Analysis of Real Estate Sales in Connecticut

Collaborators: Arin Zeng, Helen Gu, Jennifer Li, Jason Chen

Introduction

The main purpose of this project is to take you through the entire pipeline of a data science journey. We will focus on how unemployment rates, interest rates, location, among other factors, affect the sales ratio of real estate homes in Connecticut. Amidst a nation where the population is rapidly growing, homes and other real-estate properties are increasing in demand while becoming a more popular investment area. The real-estate market can appear to be a world of mystery, and intimidating to navigate. As a home-buyer, you want to understand the factors to weigh in to purchase or invest in real-estate; as a home-seller, you want to understand the factors to weigh in to sell your real-estate. We want to appeal to these two audiences in particular, as we work to uncover the mystery surrounding the real-estate market. We will be focusing on data sets specific to Connecticut, but our tutorial will easily be applicable to studying the real-estate market in other states.

The purpose of this tutorial is to analyze factors that affect sales ratio in the real-estate market, and examine its relationship with real-estate sales and profits. We will explore various methods of graphing and visualizing possible relationships and trends in the data to better understand the factors that impact the real-estate market. Data science is a perfect tool for this task because it allows us to easily unpack and organize messy and complicated data to synthesize more digestible descriptions of the real-estate market that market participants can refer to in their daily decisions.

Imports

```
# import necessary libraries
import pandas as pd
from sodapy import Socrata
import requests
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import folium
from plotnine import *
```

Data Collection: Real Estate Sales

The data science pipeline begins with collecting data for analysis. We first extract data from a dataset of Real estate sales in Connecticut from the years 2001-2020, provided by

Connecticut Open Data. This data provides insight into the sales prices of homes, what their assessed value was, when they were sold, what the sales ratio was, and more. To retrieve the data, we use an api client to get the records and store them into a dataframe for further processing.

!! Only run once !! !! Start !!

```
# Use the library to set up api client connection with the Connecticut
Open Data website
client = Socrata("data.ct.gov", None)

# Returns a JSON of records retrieved from the api up to 1000000
records since the
# dataset has a total of 997k records
results = client.get("5mzw-sjtu", limit=1000000)

# Convert to pandas DataFrame
results_df = pd.DataFrame.from_records(results)
results_df.head()
```

```
WARNING:root:Requests made without an app_token will be subject to
strict throttling limits.

---------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
Cell In[273], line 6
      2 client = Socrata("data.ct.gov", None)
      4 # Returns a JSON of records retrieved from the api up to
1000000 records since the
      5 # dataset has a total of 997k records
----> 6 results = client.get("5mzw-sjtu", limit=1000000)
      8 # Convert to pandas DataFrame
      9 results_df = pd.DataFrame.from_records(results)

File
~/Documents/cmsc320/lib/python3.10/site-packages/sodapy/socrata.py:412
, in Socrata.get(self, dataset_identifier, content_type, **kwargs)
    409 params.update(kwargs)
    410 params = utils.clear_empty_values(params)
--> 412 response = self._perform_request(
    413     "get", resource, headers=headers, params=params
    414 )
    415 return response

File
~/Documents/cmsc320/lib/python3.10/site-packages/sodapy/socrata.py:551
, in Socrata._perform_request(self, request_type, resource, **kwargs)
    548 # set a timeout, just to be safe
```

```
    549 kwargs["timeout"] = self.timeout
--> 551 response = getattr(self.session, request_type)(uri, **kwargs)
    553 # handle errors
    554 if response.status_code not in (200, 202):

File
~/Documents/cmsc320/lib/python3.10/site-packages/requests/sessions.py:
600, in Session.get(self, url, **kwargs)
    592 r"""Sends a GET request. Returns :class:`Response` object.
    593
    594 :param url: URL for the new :class:`Request` object.
    595 :param \*\*kwargs: Optional arguments that ``request`` takes.
    596 :rtype: requests.Response
    597 """
    599 kwargs.setdefault("allow_redirects", True)
--> 600 return self.request("GET", url, **kwargs)

File
~/Documents/cmsc320/lib/python3.10/site-packages/requests/sessions.py:
587, in Session.request(self, method, url, params, data, headers,
cookies, files, auth, timeout, allow_redirects, proxies, hooks,
stream, verify, cert, json)
    582 send_kwargs = {
    583     "timeout": timeout,
    584     "allow_redirects": allow_redirects,
    585 }
    586 send_kwargs.update(settings)
--> 587 resp = self.send(prep, **send_kwargs)
    589 return resp

File
~/Documents/cmsc320/lib/python3.10/site-packages/requests/sessions.py:
701, in Session.send(self, request, **kwargs)
    698 start = preferred_clock()
    700 # Send the request
--> 701 r = adapter.send(request, **kwargs)
    703 # Total elapsed time of the request (approximately)
    704 elapsed = preferred_clock() - start

File
~/Documents/cmsc320/lib/python3.10/site-packages/requests/adapters.py:
486, in HTTPAdapter.send(self, request, stream, timeout, verify, cert,
proxies)
    483     timeout = TimeoutSauce(connect=timeout, read=timeout)
    485 try:
--> 486     resp = conn.urlopen(
    487         method=request.method,
    488         url=url,
    489         body=request.body,
    490         headers=request.headers,
```

```
    491         redirect=False,
    492         assert_same_host=False,
    493         preload_content=False,
    494         decode_content=False,
    495         retries=self.max_retries,
    496         timeout=timeout,
    497         chunked=chunked,
    498     )
    500 except (ProtocolError, OSError) as err:
    501     raise ConnectionError(err, request=request)

File
~/Documents/cmsc320/lib/python3.10/site-packages/urllib3/connectionpoo
l.py:790, in HTTPConnectionPool.urlopen(self, method, url, body,
headers, retries, redirect, assert_same_host, timeout, pool_timeout,
release_conn, chunked, body_pos, preload_content, decode_content,
**response_kw)
    787 response_conn = conn if not release_conn else None
    789 # Make the request on the HTTPConnection object
--> 790 response = self._make_request(
    791     conn,
    792     method,
    793     url,
    794     timeout=timeout_obj,
    795     body=body,
    796     headers=headers,
    797     chunked=chunked,
    798     retries=retries,
    799     response_conn=response_conn,
    800     preload_content=preload_content,
    801     decode_content=decode_content,
    802     **response_kw,
    803 )
    805 # Everything went great!
    806 clean_exit = True

File
~/Documents/cmsc320/lib/python3.10/site-packages/urllib3/connectionpoo
l.py:536, in HTTPConnectionPool._make_request(self, conn, method, url,
body, headers, retries, timeout, chunked, response_conn,
preload_content, decode_content, enforce_content_length)
    534 # Receive the response from the server
    535 try:
--> 536     response = conn.getresponse()
    537 except (BaseSSLError, OSError) as e:
    538     self._raise_timeout(err=e, url=url,
timeout_value=read_timeout)

File
~/Documents/cmsc320/lib/python3.10/site-packages/urllib3/connection.py
```

```
:454, in HTTPConnection.getresponse(self)
    451 from .response import HTTPResponse
    453 # Get the response from http.client.HTTPConnection
--> 454 httplib_response = super().getresponse()
    456 try:
    457     assert_header_parsing(httplib_response.msg)

File
/nix/store/pg1za7jn6cl9nlkvg51hclwxyiswiq34-python3-3.10.7/lib/python3
.10/http/client.py:1374, in HTTPConnection.getresponse(self)
   1372 try:
   1373     try:
-> 1374         response.begin()
   1375     except ConnectionError:
   1376         self.close()

File
/nix/store/pg1za7jn6cl9nlkvg51hclwxyiswiq34-python3-3.10.7/lib/python3
.10/http/client.py:318, in HTTPResponse.begin(self)
    316 # read until we get a non-100 response
    317 while True:
--> 318     version, status, reason = self._read_status()
    319     if status != CONTINUE:
    320         break

File
/nix/store/pg1za7jn6cl9nlkvg51hclwxyiswiq34-python3-3.10.7/lib/python3
.10/http/client.py:279, in HTTPResponse._read_status(self)
    278 def _read_status(self):
--> 279     line = str(self.fp.readline(_MAXLINE + 1), "iso-8859-1")
    280     if len(line) > _MAXLINE:
    281         raise LineTooLong("status line")

File
/nix/store/pg1za7jn6cl9nlkvg51hclwxyiswiq34-python3-3.10.7/lib/python3
.10/socket.py:705, in SocketIO.readinto(self, b)
    703 while True:
    704     try:
--> 705         return self._sock.recv_into(b)
    706     except timeout:
    707         self._timeout_occurred = True

File
/nix/store/pg1za7jn6cl9nlkvg51hclwxyiswiq34-python3-3.10.7/lib/python3
.10/ssl.py:1274, in SSLSocket.recv_into(self, buffer, nbytes, flags)
   1270     if flags != 0:
   1271         raise ValueError(
   1272           "non-zero flags not allowed in calls to recv_into()
on %s" %
   1273           self.__class__)
```

```
-> 1274     return self.read(nbytes, buffer)
   1275 else:
   1276     return super().recv_into(buffer, nbytes, flags)

File
/nix/store/pg1za7jn6cl9nlkvg51hclwxyiswiq34-python3-3.10.7/lib/python3
.10/ssl.py:1130, in SSLSocket.read(self, len, buffer)
   1128 try:
   1129     if buffer is not None:
-> 1130         return self._sslobj.read(len, buffer)
   1131     else:
   1132         return self._sslobj.read(len)

KeyboardInterrupt:
```

Since the original dataset consists of over 990k data points, the data takes much too long to filter and process, so we can take a sampling of the data by selecting every hundredth row in order to reduce the size of the dataset. Then the data is stored into a csv for ease of future analysis.

```python
# take every hundredth row in the dataframe and save to a csv
df = results_df.iloc[::100]
df.to_csv("conn_real_estate_sample.csv", index=False)
```

!! End !!

```python
# read data into the dataframe from the csv file
results_df = pd.read_csv("data/conn_real_estate_sample.csv")
results_df
```

```
      serialnumber  listyear              daterecorded       town  
0            20001      2020  2020-10-05T00:00:00.000    Andover  \
1          2020072      2020  2020-11-25T00:00:00.000    Ansonia   
2          2020221      2020  2021-05-25T00:00:00.000    Ansonia   
3          2020361      2020  2021-09-21T00:00:00.000    Ansonia   
4           200047      2020  2020-10-23T00:00:00.000       Avon   
...            ...       ...                      ...        ...   
7071         10011      2001  2001-10-05T00:00:00.000   Woodbury   
7072         10239      2001  2002-04-02T00:00:00.000   Woodbury   
7073         10158      2001  2002-07-19T00:00:00.000   Woodbury   
7074         10067      2001  2001-12-31T00:00:00.000  Woodstock   
7075         10211      2001  2002-08-07T00:00:00.000  Woodstock   

                  address  assessedvalue  saleamount  salesratio  
propertytype
0             303 LAKE RD         121300    210000.0    0.577600  
Residential  \
1       7 JASON WRIGHT DR         172500    330000.0    0.522700  
Residential
2          8 BERKSHIRE RD         108300    245000.0    0.442000  
Residential
```

```
3          5 BIRCHWOOD DR         203700    335000.0    0.608000
Residential
4          21 CLIFF DRIVE         155390    249000.0    0.624000
Residential
...                   ...            ...         ...         ...
...
7071     57 OLD GRASSY HL         190630    418962.0    0.455005
NaN
7072  86 29 WASHINGTON RD          36580     80000.0    0.457250
NaN
7073       58 SHERMAN HTS          96370    240000.0    0.401542
NaN
7074     12 WAINWRIGHT DR          67930    146000.0    0.465274
NaN
7075           564 RT 198         117550    199500.0    0.589223
NaN

     residentialtype
geo_coordinates
0      Single Family  {'type': 'Point', 'coordinates': [-72.35327,
4...  \
1      Single Family
NaN
2      Single Family
NaN
3      Single Family
NaN
4      Single Family  {'type': 'Point', 'coordinates': [-72.87619,
4...
...              ...                                                ..
.
7071             NaN
NaN
7072             NaN
NaN
7073             NaN
NaN
7074             NaN
NaN
7075             NaN  {'type': 'Point', 'coordinates': [-72.07593,
4...

      :@computed_region_dam5_q64j  :@computed_region_nhmp_cq6b
0                            38.0                        246.0  \
1                             NaN                          NaN
2                             NaN                          NaN
3                             NaN                          NaN
4                          1041.0                         46.0
...                           ...                          ...
7071                          NaN                          NaN
```

```
7072                            NaN                          NaN
7073                            NaN                          NaN
7074                            NaN                          NaN
7075                           39.0                         72.0

      :@computed_region_m4y2_whse  :@computed_region_snd5_k6zv
nonusecode
0                             1.0                          1.0
NaN  \
1                             NaN                          NaN
NaN
2                             NaN                          NaN
NaN
3                             NaN                          NaN
NaN
4                             4.0                          1.0
NaN
...                           ...                          ...
...
7071                          NaN                          NaN
NaN
7072                          NaN                          NaN
NaN
7073                          NaN                          NaN
NaN
7074                          NaN                          NaN
NaN
7075                        169.0                          5.0
NaN

     remarks opm_remarks
0        NaN         NaN
1        NaN         NaN
2        NaN         NaN
3        NaN         NaN
4        NaN         NaN
...      ...         ...
7071     NaN         NaN
7072     NaN         NaN
7073     NaN         NaN
7074     NaN         NaN
7075     NaN         NaN

[7076 rows x 18 columns]
```

