

University of Texas at Arlington

MavMatrix

Computer Science and Engineering Theses

Computer Science and Engineering Department

2022

Data Discovery Analysis on Complex Time Series Data

Peter Lawrence Severynen

Follow this and additional works at: https://mavmatrix.uta.edu/cse_theses



Part of the [Computer Sciences Commons](#)

Recommended Citation

Severynen, Peter Lawrence, "Data Discovery Analysis on Complex Time Series Data" (2022). *Computer Science and Engineering Theses*. 505.

https://mavmatrix.uta.edu/cse_theses/505

This Thesis is brought to you for free and open access by the Computer Science and Engineering Department at MavMatrix. It has been accepted for inclusion in Computer Science and Engineering Theses by an authorized administrator of MavMatrix. For more information, please contact leah.mccurdy@uta.edu, erica.rousseau@uta.edu, vanessa.garrett@uta.edu.

Data Discovery Analysis on Complex Time Series Data

Peter L. Severynen

Department of Computer Science and Engineering,

The University of Texas at Arlington

December 2022

Abstract

Complex time series are a ubiquitous form of data in the modern world. They have wide application across many different fields of scientific inquiry and business endeavor. Time series are used to understand and forecast weather patterns, voting patterns, computer network traffic, population health outcomes, demographic changes, the results of scientific experiments, and the performance of stocks and mutual funds. But time series can be difficult to analyze by conventional methods when the data is multivariate, incomplete, or in different formats. To address these issues, an investigation of several multivariate time series datasets was performed using the methods of automatic data discovery and derivative-based analysis. Interactive maps were constructed which displayed the results of the study. Conclusions were drawn and discussed, and an explanation was given of how this method can be applied to other multivariate time series datasets and real-world problems.

Keywords: time series, real estate, stocks, inflation, correlation, analysis, algorithm

Table of Contents

Abstract	II
Data Discovery Analysis on Complex Time Series Data	VI
Introduction.....	1
Part I: Applications of Time Series and Time Series Analysis.....	1
Part II: Data and Methods	6
Figure 1. Components of the Dow Jones Industrial Average Sorted by Stock Name	8
Why was Time Series Analysis Chosen as the Method of Inquiry for These Datasets?.	9
Research Journey	10
Research Questions	11
The Data Pipeline.....	12
Features that will be extracted from the Real Estate Dataset.....	13
Development of Methods and Algorithms	13
Map 1. The 120 Largest U.S. Metropolitan Areas Grouped by the Time to Double	17
Map 2. The 120 Largest U.S. Metropolitan Areas Grouped by Their Senate	
Representation.....	19
Table 1. Average Doubling Time of the Metropolitan Areas by Political Party of their	
Senate Representation	20
Map 3. The 120 Largest U.S. Metropolitan Areas Grouped by Population.....	21
Figure 2. Graph of the First Derivative for Denver, CO.....	25

Figure 3. Graph of the Second Derivative for Denver, CO	25
Figure 4. Graph of the First Derivative for Miami, FL.....	27
Figure 5. Graph of the Second Derivative for Miami, FL	28
Figure 6. Graph of the First Derivative for Austin, TX	29
Figure 7. Graph of the Second Derivative for Austin, TX.....	30
Table 2. Error Ratios for Different Increase in Median Price Threshold Values	32
Map 4. The Housing Bubble in California, Arizona, and Nevada in the early 2000s	33
Map 5. The Housing Bubble in Florida in the early 2000s.....	34
Map 6. The Housing Bubble in Central Florida from 2001 to 2005.....	35
Discussion	35
Figure 8. Graph of the Daily Closing Price of the Dow Jones Industrial Average ...	37
Part III: Other Methods for Performing Time Series Analysis	39
Support Vector Machine	40
Neural Network.....	41
Part IV: Time Series Forecasting	44
Part V: Conclusion	45
Footnotes	47
References	48
Python Coding References.....	57

Appendix I: M.S. Thesis Defense PowerPoint Slides.....	69
Appendix II: Source Code	115
ca_reader.py	115
djia_derivatives.py	116
highest.py	121
inflation_derivatives.py	128
ml1.py	130
plot_data.py.....	132
process_dija.py.....	134
rel.py	143

Data Discovery Analysis on Complex Time Series Data

Introduction

The Organisation for Economic Co-operation and Development (OECD) defines a time series as “a set of regular time-ordered observations of a quantitative characteristic of an individual or collective phenomenon taken at successive, in most cases equidistant, periods / points of time.” (Time Series). A time series may be either univariate or multivariate. In a univariate time series, there is one variable of interest. In a multivariate time series, there are more than one variables of interest. The distinguishing characteristic about time series data is that, as Lazzeri observes, “time isn’t just a metric, but a primary axis.” (Lazzeri, 2020). Indeed, time is a central element of every time series dataset. In a time series, the observations have a temporal ordering and may have a temporal dependency on preceding observations. For example, if the daily high temperatures in a metropolitan area are recorded for a month, and this data is represented as a time series, then the observations have a temporal dependency on one another. If it was hot yesterday, then it is more likely that it will be hot today. When this temporal dependency aspect of time series is present in a dataset, it makes it possible to predict future values of the time series (with varying degrees of accuracy) based on previous values of the time series. This is called time series forecasting. Time series’ predictive power is one of the factors that have led to their widespread use.

Part I: Applications of Time Series and Time Series Analysis

There are many situations in which it is necessary to analyze different time series data using data discovery, data mining, and machine learning (ML) techniques and methods. For example, weather patterns, climate change, voting patterns, computer network traffic, commuter traffic, healthcare data, demographics, scientific observations, and business/financial applications,

etc. may all be represented as time series. In order to extract useful information from these time series and derive meaning from them, it is necessary to perform time series analysis on them.

Weather forecasting is the quintessential application of time series data. Weather stations around the world collect data on temperature, precipitation, humidity, air pressure, and other atmospheric parameters at regular intervals. This atmospheric data constitutes time series. These time series are aggregated and fed as input to supercomputers run by the National Oceanic and Atmospheric Administration (NOAA), a division of the U.S. Department of Commerce. By analyzing atmospheric time series and performing time series forecasting, accurate weather forecasts may be obtained. These forecasts are vital to the economic well-being of farmers, ranchers, builders, and others in weather-sensitive occupations and industries. In fact, according to NOAA,

Each year, the United States averages some 10,000 thunderstorms, 5,000 floods, 1,300 tornadoes and 2 Atlantic hurricanes, as well as widespread droughts and wildfires.

Weather, water and climate events, cause an average of approximately 650 deaths and \$15 billion in damage per year and are responsible for some 90 percent of all presidentially-declared disasters. About one-third of the U.S. economy – some \$3 trillion – is sensitive to weather and climate. National Weather Service (NWS) provides weather, hydrologic, and climate forecasts and warnings for the United States, its territories, adjacent waters and ocean areas, for the protection of life and property and the enhancement of the national economy. (NOAA, 2022)

The National Weather Service is a division of NOAA, and it provides timely and essential weather forecasts and other weather-related information to all Americans. For example, the National Weather Service accurately predicted the paths of Hurricane Ian and Hurricane Nicole,

which devastated Florida in 2022. Because the National Weather Service issued a hurricane watch for the affected counties, many residents evacuated, and thousands of lives were saved as a result! This would not have been possible without modern time series analysis and forecasting algorithms.

The scientific consensus is that extreme weather events such as these are growing in frequency and severity due to climate change. Here too, time series analysis is helpful for understanding the origin and evolution of climate change. Stips et al. asserts that “the results of investigating the information flow between the major radiative forcing’s and the GMTA [global mean surface air temperature anomalies] time series clearly show that total Green House Gases (GHG), dominated in particular by CO₂ forcing, is the main driver of changing global surface air temperature” (Stips et al., 2015). Kremer et al. used a clustering technique to detect climate change in multivariate time series data from hydrology, meteorology, and oceanography (Kremer et al., 2010).

Time series analysis is also helpful in understanding voting patterns and how they change over time. For example, as set forth in the U.S. Constitution, the U.S. House of Representatives currently has 438 members who are elected from their respective congressional districts to serve two-year terms (U.S. Const. art. I, § 2.). The party affiliation of the winning candidate in successive elections forms a time series. By analyzing these time series for each congressional district, it can be determined whether a particular district leans Democratic or Republican. Political strategists use this information to target campaign advertising funds in the geographic areas where they believe that these funds will be most effective. This information also informs candidates’ voter mobilization efforts.

The role of time series analysis in understanding computer network traffic, congestion, and cyberattacks has been well-documented. Ntlangu and Baghai-Wadji trace the history of using time series to model network traffic and explain the theoretical foundations of this approach (Ntlangu & Baghai-Wadji, 2017). The number of packets traveling across a network node at predefined time intervals constitute a time series. By measuring this time series, it can be determined whether the network is congested at that node and will ultimately slow down. Moreover, as Shen observes, time series forecasting may be used to predict network traffic. (Shen et al., 2009). Time series are also an integral part of computer network security. An anomalous and sudden increase in traffic from a particular subnet or range of IP addresses may indicate a distributed denial of service (DDoS) attack on a particular server or Web site. Time series help network security professionals detect and prepare for such cyberattacks. Wu et al. used time series and fuzzy logic to improve the accuracy of “network threat frequency forecasting” (Wu et al., 2008). This work is important because it makes network connections faster and more secure.

Just as time series are used to understand and model the movement of traffic in the digital world, time series may also be used to understand and model the movement of traffic in the physical world. Time series data about traffic flow is frequently used by civil engineers when designing new transportation infrastructure and improving existing transportation infrastructure. This applies equally well in both urban and rural settings. Moreover, Bawaneh and Simon created an algorithm for detecting anomalies in city traffic, such as automobile accidents, using time series data (Bawaneh & Simon, 2019). Once civil engineers know where trouble spots frequently occur, they can recommend solutions to policymakers including widening roads,

installing crash impact barrels, and adjusting traffic signage. This illustrates one of the many practical benefits of time series analysis.

Time series analysis is also an important tool in biomedical research and public health policy planning. As Ferenti observes, “Many biomedical data are available as time series, especially in the field of public health and epidemiology, where indicators are usually collected over time.” (Ferenti, 2017). For example, the number of cases of an infectious disease, the number of positive tests, and the number of deaths attributable to the disease over time are all important factors that may be represented as time series.

Specifically, state public health departments routinely publish datasets of COVID-19 cases, tests, and deaths. An example of such data for the state of California may be found at CHHS Open Data, a website of the California Department of Public Health (California Department of Public Health, 2022). Models and analyses based on these time series datasets inform policymakers when they are tasked with making decisions about masking, social distancing, and other public health measures that are designed to reduce transmission and prevent the healthcare system from becoming overburdened. By analyzing these time series datasets, public health experts and data scientists can track case numbers over time and assess whether COVID-19 cases are trending upward or downward as new variants evolve. Oladunni et al. modeled how COVID-19 affects different communities and demographic groups (Oladunni et al., 2021). This is important because it can potentially lead to increased health equity and targeted outreaches and educational campaigns to underserved communities (racial and ethnic minorities, rural areas, tribal lands, etc.). Ibrahim et al. describe how “data analytics, properly applied, hold great potential to target inequities and reduce disparities [in health care]” (Ibrahim, 2020). Timely,

comprehensive data and accurate, high-quality analysis are prerequisites for sound and effective public health policy.































Time series analysis and forecasting is also applicable to a wide variety of economic and financial data both at an economy-wide level and at the level of individual companies and corporations. Furthermore, many kinds of economic data may be represented as time series. As Hayes observes, “[h]istorical stock prices, earnings, gross domestic product (GDP), or other sequences of financial or economic data can be analyzed as a time series.” (Hayes, 2022). These are metrics that apply to the U.S. economy as a whole. On a smaller level, time series are also used to model various financial performance metrics for individual companies or groups of companies within a specific sector of the economy. For example, the quarterly sales volume of a publicly traded company, or the monthly production of Japanese steelmakers may be modeled as a time series. Unlike the macroeconomic factors stated above, these metrics apply to a particular company or economic sector. Time series analysis helps companies make projections about what their quarterly sales, revenue, and earnings are expected to be for the next year.

Part II: Data and Methods

To evaluate the merits and applicability of derivative-based analysis to a broad set of problems, three datasets were used for this study: the Zillow Single-Family Homes Time Series Metro & U.S. dataset, the Dow Jones Industrial Average (DJIA) dataset, and the Consumer Price Index (Inflation) dataset. The first dataset describes the monthly median sale price of single-family homes in the 911 largest U.S. metropolitan areas. The dataset covers the period from January 1996 to August 2021. This data was collected by the real estate company Zillow and made publicly available (Zillow, 2021). The second dataset consists of the opening and closing

prices of the Dow Jones Industrial Average (DJIA). Ganti describes the DJIA as “a price-weighted index that tracks 30 large, publicly-owned [publicly-traded] companies trading on the New York Stock Exchange and the Nasdaq...[It] serve[s] as a proxy for the broader U.S. economy” (Ganti, 2022). The current composition of the index is shown in Figure 1.

Figure 1. Components of the Dow Jones Industrial Average Sorted by Stock Name

No.	Stock	Rank	% Weight in the Index	Bar Graph of % Weight in the Index
1	3M	21	2.49	
2	American Express	14	2.99	
3	Amgen	4	5.55	
4	Apple	16	2.90	
5	Boeing	13	3.33	
6	Caterpillar	7	4.59	
7	Chevron	11	3.59	
8	Cisco Systems	27	0.94	
9	Coca-Cola	25	1.21	
10	Disney	24	1.86	
11	Dow	26	1.00	
12	Goldman Sachs	2	7.42	
13	Home Depot	3	6.28	
14	Honeywell International	8	4.23	
15	IBM	18	2.88	
16	Intel	30	0.58	
17	Johnson & Johnson	12	3.42	
18	JPMorgan Chase	20	2.61	
19	McDonalds	5	5.31	
20	Merck	22	2.07	
21	Microsoft	6	4.74	
22	NIKE B	23	2.05	
23	Procter & Gamble	19	2.81	
24	Salesforce	17	2.88	
25	Travelers	10	3.62	
26	UnitedHealth Group	1	10.11	
27	Verizon Communications	29	0.76	
28	Visa A	9	4.07	
29	Walgreens Boots Alliance	28	0.81	
30	Walmart	15	2.93	

The 30 companies that comprise the Dow Jones Industrial Average and their current weights.^[1]

The DJIA dataset covers the time period from December 31, 1999 to February 11, 2022. The DJIA dataset was obtained from Yahoo! Finance (Dow Jones Industrial Average (^DJI)). The third dataset lists the inflation rate as measured by the Consumer Price Index (CPI) for each month from January 1996 to December 2021; it was obtained from usinflationcalculator.com (Historical Inflation Rates: 1914-2022, 2022).

Why was Time Series Analysis Chosen as the Method of Inquiry for These Datasets?

For this study, time series analysis was chosen because of the nature of the datasets to be investigated. Lazzeri notes, “[T]ime series analysis is about identifying the intrinsic structure and extrapolating the hidden traits of your time series data in order to get helpful information from it.” (Lazzeri, 2020) For the purposes of this study, each dataset was treated as separate and independent from the others, that is, there are no known dependencies of one dataset upon another. However, all the datasets may be influenced by broader economic trends, which are beyond the scope of this analysis.

Since the datasets are multivariate time series, this presents unique challenges and opportunities. Two of the challenging aspects of analyzing these datasets were that they exist in different formats and use different timescales for their observations. While all the datasets are stored as comma separated values (CSV) files, one of the datasets uses time as the horizontal axis, while the others use time as the vertical axis. Time is also measured in different increments by the disparate datasets. In the real estate and inflation datasets, the observations were recorded on a timescale of months, while in the DJIA dataset, the observations were recorded on a timescale of days. This renders the datasets incompatible for a comparative analysis at the closest level of granularity.

The multivariate nature of the datasets also afforded unique opportunities for analysis that would not have been possible otherwise. For example, as the real estate dataset included geospatial, temporal, and financial components, correlations could be drawn between the three. It was possible to track which regions appreciated faster than others because of the intersection of geography, time, and price.

Research Journey

When this research project began, the initial intent was to develop an online home buyer's assistant that could show properties and recommend offer prices. This entailed a large amount of research into home prices and the fundamentals of the real estate market. Over time this goal evolved significantly as additional data was incorporated into the data discovery and analysis process. A set of color-coded interactive maps were created, which showed the 120 largest U.S. metropolitan areas grouped by different parameters. Normalization of the data and computing the first and second derivatives made it possible to make meaningful comparisons within the dataset. Furthermore, as the researcher continued to explore the data, he found that some of his initial assumptions were validated by the analysis while others were not. For example, the median price of a single-family home doubled faster in California and Florida than it did in the Great Plains states ([Map 1](#)). However, most metropolitan areas in Texas and Louisiana did not double faster than metropolitan areas of similar size in other states, even though these states are located on the Gulf Coast of the United States and have prosperous energy industries. Eventually, as more data was incorporated, broader patterns and trends began to emerge. For example, there has been a general upward trend in asset prices over the 21 years

from 2000 to 2021. However, there have also been times of sudden decreases in asset prices (e.g. stock market crashes and housing price crashes).

First, the method of derivative-based analysis will be applied to the real estate dataset, the DJIA dataset, and the inflation dataset to answer the associated research questions. The results will be discussed, and conclusions will be drawn.

Research Questions

There were ten major research questions that were addressed in this analysis:

1. “In a given metropolitan area, which 1-year, 2-year, 5-year, or 10-year period had the greatest increase in normalized price?” and “What were the highest-performing metropolitan areas in a given 1-year, 2-year, 5-year, or 10-year period.”
2. “Given a certain amount of money to invest, which metropolitan area offered the best returns over the entire 24-year period?”
3. “Given two metropolitan areas, which one had a greater increase in median price over the entire 24-year period?”
4. “In a given metropolitan area, how many years will it take for the median price for a single-family home to double?”
5. “Given the price data for a metropolitan area for 2021, will prices in that metropolitan area rise or fall in the next year?”
6. “Is there a relationship between doubling time and the political party affiliation of the senators that represented the metropolitan areas and their respective states during the years in which the median price of a single-family home doubled?”

7. “Is there a statistically significant correlation between doubling time and the population size of the metropolitan area, i.e. do smaller metropolitan areas double in price faster than larger ones?”
8. “What is the regional distribution of the metropolitan areas that doubled in price, e.g. did metropolitan areas on the East and West Coasts of the United States double in price faster than metropolitan areas in the interior of the country?”
9. “Can it be determined from the available data when there is an asset price bubble in a particular metropolitan area or region?”
10. “How can an asset price bubble be defined mathematically?”

The first five research questions are important because they may be used by potential real estate investors to determine which metropolitan areas are likely to have the fastest return on investment. Leusin defines “return on investment” as “the gain or loss a real estate property generates minus its initial costs over a specific period of time” (Leusin, 2017). The last five questions are important because they deal with larger political, social, and economic trends.

The Data Pipeline

The first step in the data pipeline was preprocessing the data. Hagan explains that “Data preprocessing consists of such steps as normalization, non-linear transformations, feature extraction, coding of discrete inputs/targets, handling missing data, etc.” (Hagan, 2014). For this analysis, data preprocessing included data cleaning, type conversion, and feature extraction. First, the data was read into the program from a CSV file stored on disk. Once the data was read into the program, it was stored in a Pandas DataFrame for easy access and manipulation. Each of the fields in the DataFrame was initially interpreted as a string. Rows with missing values were

dropped. Then the program performed automatic type conversion from string to float so that the powerful mathematical and linear algebra functions of Numpy could be used on the numerical features in the dataset.

Features that will be extracted from the Real Estate Dataset

Next, features were extracted from the real estate dataset. The following five features were chosen to compare metropolitan areas from the real estate dataset because they were the most informative: time, median price, normalized median price, state, and region. Normalized median price and region are derived features.

Development of Methods and Algorithms

In order to answer the research questions, several methods and algorithms were developed. Initially fixed-length time periods such as years were used to partition and analyze the real estate dataset at a higher level of granularity. Then these were generalized to multi-year spans of time. One of the challenges was the different formats of the datasets. Like all tabular datasets, the real estate dataset consists of rows and columns. Each row represents a different metropolitan area. There are 911 metropolitan areas in all. The first five columns are features, such as *RegionName*, *RegionType*, and *StateName*. The remaining columns are the months that the price data was recorded for. Therefore, the real estate dataset is unique among the datasets because in this dataset time is the horizontal axis; whereas, in the other datasets time is the vertical axis. The scale of time is different, too, among the three datasets. In the CPI dataset, time is measured in months. In the real estate dataset, time is measured in months. In the DJIA dataset, time is measured in days. All of these timescales are important, but in order to make meaningful comparisons, years were chosen as the common timescale.

It was important to normalize the prices for the real estate dataset because housing prices can vary widely from one market to another. For example, the median price of a single-family home in Selma, Alabama in August 2021 (the most recent month for which data is available in the real estate dataset) was \$60,503. The median price of a single-family home in San Jose, CA in August 2021 was \$1,564,395 – over \$1.5 million dollars! Normalization helps to even-out these differences so that appreciation and depreciation trends may be seen over time. Normalization yields an apples-to-apples comparison instead of an apples-to-oranges comparison.

There are different kinds of normalization, though. Two of the most frequently used types of normalization for vectors are L1 normalization and L2 normalization. L1 normalization takes the sum of the absolute values of the vector to be normalized. L2 normalization sums the squares of the absolute values of the elements of the vector.

$$|\mathbf{x}|_1 = \sum_{r=1}^n |x_r|$$

Eq. 1. The formula for L1 normalization.^[2]

$$|\mathbf{x}| = \sqrt{\sum_{k=1}^n |x_k|^2}$$

Eq. 2. The formula for L2 normalization.^[3]

For this analysis, a different method of normalization was chosen. The maximum price of a single-family home in a given metropolitan area across a fixed time period (one or more years) was found. Then each price was divided by the maximum price in order to normalize it.

The first question has two parts: “In a given metropolitan area, which 1-year, 2-year, 5-year, or 10-year period had the greatest increase in normalized price?” and “What were the highest-performing metropolitan areas in a given 1-year, 2-year, 5-year, or 10-year period.” To answer the first part of this question, the following algorithm was designed:

Algorithm 1a

- 1) partition the dataset into user-specified equal length time periods
- 2) for each period
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the specified timespan
 - b) change in price = final price - initial price
 - c) store the name of the of the metropolitan area and the change in its normalized price in a data structure
- 3) sort the metropolitan areas in descending order by change in price.
- 4) iterate through the periods comparing the increase in normalized price in each period for that metropolitan area.
- 5) return the highest increase in normalized price and the period in which it occurred

To answer the second part, the following algorithm was used:

Algorithm 1b

- 1) partition the dataset into user-specified equal length time periods
- 2) For a user-defined period
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the specified timespan
 - b) change in price = final price - initial price
 - c) store the name of the metropolitan area and the change in its normalized price in a data structure
- 3) sort the metropolitan areas in descending order by change in price
- 4) print the user-specified number of metropolitan areas and their respective increases in normalized price

After these algorithms were implemented, manual testing was performed on them, and they were found to work on any number of metropolitan areas in the dataset.

The following algorithm was used to answer the second research question:

Algorithm 2

- 1) set a user-defined threshold for the maximum initial starting price
- 2) filter the dataset based on the threshold
- 3) for each metropolitan area under the threshold
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the entire timespan of the dataset
 - b) change in price = final price - initial price

- c) store the name of the metropolitan area and the change in its normalized price in a data structure
- 4) sort the metropolitan areas in descending order by change in price
- 5) return the name of the metropolitan area with the highest increase in normalized price and its change in price

To answer the third research question, the following algorithm was used:

Algorithm 3

- 1) for each metropolitan area in the dataset
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the entire timespan of the dataset
 - b) change in price = final price - initial price
 - c) store the name of the metropolitan area and the change in its normalized price in a data structure
- 2) sort the metropolitan areas in descending order by change in price
- 3) define two variables to store the increase in normalized price for the two user-specified metropolitan areas
- 4) iterate through the sorted list of metropolitan areas, updating each metropolitan area's price variable
- 5) choose the metropolitan area that had the highest increase in normalized price over the entire timespan of the dataset
- 6) print the names and the respective increases in normalized price for the two metropolitan areas

To compute the shortest time interval for the median price of a single-family home to double, the following algorithm was used:

Algorithm 4

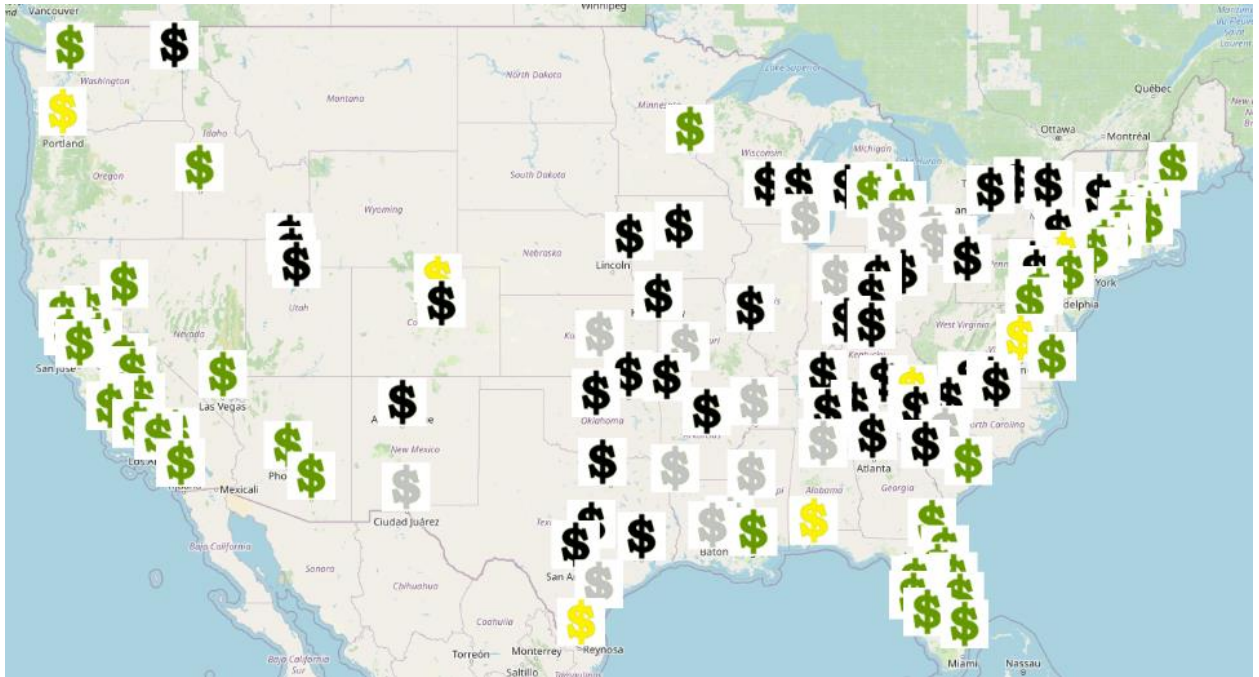
- 1) find the minimum monthly price for a user-specified metropolitan area
- 2) iterate through the succeeding columns until a price greater than or equal to twice the minimum price is found
- 3) count the number of months that occurred between these two price points
- 4) divide the number of months by 12 to obtain the doubling time in years

round the doubling time in years to the desired precision

There is also a geospatial component to this analysis because the state and region in which each metropolitan area is located were considered when assessing appreciation rates.

In this analysis, the doubling time was rounded to the nearest tenth and the results were displayed in Map 1.

Map 1. The 120 Largest U.S. Metropolitan Areas Grouped by the Time to Double



Generated with Leaflet.js^[4]

Legend

Green: Metropolitan areas where the prices doubled in 0 to 10 years.

Yellow: Metropolitan areas where the prices doubled in 10 to 20 years.

Black: Metropolitan areas where the prices doubled in 20 to 30 years.

Gray: Metropolitan areas where the prices did not double during the years for which there is data available.

This map shows the largest 120 U.S. metropolitan areas by population grouped by the time for the normalized median price of a single-family home in that metropolitan area to double. The metropolitan areas denoted by the green dollar sign icon doubled in price the fastest, and the metropolitan areas denoted by the black dollar sign icon doubled in price the slowest.

Doubling times varied widely between states and within certain states. Among the 120 most populous metropolitan areas in the dataset, the shortest doubling time was San Jose, CA (4.5 years). The longest doubling time was undefined because there were some metropolitan areas (e.g. Chicago, IL and Cleveland, OH) that did not double in price during the 25 years that are recorded in the dataset, though prices in these metropolitan areas did generally increase.

To answer the fifth research question, the following algorithm was used:

Algorithm 5

- 1) compute the derivative of normalized price with respect to time for each year in the dataset for a user-specified metropolitan area
- 2) compute the median derivative for that metropolitan area over the past 24 years
- 3) If the derivative of price with respect to in 2021 was *more than 20% above* the median derivative for that metropolitan area over the past 24 years, then the next year was predicted to be a good (above average) year for price increases in that metropolitan area.
- 4) If the derivative of price with respect to time in 2021 was *more than 20% below* the median derivative for that metropolitan area over the past 24 years, then the next year was predicted to be a poor (below average) year for price increases in that metropolitan area.
- 5) If the derivative of price with respect to time in 2021 was *between 80% and 120%* of the median derivative for that metropolitan area over the past 24 years, then the next year was predicted to be a fair (average) year for price increases in that metropolitan area.

Algorithm 6

- 1) Assign the states to regions
- 2) For each state, take the metropolitan area with the highest median price and the lowest median price and compare them to the metropolitan area to the highest and lowest median price respectively in every other state in that region.

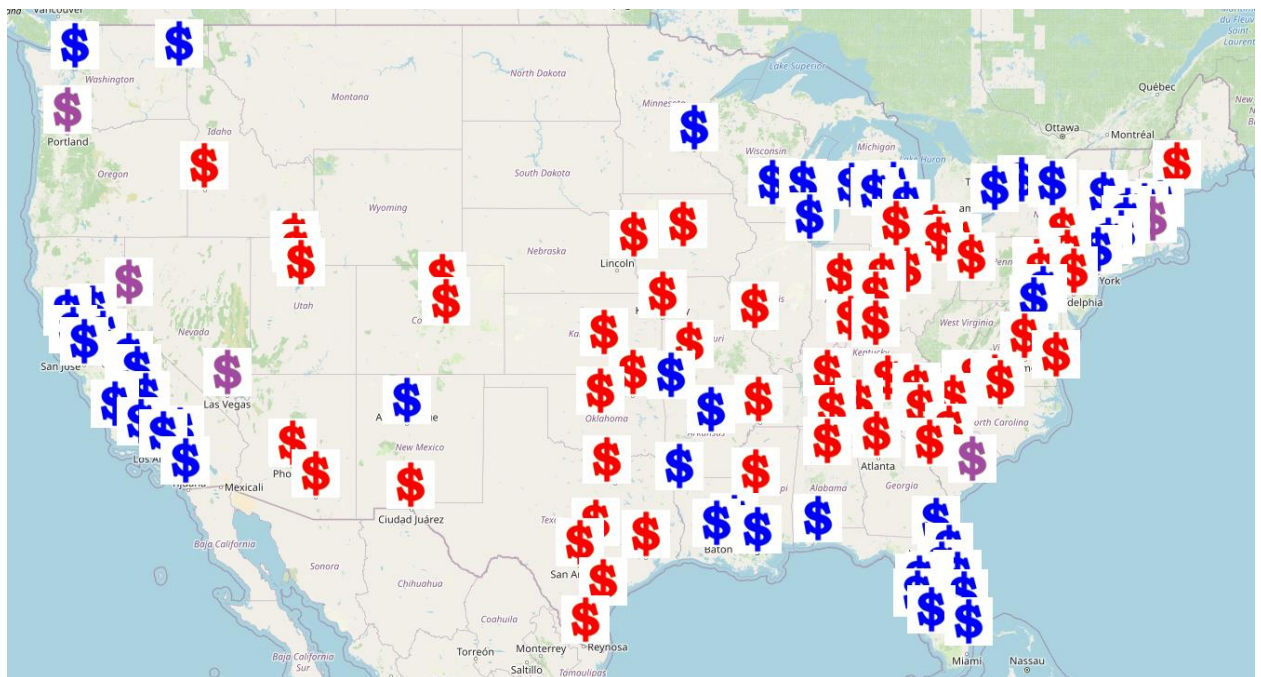
By studying these derivatives, the 2008-2009 housing crash can be clearly seen in metropolitan areas across the country.

For example, in Las Vegas, NV, the derivative of normalized price w.r.t time was negative from 2007 to 2011, and the median price of single-family homes was decreasing in Las Vegas during this time. This sharp decrease in median home prices had long-lasting effects on the

residential real estate market in Las Vegas. The median price of a single-family in the metropolitan area did not reach its pre-crash high of \$370,412 again until July 2021. It took approximately 15 years for real estate prices to recover in Las Vegas.

To ascertain whether there was any correlation between political representation and the rate at which residential housing prices increased in the 120 largest U.S. metropolitan areas, the party affiliation of the U.S. senators that represented these metropolitan areas' states during the periods in which the median price of a single-family home doubled were determined. The median price of single-family homes in metropolitan areas on the East and West Coasts of the U.S. tended to double faster than those in the interior of the country.

Map 2. The 120 Largest U.S. Metropolitan Areas Grouped by Their Senate Representation



Generated with Leaflet.js^[4]

Legend

Red: Metropolitan areas that are located in a state that was represented by two Republican senators during the years in which the prices in that metropolitan area doubled.

Blue: Metropolitan areas that are located in a state that was represented by two Democratic senators during the years in which the prices in that metropolitan area doubled.

Purple: Metropolitan areas that are located in a state that was represented by one Republican and one Democratic senator during the years in which the prices in that metropolitan area doubled.

This map shows the largest 120 U.S. metropolitan areas by population grouped by the political affiliation of their state's U.S. Senate delegation. Red means two Republican senators, blue means two Democratic senators, and purple means one Republican senator and one Democratic senator.

The coastal metropolitan areas are more likely to support Democratic political candidates in Senate races as shown in Map 2, but correlation does not prove causation and other factors could be at play (e.g. education, income level, immigration, and local industries). The average doubling time of the metropolitan areas and their Senate representation during that time is shown in Table 1.

Table 1. Average Doubling Time of the Metropolitan Areas by Political Party of their Senate Representation

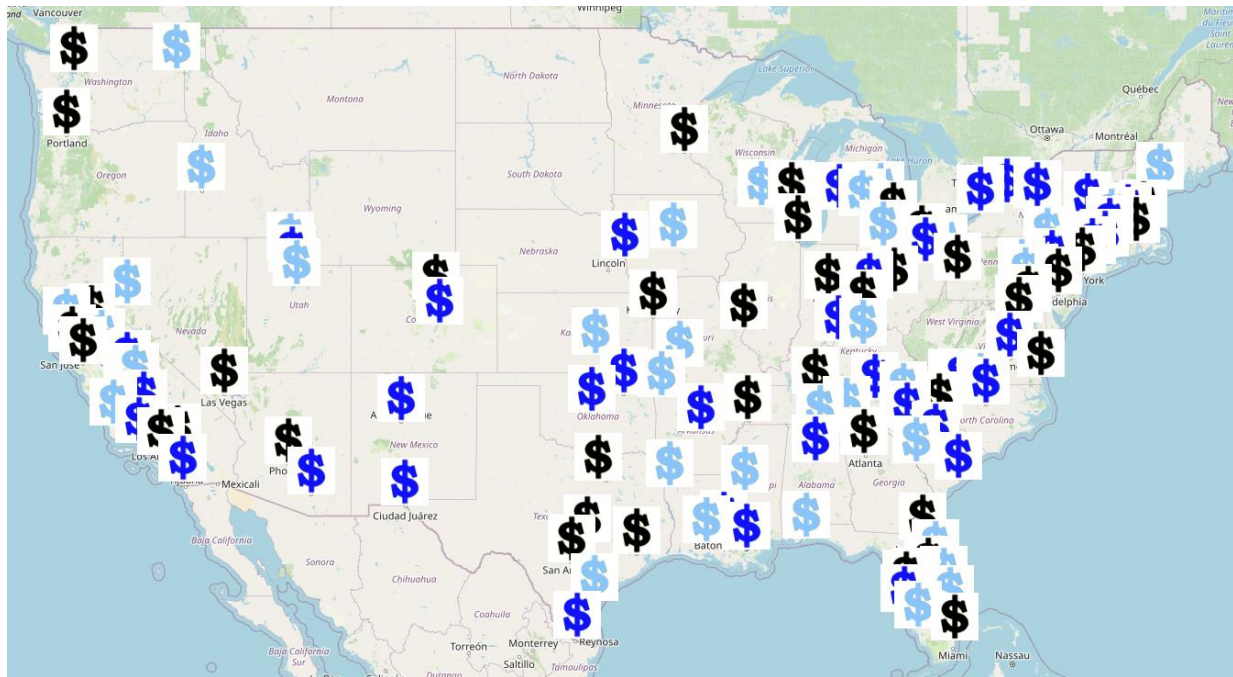
Political Representation in the U.S. Senate	Average Doubling Time in Years
Two Democratic senators	11.2
Two Republican senators	20.7
One Democratic and one Republican senator	7.7

This table shows the average time for the normalized median price of a single-family home to double in the 120 largest U.S. metropolitan areas compared to the political affiliation of their Senate representation.

Interestingly, the average doubling time of the metropolitan areas represented by Democratic and Republican senators (7.7) was lower than either the average doubling time of the metropolitan areas represented by Democratic senators (11.2 years), or the metropolitan areas represented by Republican senators (20.7 years).

To answer the question, “Is there a statistically significant correlation between doubling time and the population size of the metropolitan area, i.e. do smaller metropolitan areas double in price faster than larger ones?” the correlation between population size and doubling time was computed. The metropolitan areas are sorted by population size on Map 3.

Map 3. The 120 Largest U.S. Metropolitan Areas Grouped by Population



Generated with Leaflet.js ^[4]

Legend
Black: Metropolitan areas in the 1 to 40 largest U.S. metropolitan areas by population.
Dark Blue: Metropolitan areas in the 40 to 80 largest U.S. metropolitan areas by population.
Light Blue: Metropolitan areas in the 80 to 120 largest U.S. metropolitan areas by population.

This map shows the largest 120 U.S. metropolitan areas by population grouped by the population size of the metropolitan area. The metropolitan areas denoted by the black dollar sign icon were the most populous, and the metropolitan areas denoted by the light blue dollar sign icon were the least populous.

