Zillow Real Estate Project Overview

By Alex Thompson, Rohil Navani, and Hailey Nguyen

Project Goal

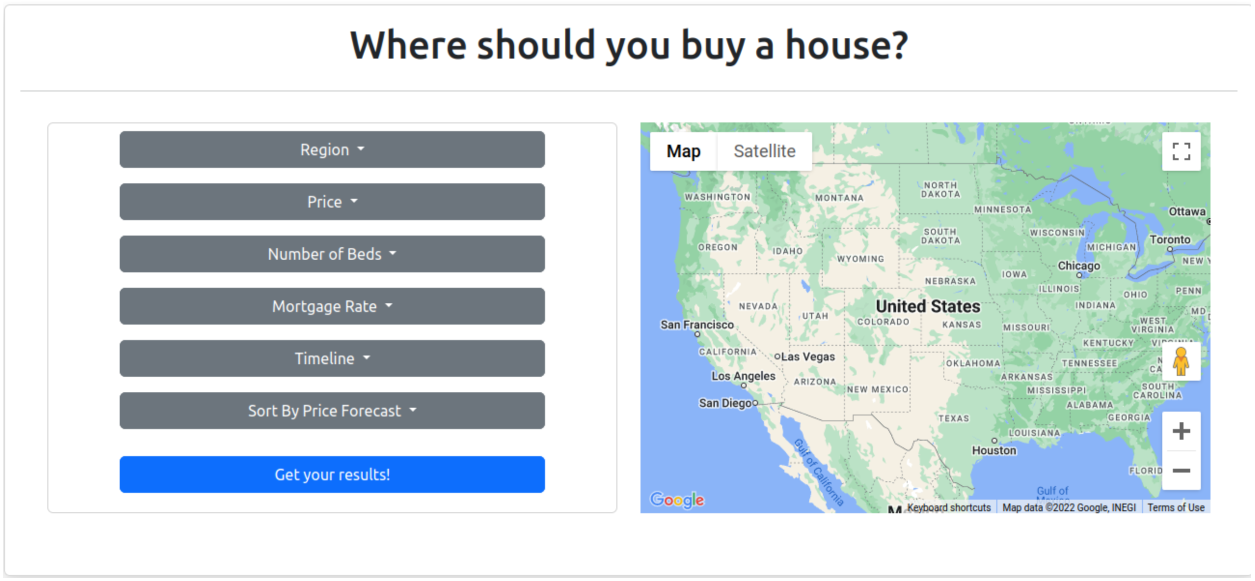
The goal of this project is to make a program that helps the user analyze housing affordability. The user should be able to provide data to the program which can be used to determine if the user can afford housing in the location given as input. As a secondary goal, we also wanted to give price change predictions in the future.

Project Design

The project was designed with user accessibility in mind. We hosted a website on the virtual machine provided by Professor Moskalev, and designed a frontend with visual elements so that it is easy for any user to navigate and give the necessary input. The frontend was created using Flask and Apache2.