Data Processing: Real Estate Sales

Now that we have the data, our next step is to clean up our data in order to get rid of unnecessary columns or rows, as well as extraneous values. Looking at the table, there are multiple columns that don't quite seem to make sense as well as columns that have little effect on our analysis such as serial number, computed regions, and remarks. So we can go ahead and drop these columns from our dataframe since we don't need them going forward. Within some columns, there is also data where all the information is included, but it is formatted in a way that is difficult for us to process or includes extra data. For example, the date recorded field includes the time of the record as well; however it appears to be 00:00:00 for every entry and since it is the date that we are looking at, the time is not pertinent to this field, so we can strip the time from the date field to clean it up. We also separate out the year and the month into new fields of their own for more general analysis.

```
# drop unnecessary columns
df = results_df.drop(columns = ['serialnumber',
                      ':@computed_region_dam5_q64j',
':@computed_region_nhmp_cq6b',
                      ':@computed_region_m4y2_whse',
':@computed_region_snd5_k6zv',
                     'remarks', 'opm_remarks'])

# strip time from date string
df['daterecorded'] = df['daterecorded'].str[:10]

# separate date into two fields and convert into datetime objects
df['Year'] = pd.to_datetime(df['daterecorded'], format="%Y-%m-
%d").dt.strftime('%Y')
df['Month'] = pd.to_datetime(df['daterecorded'], format="%Y-%m-
%d").dt.strftime('%m')
df
```

```
      listyear daterecorded       town              address
assessedvalue
0         2020   2020-10-05    Andover          303 LAKE RD
121300  \
1         2020   2020-11-25    Ansonia    7 JASON WRIGHT DR
172500
2         2020   2021-05-25    Ansonia       8 BERKSHIRE RD
108300
3         2020   2021-09-21    Ansonia       5 BIRCHWOOD DR
203700
4         2020   2020-10-23       Avon       21 CLIFF DRIVE
155390
...        ...          ...        ...                  ...
...
7071      2001   2001-10-05   Woodbury     57 OLD GRASSY HL
190630
7072      2001   2002-04-02   Woodbury  86 29 WASHINGTON RD
36580
7073      2001   2002-07-19   Woodbury       58 SHERMAN HTS
96370
7074      2001   2001-12-31  Woodstock     12 WAINWRIGHT DR
67930
7075      2001   2002-08-07  Woodstock           564 RT 198
```

```
117550

      saleamount  salesratio propertytype residentialtype  
0       210000.0    0.577600  Residential   Single Family  \
1       330000.0    0.522700  Residential   Single Family   
2       245000.0    0.442000  Residential   Single Family   
3       335000.0    0.608000  Residential   Single Family   
4       249000.0    0.624000  Residential   Single Family   
...          ...         ...          ...             ...   
7071    418962.0    0.455005          NaN             NaN   
7072     80000.0    0.457250          NaN             NaN   
7073    240000.0    0.401542          NaN             NaN   
7074    146000.0    0.465274          NaN             NaN   
7075    199500.0    0.589223          NaN             NaN   

                                        geo_coordinates  nonusecode  
Year  
0     {'type': 'Point', 'coordinates': [-72.35327, 4...         NaN  
2020  \
1                                                   NaN         NaN  
2020  
2                                                   NaN         NaN  
2021  
3                                                   NaN         NaN  
2021  
4     {'type': 'Point', 'coordinates': [-72.87619, 4...         NaN  
2020  
...                                                 ...         ...  
...  
7071                                                NaN         NaN  
2001  
7072                                                NaN         NaN  
2002  
7073                                                NaN         NaN  
2002  
7074                                                NaN         NaN  
2001  
7075  {'type': 'Point', 'coordinates': [-72.07593, 4...         NaN  
2002  

     Month  
0       10  
1       11  
2       05  
3       09  
4       10  
...    ...  
7071    10  
7072    04  
7073    07  
```

```
7074    12
7075    08

[7076 rows x 13 columns]
```

Another step we can take is to separate out the x and y coordinates for the geo coordinates field. The geo coordinates field also has encoded all necessary information; however, the formatting of the field makes it difficult for us to directly examine the coordinate information. So in order to extract the coordinate information, we can use string operations to extract the x and y coordinates, which are placed into their own respective fields. This way, we can better examine where the coordinates of a specific record belong.

```
# extract the coordinates from the geo coordinates field and separate
into x and y coordinates
df['coord'] = df['geo_coordinates'].astype(str).str.extract('\
[(.*?)\]', expand=False).str.strip()
df[['x_coord','y_coord']] = df.coord.str.split(",
",expand=True).astype(float)
df = df.drop(columns = ['geo_coordinates', 'coord'])
df.head()
```

```
   listyear daterecorded     town            address  assessedvalue  \
0      2020   2020-10-05  Andover        303 LAKE RD         121300   
1      2020   2020-11-25  Ansonia  7 JASON WRIGHT DR         172500   
2      2020   2021-05-25  Ansonia     8 BERKSHIRE RD         108300   
3      2020   2021-09-21  Ansonia     5 BIRCHWOOD DR         203700   
4      2020   2020-10-23     Avon     21 CLIFF DRIVE         155390   

   saleamount  salesratio propertytype residentialtype  nonusecode 
Year  
0    210000.0      0.5776  Residential   Single Family         NaN 
2020  \
1    330000.0      0.5227  Residential   Single Family         NaN 
2020   
2    245000.0      0.4420  Residential   Single Family         NaN 
2021   
3    335000.0      0.6080  Residential   Single Family         NaN 
2021   
4    249000.0      0.6240  Residential   Single Family         NaN 
2020   

  Month   x_coord   y_coord  
0    10 -72.35327  41.71416  
1    11       NaN       NaN  
2    05       NaN       NaN  
3    09       NaN       NaN  
4    10 -72.87619  41.80986  
```

An important part of examining the data is also to look at the distribution of data to make sure that our data has a good distribution of records and also doesn't include any extraneous values. This we can see in a summary of the dataframe seen below. Most of the data appears to be normal; however, in our sales ratio, there is clearly a max value that lies far beyond the range of the other sales ratio values. We can confirm that it is an outlier by calculating the third quartile + (1.5 * IQR) which gives us an upper fence where outliers would lie beyond. The third quartile + (1.5 * IQR) is equal to 0.70 + 1.5(0.70 - 0.49) = 1.015, so it is evident that this data point, the max of 7.9 is an outlier and we can remove it from the dataset and also use this upper fence to filter out unreasonably high values.

```
# display a summary of the dataframe
df.describe()
```

```
          listyear  assessedvalue    saleamount   salesratio
nonusecode
count  7076.000000   7.076000e+03  7.076000e+03  7076.000000
0.0  \
mean   2009.954211   2.306462e+05  3.838914e+05     0.606059
NaN
std       6.366022   4.231461e+05  7.206351e+05     0.195015
NaN
min    2001.000000   0.000000e+00  0.000000e+00     0.000000
NaN
25%    2004.000000   9.281500e+04  1.670000e+05     0.485414
NaN
50%    2009.000000   1.449950e+05  2.470000e+05     0.597535
NaN
75%    2016.000000   2.329325e+05  3.821250e+05     0.704150
NaN
max    2020.000000   1.540000e+07  2.489893e+07     7.915833
NaN

           x_coord      y_coord
count  1403.000000  1403.000000
mean    -72.887949    41.485677
std       0.428388     0.258001
min     -73.671740    41.012320
25%     -73.221735    41.280930
50%     -72.907140    41.483920
75%     -72.634795    41.709425
max     -71.810420    42.033330
```

```
# remove sales ratio outliers
df = df[df['salesratio'] < 1.015]
df
```

```
      listyear daterecorded       town              address
assessedvalue
0         2020   2020-10-05    Andover          303 LAKE RD
121300  \
1         2020   2020-11-25    Ansonia    7 JASON WRIGHT DR
172500
```

```
2          2020   2021-05-25    Ansonia       8 BERKSHIRE RD
108300
3          2020   2021-09-21    Ansonia       5 BIRCHWOOD DR
203700
4          2020   2020-10-23       Avon       21 CLIFF DRIVE
155390
...         ...          ...        ...                  ...
...
7071       2001   2001-10-05   Woodbury     57 OLD GRASSY HL
190630
7072       2001   2002-04-02   Woodbury  86 29 WASHINGTON RD
36580
7073       2001   2002-07-19   Woodbury       58 SHERMAN HTS
96370
7074       2001   2001-12-31  Woodstock     12 WAINWRIGHT DR
67930
7075       2001   2002-08-07  Woodstock           564 RT 198
117550

      saleamount  salesratio propertytype residentialtype  nonusecode
Year
0       210000.0    0.577600  Residential   Single Family         NaN
2020  \
1       330000.0    0.522700  Residential   Single Family         NaN
2020
2       245000.0    0.442000  Residential   Single Family         NaN
2021
3       335000.0    0.608000  Residential   Single Family         NaN
2021
4       249000.0    0.624000  Residential   Single Family         NaN
2020
...          ...         ...          ...             ...         ...
...
7071    418962.0    0.455005          NaN             NaN         NaN
2001
7072     80000.0    0.457250          NaN             NaN         NaN
2002
7073    240000.0    0.401542          NaN             NaN         NaN
2002
7074    146000.0    0.465274          NaN             NaN         NaN
2001
7075    199500.0    0.589223          NaN             NaN         NaN
2002

     Month   x_coord   y_coord
0       10 -72.35327  41.71416
1       11       NaN       NaN
2       05       NaN       NaN
3       09       NaN       NaN
4       10 -72.87619  41.80986
```

```
...    ...        ...       ...
7071    10        NaN       NaN
7072    04        NaN       NaN
7073    07        NaN       NaN
7074    12        NaN       NaN
7075    08 -72.07593  41.95063

[6954 rows x 14 columns]
```

Data Collection: Interest Rates

In order to better gauge factors that affect real estate sales, we can take a look at historical interest rates and how they fit in with our real estate data. Since interest rates commonly affect the market and how people buy and sell, it can be useful to examine interest rates in relation to the real estate sales to determine what kind of relationship exists interest rates and real estate sales.

