Lab 0: Syntax and Introduction

Getting and loading online data, syntax

```
DOWNLOAD_URL = "http://www.everyday-beat.org/ginsberg/poems/howl.txt"
SAVE_URL = "howl.txt"

from urllib.request import urlopen
downloadedText = urlopen(DOWNLOAD_URL).read().decode('utf-8')
with open(SAVE_URL, "w") as file:
    file.write(downloadedText)

# LOAD DATA
with open("howl.txt", "r") as poemFile:
    poemText = poemFile.read()
print(poemText[0:500])

# SPLIT TEXT
poemLines = poemText.splitlines() # into list of lines
poemWords = poemText.split() # into list of words

# COUNTING THE NUMBER OF OCCURRENCES OF A WORD
wordsCounter = 0
for word in poemWords:
    # strip non-alphanumeric character, and convert to lower case
    if word.strip(""" ,.*()[]!@#$%^&*{}?'`"-""").lower() == "word":
        wordsCounter += 1

# COUNTING THE NUMBER OF LINES STARTING WITH A WORD
linesCounter = 0
for line in poemLines:
    if line.lower().strip().startswith("who"):
        linesCounter += 1

# COUNTING UNIQUE WORDS
cleanWords = [word.strip(""" ,.*()[]!@#$%^&*{}?'`"-""").lower()
              for word in poemWords]

# COUNTING WITH COUNTER
from collections import Counter
wordsCounter = Counter(cleanWords)
wordsCounter["word"] # number of occurrences of "word"

# FIND IN TEXT
poemText.find("word")

# REMOVE ITEM FROM LIST
del cleanWords[2]           # by index
cleanWords.remove("word")   # by value
```

```
# LIST COMPREHENSIONS
[num**2 for num in range(1,6) if num % 2 != 0]          # [1, 4, 9, 16, 25]
[num**2 if (num % 2 != 0) else 0 for num in range(1,6)] # [1, 0, 9, 0, 25]

# DICTIONARY COMPREHENSIONS
{num: num**2 for num in range(1,6)}                     # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
# Collect keys if they have values meeting certain criteria, e.g., people's names whose
age is under 30)?
peopleAge = {'Sam':25,'Luke':35,'Judy':50,'Paul':18}
peopleUnder30 = {key:value for (key,value) in peopleAge.items() if value <= 30}
for (name,age) in peopleUnder30.items():
    print('Name: {:<6}, Age:{:<5} '.format(name,age))
```

Using **list comprehensions**, write a code that takes a list of names and returns a list of "Hello name" for each name that has more than 6 characters.

```
names = ["name1", "longname2", "verylongname3", "name4", "longlonglong5"]
namesNew = [f"Hello {k}" for k in names if len(k) > 6]
```

There are four **collection data types** in the Python programming language:

1. List is a collection which is ordered and mutable(changeable). Allows duplicate members.
2. Tuple is a collection which is ordered and immutable(unchangeable). Allows duplicate members.
3. Set is a collection which is unordered, unidexed and mutable and . No duplicate members.
4. Dictionary is a collection of key-value pairs which is unordered, mutable and indexed (values are indexed by keys). No duplicate keys, but allows duplicate values.

Map applies a function to all items in a list: map(func, input_list)

```
# MAPPING - map is lazily evaluated
def square(num):
    return num**2
items = [1, 2, 3, 4, 5]
squaredItems_bymap = list(map(square, items))
print(squaredItems_bymap)
```

Lambda variables: simple return-based functions.

```
def longMultiply(num):
    return num ** 2
shortMultiply = lambda num: num ** 2
```

With the **filter** function

```
names = ["name1", "longname2", "verylongname3", "name4", "longlonglong5"]

# Filter names with more than 6 characters
filtered_names = filter(lambda k: len(k) > 6, names)

# Add "Hello " to each name
namesNew = list(map(lambda k: f"Hello {k}", filtered_names))
```

Numpy

```
import numpy as np

# LIST TO NUMPY ARRAY
my2dlist = [[1,2,3],[4,5,6]]
my2darray = np.array(my2dlist)

# RANGE SUBINDEXIN
# i:j:k where i is the starting index, j is the stopping index, and k is the step (k !=
0).
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x[1:7:2])

# MATRIX OPERATIONS
x = np.array([[11,22],[99,88]])
xI = np.linalg.inv(x) # inverse
x = x.T               # transpose
identityMatrix = x@xI # matrix multiply
identityMatrix = matmul(x, xI)
```

Pandas

```
import pandas as pd

# READ CSV
url = 'https://raw.githubusercontent.com/MIE223-2024/course-
datasets/main/arabica_data.csv'
df = pd.read_csv(url, index_col=0, sep="\s+") # sep="\s+" : use one or more spaces as
delimiter
df.head()     # first 5 rows
df.describe() # summary statistics
df['Species'] # access specific column
set(df['Variety'].values) # raw values of a column
df['Variety'].unique()    # unique values of a column

# CREATE COPY
df_copy = df.copy()
```

Imputation

```
# IMPUTATION IN PYTHON
df['vals_imputed'] = df['vals'].fillna(avg_altitude_mean_meters) # replace vals with the
name of your column

# GET AVG RATING BY COUNTRY
df_clean.groupby('Country of Origin')['quality_score'].mean()
```

Lab 1: More on Data Manipulation Tools for Data Science with Python

More on Numpy

```
import numpy as np

# NUMPY ARRAY ATTRIBUTES
my_2D_arr = np.array([[7, 1, 4], [5, 5, 9], [0, 8, 3], [2, 6, 2]])
my_2D_arr.shape # Shape of my_2D_arr is: (4, 3)
my_2D_arr.size  # Size: 12
len(my_2D_arr)  # Length: 4

# CONSTRUCT ARRAY
np.arange(1, 10, 2)     # from range of numbers
np.linspace(1, 10, 10)  # equally-spaced elements
np.logspace(1, 10, 10)  # logarithmically-spaced elements
np.zeros(10)            # zero array
np.eye(5)               # identity matrix (same as np.identity(5))
np.diag([1, 2, 3, 4])   # diagonal matrix
my_rand_arr = np.random.randint(low=0, high=9, size=10) # sample from a uniform
distribution

# ARRAY OPERATIONS
np.random.seed(42)      # set seed for reproducibility
np.random.choice([1, 3, 5, 8, 2, 4], size=3, replace=False, p=None) # sample from list of
choices
np.sort(my_rand_arr)    # returns a sorted copy of array
my_rand_arr.argsort()   # returns indices of a sorted array
my_rand_arr.sort()      # returns None, but mutates the array by sorting it.

# AGGREGATING FUNCTIONS
# axis can be specified. axis=0 for row, axis=1 for column
my_rand_arr.sum()
my_rand_arr.mean()
my_rand_arr.std()
my_rand_arr.min()
my_rand_arr.max()

# RESHAPE / FLATTEN ARRAY
my_1D_arr_2.reshape(-1, 2) # using -1 as the number of rows means as many rows as would be
compactible with the given number of columns
my_2D_arr.reshape(-1) # flattens the array to a 1-D array. This can also be achieved using
the .flatten or .ravel methods
```

Numpy can broadcast (vectorize) most arithmetic operations over an array. Some common arithmetic operations:

```
print(f"my_rand_arr is: {my_rand_arr}")
print(f"Adding 2 to my_rand_arr gives: {my_rand_arr + 2}")
print(f"Subtracting 1 to my_rand_arr gives: {my_rand_arr - 1}")
print(f"Multiplying my_rand_arr by 3 gives: {my_rand_arr * 3}")
```

Arithmetic Operations can also be done between two arrays. These operations are also called **element-wise** operations and the two arrays must have the same length (or be broadcastable) for this to be possible.

```
print(f"arr1 + arr2 gives: {arr1 + arr2}")
```

Applying custom function to elements of an array using `.vectorize` method.

```
def my_func(x: float) -> float:
    return 3*x**2 - 1
my_vectorized_func = np.vectorize(my_func)

a = my_3D_arr.max(axis=0) # creating a 2-D array from my_3D_arr by taking the max value
along axis 0
# horizontally stacking existing arrays b and c
np.hstack([b, c])
```

Comparing runtimes

```
import time

t0 = time.time()
prime_numbers = get_prime_numbers(n)
print(f"Prime numbers up to {n} generated with for-loop after: {time.time() - t0} secs")
```

NumPy Arrays are faster than Python Lists because:

1. An array is a collection of homogeneous data-types that are stored in contiguous memory locations.
2. Python's list in is a collection of heterogeneous data types stored in non-contiguous memory locations.

More on Pandas

Pandas Series is a pandas data structure that is the building block of dataframes.

```
df[['A', 'B']]  # retrive columns A and B
df.loc[0]       # returns the first row of the DataFrame
df.loc[:, 'A']  # returns column A

# SERIES
s = pd.Series([10, 3, 1, 5, 7, -2])
s[s > 3]        # filtering the series
s.sort_values() # sorting the series by its values

# SERIES TO DF
df = s.to_frame()
```

```
# DF from DICTIONARY
df_dict = pd.DataFrame(
    {
        'A': [2, 2, 8, 22, 6, 5],
        'B': [2, -1, 3, 5, 10, 4],
        'C': ['cat', 'dog', 'bird', 'dog', 'cat', 'fish'],
        'D': ['Ragdoll', 'Golden Retriever', 'Robin', 'Bulldog ', 'Bengal', 'Bluefish']
    },
    index=[11, 12, "three", -1,  10, 10]
    # you can also specify column names if you're passing list of lists to a DataFrame.
)
# COLUMN OPERATIONS IN DF
df_dict.drop("D", axis=1)
df_dict.sort_values(by="C")
```

Filtering

~ = not, & = and, | = or

```
df2[(df2["C"] == "cat") & (df2['D'] == "Ragdoll")]
df2.loc[df2["A"] == 22] = 10          # Replace all values in rows where column "A" equals
22 with 10.
df2.loc[df2["A"] == 10, "C"] = "dog"  # Replace values in col "C" with "dog" only in rows
where col "A"==10.
```

```
titanic_df.count()    # non-null values
titanic_df.nunique()  # number of unique values per column
titanic_df["Sex"].value_counts() # count of unique values in a particular column
```

Imputation, Missing values

```
# replacing missing values in 'Age' column with the mean because its values are continuous
titanic_df['Age'] = titanic_df['Age'].fillna(titanic_df['Age'].mean())

# replacing missing values in 'Embarked' column with the mode because its values are
discrete
titanic_df['Embarked'] = titanic_df['Embarked'].fillna(titanic_df['Embarked'].mode()[0])

# dropping the 'Cabin' column because it has many missing values
titanic_df.drop("Cabin", axis=1, inplace=True)
titanic_df.isnull().sum()

# GET FULL FORM OF EMBARKMENT PORT
def get_port_of_embarkment(embarked: str) -> str:
    ports = {"C": "Cherbourg", "Q": "Queenstown", "S": "Southampton"}
    return ports.get(embarked, None)
# use the .apply function
titanic_df['Embarked'] = titanic_df['Embarked'].apply(get_port_of_embarkment)
```

```
# IMPUTE WITH LAMBDA
titanic_df['Survived'].apply(lambda x: "Yes" if x == 1 else "No").head(10)
# AGGREGATION, .agg function to rename the columns of the resulting DataFrame after
applying grouby.
titanic_df.groupby("Survived").agg(avg_age=("Age", "mean"), max_age=("Age", "max"),
avg_fare=("Fare", "mean"), max_fare=("Fare", "max"))
```

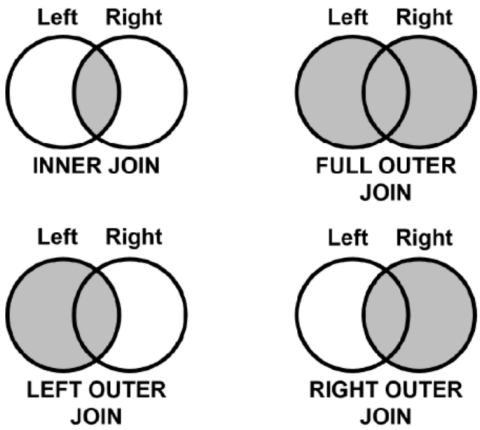
Unconditional vs. conditional imputation

Be careful when imputing missing values as they can radically change the distribution of your data. This mostly depends on the value you are imputing with and on the % of missing values. If there are many missing values, it will largely change the distribution after imputing thereby intriducing false relationships (especially for MCAR).

Conditional imputation: imputing a feature condtioned on another feature. Doing conditional imputation requires domain knowledge and analyzing features to see any relationship.

```
# Impute with median
median_col = df_copy['col'].median() # calculating the median
df_copy['col'].fillna(median_col, inplace=True) # replacing missing values
df_copy['col'].hist()   # check with a histogram

df_copy_2['col'] = df_copy_2['col'].fillna(df_copy_2.groupby('diff_col')
['col'].transform('median'))
```

Concatenation




```
# default 'how' is inner join
df = pd.merge(project_members_df, employee_df, left_on="employee_id", right_on="id",
how="inner")
df = pd.merge(employee_df, project_members_df, left_on="id", right_on="employee_id",
how="left")
```

Lab 2: Data Cleaning with Python

Viewing files with Python's requests

```
response = requests.get("https://raw.githubusercontent.com/MIE223-2025/course-
datasets/main/auto-mpg.data")
for line in response.iter_lines():
  line.decode()
```

Operating on rows and columns to clean data

```
# WORKING WITH COLUMN TYPES
df.select_dtypes(include = ["float64"]).columns   # select float64 columns
df['horsepower'] = df['horsepower'].astype(float) # cast to float
df.dtypes                                         # display column types

# LAMBDA FUNCTIONS
df['model year'] = df['model year'].apply(lambda x: "19"+str(x))
df['origin'] = df['origin'].apply(lambda x: {1: "American", 2: "European", 3: "Japanese"}
[x])

# DERIVED FEATURES - MAPPING TO CORRECT VALUES
car_company_mapper = {
    "capri": "ford", # capri cars were manufactured by ford
    "chevroelt": "chevrolet", # misspelling
    "chevy": "chevrolet",  # misspelling
}
# Check the current unique names
np.sort(df['car name'].apply(lambda x: x.strip().split()[0]).unique())

df['car company'] = df['car company'].apply(lambda x: car_company_mapper.get(x, x))
```

Normalization

1. **Data centering (`df[col] - df[col].mean()`)**

- Used when you want to center data around zero but retain its original scale.
- Useful for models sensitive to absolute values but needing mean-centered data (e.g., PCA).

2. **Min-max scaling (`(df[col] - df[col].min()) / (df[col].max() - df[col].min())`)**

- Used when you need to scale data to a fixed range [0,1].
- Common for machine learning algorithms like neural networks that perform better with bounded inputs.

3. **Z-score (`(df[col] - df[col].mean()) / df[col].std()`)**

- Used when you need to standardize data to have a mean of 0 and standard deviation of 1.

- Useful for algorithms like SVMs and k-means clustering that assume normally distributed data.

4. **Log normalization (`np.log(df[col])`)**

- Used when data is highly skewed or follows a power-law distribution.
- Helps compress large values and spread out smaller ones, improving interpretability and model performance.

```
# NORMALIZATION
df[col] - df[col].mean()  # Data centering: subtracting the mean
(df[col] - df[col].min()) / (df[col].max() - df[col].min())   # Minmax scaling
(df[col] - df[col].mean()) / df[col].std()      # Z-score
np.log(df[col])           # Log normalization
```

Lab 3: Data Visualization and Feature Analysis

- Matplotlib: object-based (explicit) vs. function-based (implicit) interfaces
    - Object-based, pyplot: create a Figure and one or more Axes objects, then explicitly use methods on these objects to add data, configure limits, set labels etc.
        - **Axes = plt.gca()**: get current axes to modify
        - **Fig = plt.gcf()**: get current figures to modify
    - Function-based, axes: create a Figure and one or more Axes objects, then explicitly use methods on these objects to add data, configure limits, set labels etc.
- Handling missing values
    - Replace with the mode of the column: **df[column_name] = df['Embarked'].fillna(df['‘column_name’].mode().iloc[0])**
    - Checking for missing values: **df.isnull().sum() / titanic_df.shape[0]**
- Univariate plots
    - Continuous:
        - Histogram: frequency distribution
        - Density plots (kde): smoothed histogram, generalized shape of a distribution
        - Boxplot: easily see percentiles and outliers.
        - Violin plot: like box plot. Created using a kernel density estimate of the underlying distribution.
    - Discrete:
        - Bar plot (plt.bar or plt.barh for horizontal)
        - Pie chart: visualizing the proportion of different categories in a dataset.
- Bivariate plots
    - Continuous vs. continuous
        - Scatter
        - Contingency Table Heatmap / Cross-tabulation Heatmap: visualizing the relationship between two categorical variables.
        - Pairplot: correlogram.
    - Discrete vs. continuous
        - Overlapping Histogram
        - Multiple boxplots
- Feature analysis
    - Continuous vs continuous
        - Correlation coefficient: **np.corrcoef(df['col1'].values, df['col2'].values)** # correlation matrix

        - Correlation matrix: **corr_arr = df.corr(method='pearson', numeric_only=True)**
    - Discrete vs discrete
        - Mutual information: define a function for that

```
def cal_PMI(var: str, target_var: str = "Survived", data: pd.DataFrame =
titanic_df) -> pd.DataFrame:
    contingency_table = pd.crosstab(index=data[var], columns=data[target_var]) #
create a contingency table

    n = contingency_table.sum().sum() # total number of data points

    p_of_x_y =  contingency_table / n # calculate joint probability P(x,y)
    p_of_x_y = p_of_x_y.replace(0, 1E-5) # replace zeros because log(0) is
undefined

    p_of_x = contingency_table.sum(axis=1) / n # calculate marginal probability
P(x)
    p_of_y = contingency_table.sum(axis=0) / n # calculate marginal probability
P(y)

    p_of_y_given_x = p_of_x_y.div(p_of_x, axis=0) # calculate conditional
probability P(y|x)
    pmi_df = np.log(p_of_y_given_x.div(p_of_y, axis=1)) # calculate PMI for all
pairs of x and y

    return pmi_df
```

Function-based interface

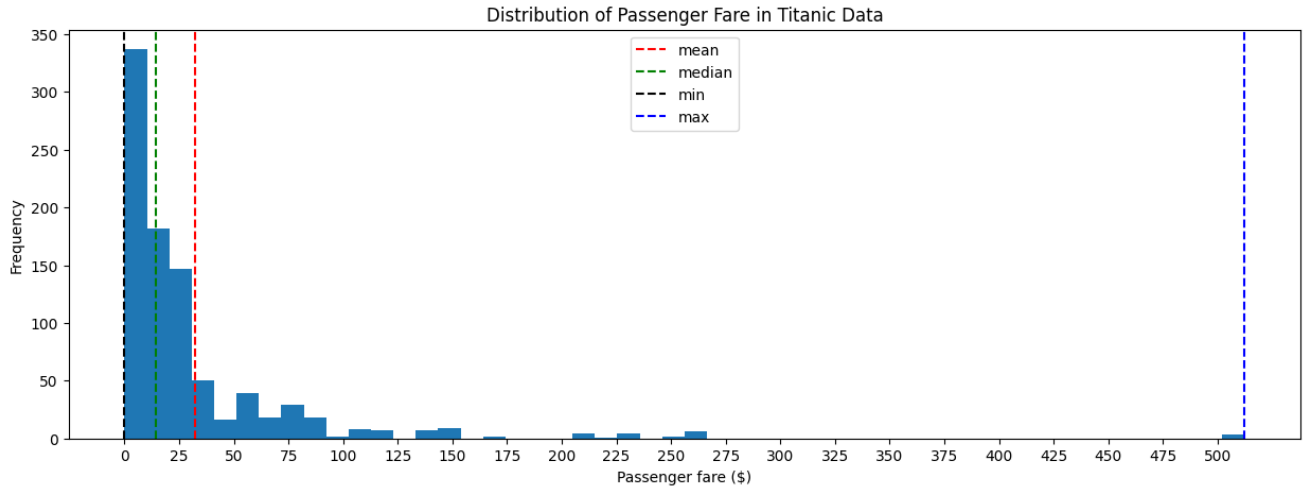
```
xpoints = np.array([0, 5])
ypoints = np.array([-25, 100])

plt.scatter(xpoints, ypoints)
plt.plot(xpoints, ypoints, marker='*', color="r", linestyle="--")   # line plot w/ configs

# CONFIGS
# ls: linestyle
# c: color
# ms: markersize
# mec: markeredgecolor
# mfc: markerfacecolor
```

Object-based interface

Histogram

```
plt.figure(figsize=(15, 5))
plt.hist(titanic_df['Fare'], bins=50)
```

```
plt.xlabel("Passenger fare ($)")
plt.xticks(np.arange(0, titanic_df['Fare'].max(), 25))
plt.ylabel("Frequency")
plt.title("Distribution of Passenger Fare in Titanic Data")

plt.axvline(titanic_df['Fare'].mean(), color='red', ls='--', label='mean')
plt.axvline(titanic_df['Fare'].median(), color='green', ls='--', label='median')
plt.axvline(titanic_df['Fare'].min(), color='black', ls='--', label='min')
plt.axvline(titanic_df['Fare'].max(), color='blue', ls='--', label='max')

plt.legend(loc="upper center")

plt.show()
```



Density plot

```
ax = sns.histplot(data = titanic_df,
                  x =  'Age',
                  kde=True # creates a smoothed line that represents the distribution of
the data
    )
```

Box-plot

```
ax = sns.boxplot(data=titanic_df, # dataframe
                 x='Age' # column we care about
    )
```

Bar plot

```
count_by_sex = titanic_df['Sex'].value_counts()
plt.bar(count_by_sex.index, count_by_sex.values)
plt.barh(count_by_sex.index, count_by_sex.values) # horizontal bar diagram
```

Pie chart

```
Sex_Pclass_count = titanic_df["Sex_Pclass"].value_counts() / titanic_df.shape[0]

plt.pie(Sex_Pclass_count.values, labels=Sex_Pclass_count.index,
        autopct='%.1f%%', # display percent on the pie chart up to 1 decimal place
        explode=(0, 0, 0, 0, 0.1, 0)) # make one of the slices pop out
```

Pairplot (Correlogram)

visualize all pairs of continuous variables pairplots.

```
sns.pairplot(iris_df, hue="species", height=1.5);
```

Correlation matrix heatmap

```
ax = sns.heatmap(data=corr_arr,
                 cmap='coolwarm', # colorscheme
                 annot=True,# display the correlation values on the heatmap
                 fmt=".2f", # format the correlation values to 2 decimal places
                 xticklabels=index,
                 yticklabels=columns,
                 center=0) # center the colorbar at 0 correlation
```

Subplots

```
plt.subplot(1, 2, 2)    # 1st row, 2nd column, 2nd subplot in the figure
```

Lab 4: Natural Language

- NLTK is a comprehensive Python library for working with natural language content.
- Word tokenization: Splitting a string into a substring
  - **punkt_tokenized = word_tokenize(example_text) # intelligent**
  - **regex_tokenized = wordpunct_tokenize(example_text) # puntuation only**
  - **newsgroups_df['words'] = newsgroups_df['text'].apply(lambda x: nltk.word_tokenize(x))**
- Stopwords: "a", "the", "and", and other words that we want to remove since they don't carry much meaning.
  - **stop_words = set(stopwords.words("english"))**

```
import re
import nltk

nltk.download("stopwords")
from nltk.corpus import stopwords

# Get frequency of all non stopwords in dataset
token_count_ng_nsp = Counter()

# Create no stopwords column
newsgroups_df['words_nsp'] = newsgroups_df['words'].apply(lambda x: [word for word in x if
word not in
stop_words]) # Removes stopwords
for words in newsgroups_df['words_nsp']:
  token_count_ng_nsp.update(words)

# Put the frequency of all words in a dataframe
word_freq_ng_nsp = pd.DataFrame.from_dict(token_count_ng_nsp, orient='index', columns=
['frequency']).sort_values(by='frequency', ascending=False)
```

- Stemming: extracting the root of a word for its meaning.
  - **stemmer = PorterStemmer(); # or SnowballStemmer for higher accuracy**
  - **stemmer.stem(word); # word is a string**
  - False Positive: Overstemming. False Negative: Understemming.
    - Overstemming: university and universal stem to the same word (univers) despite both meaning different things.
    - Understemming: discovered and discoveries stem to different words despite both starting with the same base word (discover).

```
stemmer = PorterStemmer()
punkt_example_words = [word.lower() for word in word_tokenize(example_text)]

stemmed_words = []
```

```
for word in punkt_example_words:
  stemmed_words.append(stemmer.stem(word))
  #goes through tokenized words and takes the stemmed version

# display as columns of a df
stemmer_df = pd.DataFrame({'word': punkt_example_words, 'stemmed_word': stemmed_words})

# SnowballStemmer
from nltk.stem.snowball import SnowballStemmer
snowball_stemmer = SnowballStemmer("english")
words = ["Running", "Studies", "Happily", "Organization", "Generous",
         "Nationality", "Flying", "Leaves", "Quickly", "Argument"]
stemmed_words = [stemmer.stem(word) for word in words]
for original, stemmed in zip(words, stemmed_words):
    print(f"{original} → {stemmed}")
```

- Lemmatization: extract meaning of word.
  - Produces an actual word (rather than just a stem) which carries the core meaning of the original word.

```
lemmatizer = WordNetLemmatizer()

lemmatized_words = []
for word in punkt_example_words:
  lemmatized_words.append(lemmatizer.lemmatize(word))

# add to stem df
stemmer_df['lemmatized_word'] = lemmatized_words
```

Lemmatization vs. Stemming

```
stemmer_df[stemmer_df['word'].isin(["students", "discovered", "discoveries", "assignment",
"assigned", "university", "universal"])].sort_values(by='word')
```

	word	stemmed_word	lemmatized_word
20	assigned	assign	assigned
9	assignment	assign	assignment
25	discovered	discov	discovered
28	discoveries	discoveri	discovery
1	students	student	student
34	universal	univers	universal
4	university	univers	university

- Part of speech tagging
  - **sample_pos_tags = nltk.pos_tag(punkt_tokenized)** # yes, that's it
  - There are 8 POS in English. NLTK uses the Brown Corpus for POS tagging.

1. Noun (NN): Is a person, place, or thing
2. Pronoun (PRP): Replaces a noun
3. Adjective (JJ): Gives information about what a noun is like
4. Verb (VB): Is an action or a state of being
5. Adverb (RB): Gives information about a verb, an adjective, or another adverb
6. Preposition: Gives information about how a noun is related to another word in the sentence
7. Conjunction: Connects words, phrases, or clauses
8. Interjection: Expresses emotion or surprise

- Chunking: Chunking uses POS tags to group words together.
  - Noun phrases: Group of words that acts like nouns in a sentence.
  - **sent_pos_tags = [nltk.pos_tag(word_tokenize(sent)) for sent in sent_tokenized]** # yes, that's it

    The nltk.pos_tag function takes a list of words (or tokens) as input and returns a list of tuples. Each tuple contains a word and its corresponding POS tag.
- Zipf plot: word frequency distribution.

```
# get frequency of all words in dataset
token_count_ng = Counter()

for words in newsgroups_df['words']:
  token_count_ng.update(words)

# put the frequency of all words in a dataframe
word_freq_ng = pd.DataFrame.from_dict(token_count_ng, orient='index', columns=
['frequency']).sort_values(by='frequency', ascending=False)
#orient because keys of the dictionary represent the index of the dataframe
word_freq_ng.head()
```

- Mutual Information (MI): information that is shared in the non-linear realtionships between tokens and topics.

