Lab 4. Data Acquisition. Descriptive statistics.

Learning Objectives

Data acquisition and descriptive statistics on structured and unstructured datasets.

Evaluation – 7.5% of total course mark

Scoring Rubric

Category	Criteria	Maximum (Points)
Lab Examples	All python code is exceptionally well organized and very easy to follow. Make sure you provide comments for the examples.	**2**
Lab Questions	4 lab questions answered correctly, and python code is provided. Q1 – 0.8 points, Q2 – 1 point, Q3-4 - 0.6 points per question.	**3**

Use Jupyter notebook for this lab and include code, text, your comments, observations, and visualizations. Please make sure that the submitted notebooks have been run and the cell outputs are visible. Once created, submit it in pdf format.

Instructions

Data Acquisition

The pandas read_html() function is a quick and convenient way to turn an HTML table into a pandas DataFrame. This function can be useful for quickly incorporating tables from various websites without figuring out how to scrape the site's HTML. However, there can be some challenges in cleaning and formatting the data before analyzing it. In this lab, you will learn how to use pandas read_html() to read and clean several Wikipedia HTML tables so that you can use them for further numeric analysis.

For the first example, we will try to parse this table from the Politics section on the Minnesota wiki page.

The basic usage is of pandas read_html is pretty simple and works well on many Wikipedia pages since the tables are not complicated. To get started, I am including some extra imports we will use for data cleaning for more complicated examples:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from unicodedata import normalize

table_MN = pd.read_html('https://en.wikipedia.org/wiki/Minnesota')
table_MN
```

The unique point here is that table_MN is a list of all the tables on the page:

```
print(f'Total tables: {len(table_MN)}')
```

It can be challenging to find the one you need. To make the table selection easier, use the match parameter to select a subset of tables. We can use the caption "Election results from statewide races" to select the table:

```
table_MN = pd.read_html('https://en.wikipedia.org/wiki/Minnesota', match='Election
results from statewide races')
len(table_MN)
```

Pandas makes it easy to read in the table and also handles the year column that spans multiple rows. This is an example where it is easier to use pandas than to try to scrape it all yourself.

Overall, this looks ok until we look at the data types with df.info(). We need to convert the GOP, DFL and Other columns to numeric values if we want to do any analysis.

```
df = table_MN[0]
df.head()
```

If you try, you'll get an error

```
df['GOP'].astype('float')
```

The most likely culprit is the % . You can get rid of it using pandas replace() function.

```
df['GOP'].replace({'%':''}, regex=True).astype('float')
```

Note, that we had to use the regex=True parameter for this to work since the % is a part of the string and not the full string value. Now, we can call replace all the % values and convert to numbers using pd.to_numeric() and apply()

```
df = df.replace({'%': ''}, regex=True)
df[['GOP', 'DFL', 'Others']] = df[['GOP', 'DFL', 'Others']].apply(pd.to_numeric)
df.info()
```

```
df.head()
```

This basic process works well. The next example is a little trickier. The previous example showed the basic concepts. Frequently more cleaning is needed. Here is an example that was a little trickier. This example continues to use Wikipedia but the concepts apply to any site that has data in an HTML table.

What if we wanted to parse the US GDP table from https://en.wikipedia.org/wiki/Economy_of_the_United_States?

This one was a little harder to use match to get only one table but matching on 'Nominal GDP' gets the table we want as the first one in the list.

```
table_GDP =
pd.read_html('https://en.wikipedia.org/wiki/Economy_of_the_United_States',
match='Nominal GDP')
df_GDP = table_GDP[0]
df_GDP.info()
```

Not surprisingly we have some cleanup to do. We can try to remove the % like we did last time:

```
df_GDP['GDP growth(real)'].replace({'%': ''}, regex=True).astype('float')
```

Unfortunately we get this error: The issue here is that we have a hidden character, xa0 that is causing some errors. This is a “non-breaking Latin1 (ISO 8859-1) space”. One option is directly removing the value using replace. It won’t work with other characters in the future. The second option is to use normalize to clean this value. We’ll build a small function to clean all the text values:

```
from unicodedata import normalize

def clean_normalize_whitespace(x):
    if isinstance(x, str):
        return normalize('NFKC', x).strip()
    else:
        return x
```

We can run this function on the entire DataFrame using applymap. Be cautious about using applymap This function is very slow so you should be judicious in using it.