Another method of acquiring data is to scrape it from websites. Since First Republic provides a table of historical interest rates on their website, we can scrape the interest rates off their website using webscraping libraries like selenium and BeautifulSoup. Once the html from the page has been scraped, we can then parse the html to identify the table and pick out the data that we need from the table. Web scraping is a useful technique for gathering data from websites online where the dataset may not be available to download and is commonly used for data collection. Check out more information on webscraping using selenium and BeautifulSoup here.

```
# url of the page we are scraping
url = "https://www.firstrepublic.com/finmkts/historical-interest-
rates"

# initiating the webdriver. Parameter includes the path of the
webdriver.
driver = webdriver.Chrome('./chromedriver')
driver.get(url)

# ensures that the page is loaded
time.sleep(5)

# gets the html of the page
html = driver.page_source

# apply bs4 to html variable
soup = BeautifulSoup(html, "html.parser")
```

```
C:\Users\helen\AppData\Local\Temp\ipykernel_14968\1535766723.py:4:
DeprecationWarning: executable_path has been deprecated, please pass
in a Service object
```

```
# find all table tag in the html
table = soup.find_all("table")
```

```
# take the third table tag and isolate only the necessary data and
divide
# the data into rows
arr = str(table[2]).split("<tbody>")
arr = arr[1].split("\n")
arr = arr[-2].split("<tr>")
arr = arr[1:]

# make a dataframe
df1 = pd.DataFrame(arr)

# iterate over the dataframe
for i, row in df1.iterrows():
    # split the entries
    arr = row[0].split("<td>")

    # get the month/year value
    date = arr[1][:-5]

    # break if outside of our date range
    if date[-4:] == '2000':
        break

    # get the interest rate
    prime_rate = float(arr[4][:-11])

    # parse the date and split into year and month
    df1.at[i, 'Year'] = pd.to_datetime(date, format="%m/%Y").year
    df1.at[i, 'Month'] = pd.to_datetime(date, format="%m/%Y").month

    # set the interest rate
    df1.at[i, 'primeRate'] = prime_rate

# drop all na values and unnecessary columns
df1.dropna(inplace=True)
df1.drop(columns=[0], inplace=True)

# cast types
df1['Year'] = df1['Year'].astype(int)
df1['Month'] = df1['Month'].astype(int)

# write to csv
df_rate = df1.copy()
df_rate.to_csv("data/conn_interest_rate.csv", index=False)
df_rate
```

```
     Year  Month  primeRate
0    2023      3       7.75
1    2023      2       7.75
2    2023      1       7.50
3    2022     12       7.00
4    2022     11       7.00
..    ...    ...        ...
262  2001      5       7.50
263  2001      4       7.50
264  2001      3       8.00
265  2001      2       8.50
266  2001      1       9.00

[267 rows x 3 columns]
```

```
# if web driver error
df_rate = pd.read_csv("data/conn_interest_rate.csv")
```

Data Collection: Unemployment Rates

Another factor that appears to impact the real estate market is the state of the economy which correlates to unemployment rates. It is also possible that with higher unemployment rates, there will be less people who are financially capable of purchasing real estate, resulting in lower sales. To analyze whether a relationship truly exists between unemployment rates real estate sales, we first start by collecting data on unemployment rates. The U.S. Bureau of Labor Statistics provides a dataset of labor statistics which we use below.

```
# read unemployment data into the dataframe
unemployment_df = pd.read_csv("data/conn_unemployment.csv")
unemployment_df.head()
```

```
   Year  Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
0  2001  2.3  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.3  3.4  3.5
1  2002  3.6  3.8  3.9  4.1  4.2  4.3  4.5  4.6  4.8  4.9  5.0  5.1
2  2003  5.2  5.3  5.4  5.5  5.5  5.6  5.6  5.6  5.5  5.5  5.4  5.3
3  2004  5.3  5.3  5.2  5.2  5.2  5.1  5.1  5.0  5.0  4.9  4.9  4.9
4  2005  4.9  4.9  4.9  4.9  4.8  4.8  4.8  4.8  4.8  4.7  4.7  4.6
```

Looking at the data, we can see the current unemployment data is not tidy, because we do not fit the criteria for tidy data which includes the following principles of tidy data:

1. Every column is a variable
2. Every row is an observation
3. Every cell is a single value

To make the unemployment data tidy, we need to convert our columns into rows. This can be done in an process called melting which makes it so that the months become values and we end up with the year, month, and unemployment rate as columns. To learn more about tidy data click here.

```
# identify the columns of the dataframe
unemployment_df.columns

Index(['Year', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep',
       'Oct', 'Nov', 'Dec'],
      dtype='object')

# melt the dataframe
unemployment_df = pd.melt(unemployment_df,
        id_vars=['Year'],
        value_vars=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                    'Aug', 'Sep', 'Oct', 'Nov', 'Dec'],
        var_name='Month',
        value_name='UnemploymentRate')

# drop missing values
unemployment_df.dropna(inplace=True)
unemployment_df.head()

   Year Month  UnemploymentRate
0  2001   Jan               2.3
1  2002   Jan               3.6
2  2003   Jan               5.2
3  2004   Jan               5.3
4  2005   Jan               4.9

# array of months
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                    'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

# format the year and month into datetime objects
for i, row in unemployment_df.iterrows():
    unemployment_df.at[i, 'Year'] = pd.to_datetime(row['Year'],
format="%Y").year
    unemployment_df.at[i, 'Month'] =
pd.to_datetime((int(months.index(row['Month'])) + 1),
format="%m").month

unemployment_df.head()

   Year Month  UnemploymentRate
0  2001     1               2.3
1  2002     1               3.6
2  2003     1               5.2
3  2004     1               5.3
4  2005     1               4.9
```

Now that we have retrieved real estate sales data, interest rate data, and unemployment rate data, we can combine all the datasets into one dataset, where we compare the year and the month to insert the correct unemployment rate and interest rate. To do this, we can perform a join using the pandas merge function and join on the columns year and month, since those are the columns that we want to be cross referenced when merging the datasets. Joining datasets is an essential part of being able to compare data across different datasets. Learn more about joins here.

```
# drop missing values
df_copy = df.copy().dropna(subset=['Year', 'Month'])
df_copy['Year'] = df_copy['Year'].astype(int)
df_copy['Month'] = df_copy['Month'].astype(int)


# join the real estate data and unemployment data
df_merge = pd.merge(df_copy, unemployment_df, on=['Year', 'Month'])
df_merge
```

```
      listyear daterecorded         town              address
0         2020   2020-10-05      Andover          303 LAKE RD  \
1         2020   2020-10-23         Avon       21 CLIFF DRIVE
2         2020   2020-10-09  Beacon Falls      19 SUSAN STREET
3         2020   2020-10-13       Bethel   57 CHESTNUT STREET
4         2020   2020-10-21     Branford       2E ANCHOR REEF
...        ...          ...          ...                  ...
6948      2001   2002-03-20   Torrington  616 MIGEON AVE UT 10
6949      2001   2002-03-25     Trumbull      6 PEPPERIDGE RD
6950      2001   2002-03-01     Westport      92 NEWTOWN TPKE
6951      2001   2002-03-11 Wethersfield         703 RIDGE RD
6952      2001   2002-03-27   Woodbridge   27 RICHARD SWEET DR

      assessedvalue  saleamount  salesratio propertytype
residentialtype
0            121300    210000.0    0.577600  Residential   Single
Family  \
1            155390    249000.0    0.624000  Residential   Single
Family
2             32640     45000.0    0.725300  Residential   Single
Family
3            194110    365000.0    0.531800  Residential   Single
Family
4            256000    480000.0    0.533300  Residential
Condo
...             ...         ...         ...          ...              .
..
6948          17100     32900.0    0.519757          NaN
NaN
6949         168500    309900.0    0.543724          NaN
NaN
6950         385900    635000.0    0.607717          NaN
NaN
6951         131400    245000.0    0.536327          NaN
NaN
```

```
6952        307650    439000.0    0.700797         NaN
NaN

      nonusecode  Year Month   x_coord   y_coord  UnemploymentRate
0            NaN  2020    10 -72.35327  41.71416               7.9
1            NaN  2020    10 -72.87619  41.80986               7.9
2            NaN  2020    10       NaN       NaN               7.9
3            NaN  2020    10 -73.40632  41.36916               7.9
4            NaN  2020    10 -72.67344  41.72606               7.9
...          ...   ...   ...       ...       ...               ...
6948         NaN  2002     3       NaN       NaN               3.9
6949         NaN  2002     3       NaN       NaN               3.9
6950         NaN  2002     3       NaN       NaN               3.9
6951         NaN  2002     3       NaN       NaN               3.9
6952         NaN  2002     3       NaN       NaN               3.9

[6953 rows x 15 columns]
```

```
# join the merged datasets and also the interest rate dataset to
combine all three datasets
df_merge = pd.merge(df_merge, df_rate, on=['Year', 'Month'])
df_merge
```

```
      listyear daterecorded          town              address  \
0         2020   2020-10-05       Andover          303 LAKE RD
1         2020   2020-10-23          Avon       21 CLIFF DRIVE
2         2020   2020-10-09  Beacon Falls      19 SUSAN STREET
3         2020   2020-10-13        Bethel    57 CHESTNUT STREET
4         2020   2020-10-21      Branford       2E ANCHOR REEF
...        ...          ...           ...                  ...
6948      2001   2002-03-20    Torrington  616 MIGEON AVE UT 10
6949      2001   2002-03-25      Trumbull        6 PEPPERIDGE RD
6950      2001   2002-03-01      Westport        92 NEWTOWN TPKE
6951      2001   2002-03-11  Wethersfield          703 RIDGE RD
6952      2001   2002-03-27    Woodbridge    27 RICHARD SWEET DR

      assessedvalue  saleamount  salesratio propertytype
residentialtype
0            121300    210000.0    0.577600  Residential   Single
Family  \
1            155390    249000.0    0.624000  Residential   Single
Family
2             32640     45000.0    0.725300  Residential   Single
Family
3            194110    365000.0    0.531800  Residential   Single
Family
4            256000    480000.0    0.533300  Residential
Condo
...             ...         ...         ...          ...             .
..
```

```
6948          17100     32900.0    0.519757         NaN
NaN
6949         168500    309900.0    0.543724         NaN
NaN
6950         385900    635000.0    0.607717         NaN
NaN
6951         131400    245000.0    0.536327         NaN
NaN
6952         307650    439000.0    0.700797         NaN
NaN

      nonusecode  Year Month   x_coord   y_coord  UnemploymentRate
primeRate
0            NaN  2020    10 -72.35327  41.71416               7.9
3.25
1            NaN  2020    10 -72.87619  41.80986               7.9
3.25
2            NaN  2020    10       NaN       NaN               7.9
3.25
3            NaN  2020    10 -73.40632  41.36916               7.9
3.25
4            NaN  2020    10 -72.67344  41.72606               7.9
3.25
...          ...   ...   ...       ...       ...               ...
...
6948         NaN  2002     3       NaN       NaN               3.9
4.75
6949         NaN  2002     3       NaN       NaN               3.9
4.75
6950         NaN  2002     3       NaN       NaN               3.9
4.75
6951         NaN  2002     3       NaN       NaN               3.9
4.75
6952         NaN  2002     3       NaN       NaN               3.9
4.75

[6953 rows x 16 columns]
```

Exploratory Analysis & Data Visualization

First, let's get a feel for exploring the various factors that may affect the sales ratio of real-estate units in Connecticut. We will explore the residential types of the units, creating a dataframe that shows the number of each residential type {Condo, Single Family, Two Family, Three Family, Four Family} for each town. To do so, we will need to create a unique list of residential types, capture the number of real-estate units that match each residential type, and create a new column to display the collective counts.

```
# Get a unique list of the residential types
cleanedList = [x for x in results_df.residentialtype.unique() if x ==
x]
```

