Computational Notebooks

- What?
  - Notebook encapsulates Code + Results + Narrative
  - Consists of a series of 2 types of cells: code+result cell, or markdown cell
  - Example: https://nbviewer.org/gist/nealcaren/5105037
- Why?
  - Provenance = Sequence of code cells and intermediate results leading up to a solution
  - Incremental, Interactive, Collaborative, Reproducable science, ...
- Use cases
  - Data science process
  - Reporting and presentation
- Tools:
  - Jupyter, colab.research.google.com, deepnote.com, observablehq.com, ...

QAC

1. Question:
   - Question, problem, hypothesis, claim, context, motivation
   - Definitions, data, methods to be used
   - Rationale, assumptions, biases
   - Qualitative in nature, with definitions that prepare for quantitative Analysis section

2. Analysis:
   - Data processing, analysis, visualization
   - Documented code and results
   - Summary visuals
   - Quantitative in nature

3. Conclusion:
   - Answer the Question, summary, interpretation, re-contextualization, story, relate to domain knowledge
   - Uncertainty, limitations, caveats, biases
   - New problems, updates to the Question, follow-up questions, next steps
   - Qualitative in nature, shifting quantitative result back into qualitative question & answer

4. Repeat. QAC-QAC-QAC-...
   - Break down a large problem into parts
   - Alternative approaches to a problem
   - Sequence of related problems, "vignettes"
   - Follow-up problems

Data Processing

Data structures:

- List: sequence of elements, mutable “[1.2.3]”

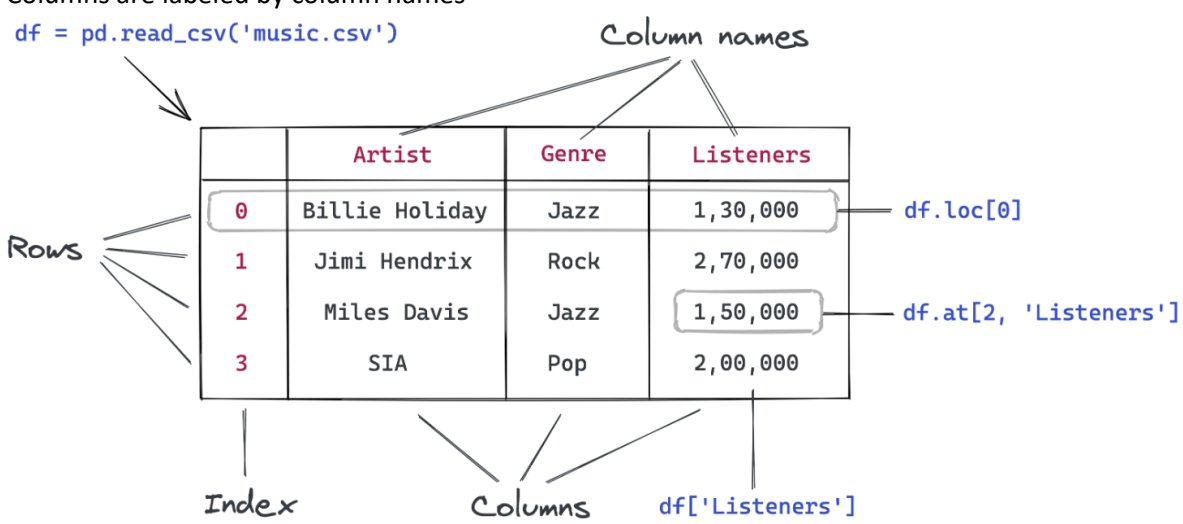
    - Slice: select data by index, [start:end:step]
    - Reduce: Interpret a list as a value (mean(), sum(), len())
    - Map: map(function, list), apply function to all list elements. Can be done with list comprehension.
    - Filter: filter(func, list), select elements that return true from func. Can be done with list comprehension.
    - Sort: sorted(list) or list.sort()
- Tuples: are immutable but more efficient then list.
- Dict: key value pairs
- Set: list where elements do not repeat, special functions can be applied to it.
- Strings: char array
- Data table: list of tuples. Created by the zip operation. Can be row major (list of rows) or column major (list of columns). LHS of tuple are elements from first list and RHS of tuple are elements from the second list.

Functional vs Loops:

- Abstraction: loops are low level
- Composable, re-usable: z = f(g(x))
- Provenance: maintain copies of lists
- Usability: focus on math, not data structure maintenance
- Optimization: native code
- Efficiency: parallel computing
- More efficiency: Lazy evaluation

Pandas Dataframe (n*p data structure)

- Rows (axis=0) and columns (axis = 1)
- n = cardinality (rows)
- p = dimensionality (columns)
- Rows are labeled by index
- Columns are labeled by column names



-

Useful dataframe operations

- Pandas.read_csv(fname)
    - Reads a csv file and returns a dataframe
- df.shape(df)
    - Returns a type (row count, column count). Index and column name is not included.
- df.describe()
    - Gives count, mean, std, min, percentile data on dataframe columns.

- df.columns
    - Returns Pandas Index object containing an array of column names and data type)
- df.info()
    - Shows class type, range of rows and # of rows, number of columns, each column's #, name, non null count, data type. Also shows total data types of columns and memory usage.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3146 entries, 0 to 3145
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Name                3146 non-null   object
 1   State               3146 non-null   object
 2   Population          3146 non-null   int64
 3   Pop2000             3146 non-null   int64
 4   IncomePerCapita     3146 non-null   int64
 5   PercentCollegeGrad  3146 non-null   float64
 6   MedianRent          3146 non-null   int64
 7   CommuteTime         3146 non-null   float64
 8   LandArea            3146 non-null   float64
dtypes: float64(3), int64(4), object(2)
memory usage: 221.3+ KB
```

- pandas.options.*
    - Various options for changing how dataframes are displayed. Reset with pandas.reset_option(*)
    - pandas.describe_option(*) describes a category of options
- df.sort_values("column name", ascending=bool)
    - Sort dataframe by a column
- df.columnName
    - Returns the every row value of a column and how many exists and column data type.
- df.iloc[startRow:endRow, startCol:endCol]
    - Get a slice of a table from row/column in range(start, end).
    - Takes numerical values only
- df.iloc[num]
    - Single number returns just a specify row with all of its columns
- df.iloc[:num]
    - Return the first 10 rows, returns all columns if second param is not specified.
- df.loc[rowlabel, collabel]
    - Returns a row whose label matches label and columns who column matches label
- Note that num and label is interchangeable with a list if specific rows are desired. The x:y construct in iloc and loc can be replaced entirely by a list of indexes. x:y construct in loc takes all rows/columns inbetween exactly like how iloc works.
- df.col.mean() or numpy.mean(df.col)
    - Reduces a column into a value
- df.mean(index=0|1)
    - Index 0 mean get average of all column values

```
Population            98139.077559
Pop2000               89453.879847
IncomePerCapita       30019.454228
PercentCollegeGrad       18.685251
MedianRent              453.595041
CommuteTime              22.597934
LandArea               1122.665928
CollegeGrad           26994.137317
dtype: float64
```

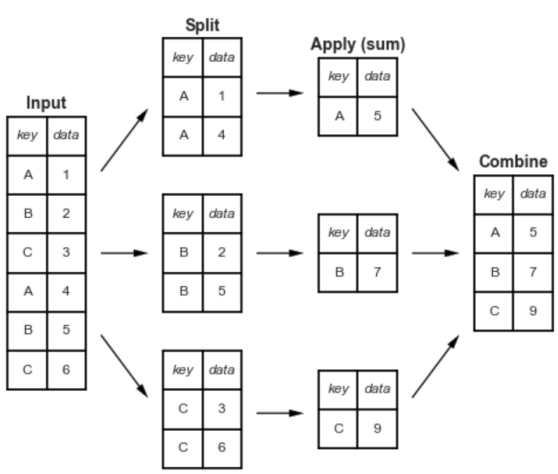
    - Index 1 means get the average of all row's column values

```
0         5330.74250
1         6283.71000
2        86910.87250
3        12986.76250
4         6544.65500
             ...
3141     23467.97250
3142     11211.27000
3143      7792.16875
3144      7508.21125
3145    190087.06875
Length: 3146, dtype: float64
```

- dt.column.idxmax()
    - Gets the index of the row who has the column with the greatest value. Use as parameter in dt.loc[] to get the entire row, not the index.
- df.column.map(func)
    - Returns a column whose values were manipulated by func
- df.column1 + df.column2
    - Adds the 2 columns together with addition operation
- df + df
    - Adds 2 dataframes together if they have the same dimensions and are all values
- df.applymap(func)
    - map every single value in the table to func
- df.apply(func, axis)
    - When axis = 0, then function converts a column to another column or a scaler
    - When axis = 1, then function converts a row to another row or scaler
- df[‘new column] = …
    - Adds an column to the dataframe or sets a current column to a new column. Column length must match table length.
- df[list[bool]]
    - When provided a list of bool, return only rows whose position corresponds to the same array position which is set to True.
    - Doing dt.column == … or data.column < or combining list with & creates filtered dataframes with the conditionals specified.

GroupBy

- Group rows that have the same key.
- Afterwards, these groups can all be reduced into a groups with only 1 row then combined into a single dataframe.



-
- df.groupby(“Column)
    - Groups all rows into groups based on column name
- g.size()

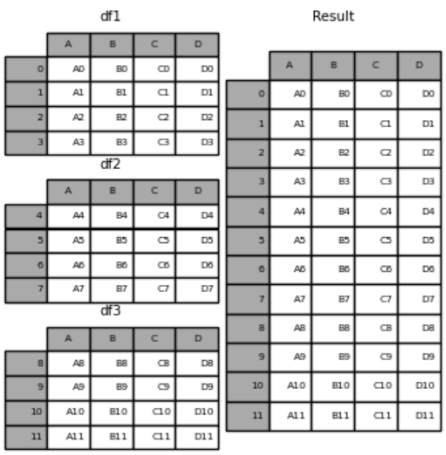
- Returns the key of each group and size of each group