Noun phrase extraction via RegEx and POS

```
NP: {<DT>?<JJ>*<NN>}
```

Breakdown: \ **NP**: noun phrase \ **DT**: determiner (ex. "a", "the") \ **?**: says that DT is optional \ **JJ**: adjective \ *****: there can be any number of adjectives \ **NN**: noun \ **+**: match at least 1 noun \

Visualize with POS tree

```
np_grammar = "NP: {?*+}" #0 or more adjectives
cp = nltk.RegexpParser(np_grammar) #creates a chunk parser based on regular expressions
for sent_tags in sent_pos_tags:
  #loops over, where each sentence is represented as a list of POS tagged words
  np_chunks = cp.parse(sent_tags)
```

```
  #for each sentence, cp parses the POS tagged words and identifies noun phrases
  #result is stored in np_chunks
  display(np_chunks) #displayed the parsed tree structure of the sentence,
  #highlighting identified noun phrases
  # print NPs
  for subtree in np_chunks.subtrees():
    if subtree.label() == 'NP':
      tokens, pos = zip(*subtree.leaves())
      #separates tokens (words) and their corresponding POS tags
      #prints tokens (words) that constitute each noun phrase
      print(tokens)

#.leaves() returns a list of all the leaves in the subtree. each leaf is represented
#as a tuple containing the word and its associated POS tag

#zip(*subtree.leaves()): zip combines elements of multiple iterables into tuples
#combines words and POS tags from the leaves
#* unpacks the list of tuples into separate lists of words and POS tags
#tokens, pos = zip(*subtree.leaves()): The unpacking operation assigns the first
#component of each tuple (words) to the variable tokens and the second component
#(POS tags) to the variable pos. This line essentially separates the words and
#POS tags of the leaves into two lists

#For example, if subtree.leaves() returns [(('The', 'DT'), ('cat', 'NN'))], the
#unpacking operation would result in tokens = ('The', 'cat') and pos = ('DT', 'NN')
```

1. **Parse tree (Syntactic tree)**

- A **parse tree** represents the **full syntactic structure** of a sentence based on a formal grammar (like Context-Free Grammar, CFG).
- It **breaks down a sentence hierarchically** into phrases and components such as noun phrases (NP), verb phrases (VP), and sentence (S).
- The tree is **derived from the grammar rules** of a language.

2. POS (Part-of-Speech) Tree

- A **POS tree** is a shallower tree that only represents the **parts of speech** (POS tags) of words in a sentence.
- Each word is labeled with its **grammatical category** such as noun (NN), verb (VB), adjective (JJ), etc.
- It does **not show phrase-level structures** like NP (Noun Phrase) or VP (Verb Phrase).

Mutual Information

Mutual Information (MI) tells us the information that is shared in the non-linear relationships between tokens and topics. We will calculate the MI of the non-stopwords in the data with the topics. We will use the top 500 most common words (tokens) to do this.

```
# Using topk words, we will do mutual information
# get the top 500 words by frequency
word_freq_ng_nsp['word'] = word_freq_ng_nsp.index
```

```
#creates a new column containing the words from the dataframe
topk = [tuple(x) for x in word_freq_ng_nsp.head(500).to_numpy()]

#selects top 500 rows from the df using head(500)
# to_numpy converts the selected rows to a numpy array
#for each row in the array, it creates a tuple where x represents the values in that row
#resulting list topk contains tuples, where each tuple represents a row of the top 500
words along with their frequencies

# create a binary list for each row in the dataset to indicate if it contains each word in
the top 500
freqText = []
for entry in newsgroups_df['words_nsp']:
    tempCounter = Counter(entry)
    topkinText = [1 if tempCounter[word] > 0 else 0 for (wordCount,word) in topk]
    freqText.append(topkinText)
    #appends the binary frequency representation of the current document to the freqText
list

# Turn these lists into a DataFrame
freqTextDf = pd.DataFrame(freqText)
dfName = []
for c in topk:
    dfName.append(c[1])
freqTextDf.columns = dfName

# Merge DataFrame with dataset DataFrame
finalTextDf = newsgroups_df[['text', 'target','topic']].join(freqTextDf)

def getMI(topk, df, label_column='topic'):
    unique_labels = df[label_column].unique()

    # create a placeholder df
    overallDf = pd.DataFrame()
    #empty df to store the overall MI scores for all topics and words

    # loop through each topic
    for topic in unique_labels:
      miScore = []

      # create a binary column for the current topic
      label_col = df[label_column].copy()
      #indicating whether each entry in the original label column matches the
      #current topic

      label_col[label_col != topic] = 0
#This line sets all values in the label_col column that do not equal the specified topic
to 0.
#marking all entries that don't correspond to the current topic as 0.
      label_col[label_col == topic] = 1
```

```
#calculate MI scores for each word with the current topic
#loops through each word in the top k words
#computes the MI score between the binary label column and the column corresponding to the
current word
#appends the word, MI score, and topic to the 'miscore' list

      # get the MI score for each word with the current topic
      for word in topk:
          miScore.append([word[1]]+[metrics.mutual_info_score(label_col, df[word[1]])] +
[topic])
# word[1] retrieves the actual word from the tuple 'word'
# calculates MI between binary column label_col and column in df corresponding to the
current word
# creates a list with three elements: word, MI score for the current word and topic, the
topic

      # combine the scores of all words for the topic into the df
      miScoredf = pd.DataFrame(miScore).sort_values(1,ascending=0)
      miScoredf.columns = ['Word','MI Score','Topic']
      overallDf = pd.concat([overallDf, miScoredf])

    return overallDf
mi_scores = getMI(topk, finalTextDf)
```

Lab 5: Sentiment analysis

- Analyze sentiment with Vader: rule-based model.
  - Access vader with NLTK: **sid = SentimentIntensityAnalyzer()**
  - Polarity score of a string: **sid.polarity_scores(string)**

```
# Use vader to evaluated sentiment of reviews
def evalSentences(sentences, to_df=False, columns=[]):

    # Instantiate an instance to access SentimentIntensityAnalyzer class
    # from vader in nltk
    sid = SentimentIntensityAnalyzer()
    pdlist = []

    if to_df:
        for sentence in tqdm(sentences):
            ss = sid.polarity_scores(sentence)
            pdlist.append([sentence]+[ss['compound']])
        reviewDf = pd.DataFrame(pdlist)
        reviewDf.columns = columns
        return reviewDf
    else:
        for sentence in tqdm(sentences):
            print("\n" + sentence)
            ss = sid.polarity_scores(sentence)
            for k in sorted(ss):
                print('{0}: {1}, '.format(k, ss[k]), end='')
            print()

reviews = hotelDf['reviewColumn'].values
reviewDF = evalSentences(reviews, to_df=True, columns=['reviewCol','vader'])
```

- Find the words the convery the highest sentiment by Mutual Information
  - Step 1: Count the most common words:

```
counter = Counter()
for review in reviews: # go through each review and feed words into counter
  counter.update([word.lower()
  for word in word_tokenize(review):
    if word.lower() not in stop and len(word) > 2]):
      topk = counter.most_common(kwords) # kwords is an integer.
```

  - Step 3: Compute MI:

```
for word in topk:
    # label_column: ground truth/sentiment label.
    miScore.append([word[0]]+[metrics.mutual_info_score(df[label_column], df[word[0]])])
```

- Computing PMI:

```
def demo_pmiCal(df,word):
    pmilist=[]
    N = df.shape[0]
    for sentiment in ['positive','negative']:
        for word_present in [False,True]:
            px = sum(df['groundTruth']==sentiment)
            py = sum(df[word]==word_present)
            pxy = len(df[(df['groundTruth']==sentiment) & (df[word]==word_present)])

            if pxy==0:#Log 0 cannot happen
                pmi = math.log((pxy+0.0001)*N/(px*py))
            else:
                pmi = math.log(pxy*N/(px*py))

            pmilist.append([sentiment]+[word_present]+[px]+[py]+[pxy]+[pmi])

    # assemble the results into a dataframe
    pmidf = pd.DataFrame(pmilist)
    pmidf.columns = ['sentiment (x)','word_present (y)','px','py','pxy','pmi']
    return pmidf

demo_pmiCal(finaldf,'great')
```

Get TopK words based on review sentiment

```
def getTopK(df, kwords, label_value, label_column='groundTruth', operation=operator.eq,
value_column='reviewCol'):
    stop = get_stop_words()
    counter = Counter()
    reviews = df.loc[operation(df[label_column],label_value)][value_column]
    for review in reviews:
          counter.update([word.lower()
                          for word
                          in word_tokenize(review)
                          if word.lower() not in stop and len(word) > 2])
    topk = counter.most_common(kwords)
    return topk
```

```
#We are only intereseted in this three column for overall analysis
itemAnalysisDf = finaldf[['reviewCol','groundTruth','vader']]
itemAnalysisDf.head()
```

Chunking: PerceptronTagger in NLTK

- NBAR matches any phrase that starts with any number of nouns (POS tag starts with NN) or adjectives (JJ) and ends with a noun (POS tag starts with NN).
    - NBAR: {<NN.|JJ><NN.*>} # Nouns and Adjectives, terminated with Nouns

- NP matches any phrase that is NBAR or 2 NBAR phrases connected with in/of/etc
  - NP: {} {} # Above, connected with in/of/etc...
- chunker = nltk.RegexpParser(grammar_with_regex)

Put results above in a tree:

```
grammar = r"""
    NBAR:
        {<NN.*|JJ>*<NN.*>}  # Nouns and Adjectives, terminated with Nouns

    NP:
        {<NBAR>}
        {<NBAR><IN><NBAR>}  # Above, connected with in/of/etc...
"""
chunker = nltk.RegexpParser(grammar)

# to make the results more useable, we clean up the tree results shown above.
lemmatizer = nltk.WordNetLemmatizer()
stemmer = nltk.stem.porter.PorterStemmer()
stopword_list = get_stop_words()

# generator, create item one a time
def get_terms(tree):
    for leaf in leaves(tree):

        term = [normalise(w) for w,t in leaf if acceptable_word(w) ]
        # Phrase only
        if len(term)>1:
            yield term

# generator, generate leaves one by one
def leaves(tree):
    """Finds NP (nounphrase) leaf nodes of a chunk tree."""
    for subtree in tree.subtrees(filter = lambda t: t.label()=='NP' or
t.label()=='JJ' or t.label()=='RB'):
        yield subtree.leaves()

# stemming, lematizing, lower case...
def normalise(word,lemmatizer=lemmatizer, stemmer=stemmer):
    """Normalises words to lowercase and stems and lemmatizes it."""
    word = word.lower()
    word = stemmer.stem(word)
    word = lemmatizer.lemmatize(word)
    return word

# stop-words and length control
def acceptable_word(word, stopword_list=stopword_list):
    """Checks conditions for acceptable word: length, stopword."""
    accepted = bool(2 <= len(word) <= 40
        and word.lower() not in stopword_list)
    return accepted
```

```
# Flatten phrase lists to get tokens for analysis
def flatten_phrase_lists(npTokenList):
    finalList =[]
    for phrase in npTokenList:
        token = ''
        for word in phrase:
            token += word + ' '
        finalList.append(token.rstrip())
    return finalList
```

- Time series: rolling mean of ratings.
  - Get the running mean: df['ratingScore'].rolling(3).mean().plot(ax=ax[1],label=hotel)
  - Hold the last value constant in time periods when no new values are added: .fillna(method='ffill')

```
# It can be useful to see when reviews were being made and how the ratings changed
using a *running* mean
fig, ax = plt.subplots(2, 1, figsize=(15,7),
                       sharex=True,
                       gridspec_kw={
                           'height_ratios': [1, 2]})

for hotel in five_hotels[:3]:
  _df = hotelDf[hotelDf['hotelName']==hotel].set_index('date_stamp')
  _df.index = pd.to_datetime(_df.index)
  _df = _df.sort_index()
  _df['count'] = 1
  _df['count'].cumsum().plot(ax=ax[0],label=hotel, marker='o')
  _df['ratingScore'].rolling(3).mean().plot(ax=ax[1],label=hotel)

ax[1].set_ylabel('Avg Rating')
ax[0].set_ylabel('Number of Reviews')
plt.legend()



# Rating average per year across all hotels over time
# It can be useful to see when reviews were being made and how the ratings changed
fig, ax = plt.subplots(2, 1, figsize=(15,7),
                       sharex=True,
                       gridspec_kw={
                           'height_ratios': [1, 2]})

_df = hotelDf.set_index('date_stamp')
_df.index = pd.to_datetime(_df.index)
_df = _df.sort_index()
_df['count'] = 1
_df_yearly =
_df.groupby(pd.Grouper(freq='Y')).agg({'count':'sum','ratingScore':'mean'})
_df_yearly['ratingScore'] = _df_yearly['ratingScore'].fillna(method='ffill') # hold
the last rating constant in years with no reviews
```

```
_df_yearly['count'].plot(ax=ax[0],marker='o')
_df_yearly['ratingScore'].plot(ax=ax[1])

ax[1].set_ylabel('Avg Rating')
ax[0].set_ylabel('Number of Reviews')
plt.legend()
```

- Geolocation
  - locator = Photon(user_agent='myGeocoder')
  - geocode = RateLimiter(locator.geocode, min_delay_seconds=1) # delay between geocoding calls
  - geo_rating_df['location'] = geo_rating_df['formed_address'].apply(geocode)
  - geo_rating_df['point'] = geo_rating_df['location'].apply(lambda loc: tuple(loc.point) if loc else None)
  - geo_rating_df[['latitude', 'longitude', 'altitude']] = pd.DataFrame(geo_rating_df['point'].tolist(), index=geo_rating_df.index)

Lab 6: Time series analysis

- Resampling: adjusting the time scale of the time-series data.
  - Upsampling: adjusting the time-scale from a lower frequency to a higher frequency (e.g. months to weeks). Done with linear interpolation.
  - Downsampling: adjusting the time scale from a higher frequency to a lower frequency (e.g. months to years)
    - Compute an average for weekly downsampling: stocks_weekly = stocks_df.groupby('company_name').resample('W').mean()
- Smoothing: reduce the effects of noise in the data to help uncover trends.
  - Moving average: simple and weighted. Causal vs non-causal (causal includes future values in time window).

Moving Average

A moving average is similar to the concept of a rolling window function from the previous lab.

Simple Moving Average:

The simplest definition of a moving average over a time window k (e.g. k = 3 days, k = 5 months) is as follows:

$$S_t = \frac{\sum_{i=1}^{k} X_{t-k+i}}{k}$$

where X_{t-k+i} corresponds to values (or observations) within that rolling window.

Weighted Moving Average:

A weighted moving average assigns a weight to each value in the time window.

$$S_t = \sum_{i=1}^{k} w_{t-k+i} \cdot X_{t-k+i}$$

Note that $\sum_{i=1}^{k} w_{t-k+i} = 1$. Often, higher weights are assigned to values at times closer to t.

Note: Downsampling using average seems to have an indirect smoothing effect on the data.

```
stocks_rolling_week = stocks_df_grouped.rolling(window='5D',
min_periods=1).mean()
# min_periods=1 means that if there are less than 5 days, it will still
calculate the mean

# Plot original and rolling col by company on same subplot
stocks_df_grouped = stocks_df.groupby('company_name', group_keys=True)
fig, ax = plt.subplots(num_groups, 1, figsize=(15,15))
for i, (name, group) in enumerate(stocks_df_grouped):
    group.plot(ax=ax[i], y='col', label='Original', title=f"{name} col",
xlabel="Date", ylabel="col")
    stocks_rolling_week.loc[name].plot(ax=ax[i], y='col', label='5D Rolling',
xlabel="Date", ylabel="col")
```

    - Exponential smoothing: s_t = ax_t + (1-a)s_(t-1).
        - More computationally efficient than moving average.
        - Estimate for s_t affected by all values, as opposed to just recent values.
        - Easier to fine-tune.
        - **stocks_exp_01 = stocks_df_grouped.ewm(alpha=0.1, adjust=False).mean()** # ewm, adjust=False mandatory for recursive calculation
- Time series shifting/lagging: translating a time signal n time steps forward/backward.
    - **stocks_lag_60_forward = stocks_df_grouped.apply(lambda x: x.shift(periods=60)) # next 60 periods of time**
- Correlation: 2 common patterns are trends and seasonality.
    - Trends: consistent growth or decline over time (often linear)
    - Seasonality: periodic pattern (like a sine wave)
    - Autocorrelation: detect possible trends and/or seasonality. Correlation between the original time-series data and a time-shifted version of it. **pd.plotting.autocorrelation_plot(group['Adj Close'])**
        - Seasonal: Remove values that aren't at the index intersection: index_intersection = city_1_df.index.intersection(city_2_df.index).intersection(city_3_df.index)
        - The periodic peaks/valleys in autocorrelation plots is evidence of seasonality. The period is about 12 months which is a calendar year. The peaks occur when lagging the data aligns the seasons (e.g. winter-winter, summer-summer). The valleys occur when lagging the data misaligns the seasons (e.g. winter-summer, summer-winter). Note that the decreasing magnitude of the peaks/valleys suggest the presence of a trend as well (e.g. warmer temperatures due to climate change).
    - Cross-correlation accomplishes the same thing as autocorrelation, except with two different time-series data sources.
        - ccf_12 = ccf(city_1_monthly['AvgTemperature'], city_2_monthly['AvgTemperature'])
        - e.g. The plot shows that the seasonal cycle for Toronto/Ottawa and Sydney differ by 6 months, while Toronto and Ottawa share the same seasonal cycle.

          Plotting lag plots

          ```
          pd.plotting.lag_plot(city_1_monthly['AvgTemperature'], ax=ax[0], lag=12) # lag = 12 means 12 months
          ```

Lab 7: Graphs & Networks

- Graphs/Network structure: collection of nodes that have data and are connected to other nodes.
  - Nodes: any hashable object (e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.)
- Ebunch: iterable container (list) of edge-tuples.

```
G = nx.Graph()                  # Undirected graph

# Adding one node at a time to graph
G.add_node("A")
G.add_node(1)

# Adding several nodes at a time
G.add_nodes_from(["B","C"])
G.add_nodes_from("Hi")            # This will add node H and node i

# Adding edges
G.add_edge("A", "B")              # Start of edge, end of edge
e = ("B", "C")
G.add_edge(*e)                    # Unpack edge tuple*

# Add an ebunch
ebunch = [(1, 2), (2, 3), (3, 4)]
G.add_edges_from(ebunch)
G.add_edges_from([(1, "A"),(1, "B")]) # List of tuples. Edges from 1 to "A" and 1 to "B"


# Graph information
print("Number of nodes:", G.number_of_nodes(), ", they are:", list(G.nodes))
print("Number of edges:", G.number_of_edges(), ", they are:", list(G.edges))

# Visualize the graph
nx.draw_networkx(G)

# Clear graph
G.clear()
```

Adjacency lists with sparse matrices in Scipy

```
from scipy.sparse import coo_matrix

# Represent adjacency list with dictionary
adj_list = {
    0: [1, 2],
    1: [0],
    2: [1]
}
```

```
rows = []
cols = []

for src, neighbors in adj_list.items():
    for dst in neighbors:
        rows.append(src)
        cols.append(dst)

data = np.ones(len(rows))  # or use edge weights here instead of default ones
num_nodes = max(max(rows), max(cols)) + 1
adj_matrix = coo_matrix((data, (rows, cols)), shape=(num_nodes, num_nodes))
adj_matrix = adj_matrix.tocsr()
```

Degrees and Adjacency

- Degree: number of edges that are connected to the node. Aka valency.

```
Directed_G = nx.DiGraph()         # Directed graph
Directed_G.add_edge(1, 2)         # Edge from 1 to 2

# Adjacency: outgoing edges for each node in directional graphs
for k, v in Directed_G.adjacency():
    print("Node", k, "is adjacent to", list(v)) # Node 1 is adjacent to [2], Node 2 is
adjacent to [].
# Display in-degree and out-degree.
for k,v, in Directed_G.in_degree():
  print("Node", k , "has in-degree", v)
for k,v, in Directed_G.out_degree():
  print("Node", k , "has out-degree", v)

# Cast directed graph to undirected graph
Undirected_G = Directed_G.to_undirected()
for k, v in Undirected_G.adjacency():
    print("Node", k, "is adjacent to", list(v))
# Display degree
for k,v, in Undirected_G.degree():
  print("Node", k , "has degree", v)
```

Attributes

- Attributes: information about graph, edge, or node.

> ⚠ **Warning**
>
> You can't edit attributes for things that haven't been initialized. They won't get initialized automatically.

```
# You can set the attributes to be equal to any key:value pair.
G = nx.Graph(day="Friday", mood="happy", is_cool=True, weight_of_universe=42)
```

```
# Modify graph attributes
G.graph['day'] = "Monday"

# Node attributes
G.add_node(1, color="blue", size=10)
G.nodes[1]["color"] = "blue"
G.nodes[1]["size"] = 10
nx.set_node_attributes(G, {1: {"color": "green"}, 2: {"color": "yellow"}})
print(G.nodes[1])       # {'color': 'blue', 'size': 10}

# Edge attributes
G.add_edge(1, 2, weight=5, label="friend")
G[1][2]["weight"] = 5
G[1][2]["label"] = "friendship"
nx.set_edge_attributes(G, {(1, 2): {"weight": 10}})
print(G[1][2])          # {'weight': 5, 'label': 'friendship'}
G.get_edge_data(1, 2)   # {'weight': 5, 'label': 'friendship'} (dictionary of attributes)

# Get all the edge weights, gives us position w.r.t a certain layout
edge_weight = nx.get_edge_attributes(G,"weight")

# Display nodes/edges with attributes
list(G.nodes(data=True))[:5]      # [(1, {'color': 'red'}), (2, {'color': 'blue'})]
list(G.edges(data=True))[:5]      # [(1, 2, {'weight': 4}), (2, 3, {})]
```

Graph visualisations

```
from matplotlib.lines import Line2D

# color the nodes by faction
# This color list will be passed to the draw function: color[i] -> nodes[i]'s color
color = []
for node in G.nodes():
    # Main nodes
    if node==0:
        color.append("C0")
    elif node==33:
        color.append("C1")
    else:
      # Other nodes
      if G.nodes.data()[node]["club"] == "Mr. Hi":
          color.append("C2")
      else:
          color.append("C3")

# Set Different Marker and Legend for the two main nodes
main_node1_legend = Line2D(
    [],
    [],
    markerfacecolor="C0",
```

```
    markeredgecolor="C0",
    marker="o",
    linestyle="None",
    markersize=14,
)
main_node2_legend = Line2D(
    [],
    [],
    markerfacecolor="C1",
    markeredgecolor="C1",
    marker="o",
    linestyle="None",
    markersize=14,
)

# Set Different Marker and Legend for the two club members
other_node1_member = Line2D(
    [],
    [],
    markerfacecolor="C2",
    markeredgecolor="C2",
    marker="o",
    linestyle="None",
    markersize=8,
)
other_node2_member = Line2D(
    [],
    [],
    markerfacecolor="C3", # Set color
    markeredgecolor="C3",
    marker="o",
    linestyle="None",
    markersize=8,
)
```

Graph layouts

- Graph layouts: mapping function of nodes -> coordinate positions.
    - Spectral embeddings: good for partitioning the graph into clusters.
    - Spring layout: node separation.
    - Circular layout: better visualization on edges and degree of nodes.

```
fig, ax = plt.subplots(figsize=(10, 10))
ax.legend(
    [main_node1_legend, main_node2_legend, other_node1_member, officer_club_legend],
    ["Main node 1","Main Node 2", "Main node 1's club", "Main node 2's club"],
    loc="upper left",
    bbox_to_anchor=(1, 1),
    title="Faction",
)
```

```
# Spectral embeddings
ax.set_title("Spectral Embeddings Layout", fontsize=10)
nx.draw_spectral(
    G, with_labels=True, node_color=color, ax=ax, font_color="white", node_size=800
)
plt.show()

# Spring layout
ax.set_title("Spring Layout", fontsize=10)
nx.draw_spring(
    G, with_labels=True, node_color=color, ax=ax, font_color="white", node_size=800
)
plt.show()

# Kamada-Kawai layout
ax.set_title("Kamada Kawai Layout", fontsize=10)
nx.draw_kamada_kawai(
    G, with_labels=True, node_color=color, ax=ax, font_color="white", node_size=800
)
plt.show()

# Circular layout
ax.set_title("Circular Layout", fontsize=10)
nx.draw_circular(
    G, with_labels=True, node_color=color, ax=ax, font_color="white", node_size=800
)
plt.show()
```

Centrality

- Centrality: Highly central nodes usually play a key role of a network, serving as hubs for different network dynamics. Measures of centrality:
    - Degree: amount of neighbors of the node. Fraction of nodes a node is connected to.

```
deg_centrality = nx.degree_centrality(G)
list(deg_centrality.items())[:5]
plt.bar(deg_centrality.keys(), deg_centrality.values(), color='g')

# Top-5 nodes by degree
sorted(deg_centrality.items(), key=lambda x:x[1], reverse=True)[0:5]
```

    - PageRank: iterative circles of neighbors. A higher PageRank score represents influential nodes who can spread their content to a community much faster compared to nodes with lower PageRank score.

```
pr_centrality = nx.pagerank(G)
draw_centrality(G, pr_centrality, node_scale=2e4)
```

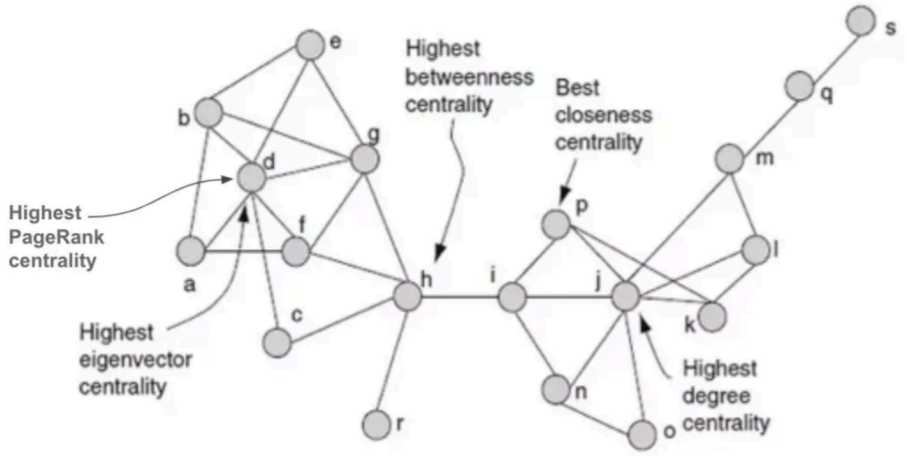
- Closeness: the level of closeness to all of the nodes. Reciprocal of the average shortest path distance to node from all other nodes. Closeness centrality of a node u is the reciprocal of the average shortest path distance to u over all $n-1$ reachable nodes. $C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}, d(v,u)$ is shortest path between v and u.