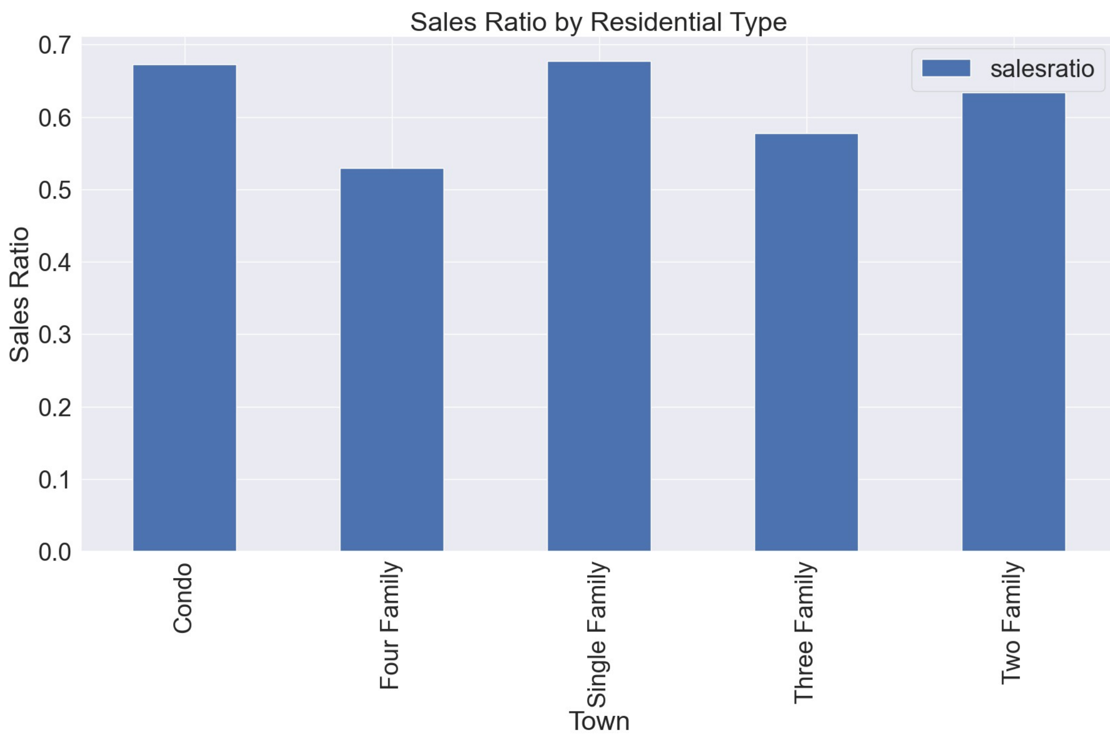
```python
# Create a dataframe with just the town and residential type columns
residential_df = results_df[['town','residentialtype']]

# Plus residential type counts for each town
count_resident_df =
residential_df.groupby(['town','residentialtype']).size().reset_index(
name='count')

# Shows the amount of each residential type for each town where each
column is a residential type and another column for the town
residential_df = count_resident_df.pivot_table(values='count',
index='town', columns='residentialtype',fill_value=0)

temp = residential_df
```

```
residential_df

residentialtype  Condo  Four Family  Single Family  Three Family  Two
Family
town

Andover              0            0              4             0
0
Ansonia              0            0             17             2
1
Ashford              0            0              4             0
1
Avon                11            0             29             0
0
Barkhamsted          0            0              1             0
0
...                ...          ...            ...           ...
...
Windsor Locks        6            0             11             0
0
Wolcott              1            0              9             0
0
Woodbridge           0            0             13             0
0
Woodbury             5            0              6             0
0
Woodstock            0            0             12             0
0

[166 rows x 5 columns]
```

Now that we've organized this data to show the number of each residential types for each town, let's make a graph to better visualize the data. We can make a bar graph to visualize the average sales ratio vs each residential type. We'll start by grouping entries with the same residential type, and aggregating their sales ratios to calculate their mean/average value.

```
# real-estate unit entries grouped by their residential type vs and
average sales ratio
residential_df =
results_df.groupby('residentialtype').agg({'salesratio': 'mean'})

# made into a bar graph
residential_df.plot.bar(figsize=(20,10), title='Sales Ratio by
Residential Type', ylabel='Sales Ratio', xlabel='Town')
```

```
<Axes: title={'center': 'Sales Ratio by Residential Type'},
xlabel='Town', ylabel='Sales Ratio'>
```



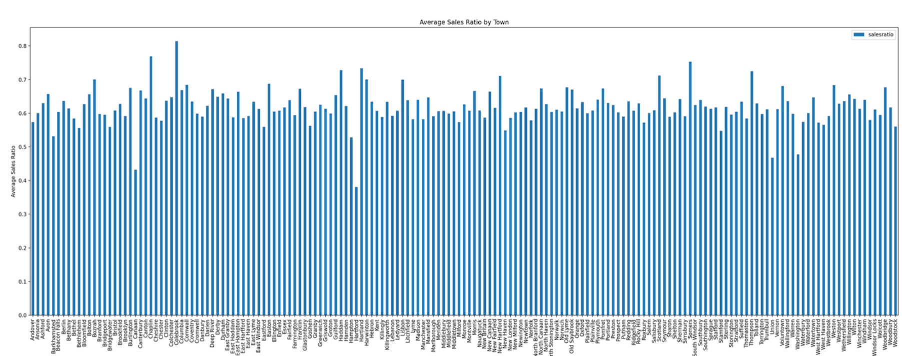
As we can see from this bar graph, sales ratio falls consistently within a tight range of 0.53 and 0.68 across the five residential types, suggesting that there is a weak or no relationship between residential type and sales ratio. Moving forward, we can infer that residential type has little effect on sales ratio, and is a weak factor if at all for sales ratio.

Next, we will examine the relationship of location and the average sale ratio of homes. We will first make a bar graph of towns vs their average sales ratio, and then we will look at how sales ratio differs with locations not guided by town borders using a map. Let's start by making a bar graph with the town and salesratio columns!

```
# make table with just town and salesratio columns
df4 = df_merge[['town', 'salesratio']]
```

```python
# convert all salesratio values to floats
df4['salesratio'] = df4['salesratio'].astype(float)

# group by town and average the salesratio
df4 = df4.groupby('town').mean()

# remove the rows where town is '***Unknown***'
df4 = df4[df4.index != '***Unknown***']

df4.head()

# bar graph with town vs salesratio
df4.plot.bar(figsize=(30,10), title='Average Sales Ratio by Town')
plt.xlabel('Town')
plt.ylabel('Average Sales Ratio')
plt.show()
```

```
/var/folders/04/hdjfckkd4fd__5521d_kggjh0000gn/T/
ipykernel_53103/2815784812.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
```

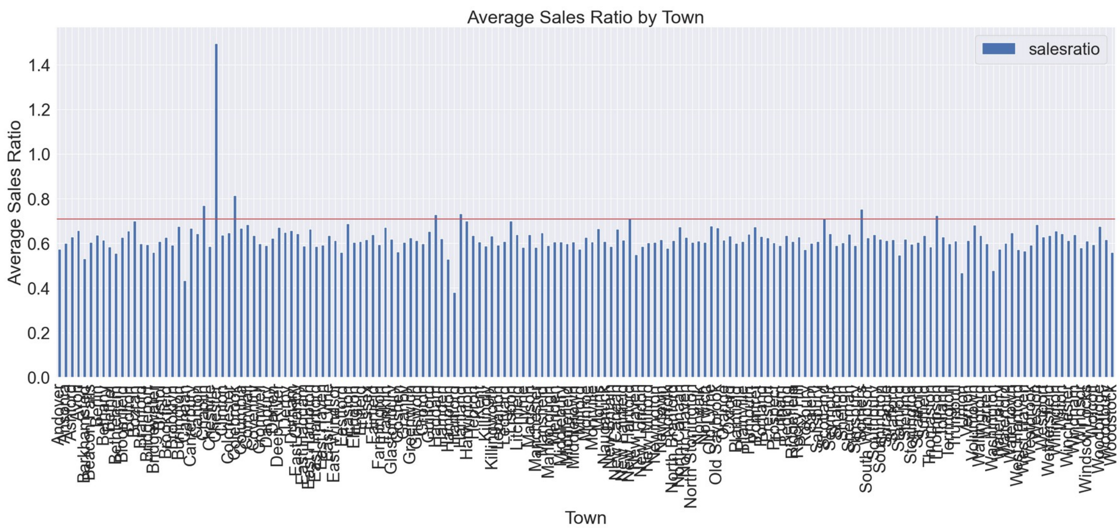



```python
# summary of bar graph
df4.describe()
```

	salesratio
count	169.000000
mean	0.624507
std	0.084827
min	0.380444
25%	0.595276
50%	0.613225
75%	0.641801
max	1.494922

From the graph summary, we can find that an upper outlier = Q3 + 1.5*IQR = 0.640661 + 1.5*(0.640661-0.595011) = 0.709136. We can then refer to the bar graph to find any towns that have sales ratio values that exceed the upper outlier values. To make it easier to identify these values, let's plot a horizontal line on our bar graph at y=0.709136.

```
df4.plot.bar(figsize=(30,10), title='Average Sales Ratio by Town')
plt.xlabel('Town')
plt.ylabel('Average Sales Ratio')

# add a horizontal line y=0.709136 to plt
plt.axhline(y=0.709136, color='r', linestyle='-')

plt.show()
```



We can easily identify the towns that have average sales ratios that are noticeably higher than the rest: Chaplin, Colebrook, Haddam, Hartland, New Hartford, Scotland, Somers, and Thompson. Knowing this, we can include town as a factor that more likely affects sales ratios of real-estate in Connecticut (and to sales ratios in general).

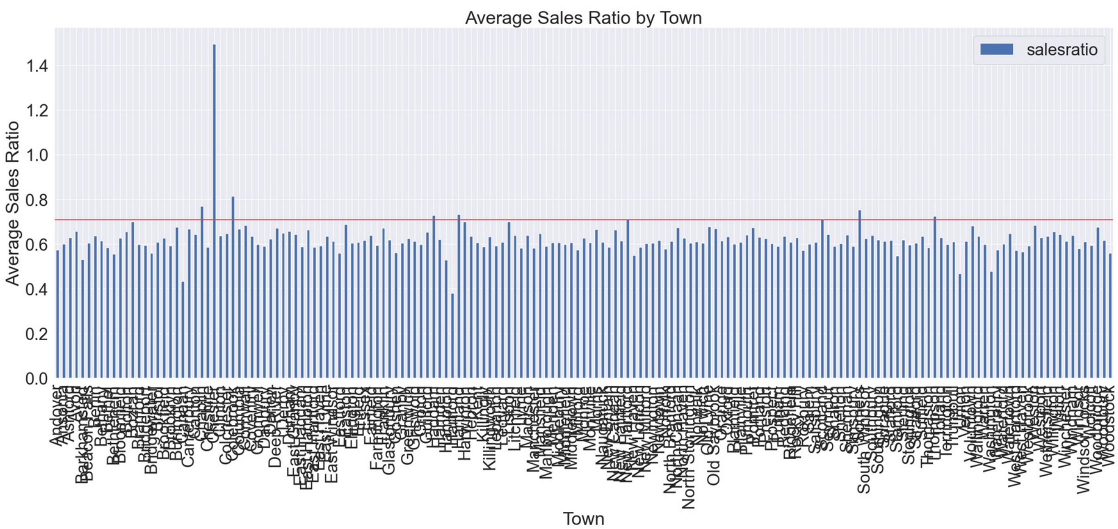
We can further explore the effect of location on sales ratio by creating a map of plotted addresses of real-estate units in Connecticut and focusing on the distribution of spread of sales ratios. We're going to make a map of each house's location, using its longitude (y_coord) and latitude (x_coord) values, and label each point/marker with its address name, on a map of Connecticut. This way, we can get a better visual of where houses are generally located in the state.

```
# summary of bar graph
df4.describe()
```

```
       salesratio
count  169.000000
mean     0.624507
std      0.084827
min      0.380444
```

```
25%      0.595276
50%      0.613225
75%      0.641801
max      1.494922
```

From the graph summary, we can find that an upper outlier = Q3 + 1.5*IQR = 0.640661 + 1.5*(0.640661-0.595011) = 0.709136. We can then refer to the bar graph to find any towns that have sales ratio values that exceed the upper outlier values. To make it easier to identify these values, let's plot a horizontal line on our bar graph at y=0.709136.



We can easily identify the towns that have average sales ratios that are noticeably higher than the rest: Chaplin, Colebrook, Haddam, Hartland, New Hartford, Scotland, Somers, and Thompson. Knowing this, we can include town as a factor that more likely affects sales ratios of real-estate in Connecticut (and to sales ratios in general).

We can further explore the effect of location on sales ratio by creating a map of plotted addresses of real-estate units in Connecticut and focusing on the distribution of spread of sales ratios.