```
df_GDP = df_GDP.applymap(clean_normalize_whitespace)
```

The applymap function is a very inefficient pandas function. You should not use it very often but in this case, the DataFrame is small and cleaning like this is tricky so I think it is a useful trade-off. One thing that applymap misses is the column names. Let’s look at one column in more detail:

```
df_GDP.columns[7]
```

We have that dreaded xa0% in the column names. There are a couple of ways we could go about cleaning the columns but I’m going to use clean_normalize_whitespace() on the columns by converting the column to a series and using apply to run the function. Future versions of pandas may make this a little easier.

```
df_GDP.columns = df_GDP.columns.to_series().apply(clean_normalize_whitespace)
df_GDP.columns[7]
```

Now we have some of the hidden characters cleaned out. Let’s try it out again:

```
df_GDP['GDP growth(real)'].replace({'%': ''}, regex=True).astype('float')
```

This error is really tricky. If you look really closely, you might be able to tell that the − looks a little different than the - . It’s hard to see but there is actually a difference between the unicode dash and minus. Fortunately, we can use replace to clean that up too:

```
df_GDP['GDP growth(real)'].replace({'%': '', '−': '-'}, regex=True).astype('float')
```

One other column we need to look at is the Year column. For 2020, it contains “2020 (est)” which we want to get rid of. Then convert the column to an int. We can add to the dictionary but have to escape the parentheses since they are special characters in a regular expression:

```
df['Year'].replace({'%': '', '−': '-', '\(est\)': ''}, regex=True).astype('int')
```

Before we wrap it up and assign these values back to our DataFrame, there is one other item to discuss. Some of these columns should be integers and some are floats. If we use pd.numeric() we don’t have

that much flexibility. Using astype() we can control the numeric type but we don't want to have to manually type this for each column.

The astype() function can take a dictionary of column names and data types. This is really useful. Here is how we can define the column data type mapping:

```
col_type = {
    'Year': 'int',
    'Nominal GDP(billions USD)': 'float',
    'GDP per capita(USD)': 'float',
    'GDP growth(real)': 'float',
    'Inflation rate(in %)': 'float',
    'Unemployment(in %)': 'float',
    'Budget balance(in % of GDP)[116]': 'float',
    'Government debt held by public(in % of GDP)[117]': 'float',
    'Current accountbalance(in % of GDP)': 'float'
}
```

Here's a quick hint. Typing this dictionary is slow. Use this shortcut to build up a dictionary of the columns with float as the default value:

```
dict.fromkeys(df_GDP.columns, 'float')
```

We can also create a single dictionary with the values to replace:

```
clean_dict = {'%': '', '-': '-', '\(est\)': ''}
```

Now we can call replace on this DataFrame, convert to the desired type and get our clean numeric values:

```
df_GDP = df_GDP.replace(clean_dict, regex=True).replace({
    '-n/a ': np.nan
}).astype(col_type)
df_GDP.info()
```

Just to prove it works, we can plot the data too:

```
plt.style.use('seaborn-whitegrid')
df_GDP.plot.line(x='Year', y=['Inflation rate(in %)', 'Unemployment(in %)'])
```

Question 1. Find any other HTML data table online that potentially can be useful for your assignments project. Read it using read_html() function and apply appropriate cleaning.

There are special Python packages for data acquisition from different sources including social media:

Package Name	Description	Online Resources
tweepy	a Python library for accessing Twitter APIs	https://www.tweepy.org/
facebook-sdk	a Python library for accessing the Facebook APIs	https://facebook-sdk.readthedocs.io/en/latest/
python-instagram	a Python library for accessing the Instagram APIs	https://github.com/facebookarchive/python-instagram

Question 2. Implement an example of data retrieval using one of the packages listed in the table.

Descriptive statistics

Descriptive statistics are measures that summarize important features of data, often with a single number. Producing descriptive statistics is a common first step to take after cleaning and preparing a data set for analysis.

Measures of Center

Measures of center are statistics that give us a sense of the "middle" of a numeric variable. In other words, centrality measures give you a sense of a typical value you'd expect to see. Common measures of center include the mean, median and mode. The mean is simply an average: the sum of the values divided by the total number of records. As we've seen in previous lessons we can use df.mean() to get the mean of each column in a DataFrame:

```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
mtcars = pd.read_csv("../input/mtcars/mtcars.csv")
mtcars =  mtcars.rename(columns={'Unnamed: 0': 'model'})
mtcars.index = mtcars.model
del mtcars["model"]

mtcars.head()
```

```
mtcars.mean()                  # Get the mean of each column
```

We can also get the means of each row by supplying an axis argument:

```
mtcars.mean(axis=1)            # Get the mean of each row
```

The median of a distribution is the value where 50% of the data lies below it and 50% lies above it. In essence, the median splits the data in half. The median is also known as the 50% percentile since 50% of the observations are found below it. As we've seen previously, you can get the median using the df.median() function:

```
mtcars.median()                 # Get the median of each column
```

Again, we could get the row medians across each row by supplying the argument axis=1.

Although the mean and median both give us some sense of the center of a distribution, they aren't always the same. The median always gives us a value that splits the data into two halves while the mean is a numeric average so extreme values can have a significant impact on the mean. In a symmetric distribution, the mean and median will be the same. Let's investigate with a density plot:

```
norm_data = pd.DataFrame(np.random.normal(size=100000))

norm_data.plot(kind="density",
               figsize=(10,10));


plt.vlines(norm_data.mean(),     # Plot black line at mean
           ymin=0,
           ymax=0.4,
           linewidth=5.0);

plt.vlines(norm_data.median(),   # Plot red line at median
           ymin=0,
           ymax=0.4,
           linewidth=2.0,
           color="red");
```

In the you got the mean and median are both so close to zero that the red median line lies on top of the thicker black line drawn at the mean.

Question 3: Add legend to the plot with caption for all lines on the plot.

In skewed distributions, the mean tends to get pulled in the direction of the skew, while the median tends to resist the effects of skew:

```
skewed_data = pd.DataFrame(np.random.exponential(size=100000))

skewed_data.plot(kind="density",
                 figsize=(10,10),
                 xlim=(-1,5));


plt.vlines(skewed_data.mean(),     # Plot black line at mean
           ymin=0,
           ymax=0.8,
```

```
            linewidth=5.0);

plt.vlines(skewed_data.median(),    # Plot red line at median
           ymin=0,
           ymax=0.8,
           linewidth=2.0,
           color="red");
```

The mean is also influenced heavily by outliers, while the median resists the influence of outliers:

```
norm_data = np.random.normal(size=50)
outliers = np.random.normal(15, size=3)
combined_data = pd.DataFrame(np.concatenate((norm_data, outliers), axis=0))

combined_data.plot(kind="density",
              figsize=(10,10),
              xlim=(-5,20));


plt.vlines(combined_data.mean(),     # Plot black line at mean
           ymin=0,
           ymax=0.2,
           linewidth=5.0);

plt.vlines(combined_data.median(),    # Plot red line at median
           ymin=0,
           ymax=0.2,
           linewidth=2.0,
           color="red");
```

Since the median tends to resist the effects of skewness and outliers, it is known a "robust" statistic. The median generally gives a better sense of the typical value in a distribution with significant skew or outliers.

The mode of a variable is simply the value that appears most frequently. Unlike mean and median, you can take the mode of a categorical variable and it is possible to have multiple modes. Find the mode with df.mode():

```
mtcars.mode()
```

The columns with multiple modes (multiple values with the same count) return multiple values as the mode. Columns with no mode (no value that appears more than once) return NaN.

Measures of Spread

Measures of spread (dispersion) are statistics that describe how data varies. While measures of center give us an idea of the typical value, measures of spread give us a sense of how much the data tends to diverge from the typical value. One of the simplest measures of spread is the range. Range is the distance between the maximum and minimum observations:

```
max(mtcars["mpg"]) - min(mtcars["mpg"])
```

As noted earlier, the median represents the 50th percentile of a data set. A summary of several percentiles can be used to describe a variable's spread. We can extract the minimum value (0th percentile), first quartile (25th percentile), median, third quartile(75th percentile) and maximum value (100th percentile) using the quantile() function:

```
five_num = [mtcars["mpg"].quantile(0),
            mtcars["mpg"].quantile(0.25),
            mtcars["mpg"].quantile(0.50),
            mtcars["mpg"].quantile(0.75),
            mtcars["mpg"].quantile(1)]

five_num
```

Since these values are so commonly used to describe data, they are known as the "five number summary". They are the same percentile values returned by df.describe():

```
mtcars["mpg"].describe()
```

Interquartile (IQR) range is another common measure of spread. IQR is the distance between the 3rd quartile and the 1st quartile:

```
mtcars["mpg"].quantile(0.75) - mtcars["mpg"].quantile(0.25)
```

The boxplots we learned to create in the lesson on plotting are just visual representations of the five number summary and IQR:

```
mtcars.boxplot(column="mpg",
               return_type='axes',
               figsize=(8,8))
```

Question 4: Plot boxplots for all numerical columns in the dataset and provide your comments on what you observe.

Variance and standard deviation are two other common measures of spread. The variance of a distribution is the average of the squared deviations (differences) from the mean. Use df.var() to check variance:

```
mtcars["mpg"].var()
```

The standard deviation is the square root of the variance. Standard deviation can be more interpretable than variance, since the standard deviation is expressed in terms of the same units as the variable in question while variance is expressed in terms of units squared. Use df.std() to check the standard deviation:

```
mtcars["mpg"].std()
```

Since variance and standard deviation are both derived from the mean, they are susceptible to the influence of data skew and outliers. Median absolute deviation is an alternative measure of spread based on the median, which inherits the median's robustness against the influence of skew and outliers. It is the median of the absolute value of the deviations from the median:

```
abs_median_devs = abs(mtcars["mpg"] - mtcars["mpg"].median())

abs_median_devs.median() * 1.4826
```

Note: The MAD is often multiplied by a scaling factor of 1.4826.

Skewness and Kurtosis

Beyond measures of center and spread, descriptive statistics include measures that give you a sense of the shape of a distribution. Skewness measures the skew or asymmetry of a distribution while kurtosis measures how much data is in the tails of a distribution vs. the center. We won't go into the exact calculations behind skewness and kurtosis, but they are essentially just statistics that take the idea of variance a step further: while variance involves squaring deviations from the mean, skewness involves cubing deviations from the mean and kurtosis involves raising deviations from the mean to the 4th power.

Pandas has built in functions for checking skewness and kurtosis, df.skew() and df.kurt() respectively:

```
mtcars["mpg"].skew()  # Check skewness
```

```
mtcars["mpg"].kurt()  # Check kurtosis
```

To explore these two measures further, let's create some dummy data and inspect it:

```
norm_data = np.random.normal(size=100000)
skewed_data = np.concatenate((np.random.normal(size=35000)+2,
                              np.random.exponential(size=65000)),
                              axis=0)
uniform_data = np.random.uniform(0,2, size=100000)
peaked_data = np.concatenate((np.random.exponential(size=50000),
                              np.random.exponential(size=50000)*(-1)),
                              axis=0)

data_df = pd.DataFrame({"norm":norm_data,
                        "skewed":skewed_data,
                        "uniform":uniform_data,
                        "peaked":peaked_data})
```

```
data_df.plot(kind="density",
             figsize=(10,10),
             xlim=(-5,5));
```

Now let's check the skewness of each of the distributions. Since skewness measures asymmetry, we'd expect to see low skewness for all of the distributions except the skewed one, because all the others are roughly symmetric:

```
data_df.skew()
```

Now let's check kurtosis.

```
data_df.kurt()
```

As we can see from the output, the normally distributed data has a kurtosis near zero, the flat distribution has negative kurtosis and the two distributions with more data in the tails vs the center have higher kurtosis.

Descriptive statistics help you explore features of your data, like center, spread and shape by summarizing them with numerical measurements. Descriptive statistics help inform the direction of an analysis and let you communicate your insights to others quickly and succinctly. In addition, certain values, like the mean and variance, are used in all sorts of statistical tests and predictive models.