> ⚠ **Warning**
>
> Notice that the closeness distance function computes the incoming distance to u for directed graphs. To use outward distance, act on G.reverse().

```
cl_centrality = nx.closeness_centrality(G)
draw_centrality(G, cl_centrality, node_scale=2e3)
```

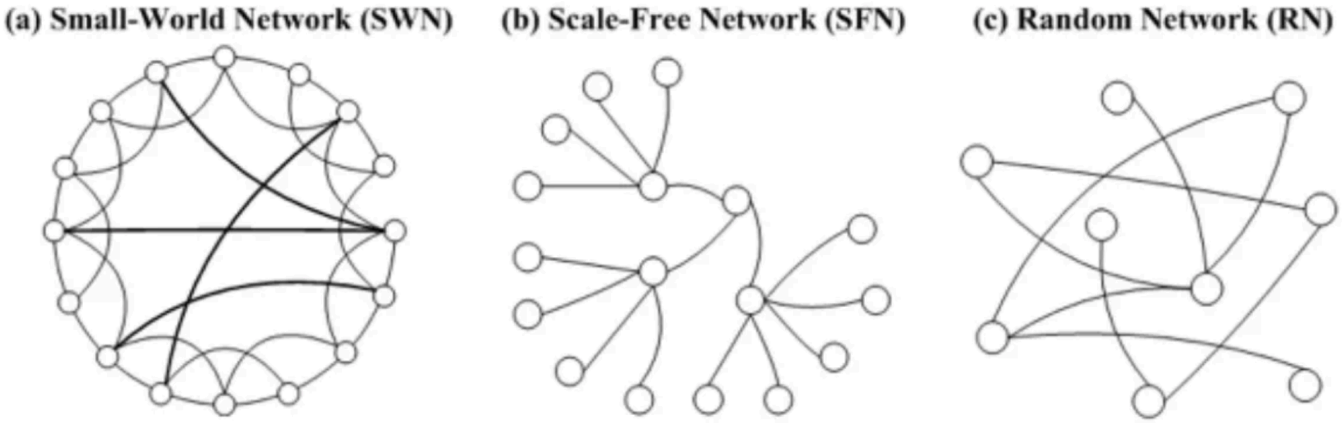
- Betweenness: the amount of short paths going through the node. Sum of the fraction of all-pairs shortest paths that pass through the node.

```
def draw_centrality(G: nx.Graph, centrality_measure: dict, node_scale: float):
    """
    G: the graph to draw
    centrality_measure: your centrality_measure returned from nx.centrality_measures
    node_scale: the scale of the node size
    """
    fig, ax = plt.subplots(figsize=(12, 10))

    # centrality, cast it to a np.array of floats
    centrality = np.fromiter(centrality_measure.values(), float)

    # plot
    pos = nx.spring_layout(G, seed=42)

    # node_color=centrality and cmap=plt.cm.copper will map the centrality to a linear
colorspace
    # node_size=centrality*node_scale will scale the node_size w.r.t. the centrality
    nx.draw(G, pos, ax=ax, node_color=centrality, node_size=centrality*node_scale,
cmap=plt.cm.copper)

    # we also draw the node labels
    nx.draw_networkx_labels(G, pos, ax=ax, font_size=10, font_color="white")

    # colorbar
    sub_ax = fig.add_axes([0.95, 0.13, 0.01, 0.09])
    plt.colorbar(plt.cm.ScalarMappable(cmap=plt.cm.copper),sub_ax)
    plt.show()
```



Real-world Networks

- Scale Free networks with power-law degree distribution have a skewed population with a few highly-connected nodes (such as social-influences) and a lot of loosely-connected nodes.
  - The degree distribution of a real social network can (usually) fit a power law distribution in this form: $f(k) = ck^{\alpha}$, k: degree of nodes, c: constant, α: law's exponent.
- Small World phenomenon claims that real networks often have very short paths (in terms of number of hops) between any connected network members.



- Homophily: tendency of individuals to associate and bond with similar others, which results in similar properties among neighbors.

```
# Get hashtags
from pprint import pprint

def print_hashtags(tweets_df,top=100):
  allTweets = tweets_df["text"].str.cat(sep=' ')
  tweetWords = [word.strip("""  ,.:'\";""").lower() for word in allTweets.split()]
  hashTags = [word for word in tweetWords if word.startswith("#")]
  hashTagsCounter = Counter(hashTags)
  pprint(hashTagsCounter.most_common(top))

print_hashtags(tweets_df, 40)
```

```
trumpTag = tweets_df[tweets_df["hashtags"].str.lower().str.contains("'trump'", na=False,
regex=False)].copy()

# Using mentions on Twitter to build a graph
def addMentionedColumn(df):
    def mentionsList(txt):
        allWords = [word.strip(""" ,.:'\";""").lower() for word in txt.split()]
        allNames = [word.strip("@") for word in allWords if word.startswith("@")]
        uniqueNames = list(set(allNames))
        return uniqueNames

    df["mentioned"] = df["text"].apply(mentionsList)
addMentionedColumn(trumpTag)
trumpTag = trumpTag[["date","user_followers","user_name","text","mentioned"]]
trumpTag.head(100)

# Mention graph
def mentionGraph(df):
    g = nx.Graph()
    for (index, date,user_followers, user, tweet, mentionedUsers) in df.itertuples():
        for mentionedUser in mentionedUsers:
            if mentionedUser in df['user_name']:
              print(mentionedUser)
            if (user in g) and (mentionedUser in g[user]):
                g[user][mentionedUser]["numberMentions"] += 1
            else:
                g.add_edge(user, mentionedUser, numberMentions=1)
    return g
trumpGraph = mentionGraph(trumpTag)
```

A **degree distribution** is the probability distribution of nodes' degrees over the whole network

```
def plot_degree_dist(G):
    degrees = [G.degree(n) for n in G.nodes()]
    counts, edges, bars = plt.hist(degrees,bins=30)
    plt.bar_label(bars)
    plt.xlabel("Degree")
    plt.ylabel("Frequency")
    plt.show()

# Fit to a power law distribution
fit_function = pwl.Fit(degree, xmin=1.1)
print("Fitted alpha:", fit_function.power_law.alpha)
print("Fitted start from x:",fit_function.power_law.xmin)
```

Lab 8: Knowledge Graphs

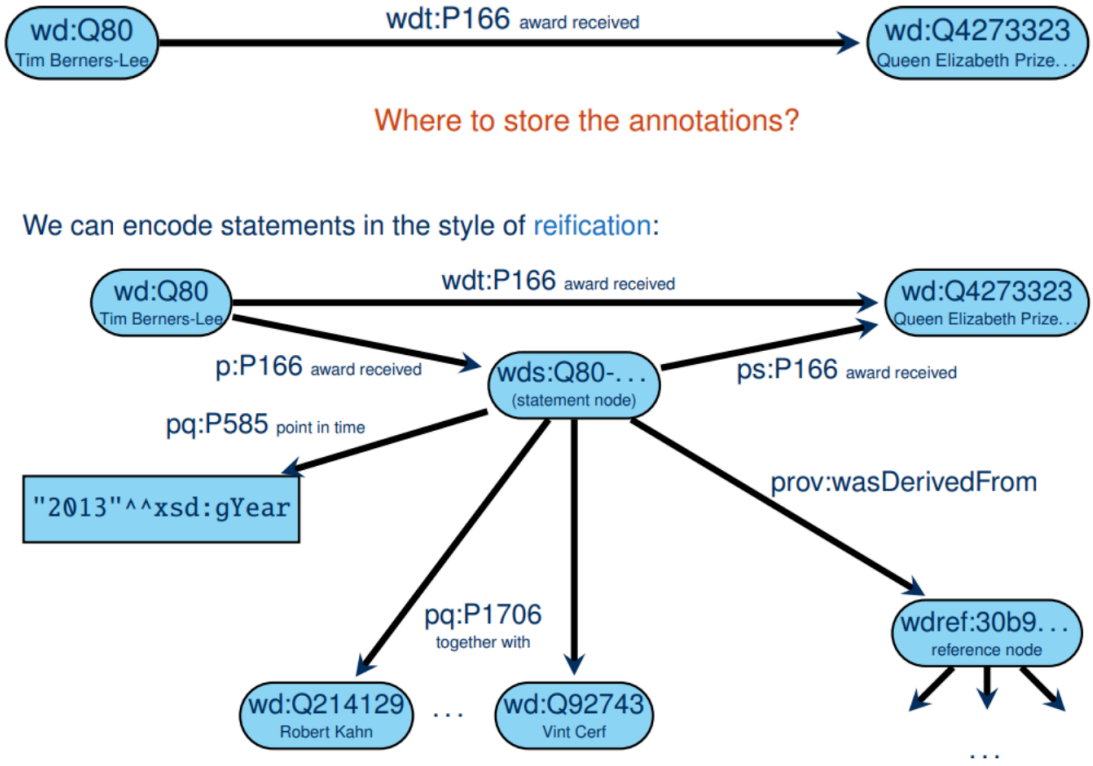
- Knowledge graphs: way of storing data that results from an information extraction task. Often implemented with triples (subject, predicate, object). Directional. Benefits:
    - Enhanced Question Answering (QA) Systems: Knowledge Graphs enable more accurate and context-aware answers in QA systems, improving user experience in search engines, virtual assistants, and customer support tools.
    - Improved Entity Resolution: By mapping complex relationships and various attributes of entities, Knowledge Graphs can be used in accurately identifying, linking, and disambiguating entities. This is crucial in domains like e-commerce, content management, and data integration.

```
# Construct a knowledge graph
G = nx.DiGraph()
G.add_edges_from([
    ('Tim', 'MIE223', {'label': 'enrolled'}), # order of the triplet different, but
concept holds.
    ('Joan', 'MIE223', {'label': 'enrolled'}),
    ('Scott', 'MIE223', {'label':'teaches'})
])
pos = nx.spring_layout(G)  # positions for all nodes
nx.draw(G, pos=pos, with_labels=True, node_color='lightblue', edge_color='gray',
font_size=10, font_weight='bold')
nx.draw_networkx_edge_labels(G, pos=pos, edge_labels={(u, v): d['label'] for u, v, d in
G.edges(data=True)})
plt.title("Knowledge Graph of MIE223")
plt.show()


# Get list of enrolled people
[subject for subject, predicate, value in G.in_edges('MIE223', data=True) if
value['label'] == 'enrolled']
```

WikiData

- WikiData: Wikipedia's knowledge graph, large graph (August 2019: >733M statements on >58M entities).
    - Identifiers: **QIDs**: Entities; **PIDs**: Relations.

- International, stable, convenient.



1. **Main Entities**

- `wd:Q80` **(Tim Berners-Lee)**: Represents a Wikidata entity for Tim Berners-Lee.
- `wd:Q4273323` **(Queen Elizabeth Prize)**: Represents the award received.
- `wds:Q80-...` **(statement node)**: Represents the detailed statement that Tim Berners-Lee received the award.

2. **Main Relationship**

- `wdt:P166` **(award received)**: Indicates that Tim Berners-Lee received the Queen Elizabeth Prize.
- This is a **simple** statement but lacks additional details like the year or co-recipients.

3. **Reification (Detailed Representation)**: Reification allows Wikidata (or any structured knowledge system) to store **contextual details** about a relationship. Instead of just storing a **subject-predicate-object** statement (`Tim Berners-Lee → received → Queen Elizabeth Prize`), a **statement node** (`wds:Q80-...`) is introduced (e.g. time-specific details, additional related entities, provenance). This allows adding more qualifiers:

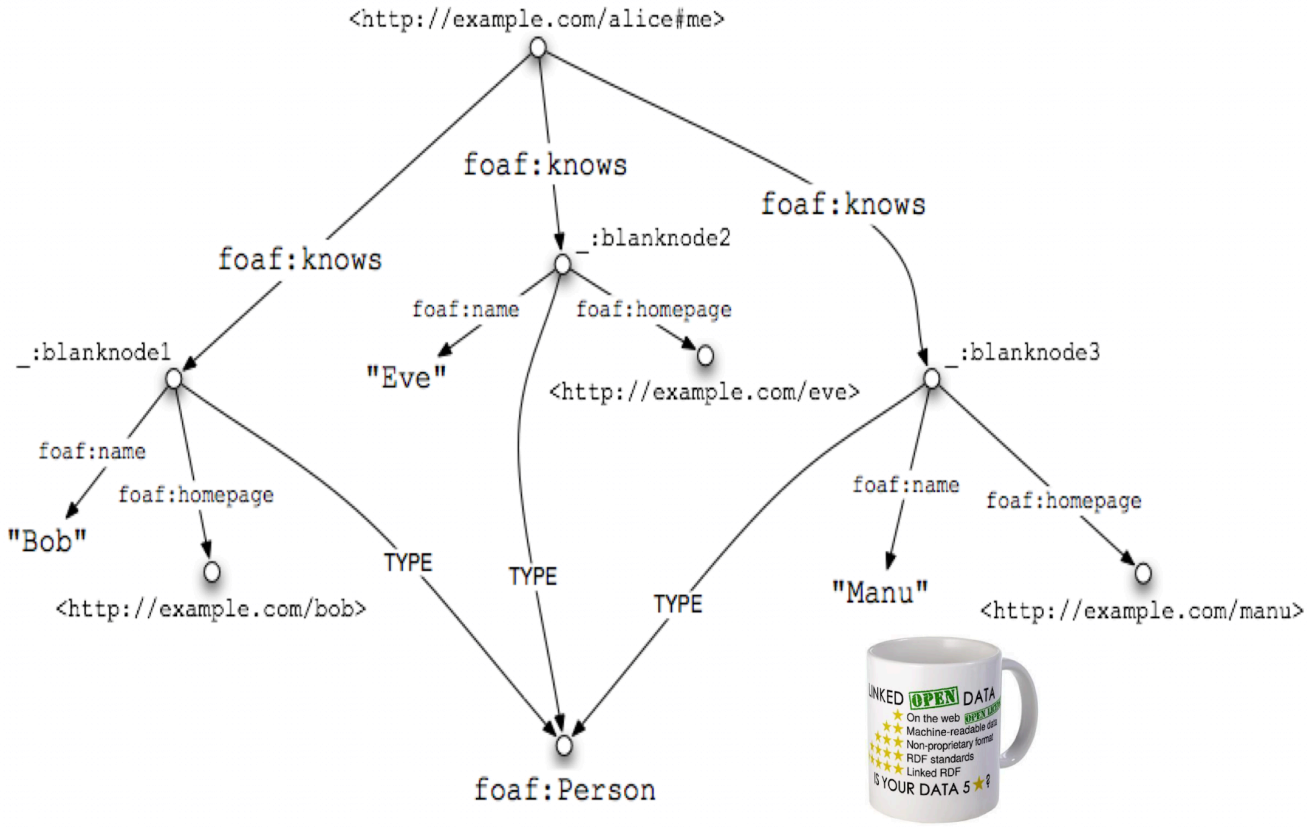
- `pq:P585` **(point in time)** → `"2013"^^xsd:gYear` (awarded in 2013).
- `pq:P1706` **(together with)** → `wd:Q214129 (Robert Kahn)`, `wd:Q92743 (Vint Cerf)` (other co-recipients).

4. **Provenance (Source Information)**: where does information come from?
prov:wasDerivedFrom → wdref:30b9... (reference node): Links to a reference providing proof for the statement.

In DBpedia, most widely used linking predicates are `owl:sameAs`, `rdfs:seeAlso`, `foaf:knows`:

`foaf:knows`	**Used in FOAF (Friend-of-a-Friend) to say person A knows person B**
`owl:sameAs`	Entity A is the same as Entity B (used to align across KGs/websites)
`rdfs:seeAlso`	There's additional related information elsewhere

RDFa: lightweight version of RDF.



With RDFa, we can make these statements in a web page.

SPARQL

- SPARQL: Query language for live queries on WikiData.

Setting up SPARQL

```
from qwikidata.entity import WikidataItem, WikidataLexeme, WikidataProperty
from qwikidata.linked_data_interface import get_entity_dict_from_api
import sys
import seaborn as sns
import folium # mapping
from folium.plugins import HeatMap
import missingno as msno
from SPARQLWrapper import SPARQLWrapper, JSON
import matplotlib.pyplot as plt
from typing import List, Dict
import json
import ast
```

```
class WikiDataQueryResults:
    """
    A class that can be used to query data from Wikidata using SPARQL and return the
results as a Pandas DataFrame or a list
    of values for a specific key.
    """
    def __init__(self, query: str):
        """
        Initializes the WikiDataQueryResults object with a SPARQL query string.
        :param query: A SPARQL query string.
        """
        self.user_agent = "WDQS-example Python/%s.%s" % (sys.version_info[0],
sys.version_info[1])
        self.endpoint_url = "https://query.wikidata.org/sparql"
        self.sparql = SPARQLWrapper(self.endpoint_url, agent=self.user_agent)
        self.sparql.setQuery(query)
        self.sparql.setReturnFormat(JSON)

    def __transform2dicts(self, results: List[Dict]) -> List[Dict]:
        """
        Helper function to transform SPARQL query results into a list of dictionaries.
        :param results: A list of query results returned by SPARQLWrapper.
        :return: A list of dictionaries, where each dictionary represents a result row and
has keys corresponding to the
        variables in the SPARQL SELECT clause.
        """
        new_results = []
        for result in results:
            print(result)
            break
            new_result = {}
            for key in result:
                new_result[key] = result[key]['value']
            new_results.append(new_result)
        return new_results

    # Function to remove invalid control characters
    def __remove_invalid_chars(text):
        return ''.join(char for char in text if ord(char) > 31 or char == '\n' or char ==
'\t')

    def _load(self) -> List[Dict]:
      """
      Helper function that loads the data from Wikidata using the SPARQLWrapper library,
      and transforms the results into a list of dictionaries.
      """
      results = self.sparql.queryAndConvert()['results']['bindings']
      return results  # Return directly as Python list


    def load_as_dataframe(self) -> pd.DataFrame:
      """
```

```
        Executes the SPARQL query and returns the results as a Pandas DataFrame.
        """
        results = self._load()
        df = pd.DataFrame(results)  # Convert to DataFrame directly

        # Ensure values are extracted properly
        df = df.applymap(lambda x: x['value'] if isinstance(x, dict) and 'value' in x else
x)

        return df
```

SPARQL Queries

```
SELECT distinct ?itemWeWantToRetrieve ?otherItemWeWantToRetrive
WHERE{
  scott teaches ?itemWeWantToRetrieve.
  ?itemWeWantToRetrieve isPartOf otherItemWeWantToRetrive.
}
limit ...
```

- Given this query, SPARQL will aim to find items that can complete the triplet and save it into variables (marked by the `?` marks).
- `distinct` means that we only want unique items. This can be removed if we don't care about only getting unique items.
- `limit` sets a limit on the number of tuples/items we retreive (this can also be removed if its not necessary).

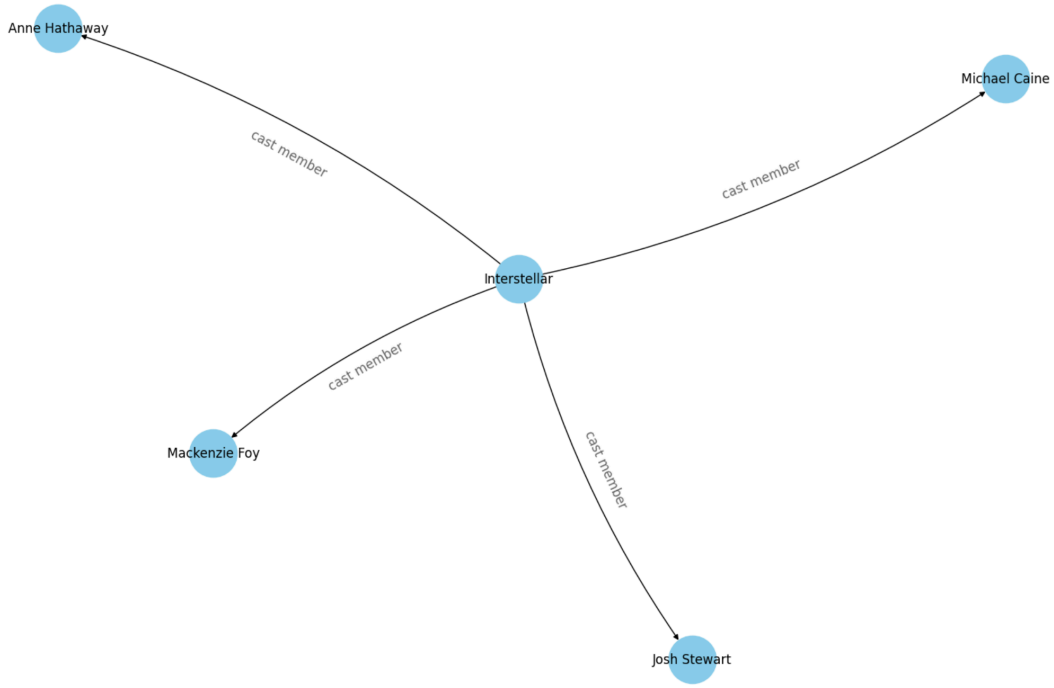
Queries and their interpretations

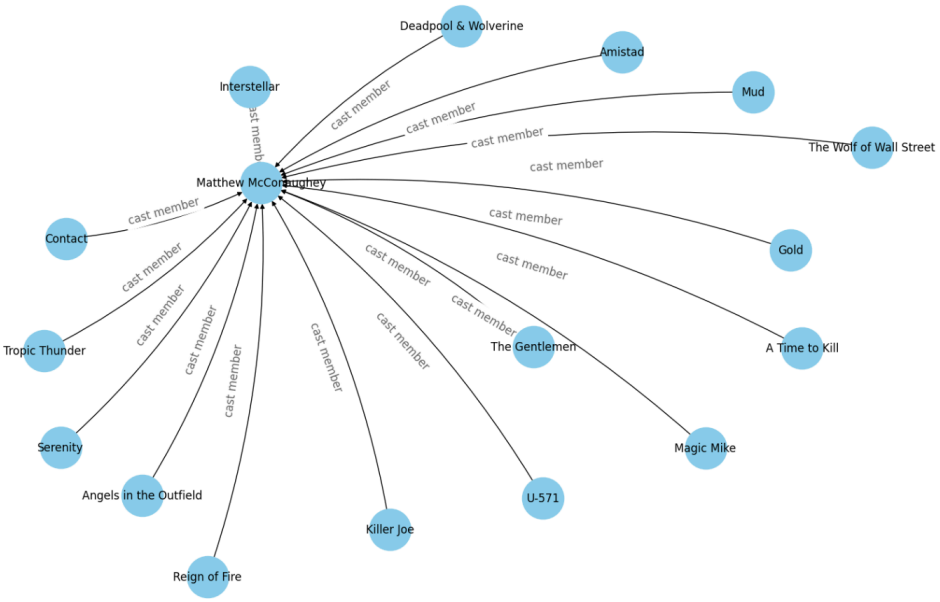
```
# Query 1: "List up to 100 unique actors (by ID and English name) who appear in the film
Interstellar, also retrieving the movie’s English title."

movieQuery = """
SELECT distinct ?actor ?actorLabel ?movieLabel
WHERE {
  # wd:Q13417189: [Entity] Interstellar (film) - https://www.wikidata.org/wiki/Q13417189
  # wdt:P161: [Relation] cast member https://www.wikidata.org/wiki/Property:P161

  wd:Q13417189 wdt:P161 ?actor.
  ?movie wdt:P161 ?actor

  # This line will map the actor (PIDs) to their labels actorLabel (actual names) in
English.
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
# We limit the number of returned results to 100
limit 100
"""
# This query will give us what you specified in after the select: ?actor and ?actorLabel
Interstellar_df = WikiDataQueryResults(movieQuery).load_as_dataframe()
```

```
# Query 2: "Find all movies that feature Matthew McConaughey, along with each movie’s
genre and box office revenue, and sort them from highest to lowest box office."

actorQuery = """
SELECT distinct ?movie ?movieLabel ?genreLabel ?boxOffice
WHERE {
  # Movie - cast member -> Matthew McConaughey
  ?movie wdt:P161 wd:Q188955.
  # Movie - box office -> ?boxOffice (a number)
  ?movie wdt:P2142 ?boxOffice.
  # Movie - genre -> ?genre
  ?movie wdt:P136 ?genre.

  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
# Order by descending box office
ORDER BY DESC(?boxOffice)
#limit 10 (this is a comment btw)
"""
Matthew_df = WikiDataQueryResults(actorQuery).load_as_dataframe()
# Matthew's movies
Matthew_movie_df = Matthew_df.drop_duplicates(subset=["movieLabel"])
Matthew_movie_df["actor"]="Matthew McConaughey"
Matthew_movie_df["relation"]="cast member"

df_to_KG(Matthew_movie_df, "movieLabel", "actor", "relation") # defined in next block

# Query 3: "List up to 10,000 films released after 2005 that earned more than $10 million,
along with their actors and genres."
hotmovieQuery = """
SELECT distinct ?movieLabel ?actorLabel ?genreLabel
WHERE {
  ?movie wdt:P31 wd:Q11424;        # Instance of a film wd:Q11424 is a film
         wdt:P136 ?genre;          # Genre - replace wd:Q130232. with the Wikidata ID for
drama or comedy
         wdt:P161 ?actor;          # Actor in the film
         wdt:P2142 ?boxOffice;     # Movie - box office -> ?boxOffice (a number)
         wdt:P577 ?releaseDate.     # Film's release date
  FILTER(YEAR(?releaseDate) > 2005) # Release date after 2005
  FILTER(?boxOffice > 10000000)     # Box office larger than 10M, so we can assume it's
more likely we've heard about the titles
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}

limit 10000
"""

# Multiple queries: iterate over list of QIDs, retrieve information.
def find_movie_info(movie_QIDs):
```

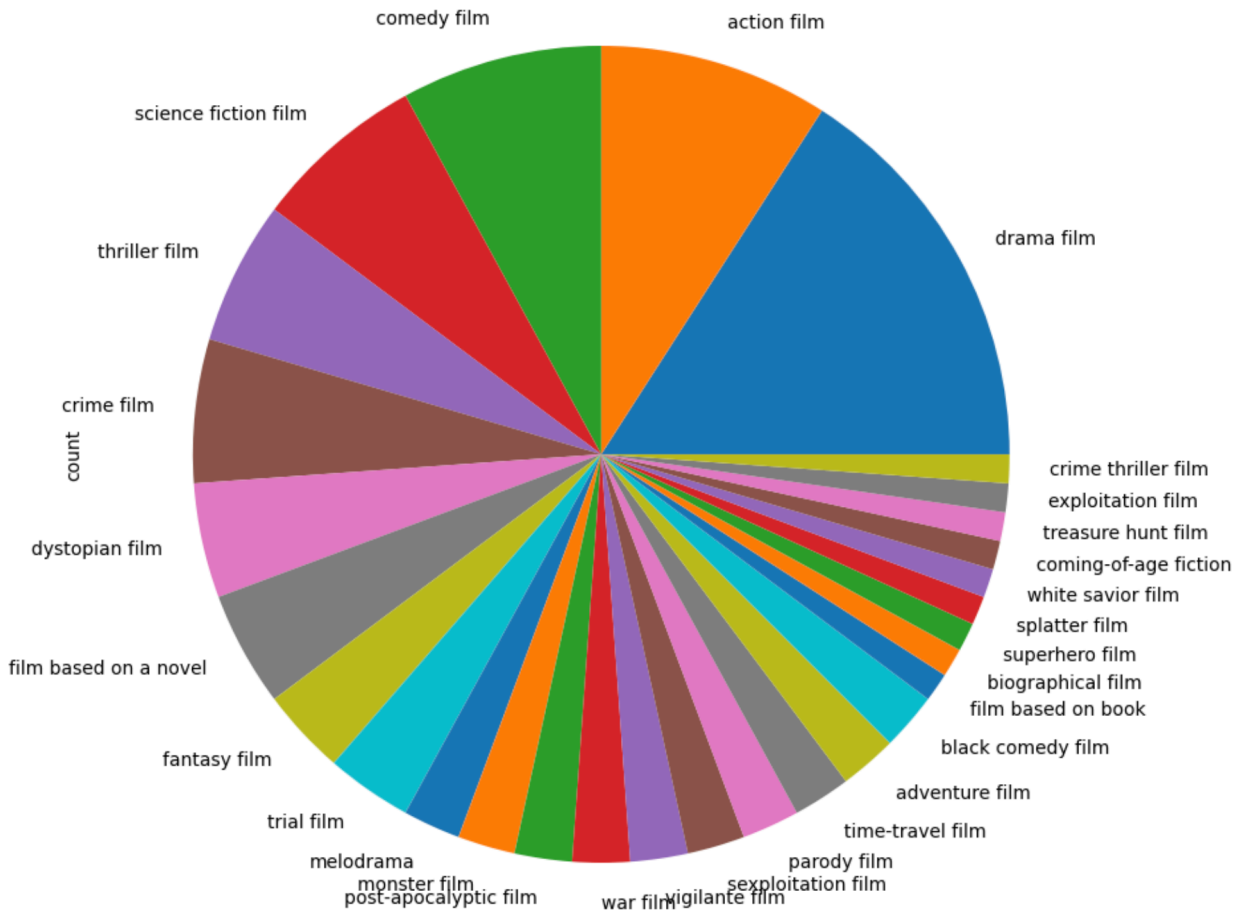
```
  """
  movie_QIDs is a list of QIDs [ "Qxxx", "Qyyyy", ...]
  """
  for movie_QID in movie_QIDs:
    # return all movies in the list with its actors and genres
    testQuery = f"""
    SELECT distinct ?actorLabel ?genreLabel
    WHERE {{
      wd:{movie_QID} wdt:P136 ?genre;      # Movie - genre -> ?genre
      wd:{movie_QID} wdt:P161 ?actor.      # Movie - cast member -> ?actor

      SERVICE wikibase:label {{ bd:serviceParam wikibase:language "en". }}
    }}
    """
    ret_df = WikiDataQueryResults(testQuery).load_as_dataframe()
    return ret_df
```

```
# Create a knowledge graph of the dataframe
def df_to_KG(df, source_col, target_col, relation_col):
  """
  Source - relation -> target
  """
  G = nx.from_pandas_edgelist(df, source=source_col, target=target_col,
                              edge_attr=relation_col,
                              create_using=nx.DiGraph())
  plt.figure(figsize=(15,10))
  pos = nx.spring_layout(G, k=1)
  node_color = "skyblue"
  edge_color = "black"
  nx.draw(G, pos=pos, with_labels=True, node_color=node_color,
          edge_color=edge_color, cmap=plt.cm.Dark2,
          node_size=2000, connectionstyle='arc3,rad=0.1')

  nx.draw_networkx_edge_labels(G, pos=pos, label_pos=0.5,
                           edge_labels=nx.get_edge_attributes(G,'relation'),
                           font_size=12, font_color='black', alpha=0.6)
  plt.show()

# We add the source node and relation back
Interstellar_df["movie"] = "Interstellar"
Interstellar_df["relation"] = "cast member"

# Plot the knowledge graph
df_to_KG(Interstellar_df, "movie", "actorLabel", "relation")
```






```
# Pie chart of movie genres Matthew starred in

# Counting each genre
genre_counts = Matthew_df['genreLabel'].value_counts()
# Normalizing the counts
Matthew_normalized_genre_vector = genre_counts / genre_counts.sum()
Matthew_normalized_genre_vector.plot(kind='pie' , figsize=(10,10))
```



Calculating cosine similarity between two actors by their genre of movie

```
def actor_cosine_similarity(vector1, vector2):
  # Combine the indices (genres) and remove duplicates
  all_genres = vector1.index.union(vector2.index)

  # Reindex both series to have the same genres
  actor1_series = vector1.reindex(all_genres, fill_value=0)
  actor2_series = vector2.reindex(all_genres, fill_value=0)

  # Calculate the cosine similarity. scipy.spatial.distance.cosine is cosine distance
  # cosine similarity is just (1 - cosine distance)
  pair_cosine_similarity = 1 - cosine(actor1_series, actor2_series)

  print(f'Cosine similarity: {pair_cosine_similarity:.2f}')

actor_cosine_similarity(Matthew_normalized_genre_vector,Jackie_normalized_genre_vector)
```

MovieLens

The datasets describe ratings and free-text tagging activities from MovieLens, a movie recommendation service.