```
State
AK      29
AL      67
AR      75
AZ      15
```

- len(g)
  - Returns the number of groups
- g.Column.sum()
  - Get each group's sum of their member's column value
- g.aggregate({"Column1": sum, "Column2":numpy.mean, "Column3":first, …})
  - Reduce each group into a single row with dict parameter and combine into a single dataframe. The dict parameter values take in a list of values and returns the result as a scaler.

Union

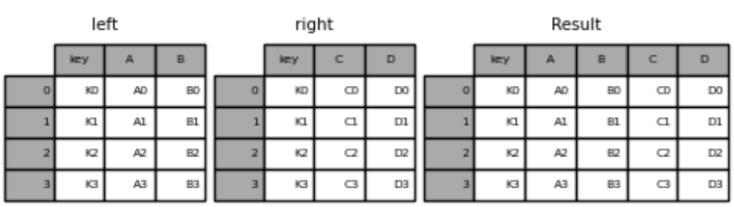
- Combine tables vertically
- Results in table with n+m rows where table 1 has n rows and table 2 has m rows.



- pandas.concat([c,s])
  - Union dataframes c and s. If there are columns in c that don't exist in s and vice versa, they will be NaN.
- Note that c.append(s) does the same thing as concat but append will be deprecated soon.

Join

- Combine tables horizontally.
- Results in table with n+m cols where table 1 has n cols and table 2 has m cols.



- pandas.merge(leftdf, rightdf, how, left_on, right_on, suffixes)
  - Join leftdf and rightdf together
  - How: String; join type

`left`	`LEFT OUTER JOIN`	Use keys from left frame only
`right`	`RIGHT OUTER JOIN`	Use keys from right frame only
`outer`	`FULL OUTER JOIN`	Use union of keys from both frames
`inner`	`INNER JOIN`	Use intersection of keys from both frames
`cross`	`CROSS JOIN`	Create the cartesian product of rows of both frames

        -
        - Inner: Only keep rows that exist in both dataframes
        - Left: Keep rows that exists in leftdf, rows that only exists in rightdf are dropped
        - Outer: Keep all rows
        - Rows from inner, left, and outer that exists only in one dataframe will have their missing column values as NaN
    - left_on: left dataframe key column
    - right_on: right dataframe key column
    - suffixes: If 2 columns with the same label exists in both dataframes, add a suffix unique to left and right dataframes to the joined columns. For example, suffix=('_s', '_c')
- Join types
    - 1:1
        - Leftdf's keys do not match any of rightdf keys
        - Left: key = a, col = data_a | Right: key=a, col=data_b
            - Join: key = a, col = data_a, data_b
    - 1:m
        - Leftdf's keys match multiple of rightdf keys
        - Left: key = a, col = data_a | Right: key=a, col="b1", key a, col="b2"
            - Join: key = a, col= data_a,"b1" && key = a, col= data_a,"b2"
    - m:m
        - Multiple leftdf's keys match multiple of rightdf keys
        - Left: key = a, col = "a1", key = a, col="a1" | Right: key=a, col="b1", key a, col="b2"
            - Join: key = a, col="a1", "b1" && key = a, col="a1", "b2"&& key = a, col="a2","b1" && key = a, col="a2" ,"b2"

Zip operation

- Used to combine lists
- When given n lists, create a tuple with n columns of the length of the provided lists.
    - Zip([a,b,c], [1,2,3]) == ((a,1),(b,2),(c,3))
- When provided a single list, create a tuple with 1 column of the length of the provided list.
- End result can be converted to list with list(result)

How to parse data from text files

- Open it first
    - Text = open("./path").
- Read, splitlines, or split
    - Text.read()
- List comprehension with tuple assignment to create a data table.
    - dt3 = [(c,s,int(p)) for (c,s,p) in dt2]

Zscore

- $$zscore(x) = \frac{(x - mean)}{stdev}$$
- Normalize data so mean=0 and stddev=1.
- Process
  - Get the mean and standard deviation of data column to normalize
    - ```
      dt.Population.mean(), dt.Population.std()
      ```
  - Subtract data by mean and then divide by standard deviation
    - ```
      (dt.Population - dt.Population.mean()) / dt.Population.std()
      ```
  - Whole tables may be done instead of columns if they contain only quantitative data
    -

	CollegeGrad	CommuteTime	IncomePerCapita	LandArea	MedianRent	PercentCollegeGrad	Pop2000	Population
count	3.146000e+03	3.146000e+03	3.146000e+03	3.146000e+03	3.146000e+03	3.146000e+03	3.146000e+03	3.146000e+03
mean	2.075051e-17	-3.310342e-15	-1.496295e-17	-6.659221e-17	-7.509709e-17	-9.500065e-16	-7.177984e-17	-3.761913e-17
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00

Dirty data:

How to find?

- Wrong data types
- Strange data distributions, summary, and stats
- Presence of numpy.nan

How to clean?

- missing data: set to NaN
- invalid data: set to NaN
- extreme data: set to NaN
- wrong data types: convert, e.g. int("47")
- wrong data units: math, e.g 1GB == 1,000,000,000b
- duplicated data: eliminate
- messy categories: standardize, e.g. "VA"→"Virginia"
- messy text: whitespace, punctuation, unusual chars, emojis

Practical things for handling missing data

- NaN
- Isnull() notnull() to find missing data
- Can skip NaN data (sum(skipna=True))
- Remove rows with missing data (dropna())
- Impute replacement value (fillna())

Values to impute for missing data:

- Fixed value 0
- Use other data like mean or median in the same category
- Use regression to predict data value that could've been there
- Sample from column distribution, value frequencies, data simulation.
- Similar row, multidimensional nearest neighbor.

Common dirty table problems:

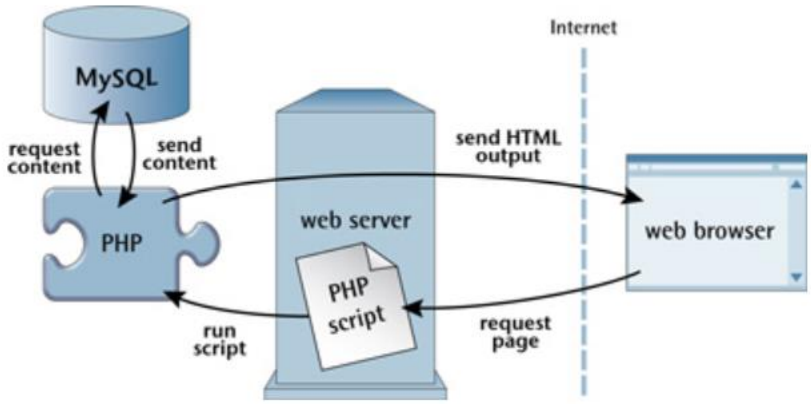
- Column headers are values, not variable names
- Multiple variables in one column
- Variables stored in both rows and columns
- Multiple types of observable units in same table
- Single observable unit in multiple tables

Strategies for cleaning columns:

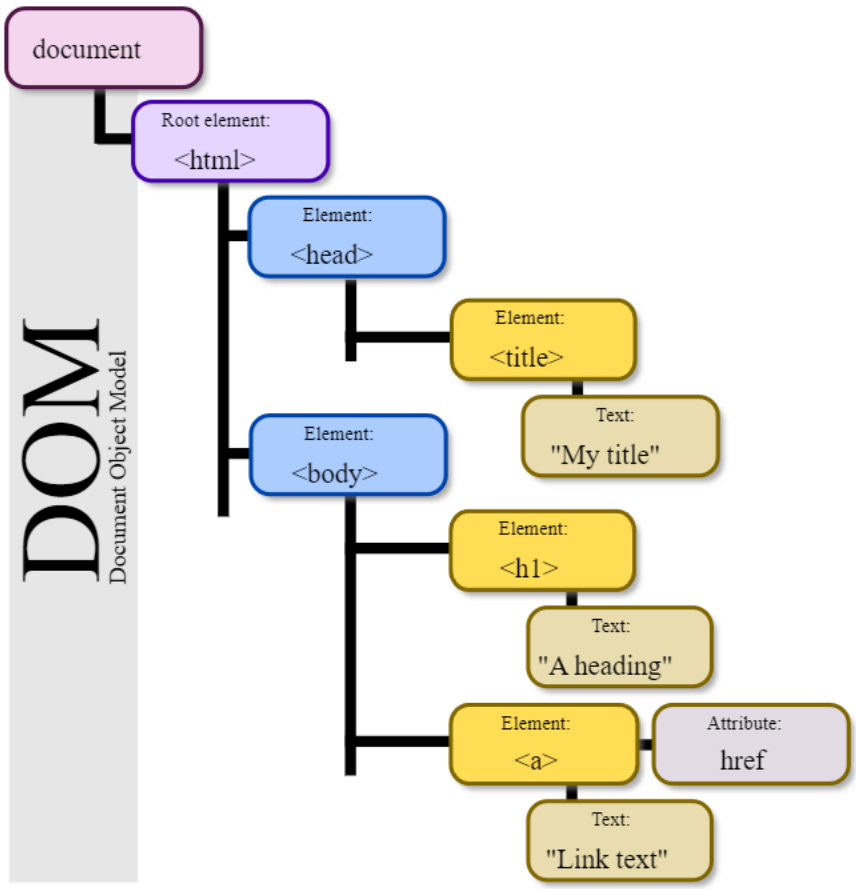
- df.dtypes
  - Shows all data types, mismatched column data type can be seen here
- df.column = df.column.map(lambda s: int(s.replace("token", "replacement")))
  - Replace a column with string values in non int format into an int column.
- df.describe()
  - Shows lots of computed per column like min, max, mean, std. Can reveal the presence of unusual data which can be filtered and imputed.
- df.column.mask(bool | list[bool], numpy.NaN)
  - Replace all column values with numpy.NaN if the conditional is true.
  - Example conditional: (df.Name.str.endswith('VA')) & (df.Income == 0)
- df.column.func(skipna=bool)
  - If a reduction function (such as sum) is applied to a column, skipna when true will skip all NaN while when false, will return NaN if a NaN exists in the column.
- df.isnull().any()
  - Shows which columns contain null data
  - Isnull() transforms a dataframe into a true/false matrix where true is where null exists and any() shows columns with any True values.
  - Isnull() can be combined with filtering to show which rows specifically have null values
- df.column.fillna(df.column.mean())
  - Replace all NaN values in a column with the average value of that column

Web scraping

Web publishing process:

- Data -> (server side scripts) -> HTML -> (browser) -> visual page
  - 

HTML DOM

- Structure and page layout in hierarchy view of tags and elements.
  - 

How to obtain data?

- Download, Web DB APIs, and scraping

How web scraping works:

- Web scraping process: fetch, parse, crawl
- Algorithmically parse HTML DOM, search for unique tags and navigate DOM structure
- Common parsing patterns of page -> data
  - Page ➔ data row: product image
  - Page ➔ k data rows: paginated search result list
  - Page ➔ Data column: index page, list
- Cautions:
  - Blocked by server
  - Human or cyber attack?
  - Retrieved page differs from use of JS
- Ethical concerns:
  - Does provider allow scraping?
  - Commercial restrictions?
  - Intellectual property?
  - Rapid crawling harm?
  - Privacy issues?

Web scraping process

1. Fetch data
    a. Requests or Selenium
    b. GET request URL

```
### Example:  Yelp reviews
import requests
```

```
url = "https://www.yelp.com/search?find_desc=Restaurants&find_loc=Blacksburg%2C+VA+24060&ns=1&sortby=rating"
page = requests.get(url)
```

```
page.reason
```

```
'OK'
```

```
page.text
```

...

c.

```
open('page.html', 'w').write(page.text)
```

2. Parse response body
    a. BS4
    b. Two strategies: look for unique tag attributes or navigate DOM tree top down.

```
tag = soup.find('tag', {dictionary of identifiers})
```

```
tags = soup.find_all('tag', {dictionary of identifiers})
```

c.

```
## Example: Find the restaurant names

tags = soup.find_all('a', {"class": "css-1m051bw", "rel": "noopener"})
tags
```

```
[<a class="css-1m051bw" href="/biz/taqueria-el-paso-christiansburg?osq=Restaurants" name="Taqueria el Paso" rel="noopener" targ
et="_blank">Taqueria el Paso</a>,
```

```
childtag = tag.contents[childnumber]
```

d.

```
## Example:  Navigate from the restaurant name to the restaraunt review counts
parent = tags[0].parent.parent.parent.parent.parent.parent
```

```
parent.contents[1]
```

```
<div class="border-color--default__09f24__NPAKY"><div aria-hidden="false" class="display--inline-block__09f24__fEDiJ border-col
or--default__09f24__NPAKY"><div class="border-color--default__09f24__NPAKY"><div class="display--inline-block__09f24__fEDiJ mar
```

3. Crawl
    a. Scrape many pages by:
        i. Parsed hyperlinks on index page
        ii. URL query string

b.

```
## Example: crawl to each restaurant's review page
url = "https://www.yelp.com/search?find_desc=Restaurants&find_loc=Blacksburg%2C+VA+24060&ns=1&sortby=rating&start="
urli = url + str(40)
urli
```

```
'https://www.yelp.com/search?find_desc=Restaurants&find_loc=Blacksburg%2C+VA+24060&ns=1&sortby=rating&start=40'
```

```
## Exercise: crawl paginated lists: restaurants 1-10, 11-20, 21,30, ...