We're going to make a map of each house's location, using its longitude (y_coord) and latitude (x_coord) values, and label each point/marker with its address name. This way, we can get a better visual of where houses are generally located in Connecticut.

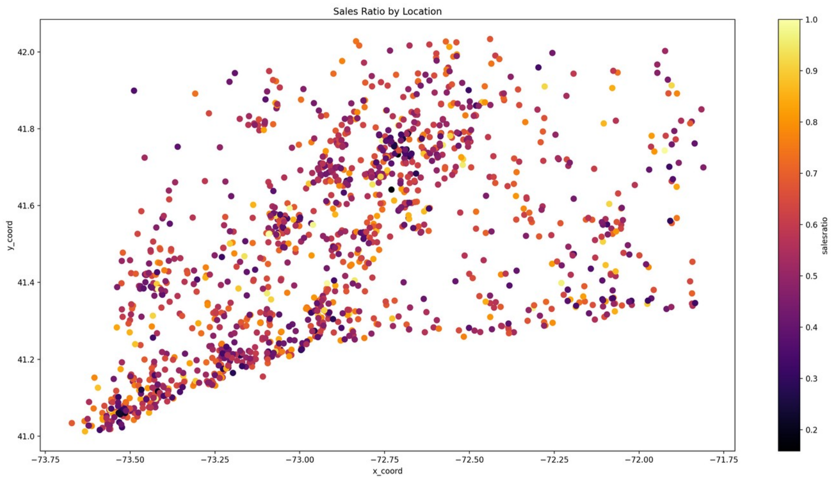
```python
# remove all rows with nan as x_coord or y_coord
df5 = df_merge.dropna(subset=['x_coord', 'y_coord'])\

# create a map centered on connecticut
m = folium.Map(location=[41.6, -72.7], zoom_start=9)

# plot all addresses on the map
for i, row in df5.iterrows():
    folium.CircleMarker(
```

```
        [row['y_coord'], row['x_coord']],
        radius=5,
        popup=row['salesratio'],
        color='blue',
        fill=True,
        fill_color='#3186cc',
        fill_opacity=0.7,
        tooltip=row['salesratio']
    ).add_to(m)

m
```

```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
Cell In[329], line 5
      2 df5 = df_merge.dropna(subset=['x_coord', 'y_coord'])\
      4 # create a map centered on connecticut
----> 5 m = folium.Map(location=[41.6, -72.7], zoom_start=9)
      7 # plot all addresses on the map
      8 for i, row in df5.iterrows():

NameError: name 'folium' is not defined
```

Now we can see the relative locations of each address on a map of Connecticut, with their sales ratios visible when we hover over each point. The map looks so cool to code up! But it would be more helpful if we could see each address's sales ratio more explicitly. Why don't we plot the points onto a graph with x_coord and y_coord as our axes, and include a color scale to indicate how high/low the points' sales ratios are? Let's try it out!

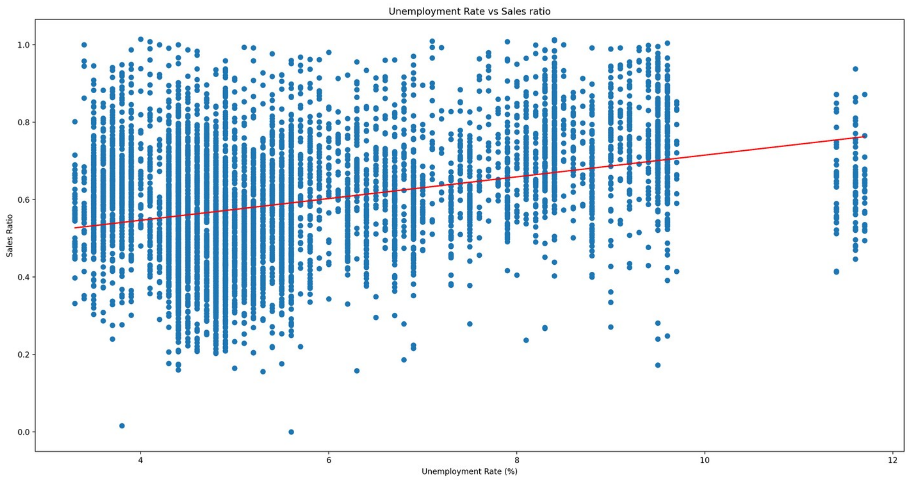
```
# creates a scatter plot of x_coord vs y_coord, where the color of the
points is determined by the salesratio
df_merge.plot.scatter(x='x_coord', y='y_coord', c='salesratio',
colormap='inferno', figsize=(20,10), title='Sales Ratio by Location',
s=50)
plt.xlabel('x_coord')
plt.ylabel('y_coord')
plt.show()
```



Now we can see that the real-estate units with a lower sales ratio are clustered around the bottom left and the center of Connecticut. Observing this graph, we can further support that the location of the real-estate unit is likely to have an impact on the sales ratio of the unit.

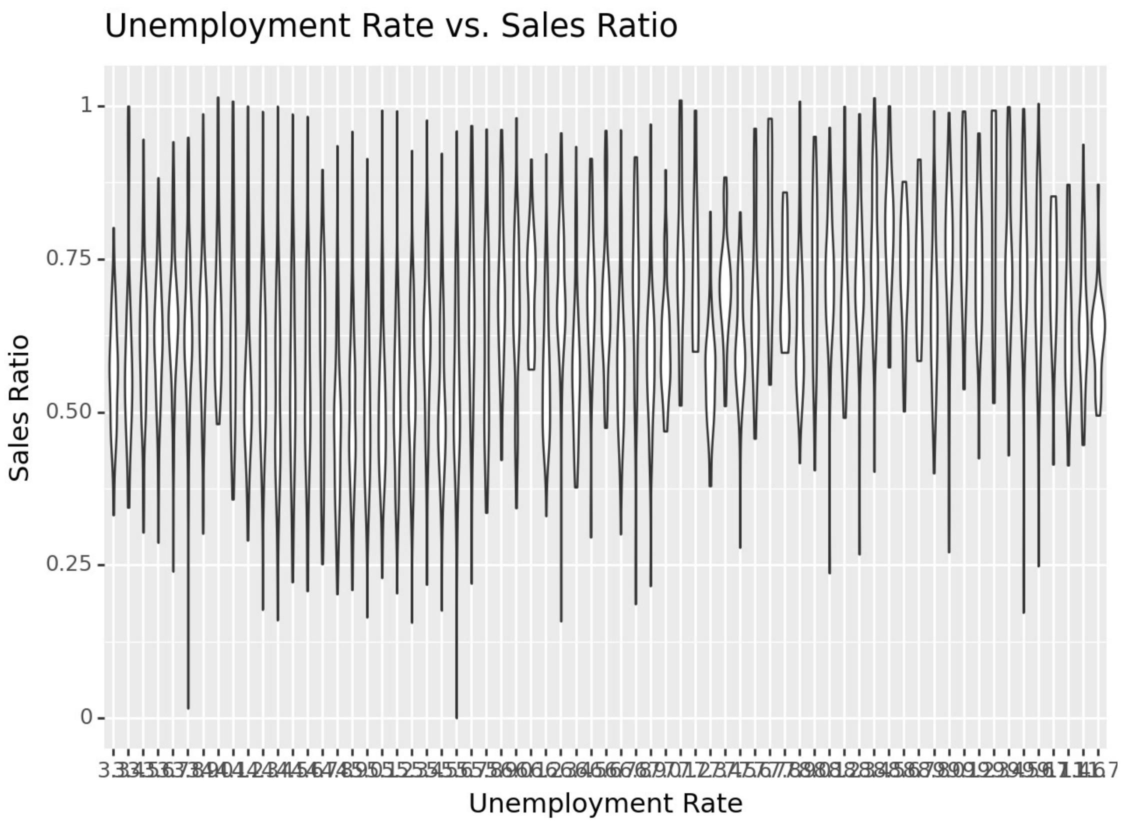
Other factors that could potentially correlate with real estate sales include the unemployment rate, as discussed above. So, how can we examine this relationship? We can plot the unemployment rates in relationship to the sales ratio and try to observe a relationship between the two.

```
# unemployment rate vs sales ratio labels
plt.figure(figsize=(20, 10))
plt.title("Unemployment Rate vs Sales ratio")
plt.xlabel("Unemployment Rate (%)")
plt.ylabel("Sales Ratio")

# plot unemployment rate vs sales ratio in a scatter plot
plt.scatter(df_merge['UnemploymentRate'], df_merge['salesratio'])

# create a linear regression line and plot it
z = np.polyfit(df_merge['UnemploymentRate'], df_merge['salesratio'],
1)
p = np.poly1d(z)
plt.plot(df_merge['UnemploymentRate'],p(df_merge['UnemploymentRate']),
"r")
plt.show()
```



Looking at the points without the linear regression line, we can already see that there is a slightly positive relationship between unemployment rate and the sales ratio, although not completely clear. This is highlighted in the linear regression line because there is a clear positive slope in the line. Thus we have established that there is a slight positive correlation between these two variables.

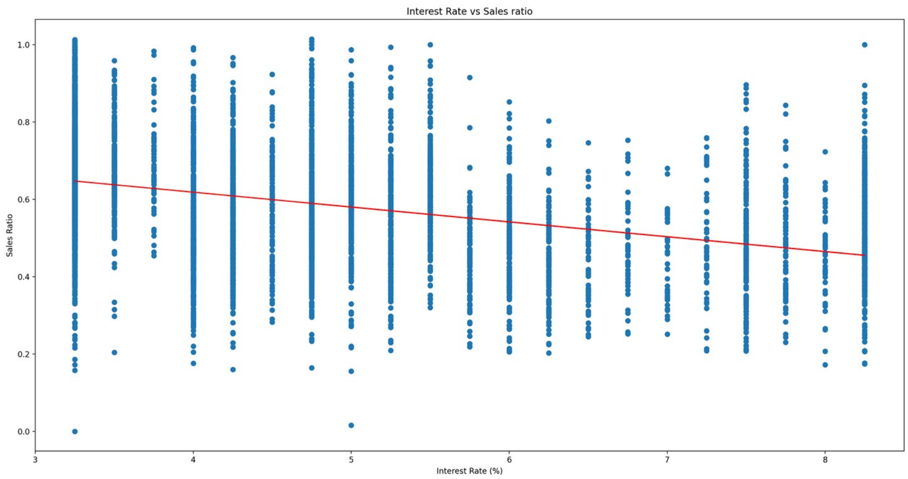
```
# get a copy of the dataframe
df = df_merge.copy()

# cast type of unemployment rate to object
df['UnemploymentRate'] = df['UnemploymentRate'].astype(object)

# create a violin plot of the points for each discrete unemployment
rate
ggplot(data=df, mapping=aes(x='UnemploymentRate', y='salesratio')) +\
geom_violin() +\
labs(title="Unemployment Rate vs. Sales Ratio",
     x = "Unemployment Rate",
     y = "Sales Ratio"),
```




```
(<Figure Size: (640 x 480)>,)
```