Data processing

```
def strp_parenthesis(x):
    '''
    Function to clean up extra parenthesis at the beginning and end of the string.
    '''
    return (x.str.strip('()'))

# extract year
movies['year'] = movies['title'].str.extract('(\(\d{4}\))').apply(strp_parenthesis)

# converting year string to datetime 'year' values
movies['year'] = pd.to_datetime(movies['year'] , format='%Y')

# Movies that do not have Year information available.
movies['year'].isnull().sum()

# Dropping such records with no Year information on them
movies.dropna(subset=['year'] , inplace=True)

# Converting year column from datetime to Year
movies['year'] = movies['year'].dt.year

# Display the year range of the movies
print('min year:' , movies['year'].min() , ' |  max year:', movies['year'].max())

# Get average ratings
avg_ratings = ratings[['movieId' , 'rating']].groupby('movieId').mean().round(2)
movies = movies.merge(avg_ratings, how='inner', on='movieId')

# Get count of ratings
count_of_ratings = ratings[['movieId' , 'userId']].groupby('movieId').count()
count_of_ratings.rename(columns={'userId' : 'Count of Ratings'}, inplace=True)

# Joining count of ratings with movies names
# Top 25 most rated movies of all time
movies = movies.merge(count_of_ratings, on='movieId')

# Get most popular movie genres after 2005
from collections import Counter
# Fetch unique list of genres available in the data  - total available genres
unique_genre =
pd.DataFrame(movies_after_2005['genres'].str.split('|').to_list()).stack().unique()
# Number of movies per genre
genres_counts =
Counter(pd.DataFrame(movies_after_2005['genres'].str.split('|').to_list()).stack().to_list
())

# Convert it back to a dataframe, sort by counts
df = pd.DataFrame(genres_counts.items(), columns=
['genre','movie_counts']).sort_values('movie_counts', ascending=False)

# Bar plot of top movie genres after 2005
bar_genres = df.plot.bar(x='genre' , rot = 60, figsize = (16,4), grid = True,
```

```
                        colormap = 'tab10_r',
                        title='Movie Counts per Genre',
                        fontsize = 14,  )
```

MI Analysis for MovieLens

```
from sklearn.metrics import mutual_info_score

# Start by building a table of movie,actor,and genre from wikiData
hotmovieQuery = """
SELECT distinct ?movieLabel ?actorLabel ?genreLabel
WHERE {
  ?movie wdt:P31 wd:Q11424;         # Instance of a film wd:Q11424 is a film
         wdt:P136 ?genre;           # Genre - replace wd:Q130232. with the Wikidata ID for
drama or comedy
         wdt:P161 ?actor;           # Actor in the film
         wdt:P2142 ?boxOffice;      # Movie - box office -> ?boxOffice (a number)
         wdt:P577 ?releaseDate.     # Film's release date
  FILTER(YEAR(?releaseDate) > 2005) # Release date after 2005
  FILTER(?boxOffice > 10000000)     # Box office larger than 10M, so we can assume it's
more likely we've heard about the titles
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
limit 10000
"""

hotmovie_df = WikiDataQueryResults(hotmovieQuery).load_as_dataframe()
MI_df = hotmovie_df.copy()

# Create a binary encoding for Drama and Comedy for each actor
MI_df['is_drama'] = MI_df['genreLabel'].apply(lambda x: 1 if x == 'drama film' else 0)
MI_df['is_comedy'] = MI_df['genreLabel'].apply(lambda x: 1 if x == 'comedy film' else 0)

# Create a binary column for each actor per movie
actor_movie_matrix = pd.pivot_table(MI_df, index='movieLabel', columns='actorLabel',
aggfunc='size', fill_value=0)

# Aggregate these values at the movie level
genre_matrix = MI_df.groupby('movieLabel')[['is_drama', 'is_comedy']].max()

# Merge the actor and genre matrices
actor_movie_matrix = actor_movie_matrix.merge(genre_matrix, left_index=True,
right_index=True)

mi_scores = {}

# Iterate over actor columns to calculate MI with 'is_drama' and 'is_comedy'
for actor in actor_movie_matrix.columns:
    if actor != 'is_drama' and actor != 'is_comedy':
        drama_mi = mutual_info_score(actor_movie_matrix[actor],
actor_movie_matrix['is_drama'])
```

```
        comedy_mi = mutual_info_score(actor_movie_matrix[actor],
actor_movie_matrix['is_comedy'])
        mi_scores[actor] = {'Drama_MI_Score': drama_mi, 'Comedy_MI_Score': comedy_mi}

# Convert MI scores to a DataFrame
mi_df = pd.DataFrame.from_dict(mi_scores, orient='index')

# Sort actors by MI score in descending order, show their relation to Drama
mi_df.sort_values(by='Drama_MI_Score', ascending=False)[["Drama_MI_Score"]].head(10)

# Sort actors by MI score in descending order, show their relation to Comedy
mi_df.sort_values(by='Comedy_MI_Score', ascending=False)[["Comedy_MI_Score"]].head(10)
```

	Drama_MI_Score	Comedy_MI_Score
Aamir Khan	0.002940	0.005494
Aaron Abrams	0.002940	0.001097
Aaron Ashmore	0.002940	0.001097
Aaron Douglas	0.002483	0.001097
Aaron Himelstein	0.002483	0.001097
...	...	...
Óscar Jaenada	0.007889	0.003305
Örvendi Cintia	0.002940	0.001097
Şafak Pekdemir	0.002940	0.001097
Željko Ivanek	0.002940	0.005494
ענבל ורוני	0.002483	0.005494

2875 rows × 2 columns

Similar movies

Similar according to tags

Measuring how similar movies are based on overlapping tags.

```
# We want movies that have enough ratings for us to analyze
# Image below after this line
movies_after_2005_5000ratings = movies_after_2005[ (movies_after_2005['Count of
Ratings']>5000)]

# Sync tags with movies_after_2005_5000ratings
tags_after_2005_5000ratings =
tags[tags['movieId'].isin(movies_after_2005_5000ratings['movieId'])]
```

	movieId	title	genres	year	rating	Count of Ratings
10677	44191	V for Vendetta (2006)	Action|Sci-Fi|Thriller|IMAX	2006	3.91	25990
10679	44195	Thank You for Smoking (2006)	Comedy|Drama	2006	3.82	10402
10681	44199	Inside Man (2006)	Crime|Drama|Thriller	2006	3.89	8551
10700	44555	Lives of Others, The (Das leben der Anderen) (...	Drama|Romance|Thriller	2006	4.20	9177
10718	44665	Lucky Number Slevin (2006)	Crime|Drama|Mystery	2006	3.85	8994
...	...	...	...	...	...	...
39918	164179	Arrival (2016)	Sci-Fi	2016	4.03	10087
40872	166528	Rogue One: A Star Wars Story (2016)	Action|Adventure|Fantasy|Sci-Fi	2016	3.78	6850
41599	168250	Get Out (2017)	Horror	2017	3.96	5010
41600	168252	Logan (2017)	Action|Sci-Fi	2017	3.89	6739
45236	176371	Blade Runner 2049 (2017)	Sci-Fi	2017	3.91	5119

215 rows × 6 columns

```
# Simply count the number of overlapping tags
# Group by 'movieId' and aggregate the tags into a list
tags_grouped = tags_after_2005_5000ratings.groupby('movieId')
['tag'].apply(list).reset_index()

# Initialize an empty DataFrame for similarity scores
similarity_matrix = pd.DataFrame(index=tags_grouped['movieId'],
columns=tags_grouped['movieId'])

# Fill the matrix with similarity scores
for i in tags_grouped['movieId']:
    for j in tags_grouped['movieId']:
        if i != j:
            # Intersection of tags
            similarity_matrix.at[i, j] = len(set(tags_grouped[tags_grouped['movieId'] ==
i]['tag'].values[0]) & set(tags_grouped[tags_grouped['movieId'] == j]['tag'].values[0]))
```

The similarity matrix based on number of overlapping tags

movieId	44191	44195	44199	44555	44665	45186	45447	45499	45517	45672	...	134853	139385	142488	148626
movieId															
44191	NaN	15	4	17	6	8	5	8	5	2	...	2	6	2	6
44195	15	NaN	4	4	1	3	7	4	6	2	...	4	4	3	12
44199	4	4	NaN	3	4	4	0	6	2	2	...	2	1	3	2
44555	17	4	3	NaN	5	2	3	0	3	1	...	4	1	0	4
44665	6	1	4	5	NaN	2	1	1	0	1	...	0	3	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
164179	5	2	4	7	3	1	3	1	0	6	...	9	7	3	4
166528	8	1	1	2	0	4	2	7	2	0	...	8	2	1	2
168250	3	8	4	3	6	2	3	1	3	3	...	8	8	4	5
168252	4	0	1	2	1	1	1	10	3	3	...	4	2	0	2
176371	13	4	1	5	5	1	2	2	2	1	...	5	9	2	2

215 rows × 215 columns

Get the 50 most similar movies

```
N=50

# Convert the matrix into a stack of pairs of movies
```

```
similarity_pairs = similarity_matrix.stack()

# Sort the pairs by similarity score
sorted_pairs = similarity_pairs.sort_values(ascending=False)

# Get the top N pairs
top_similar_movies = sorted_pairs.head(N)

# Map back to titles
movie_titles = dict(zip(movies_after_2005_5000ratings['movieId'],
movies_after_2005_5000ratings['title']))
top_similar_movies.index = [(movie_titles[i], movie_titles[j]) for i, j in
top_similar_movies.index]
```

	0
(Dark Knight Rises, The (2012), Dark Knight, The (2008))	44
(Dark Knight, The (2008), Dark Knight Rises, The (2012))	44
(Hobbit: The Desolation of Smaug, The (2013), Hobbit: An Unexpected Journey, The (2012))	41
(Hobbit: An Unexpected Journey, The (2012), Hobbit: The Desolation of Smaug, The (2013))	41
(Avengers: Age of Ultron (2015), Avengers, The (2012))	40
(Avengers, The (2012), Avengers: Age of Ultron (2015))	40
(Interstellar (2014), Inception (2010))	39
(Harry Potter and the Half-Blood Prince (2009), Harry Potter and the Order of the Phoenix (2007))	39
(Harry Potter and the Order of the Phoenix (2007), Harry Potter and the Half-Blood Prince (2009))	39
(Inception (2010), Interstellar (2014))	39
(Iron Man 2 (2010), Iron Man (2008))	38
(Iron Man (2008), Iron Man 2 (2010))	38
(Dark Knight, The (2008), Inception (2010))	38

Similar according to user preferences

This approach computes the similarity by determining the set intersection cardinality, which refers to the number of common elements, between item columns in user-item matrices, utilizing tools like NumPy for efficient computation.

```
# Next, we grab the ratings
# We do not need timestamp
ratings_mat = ratings.drop(columns=['timestamp']).copy()

# We only care about movieIds in movies_after_2005_1000ratings
ratings_mat = ratings_mat[
ratings_mat['movieId'].isin(movies_after_2005_5000ratings["movieId"]) ]

# And we think rating >= 4 means the user really like the movie
ratings_mat['user_like'] = (ratings_mat['rating'] >= 4).astype(int)
```

	userId	movieId	rating	user_like
586	3	44191	4.0	1
587	3	45186	3.5	0
588	3	45447	3.5	0
589	3	45499	4.0	1
591	3	45517	4.0	1
...	...	...	...	...
24999909	162540	68954	5.0	1
24999912	162540	69844	4.5	1
25000089	162541	45517	4.5	1
25000090	162541	50872	4.5	1
25000093	162541	58559	4.0	1

2237641 rows × 4 columns

```
# We now create the user rating matrix, where rows are users and cols are movies
ratings_mat = ratings_mat.pivot(index='userId', columns='movieId', values='user_like')
ratings_mat.fillna(0, inplace=True)
```

movieId	44191	44195	44199	44555	44665	45186	45447	45499	45517	45672	...	134853	139385	142488	148626
userId															
3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	...	1.0	1.0	0.0	1.0
4	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0	1.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
162534	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0
162536	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	1.0
162538	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
162540	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0
162541	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0

79149 rows × 215 columns

```
# Rows: Users, Columns: Movies, Cell value: 1 if liked, 0 otherwise
# Convert to np for faster computation
ratings_array = ratings_mat.values

# Compute the dot product of the matrix with its transpose
# This gives a matrix where each (i, j) element is the count of users who liked both movie
i and j
similarity_matrix = np.matmul(ratings_array.T, ratings_array)

# Number of movies
num_movies = similarity_matrix.shape[0]

# Extract movie pairs and their overlap counts
```

```
movie_pairs = []
for i in range(num_movies):
    for j in range(i + 1, num_movies):
        movie_pairs.append(((ratings_mat.columns[i], ratings_mat.columns[j]),
similarity_matrix[i, j]))

# Sort movie pairs by overlap count in descending order
top_movie_pairs = sorted(movie_pairs, key=lambda x: x[1], reverse=True)[:N]

# Create a dictionary for movie ID to title mapping
movie_titles = pd.Series(movies_after_2005_5000ratings.title.values,
index=movies_after_2005_5000ratings.movieId).to_dict()

# Replace movie IDs with titles in the top 20 pairs
top_movie_pairs_with_titles = [((movie_titles[pair[0][0]], movie_titles[pair[0][1]]),
pair[1]) for pair in top_movie_pairs]
```

```
[(('Dark Knight, The (2008)', 'Inception (2010)'), 18789.0),
 (('Inception (2010)', 'Interstellar (2014)'), 12370.0),
 (('Departed, The (2006)', 'Dark Knight, The (2008)'), 12327.0),
 (('Dark Knight, The (2008)', 'WALL·E (2008)'), 12069.0),
 (('Dark Knight, The (2008)', 'Dark Knight Rises, The (2012)'), 11817.0),
 (('Dark Knight, The (2008)', 'Iron Man (2008)'), 11651.0),
 (('Prestige, The (2006)', 'Dark Knight, The (2008)'), 11536.0),
 (('WALL·E (2008)', 'Inception (2010)'), 11129.0),
 (('Prestige, The (2006)', 'Inception (2010)'), 11078.0),
 (('Dark Knight, The (2008)', 'Inglourious Basterds (2009)'), 11069.0),
 (('Inglourious Basterds (2009)', 'Inception (2010)'), 10724.0),
 (('Departed, The (2006)', 'Inception (2010)'), 10586.0),
```

Using a dot product for rating-based similarities can introduce biases, specifically **popularity** and **temporal** bias.

Popularity bias: users rate mostly what is popular

Temporal bias: users only provide ratings in a short temporal range. We're always going to get the temporal bias, but normalizing for vector length (=popularity) should improve the results. To reduce these biases, methods that explicitly incorporate normalization, such as Jaccard and Cosine Similarity, can be effective.

```
from sklearn.metrics.pairwise import cosine_similarity

N = 50

# Rows: Users, Columns: Movies, Cell value: 1 if liked, 0 otherwise
# Convert to np array for faster computation
ratings_array = ratings_mat.values

# Compute cosine similarity matrix
similarity_matrix = cosine_similarity(ratings_array.T)

# Number of movies
num_movies = similarity_matrix.shape[0]

# Extract movie pairs and their similarity scores
movie_pairs = []
```

```
for i in range(num_movies):
    for j in range(i + 1, num_movies):
        movie_pairs.append(((ratings_mat.columns[i], ratings_mat.columns[j]),
similarity_matrix[i, j]))

# Sort movie pairs by similarity score in descending order
top_movie_pairs = sorted(movie_pairs, key=lambda x: x[1], reverse=True)[:N]

# Create a dictionary for movie ID to title mapping
movie_titles = pd.Series(movies_after_2005_5000ratings.title.values,
index=movies_after_2005_5000ratings.movieId).to_dict()

# Replace movie IDs with titles in the top N pairs
top_movie_pairs_with_titles = [((movie_titles[pair[0][0]], movie_titles[pair[0][1]]),
pair[1]) for pair in top_movie_pairs]
```

```
[(('Harry Potter and the Deathly Hallows: Part 1 (2010)',
   'Harry Potter and the Deathly Hallows: Part 2 (2011)'),
  0.7372551410118866),
 (('Harry Potter and the Half-Blood Prince (2009)',
   'Harry Potter and the Deathly Hallows: Part 1 (2010)'),
  0.7358257408011523),
 (('Harry Potter and the Order of the Phoenix (2007)',
   'Harry Potter and the Half-Blood Prince (2009)'),
  0.6847767746526652),
 (('Harry Potter and the Half-Blood Prince (2009)',
   'Harry Potter and the Deathly Hallows: Part 2 (2011)'),
  0.673121750382305),
 (('Hobbit: An Unexpected Journey, The (2012)',
   'Hobbit: The Desolation of Smaug, The (2013)'),
  0.6473379421184209),
 (('Harry Potter and the Order of the Phoenix (2007)',
   'Harry Potter and the Deathly Hallows: Part 1 (2010)'),
  0.6244028835537182),
 (('Avengers: Infinity War - Part I (2018)', 'Thor: Ragnarok (2017)'),
  0.6228104734123671),
 (('Harry Potter and the Order of the Phoenix (2007)',
   'Harry Potter and the Deathly Hallows: Part 2 (2011)'),
  0.6102875135559169),
 (('Dark Knight, The (2008)', 'Inception (2010)'), 0.6082499354128318),
 (('Hobbit: The Desolation of Smaug, The (2013)',
   'The Hobbit: The Battle of the Five Armies (2014)'),
  0.600433783763524),
 (('The Hunger Games (2012)', 'The Hunger Games: Catching Fire (2013)'),
  0.5867973183633142),
 (("Pirates of the Caribbean: Dead Man's Chest (2006)",
   "Pirates of the Caribbean: At World's End (2007)"),
  0.5748223247275084),
```

Lab 9: Spatial Data

Vector data

It describes the features of geographic locations on Earth through the use of discrete geometries (primitives).

Some of the most common vector data formats are:

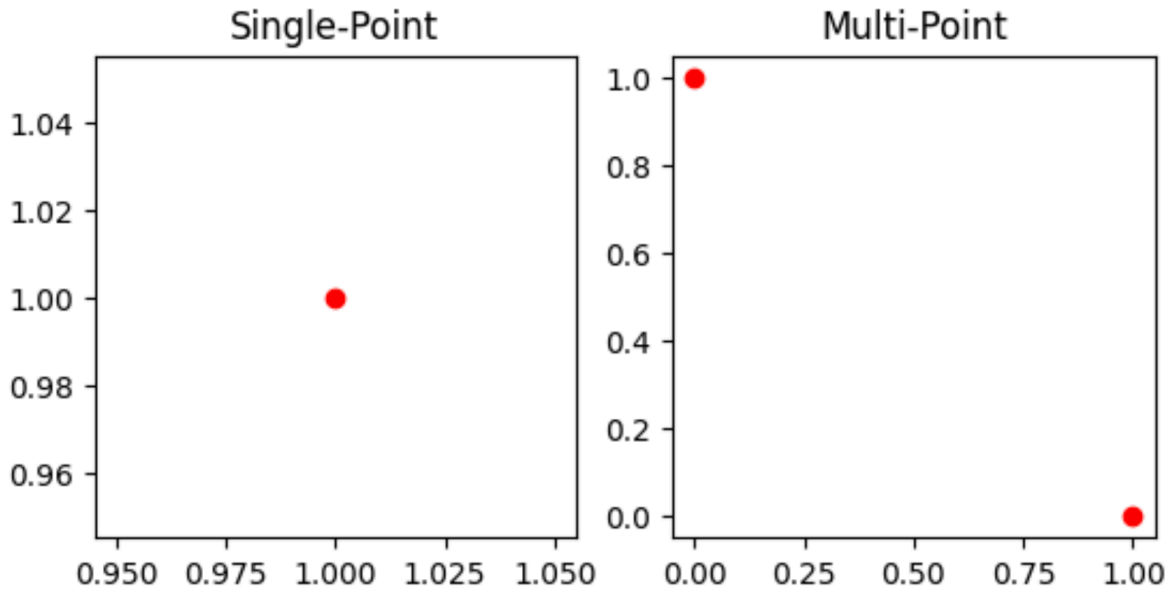
- Shapefile. file-based data format that stores a set of features sharing a common geometry type (point, line, or polygon), possessing the same attributes, and occupying a shared spatial extent. As the industry standard, shapefiles are the most common vector data format. It (usually) comprises three files that are usually provided in a zip file:
  - The .**shp** file contains shape geometry. Actual vector shapes.
  - The **.dbf** file holds attributes for each geometry, e.g. name, types.
  - The **.shx** file or shape index file helps link the attributes to the shapes.
- GeoJSON. It's a newer format for geospatial data released in 2016. Unlike shapefiles, GeoJSON is a single file, making it easier to work with.

Vector shapes: can be decomposed into 3 different geometric primitives. We can use multiple representations of some feature.