pages = [requests.get(url + str(i*10)) for i in range(5)]
pages
```

```
[<Response [200]>,
 <Response [200]>,
 <Response [200]>,
 <Response [200]>,
 <Response [200]>]
```

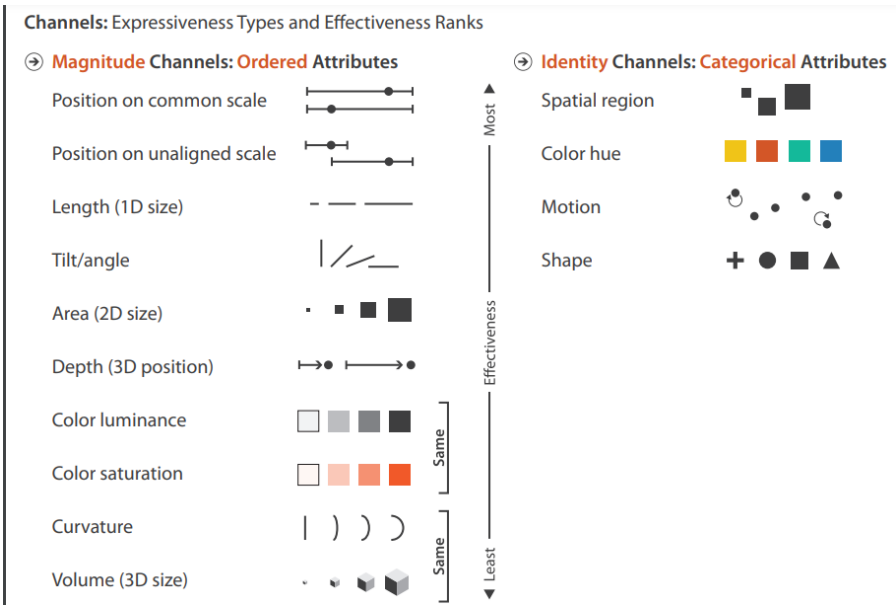
Data Visualization

Goal of visualization: Organize large amounts of data visually to make patterns apparent.

- Rows are shapes

- Columns are shape attributes.

Munzner's ranking of visual encodings:

- Quantitative == magnitude
- Qualitative == categorical
- 

Jupyter visualizations

- %matplotlib inline
    - Reliable static visualization
- %matplotlib notebook
    - Interactive

Plotting steps

- Prepare a dataframe
- Call dataframe.plot.plot_type(plot specific settings)
    - Returns an axes subplot object of a plot created by the call.

Bar chart

- Categorical unique key column -> x-axis
- Quantitative column -> y-axis
- Use case: Used to compare values
- Variations
    - Multi set bar
    - Stacked bar
    - 100% stacked bar
- Pie chart
    - Ratio to sum
    - Key are slices, quantitative is slice angle.

Scatterplot

- Quantitative data in both x-axis and y-axis
- Use case: correlation, data correlation
- Variations
    - Bubble chart
        - Quantitative column is dot size
        - Quantitative or categorical data is dot color.

Line

- Time or continuous quantitative column -> x-axis
- Multiple quantitative columns -> y-axis and color
- Use case: time trend
- Variations
    - Area
    - Stacked area.

Color mapping

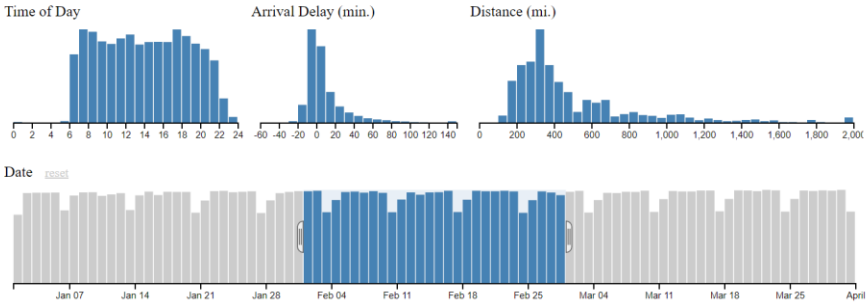
- Hue (wavelength) perceived categorically, intensity perceived quantitatively as amplitude.
- Mapping color with colormaps
    - Categorical data -> hue: Qualitative, cycle colormaps
    - Quantitative -> saturation or luminance: sequential, diverging +/- colormaps
- Avoid
    - Red-green colormaps
    - Rainbow colormaps for quantitative data

Annotating plots (axes subplot object)

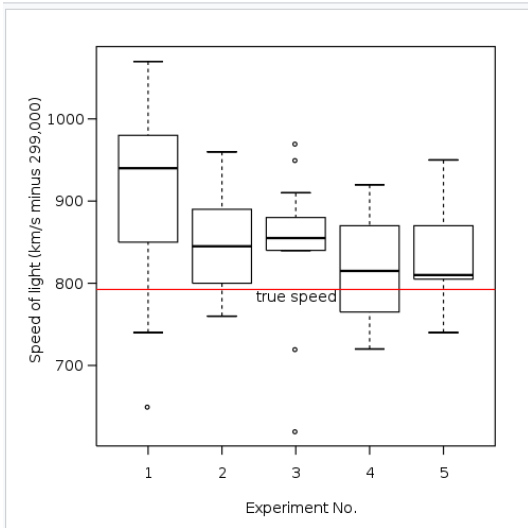
- Use axes object returned from the plot
- ax.text(), ax.add_line, ax,add_path(), ax.annotate.

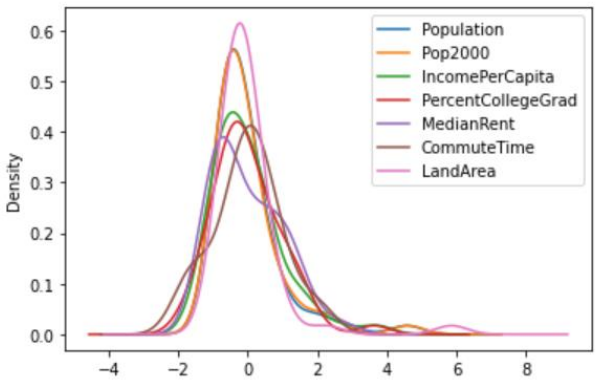
Multidimensional Visualizations

Multi-variate distributions

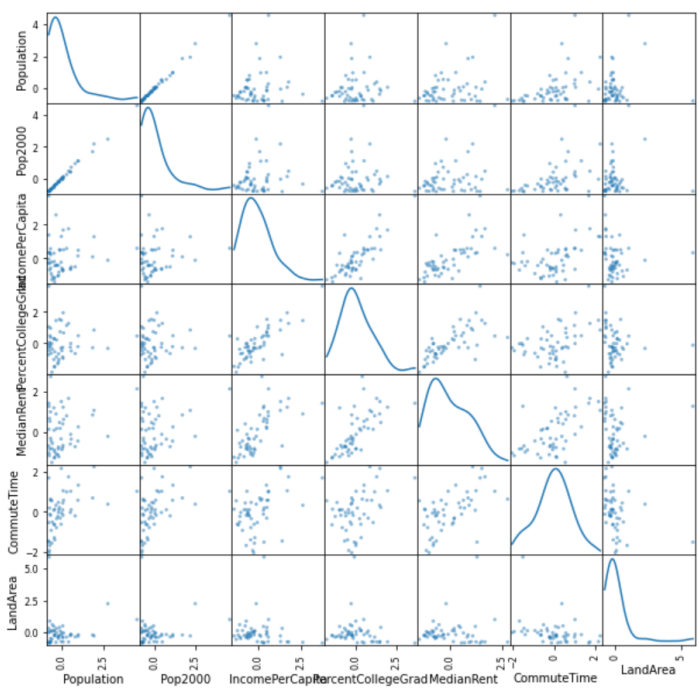
- Histogram matrix
    - 

- Boxplots

- 

- KDEs
  - 

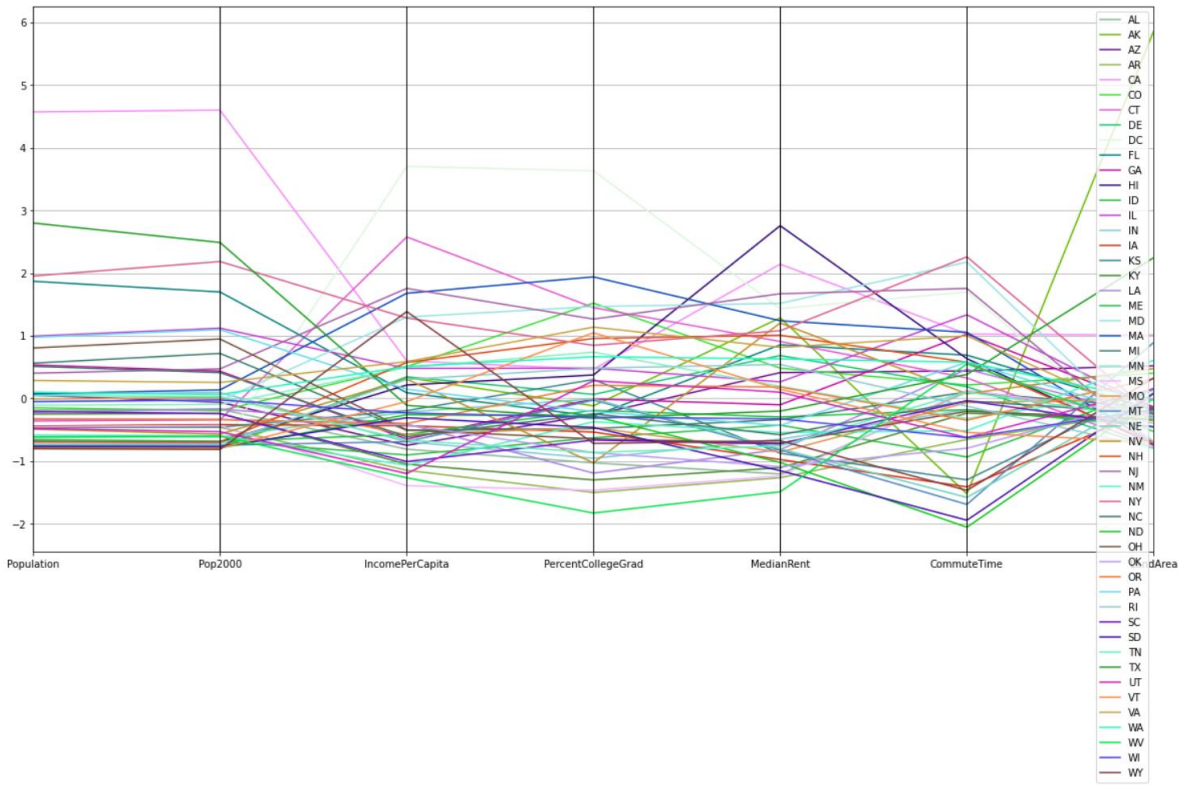
Scatterplot matrix

- Scatterplot all pairs of variables
- 

- Need to specify what the diagonal should be, is KDE for diagonal above.

Parallel coordinates

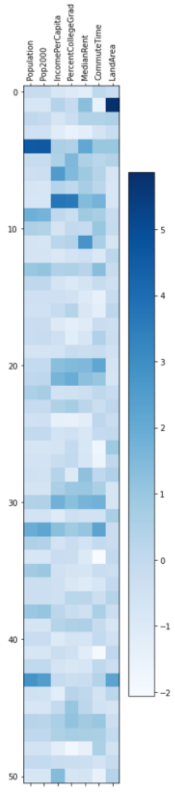
- Parallel axes with data rows as connected lines



-

Heat map

- Spreadsheets with color



-

Trees and Graphs

Types of data:

1D – text documents, lists

2D – maps

3D – real world objects

MD – databases

Temporal – timelines

Tree

Network

Tree vs Graph

Trees:

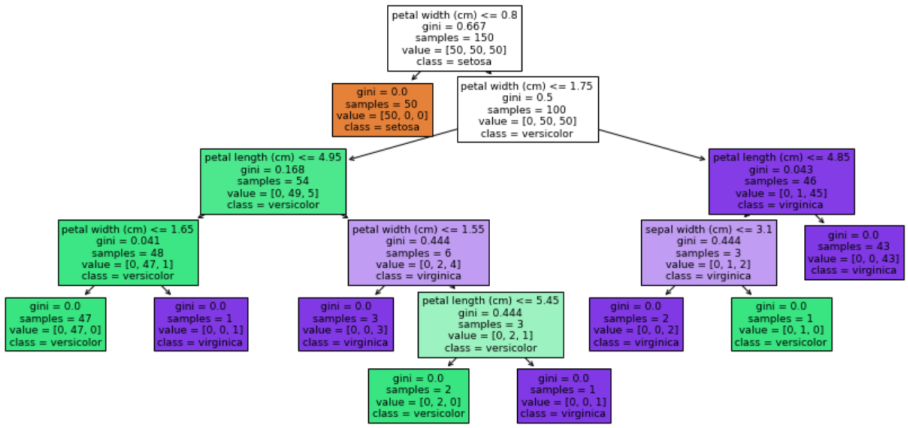
- Nonlinear
- Set of nodes and edges
- Unique root node
- Node in a tree can have 2 children or more
- Trees do not have cycles

Graphs

- Nonlinear
- Set of nodes and edges
- No root node
- Node can have few or many edges
- Graphs can have cycles

Decision Tree Visualization

- Generate tree with attribute and label data.
- Use plot_tree to visualize data
- Tree.export_text(decisionTree) prints tree in text form.

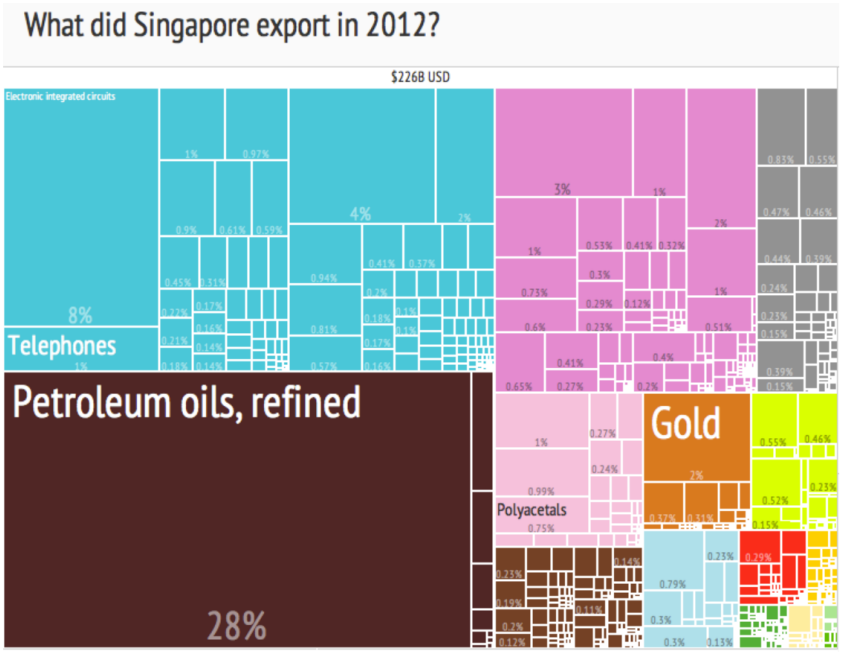


-

Treemaps

Area plots data

1. Categorical data for relationships among data
2. Display magnitude or importance of categories of one can understand the enormity of a categort
3. When data is large, treemaps provide clean and informative insight into data.



Steps:

1. Use plotly.express px.treemap(d, path=[px.Constant("Animal"), 'Type', "Name"], values=dim, color=Type)

- d is the dataset to be visualized, should have only positive, nonnegative values in the value that will be visualized.
- Path is the layers from the outside of the map to the inside. Constant is for constant label while non constant labels are taken from the dataframe.
- Values is which attribute is to be visualized, this is seen in how large boxes are.
- Color is used to distinguish groups from each other.
- There are numerous other ways to color treemaps to illustrate a hierarchy.

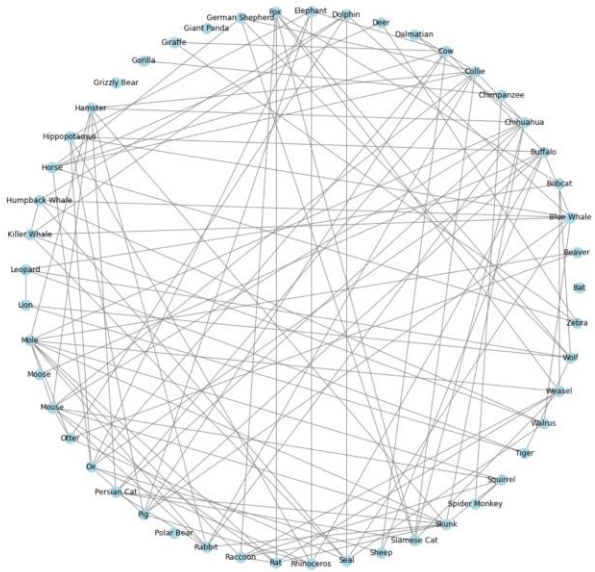
Graphs

- Normalize dataset.
  - ```
    norm = (table - table.mean()) / table.std()
    ```
- Generate a distance matrix from the normalized data.
  - ```
    dist = pairs.manhattan_distances(norm)
    dist = pd.DataFrame(dist, index=norm.index, columns=norm.index)
    ```
- Use nx.from_pandas_adjacency(dist) to generate graph
  - ```
    import networkx as nx
    G = nx.from_pandas_adjacency(dist)
    ```
- Use nx.draw to draw the graph.
  - ```
    plt.figure(figsize=(15,15))
    nx.draw(G, with_labels=True, edge_color='grey', node_color='lightblue')
    ```

- Filtering distance matrix to use a certain distance and using nx.draw_circular makes a more clean graph.