To compute the correlation between population size and doubling time, first a subset of the 120 most populous metropolitan areas was taken and plotted on a map. In the dataset, the metropolitan areas are sorted in decreasing order of population size. The most populous metropolitan area in the top 120 was New York, NY with a population of 20.1 million in its metropolitan area and *size_rank* 1 (U.S. Census Bureau, 2022). The least populous metropolitan area in the top 120 was Huntsville, AL with a population of 200,574 and *size_rank* 121 (U.S. Census Bureau, 2022). The population of each metropolitan area was not included in the dataset; therefore, the variable *size_rank* (relative metropolitan area size based on population) was used as a proxy for population. For the correlation calculation, the independent variable was *size_rank* and the dependent variable was doubling time. The Pearson correlation coefficient between *size_rank* and doubling time was 0.11. This means that the *size_rank* of the metropolitan area was not correlated with the doubling time of the metropolitan area, hence smaller metropolitan areas did not double in price faster than larger ones.

Next, the eighth research question, “What is the regional distribution of the metropolitan areas that doubled in price, e.g. did metropolitan areas on the East and West Coasts of the United States double in price faster than metropolitan areas in the interior of the country?” was investigated. Seventy-six percent of the metropolitan areas where the median price of a single-family home doubled in less than 10 years were located in a state that borders either the Atlantic Ocean or the Pacific Ocean.

The metropolitan areas that did not follow this pattern were Boise City, ID (7.2 years); Phoenix, AZ (9.3 years); Tucson, AZ (9.7 years); Minneapolis-St. Paul, MN (8.1 years); New Orleans, LA (9.6 years); Detroit, MI (7.9 years); Flint, MI (9.1 years); and Lansing, MI (8.7 years). With the exception of New Orleans, LA, these metropolitan areas are all located inland, i.e. not in a coastal state.

Some of the metropolitan areas with sub-10-year doubling times may be outliers. For example, New Orleans is the only metropolitan area on the Gulf Coast of the United States that had a doubling time of less than 10 years. In New Orleans the doubling time ran from January 1996 to August 2005. Detroit, MI; Flint, MI; and Lansing, MI may also be outliers because the metropolitan areas nearby them had doubling times that were twice as long as these three metropolitan areas in Michigan did.

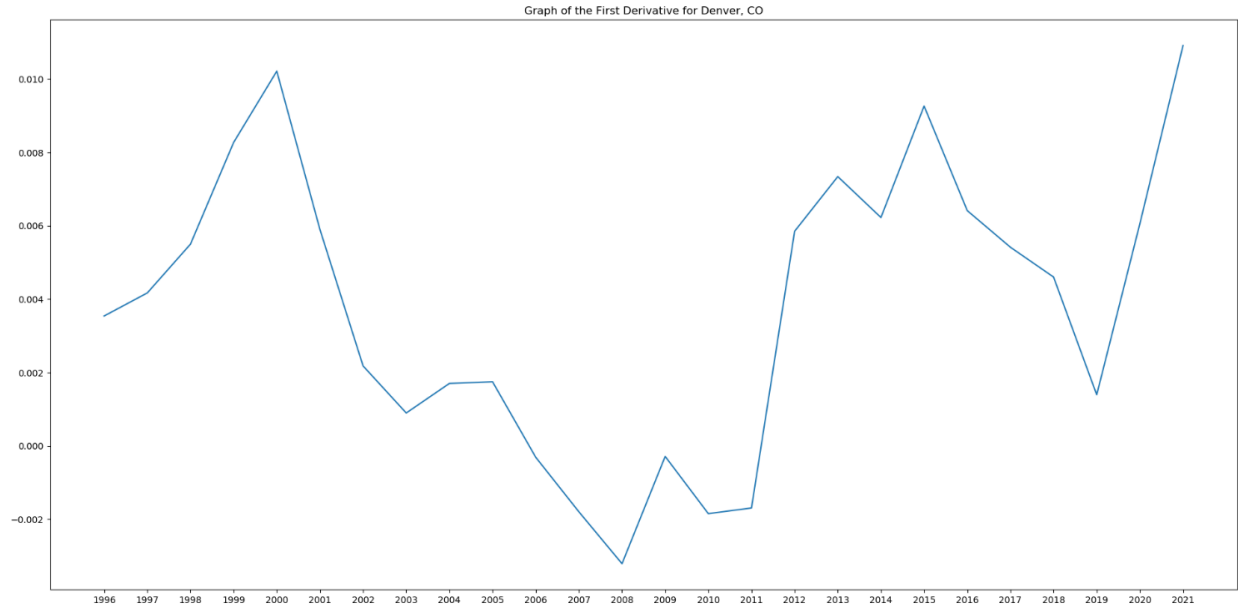
Metropolitan areas in the interior of the country had longer times than metropolitan areas on the coasts. Only 24 percent of metropolitan areas that had a doubling time of less than 10 years were located in non-coastal states. The average doubling time of metropolitan areas in these states was 13.8 years.

Based on the analysis that was performed, several important patterns can be identified. There was a sharp and continued decrease in the median price of single-family homes in many metropolitan areas across the United States in the period 2008-2009. Metropolitan areas in states that bordered the Atlantic Ocean or the Pacific Ocean were significantly more likely to double in price than metropolitan areas in non-coastal states. Doubling times varied widely between states and within certain states. Metropolitan areas that were represented by two Democratic senators or one Republican senator and one Democratic senator doubled much faster on average than

metropolitan areas that were represented by two Republican senators. The *size_rank* of a metropolitan area was not correlated with its doubling time.

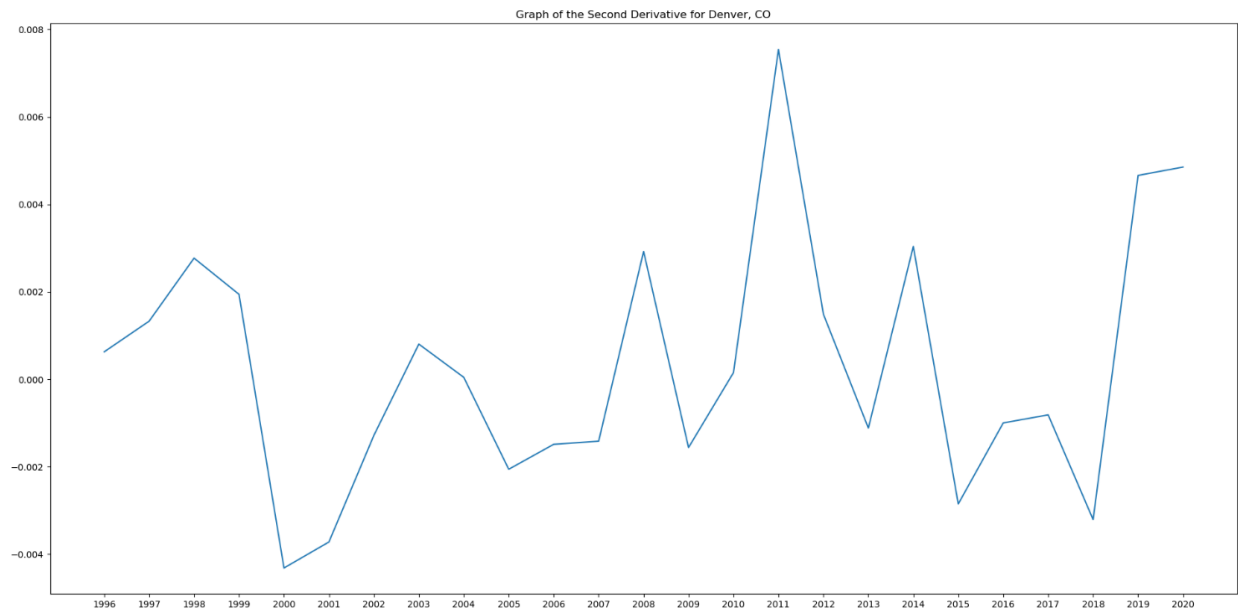
To understand how the rate of appreciation or depreciation in each real estate market changes over time, the first and second derivatives of price with respect to (w.r.t) time were calculated and compared. The first derivative shows whether the prices are increasing or decreasing. It gives the slope of the price function. The second derivative shows whether the change in price is accelerating or decelerating. The second derivative also gives the concavity of the price function. May and Bart explain, “[A] graph is concave up if the line between two points is above the graph, or alternatively if the first derivative is increasing...[A] graph is concave down if the line between two points is below the graph, or alternatively if the first derivative is decreasing. In determining [whether] a curve is concave up or concave down, we want to take the second derivative of a function, or the derivative of the derivative.” (May & Bart, 2022) The first and second derivatives for different metropolitan areas in the dataset were analyzed. The following graphs show the first and second derivatives of normalized price with respect to time for select metropolitan areas.

Figure 2. Graph of the First Derivative for Denver, CO



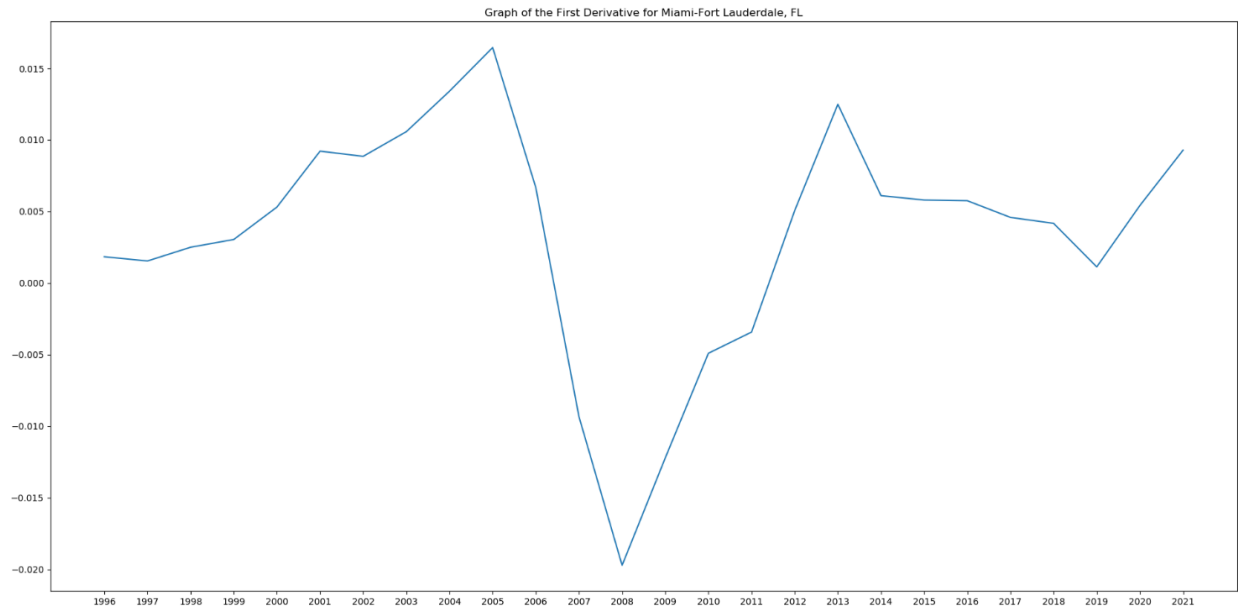
This graph shows the first derivative of the normalized median price of a single-family home in Denver, CO from January 31, 1996 to August 31, 2021. The derivative reached its lowest point in 2008, and it reached its highest point in 2021. Single-family homes in Denver were depreciating in 2007.

Figure 3. Graph of the Second Derivative for Denver, CO



This graph shows the second derivative of the normalized median price of a single-family home in Denver, CO from January 31, 1996 to August 31, 2021. The second derivative changed more abruptly than the first derivative at its local maxima and minima. This is reflected in the fact that the normalized price is still increasing, albeit at slower rate, when the first derivative is positive and the second derivative is negative.

Figure 4. Graph of the First Derivative for Miami, FL



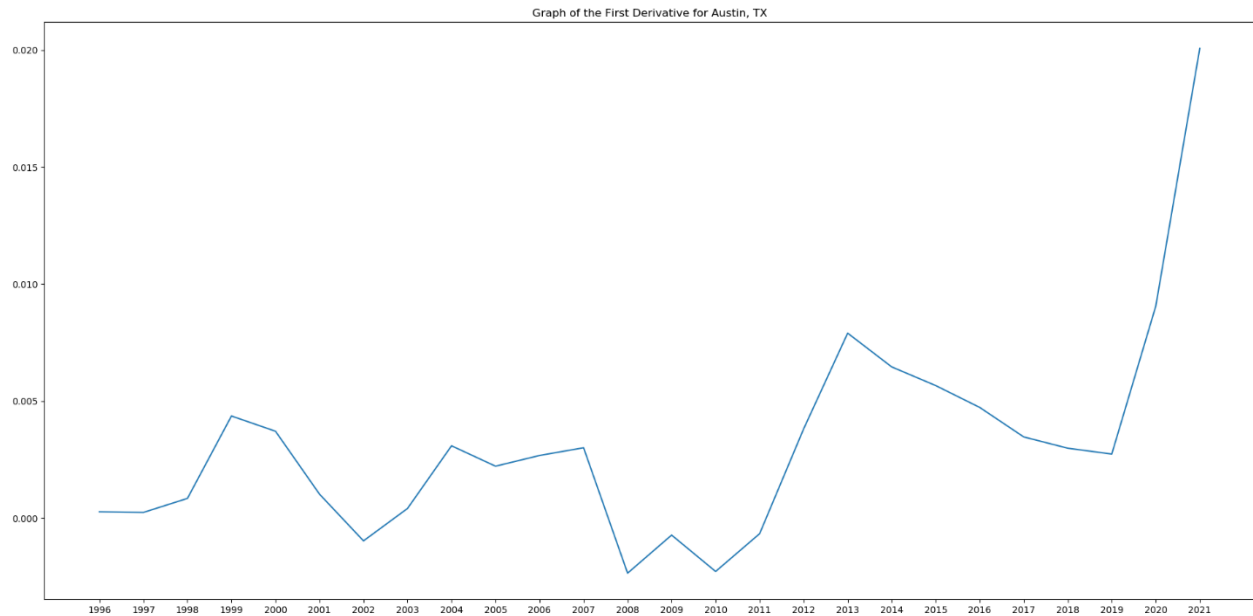
This graph shows the first derivative of the normalized median price of a single-family home in Miami, FL from January 31, 1996 to August 31, 2021. “A housing bubble, or real estate bubble,” has been defined as,” a run-up in housing prices fueled by demand, speculation, and exuberant spending to the point of collapse.” (Investopedia Team, 2020, December 25). The characteristic “V-shape” in the graph is typical of real estate markets that experienced a housing price bubble followed by a housing price crash.

Figure 5. Graph of the Second Derivative for Miami, FL



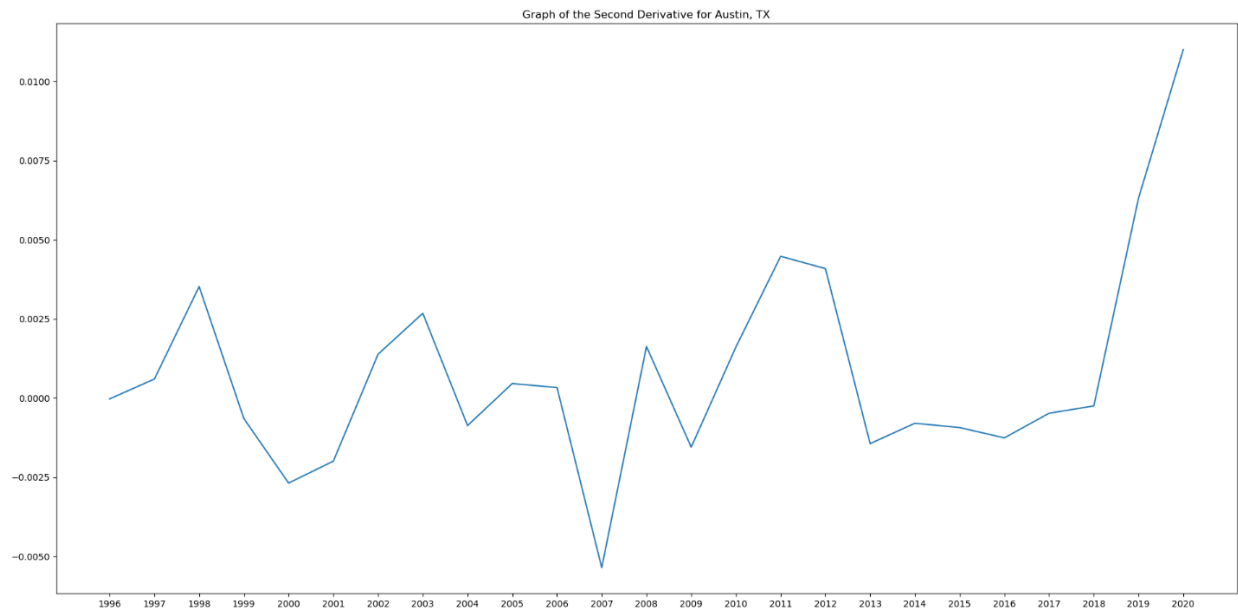
This graph shows the second derivative of the normalized median price of a single-family home in Miami, FL from January 31, 1996 to August 31, 2021. The graph of the second derivative for this metropolitan area is similar to the graph of the first derivative, but it is more staggered even in the recovery phase (the years 2010-2020).

Figure 6. Graph of the First Derivative for Austin, TX



This graph shows the first derivative of the normalized median price of a single-family home in Austin, TX from January 31, 1996 to August 31, 2021. The graph of the first derivative for Austin, TX is more stable than the graph of the first derivative for Miami, FL during the same time, but there is a rapid increase in the normalized median price from 2020-2021. This may indicate that a housing bubble was forming in this metropolitan area.

Figure 7. Graph of the Second Derivative for Austin, TX



This graph shows the second derivative of the normalized median price of a single-family home in Austin, TX from January 31, 1996 to August 31, 2021. From 2018 to 2020, the normalized median price of a single-family home was increasing at an increasing rate in Austin, TX. This provides further evidence that a housing bubble was forming in this metropolitan area at that time.

In all the metropolitan areas displayed, there was a sharp decrease in housing prices during the 2008-2009 recession, also known as the “Great Recession.” For a while, the median price of a single-family home in cyclical markets such as Miami, FL was in “free fall” that is, the price was decreasing at an increasing rate. This can be determined from those points in the dataset at which the first derivative of price with respect to time is negative and the second derivative of price with respect to time is positive.

Coastal states were defined as U.S. states that bordered either the Atlantic or the Pacific Ocean. Non-coastal states were defined as U.S. states that did not border the Atlantic nor the Pacific Ocean. It was observed that the median price of single-family homes grew faster in coastal states than in non-coastal states.

Algorithm 9.

“A housing bubble, or real estate bubble,” has been defined as,” a run-up in housing prices fueled by demand, speculation, and exuberant spending to the point of collapse.” (Investopedia Team, 2020, December 25). To identify when a real estate bubble occurred in a particular region and answer the ninth and tenth research questions, the first and second derivatives were computed for the metropolitan areas in the real estate dataset in order to determine the appreciation rate, or the increase in normalized median price in each metropolitan area. The following thresholds were tested for the increase in normalized median price: 10%, 20%, and 25%, and the error ratio was computed for each threshold. If the real estate prices in a metropolitan area grew as fast as predicted based on the derivative from the previous year(s), then the prediction was considered to be correct. If the real estate prices in a metropolitan area grew much slower or much faster than predicted based on the derivative from the previous year(s), then the prediction was considered to be incorrect. Error ratio is defined as the number of

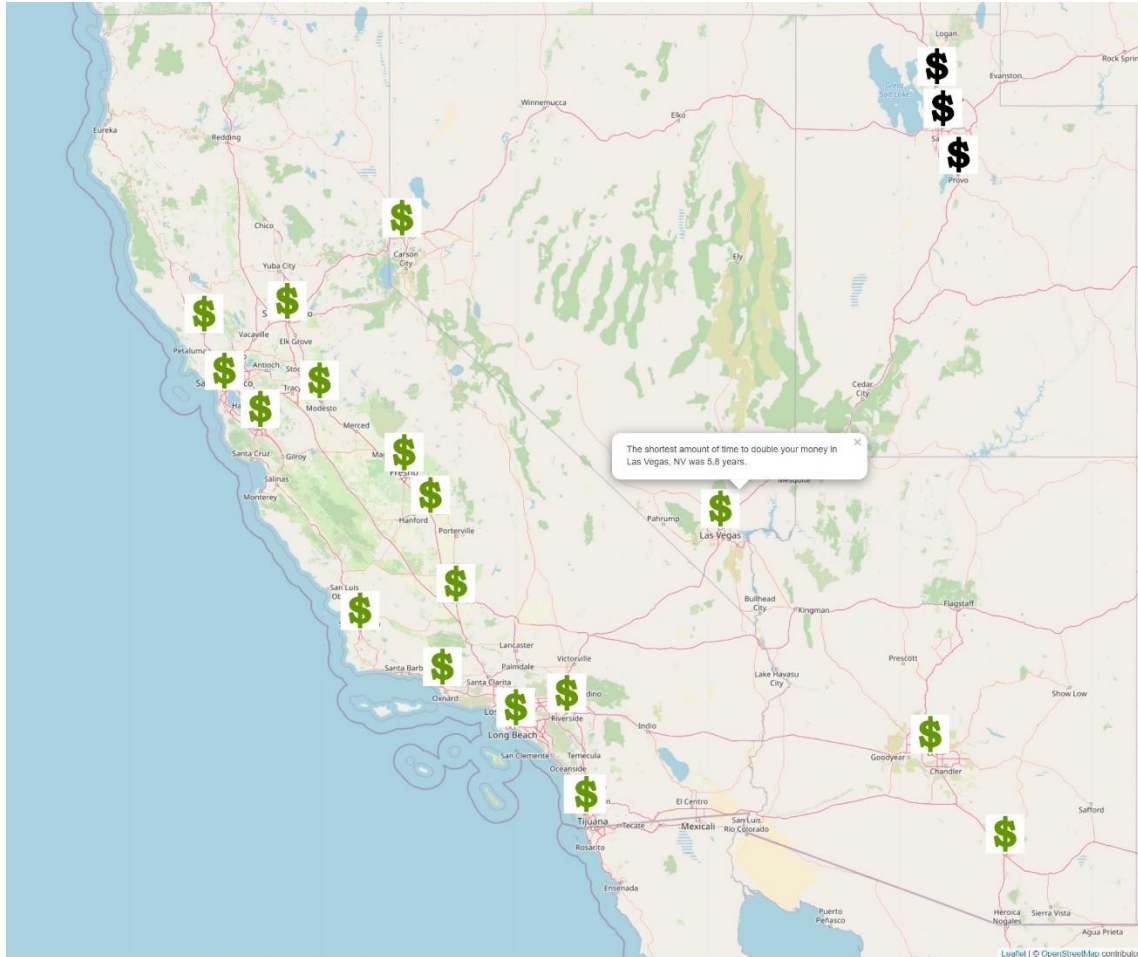
incorrect predictions divided by the total number of predictions. The results of this portion of the analysis are displayed in Table 2.

Table 2. Error Ratios for Different Increase in Median Price Threshold Values

Threshold	Error Ratio
10%	47.0%
20%	50.3%
25%	52.9%

A *bubble* year was defined as a one-year period in which the following two conditions were met: the derivative of normalized price with respect to time in that year was greater than or equal to 120% of the median derivative of normalized price with respect to time in that metropolitan area over the entire 20-year span, and the first and second derivatives were positive. This means that the normalized median price was increasing at an increasing rate. Bubbles may begin or end at any time and may last for more or less than one year (12 months), but one year was a standard measure of time that could be used to compare growth rates in different metropolitan areas within a given region. These indicators were highly correlated with the appearance of real estate bubbles. For example, in the West Coast region, Los Angeles-Long Beach-Anaheim, CA; Riverside, CA; Sacramento, CA; Las Vegas, NV; Reno, NV; and Tucson, AZ all experienced a housing bubble from 2003 to 2005, as shown in Map 4.

Map 4. The Housing Bubble in California, Arizona, and Nevada in the early 2000s



The median price of a single-family home doubled in less than 10 years in the metropolitan areas depicted in green.

It should be noted that the bubble may have begun prior to 2003 or ended after 2005 in specific metropolitan areas. Because the real estate data was reported monthly, it was not possible to determine the exact start and end dates of the bubble. Nearby metropolitan areas (e.g. Phoenix, AZ, San Diego, CA) also experienced a housing bubble at approximately the same time (+/- 1 year). Therefore, it is concluded that there was a housing bubble in the states of California, Arizona, and Nevada in the early-to-mid 2000s.

A similar pattern was seen in Florida from 2001 to 2006. The conditions for a bubble year were met in most large metropolitan areas in Florida in the first half of the 2000s as shown in Map 5. The bubble there usually lasted from 3 to 5 years.

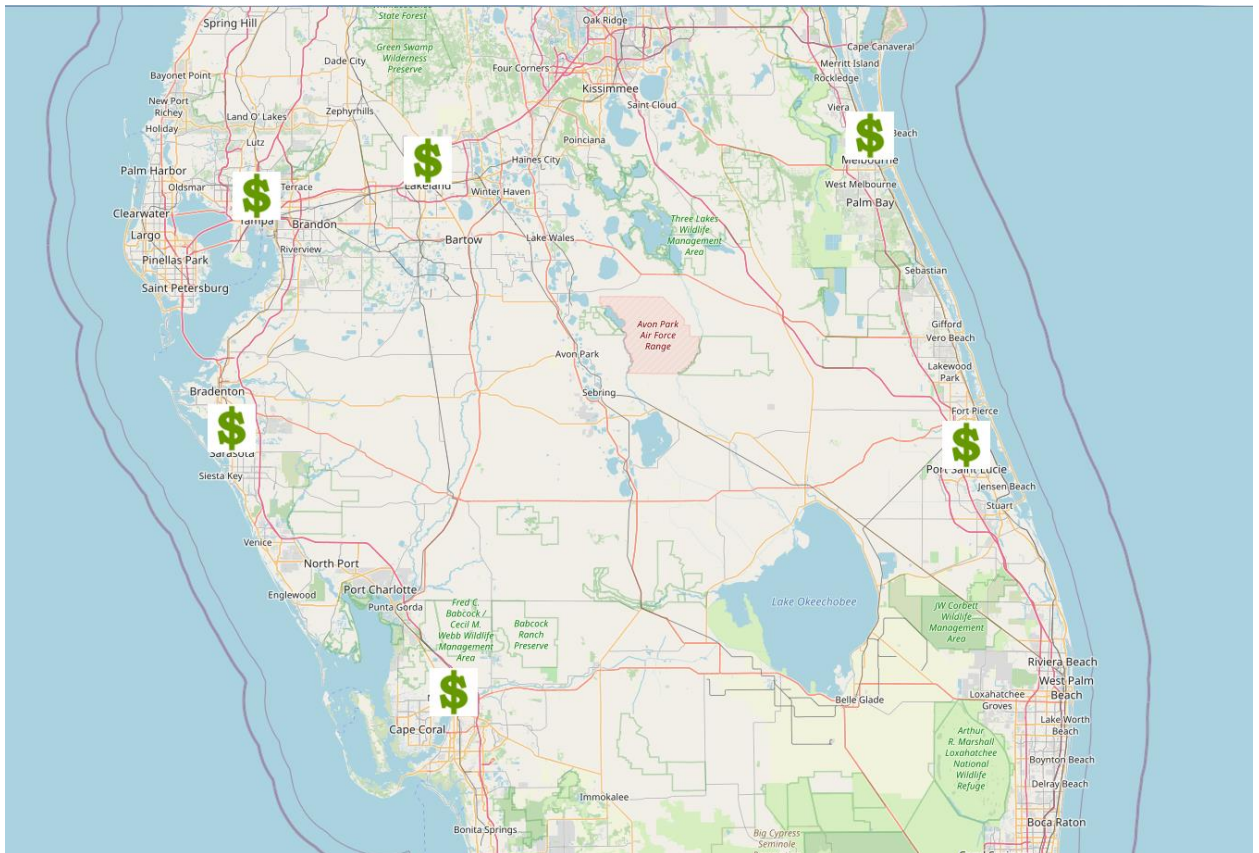
Map 5. The Housing Bubble in Florida in the early 2000s



Housing bubbles occurred in Daytona Beach, Fort Myers, Jacksonville, Melbourne, Miami-Fort Lauderdale, North Port-Sarasota-Bradenton, Orlando, Port St. Lucie, and Tampa, FL in the first half of the 2000s. Pensacola, FL was an outlier. Housing prices there doubled more slowly than they did in peninsular Florida.

Of particular interest were the metropolitan areas in central Florida. This region experienced a sustained housing bubble from 2001 to 2005 as shown in Map 6.

Map 6. The Housing Bubble in Central Florida from 2001 to 2005



There was a housing bubble in Fort Myers, Melbourne, North Port-Sarasota-Bradenton, Port St. Lucie, and Tampa, FL from 2001 to 2005. Lakeland, FL also experienced a housing bubble, but it did not last from 2001 to 2005.

Discussion

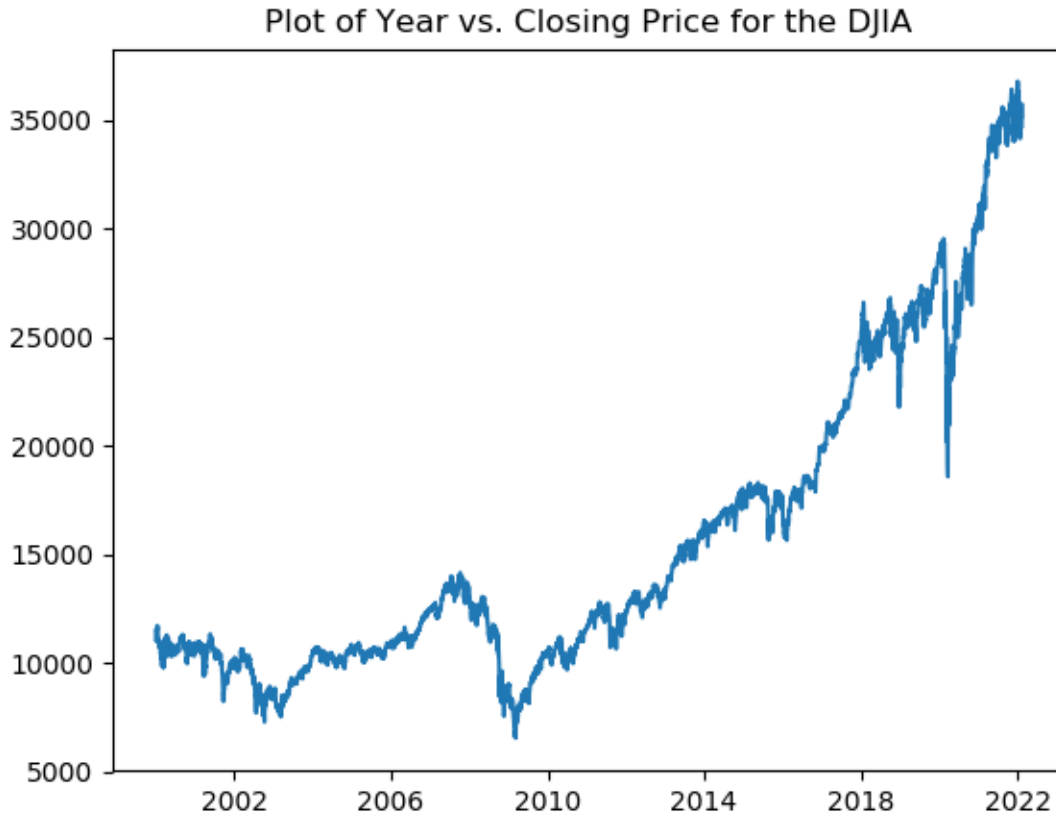
Real estate markets are first and foremost local to a specific metropolitan area and county, but they are also affected by national and regional trends. For example, the real estate bubble in the early-to-mid 2000s affected many metropolitan areas in the West Coast and Florida. This is an example of regional trends in the real estate market.

An asset price bubble may occur in any market: residential real estate, commercial real estate, stocks and bonds, precious metals, etc. For example, an asset price bubble that occurs on a

national stock market or publicly traded exchange will take the form of a stock price bubble (a.k.a. stock market bubble). The characteristics and drivers of such a bubble are often the same as the characteristics and drivers of a real estate bubble (e.g. speculation drives up the price of the asset rapidly).

In the DJIA dataset, each row is a trading day. The columns represent the features: the opening price that day, the highest price that day, the lowest price that day, the closing price that day, the adjusted closing price that day, and the trading volume that day. Ganti observes that the difference between the closing price and the adjusted closing price is, “The adjusted closing price amends a stock's closing price to reflect that stock's value after accounting for any corporate actions.” (Ganti, 2020). For this analysis, the raw closing price was used for calculating derivatives. This raw closing price over time is shown in Figure 8.

Figure 8. Graph of the Daily Closing Price of the Dow Jones Industrial Average



This figure shows the closing prices (in points) of the Dow Jones Industrial Average (DJIA) from December 31, 1999 to February 11, 2022.

Asset price bubbles can be detected by derivative-based analysis in the Dow Jones Industrial Average dataset. To identify when the Dow Jones Industrial Average experienced a stock price bubble, the first and second derivatives of closing price with respect to time and the annual percentage change in the DJIA were computed and compared (Kenton, 2022). The median derivative for the closing price with respect to time was also computed and a threshold of 120% was set, that is, if the derivative of closing price with respect to time was greater than or

equal to 120% of the median derivative of closing price with respect to time across the entire 21-year timeframe, then it was considered a bubble year. The median of the first derivative of price with respect to time was 5.22. The median of the second derivative of price with respect to time was 1.93. The only year in which the first derivative was greater than or equal to 120% of the median derivative was 2022. However, this is likely due to the incomplete data available in the dataset concerning the performance of the DJIA in 2022. The dataset only contains data about the closing price for the DJIA from January 3rd through February 11th, 2022. Also, the DJIA was close to its all-time high of \$36,799.65 during this period. It is likely that if data concerning the performance of the DJIA for the rest of 2022 was included, the first derivative for 2022 would not be greater than or equal to 120% of the median derivative for the 2000-2022 timeframe. Derivative-based analysis, with a threshold of 20% as used in this study, was better at detecting asset price bubbles in the real estate market than in the stock market. But this may be due to the incompleteness of the stock market data provided rather than an inherent weakness in the algorithm itself.

When a large asset price bubble bursts, it sends shockwaves throughout the U.S. economy. But the observed pattern in all three of the economic/financial datasets has been a general upward trend. That is, as time passes (t increases), real estate prices *usually* increase, stock prices *usually* increase, and the price of goods and services *usually* increase. However, it is not a smooth linear increase because real estate and stock prices have also seen temporary, and sometimes sharp, decreases in value, e.g. the 2008-2009 financial crisis and recession.

Part III: Other Methods for Performing Time Series Analysis

As shown in Part I, other methods have been proposed for multivariate time series data analysis and prediction.

As Ferenti explains, “The methods of time series analysis can be very broadly divided into two categories: time-domain and frequency-domain methods. Frequency-domain methods are based on converting the time series, classically using Fourier transform, to a form where the time series is represented as the weighted sum of sinusoids... The vast majority of time series analyses, however, apply time-domain methods.” (Ferenti, 2017)

$$\int_{-\infty}^{\infty} F(k) e^{2\pi i k x} dk$$

Eq. 3. Fourier Transform formula ^[5]

Autocorrelation, cross-correlation, and regression analysis are time domain methods. Two types of regression analysis that are frequently employed on time series data are support vector machines and neural networks.

In this section, these machine learning representations will be explained at a high-level and a prospective approach for how they may be applied to the real estate dataset, the DJIA dataset, and the inflation dataset will be described. Support vector machines are frequently used for classification, but their use for regression has also been well-documented (Bishop, 2006; Ristanoski et al., 2013). A neural network of two or more layers can approximate any function, and this machine learning representation is frequently used for regression tasks also.

Support Vector Machine

In Support Vector Machine regression, a support vector machine is trained by showing it examples of the data that it is supposed to perform regression on (see Figure 9).

Figure 9. High-level Concept of a Support Vector Machine (SVM)

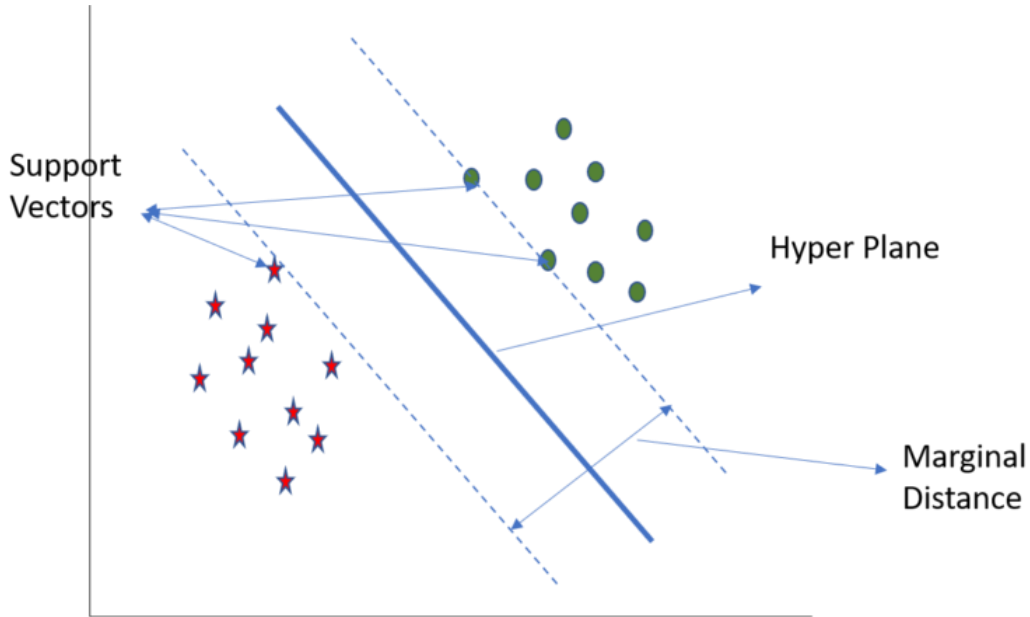


Image Credit: Gaurav Chauhan (Chauhan, 2021) <https://machinelearninghd.com/sklearn-svm-starter-guide/>

For the economic/financial datasets, this data would be the median price of a single-family home in the 911 largest U.S. metropolitan areas from 1996 to 2021 and the closing prices of the Dow Jones Industrial Average from 2000 to 2021. A support vector machine can be trained on these datasets and used to perform regression analysis on them. Support vector machines are resistant to outliers because “the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function ignores samples whose prediction is close to their target,” according to the scikit-learn documentation (Pedregosa et al., 2011).