1. Point: one coordinate pair representing a specific location in a coordinate. Individual locations.

```
# Given the x and y coordinates, output the 'geometry' column
geometry = gpd.points_from_xy(x=[1], y=[1])
# Initialize the GeoDataFrame
gdf_from_xy = gpd.GeoDataFrame(geometry=geometry)

gdf_from_shapely = gpd.GeoDataFrame(geometry=[Point(1, 0), Point(0, 1)])
system.gdf_from_shapely = gpd.GeoDataFrame(geometry=[Point(1, 0), Point(0, 1)])

print(f'The X coordinate for first point is {gdf_from_shapely.geometry.x[0]}')    # 1.0
print('The Y coordinate for points are', gdf_from_shapely['geometry'].y.tolist()) # [0.0,
1.0]

# Point visualization
plt.subplot(121)
gdf_from_xy.plot(ax=plt.gca(), color='r')
plt.title('Single-Point')
plt.subplot(122)
gdf_from_shapely.plot(ax=plt.gca(), color='r')
plt.title('Multi-Point')
plt.show()
```

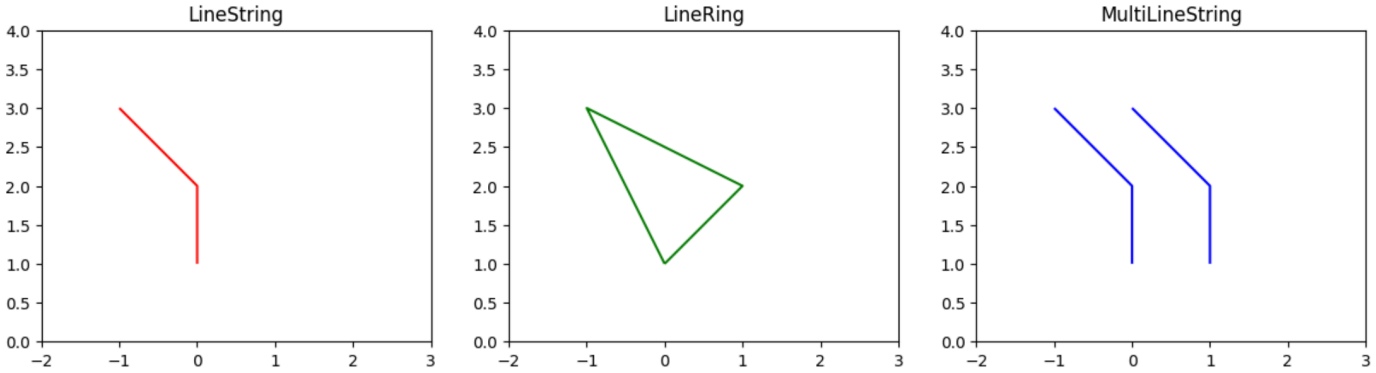
	geometry
0	POINT (1 0)
1	POINT (0 1)



2. LineString: sequence of vertices. Can find distances and intersections between geometries. A series of connected points describing things like roads or streams. Polyline: one or more strings.

```
from shapely.geometry import LineString,MultiLineString

# Create a line out of connecting 3 points
LineString([Point(0, 1), # start point
      Point(0,2), # intermediate point (can be more than one points)
      Point(-1,3)]) # end point

# Create a line out of a list of points
points_to_line = [Point(53,65), Point(50, 70), Point(46,73), Point(45,79), Point(40,79),
Point(35,85)]
line_from_points = LineString(points_to_line)

line_ring = LineString([Point(0, 1), Point(1,2), Point(-1,3), Point(0, 1)]) # line in a
cricle

# Multilines in one data structure (can use tuple instead of Point)
multi_line = MultiLineString([[(0, 1), (0,2), (-1,3)], # first line
                              [(1, 1), (1,2), (0,3)]]) # second line
# Visual
plt.figure(figsize=(15, 5))
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
gpd.GeoSeries([line]).plot(ax=ax[0], color='r')
ax[0].set_title('LineString')
gpd.GeoSeries([line_ring]).plot(ax=ax[1], color='g')
ax[1].set_title('LineRing')
gpd.GeoSeries([multi_line]).plot(ax=ax[2], color='b')
ax[2].set_title('MultiLineString')
plt.setp(ax, xlim=(-2, 3), ylim=(0, 4))
plt.show()
```

```
<Figure size 1500x500 with 0 Axes>
```




```
# Print line attributes

# Length
print('Length: ', line.length)          # Length:  2.414213562373095
# Minimum bounding region (minx, miny, maxx, maxy)
print('Bound: ', line.bounds)           # Bound:  (-1.0, 1.0, 0.0, 3.0)
# Extracting the coordinates of points in the line
print('Coordinate of points within the line: ',
      [(x,y) for x,y in zip(line.coords.xy[0],line.coords.xy[1])] )
      # Coordinate of points within the line:  [(0.0, 1.0), (0.0, 2.0), (-1.0, 3.0)]

print('Is a \'LineRing\' intersecting with the \'LineString\': ',
line.intersects(line_ring)) # True
print('Is the Point(0,0) within the \'LineString\': ', line.contains(Point(0, 0)))
# False
```

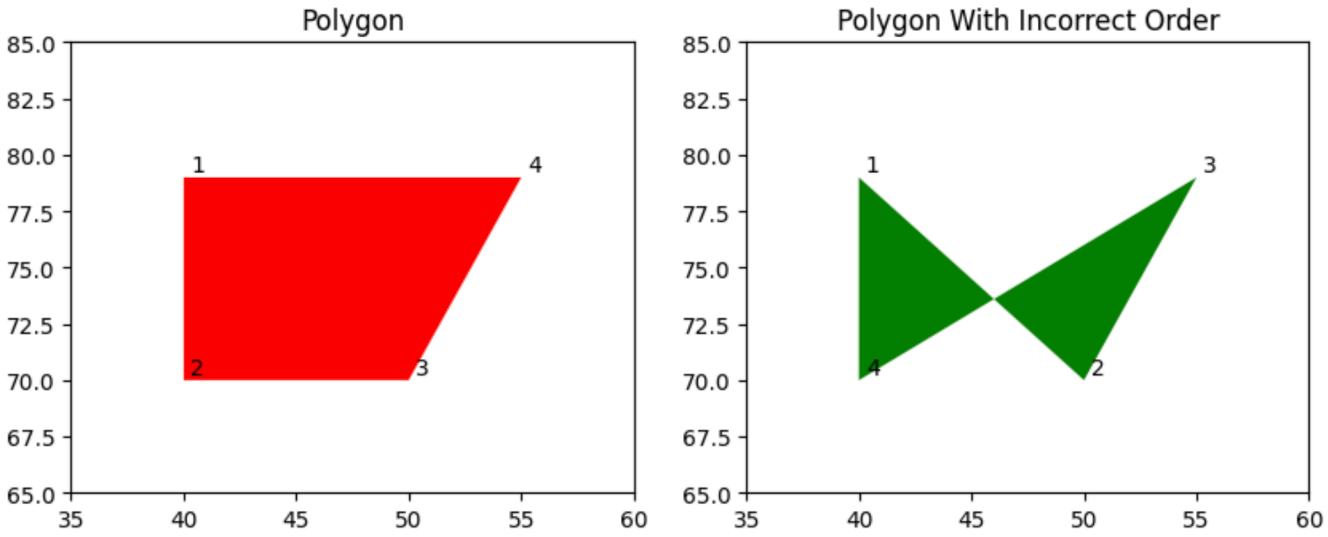
3. Polygons: area defined by three or more line segments, each with a starting and ending pair of coordinates that match. Polyine that starts and ends at the same point. Polygons can have holes. Formed by a closed line that encircles an area, such as the boundaries of a country. Additionally, when one feature consists of multiple geometries, we call it **MultiPolygon**.

```
from shapely.geometry import Polygon, MultiPolygon
# the points in polygon must be in the correct order
polygon = Polygon([Point(40,79), Point(40, 70),Point(50,70), Point(55,79)])

print('Area: ', polygon.area)
print('Bound: ', polygon.bounds)
print('Centroid: ', polygon.centroid)
print('Exterior: ', polygon.exterior)
print('Points that form the Polygon: ', [(x,y) for x,y in
zip(polygon.exterior.xy[0],polygon.exterior.xy[1])])

print('Is a Polygon intersecting with the LineString: ', polygon.intersects(line1)) #
check if polygon intersects with the line
print('Is the Point(40, 75) within the Polygon: ', polygon.contains(Point(40, 75))) #
check if point is within the polygon
```

Area: 112.5
Bound: (40.0, 70.0, 55.0, 79.0)
Centroid: POINT (46.333333333333336 74.8)
Exterior: LINEARRING (40 79, 40 70, 50 70, 55 79, 40 79)
Points that form the Polygon: [(40.0, 79.0), (40.0, 70.0), (50.0, 70.0), (55.0, 79.0), (40.0, 79.0)]
Is a Polygon intersecting with the LineString: True
Is the Point(40, 75) within the Polygon: False

```
# Similar to the LineString, we can create a polygon using a list of tuples.
polygon_failed = Polygon([(40,79), (50,70), (55,79), (40, 70)]) # order matters!

print('Area: ', polygon_failed.area)    # 0.0
print('Bound: ', polygon_failed.bounds) # (40.0, 70.0, 55.0, 79.0)
```

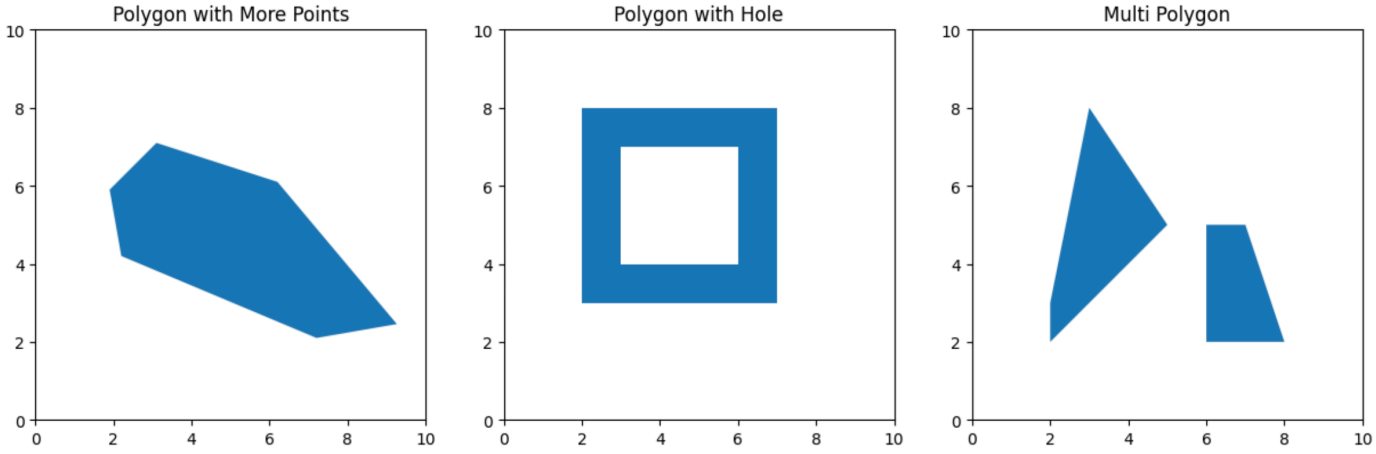
```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))

# Plot polygons and annotate vertices for the first subplot
for data, poly in [(ax[0], polygon), (ax[1], polygon_failed)]:
    gpd.GeoSeries([poly]).plot(ax=data, color='r' if data == ax[0] else 'g')
    data.set_title('Polygon' if data == ax[0] else 'Polygon With Incorrect Order')
    # Annotate vertices
    for x,y,label in zip(poly.exterior.coords.xy[0], poly.exterior.coords.xy[1], ['1',
'2', '3', '4']):
        data.annotate(label, xy=(x, y), xytext=(3, 3), textcoords="offset points")

plt.setp(ax, xlim=(35, 60), ylim=(65, 85))
plt.show()
```



Polygon with holes

```
# Polygon with more points
polygon_complex = Polygon([(2.2, 4.2), (7.2, 2.1), (9.26, 2.456),(6.2, 6.1),(3.1, 7.1),
(1.9, 5.9)])
```

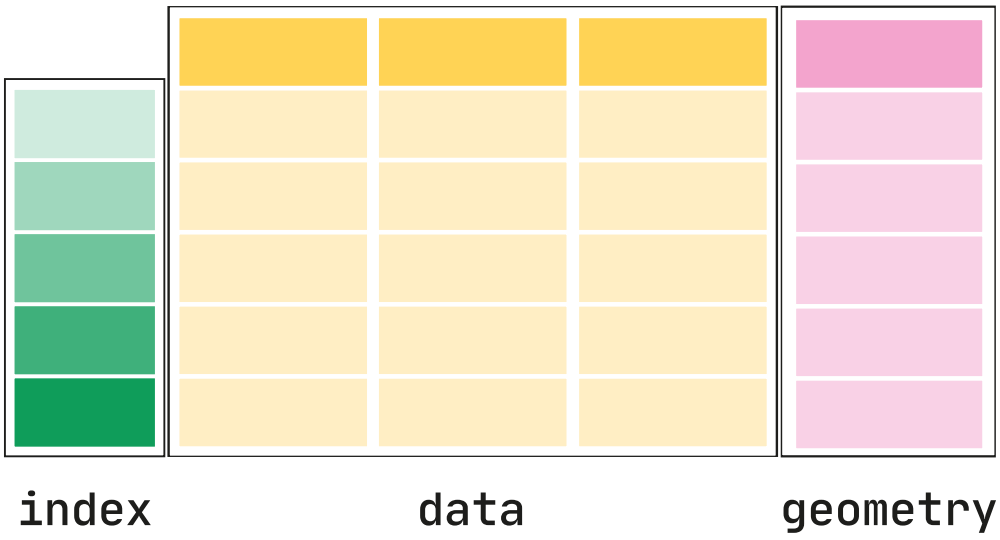
```
# Create a polygon with a hole
exterior = [(2, 8), (7, 8), (7, 3), (2, 3)]
interior = [(3, 7), (3, 4), (6, 4), (6, 7)]
polygon_with_hole = Polygon(shell=exterior, holes=[interior])

# Multi-polygons in one data structure
ploygon1 = Polygon([(2, 3), (3, 8), (5, 5), (2, 2)])
ploygon2 = Polygon([(7, 5), (6, 5), (6, 2), (8, 2)])
multi_polygon = MultiPolygon([ploygon1, ploygon2])

# Visualize
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
gpd.GeoSeries([polygon_complex]).plot(ax=ax[0])
ax[0].set_title('Polygon with More Points')
gpd.GeoSeries([polygon_with_hole]).plot(ax=ax[1])
ax[1].set_title('Polygon with Hole')
gpd.GeoSeries([multi_polygon]).plot(ax=ax[2])
ax[2].set_title('Multi Polygon')
plt.setp(ax, xlim=(0, 10), ylim=(0, 10))
plt.show()
```



Raster data

Encodes the world as a continuous surface represented by a grid, such as the pixels of an image. Each piece of the grid can be either a continuous value (such as an elevation value) or a categorical classification (such as land cover classifications). Classic examples include altitude data or satellite images.

GeoPandas



A GeoDataFrame is a subclass of pandas.DataFrame. Each column in geopandas.GeoDataFrame is a geopandas.GeoSeries. Every dataset (raster or vector) has attributes.

> ⚠ **Warning**
>
> Although a GeoDataFrame can have multiple GeoSeries, only one column is considered the active geometry, meaning that all the spatial operations will be based on that column.

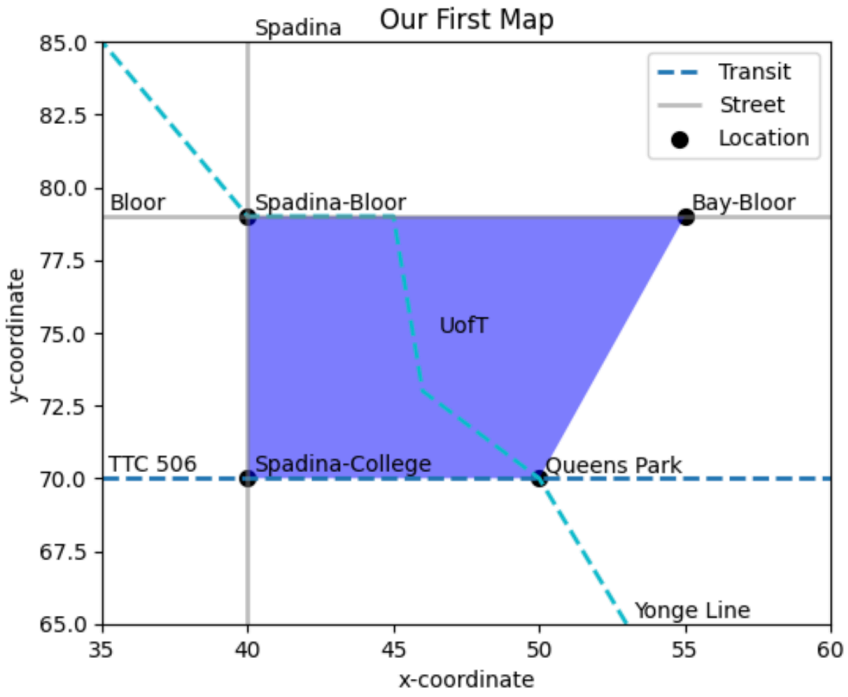
```
data = {'name': ['Spadina-Bloor', 'Spadina-College','Bay-Bloor','Queen Park'], # column 1
          'geometry': [Point(40,79), Point(40, 70), Point(55,79),Point(50,70)]} # shape
# Create a GeoDataFrame from the data
gdf_city = gpd.GeoDataFrame(data)

# Create a GeoDataFrame from lists
# Include both lines and points in a single GeoDataFrame.
point = [Point(40,79), Point(40, 70), Point(55,79), Point(50,70)] # four locations we had
before
point_name = ['Spadina-Bloor', 'Spadina-College','Bay-Bloor','Queens Park']
gdf_line = gpd.GeoDataFrame({'name': ['Spadina', 'Bloor', 'TTC 506', 'Yonge Line'] +
point_name,
   # ^combine two lists of shapes name
   'properties':
['Street','Street','Transit','Transit','Location','Location','Location','Location'],
   # ^type of the shape
   'geometry':[line1, line2, line3, line4] + point }) # combine two lists of shapes
```

```
# Check the type of the geometry in the GeoDataFrame
gdf_line.type # or gdf_line.geom_type

# Check the type of the geometry in the GeoDataFrame
gdf_map.type

# Plot the GeoDataFrame geometry
gdf_city.plot()
# add a label to each point
for x, y, label in zip(gdf_city.geometry.x, gdf_city.geometry.y, gdf_city.name):
```

```
    plt.gca().annotate(label, xy=(x, y), xytext=(3, 3),
    textcoords="offset points")
plt.ylim(65, 85)
plt.xlim(35, 60)
plt.xlabel('x-coordinate')
plt.ylabel('y-coordinate')
plt.show()
```

Building a map

```
def plot_from_gdf(gdf) -> None:
    for prop, color, linestyle, alpha in [('Transit', 'tab10', '--', 1), ('Street',
'grey', '-', 0.5),
                                          ('Location', 'black', '-', 1), ('Area', 'blue',
'-', 0.5)]:
        try:
            gdf[gdf['properties'] == prop].plot(color=color, lw=2, ls=linestyle,
alpha=alpha, ax=plt.gca())
        except:
            gdf[gdf['properties'] == prop].plot(cmap=color, lw=2, ls=linestyle,
alpha=alpha, ax=plt.gca())

    switch = {
        Point: lambda s: (s.x, s.y),
        LineString: lambda s: (s.xy[0][0], s.xy[1][0]),
        Polygon: lambda s: (s.centroid.x, s.centroid.y) # if the shape is a polygon,
annotate at the centroid
        }

    for shape, label in zip(gdf.geometry, gdf.name):
        xy = switch.get(type(shape))(shape)
        if xy:
            plt.annotate(label, xy=xy, xytext=(3, 3), textcoords="offset points")

    plt.xlim(gdf.total_bounds[0], gdf.total_bounds[2])
    plt.ylim(gdf.total_bounds[1], gdf.total_bounds[3])
    plt.legend(['Transit', 'Street', 'Location'])
    plt.show()

plt.xlabel('x-coordinate')
plt.ylabel('y-coordinate')
plt.title('Our First Map')

plot_from_gdf(gdf_map)
```



Finding distance and intersection between locations (This is why Combining `Point` and `LineString` with other features into a single `GeoDataFrame` can be very useful in analysis)

```
# Given two locations, we can calculate the distance between them
def distance_between_two_locations(gdf, location1, location2) -> float:
    # Extract the geometry of the two locations
    geom1 = gdf[gdf['name'] == location1].geometry.values[0] #convert GeoSeries to
shapely.geometry
    geom2 = gdf[gdf['name'] == location2].geometry.values[0]
    # Computes the Cartesian distance between two geometries.
    return geom1.distance(geom2)

# Given two lines, we can calculate the intersection between them
def intersection_between_two_lines(gdf, location1, location2) -> list:
    geom1 = gdf[gdf['name'] ==  location1].geometry.values[0]
    geom2 = gdf[gdf['name'] ==  location2].geometry.values[0]
    # Returns the geometry that is shared between input geometries
    intersecton = geom1.intersection(geom2)
    return [(x,y) for x,y in zip(intersecton.xy[0],intersecton.xy[1])]

distance_between_two_locations(gdf_line, 'Spadina-Bloor', 'Spadina-College')  # 9.0
intersection_between_two_lines(gdf_line, 'TTC 506', 'Yonge Line')             # [(50.0,
70.0)]
```

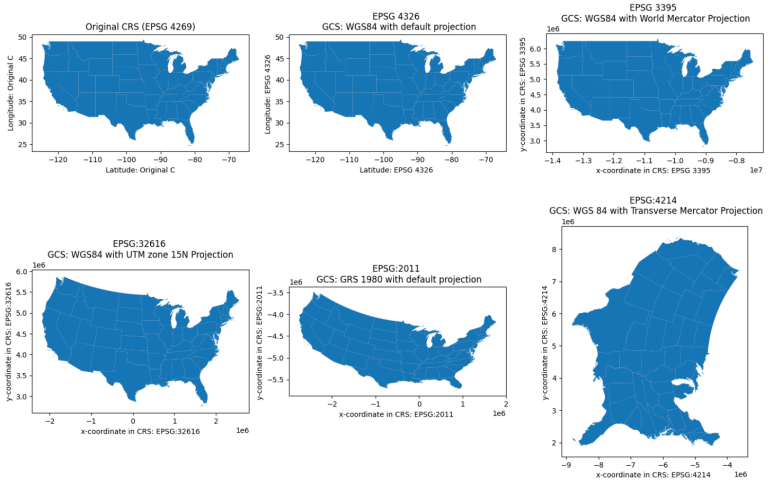
Coordinate Reference Systems (CRS)

2 main categories of CRS:

1. Geographic coordinates (GCS): aka geodetic datum. They define a global position in degrees of latitude and longitude relative to the equator and the prime meridian. With this system, we can easily specify any location on earth. It is used widely, for example, in GPS. The most popular CRS is EPSG:4326, also called WGS84.

    - Positions on the Earth's surface are gauged in angular units from the center of the Earth in relation to two planes: the equatorial plane and the prime meridian plane.
2. Projected coordinates (PCS): projecting GCS onto a 2D plane. While Earth is round, we usually represent it on a two-dimension map. Projected coordinates express locations in X and Y dimensions, thereby allowing us to work with a length unit, such as meters, instead of degrees, which makes the analysis more convenient and effective.
    - However, moving from the three-dimensional Earth to a two-dimensional map will inevitably result in distortions. That's why there are different approaches to creating projected coordinates. For example, many countries have adopted a standard projected CRS for their particular geography.

```
# CRS not defined by default.
print('CRS from our line map:', gdf_line.crs)
gdf_line.set_crs(epsg=4326,inplace=True)
print('CRS from our line map:', gdf_line.crs)

# Change CRS in the GeoDataFrame (sepcify the EPSG code)
gdf_line.to_crs(epsg=2011,inplace=True)
print('CRS from our line map:', gdf_line.crs)
```



Geofencing

Creating a virtual geographical boundary on a map to detect when users (points) enter or leave this boundary. Geofencing allows for tracking whether a point, such as a user's device location, is within a specific region or not. Additionally, given a region, it enables the generation of a list of users within that region. Has these purposes:

- Triggering marketing actions or advertisements for specific geographic locations.
- Enhancing augmented reality experiences, such as providing information about activities in a park as users walk through it.
- Tracking traffic flow between different regions of a city or geographical area, such as monitoring the number of people commuting from Scarborough to downtown Toronto.
- Ecological tracking of GPS-tagged animals, like monitoring the movements of whales entering a bay and studying their seasonal patterns.

Checking if something is inside a polygon

```
# Given location (name/coordinate) and area (geofence), we can check whether the location
is within area
def location_within_area(gdf, location, area) -> bool:
     # If given Point, use it, otherwise, get the geometry of the location
    geom_location = location if isinstance(location, Point) else gdf[gdf['name'] ==
location].geometry.values[0]
    geom_area = gdf[gdf['name'] == area].geometry.values[0]
    return geom_area.contains(geom_location)

print('Is the Spadina-Bloor within the UofT: ', location_within_area(gdf_map, 'Spadina-
Bloor', 'UofT'))
print('Is the Point(45, 75) within the UofT: ', location_within_area(gdf_map, Point(45,
75), 'UofT'))
# You can directly use the Point object

# Or iterate through a list of locations
user_location = [Point(50, 77), Point(31, 20), Point(47, 77), Point(50, 70)]
for location in user_location:
    print(f'Is the location {location} within the UofT: ', location_within_area(gdf_map,
location, 'UofT'))
```

Is the Spadina-Bloor within the UofT: False
Is the Point(45, 75) within the UofT: True
Is the location POINT (50 77) within the UofT: True
Is the location POINT (31 20) within the UofT: False
Is the location POINT (47 77) within the UofT: True
Is the location POINT (50 70) within the UofT: False

```
# Print all the locations within the area (geo-fence)
def get_all_location_within_area(gdf, area) -> None:

    # Get the geometry of the area
    if isinstance(area, Polygon):
        geom_area = area
    else:
        geom_area = gdf[gdf['name'] == area].geometry.values[0]

    # Iterate through the GeoDataFrame to find all the locations within the area
    for _, row in gdf.iterrows():
        if geom_area.contains(row.geometry) and isinstance(row.geometry, Point):
            print(row['name'], ' is within ', area)

# Add a new location to the existing GeoDataFrame
new_location = gpd.GeoDataFrame({'name': 'Robarts Library', 'properties':
                                 'Location', 'geometry': Point(43, 75)}, index=[0])
gdf_map = pd.concat([gdf_map, new_location], ignore_index=True)
get_all_location_within_area(gdf_map, 'UofT')  # Robarts Library  is within  UofT
```

Geocoding

Geocoding is the process of converting addresses (like "1600 Amphitheatre Parkway, Mountain View, CA") into geographic coordinates (like latitude 37.423021 and longitude -122.083739), which you can use to place markers on a map, or position the map.