```
distFlitered = dist.applymap(lambda v: v if 0<v<25 else 0)
G = nx.from_pandas_adjacency(distFlitered)
```

```
plt.figure(figsize=(15,15))
nx.draw_circular(G, with_labels=True, edge_color='grey', node_color='lightblue')
```

  - 

-

Force Directed Graphs

- Assign spring force to edges
- Assign repel force to nodes
- Assign centering force to entire graph
- Simulate forces until energy in system is minimized.

```
from pyvis.network import Network
net = Network(notebook=True)
net.from_nx(G)
net.show("example.html")
```

```
Local cdn resources have problems on chrome/safari when used in jupyter-notebook.
```

Interactive Visualization

Goal: respond to interactive events in plots or widgets

Callback functions

- Create plot/widget
- Define callback function
- Register callback function for an event in the plot or widget

Example: Picking

1. Get a dataframe, get its axis subplot through .plot.scatter() or a related function. Set picker=true
   a.
```
ax = data.plot.scatter(x='longitude', y='latitude', picker=True)
```
2. Add a class back function which is the event handler for a pick event
   a.
```
#class back function
def pick(event):
    text.set_text(data.tweet.iloc[event.ind[0]])
```
3. Register the event handler
   a.
```
# register
ax.figure.canvas.mpl_connect('pick_event', pick)
```

Linear Regression

Univariate Distribution Analysis

- Obtain from df.describe, measures of data
  - $mean(x) = \bar{x} = \mu(x) = \frac{1}{n}\sum x_i$
  - $std(x) = \sigma(x) = RMSE(x, \bar{x}) = \sqrt{\frac{1}{n}\sum(x_i - \bar{x})^2}$
    - https://en.wikipedia.org/wiki/68–95–99.7_rule
  - $var(x) = \sigma(x)^2$
  - $zscore(x_i) = \frac{x_i - \bar{x}}{\sigma(x)}$
-

Bivariate Correlation Analysis

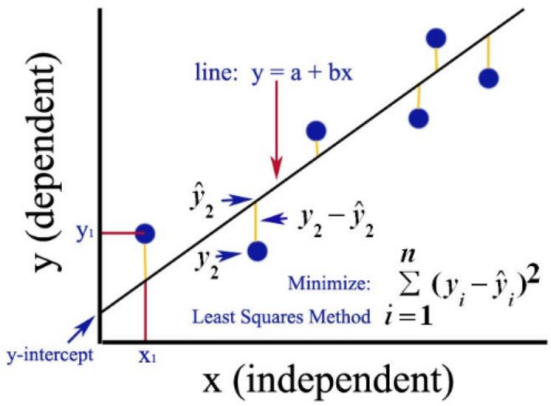
Pearson Correlation coefficient

- Center of mass = $(mean(x), mean(y), \dots)$
- $cov(x, y) = \frac{1}{n}\sum(x_i - \bar{x})(y_i - \bar{y})$
- **Pearson Correlation**
  - https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
  - $r(x, y) = \frac{cov(x,y)}{\sigma(x)\sigma(y)} = cov(zscore(x), zscore(y))$
  - $r(x, y) = [-1, 1]$
    - -1 = negative linear correlation
    - 0 = no linear correlation
    - 1 = positive linear correlation
-
  - Correlation matrix = correlations between all pairs of columns

Linear Regression Analysis

- Regression is unsupervised machine learning
  - Linear relationship model
  - Learn to predict outs (y) given inputs (x….)
    - Labeled data (y=labels)

Simple linear regression



- Predict y given x
- y=mx+b
- Steps
  - Create sklearn model with LinearRegression().fit(y,x)
    - First argument is the label data used to predict the first argument
    - Second argument is column to predict

```
lm = LinearRegression().fit(s[['PercentCollegeGrad']], s.IncomePerCapita)
lm.coef_, lm.intercept_
```

    - `(array([995.54255058]), 10791.531798151806)`
  - Model then can be used to predict values like:

```
lm.predict([[33.4]])
```

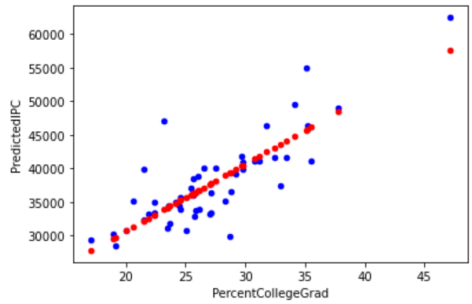
```
/opt/anaconda3/lib/pytho
ression was fitted with
  warnings.warn(
```

    - `array([44042.65298743])`
    - Or `33.4 * lm.coef_[0] + lm.intercept_` where the first number is the y value of the row column to predict, returns the predicted x value
  - Or create a new column of just predicted values with df[“prediction”]=lm.predict(y column)

```
s['PredictedIPC'] = lm.predict(s[['PercentCollegeGrad']])
```

```
ax = s.plot.scatter(x='PercentCollegeGrad',
                    y='IncomePerCapita', color='blue')
s.plot.scatter(x='PercentCollegeGrad',
               y='PredictedIPC', ax=ax, color='red')
```

```
<AxesSubplot:xlabel='PercentCollegeGrad', ylabel='PredictedIPC'>
```



- Compute residual, the m,b that minimizes the loss

```
s['Residual'] = s.PredictedIPC - s.IncomePerCapita
(s.Residual**2).sum()
```

```
723001434.7839286
```

- Perform pearson correlation coef with lm.score(y column, x column) or manually

```
(s.IncomePerCapita.corr(s.PredictedIPC))**2
```

```
0.6739674104888136
```

```
lm.score(s[['PercentCollegeGrad']], s.IncomePerCapita)
```

```
0.6739674104888135
```

Polynomial Regression

- Predict y given x,x^2,x^3
- Steps
  - Use PolynomialFeatures(deg,include_bias=False).fit_transform(y column)

```
X = PolynomialFeatures(degree=2, include_bias=False).fit_transform(s[['PercentCollegeGrad']])
```

  - Create a linear model using the polynomial model and x column

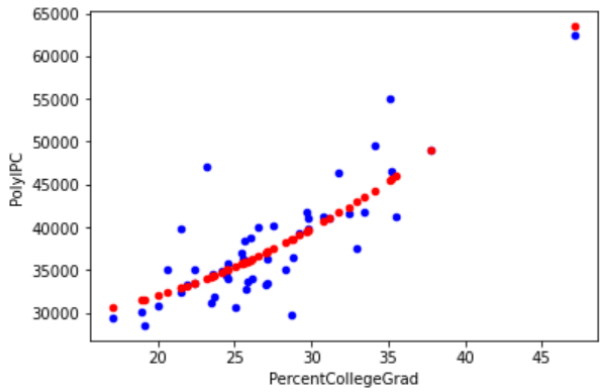
```
pm = LinearRegression().fit(X, s.IncomePerCapita)
```

  - Call pm.predict(polynomial model) to get predicted x column

```
s['PolyIPC'] = pm.predict(X)
```

```
ax = s.plot.scatter(x='PercentCollegeGrad',
                    y='IncomePerCapita', color='blue')
s.plot.scatter(x='PercentCollegeGrad',
               y='PolyIPC', ax=ax, color='red')
```

```
<AxesSubplot:xlabel='PercentCollegeGrad', ylabel='PolyIPC'>
```



- Pearson correlation by pm.score(Polynomial model, x column)
  - ```
    pm.score(X, s.IncomePerCapita)
    ```

Multiple Linear Regression

- Linear regression but use multiple y columns

```
: d = s[['PercentCollegeGrad', 'MedianRent']]
```

```
: mm = LinearRegression().fit(d, s.IncomePerCapita)
```

```
: mm.predict(d)
  mm.score(d, s.IncomePerCapita)
```

- ```
  : 0.6963076560360941
  ```

Distance

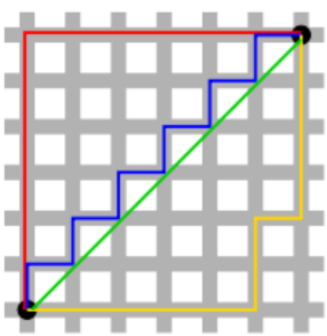
Similarity

- Similarity == Compare data rows
  - Clusters, outliers, nearest neightbor

Distance Metrics

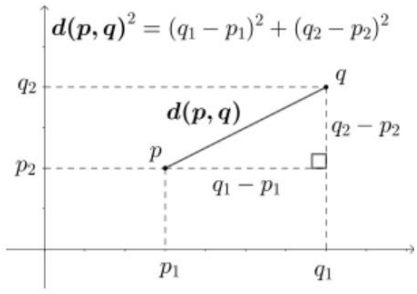
- dist(P,Q) is distance between P and Q
- To be a proper distance metric, function dist must have
  - Postivity: dist(P,Q) > 0
  - Symmetry: dist(P,Q) = dist(Q,P)
  - Identity-discerning: dist(P,Q)=0 -> P = Q
  - Triangle inequality: dist(P,R) <= dist(P,Q) + dist(Q,R)
- Must normalize data first

L1 Manhattan distance

- $dist_{L1}(P,Q) = \sum_i |P_i - Q_i|$
- 
- To find L1 of 2 rows, subtract one row from another row, get the abs value then compute the same of columns
  - ```
    (norm.loc['VA'] - norm.loc['MD']).abs().sum()
    ```

L2 Euclidean

- $_2(P,Q) = \sqrt{\sum_i (P_i - Q_i)^2}$
- Diagonal distance between 2 points
    ```

- 

- To find L2 of 2 rows, subtract one row form another row, square all columns then get the sum of columns. Then get the square root.
  - ```
    ((norm.loc['VA'] - norm.loc['MD']) ** 2).sum() ** 0.5
    ```

Lp (Minkowski distance)

- $dist_{Lp}(P,Q) = (\sum_i |P_i - Q_i|^p)^{1/p}$
- Generalized formula for Lp
- P = 1 is manhatten, p = 2 is Euclidean.

Hamming Distance

- String matching distance which computes the number of equal entries.
- $$dist_{Ham}(P,Q) = \sum_i \begin{cases} 0, & \text{if } P_i == Q_i \\ 1, & \text{otherwise} \end{cases}$$
- Process
  - Get columns in a row that are not in another row then get the number of columns remaining
  - ```
    (cities.loc["Washington DC"] != cities.loc["New York"]).sum()
    ```

Weighted Distance

- Apply a weight wi to each data column i
- $\sum_i w_i = 1$

  Weighted L2: $dist_w(P,Q) = \sqrt{\sum_i w_i (P_i - Q_i)^2}$

  Mahalanobis distance: $w_i = 1/\sigma_i^2$, same as L2 of z-score normalized data

Distance Matrix

- Compute the distance of all pairs
- Process
  - Nest for-loop, outer loop goes through rows "r" and inner loop goes through all columns "c"
  - Within the inner for loop, matrix.iloc[r,c] = distance formula with row and column
    - ```
      for r in range(len(states.index)):
      #    matrix.iloc[r, r] = 0
          for c in range(r):
              matrix.iloc[r, c] = matrix.iloc[c, r] = (norm.iloc[r] - norm.iloc[c]).abs().sum()
      ```
    - This creates NaN in the diagonal
  - Use distanceMatrix['row'].idcmin() to find the row that is most similar to another row. The discovered row may be more similar to another row however.

Computational Alternative

- Sklearn has functions to directly compute the distance matrix just by providing the normalized data which can be over 1000x faster than nested for loops.
      ```

Dimension Reduction

Goal: reduce number of columns a table has

- 2 or 3 columns for a scatterplot visualization

MDS

- Multi-dimensional scaling, distance preservation
- Visualize distance against points.
- Parameters
    - High dimension distance matrix
    - Low dimension Euclidean distance function
- Find
    - Low dimensional points with the least amount of stress
    - Stress is projection error
    - Stress = 0 is best projection
    - Stress = 1 is null project where pi==pj
    - Goal is stress < 0.1
- Optimization algorithm
    - Randomly initialize all pi
    - Gradient descent to local minimum
        - Compute force vector on each point pi
        - Nudge each point pi by its force vector
        - Re-compute stress
        - Lower stress, repeat cycle until max_iter
    - Save result
    - Got to beginning until n_init
    - Select result with the least sress
- O(n^2 * max_iter * n_init)
- Generalizable to various distance functions
- Relative distances, scale and rotation invariant.
- Data should be normalized

Compute MDS by Hand

- Sample matrix

	C1	C2	C3	C4
A	0	0	0	0
B	1	1	1	1
C	1	1	1	2
D	0	1	3	1

    -
- L1 distance matrix for all rows

	A	B	C	D
A	0			
B	4	0		
C	5	1	0	
D	5	3	4	0

- Draw points in 2D to preserve pairwise distance

```
y
|
| A       B  C
|
|
|
|        D
+-------------------- x
```

- Get 2D coordinates of drawn points

	x	y
A	0	4
B	4	4
C	5	5
D	4	0

- Fill L2 euclidean distance matrix for the drawn points

	A	B	C	D
A	0			
B	4	0		
C	6	1	0	
D	5	4	4	0

- Compute stress:

$$Stress = \left(\frac{\sum_{i,j} (d_{ij} - ||p_i - p_j||)^2}{\sum_{i,j} d_{ij}^2} \right)^{1/2}$$

Stress = 0.15

