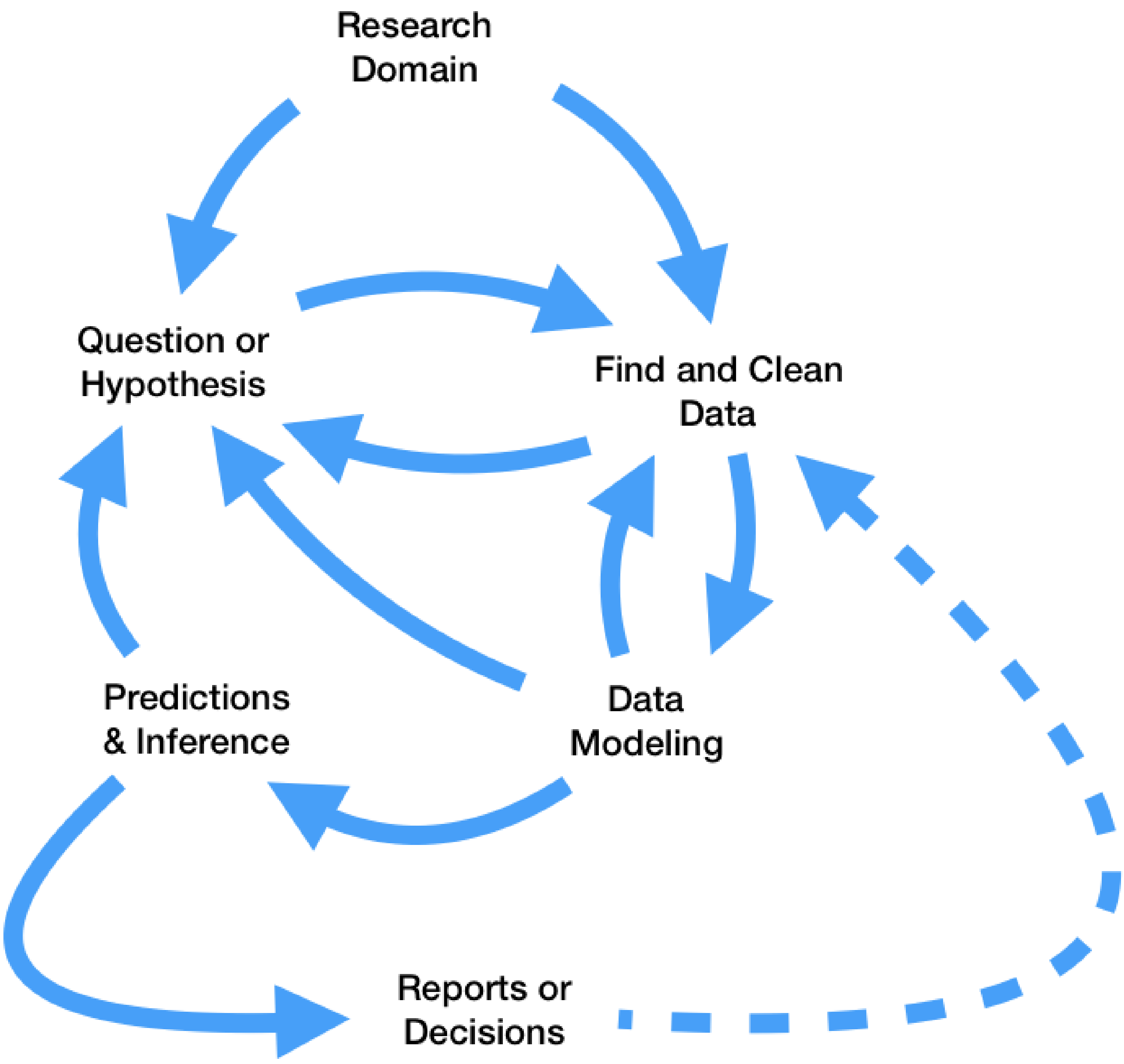
# DSC 80 Review Guide

Course Website **A Toolkit for DSC in practice!**

This document is made as a student studying in Suraj Rampure's DSC 80, 2022 Spring at UCSD.



- Some data collecting process is in appendix

## Pandas:

```python
import pandas as pd
import numpy as np
```

Check out the Official Website and the Official Website API

## *Data Type Comparison*

| Pandas dtype | Python type | NumPy type | SQL type | Usage |
|---|---|---|---|---|
| int64 | int | int_, int8,...,int64, uint8,...,uint64 | INT, BIGINT | Integer numbers |
| float64 | float | float_, float16, float32, float64 | FLOAT | Floating point numbers |
| bool | bool | bool_ | BOOL | True/False values |
| datetime64 | NA | datetime64[ns] | DATETIME | Date and time values |
| timedelta[ns] | NA | NA | NA | Differences between two datetimes |
| category | NA | NA | ENUM | Finite list of text values |
| object | str | string, unicode | NA | Text |
| object | NA | object | NA | Mixed types |

## *Series*

1 dimensional (columnar) array

### Series are "slices"

- Rows and columns of DataFrame are stored as `pd.Series`
- A `pd.Series` object is a one-dimensional sequence with labels (index).

### Initializing a Series

- the function `pd.Series` can create a new Series given either an existing sequence or dictionary.
  - When using `list`, the `pd.Series` directly use them in order
  - When using `dict`, the `pd.Series` will use dictionary key as index and use values for values
- You can consider the `name` attribute of a `pd.Series` like a name of "this column"

```python
# some important parameters
pd.Series(data = None, index = None, dtype = None, name = None)
```

- `data`: the list or dictionary for data
- `index`: a list like object for index
- `dtype`: specify the datatype of the `pd.Sereisobject`
- `name`: name of the `pd.Series`

# *Index*

immutable sequence of column/row labels

# *DataFrame*

2 dimensional tables

## Initialize

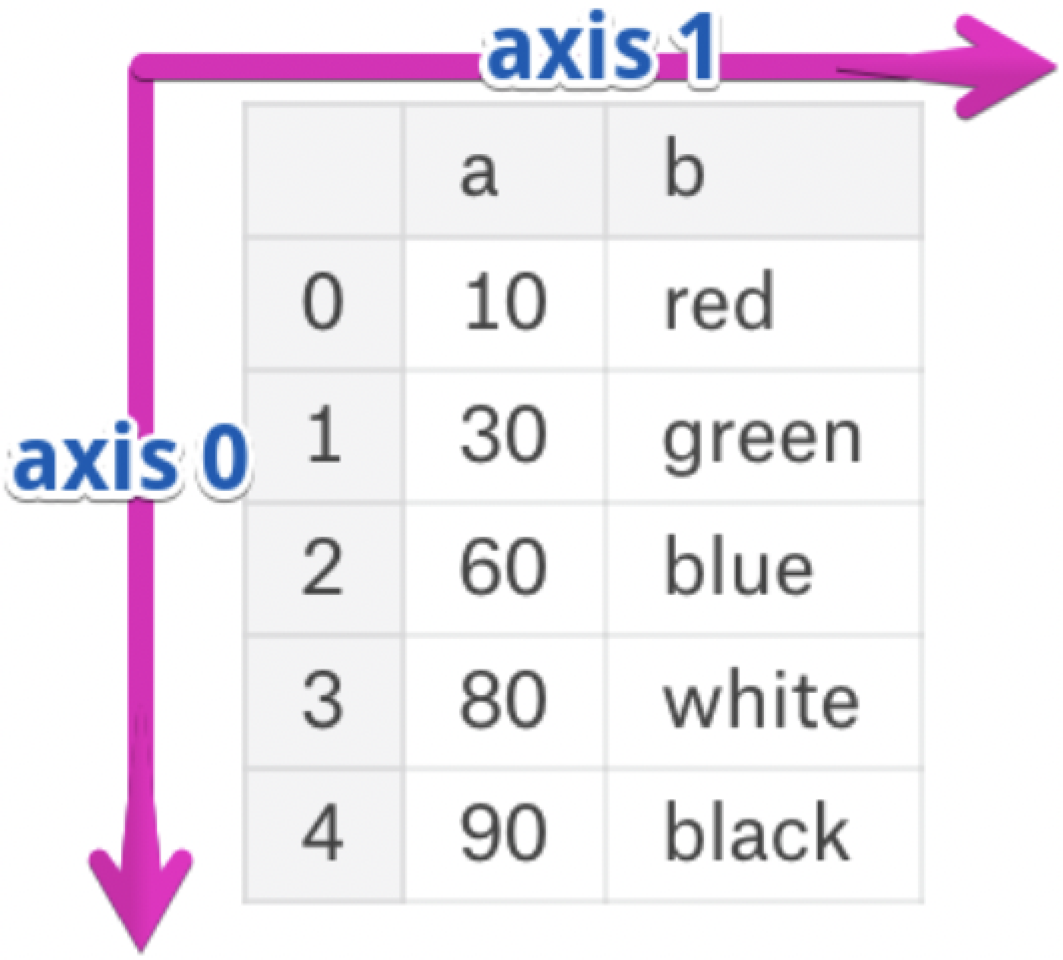
- `pd.DataFrame` initializes a DataFrame using either:
  - a list of rows (default integer row / column label), or
  - a dictionary of columns (consider each key-value for one column)

```
# some important parameters
pd.DataFrame(data = None, index = None, columns = None, dtype = None)
```

- `data`: list of rows / dictionary of columns... *ndarray, iterable...*
- `index`: a list-like object for index
- `columns`: a list-like object for columns
- `dtype`: specify the data type (only a single value is allowed)
  - Consider `astype` for manipulate

## Axis: Index & Columns

- rows and columns of a `DataFrame` are both stored as `Series`
- **axis** specify the direction of slicing:
  - Axis 0 refers to index
  - Axis 1 refers to columns



|   | a  | b     |
|---|----|-------|
| 0 | 10 | red   |
| 1 | 30 | green |
| 2 | 60 | blue  |
| 3 | 80 | white |
| 4 | 90 | black |

- See all the index in one `DataFrame`:
    - `df.index`
- See all the columns (name) in one `DataFrame`:
    - `df.columns`

## Selection

### Selecting Columns

- With column name:

```
df['col_name'] # this returns one column as a pd.Series object with selected col_name
df[[col_lst]] # this returns a dataframe with selected columns
```

- With column position

```
df.iloc[:, i] # this returns the ith column as a pd.Series object
df.iloc[:, [i_lst]] # this returns a dataframe with columns in the i position list
```

will explain the `.iloc` method later

### Selecting Rows

- With Index (row) name:

```
df.loc['index_name'] # this returns the row with index name as a pd.Series object
df.loc[[index_lst]] # this returns a dataframe with selected rows
```

- With row position

```
df.iloc[i] # this returns the ith row as a pd.Series object
df.iloc[[i_lst]] # this returns a dataframe with rows in the i position list
```

### Selecting Simultaneously

- All previous methods can directly go after another one, so simply use together can selecting simultaneously
- by `loc`

```
df.loc[<row selection>, <column selection>]
# the selection perform similar like before: single value or list
# this allows you to preform selection at one method:
# with row and column name
# allows ":" standing for "no conditions" (allows all to pass)
```

- by `iloc`

```
df.iloc[<integer row selection>, <integer column selection>]
# works the same as loc but replaced selection by position
```

### Selecting by Boolean Sequence

```
df[boolean_seq]
df.loc[boolean_seq]  # performs the same, all selecting on rows
# boolean seqeunce query will always result in dataframe even only one True
```

- when using `[]` after a dataframe with a boolean list, it will become a filter to filter out only rows with True values.
- The length of the sequence must be the same as the number of rows

### Querying

- Querying is the act of selecting rows in a DataFrame that satisfy certain condition(s).
- There are multiple methods (like comparison) can produce a boolean sequence from a DataFrame: use them as **selecting by boolean sequence**

**All previous discussing method will result in error when the value is missing in row/column as `ValueError`.**

## Efficiency

- loop over DataFrame rows is always very low in efficient: use vectorized operations (some are through NumPy)

```
# here provide some methods to measure time on IPython (jupyter notebook)

%timeit # this measures the time spending on the small piece of code come after

%time # this measures the time of first line of code in the cell

%%time # this measures the time of the entire cell
```

## Data Type

| Pandas dtype | Python type | NumPy type | SQL type | Usage |
|---|---|---|---|---|
| int64 | int | int_, int8,...,int64, uint8,...,uint64 | INT, BIGINT | Integer numbers |
| float64 | float | float_, float16, float32, float64 | FLOAT | Floating point numbers |
| bool | bool | bool_ | BOOL | True/False values |
| datetime64 | NA | datetime64[ns] | DATETIME | Date and time values |
| timedelta[ns] | NA | NA | NA | Differences between two datetimes |
| category | NA | NA | ENUM | Finite list of text values |
| object | str | string, unicode | NA | Text |
| object | NA | object | NA | Mixed types |

- `.dtypes` directly gives the datatype of a Series or DataFrame
- datatype determines what operations can be applied
- `pandas` make guess on datatype and is sometimes wrong: manually correct them for efficiency and calculation correction
- `pandas` like correctness and ease-of-use, it like to create type of `object` which preserved the original data types

### Type Conversion

`.astype`: change data type of a Series

```
# Some important parameters
Series.astype(dtype, copy = True, errors='raise')
DataFrame.astype(dtype, copy = True, errors='raise')
# errors: {'raise', 'ignore'}
```

- Both Object try to cast the entire object into one single type
- Notice the default value of `copy` is True, this **is not an in-place method**

### Memory & Performance Management

arbitrary assign NumPy data types for better memory management and fast computation.