Website Features

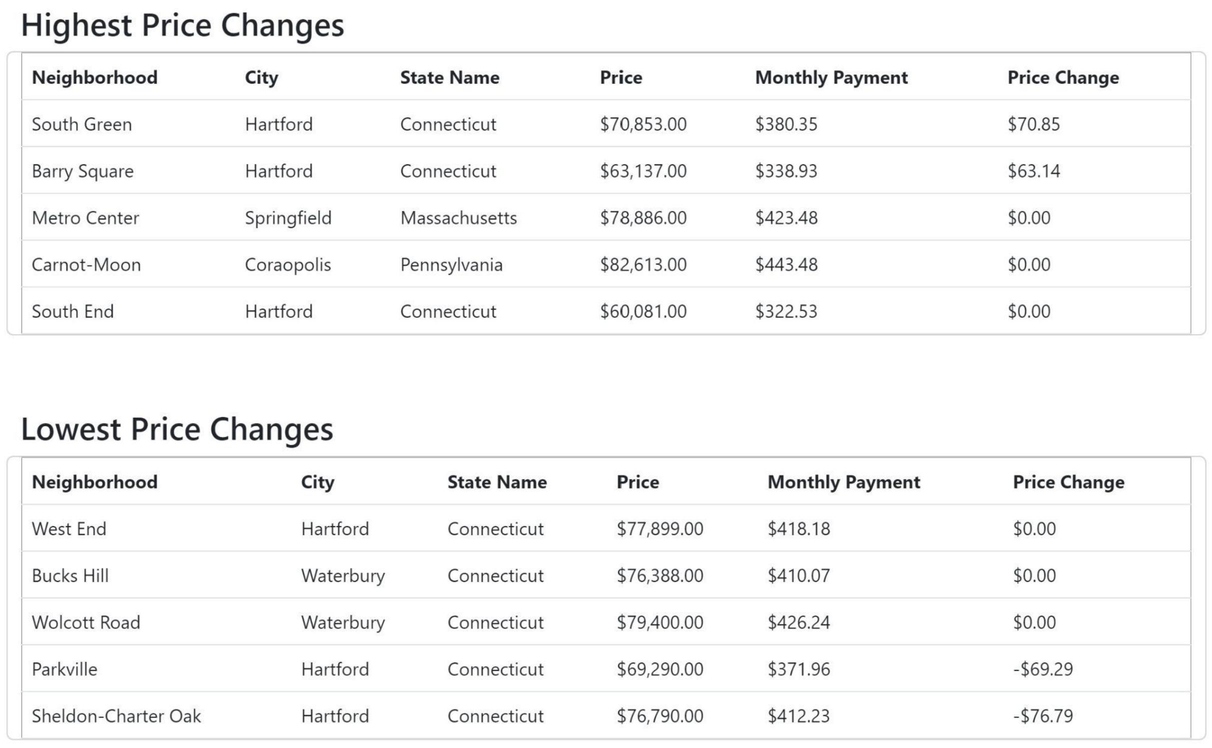
Below is a picture of our website UI:



As you can see, we take a multitude of user inputs:

- Preferred Region - Takes the geographic region the user wants to view prices for, based on what area of the country they would want to live in
- Price Range - Takes the budget range that would be acceptable for the user
- Mortgage Timeline and Interest Rate - Takes the information necessary to calculate monthly payments for housing purchase
- Number of bedrooms - Takes the number of bedrooms the user needs in their house
- Sort by Price Forecast - Asks if you want to sort by forecasted price or not

We then give the results to the user in a table. Sometimes, if there aren't enough options, there will be fewer than 5 suggestions, sometimes even 1 (example attached below). Depending on user input, this displays either price information or price forecasting information:

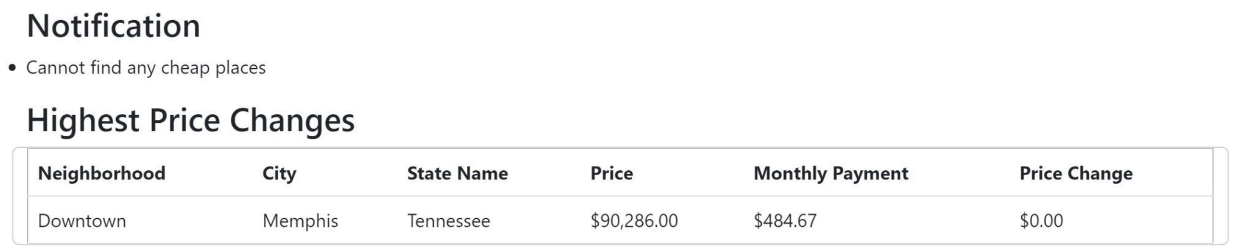


Highest Price Changes

Neighborhood	City	State Name	Price	Monthly Payment	Price Change
South Green	Hartford	Connecticut	$70,853.00	$380.35	$70.85
Barry Square	Hartford	Connecticut	$63,137.00	$338.93	$63.14
Metro Center	Springfield	Massachusetts	$78,886.00	$423.48	$0.00
Carnot-Moon	Coraopolis	Pennsylvania	$82,613.00	$443.48	$0.00
South End	Hartford	Connecticut	$60,081.00	$322.53	$0.00