The data about real estate prices and stock prices may be partitioned according to an 80-10-10 split between the training set, the test set, and the validation set. Lazzeri said, “data scientists usually transform their time series data sets into a supervised learning [problem] by exploiting previous time steps and using them as input and then leveraging the next time step as output of the model.” (Lazzeri, 2020). The svm.SVR module from scikit-learn may be used to make predictions about future real estate and stock prices (Pedregosa et al., 2011).

Neural Network

Neural networks can be used to perform nonlinear regression on a dataset. The same data would be used to train the neural network that was used to train the support vector machine, although the preprocessing steps might differ slightly. The use of neural networks for time series analysis has been well-documented (Katarya & Rastogi, S., 2018; Shterev, Metchkarski, N.S., & Koparanov, K.A., 2022; Ghanbari & Borna 2021; Li, L., Huang, S., Ouyang, Z., & Li, N. 2022; Kasfi, K. T., Hellicar, A., & Rahman, A., 2016). According to Lazzeri, the first step in using a neural network on time series data is to “reframe [the] time series forecasting problem as a supervised learning problem.” (Lazzeri, 2020). Most neural network representations are designed to work on supervised learning problems.

In a neural network there is an input layer, one or more hidden layers, and an output layer as shown in Figure 10.

Figure 10. A Feed-forward Neural Network

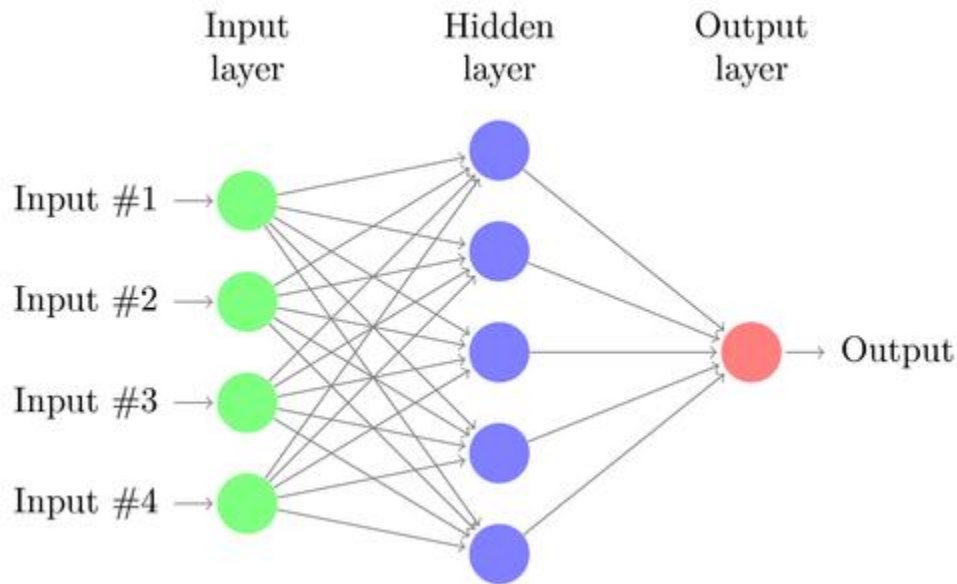


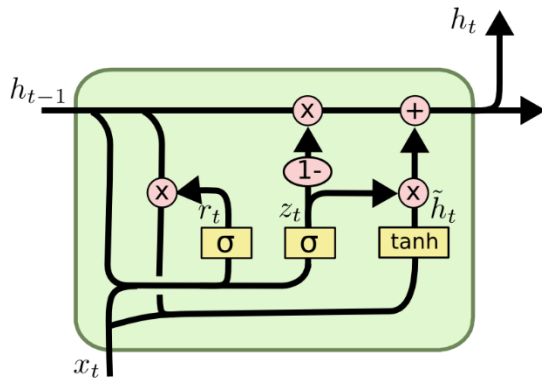
Image Credit: Gaurav Chauhan (Chauhan, 2021) <https://machinelearninghd.com/sklearn-svm-starter-guide/>

“Neural networks learn by adjusting the weights,” (Kamangar, F., personal communication, September 2020). TensorFlow is a framework for training neural network machine learning models. It is based on the concept of a computational graph. Goodfellow et al. said, “In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor” (Goodfellow et al., 2016). TensorFlow performs automatic numerical differentiation by backpropagating partial derivatives of tensors through a computational graph. Keras is an easy-to-use API for TensorFlow. These tools may be used to build neural networks quickly and efficiently.

Two important neural network architectures for regression problems are feed-forward neural networks and long-short term memory (LSTM) neural networks, which are a type of recurrent neural networks (RNN). Russell and Norvig contrast the two architectures as follows:

A **feed-forward network** has connections only in one direction--that is, it forms a directed acyclic graph. Every node receives input from “upstream” nodes and delivers output to “downstream” nodes; there are no loops. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A **recurrent network**, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response to the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory.” (Russell & Norvig, 2009).

Figure 11. Long Term Short Term Memory (LSTM) Architecture



$$z_t = \sigma (W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Image Credit: Christopher Olah (2015, August 27). <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

It is proposed to train a LSTM neural network using TensorFlow and Keras to perform nonlinear regression on the real estate and stock price datasets and the performance be compared to that of the support vector machine using the R^2 -score as the quantitative measure of performance.

Part IV: Time Series Forecasting

Often, time series analysis is performed with the intent of predicting future values that the time series may take, which is called time series forecasting. Baheti & Toshniwal remark, “Identifying potential trends in time series is important because it imparts knowledge about what has taken place in the past and what will take place in time to come.” (Baheti & Toshniwal, 2014). Some of the applications of time series forecasting were discussed in Part I. Now the methods used to perform such forecasting will be discussed in more detail.

As Lazzeri observes, “[T]here are four main categories of components in time series analysis: long-term movement or trend, seasonal short-term movements, cyclical short-term movements, and random or irregular fluctuations.” (Lazzeri, 2020) It has been shown that the real estate dataset and the DJIA dataset both demonstrate an upward trend. This was observed in each of the datasets individually as well as across the approximately 21 years in which they overlap. An example of seasonal short-term movements would be homebuyers buying fewer new homes in the winter and prices decreasing during the winter months (December, January, February) because of decreased demand. More specialized techniques for time series analysis might be able to detect this phenomenon in the real estate dataset, but it was not seen via the derivative-based analysis in this study because the level of granularity was annual rather than monthly. Cyclical short-term movements could be the periodic ups and downs in the closing price of the DJIA. Over long timescales (several years or more) these short-term fluctuations average out. But they are significant to day traders and financial commentators. Random fluctuations could be considered noise in the data. Preprocessing was performed on the datasets to remove missing values and incomplete rows. Potential sources of noise and specific denoising algorithms are beyond the scope of this study.

Part V: Conclusion

Time series analysis and forecasting are important to a wide range of fields across science, industry, technology, politics, economics, and the environment. The methods of time series analysis are diverse and multifaceted. By comparing the derivatives of time series data, researchers can ascertain whether a particular feature (e.g. the price of an asset) is increasing or decreasing in value and the rate at which this change is occurring. But the most valuable insights come as data is aggregated over many experimental units and over a long period of time. Then a general trend in the data points usually emerges. For example, when the economic datasets were analyzed using derivative-based analysis it was observed that the closing price of the Dow Jones Industrial Average increased from \$11,497.12 on Dec 31, 1999 to \$34,738.06 on February 11, 2022. The inflation rate, as measured by the Consumer Price Index fluctuated from a low of 2.7% in December 1999 to a high of 8.5% in March 2022. Therefore, there was a general upward trend in the economic datasets.

Another insight that came out of this study was the importance of normalizing the data prior to performing computations and detailed analysis on it. Normalization is very important when the range of values of the variable of interest in a dataset is large. It permits apples-to-apples comparisons of values in a dataset.

Many representations, methods, and algorithms were researched during the course of this study. Some of them work better on time series data than others do. Sometimes modifying the structure of a representation can significantly improve its performance on time series data. For example, LSTMs have been shown to outperform feed-forward neural networks because the former take into the account the temporal component of the data (Lazzeri, 2020).

Derivative-based analysis was chosen for this study because it was simple to implement, easy to explain, and did not require time-consuming training iterations, massive amounts of training data, nor powerful GPUs. When data is aggregated over many experimental units and over a long period of time, a general trend in the data points usually emerges. Derivative-based analysis works well for identifying trends in the data and inflection points in the functions that model that data. For example, when the first and second derivatives change from being positive to negative, it may indicate that a market correction is imminent. However, derivative-based analysis is primarily a tool for understanding the past and identifying patterns in data, rather than forecasting the future, as it has limited predictive power. It was easier for the program that computed the first and second derivatives to detect when a bubble had occurred, than it was for it to predict when the *next* bubble would occur or how long it would last. Nevertheless, derivative-based analysis is an important and useful tool that data scientists and machine learning practitioners can use to provide new insights into time series data. In the future, this method could be added on to ML algorithms or incorporated as a library function in one of the popular machine learning libraries for Python. Derivative-based analysis may be used as an early-stage exploratory analysis tool to assist ML engineers in understanding their data and choosing more specific forecasting methods (e.g.) LSTMs, ARIMA, or regression algorithms.

Footnotes

^[1] *Index Component Weights of Stocks in the Dow Jones Industrial Average*

indexArb <https://indexarb.com/indexComponentWtsDJ.html>

Accessed 2022, November 19. Copyright 2000-2022 Ergo Inc. All Rights Reserved Worldwide.

Dow JonesSM and Dow Jones Industrial AverageSM are famous, well-known, and internationally recognized trademarks of Dow Jones & Company, Inc. and have been licensed for use by Ergo Inc.

^[2] Weisstein, 2022, “ L^1 -Norm”

^[3] Weisstein, 2022, “ L^1 -Norm”

^[4] <https://leafletjs.com/examples/custom-icons/>

^[5] Weisstein, 2022, “Fourier Transform”

References

A

Abadi, M. (2018, May 10). *Even the US government can't agree on how to divide up the states into regions*. Business Insider. <https://www.businessinsider.com/regions-of-united-states-2018-5>

Agafonkin, V. (2022, September 21). Leaflet.js 1.9. <https://github.com/Leaflet/Leaflet>

B

Baheti, A. & Toshniwal, D. (2014). Trend analysis of time series data using data mining techniques [Abstract], *2014 IEEE International Congress on Big Data*, 430-437, doi: 10.1109/BigData.Congress.2014.69.

Bawaneh M. & Simon, V. (2019). Anomaly detection in smart city traffic based on time series analysis [Abstract], *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 1-6, doi: 10.23919/SOFTCOM.2019.8903822.

Bishop, C. (2006). *Pattern recognition and machine learning*. Springer.

Brownlee, J. (2018, April 27). *How to Calculate Correlation Between Variables in Python*. Machine Learning Mastery. <https://machinelearningmastery.com/how-to-use-correlation-to-understand-the-relationship-between-variables/>

Brownlee, J. (2020, November 4). *Curve fitting with Python*. Machine Learning Mastery. <https://machinelearningmastery.com/curve-fitting-with-python/>

C

Chauhan, G. (2021, March 16). *Sklearn svm – Starter Guide* Machine Learning HD. <https://machinelearninghd.com/sklearn-svm-starter-guide/>

Roesel, D. (2017, October 24). *Curve fitting in Python using SciPy and Matplotlib*.

David.roesel.cz. <https://david.roesel.cz/notes/posts/curve-fitting-python-scipy-matplotlib/>

California Department of Public Health (2022, November 8). *COVID-19 Time-Series Metrics by County and State*. CHHS Open Data. <https://data.chhs.ca.gov/dataset/covid-19-time-series-metrics-by-county-and-state>

D

Dollar sign icon. Retrieved from October 26, 2021 from

<https://www.publicdomainpictures.net/en/view-image.php?image=82321&picture=dollar-sign>

E

F

Farwaji, S. (2020, May 13). *What is the Rule of 72 in Real Estate*. Mashvisor.com.

<https://www.mashvisor.com/blog/rule-of-72/>

Ferenti, T. (2017). Biomedical applications of time series analysis [Abstract], *IEEE 30th Neumann Colloquium (NC)*, 000083-000084, doi: 10.1109/NC.2017.8263256.

Federal Communications Commission (2016, August 25). *Degrees Minutes Seconds to/from Decimal Degrees*. Retrieved November 8, 2022 from

<https://www.fcc.gov/media/radio/dms-decimal>

G

Ganti, A. (2020, December 28). *Adjusted Closing Price*. Investopedia.

https://www.investopedia.com/terms/a/adjusted_closing_price.asp

Ganti, A. (2022, October 5). *What is the Dow Jones Industrial Average (DJIA)?* Investopedia.

<https://www.investopedia.com/terms/d/djia.asp>

Ghanbari R. & Borna K. (2021). Multivariate time-series prediction using LSTM neural networks. *26th International Computer Conference, Computer Society of Iran (CSICC)*, 1-5, doi: 10.1109/CSICC52343.2021.9420543.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Gupta, N. (2022). *Capital Appreciation*. Wall Street Mojo.

<https://www.wallstreetmojo.com/capital-appreciation/>

H

Hagan, M. T., Demuth, H. B., Beale, M. H., & De Jesús, O. (2014). *Neural Network Design* (2nd ed). Martin T. Hagan and Howard B. Demuth.

Hages, C. (2020, September 26). *How to: Numerical derivative in Python*. YouTube.

<https://www.youtube.com/watch?v=utRKIIlOZbtw>

Hall, M. (2022, January 27). *What Does the Dow Jones Industrial Average Measure?*

Investopedia. <https://www.investopedia.com/ask/answers/050115/what-does-dow-jones-industrial-average-measure.asp>

Hayes, A. (2022, June 12) *What Is a Time Series and How Is It Used to Analyze Data?*

Investopedia. <https://www.investopedia.com/terms/t/timeseries.asp>

Historical Inflation Rates: 1914-2022. (2022, November 8) US Inflation Calculator. Retrieved

November 8, 2022 from <https://www.usinflationcalculator.com/inflation/historical-inflation-rates/>

I

Ibrahim, S. A., Charlson, M. E., & Neill, D. B. (2020). Big Data Analytics and the Struggle for Equity in Health Care: The Promise and Perils. *Health equity*, 4(1), 99–101.

<https://doi.org/10.1089/heq.2019.0112>

Investopedia Team. (2020, December 25). *What Is the Housing Bubble? Definition, Causes and Recent Example*. Investopedia.com.

https://www.investopedia.com/terms/h/housing_bubble.asp

J

K

Kamangar, F. (2020, September). *CSE 5368 Neural Networks*. CSE Department. The University of Texas at Arlington.

Kasirat, K.T., Hellicar, A., & Rahman A. (2016). Convolutional neural network for time series cattle behaviour classification. *Proceedings of the Workshop on Time Series Analytics and Applications (TSAA '16)*. Association for Computing Machinery, New York, NY, USA, 8–12. <https://doi-org.ezproxy.uta.edu/10.1145/3014340.3014342>

Katarya, R. & Rastogi, S., (2018). A study on neural networks approach to time-series analysis. *2nd International Conference on Inventive Systems and Control (ICISC)*, 116-119, doi: 10.1109/ICISC.2018.8399024.

Kenton, W. *Percentage changes and how to calculate them*. (2022, August 31). Investopedia. <https://www.investopedia.com/terms/p/percentage-change.asp>

Kremer, H., Gunnemann, S. & Seidl, T. (2010) Detecting Climate Change in Multivariate Time Series Data by Novel Clustering and Cluster Tracing Techniques [Abstract]. *IEEE International Conference on Data Mining Workshops*, 96-97, doi: 10.1109/ICDMW.2010.39.

L

- Lazzeri, F. (2020). *Machine learning for time series forecasting with Python*. Hoboken, NJ, United States, Wiley, 2020.
- Leusin, Y. (2017, December 17). *How do you calculate rate of return on investment in real estate?* Mashvisor. <https://www.mashvisor.com/blog/calculate-rate-of-return-on-investment-real-estate/>
- Li, L., Huang, S., Ouyang Z., & Li., N. (2022). A deep learning framework for non-stationary time series prediction. *3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*, 2022, pp. 339-342, doi: 10.1109/CVIDLICCEA56201.2022.9824863.
- Lynn, Shane. (2022). *Read CSV data quickly into Pandas DataFrames with read_csv*. Retrieved November 8, 2022 from <https://www.shanelynn.ie/python-pandas-read-csv-load-data-from-csv-files/>
- Lynn, Shane. (2022). *Pandas iloc and loc – quickly select rows and columns in dataframes*. Retrieved November 8, 2022 from <https://www.shanelynn.ie/pandas-iloc-loc-select-rows-and-columns-dataframe/>
- M**
- Maina, S. (2021, October 31). *Filter a Pandas dataframe by a partial string or pattern in 8 ways*. Towards Data Science. <https://towardsdatascience.com/8-ways-to-filter-a-pandas-dataframe-by-a-partial-string-or-pattern-49f43279c50f>
- May, M & Bart, A. (2022, October 8). *4.5 The second derivative and concavity*. Business Calculus with Excel. <https://mathstat.slu.edu/~may/ExcelCalculus/sec-4-5-SecondDerivativeConcavity.html>

Moreno, A. (2020, July 20). *Data normalization with Pandas and Scikit-Learn: The complete guide to clean datasets — Part 1*. Towards Data Science

<https://towardsdatascience.com/data-normalization-with-pandas-and-scikit-learn-7c1cc6ed6475>

N

Nielsen, B. (2022, September 3). *What Causes a Real Estate Bubble?* Investopedia.com.

https://www.investopedia.com/articles/07/housing_bubble.asp

NOAA. (2022, September 1). *Weather*. National Oceanic and Atmospheric Administration.

<https://www.noaa.gov/weather>.

Ntlangu, M. B., & Baghai-Wadji, A. (2017). Modelling Network Traffic Using Time

Series Analysis: A Review [Abstract]. *Proceedings of the International Conference on Big Data and Internet of Thing (BDIOT2017)*. Association for Computing Machinery, New York, NY, USA, 209–215. <https://doi-org.ezproxy.uta.edu/10.1145/3175684.3175725>

O

Olah, C. (2015, August 27). *Understanding LSTM Networks*. Colah's blog.

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Oladunni, T., Denis, M., Ososanya, E., & Adesina, J. (2021, May 18). A Time Series Analysis and Forecast of COVID-19 Healthcare Disparity. medRxiv. doi:

<https://doi.org/10.1101/2021.05.13.21257189>

P

Pandas documentation. 2022. *Getting Started*. NumFOCUS, Inc. Retrieved November 8, 2022

from https://pandas.pydata.org/docs/getting_started/intro_tutorials/03_subset_data.html

Pandas documentation. 2022. *Merge, join, concatenate, and compare*. User Guide. NumFOCUS, Inc. Retrieved November 8, 2022 from https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Pandas documentation. 2022. *Indexing and selecting data*. User Guide. NumFOCUS, Inc. Retrieved November 8, 2022 from https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html

Pandas documentation. 2022. *pandas.DataFrame.to_numpy*. API reference. NumFOCUS, Inc. Retrieved November 8, 2022 from https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_numpy.html

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Duchesnay, E. (2011). [Scikit-learn: Machine Learning in Python](#) *JMLR* 12, pp. 2825-2830, 2011.

Pathak, P. (2020, May 17). *How to generate lat and long coordinates of city without using APIS in Python*. Medium.com. <https://medium.com/analytics-vidhya/how-to-generate-lat-and-long-coordinates-of-city-without-using-apis-25ebabcaf1d5>

Q

R

Ristanoski, G., Liu, W., & Bailey, J. (2013). A time-dependent enhanced support vector machine for time series regression [Abstract]. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '13)*.

Association for Computing Machinery, New York, NY, USA, 946–954. <https://doi-org.ezproxy.uta.edu/10.1145/2487575.2487655>

Russell, S. J., Norvig, P. (2009). *Artificial intelligence: A modern approach (3rd ed)*. Pearson.

S

Shen, F. K., Zhang W., & Chang, P. (2009). An Engineering Approach to Prediction of Network Traffic Based on Time-Series Model [Abstract]. *2009 International Joint Conference on Artificial Intelligence*, 432-435, doi: 10.1109/JCAI.2009.104.

Shivam, K. (2021, June 29). *Python / Pandas Series.abs()* Geeks for Geeks.

<https://www.geeksforgeeks.org/python-pandas-series-abs/>

Shterev, V. A., Metchkarski, N. S., & Koparanov, K. A. (2022). Time Series Prediction with Neural Networks: a Review. *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, 1-4, doi: 10.1109/ICEST55168.2022.9828735.

Smith, A. *How to print an entire pandas dataframe in Python*. Retrieved from November 8, 2022 from <https://www.adamsmith.haus/python/answers/how-to-print-an-entire-pandas-dataframe-in-python>

Stewart, M. (2019, February 26). *Simple Introduction to Convolutional Neural Networks*.

Towards Data Science. <https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac>

Stips., A.K., Macias, D., Coughlan, C., and E. Garcia-Gorriz, (2015). Global climate change: Analyzing anthropogenic warming and causality [Abstract], *OCEANS 2015 - Genova, 2015*, 1-4, doi: 10.1109/OCEANS-Genova.2015.7271393.

T

Time Series. (2022, November 8) stats.oecd.org. Retrieved November 8, 2022 from

<https://stats.oecd.org/glossary/detail.asp?ID=2708>

Trend. (2022, November 8) stats.oecd.org. Retrieved November 8, 2022 from

<https://stats.oecd.org/glossary/detail.asp?ID=2769>

The Coding Team. (2019, May 2019). *1.5 Mapping Geolocation with Leaflet.js - Working with Data and APIs in JavaScript*. YouTube.

<https://www.youtube.com/watch?v=nZaZ2dB6pow>

U

U.S. Census Bureau. (2022, May 16) *City and Town Population Totals: 2020-2021*. census.gov

<https://www.census.gov/data/tables/time-series/demo/popest/2020s-total-cities-and-towns.html>

U.S. Const. art. I, § 2.

V

W

Wang, J. (2020, April 5). *How To Analyse A Single Time Series Variable: Time Series Modeling*

With Python Code. Towards Data Science. [https://towardsdatascience.com/how-to-](https://towardsdatascience.com/how-to-analyse-a-single-time-series-variable-11dcca7bf16c)

[analyse-a-single-time-series-variable-11dcca7bf16c](https://towardsdatascience.com/how-to-analyse-a-single-time-series-variable-11dcca7bf16c)

Weisstein, E. W. 2022. *Fourier Transform*. From MathWorld--A Wolfram Web Resource.

<https://mathworld.wolfram.com/FourierTransform.html>

Weisstein, E. W. 2022. *L^1 -Norm*. From MathWorld--A Wolfram Web Resource.

<https://mathworld.wolfram.com/L1-Norm.html>

Weisstein, E. W. 2022. *L^2 -Norm*. From MathWorld--A Wolfram Web Resource.

<https://mathworld.wolfram.com/L2-Norm.html>

Wu, F., Zhong, Y., & Wu, Q., (2008). Network Threat Frequency Forecasting Based on Fuzzy Time Series and D-S Evidence Theory [Abstract]. *2008 International Conference on Computer Science and Software Engineering*, 754-757, doi: 10.1109/CSSE.2008.284.

Wu, S. (2020, May 23). *3 Best metrics to evaluate regression model?* Towards Data Science. <https://towardsdatascience.com/what-are-the-best-metrics-to-evaluate-your-regression-model-418ca481755b>

X

Y

Dow Jones Industrial Average (^DJI). (2022, December 1). yahoo! finance.

<https://finance.yahoo.com/quote/%5EDJI/history/>

Z

Zeger, S. L., Irizarry, R., & Peng, R. D. (2006). On time series analysis of public health and biomedical data. *Annual review of public health*, 27, 57–79.

<https://doi.org/10.1146/annurev.publhealth.26.021304.144517>

Zillow. 2021. *Housing Data: ZHVI Single-Family Homes Time Series Metro & U.S.* [Data set].

Zillow.com. <https://www.zillow.com/research/data/>

Python Coding References

Retrieved [November 19, 2022] from

<http://www2.gcc.edu/dept/math/faculty/BancroftED/buscalc/chapter2/section2-6.php>

Retrieved [November 19, 2022] from

<https://appdividend.com/2020/11/02/python-string-append/>

Retrieved [November 19, 2022] from

<https://blog.container-solutions.com/understanding-volumes-docker>

Retrieved [November 19, 2022] from

<https://bobbyhadz.com/blog/python-valueerror-could-not-convert-string-to-float>

Retrieved [November 19, 2022] from

<https://boscacci.medium.com/why-and-how-to-make-a-requirements-txt-f329c685181e>

Retrieved [November 19, 2022] from

https://colab.research.google.com/drive/1UCJt8EYjlzCs1H1d1X0iDGYJsHKwu-NO#scrollTo=RX2SB_2O1jx7

Retrieved [November 19, 2022] from

<https://datascienceparichay.com/article/get-column-names-as-list-in-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://datatofish.com/numpy-array-to-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://geeksforgeeks.org/python-check-if-a-file-or-directory-exists-2/>

Retrieved [November 19, 2022] from

<https://gis.stackexchange.com/questions/54405/is-there-a-way-to-populate-a-spreadsheet-of-city-names-with-their-latitudes-and>

Retrieved [November 19, 2022] from

https://howtothink.readthedocs.io/en/latest/PvL_H.html

Retrieved [November 19, 2022] from

<https://itsmycode.com/python-valueerror-could-not-convert-string-to-float/>

Retrieved [November 19, 2022] from

<https://numpy.org/doc/stable/reference/generated/numpy.flip.html>

Retrieved [November 19, 2022] from

<https://numpy.org/doc/stable/reference/generated/numpy.subtract.html>

Retrieved [November 19, 2022] from

<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.append.html>

Retrieved [November 19, 2022] from

<https://pandas.pydata.org/docs/reference/api/pandas.Series.divide.html>

Retrieved [November 19, 2022] from

<https://pandas.pydata.org/docs/reference/api/pandas.Series.idxmin.html>

Retrieved [November 19, 2022] from

<https://pandas.pydata.org/docs/reference/api/pandas.Series.max.html>

Retrieved [November 19, 2022] from

https://pandas.pydata.org/docs/reference/api/pandas.Series.sort_values.html

Retrieved [November 19, 2022] from

<https://pandas.pydata.org/docs/reference/api/pandas.Timestamp.html>

Retrieved [November 19, 2022] from

<https://pynative.com/python-datetime-to-seconds/>

Retrieved [November 19, 2022] from

<https://pythonguides.com/matplotlib-time-series-plot/>

Retrieved [November 19, 2022] from

<https://pythonspeed.com/articles/activate-virtualenv-dockerfile/>

Retrieved [November 19, 2022] from

https://raw.githubusercontent.com/CodingTrain/Intro-to-Data-APIs-JS/source/module1/03_fetch_json/index.html

Retrieved [November 19, 2022] from

<https://realpython.com/numpy-scipy-pandas-correlation-python/>

Retrieved [November 19, 2022] from

<https://realpython.com/python-wheels/#python-packaging-made-better-an-intro-to-python-wheels>

Retrieved [November 19, 2022] from

<https://runnable.com/docker/python/dockerize-your-python-application>

Retrieved [November 19, 2022] from

<https://sparkbyexamples.com/pandas/print-pandas-dataframe-without-index/>

Retrieved [November 19, 2022] from

<https://stackabuse.com/python-get-number-of-elements-in-a-list/>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/53513/how-do-i-check-if-a-list-is-empty>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/151199/how-to-calculate-number-of-days-between-two-given-dates>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/466345/converting-string-into-datetime>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/493386/how-to-print-without-a-newline-or-space>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/7152762/how-to-redirect-print-output-to-a-file>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/9159757/can-i-add-comments-to-a-pip-requirements-file>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/11285613/selecting-multiple-columns-in-a-pandas-dataframe>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/16476924/how-to-iterate-over-rows-in-a-dataframe-in-pandas>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/17071871/how-do-i-select-rows-from-a-dataframe-based-on-column-values>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/20444087/right-way-to-reverse-a-pandas-dataframe>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/21285380/find-column-whose-name-contains-a-specific-string>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/22341271/get-list-from-pandas-dataframe-column-or-row>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/24101524/finding-median-of-list-in-python>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/25852044/converting-pandas-tslib-timestamp-to-datetime-python>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/31037298/pandas-get-column-average-mean>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/31929538/how-to-subtract-datetimes-timestamps-in-python>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/33246771/convert-pandas-data-frame-to-series>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/33814887/python-pandas-printing-out-values-of-each-cells>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/33834727/how-do-you-extract-only-the-date-from-a-python-datetime>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/37093454/typeerror-must-be-string-not-datetime-datetime-when-using-strptime>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/38387529/how-to-iterate-over-pandas-series-generated-from-groupby-size>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/42286972/converting-from-pandas-dataframe-to-tensorflow-tensor-object>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/56947333/how-to-remove-commas-from-all-the-column-in-pandas-at-once>

Retrieved [November 19, 2022] from

<https://stackoverflow.com/questions/68357090/getting-an-error-saying-could-not-build-wheels-for-numpy-which-use-pep-517-and>

Retrieved [November 19, 2022] from

<https://thispointer.com/drop-first-row-of-pandas-dataframe-3-ways/>

Retrieved [November 19, 2022] from

<https://thispointer.com/how-to-create-and-initialize-a-list-of-lists-in-python/>

Retrieved [November 19, 2022] from

<https://thispointer.com/python-how-to-get-last-n-characters-in-a-string/>

Retrieved [November 19, 2022] from

<https://www.activestate.com/resources/quick-reads/how-to-access-an-element-in-pandas/>

Retrieved [November 19, 2022] from

<https://www.askpython.com/python/string/check-string-contains-substring-python>

Retrieved [November 19, 2022] from

<https://www.codegrepper.com/code-examples/python/python+color+text+on+windows>

Retrieved [November 19, 2022] from

<https://www.digitalocean.com/community/tutorials/how-to-build-and-deploy-a-flask-application-using-docker-on-ubuntu-20-04>

Retrieved [November 19, 2022] from

<https://www.digitalocean.com/community/tutorials/how-to-install-nginx-on-ubuntu-18-04>

Retrieved [November 19, 2022] from

<https://www.freecodecamp.org/news/how-to-substring-a-string-in-python/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/accessing-elements-of-a-pandas-series/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/check-if-element-exists-in-list-in-python/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/convert-the-column-type-from-string-to-datetime-format-in-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/create-a-pandas-dataframe-from-lists/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/different-ways-to-iterate-over-rows-in-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/drop-rows-from-pandas-dataframe-with-missing-values-or-nan-in-columns/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/find-maximum-values-position-in-columns-and-rows-of-a-dataframe-in-pandas/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/floor-ceil-function-python/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/get-column-index-from-column-name-of-a-given-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/how-to-add-one-row-in-an-existing-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/how-to-convert-datetime-to-integer-in-python/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/how-to-get-column-names-in-pandas-dataframe/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-broadcasting-with-numpy-arrays/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-dictionary/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-get-key-from-value-in-dictionary/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-numpy-nanmean-function/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-program-convert-string-list/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-program-to-convert-a-list-to-string/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-program-to-sort-a-list-of-tuples-by-second-item/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-removing-first-element-of-list/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/python-string-upper/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/sort-in-python/>

Retrieved [November 19, 2022] from

<https://www.geeksforgeeks.org/ways-to-filter-pandas-dataframe-by-column-values/>

Retrieved [November 19, 2022] from

<https://www.golinuxcloud.com/pandas-convert-column-to-float/>

Retrieved [November 19, 2022] from

<https://www.kite.com/python/answers/how-to-find-the-max-value-of-a-pandas-dataframe-column-in-python>

Retrieved [November 19, 2022] from

<https://www.kite.com/python/answers/how-to-get-the-part-of-a-string-before-a-specific-character-in-python>

Retrieved [November 19, 2022] from

<https://www.kite.com/python/answers/how-to-print-a-float-with-two-decimal-places-in-python>

Retrieved [November 19, 2022] from

<https://www.kite.com/python/answers/how-to-write-to-an-html-file-in-python>

Retrieved [November 19, 2022] from

<https://www.linode.com/docs/guides/how-to-use-dockerfiles/>

Retrieved [November 19, 2022] from

<https://www.marsja.se/pandas-convert-column-to-datetime/>

Retrieved [November 19, 2022] from

<https://www.pluralsight.com/guides/how-to-use-gitignore-file>

Retrieved [November 19, 2022] from

<https://www.programiz.com/python-programming/methods/list/index>

Retrieved [November 19, 2022] from

<https://www.section.io/engineering-education/how-to-share-data-between-a-docker-container-and-the-host-computer/#step-1-lets-make-a-directory-where-we-will-mount-with-the-container>

Retrieved [November 19, 2022] from

<https://www.statology.org/pandas-filter-rows-containing-string/>

Retrieved [November 19, 2022] from

https://www.tensorflow.org/tutorials/load_data/pandas_dataframe

Retrieved [November 19, 2022] from

<https://www.tutorialspoint.com/how-to-set-x-axis-values-in-matplotlib-python>

Retrieved [November 19, 2022] from

https://www.w3schools.com/js/js_object_properties.asp

Retrieved [November 19, 2022] from

https://www.w3schools.com/jsref/jsref_if.asp

Retrieved [November 19, 2022] from

https://www.w3schools.com/python/gloss_python_escape_characters.asp

Retrieved [November 19, 2022] from

https://www.w3schools.com/python/matplotlib_labels.asp

Retrieved [November 19, 2022] from

https://www.w3schools.com/python/ref_string_split.asp

Slide 1

Data Discovery Analysis on Complex Time Series Data

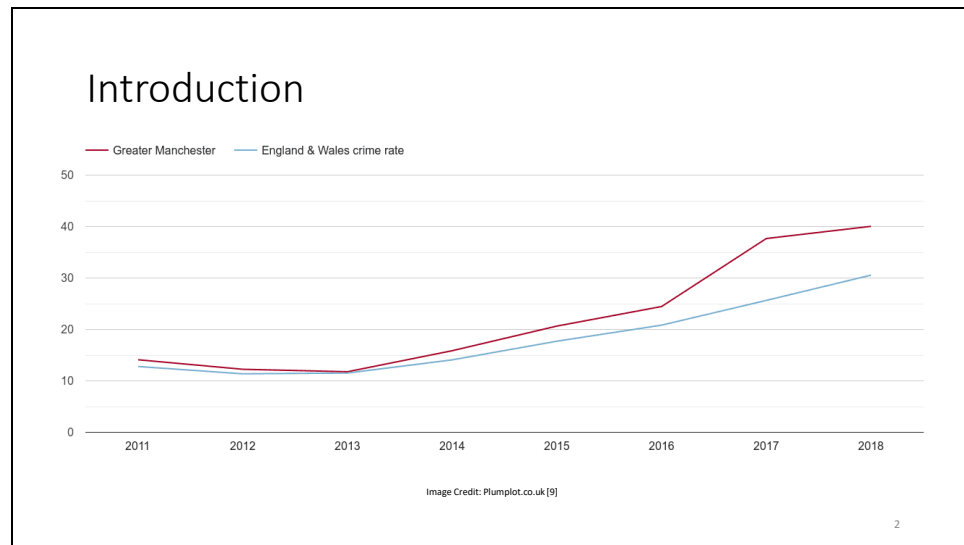
Peter L. Severynen

Department of Computer Science and Engineering

The University of Texas at Arlington

December 2, 2022

Slide 2



-> Complex time series are a ubiquitous form of data in the modern world. They have wide application across many different fields of scientific inquiry and business endeavor. ->

Applications of Time Series

- weather patterns
- climate change
- voting patterns
- computer network traffic
- healthcare data
- demographics

3

Time series are used to understand and forecast -> weather patterns, -> climate change, -> voting patterns, -> computer network traffic, -> population health outcomes, -> demographic changes ->

Applications of Time Series

- scientific observations
- economic data

4

-> the results of scientific experiments, and -> the performance of stocks and mutual funds. But time series can be difficult to analyze by conventional methods when the data is multivariate, incomplete, or in different formats. To address these issues, an investigation of several multivariate time series datasets was performed using the methods of automatic data discovery and derivative-based analysis. Interactive maps were constructed which displayed the results of the study. Conclusions were drawn and discussed, and an explanation was given of how this method can be applied to other multivariate time series datasets and real-world problems. ->

Slide 5

Applications of Time Series: Weather forecasting

- Input: measurements of temperature, precipitation, humidity, air pressure, etc
- Processing: NOAA supercomputers
- Output: detailed weather forecasts



NOAA supercomputers used for weather forecasting

Image Credit: NOAA/NOAA News September 8, 2009 [1]

5

-> This atmospheric data constitutes time series. -> These time series are aggregated and fed as input to supercomputers run by the National Oceanic and Atmospheric Administration (NOAA), a division of the U.S. Department of Commerce. -> These supercomputers generate weather forecasts, but accurate weather forecasts are more than just a convenience.

Slide 6

Applications of Time Series: Weather forecasting

Significance of Accurate Forecasts

- Hurricane Ian and Hurricane Nicole devastated Florida in 2022
- Thousands of lives were saved because the National Weather Service accurately predicted the paths of the storms and residents in the affected areas evacuated



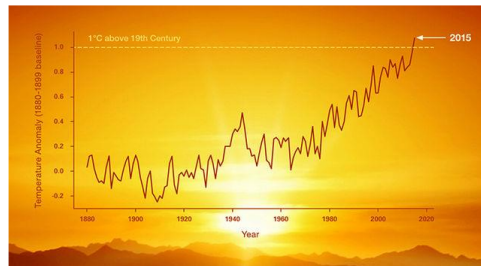
Hurricane Ian in 2022

Image Credit: NOAA Geostationary Operational Environmental Satellites (GOES) [2] 6

-> Each year, the United States averages some 10,000 thunderstorms, 5,000 floods, 1,300 tornadoes and -> 2 Atlantic hurricanes, as well as widespread droughts and wildfires. Weather, water and climate events, cause an average of approximately 650 deaths and \$15 billion in damage per year, and are responsible for some -> 90 percent of all presidentially-declared disasters. About one-third of the U.S. economy – some \$3 trillion – is sensitive to weather and climate. The scientific consensus is that extreme weather events such as these are growing in frequency and severity due to climate change. ->

Applications of Time Series: Climate change

- Scientists use time series analysis to understand the origin and evolution of climate change
- Earth's global mean surface temperature can be modeled as a time series.
- This temperature has been rising since the 1930s.