## Useful Methods & Attributes

Some General Parameters:

- `inplace`: to decide whether this method works on the original object or on a copy
- `ignore_index`: reset the index of the returned object

### Overall Information

```
# DataFrame & Series shared methods [Using more default type to demonstrate]
DataFrame.head(i) # this gives the first i rows of dataframe, default 5
DataFrame.tail(i) # this gives the last i rows of dataframe, defualt 5
DataFrame.shape # return the shape (#row, #col) of dataframe
DataFrame.size # return the number of entries
Series.count() # return the number of non-null entries in the Series
Series.nunique() # returns the number of unique values in the Series
Series.value_counts() # returns a Series of counts of unique values
Series.describe() # returns a Series / DataFrame of descriptive stats of values
DataFrame.plot(x, t, kind) # see documentation below

# Method for Series only
Series.unique() # return the unique values in the Series

# Method for DataFrame only
DataFrame.info() # provides multiple information on the dataframe including column data type,
memory usage, ...
pd.plotting.scatter_matrix(df) # make sure no categorical columns in df:
    # Gives numerical data correlation toward each other.
```

- Documentation on Plotting (`df.plot()`)

### Easy Manipulations

```
# DataFrame & Series shared methods [Using more default type to demonstrate]
DataFrame.sort_values(by, ascending = True, inplace = False, kind = 'quicksort', ignore_index =
False)
    # Sort rows by specified column values
    # specify by which column when applying on DataFrame
    # kind: {'quicksort', 'mergesort', 'heapsort', 'stable'}
DataFrame.astype(dtype, copy = True, errors='raise') # errors: {'raise', 'ignore'}
DataFrame.drop_duplicates(keep={'first', 'last', False}, inplace=False, ignore_index=False)
    # drop duplicate rows
DataFrame.reset_index(drop=False, inplace=False) # drop: maintain the original index col?
DataFrame.fillna(value=None, method=None, axis=None, inplace=False)
    # method: {'backfill', 'bfill', 'pad', 'ffill', None}
    # axis: {0 or 'index', 1 or 'columns'}

# DataFrame only
DataFrame.assign(**dic) # assign new columns to a DataFrame with the use of dictionary
    # Returns a copy
    # Direct change on object: use a dictionary-like method
```

## *DataFrame Manipulation & Their Methods*

## Concepts

- **Granularity** refers to the level of detail present in data.
  - Fine: small details
  - Coarse: bigger picture
  - Data Creation should seek for **finer granularity**

## Grouping & GroupBy

*Manipulate Granularity*

Three Steps: **Split, Apply, Combine** in `groupby` method

- **Split** breaks up and "groups" the rows of a DataFrame according to the specified key. There is one "group" for every unique value of the key.
- **Apply** uses a function (e.g. aggregation, transformation, filtering) within the individual groups.
- **Combine** stitches the results of these operations into an output DataFrame.

### "split": `GroupBy` Object

```
df.groupby(key)
```

**Methods and Attributes**:

- `.groups`: a dictionary in which the keys are group names and the values are lists of row labels
- `.get_groups(key)`: a DataFrame with only the values for the given key
- to obtain groups with for-loop:

  -
    ```
    df_groups = df.groupby(key)
    for group, sub_df in df_groups:
        # group is the group lable in key
        # sub_df is the DataFrame in the group
    ```

### "apply & combine": Aggregate Methods

- The most common operation applied to each group is an **aggregation**
  - Aggregation refers to the process of reducing many values to one: **increase granularity**
- To perform an aggregation, use an aggregator method on the `DataFrameGroupBy` object
  - e.g. `.mean()`, `.max()`, `.median()`
  - check official documentation for full list of aggregate methods

**Column Selection**

- By default, the aggregator will be applied to **all** columns that it can be applied to.
  - `max` and `min` are defined on strings, while `median` and `mean` are not.
- If we only care about one column, we can select column before aggregating to save time: `DataFrameGroupBy` objects support `[]` notation
  - `df.groupby(key)[col]`

**Other Important Aggregate Methods**

- `aggregate`: the documentation
  - aggregates using one or more operations. Examples:

        -
          ```
          df.groupby(key).aggregate({"col_name1": "method", "col_name2": "method"})
              # above: apply different agg methods toward different columns
          df.groupby(key).aggregate(["method1", "method2", ...])
              # above: apply multiple agg methods on the groupby objects
          ```

    - agg is the same as `aggregate`
- `transform`: the documentation
    - split into groups and apply operations for each group, then return a DataFrame with original size
- `filter`: the documentation
    - Keep only groups that satisfy a particular condition
        - Use a `boolean` function to filter
- `apply`: the documentation
    - The `apply` method is a generalization of `aggregate`, `transform`, and `filter`.
    - it is slower than other aggregation and transformation methods, so use those instead whenever possible, and **avoid `apply`**.

## Pivoting and Pivot Table

Aggregate based on two columns.

- `pivot_table` aggregates a DataFrame into a spreadsheet-style table using two columns: the documentation
    - Rows are no longer correspond to observations

```
df.pivot_table(index=index_col,
               columns=columns_col,
               values=values_col,
               aggfunc=func)
```

- `pivot` only reshap DataFrame organized by given index / column values.
    - Now both rows and columns helps identify one specific observations

```
df.pivot(index=index_col,
         columns=columns_col,
         values=values_col)
```

**Other Reshaping Methods**:

- `melt`: un-pivots a DataFrame
- `stack`: pivots multi-level columns to multi-indicies.
- `unstack`: pivots multi-indices to columns
- ...

## Combining Data

### `pd.concat`: Simple Combine

Check out the documentation

