Live Better : Refine Your House Search

Adam Romayor, Varun Patwari, Annmary Sebastian, Tejas Mahajan
Department of Software Engineering, San Jose State University
San Jose, CA
adam.romayor@sjsu.edu, varun.patwari@sjsu.edu, annmary.sebastian@sjsu.edu
tejas.mahajan@sjsu.edu

Abstract — Tremendous amounts of data related to housing is generated on a daily basis. If analyzed effectively it enables us to answer the fundamental questions about the livability of a particular area. In order to achieve the same, the generated data must be extracted, cleaned, properly structured and then analyzed using various data models. This report focuses on the process of calculating the livability index of houses. This improved technique will mainly benefit home buyers for choosing better homes.

I. INTRODUCTION

Live Better is a tool that helps home buyers decide on which house to buy based on their preferences. The application provides options for the users to rate their preferences and a livability index for the house is generated based on those preferences.

II. ARCHITECTURE

There are two major components: Backend for database queries and user interface for the application.

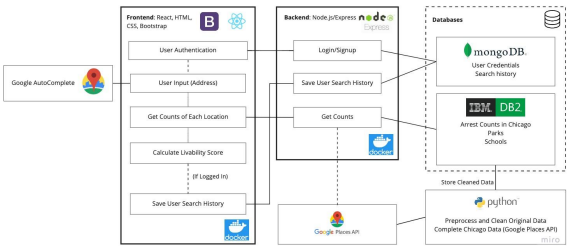

Figure 1. Project Architecture

The frontend of the project was written in React, HTML, Bootstrap and CSS. The backend was written in JavaScript using the Node.js and Express Framework. Two databases were used in this application. MongoDB was used for user authentication and IBM Db2 was used to store all the data used by the application. The frontend and backend were each in their own Docker Container, and could be built by using the docker-compose up command [6], [10].

III. DATA COLLECTION

The data was collected initially through research, where we used the Chicago Crime database to obtain the crime information regarding locations within Chicago [7]. School data was collected through Homeland Infrastructure Foundation-Level Data [1]. Park data was collected using the ParkServe dataset [2]. The school and park datasets are available for locations across the entire United States. Later, we used the Google Places API [9] and collected more information like, hospitals, restaurants, supermarkets, gyms, hike trails, bike trails, ATM and bus stations for all the location values we gathered from the Chicago database. All this information gathered was merged efficiently to create the final database.

For locations outside Chicago, we currently do not have the crime information. The rest of the information for such locations are queried from the Google Places API.

IV. DATA PROCESSING

The initial dataset that we began to work with is the Chicago Crime Database. We removed all columns that were not necessary for our analysis. The only features retained from the database were the latitude and longitude values, location information and the arrest count for a location. The missing information in the dataset was filled up using multiple imputation techniques. Then we added 10 more features to our dataset which includes restaurant, school, bus_station, atm, supermarket, park, gym, hospital, hike trail and bike trail. This information was fetched from the Google Places API using the latitude and longitude values. Later on we calculated the number of instances of individual features present in the location. Further analysis and visualisation on the dataset was performed using visualization libraries which includes matplotlib and seaborn. The below figure represents the distribution of features across the various locations.

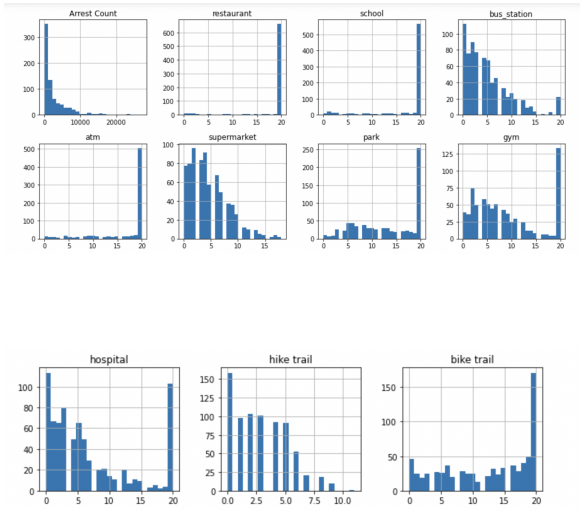

Figure 2. Features by location.