Lowest Price Changes

Neighborhood	City	State Name	Price	Monthly Payment	Price Change
West End	Hartford	Connecticut	$77,899.00	$418.18	$0.00
Bucks Hill	Waterbury	Connecticut	$76,388.00	$410.07	$0.00
Wolcott Road	Waterbury	Connecticut	$79,400.00	$426.24	$0.00
Parkville	Hartford	Connecticut	$69,290.00	$371.96	-$69.29
Sheldon-Charter Oak	Hartford	Connecticut	$76,790.00	$412.23	-$76.79

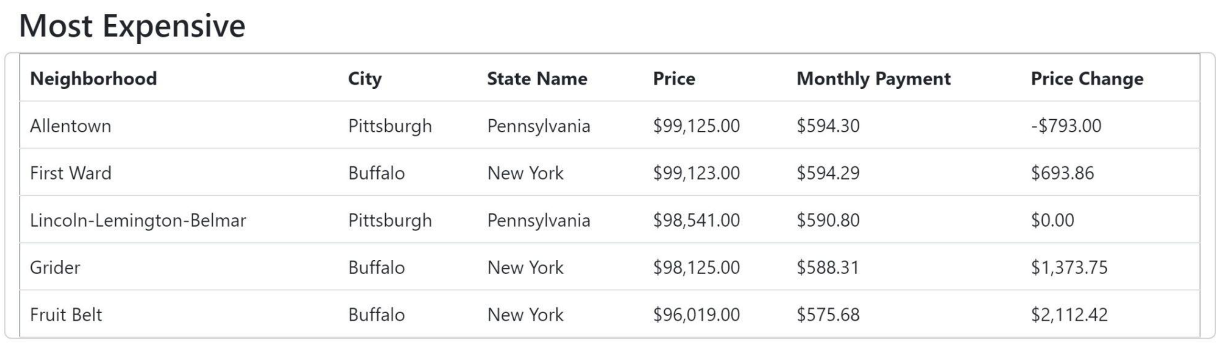


Notification

- Cannot find any cheap places

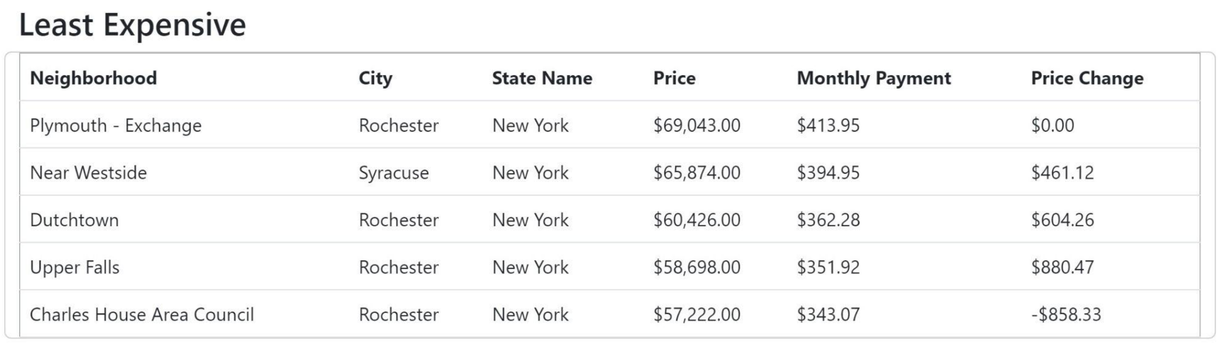
Highest Price Changes

Neighborhood	City	State Name	Price	Monthly Payment	Price Change
Downtown	Memphis	Tennessee	$90,286.00	$484.67	$0.00



Most Expensive

Neighborhood	City	State Name	Price	Monthly Payment	Price Change
Allentown	Pittsburgh	Pennsylvania	$99,125.00	$594.30	-$793.00
First Ward	Buffalo	New York	$99,123.00	$594.29	$693.86
Lincoln-Lemington-Belmar	Pittsburgh	Pennsylvania	$98,541.00	$590.80	$0.00
Grider	Buffalo	New York	$98,125.00	$588.31	$1,373.75
Fruit Belt	Buffalo	New York	$96,019.00	$575.68	$2,112.42