```
(2 / (4**2 + 5 ** 2 + 5 ** 2 + 1 + 3 ** 2 + 4 ** 2)) ** 0.5
```

```
0.14744195615489714
```

Compute MDS Computationally

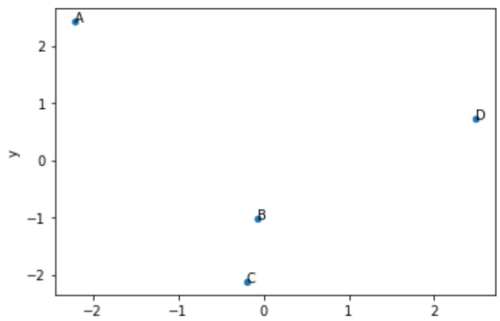
1. Generate L1 distance matrix
   a. DistHD = Sklearn.metrixs.pairwise.manhattan_distances(dataHD)
   b. distHD = pandas.DataFrame(distHD, columns=dataHD.index, index=dataHD.index)
   c.

	A	B	C	D
A	0.0	4.0	5.0	5.0
B	4.0	0.0	1.0	3.0
C	5.0	1.0	0.0	4.0
D	5.0	3.0	4.0	0.0

2. Apply MDS
    a. mds = sklearn.manifold.MDS(n_components=2, dissimilarity='precomputed')
        i. n_components is target dimensionality
        ii. max_iter is maximum iterations per init
        iii. n_init is number of inits to make
        iv. dissimilarity precomputed means distance matrix computed already and used as parameters in fit_transform later. Default is Euclidean distance.
    b. data2D=mds.fit_transform(dataHD)
    c. data2D=pandas.DataFrame(data2D,columns=['x','y'], index-dataHD.index)
    d.

	x	y
A	-2.206880	2.420670
B	-0.077155	-1.024468
C	-0.194165	-2.121498
D	2.478200	0.725296

3. Plot points
    a.



4. Compute stress
    a. Get 2D Euclidean distance matrix

```
## 2D euclidean distances in the plot
dist2D = sklearn.metrics.pairwise.euclidean_distances(data2D)
dist2D = pandas.DataFrame(dist2D, index=data2D.index, columns=data2D.index)
dist2D
```

    i.

	A	B	C	D
A	0.000000	4.050272	4.968130	4.982396
B	4.050272	0.000000	1.103252	3.097017
C	4.968130	1.103252	0.000000	3.904583
D	4.982396	3.097017	3.904583	0.000000

    b. You can check the error in distance made by MDS by subtracting L1 distance matrix by L2 euclidean distance matrix.

```
## Distance error
distHD - dist2D
```

    i.

	A	B	C	D
A	0.000000	-0.050272	0.031870	0.017604
B	-0.050272	0.000000	-0.103252	-0.097017
C	0.031870	-0.103252	0.000000	0.095417
D	0.017604	-0.097017	0.095417	0.000000

    c. Compute stress by following formula
    i.

$$Stress = \left(\frac{\sum_{i,j} (d_{ij} - ||p_i - p_j||)^2}{\sum_{i,j} d_{ij}^2} \right)^{1/2}$$

```
## Stress function
((((distHD - dist2D) ** 2).sum().sum()) / (distHD ** 2).sum().sum()) ** 0.5
```

ii. 0.018948066294410076

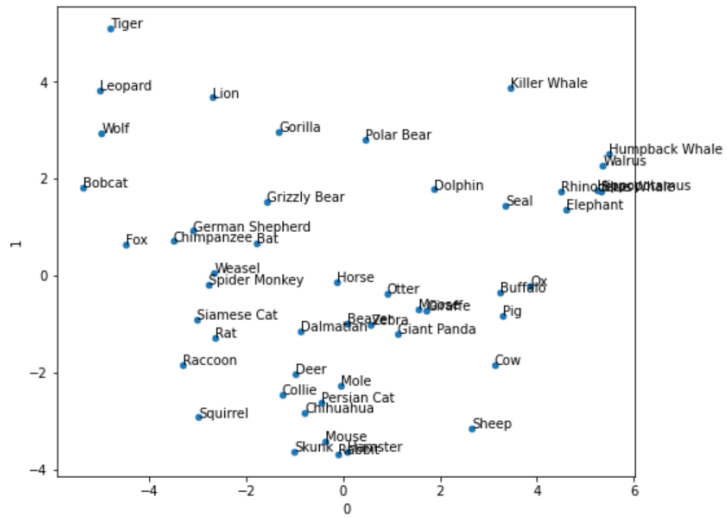
Weighted MDS

- Interactive, dynamic weights.
- MDS with weighted distance function
  - Weight wi for data column i that represents importance.
  - Weighted L1:
    - $dist(P, Q) = \sum_i w_i |P_i - Q_i|$
- Forward projection: given dimension weights, find projection that minimizes error in weighted pairwise distances
- Inverse projection: given a user modified projection, find dimension weights that best produce that projection
- Andromeda

Principal Component Analysis (PCA)

- Compute new axes (principal components)
  - Linear combination of the column
  - Orthogonal to each other
  - Maximize explained variance
  - Choose first 2 or 3 PCs for visualization
- Goal: find orthogonal axes that capture the max variance, preserve the original data's variance.
- Concept
  - Rotate data such that maximum variance is on X
  - Project onto X, save as PCi
  - Project out X
  - Go to 1
- Computation
  - Correlation matrix
  - Eigenvectors, eigenvalues
  - Selection eigenvectors for largest k eigenvalues
    - Explained variance == eigenvalues
  - Project data onto eigenvectors
- Assumes L2 distance
- Steps:
  - Get normalize matrix
  - Generate PCA
    -
```
pca = sklearn.decomposition.PCA()
tablePCA = pca.fit_transform(norm)
tablePCA = pandas.DataFrame(tablePCA, index = norm.index)
```
  - Out of the large generated PCA table, plot the first 2 PCs

```
## Plot the first 2 PCs as x,y
ax = tablePCA.plot.scatter(0, 1, figsize=(8,8))
ax.axis('scaled')
for i, r in tablePCA.iterrows():
    ax.text(r[0], r[1], i)
```



- View ratio of explained variance of each PC

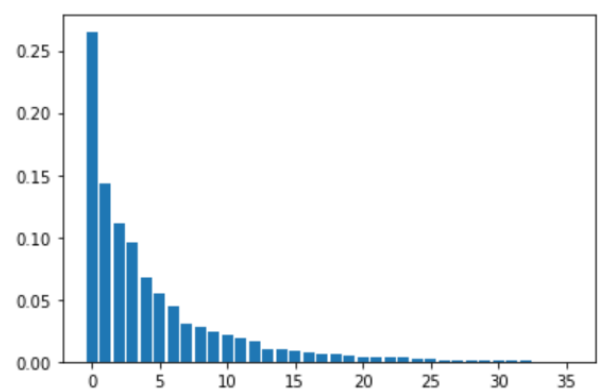
```
pca.explained_variance_ratio_
```

```
array([2.65244039e-01, 1.42694950e-01, 1.10905281e-01, 9.61110824e-02,
       6.74820395e-02, 5.42212832e-02, 4.48155345e-02, 2.99485677e-02,
       2.75763198e-02, 2.38994981e-02, 2.20542900e-02, 1.84994193e-02,
       1.66528802e-02, 1.04816482e-02, 9.84873917e-03, 9.14239312e-03,
       7.66148727e-03, 6.53358472e-03, 5.57713564e-03, 4.47489584e-03,
       4.21471615e-03, 3.63697800e-03, 3.34482819e-03, 3.02683385e-03,
       2.74520290e-03, 1.90350153e-03, 1.42935190e-03, 1.39932360e-03,
       1.20026568e-03, 9.30626558e-04, 6.69070737e-04, 4.92188940e-04,
       4.25731995e-04, 2.99617845e-04, 2.92736235e-04, 1.63957538e-04])
```

- Plot a scree plot

```
matplotlib.pyplot.bar(tablePCA.columns, pca.explained_variance_ratio_)
```

```
<BarContainer object of 36 artists>
```



Clustering:

What is clustering: reduce rows of a large table by aggregating similar rows in k < n clusters where the cluster centroids approximate the full data.

- Rows are clustered based on high dimensional distances
- Cluster labels = ID# of each cluster
- Cluster membership = assign each row a cluster label for use by GroupBy
- Cluster Centroid = mean of the rows in the cluster
- K = number of clusters

K-means clustering

Selects K clusters that minimize the total within cluster **variance** (rows assigned to their nearest L2 cluster centroid)

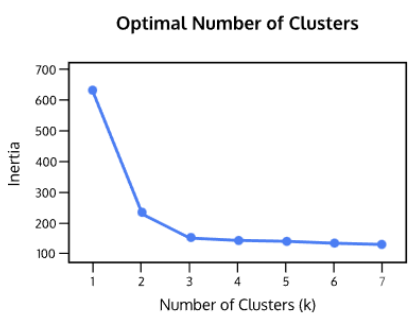
Given:

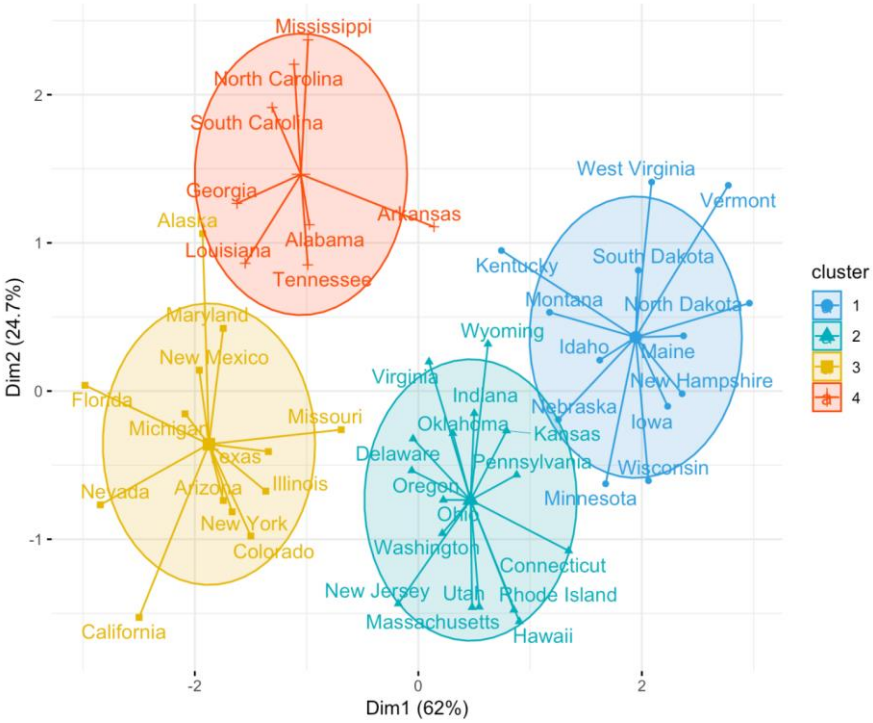
- n high dimensional points P_i where i = 1..n
- k << n clusters
- high dimensional Euclidean distance formula $|P_i - P_j|$

Find:

- Cluster labels for each point L_i = 1…k
- A low number of clusters and low inertia is ideal.
- Cluster centroids μ_j, where j=1…k minimizing the total within cluster variance (WCV) $\sum_i ||P_i - \mu_{L_i}||^2$

Notes:

- Z-score normalization
- Cluster labels used for GroupBy
- Centroid are means of group
- Type of unsupervised machine learning, groups data points into clusters based on similarity (Distance)
- Performance measured by inertia (distance between each data point and its centroid). Ideally low number of clusters and low inertia but as number of clusters increases, inertia decreases. Elbow method for optimal cluster count (3 in graph below). Look for the point before the stagnation of slope begins.
  - 



Steps

1) Normalize data
   a.
```
### normalize
quant = table.drop(columns='Type')
norm = (quant - quant.mean())/quant.std()
norm.head()
```
2) Use k-means clustering
   a.
```
## Cluster with KMeans
import sklearn.cluster

km = sklearn.cluster.KMeans(n_clusters=5)
labels = km.fit_predict(norm)
labels = pandas.DataFrame(labels, columns=['Cluster'], index=norm.index)
labels.sort_values('Cluster')
```
   b. Need to determine how many clusters are desired
   c. Results:
      i. Row and their cluster label:

1.

Name	Cluster
Elephant	0
Ox	0
Hippopotamus	0
Buffalo	0
Cow	0
Sheep	0
Rhinoceros	0
Pig	0
Moose	0
Giant Panda	0

ii. Number of rows in each cluster

1.

```
labels.Cluster.value_counts()
```

```
3    13
2    11
4    11
0     8
1     6
Name: Cluster, dtype: int64
```

iii. Inertia for 5 clusters

1.