Global temperature anomaly from 1880 to 2015

Image credit: NASA/JPL-Caltech [3]

7

-> -> Climate change can be modeled using time series ->

Applications of Time Series: Climate change

- This is due to the greenhouse effect.
- The atmospheric concentration of CO₂ and other greenhouse gases (GHGs) can be modeled as a time series.
- Kremer et al. used a clustering technique to detect climate change in multivariate time series data from hydrology, meteorology, and oceanography (Kremer et al., 2010).

The greenhouse effect

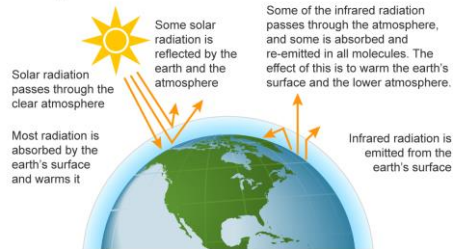


Image Credit: U.S. Environmental Protection Agency [4]

8

-> As human beings pump carbon dioxide gas into the Earth's atmosphere by burning fossil fuels, -> this carbon dioxide absorbs infrared radiation and re-emits it, -> warming the planet's surface and increasing the global mean surface temperature. ->

Applications of Time Series: Climate change

- It was constructed in 1935 as a reservoir for Hoover Dam
- The water level has dropped dramatically since 1983



Lake Mead in 2010

Image Credit: Cmpxchg8b, [5]

9

-> This is Lake Mead. -> It shows the visible effects of climate change. -> -> I went there when I was a child, and the water level was above the so-called “bathtub ring” of white mineralized -> rock around the shore of the lake. The water level in Lake Mead may be represented as a time series

Slide 10

Applications of Time Series: Climate change

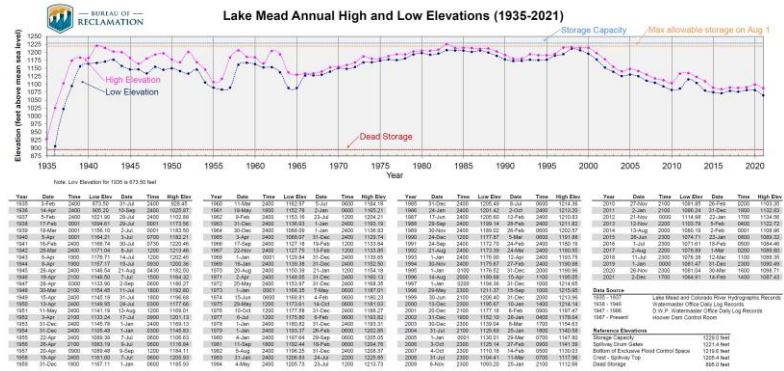


Image Credit: U.S. Bureau of Reclamation [6]

10

This time series is shown here. ->

Applications of Time Series: Voting Patterns

- Time series analysis is also helpful in understanding voting patterns and how they change over time.
- The party affiliation of the winning candidate in successive elections forms a time series.
- By analyzing these time series for each congressional district, it can be determined which political party is favored in that district.
- This information is used to target campaign advertising and plan voter mobilization efforts

11

Applications of Time Series: Network Traffic

- Time series analysis is used to model computer network traffic, congestion, and cyberattacks.
- By monitoring the number of packets traveling across a network node at predefined intervals, it can be determined whether the network is congested at that node and will ultimately slow down.
- An anomalous and sudden increase in traffic from a particular subnet or range of IP addresses may indicate a distributed denial of service (DDoS) attack on a particular server or Web site.
- Time series help network security professionals detect and prepare for such cyberattacks.

12

Applications of Time Series: Public Health Policy

- The number of cases of an infectious disease, the number of positive tests, and the number of deaths attributable to the disease over time are all important factors that may be represented as time series.
- State public health departments routinely publish datasets of COVID-19 cases, tests, and deaths.
- Models and analyses based on these time series datasets inform decisions on public health measures that are designed to reduce transmission and prevent the healthcare system from becoming overburdened.
- Timely, comprehensive data and accurate, high-quality analysis are prerequisites for sound and effective public health policy.

Applications of Time Series: Economic Data

- Many kinds of economic data may also be represented as time series:
- Residential real estate prices in large U.S. metropolitan areas
- The price of the Dow Jones Industrial Average
- The Consumer Price Index (CPI)

14

For this analysis, I will focus on three types of economic data

The Real Estate Dataset



Single-family home

Image Credit: Sanjib Lemar

- The Zillow Single-Family Homes Time Series Metro & U.S. dataset describes the monthly sale price of single-family homes in the 911 largest U.S. metropolitan areas from January 1996 to August 2021.

15

This data was collected by the real estate company Zillow and obtained from its website, [zillow.com](https://www.zillow.com). A single-family home is a free-standing residential structure designed to be occupied by one group of biologically-related individuals. It is the preferred type of housing for most middle-class and upper-middle class families in the United States.

Slide 16

The Real Estate Dataset

There are five features in this dataset in addition to the monthly prices in the metropolitan areas:

- RegionID
- SizeRank
- RegionName
- RegionType
- StateName

RegionID	RegionName	RegionType	StateName	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19	A20	A21	A22	A23	A24	A25	A26	A27	A28	A29	A30	A31	A32	A33	A34	A35	A36
1	New York, NY	MSA	NY	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000
2	New Jersey, NJ	MSA	NJ	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000
3	Connecticut, CT	MSA	CT	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000

-> RegionID is a unique identifier assigned to each metropolitan area. -> SizeRank is the relative population ranking of the metropolitan area in the dataset. -> RegionName is the name of the metropolitan area. A metropolitan area may consist of one city or multiple cities in the same geographical area. -> RegionType shows whether the data in that row is for the whole country or a single metropolitan statistical area (MSA). -> StateName is the name of the state where the metropolitan area is located. The median price of a single-family home in that metropolitan area each month is listed in the columns on the right. ->

About the DJIA

The Dow Jones Industrial Average (DJIA) is “a price-weighted index that tracks 30 large...[publicly-traded] companies trading on the New York Stock Exchange and the Nasdaq...[and it] serve[s] as a proxy for the broader U.S. economy” (Ganti, 2022).

No.	Stock	Rank	% Weight in the Index	Bar Graph of % Weight in the Index
1	3M	21	2.49	<div></div>
2	American Express	14	2.99	<div></div>
3	Amgen	4	5.55	<div></div>
4	Apple	16	2.90	<div></div>
5	Boeing	13	3.33	<div></div>
6	Caterpillar	7	4.59	<div></div>
7	Chevron	11	3.59	<div></div>
8	Cisco Systems	27	0.94	<div></div>
9	Coca-Cola	25	1.21	<div></div>
10	Disney	24	1.86	<div></div>
11	Dow	26	1.00	<div></div>
12	Goldman Sachs	2	7.42	<div></div>
13	Home Depot	3	6.28	<div></div>
14	Honeywell International	8	4.23	<div></div>
15	IBM	18	2.88	<div></div>
16	Intel	30	0.58	<div></div>
17	Johnson & Johnson	12	3.42	<div></div>
18	JPMorgan Chase	20	2.61	<div></div>
19	McDonalds	5	5.31	<div></div>
20	Merck	22	2.07	<div></div>
21	Microsoft	6	4.74	<div></div>
22	NIKE B	23	2.05	<div></div>
23	Procter & Gamble	19	2.81	<div></div>
24	Salesforce	17	2.88	<div></div>
25	Travelers	10	3.62	<div></div>
26	UnitedHealth Group	1	10.11	<div></div>
27	Verizon Communications	29	0.76	<div></div>
28	Visa A	9	4.07	<div></div>
29	Walgreens Boots Alliance	28	0.81	<div></div>
30	Walmart	15	2.93	<div></div>

Here we see the current components of the Dow Jones Industrial Average and their weights in the index. The three largest components by weight are: UnitedHealth Group (10.11%), Goldman Sachs (7.42%), and Home Depot (6.28%). Dow Chemical is 1% of the index. These weights are assigned and regularly updated by Dow Jones and Company, Inc, which publishes the Dow Jones Industrial Average.

Slide 18

The DJIA dataset

There are seven features in this dataset:

- Date
- Open
- High
- Low
- Close*
- Adj Close**
- Volume

	A	B	C	D	E	F	G	H
1	Date	Open	High	Low	Close*	Adj Close**	Volume	
2	11-Feb-22	35,267.89	35,431.15	34,620.52	34,738.06	34,738.06	398,860,000	
3	10-Feb-22	35,630.81	35,800.24	35,100.72	35,241.59	35,241.59	411,620,000	
4	9-Feb-22	35,614.90	35,824.28	35,614.90	35,768.06	35,768.06	333,610,000	
5	8-Feb-22	35,160.68	35,544.89	35,090.42	35,462.78	35,462.78	319,190,000	
6	7-Feb-22	35,108.38	35,325.01	34,993.98	35,091.13	35,091.13	328,910,000	
7	4-Feb-22	35,095.74	35,333.55	34,799.08	35,089.74	35,089.74	349,880,000	
8	3-Feb-22	35,520.08	35,535.94	35,071.06	35,111.16	35,111.16	381,020,000	
9	2-Feb-22	35,378.19	35,679.20	35,290.12	35,629.33	35,629.33	359,540,000	
10	1-Feb-22	35,151.47	35,441.09	34,977.95	35,405.24	35,405.24	386,080,000	
11	31-Jan-22	34,691.17	35,148.14	34,496.10	35,131.86	35,131.86	468,070,000	
12	28-Jan-22	34,135.24	34,731.77	33,807.51	34,725.47	34,725.47	568,440,000	
13	27-Jan-22	34,261.75	34,773.32	34,007.78	34,160.78	34,160.78	527,780,000	
14	26-Jan-22	34,520.82	34,815.67	33,876.48	34,168.09	34,168.09	546,330,000	
15	25-Jan-22	34,186.64	34,591.04	33,545.52	34,297.73	34,297.73	506,430,000	
16	24-Jan-22	34,070.61	34,420.99	33,150.33	34,364.50	34,364.50	664,010,000	
17	21-Jan-22	34,701.69	34,896.67	34,229.55	34,265.37	34,265.37	523,880,000	
18	20-Jan-22	35,102.66	35,490.20	34,670.12	34,715.39	34,715.39	369,070,000	

-> Date is the trading day on which this data was recorded. -> Open is the price of the DJIA at the opening of the stock exchange on that trading day. -> High is the highest price that it saw on that trading day. -> Low is the lowest price that it saw on that trading day. -> Close* is the price of the DJIA when the market closed, not accounting for any corporate actions, such as stock splits. -> Adjusted Close is the price after accounting for corporate actions. -> Volume is the number of shares traded that day. ->

The Inflation Dataset

- The Consumer Price Index dataset lists the inflation rate as measured by the Consumer Price Index (CPI) for each month from January 1996 to December 2021.

Image Credit: Jernej Furman [8]



Inflation has hit record highs for the past two years and has become a major political issue.

Slide 20

The Consumer Price Index (CPI) Dataset

- The inflation rate is given as a percentage change in prices for each month in the dataset.
- The *Monthly_Average* column shows the average inflation rate for each year

Year	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	Monthly_Average
1996	2.7	2.7	2.8	2.9	2.9	2.8	3	2.9	3	3	3.3	3	3
1997	3	3	2.8	2.5	2.2	2.3	2.2	2.2	2.2	2.1	1.8	1.7	2.3
1998	1.6	1.4	1.4	1.4	1.7	1.7	1.7	1.6	1.5	1.5	1.5	1.6	1.6
1999	1.7	1.6	1.7	2.3	2.1	2	2.1	2.3	2.6	2.6	2.6	2.7	2.2
2000	2.7	3.2	3.8	3.1	3.2	3.7	3.7	3.4	3.5	3.4	3.4	3.4	3.4
2001	3.7	3.5	2.9	3.3	3.6	3.2	2.7	2.7	2.6	2.1	1.9	1.6	2.8
2002	1.1	1.1	1.5	1.6	1.2	1.1	1.5	1.6	1.5	2	2.2	2.4	1.6
2003	2.6	3	3	2.2	2.1	2.1	2.1	2.2	2.3	2	1.8	1.9	2.3
2004	1.9	1.7	1.7	2.3	3.1	3.3	3	2.7	2.5	3.2	3.5	3.3	2.7
2005	3	3	3.1	3.5	2.8	2.5	3.2	3.6	4.7	4.3	3.5	3.4	3.4
2006	4	3.6	3.4	3.5	4.2	4.3	4.1	3.8	2.1	1.3	2	2.5	3.2
2007	2.1	2.4	2.8	2.6	2.7	2.7	2.4	2	2.8	3.5	4.3	4.1	2.8
2008	4.3	4	4	3.9	4.2	5	5.6	5.4	4.9	3.7	1.1	0.1	3.8
2009	0	0.2	-0.4	-0.7	-1.3	-1.4	-2.1	-1.5	-1.3	-0.2	1.8	2.7	-0.4
2010	2.6	2.1	2.3	2.2	2	1.1	1.2	1.1	1.1	1.2	1.1	1.5	1.6
2011	1.6	2.1	2.7	3.2	3.6	3.6	3.6	3.8	3.9	3.5	3.4	3	3.2
2012	2.9	2.9	2.7	2.3	1.7	1.7	1.4	1.7	2	2.2	1.9	1.7	2.1
2013	1.6	2	1.5	1.1	1.4	1.8	2	1.5	1.2	1	1.2	1.5	1.5
2014	1.6	1.1	1.5	2	2.1	2.1	2	1.7	1.7	1.7	1.3	0.8	1.6
2015	-0.1	0	-0.1	-0.2	0	0.1	0.2	0.2	0	0.2	0.5	0.7	0.1
2016	1.4	1	0.9	1.1	1	1	0.8	1.1	1.5	1.6	1.7	2.1	1.3
2017	2.5	2.7	2.4	2.2	1.9	1.6	1.7	1.9	2.2	2	2.2	2.1	2.1
2018	2.1	2.2	2.4	2.5	2.8	2.9	2.9	2.7	2.3	2.5	2.2	1.9	2.4
2019	1.6	1.5	1.9	2	1.8	1.6	1.8	1.7	1.7	1.8	2.1	2.3	1.8
2020	2.5	2.3	1.5	0.3	0.1	0.6	1	1.3	1.4	1.2	1.2	1.4	1.2
2021	1.4	1.7	2.6	4.2	5	5.4	5.4	5.3	5.4	6.2	6.8	7	4.7
2022	7.5	7.9	8.5										

Research Questions

- “Given two metropolitan areas, which one had a greater increase in median price over the entire 24-year period?”
- “In a given metropolitan area, how many years will it take for the median price for a single-family home to double?”
- “Given the price data for a metropolitan area for 2021, will prices in that metropolitan area rise or fall in the next year?”

21

Several research questions were posed for this analysis, but I will focus on the last three.

Research Questions (cont.)

- “Is there a relationship between doubling time and the political party affiliation of the senators that represented the metropolitan areas and their respective states during the years in which the median price of a single-family home doubled?”
- “Is there a statistically significant correlation between doubling time and the population size of the metropolitan area, i.e. do smaller metropolitan areas double in price faster than larger ones?”
- “What is the regional distribution of the metropolitan areas that doubled in price, e.g. did metropolitan areas on the East and West Coasts of the United States double in price faster than metropolitan areas in the interior of the country?”
- “Can it be determined from the available data when there is a housing price bubble in a particular metropolitan area or region?”
- “How can a housing price bubble be defined mathematically?”

22

These three questions are important because they give insight into the regional trends that were occurring in the U.S. residential housing market at that time. Each method I tested made a unique contribution to the overall analysis. First, I implemented the following algorithms to answer the specific questions, and these gave me insight into the general trends.

Features from the Real Estate Dataset

- time
- median price
- normalized median price
- state
- region

23

These features were chosen to compare metropolitan areas from the real estate dataset because they were the most informative. Normalized median price and region are derived features.

Development of Methods and Algorithms

24

In order to answer the research questions, several methods and algorithms were developed.

Development of Methods and Algorithms

Algorithm 1

- 1) partition the dataset into user-specified equal length time periods
- 2) For a user-defined period
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the specified timespan
 - b) $\text{change in price} = \text{final price} - \text{initial price}$
 - c) store the name of the metropolitan area and the change in its normalized price in a data structure
- 3) sort the metropolitan areas in descending order by change in price
- 4) print the user-specified number of metropolitan areas and their respective increases in normalized price

25

-> In order to determine which metropolitan areas appreciated the fastest in a given time period, the following algorithm was developed. -> I partitioned the dataset into time periods. -> I normalized the prices in each time period. -> I computed the change in price -> I sorted the metropolitan areas by the change in price. -> And I printed the results for the user. ->

Development of Methods and Algorithms

Algorithm 2

- 1) set a user-defined threshold for the maximum initial starting price
- 2) filter the dataset based on the threshold
- 3) for each metropolitan area under the threshold
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the entire timespan of the dataset
 - b) $\text{change in price} = \text{final price} - \text{initial price}$
 - c) store the name of the metropolitan area and the change in its normalized price in a data structure
- 4) sort the metropolitan areas in descending order by change in price
- 5) return the name of the metropolitan area with the highest increase in normalized price and its change in price

26

-> To find which city offered the best return on investment over the timespan of the real estate dataset, -> I filtered the cities based on the desired investment amount. -> I normalized the prices. -> I computed the change in price -> I sorted the metropolitan areas, by the change in price. -> And I printed the results to the user. ->

Development of Methods and Algorithms

Algorithm 3

- 1) for each metropolitan area in the dataset
 - a) normalize the prices by dividing each price by the maximum price for its metropolitan area over the entire timespan of the dataset
 - b) $\text{change in price} = \text{final price} - \text{initial price}$
 - c) store the name of the metropolitan area and the change in its normalized price in a data structure
- 2) sort the metropolitan areas in descending order by change in price
- 3) define two variables to store the increase in normalized price for the two user-specified metropolitan areas
- 4) iterate through the sorted list of metropolitan areas, updating each metropolitan area's price variable
- 5) choose the metropolitan area that had the highest increase in normalized price over the entire timespan of the dataset
- 6) return the names and the respective increases in normalized price for the two metropolitan areas

27

In order to determine which of a PAIR of metropolitan areas had a greater increase in median price over the timespan of the dataset, a similar algorithm was used. The main difference in the two algorithms is that TWO variables were used to store the increase in normalized price the increases in normalized price for both metropolitan areas was returned.

Development of Methods and Algorithms

Algorithm 4

- 1) find the minimum monthly price for a user-specified metropolitan area
- 2) iterate through the succeeding columns until a price greater than or equal to twice the minimum price is found
- 3) count the number of months that occurred between these two price points
- 4) divide the number of months by 12 to obtain the doubling time in years
- 5) round the doubling time in years to the desired precision

28

To determine the doubling time for a metropolitan area,

Development of Methods and Algorithms

Algorithm 5

- 1) compute the derivative of normalized price with respect to time for each year in the dataset for a user-specified metropolitan area
- 2) compute the median derivative for that metropolitan area over the timespan of the dataset
- 3) If the derivative of price with respect to time in the current year was *more than 20% above* the median derivative for that metropolitan area over the past 24 years, then the next year was predicted to be a good (above average) year for price increases in that metropolitan area.
- 4) If the derivative of price with respect to time in the current year was *more than 20% below* the median derivative for that metropolitan area over the past 24 years, then the next year was predicted to be a poor (below average) year for price increases in that metropolitan area.
- 5) If the derivative of price with respect to time in the current year was *between 80% and 120%* of the median derivative for that metropolitan area over the past 24 years, then the next year was predicted to be a fair (average) year for price increases in that metropolitan area.

29

-> To determine whether prices in a given metropolitan area would rise or fall in the next year, -> I normalized the prices -> took derivatives -> and compared the derivative for the current year -> with the median derivative for that metropolitan area. ->

Results of the Analysis

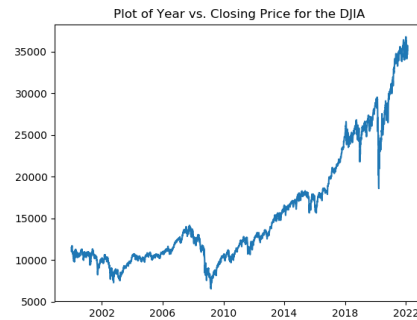
- The DJIA Dataset
- The CPI Dataset
- The Real Estate Dataset
 - Interactive maps of the 120 largest metropolitan areas
 - Special focus area: The West Coast in 2003-2005
 - Special focus area: Florida in 2001-2006

30

The results of this analysis, will be broken into five parts. -> First, the results of the DJIA Dataset will be discussed. -> Then the results of the CPI Dataset will be discussed. -> -> Next, the results of the analysis on the real estate will be discussed with a special focus on the West Coast and Florida in the first half of the 2000s.

Results of the Analysis

- There was a general upward trend in the closing price of the DJIA punctuated by large drops in 2009 and 2020.



31

This is a graph of the closing price of the Dow Jones Industrial average from 2000 to 2022. As you can see, there is a general upward trend in the price as time increases. Also notice the two dramatic drops in the price in 2009 and 2020. The former was caused by the 2008-2009 recession, also known as the “Great Recession.” The latter occurred during the first year of the COVID-19 pandemic.

Results of the Analysis

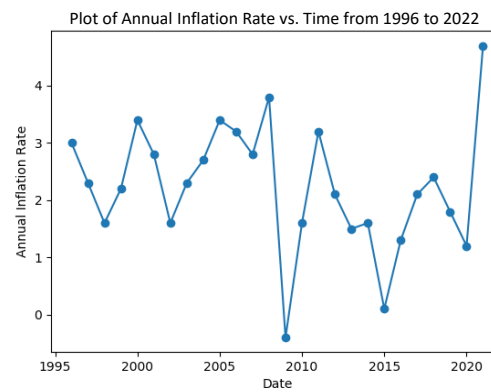
- There was a general upward trend in the closing price of the DJIA punctuated by large drops in 2009 and 2020.



32

This is a graph of the closing price of the Dow Jones Industrial average from 2000 to 2022. As you can see, there is a general upward trend in the price as time increases. Also notice the two dramatic drops in the price in 2009 and 2020. The former was caused by the 2008-2009 recession, also known as the “Great Recession.” The latter occurred during the first year of the COVID-19 pandemic.

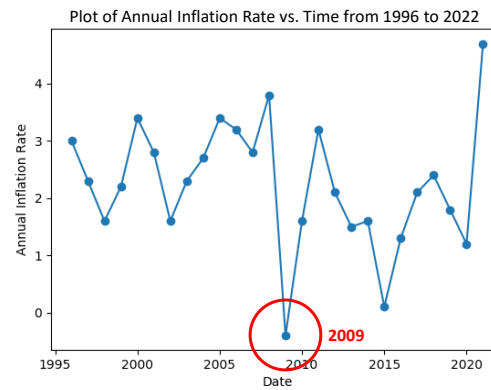
Results of the Analysis



33

This graph shows the Annual Inflation Rate as measured by the Consumer Price Index

Results of the Analysis



34

As you can see from the graph, the annual inflation rate fell precipitously in 2009. Also, the annual inflation rate has risen sharply since 2020.

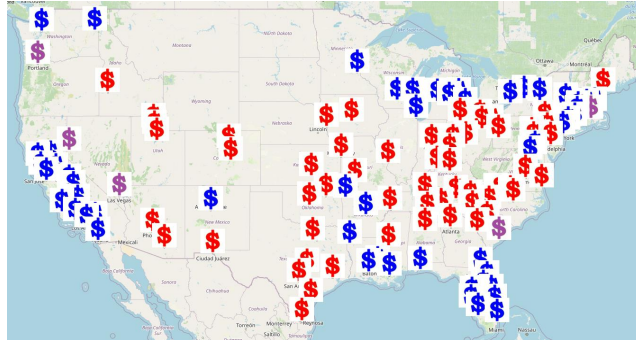
The 120 Largest U.S. Metropolitan Areas Grouped by the Time to Double



35

This map shows the largest 120 U.S. metropolitan areas by population grouped by the time for the normalized median price of a single-family home in that metropolitan area to double. The metropolitan areas denoted by the green dollar sign icon doubled in price the fastest, and the metropolitan areas denoted by the black dollar sign icon doubled in price the slowest.

The 120 Largest U.S. Metropolitan Areas Grouped by Their Senate Representation



36

This map shows the largest 120 U.S. metropolitan areas by population grouped by the political affiliation of their state's U.S. Senate delegation. Red means two Republican senators, blue means two Democratic senators, and purple means one Republican senator and one Democratic senator.

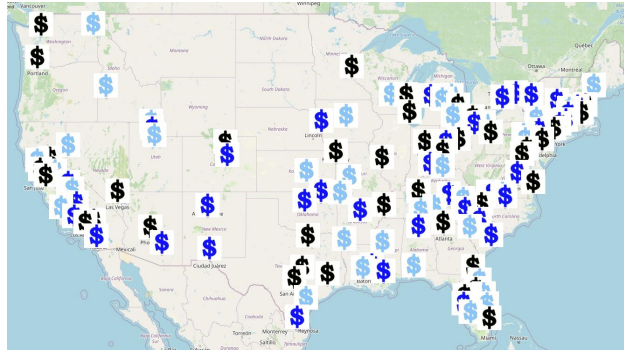
Average Doubling Time of the Metropolitan Areas by Political Party of their Senate Representation

Political Representation in the U.S. Senate	Average Doubling Time in Years
Two Democratic senators	11.2
Two Republican senators	20.7
One Democratic and one Republican senator	7.7

37

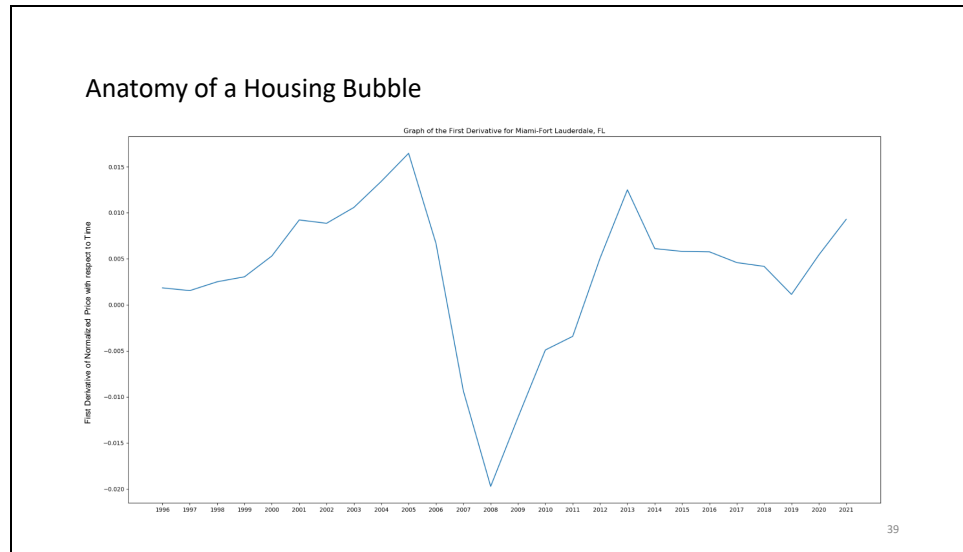
This table shows the average time for the normalized median price of a single-family home to double in the 120 largest U.S. metropolitan areas compared to the political affiliation of their Senate representation. The average doubling time of the metropolitan areas represented by Democratic and Republican senators (7.7) was lower than either the average doubling time of the metropolitan areas represented by Democratic senators (11.2 years), or the metropolitan areas represented by Republican senators (20.7 years). While the average doubling time for metropolitan areas represented by Republican senators was almost twice as long as the average doubling time of the metropolitan areas represented by Democratic senators.

The 120 Largest U.S. Metropolitan Areas Grouped by Population



38

This map shows the largest 120 U.S. metropolitan areas by population grouped by the population size of the metropolitan area. The metropolitan areas denoted by the black dollar sign icon were the most populous, and the metropolitan areas denoted by the light blue dollar sign icon were the least populous.

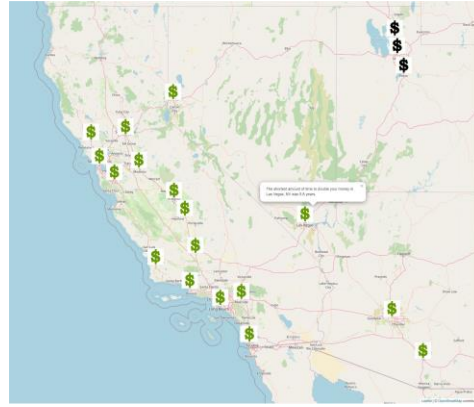


“A housing bubble, or real estate bubble,” has been defined as,” a run-up in housing prices fueled by demand, speculation, and exuberant spending to the point of collapse.” This graph shows the first derivative of the normalized median price of a single-family home in Miami, FL from January 31, 1996 to August 31, 2021. The characteristic “V-shape” in the graph is typical of real estate markets that experienced a housing bubble.

The Housing Bubble on the West Coast

Housing bubbles occurred in the following metropolitan areas in the West Coast region between 2002 and 2006:

- Las Vegas, NV
- Los Angeles-Long Beach-Anaheim, CA
- Phoenix, AZ
- Reno, NV
- Riverside, CA
- Sacramento, CA
- San Diego, CA
- Tucson, AZ



For this analysis, a year was considered a bubble year if the following two conditions were met: the derivative of normalized price with respect to time in that year was greater than or equal to 120% of the median derivative of normalized price w.r.t. time in that metropolitan area over the entire 20-year span, and the first and second derivatives of normalized price w.r.t. time were positive. The metropolitan areas denoted by the green icons doubled in price in less than 10 years. These are the same areas that experienced a housing bubble between 2002 and 2006.

The Housing Bubble in Florida

Housing bubbles occurred in the following metropolitan areas in Florida between 2001 and 2006:

- Daytona Beach
- Fort Myers
- Jacksonville
- Melbourne
- Miami-Fort Lauderdale
- North Port-Sarasota-Bradenton
- Orlando
- Port St. Lucie
- Tampa



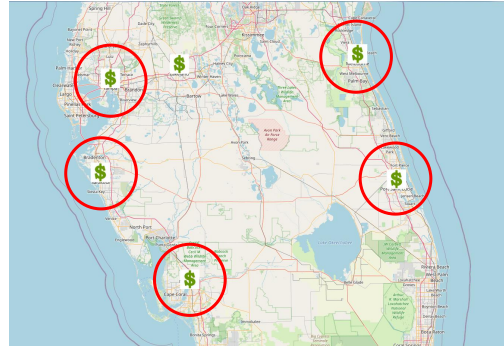
41

A similar pattern was seen in Florida between 2001 and 2006. The conditions for a bubble year were met in most large metropolitan areas in Florida in the first half of the 2000s. The bubble usually lasted from 3 to 5 years. Pensacola was an outlier. Housing prices there doubled more slowly than they did in peninsular Florida.

The Housing Bubble in Florida

A housing bubble occurred in these metropolitan areas from 2001 to 2005:

- Fort Myers
- Melbourne
- North Port-Sarasota-Bradenton
- Port St. Lucie
- Tampa



42

Of particular interest were the metropolitan areas in central Florida. -> This region experienced a sustained housing bubble from 2001 to 2005. In Fort Myers -> -> Melbourne -> -> North Port-Sarasota-Bradenton -> -> Port St. Lucie -> -> and Tampa -> -> Lakeland also experienced a housing bubble, but it did not last from 2001 to 2005.

Alternative Methods for Time Series Analysis

- Support Vector Machine
- Neural Network

43

A time series dataset may be transformed into a supervised learning problem and fed as input to a support vector machine or a neural network to perform regression on it. This has the added advantage of being able to predict future values of the time series in a process called “time series forecasting.”

Summary

Advantages

- simple to implement
- easy to explain
- can be performed in resource-constrained environments
- works well for identifying trends in data and inflection points
- provides new insights into time series data

Disadvantages

- limited predictive power

44

By comparing the derivatives of time series data, researchers can ascertain whether a particular feature (e.g. the price of an asset) is increasing or decreasing in value and the rate at which this change is occurring. But the most valuable insights come as data is aggregated over many experimental units and over a long period of time. Then a general trend in the data points usually emerges.

Further Research

- Performing a regional analysis of the real estate dataset
- Correlation analysis of real estate, DJIA, and inflation datasets

References

- [1] NOAA Photo Library. <https://www.flickr.com/photos/noaaphotolib/9669271621/>
- [2] NOAA Geostationary Operational Environmental Satellites (GOES) 16, 17 & 18 was accessed on November 30, 2022 from <https://registry.opendata.aws/noaa-goes>.
- [3] Jet Propulsion Laboratory, California Institute of Technology. <https://www.jpl.nasa.gov/edu/teach/activity/graphing-global-temperature-trends/>
- [4] U.S. Energy Administration. <https://www.eia.gov/energyexplained/energy-and-the-environment/greenhouse-gases.php>
- [5] Cmpxchg8b. https://commons.wikimedia.org/wiki/File:Lake_Mead_October_2010.jpg
- [6] U.S. Bureau of Reclamation. https://www.usbr.gov/lc/region/g4000/lakemead_line.pdf
- [7] Sanjib Lemar. https://commons.wikimedia.org/wiki/File:Single-family_home.jpg
- [8] Jernej Furman. Flickr.com. <https://www.flickr.com/photos/91261194@N06/49866200962>
- [9] Plumplot.co.uk. <https://www.plumplot.co.uk/Greater-Manchester-violent-crime-statistics.html>

46

The image on the first slide shows “Annual crime rate per 1000 workday people”

Appendix II: Source Code

ca_reader.py

```
1
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import sys
5 import os
6 import numpy as np
7 from dateutil import parser
8
9 date_format = "%m/%d/%Y"
10 COASTAL = False
11
12 def Rank_Cities_in_Descending_Order(t):
13     t.sort(key = lambda x: x[1], reverse=True)
14     return t
15
16 if __name__=="__main__":
17     user_preference = input("Do you want cities in coastal or non-coastal states? ")
18     user_preference = user_preference.upper()
19     if user_preference == "COASTAL":
20         COASTAL = True
21     else:
22         COASTAL = False
23     mode = 1
24     period_length = 1
25
26     #period_name = input_year
27     str_number_of_cities_to_list = "911"
28
29     # process the dataframe
30     original_df = pd.read_csv("Metro_zhvi_uc_sfr_tier_0.33_0.67_sm_sa_month.csv")
31
32     # pre-processing steps
33     # step 1: get the column names
34     complete_rows_df = original_df
35     size_rank_column_df = original_df[['SizeRank']]
36     region_name_column_df = original_df[['RegionName']]
37     size_rank_column_series = size_rank_column_df.squeeze()
38     region_name_column_series = region_name_column_df.squeeze()
39     region_name_column_list = region_name_column_series.tolist()
40     number_of_rows = original_df.shape[0]
41     city_name_size_rank_dict = {}
42     for i in range(0, number_of_rows):
43         city_name_size_rank_dict[region_name_column_series[i]] = size_rank_column_series
44         [i]
45
46     # step 2: drop "RegionID", "SizeRank", "RegionType", and "StateName" columns
47     df = complete_rows_df.drop(columns=['RegionID', 'SizeRank', 'RegionType'])
48     df = df.iloc[1:, :]
```

```

48 if COASTAL == True:
49 df = df[df["RegionName"].str.contains(
"CA|OR|WA|AK|HI|ME|NH|MA|RI|CT|NY|NJ|DE|MD|VA|NC|SC|GA|FL|TX|LA|MS|AL")]
50 else:
51 df = df[df["RegionName"].str.contains(
"ID|MT|ND|MN|WI|MI|VT|WY|SD|IA|PA|NV|UT|CO|NE|IL|IN|OH|KS|MO|KY|WV|AZ|NM|OK|AR|TN
|DC")]
52 print(df.head())
53
54 #if you want cities in a single state, do
55
56 #ca_cities = df[df["RegionName"].str.contains("CA")]

```

djia_derivatives.py

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import sys
5 from datetime import datetime
6 import statistics
7
8 if __name__ == "__main__":
9 djia_df = pd.read_csv("DJIA2000-2022FebCSV.csv", parse_dates=[0]) # DJIA
opening/closing price dataset
10 djia_df['Close*'] = djia_df['Close*'].str.replace(',', '') # remove commas before
conversion to float
11 closing_price_series = djia_df['Close*']
12 closing_price_array = closing_price_series.to_numpy('float64')
13 djia_array = djia_df['Close*'].astype('float64')
14 first_derivative_list = []
15 current_date_and_first_deriv_tuple_list = []
16 second_derivative_list = []
17 current_date_and_second_deriv_tuple_list = []
18 time_intervals = []
19
20 for i in range(len(closing_price_array) - 1):
21 initial_price = closing_price_array[i+1] # because the dataset is in reverse
time order
22 final_price = closing_price_array[i]
23 change_in_price = final_price - initial_price
24 initial_time = datetime.strptime(str(djia_df.iloc[i+1, 0]), "%Y-%m-%d %H:%M:%S")
25 final_time = datetime.strptime(str(djia_df.iloc[i, 0]), "%Y-%m-%d %H:%M:%S")
26 change_in_time = final_time - initial_time
27 dp_dt = change_in_price / change_in_time.days
28 time_intervals.append(initial_time)
29 if dp_dt >= 0:
30 is_positive = True
31 else:
32 is_positive = False
33 first_derivative_list.append(dp_dt)
34 current_date = djia_df.iloc[i, 0]

```

```

35 current_date_and_first_deriv_tuple = (current_date, dp_dt, is_positive)
36 current_date_and_first_deriv_tuple_list.append(
current_date_and_first_deriv_tuple)
37
38 first_derivative_array = np.array(first_derivative_list)
39 for i in range(len(first_derivative_array) - 1):
40 initial_price = first_derivative_array[i]
41 final_price = first_derivative_array[i+1]
42 change_in_price = final_price - initial_price # this is actually the change in
the derivative of price w.r.t time
43 initial_time = datetime.strptime(str(djia_df.iloc[i+2, 0]), "%Y-%m-%d %H:%M:%S")
44 final_time = datetime.strptime(str(djia_df.iloc[i, 0]), "%Y-%m-%d %H:%M:%S")
45 change_in_time = final_time - initial_time
46 dp_dt = change_in_price / change_in_time.days
47 if dp_dt >= 0:
48 is_positive = True
49 else:
50 is_positive = False
51 second_derivative_list.append(dp_dt)
52 current_date = djia_df.iloc[i, 0]
53 current_date_and_second_deriv_tuple = (current_date, dp_dt, is_positive)
54 current_date_and_second_deriv_tuple_list.append(
current_date_and_second_deriv_tuple)
55
56 # lists of first derivatives
57 the_2000_derivatives = []
58 the_2001_derivatives = []
59 the_2002_derivatives = []
60 the_2003_derivatives = []
61 the_2004_derivatives = []