```
# only list some important parameters
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, names=None, sort=False,
copy=True) # join: {'inner', 'outer'}
```

- **Vertical Concatenating**: by default, the **rows of objects are stacked on top of one another**.

    - `objs` can be a list of DataFrames (even with Series)
    - index will keep if `ignore_index=False`
    - if columns cannot match: will keep both columns and fill with `np.NaN` values
    - `pd.concat` is not super efficient, combine all elements only once will be a better choice
- **Horizontal Concatenating**: when `axis = 1`
    - concatenate the columns: **direct matching indexes regardless of order and any column values**

### `pd.merge`: Combine with Values

Check out the documentation

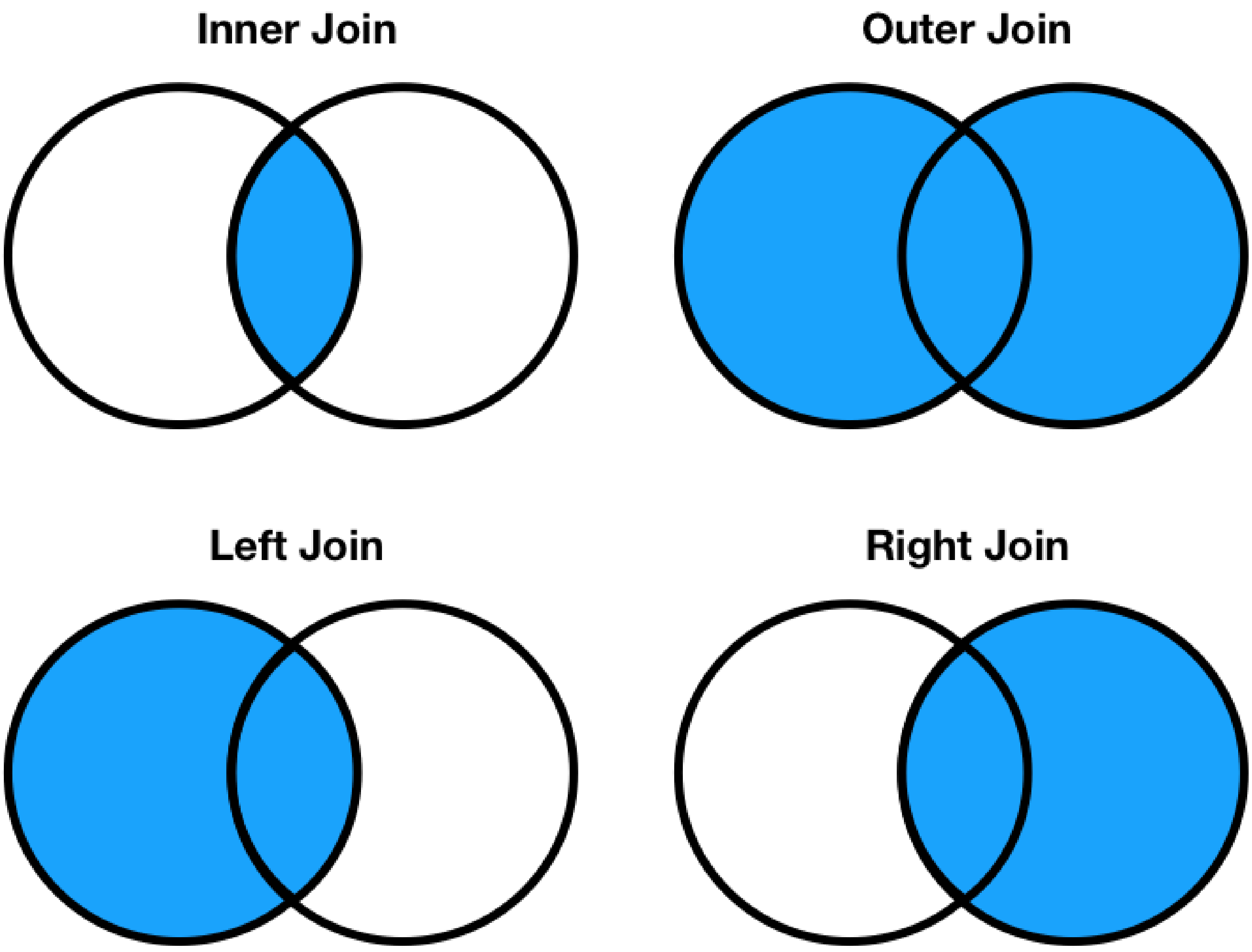
```
# only important parameters
pd.merge(left, right, how='inner', on, left_on, right_on, left_index, right_index, sort=False,
copy=True, validate) # how: {'inner', 'outer', 'left', 'right'}
    # validate: {"one_to_one" or "1:1", "one_to_many" or "1:m", "many_to_one" or "m:1",
"many_to_many" or "m:m"}
```

Explain: **Join (merge)**:

- A **join** creates a new DataFrame by combining the rows of two DataFrames.
- A join is appropriate when we have two sources of information
    - about the same individuals, that is
    - linked by a common column.
- The common column is called the **join key**.

**Different join types handle mismatches differently**: There are four types of joins.

- **Inner:** keep **only** matching keys (intersection).
- **Outer:** keeps **all** keys in both DataFrames (union).
- **Left:** keep all keys in the left DataFrame, whether or not they are in the right DataFrame.
- **Right:** keep all keys in the right DataFrame, whether or not they are in the left DataFrame.



**Different number of values**: (Many-to-one & many-to-many joins)

- Will lead to multiplication of all value combinations

## Exploratory Data Analysis

### *General Discussion on* ***Data***

*Data are the result of measurements that must be recorded.*

- Data is **always** an imperfect record of the underlying processing being measures

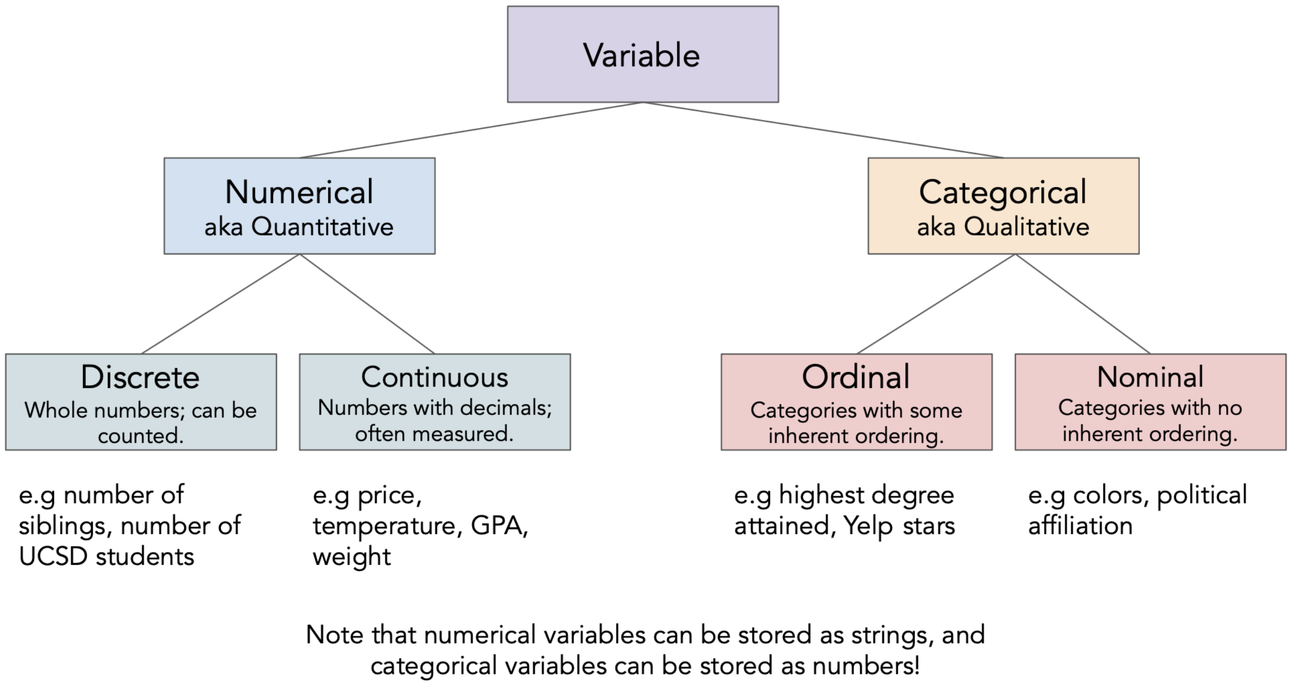
**Data Generating Process** is the underlying, real-world (probabilistic) mechanism that generates observed data.

- Observed is an incomplete artifact of the data generating process
- **A data generating process is what a statistical model attempts to describe.**

It's also important to understand the "story" of how a dataset came to be, or the **provenance** (**can we trust our data?**) of the data. Specifically:

1. Assumptions about the data generating process
2. How the initial values in the dataset came to be
3. How many data processing or storage decisions affected the values in the dataset.

## *Kinds of Data*


Variable
Numerical
aka Quantitative
Categorical
aka Qualitative
Discrete
Whole numbers; can be counted.
Continuous
Numbers with decimals; often measured.
Ordinal
Categories with some inherent ordering.
Nominal
Categories with no inherent ordering.
e.g number of siblings, number of UCSD students
e.g price, temperature, GPA, weight
e.g highest degree attained, Yelp stars
e.g colors, political affiliation
Note that numerical variables can be stored as strings, and categorical variables can be stored as numbers!

## *Missing Data*

### Relative Methods

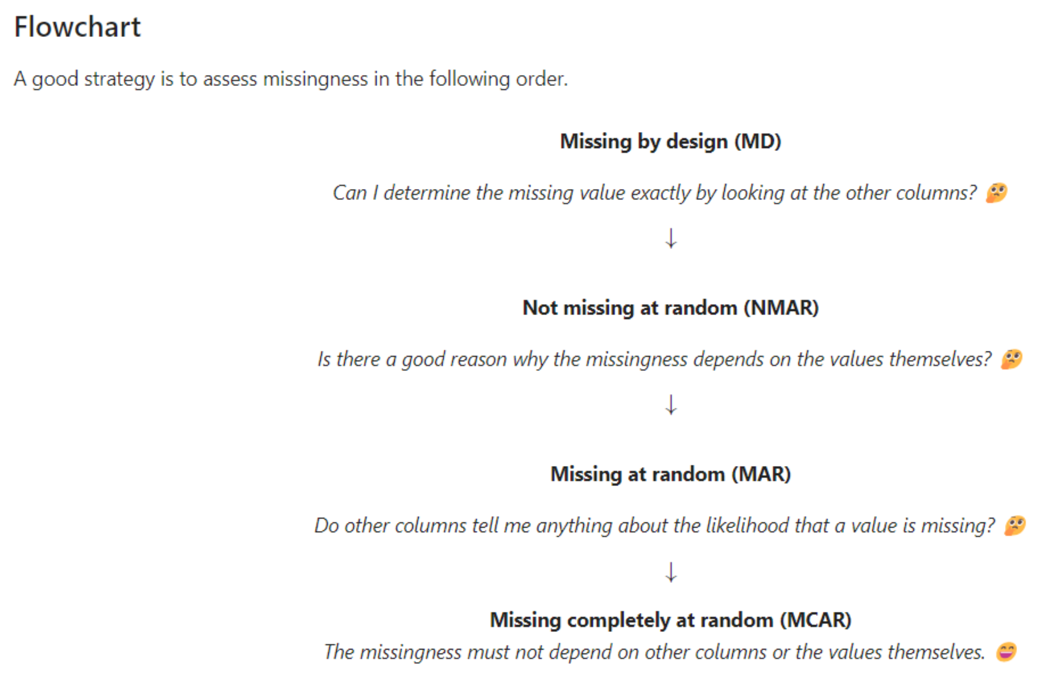
```
pd.isnull()
df.isna()
```

## *Types of Missingness*

There are four key ways in which values can be missing. It is important to distinguish between these types so that we can correctly **handle** missing data (Lecture 13).

- **Missing by design (MD)**.
  - the designers of the data collection process **intentionally decided to not collect data in that column**,
  - because it can be recovered from other columns.
- **Not missing at random (NMAR)**.
  - Also called "non-ignorable" (NI), especially in Lab 4.
  - The chance that a value is missing **depends on the actual missing value**
- **Missing at random (MAR)**.
  - Also called "conditionally ignorable".
  - The chance that a value is missing **depends on other columns**, but **not** the actual missing value itself.
- **Missing completely at random (MCAR)**.
  - Also called "unconditionally ignorable".
  - The chance that a value is missing is completely **independent** of
    - other columns, and

- the actual missing value.



### Flowchart

A good strategy is to assess missingness in the following order.

**Missing by design (MD)**

*Can I determine the missing value exactly by looking at the other columns?* 🤔

↓

**Not missing at random (NMAR)**

*Is there a good reason why the missingness depends on the values themselves?* 🤔

↓

**Missing at random (MAR)**

*Do other columns tell me anything about the likelihood that a value is missing?* 🤔

↓

**Missing completely at random (MCAR)**
*The missingness must not depend on other columns or the values themselves.* 😄

# *Tests*

## Hypothesis Test

### Null Hypothesis

an initial or default belief as to how data were generated.

- The null hypothesis must be a **probability model.** i.e. something that we can simulate under.

### Alternative Hypothesis

a different viewpoint as to how data were generated.

### Test Statistics

The number that we compute in each repetition of an experiment, to help us make a decision

- Try to **avoid "two-tailed tests"**
  - Pure Number
  - Proportions
  - Difference from expected number

**Total Variation Distance**: *describes the distance between two categorical distributions.*

- Defined as "**difference between the two proportions in each category, add up the absolute values of all the differences, and then divide the sum by 2**"

### Cutoffs:

- or **significance level**: the threshold for rejecting hypothesis or not based on p-value

### Example of Hypothesis Tests

```python
# set up data: simulation on flipping a coin
np.random.seed(42)
flips = pd.DataFrame(np.random.choice(['H', 'T'], p=[0.55, 0.45], size=(114, 1)), columns=
['result'])
flips.head()

# Step 1: Generate Null Hypothesis Distribution
# 100,000 times, we want to flip a coin 114 times
results = []

for _ in range(100000):
    simulation = np.random.choice(['H', 'T'], size=114)
    sim_heads = (simulation == 'H').sum()  # Test statistic
    results.append(sim_heads)

# Step 2: Calculate the p-value
p = (np.array(results) >= obs).mean()

# Step 3: Plotting the Empircal distribution of the test statistic
pd.Series(results).plot(kind='hist',
                        density=True,
                        bins=np.arange(35, 76, 1),
                        ec='w',
                        title='Number of Heads in 114 Flips of a Fair Coin');
obs = (flips['result'] == 'H').sum()
plt.axvline(x=obs, color='red', linewidth=2);

# Speed Everything Up
flips_fast = flips.replace({'H': 1, 'T': 0})
def flip_114(N):
    return np.random.choice([1, 0], size=(N, 114))

# Flips a fair coin 100,000 * 114 times
simulations = pd.DataFrame(flip_114(100000))

# Compute test statistics
# Note that axis=1 will take the sum of each row of 114, which is what we want
results_fast = simulations.sum(axis=1)

# function for calculating total_variation_distance
def total_variation_distance(dist1, dist2):
    '''Given two categorical distributions,
    both sorted with same categories, calculates the TVD'''
    return np.sum(np.abs(dist1 - dist2)) / 2
```

### Notes:

- We **can't prove the null**:
  - We can only either **reject the null** or **fail to reject the null**
  - Cheek out the speed up example code on the lecture notes
    - Basic Idea: whenever you can direct generate a 2-D array representing the whole random process by NumPy, it will be more efficient.

### The hypothesis testing "recipe" (conclusion)

Faced with a question about the data raised by an observation...

1. Carefully pose the question as a testable "yes or no" hypothesis.
2. Decide on a **test statistic** that helps differentiate between instances that would affirm or reject the hypothesis.
3. Create a probability model for the data generating process that reflects the "known behavior" of the process.
4. Simulate the data generating process using this probability model (the "**null hypothesis**").
5. Assess if the observation is consistent with the simulations by computing a **p-value**.

---

## Permutation Test

**Given two observed samples, are they fundamentally different, or could they have been generated by the same process?**

### Null Hypothesis:

Two observed samples are from the same distribution.

- We don't know what the null distribution is.

### Alternative Hypothesis:

Two observed samples are not from the same distribution.

### Test Statistics

- **Difference in group means**.
- **TVD**
- **Kolmogorov-Smirnov test statistic**
  - roughly defined as the **largest difference between two CDFs**
    - **CDF**: cumulative distribution function

```
# to calculate KS test statistic
from scipy.stats import ks_2samp
obs_ks = ks_2samp(gpA, gpB).statistic
```

### Example of Permutation Test

```
# smoking_and_weight is a dataframe with whether is a smoker and baby birth weight

# visualize distribution
title = "Birth Weight by Mother's Smoking Status"

(
    smoking_and_birthweight
    .groupby('Maternal Smoker')['Birth Weight']
    .plot(kind='hist', density=True, legend=True,
          ec='w', bins=np.arange(50, 200, 5), alpha=0.75,
          title=title)
);

# begin of permutation test
n_repetitions = 500
```

```
differences = []
for _ in range(n_repetitions):

    # Step 1: Shuffle the weights
    shuffled_weights = (
        smoking_and_birthweight['Birth Weight']
        .sample(frac=1)
        .reset_index(drop=True) # Be sure to reset the index! (Why?)
    )

    # Step 2: Put them in a DataFrame
    shuffled = (
        smoking_and_birthweight
        .assign(**{'Shuffled Birth Weight': shuffled_weights})
    )

    # Step 3: Compute the test statistic
    group_means = (
        shuffled
        .groupby('Maternal Smoker')
        .mean()
        .loc[:, 'Shuffled Birth Weight']
    )
    difference = group_means.diff().iloc[-1]

    # Step 4: Store the result
    differences.append(difference)

# Plot Permutation Test Result
title = 'Mean Differences in Birth Weights (Smoker - Non-Smoker)'
pd.Series(differences).plot(kind='hist', density=True, ec='w', bins=10, title=title)
plt.axvline(x=observed_difference, color='red', linewidth=3);

# Speed Things Up
is_smoker = smoking_and_birthweight['Maternal Smoker'].values
weights = smoking_and_birthweight['Birth Weight'].values
n_smokers = is_smoker.sum()
n_non_smokers = 1174 - n_smokers

is_smoker_permutations = np.column_stack([
    np.random.permutation(is_smoker)
    for _ in range(3000)
]).T

mean_smokers = (weights * is_smoker_permutations).sum(axis=1) / n_smokers
mean_non_smokers = (weights * ~is_smoker_permutations).sum(axis=1) / n_non_smokers
ultra_fast_differences = mean_smokers - mean_non_smokers
```

## Notes:

- **Cannot** directly conclude the **causes** (be aware of possible cofounding factors)
- **We only reject (or fail to reject) the null hypothesis that the two groups come from the same distribution**.
- There are shortcuts to use **pivot_table** for calculate **TVD** when dealing with categorical values
- Check out the lecture 10 note for basic concepts, check out the lecture 11 note for speeding up, check out the lecture 12 note for Kolmogorov-Smirnov test statistic

### The permutation test "recipe" (conclusion)

- In a **permutation test**, we generate new data by **shuffling group labels** .
- On each shuffle, we'll compute our test statistic.
- If we shuffle many times and compute our test statistic each time, we will approximate the distribution of the test statistic.
- We can them compare our observed statistic to this distribution, as in any other hypothesis test.

## SKLearn

Check out the Official Website and the Official Website API

Scikit-learn: simple and efficient tools for predictive data analysis; free software machine learning library

Later Summary will directly use SKLearn in practice.

## Data Preparation before Feature Engineering

### *Data Cleaning*

Data Cleaning is the process of transforming data so that it best represent the underlying **data generating process**.

- Data Cleaning requires an understanding of the **data generating process** and the **provenance**.
- Keys to Data Cleaning
  - The **structure** of the recorded data.
  - The **encoding** and **format** of the values in the data.
    - Change to correct **data structure**
  - Corrupt and "**incorrect**" data, and missing values
    - Does the data contain unrealistic values: that said is it **"faithful"** to the DGP?
      - Does the data violate obvious dependencies?
      - Was the data entered by hand?
      - Are there obvious signs of data falsification (aka "curbstoning")?
      - Outliers are not Unfaithful Data:
        - Consistently "incorrect" values
        - Abnormal artifacts from the data collection process
        - Unreasonable Outliers
    - Missing Values: lost values (leaving blank)
      - Correct identify them with different forms
      - formally use `np.NaN` to represent

#### Relative Methods