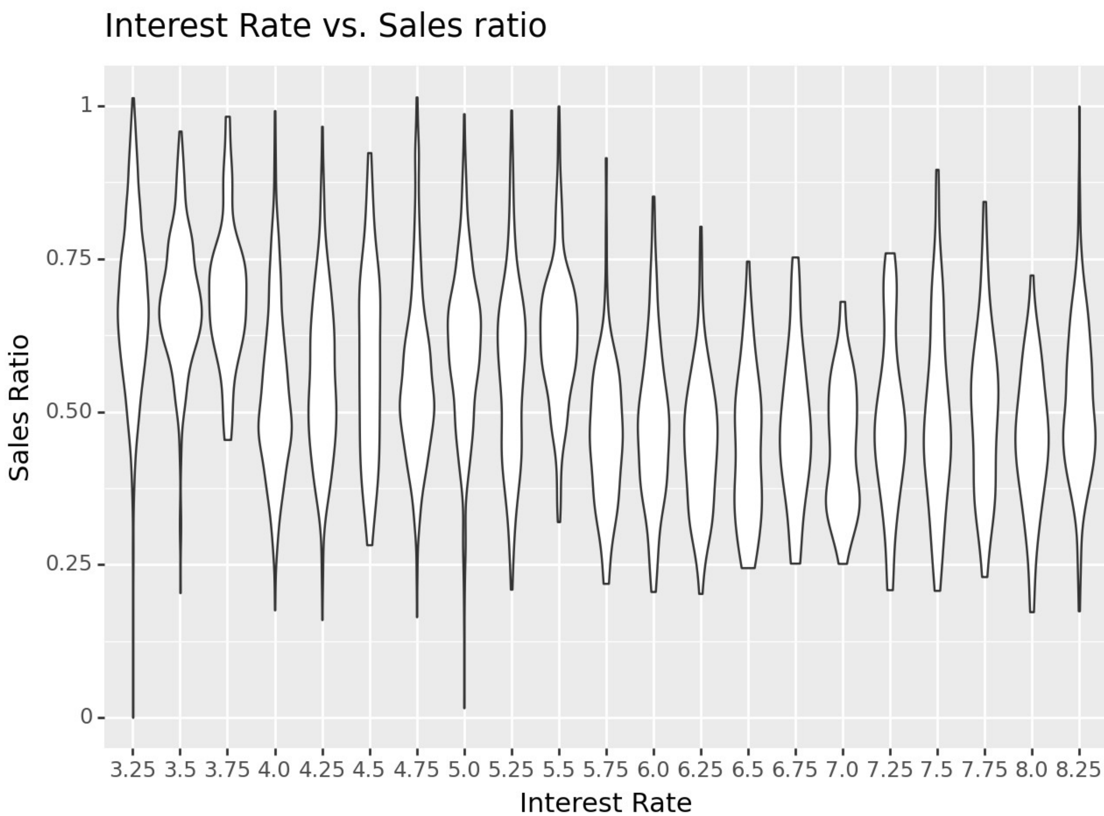
Next, we examine the relationship between interest rate and real estate sales. So if we make a scatter plot of the relationship between interest rate and real estate, we can better observe what sort of relationship exists between the two by observing overall trends.

```python
# plot interest rate vs sales ratio
plt.figure(figsize=(20, 10))
plt.title("Interest Rate vs Sales ratio")
plt.xlabel("Interest Rate (%)")
plt.ylabel("Sales Ratio")

# create a scatter plot
plt.scatter(df_merge['primeRate'], df_merge['salesratio'])

# create a linear regression line
z = np.polyfit(df_merge['primeRate'], df_merge['salesratio'], 1)
p = np.poly1d(z)
plt.plot(df_merge['primeRate'],p(df_merge['primeRate']),"r")
plt.show()
```



In this graph, there does also appear to be a relationship between interest rate and sales; however, this relationship is negative. This relationship is also very subtle and is harder to observe without the linear regression line. A good majority of the points appear a distance away from the linear regression line and the points are all spread out in very similar ranges for each discrete interest rate.

```
# make a copy of the dataframe
df = df_merge.copy()

# cast type to object
df['primeRate'] = df['primeRate'].astype(object)

# make violin plot of interest rate vs sales ratio
ggplot(data=df, mapping=aes(x='primeRate', y='salesratio')) +\
geom_violin() +\
labs(title="Interest Rate vs. Sales ratio",
     x = "Interest Rate",
     y = "Sales Ratio")
```




```
<Figure Size: (640 x 480)>
```

On the violin plot, the negative relationship is made slightly more clear since we can see where the bulk of the points are distributed across the interest rates and it does seem to clearly indicate that the bulk of the points seem to shift downwards as the interest rates decrease.

From the above line graph, we can see each town's sales ratio vs time. We can see that each town's sales ratios tend to hover between 0.3 and 1.3 throughout 2001-2020. Also, the sales ratios for each town are more similar in value towards 2001, and vary more towards 2020. Was there a general trend? Perhaps there was a slightly positive trend, where the average sales ratio across all towns increased. Other than that, we can see that each line representing a town tends to fluctuate up and down more suddenly, rather than increasing/decreasing slowly. This finding supports that there are other factors at play in shaping the sales ratio in each town besides time.

Hypothesis Testing

Now that we've gained a better understanding of the dataset through analysis of various factors' impact on sales ratio, let's move onto Hypothesis Testing!

Our null hypothesis is that there is no relationship between sales ratio and the given factor.

Some alternative hypotheses we can make are:

1. Sales ratio is correlated with location
2. Sales ratio is correlated with unemployment rate
3. Sales ratio is correlated with interest rate

Null hypothesis: There is no relationship between sales ratio and location.

Alternative hypothesis 2: Sales ratio is correlated with location.

Let's figure out if we should reject the null hypothesis and accept the alternative hypothesis 2!

```
# TO DO, + post markdown comment
```

Null hypothesis: There is no relationship between sales ratio and unemployment rate.

Alternative hypothesis 2: Sales ratio is correlated with unemployment rate.

Let's figure out if we should reject the null hypothesis and accept the alternative hypothesis 2!

```
import statsmodels.formula.api as smf
model = smf.ols(formula="salesratio ~ UnemploymentRate",
data=df_merge).fit()
model.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results

==============================================================================
========
Dep. Variable:             salesratio   R-squared:
0.118
Model:                            OLS   Adj. R-squared:
0.118
Method:                 Least Squares   F-statistic:
945.2
Date:                Fri, 12 May 2023   Prob (F-statistic):
6.25e-195
Time:                        20:14:53   Log-Likelihood:
2758.5
No. Observations:                7074   AIC:
-5513.
Df Residuals:                    7072   BIC:
-5499.
Df Model:                           1

Covariance Type:            nonrobust
```

```
==============================================================================
                         coef    std err          t      P>|t|
[0.025      0.975]
------------------------------------------------------------------------------
Intercept              0.5234      0.003    158.854      0.000
0.517       0.530
UnemploymentRate       0.0034      0.000     30.744      0.000
0.003       0.004
==============================================================================
Omnibus:                     1375.674   Durbin-Watson:
1.322
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
5108.473
Skew:                           0.940   Prob(JB):
0.00
Kurtosis:                       6.715   Cond. No.
50.2
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
"""
```

The p-value for the coefficient of unemployment rate is 0.000, which is less than alpha=0.05. This suggests that there is a statistically significant relationship between sales ratio and unemployment rate, so we should reject the null hypothesis and accept alternative hypothesis 2.

Null hypothesis: There is no relationship between sales ratio and interest rate.

Alternative hypothesis 3: Sales ratio is correlated with interest rate.

Let's figure out if we should reject the null hypothesis and accept the alternative hypothesis 3!

```python
import statsmodels.formula.api as smf
model = smf.ols(formula="salesratio ~ primeRate", data=df_merge).fit()
model.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
```

```
==============================================================================
========
Dep. Variable:             salesratio   R-squared:
0.142
Model:                            OLS   Adj. R-squared:
0.142
Method:                 Least Squares   F-statistic:
1168.
Date:                Fri, 12 May 2023   Prob (F-statistic):
4.38e-237
Time:                        20:14:40   Log-Likelihood:
2855.5
No. Observations:                7074   AIC:
-5707.
Df Residuals:                    7072   BIC:
-5693.
Df Model:                           1

Covariance Type:            nonrobust

==============================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
--------
Intercept      0.8008      0.006    132.578      0.000       0.789
0.813
primeRate     -0.0424      0.001    -34.178      0.000      -0.045
-0.040
==============================================================================
========
Omnibus:                     1428.558   Durbin-Watson:
1.357
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
5663.661
Skew:                           0.956   Prob(JB):
0.00
Kurtosis:                       6.945   Cond. No.
15.9
==============================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
"""
```

The p-value for the coefficient of prime (interest) rate is 0.000, which is less than alpha=0.05. This suggests that there is a statistically significant relationship between sales ratio and interest rate, so we should reject the null hypothesis and accept alternative hypothesis 3.

Machine Learning

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()

# normalize the towns column of df_merge using sklearn
le.fit(df_merge['town'])
df_merge['town'] = le.transform(df_merge['town'])

# normalize the residentialtype column of df_merge using sklearn
le.fit(df_merge['residentialtype'])
df_merge['residentialtype'] =
le.transform(df_merge['residentialtype'])

df_merge
```

```
      listyear daterecorded  town                address  assessedvalue

0         2020   2020-10-05     0           303 LAKE RD           2008
\
1         2020   2020-10-23     3         21 CLIFF DRIVE          2801

2         2020   2020-10-09     5        19 SUSAN STREET           195

3         2020   2020-10-13     8    57 CHESTNUT STREET          3480

4         2020   2020-10-21    13         2E ANCHOR REEF          4190

...        ...          ...   ...                   ...            ...

7069      2001   2002-03-20   142  616 MIGEON AVE UT 10             55

7070      2001   2002-03-25   143        6 PEPPERIDGE RD          3055

7071      2001   2002-03-01   157        92 NEWTOWN TPKE          4890

7072      2001   2002-03-11   158           703 RIDGE RD          2272

7073      2001   2002-03-27   166    27 RICHARD SWEET DR          4558


      saleamount  salesratio propertytype  residentialtype  nonusecode
Year
0       210000.0    0.577600  Residential                2         NaN
2020  \
1       249000.0    0.624000  Residential                2         NaN
2020
```

```
2        45000.0    0.725300  Residential                2         NaN
2020
3       365000.0    0.531800  Residential                2         NaN
2020
4       480000.0    0.533300  Residential                0         NaN
2020
...          ...         ...          ...              ...         ...
...
7069     32900.0    0.519757          NaN                5         NaN
2002
7070    309900.0    0.543724          NaN                5         NaN
2002
7071    635000.0    0.607717          NaN                5         NaN
2002
7072    245000.0    0.536327          NaN                5         NaN
2002
7073    439000.0    0.700797          NaN                5         NaN
2002

     Month   x_coord   y_coord  UnemploymentRate  primeRate
0       10 -72.35327  41.71416                46       3.25
1       10 -72.87619  41.80986                46       3.25
2       10       NaN       NaN                46       3.25
3       10 -73.40632  41.36916                46       3.25
4       10 -72.67344  41.72606                46       3.25
...    ...       ...       ...               ...        ...
7069     3       NaN       NaN                 6       4.75
7070     3       NaN       NaN                 6       4.75
7071     3       NaN       NaN                 6       4.75
7072     3       NaN       NaN                 6       4.75
7073     3       NaN       NaN                 6       4.75

[7074 rows x 16 columns]
```

```
relevant_cols = df_merge[['salesratio', 'UnemploymentRate',
'primeRate', 'town', 'residentialtype']]

relevant_cols = relevant_cols.dropna(subset=['salesratio',
'UnemploymentRate', 'primeRate', 'town', 'residentialtype'])

# combine all sales ratio, unemployment rate, and interest rate
grouped by year for better visualization
ml_df = relevant_cols[['UnemploymentRate', 'primeRate', 'town',
'residentialtype', 'salesratio']].groupby(by=['town'], as_index=False)
ml_df.head(10)
```

```
      UnemploymentRate  primeRate  town  residentialtype  salesratio
0                   46       3.25     0                2    0.577600
1                   46       3.25     3                2    0.624000
2                   46       3.25     5                2    0.725300
```

```
3                   46       3.25     8                2    0.531800
4                   46       3.25    13                0    0.533300
...                ...        ...   ...              ...         ...
6965                12       4.75    25                5    0.408800
7009                 1       5.50    12                5    0.575189
7019                 1       5.50    67                5    0.468841
7023                 1       5.50    86                5    0.582040
7061                 6       4.75   124                5    0.449714

[1535 rows x 5 columns]
```

```python
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score

# function for modeling with linear regression
def modelLinReg(X, Y):
  # split the data into training/testing sets
  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.3)

  # create linear regression object
  model = linear_model.LinearRegression()

  # train the model using the training sets
  model.fit(X_train, Y_train)

  # make predictions using the testing set
  Y_pred = model.predict(X_test)

  print('Ordinary Least Squares (OLS)')
  print('Coefficients: ', model.coef_[0])
  print('Intercept: ', model.intercept_)
  print('Mean squared error: %.2f'
        % mean_squared_error(Y_test, Y_pred))
  print('Coefficient of determination: %.2f'
        % r2_score(Y_test, Y_pred))