62 the_2005_derivatives = []
63 the_2006_derivatives = []
64 the_2007_derivatives = []
65 the_2008_derivatives = []
66 the_2009_derivatives = []
67 the_2010_derivatives = []
68 the_2011_derivatives = []
69 the_2012_derivatives = []
70 the_2013_derivatives = []
71 the_2014_derivatives = []
72 the_2015_derivatives = []
73 the_2016_derivatives = []
74 the_2017_derivatives = []
75 the_2018_derivatives = []
76 the_2019_derivatives = []
77 the_2020_derivatives = []
78 the_2021_derivatives = []
79 the_2022_derivatives = []
80
81 print("Table of the first derivative of closing price w.r.t. time for the DJIA
dataset.\n")
82 print("-----")

```

```

83 for i in range(0, 30):
84 entry = current_date_and_first_deriv_tuple_list[i]
85 date_obj = entry[0]
86 deriv = entry[1]
87 deriv_2f = "{:.2f}".format(deriv)
88 d = date_obj.date()
89 if d.year == 2000:
90 the_2000_derivatives.append(deriv)
91 if d.year == 2001:
92 the_2001_derivatives.append(deriv)
93 if d.year == 2002:
94 the_2002_derivatives.append(deriv)
95 if d.year == 2003:
96 the_2003_derivatives.append(deriv)
97 if d.year == 2004:
98 the_2004_derivatives.append(deriv)
99 if d.year == 2005:
100 the_2005_derivatives.append(deriv)
101 if d.year == 2006:
102 the_2006_derivatives.append(deriv)
103 if d.year == 2007:
104 the_2007_derivatives.append(deriv)
105 if d.year == 2008:
106 the_2008_derivatives.append(deriv)
107 if d.year == 2009:
108 the_2009_derivatives.append(deriv)
109 if d.year == 2010:
110 the_2010_derivatives.append(deriv)
111 if d.year == 2011:
112 the_2011_derivatives.append(deriv)
113 if d.year == 2012:
114 the_2012_derivatives.append(deriv)
115 if d.year == 2013:
116 the_2013_derivatives.append(deriv)
117 if d.year == 2014:
118 the_2014_derivatives.append(deriv)
119 if d.year == 2015:
120 the_2015_derivatives.append(deriv)
121 if d.year == 2016:
122 the_2016_derivatives.append(deriv)
123 if d.year == 2017:
124 the_2017_derivatives.append(deriv)
125 if d.year == 2018:
126 the_2018_derivatives.append(deriv)
127 if d.year == 2019:

128 the_2019_derivatives.append(deriv)
129 if d.year == 2020:
130 the_2020_derivatives.append(deriv)
131 if d.year == 2021:
132 the_2021_derivatives.append(deriv)
133 if d.year == 2022:
134 the_2022_derivatives.append(deriv)

```

```

135
136 #print('|', date_obj.date(), '|', deriv_2f, '|')
137
138 print("| ... | ... |")
139 print("-----")
140 print(len(current_date_and_first_deriv_tuple_list), 'rows.')
141 print("\n")
142 first_entry = current_date_and_first_deriv_tuple_list[0]
143 part0 = first_entry[0]
144 #print('part0:', part0)
145 first_median = statistics.median(first_derivative_list)
146 first_median_2f = "{:.2f}".format(first_median)
147 print("The median of the first derivative of price w.r.t time is", first_median_2f)
148
149 years_with_derivatives_greater_than_1_20_list = []
150
151 for deriv in the_2000_derivatives:
152     if deriv >= 1.2 * first_median:
153         years_with_derivatives_greater_than_1_20_list.append(2000)
154 for deriv in the_2001_derivatives:
155     if deriv >= 1.2 * first_median:
156         years_with_derivatives_greater_than_1_20_list.append(2001)
157 for deriv in the_2002_derivatives:
158     if deriv >= 1.2 * first_median:
159         years_with_derivatives_greater_than_1_20_list.append(2002)
160 for deriv in the_2003_derivatives:
161     if deriv >= 1.2 * first_median:
162         years_with_derivatives_greater_than_1_20_list.append(2003)
163 for deriv in the_2004_derivatives:
164     if deriv >= 1.2 * first_median:
165         years_with_derivatives_greater_than_1_20_list.append(2004)
166 for deriv in the_2005_derivatives:
167     if deriv >= 1.2 * first_median:
168         years_with_derivatives_greater_than_1_20_list.append(2005)
169 for deriv in the_2006_derivatives:
170     if deriv >= 1.2 * first_median:
171         years_with_derivatives_greater_than_1_20_list.append(2006)
172 for deriv in the_2007_derivatives:
173     if deriv >= 1.2 * first_median:
174         years_with_derivatives_greater_than_1_20_list.append(2007)
175 for deriv in the_2008_derivatives:
176     if deriv >= 1.2 * first_median:
177         years_with_derivatives_greater_than_1_20_list.append(2008)
178 for deriv in the_2009_derivatives:
179     if deriv >= 1.2 * first_median:
180         years_with_derivatives_greater_than_1_20_list.append(2009)
181 for deriv in the_2010_derivatives:
182     if deriv >= 1.2 * first_median:
183         years_with_derivatives_greater_than_1_20_list.append(2010)
184 for deriv in the_2011_derivatives:
185     if deriv >= 1.2 * first_median:
186         years_with_derivatives_greater_than_1_20_list.append(2011)
187 for deriv in the_2012_derivatives:

```

```

188 if deriv >= 1.2 * first_median:
189     years_with_derivatives_greater_than_1_20_list.append(2012)
190 for deriv in the_2013_derivatives:
191     if deriv >= 1.2 * first_median:
192         years_with_derivatives_greater_than_1_20_list.append(2013)
193 for deriv in the_2014_derivatives:
194     if deriv >= 1.2 * first_median:
195         years_with_derivatives_greater_than_1_20_list.append(2014)
196 for deriv in the_2015_derivatives:
197     if deriv >= 1.2 * first_median:
198         years_with_derivatives_greater_than_1_20_list.append(2015)
199 for deriv in the_2016_derivatives:
200     if deriv >= 1.2 * first_median:
201         years_with_derivatives_greater_than_1_20_list.append(2016)
202 for deriv in the_2017_derivatives:
203     if deriv >= 1.2 * first_median:
204         years_with_derivatives_greater_than_1_20_list.append(2017)
205 for deriv in the_2018_derivatives:
206     if deriv >= 1.2 * first_median:
207         years_with_derivatives_greater_than_1_20_list.append(2018)
208 for deriv in the_2019_derivatives:
209     if deriv >= 1.2 * first_median:
210         years_with_derivatives_greater_than_1_20_list.append(2019)
211 for deriv in the_2020_derivatives:
212     if deriv >= 1.2 * first_median:
213         years_with_derivatives_greater_than_1_20_list.append(2020)
214 for deriv in the_2021_derivatives:
215     if deriv >= 1.2 * first_median:
216         years_with_derivatives_greater_than_1_20_list.append(2021)
217 for deriv in the_2022_derivatives:
218     if deriv >= 1.2 * first_median:
219         years_with_derivatives_greater_than_1_20_list.append(2022)
220
221 years_with_derivatives_greater_than_1_20_set = set(
years_with_derivatives_greater_than_1_20_list)
222 print('Years in which the first derivative of closing w.r.t. time was >= 120
percent of the median derivative: ', years_with_derivatives_greater_than_1_20_set)
223
224 print("Table of the second derivative of closing price w.r.t. time for the DJIA
dataset.\n")
225 print("-----")
226 for i in range(0, len(current_date_and_second_deriv_tuple_list)):
227     entry = current_date_and_second_deriv_tuple_list[i]
228     date_obj = entry[0]
229     deriv = entry[1]
230     deriv_2f = "{:.2f}".format(deriv)
231
232     print(" | ... | ... | ")
233     print("-----")
234     print(len(current_date_and_second_deriv_tuple_list), 'rows.')
235     print("\n")
236 second_median = statistics.median(second_derivative_list)

```

```

237 second_median_2f = "{:.2f}".format(second_median)
238 print("The median of the second derivative of price w.r.t time is", second_median_2f)
239
240 x = time_intervals
241 y = first_derivative_array
242 my_xticks = np.linspace(0, 56000, 1)
243 plt.xticks(x, my_xticks)
244 plt.plot(x, y)
245 plt.show()
246

```

highest.py

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import sys
4 import os
5 import numpy as np
6 from dateutil import parser
7
8 if __name__ == "__main__":
9
10     chosen_year = input("Which year? ")
11     int_chosen_year = int(chosen_year)
12
13     mode = 1
14     period_length = 1
15
16     # process the dataframe
17     original_df = pd.read_csv("Metro_zhvi_uc_sfr_tier_0.33_0.67_sm_sa_month.csv")
18
19     # pre-processing steps
20     # step 1: get the column names
21     complete_rows_df = original_df
22     size_rank_column_df = original_df[['SizeRank']]
23     region_name_column_df = original_df[['RegionName']]
24     size_rank_column_series = size_rank_column_df.squeeze()
25     region_name_column_series = region_name_column_df.squeeze()
26     region_name_column_list = region_name_column_series.tolist()
27     number_of_rows = original_df.shape[0]
28     city_name_size_rank_dict = {}
29     for i in range(0, number_of_rows):
30         city_name_size_rank_dict[region_name_column_series[i]] = size_rank_column_series
31         [i]
32
33     # step 2: drop "RegionID", "SizeRank", "RegionType", and "StateName" columns
34     df = complete_rows_df.drop(columns=['RegionID', 'SizeRank', 'RegionType',
35                                         'StateName'])
36     df = df.iloc[1:, :] # remove the first row because it doesn't have a state
37     #if COASTAL == True:
38     # df =
39     df[df["RegionName"].str.contains("CA|OR|WA|AK|HI|ME|NH|MA|RI|CT|NY|NJ|DE|MD|VA|NC|SC|

```

```

GA|FL|TX|LA|MS|AL"))
37 #else:
38 # df =
df[df["RegionName"].str.contains("ID|MT|ND|MN|WI|MI|VT|WY|SD|IA|PA|NV|UT|CO|NE|IL|IN|
OH|KS|MO|KY|WV|AZ|NM|OK|AR|TN|DC"))
39
40 str_number_of_cities_to_list = df.shape[0]
41
42 # create a numeric dataframe consisting of only floats
43 float_df = df.drop(columns=['RegionName'])
44
45 periods = []
46 period_dict = {}
47
48 # partition the floating point dataframe into 25 1-year periods
49 for i in range(0,180,12):
50 period = float_df.iloc[:, i:i+12]
51 periods.append(period)
52 period_dict[0] = '1996'
53 period_dict[1] = '1997'
54 period_dict[2] = '1998'
55 period_dict[3] = '1999'
56 period_dict[4] = '2000'
57 period_dict[5] = '2001'
58 period_dict[6] = '2002'
59 period_dict[7] = '2003'
60 period_dict[8] = '2004'
61 period_dict[9] = '2005'

62 period_dict[10] = '2006'
63 period_dict[11] = '2007'
64 period_dict[12] = '2008'
65 period_dict[13] = '2009'
66 period_dict[14] = '2010'
67 period_dict[15] = '2011'
68 period_dict[16] = '2012'
69 period_dict[17] = '2013'
70 period_dict[18] = '2014'
71 period_dict[19] = '2015'
72 period_dict[20] = '2016'
73 period_dict[21] = '2017'
74 period_dict[22] = '2018'
75 period_dict[23] = '2019'
76 period_dict[24] = '2020'
77 print('periods', periods[0])
78 # the outer list is the states
79 # the inner list is the cities in the state
80 # Washington, DC will be treated as a state
81 list_of_state_high_tuples = [[] for i in range(51)]
82 list_of_state_low_tuples = [[] for i in range(51)]
83
84 # get the city names and put them in a list
85 city_names_list = df['RegionName'].values.tolist()

```

```

86 #big_city_list = city_names_list[1:40]
87 #all_period_list = []
88 for each_city in city_names_list:
89     abs_price_inc_list = []
90     highest_prices = []
91     lowest_prices = []
92     for i in range(0, len(period)):
93         #print('periods[j] is a', type(periods[j]))
94         #print('the columns of this period are:', period.columns)
95         first = i
96         last = i+1
97         row = period.iloc[first:last, :]
98         # convert row to Panadas series
99         row_series = row.squeeze(axis=0)
100         rowmax = row_series.max()
101         rowmin = row_series.min()
102         city_names_list[i]
103         name_price_tuple_high = (city_names_list[i], rowmax)
104         name_price_tuple_low = (city_names_list[i], rowmin)
105         highest_prices.append(name_price_tuple_high)
106         lowest_prices.append(name_price_tuple_low)
107         if name_price_tuple_high[0] == each_city:
108             this_city_name = name_price_tuple_high[0]
109             state_abbr = this_city_name[-2:]
110             #print('highest price for that city was', name_price_tuple_high[1],
111                   'in', year)
111             if state_abbr == "AL":
112                 list_of_state_high_tuples[0].append(name_price_tuple_high)
113             if state_abbr == "AK":
114                 list_of_state_high_tuples[1].append(name_price_tuple_high)
115             if state_abbr == "AZ":
116                 list_of_state_high_tuples[2].append(name_price_tuple_high)
117             if state_abbr == "AR":
118                 list_of_state_high_tuples[3].append(name_price_tuple_high)
119             if state_abbr == "CA":
120                 list_of_state_high_tuples[4].append(name_price_tuple_high)
121             if state_abbr == "CO":
122                 list_of_state_high_tuples[5].append(name_price_tuple_high)
123             if state_abbr == "CT":
124                 list_of_state_high_tuples[6].append(name_price_tuple_high)
125             if state_abbr == "DC":
126                 list_of_state_high_tuples[7].append(name_price_tuple_high)

127             if state_abbr == "DE":
128                 list_of_state_high_tuples[8].append(name_price_tuple_high)
129             if state_abbr == "FL":
130                 list_of_state_high_tuples[9].append(name_price_tuple_high)
131             if state_abbr == "GA":
132                 list_of_state_high_tuples[10].append(name_price_tuple_high)
133             if state_abbr == "HI":
134                 list_of_state_high_tuples[11].append(name_price_tuple_high)
135             if state_abbr == "ID":
136                 list_of_state_high_tuples[12].append(name_price_tuple_high)

```

```

137 if state_abbr == "IL":
138 list_of_state_high_tuples[13].append(name_price_tuple_high)
139 if state_abbr == "IN":
140 list_of_state_high_tuples[14].append(name_price_tuple_high)
141 if state_abbr == "IA":
142 list_of_state_high_tuples[15].append(name_price_tuple_high)
143 if state_abbr == "KS":
144 list_of_state_high_tuples[16].append(name_price_tuple_high)
145 if state_abbr == "KY":
146 list_of_state_high_tuples[17].append(name_price_tuple_high)
147 if state_abbr == "LA":
148 list_of_state_high_tuples[18].append(name_price_tuple_high)
149 if state_abbr == "ME":
150 list_of_state_high_tuples[19].append(name_price_tuple_high)
151 if state_abbr == "MD":
152 list_of_state_high_tuples[20].append(name_price_tuple_high)
153 if state_abbr == "MA":
154 list_of_state_high_tuples[21].append(name_price_tuple_high)
155 if state_abbr == "MI":
156 list_of_state_high_tuples[22].append(name_price_tuple_high)
157 if state_abbr == "MN":
158 list_of_state_high_tuples[23].append(name_price_tuple_high)
159 if state_abbr == "MS":
160 list_of_state_high_tuples[24].append(name_price_tuple_high)
161 if state_abbr == "MO":
162 list_of_state_high_tuples[25].append(name_price_tuple_high)
163 if state_abbr == "MT":
164 list_of_state_high_tuples[26].append(name_price_tuple_high)
165 if state_abbr == "NE":
166 list_of_state_high_tuples[27].append(name_price_tuple_high)
167 if state_abbr == "NV":
168 list_of_state_high_tuples[28].append(name_price_tuple_high)
169 if state_abbr == "NH":
170 list_of_state_high_tuples[29].append(name_price_tuple_high)
171 if state_abbr == "NJ":
172 list_of_state_high_tuples[30].append(name_price_tuple_high)
173 if state_abbr == "NM":
174 list_of_state_high_tuples[31].append(name_price_tuple_high)
175 if state_abbr == "NY":
176 list_of_state_high_tuples[32].append(name_price_tuple_high)
177 if state_abbr == "NC":
178 list_of_state_high_tuples[33].append(name_price_tuple_high)
179 if state_abbr == "ND":
180 list_of_state_high_tuples[34].append(name_price_tuple_high)
181 if state_abbr == "OH":
182 list_of_state_high_tuples[35].append(name_price_tuple_high)
183 if state_abbr == "OK":
184 list_of_state_high_tuples[36].append(name_price_tuple_high)
185 if state_abbr == "OR":
186 list_of_state_high_tuples[37].append(name_price_tuple_high)
187 if state_abbr == "PA":
188 list_of_state_high_tuples[38].append(name_price_tuple_high)
189 if state_abbr == "RI":

```

```

190 list_of_state_high_tuples[39].append(name_price_tuple_high)
191 if state_abbr == "SC":
192 list_of_state_high_tuples[40].append(name_price_tuple_high)
193 if state_abbr == "SD":

194 list_of_state_high_tuples[41].append(name_price_tuple_high)
195 if state_abbr == "TN":
196 list_of_state_high_tuples[42].append(name_price_tuple_high)
197 if state_abbr == "TX":
198 list_of_state_high_tuples[43].append(name_price_tuple_high)
199 if state_abbr == "UT":
200 list_of_state_high_tuples[44].append(name_price_tuple_high)
201 if state_abbr == "VT":
202 list_of_state_high_tuples[45].append(name_price_tuple_high)
203 if state_abbr == "VA":
204 list_of_state_high_tuples[46].append(name_price_tuple_high)
205 if state_abbr == "WA":
206 list_of_state_high_tuples[47].append(name_price_tuple_high)
207 if state_abbr == "WV":
208 list_of_state_high_tuples[48].append(name_price_tuple_high)
209 if state_abbr == "WI":
210 list_of_state_high_tuples[49].append(name_price_tuple_high)
211 if state_abbr == "WY":
212 list_of_state_high_tuples[50].append(name_price_tuple_high)
213
214 # AL is index 0 in list_of_state_high_tuples
215 # WY is index 50 in list_of_state_high_tuples
216 if name_price_tuple_low[0] == each_city:
217 this_city_name = name_price_tuple_low[0]
218 state_abbr = this_city_name[-2:]
219 if state_abbr == "AL":
220 list_of_state_low_tuples[0].append(name_price_tuple_low)
221 if state_abbr == "AK":
222 list_of_state_low_tuples[1].append(name_price_tuple_low)
223 if state_abbr == "AZ":
224 list_of_state_low_tuples[2].append(name_price_tuple_low)
225 if state_abbr == "AR":
226 list_of_state_low_tuples[3].append(name_price_tuple_low)
227 if state_abbr == "CA":
228 list_of_state_low_tuples[4].append(name_price_tuple_low)
229 if state_abbr == "CO":
230 list_of_state_low_tuples[5].append(name_price_tuple_low)
231 if state_abbr == "CT":
232 list_of_state_low_tuples[6].append(name_price_tuple_low)
233 if state_abbr == "DC":
234 list_of_state_low_tuples[7].append(name_price_tuple_low)
235 if state_abbr == "DE":
236 list_of_state_low_tuples[8].append(name_price_tuple_low)
237 if state_abbr == "FL":
238 list_of_state_low_tuples[9].append(name_price_tuple_low)
239 if state_abbr == "GA":
240 list_of_state_low_tuples[10].append(name_price_tuple_low)
241 if state_abbr == "HI":

```

```

242 list_of_state_low_tuples[11].append(name_price_tuple_low)
243 if state_abbr == "ID":
244 list_of_state_low_tuples[12].append(name_price_tuple_low)
245 if state_abbr == "IL":
246 list_of_state_low_tuples[13].append(name_price_tuple_low)
247 if state_abbr == "IN":
248 list_of_state_low_tuples[14].append(name_price_tuple_low)
249 if state_abbr == "IA":
250 list_of_state_low_tuples[15].append(name_price_tuple_low)
251 if state_abbr == "KS":
252 list_of_state_low_tuples[16].append(name_price_tuple_low)
253 if state_abbr == "KY":
254 list_of_state_low_tuples[17].append(name_price_tuple_low)
255 if state_abbr == "LA":
256 list_of_state_low_tuples[18].append(name_price_tuple_low)
257 if state_abbr == "ME":
258 list_of_state_low_tuples[19].append(name_price_tuple_low)
259 if state_abbr == "MD":

260 list_of_state_low_tuples[20].append(name_price_tuple_low)
261 if state_abbr == "MA":
262 list_of_state_low_tuples[21].append(name_price_tuple_low)
263 if state_abbr == "MI":
264 list_of_state_low_tuples[22].append(name_price_tuple_low)
265 if state_abbr == "MN":
266 list_of_state_low_tuples[23].append(name_price_tuple_low)
267 if state_abbr == "MS":
268 list_of_state_low_tuples[24].append(name_price_tuple_low)
269 if state_abbr == "MO":
270 list_of_state_low_tuples[25].append(name_price_tuple_low)
271 if state_abbr == "MT":
272 list_of_state_low_tuples[26].append(name_price_tuple_low)
273 if state_abbr == "NE":
274 list_of_state_low_tuples[27].append(name_price_tuple_low)
275 if state_abbr == "NV":
276 list_of_state_low_tuples[28].append(name_price_tuple_low)
277 if state_abbr == "NH":
278 list_of_state_low_tuples[29].append(name_price_tuple_low)
279 if state_abbr == "NJ":
280 list_of_state_low_tuples[30].append(name_price_tuple_low)
281 if state_abbr == "NM":
282 list_of_state_low_tuples[31].append(name_price_tuple_low)
283 if state_abbr == "NY":
284 list_of_state_low_tuples[32].append(name_price_tuple_low)
285 if state_abbr == "NC":
286 list_of_state_low_tuples[33].append(name_price_tuple_low)
287 if state_abbr == "ND":
288 list_of_state_low_tuples[34].append(name_price_tuple_low)
289 if state_abbr == "OH":
290 list_of_state_low_tuples[35].append(name_price_tuple_low)
291 if state_abbr == "OK":
292 list_of_state_low_tuples[36].append(name_price_tuple_low)
293 if state_abbr == "OR":

```

```

294 list_of_state_low_tuples[37].append(name_price_tuple_low)
295 if state_abbr == "PA":
296 list_of_state_low_tuples[38].append(name_price_tuple_low)
297 if state_abbr == "RI":
298 list_of_state_low_tuples[39].append(name_price_tuple_low)
299 if state_abbr == "SC":
300 list_of_state_low_tuples[40].append(name_price_tuple_low)
301 if state_abbr == "SD":
302 list_of_state_low_tuples[41].append(name_price_tuple_low)
303 if state_abbr == "TN":
304 list_of_state_low_tuples[42].append(name_price_tuple_low)
305 if state_abbr == "TX":
306 list_of_state_low_tuples[43].append(name_price_tuple_low)
307 if state_abbr == "UT":
308 list_of_state_low_tuples[44].append(name_price_tuple_low)
309 if state_abbr == "VT":
310 list_of_state_low_tuples[45].append(name_price_tuple_low)
311 if state_abbr == "VA":
312 list_of_state_low_tuples[46].append(name_price_tuple_low)
313 if state_abbr == "WA":
314 list_of_state_low_tuples[47].append(name_price_tuple_low)
315 if state_abbr == "WV":
316 list_of_state_low_tuples[48].append(name_price_tuple_low)
317 if state_abbr == "WI":
318 list_of_state_low_tuples[49].append(name_price_tuple_low)
319 if state_abbr == "WY":
320 list_of_state_low_tuples[50].append(name_price_tuple_low)
321
322 # Iterate through each list to find the highest and lowest price in each state
323 # in each 1-year period
324 # repeat this for loop 50x, once for each state
325 for i in range(0, len(list_of_state_high_tuples[0])):
326     # for each state list, find the highest-priced city
327     for item in list_of_state_high_tuples[0]:
328         current_highest_city_for_this_state = ""
329         current_high_for_this_state = 0
330         if item[1] >= current_high_for_this_state:
331             current_high_for_this_state = item[1]
332             current_highest_city_for_this_state = item[0]
333         # TODO: Add a dict that maps state abbreviations to state names
334         if ((current_highest_city_for_this_state != "") and (
335             current_high_for_this_state != 0)):
336             print("The city with the highest price in",
337                 current_highest_city_for_this_state[-2:], "is",
338                 current_highest_city_for_this_state, ".", "It had a high price of",
339                 current_high_for_this_state, "in", chosen_year)
340
341     for i in range(0, len(list_of_state_low_tuples)):
342         # for each state list, find the lowest-priced city
343         for item in list_of_state_low_tuples[i]:
344             current_lowest_city_for_this_state = ""
345             current_low_for_this_state = 0

```

```

341 if item[1] <= current_low_for_this_state:
342     current_low_for_this_state = item[1]
343     current_lowest_city_for_this_state = item[0]
344 if ((current_lowest_city_for_this_state != "") and (
current_low_for_this_state != 0)):
345     print("The city with the lowest price in",
current_lowest_city_for_this_state[-2:], "is",
current_lowest_city_for_this_state, ".", "It had a low price of",
current_low_for_this_state, "in", chosen_year)
346
347
348

```

inflation_derivatives.py

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import sys
5 from datetime import datetime
6
7 if __name__ == "__main__":
8     acceptable_inputs = ['JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG', 'SEP',
'OCT', 'NOV', 'DEC', 'Monthly_Average']
9     month_dict = {'JAN': 'January', 'FEB': 'February', 'MAR': 'March', 'APR': 'April',
'MAY': 'May', 'JUN': 'June', 'JUL': 'July', 'AUG': 'August', 'SEP': 'September',
'OCT': 'October', 'NOV': 'November', 'DEC': 'December', 'Monthly_Average':
'Monthly_Average'}
10    month = sys.argv[1]
11    if month != "Monthly_Average": # input validation
12        month = month.upper()
13    if month not in acceptable_inputs:
14        print("Invalid input! Exiting...")
15        sys.exit()
16    inflation_df = pd.read_csv("Inflation_1996-2021.csv", parse_dates=[0]) # inflation
dataset
17    month_series = inflation_df[month]
18    month_array = month_series.to_numpy('float64')
19    inflation_array = inflation_df[month].astype('float64')
20    first_derivative_list = []
21    current_year_and_first_deriv_tuple_list = []
22    second_derivative_list = []
23    current_year_and_second_deriv_tuple_list = []
24
25    for i in range(len(month_array) - 1):
26        intial_cpi = month_array[i]
27        final_cpi = month_array[i+1]
28        change_in_cpi = final_cpi - intial_cpi
29        initial_time = inflation_df.iloc[i, 0]
30        final_time = inflation_df.iloc[i+1, 0]
31        change_in_time = 1 # one year
32        dp_dt = change_in_cpi / change_in_time

```

```

33 first_derivative_list.append(dp_dt)
34 current_year = initial_time
35 current_year_and_first_deriv_tuple = (current_year, dp_dt)
36 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
37
38 first_derivative_array = np.array(first_derivative_list)
39 for i in range(len(first_derivative_array) - 1):
40 initial_cpi = first_derivative_array[i]
41 final_cpi = first_derivative_array[i+1]
42 change_in_cpi = final_cpi - initial_cpi # this is actually the change in the
derivative of price w.r.t time
43 initial_time = inflation_df.iloc[i, 0]
44 final_time = inflation_df.iloc[i+1, 0]
45 change_in_time = 1 # one year
46 dp_dt = change_in_cpi / change_in_time
47 second_derivative_list.append(dp_dt)
48 current_year = initial_time
49 current_year_and_second_deriv_tuple = (current_year, dp_dt)
50 current_year_and_second_deriv_tuple_list.append(
current_year_and_second_deriv_tuple)
51
52 if month == 'Monthly_Average':
53 print("Table of the first derivative of CPI w.r.t. time for the inflation
dataset for the monthly average.\n")
54 else:
55 print("Table of the first derivative of CPI w.r.t. time for the inflation
dataset for the month of ", month_dict[month], '\n', sep='')
56 print("-----")
57 for i in range(0, len(current_year_and_first_deriv_tuple_list)):

58 entry = current_year_and_first_deriv_tuple_list[i]
59 date_obj = entry[0]
60 deriv = entry[1]
61 deriv_2f = "{:.2f}".format(deriv)
62 print('|', date_obj.date(), '|', deriv_2f, '|')
63 #print("| ... | ... |")
64 print("-----")
65 print(len(current_year_and_first_deriv_tuple_list), 'rows.')
66 print("\n")
67
68 plt.rcParams["figure.figsize"] = [7.00, 3.50]
69 plt.rcParams["figure.autolayout"] = True
70 #x = np.arange(2000, 2022, 1, int)
71 #x = [2000, 2001, 2002, 2003, 2004, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011,
2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022]
72 #y = first_derivative_list[0:-2]
73 #default_x_ticks = range(len(x))
74 #plt.plot(default_x_ticks, y)
75 #plt.xticks(default_x_ticks, x)
76 #plt.show()
77
78 if month == 'Monthly_Average':

```

```

79 print("Table of the first derivative of CPI w.r.t. time for the inflation
dataset for the monthly average.\n")
80 else:
81 print("Table of the first derivative of CPI w.r.t. time for the inflation
dataset for the month of ", month_dict[month], '.\n', sep='')
82 print("-----")
83 for i in range(0, len(current_year_and_second_deriv_tuple_list)):
84 entry = current_year_and_second_deriv_tuple_list[i]
85 date_obj = entry[0]
86 deriv = entry[1]
87 deriv_2f = "{:.2f}".format(deriv)
88 print('|', date_obj.date(), '|', deriv_2f, '|')
89
90 #print("| ... | ... |")
91 print("-----")
92 print(len(current_year_and_second_deriv_tuple_list), 'rows.')
93 print("\n")
94
95 plt.rcParams["figure.figsize"] = [7.00, 3.50]
96 plt.rcParams["figure.autolayout"] = True
97 x = np.arange(2000, 2022, 1, int)
98 x = [2000, 2001, 2002, 2003, 2004, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011,
2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]
99 y = second_derivative_list[0:-3]
100 default_x_ticks = range(len(x))
101 plt.plot(default_x_ticks, y)
102 plt.xticks(default_x_ticks, x)
103 plt.show()
104

```

ml1.py

```

1 import numpy as np
2 import pandas as pd
3 from sklearn.linear_model import LinearRegression
4 import datetime
5 import matplotlib.pyplot as plt
6 import sys
7
8 if __name__ == "__main__":
9
10 # read the datasets into Pandas dataframes
11 djia_df = pd.read_csv("DJIA2000-2022FebCSV.csv", parse_dates=[0]) # DJIA
opening/closing price dataset
12 cpi_df = pd.read_csv("Inflation_1996-2021.csv") # CPI inflation dataset
13 re_df = pd.read_csv("Metro_zhvi_uc_sfr_tier_0.33_0.67_sm_sa_month.csv") # Zillow
real estate dataset
14
15 # extract the numeric features from each dataframe
16 numeric_feature_names_of_the_djia_df = ['Open', 'High', 'Low', 'Close*', 'Adj
Close**', 'Volume']
17

```

```

18 numeric_feature_names_of_the_inflation_df = ['Year', 'JAN', 'FEB', 'MAR', 'APR',
'MAY', 'JUN', 'JUL', 'AUG', 'SEP', 'OCT', 'NOV', 'DEC', 'Monthly_Average']
19 numeric_feature_names_of_the_real_estate_df = ['1/31/1996', '2/29/1996', '3/31/1996',
, '4/30/1996', '5/31/1996', '6/30/1996', '7/31/1996', '8/31/1996', '9/30/1996',
'10/31/1996', '11/30/1996', '12/31/1996', '1/31/1997', '2/28/1997', '3/31/1997',
'4/30/1997', '5/31/1997', '6/30/1997', '7/31/1997', '8/31/1997', '9/30/1997',
'10/31/1997', '11/30/1997', '12/31/1997', '1/31/1998', '2/28/1998', '3/31/1998',
'4/30/1998', '5/31/1998', '6/30/1998', '7/31/1998', '8/31/1998', '9/30/1998',
'10/31/1998', '11/30/1998', '12/31/1998', '1/31/1999', '2/28/1999', '3/31/1999',
'4/30/1999', '5/31/1999', '6/30/1999', '7/31/1999', '8/31/1999', '9/30/1999',
'10/31/1999', '11/30/1999', '12/31/1999', '1/31/2000', '2/29/2000', '3/31/2000',
'4/30/2000', '5/31/2000', '6/30/2000', '7/31/2000', '8/31/2000', '9/30/2000',
'10/31/2000', '11/30/2000', '12/31/2000', '1/31/2001', '2/28/2001', '3/31/2001',
'4/30/2001', '5/31/2001', '6/30/2001', '7/31/2001', '8/31/2001', '9/30/2001',
'10/31/2001', '11/30/2001', '12/31/2001', '1/31/2002', '2/28/2002', '3/31/2002',
'4/30/2002', '5/31/2002', '6/30/2002', '7/31/2002', '8/31/2002', '9/30/2002',
'10/31/2002', '11/30/2002', '12/31/2002', '1/31/2003', '2/28/2003', '3/31/2003',
'4/30/2003', '5/31/2003', '6/30/2003', '7/31/2003', '8/31/2003', '9/30/2003',
'10/31/2003', '11/30/2003', '12/31/2003', '1/31/2004', '2/29/2004', '3/31/2004',
'4/30/2004', '5/31/2004', '6/30/2004', '7/31/2004', '8/31/2004', '9/30/2004',
'10/31/2004', '11/30/2004', '12/31/2004', '1/31/2005', '2/28/2005', '3/31/2005',
'4/30/2005', '5/31/2005', '6/30/2005', '7/31/2005', '8/31/2005', '9/30/2005',
'10/31/2005', '11/30/2005', '12/31/2005', '1/31/2006', '2/28/2006', '3/31/2006',
'4/30/2006', '5/31/2006', '6/30/2006', '7/31/2006', '8/31/2006', '9/30/2006',
'10/31/2006', '11/30/2006', '12/31/2006', '1/31/2007', '2/28/2007', '3/31/2007',
'4/30/2007', '5/31/2007', '6/30/2007', '7/31/2007', '8/31/2007', '9/30/2007',
'10/31/2007', '11/30/2007', '12/31/2007', '1/31/2008', '2/29/2008', '3/31/2008',
'4/30/2008', '5/31/2008', '6/30/2008', '7/31/2008', '8/31/2008', '9/30/2008',
'10/31/2008', '11/30/2008', '12/31/2008', '1/31/2009', '2/28/2009', '3/31/2009',
'4/30/2009', '5/31/2009', '6/30/2009', '7/31/2009', '8/31/2009', '9/30/2009',
'10/31/2009', '11/30/2009', '12/31/2009', '1/31/2010', '2/28/2010', '3/31/2010',
'4/30/2010', '5/31/2010', '6/30/2010', '7/31/2010', '8/31/2010', '9/30/2010',
'10/31/2010', '11/30/2010', '12/31/2010', '1/31/2011', '2/28/2011', '3/31/2011',
'4/30/2011', '5/31/2011', '6/30/2011', '7/31/2011', '8/31/2011', '9/30/2011',
'10/31/2011', '11/30/2011', '12/31/2011', '1/31/2012', '2/29/2012', '3/31/2012',
'4/30/2012', '5/31/2012', '6/30/2012', '7/31/2012', '8/31/2012', '9/30/2012',
'10/31/2012', '11/30/2012', '12/31/2012', '1/31/2013', '2/28/2013', '3/31/2013',
'4/30/2013', '5/31/2013', '6/30/2013', '7/31/2013', '8/31/2013', '9/30/2013',
'10/31/2013', '11/30/2013', '12/31/2013', '1/31/2014', '2/28/2014', '3/31/2014',
'4/30/2014', '5/31/2014', '6/30/2014', '7/31/2014', '8/31/2014', '9/30/2014',
'10/31/2014', '11/30/2014', '12/31/2014', '1/31/2015', '2/28/2015', '3/31/2015',
'4/30/2015', '5/31/2015', '6/30/2015', '7/31/2015', '8/31/2015', '9/30/2015',
'10/31/2015', '11/30/2015', '12/31/2015', '1/31/2016', '2/29/2016', '3/31/2016',
'4/30/2016', '5/31/2016', '6/30/2016', '7/31/2016', '8/31/2016', '9/30/2016',
'10/31/2016', '11/30/2016', '12/31/2016', '1/31/2017', '2/28/2017', '3/31/2017',
'4/30/2017', '5/31/2017', '6/30/2017', '7/31/2017', '8/31/2017', '9/30/2017',
'10/31/2017', '11/30/2017', '12/31/2017', '1/31/2018', '2/28/2018', '3/31/2018',

'4/30/2018', '5/31/2018', '6/30/2018', '7/31/2018', '8/31/2018', '9/30/2018',
'10/31/2018', '11/30/2018', '12/31/2018', '1/31/2019', '2/28/2019', '3/31/2019',
'4/30/2019', '5/31/2019', '6/30/2019', '7/31/2019', '8/31/2019', '9/30/2019',
'10/31/2019', '11/30/2019', '12/31/2019', '1/31/2020', '2/29/2020', '3/31/2020',
'4/30/2020', '5/31/2020', '6/30/2020', '7/31/2020', '8/31/2020', '9/30/2020',

```