```
## total within-cluster variance
km.inertia_
```

```
967.7135589348841
```

iv. Centroid stats for the 5 clusters

```
## cluster centroids (of normalized data)
pandas.DataFrame(km.cluster_centers_, columns=norm.columns)
```

	Active	Agility	Arctic	Bipedal	Bulbous	Coastal	Domestic	Fierce	Forager	Forest	...	Size	Smart	Speed	Spots	S
0	-0.051321	-0.165690	-0.439498	-0.031938	-0.079789	-0.478570	1.049872	-0.541573	-0.071006	-0.128840	...	-0.844449	-0.231671	-0.175100	0.132424	-0.2
1	-0.541880	-0.324242	1.816529	-0.221619	0.893240	1.742341	-0.331506	-0.235591	-0.675374	-0.850432	...	0.666640	0.348435	-0.223443	-0.060775	-0.2
2	0.968602	1.266157	-0.119330	-0.458377	-0.774621	-0.508861	-0.392616	1.819681	0.011518	0.045485	...	0.310478	0.513655	1.169673	0.335572	2.1
3	-0.886156	-0.934488	-0.401406	-0.518055	0.432262	-0.425529	-0.050510	-0.481663	-0.343477	-0.540522	...	0.761488	-0.875131	-0.795349	0.039168	-0.5
4	0.810153	0.669453	-0.347075	0.924266	-0.567305	-0.042913	-0.494561	0.225325	0.853056	1.202837	...	-0.587075	0.598378	0.519984	-0.287826	-0.0

Lloyd's algorithm for K-means

Steps

1. K random points as initial centroids: $\mu_j = P_{random}$
2. Assign each point to nearest centroid $L_i=j$ where $argmin_j||P_i-\mu_j||$
3. Recalculate centroids based on cluster membership $\mu_j=mean(\{P_i:L_i=j\})$
4. Repeat 2 until convergence (total centroid movement below threshold)

Notes:

- Heuristic optimization solution, local minima = different results based on randomized initialization
- Euclidean distance

- O(npki) where i=number of iterations, fast

Steps done without scikit:

1. Choose k random centroids:

```
### 1. Let initial k=2 random centroids be points A and B.
centroids = pandas.DataFrame([(0,0,0),(1,2,2)],
                             columns=['Cluster','X','Y',]).set_index('Cluster')
centroids
```

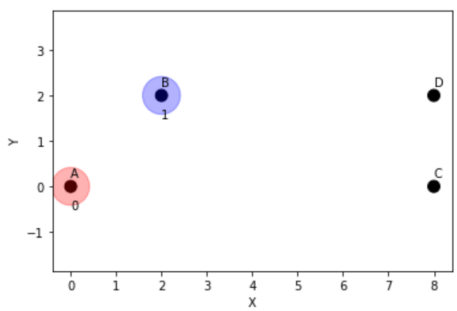
a.

	X	Y
Cluster		
0	0	0
1	2	2

2. Visualize centroids:

```
### visualize
ax = data.plot.scatter(x='X', y='Y', s=100, c=data.Cluster.map({None:'black',0:'red',1:'blue'}))
[ax.text(r.X, r.Y+0.2, i) for (i,r) in data.iterrows()]
centroids.plot.scatter(x='X', y='Y', s=1000, alpha=0.3, c=centroids.index.map({0:'red',1:'blue'}), ax=ax).axis('equal')
a=[ax.text(r.X, r.Y-0.5, i) for (i,r) in centroids.iterrows()]
```

a.



3. Assign points to nearest centroid

```
### 2. Assign points to nearest centroid
data.Cluster = [0, 0, 1, 1]
data
```

a.

	X	Y	Cluster
Point			
A	0	0	0
B	2	2	0
C	8	0	1
D	8	2	1

4. Recompute new centroids

```
### 3. Re-compute new centroids.
centroids.X = [ 1, 8]
centroids.Y = [ 1, 1]
centroids
```

a.

	X	Y
Cluster		
0	1	1
1	8	1

5. Repeat point assignment until no more changes occur
   a. $WCV = \sum_i ||P_i - \mu_{L_i}||_2$

Computational steps:

1. Run k-means algorithm:

```
### Run K-Means
km = sklearn.cluster.KMeans(n_clusters=2)
labels = km.fit_predict(data[['X','Y']])  # execute kmeans algorithm here
data['Cluster'] = labels
data
```

Point	X	Y	Cluster
A	0	0	0
B	2	2	0
C	8	0	1
D	8	2	1

a.

2. Look at cluster centroids and inertia

```
### cluster centroids
km.cluster_centers_
pandas.DataFrame(km.cluster_centers_, columns=['X','Y'])
```

	X	Y
0	1.0	1.0
1	8.0	1.0

a.

$$WCV = \sum_i ||P_i - \mu_{L_i}||^2$$

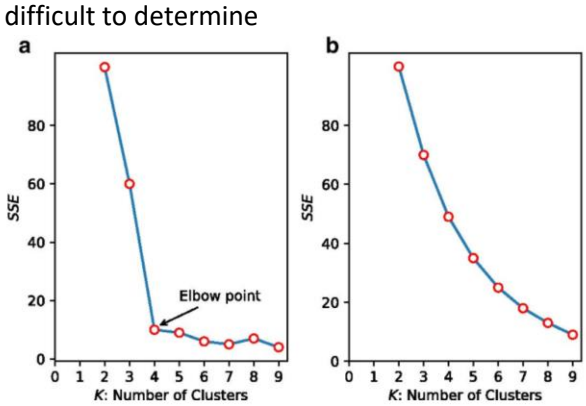
```
## total within-cluster variance
km.inertia_
```

```
6.0
```

b.

Elbow Method for number of clusters

- K selected at elbow
  - maximum ratio of slopes of adjecent segments k-1, k, k+1
  - k where $argmax_k \frac{WCV(k)-WCV(k-1)}{WCV(k+1)-WCV(k)}$
- Can be a difficult to determine



Hierarchical Agglomerative Clustering

- Hierarchical structure of clusters. Small group of cluster and a big cluster encompassing the group of clusters.

- Produces a dendrogram
- Iteratively group nearest neighbors until all points are in 1 big group.
    - Distance between points, L1, L2
    - Linkage: distance formula between 2 clusters
        - Single linkage: distance between nearest points
        - Complete: distance between farthest points
        - Centroid: distance between centroids (Euclidean)
- Steps
    - Determine which linkage method to use, distance metric to cluster rows, and the data rows themselves.

```
:
###
data[['X','Y']]
```

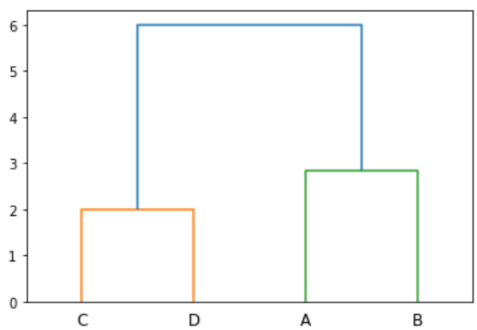
:

	X	Y
Point		
A	0	0
B	2	2
C	8	0
D	8	2

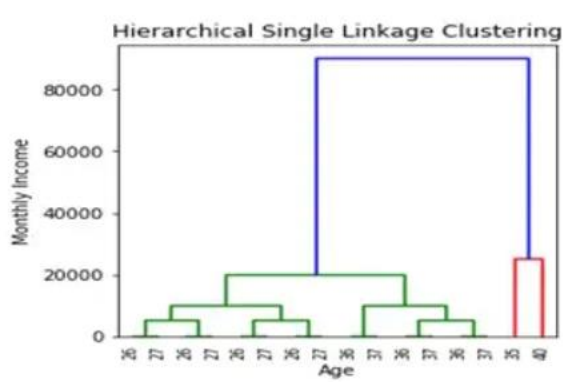
```
:
## Using scipy
import scipy.cluster.hierarchy
import matplotlib.pyplot

tree = scipy.cluster.hierarchy.linkage(data[['X','Y']], method='single', metric='euclidean')
scipy.cluster.hierarchy.dendrogram(tree, labels=data.index)
```

```
: {'icoord': [[5.0, 5.0, 15.0, 15.0],
   [25.0, 25.0, 35.0, 35.0],
   [10.0, 10.0, 30.0, 30.0]],
  'dcoord': [[0.0, 2.0, 2.0, 0.0],
   [0.0, 2.8284271247461903, 2.8284271247461903, 0.0],
   [2.0, 6.0, 6.0, 2.8284271247461903]],
  'ivl': ['C', 'D', 'A', 'B'],
  'leaves': [2, 3, 0, 1],
  'color_list': ['C1', 'C2', 'C0'],
  'leaves_color_list': ['C1', 'C1', 'C2', 'C2']}
```



-
- Outliers are the last ones to get converged as indicated in below for age 35 and 40 with high income.

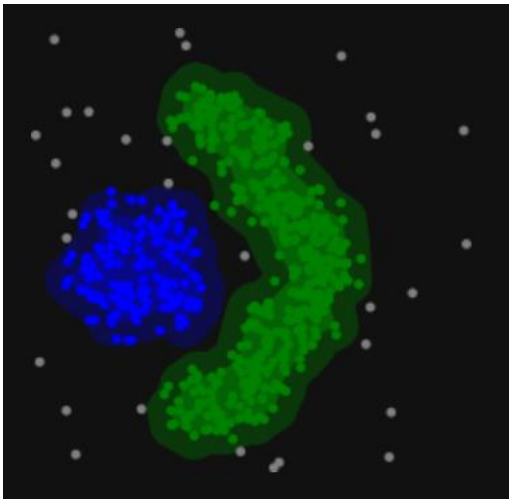


-

    - Color used correlates with threshold value (numerical axis). This is the distance between points and cluster (A is around 3 away from B)

Density-Based Clustering (DBSCAN)

- Goal: find densely packed group of points and ignore all outliers
- Density
    - A point has min_samples neighbors within a distance of eps
        - Min_samples: minimum number of neighbors
        - Eps: max distance to neighbors
    - Clusters with less than min_samples members are outliers
- Linear scan points
    - When a point is found, recursively scan its neighbors until cluster is completed.



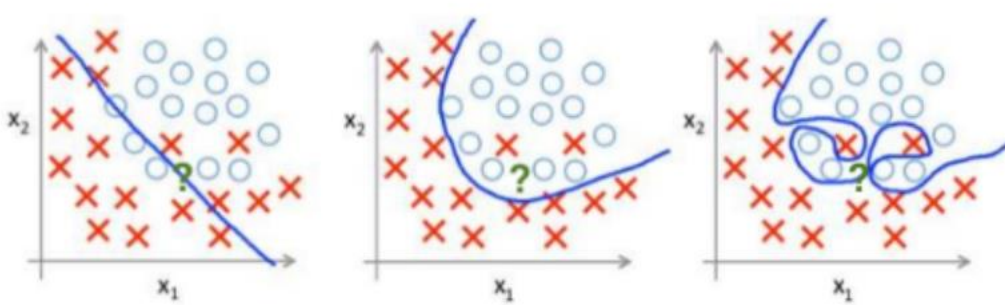
Process

- Create model with eps, min_samples, and distance metric
- Predict normalized data
-

```
import sklearn.cluster
# Create model first
dbscan = sklearn.cluster.DBSCAN(eps=5, min_samples=3, metric='euclidean') # Greater eps lowers outliers
labels = dbscan.fit_predict(norm) # -1 means outliers,
labels = pandas.DataFrame(labels, index=norm.index, columns=['Cluster'])
labels.sort_values('Cluster') # Around half datapoints are outliers, there are 2 clusters
```

Classification

Classification seeks to resolve issues with determining which class a data instance is part of:

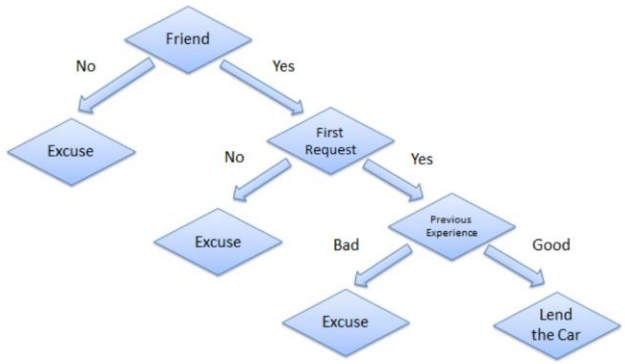


Decision Trees

- Simple and powerful
- Regression and classification

- Unstable

Trees work like a bunch of if else statements, if condition 1 is true, goto condition 2, else, end. The following scenario is about lending someone a car.



Advantage:

1. Inherently explainable
2. Regression and classification
3. Less training needed
4. Classify non linearly (non-similar) separable data.
5. Fast and efficient compared to other classification algorithms.

Steps:

1. Load a database
2. Make a dataframe where each row is an entity and columns are attributes
3. Generate a series where the entity type at position x correlates to attributes at row x in the previous database.
   a.
```
x = pandas.DataFrame(iris.data, columns=iris.feature_names)
y = pandas.Series(iris.target, name='Species').map(lambda s: iris.target_names[s])
x
```

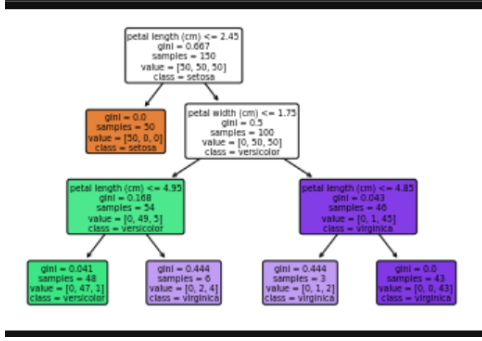
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

4. Create a sklearn decision tree classifier with depth equal to the number of decisions to make (entity types)
   a.
```
tree = sklearn.tree.DecisionTreeClassifier(max_depth=3)
```
5. Fit the tree to the dataframe and the series

a.
```
tree.fit(x, y)
```

6. You can plot the tree using plot_tree

a.
```
a = sklearn.tree.plot_tree(tree, filled=True, rounded=True,
                           feature_names=tree.feature_names_in_, class_names=tree.classes_)
```



7. Acquire prediction on rows using tree.predict(Rows)

a.
```
tree.predict([[5, 5, 1, 3] , [5, 5, 1, 1]])
```