School data was three separate datasets, Private Schools, Public Schools, and College and University data. This was combined into a single table consisting of the school name and their latitude and longitude values. Park data had addresses, but did not include any latitude and longitude values. In order to complete the park database, the Nominatim OpenStreetMap API [3] was used to convert every address into the corresponding latitude and longitude values. Any address that did not return those values was dropped from the database. In the end, the park database totaled 119,036 rows, consisting of the park name, and it's respective latitude and longitude location. All three final databases, Chicago Data, School Data, and Park Data were stored in IBM Db2.

V. BACKEND

The backend of this application used the Node.js framework. Database calls were handled in the backend of this application.

The first step was to correctly set up the IBM Db2 database. Within the database, there were three separate tables: Chicago, Parks and Schools. The Chicago table contained arrest count information, along with counts of the nearby places as explained in Section III. The Parks and Schools tables each contained the name of the place, and their location in latitude and longitude coordinates. In Node.js the command npm install ibm_db installed the drivers necessary to connect to the database. Once the database was connected, the backend could query from each table for their corresponding information [4].

From the backend, two endpoints were exposed, to access the IBM Db2 database. The first endpoint is /chicago. This takes latitude and longitude values, and queries the Chicago table. This will find the row from the Chicago table that is closest to the input location, as long as that row is within 5 miles of the input. If it's greater than 5 miles, this query will turn up empty.

The second endpoint is /queries. This also takes latitude and longitude values, but this makes two queries. The first query accesses the Parks table and finds the number of parks within

two miles. The second query finds the number of schools within two miles, by accessing the Schools table. The queries utilize the Haversine Formula, to convert latitude and longitude values into distance in miles [5].

A third endpoint is used to complete the rest of the data. This endpoint is /google-search and is only accessed when a user is searching a location that is outside of Chicago. This query makes eight separate API calls to the Google Places API to find the number of restaurants, bus stations, atms, grocery stores, gyms, and hospitals within one mile. Additionally it finds the number of hike trails and bike trails within two miles. This is the same information stored in the Chicago table, so this completes the information for the rest of the country. API calls to Google were made using axios.get.

MongoDB is used to store the user's credentials at signup and when the user tries to login with an email and password. Those inputs are checked with the records in the database and the user is authenticated.

When a score for a particular place according to user preferences is generated, the /savescore API is called which stores the score in MongoDB. But before storing the data, the data is validated using the mongoose ODM (Object Data Modelling). Whenever a user wants to see previous scores those data are retrieved from MongoDB.

VI. CLIENT

In the first phase of the project, we created designs of the application using Figma to get a better idea about the frontend implementation.

The application includes a home page, search page, results page, history page, signup page, and a login page. The home-page has an option to get a trial of the application with limited features. When a user attempts to search a location, a pop-up appears where the user can set the importance of various aspects of a location and the livability result is calculated based on these preferences.

The result is displayed as a score out of 10 along with the details of the nearby places. The user can change the filters on the results page and can get a newly evaluated score based on the changes in real time. The history page saves the users search history along with the livability score which can be accessed at any point by the user.

A. Home Page

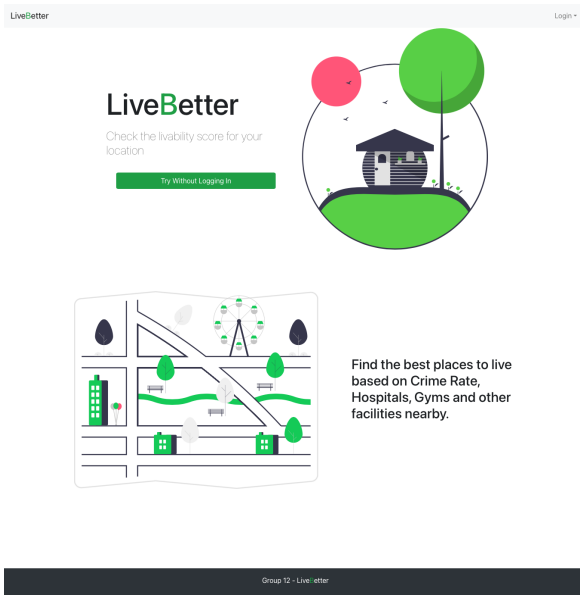

Figure 3. Landing page of the application.