```

'10/31/2020', '11/30/2020', '12/31/2020', '1/31/2021', '2/28/2021', '3/31/2021',
'4/30/2021', '5/31/2021', '6/30/2021', '7/31/2021', '8/31/2021']
20
21 numeric_features_of_djia_df = djia_df[numeric_feature_names_of_the_djia_df]
22 numeric_features_of_inflation_df = cpi_df[numeric_feature_names_of_the_inflation_df]
23 numeric_features_of_real_estate_df = re_df[
numeric_feature_names_of_the_real_estate_df]
24
25 djia_array = numeric_features_of_djia_df.to_numpy()
26 cpi_array = numeric_features_of_inflation_df.to_numpy()
27 re_array = numeric_features_of_real_estate_df.to_numpy()
28
29 print('djia_array.shape:', djia_array.shape)
30 print('cpi_array.shape:', cpi_array.shape)
31 print('re_array.shape:', re_array.shape)
32
33 # plot each dataset as a timeseries in matplotlib
34 if sys.argv[1] == 're':
35     # plot the real estate dataset
36     list_of_columns = list(re_df.columns)
37     print(list_of_columns)
38
39 if sys.argv[1] == 'djia':
40     # plot the DJIA dataset
41     djia_df["Date"] = djia_df["Date"].astype("datetime64")
42     djia_df = djia_df.set_index("Date")
43     djia_df['Close*'] = djia_df['Close*'].str.replace(',', '')
44     djia_df['Close*'] = djia_df['Close*'].str.replace('.', '')
45     djia_df["Close*"] = djia_df["Close*"].astype("float64")
46     plt.plot(djia_df["Close*"], marker='o')
47     plt.xlabel("Date")
48     plt.ylabel("Closing Price")
49     plt.title("Plot of Daily Closing Price vs. Time for the Dow Jones Industrial
Average from 2000 to 2021")
50     plt.show()
51
52 if sys.argv[1] == 'cpi':
53     # plot a dataset
54     cpi_df["Year"] = cpi_df["Year"].astype("datetime64[ns]")
55     cpi_df = cpi_df.set_index("Year")
56     plt.plot(cpi_df["Monthly_Average"], marker='o')
57     plt.xlabel("Date")
58     plt.ylabel("Annual Inflation Rate")
59     plt.title("Plot of the Consumer Price Index (CPI) vs. Time from 2000 to 2021")
60     plt.show()

```

plot_data.py

```

1 #!/usr/bin/env python3
2
3 import pandas as pd
4 import matplotlib.pyplot as plt

```

```

5 import sys
6 import os
7 import numpy as np
8 from dateutil import parser
9
10 if __name__ == "__main__":
11     filename = sys.argv[1]
12     choice = sys.argv[2]
13     col_name = str(choice)
14     acc_inputs = ['Open', 'High', 'Low', 'Close*', 'Adj Close**', 'Volume', 'Year', 'JAN',
15 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG', 'SEP', 'OCT', 'NOV', 'DEC',
16 'Monthly_Average']
17 if col_name not in acc_inputs:
18     print("Invalid input!!! Exiting...")
19     sys.exit()
20 df = pd.read_csv(str(filename))
21
22 if filename == "Inflation_1999-2021.csv":
23     if str(choice) == 'Year':
24         selected_year = input('Enter a year in range [1996, 2021]: ')
25         int_selected_year = int(selected_year)
26         # 1996 is index 0
27         index = int_selected_year - 1996
28         # print the row for that year
29         subset = df.iloc[index,:]
30     else:
31         selected_month = str(choice)
32         first_col = df['Year']
33         second_col = df[selected_month]
34         subset = pd.concat([first_col, second_col], axis=1)
35     print(subset)
36
37 if filename == "DJIA2000-2022FebCSV.csv":
38     print(df.head())
39     print()
40     print('There are', df.shape[0], 'rows in the dataset.')
41     print('There are', df.shape[1], 'columns in the dataset.')
42     dates = df["Date"]
43     d_list = dates.to_list()
44     ndl = []
45     for d in d_list:
46         numb = parser.parse(d)
47         ndl.append(numb)
48     nda = np.array(ndl)
49
50 closing_prices = df[col_name]
51 cpl = closing_prices.to_list()
52 cpf = []
53 for x in cpl:
54     num = x.replace(',', '')
55     cpf.append(float(num))
56 cpa = np.array(cpf)
57 title_str = 'Plot of Year vs. Closing Price for the DJIA'

```

```

56 plt.title(title_str)
57 plt.plot(nda, cpa)
58 plt.show()
59
60 if filename == "statewide-covid-19-cases-deaths-tests.csv":
61     print(df.head())
62     print()
63     print('There are', df.shape[0], 'rows in the dataset.')
64     print('There are', df.shape[1], 'columns in the dataset.')
65     dates = df["Date"]

66 d_list = dates.to_list()
67 ndl = []
68 for d in d_list:
69     numb = parser.parse(d)
70     ndl.append(numb)
71     nda = np.array(ndl)
72
73 closing_prices = df[col_name]
74 cpl = closing_prices.to_list()
75 cpf = []
76 for x in cpl:
77     num = x.replace(',', '')
78     cpf.append(float(num))
79     cpa = np.array(cpf)
80     title_str = 'Plot of Date vs. '+col_name+' for the DJIA'
81     plt.title(title_str)
82     plt.plot(nda, cpa)
83     plt.show()

```

process_dija.py

```

1 #!/usr/bin/env python3
2
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import sys
6 import os
7 import numpy as np
8 from dateutil import parser
9
10 date_format = "%m/%d/%Y"
11 COASTAL = False # determines whether we include only cities in coastal or non-coastal
states in our analysis
12
13 def Rank_Cities_in_Descending_Order(t):
14     t.sort(key = lambda x: x[1], reverse=True)
15     return t
16
17 if __name__ == "__main__":
18
19     filename = sys.argv[1]

```

```

20 df = pd.read_csv(str(filename))
21 col_name = "Close*"
22 #print(df.head())
23 print()
24 #print('There are', df.shape[0], 'rows in the dataset.')
25 #print('There are', df.shape[1], 'columns in the dataset.')
26 dates = df["Date"]
27 d_list = dates.to_list()
28 ndl = []
29 for d in d_list:
30     numb = parser.parse(d)
31     ndl.append(numb)
32     nda = np.array(ndl)
33
34 # calculate the stock market appreciation rate for selected year
35
36 closing_prices = df[col_name]
37
38 percentage_change_dict = {}
39
40 # 2000
41 last_day_price = df["Close*"][5315 - 2]
42 first_day_price = df["Close*"][5566 - 2]
43 last_day_price_no_comma = last_day_price.replace(',', '')
44 first_day_price_no_comma = first_day_price.replace(',', '')
45 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
46 percentage_change = (diff / float(first_day_price_no_comma)) * 100
47 percentage_change_2f = "{:.2f}".format(percentage_change)
48 percentage_change_dict[2000] = percentage_change_2f
49
50 # 2001
51 last_day_price = df["Close*"][5067 - 2]
52 first_day_price = df["Close*"][5314 - 2]
53 last_day_price_no_comma = last_day_price.replace(',', '')
54 first_day_price_no_comma = first_day_price.replace(',', '')
55 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
56 percentage_change = (diff / float(first_day_price_no_comma)) * 100
57 percentage_change_2f = "{:.2f}".format(percentage_change)
58 percentage_change_dict[2001] = percentage_change_2f
59
60 # 2002
61 last_day_price = df["Close*"][4815 - 2]
62 first_day_price = df["Close*"][5066 - 2]
63 last_day_price_no_comma = last_day_price.replace(',', '')
64 first_day_price_no_comma = first_day_price.replace(',', '')
65 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
66 percentage_change = (diff / float(first_day_price_no_comma)) * 100
67
68 percentage_change_2f = "{:.2f}".format(percentage_change)
69 percentage_change_dict[2002] = percentage_change_2f
70
71 # 2003
72 last_day_price = df["Close*"][4563 - 2]

```

```

72 first_day_price = df["Close*"][4814 - 2]
73 last_day_price_no_comma = last_day_price.replace(',', '')
74 first_day_price_no_comma = first_day_price.replace(',', '')
75 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
76 percentage_change = (diff / float(first_day_price_no_comma)) * 100
77 percentage_change_2f = "{:.2f}".format(percentage_change)
78 percentage_change_dict[2003] = percentage_change_2f
79
80 # 2004
81 last_day_price = df["Close*"][4311 - 2]
82 first_day_price = df["Close*"][4562 - 2]
83 last_day_price_no_comma = last_day_price.replace(',', '')
84 first_day_price_no_comma = first_day_price.replace(',', '')
85 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
86 percentage_change = (diff / float(first_day_price_no_comma)) * 100
87 percentage_change_2f = "{:.2f}".format(percentage_change)
88 percentage_change_dict[2004] = percentage_change_2f
89
90 # 2005
91 last_day_price = df["Close*"][4059 - 2]
92 first_day_price = df["Close*"][4310 - 2]
93 last_day_price_no_comma = last_day_price.replace(',', '')
94 first_day_price_no_comma = first_day_price.replace(',', '')
95 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
96 percentage_change = (diff / float(first_day_price_no_comma)) * 100
97 percentage_change_2f = "{:.2f}".format(percentage_change)
98 percentage_change_dict[2005] = percentage_change_2f
99
100 # 2006
101 last_day_price = df["Close*"][3808 - 2]
102 first_day_price = df["Close*"][4058 - 2]
103 last_day_price_no_comma = last_day_price.replace(',', '')
104 first_day_price_no_comma = first_day_price.replace(',', '')
105 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
106 percentage_change = (diff / float(first_day_price_no_comma)) * 100
107 percentage_change_2f = "{:.2f}".format(percentage_change)
108 percentage_change_dict[2006] = percentage_change_2f
109
110 # 2007
111 last_day_price = df["Close*"][3557 - 2]
112 first_day_price = df["Close*"][3807 - 2]
113 last_day_price_no_comma = last_day_price.replace(',', '')
114 first_day_price_no_comma = first_day_price.replace(',', '')
115 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
116 percentage_change = (diff / float(first_day_price_no_comma)) * 100
117 percentage_change_2f = "{:.2f}".format(percentage_change)
118 percentage_change_dict[2007] = percentage_change_2f
119
120 # 2008
121 last_day_price = df["Close*"][3304 - 2]
122 first_day_price = df["Close*"][3556 - 2]
123 last_day_price_no_comma = last_day_price.replace(',', '')
124 first_day_price_no_comma = first_day_price.replace(',', '')

```

```

125 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
126 percentage_change = (diff / float(first_day_price_no_comma)) * 100
127 percentage_change_2f = "{:.2f}".format(percentage_change)
128 percentage_change_dict[2008] = percentage_change_2f
129
130 # 2009
131 last_day_price = df["Close*"][3052 - 2]
132 first_day_price = df["Close*"][3303 - 2]
133 last_day_price_no_comma = last_day_price.replace(',', '')

134 first_day_price_no_comma = first_day_price.replace(',', '')
135 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
136 percentage_change = (diff / float(first_day_price_no_comma)) * 100
137 percentage_change_2f = "{:.2f}".format(percentage_change)
138 percentage_change_dict[2009] = percentage_change_2f
139
140 # 2010
141 last_day_price = df["Close*"][2800 - 2]
142 first_day_price = df["Close*"][3051 - 2]
143 last_day_price_no_comma = last_day_price.replace(',', '')
144 first_day_price_no_comma = first_day_price.replace(',', '')
145 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
146 percentage_change = (diff / float(first_day_price_no_comma)) * 100
147 percentage_change_2f = "{:.2f}".format(percentage_change)
148 percentage_change_dict[2010] = percentage_change_2f
149
150 # 2011
151 last_day_price = df["Close*"][2548 - 2]
152 first_day_price = df["Close*"][2799 - 2]
153 last_day_price_no_comma = last_day_price.replace(',', '')
154 first_day_price_no_comma = first_day_price.replace(',', '')
155 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
156 percentage_change = (diff / float(first_day_price_no_comma)) * 100
157 percentage_change_2f = "{:.2f}".format(percentage_change)
158 percentage_change_dict[2011] = percentage_change_2f
159
160 # 2012
161 last_day_price = df["Close*"][2298 - 2]
162 first_day_price = df["Close*"][2547 - 2]
163 last_day_price_no_comma = last_day_price.replace(',', '')
164 first_day_price_no_comma = first_day_price.replace(',', '')
165 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
166 percentage_change = (diff / float(first_day_price_no_comma)) * 100
167 percentage_change_2f = "{:.2f}".format(percentage_change)
168 percentage_change_dict[2012] = percentage_change_2f
169
170 # 2013
171 last_day_price = df["Close*"][2046 - 2]
172 first_day_price = df["Close*"][2297 - 2]
173 last_day_price_no_comma = last_day_price.replace(',', '')
174 first_day_price_no_comma = first_day_price.replace(',', '')
175 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
176 percentage_change = (diff / float(first_day_price_no_comma)) * 100

```

```

177 percentage_change_2f = "{:.2f}".format(percentage_change)
178 percentage_change_dict[2013] = percentage_change_2f
179
180 # 2014
181 last_day_price = df["Close*"][1794 - 2]
182 first_day_price = df["Close*"][2045 - 2]
183 last_day_price_no_comma = last_day_price.replace(',', '')
184 first_day_price_no_comma = first_day_price.replace(',', '')
185 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
186 percentage_change = (diff / float(first_day_price_no_comma)) * 100
187 percentage_change_2f = "{:.2f}".format(percentage_change)
188 percentage_change_dict[2014] = percentage_change_2f
189
190 # 2015
191 last_day_price = df["Close*"][1542 - 2]
192 first_day_price = df["Close*"][1793 - 2]
193 last_day_price_no_comma = last_day_price.replace(',', '')
194 first_day_price_no_comma = first_day_price.replace(',', '')
195 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
196 percentage_change = (diff / float(first_day_price_no_comma)) * 100
197 percentage_change_2f = "{:.2f}".format(percentage_change)
198 percentage_change_dict[2015] = percentage_change_2f
199
200 # 2016
201 last_day_price = df["Close*"][1290 - 2]
202 first_day_price = df["Close*"][1541 - 2]
203 last_day_price_no_comma = last_day_price.replace(',', '')
204 first_day_price_no_comma = first_day_price.replace(',', '')
205 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
206 percentage_change = (diff / float(first_day_price_no_comma)) * 100
207 percentage_change_2f = "{:.2f}".format(percentage_change)
208 percentage_change_dict[2016] = percentage_change_2f
209
210 # 2017
211 last_day_price = df["Close*"][1039 - 2]
212 first_day_price = df["Close*"][1289 - 2]
213 last_day_price_no_comma = last_day_price.replace(',', '')
214 first_day_price_no_comma = first_day_price.replace(',', '')
215 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
216 percentage_change = (diff / float(first_day_price_no_comma)) * 100
217 percentage_change_2f = "{:.2f}".format(percentage_change)
218 percentage_change_dict[2017] = percentage_change_2f
219
220 # 2018
221 last_day_price = df["Close*"][788 - 2]
222 first_day_price = df["Close*"][1038 - 2]
223 last_day_price_no_comma = last_day_price.replace(',', '')
224 first_day_price_no_comma = first_day_price.replace(',', '')
225 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
226 percentage_change = (diff / float(first_day_price_no_comma)) * 100
227 percentage_change_2f = "{:.2f}".format(percentage_change)
228 percentage_change_dict[2018] = percentage_change_2f

```

```

229
230 # 2019
231 last_day_price = df["Close*"][536 - 2]
232 first_day_price = df["Close*"][787 - 2]
233 last_day_price_no_comma = last_day_price.replace(',', '')
234 first_day_price_no_comma = first_day_price.replace(',', '')
235 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
236 percentage_change = (diff / float(first_day_price_no_comma)) * 100
237 percentage_change_2f = "{:.2f}".format(percentage_change)
238 percentage_change_dict[2019] = percentage_change_2f
239
240 # 2020
241 last_day_price = df["Close*"][283 - 2]
242 first_day_price = df["Close*"][535 - 2]
243 last_day_price_no_comma = last_day_price.replace(',', '')
244 first_day_price_no_comma = first_day_price.replace(',', '')
245 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
246 percentage_change = (diff / float(first_day_price_no_comma)) * 100
247 percentage_change_2f = "{:.2f}".format(percentage_change)
248 percentage_change_dict[2020] = percentage_change_2f
249
250 # 2021
251 last_day_price = df["Close*"][31 - 2]
252 first_day_price = df["Close*"][282 - 2]
253 last_day_price_no_comma = last_day_price.replace(',', '')
254 first_day_price_no_comma = first_day_price.replace(',', '')
255 diff = float(last_day_price_no_comma) - float(first_day_price_no_comma)
256 percentage_change = (diff / float(first_day_price_no_comma)) * 100
257 percentage_change_2f = "{:.2f}".format(percentage_change)
258 percentage_change_dict[2021] = percentage_change_2f
259
260 input_year = input("Which year do you want to search for? Enter a year in [2000,
2010]: ")
261 if (int(input_year) < 2000) or (int(input_year) > 2010): # check the years 2010-2020
262     print("Year outside acceptable range. Exiting...")
263     sys.exit()
264 coastal_preference = input("Do you want cities in coastal or non-coastal states? ")
265 coastal_preference = coastal_preference.upper()
266 if coastal_preference == "COASTAL":
267     COASTAL = True
268 else:
269     COASTAL = False
270 print("In ", int(input_year), ", the change in the Dow Jones Industrial Average
was: ", percentage_change_dict[int(input_year)], " percent.", sep = "")
271
272 # calculate the real estate appreciation rate for selected year(s)
273
274 # add color to the terminal
275 os.system('color')
276
277 mode = 1
278 period_length = 1

```

```

279
280 period_name = input_year
281
282 # process the dataframe
283 original_df = pd.read_csv("Metro_zhvi_uc_sfr_tier_0.33_0.67_sm_sa_month.csv")
284
285 # pre-processing steps
286 # step 1: get the column names
287 complete_rows_df = original_df
288 size_rank_column_df = original_df[['SizeRank']]
289 region_name_column_df = original_df[['RegionName']]
290 size_rank_column_series = size_rank_column_df.squeeze()
291 region_name_column_series = region_name_column_df.squeeze()
292 region_name_column_list = region_name_column_series.tolist()
293 number_of_rows = original_df.shape[0]
294 city_name_size_rank_dict = {}
295 for i in range(0, number_of_rows):
296 city_name_size_rank_dict[region_name_column_series[i]] = size_rank_column_series
[i]
297
298 # step 2: drop "RegionID", "SizeRank", "RegionType", and "StateName" columns
299 df = complete_rows_df.drop(columns=['RegionID', 'SizeRank', 'RegionType',
'StateName'])
300 df = df.iloc[1:, :] # remove the first row because it doesn't have a state
301 if COASTAL == True:
302 df = df[df["RegionName"].str.contains(
"CA|OR|WA|AK|HI|ME|NH|MA|RI|CT|NY|NJ|DE|MD|VA|NC|SC|GA|FL|TX|LA|MS|AL")]
303 else:
304 df = df[df["RegionName"].str.contains(
"ID|MT|ND|MN|WI|MI|VT|WY|SD|IA|PA|NV|UT|CO|NE|IL|IN|OH|KS|MO|KY|WV|AZ|NM|OK|AR|TN
|DC")]
305
306 str_number_of_cities_to_list = df.shape[0]
307
308 # create a numeric dataframe consisting of only floats
309 float_df = df.drop(columns=['RegionName'])
310
311 periods = []
312 period_dict = {}
313
314 # partition the floating point dataframe into 25 1-year periods
315 for i in range(0,180,12):
316 period = float_df.iloc[:, i:i+12]
317 periods.append(period)
318 period_dict[0] = '1996'
319 period_dict[1] = '1997'
320 period_dict[2] = '1998'
321 period_dict[3] = '1999'
322 period_dict[4] = '2000'
323 period_dict[5] = '2001'
324 period_dict[6] = '2002'
325 period_dict[7] = '2003'
326 period_dict[8] = '2004'

```

```

327 period_dict[9] = '2005'

328 period_dict[10] = '2006'
329 period_dict[11] = '2007'
330 period_dict[12] = '2008'
331 period_dict[13] = '2009'
332 period_dict[14] = '2010'
333 period_dict[15] = '2011'
334 period_dict[16] = '2012'
335 period_dict[17] = '2013'
336 period_dict[18] = '2014'
337 period_dict[19] = '2015'
338 period_dict[20] = '2016'
339 period_dict[21] = '2017'
340 period_dict[22] = '2018'
341 period_dict[23] = '2019'
342 period_dict[24] = '2020'
343
344 # get the city names and put them in a list
345 city_names_list = df['RegionName'].values.tolist()
346 big_city_list = city_names_list[1:40]
347 all_period_list = []
348 for period in periods:
349     # declare a list to store the normalized price increase for each city for that
    period
350 abs_price_inc_list = []
351 for i in range(0, len(period)):
352     first = i
353     last = i+1
354     row = period.iloc[first:last, :]
355     # convert row to Pandas series
356     row_series = row.squeeze(axis=0)
357     rowmax = row_series.max()
358     # normalized row for a city
359     norm_series = row_series.divide(rowmax)
360     limit = len(norm_series) - 1
361     first_element_in_row = norm_series[0]
362     last_element_in_row = norm_series[limit]
363     difference_between_the_elements = last_element_in_row - first_element_in_row
364     city_names_list[i]
365     name_price_tuple = (city_names_list[i], difference_between_the_elements)
366     abs_price_inc_list.append(name_price_tuple)
367     sorted_period_list = Rank_Cities_in_Descending_Order(abs_price_inc_list)
368     all_period_list.append(sorted_period_list)
369     for k in range(0, len(all_period_list) - 1):
370         # get the kth period's sorted list of city name-price increase pairs
371         current_period = all_period_list[k]
372         highest_city_price_inc_pair = current_period[0]
373         highest_city = highest_city_price_inc_pair[0]
374         highest_city_increase = highest_city_price_inc_pair[1]
375
376     fp_number_of_cities_to_list = float(str_number_of_cities_to_list)
377     number_of_cities_to_list = int(np.ceil(fp_number_of_cities_to_list))

```

```

378 period_list = list(period_dict.keys())
379 year_list = list(period_dict.values())
380 idx = year_list.index(period_name)
381 period_number = period_list[idx]
382 city_names_list = df['RegionName'].values.tolist()
383 # input validation
384 if number_of_cities_to_list <= 0:
385     print("You are trying to list too few cities! Exiting...")
386     sys.exit()
387 if number_of_cities_to_list > len(city_names_list):
388     print("You are trying to list more cities than there are in the dataset!
Exiting...")
389     sys.exit()
390 period = periods[period_number]
391 # declare a list to store the normalized price increase for each city for that period
392 abs_price_inc_list = []

393 for i in range(0, len(period)):
394     first = i
395     last = i+1
396     row = period.iloc[first:last, :]
397     # convert row to Pandas series
398     row_series = row.squeeze(axis=0)
399     rowmax = row_series.max()
400     # normalized row for a city
401     norm_series = row_series.divide(rowmax)
402     limit = len(norm_series) - 1
403     first_element_in_row = norm_series[0]
404     last_element_in_row = norm_series[limit]
405     difference_between_the_elements = last_element_in_row - first_element_in_row
406     city_names_list[i]
407     name_price_tuple = (city_names_list[i], difference_between_the_elements)
408     abs_price_inc_list.append(name_price_tuple)
409     sorted_list = Rank_Cities_in_Descending_Order(abs_price_inc_list)
410     avg_inc_in_normalized_price_list = []
411     for j in range(0, number_of_cities_to_list):
412         highest_city_price_inc_pair = sorted_list[j]
413         highest_city = highest_city_price_inc_pair[0]
414         highest_city_increase = highest_city_price_inc_pair[1]
415         avg_inc_in_normalized_price_list.append(highest_city_increase)
416         avg_inc_in_normalized_price_array = np.array(avg_inc_in_normalized_price_list)
417         avg_inc_in_normalized_price = np.nanmean(avg_inc_in_normalized_price_array)
418         avg_inc_in_normalized_price_2f = "{:.2f}".format(avg_inc_in_normalized_price * 100)
419         print("The average increase in the normalized median price of single family homes
was: ", avg_inc_in_normalized_price_2f, " percent.", sep="")
420
421 # calculate the inflation rate for the selected year(s)
422
423 filename2 = sys.argv[2]
424 df2 = pd.read_csv(str(filename2))
425 int_selected_year = int(input_year)
426 selected_month = "Monthly_Average"
427 first_col = df2['Year']

```

```

428 second_col = df2[selected_month]
429 subset = pd.concat([first_col, second_col], axis=1)
430 # 1996 is index 0
431 index = int_selected_year - 1996
432 row = df2[index:index+1]
433 inflation_rate_df = row['Monthly_Average']
434 inflation_rate = inflation_rate_df.to_string(index=False)
435 print("The inflation rate was", inflation_rate, " percent.", sep="")

```

rel.py

```

1 #!/usr/bin/env python
2
3 import numpy as np
4 import pandas as pd
5 import matplotlib.pyplot as plt
6 import sys
7 import os
8 import termcolor as tm
9 import scipy.stats
10 from datetime import datetime
11 from OSMPythonTools.nominatim import Nominatim
12
13 # named constants
14
15 date_format = "%m/%d/%Y"
16
17 # sorting functions
18
19 def Rank_Cities_in_Descending_Order(t):
20     t.sort(key = lambda x: x[1], reverse=True)
21     return t
22
23 def Rank_Cities_in_Ascending_Order(t):
24     t.sort(key = lambda x: x[1], reverse=False)
25     return t
26 #TODO: Add a dict that maps user-friendly string command line arguments to numeric modes
27
28 if __name__ == "__main__":
29     # add color to the terminal
30     os.system('color')
31     # take command line arguments
32     cmd = str(sys.argv[1])
33     mode = int(cmd)
34     period_length = 10 # default value
35     # choose mode and take corresponding user input
36     if (mode < 0) or (mode > 20):
37         print('Invalid command line argument! Exiting...')
38         sys.exit()
39     if mode == 1:
40         period_length_arg = input('Select period length of 1, 2, 5, or 10 years: ')
41         period_length = int(period_length_arg)

```

```

42 # input validation
43 if (period_length < 1) or (period_length > 10):
44     period_length = 10 # default value
45     city1 = input("Enter a city and state: ")
46     elif mode == 2:
47         period_length_arg = input('Select period length of 1, 2, 5, or 10 years: ')
48         period_length = int(period_length_arg)
49     # input validation
50     if (period_length < 1) or (period_length > 10):
51         period_length = 10 # default value
52     if period_length == 1:
53         period_name = input("Enter a 1-year period between 1996 and 2020 (in YYYY
format): ")
54     elif period_length == 2:
55         period_name = input("Enter a 2-year period between 1996 and 2020 (in
YYYY-YYYY format): ")
56     elif period_length == 5:
57         period_name = input("Enter a 5-year period between 1996 and 2020 (in
YYYY-YYYY format): ")
58     elif period_length == 10:
59         period_name = input("Enter a 10-year period between 1996 and 2020 (in
YYYY-YYYY format): ")
60     str_number_of_cities_to_list = input("How many cities do you want to list? ")
61     elif mode == 3:
62         amount = input("Enter an amount to invest (in thousands): ")
63     elif mode == 4:

64     city1 = input("Enter a city and state: ")
65     city2 = input("Enter another city and state: ")
66     elif mode == 5:
67         city1 = input("Enter a city and state: ")
68     elif mode == 6:
69         city1 = input("Enter a city and state: ")
70     elif mode == 7:
71         city1 = input("Enter a city and state: ")
72     elif mode == 8:
73         city1 = input("Enter a city and state: ")
74         str_requested_year = input("Enter a year: ")
75         requested_year = int(str_requested_year)
76     elif mode == 9:
77         city1 = input("Enter a city and state: ")
78         str_requested_years = input("Enter a series of years (separated by spaces): ")
79         requested_years_string_list = list(str_requested_years.split(' '))
80         requested_years = [int(i) for i in requested_years_string_list]
81     elif mode == 10:
82         city1 = input("Enter a city and state: ")
83     elif mode == 11:
84         city1 = input("Enter a city and state: ")
85     elif mode == 13:
86         upper_bound_str = input("Enter the upper bound: ")
87         lower_bound_str = input("Enter the lower bound: ")
88         upper_bound = float(upper_bound_str)
89         lower_bound = float(lower_bound_str)

```

```

90 if lower_bound > upper_bound:
91     print("Invalid input!!!")
92     sys.exit(0)
93 elif mode == 15:
94     city1 = input("Enter a city and state: ")
95
96 # process the dataframe
97 original_df = pd.read_csv("Metro_zhvi_uc_sfr_tier_0.33_0.67_sm_sa_month.csv")
98 senators_df = pd.read_csv("Senators_Political_Affiliations_1996_2021.csv")
99
100 # define the regions of the United States
101 west_coast_states = ["WA", "OR", "CA", "NV", "AZ", "AK", "HI"]
102 rocky_mountain_states = ["MT", "ID", "WY", "UT", "CO"]
103 gulf_coast_states = ["NM", "TX", "LA", "AR", "MS", "AL"]
104 midwest_states = ["ND", "SD", "NE", "KS", "OK", "MN", "IA", "MO", "WI", "IL", "IN",
105 "OH", "MI", "KY", "TN"]
106
107 #index 0 is 1996
108 AL_series = senators_df["AL"]
109 AL_senators = AL_series.tolist()
110
111 AR_series = senators_df["AR"]
112 AR_senators = AR_series.tolist()
113
114 AZ_series = senators_df["AZ"]
115 AZ_senators = AZ_series.tolist()
116
117 CA_series = senators_df["CA"]
118 CA_senators = CA_series.tolist()
119
120 CO_series = senators_df["CO"]
121 CO_senators = CO_series.tolist()
122
123 CT_series = senators_df["CT"]
124 CT_senators = CT_series.tolist()
125
126 FL_series = senators_df["FL"]
127 FL_senators = FL_series.tolist()
128
129
130 GA_series = senators_df["GA"]
131 GA_senators = GA_series.tolist()
132
133 HI_series = senators_df["HI"]
134 HI_senators = HI_series.tolist()
135
136 IA_series = senators_df["IA"]
137 IA_senators = IA_series.tolist()
138
139 ID_series = senators_df["ID"]
140 ID_senators = ID_series.tolist()

```

```

140
141 IL_series = senators_df["IL"]
142 IL_senators = IL_series.tolist()
143
144 IN_series = senators_df["IN"]
145 IN_senators = IN_series.tolist()
146
147 KS_series = senators_df["KS"]
148 KS_senators = KS_series.tolist()
149
150 KY_series = senators_df["KY"]
151 KY_senators = KY_series.tolist()
152
153 LA_series = senators_df["LA"]
154 LA_senators = LA_series.tolist()
155
156 ME_series = senators_df["ME"]
157 ME_senators = ME_series.tolist()
158
159 MD_series = senators_df["MD"]
160 MD_senators = MD_series.tolist()
161
162 MA_series = senators_df["MA"]
163 MA_senators = MA_series.tolist()
164
165 MI_series = senators_df["MI"]
166 MI_senators = MI_series.tolist()
167
168 MN_series = senators_df["MN"]
169 MN_senators = MN_series.tolist()
170
171 MS_series = senators_df["MS"]
172 MS_senators = MS_series.tolist()
173
174 MO_series = senators_df["MO"]
175 MO_senators = MO_series.tolist()
176
177 NE_series = senators_df["NE"]
178 NE_senators = NE_series.tolist()
179
180 NV_series = senators_df["NV"]
181 NV_senators = NV_series.tolist()
182
183 NJ_series = senators_df["NJ"]
184 NJ_senators = NJ_series.tolist()
185
186 NM_series = senators_df["NM"]
187 NM_senators = NM_series.tolist()
188
189 NY_series = senators_df["NY"]
190 NY_senators = NY_series.tolist()
191
192 NC_series = senators_df["NC"]

```

```

193 NC_senators = NC_series.tolist()
194
195 OH_series = senators_df["OH"]
196 OH_senators = OH_series.tolist()
197
198 OK_series = senators_df["OK"]
199 OK_senators = OK_series.tolist()
200
201 OR_series = senators_df["OR"]
202 OR_senators = OR_series.tolist()
203
204 PA_series = senators_df["PA"]
205 PA_senators = PA_series.tolist()
206
207 RI_series = senators_df["RI"]
208 RI_senators = RI_series.tolist()
209
210 SC_series = senators_df["SC"]
211 SC_senators = SC_series.tolist()
212
213 TN_series = senators_df["TN"]
214 TN_senators = TN_series.tolist()
215
216 TX_series = senators_df["TX"]
217 TX_senators = TX_series.tolist()
218
219 UT_series = senators_df["UT"]
220 UT_senators = UT_series.tolist()
221
222 VA_series = senators_df["VA"]
223 VA_senators = VA_series.tolist()
224
225 WA_series = senators_df["WA"]
226 WA_senators = WA_series.tolist()
227
228 WI_series = senators_df["WI"]
229 WI_senators = WI_series.tolist()
230
231 # pre-processing steps
232 # step 1: get the column names
233 complete_rows_df = original_df
234 size_rank_column_df = original_df[['SizeRank']]
235 region_name_column_df = original_df[['RegionName']]
236 size_rank_column_series = size_rank_column_df.squeeze()
237 region_name_column_series = region_name_column_df.squeeze()
238 region_name_column_list = region_name_column_series.tolist()
239 number_of_rows = original_df.shape[0]
240 city_name_size_rank_dict = {}
241 for i in range(0, number_of_rows):
242     city_name_size_rank_dict[region_name_column_series[i]] = size_rank_column_series
    [i]
243

```

```

244 # step 2: drop "RegionID", "SizeRank", "RegionType", and "StateName" columns
245 df = complete_rows_df.drop(columns=['RegionID', 'SizeRank', 'RegionType',
'StateName'])
246
247 # create a numeric dataframe consisting of only floats
248 float_df = df.drop(columns=['RegionName'])
249
250 periods = []
251 period_dict = {}
252
253 if period_length == 1:
254 # partition the floating point dataframe into 25 1-year periods
255 for i in range(0,180,12):
256 period = float_df.iloc[:, i:i+12]
257 periods.append(period)
258 period_dict[0] = '1996'
259 period_dict[1] = '1997'
260 period_dict[2] = '1998'

261 period_dict[3] = '1999'
262 period_dict[4] = '2000'
263 period_dict[5] = '2001'
264 period_dict[6] = '2002'
265 period_dict[7] = '2003'
266 period_dict[8] = '2004'
267 period_dict[9] = '2005'
268 period_dict[10] = '2006'
269 period_dict[11] = '2007'
270 period_dict[12] = '2008'
271 period_dict[13] = '2009'
272 period_dict[14] = '2010'
273 period_dict[15] = '2011'
274 period_dict[16] = '2012'
275 period_dict[17] = '2013'
276 period_dict[18] = '2014'
277 period_dict[19] = '2015'
278 period_dict[20] = '2016'
279 period_dict[21] = '2017'
280 period_dict[22] = '2018'
281 period_dict[23] = '2019'
282 period_dict[24] = '2020'
283
284 if period_length == 2:
285 # partition the floating point dataframe into 25 2-year periods
286 for i in range(0,180,12):
287 period = float_df.iloc[:, i:i+24]
288 periods.append(period)
289 period_dict[0] = '1996-1997'
290 period_dict[1] = '1997-1998'
291 period_dict[2] = '1998-1999'
292 period_dict[3] = '1999-2000'
293 period_dict[4] = '2000-2001'
294 period_dict[5] = '2001-2002'

```

```

295 period_dict[6] = '2002-2003'
296 period_dict[7] = '2003-2004'
297 period_dict[8] = '2004-2005'
298 period_dict[9] = '2005-2006'
299 period_dict[10] = '2006-2007'
300 period_dict[11] = '2007-2008'
301 period_dict[12] = '2008-2009'
302 period_dict[13] = '2009-2010'
303 period_dict[14] = '2010-2011'
304 period_dict[15] = '2011-2012'
305 period_dict[16] = '2012-2013'
306 period_dict[17] = '2013-2014'
307 period_dict[18] = '2014-2015'
308 period_dict[19] = '2015-2016'
309 period_dict[20] = '2016-2017'
310 period_dict[21] = '2017-2018'
311 period_dict[22] = '2018-2019'
312 period_dict[23] = '2019-2020'
313 period_dict[24] = '2020-2021'
314
315 if period_length == 5:
316     # partition the floating point dataframe into 20 5-year periods
317     for i in range(0,252,12):
318         period = float_df.iloc[:, i:i+60]
319         periods.append(period)
320         period_dict[0] = '1996-2001'
321         period_dict[1] = '1997-2002'
322         period_dict[2] = '1998-2003'
323         period_dict[3] = '1999-2004'
324         period_dict[3] = '2000-2005'
325         period_dict[4] = '2001-2006'
326         period_dict[5] = '2002-2007'
327         period_dict[6] = '2003-2008'
328
328 period_dict[7] = '2004-2009'
329 period_dict[8] = '2005-2010'
330 period_dict[9] = '2006-2011'
331 period_dict[10] = '2007-2012'
332 period_dict[11] = '2008-2013'
333 period_dict[12] = '2009-2014'
334 period_dict[13] = '2010-2015'
335 period_dict[14] = '2011-2016'
336 period_dict[15] = '2012-2017'
337 period_dict[16] = '2013-2018'
338 period_dict[17] = '2014-2019'
339 period_dict[18] = '2015-2020'
340 period_dict[19] = '2016-2021'
341
342 if period_length == 10:
343     # partition the floating point dataframe into 15 10-year periods
344     for i in range(0,180,12):
345         period = float_df.iloc[:, i:i+132]
346         periods.append(period)

```

```

347 period_dict[0] = '1996-2006'
348 period_dict[1] = '1997-2007'
349 period_dict[2] = '1998-2008'
350 period_dict[3] = '1999-2009'
351 period_dict[4] = '2000-2010'
352 period_dict[5] = '2001-2011'
353 period_dict[6] = '2002-2012'
354 period_dict[7] = '2003-2013'
355 period_dict[8] = '2004-2014'
356 period_dict[9] = '2005-2015'
357 period_dict[10] = '2006-2016'
358 period_dict[11] = '2007-2017'
359 period_dict[12] = '2008-2018'
360 period_dict[13] = '2009-2019'
361 period_dict[14] = '2010-2020'
362
363 # get the city names and put them in a list
364 city_names_list = df['RegionName'].values.tolist()
365 big_city_list = city_names_list[1:40]
366 all_period_list = []
367 for period in periods:
368     # declare a list to store the normalized price increase for each city for that
    period
369     abs_price_inc_list = []
370     for i in range(0, len(period)):
371         first = i
372         last = i+1
373         row = period.iloc[first:last, :]
374         # convert row to Pandas series
375         row_series = row.squeeze(axis=0)
376         rowmax = row_series.max()
377         # normalized row for a city
378         norm_series = row_series.divide(rowmax)
379         limit = len(norm_series) - 1
380         first_element_in_row = norm_series[0]
381         last_element_in_row = norm_series[limit]
382         difference_between_the_elements = last_element_in_row - first_element_in_row
383         city_names_list[i]
384         name_price_tuple = (city_names_list[i], difference_between_the_elements)
385         abs_price_inc_list.append(name_price_tuple)
386         sorted_period_list = Rank_Cities_in_Descending_Order(abs_price_inc_list)
387         all_period_list.append(sorted_period_list)
388         for k in range(0, len(all_period_list) - 1):
389             # get the kth period's sorted list of city name-price increase pairs
390             current_period = all_period_list[k]
391             highest_city_price_inc_pair = current_period[0]
392             highest_city = highest_city_price_inc_pair[0]
393             highest_city_increase = highest_city_price_inc_pair[1]
394
395 # the modes
396 if mode == 1:
397     current_highest_price = 0

```