```
C:\Anaconda\anaconda3\lib\site-packages\sklear
with feature names
  warnings.warn(
```

```
array(['setosa', 'setosa'], dtype=object)
```

Gini impurity score:

- Quantifies impurity of the node
- 0 when node is pure, within node, only single class of samples exists. This is the best gini score.
- >0 when samples contained are distributed into different classes.

Comparing models - Decision tree hyper parameters:

- Criterion: how impurity of a split is measured, default is gini but entropy can be used
- Splitter: how decision tree searches the features for a split
- Max_depth: max tree depth
- Min_samples_split: minimum number of samples a node needs to be considered a leaf
- Max_features: number of features to consider when looking for best split.

Confusion Matrix

- Compare frequencies of actual vs predicted

- actual on rows
- predicted on cols

- https://en.wikipedia.org/wiki/Confusion_matrix

Vowel produced \ Perceived vowel	i	e	a	o	u
i	15		1		
e	1		1		
a			79	5	
o			4	15	3
u				2	2

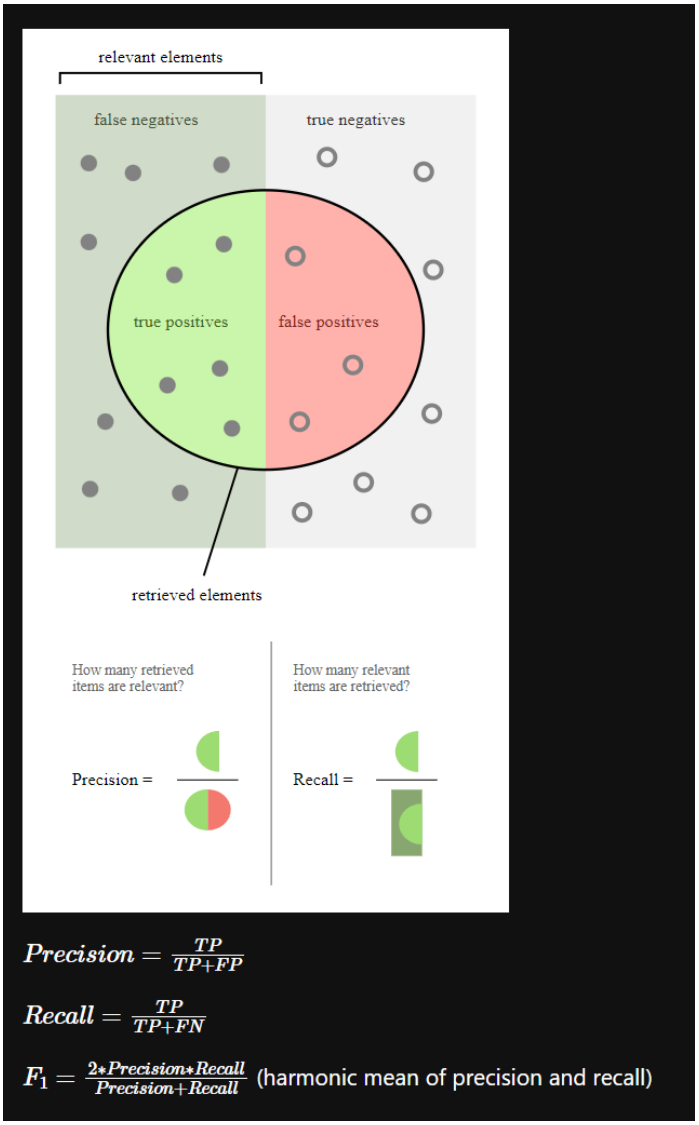
- Create confusion matrix by sklearn function with predicted labels and actual labels

```
conf = sklearn.metrics.confusion_matrix(y_test, y_pred, labels=tree.classes_)
conf = pandas.DataFrame(conf, index=tree.classes_, columns=tree.classes_)
conf
```

	setosa	versicolor	virginica
setosa	24	0	0
versicolor	0	24	0
virginica	0	1	26

Precision and Recall

- Measure classification accuracy (F-Score)



-
- Process
  - Print classification report and use above formula for F-score (it is automatically computed)
  -

```
print(sklearn.metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        24
  versicolor       0.96      1.00      0.98        24
   virginica       1.00      0.96      0.98        27

    accuracy                           0.99        75
   macro avg       0.99      0.99      0.99        75
weighted avg       0.99      0.99      0.99        75
```

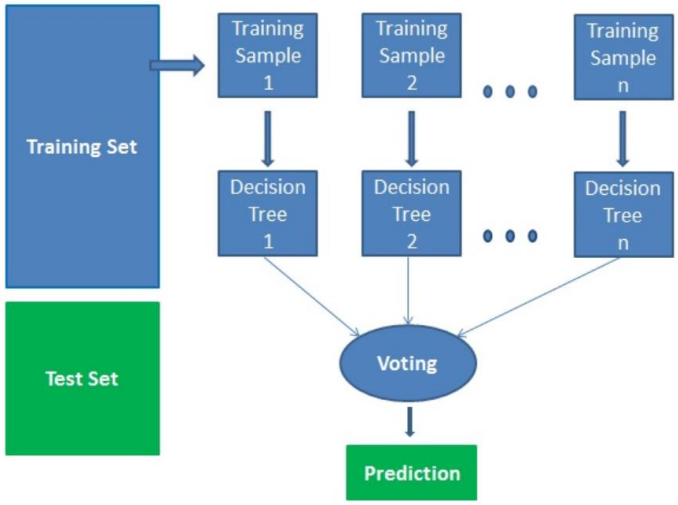
```
20 / 23
```

Random Forests

Decision trees are unstable due to data dependency, height, and other parameters. Random forests use multiple decision trees and relies on majority decision.

Process:

1. Select random samples from dataset
2. Construct a decision tree for each sample and get a prediction result from each data tree
3. Perform a vote for each predicted result
4. Select the prediction result with the most votes as final prediction.



Advantages:

- Highly accurate and robust
- No overfitting, takes average of all prediction that cancels the biases.
- Algorithm used in both classification and regression problems.
- Random forests handle missing values by median for continuous variables and proximity weighted average for missing values.
- Get relative feature importance for most contributing features for the classifier.

Disadvantages:

- Slow due to multiple trees, all forests need to process input and then vote.
- Hard to interpret compared to decision tree.

Steps:

1. Generate sklearn ensemble RandomForestClassifier with number of decision trees to use
   a. ```
      rf = sklearn.ensemble.RandomForestClassifier(n_estimators=49)
      ```
2. Fit to training data
   a. ```
      rf.fit(x_train, y_train)
      ```
3. Use model to predict testing data.
   a. ```
      y_predrf = rf.predict(x_test)
      ```
4. Generate confusion matrix and classification report for results.
      ```

a.

```
conf = sklearn.metrics.confusion_matrix(y_test, y_predrf, labels=tree.classes_)
conf = pandas.DataFrame(conf, index=tree.classes_, columns=tree.classes_)
conf
```

	setosa	versicolor	virginica
setosa	24	0	0
versicolor	0	24	0
virginica	0	1	26

```
print(sklearn.metrics.classification_report(y_test, y_predrf))
```

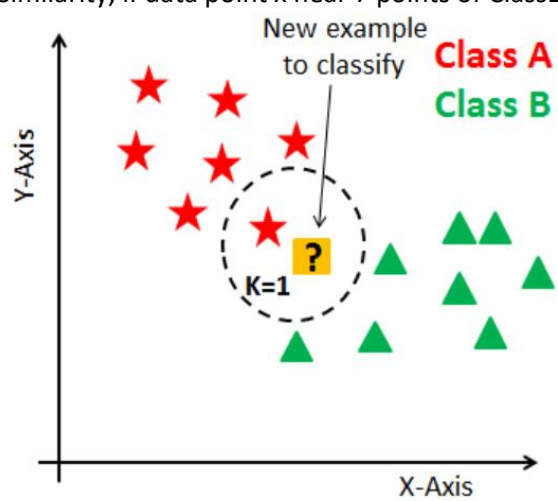
```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        24
  versicolor       0.96      1.00      0.98        24
   virginica       1.00      0.96      0.98        27

    accuracy                           0.99        75
   macro avg       0.99      0.99      0.99        75
weighted avg       0.99      0.99      0.99        75
```

K-Nearest neighbors (kNN)

- Similarity, if data point x near 7 points of Class1 and 5 points of Class2, likely x belongs to Class 1



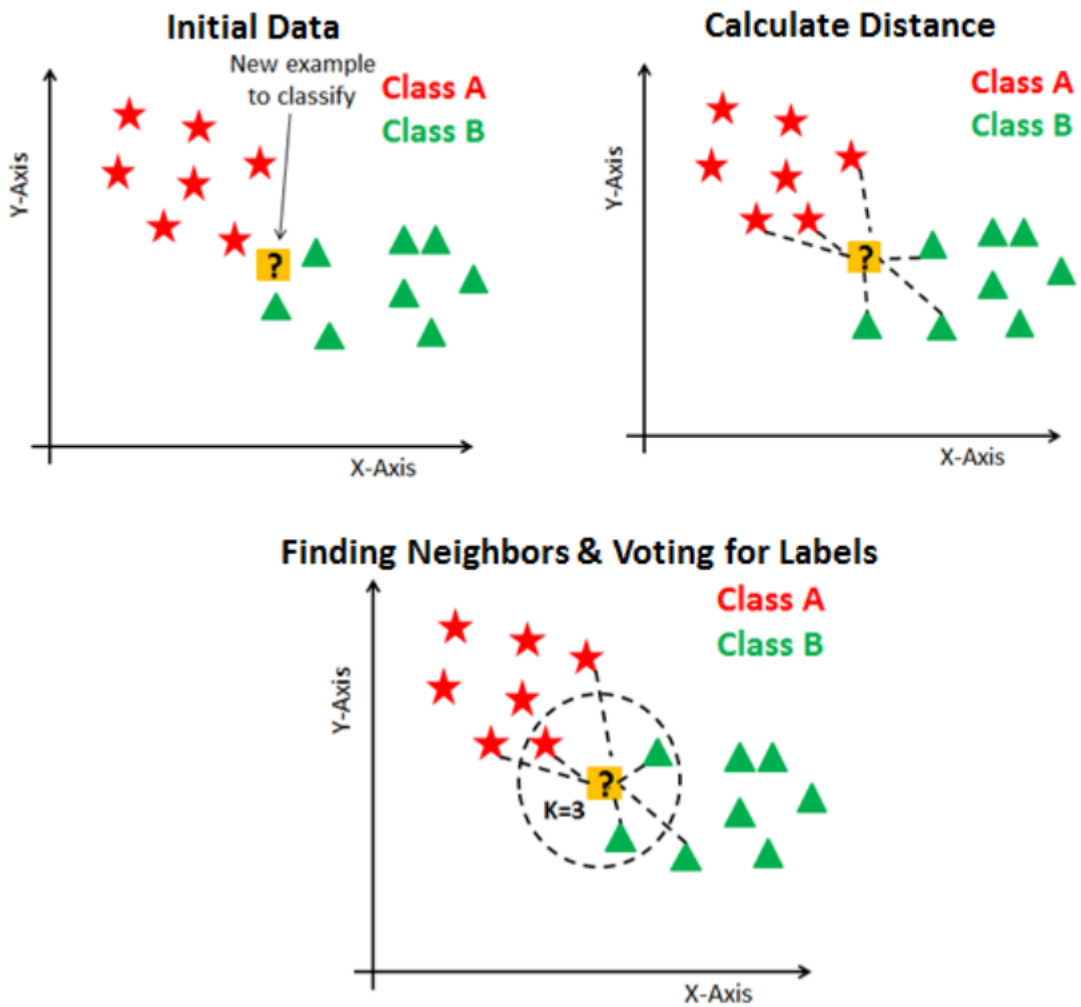
-

Non-parametric: no assumption for underlying data distribution, model determined from dataset

Lazy algorithm: no training data needed for model generation, all training data in testing phase, training fast but testing slow. KNN needs to scan all data points requiring memory.

Process:

1. Measure distance
2. Find closest labels
3. Vote.



Steps:

1. Generate sklearn.neighbors.KNeighborsClassifier with number of neighbors and distance as weight
   a.
   ```
   knn = sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='distance')
   ```
2. Fit on training data
   a.
   ```
   knn.fit(x_train, y_train)
   ```
3. Predict test data.
   a.
   ```
   y_predknn = knn.predict(x_test)
   y_predknn
   ```
4. Generate confusion matrix and classification report.

a.