Least Expensive

Neighborhood	City	State Name	Price	Monthly Payment	Price Change
Plymouth - Exchange	Rochester	New York	$69,043.00	$413.95	$0.00
Near Westside	Syracuse	New York	$65,874.00	$394.95	$461.12
Dutchtown	Rochester	New York	$60,426.00	$362.28	$604.26
Upper Falls	Rochester	New York	$58,698.00	$351.92	$880.47
Charles House Area Council	Rochester	New York	$57,222.00	$343.07	-$858.33

Data Sources

We sourced our data from Zillow's housing data[1], which gave us housing information for the entire United States. We used this data to analyze prices in the area that the user gives as input. Since we could not use Zillow's API due to its high price, we downloaded CSV files from the website that we used. We downloaded 5 csv files: the 5 different Zillow Home Value Index files that had a specific number of bedrooms (so the 1 bedroom ZHVI csv, the 2 bedroom ZHVI csv, etc). These files are the first category of CSV files on the website, the home value category. We also sourced pricing forecast data from Zillow, from the same website. Finally, we took a csv from here which maps states to their regions in the US.

Data Ingestion and Transformation

First we took the zillow data on neighborhood pricing and merged all of it together. Then we ran positionstack api queries to cache the latitudes and longitudes of each of the neighborhoods in this file. Since positionstack did not give us zipcodes with these original queries, we had to run additional queries to get zipcodes from the longitudes and latitudes. Finally, we merged this data with the zillow price forecast data, which was given to use by zipcodes in order to get forecasted price changes for one year in the future. Our application allows the user to run queries to sort through our data and pick out neighborhoods that adhere to their search criteria. Additionally, we have written the code such that a person can simply change the configuration files and add additional datasets, as well as additional query parameters so that they can search for more personalized results.

[1] https://www.zillow.com/research/data/

Files, functions, and descriptions:

This is an overview of each file and what its purpose is.

- data_app.py
    - Routes:
        - "/": This is our user landing page, the function simply renders templates/form_with_map.html, sending the google maps api key in on render.
        - "/api": This route is used to process user requests. Once a user enters their info, a request is sent here. We take this input and feed it to the make_query function from data_app.py and return the resulting dataframe as a json object.
    - Runs Flask application
- search.py
    - In this file, we write 2 classes in order to manage our data and features selections
        - Feature(name, data, error)
            - Name: the data name when it show up on the website
            - Data: the available value we have in the dataset of that features (Ex: number of beds from 1 to 5)
            - Error: the error that will show to user when there are conflict related to the data (Ex: cannot find the required number of beds)
            - Function get_name():
                - Return the name of the Feature
            - Function get_data():
                - Return the data of the Feature
            - Function get_error()
                - Return the error of the Feature
        - Limit(**kwargs)
            - This class takes as many parameters as needed as it will get all the Features that our website needs and store them in a dictionary. Currently, our website use 6 Features (Regions, Price, Number of beds, Mortgage, Timeline, and Forecast)
            - Function check_param(param)
                - Check if a request for our website has a valid data list or not as well as check if the request has enough parameters to make a search or not.
                - Return the argument if it is a valid request, if not, return a list of errors
            - Function param()
                - Return a dictionary in which each pair of key and value is:

                - Key: same key as the Limit's dictionary
                - Value: a pair of tuple with name of the features and the list of feature's data

- data_work.py
    - In this file, we do all of our data processing and querying.
    - Functions:
        - format_money(amount)
            - Takes in a float amount, and formats it as a string with a dollar sign, only two decimal places, and commas every three characters.
        - get_unique_neighborhoods(dataset)
            - dataset should be a pandas dataframe of our raw merged zillow files.
            - This function will find all of the unique neighborhoods, along with their corresponding cities and states, saving this new dataframe to data/other/unique_neighborhoods.csv
        - add_latlon()
            - Here, we read in data/other/unique_neighborhoods.csv and use the positionstack api to get latitudes and longitutdes for each of the unique neighborhoods. Then we save the new data to data/other/unique_neighborhoods_with_latlon.csv
            - We need these longitudes and latitudes for the google maps api to drop markers on the map in form_with_map.html
        - add_zipcodes()
            - Now we use the positionstack api once again to get zipcodes. We read in data/other/unique_neighborhoods_with_latlon.csv and get zipcodes for each longitude and latitude with the api, saving this new data to data/other/unique_neighborhoods_w_zip_latlon.csv
        - create_zipcode_mapping(dataset)
            - Dataset is a pandas dataframe formed from the merged zillow csv files
            - This function calls get_unique_neighborhoods, add_latlon, and add_zipcodes in that order to generate data/other/unique_neighborhoods_w_zip_latlon.csv if it does not exist yet.
        - get_latest_date(dates)
            - Gets the latest date in a list of dates, returning a string containing the latest date.
        - merge_zillow_data()