Reverse geo-coding is the opposite process. It involves converting geographic coordinates into a human-readable address. So, you input latitude and longitude coordinates, and the reverse geocoding service provides you with the corresponding address or location information.

```
from geopy.geocoders import Nominatim

# Geo-coding
geolocator = Nominatim(user_agent="my_app")
location = geolocator.geocode("27 King's College Cir, Toronto, ON M5S 1A1") # UofT address
print(location.latitude, location.longitude) # 43.6607231 -79.39592001845992

# Reverse geo-coding
address = geolocator.reverse("43.6452, -79.3806")
print(address.address)  # 140, Bay Street, St Lawrence-East Bayfront-The Islands, Spadina—
Fort York, Toronto, Golden Horseshoe, Ontario, M5J 2L5, Canada
```

Geo-coding can be used for geo-fencing:

```
import folium
from shapely.geometry import Polygon

address_lst = ['290 Bremner Blvd, Toronto, ON M5V 3L9',
 '1 Austin Terrace, Toronto, ON M5R 1X8',
 '770 Don Mills Rd., North York, ON M3C 1T3',
 '77 Wynford Dr, North York, ON M3C 1K1']

# Geo-coding to latitude and longitude
geolocator = Nominatim(user_agent="my_app", timeout=10) # if code fails, increase this
number
address_coded = [geolocator.geocode(address) for address in address_lst]

# Create list of point objects using this geo-coding
list_of_points = [Point(i.latitude,i.longitude) for i in address_coded]

# Define coordinates for downtown Toronto boundary
downtown_toronto_boundary = [(43.68332301188323, -79.41832982140303),
 (43.63659467805622, -79.3998381602896), (43.65214101062051, -79.34847901920116),
  (43.69235150490925, -79.3692961282509)]
# Create a polygon
downtown_polygon = Polygon(downtown_toronto_boundary)

for point in list_of_points:
  if downtown_polygon.contains(point):
    folium.Marker((point.x, point.y),
popup='within',icon=folium.Icon(color='blue')).add_to(map_toronto)
```

```
  else:
    folium.Marker((point.x, point.y),
popup='out',icon=folium.Icon(color='red')).add_to(map_toronto)
map_toronto
```

Spatial analysis

COVID-19 cases example

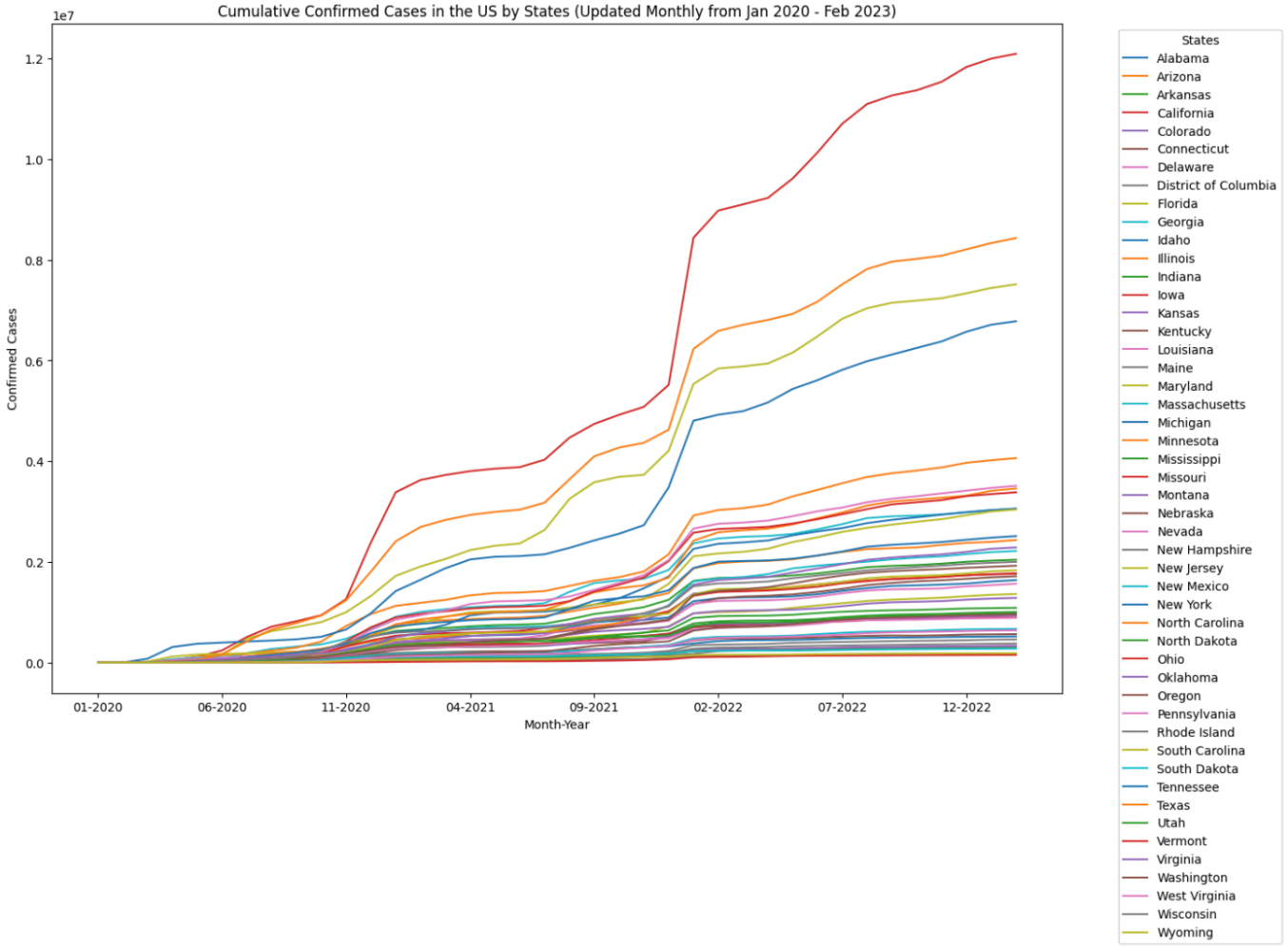
```
confirmed_case = pd.read_csv('https://github.com/MIE223-2024/course-
datasets/raw/main/time_series_covid19_confirmed_US.csv')

# check the shape of the data
print(f'Number of Columns: {confirmed_case.shape[1]} and Number of Samples:
{confirmed_case.shape[0]}')

# missing values
missing = pd.isnull(confirmed_case).sum()
print('Missing: ', [i for i in missing[missing  > 0].items()])

# since the data is for US, we expect 1
print('Number of unique Country_Region: ', confirmed_case['Country_Region'].nunique())

# number of states in the US
print('Number of unique Province_State: ', confirmed_case['Province_State'].nunique())

states_in_shapefile = gdf_us_by_states['NAME'].unique()
states_in_confirmed_case = confirmed_case['Province_State'].unique()

# Check the difference between the two lists
print('States in the confirmed case but not in the shape files: ',
set(states_in_confirmed_case) - set(states_in_shapefile))
print('States in the shape files but not in the confirmed case: ',
set(states_in_shapefile) - set(states_in_confirmed_case))

# dropping columns that are not needed in this analysis
confirmed_case.drop(columns=['UID', 'iso2', 'iso3', 'code3', 'FIPS', 'Admin2',
'Country_Region', 'Lat', 'Long_', 'Combined_Key'], inplace=True)

# remove the rows that are not mainland states
confirmed_case = confirmed_case[~confirmed_case['Province_State'].isin(['Diamond
Princess', 'Grand Princess','Northern Mariana Islands', 'American Samoa', 'Guam', 'Virgin
Islands', 'Puerto Rico', 'Alaska', 'Hawaii'])]

# aggregate the data by states (from counties to states)
confirmed_case_by_states = confirmed_case.groupby('Province_State').sum()
confirmed_case_by_states.head()

# To simplify the dataset for this lab, we will aggregate each sample from daily cases to
monthly cases.
# merge by month
```

```
confirmed_case_by_states = confirmed_case_by_states.T
#reforamt the index to datetime
confirmed_case_by_states.index =  pd.to_datetime(confirmed_case_by_states.index,
format='%m/%d/%y')
# Since it's total cases from the first day to the current date,
# we only need to keep the data for the last day of each month to represent the entire
month.
confirmed_case_by_states = confirmed_case_by_states.resample('M').last()
# drop the last incomplete month
confirmed_case_by_states = confirmed_case_by_states.iloc[:-1].T.reset_index()
# rename the columns to the month and year
confirmed_case_by_states.columns = ['Province_State'] + [x.strftime('%m-%Y') for x in
confirmed_case_by_states.columns[1:]]
confirmed_case_by_states.head()
```

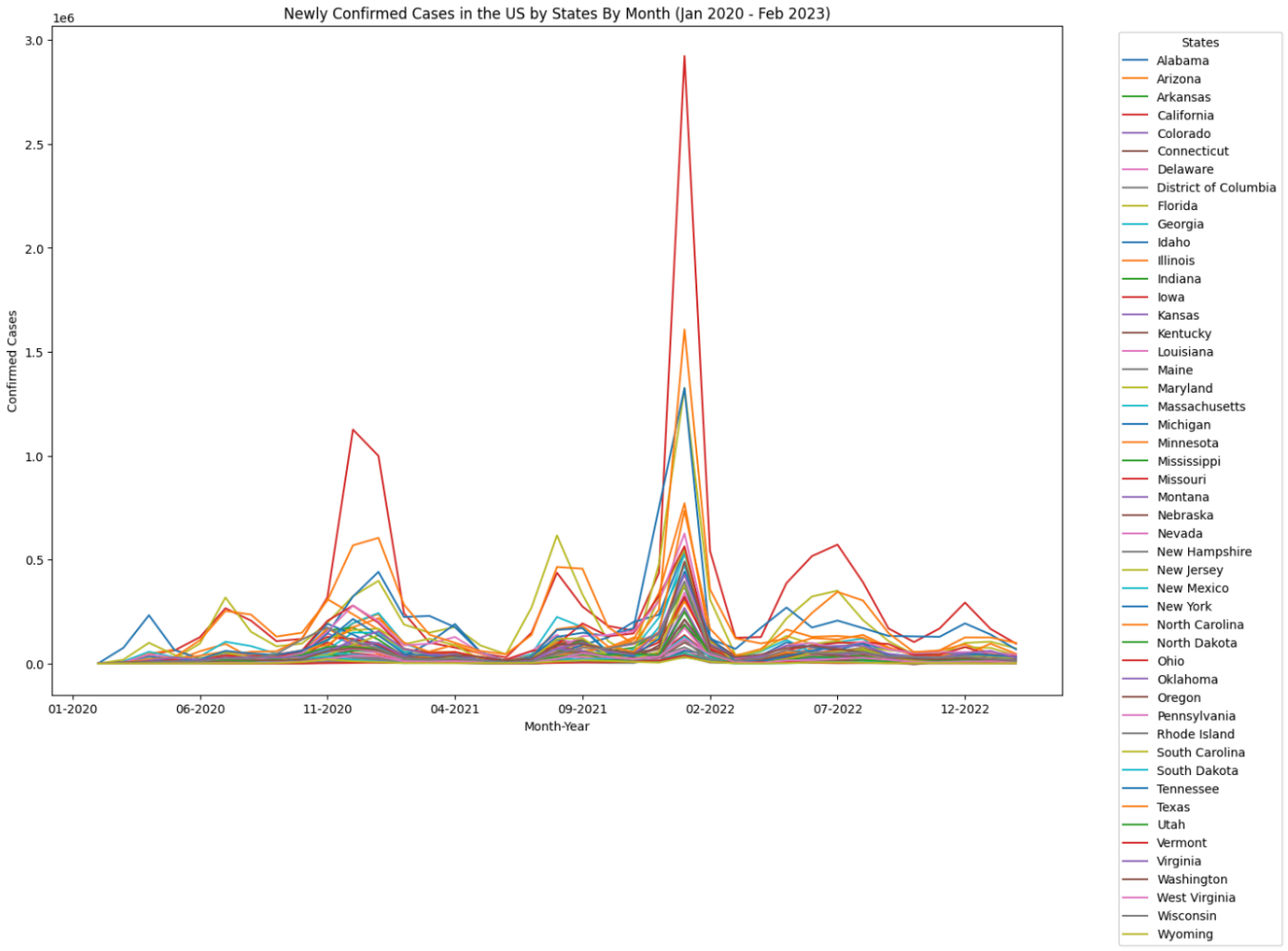
Time series plot

- Large sample sizes will result in a very long legend that is difficult to interpret and compare.
- We lose the geographical information associated with each sample in the time series plot, preventing us from examining their spatial relationships.

```
# plot the confirmed cases in the US
tem = confirmed_case_by_states.T.iloc[1:]
tem.columns = confirmed_case_by_states['Province_State']
tem.plot(figsize=(15, 10))
plt.title('Cumulative Confirmed Cases in the US by States (Updated Monthly from Jan 2020 -
Feb 2023)')
# improve legend visibility
plt.legend(title='States', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xlabel('Month-Year')
plt.ylabel('Confirmed Cases')
plt.show()
```




```
# Calculating the difference in confirmed cases between months
tem = confirmed_case_by_states.T.iloc[1:].diff()
tem.columns = confirmed_case_by_states['Province_State']
tem.plot(figsize=(15, 10))
plt.title('Newly Confirmed Cases in the US by States By Month (Jan 2020 - Feb 2023)')
# improve legend visibility
plt.legend(title='States', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xlabel('Month-Year')
plt.ylabel('Confirmed Cases')
plt.show()
```



Choropleth

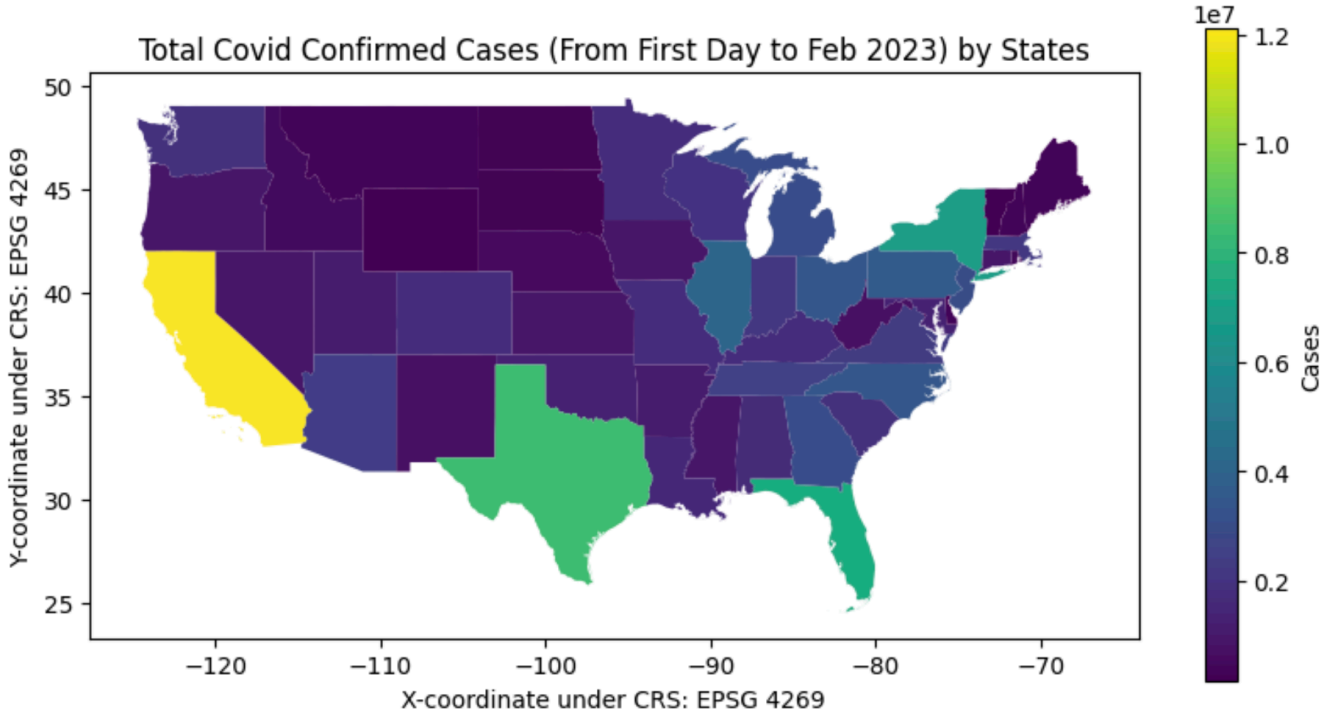
Choropleth: maps where the color of each shape is based on the value of an associated variable.

Merge the geospatial data with the cases first:

```
gdf_us_by_states = gdf_us_by_states.rename(columns={'NAME': 'Province_State'})
# Merge the GeoDataFrame with the confirmed cases DataFrame based on the State name
gdf_covid_confirmed = gdf_us_by_states[['Province_State',
'geometry']].merge(confirmed_case_by_states)
#Make sure you convert the DataFrame to GeoDataFrame after merging (in default, it's a
DataFrame)
gdf_covid_confirmed = gpd.GeoDataFrame(gdf_covid_confirmed)
gdf_covid_confirmed.head()
```

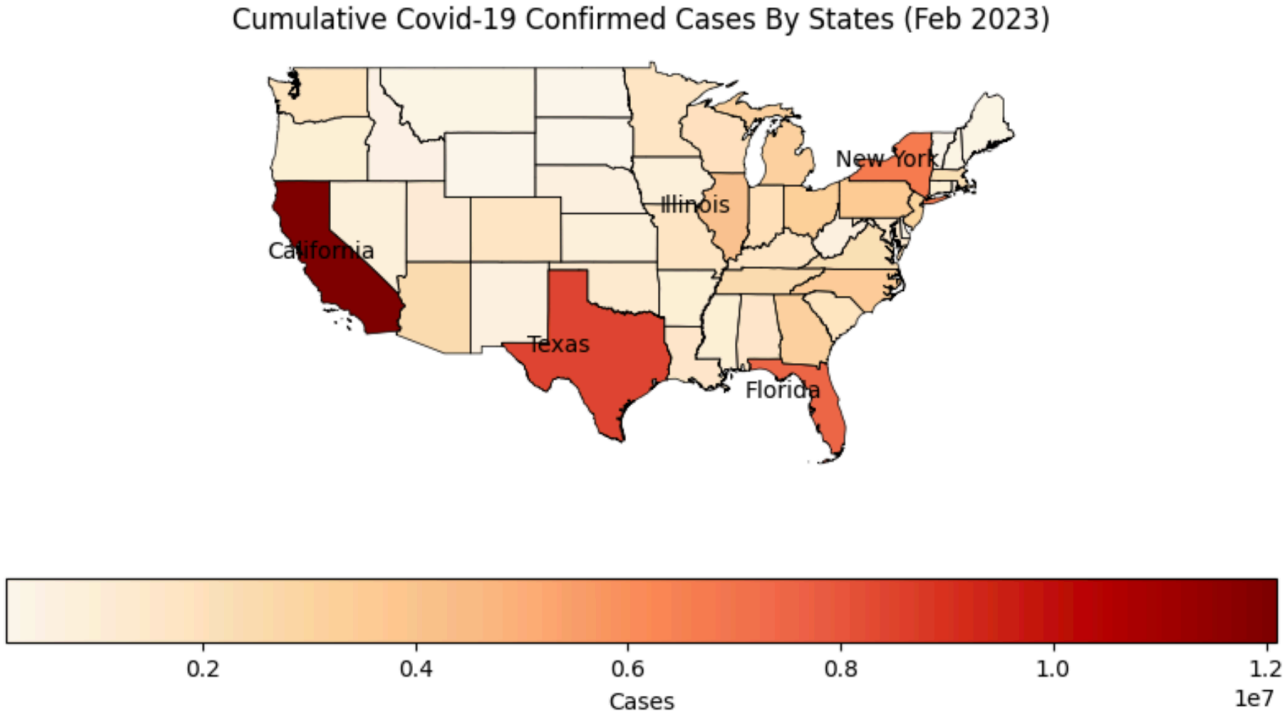
Create a choropleth

```
gdf_covid_confirmed.plot(figsize=(10, 5), column='02-2023',
                         legend=True, # include the legend
                         legend_kwds={"label": "Cases"}) # legend style
plt.title('Total Covid Confirmed Cases (From First Day to Feb 2023) by States')
plt.xlabel('X-coordinate under CRS: EPSG 4269')
plt.ylabel('Y-coordinate under CRS: EPSG 4269')

plt.show()
```



Add some shenanigans to the map (cmap, labels, etc)

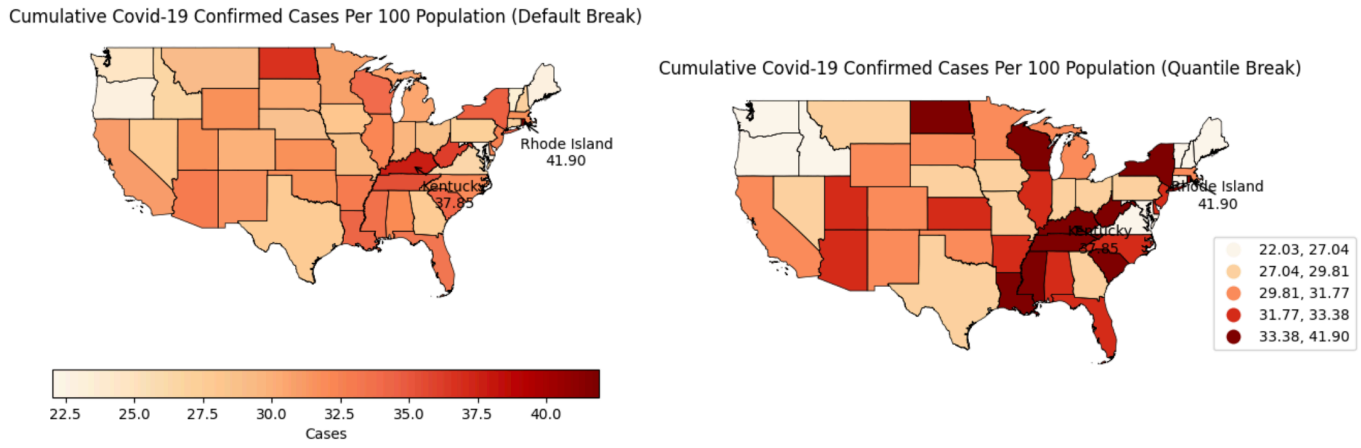
```
gdf_covid_confirmed.plot(figsize=(10, 5), column='02-2023',
                         legend=True,cmap='OrRd',
                         legend_kwds={"label": "Cases", "orientation": "horizontal"}) #
change the legend style

# plot the boundary of the states
gdf_covid_confirmed.boundary.plot(figsize=(10, 5),ax = plt.gca(), color='k',
linewidth=0.5)

# label the top 5 states with the highest confirmed cases
max_cases_state = gdf_covid_confirmed.nlargest(5, '02-2023')
for x, y, label in zip(max_cases_state.geometry.centroid.x,
                       max_cases_state.geometry.centroid.y,
                       max_cases_state['Province_State']):
    plt.gca().annotate(label, xy=(x, y), xytext=(-30, 0), textcoords="offset points")

plt.title('Cumulative Covid-19 Confirmed Cases By States (Feb 2023)')
plt.axis('off') # Or we can remove the axis
plt.show()
```



Redefine default colour intervals with the `scheme` parameter (e.g. base each color on the quantiles of the given data).

```
# get current month's data
current_month = gdf_covid_confirmed[['Province_State','02-2023','geometry','Population']]
# calculate the confirmed cases per 100 people in the state
current_month['Density'] = current_month['02-2023'] / current_month['Population'] * 100

# visualize the density
fig,ax  = plt.subplots(1, 2, figsize=(15, 5))

# By default scheme
current_month.plot(ax=ax[0], column='Density', cmap='OrRd',legend=True,
                   figsize=(5, 5),legend_kwds={"label": "Cases", "orientation":
"horizontal"})
gdf_covid_confirmed.boundary.plot(ax = ax[0], color='k', linewidth=0.5) # plot the
boundary of the states
ax[0].set_title('Cumulative Covid-19 Confirmed Cases Per 100 Population (Default Break)')
ax[0].axis('off')

# By quantile scheme
current_month.plot(figsize=(5, 5), column='Density',scheme='quantiles',
                   cmap='OrRd',legend=True,ax=ax[1],
                   legend_kwds={"bbox_to_anchor": (1.2, 0.5)}) #ensure the legend does not
block the map
# plot the boundary of the states
gdf_covid_confirmed.boundary.plot(ax = ax[1], color='k', linewidth=0.5)
ax[1].set_title('Cumulative Covid-19 Confirmed Cases Per 100 Population (Quantile Break)')
ax[1].axis('off') # Or we can remove the axis

# label the top 2 states with the highest confirmed cases by its name and value
max_cases_state = current_month.nlargest(2, 'Density') # get the top 2 states
```

```
for x, y, name, value in zip(max_cases_state.geometry.centroid.x,
                             max_cases_state.geometry.centroid.y,
                             max_cases_state['Province_State'],
                             max_cases_state['Density']):
    ax[0].annotate(f'{name}\n{value:.2f}', xy=(x, y), xytext=(30, -30),
                   textcoords="offset points", ha='center',
arrowprops=dict(arrowstyle='->'))
    ax[1].annotate(f'{name}\n{value:.2f}', xy=(x, y), xytext=(20, -20),
                  textcoords="offset points", ha='center', arrowprops=dict(arrowstyle='-
>'))
plt.show()
```



Choropleth across time

Map choropleth across time by putting multiple choropleths side by side with each other.

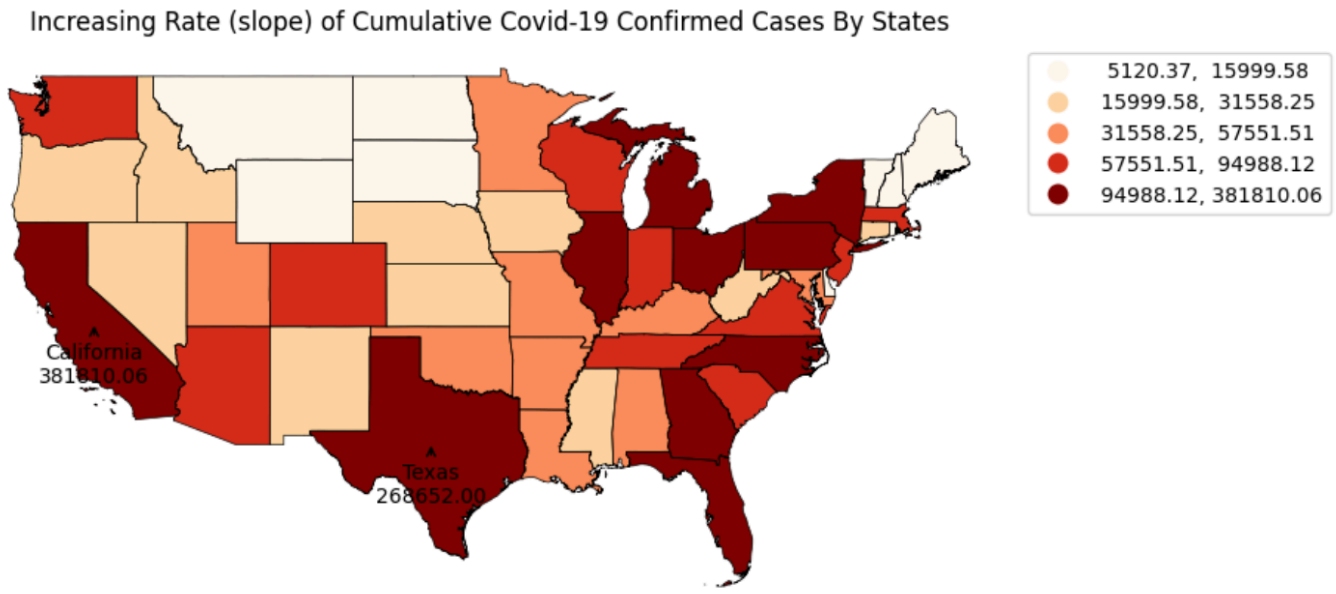
```
# get current month's data
current_month = gdf_covid_confirmed[['Province_State','02-2023','geometry','Population']]
# calculate the confirmed cases per 100 people in the state
current_month['Density'] = current_month['02-2023'] / current_month['Population'] * 100

# visualize the density
fig,ax  = plt.subplots(1, 2, figsize=(15, 5))

# By default scheme
current_month.plot(ax=ax[0], column='Density', cmap='OrRd',legend=True,
                   figsize=(5, 5),legend_kwds={"label": "Cases", "orientation":
"horizontal"})
gdf_covid_confirmed.boundary.plot(ax = ax[0], color='k', linewidth=0.5) # plot the
boundary of the states
ax[0].set_title('Cumulative Covid-19 Confirmed Cases Per 100 Population (Default Break)')
ax[0].axis('off')

# By quantile scheme
current_month.plot(figsize=(5, 5), column='Density',scheme='quantiles',
                   cmap='OrRd',legend=True,ax=ax[1],
                   legend_kwds={"bbox_to_anchor": (1.2, 0.5)}) #ensure the legend does not
block the map
# plot the boundary of the states
```

```
gdf_covid_confirmed.boundary.plot(ax = ax[1], color='k', linewidth=0.5)
ax[1].set_title('Cumulative Covid-19 Confirmed Cases Per 100 Population (Quantile Break)')
ax[1].axis('off') # Or we can remove the axis

# label the top 2 states with the highest confirmed cases by its name and value
max_cases_state = current_month.nlargest(2, 'Density') # get the top 2 states
for x, y, name, value in zip(max_cases_state.geometry.centroid.x,
                             max_cases_state.geometry.centroid.y,
                             max_cases_state['Province_State'],
                             max_cases_state['Density']):
    ax[0].annotate(f'{name}\n{value:.2f}', xy=(x, y), xytext=(30, -30),
                   textcoords="offset points", ha='center',
arrowprops=dict(arrowstyle='->'))
    ax[1].annotate(f'{name}\n{value:.2f}', xy=(x, y), xytext=(20, -20),
                  textcoords="offset points", ha='center', arrowprops=dict(arrowstyle='-
>'))

plt.show()
```

Choropleth by simple regression: utilizing the slope of the fitted regression line as the representation of each sample, we can estimate the rate of increase for each state over time.