```
# convert to numerical data
pd.to_numeric(arg, errors)
    # errors: {'ignore', 'raise', 'coerce'} (coerce will give np.NaN)
# convert to datetime
pd.to_datetime(args, errors, dayfirst, yearfirst, utc, format, exact)
    # errors: {'ignore', 'raise', 'coerce'} (coerce will give np.NaN)
# str methods
pd[col].str # this allows applies of functions on indivdual row (string) values
str.zfill(l) # adds zeros to the start of a string until it reaches specified length
str.split(char) # split by speicifed char

#  naive methods for missing values: drop & fill
df.dropna(axis, how='any', inplace=False) # how: {'any', 'all'}
df.fillna(value=None, method=None, axis=None, inplace=False)
```

## *Dealing With Missing Values*

### Dropping missing values can be one solution:

- unbiased only when the data is **MCAR**

## Solution 2: Imputation

### Kinds of imputation

- There are three main types of imputation, two of which we will focus on today:
  - **Imputation with a single value: mean, median, mode.**
  - **Imputation with a single value, using a model: regression, kNN.**
  - **Probabilistic imputation by drawing from a distribution.**
- Each has upsides and downsides, and **each works differently with different types of missingness**.

#### Mean imputation

- Mean imputation is the act of filling in missing values in a column with the mean of the observed values in that column.
- This strategy:
  - 👍 Preserves the mean of the observed data, for all types of missingness.
  - 👎 Decreases the variance of the data, for all types of missingness.
  - 👎 Creates a biased estimate of the true mean when the data are not MCAR.

-
  ```
  df_filled = df.fillna(df[col].mean())
  ```
- Improvements: within-group (conditional) mean imputation

**Probabilistic imputation**:

Imputing missing values using distributions

- So far, each missing value in a column has been filled in with a constant value.
  - This creates "spikes" in the imputed distributions.
- Idea: We can **probabilistically** impute missing data from a distribution.
  - We can fill in missing data by drawing from the distribution of the ***non-missing** data.
  - There are 5 missing values? Pick 5 values from the data that aren't missing.
    - How? Using `.sample`.

-
  ```python
  # example code
  fill_values = heights_mcar.child.dropna().sample(num_null, replace=True)

  # Find the positions where values in heights_mcar are missing
  fill_values.index = heights_mcar.loc[heights_mcar['child'].isna()].index

  # Fill in the missing values
  heights_mcar_dfilled = heights_mcar.fillna({'child': fill_values.to_dict()})  # fill the vals
  ```

**Multiple Imputation**:

Steps:

1. Start with observed and incomplete data.
2. Create several **imputed** versions of the data through a probabilistic procedure.
   - The imputed datasets are identical for the observed data entries.
   - They differ in the imputed values.
   - The differences reflect our **uncertainty** about what value to impute.
3. Then, estimate the parameters of interest for **each** imputed dataset.
   - For instance, the mean, standard deviation, median, etc.
4. Finally, pool the m parameter estimates into one estimate.

# *Dealing With Text Data*

## Quantifying Text Data

*How do we turn a text document into a vector of numbers?*

- A **design matrix** consists of one row per "data point", and one column per "feature"

### Step 1: Punctuation

unnecessary punctuation that we can remove

### Step 2: "Glue" words

Remove words like "to", "the", and "for"

### Step 3: Abbreviations (or akas)

They should be turned into the same format: requires manual labor....

### To canonicalize job titles, we'll start by:

- converting to lowercase,
- removing each occurrence of `'to'`, `'the'`, and `'for'`,
- replacing each non-letter/digit/space character with a space, and
- replacing each sequence of multiple spaces with a single space.

```
jobtitles = (
    jobtitles
    .str.lower()
    .str.replace(r'\bto|\bthe|\bfor', '', regex=True)
    .str.replace('[^A-Za-z0-9 ]', ' ', regex=True)
    .str.replace(' +', ' ', regex=True)     # ' +' matches 1 or more occurrences of a space
    .str.strip()                            # Removes leading/trailing spaces if present
)
```

## Bags of Words 👜

Two job titles are similar if they contain similar words.

### Count Matrix:

- 1 row per text
- 1 column per **unique** word
- the value in entries are the occurrences of `word` in `text`

```
# following is example code for count matrix
all_words = jobtitles.str.split().sum()
unique_words = pd.Series(all_words).value_counts()

# Created using a dictionary to avoid a "DataFrame is highly fragmented" warning.
counts_dict = {}
for word in unique_words.index:
    re_pat = fr'\b{word}\b'
    counts_dict[word] = jobtitles.str.count(re_pat).astype(int).tolist()

counts_df = pd.DataFrame(counts_dict)
```

### Similarities: Dot Product

**Dot Product**: sum of individual term products.

**Geometric** interpretation:

$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}$$

***Big number means more similar***

**Note:** Sometimes, you will see the **cosine distance** being used. It is the complement of cosine similarity:

$$\textbf{dist}(\vec{a},\vec{b}) = 1 - \cos\theta$$

### A recipe for computing similarities

Given a set of texts, to find the **most similar** text to one text TT in particular:

- Use the bag of words model to create a counts matrix. Specifically:
  - Create an index out of **all** distinct words used across all texts.
  - Create a single vector for each text by counting the number of occurrences of each distinct word.
- Compute the cosine similarity between text TT and all other texts.
- The other text with the greatest cosine similarity is the most similar, under the bag of words model.

### Pitfalls of the bag of words model

Remember, the key assumption underlying the bag of words model is that **two texts are similar if they share many words in common**.

- The bag of words model doesn't consider **order**.
- The job titles `'asst fire chief'` and `'chief fire asst'` are treated as the same.
- The bag of words model treats all words as being equally important.
  - `'asst'` and `'fire'` have the same importance, even though `'fire'` is probably more important in describing someone's job title.
- The bag of words model doesn't consider the **meaning** of words.
  - `'I love data science'` and `'I hate data science'` share 75% of their words, but have very different meanings.

## TF-IDF

**Quantify** how **important** a word(term) is to a document. (working on a single document...)

The **term frequency-inverse document frequency (TF-IDF)** of word $t$ in document dd is the product:

$$\text{tfidf}(t, d) = \text{tf}(t, d) \cdot \text{idf}(t)$$

### Term Frequency

- The **term frequency** of a word (term) $t$ in a document dd, denoted $tf(t, d)$ is the proportion of words in document dd that are equal to $t$.

$$\text{tf}(t, d) = \frac{\text{number of occurrences of } t \text{ in } d}{\text{total number of words in } d}$$

### Inverse Document Frequency

- For a word t in a set of documents $d_1, d_2, \ldots$ is

$$\text{idf}(t) = \log\left(\frac{\text{total number of documents}}{\text{number of documents in which } t \text{ appears}}\right)$$

## *A More General Discussion on Features*

- A **feature** is a measurable property or characteristic of a phenomenon being observed.
  - Other words for "feature" include "(explanatory) variable" and "attribute".
- In DataFrames, features typically correspond to **columns**, while rows typically correspond to different individuals.
- There are two types of features:
  - Features that come as part of a dataset
  - Features that we **create**

## What makes a good feature?

- A good feature should be...
  - Faithful to the data generating process.
  - Strongly associated to the phenomenon of interest.
  - Easily used in standard modeling techniques (e.g. quantitative and scaled).
- Often times, the columns in a dataset aren't good features on their own. In such cases, we may need to "engineer" features that are useful.

## Feature Engineering

**Feature engineering** is the act of finding **transformations** that transform data into effective **quantitative variables**.

- A "good" choice of features depends on many factors:
  - The kind of data (quantitative, ordinal, nominal),
  - The relationship(s) and association(s) being modeled,
  - The model type (e.g. linear models, decision tree models, neural networks).

### *General Transformer Idea*

### *Transformer classes*

- Transformers take in "raw" data and output "processed" data. They are used for creating features.
  - The input should be a **multi-dimensional** `numpy` array.
    - Inputs can be DataFrames, but `sklearn` only looks at the values (i.e. it calls `to_numpy()` on input DataFrames).
  - The output is a `numpy` array (never a DataFrame or Series).

### *One Hot Encoding*

- One-hot encoding is a transformation that turns a categorical feature into several binary features.
- 1 indicate one row has the value the column represents, 0 indicate don't have the value
- One-Hot encoding is also called "dummy encoding"

## SKLearn Code

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
ohe.fit(X) # this already returns the fitted object and stored a sparse matrix
```

## *Binarizer*

The `Binarizer` transformer allows us to map a quantitative sequence to a sequence of 1s and 0s, depending on whether values are above or below a threshold

### SKLearn Code

```
from sklearn.preprocessing import Binarizer
bi = Binarizer(threshold = n)
transformed = bi.transform(X)
```

## *StdScaler*

**Standardizes** the data using mean and standard deviation (z-score).

- `StdScaler` requires the knowledge over mean and std over a larger dataset:
  - It needed to be `fit` before we `transform`

### SKLearn Code

```
from sklearn.preprocessing import StandardScaler
stdscaler = StandardScaler()
stdscaler.fit(X_train)
stdscaler.transform(X)
```

## *PolynomialFearues*

Reduce a feature from degree d to linear.

```
from sklearn.preprocessing import PolynomialFeatures
pf = PolynomialFeatures(d)
pd.transform(X)
```

## *CountVectorizer*

Entries in the `'summary'` column are not currently quantitative! We can use the bag-of-words encoding to create quantitative features out of each `'summary'`. Instead of performing a bag-of-words encoding manually as we did before, we can rely on `sklearn`'s `CountVectorizer`.

```
from sklearn.feature_extraction.text import CountVectorizer

example_corp = ['hey hey hey my name is billy',
                'hey billy how is your dog billy']
count_vec = CountVectorizer()
count_vec.fit(example_corp)
count_vec.vocabulary_ # give the word and corresponded index
# dataframe of the word count matrix
pd.DataFrame(count_vec.transform(example_corp).toarray(),
             columns=pd.Series(count_vec.vocabulary_).sort_values().index)
```

### *Dealing with Multicollinearity*

- Multicollinearity occurs when features in a regression model are **highly correlated** with one another.
  - In other words, multicollinearity occurs when **a feature can be predicted using a linear combination of other features, fairly accurately**.
- When multicollinearity is present in the features, the **coefficients in the model** are uninterpretable – they have no meaning.
  - A "slope" represents "the rate of change of yy with respect to a feature", when all other features are held constant – but if there's multicollinearity, you can't hold other features constant.
- Note: Multicollinearity doesn't impact a model's predictions!
  - It doesn't impact a model's ability to generalize to unseen data.
  - If features are multicollinear in the training data, they will probably be multicollinear in the test data too.
- Solutions:
  - Manually remove highly correlated features.
  - Use a dimensionality reduction technique (such as PCA) to automatically reduce dimensions.
- Multicollinearity is present when performing one-hot encoding; a solution is to drop **one one-hot-encoded column for each original categorical feature**.