The top right corner has the option to login or sign up. A new user can also try the product without logging in, for limited access.

B. Search Page

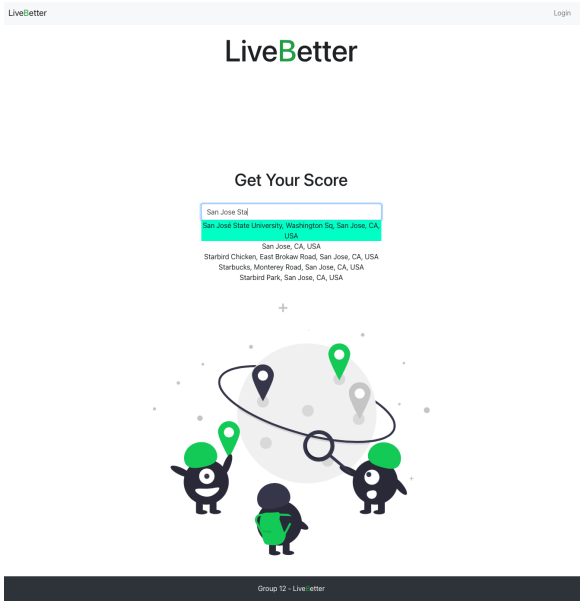

Figure 4. Search page of the application.

A user can type in the address they want to find the livability index for. The Google Autocomplete API is used to assist the user when they type an address. This allows the user to search any address, and acts as an input validation. The API provides several details, such as the latitude and longitude values that are passed to the backend to query the databases.

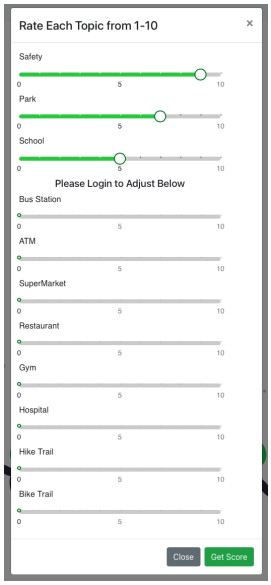

Figure 5. Trial user input preferences.

Figure 5 shows a user that is trying the application without logging in. Once an address is inputted, the user can assign a preference for each topic shown. Since this is only the trial version, the user can only adjust the sliders for safety, park, and school.

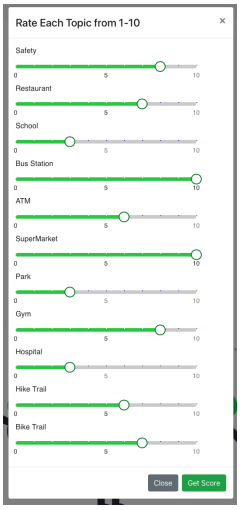

Figure 6. Logged in user input preferences.

Figure 6 shows a logged in user's input preferences. They have access to all features and can individually assign a rating to each topic. If a user leaves all values as zero, each feature is treated with equal weight. The sliders are implemented using the Material UI slider component. In Figure 5, the sliders that can't be changed are set to a disabled state. Once a user clicks "Get Score" they are redirected to a loading page, where the backend calls are made to get the data necessary for the score calculation.

C. Loading Page

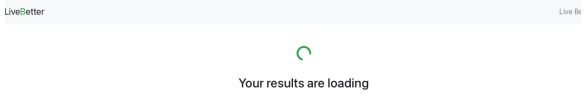

Figure 7. Loading Page while the application queries for all data.