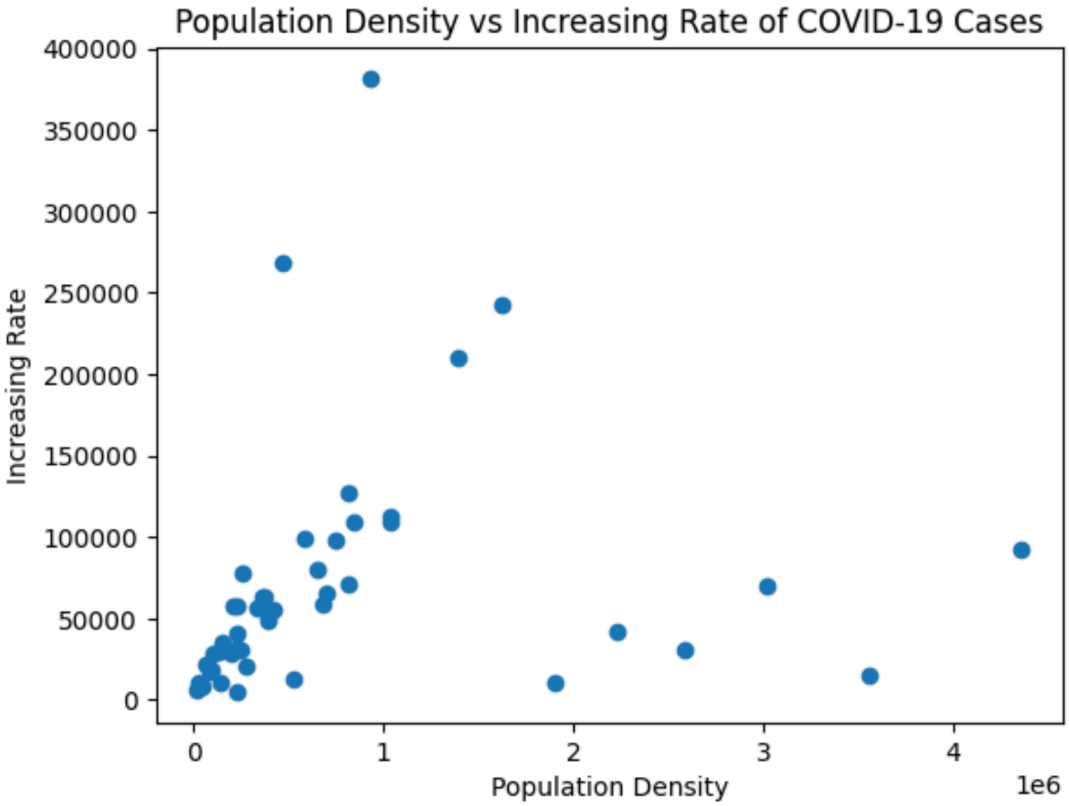
```
def linear_regression(sample):
    x = range(len(sample))
    y = sample
    mean_x = sum(x) / len(x)
    mean_y = sum(y) / len(y)

    # Calculate the slope (m) and intercept (b) of the regression line
    numerator = sum((xi - mean_x) * (yi - mean_y) for xi, yi in zip(x, y))
    denominator = sum((xi - mean_x) ** 2 for xi in x)
    slope = numerator / denominator
    intercept = mean_y - slope * mean_x

    return slope, intercept

# Calculate the linear regression for the total cases in Alabama
col = [col for col in gdf_covid_confirmed.columns if col not in ['Province_State',
'geometry', 'Population']]
slope, intercept = linear_regression(gdf_covid_confirmed.loc[0,col].tolist()) # get the
slope and intercept

gdf_increase_rate = gdf_covid_confirmed.copy()
# Calculate the slope for each state
col = [col for col in gdf_increase_rate.columns if col not in ['Province_State',
'geometry', 'Population']]
gdf_increase_rate['slope'] = gdf_increase_rate.apply(lambda x: linear_regression(x[col])
[0], axis=1)

# Choropleth based on the slope
gdf_increase_rate.plot(figsize=(10, 5), column='slope', cmap='OrRd',
```

```
                        legend=True,scheme='quantiles',
                          legend_kwds={"bbox_to_anchor": (1, 1),
                                       'loc': 'upper left'})
gdf_increase_rate.boundary.plot(color='k', linewidth=0.5,ax = plt.gca())

# Label the top 2 states with the highest increase rate
max_cases_state = gdf_increase_rate.nlargest(2, 'slope') # get the top 2 states
for x, y, name, value in zip(max_cases_state.geometry.centroid.x,
                             max_cases_state.geometry.centroid.y,
                             max_cases_state['Province_State'],
                             max_cases_state['slope']):
    plt.gca().annotate(f'{name}\n{value:.2f}', xy=(x, y), xytext=(0, -30),
                       textcoords="offset points", ha='center',
arrowprops=dict(arrowstyle='->'))

plt.title('Increasing Rate (slope) of Cumulative Covid-19 Confirmed Cases By States')
plt.axis('off')
plt.show()
```



Detecting correlation with a choropleth

```
# Correlation between population density and confirmed cases
tem = gdf_increase_rate.merge(gpf_pop_density, on='Province_State')
# Drop the outlier
tem = tem[tem['Province_State'] != 'District of Columbia']
plt.scatter(tem['Pop Density'], tem['slope'])
plt.title('Population Density vs Increasing Rate of COVID-19 Cases')
plt.xlabel('Population Density')
plt.ylabel('Increasing Rate')
plt.show()

# Correlogram
tem[['Pop Density', 'slope']].corr()
```


Population Density vs Increasing Rate of COVID-19 Cases
Increasing Rate
400000
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
Population Density
1e6

Lab 10: More on Choropleth, Folium Map, and Raster Data

Toronto Transportation Fairness

```
# Get Toronto Forward Sortation Area (FSA)
toronto_FSA = gpd.read_file('lfsa000a16a_e')
# 'CFSAUID' start with m
toronto_FSA = toronto_FSA[toronto_FSA['CFSAUID'].str.startswith('M')]
toronto_FSA.plot(figsize=(5, 5), legend=True)
plt.title('Toronto Forward Sortation Area (FSA)')
plt.axis('off')
plt.show()

# Checking CRS
toronto_FSA.crs

# Change CRS
toronto_FSA.to_crs(epsg=4326, inplace=True)
```

Open-Street Map API

```
import osmnx as ox

place = "City of Toronto, Ontario, Canada" # define the place for data extraction
tags = {'network':'TTC'} # define the Keys, and value of Keys
gdf_TTC = ox.features.features_from_place(place, tags).reset_index() # extract the
geodataframe
```

Problem 1: The TTC network dataset includes over 8000 geo-entities like bus routes, stations, and garages, with varied features across types—resulting in many NaN values. To reduce missing data, we can select only common features like `name`, `element type`, and `geometry`.

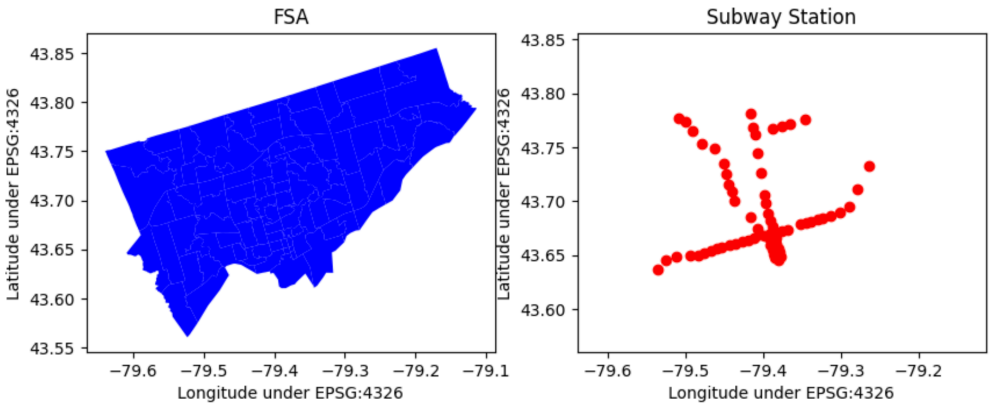
Problem 2: Since we're only interested in the ~70 TTC subway stations, we can filter the dataset using the `'subway'` column, which is marked `'yes'` for relevant entries. Alternatively, using multiple tags from the OSM database returns a union of entities, so we'd need to manually remove rows with nulls to isolate the subway stations.

```
# We want data containing only the TTC subway stations.
place = "City of Toronto, Ontario, Canada" # define the place for data extraction
tags = {'network':'TTC','subway':'yes'} # define the feature, and value of feature
gdf_subway_from_query = ox.features.features_from_place(place, tags).reset_index() #
extract the data
gdf_subway_from_query = gdf_subway_from_query[pd.notnull(gdf_subway_from_query["network"])
& pd.notnull(gdf_subway_from_query["subway"])] # remove the null values
gdf_subway_from_query.shape

# We only want to keep Point, not LineString or Polygon
```

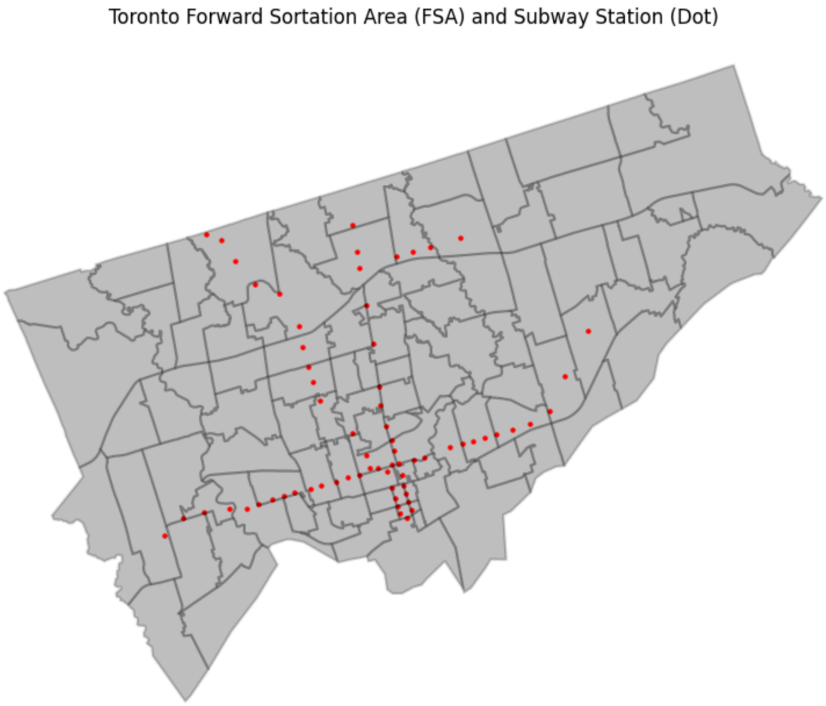
```
gdf_subway_station = gdf_subway[gdf_subway.geometry.type == 'Point'][['name', 'geometry']]
# filter the data and only keep the name and geometry
print(gdf_subway_station.shape) # print the shape of the data
gdf_subway_station.head() # print the first 5 rows of the data

# Drop row with duplicate name
gdf_subway_station =
gdf_subway_station.drop_duplicates(subset='name').reset_index(drop=True)

# Rename the columns
toronto_FSA.columns = ['name', 'geometry']
# Create a new column called property_type to help us distinguish the data
toronto_FSA['property_type'] = 'FSA'
gdf_subway_station['property_type'] = 'subway_station'
# Concatenate the data
gdf_all = pd.concat([toronto_FSA, gdf_subway_station  ], axis=0, ignore_index=True)
```

Visualization of TFSA and subway

```
fig, ax = plt.subplots(1,2, figsize=(10, 5))
gdf_all[gdf_all['property_type'] == 'FSA'].plot(figsize=(5, 5), legend=True, color='blue',
ax=ax[0])
gdf_all[gdf_all['property_type'] == 'subway_station'].plot(figsize=(5, 5), legend=True,
color='red', ax=ax[1])
ax[0].set_title('FSA')
ax[1].set_title('Subway Station')
ax[0].set_xlabel('Longitude under EPSG:4326')
ax[0].set_ylabel('Latitude under EPSG:4326')
ax[1].set_xlabel('Longitude under EPSG:4326')
ax[1].set_ylabel('Latitude under EPSG:4326')
#allow both plots have the same boundary
ax[1].set_xlim(gdf_all[gdf_all['property_type'] == 'FSA'].total_bounds[0],
gdf_all[gdf_all['property_type'] == 'FSA'].total_bounds[2])
ax[1].set_ylim(gdf_all[gdf_all['property_type'] == 'FSA'].total_bounds[1],
gdf_all[gdf_all['property_type'] == 'FSA'].total_bounds[3])
plt.show()
```



```
# plot the FSA shapefile and Subway Station point in the same plot
gdf_all[gdf_all['property_type'] == 'FSA'].plot(figsize=(10, 10), legend=True, color =
'grey',alpha=0.5)
gdf_all[gdf_all['property_type'] == 'subway_station'].plot(figsize=(10, 10), legend=True,
                                                           color='red',
ax=plt.gca(),alpha=1, markersize=5)
# boundary of the shapefile
gdf_all[gdf_all['property_type'] == 'FSA'].boundary.plot(figsize=(10, 10), color =
'black', ax=plt.gca(),alpha=0.2)

plt.title('Toronto Forward Sortation Area (FSA) and Subway Station (Dot)')
plt.axis('off')
plt.show()
```




```
# Show edges only
ox.plot_graph(street_map, figsize=(10, 10), bgcolor='w',node_size=1, node_alpha=0,
              edge_linewidth=0.5, edge_color='k') #show edge only, change node_alpha to 0

# Show nodes (intersections) only
ox.plot_graph(street_map, figsize=(10, 10), bgcolor='w',node_size=1, node_alpha=1,
node_color='r',
              edge_linewidth=0.5, edge_alpha=0) #show node only, change the edge_alpha to
0

# Show both nodes and edges
ox.plot_graph(street_map, figsize=(10, 10), bgcolor='w',node_size=1, node_alpha=0.5,
node_color='r',
              edge_linewidth=0.5, edge_color='k')

# Get the bike network within the boundary
```

```
ox.plot_graph(ox.graph_from_bbox(bbox = [43.71, 43.72, -79.41, -79.40],
network_type='bike'),
              figsize=(10, 10),node_size=50, node_alpha=1, node_color='r',
              edge_linewidth=0.5, edge_color='w')

# Show CRS
street_map.graph['crs']
print('Number of nodes:', street_map.number_of_nodes()) # print the number of nodes
print('Number of edges:', street_map.number_of_edges()) # print the number of edges
```

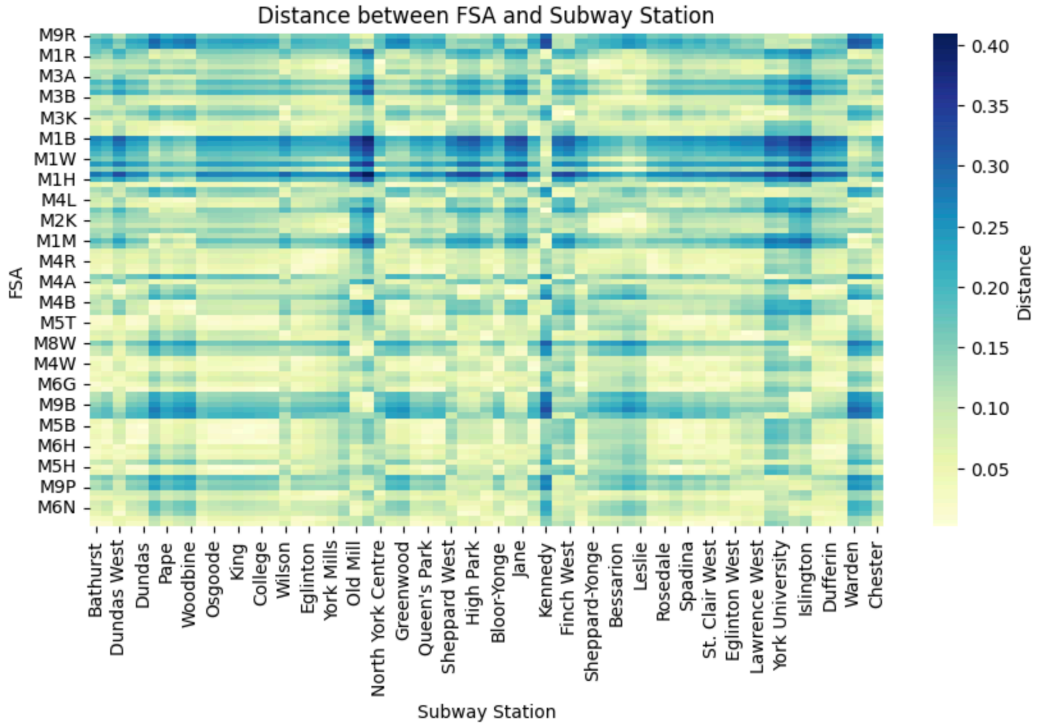
Calculating distances between polygons: centroids

```
def get_centroid(geom):
  if geom.geom_type == 'Polygon' or geom.geom_type == 'MultiPolygon':
    return geom.centroid
  else:
    return geom

gdf_all_with_centroid['centroid'] = gdf_all_with_centroid['geometry'].apply(get_centroid)
```

Distance matrix between all the FSA centroids and each subway station

```
fsa_centroid = gdf_all_with_centroid[gdf_all_with_centroid['property_type'] == 'FSA']
[['name', 'centroid']]
subway_station_centroid = gdf_all_with_centroid[gdf_all_with_centroid['property_type'] ==
'subway_station'][['name', 'centroid']]

# Calculate the distance between each FSA and subway station
def get_distance_matrix(fsa_centroid, subway_station_centroid) -> pd.DataFrame:
    '''
    function is used to calculate the distance matrix between each FSA and subway station
    fsa_centroid: dataframe with the centroid of FSA
    subway_station_centroid: dataframe with  the centroid of subway station
    return: a dataframe with the distance between each FSA and subway station
    '''
    distance_matrix = np.zeros((fsa_centroid.shape[0], subway_station_centroid.shape[0]))
```

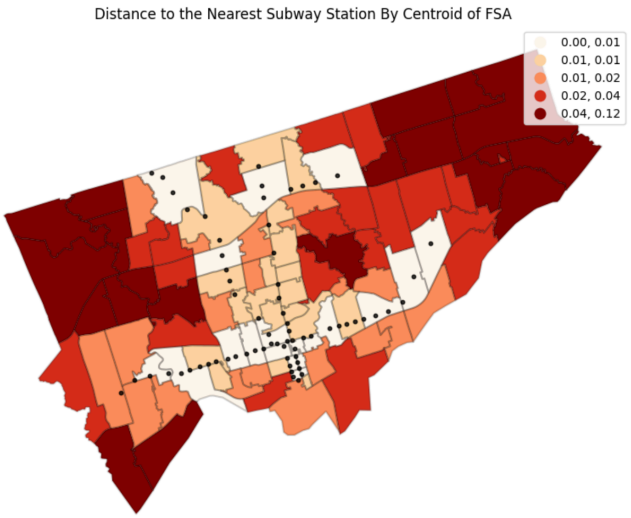
```
    for i in range(fsa_centroid.shape[0]):
        for j in range(subway_station_centroid.shape[0]):
            distance_matrix[i, j] = fsa_centroid.iloc[i]
['centroid'].distance(subway_station_centroid.iloc[j]
['centroid'])
    distance_matrix = pd.DataFrame(distance_matrix, index=fsa_centroid['name'].to_list(),
columns=subway_station_centroid['name'])
    return distance_matrix

distance_matrix = get_distance_matrix(fsa_centroid, subway_station_centroid)

# Visualize the matrix
import seaborn as sns
plt.figure(figsize=(10, 5))
sns.heatmap(distance_matrix, cmap='YlGnBu', cbar_kws={'label': 'Distance'})
plt.title('Distance between FSA and Subway Station')
plt.xlabel('Subway Station')
plt.ylabel('FSA')
plt.show()
```



Obtain distance from each FSA to closest station

```
# Create a new pandas series to store the nearest distance
smallest_distance = distance_matrix.min(axis=1)
# Assign the name of the series for future merging
smallest_distance.name = 'distance_to_nearest_subway_station'
# Obtain all row that is a foward sortation area
gdf_FSA = gdf_all[gdf_all_with_centroid['property_type'] == 'FSA']

# Merge the distance series with the FSA shape data
```

```
gdf_smallest_distance = gdf_FSA.merge(smallest_distance, left_on='name',
right_index=True).rename(columns={0:
'distance_to_nearest_subway_station'})

# Get quantiles
quantile = gdf_smallest_distance['distance_to_nearest_subway_station'].quantile([0, 0.25,
0.5, 0.75,
1]).to_list()
# Visualize
gdf_smallest_distance.plot(column='distance_to_nearest_subway_station', legend=True,
scheme='quantiles',
              cmap='OrRd', figsize=(10, 10))
gdf_smallest_distance.boundary.plot(figsize=(10, 10), color = 'black',
ax=plt.gca(),alpha=0.2)
# add subway station
gdf_all[gdf_all['property_type'] == 'subway_station'].plot(ax=plt.gca(), color='black',
alpha=0.8, markersize=10)
plt.title('Distance to the Nearest Subway Station By Centroid of FSA')
plt.axis('off')
plt.show()
```



Shortest paths with NetworkX

```
import networkx as nx

# you dont need to convert the graph to networkx graph, you can directly use the graph
from osmnx
shortest_path = nx.shortest_path(street_map, FSA_node , subway_station_node,
weight='length')
ox.plot_graph_route(street_map, shortest_path,figsize=(10, 10), bgcolor='w',node_size=1,
node_alpha=0.3,
              edge_linewidth=0.5, edge_color='k')
plt.show()

distance = nx.shortest_path_length(street_map, FSA_node , subway_station_node,
weight='length')
print('The shortest path length between FSA and subway station is:', distance)
```

Shortest distance between 2 locations

```
def distance_of_shortest_path(graph, fsa_name, subway_station_name, fsa_centroid, subway_station_centroid) -> float:
    '''
    This function will calculate the shortest path length between a FSA and a subway station
    Args:
    graph: the graph object of the route map
    fsa_name: the name of the FSA
    subway_station_name: the name of the subway station
    fsa_centroid: a dataframe of the centroid of all FSA
    subway_station_centroid: a dataframe of the centroid of all subway station
    Returns:
    distance: the shortest path length
    '''
    fsa = fsa_centroid[fsa_centroid['name'] == fsa_name]['centroid'].values[0] # get the centroid of the FSA by given name
    fsa_coords = (fsa.x, fsa.y) # get the coordinates of the centroid
    subway_station = subway_station_centroid[subway_station_centroid['name'] == subway_station_name]['centroid'].values[0] # get the centroid of the subway station by given name
    subway_station_coords = (subway_station.x, subway_station.y) # get the coordinates of the centroid
    fsa_node = ox.distance.nearest_nodes(graph, fsa_coords[0], fsa_coords[1]) # get the nearest node ID in the graph
    subway_station_node = ox.distance.nearest_nodes(graph, subway_station_coords[0], subway_station_coords[1]) # get the nearest node ID in the graph
    distance = nx.shortest_path_length(graph, fsa_node , subway_station_node, weight='length') # calculate the shortest path length
    route = nx.shortest_path(graph, fsa_node , subway_station_node, weight='length') # get the route
    return distance, route #return the shortest path length and the route

distance, route  = distance_of_shortest_path(street_map, 'M1B', 'Dundas', fsa_centroid, subway_station_centroid)
```

```
# Visualization
ox.plot_graph_route(street_map, route,figsize=(10, 10), bgcolor='w',node_size=1,
node_alpha=0.3,
              edge_linewidth=0.5, edge_color='k')
```

Folium maps

- Visualize data that's been manipulated in Python on an interactive map.
- Enables binding of data to map for choropleth visualization; passing rich vector visualizations as markers on a map.

Tileset: visual appearance and style of the map. The default tileset for Folium is OpenStreetMap, you can use `tiles=` parameter to change the tileset

LayerControl: allows users to switch between different tile sets and toggle layers on and off, providing greater flexibility and customization options for their viewing experience.

```
import folium as fm

# Create a map object
m = fm.Map(location=[43.7, -79.4], zoom_start=12)
cartodb_positron = fm.Map(location=[43.7, -79.4], zoom_start=12,
             tiles='cartodb positron', zoom_control=False, scrollWheelZoom=False)
dark_matter = fm.Map(location=[43.7, -79.4], zoom_start=12, scrollWheelZoom=False)
fm.TileLayer('cartodb dark_matter').add_to(dark_matter) # .add_to adds layer to map

fm.LayerControl().add_to(distance_nearest_ttc_map) # add the layer control
```

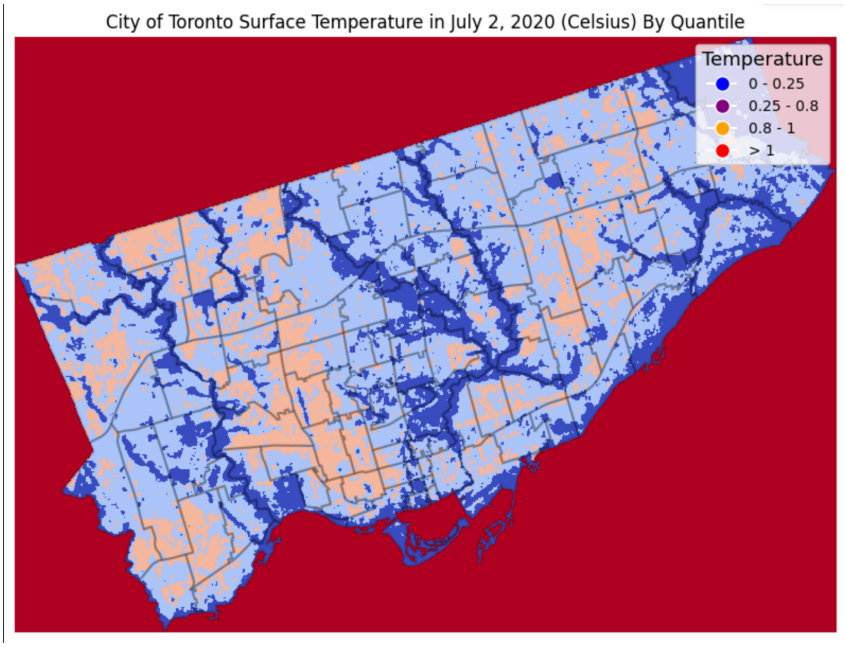
Create a choropleth

```
fm.Choropleth( # create a choropleth layer
    geo_data=FSA_json, # feed in the json data for vector shape
    name='Distance to the Nearest Subway Station', # name of the layer
    data=gdf_smallest_distance, # feed in the dataframe for feature properties
    columns=['name', 'distance_to_nearest_subway_station'], # columns to use from the
dataframe
    key_on='feature.properties.name', # key to match the json data and dataframe
    fill_color='OrRd', # color for the choropleth
    fill_opacity=0.4, # opacity of the fill color
    line_opacity=0.2, # opacity of the boundary line
    bins = quantile, # bins to classify the data
    legend_name='Distance to the Nearest Subway Station' # name of the legend
).add_to(distance_nearest_ttc_map) # add the layer to the map
distance_nearest_ttc_map # display the map
```