```

398 index_of_best_period = 0
399 for i in range(0, len(all_period_list) - 1):
400     current_period = all_period_list[i]
401     for j in range(0, len(current_period)):
402         this_tuple = current_period[j]
403         if this_tuple[0] == city1:
404             price_in_period_i = this_tuple[1]
405             if price_in_period_i > current_highest_price:
406                 current_highest_price = price_in_period_i
407                 index_of_best_period = i
408             current_highest_price_3f = "{:.3f}".format(current_highest_price)
409             print("The best ", period_length, "-year span to invest in ", city1, " was ",
                  period_dict[index_of_best_period], ". It had an increase in normalized price of
                  ", current_highest_price_3f, " during that period.", sep="")
410
411 if mode == 2:
412     fp_number_of_cities_to_list = float(str_number_of_cities_to_list)
413     number_of_cities_to_list = int(np.ceil(fp_number_of_cities_to_list))
414     period_list = list(period_dict.keys())
415     year_list = list(period_dict.values())
416     idx = year_list.index(period_name)
417     period_number = period_list[idx]
418     city_names_list = df['RegionName'].values.tolist()
419     # input validation
420     if number_of_cities_to_list <= 0:
421         print("You are trying to list too few cities! Exiting...")
422         sys.exit()
423     if number_of_cities_to_list > len(city_names_list):
424         print("You are trying to list more cities than there are in the dataset!
                  Exiting...")
425         sys.exit()
426     period = periods[period_number]
427     # declare a list to store the normalized price increase for each city for that
    period
428     abs_price_inc_list = []
429     for i in range(0, len(period)):
430         first = i
431         last = i+1
432         row = period.iloc[first:last, :]
433         # convert row to Pandas series
434         row_series = row.squeeze(axis=0)
435         rowmax = row_series.max()
436         # normalized row for a city
437         norm_series = row_series.divide(rowmax)
438         limit = len(norm_series) - 1
439         first_element_in_row = norm_series[0]
440         last_element_in_row = norm_series[limit]
441         difference_between_the_elements = last_element_in_row - first_element_in_row
442         city_names_list[i]
443         name_price_tuple = (city_names_list[i], difference_between_the_elements)
444         abs_price_inc_list.append(name_price_tuple)
445         sorted_list = Rank_Cities_in_Descending_Order(abs_price_inc_list)
446         print("The", number_of_cities_to_list, "highest performing cities in",

```

```

period_name, 'were:\n')
447 for j in range(0, number_of_cities_to_list):
448     highest_city_price_inc_pair = sorted_list[j]
449     highest_city = highest_city_price_inc_pair[0]
450     highest_city_increase = highest_city_price_inc_pair[1]
451     highest_city_increase_3f = "{:.3f}".format(highest_city_increase)
452     print(j+1, highest_city, 'which had an increase in normalized price of',
highest_city_increase_3f)
453
454 if mode == 3:

455 # filter the rows of the dataframe based on the initial amount entered
456 reduced_df = float_df.loc[float_df['1/31/1996'] <= ( float(amount) * 1000) ]
457 # get the city names and put them in a list
458 city_names_list = df['RegionName'].values.tolist()
459 # declare a list to store the normalized price increase for each city
460 abs_price_inc_list = []
461 for i in range(0, reduced_df.shape[0]):
462     first = i
463     last = i+1
464     row = reduced_df.iloc[first:last, :]
465     # convert row to Pandas series
466     row_series = row.squeeze(axis=0)
467     rowmax = row_series.max()
468     # normalized row for a city
469     norm_series = row_series.divide(rowmax)
470     limit = len(norm_series) - 8
471     first_element_in_row = norm_series[0]
472     last_element_in_row = norm_series[limit]
473     difference_between_the_elements = last_element_in_row - first_element_in_row
474     city_names_list[i]
475     name_price_tuple = (city_names_list[i], difference_between_the_elements)
476     abs_price_inc_list.append(name_price_tuple)
477     sorted_list = Rank_Cities_in_Descending_Order(abs_price_inc_list)
478     highest_city_price_inc_pair = sorted_list[0]
479     highest_city = highest_city_price_inc_pair[0]
480     highest_city_increase = highest_city_price_inc_pair[1]
481     highest_city_increase_3f = "{:.3f}".format(highest_city_increase)
482     print('The best place to invest', amount, 'thousand dollars in 1996-2020 was',
highest_city, 'which had an increase in normalized price of',
highest_city_increase_3f)
483
484 if mode == 4:
485 # get the city names and put them in a list
486 city_names_list = df['RegionName'].values.tolist()
487 # declare a list to store the normalized price increase for each city
488 abs_price_inc_list = []
489 for i in range(0, float_df.shape[0]):
490     first = i
491     last = i+1
492     row = float_df.iloc[first:last, :]
493     # convert row to Panadas series
494     row_series = row.squeeze(axis=0)

```

```

495 rowmax = row_series.max()
496 # normalized row for a city
497 norm_series = row_series.divide(rowmax)
498 limit = len(norm_series) - 8
499 first_element_in_row = norm_series[0]
500 last_element_in_row = norm_series[limit]
501 difference_between_the_elements = last_element_in_row - first_element_in_row
502 name_price_tuple = (city_names_list[i], difference_between_the_elements)
503 abs_price_inc_list.append(name_price_tuple)
504 sorted_list = Rank_Cities_in_Descending_Order(abs_price_inc_list)
505
506 current_highest_price_for_city1 = 0
507 current_highest_price_for_city2 = 0
508 for j in range(0, len(sorted_list)):
509     this_tuple = sorted_list[j]
510     if this_tuple[0] == city1:
511         price_in_period_i = this_tuple[1]
512         if price_in_period_i > current_highest_price_for_city1:
513             current_highest_price_for_city1 = price_in_period_i
514         if this_tuple[0] == city2:
515             price_in_period_j = this_tuple[1]
516             if price_in_period_j > current_highest_price_for_city2:
517                 current_highest_price_for_city2 = price_in_period_j
518             current_highest_price_for_city1_3f = "{:.3f}".format(
current_highest_price_for_city1)
519             current_highest_price_for_city2_3f = "{:.3f}".format(
current_highest_price_for_city2)
520             if current_highest_price_for_city1 > current_highest_price_for_city2:
521                 print(city1, 'was a better place to invest than', city2, 'in 1996-2020.')
522                 print(city1, 'had an increase in normalized price of',
current_highest_price_for_city1_3f)
523                 print(city2, 'had an increase in normalized price of',
current_highest_price_for_city2_3f)
524             if current_highest_price_for_city2 > current_highest_price_for_city1:
525                 print(city2, 'was a better place to invest than', city1, 'in 1996-2020.')
526                 print(city2, 'had an increase in normalized price of',
current_highest_price_for_city2_3f)
527                 print(city1, 'had an increase in normalized price of',
current_highest_price_for_city1_3f)
528
529 if mode == 5:
530     # Find how long it will take to double your initial investment in city1
531     idx = city_names_list.index(city1)
532     first = 0
533     last = float_df.shape[1]
534     row = float_df.iloc[idx, :]
535     # convert row to Panadas series
536     row_series = row.squeeze(axis=0)
537     rowmax = row_series.max()
538     rowmin = row_series.min()
539     string_starting_idx = row.idxmin()
540     numeric_starting_idx = 0

```

```

541 numeric_ending_idx = 0
542
543 month_dict = {}
544 months = float_df.columns.values.tolist()
545 for i in range(0, len(months)):
546     month_dict[months[i]] = i
547     numeric_starting_idx = month_dict[string_starting_idx]
548     double_price = rowmin * 2
549     current_price = 999999
550     for j in range(numeric_starting_idx, len(months)):
551         current_price = row[j]
552         if current_price >= double_price:
553             numeric_ending_idx = j
554             break
555     #convert numeric_ending_idx to string_ending_idx
556     string_ending_idx = months[numeric_ending_idx]
557     min_years = (numeric_ending_idx - numeric_starting_idx) / 12
558     start_year = string_starting_idx[-4:]
559     end_year = string_ending_idx[-4:]
560     int_start_year = int(start_year)
561     int_end_year = int(end_year)
562     int_start_index = int_start_year - 1996
563     int_end_index = int_end_year - 1996
564     senators = []
565     state_abbr = city1[-2:]
566     if state_abbr == 'AL':
567         senators = AL_senators[int_start_index:int_end_index]
568
569     if state_abbr == 'AZ':
570         senators = AZ_senators[int_start_index:int_end_index]
571
572     if state_abbr == 'AR':
573         senators = AR_senators[int_start_index:int_end_index]
574
575     if state_abbr == 'CA':
576         senators = CA_senators[int_start_index:int_end_index]
577
578     if state_abbr == 'CO':
579         senators = CO_senators[int_start_index:int_end_index]
580
581     if state_abbr == 'CT':
582         senators = CT_senators[int_start_index:int_end_index]
583
584     if state_abbr == 'FL':
585         senators = FL_senators[int_start_index:int_end_index]
586
587     if state_abbr == 'GA':
588         senators = GA_senators[int_start_index:int_end_index]
589
590     if state_abbr == 'HI':
591         senators = HI_senators[int_start_index:int_end_index]
592

```

```

593 if state_abbr == 'ID':
594 senators = ID_senators[int_start_index:int_end_index]
595
596 if state_abbr == 'IL':
597 senators = IL_senators[int_start_index:int_end_index]
598
599 if state_abbr == 'IN':
600 senators = IN_senators[int_start_index:int_end_index]
601
602 if state_abbr == 'IA':
603 senators = IA_senators[int_start_index:int_end_index]
604
605 if state_abbr == 'KS':
606 senators = KS_senators[int_start_index:int_end_index]
607
608 if state_abbr == 'KY':
609 senators = KY_senators[int_start_index:int_end_index]
610
611 if state_abbr == 'LA':
612 senators = LA_senators[int_start_index:int_end_index]
613
614 if state_abbr == 'ME':
615 senators = ME_senators[int_start_index:int_end_index]
616
617 if state_abbr == 'MD':
618 senators = MD_senators[int_start_index:int_end_index]
619
620 if state_abbr == 'MA':
621 senators = MA_senators[int_start_index:int_end_index]
622
623 if state_abbr == 'MI':
624 senators = MI_senators[int_start_index:int_end_index]
625
626 if state_abbr == 'MN':
627 senators = MN_senators[int_start_index:int_end_index]
628
629 if state_abbr == 'MS':
630 senators = MS_senators[int_start_index:int_end_index]
631
632 if state_abbr == 'MO':
633 senators = MO_senators[int_start_index:int_end_index]
634
635 if state_abbr == 'NE':
636 senators = NE_senators[int_start_index:int_end_index]
637
638 if state_abbr == 'NV':
639 senators = NV_senators[int_start_index:int_end_index]
640
641 if state_abbr == 'NJ':
642 senators = NJ_senators[int_start_index:int_end_index]
643
644 if state_abbr == 'NM':
645 senators = NM_senators[int_start_index:int_end_index]

```

```

646
647 if state_abbr == 'NY':

648 senators = NY_senators[int_start_index:int_end_index]
649
650 if state_abbr == 'NC':
651 senators = NC_senators[int_start_index:int_end_index]
652
653 if state_abbr == 'OH':
654 senators = OH_senators[int_start_index:int_end_index]
655
656 if state_abbr == 'OK':
657 senators = OK_senators[int_start_index:int_end_index]
658
659 if state_abbr == 'OR':
660 senators = OR_senators[int_start_index:int_end_index]
661
662 if state_abbr == 'PA':
663 senators = PA_senators[int_start_index:int_end_index]
664
665 if state_abbr == 'RI':
666 senators = RI_senators[int_start_index:int_end_index]
667
668 if state_abbr == 'SC':
669 senators = SC_senators[int_start_index:int_end_index]
670
671 if state_abbr == 'TN':
672 senators = TN_senators[int_start_index:int_end_index]
673
674 if state_abbr == 'TX':
675 senators = TX_senators[int_start_index:int_end_index]
676
677 if state_abbr == 'UT':
678 senators = UT_senators[int_start_index:int_end_index]
679
680 if state_abbr == 'VA':
681 senators = VA_senators[int_start_index:int_end_index]
682
683 if state_abbr == 'WA':
684 senators = WA_senators[int_start_index:int_end_index]
685
686 if state_abbr == 'WI':
687 senators = WI_senators[int_start_index:int_end_index]
688 if min_years <= 0:
689 print('The shortest amount of time to double your money in', city1, 'is
unknown. Senate representation during the doubling time cannot be
determined.')
690 else:
691 min_years_1f = "{:.1f}".format(min_years)
692 print('The minimum price was:', rowmin)
693 print('The minimum price occurred on:', string_starting_idx)
694 if city1 == 'Washington, DC':
695 print(city1, 'This city does not vote for U.S. senators.')

```

```

696 else:
697     print(city1, 'was represented by', senators, 'in the U.S. Senate during
this time.')
698     print('A price greater than or equal to twice the minimum price first
occured on:', string_ending_idx)
699     print('The shortest amount of time to double your money in', city1, 'was',
min_years_1f, 'years.')
700
701
702 if mode == 6:
703     # compute the derivative of normalized price w.r.t. time for a city, determine
its good years and poor years in the dataset,
704     # and make a prediction about next year based on the derivative in 2021.
705     sz = city_name_size_rank_dict[city1]
706     derivative_list = []
707     current_year_and_deriv_tuple_list = []
708     good_years_list = []

709     fair_years_list = []
710     poor_years_list = []
711     current_year = 1996
712     print('TABLE OF THE FIRST DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
713     print('-----')
714     for i in range(0,300,12):
715         year = float_df.iloc[:, i:i+12]
716         row = year.iloc[sz, :]
717         rowmax = row.max()
718         norm_series = row.divide(rowmax)
719         initial_price = norm_series[0]
720         final_price = norm_series[len(norm_series) - 1]
721         change_in_price = final_price - initial_price
722         change_in_time_days = 12
723         dp_dt = change_in_price / change_in_time_days
724         derivative_list.append(dp_dt)
725         current_year_and_deriv_tuple = (current_year, dp_dt)
726         current_year_and_deriv_tuple_list.append(current_year_and_deriv_tuple)
727         print('|', current_year, '|', dp_dt, '|')
728         current_year = current_year + 1
729
730     partial_year = float_df.iloc[:, 300:308]
731     row = partial_year.iloc[sz, :]
732     rowmax = row.max()
733     norm_series = row.divide(rowmax)
734     initial_price = norm_series[0]
735     final_price = norm_series[len(norm_series) - 1]
736     change_in_price = final_price - initial_price
737     change_in_time_days = 12
738     dp_dt = change_in_price / change_in_time_days
739     derivative_list.append(dp_dt)
740     current_year_and_deriv_tuple = (current_year, dp_dt)
741     current_year_and_deriv_tuple_list.append(current_year_and_deriv_tuple)
742     print('|', current_year, '|', dp_dt, '|')

```

```

743 current_year = current_year + 1
744 print('-----')
745 derivative_series = pd.Series(derivative_list)
746 series_median = derivative_series.median()
747 print('\n\nThe median derivative in', city_names_list[sz], 'was:', series_median)
748 current_year = current_year_and_deriv_tuple_list
749 for j in range(0, len(derivative_list)):
750     current_tuple = current_year_and_deriv_tuple_list[j]
751     current_year = current_tuple[0]
752     # a year is considered a "good year" if the derivative of price w.r.t time
    is greater than or equal to 20% above the median derivative
753     if current_tuple[1] >= series_median * 1.2:
754         good_years_list.append(current_year)
755     # a year is considered a "fair year" if the derivative of price w.r.t time
    is between 20% below the median derivative and 20% above the median
    derivative
756     if (current_tuple[1] > series_median * 0.8) and (current_tuple[1] <
    series_median * 1.2):
757         fair_years_list.append(current_year)
758     # a year is considered a "poor year" if the derivative of price w.r.t time
    is less than or equal to 20% below the median derivative
759     if current_tuple[1] <= series_median * 0.8:
760         poor_years_list.append(current_year)
761
762     current_tuple = current_year_and_deriv_tuple_list[len(
    current_year_and_deriv_tuple_list) - 1]
763     current_year = current_tuple[0]
764
765     print(tm.colored('The good years to invest in', 'green'), tm.colored(
    city_names_list[sz], 'green'), tm.colored('were:', 'green'), tm.colored(
    good_years_list, 'green'))
766     print(tm.colored('The fair years to invest in', 'yellow'), tm.colored(
    city_names_list[sz], 'yellow'), tm.colored('were:', 'yellow'), tm.colored(
    fair_years_list, 'yellow'))
767     print(tm.colored('The poor years to invest in', 'red'), tm.colored(
    city_names_list[sz], 'red'), tm.colored('were:', 'red'), tm.colored(
    poor_years_list, 'red'))
768
769     # code to plot the derivative graph
770     x_axis_values = np.arange(1996, 2022, 1)
771     plt.rcParams["figure.figsize"] = [7.00, 3.50]
772     plt.rcParams["figure.autolayout"] = True
773     x = x_axis_values
774     y = derivative_list
775     default_x_ticks = range(len(x))
776     plt.plot(default_x_ticks, y)
777     title_str = "Graph of the First Derivative for " + city_names_list[sz]
778     plt.title(title_str)
779     plt.xticks(default_x_ticks, x)
780     plt.show()
781
782

```

```

783 if mode == 7:
784
785 # compute the second derivative of normalized price w.r.t. time for a city.
786 sz = city_name_size_rank_dict[city1]
787 first_derivative_list = []
788 second_derivative_list = []
789 current_year_and_first_deriv_tuple_list = []
790 current_year_and_second_deriv_tuple_list = []
791 current_year = 1996
792 print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
793 print('-----')
794 for i in range(0,308,12):
795 year = float_df.iloc[:, i:i+12]
796 if current_year == 2021:
797 year = float_df.iloc[:, i:i+8]
798 row = year.iloc[sz, :]
799 rowmax = row.max()
800 norm_series = row.divide(rowmax)
801 initial_price = norm_series[0]
802 final_price = norm_series[len(norm_series) - 1]
803 change_in_price = final_price - initial_price
804 change_in_time_days = 12
805 dp_dt = change_in_price / change_in_time_days
806 first_derivative_list.append(dp_dt)
807 current_year_and_first_deriv_tuple = (current_year, dp_dt)
808 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
809 current_year = current_year + 1
810 current_year = 1996
811 for j in range(0, 25):
812 change_in_derivative_of_price = first_derivative_list[j + 1] -
first_derivative_list[j]
813 change_in_time_years = 1 # year
814 d2p_dt2 = change_in_derivative_of_price / change_in_time_years
815 second_derivative_list.append(d2p_dt2)
816 current_year_and_second_deriv_tuple = (current_year, d2p_dt2)
817 current_year_and_second_deriv_tuple_list.append(
current_year_and_second_deriv_tuple)
818 print('|', current_year, '|', d2p_dt2, '|')
819 current_year = current_year + 1
820 # code to plot the derivative graph
821 x_axis_values = np.arange(1996, 2021, 1)
822 plt.rcParams["figure.figsize"] = [7.00, 3.50]
823 plt.rcParams["figure.autolayout"] = True
824 x = x_axis_values
825 y = second_derivative_list

826 default_x_ticks = range(len(x))
827 plt.plot(default_x_ticks, y)
828 title_str = "Graph of the Second Derivative for " + city_names_list[sz]
829 plt.title(title_str)
830 plt.xticks(default_x_ticks, x)

```

```

831 plt.show()
832
833 partial_year = float_df.iloc[:, 300:308]
834 row = partial_year.iloc[sz, :]
835 rowmax = row.max()
836 norm_series = row.divide(rowmax)
837 initial_price = norm_series[0]
838 final_price = norm_series[len(norm_series) - 1]
839 change_in_price = final_price - initial_price
840 change_in_time_days = 12
841 dp_dt = change_in_price / change_in_time_days
842 first_derivative_list.append(dp_dt)
843 current_year_and_first_deriv_tuple = (current_year, dp_dt)
844 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
845 print('|', current_year, '|', dp_dt, '|')
846 current_year = current_year + 1
847 print('-----')
848 first_derivative_series = pd.Series(first_derivative_list)
849 series_median = first_derivative_series.median()
850 print('\nThe median derivative in', city_names_list[sz], 'was:', series_median)
851 current_year = current_year_and_first_deriv_tuple_list
852
853
854 if mode == 8:
855     # compute the second derivative of normalized price w.r.t. time for a city for
one year
856     sz = city_name_size_rank_dict[city1]
857     first_derivative_list = []
858     second_derivative_list = []
859     current_year_and_first_deriv_tuple_list = []
860     current_year_and_second_deriv_tuple_list = []
861     current_year = 1996
862     #print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
863     #print('-----')
864     for i in range(0,308,12):
865         year = float_df.iloc[:, i:i+12]
866         if current_year == 2021:
867             year = float_df.iloc[:, i:i+8]
868         row = year.iloc[sz, :]
869         rowmax = row.max()
870         norm_series = row.divide(rowmax)
871         initial_price = norm_series[0]
872         final_price = norm_series[len(norm_series) - 1]
873         change_in_price = final_price - initial_price
874         change_in_time_days = 12
875         dp_dt = change_in_price / change_in_time_days
876         first_derivative_list.append(dp_dt)
877         current_year_and_first_deriv_tuple = (current_year, dp_dt)
878         current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
879         current_year = current_year + 1

```

```

880 current_year = 1996
881 for j in range(0, 25):
882     change_in_derivative_of_price = first_derivative_list[j + 1] -
first_derivative_list[j]
883     change_in_time_years = 1 # year
884     d2p_dt2 = change_in_derivative_of_price / change_in_time_years
885     second_derivative_list.append(d2p_dt2)
886     current_year_and_second_deriv_tuple = (current_year, d2p_dt2)
887     current_year_and_second_deriv_tuple_list.append(

current_year_and_second_deriv_tuple)
888 if requested_year == current_year:
889     print('The second derivative of the normalized price w.r.t time in',
city1, 'in', requested_year, 'was:', d2p_dt2)
890     #print('|', current_year, '|', d2p_dt2, '|')
891     current_year = current_year + 1
892
893     plt.plot(second_derivative_list)
894     #plt.show()
895     partial_year = float_df.iloc[:, 300:308]
896     row = partial_year.iloc[sz, :]
897     rowmax = row.max()
898     norm_series = row.divide(rowmax)
899     initial_price = norm_series[0]
900     final_price = norm_series[len(norm_series) - 1]
901     change_in_price = final_price - initial_price
902     change_in_time_days = 12
903     dp_dt = change_in_price / change_in_time_days
904     first_derivative_list.append(dp_dt)
905     current_year_and_first_deriv_tuple = (current_year, dp_dt)
906     current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
907     #print('|', current_year, '|', dp_dt, '|')
908     current_year = current_year + 1
909     #print('-----')
910     first_derivative_series = pd.Series(first_derivative_list)
911     series_median = first_derivative_series.median()
912     #print('\n\nThe median derivative in', city_names_list[sz], 'was:', series_median)
913     current_year = current_year_and_first_deriv_tuple_list
914
915 if mode == 9:
916     # compute the second derivative of normalized price w.r.t. time for a city for
several years.
917     sz = city_name_size_rank_dict[city1]
918     first_derivative_list = []
919     second_derivative_list = []
920     current_year_and_first_deriv_tuple_list = []
921     current_year_and_second_deriv_tuple_list = []
922     current_year = 1996
923     #print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
924     #print('-----')
925     for i in range(0, 308, 12):

```

```

926 year = float_df.iloc[:, i:i+12]
927 if current_year == 2021:
928 year = float_df.iloc[:, i:i+8]
929 row = year.iloc[sz, :]
930 rowmax = row.max()
931 norm_series = row.divide(rowmax)
932 initial_price = norm_series[0]
933 final_price = norm_series[len(norm_series) - 1]
934 change_in_price = final_price - initial_price
935 change_in_time_days = 12
936 dp_dt = change_in_price / change_in_time_days
937 first_derivative_list.append(dp_dt)
938 current_year_and_first_deriv_tuple = (current_year, dp_dt)
939 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
940 current_year = current_year + 1
941 current_year = 1996
942 for j in range(0, 25):
943 change_in_derivative_of_price = first_derivative_list[j + 1] -
first_derivative_list[j]
944 change_in_time_years = 1 # year
945 d2p_dt2 = change_in_derivative_of_price / change_in_time_years
946 second_derivative_list.append(d2p_dt2)
947 current_year_and_second_deriv_tuple = (current_year, d2p_dt2)

948 current_year_and_second_deriv_tuple_list.append(
current_year_and_second_deriv_tuple)
949 if current_year in requested_years:
950 print('The second derivative of the normalized price w.r.t time in',
city1, 'in', current_year, 'was:', d2p_dt2)
951 #print('|', current_year, '|', d2p_dt2, '|')
952 current_year = current_year + 1
953
954 plt.plot(second_derivative_list)
955 plt.show()
956 partial_year = float_df.iloc[:, 300:308]
957 row = partial_year.iloc[sz, :]
958 rowmax = row.max()
959 norm_series = row.divide(rowmax)
960 initial_price = norm_series[0]
961 final_price = norm_series[len(norm_series) - 1]
962 change_in_price = final_price - initial_price
963 change_in_time_days = 12
964 dp_dt = change_in_price / change_in_time_days
965 first_derivative_list.append(dp_dt)
966 current_year_and_first_deriv_tuple = (current_year, dp_dt)
967 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
968 #print('|', current_year, '|', dp_dt, '|')
969 current_year = current_year + 1
970 #print('-----')
971 first_derivative_series = pd.Series(first_derivative_list)
972 series_median = first_derivative_series.median()

```

```

973 #print('\n\nThe median derivative in', city_names_list[sz], 'was:', series_median)
974 current_year = current_year_and_first_deriv_tuple_list
975
976 if mode == 10:
977 # compute the second derivative of normalized price w.r.t. time for a city and
display its +/- sign
978 sz = city_name_size_rank_dict[city1]
979 first_derivative_list = []
980 second_derivative_list = []
981 positive_second_derivative_list = []
982 negative_second_derivative_list = []
983 current_year_and_first_deriv_tuple_list = []
984 current_year_and_second_deriv_tuple_list = []
985 current_year = 1996
986 #print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
987 #print('-----')
988 for i in range(0,308,12):
989 year = float_df.iloc[:, i:i+12]
990 if current_year == 2021:
991 year = float_df.iloc[:, i:i+8]
992 row = year.iloc[sz, :]
993 rowmax = row.max()
994 norm_series = row.divide(rowmax)
995 initial_price = norm_series[0]
996 final_price = norm_series[len(norm_series) - 1]
997 change_in_price = final_price - initial_price
998 change_in_time_days = 12
999 dp_dt = change_in_price / change_in_time_days
1000 first_derivative_list.append(dp_dt)
1001 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1002 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1003 current_year = current_year + 1
1004 current_year = 1996
1005 for j in range(0, 25):
1006 change_in_derivative_of_price = first_derivative_list[j + 1] -
first_derivative_list[j]
1007 change_in_time_years = 1 # year

1008 d2p_dt2 = change_in_derivative_of_price / change_in_time_years
1009 if d2p_dt2 > 0:
1010 positive_second_derivative_list.append(current_year)
1011 else:
1012 negative_second_derivative_list.append(current_year)
1013 second_derivative_list.append(d2p_dt2)
1014 current_year_and_second_deriv_tuple = (current_year, d2p_dt2)
1015 current_year_and_second_deriv_tuple_list.append(
current_year_and_second_deriv_tuple)
1016 current_year = current_year + 1
1017
1018 #plt.plot(second_derivative_list)
1019 #plt.show()

```

```

1020 partial_year = float_df.iloc[:, 300:308]
1021 row = partial_year.iloc[sz, :]
1022 rowmax = row.max()
1023 norm_series = row.divide(rowmax)
1024 initial_price = norm_series[0]
1025 final_price = norm_series[len(norm_series) - 1]
1026 change_in_price = final_price - initial_price
1027 change_in_time_days = 12
1028 dp_dt = change_in_price / change_in_time_days
1029 first_derivative_list.append(dp_dt)
1030 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1031 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1032 #print('|', current_year, '|', dp_dt, '|')
1033 current_year = current_year + 1
1034 #print('-----')
1035 first_derivative_series = pd.Series(first_derivative_list)
1036 series_median = first_derivative_series.median()
1037 #print('\nThe median derivative in', city_names_list[sz], 'was:', series_median)
1038 current_year = current_year_and_first_deriv_tuple_list
1039 print("The second derivative was positive in these years:",
positive_second_derivative_list)
1040 print("The second derivative was negative in these years:",
negative_second_derivative_list)
1041
1042 if mode == 11:
1043 # compute the second derivative of normalized price w.r.t. time for a city.
1044 # print whether it's increasing-increasing, decreasing-decreasing, etc
1045 sz = city_name_size_rank_dict[city1]
1046 first_derivative_list = []
1047 second_derivative_list = []
1048 inc_inc_years = []
1049 inc_dec_years = []
1050 dec_inc_years = []
1051 dec_dec_years = []
1052 negative_second_derivative_list = []
1053 current_year_and_first_deriv_tuple_list = []
1054 current_year_and_second_deriv_tuple_list = []
1055 current_year = 1996
1056 #print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
1057 #print('-----')
1058 for i in range(0,308,12):
1059 year = float_df.iloc[:, i:i+12]
1060 if current_year == 2021:
1061 year = float_df.iloc[:, i:i+8]
1062 row = year.iloc[sz, :]
1063 rowmax = row.max()
1064 norm_series = row.divide(rowmax)
1065 initial_price = norm_series[0]
1066 final_price = norm_series[len(norm_series) - 1]
1067 change_in_price = final_price - initial_price
1068 change_in_time_days = 12

```

```

1069 dp_dt = change_in_price / change_in_time_days

1070 first_derivative_list.append(dp_dt)
1071 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1072 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1073 current_year = current_year + 1
1074 current_year = 1996
1075 for j in range(0, 25):
1076 change_in_derivative_of_price = first_derivative_list[j + 1] -
first_derivative_list[j]
1077 change_in_time_years = 1 # year
1078 d2p_dt2 = change_in_derivative_of_price / change_in_time_years
1079 if ((first_derivative_list[j] > 0) and (d2p_dt2 > 0)):
1080 inc_inc_years.append(current_year)
1081 elif ((first_derivative_list[j] > 0) and (d2p_dt2 < 0)):
1082 inc_dec_years.append(current_year)
1083 elif ((first_derivative_list[j] < 0) and (d2p_dt2 > 0)):
1084 dec_inc_years.append(current_year)
1085 elif ((first_derivative_list[j] < 0) and (d2p_dt2 < 0)):
1086 dec_dec_years.append(current_year)
1087 current_year_and_second_deriv_tuple = (current_year, d2p_dt2)
1088 current_year_and_second_deriv_tuple_list.append(
current_year_and_second_deriv_tuple)
1089 current_year = current_year + 1
1090
1091 #plt.plot(second_derivative_list)
1092 #plt.show()
1093 partial_year = float_df.iloc[:, 300:308]
1094 row = partial_year.iloc[sz, :]
1095 rowmax = row.max()
1096 norm_series = row.divide(rowmax)
1097 initial_price = norm_series[0]
1098 final_price = norm_series[len(norm_series) - 1]
1099 change_in_price = final_price - initial_price
1100 change_in_time_days = 12
1101 dp_dt = change_in_price / change_in_time_days
1102 first_derivative_list.append(dp_dt)
1103 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1104 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1105 #print('|', current_year, '|', dp_dt, '|')
1106 current_year = current_year + 1
1107 #print('-----')
1108 first_derivative_series = pd.Series(first_derivative_list)
1109 series_median = first_derivative_series.median()
1110 #print('\nThe median derivative in', city_names_list[sz], 'was:', series_median)
1111 current_year = current_year_and_first_deriv_tuple_list
1112 print("The median price of a single-family home was increasing at an increasing
rate in: ", inc_inc_years)
1113 print("The median price of a single-family home was increasing at an decreasing
rate in: ", inc_dec_years)
1114 print("The median price of a single-family home was decreasing at an increasing

```

```

rate in: ", dec_inc_years)
1115 print("The median price of a single-family home was decreasing at an decreasing
rate in: ", dec_dec_years)
1116
1117 if mode == 12:
1118 complete_rows_df = original_df.dropna()
1119 state_names_df = complete_rows_df.drop(columns=['RegionID', 'RegionType',
'SizeRank'])
1120 west_coast_region_df = state_names_df.loc[state_names_df['StateName'].isin(
west_coast_states)]
1121 rocky_mountain_region_df = state_names_df.loc[state_names_df['StateName'].isin(
rocky_mountain_states)]
1122 gulf_coast_region_df = state_names_df.loc[state_names_df['StateName'].isin(
gulf_coast_states)]
1123 midwest_region_df = state_names_df.loc[state_names_df['StateName'].isin(
midwest_states)]

1124 east_coast_region_df = state_names_df.loc[state_names_df['StateName'].isin(
east_coast_states)]
1125
1126 current_year = 1996
1127
1128 west_coast_float_df = west_coast_region_df
1129 rocky_mountain_float_df = rocky_mountain_region_df.drop(columns=['StateName'])
1130 gulf_coast_float_df = gulf_coast_region_df.drop(columns=['StateName'])
1131 midwest_float_df = midwest_region_df.drop(columns=['StateName'])
1132 east_coast_float_df = east_coast_region_df.drop(columns=['StateName'])
1133
1134 CA_highs = []
1135 CA_lows = []
1136 # for loop for west coast
1137 for i in range(0,300,12):
1138 year_df = west_coast_float_df.iloc[:, i:i+12]
1139 for index, row in year_df.iterrows():
1140 prices = row[2:]
1141 maximum_price = prices.max()
1142 normalized_prices = prices.divide(maximum_price)
1143 highest_price_for_this_city = normalized_prices.max()
1144 lowest_price_for_this_city = normalized_prices.min()
1145 city_name = row[0]
1146 city_highest_price_tuple = (city_name, highest_price_for_this_city)
1147 city_lowest_price_tuple = (city_name, lowest_price_for_this_city)
1148 # append the tuple to a list
1149 if city_name[-2:] == 'CA':
1150 CA_highs.append(city_highest_price_tuple)
1151 CA_lows.append(city_lowest_price_tuple)
1152
1153 print('CA highs', CA_highs)
1154 print('CA lows', CA_lows)
1155
1156 if mode == 13:
1157 threshold = (upper_bound + lower_bound) / 2
1158 # total number of cities seen so far

```

```

1159 total_count = 1
1160 # total number of cities mis-predicted so far
1161 error_count = 0
1162 # list to store sum of squared errors
1163 see_list = []
1164
1165 for city1, sz in city_name_size_rank_dict.items():
1166 # compute the derivative of normalized price w.r.t. time for a city,
1167 # determine its good years and poor years in the dataset,
1168 # and make a prediction about next year based on the derivative in
1169 2020.
1170 derivative_list = []
1171 current_year_and_deriv_tuple_list = []
1172 good_years_list = []
1173 fair_years_list = []
1174 poor_years_list = []
1175 current_year = 1996
1176 #print('TABLE OF THE DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
1177 city_names_list[sz])
1178 #print('-----')
1179 for i in range(0,288,12):
1180 year = float_df.iloc[:, i:i+12]
1181 row = year.iloc[sz, :]
1182 rowmax = row.max()
1183 norm_series = row.divide(rowmax)
1184 initial_price = norm_series[0]
1185 final_price = norm_series[len(norm_series) - 1]
1186 change_in_price = final_price - initial_price
1187 change_in_time_days = 12
1188 dp_dt = change_in_price / change_in_time_days
1189 derivative_list.append(dp_dt)
1190
1191 current_year_and_deriv_tuple = (current_year, dp_dt)
1192 current_year_and_deriv_tuple_list.append(current_year_and_deriv_tuple)
1193 #print('|', current_year, '|', dp_dt, '|')
1194 current_year = current_year + 1
1195
1196 #print('-----')
1197 derivative_series = pd.Series(derivative_list)
1198 series_median = derivative_series.median()
1199 #print('\nThe median derivative in', city_names_list[sz], 'was:',
1200 series_median)
1201 current_year = current_year_and_deriv_tuple_list
1202 for j in range(0, len(derivative_list)):
1203 current_tuple = current_year_and_deriv_tuple_list[j]
1204 current_year = current_tuple[0]
1205 # a year is considered a "good year" if the derivative of price w.r.t
1206 time is greater than or equal to 10% above the median derivative
1207 if current_tuple[1] >= series_median * 1.1:
1208 good_years_list.append(current_year)
1209 # a year is considered a "fair year" if the derivative of price w.r.t
1210 time is between 10% below the median derivative and 10% above the
1211 median derivative

```