The loading page is shown while the frontend accesses the backend functions. The first query accesses the /queries endpoint. This queries the Schools table and Parks table in the IBM Db2 database. This retrieves the counts for schools and parks within two miles. The second backend call queries the Chicago table in the IBM Db2 database. This retrieves the arrest counts, and remaining features available. If this query comes up empty, the third api is called, /google-search. This uses the Google Places API to retrieve the final information, since it is not available in the Chicago table. A trial user only accesses the Chicago, Schools, and Parks tables, so no Google Places API calls will be made.

A separate loading page is created, so the application only retrieves this data once. Refreshing the result page won't force the application to make additional calls to the database or the Google Places API.

D. Result Page

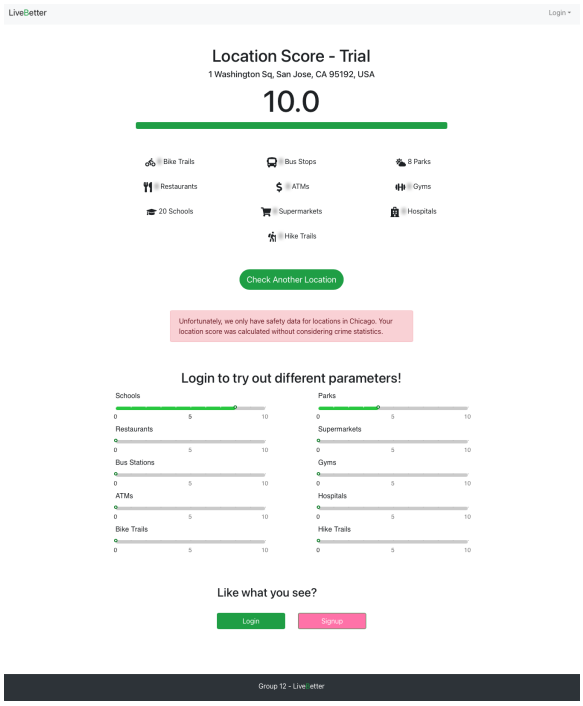

Figure 8. Trial user result page.

Figure 8 displays the results for a trial user. The score is 10 out of 10 and shows the counts for Schools and Parks within two miles. There is an alert in red that explains safety data is unavailable for locations that are not in Chicago. This check is done through Frontend code, and appears when the query to the Chicago table comes up empty. The bottom of the page shows the user preferences, so the user knows what weights they input. However, since this is a trial, the user is not allowed to change the weights once the score is calculated. The counts of other locations like bus stops, hospitals, etc. are all blurred out. On this page, the user can directly login or sign up for an account.

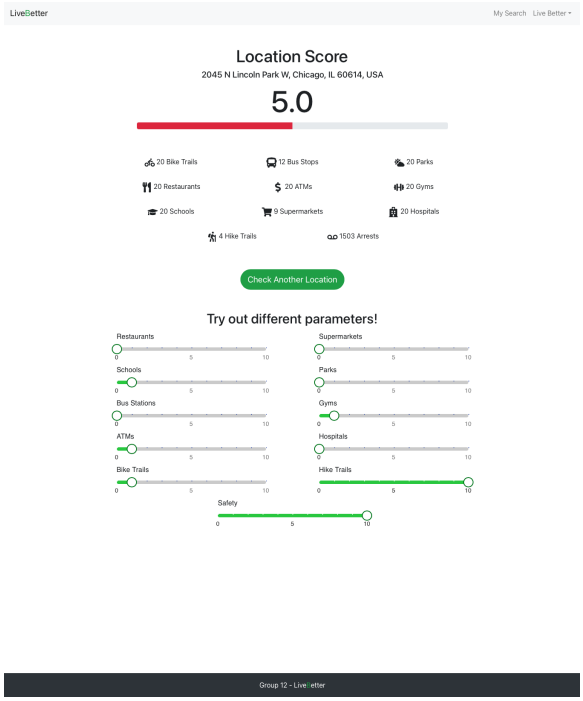

Figure 9. Logged in user result page.