# function for modeling with ridge regression
def modelRidgeReg(X, Y):
  # split the data into training/testing sets
  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.3)

  # create linear regression object
  model = linear_model.Ridge(alpha=0.5)

  # train the model using the training sets
  model.fit(X_train, Y_train)
```

```
  # make predictions using the testing set
  Y_pred = model.predict(X_test)

  print('Ridge Regression')
  print('Coefficients: ', model.coef_[0])
  print('Intercept: ', model.intercept_)
  print('Mean squared error: %.2f'
        % mean_squared_error(Y_test, Y_pred))
  print('Coefficient of determination: %.2f'
        % r2_score(Y_test, Y_pred))

# function for modeling with lasso regression
def modelLassoReg(X, Y):
  # split the data into training/testing sets
  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.3)

  # create linear regression object
  model = linear_model.Lasso(alpha=0.1)

  # train the model using the training sets
  model.fit(X_train, Y_train)

  # make predictions using the testing set
  Y_pred = model.predict(X_test)

  print('Lasso Regression')
  print('Coefficients: ', model.coef_[0])
  print('Intercept: ', model.intercept_)
  print('Mean squared error: %.2f'
        % mean_squared_error(Y_test, Y_pred))
  print('Coefficient of determination: %.2f'
        % r2_score(Y_test, Y_pred))

# function for modeling with elastic net regression
def modelElasticNetReg(X, Y):
  # split the data into training/testing sets
  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.3)

  # create linear regression object
  model = linear_model.ElasticNet(alpha=0.1)

  # train the model using the training sets
  model.fit(X_train, Y_train)

  # make predictions using the testing set
  Y_pred = model.predict(X_test)

  print('Elastic Net Regression')
```

```
    print('Coefficients: ', model.coef_[0])
    print('Intercept: ', model.intercept_)
    print('Mean squared error: %.2f'
          % mean_squared_error(Y_test, Y_pred))
    print('Coefficient of determination: %.2f'
          % r2_score(Y_test, Y_pred))

X = np.array(df_merge[['UnemploymentRate', 'primeRate',
'residentialtype', 'town']])
Y = np.array(df_merge[['salesratio']])

modelLinReg(X, Y)
print("\n")

modelRidgeReg(X, Y)
print("\n")

modelLassoReg(X, Y)
print("\n")

modelElasticNetReg(X, Y)
print("\n")
```

```
Ordinary Least Squares (OLS)
Coefficients:  [ 0.0010291  -0.02316451 -0.03244144 -0.00011672]
Intercept:  [0.79385828]
Mean squared error: 0.02
Coefficient of determination: 0.27


Ridge Regression
Coefficients:  [ 1.00692376e-03 -2.38438584e-02 -3.19109917e-02 -
2.36004104e-05]
Intercept:  [0.78928174]
Mean squared error: 0.02
Coefficient of determination: 0.28


Lasso Regression
Coefficients:  0.003035298441493321
Intercept:  [0.53674553]
Mean squared error: 0.03
Coefficient of determination: 0.13


Elastic Net Regression
Coefficients:  0.0027242910687823333
Intercept:  [0.59394055]
Mean squared error: 0.02
Coefficient of determination: 0.19
```

```
import seaborn as sns

sns.set(font_scale=2.5)
sns.pairplot(ml_df, height=10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[276], line 4
      1 import seaborn as sns
      3 sns.set(font_scale=2.5)
----> 4 sns.pairplot(ml_df, height=10)

File ~/Documents/cmsc320/lib/python3.10/site-packages/seaborn/axisgrid.py:2098, in pairplot(data, hue, hue_order, palette, vars, x_vars, y_vars, kind, diag_kind, markers, height, aspect, corner, dropna, plot_kws, diag_kws, grid_kws, size)
   2095     warnings.warn(msg, UserWarning)
   2097 if not isinstance(data, pd.DataFrame):
-> 2098     raise TypeError(
   2099         f"'data' must be pandas DataFrame object, not: {type(data)}")
   2101 plot_kws = {} if plot_kws is None else plot_kws.copy()
   2102 diag_kws = {} if diag_kws is None else diag_kws.copy()

TypeError: 'data' must be pandas DataFrame object, not: <class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

```
# make a linear regression model for sales ratio = m1 * unemployment rate + m2 * prime rate + m3 * town + m4 * residential type + b
model = smf.ols(formula="salesratio ~ UnemploymentRate + primeRate + town + residentialtype", data=ml_df).fit()
model.summary()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[278], line 2
      1 # make a linear regression model for sales ratio = m1 * unemployment rate + m2 * prime rate + m3 * town + m4 * residential type + b
----> 2 model = smf.ols(formula="salesratio ~ UnemploymentRate + primeRate + town + residentialtype", data=ml_df).fit()
      3 model.summary()
```

```
File
~/Documents/cmsc320/lib/python3.10/site-packages/statsmodels/base/
model.py:203, in Model.from_formula(cls, formula, data, subset,
drop_cols, *args, **kwargs)
    200 if missing == 'none':  # with patsy it's drop or raise. let's
raise.
    201     missing = 'raise'
--> 203 tmp = handle_formula_data(data, None, formula, depth=eval_env,
    204                           missing=missing)
    205 ((endog, exog), missing_idx, design_info) = tmp
    206 max_endog = cls._formula_max_endog

File
~/Documents/cmsc320/lib/python3.10/site-packages/statsmodels/formula/
formulatools.py:66, in handle_formula_data(Y, X, formula, depth,
missing)
     63         result = dmatrices(formula, Y, depth,
return_type='dataframe',
     64                            NA_action=na_action)
     65     else:
---> 66         result = dmatrices(formula, Y, depth,
return_type='dataframe',
     67                            NA_action=na_action)
     69 # if missing == 'raise' there's not missing_mask
     70 missing_mask = getattr(na_action, 'missing_mask', None)

File
~/Documents/cmsc320/lib/python3.10/site-packages/patsy/highlevel.py:30
9, in dmatrices(formula_like, data, eval_env, NA_action, return_type)
    299 """Construct two design matrices given a formula_like and
data.
    300
    301 This function is identical to :func:`dmatrix`, except that it
requires
   (...)
    306 See :func:`dmatrix` for details.
    307 """
    308 eval_env = EvalEnvironment.capture(eval_env, reference=1)
--> 309 (lhs, rhs) = _do_highlevel_design(formula_like, data,
eval_env,
    310                                   NA_action, return_type)
    311 if lhs.shape[1] == 0:
    312     raise PatsyError("model is missing required outcome
variables")

File
~/Documents/cmsc320/lib/python3.10/site-packages/patsy/highlevel.py:16
4, in _do_highlevel_design(formula_like, data, eval_env, NA_action,
return_type)
```

```
    162 def data_iter_maker():
    163     return iter([data])
--> 164 design_infos = _try_incr_builders(formula_like,
data_iter_maker, eval_env,
    165                                  NA_action)
    166 if design_infos is not None:
    167     return build_design_matrices(design_infos, data,
    168                                  NA_action=NA_action,
    169                                  return_type=return_type)

File
~/Documents/cmsc320/lib/python3.10/site-packages/patsy/highlevel.py:66
, in _try_incr_builders(formula_like, data_iter_maker, eval_env,
NA_action)
     64 if isinstance(formula_like, ModelDesc):
     65     assert isinstance(eval_env, EvalEnvironment)
---> 66     return design_matrix_builders([formula_like.lhs_termlist,
     67                                    formula_like.rhs_termlist],
     68                                   data_iter_maker,
     69                                   eval_env,
     70                                   NA_action)
     71 else:
     72     return None

File
~/Documents/cmsc320/lib/python3.10/site-packages/patsy/build.py:693,
in design_matrix_builders(termlists, data_iter_maker, eval_env,
NA_action)
    689 factor_states = _factors_memorize(all_factors,
data_iter_maker, eval_env)
    690 # Now all the factors have working eval methods, so we can
evaluate them
    691 # on some data to find out what type of data they return.
    692 (num_column_counts,
--> 693  cat_levels_contrasts) = _examine_factor_types(all_factors,
    694                                                 factor_states,
    695
data_iter_maker,
    696                                                 NA_action)
    697 # Now we need the factor infos, which encapsulate the
knowledge of
    698 # how to turn any given factor into a chunk of data:
    699 factor_infos = {}

File
~/Documents/cmsc320/lib/python3.10/site-packages/patsy/build.py:444,
in _examine_factor_types(factors, factor_states, data_iter_maker,
NA_action)
    442 for factor in list(examine_needed):
    443     value = factor.eval(factor_states[factor], data)
```

```
--> 444     if factor in cat_sniffers or guess_categorical(value):
    445         if factor not in cat_sniffers:
    446             cat_sniffers[factor] =
CategoricalSniffer(NA_action,
    447
factor.origin)

File
~/Documents/cmsc320/lib/python3.10/site-packages/patsy/categorical.py:
130, in guess_categorical(data)
    128 if isinstance(data, _CategoricalBox):
    129     return True
--> 130 data = np.asarray(data)
    131 if safe_issubdtype(data.dtype, np.number):
    132     return False

ValueError: setting an array element with a sequence. The requested
array has an inhomogeneous shape after 2 dimensions. The detected
shape was (169, 2) + inhomogeneous part.
```

Step 5: Interpretation: Insight & Policy Decision

Looking at the various factors that could influence real-estate sale ratios, .

This tutorial is a simplification of the complex nature of the real-estate market, and the predictions are not to be followed blindly. We would rather encourage you to use them as a base level of understanding and as a bounceboard for further studying the influences on the real-estate market.

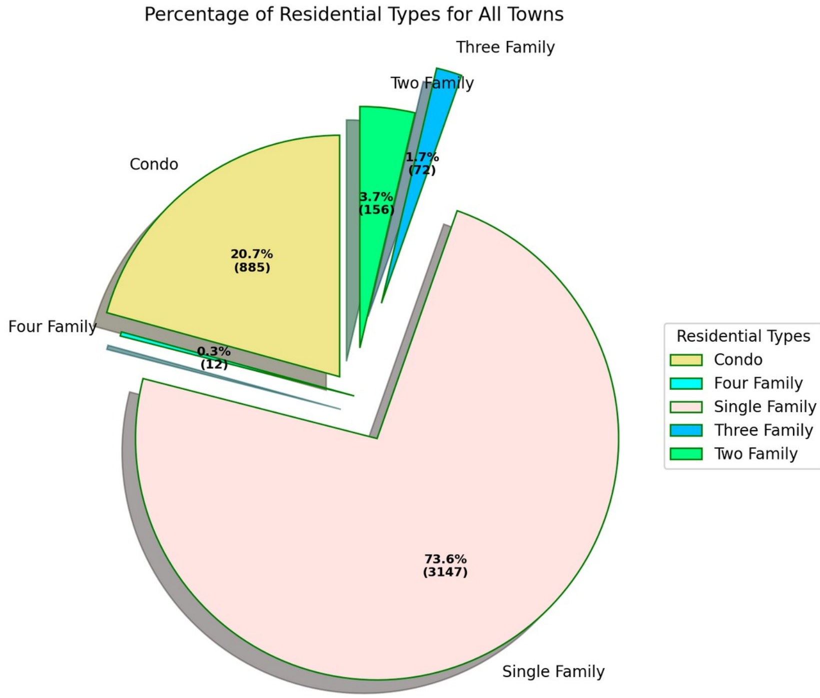
With this tutorial, we hope to have brought you insight into how to pull observations from multiple datasets to draw relationships between specific factors, and finding which model is the most useful for analyzing which factors are most relevant and should be considered the most when it comes to participating in market transactions. And most importantly, we hope that this tutorial brought you a new excitement for the possibilities that data science opens up, for topics as simple as predicting your future grades, to this project, and beyond!