```
conf = sklearn.metrics.confusion_matrix(y_test, y_predknn, labels=tree.classes_)
conf = pandas.DataFrame(conf, index=tree.classes_, columns=tree.classes_)
conf
```

	setosa	versicolor	virginica
setosa	24	0	0
versicolor	0	23	1
virginica	0	1	26

```
print(sklearn.metrics.classification_report(y_test, y_predknn))
```

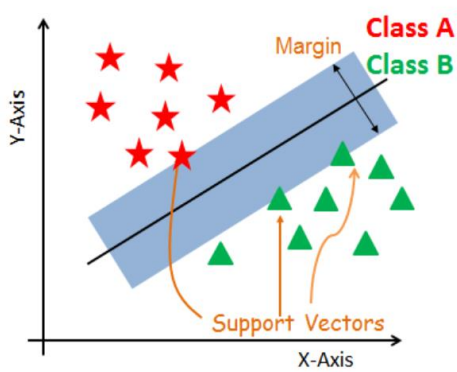
```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        24
  versicolor       0.96      0.96      0.96        24
   virginica       0.96      0.96      0.96        27

    accuracy                           0.97        75
   macro avg       0.97      0.97      0.97        75
weighted avg       0.97      0.97      0.97        75
```

Support Vector Machine (SVM)

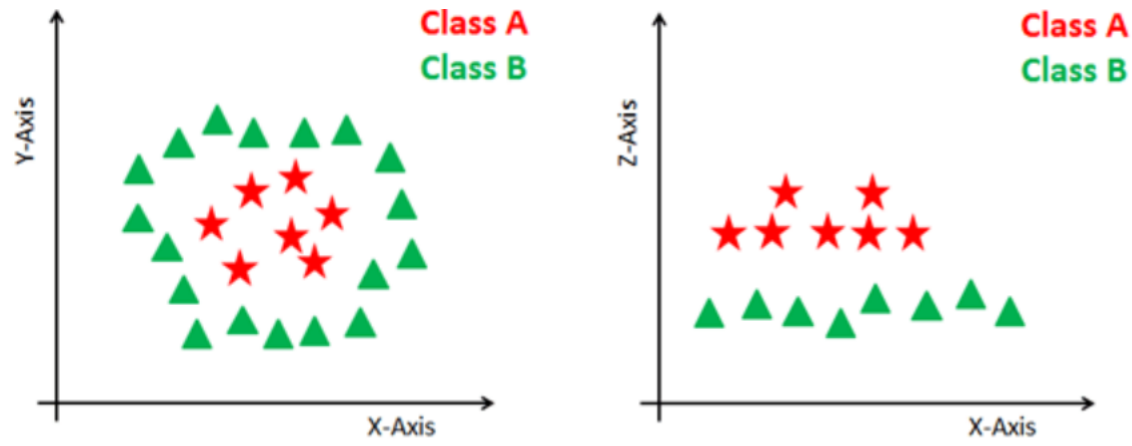
Like regression but dividing line between groups



Support Vector: data points closest to hyperplane, points define separating line better by calculating margins. Relevant to constructing of classifier.

Hyperplane: decision plane separating a set of objects from different classes.

Margin: gap between 2 lines on closest class points, perpendicular distance from line to support vectors. Larger margin is better than small margin.

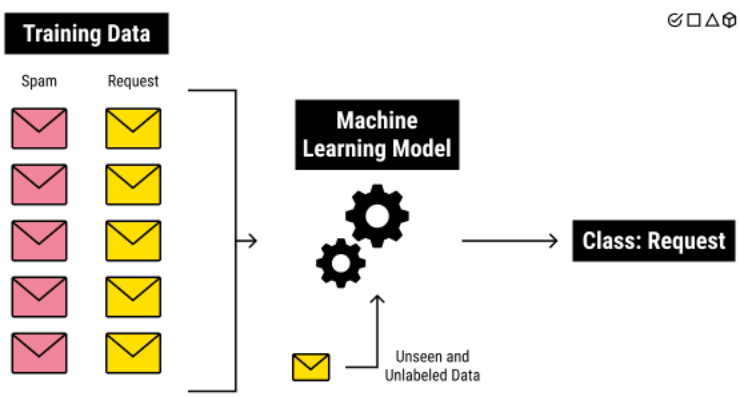


- When a hyperplane can't be used, kernel is changed.
- **Kernel**: transforms input data into required form.
- SVMs use kernel trick taking low dimensional input space, transforming into higher dimensional space.

Machine Learning:

Machine Learning Pipeline

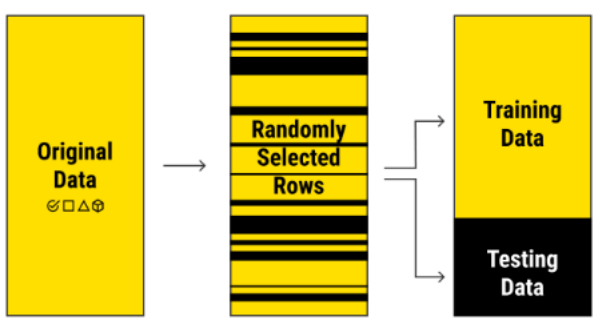
1. Training
2. Testing
3. Prediction



Evaluating Classification Models

Training and testing data:

- 80-20 split between training and testing data



-

- Evaluating prediction performance on training data leads to overfit

Steps:

1. Split data into training and testing data. Both data will have their own dataframe and series labeling the data.
2. Create a decision tree classifier with a certain amount of depth
3. Fit data only on training data, can have multiple training data.
4. You can plot the decision tree at this point.
5. Test data on test data without labels. This will return what the model thinks of each row attributes provided (labels).
6. The original labels can be appending to the dataframe with predicted labels and compared for accuracy.
7. Use a confusion matrix to compare frequencies of actual vs predicted:

	setosa	versicolor	virginica
setosa	24	0	0
versicolor	0	24	0
virginica	0	1	26

   a.
8. Measure classification accuracy by generating a classification report using testing and predicted labels.

Supervised vs. Unsupervised machine learning:

- Supervised: model trained by labeled datasets. Includes classification and regression.
- Unsupervised: algorithm detects patterns in unlabeled data and cluster related data. Includes clustering, association, and dimension reduction.

Text Analysis

Bag: unordered list, usually a list of terms in a document

Vocabulary: set of unique words in a bag

Steps for bag creation:

1. Parse and clean (whitespace, punctuation, nonalpha, case sensitivity, stemming, split into list)
   a. `text = text.replace('!', '').replace('.', '').replace('?', '').replace(',', '')`
   b. `text = text.lower()`
   c. `bag = text.split()`
2. Filter words (stop words, select useful words)
   a. `hashs = [w for w in bag if w.startswith('#')]`
3. Count word frequency.
   a. `vocab = list(set(hashs))`
   b. `freq = pandas.DataFrame([(w, hashs.count(w)) for w in vocab], columns=['hash', 'freq'])`
4. Generate bag by splitting sentences

a.

```
data['bag'] = data.tweet.map(lambda t: t.replace('|', '').replace('.', '').lower().split())
data
```

	tweet	username	datetime	latitude	longitude	id	bag
0	Calling all undergraduates interested in summe...	OurVt	2019-02-11 20:48:07	0.0	0.0	1095061945810669568	[calling, all, undergraduates, interested, in,...
1	@RATBABY_ Marisa has to pay for her rampant th...	BillySmilesXXL	2019-02-11 20:50:05	0.0	0.0	1095062439023005697	[@ratbaby_, marisa, has, to, pay, for, her, ra...
2	@djmouritzen @SasQuach_8 Well there's only one...	virginia_tech	2019-02-11 21:21:07	0.0	0.0	1095070250473848834	[@djmouritzen, @sasquach_8, well, there's, onl...
3	@therealdcunna Yes to*. Thank you David	CassellGuardVT	2019-02-11 21:26:36	0.0	0.0	1095071630257242112	[@therealdcunna, yes, to*, thank, you, david]
4	I turned 24.5 yesterday :/	vic_takes_twitr	2019-02-11 21:29:36	0.0	0.0	1095072383470321664	[i, turned, 245, yesterday, :/]

5. Add bag length

a.

```
data['len'] = data.bag.map(len)
data
```

	tweet	username	datetime	latitude	longitude	id	bag	len
0	Calling all undergraduates interested in summe...	OurVt	2019-02-11 20:48:07	0.0	0.0	1095061945810669568	[calling, all, undergraduates, interested, in,...	12
1	@RATBABY_ Marisa has to pay for her rampant th...	BillySmilesXXL	2019-02-11 20:50:05	0.0	0.0	1095062439023005697	[@ratbaby_, marisa, has, to, pay, for, her, ra...	17
2	@djmouritzen @SasQuach_8 Well there's only one...	virginia_tech	2019-02-11 21:21:07	0.0	0.0	1095070250473848834	[@djmouritzen, @sasquach_8, well, there's, onl...	21
3	@therealdcunna Yes to*. Thank you David	CassellGuardVT	2019-02-11 21:26:36	0.0	0.0	1095071630257242112	[@therealdcunna, yes, to*, thank, you, david]	6
4	I turned 24.5 yesterday :/	vic_takes_twitr	2019-02-11 21:29:36	0.0	0.0	1095072383470321664	[i, turned, 245, yesterday, :/]	5

Sentimental analysis

- Count positive and negative words
- Note that this does not use the separate bags in the matrix, it uses the one bag with all words in it
- Process
  - Get a list of positive and negative words
    -
    ```
    positives = set(open('data/positive.txt').read().split())
    negatives = set(open('data/negative.txt').read().split())
    ```
  - Get a list of words in bag which exists in the positive or negative list
    -
    ```
    p = [w for w in bag if w in positives]
    ```
  - Subtract number of positive words by number of negative words found
    -
    ```
    len(p) - len(n)
    ```
    ```
    29508
    ```

Vector Space Model

- Generate vocabulary.
- Vocabulary: list of unique terms (word) in document set

Document: vector of vocabulary term frequency

Matrix: set of documents

Term frequency matrix TF (Document x Term)

- Rows = documents
- Columns = terms in vocabulary
- Values = term frequency in document

Example: TF matrix

doc_1 = "hokies, go hokies"

doc_2 = "hokies beat duke"

TF matrix =

docID	hokies	go	beat	duke
doc_1	2	1	0	0
doc_2	1	0	1	1

Steps for TF matrix

1. Document aggregation
   a. Group documents by a certain characteristic (tweets and their username).
      i.
      ```
      g = data.groupby('username')
      ```
   b. Bags can be combined to create bigger bags.
   c. Perform groupby then aggregate on each group to aggregate each bag into a single large bag.
      i.
      ```
      userBags = g.bag.aggregate(lambda seriessofbags: [w for bag in seriessofbags for w in bag])
      ```
   d. Select the bags of certain length if desired.
      i.
      ```
      talkers = userBags[userBags.map(len) > 500]
      ```
2. Compute TF matrix
   a. Use apply() on groupby object. Function within apply returns a dataframe and apply combines the dataframes into a single dataframe.
      i.
      ```
      TF = talkers.apply(lambda talker: pandas.Series(talker).value_counts()
      ```
      f

TFIDF

Term Frequency Inverse Document Frequency

- Normalize terms by their frequency across all documents to emphasize discriminating terms and de-emphasize common terms.

DF: document frequency vector, measures number of documents each term appears in

IDF: Inverse document frequency vector: inverts DF

- IDF = log(n/DF)
- n: number of documents

- DF = 1 -> IDF = log(n)
- DF = n -> IDF – log(1) = 0
- Basically DF and IDF have a negative correlation

TFIDF: matrix computed from TF and IDF vector

Example: TF, IDF, TDIDF:

doc_1 = "hokies, go hokies"

doc_2 = "hokies beat duke"

doc_3 = "weather is good"

TF matrix =

docID	hokies	go	beat	duke
doc_1	2	1	0	0
doc_2	1	0	1	1

DF =

hokies	go	beat	duke
2	1	1	1

IDF =

hokies	go	beat	duke
0	1	1	1

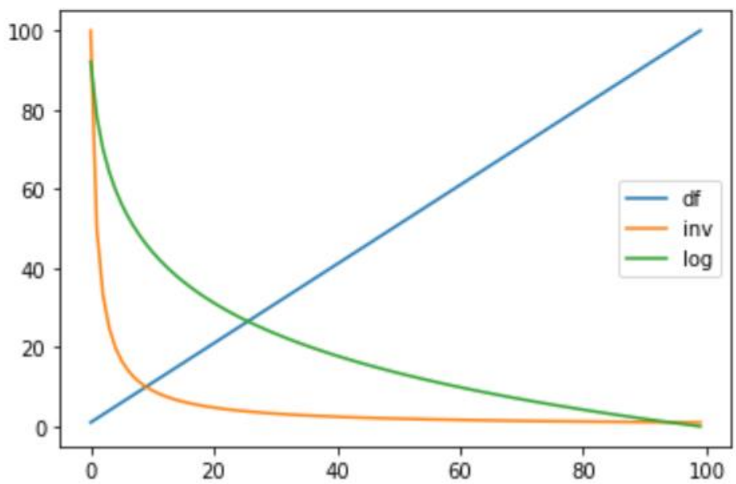
TFIDF matrix =

docID	hokies	go	beat	duke
doc_1	0	1	0	0
doc_2	0	0	1	1

Steps:

1. Compute DF through TF.count
   a. `DF = TF.count()`
2. Compute IDF through numpy.log(len(TF)/DF)
   a. `IDF = numpy.log(float(len(TF)) / DF)`
3. Compute TDIDF through TF * IDF
   a. `TFIDF = TF * IDF`

IDF Negative correlation visualized:



Cosine Similarity

Used to compare 2 documents or a query to a document

- Similar = colinear vectors
- Not similar = orthogonal
- Efficient for large and sparse VSM matrices

Similarity = d.e/|d||e|

- |d| = sqrt(d*d)
- d.e = dot product
- Similarity Range: 0 is orthogonal (not similar) and 1 is colinear (similar)
- Distance = 1 – similarity

Steps:

1. Acquire TFIDF matrix,
2. Get |d| with TFIDF as d
   a. `docmags = numpy.sqrt((TFIDF * TFIDF).sum(axis=1))`
3. Divide TFIDF matrix with |d|
   a. `nTFIDF = TFIDF.divide(docmags, axis=0)`
4. Fill all NaN with 0
   a. `nTFIDF.fillna(0.0, inplace=True)`
5. Compute dot product with matrix and the transpose of the same matrix
   a. `cosinm = nTFIDF.dot(nTFIDF.transpose())`
6. Subtract matrix with identity matrix of len(matrix)
   a. `cosinm2 = cosinm - numpy.identity(len(cosinm)`