Figure 9 displays the results for a logged in user. This score is 5 out of 10, and the progress bar changes color to red, to indicate this is a low score based on the user preferences. A score between 5 and 7.5 would turn the progress bar yellow. Anything above 7.5 is shown in green to indicate a high livability index. This page also allows the user to change their preferences to recalculate the livability score instantly.

The counts of each feature are also shown on the top of the page. The maximum count for any feature is 20, because Google Places API only returns a maximum of 20 locations. The score is able to be recalculated in real time, because when a user changes the slider, a function resets the state of the variable for that given slider. On a state change, the component that displays the score is rerendered with the new score. Every time a score is calculated or recalculated, the history is saved, and the user can view their search history by clicking on "My Search" in the top right corner. The search history is stored in the MongoDB database instead of IBM Db2, since user information is already stored in MongoDB.

One final feature to point out is that the Arrests is only shown when the address is in Chicago. The safety slider is also shown at the bottom, only for Chicago, since we were unable to find crime statistics for other cities.

E. History Page

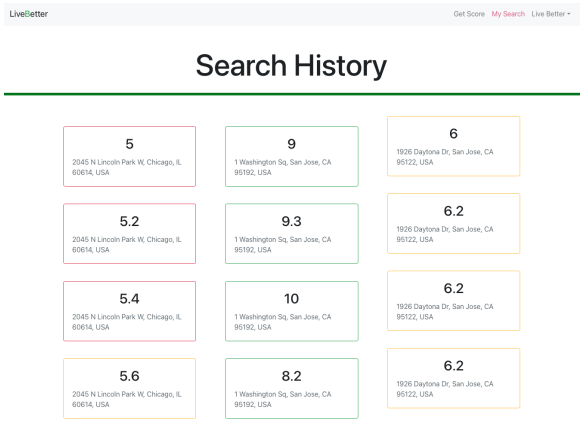

Figure 10. Search History from logged in user.

The history page is only accessible for a logged in user. Each card utilizes the React Card Component [8]. Once this page is loaded, a call to the backend accesses the MongoDB database, and lists all the search results for that particular user. The border color of each card matches the color of the progress bar from Figure 9. The card component consists of the livability index and the address that was searched. There can be multiple scores for each address, because a new score is saved every time a user changes their preferences. A future improvement to this page would also show the user preferences for each card.

VII. SCORE CALCULATION

The livability score for houses is calculated using the weighted average method. For every feature, first a weight is assigned based on its impact on the livability of a place. The number of instances of a feature in a location is then multiplied with this weight to generate the feature value. The livability index is a cumulative value of all the 11 feature values for a particular location. The livability score for all houses in the database was calculated in the similar fashion. Upon completion of the same, the data was visualized to view the number of houses inside each score level.

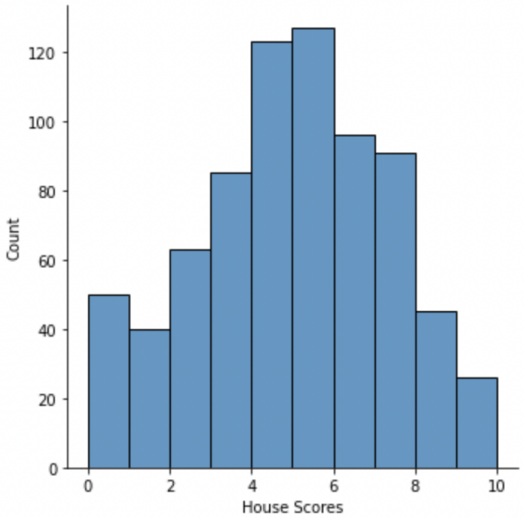

Figure 11. Housing Score Distribution

Initially, our plan was to create a Machine Learning Model to calculate the livability score. However, we decided to use a weighted average approach, since the user is assigning their own weights. To our team, it

didn't make sense to train a model, when all users have the potential of having different preferences.