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(drop='first') # direct drop first column of the ohe result
ohe.fit_transform(tips_features[['sex', 'smoker', 'day', 'time']]).toarray()
```

## Model

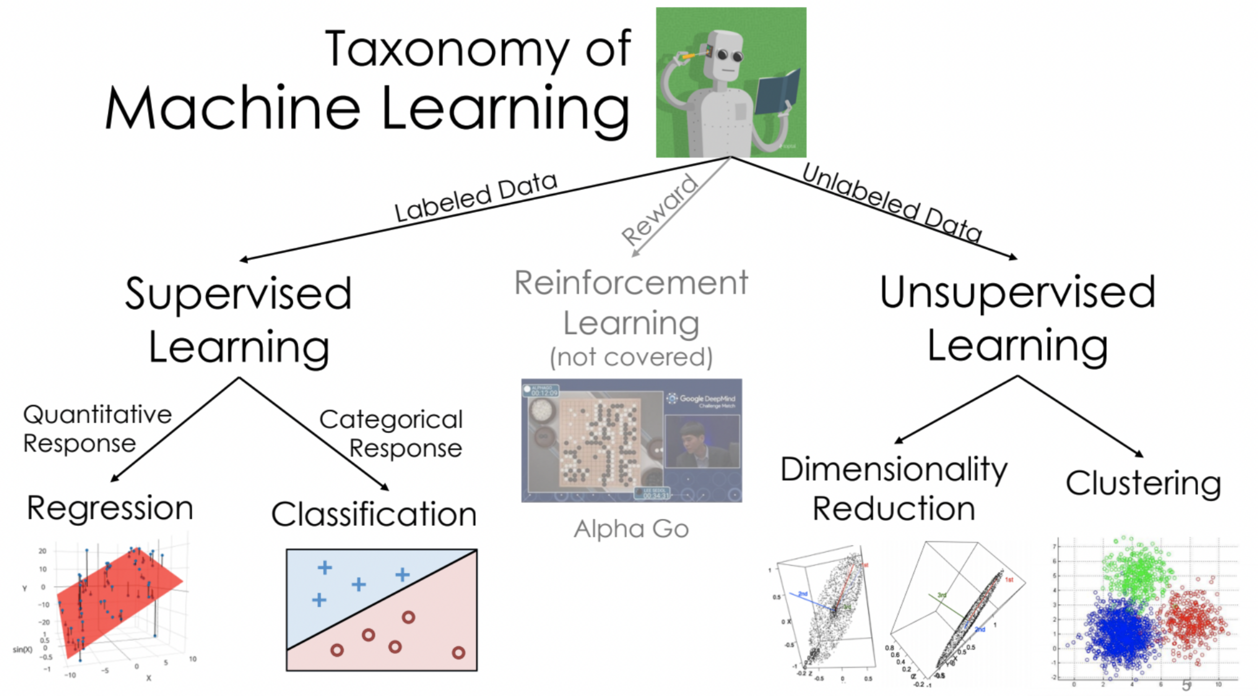
- **Data Generating Process**: The real-world phenomena that we are interested in studying.
  - *Example:* Every year, city employees are hired and fired, earn salaries and benefits, etc.
  - Unless we work for the city, we can't observe this process directly.
- **Model**: A theory about the data generating process.
  - *Example:* If an employee is XX years older than average, then they will make $100,000 in salary.
- **Fit Model**: A model that is learned from a particular set of observations, i.e. training data.

- *Example:* If an employee is 5 years older than average, they will make $100,000 in salary.
- How is this estimate determined? What makes it "good"?

## Goals of modeling

To make accurate **predictions** regarding unseen data drawn from the data generating process.

To make **inferences** about the structure of the data generating process (i.e. to understand complex phenomena).



## *General Usage of Model*

## Model classes

- `sklearn` model classes (called "estimators") behave like transformers, in that we need to instantiate and `fit` them.
- The difference is that we also need to specify what our "response" or "target" variable is, i.e. what we are trying to predict.
  - Calling `fit` is the same as "training our model".

```
# almost all of the models follow the same syntax:
# given mdl is one model

# step 1: init the model object, take LinearRegression as an example
mdl = LinearRegression()

# step 2: fit the training data
mdl.fit(X_train, y_train)

# then you can freely use the mdl object
```

## Some Common Model Methods and Attributes

- `mdl.predict(X_test)`: for a given data `X_test`, provides the prediction from the model
- `mdl.score(X, y)`: find the $R^2$ score on the predictions by `X` over `y` (for a linear model)

There are several models in the `linear_model` package; we will start with `LinearRegression`.

*LinearRegression*

## SKLearn Code

```
from sklearn.linear_model import LinearRegression

lr.score # this return the R2 score
lr.predict(X_test) # returns the predictions over X_test
lr.coef_ # Access the regression coeifficients
```

### Hyperparameters:

```
lr(data, fit_intercept=True, copy_X=True, n_jobs=None, positive=False)
```

- `copy_X`: If True, X will be copied; else, it may be overwritten.
- `n_jobs`: processors to use for the computation. Default None = 1, `n_jobs=-1` to use all processors.

## Definition:

- find the **line of best fit**, or the **regression line** that minimize Loss Function.

## Typical Loss Function:

- **Mean Squared Error**:
- **Root Mean Squared Error**: to make the unit meaningful

## Visualizing single-feature predictions example

```
sns.scatterplot(data=galton, x='father', y='childHeight', label='actual child heights')
sns.scatterplot(x=galton['father'],
                y=pred_child(galton['father']),
                label='predicted child heights'
);
```

- With `PolynomialFeatures` transformer, you can apply linear transformation toward different degrees

# *Decision Tree*

## SKLearn Code

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import DecisionTreeRegressor
```

### Hyperparameters

```
DecisionTreeClassifier(*, criterion='gini', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None,
ccp_alpha=0.0)
```

Documentation API for Classifier

Documentation API for Regressor

## Visualizing Decision Tree

```
from sklearn.tree import plot_tree
plt.figure(figsize=(10, 5))
plot_tree(dt, feature_names=X_train.columns, class_names=['no', 'yes'],
               filled=True, rounded=True, fontsize=15, impurity=False);
```

## Decision trees and overfitting

- Decision trees have a tendency to overfit. **Why is that?**
- Unlike linear classification techniques (like logistic regression or SVMs), **decision trees are non-linear**
  - They are also "non-parametric" – there are no ws to learn.
- While being trained, decision trees ask enough questions to effectively **memorize** the correct response values in the training set. However, the relationships they learn are often overfit to the noise in the training set, and don't generalize well.
- Fact: A decision tree whose depth is not restricted will achieve 100% accuracy on any training set, as long as there are no "overlapping values" in the training set.
  - Two values overlap when they have the same features xx but different response values y (e.g. if two patients have the same glucose levels and BMI, but one has diabetes and one doesn't).
- **One solution:** Make the decision tree "less complex" by limiting the maximum depth.

# *Random Forest*

- A "random forest" is a combination (or **ensemble**) of decision trees, each fit on a different **bootstrapped** resample of the training data.
- It makes predictions by aggregating the results of the individual trees (in the case of classification, by taking the **most common prediction**).

A **development** of decision tree which helps decrease probability of overfit

## SKLearn Code

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import RandomForestRegressor
```

## Hyperparameters

```
RandomForestClassifier(n_estimators=100, *, criterion='gini', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='sqrt',
max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None,
random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0,
max_samples=None)
```

Documentation API for Classifier

Documentation API for Regressor

# *Logistic Regression*

Logistic *regression* is a linear *classification?* technique that builds upon linear regression. It models **the probability of belonging to class 1, given a feature vector**:

$$P(y=1|\vec{x}) = \sigma(\underbrace{w_0 + w_1 x^{(1)} + w_2 x^{(2)} + \ldots + w_d x^{(d)}}_{\text{linear regression model}})$$

Here, $\sigma(t) = \frac{1}{1+e^t}$ is the **sigmoid** function; its outputs are between 0 and 1 (which means they can be interpreted as probabilities).

## SKLearn Code

```
from sklearn.linear_model import LogisticRegression
```

## Hyperparameters

```
LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True,
intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100,
multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)
```

Documentation API

# Wrap Up: Pipeline

### `Pipeline`s in `sklearn`

- A `Pipeline` object is instantiated using a list containing transformer(s) and a model (estimator).

```
pl = Pipeline([feat_trans1, feat_trans2, ..., mdl])
```

- Once a `Pipeline` is instantiated, you can fit all steps (transformers and model) using `fit`.

```
pl.fit(data, responses)
```

- To make predictions using **raw (untransformed) data**, use `pl.predict`.

### Creating a `Pipeline`

- To instantiate a `Pipeline`, we must provide a list with zero or more transformers followed by a single model.
  - All "steps" must have `fit` methods, and all but the last must have `transform` methods.
- The list we provide `Pipeline` with must be a list of tuples, where
  - The first element is a "name" (that we choose) for the step.
  - The second element is a transformer or estimator instance.

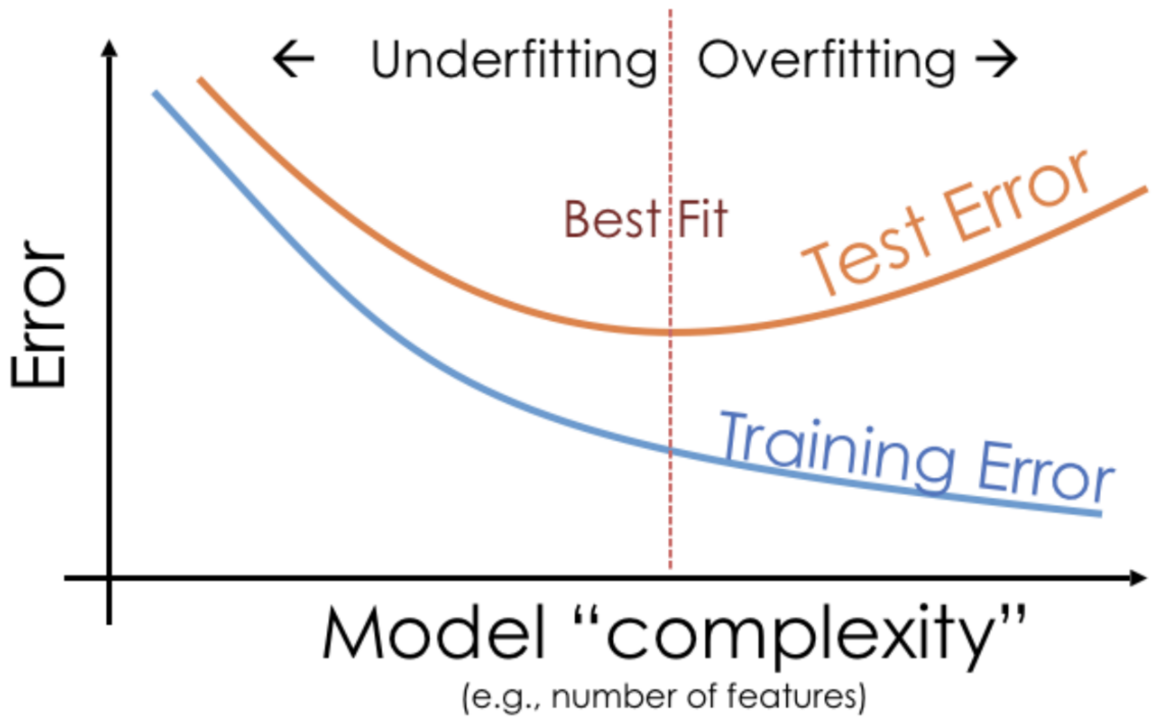
### Examples Code

```
from sklearn.pipeline import Pipeline

# a simple pipeline
pl = Pipeline([
    ('one-hot', OneHotEncoder()),
    ('lin-reg', LinearRegression())
])
pl.fit(tips_cat, tips['tip'])
pl.predict([['Male', 'Yes', 'Sat', 'Lunch']])