Raster data

Classifying the raster data

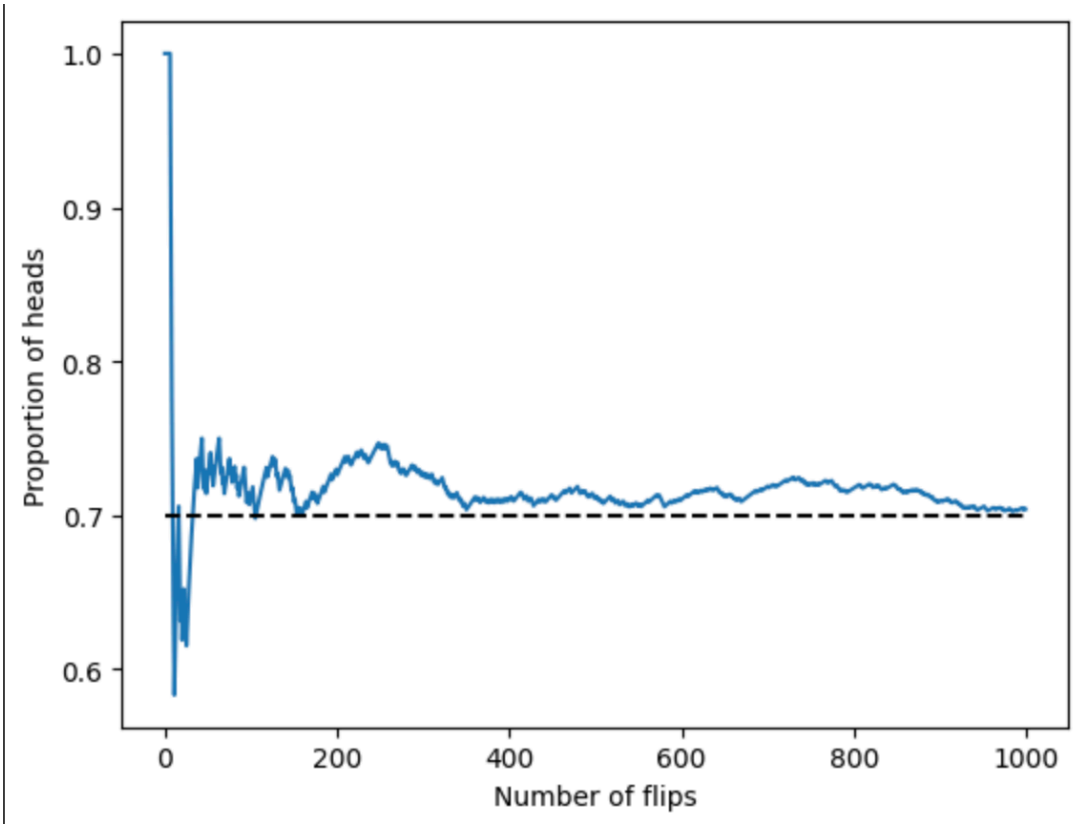
```
import xarray as xr # helper package to classify the raster data

raster_data_clipped_classified = xr.apply_ufunc(np.digitize, raster_data_clipped,
raster_data_clipped.quantile([0, 0.25, 0.8, 1]).values) #classify the data using given
quantile boundary 0 - 0.25, 0.25-0.8, 0.8 - 1, and > 1
fig, ax = plt.subplots(figsize=(10,10))
raster_data_clipped_classified.plot.imshow(ax=ax, cmap='coolwarm', add_colorbar = False)
toronto_FSA.boundary.plot(figsize=(10, 10), color = 'black', ax=ax, alpha=0.2)

# in default, legend in rasterdata is a color bar for continuous data, we will create a
custom legend
from matplotlib.lines import Line2D
colors = ['blue', 'purple', 'orange', 'red']
handles = [Line2D([0],[0],marker='o',color='w',label='0 - 0.25',
markerfacecolor=colors[0], markersize=10),
           Line2D([0],[0],marker='o',color='w',label='0.25 - 0.8',
markerfacecolor=colors[1], markersize=10),
           Line2D([0],[0],marker='o',color='w',label='0.8 - 1',markerfacecolor=colors[2],
markersize=10),
           Line2D([0],[0],marker='o',color='w',label='> 1', markerfacecolor=colors[3],
markersize=10)]

plt.legend(handles=handles, title='Temperature', title_fontsize='13', loc='upper right')
plt.title('City of Toronto Surface Temperature in July 2, 2020 (Celsius) By Quantile')
plt.axis('off')
plt.show()
```



Lab 11: Bayesian Modeling for Data Science

Frequentist perspective: Probabilities are long-run frequency of events. In other words, the probability of an event is the proportion of the times the event occurs if we repeated the experiment an infinite number of times.

```
# Function to simulate biased coin flips and plot a line graph of the proportion of heads
def biased_coin_flips(prob_heads, n_flips):

    # Simulate coin flips
    ## Generate random number between 0 and 1 n_flips times
    ## If the number is less than prob_heads, outcome of the flip is heads, otherwise
tails
    flips = np.random.rand(n_flips) < prob_heads # True (1) for heads, False (0) for tails

    # Calculate proportion of heads
    p_heads = np.cumsum(flips) / np.arange(1, n_flips + 1)

    # Plot the results
    plt.plot(p_heads)
    plt.plot(np.ones(n_flips) * prob_heads, 'k--')
    plt.xlabel('Number of flips')
    plt.ylabel('Proportion of heads')
    plt.show()

    # Return proportion of heads
    return p_heads[-1] # Last element of the array

# Set the probability of heads and the number of flips
prob_heads = 0.7
n_flips = 1000

# Call the function
p_est = biased_coin_flips(prob_heads, n_flips)

# Print the estimated probability
print('Estimated probability of heads:', p_est)
```

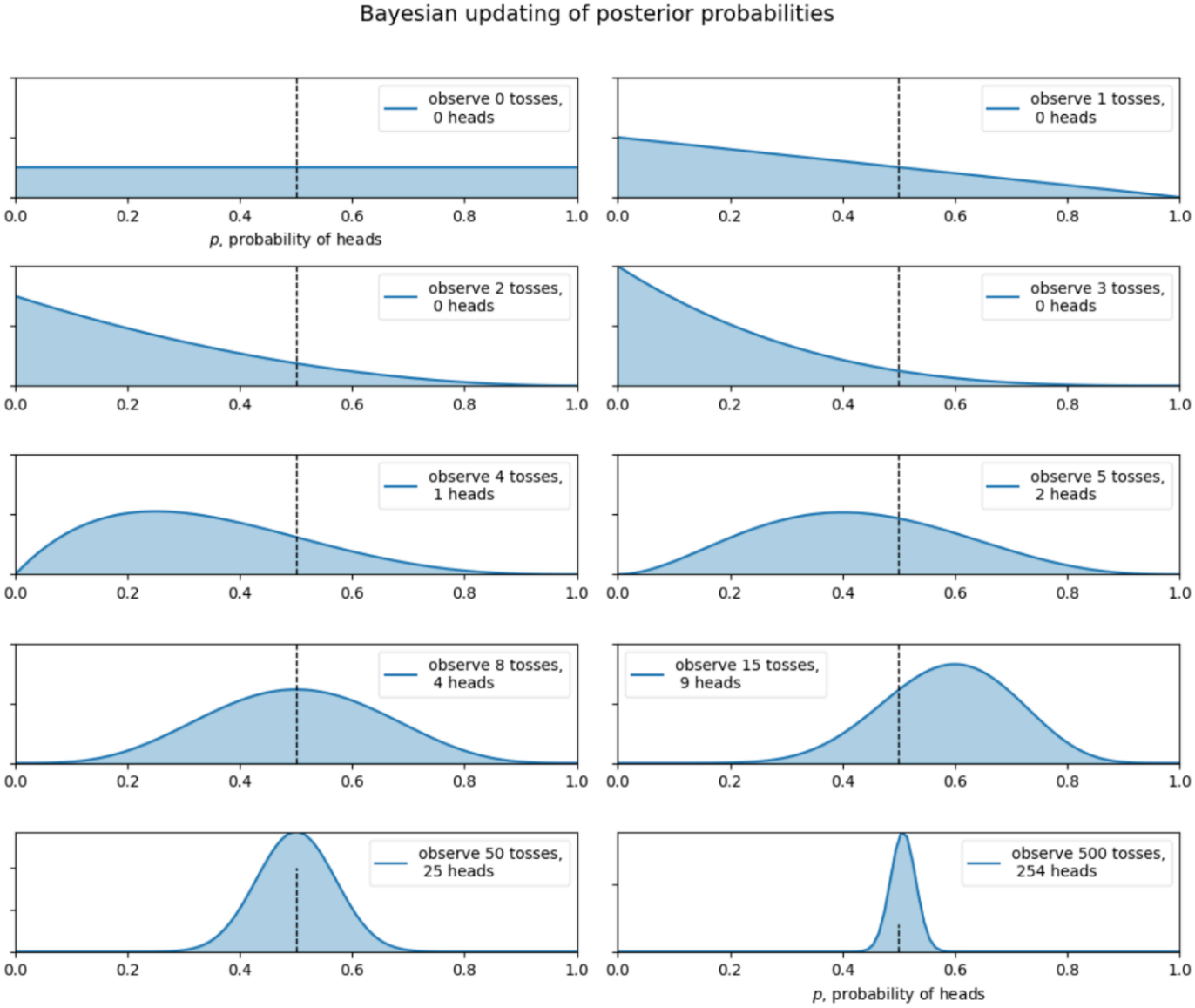



```
Estimated probability of heads: 0.704
```

Bayesian perspective: Probabilites are measures of belief or confidence of an event occurring. We possess prior beliefs about events, which are updated as we observe data. Initially, without conducting any experiments, it is fair to assume the prior distribution of p to be uniform. This means that all settings of p are equally likely.

Prior distribution: Beta distribution is the conjugate prior for Bernoulli trials.

$$P(p|\alpha,\beta) = \frac{1}{B(\alpha,\beta)} p^{\alpha-1}(1-p)^{\beta-1} \equiv Beta(p|\alpha,\beta)$$



Bayes' rule. The numerator is the "model".

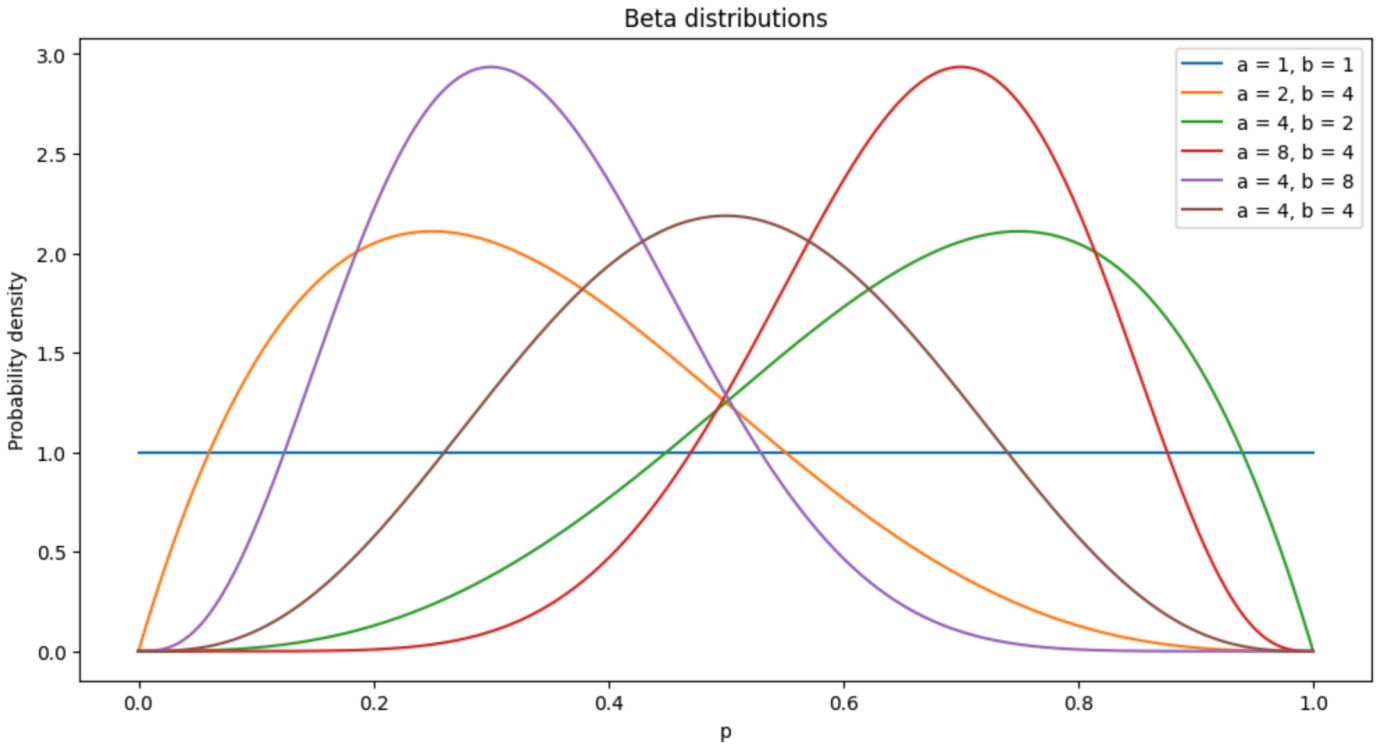
$$P(\theta|X) = \frac{P(X|\theta)P(\theta)}{P(X)}$$

Definitions of relevant variables/quantities:

- X: observation set, data, or evidence
- θ: set of model parameters
- $P(\theta)$: **prior** belief of model parameters without any evidence
- $P(\theta|X)$: **posterior** belief of model parameters given evidence
- $P(X|\theta)$: **likelihood** of data given model parameters
- $P(X)$: normalization constant (to ensure posterior distribution sums to 1)
  - Discrete case (model parameters are discrete): $\sum_{\theta \in \Theta} P(X|\theta)P(\theta)$
  - Continuous case (model parameters are continuous): $\int_{\theta \in \Theta} P(X|\theta)P(\theta)d\theta$
  - $\theta \in \Theta$ means to sum over all possible values of θ

Simulating the Beta distribution

```
# Function to simulate Beta distribution for different hyperparameters
def simulate_beta(a, b):

    # Generate 1000 points between 0 and 1
    x = np.linspace(0, 1, 1000)

    # Calculate the Beta distribution for the 1000 points
    y = stats.beta.pdf(x, a, b)

    # Return x and y
    return x, y

# List of hyperparameter settings (a, b)
hyperparameters = [(1, 1), (2, 4), (4, 2), (8, 4), (4, 8), (4, 4)]

# Plot Beta distributions on same plot
plt.figure(figsize=(12, 6))
for a, b in hyperparameters:
    x, y = simulate_beta(a, b)
    plt.plot(x, y, label=f'a = {a}, b = {b}')
plt.xlabel('p')
plt.ylabel('Probability density')
plt.title('Beta distributions')
plt.legend()
plt.show()
```

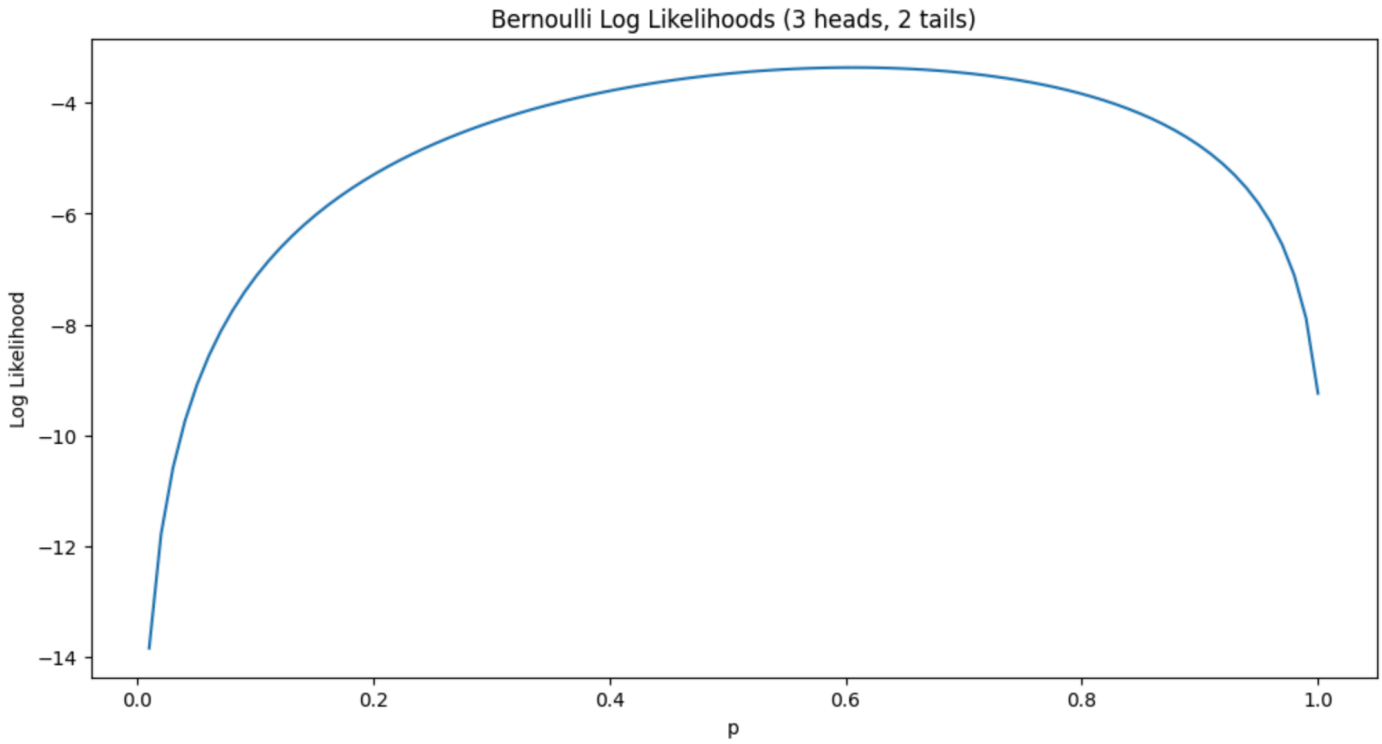


The key takeaway is that the hyperparameters influence the beliefs, and illustrates that we can define a prior that aligns with what we think the beliefs are. A breakdown of each prior in the plot above and the beliefs they convey is provided below.

- Blue ($\alpha = 1, \beta = 1$): belief that all p-values are equally likely (uniform prior)
- Orange ($\alpha = 2, \beta = 4$) + purple ($\alpha = 4, \beta = 8$): beliefs are biased towards lower values of p
- Green ($\alpha = 4, \beta = 2$) + red ($\alpha = 8, \beta = 4$): beliefs are biased towards higher values of p
- Brown ($\alpha = 4, \beta = 4$): beliefs are biased towards the coin being a fair coin ($p = 0.5$)

Simulating Bernoulli log-likelihood

```
# Function to compute Bernoulli log-likelihood for different p and outcomes
def simulate_bernoulli(p, outcomes):

    # Calculate the iid Bernoulli log likelihood for the outcomes
    y = np.sum(outcomes)*np.log(p) + np.sum(1-outcomes)*np.log(1-p)

    # Return log-likelihood
    return y

# Set the outcomes, 1 for heads and 0 for tails
outcomes = np.array([1, 0, 0, 1, 1])

# Calculate Bernoulli log likelihoods for different p values
likelihoods = []
for p in range(0, 100, 1):
    p /= 100
    likelihoods.append(simulate_bernoulli(p, outcomes))

# Plot Bernoulli log likelihoods for different p values
plt.figure(figsize=(12, 6))
plt.plot(np.linspace(0, 1, 100), likelihoods)
plt.xlabel('p')
plt.ylabel('Log Likelihood')
plt.title('Bernoulli Log Likelihoods (3 heads, 2 tails)')
plt.show()
```



Simulating Beta posterior

$$P(p|X) = Beta(p|h+\alpha, t+\beta) = \frac{1}{B(h+\alpha, t+\beta)} p^{h+\alpha-1}(1-p)^{t+\beta-1}$$

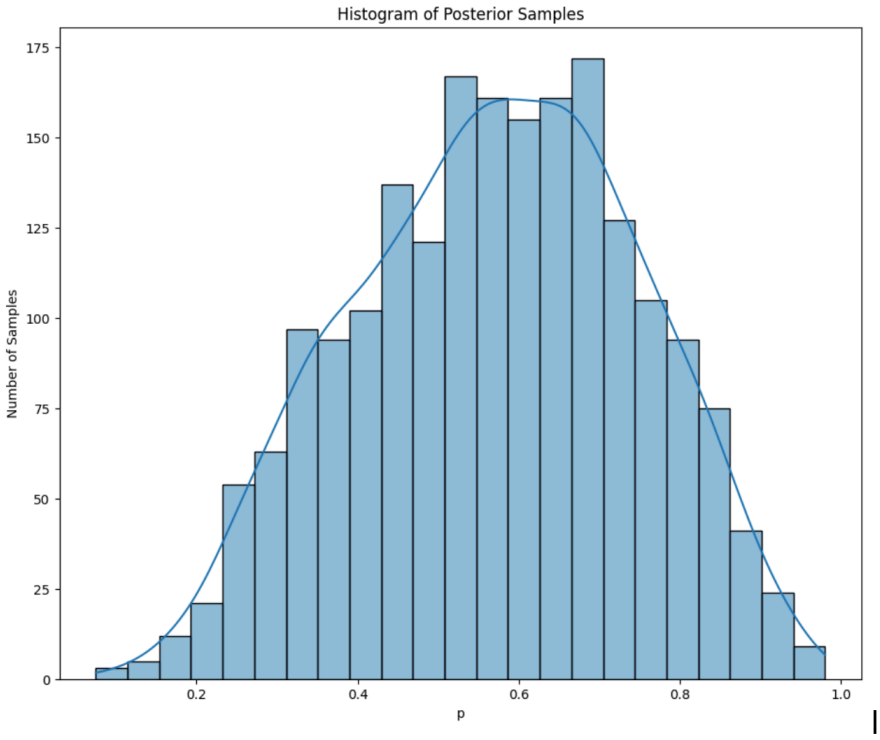
- Generative vs. Inference process:
  - Generative: formal definition of sampling from our model. Sampling is the process of randomly selecting possible events (or outcomes) of a random variable based on its probability distribution. The notation $x \sim P(X)$ means an event $X=x$ was sampled from $P(X)$.
  - Inference: computing the posterior beliefs provided the generative process and observed data samples. The inference process refers to computing the posterior beliefs, $p(\theta|X)$, provided the generative process and observed data samples. It's the application of Bayes' rule since access to the generative process implies access to definitions of prior distributions, $P(\theta)$, and the likelihood function, $P(X|\theta)$.

PyMC: Modeling Biased Coin Toss

```
# # PyMC implementation of biased coin toss

# Observed outcomes, 1 for heads and 0 for tails
outcomes = np.array([1, 0, 0, 1, 1])

# Define model
coin_flip_model = pm.Model()

with coin_flip_model:

    # Define priors
    p = pm.Beta('p', alpha=1, beta=1)
```

```
    # Define likelihood
    likelihood = pm.Bernoulli('likelihood', p=p, observed=outcomes)

    # Bayesian inference to approximate posterior (uses MCMC - outside of course scope)
    trace = pm.sample(1000, random_seed=100)

# Plot histogram of posterior samples
sns.histplot(trace.posterior['p'].values.flatten(), kde=True)
plt.xlabel('p')
plt.ylabel('Number of Samples')
plt.title('Histogram of Posterior Samples')

# Mean of posterior samples and 94% HDI
print(f"\nMean: {np.mean(trace.posterior['p'].values.flatten())}")
hdi = az.hdi(trace.posterior['p'].values.flatten())
print(f"94% HDI: {hdi}")
```

This following plot is a histogram of **samples** drawn from the **posterior**. This plot isn't as smooth as the previous PDFs because it is an approximation of the posterior (through sampling). The visualization is still useful to draw conclusion about posterior beliefs:



The posterior shows higher beliefs for values around p=0.6 (mean: p=0.57). This makes sense given 3/5 observations was heads. The 94% HDI (highest density interval) indicates that 94% of the samples (or probability density) is within the range $0.24 \leq p \leq 0.88$.

With more observed data samples (same ratio - 60% heads, 40% tails), the range of the 94% HDI decreases.

Switchpoints are used to represent **structural changes in the data generating process**—points where the statistical properties of the system change. They are particularly helpful when you suspect that something about your data changed at an unknown point in time, space, or another domain.

Use Case	Description
Change Point Detection	Identify when a time series (e.g., sales, temperature, failure rate) shifts behavior, like a new mean or variance.
Piecewise Modeling	Fit different models or parameters to different regions (e.g., before and after a switchpoint).
Hierarchical Models	Model changes in latent states or regime shifts (e.g., user behavior before vs. after a product change).
Intervention Analysis	Analyze the effect of a known or unknown intervention (policy change, marketing campaign, etc.) on some outcome.
Segmentation	Infer where different "regimes" begin and end in a signal or dataset (e.g., bull vs. bear markets).

```
# Double switchpoint model
# Order of switchpoints should be enforced
param_names = ['tau_1', 'tau_2', 'lambda_1', 'lambda_2', 'lambda_3']
double_switchpoint_model_adjusted = pm.Model()

with double_switchpoint_model_adjusted:

  # Define priors
  alpha = 1/np.mean(website_visits_data)
  tau_1 = pm.DiscreteUniform("tau_1", lower=0, upper=n_website_visits_data - 2)
  tau_2 = pm.DiscreteUniform("tau_2", lower=tau_1 + 1, upper=n_website_visits_data - 1)
  lambda_1 = pm.Exponential("lambda_1", alpha)
  lambda_2 = pm.Exponential("lambda_2", alpha)
  lambda_3 = pm.Exponential("lambda_3", alpha)
  idx = np.arange(n_website_visits_data)
  lambda_12 = pm.math.switch(idx < tau_1, lambda_1, lambda_2)
  lambda_ = pm.math.switch(idx < tau_2, lambda_12, lambda_3)

  # Define likelihood (conditioned on observed data)
  likelihood = pm.Poisson("obs", mu=lambda_, observed=website_visits_data)

  # Sample from posterior
  trace_adjusted = pm.sample(1000, random_seed=45, progressbar=False)
```

Latent Dirichlet Allocation (LDA) Topic Modelling

Latent Dirichlet Allocation (LDA): needs a document-term matrix and the number of topics estimated to be covered.

The input to the LDA algorithm is a set of documents. The output of LDA is a topic model that provides:

1. A list of topic IDs ti and the most probable words w per topic: $P(w|\text{t\_i})$
2. The probability that each topic t applies to a document d: $P(t|d)$

LDA Inference

The goal of inference is to:

1. Find $\vec{\theta}$ _ m for each document d_m.
2. Find $\vec{\psi}$ _ k for each topic z_k.

Note that the number of topics (k) must be pre-specified before running inference.

```
# Build the id2word dictionary and the corpus (map the word to id)
texts = wiki['text'].apply(lambda x: x.split(' ')).tolist()
dictionary = corpora.Dictionary(texts)
print('Number of unique tokens: ', len(dictionary))

# Remove stop words from a stop words set; merged from nltk and scikit-learn's built-in
list and words
stoplist = set(stopwords.words('english')).union(set(ENGLISH_STOP_WORDS))
stop_ids = [dictionary.token2id[stopword] for stopword in stoplist
            if stopword in dictionary.token2id]
dictionary.filter_tokens(stop_ids)

# Filter out words that appear in less than 2 documents (appear only once)
# There's also a no_above argument that we could specify, e.g. no_above = 0.5 would remove
words that appear in more than 50% of the documents
dictionary.filter_extremes(no_below = 2)

# Remove gaps in id sequence after words that were removed
dictionary.compactify()
print('Number of unique tokens: ', len(dictionary))

# Convert words to the "learned" word id
corpus = [dictionary.doc2bow(text) for text in texts]

# Training an LDA model
topic_model = LdaModel(corpus, id2word = dictionary, num_topics = 10, iterations = 200,
eval_every = None,                          random_state = 100, minimum_probability = 0)

# Top 10 most probable words per topic, topic indices start at 0
topics = topic_model.show_topics(num_words=10, formatted=False)

# Modify so topic index starts at 1 (to match upcoming visualization)
for i, topic in enumerate(topics):
    topics[i] = (topic[0]+1, topic[1])

# Topic distribution for article on Obama
obama = wiki.loc[wiki['name'] == 'Barack Obama', 'text'].tolist()[0].split()
obama_bow = topic_model.id2word.doc2bow(obama)
obama_doc_dist = topic_model[obama_bow]
for topic in obama_doc_dist:
    print(f"Topic {topic[0]+1}: {topic[1]}")

# Visualize with PyLDAvis
```

```
lda_topic_diagram = pyLDAvis.gensim_models.prepare(topic_model, corpus, dictionary,
sort_topics=False)
pyLDAvis.display(lda_topic_diagram)
lda_topic_diagram
```