The weighted average method takes into consideration the user preferences. Each weight will contribute a certain percent to the final score. For example if a user set Safety to 10, Schools to 5 and Parks to 5, the safety weight would total 50% of the final score. Schools and parks would each contribute 25% to the final score. Additionally, each feature has a maximum value. For example, the schools feature has a maximum value of 20. This means if a location has 20 schools nearby, the final score would have the full 25% contributed by schools. If a location only had 10 schools nearby, the location schools feature would only contribute a total of 12.5% to the final score, instead of the full 25%. This method is done for every feature except arrest counts.

Arrest counts are different, because higher arrest counts should have a lower livability score. This is handled by setting the maximum arrest counts to 100. Any location with 100 or more arrests will contribute 0%. This is calculated using the equation (1 - x/100). In this case, x is the total number of arrest counts. If there are 0 arrests, this value would be 1. This means the address should have the full 50% of the points, referring back to the example described before. If a location had 10 arrests, the value would be 0.9. Which means the final score would have 50% x 0.9 = 45% of the total score. In other words, this address would get 45% of the 50% available for the safety weight.

VIII. CONCLUSION

Live Better helps home buyers to choose the best place to live based on their preferences. This helps them make an informed choice while buying homes in a new location. The slider bar provided to set user preferences give the users flexibility to get the livability score based on their living habits.

IX. FURTHER IMPROVEMENTS

A. Additional Features

- The user can compare their previous location scores and related information.
- Based on the final selected location the user is provided with a list of properties with similar or higher scores near that particular location.
- Allow the user to search for more locations using the same preferences they initially set.

B. Optimizing Google API usage

We can optimize the Google Places API usage by storing information about a particular location in the database and scaling the database accordingly. Thus next time when a user searches for the same location, that information can be retrieved from the database instead of using the Places API.

X. RESOURCES

A. Github Repository
https://github.com/SJSUSpring21/LiveBetter

B. Project Link
https://live-better-272-demo.herokuapp.com

- Demo Email: livebetter@gmail.com
- Demo Password: livebetter

REFERENCES

[1] Homeland Infrastructure Foundation-Level Data, "Public Schools, Private Schools, Colleges and Universities," *Homeland Infrastructure Foundation-Level Data*. [Online]. Available: https://hifld-geoplatform.opendata.arcgis.com. [Accessed March, 2021].

[2] ParkServe, “Complete U.S. ParkServe Dataset,” *ParkServe*. [Online]. Available: https://hifld-geoplatform.opendata.arcgis.com. [Accessed March, 2021].

[3] Nominatim, “OpenStreetMap API,” *Nominatim*. [Online]. Available: https://nominatim.openstreetmap.org/ui/search.html. [Accessed March, 2021]

[4] D. VerWeire, L.Smith, B. Bigras, C. Ensel, Yorick, J. Kainz, O. Efimov, P. Hendrix, and IBM, “Node-ibm_db”, *GitHub Repository*. [Online]. Available: https://github.com/ibmdb/node-ibm_db. [Accessed April, 2021].

[5] Ollie, “Fast nearest-location finder for SQL (MySQL, PostgreSQL, SQL Server),” *plumislandmedia.net*, Oct. 31, 2014 [Online]. Available: https://www.plumislandmedia.net/mysql/haversine-mysql-nearest-loc/. [Accessed April, 2021]

[6] NodeJs, “Dockerizing a Node.js web app,” *NodeJs*. [Online]. Available: https://nodejs.org/en/docs/guides/nodejs-docker-webapp/. [Accessed April, 2021].

[7] Chicago Data Portal, “Crimes - 2001 to Present,” *Chicago Data Portal*. [Online]. Available: https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2/data. [Accessed March, 2021].

[8] React Bootstrap, “Cards,” *React Bootstrap.* [Online]. Available: https://react-bootstrap.github.io/components/cards/. [Accessed: May, 2021].

[9] F. Morales, “Google Maps: Places API,” *Medium.com*, May 18, 2020. [Online]. Available: https://medium.com/swlh/google-maps-places-api-28b8fdf28082. [Accessed: March, 2021].

[10] M. Herman, “Dockerizing a React App,” April 7, 2020. [Online]. Available: https://mherman.org/blog/dockerizing-a-react-app/. [Accessed: April, 2021].