# a slightly complicate pipeline: ColumnTransformer
from sklearn.compose import ColumnTransformer

preproc = ColumnTransformer(
    transformers = [
        ('quant', StandardScaler(), ['total_bill', 'size']),
        ('cat', OneHotEncoder(), ['sex', 'smoker', 'day', 'time'])
    ]
)
pl = Pipeline([
    ('preprocessor', preproc),
    ('lin-reg', LinearRegression())
])
pl.fit(tips_features, tips['tip'])
```

- Notice that `ColumnTransformer` can be in `Pipeline`, but a `Pipeline` can also be in `ColumnTransformer`
- `ColumnTransformer` works separately parallelly, not sequential, don't put consecutive steps in it.

# Tuning Model Hyperparameters

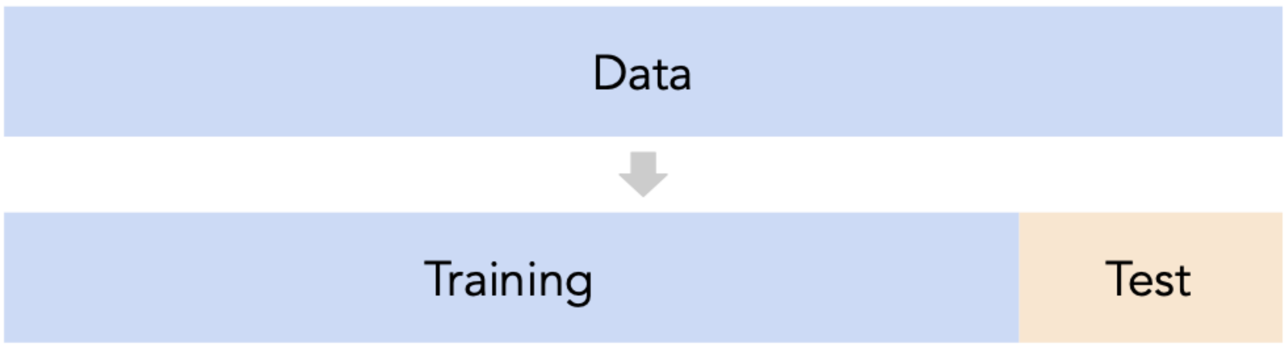
To tune model hyperparameters is to find balance between **Overfit** and **Underfit** such that our model capture the data generating process the most.



Train-Test Split

Normally, we have no access to the real test data set.

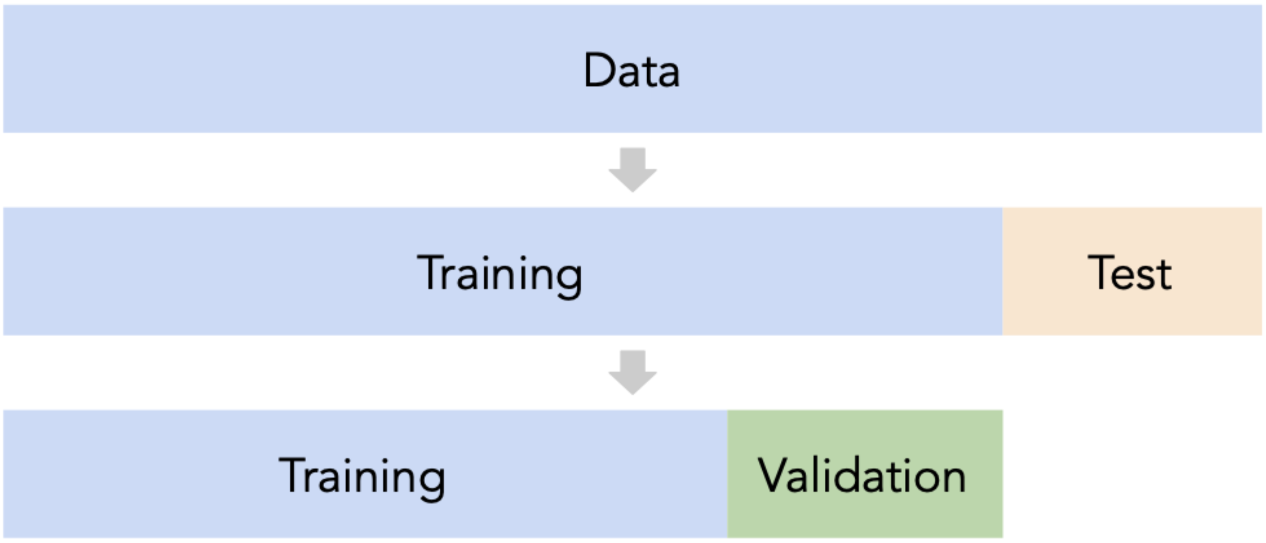
- However, to avoid overfitting (fail to generalize), we have to find a way to prevent our model gain too much information from training set [by choosing the right hyperparameters]
- Logic: split the training data into training set and the test set: and see how the model from the training set words on the test set.



However, to prevent to model only relies on that test set: we introduced *Cross Validation*:

## *Cross Validation*

## A single validation set

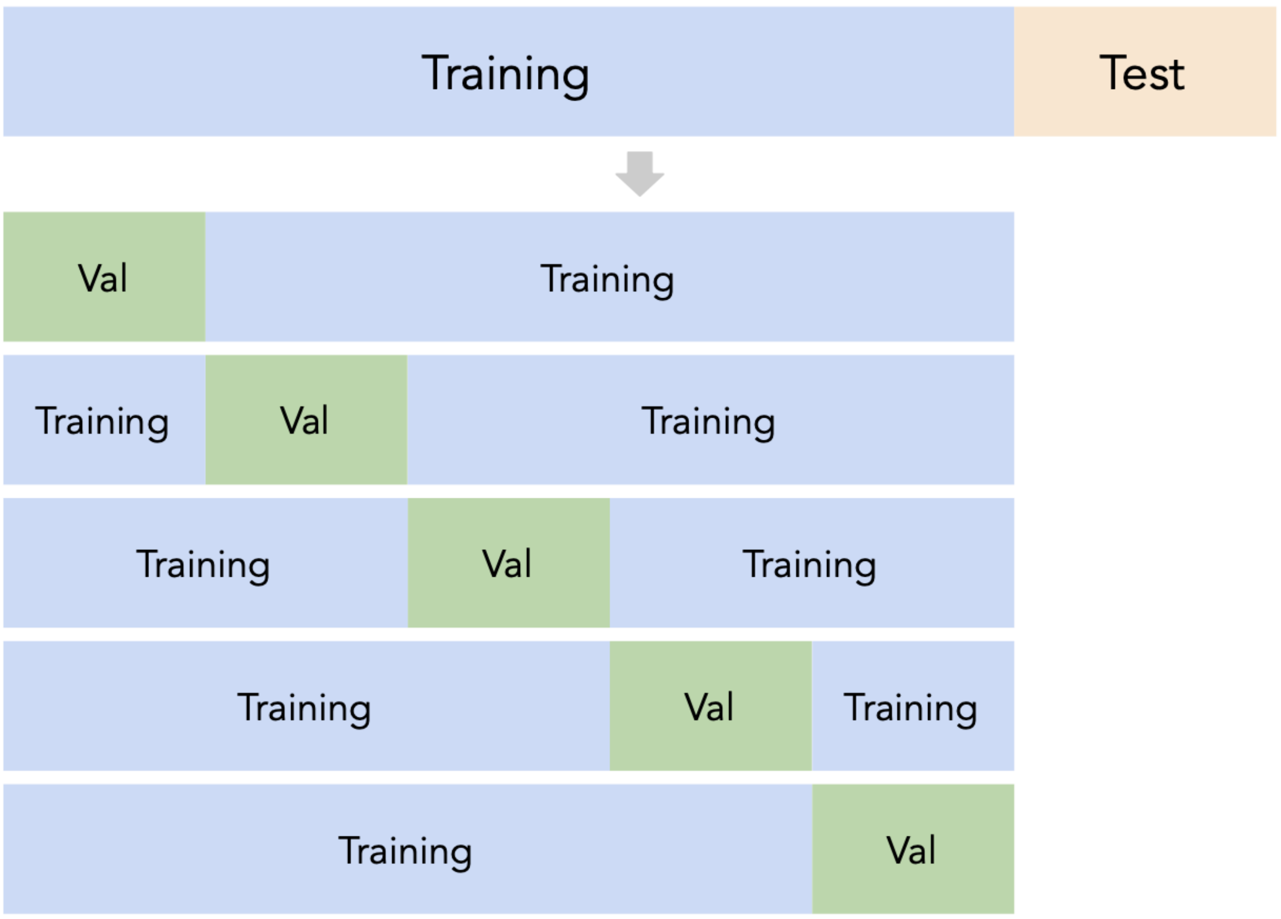


1. Split the data into three sets: **training**, **validation**, and **test**.
2. For each hyperparameter choice, **train** the model only on the **training set**, and **evaluate** the model's performance on the **validation set**.
3. Find the hyperparameter with the best **validation** performance.
4. Retrain the final model on the **training** and **validation** sets, and report its performance on the **test set**.

**Issue:** This strategy is too dependent on the **validation** set, which may be small and/or not a representative sample of the data.

## k-fold cross-validation

Instead of relying on a single validation set, we can create kk validation sets, where kk is some positive integer (5 in the following example).



Since each data point is used for training k−1 times and validation once, the (averaged) validation performance should be a good metric of a model's ability to generalize to unseen data.

**Steps**: first, **shuffle** the dataset randomly (to avoid some existed pattern) and **split** it into k disjoint groups. Then:

- For each hyperparameter:
  - For each unique group:
    - Let the unique group be the "validation set".
    - Let all other groups be the "training set".
    - Train a model using the selected hyperparameter on the training set.
    - Evaluate the model on the validation set.
  - Compute the **average** validation score (e.g. RMSE) for the particular hyperparameter.
- Choose the hyperparameter with the best average validation score.

## Example Code

### Manual

```
from sklearn.model_selection import FKold
kfold = KFold(5, shuffle=True, random_state=1)
# Manual KFold Example Code
errs_df = pd.DataFrame()

for d in range(1, 16):
    errs = []
    for train, val in kfold.split(sample_1):
        # Separate the data into a training set and validation set
        data_train, data_val = sample_1.iloc[train], sample_1.iloc[val]

        # Fit the model on the training set
        pl = Pipeline([('poly', PolynomialFeatures(d)), ('lin-reg', LinearRegression())])
        pl.fit(data_train[['x']], data_train['y'])

        # Compute the model's validation error
        val_err = mean_squared_error(data_val['y'], pl.predict(data_val[['x']]), squared=False)
        errs.append(val_err)

    errs_df[f'Deg {d}'] = errs

errs_df.index = [f'Fold {i}' for i in range(1, 6)]
```

### Semi-automatic

```
from sklearn.model_selection import FKold
from sklearn.model_selection import cross_val_score

errs_df_auto = pd.DataFrame()

for d in range(1, 16):
    pl = Pipeline([('poly', PolynomialFeatures(d)), ('lin-reg', LinearRegression())])

    # The `scoring` argument is used to specify that we want to compute the RMSE; the default is 
R^2
    # It is called "neg" RMSE because by default sklearn likes to "maximize" scores
    errs = cross_val_score(pl, sample_1[['x']], sample_1['y'], cv=5, 
scoring='neg_root_mean_squared_error')
    errs_df_auto[f'Deg {d}'] = -errs # Negate to turn positive (sklearn computed negative RMSE)

errs_df_auto.index = [f'Fold {i}' for i in range(1, 6)]
```

## *Grid Search*

**An efficient technique for trying different combinations of hyperparameters**: automatic cross validation!

`GridSearchCV` takes in:

- An **un-fit** instance of an estimator, and
- a **dictionary** of hyperparameter values to try

and performs k-fold cross-validation to find the **combination of hyperparameters with the best average validation performance**.

## Example Code

```
from sklearn.model_selection import GridSearchCV
hyperparameters = {
    'max_depth': [2, 3, 4, 5, 7, 10, 13, 15, 18, None],
    'min_samples_split': [2, 3, 5, 7, 10, 15, 20],
    'criterion': ['gini', 'entropy']
}
searcher = GridSearchCV(DecisionTreeClassifier(), hyperparameters, cv=5)
searcher.fit(X_train, y_train) # the process of grid search
searcher.best_params_  # the best parameters chosed
searcher.cv_results_['mean_test_score'] # show all test score

# Rows correspond to folds, columns correspond to hyperparameter combinations
```

```
pd.DataFrame(np.vstack([searcher.cv_results_[f'split{i}_test_score'] for i in range(5)]))

searcher.predict(X_train) # the model directly with best parameters
searcher.score(X_test, y_test) # show score
```

- `GridSearchCV` is not the only solution – see `RandomizedSearchCV` if you're curious.

## Model (Result) Evaluation

- Loss functions can all become a way of evaluating the model.
  - MSE

```
from sklearn.metrics import mean_squared_error # built-in RMSE/MSE function
```

### $R^2$: the Coefficient of Determination

- A measure of the **quality of a linear fit**.
- **Key Idea**: range from 0 to 1. **The Closer it is to 1, the better the linear fit is**
- is the **proportion of variance in y that the linear model explains**

$$R^2 = \frac{\text{var(predicted y values)}}{\text{var(actual y values)}} = [\text{correlation(predicted y values, actual values)}]^2$$

### Accuracy

$$\text{accuracy} = \frac{\text{\# data points classified correctly}}{\text{\# data points}}$$

- The `score` method of a **classifier** computes accuracy by default.

## *On Overfitting and Underfitting*

**Accuracy isn't everything!**

### Confusion Matrix

Below, we present a **confusion matrix**, which summarizes the four possible outcomes of the wolf classifier.

| | |
|---|---|
| **True Positive (TP):** • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero. | **False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up. |
| **False Negative (FN):** • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep. | **True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine. |

### Outcomes in binary classification

When performing **binary** classification, there are four possible outcomes.

(Note: A "positive prediction" is a prediction of 1, and a "negative prediction" is a prediction of 0.)

| Outcome of Prediction | Definition | True Class |
|---|---|---|
| **True** positive (TP) ✓ | The predictor **correctly** predicts the positive class. | P |
| False negative (FN) ✗ | The predictor incorrectly predicts the negative class. | P |
| **True** negative (TN) ✓ | The predictor **correctly** predicts the negative class. | N |
| False positive (FP) ✗ | The predictor incorrectly predicts the positive class. | N |

| | **Predicted Negative** | **Predicted Positive** |
|---|---|---|
| **Actually Negative** | TN ✓ | FP ✗ |
| **Actually Positive** | FN ✗ | TP ✓ |

### Code for Plotting a Confusion Matrix