        - Here we load in the zillow files which provide pricing for houses with certain numbers of bedrooms for each neighborhood and output all five files combined into one dataframe
    - generate_dataset()
        - This function takes in all of our data files and merges then into the one that we actually use to serve data to the frontend. We start calling merge_zillow_data() and joining the result with our region mapping file (for the regions shown on the frontend). Then we ensure that the zipcode mapping exists, calling create_zipcode_mapping if it does not. We then read in additional datasets from the config_data.yml file, merging these into the final dataset as well.
    - filter_data(data, args)
        - Data is our dataset from data/project_dataset.csv and args are the arguments inputted by the user on the frontend.
        - This function filters our data to remove neighborhoods that do not fulfill the user's requests, returning a dataframe without those entries.
        - It uses the config_param.yml file to filter these elements out based on the current configuration of the filters.
    - format_output(filtered_df)
        - Filtered_df is a dataframe with the results of the user's query in proper order according to the user's sorting inputs.
        - This function spits out two html tables containing the top of and bottom 5 results along with the dictionaries that represent these entires to be used for mapping.
    - make_query(args)
        - args is a dictionary of arguments read in from the user on the frontend.
        - Here, we first check that the project dataset exists, generating it if not. Then we filter the data according to the arguments using filter_data. Now if the user has requested that the results be sorted by forecasted price, we find the forecasted prices, returning a df with the 5 neighborhoods with the highest price change and the 5 neighborhoods with the lowest price changes. If the user did not request price forecasts, then we simply return a df with the 5 most expensive and the 5 least expensive neighborhoods which fit the given requirements.
- wsgi.py
    - This file runs the application to deploy with WSGI

- templates/form_with_map.html
    - This file is the template to serve our application to the user. It contains the html for the form, the google maps api call, and the tables that display the users' query results. We used Bootstrap for the styling and some personalized css/javascript for animations.
    - This file links to static/script.js and static/style.css styling and dynamic content
- static/script.js
    - In this file we first create the initMap() function that the google maps api requires to render our javascript map
    - We then ensure that the dropdown menu text changes when a user selects one of the options with event listeners.
    - Finally, we add an event listener to the submit button which triggers the animation sequence and uses fetch to send a post request to the api route along with the user input. Once we get the result, we fill in the HTML table and make it visible with another animation sequence.
- static/style.css
    - Here we just add some extra styling to the HTML and define the animations we will use to show and hide the tables.
- .env
    - Should contain three api keys:

```
POSITIONSTACK_KEY1='<positionstack_api_key1>'
POSITIONSTACK_KEY2='<positionstack_api_key2>'
GOOGLE_MAPS_KEY='<google_maps_api_key>'
```

- requirements.txt
    - Contains a a list of the python modules necessary to run our application
- config_param.yml
    - This file contains configurations for the parameters that we take in from the user. If someone wanted to add one of these, they would need to put in a name, a collection of allowed values with their corresponding backend values, a type of comparison to be used in the backend (actually filter the data), and the column that the comparison should be run on in the project dataset. By adding all of these things, a user can add a new filter to the app without writing any code.
- config_data.yml
    - This file holds configurations for some of the datasets that go into the project dataset along with some config of the final dataset itself.
    - To add another dataset, someone simply needs to provide the path to that dataset, instructions for merging it with the project dataset, and the columns to use in that dataset

    - The user can also modify the project dataset with this file by changing final_cols or adding a reformatting pair that will switch one of the column names to something else.