```

1204 if (current_tuple[1] > series_median * 0.9) and (current_tuple[1] <
series_median * 1.1):
1205 fair_years_list.append(current_year)
1206 # a year is considered a "poor year" if the derivative of price w.r.t
time is less than or equal to 10% below the median derivative
1207 if current_tuple[1] <= series_median * 0.9:
1208 poor_years_list.append(current_year)
1209
1210 current_tuple = current_year_and_deriv_tuple_list[len(
current_year_and_deriv_tuple_list) - 1]
1211 current_year = current_tuple[0]
1212
1213 # compute the derivative of price w.r.t. for the test year 2020
1214 test_year = float_df.iloc[:, 288:300]
1215 row = test_year.iloc[sz, :]
1216 rowmax = row.max()
1217 norm_series = row.divide(rowmax)
1218 initial_price = norm_series[0]
1219 final_price = norm_series[len(norm_series) - 1]
1220 change_in_price = final_price - initial_price
1221 change_in_time_days = 12
1222 dp_dt = change_in_price / change_in_time_days
1223 derivative_list.append(dp_dt)
1224 test_tuple = (2020, dp_dt)
1225
1226 # predicted condition
1227 # 0 == poor year to invest / slower than expected growth in price
1228 # 1 == fair year to invest / expected growth in price
1229 # 2 == good year to invest / faster than expected growth in price
1230 predicted_condition = 0
1231
1232 # actual condition
1233 # 0 == poor year to invest / slower than expected growth in price
1234 # 1 == fair year to invest / expected growth in price
1235 # 2 == good year to invest / faster than expected growth in price
1236 actual_condition = 0
1237
1238 # a year is considered a "good year" if the derivative of price w.r.t time
is greater than or equal to 20% above the median derivative
1239 if current_tuple[1] >= series_median * upper_bound:
1240 print(tm.colored('2020 is predicted to be a good year to invest in',
'green'), tm.colored(city_names_list[sz], 'green'), tm.colored('
because the predicted derivative is: ', 'green'), current_tuple[1])
1241 predicted_condition = 2
1242 # a year is considered a "fair year" if the derivative of price w.r.t time
is between 20% below the median derivative and 20% above the median
derivative
1243 elif (current_tuple[1] > series_median * lower_bound) and (current_tuple[1]
< series_median * upper_bound):
1244 print(tm.colored('2020 is predicted to be a fair year to invest in',
'yellow'), tm.colored(city_names_list[sz], 'yellow'), tm.colored('
because the predicted derivative is: ', 'yellow'), current_tuple[1])

```

```

1245 predicted_condition = 1
1246 # a year is considered a "poor year" if the derivative of price w.r.t time
is less than or equal to 20% below the median derivative
1247 else:
1248 print(tm.colored('2020 is predicted to be a poor year to invest in',
'red'), tm.colored(city_names_list[sz], 'red'), tm.colored(' because
the predicted derivative is: ', 'red'), current_tuple[1])
1249 predicted_condition = 0
1250 # if the actual derivative of price w.r.t. time for the year was >= the
predicted derivative of price w.r.t. for the year,
1251 # then it was a good year for that city because real estate prices grew
faster than expected
1252 if test_tuple[1] >= series_median * upper_bound:
1253 print(tm.colored('2020 was a good year to invest in', 'green'), tm.
colored(city_names_list[sz], 'green'), tm.colored(' because the
derivative was: ', 'green'), test_tuple[1])
1254 actual_condition = 2
1255 elif (test_tuple[1] > series_median * lower_bound) and (test_tuple[1] <
series_median * upper_bound):
1256 print(tm.colored('2020 was a fair year to invest in', 'yellow'), tm.
colored(city_names_list[sz], 'yellow'), tm.colored(' because the
derivative was: ', 'yellow'), test_tuple[1])
1257 actual_condition = 1
1258 # if the actual derivative of price w.r.t. time for the year was < the
predicted derivative of price w.r.t. for the year,
1259 # then it was a bad year for that city because real estate prices grew
slower than expected or decreased
1260 else:
1261 print(tm.colored('2020 was a poor year to invest in', 'red'), tm.colored
(city_names_list[sz], 'red'), tm.colored(' because the derivative was: ',
'red'), test_tuple[1])
1262 actual_condition = 0
1263 sse = np.sqrt((current_tuple[1] - test_tuple[1]) ** 2)
1264 see_list.append(sse)
1265 total_count = total_count + 1
1266 if predicted_condition != actual_condition:
1267 error_count = error_count + 1
1268 # error ratio
1269 error_ratio = error_count / total_count
1270 print("The total number of cities seen is: ", total_count)
1271 print("The total number of cities mis-predicted is: ", error_count)
1272 print("The error ratio for threshold ==", threshold, "is: ", error_ratio)
1273
1274 if mode == 15:
1275 # compute the derivative of normalized price w.r.t. time for a city, determine
its good years and poor years in the dataset,
1276 # and make a prediction about next year based on the derivative in 2021.
1277 sz = city_name_size_rank_dict[city1]
1278 derivative_list = []
1279 current_year_and_deriv_tuple_list = []
1280 good_years_list = []
1281 fair_years_list = []
1282 poor_years_list = []

```

```

1283 current_year = 1996
1284 print('TABLE OF THE FIRST DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
1285 print('-----')
1286 for i in range(0,300,12):
1287 year = float_df.iloc[:, i:i+12]
1288 row = year.iloc[sz, :]

1289 rowmax = row.max()
1290 norm_series = row.divide(rowmax)
1291 initial_price = norm_series[0]
1292 final_price = norm_series[len(norm_series) - 1]
1293 change_in_price = final_price - initial_price
1294 change_in_time_days = 12
1295 dp_dt = change_in_price / change_in_time_days
1296 derivative_list.append(dp_dt)
1297 current_year_and_deriv_tuple = (current_year, dp_dt)
1298 current_year_and_deriv_tuple_list.append(current_year_and_deriv_tuple)
1299 print('|', current_year, '|', dp_dt, '|')
1300 current_year = current_year + 1
1301
1302 partial_year = float_df.iloc[:, 300:308]
1303 row = partial_year.iloc[sz, :]
1304 rowmax = row.max()
1305 norm_series = row.divide(rowmax)
1306 initial_price = norm_series[0]
1307 final_price = norm_series[len(norm_series) - 1]
1308 change_in_price = final_price - initial_price
1309 change_in_time_days = 12
1310 dp_dt = change_in_price / change_in_time_days
1311 derivative_list.append(dp_dt)
1312 current_year_and_deriv_tuple = (current_year, dp_dt)
1313 current_year_and_deriv_tuple_list.append(current_year_and_deriv_tuple)
1314 print('|', current_year, '|', dp_dt, '|')
1315 current_year = current_year + 1
1316 print('-----')
1317 derivative_series = pd.Series(derivative_list)
1318 series_median = derivative_series.median()
1319 print('\n\nThe median derivative in', city_names_list[sz], 'was:', series_median)
1320 current_year = current_year_and_deriv_tuple_list
1321 for j in range(0, len(derivative_list)):
1322 current_tuple = current_year_and_deriv_tuple_list[j]
1323 current_year = current_tuple[0]
1324 # a year is considered a "good year" if the derivative of price w.r.t time
is greater than or equal to 20% above the median derivative
1325 if current_tuple[1] >= series_median * 1.2:
1326 good_years_list.append(current_year)
1327 # a year is considered a "fair year" if the derivative of price w.r.t time
is between 20% below the median derivative and 20% above the median
derivative
1328 if (current_tuple[1] > series_median * 0.8) and (current_tuple[1] <
series_median * 1.2):
1329 fair_years_list.append(current_year)

```

```

1330 # a year is considered a "poor year" if the derivative of price w.r.t time
is less than or equal to 20% below the median derivative
1331 if current_tuple[1] <= series_median * 0.8:
1332     poor_years_list.append(current_year)
1333
1334 current_tuple = current_year_and_deriv_tuple_list[len(
current_year_and_deriv_tuple_list) - 1]
1335 current_year = current_tuple[0]
1336
1337 #print(tm.colored('The good years to invest in', 'green'),
tm.colored(city_names_list[sz], 'green'), tm.colored('were:', 'green'),
tm.colored(good_years_list, 'green'))
1338 #print(tm.colored('The fair years to invest in', 'yellow'),
tm.colored(city_names_list[sz], 'yellow'), tm.colored('were:', 'yellow'),
tm.colored(fair_years_list, 'yellow'))
1339 #print(tm.colored('The poor years to invest in', 'red'),
tm.colored(city_names_list[sz], 'red'), tm.colored('were:', 'red'),
tm.colored(poor_years_list, 'red'))
1340
1341 # code to plot the derivative graph
1342 #x_axis_values = np.arange(1996, 2022, 1)
1343 #plt.rcParams["figure.figsize"] = [7.00, 3.50]

1344 #plt.rcParams["figure.autolayout"] = True
1345 #x = x_axis_values
1346 #y = derivative_list
1347 #default_x_ticks = range(len(x))
1348 #plt.plot(default_x_ticks, y)
1349 #title_str = "Graph of the First Derivative for " + city_names_list[sz]
1350 #plt.title(title_str)
1351 #plt.xticks(default_x_ticks, x)
1352 #plt.show()
1353
1354 # compute the second derivative of normalized price w.r.t. time for a city.
1355 # print whether it's increasing-increasing, decreasing-decreasing, etc
1356 sz = city_name_size_rank_dict[city1]
1357 first_derivative_list = []
1358 second_derivative_list = []
1359 inc_inc_years = []
1360 inc_dec_years = []
1361 dec_inc_years = []
1362 dec_dec_years = []
1363 negative_second_derivative_list = []
1364 current_year_and_first_deriv_tuple_list = []
1365 current_year_and_second_deriv_tuple_list = []
1366 current_year = 1996
1367 #print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
1368 #print('-----')
1369 for i in range(0,308,12):
1370     year = float_df.iloc[:, i:i+12]
1371     if current_year == 2021:
1372         year = float_df.iloc[:, i:i+8]

```

```

1373 row = year.iloc[sz, :]
1374 rowmax = row.max()
1375 norm_series = row.divide(rowmax)
1376 initial_price = norm_series[0]
1377 final_price = norm_series[len(norm_series) - 1]
1378 change_in_price = final_price - initial_price
1379 change_in_time_days = 12
1380 dp_dt = change_in_price / change_in_time_days
1381 first_derivative_list.append(dp_dt)
1382 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1383 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1384 current_year = current_year + 1
1385 current_year = 1996
1386 for j in range(0, 25):
1387 change_in_derivative_of_price = first_derivative_list[j + 1] -
first_derivative_list[j]
1388 change_in_time_years = 1 # year
1389 d2p_dt2 = change_in_derivative_of_price / change_in_time_years
1390 if ((first_derivative_list[j] > 0) and (d2p_dt2 > 0)):
1391 inc_inc_years.append(current_year)
1392 elif ((first_derivative_list[j] > 0) and (d2p_dt2 < 0)):
1393 inc_dec_years.append(current_year)
1394 elif ((first_derivative_list[j] < 0) and (d2p_dt2 > 0)):
1395 dec_inc_years.append(current_year)
1396 elif ((first_derivative_list[j] < 0) and (d2p_dt2 < 0)):
1397 dec_dec_years.append(current_year)
1398 current_year_and_second_deriv_tuple = (current_year, d2p_dt2)
1399 current_year_and_second_deriv_tuple_list.append(
current_year_and_second_deriv_tuple)
1400 current_year = current_year + 1
1401
1402 #plt.plot(second_derivative_list)
1403 #plt.show()
1404 partial_year = float_df.iloc[:, 300:308]
1405 row = partial_year.iloc[sz, :]
1406 rowmax = row.max()

1407 norm_series = row.divide(rowmax)
1408 initial_price = norm_series[0]
1409 final_price = norm_series[len(norm_series) - 1]
1410 change_in_price = final_price - initial_price
1411 change_in_time_days = 12
1412 dp_dt = change_in_price / change_in_time_days
1413 first_derivative_list.append(dp_dt)
1414 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1415 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1416 #print('|', current_year, '|', dp_dt, '|')
1417 current_year = current_year + 1
1418 #print('-----')
1419 first_derivative_series = pd.Series(first_derivative_list)
1420 series_median = first_derivative_series.median()

```

```

1421 #print('\nThe median derivative in', city_names_list[sz], 'was:', series_median)
1422 current_year = current_year_and_first_deriv_tuple_list
1423 #print("The median price of a single-family home was increasing at an
increasing rate in: ", inc_inc_years)
1424 #print("The median price of a single-family home was increasing at an
decreasing rate in: ", inc_dec_years)
1425 #print("The median price of a single-family home was decreasing at an
increasing rate in: ", dec_inc_years)
1426 #print("The median price of a single-family home was decreasing at an
decreasing rate in: ", dec_dec_years)
1427 for y in range(1999, 2021):
1428 if (y in good_years_list and inc_inc_years):
1429 print(y, 'was a bubble year for', city_names_list[sz])
1430
1431 if mode == 16:
1432
1433 # compute the second derivative of normalized price w.r.t. time for a city.
1434 sz = city_name_size_rank_dict[city1]
1435 first_derivative_list = []
1436 second_derivative_list = []
1437 current_year_and_first_deriv_tuple_list = []
1438 current_year_and_second_deriv_tuple_list = []
1439 current_year = 1996
1440 #print('TABLE OF THE SECOND DERIVATIVE OF THE NORMALIZED PRICE W.R.T TIME IN',
city_names_list[sz])
1441 #print('-----')
1442 for i in range(0,300,12):
1443 year = float_df.iloc[:, i:i+12]
1444 row = year.iloc[sz, :]
1445 print(row)
1446 rowmax = row.max()
1447 norm_series = row.divide(rowmax)
1448 initial_price = norm_series[0]
1449 final_price = norm_series[len(norm_series) - 1]
1450 change_in_price = final_price - initial_price
1451 change_in_time_days = 12
1452 dp_dt = change_in_price / change_in_time_days
1453 first_derivative_list.append(dp_dt)
1454 current_year_and_first_deriv_tuple = (current_year, dp_dt)
1455 current_year_and_first_deriv_tuple_list.append(
current_year_and_first_deriv_tuple)
1456 current_year = current_year + 1
1457
1458 # write a color-coded list of cities to a file based on their Senators' political
party and the doubling time
1459 elif mode == 17:
1460 default_stdout = sys.stdout
1461 output_file = open('party_list.txt', 'w')
1462 # redirect stdout to a file
1463 sys.stdout = output_file
1464
1465 red_cities_count = 0

```

```

1466 red_cities_total_years = 0
1467 red_cities_average_doubling_time = 0
1468
1469 blue_cities_count = 0
1470 blue_cities_total_years = 0
1471 blue_cities_average_doubling_time = 0
1472
1473 purple_cities_count = 0
1474 purple_cities_total_years = 0
1475 purple_cities_average_doubling_time = 0
1476
1477 for k in range(1, 122):
1478     city1 = region_name_column_list[k]
1479     state_abbr = city1[-2:]
1480     # special cases
1481     if city1 == "Denver, CO":
1482         latitude = "39.7392"
1483         longitude = "-104.9850"
1484     elif city1 == "Richmond, VA":
1485         latitude = "37.53"
1486         longitude = "-77.47"
1487     elif city1 == "Urban Honolulu, HI":
1488         latitude = "21.315603"
1489         longitude = "-157.858093"
1490     elif city1 == "Ventura, CA":
1491         latitude = "34.275"
1492         longitude = "-119.228"
1493     elif city1 == "North Port-Sarasota-Bradenton, FL":
1494         cityX = "Sarasota, FL"
1495         # find cityX's location
1496         nominatim = Nominatim()
1497         city_json = nominatim.query(cityX).toJSON()[0]
1498         latitude = city_json["lat"]
1499         longitude = city_json["lon"]
1500     elif city1 == "Minneapolis-St Paul, MN":
1501         cityY = "Minneapolis, MN"
1502         # find cityY's location
1503         nominatim = Nominatim()
1504         city_json = nominatim.query(cityY).toJSON()[0]
1505         latitude = city_json["lat"]
1506         longitude = city_json["lon"]
1507     else:
1508         # find city1's location
1509         nominatim = Nominatim()
1510         city_json = nominatim.query(city1).toJSON()[0]
1511         latitude = city_json["lat"]
1512         longitude = city_json["lon"]
1513
1514     # get city1's size rank
1515     sz = city_name_size_rank_dict[city1]
1516     # Find how long it will take to double your initial investment in city1
1517     idx = city_names_list.index(city1)
1518     first = 0

```

```

1519 last = float_df.shape[1]
1520 row = float_df.iloc[idx, :]
1521 row_series = row.squeeze(axis=0)
1522 rowmax = row_series.max()
1523 rowmin = row_series.min()
1524 string_starting_idx = row.idxmin()
1525 numeric_starting_idx = 0
1526 numeric_ending_idx = 0
1527 month_dict = {}
1528 months = float_df.columns.values.tolist()
1529 for i in range(0, len(months)):
1530     month_dict[months[i]] = i
1531     numeric_starting_idx = month_dict[string_starting_idx]
1532     double_price = rowmin * 2

1533     current_price = 999999
1534     for j in range(numeric_starting_idx, len(months)):
1535         current_price = row[j]
1536         if current_price >= double_price:
1537             numeric_ending_idx = j
1538             break
1539     #convert numeric_ending_idx to string_ending_idx
1540     string_ending_idx = months[numeric_ending_idx]
1541     min_years = (numeric_ending_idx - numeric_starting_idx) / 12
1542     start_year = string_starting_idx[-4:]
1543     end_year = string_ending_idx[-4:]
1544     int_start_year = int(start_year)
1545     int_end_year = int(end_year)
1546     int_start_index = int_start_year - 1996
1547     int_end_index = int_end_year - 1996
1548     # if the median home price in the given city did not double in during the
1549     # then use the senate representation for the entire 24-year period as the
1550     # this city
1551     if int_end_index < int_start_index:
1552         int_start_index = 0
1553         int_end_index = int(len(row) / 12)
1554     if numeric_ending_idx == 0:
1555         int_start_index = 0
1556         int_end_index = int(len(row) / 12)
1557     senators = []
1558     if state_abbr == 'AL':
1559         senators = AL_senators[int_start_index:int_end_index]
1560
1561     if state_abbr == 'AZ':
1562         senators = AZ_senators[int_start_index:int_end_index]
1563
1564     if state_abbr == 'AR':
1565         senators = AR_senators[int_start_index:int_end_index]
1566
1567     if state_abbr == 'CA':
1568         senators = CA_senators[int_start_index:int_end_index]

```

```

1569
1570 if state_abbr == 'CO':
1571 senators = CO_senators[int_start_index:int_end_index]
1572
1573 if state_abbr == 'CT':
1574 senators = CT_senators[int_start_index:int_end_index]
1575
1576 if state_abbr == 'FL':
1577 senators = FL_senators[int_start_index:int_end_index]
1578
1579 if state_abbr == 'GA':
1580 senators = GA_senators[int_start_index:int_end_index]
1581
1582 if state_abbr == 'HI':
1583 senators = HI_senators[int_start_index:int_end_index]
1584
1585 if state_abbr == 'ID':
1586 senators = ID_senators[int_start_index:int_end_index]
1587
1588 if state_abbr == 'IL':
1589 senators = IL_senators[int_start_index:int_end_index]
1590
1591 if state_abbr == 'IN':
1592 senators = IN_senators[int_start_index:int_end_index]
1593
1594 if state_abbr == 'IA':
1595 senators = IA_senators[int_start_index:int_end_index]
1596
1597 if state_abbr == 'KS':
1598 senators = KS_senators[int_start_index:int_end_index]
1599
1600 if state_abbr == 'KY':
1601 senators = KY_senators[int_start_index:int_end_index]
1602
1603 if state_abbr == 'LA':
1604 senators = LA_senators[int_start_index:int_end_index]
1605
1606 if state_abbr == 'ME':
1607 senators = ME_senators[int_start_index:int_end_index]
1608
1609 if state_abbr == 'MD':
1610 senators = MD_senators[int_start_index:int_end_index]
1611
1612 if state_abbr == 'MA':
1613 senators = MA_senators[int_start_index:int_end_index]
1614
1615 if state_abbr == 'MI':
1616 senators = MI_senators[int_start_index:int_end_index]
1617
1618 if state_abbr == 'MN':
1619 senators = MN_senators[int_start_index:int_end_index]
1620

```

```

1621 if state_abbr == 'MS':
1622     senators = MS_senators[int_start_index:int_end_index]
1623
1624 if state_abbr == 'MO':
1625     senators = MO_senators[int_start_index:int_end_index]
1626
1627 if state_abbr == 'NE':
1628     senators = NE_senators[int_start_index:int_end_index]
1629
1630 if state_abbr == 'NV':
1631     senators = NV_senators[int_start_index:int_end_index]
1632
1633 if state_abbr == 'NJ':
1634     senators = NJ_senators[int_start_index:int_end_index]
1635
1636 if state_abbr == 'NM':
1637     senators = NM_senators[int_start_index:int_end_index]
1638
1639 if state_abbr == 'NY':
1640     senators = NY_senators[int_start_index:int_end_index]
1641
1642 if state_abbr == 'NC':
1643     senators = NC_senators[int_start_index:int_end_index]
1644
1645 if state_abbr == 'OH':
1646     senators = OH_senators[int_start_index:int_end_index]
1647
1648 if state_abbr == 'OK':
1649     senators = OK_senators[int_start_index:int_end_index]
1650
1651 if state_abbr == 'OR':
1652     senators = OR_senators[int_start_index:int_end_index]
1653
1654 if state_abbr == 'PA':
1655     senators = PA_senators[int_start_index:int_end_index]
1656
1657 if state_abbr == 'RI':
1658     senators = RI_senators[int_start_index:int_end_index]
1659
1660 if state_abbr == 'SC':
1661     senators = SC_senators[int_start_index:int_end_index]
1662
1663 if state_abbr == 'TN':
1664     senators = TN_senators[int_start_index:int_end_index]
1665
1666 if state_abbr == 'TX':
1667     senators = TX_senators[int_start_index:int_end_index]
1668
1669 if state_abbr == 'UT':
1670     senators = UT_senators[int_start_index:int_end_index]
1671
1672 if state_abbr == 'VA':

```

```

1673 senators = VA_senators[int_start_index:int_end_index]
1674
1675 if state_abbr == 'WA':
1676 senators = WA_senators[int_start_index:int_end_index]
1677
1678 if state_abbr == 'WI':
1679 senators = WI_senators[int_start_index:int_end_index]
1680 # define and initialize counter variables
1681 RR_ctr = 0 # number of years the city was represented by 2 Republicans
1682 DD_ctr = 0 # number of years the city was represented by 2 Democrats
1683 DR_ctr = 0 # number of years the city was represented by 1 Democrat and 1
Republican
1684 RD_ctr = 0 # number of years the city was represented by 1 Republican and 1
Democrat
1685 RI_ctr = 0 # number of years the city was represented by 1 Republican and 1
Independent
1686 DI_ctr = 0 # number of years the city was represented by 1 Democrat and 1
Independent
1687 IR_ctr = 0 # number of years the city was represented by 1 Independent and
1 Democrat
1688 ID_ctr = 0 # number of years the city was represented by 1 Independent and
1 Democrat
1689 II_ctr = 0 # number of years the city was represented by 2 Independents
1690 for m in range(0, len(senators)):
1691 if senators[m] == 'RR':
1692 RR_ctr = RR_ctr + 1
1693 elif senators[m] == 'DD':
1694 DD_ctr = DD_ctr + 1
1695 elif senators[m] == 'DR':
1696 DR_ctr = DR_ctr + 1
1697 elif senators[m] == 'RD':
1698 RD_ctr = RD_ctr + 1
1699 elif senators[m] == 'RI':
1700 RI_ctr = RI_ctr + 1
1701 elif senators[m] == 'DI':
1702 DI_ctr = DI_ctr + 1
1703 elif senators[m] == 'IR':
1704 IR_ctr = IR_ctr + 1
1705 elif senators[m] == 'ID':
1706 ID_ctr = ID_ctr + 1
1707 elif senators[m] == 'II':
1708 II_ctr = II_ctr + 1
1709
1710 # compute the total number of years that the city was represented by a
Republican during the doubling time
1711 R_total = RR_ctr + RD_ctr + DR_ctr + RI_ctr + IR_ctr
1712 # compute the total number of years that the city was represented by a
Democrat during the doubling time
1713 D_total = DD_ctr + RD_ctr + DR_ctr + DI_ctr + ID_ctr
1714 if city1 == "Washington, DC":
1715 colorIcon = "blueIcon"
1716 elif R_total > D_total:
1717 colorIcon = "redIcon"

```

```

1718 elif D_total > R_total:
1719     colorIcon = "blueIcon"
1720 elif R_total == D_total:
1721     colorIcon = "purpleIcon"
1722 elif (R_total == 0) and (D_total == 0):
1723     colorIcon = "blackIcon"

1724
1725 string_ending_idx = months[numeric_ending_idx]
1726 min_years = (numeric_ending_idx - numeric_starting_idx) / 12
1727 if (min_years > 0) and (colorIcon == "blueIcon"):
1728     blue_cities_count = blue_cities_count + 1
1729     blue_cities_total_years = blue_cities_total_years + min_years
1730 elif (min_years > 0) and (colorIcon == "redIcon"):
1731     red_cities_count = red_cities_count + 1
1732     red_cities_total_years = red_cities_total_years + min_years
1733 elif (min_years > 0) and (colorIcon == "purpleIcon"):
1734     purple_cities_count = purple_cities_count + 1
1735     purple_cities_total_years = purple_cities_total_years + min_years
1736 if min_years <= 0:
1737     min_years_1f = "unknown"
1738 else:
1739     min_years_1f = "{:.1f}".format(min_years)
1740 if min_years_1f == "unknown":
1741     print("L.marker([, latitude, ", ", longitude, ], {icon: ", colorIcon,
1742         "}).bindPopup(\"The median home price in ", city1, " did not double
1743         between 1996 and 2021.\").addTo(map);", sep="")
1742 else:
1743     print("L.marker([, latitude, ", ", longitude, ], {icon: ", colorIcon,
1744         "}).bindPopup(\"The shortest amount of time to double your money in ",
1745         city1, " was ", min_years_1f, " years.\").addTo(map);", sep="")
1744 blue_cities_average_doubling_time = blue_cities_total_years / blue_cities_count
1745 red_cities_average_doubling_time = red_cities_total_years / red_cities_count
1746 purple_cities_average_doubling_time = purple_cities_total_years /
1747     purple_cities_count
1747 print()
1748 blue_cities_average_doubling_time_1f = "{:.1f}".format(
1749     blue_cities_average_doubling_time)
1749 red_cities_average_doubling_time_1f = "{:.1f}".format(
1750     red_cities_average_doubling_time)
1750 purple_cities_average_doubling_time_1f = "{:.1f}".format(
1751     purple_cities_average_doubling_time)
1751 print('The average doubling time in the cities represented by Democratic
1752     senators was:', blue_cities_average_doubling_time_1f, ".")
1752 print('The average doubling time in the cities represented by Republican
1753     senators was:', red_cities_average_doubling_time_1f, ".")
1753 print('The average doubling time in the cities represented by Democratic and
1754     Republican senators was:', purple_cities_average_doubling_time_1f, ".")
1754
1755 sys.stdout = default_stdout
1756 output_file.close()
1757 sys.exit()
1758

```

```

1759 # write a color-coded list of cities grouped by size
1760 elif mode == 18:
1761     default_stdout = sys.stdout
1762     output_file = open('size_list.txt', 'w')
1763     # redirect stdout to a file
1764     sys.stdout = output_file
1765     colorIcon = ""
1766     # size rank is the independent variable
1767     size_rank_list = []
1768     # doubling time is the dependent variable
1769     doubling_time_list = []
1770     for k in range(1, 122):
1771         city1 = region_name_column_list[k]
1772         # special cases
1773         if city1 == "Denver, CO":
1774             latitude = "39.7392"
1775             longitude = "-104.9850"
1776         elif city1 == "Richmond, VA":
1777             latitude = "37.53"
1778             longitude = "-77.47"
1779         elif city1 == "Urban Honolulu, HI":
1780             latitude = "21.315603"
1781             longitude = "-157.858093"
1782         elif city1 == "Ventura, CA":
1783             latitude = "34.275"
1784             longitude = "-119.228"
1785         elif city1 == "North Port-Sarasota-Bradenton, FL":
1786             cityX = "Sarasota, FL"
1787             # find cityX's location
1788             nominatim = Nominatim()
1789             city_json = nominatim.query(cityX).toJSON()[0]
1790             latitude = city_json["lat"]
1791             longitude = city_json["lon"]
1792         elif city1 == "Minneapolis-St Paul, MN":
1793             cityY = "Minneapolis, MN"
1794             # find cityY's location
1795             nominatim = Nominatim()
1796             city_json = nominatim.query(cityY).toJSON()[0]
1797             latitude = city_json["lat"]
1798             longitude = city_json["lon"]
1799         else:
1800             # find city1's location
1801             nominatim = Nominatim()
1802             city_json = nominatim.query(city1).toJSON()[0]
1803             latitude = city_json["lat"]
1804             longitude = city_json["lon"]
1805             # get city1's size rank
1806             sz = city_name_size_rank_dict[city1]
1807             # Find how long it will take to double your initial investment in city1
1808             idx = city_names_list.index(city1)
1809             first = 0
1810             last = float_df.shape[1]

```

```

1811 row = float_df.iloc[idx, :]
1812 # convert row to Panadas series
1813 row_series = row.squeeze(axis=0)
1814 rowmax = row_series.max()
1815 rowmin = row_series.min()
1816 string_starting_idx = row.idxmin()
1817 numeric_starting_idx = 0
1818 numeric_ending_idx = 0
1819
1820 month_dict = {}
1821 months = float_df.columns.values.tolist()
1822 for i in range(0, len(months)):
1823     month_dict[months[i]] = i
1824 numeric_starting_idx = month_dict[string_starting_idx]
1825 double_price = rowmin * 2
1826 current_price = 999999
1827 for j in range(numeric_starting_idx, len(months)):
1828     current_price = row[j]
1829     if current_price >= double_price:
1830         numeric_ending_idx = j
1831         break
1832 # convert numeric_ending_idx to string_ending_idx
1833 string_ending_idx = months[numeric_ending_idx]
1834 min_years = (numeric_ending_idx - numeric_starting_idx) / 12
1835 if min_years <= 0:
1836     min_years_1f = "unknown"
1837 else:
1838     size_rank_list.append(sz)
1839     doubling_time_list.append(min_years)
1840     min_years_1f = "{:.1f}".format(min_years)
1841     # if the city is in the 40 largest U.S. cities by population, color it red
1842     if sz <= 41:
1843         colorIcon = 'blackIcon'
1844     # if the city is in the second 40 largest U.S. cities by population, color
1845     # it blue
1846     elif (sz >= 42) and (sz <= 81):
1847         colorIcon = 'blueIcon'
1848     # if the city is in the third 40 largest U.S. cities by population, color
1849     # it green
1850     elif (sz >= 82) and (sz <= 122):
1851         colorIcon = 'skyBlueIcon'
1852     if min_years_1f == "unknown":
1853         print("L.marker([", latitude, ", ", longitude, "], {icon: ", colorIcon,
1854             "}).bindPopup(\"The median home price in ", city1, " did not double
1855             between 1996 and 2021.\").addTo(map);", sep="")
1856     else:
1857         print("L.marker([", latitude, ", ", longitude, "], {icon: ", colorIcon,
1858             "}).bindPopup(\"The shortest amount of time to double your money in ",
1859             city1, " was ", min_years_1f, " years.\").addTo(map);", sep="")
1860 size_rank_array = np.array(size_rank_list)
1861 doubling_time_array = np.array(doubling_time_list)
1862 r = scipy.stats.pearsonr(size_rank_array, doubling_time_array)[0]

```

```

1857 print('pearson_correlation_coefficient', r)
1858 sys.stdout = default_stdout
1859 output_file.close()
1860 sys.exit()
1861
1862 # write a color-coded list of cities grouped by doubling time
1863 elif mode == 19:
1864     default_stdout = sys.stdout
1865     output_file = open('fastest_doublers.txt', 'w')
1866     # redirect stdout to a file
1867     sys.stdout = output_file
1868     number_of_cities_in_east_coast_states_where_prices_doubled_in_less_than_10_years
1869     = 0
1870     number_of_cities_in_west_coast_states_where_prices_doubled_in_less_than_10_years
1871     = 0
1872     total_number_of_cities_where_prices_doubled_in_less_than_10_years = 0
1873     number_of_interior_state_cities = 0
1874     total_doubling_time_of_interior_state_cities = 0
1875     colorIcon = ""
1876     for k in range(1, 122):
1877         city1 = region_name_column_list[k]
1878         state_abbr = city1[-2:]
1879         # special cases
1880         if city1 == "Denver, CO":
1881             latitude = "39.7392"
1882             longitude = "-104.9850"
1883         elif city1 == "Richmond, VA":
1884             latitude = "37.53"
1885             longitude = "-77.47"
1886         elif city1 == "Urban Honolulu, HI":
1887             latitude = "21.315603"
1888             longitude = "-157.858093"
1889         elif city1 == "Ventura, CA":
1890             latitude = "34.275"
1891             longitude = "-119.228"
1892         elif city1 == "North Port-Sarasota-Bradenton, FL":
1893             cityX = "Sarasota, FL"
1894             # find cityX's location
1895             nominatim = Nominatim()
1896             city_json = nominatim.query(cityX).toJSON()[0]
1897             latitude = city_json["lat"]
1898             longitude = city_json["lon"]
1899         elif city1 == "Minneapolis-St Paul, MN":
1900             cityY = "Minneapolis, MN"
1901             # find cityY's location
1902             nominatim = Nominatim()
1903             city_json = nominatim.query(cityY).toJSON()[0]
1904             latitude = city_json["lat"]
1905             longitude = city_json["lon"]
1906         else:
1907             # find city1's location
1908             nominatim = Nominatim()

```

```

1907 city_json = nominatim.query(city1).toJSON()[0]
1908 latitude = city_json["lat"]
1909 longitude = city_json["lon"]
1910 # get city1's size rank
1911 sz = city_name_size_rank_dict[city1]
1912 # Find how long it will take to double your initial investment in city1
1913 idx = city_names_list.index(city1)
1914 first = 0
1915 last = float_df.shape[1]
1916 row = float_df.iloc[idx, :]
1917 # convert row to Pandas series
1918 row_series = row.squeeze(axis=0)
1919 rowmax = row_series.max()
1920 rowmin = row_series.min()
1921 string_starting_idx = row.idxmin()
1922 numeric_starting_idx = 0
1923 numeric_ending_idx = 0
1924
1925 month_dict = {}
1926 months = float_df.columns.values.tolist()
1927 for i in range(0, len(months)):
1928     month_dict[months[i]] = i
1929 numeric_starting_idx = month_dict[string_starting_idx]
1930 double_price = rowmin * 2
1931 current_price = 999999
1932 for j in range(numeric_starting_idx, len(months)):
1933     current_price = row[j]
1934     if current_price >= double_price:
1935         numeric_ending_idx = j
1936         break
1937 #convert numeric_ending_idx to string_ending_idx
1938 string_ending_idx = months[numeric_ending_idx]
1939 min_years = (numeric_ending_idx - numeric_starting_idx) / 12
1940 if min_years <= 0:
1941     min_years_1f = "unknown"
1942 else:
1943     min_years_1f = "{:.1f}".format(min_years)
1944 # check if the city is in an interior state
1945 if (state_abbr not in ["WA", "OR", "CA"]) and (state_abbr not in ["ME", "NH",
    "MA", "RI", "CT", "NY", "NJ", "DE", "MD", "DC", "VA", "NC", "SC", "GA",
    "FL"]):
1946     number_of_interior_state_cities = number_of_interior_state_cities + 1
1947     total_doubling_time_of_interior_state_cities =
total_doubling_time_of_interior_state_cities + min_years
1948 # if the city has a doubling time between 0 and 9.9 years, color it green
1949 if (min_years > 0) and (min_years < 10.0):
1950     colorIcon = 'greenIcon'
1951 # increment the count
1952 total_number_of_cities_where_prices_doubled_in_less_than_10_years =
total_number_of_cities_where_prices_doubled_in_less_than_10_years + 1
1953 # check if the city is in a West Coast state
1954 if state_abbr in ["WA", "OR", "CA"]:
1955

```

```

number_of_cities_in_west_coast_states_where_prices_doubled_in_less_than_10_years =
number_of_cities_in_west_coast_states_where_prices_doubled_in_less_than_10_years + 1
1956 # check if city is in an East Coast state
1957 if state_abbr in ["ME", "NH", "MA", "RI", "CT", "NY", "NJ", "DE", "MD",
"DC", "VA", "NC", "SC", "GA", "FL"]:
1958
number_of_cities_in_east_coast_states_where_prices_doubled_in_less_than_10_years =
number_of_cities_in_east_coast_states_where_prices_doubled_in_less_than_10_years + 1
1959 # if the city has a doubling time between 10.0 and 19.9 years, color it
yellow
1960 elif (min_years >= 10.0) and (min_years < 20.0):
1961 colorIcon = 'yellowIcon'
1962 # if the city has a doubling time between 20.0 and 29.9 years, color it red
1963 elif (min_years >= 20) and (min_years < 29.9):
1964 colorIcon = 'blackIcon'
1965 # if the median price for a single-family home did not double in that city
between 1996 and 2021
1966 else:
1967 colorIcon = 'grayIcon'
1968 if min_years_1f == "unknown":
1969 print("L.marker([" + latitude + ", " + longitude + "], {icon: " + colorIcon +
"}).bindPopup("The median home price in " + city1 + " did not double
between 1996 and 2021.\").addTo(map);", sep="")
1970 else:
1971 print("L.marker([" + latitude + ", " + longitude + "], {icon: " + colorIcon +
"}).bindPopup("The shortest amount of time to double your money in " +
city1 + " was " + min_years_1f + " years.\").addTo(map);", sep="")
1972
1973
number_of_cities_in_states_on_either_coast_where_prices_doubled_in_less_than_10_years =
number_of_cities_in_west_coast_states_where_prices_doubled_in_less_than_10_years
+
number_of_cities_in_east_coast_states_where_prices_doubled_in_less_than_10_years
1974 fraction_of_cities_in_coastal_states_where_prices_doubled_in_less_than_10_years
=
number_of_cities_in_states_on_either_coast_where_prices_doubled_in_less_than_10_years
/ total_number_of_cities_where_prices_doubled_in_less_than_10_years
1975
fraction_of_cities_in_coastal_states_where_prices_doubled_in_less_than_10_years_2
f = "{:.2f}".format(
fraction_of_cities_in_coastal_states_where_prices_doubled_in_less_than_10_years)
1976 print("\n",
fraction_of_cities_in_coastal_states_where_prices_doubled_in_less_than_10_years_2
f, "percent of the cities that doubled in less than 10 years were located in a
state that borders the Atlantic Ocean or the Pacific Ocean.")
1977
1978 average_doubling_time_of_interior_state_cities =

```

```
total_doubling_time_of_interior_state_cities / number_of_interior_state_cities
1979 print('average_doubling_time_of_interior_state_cities:',
average_doubling_time_of_interior_state_cities)
1980 sys.stdout = default_stdout
1981 output_file.close()
1982 sys.exit()
```