```
from sklearn import metrics
y_pred = mdl.predict(X_test)
metrics.plot_confusion_matrix(clf, X_test, y_test);
```

## Recall

proportion of actually positive instances that are correctly identified

we wish this to be as close to 1 as possible

- Better to be high in medical exams.

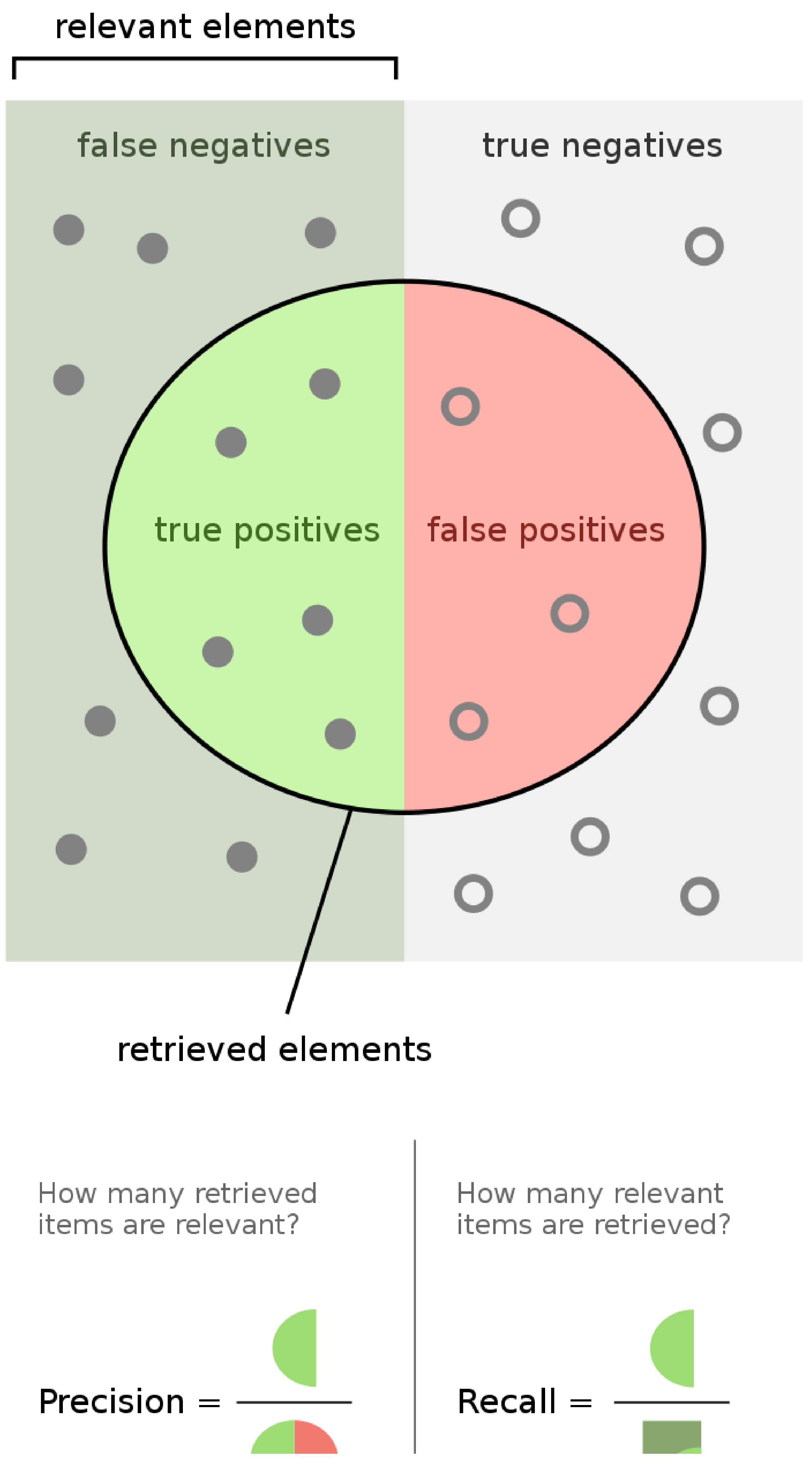
$$\text{recall} = \frac{TP}{TP + FN}$$

## Precision

proportion of actually negative instances that are correctly identified

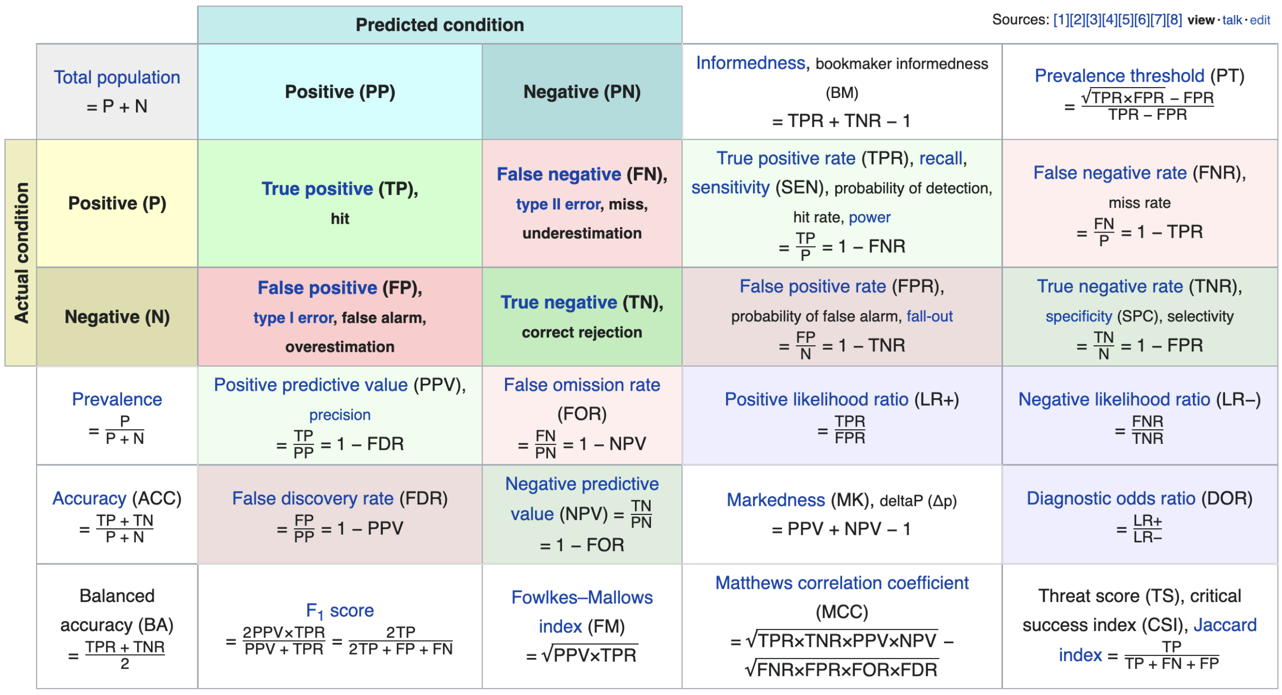
we wish this to be as close to 1 as possible

$$\text{precision} = \frac{TP}{TP + FP}$$


relevant elements
false negatives
true negatives
true positives
false positives
retrieved elements
How many retrieved items are relevant?
How many relevant items are retrieved?
Precision =
Recall =

### *Other evaluation metrics for binary classifiers*

We just scratched the surface! This excellent table from Wikipedia summarizes the many other metrics that exist.



| | | Predicted condition | | Sources: [1][2][3][4][5][6][7][8] **view** · talk · edit | |
|---|---|---|---|---|---|
| | Total population $= P + N$ | **Positive (PP)** | **Negative (PN)** | Informedness, bookmaker informedness (BM) $= TPR + TNR - 1$ | Prevalence threshold (PT) $= \frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$ |
| **Actual condition** | **Positive (P)** | **True positive (TP),** **hit** | **False negative (FN),** **type II error, miss, underestimation** | True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$ | False negative rate (FNR), miss rate $= \frac{FN}{P} = 1 - TPR$ |
| | **Negative (N)** | **False positive (FP),** **type I error, false alarm, overestimation** | **True negative (TN),** **correct rejection** | False positive rate (FPR), probability of false alarm, fall-out $= \frac{FP}{N} = 1 - TNR$ | True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$ |
| | Prevalence $= \frac{P}{P + N}$ | Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$ | False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$ | Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$ | Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$ |
| | Accuracy (ACC) $= \frac{TP + TN}{P + N}$ | False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$ | Negative predictive value (NPV) $= \frac{TN}{PN} = 1 - FOR$ | Markedness (MK), deltaP (Δp) $= PPV + NPV - 1$ | Diagnostic odds ratio (DOR) $= \frac{LR+}{LR-}$ |
| | Balanced accuracy (BA) $= \frac{TPR + TNR}{2}$ | $F_1$ score $= \frac{2PPV \times TPR}{PPV + TPR} = \frac{2TP}{2TP + FP + FN}$ | Fowlkes–Mallows index (FM) $= \sqrt{PPV \times TPR}$ | Matthews correlation coefficient (MCC) $= \sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$ | Threat score (TS), critical success index (CSI), Jaccard index $= \frac{TP}{TP + FN + FP}$ |

## Appendix 1: NumPy

"Numerical Python": commonly-used Python module that enables **fast** computation involving arrays and metrices

### *Array*

Main Object in `numpy`, they are

- homogenous (all values are of the same type), and
- (potentially) multi-dimensional

A good review of NumPy arrays

### *Relations with Pandas*

pandas is built upon numpy

- A Series in `pandas` is a `numpy` array with an index
- A DataFrame is like a dictionary of columns, each of which is a `numpy` array.
- Use `to_numpy` method to obtain the underlying `numpy` array object

### *Type Coercion*

numpy prefers homogenous data type, this lead some values to be changed format

## Appendix 2: Information from Website

- Never trust data from an unfamiliar site.
- **Never** use `eval` on "raw" data that you didn't create!

### *HTTP*

HTTP stands for **Hypertext Transfer Protocol**

- It is a **request-response** protocol
  - Protocol = set of rules

```
import requests
url = "https://ucsd.edu"
resp = requests.get(url) # resp will direct show response type
text = resp.text # a string that containing the entire response
url = resp.url # the same url we request from...
success = resp.ok # True if the request is successful
```

- **HTTP status code** return the status of requesting
  - Check the website for full code list
  - Some example: `200`: success; `404`: page not found; `500`: internal server error

### Possible Respond from `Request`

There are two ways of collecting data via requests:

- By using a published API (application programming interface).
- By scraping a webpage to collect its HTML source code.

### APIs:

- an API is a service that makes data directly available to the user in a convenient fashion.
- Advantages:
  - The data are usually clean, up-to-date, and ready to use.
  - The presence of a API signals that the data provider is okay with you using their data.
  - The data provider can plan and regulate data usage.
    - Some APIs require you to create an API "key", which is like an account for using the API.
    - APIs can also give you access to data that isn't publicly available on a webpage.
- Disadvantages:
  - APIs don't always exist for the data you want!
- An **API endpoint** is a URL of the data source that the user wants to make requests to.
- Sometimes an API can be in **JSON** methods

## *Website Format*

## JSON

**JavaScript Object Notation**: See json-schema.org for more details.

### JSON data types

- string: anything inside double quotes.
- number: any number (no difference between ints and floats).
- boolean: `true` and `false`.
- array: anything wrapped in `[]`.
- null: JSON's empty value, denoted by `null`.
- object: a collection of key-value pairs (like dictionaries).
  - Keys must be strings, values can be anything (even other objects).