For more resources and data on the real-estate market, please refer to

Appendices

```
res_type_df= results_df.groupby('residentialtype').first()
res_type_df['COUNT'] = results_df['residentialtype'].value_counts()
res_type_df.reset_index(inplace=True)
res_type_df= res_type_df[['residentialtype','COUNT']]
print(res_type_df['residentialtype'])
"""
data = res_type_df['COUNT'].values.tolist()
types = res_type_df['residentialtype'].values.tolist()
fig = plt.figure(figsize =(10, 7))
```

```
plt.pie(data, labels = types)

# show plot
plt.show()"""
explode = (0.1, 0.0, 0.2, 0.4, 0.2)

# Creating color parameters
colors = ( "khaki", "cyan", "mistyrose",
          "deepskyblue", "springgreen")
data = res_type_df['COUNT'].values.tolist()
types = res_type_df['residentialtype'].values.tolist()

wp = { 'linewidth' : 1, 'edgecolor' : "green" }

def func(pct, allvalues):
    absolute = int(pct / 100.*np.sum(allvalues))
    return "{:.1f}%\n({:d})".format(pct, absolute)

# Creating plot
fig, ax = plt.subplots(figsize =(10, 7))
wedges, texts, autotexts = ax.pie(data,
                                  autopct = lambda pct: func(pct,
data),
                                  explode = explode,
                                  labels = types,
                                  shadow = True,
                                  colors = colors,
                                  startangle = 90,
                                  wedgeprops = wp,
                                  textprops = dict(color ="black"))

# Adding legend
ax.legend(wedges, types,
          title ="Residential Types",
          loc ="center left",
          bbox_to_anchor =(1, 0, 0.5, 1))

plt.setp(autotexts, size = 8, weight ="bold")
ax.set_title("Percentage of Residential Types for All Towns\n\n\n")

# show plot
plt.show()

0            Condo
1      Four Family
2    Single Family
3     Three Family
4       Two Family
Name: residentialtype, dtype: object
```



We can also observe that the least popular type of home sale was the Four Family, accounting for only 0.3% of the total sales, and the most popular type of home sale was the Single Family, accounting for 73.6% of total sales.

We will then create a bar graph of for each of the resident types showing which town purchased the most real estate and showing the most popular residence type.

```
for type in cleanedList:

    temp_df = temp[type]

    # Turn on the grid
    temp_df.plot.bar(figsize=(70,10), title='Number of ' + type + ' 
Homes for each Town')
    plt.xlabel('Town')
    plt.ylabel('Number of ' + type + ' Sales')
    plt.show()
```

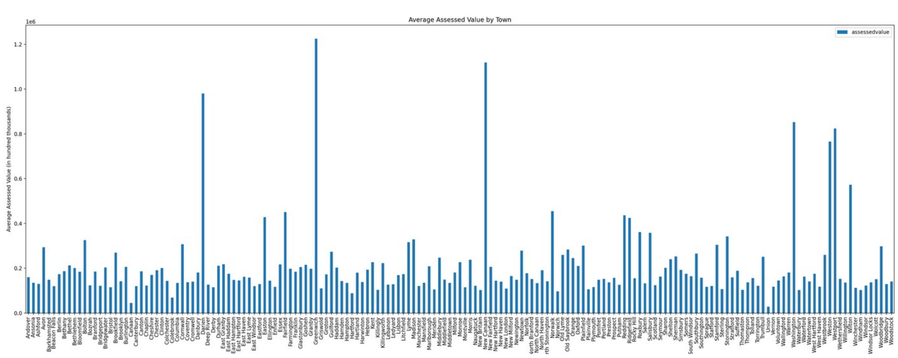
```
----------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call 
last)
Cell In[180], line 3
      1 for type in cleanedList:
```

```
----> 3     temp_df = temp[type]
      5     # Turn on the grid
      6     temp_df.plot.bar(figsize=(70,10), title='Number of ' + 
type + ' Homes for each Town')

NameError: name 'temp' is not defined
```

We can observe from each of these graphs the towns with the highest amount of Single Family, Two Family, Condo, Four Family, and Three Family are Waterbury, Bridgeport, Stamford, Killingly, and Waterbury respectively.

Next, we graphed each unique town with its average assessed value of houses, to get an overall assessment of the average value of homes as seen from their assessed amount in each town.

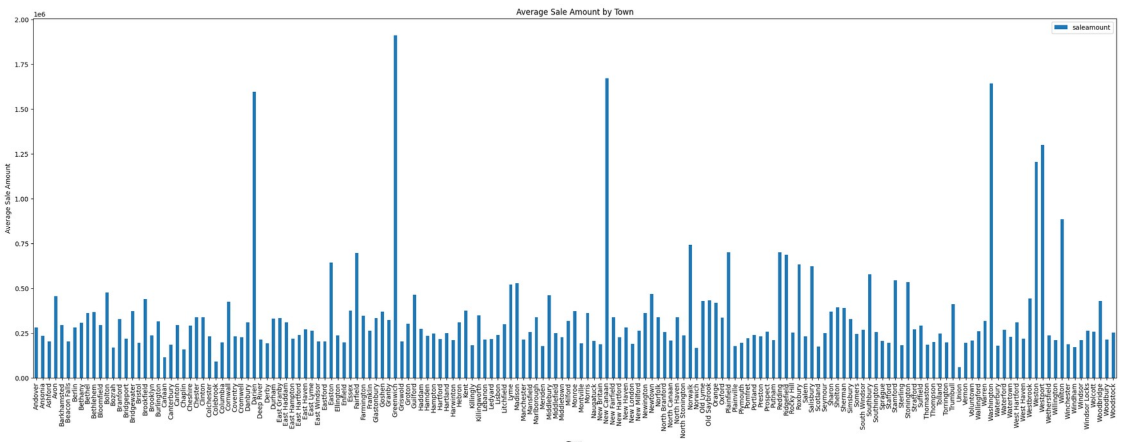
```python
# make table with just town and assessedvalue columns
df2 = df_merge[['town', 'assessedvalue']]

# group by town and average the assessedvalue
df2 = df2.groupby('town').mean()

# remove the rows where town is '***Unknown***'
df2 = df2[df2.index != '***Unknown***']

df2.head()

# bar graph with town vs assessedvalue
df2.plot.bar(figsize=(30,10), title='Average Assessed Value by Town')
plt.xlabel('Town')
plt.ylabel('Average Assessed Value (in hundred thousands)')
plt.show()
```



As we can see from the bar graph above, the average assessed value for homes in most towns is within the ranges of $150,000 and $300,000. From this graph, we can also observe that there are towns with a signficantly higher average assessed value of homes, such as Darien, Greenwich, New Canaan, etc.

Next, we graphed each unique town with its average sale amount of houses, to get an overall assessment of the average value of homes as seen from their sale amount in each town.

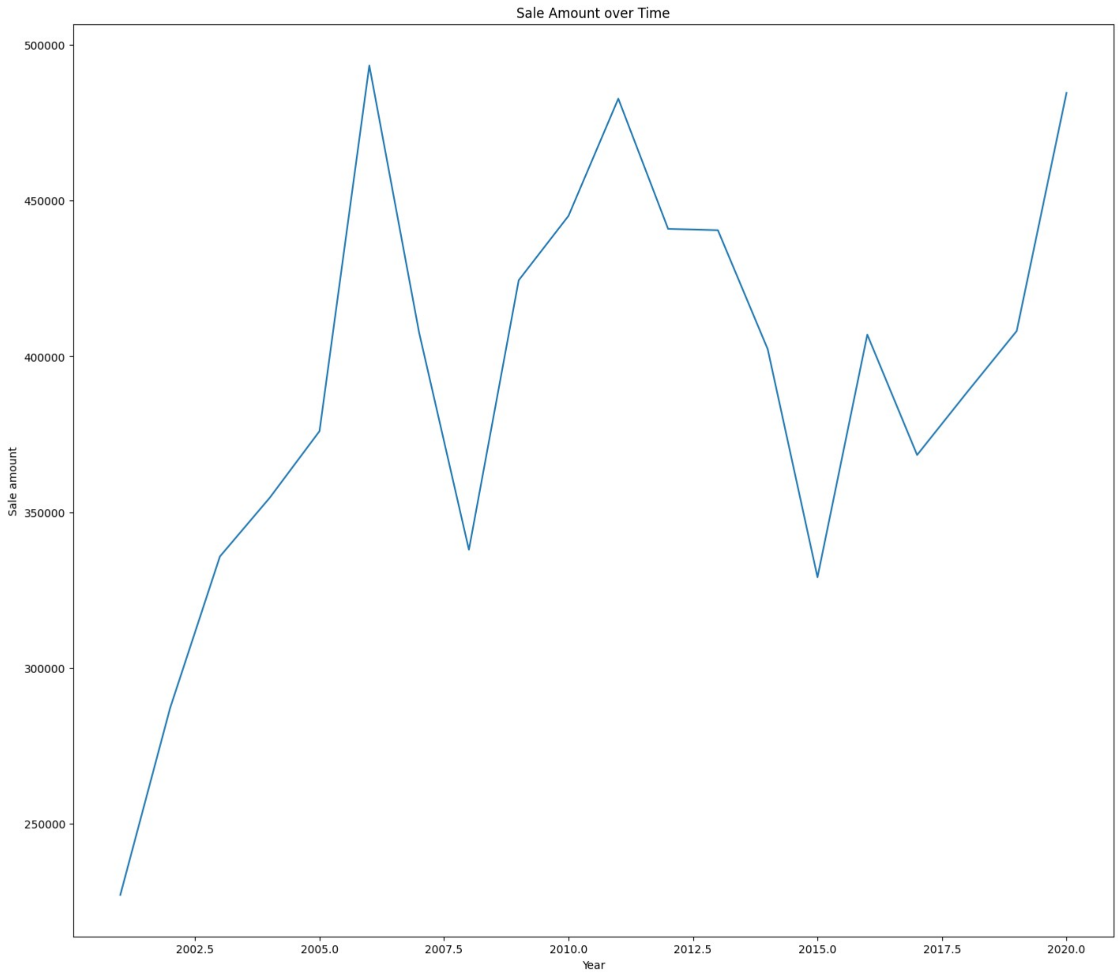
```
# make table with just town and saleamount columns
df3 = df_merge[['town', 'saleamount']]

# group by town and average the saleamount
df3 = df3.groupby('town').mean()

# remove the rows where town is '***Unknown***'
df3 = df3[df3.index != '***Unknown***']

df3.head()

# bar graph with town vs saleamount
df3.plot.bar(figsize=(30,10), title='Average Sale Amount by Town')
plt.xlabel('Town')
plt.ylabel('Average Sale Amount')
plt.show()
```



As we can see from the bar graph above, the average sales amount for homes in most towns is within the ranges of $200,000 and $400,000, a range that is shifted up from the average assessed values' range. From this graph, we can also observe that the towns with a signficantly higher average assessed value of homes, such as Darien, Greenwich, New Canaan, follow with the signficantly higher sales amounts as well.

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

# Group the DataFrame by 'listyear' and calculate the mean
'saleamount'
grouped_df = df_merge.groupby('listyear')['saleamount'].mean()

# Get unique years
years = np.unique(df['listyear'])
```

```
plt.figure(figsize=(17, 15))
plt.plot(years, grouped_df.values)

plt.xlabel('Year')
plt.ylabel('Sale amount')
plt.title("Sale Amount over Time")

plt.show()
```



From the graph above, we can see that the housing market had a sharp and steady increase from 2001 to 2006 and dropped sharply afterwards. Then, the market peaked again in 2011 before hitting a low in 2015. Since 2015, the housing sale amounts have slowly increased and have not yet returned to the highest peak around 2006.

Now we'll make a line graph of sales ratio over time for all towns. What do you expect to see in this line graph? Will there be a general trend of increasing or decreasing sales ratio over time? Let's code it up and take a look!

```python
plt.figure(figsize=(20,10))
# plot each town's salesratio vs time onto a line graph
for town in df_merge['town'].unique():
    df_town = df_merge[df_merge['town'] == town]
    df_town.sort_values(by='Year', inplace=True)
    plt.plot(df_town['Year'], df_town['salesratio'], label=town)

plt.xlabel('Year')
plt.ylabel('Sales Ratio')
plt.title("Sales Ratio over Time")
plt.legend()
plt.show()
```

```
C:\Users\helen\AppData\Local\Temp\ipykernel_14968\682255871.py:5:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
```