```python
# work json in python
import json
f = open(fp)
json_file = json.load(f) # json objects resemble Python Dictionaries
# JSON format are objects, lists, from larger group to smaller
```

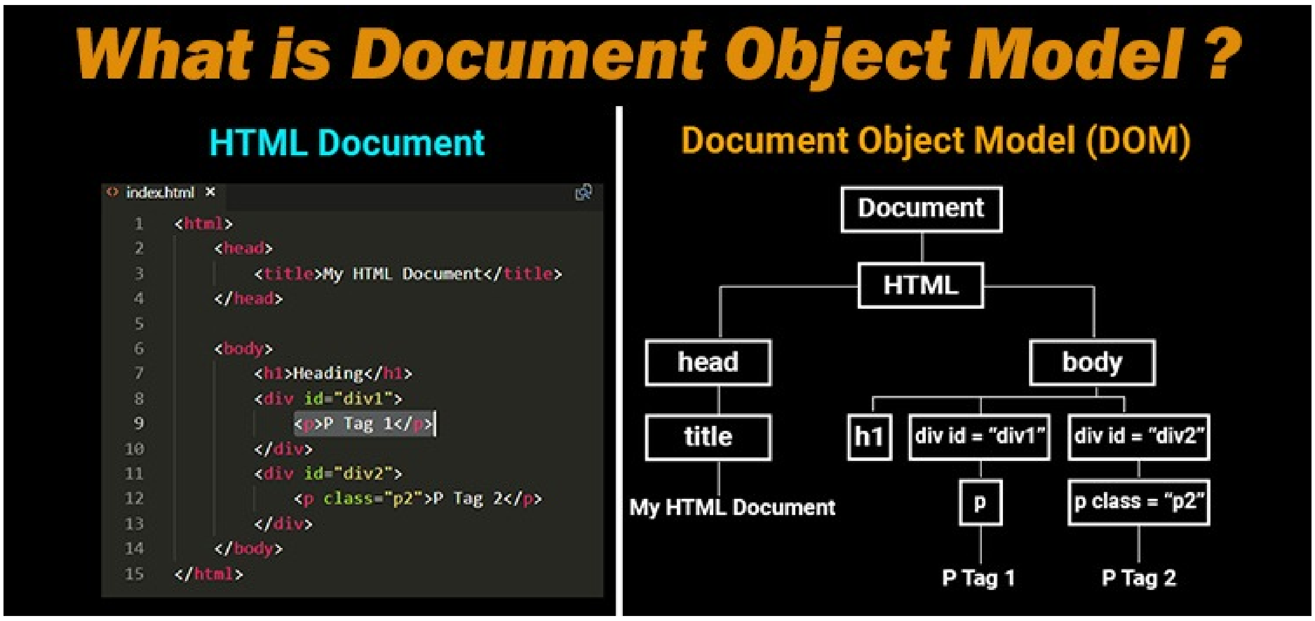
- The JSON data format needs to be **parsed**, not evaluated as a dictionary.
  - It was designed with safety in mind!

## HTML

- HTML (HyperText Markup Language) is **the** basic building block of the internet.
- It defines the content and layout of a webpage, and as such, it is what you get back when you scrape a webpage.
- See this tutorial for more details.

## The anatomy of HTML documents

- **HTML document**: The totality of markup that makes up a webpage.
- **Document Object Model (DOM)**: The internal representation of a HTML document as a hierarchical **tree** structure.
- **HTML element**: An object in the DOM, such as a paragraph, header, or title.
- **HTML tags**: Markers that denote the **start** and **end** of an element, such as `<p>` and `</p>`.



```
<html>
    <head>
        <title>My HTML Document</title>
    </head>

    <body>
        <h1>Heading</h1>
        <div id="div1">
            <p>P Tag 1</p>
        </div>
        <div id="div2">
            <p class="p2">P Tag 2</p>
        </div>
    </body>
</html>
```

### Useful tags to know

| Element | Description |
|---|---|
| `<html>` | the document |
| `<head>` | the header |
| `<body>` | the body |
| `<div>` | a logical division of the document |
| `<span>` | an *in-line* logical division |
| `<p>` | a paragraph |
| `<a>` | an anchor (hyper-link) |
| `<h1>, <h2>, ...` | header(s) |
| `<img>` | an image |

There are many, many more. See this article for examples.

# *Website Scraping*

- Scraping is the act of programmatically "browsing" the web, downloading the source code (HTML) of pages that you're interested in extracting data from.
- Advantages:
  - You can always do it!
    - e.g. Google scrapes webpages in order to make them searchable.
- Disadvantages:
  - It is often difficult to parse and clean scraped data.
    - Source code often includes a lot of content unrelated to the data you're trying to find (e.g. formatting, advertisements, other text).
  - Websites can change often, so scraping code can get outdated quickly.
  - Websites may not want you to scrape their data!
- In general, we prefer APIs.

## Python HTML Parser

- Beautiful Soup 4 is a Python HTML parser.
  - To "parse" means to "extract meaning from a sequence of symbols".

**Note**: `HTML(html_string)`: the code for `IPython` (jupyter notebook) to directly render the HTML string

### Child nodes

- Recall, HTML documents are represented as trees.
  - Each page element becomes a node in this tree.
- A `BeautifulSoup` object represents a **node** in the tree.
  - Each `BeautifulSoup` object has 0 or more child nodes.
  - To access the children of a node, use the `children` attribute.

```
import bs4
soup = bs4.BeautifulSoup(html_string)
soup.text # only return the text without tags...
soup.children # return the children of the root node, it is a iterator
```

### Depth-First Traversal through `descendants`

```
for child in soup.descendants:
    if isinstance(child, str):
        continue
    print(child.name)
```

### Finding Elements

- `find()`: finds the **first** instance of a tag
- `findall()`: return a list of all matches

```
soup.find(name=None, attrs={}, recurseive=True, text=None)
soup.findall(name=None, attrs={}, recurseive=True, text=None)
```

#### Node attributes

- The `text` attribute of a tag element gets the text between the opening and closing tags.
- The `attrs` attribute lists all attributes of a tag.
- The `get(key)` method gets the value of a tag attribute.

### Key takeaways

- Make as few requests as possible.
- Create a request and parsing plan **beforehand**.
- Create your output schema **beforehand**.
- Make requests and parse in **separate functions**!

# Appendix 3: Regular Expression

A regular expression, or **regex** for short, is a sequence of characters used to **match patterns in strings**.

- For example, `[1-9][0-9]{2}-[0-9]{3}-[0-9]{4}` matches US phone numbers of the form `'XXX-XXX-XXXX'`.
- They are very powerful and widely used.
- However, they are quite difficult to read.

## regex101.com

- However, when crafting regular expressions, it is helpful to work in an environment that provides syntax highlighting and details.
- regex101.com does exactly that – use it!
    - This link will bring you to the phone number example.

## *Regex building blocks* 📦

The four main building blocks for all regexes are shown below (table source, inspiration).

| operation | order of op. | example | matches ✓ | does not match ✗ |
|---|---|---|---|---|
| **concatenation** | 3 | `AABAAB` | `'AABAAB'` | every other string |
| **or** | 4 | `AA\|BAAB` | `'AA'`, `'BAAB'` | every other string |
| **closure** (zero or more) | 2 | `AB*A` | `'AA'`, `'ABBBBBBA'` | `'AB'`, `'ABABA'` |
| **parentheses** | 1 | `A(A\|B)AAB``(AB)*A` | `'AAAAB'`, `'ABAAB'``'A'`, `'ABABABABA'` | every other string`'AA'`, `'ABBA'` |

| operation | example | matches ✓ | does not match ✗ |
|---|---|---|---|
| **wildcard** | `.U.U.U.` | `'CUMULUS'` `'JUGULUM'` | `'SUCCUBUS'` `'TUMULTUOUS'` |
| **character class** | `[A-Za-z][a-z]*` | `'word'` `'Capitalized'` | `'camelCase'` `'4illegal'` |
| **at least one** | `bi(ll)+y` | `'billy'` `'billlllly'` | `'biy'` `'bily'` |
| **between a and b occurrences** | `m[aeiou]{1,2}m` | `'mem'` `'maam'` `'miem'` | `'mm'` `'mooom'` `'meme'` |

| operation | example | matches ✓ | does not match ✗ |
|---|---|---|---|
| **escape character** | `ucsd\.edu` | `'ucsd.edu'` | `'ucsd!edu'` |
| **beginning of line** | `^ark` | `'ark two'` `'ark o ark'` | `'dark'` |
| **end of line** | `ark$` | `'dark'` `'ark o ark'` | `'ark two'` |
| **zero or one** | `cat?` | `'ca'` `'cat'` | `'cart'` (matches `'ca'` only) |
| **built-in character classes*** | `\w+` `\d+` | `'billy'` `'231231'` | `'this person'` `'858 people'` |
| **character class negation** | `[^a-z]+` | `'KINGTRITON551'` `'1721$$'` | `'porch'` `'billy.edu'` |

## *Regex in Python*

The `re` package is built into Python. It allows us to use regular expressions to find, extract, and replace strings.

```python
import re

# some important re methods
re.search(regex, text) # returns the location and substring of the first match
re.findall(regex, text) # return all matches in a list
re.sub(regex, repl) # replace all matches with repl text

# some useful string formatting in python
f'balabala{var}'  # sub a var into the string: f string
r'bala\bala'  # raw-string to avoid special characters
```

## Capturing groups

- Surround a regex with `(` and `)` to define a **capturing group** within a regex pattern.
- Capturing groups are useful for extracting relevant parts of a string.

## Limitations of regexes

Writing a regular expression is like writing a program.

- You need to know the syntax well.
- They can be easier to write than to read.
- They can be difficult to debug.

Regular expressions are terrible at certain types of problems. Examples:

- Anything involving counting (same number of instances of a and b).
- Anything involving complex structure (palindromes).
- Parsing highly complex text structure (HTML, for instance).

### *Other Resources*

Python `re` library documentation and how-to.

regex “cheat sheet” (taken from here).

# Appendix 4: Time Series

## Python `datetime`

```
import datetime # datetime object of python
# Some important methods
datetime.datetime.now() # generate a datetime object of now
datetime.datetime.now() + datetime.timedelta(days=3, hours=5) # direct add datetime object
datetime.datetime.now().timestamp() # Unix timestamp: # of seconds from Jan 1st, 1970
```

## Time in pandas

### `pd.Timestamp`

- `pd.Timestamp` is the `pandas` equivalent of `datetime`
- `pd.to_datetime` automatically try to converts strings to `pd.Timestamp` objects
- also support direct add / subtract operation: yields `pd.Timedelta` objects
- a series of datetimes are stored as `np.datetime64` for memory and speed efficiency
- See the documentation for more details.

### `pd.Timedelta`

- For Time duraction

---

**Below sections might not be in the final exam**

# Appendix 5: Other Models

### *SVM*

Check out the tutorial

## SKLearn Code

```
from sklearn.svm import SVC
from sklearn.svm import SVR
```

### Hyperparameters

```
SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True,
probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=- 1,
decision_function_shape='ovr', break_ties=False, random_state=None)
```

Documentation API for Classifier

Documentation API for Regressor

## *Gradient Boosting Tree*

Another direction of dealing with decision tree overfitting issue(?), check out the tutorial

Also check out the old, popular boosting algorithm AdaBoost.

### SKLearn Code

```
from sklearn.ensemble import GradientBoostingClassifier
```

### Hyperparameters

```
GradientBoostingClassifier(*, loss='log_loss', learning_rate=0.1, n_estimators=100,
subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None,
random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False,
validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)
```

Documentation API for Classifier

Documentation API for Regressor

## *XGBoost*

XGBoost is a complicated but highly efficient gradient boosting tree algorithm system.

You can learn it from their official website

Paper of XGBoost

# Appendix 6: Ethics & Fairness

- Sometimes, a model performs better for certain groups than others; in such cases we say the model is **unfair**.
- Since ML models are now used in processes that significantly affect human lives, it is important that they are fair!
  - Job applications and college admissions.

- Criminal sentencing and parole grants.
- Predictive policing.
- Credit and loans.

## *How does bias occur?*

Remember, our models learn patterns from the training data. Various sources of bias may be present within training data:

- Training data may not be representative of the population.
    - There may be fewer data points for minority groups, leading to poorer model performance.
- The features chosen may be more useful in making predictions for certain groups than others.
- Training data may encode existing human biases.

As a data scientists, we might someday changed someone's life by our models, so be careful!

***The Last Part: the mathematical conclusion on the unfairness is not up yet, check out the Course Website***

# Appendix 7: Other Resources

## *Lecture Recordings*

UCSD Podcast

## *Course Readings*

- notes.dsc80.com, our course notes.
- Wes McKinney. “Python for Data Analysis”.
- DSC 10 Course Notes – great refresher on `babypandas`.
- Principles and Techniques of Data Science, the textbook for Berkeley’s Data 100 course.
- Computational and Inferential Thinking, the textbook for Berkeley’s Data 8 course.

## *Past Exams*

| Quarter | Instructor(s) | Midterm | Final |
|---|---|---|---|
| Spring 2022 | Suraj Rampure | exam, solutions | |
| Fall 2021 | Justin Eldridge | exam, solutions | exam, solutions, video |
| Spring 2021 | Justin Eldridge | exam, solutions, video | exam, solutions |
| Spring 2019 | Aaron Fraenkel, Marina Langlois | exam, solutions | |

# *Tech: Environment List and Git*

## Environment List:

```
matplotlib==3.4.3
numpy==1.21.2
otter-grader==3.1.4
pandas==1.3.3
Pillow==8.3.2
pydantic==1.8.2
PyYAML==5.4.1
requests==2.26.0
tqdm==4.62.3
urllib3==1.26.7
scikit-learn==1.0
seaborn==0.11.2
beautifulsoup4==4.10.0
```

## GitHub Repository

https://github.com/dsc-courses/dsc80-2022-sp

# *Pandas Tutor*

pandastutor.com is a new tool that allows you to visualize DataFrame operations.

- It works similarly to pythontutor.com, which you may have seen in DSC 20.

# *Intel Acceleration Over SKLearn*

*06022022* Package Documentation

If you are using CPU which at least support one of the SSE2, AVX, AVX2, AVX512 instruction sets, you can use the Intel Extension for Scikit-learn to accelerate your SKLearn.

## Installation

`pip install scikit-learn-intelex` for `PyPI`

or `conda install scikit-learn-intelex -c conda-forge` for a Anaconda environment

## Use

```python
from sklearnex import patch_sklearn
patch_sklearn()
```

*Visual Explanation of Permutation Testing*

jwilber.me/permutationtest