Chicago Service Requests Dashboard

Dominick Yacono

Summary

The following report outlines the steps taken to develop a predictive model for 311 service requests across the Chicago metropolitan area. Chicago holds 77 individual community areas, each with its own demographic and economic characteristics. From the shores of Hyde Park to the international transportation hub centered in O'Hare, Chicago's numerous communities have various, unique service needs. Community attributes and features were joined together to build a powerful XGBoost predictive machine learning algorithm.

This model aims to provide Chicago service workers the following:

1. A daily, consistent overview of predicted service requests for each of Chicago's 77 community areas.
2. A historical look at the past 2 days to contrast daily predictions with previous request numbers, and view the top types of service requests.
3. A list of the top 5 communities predicted to have the largest increase in service requests

Preparation

Background On The Project And My Interest

My desire to work on this personal project stemmed from an interest in using widely available data from the city of Chicago. Chicago's data is renowned for its comprehensive detail and transparency, providing important insights into crime, safety, and public health. The city hosts a data portal webpage, with links that connect to city departments and allow ease of access to data sources. Data can also be easily downloaded in a variety of formats, like CSV, JSON, and

Shapefiles. After working on my Citibike Tableau dashboard for New York City, I was attracted to move westward towards Chicago for my next project.

The 311 phone number is a special use access line allowing citizens of select cities like Los Angeles, Orlando, New Orleans, and Chicago to get in contact with city services for non-emergency matters. Use of the number has led cities to reduce costs and increase quick access to services. Since introducing the 311 system in 1999, Chicago has won national awards for its implementation , including the Innovations in American Government Award at Harvard University's School of Government in 2003.

As a vital instrument for the public good, the 311 phone number drew my interest with the immediate impact it provides for those who call it. Thanks to Chicago's Data Portal, I easily found the city's 311 service request database and began my work.

i. Overview of 311 Service Data and Downloading

The 311 Service Request Dataset on the Chicago Data Portal is a realtime feed of service requests. Each row pertains to a unique request, indicated by a service request number (SR Number). The dataset provides information on the type of request (aircraft noise complaint, sanitation code violation, etc.), the creation date of the request, the street address of the requester, whether the request has been completed, and more.

To obtain this live data, I used the dataset's given API endpoint, and funneled it into my code pipeline. The API has a limit of 1000 rows per requests, so I programmed by code to run through multiple requests.

The Pipeline and Predictive Model

Data Processing

Before fitting the data onto a predictive model, I worked through a variety of processing steps to clean and organize the information. First, I dropped service requests where both the requester's street address and community area are not given. These observations are sadly impossible to model, since they are not geolocated. If requests did have an origin address but no community area assigned, I used the Nominatim geocoding service to assign appropriate community area numbers to these addresses.

i. Feature Engineering and Adding Temporal Features

With each service request belonging to one of numerous service types, I simplified the categorization of these requests using a predefined dictionary. This would help me later when including the top 3 service types for each community in my visual dashboard.

To obtain more data on the demographics and characteristics of each community area, I downloaded American Community Survey data from the Chicago Data Portal. This data provides breakdowns on age, race, and income level for Chicago’s communities.

To add temporality to the prediction model, I engineered time-based features to enhance the data. These temporal features can capture the effect of devastating snow storms, or federal holidays where people are off work. I created features for the year, the day of the week, and the season. I added a binary feature for weekday or weekend, and a boolean feature for whether the current day is a federal holiday.

ii. Aggregating the Data by Community and Adding More Features

At this stage, I grouped the data by community area, obtaining the total service request counts for each area.

I also continued to add important temporal features that will help the predictive model take into account each community's past request counts. First, I created 'count_lag' features going back 6 days for each community. I also engineered rolling window features that are based on 7-day windows. For example, 'count_rolling_avg_7' provides a 7-day moving average of request counts for each community, while 'count_rolling_std_7' provides a 7-day moving std. deviation of request counts.

By creating new lag and rolling window features, 'NaN' values were introduced at the beginning for each community area's time series. I removed these rows to ensure model completeness.

iii. Final Preparation for XGBoost

For a tree-based model such as XGBoost, data is split and predicted based on a feature's values. For categorical features, values are encoded as numerical values (e.g., winter = 1, spring = 2, summer = 3, fall = 4). An issue is that this can lead a model to believe in ordinal relationships within a feature (i.e., summer and fall are worth "more" compared to winter and spring). To fix this, I employed one-hot encoding on categorical features, creating simple binary columns for each value within categories. Now, seasons like winter, spring, summer, and fall have their own columns.

I standardized all numerical features, giving each column a mean of 0 and standard deviation of 1. This helps the prediction model converge to its optimal predictions more effectively, instead of searching across a wide range of various feature data ranges.

Lastly, one of the most important steps I took was log- transforming the target variable. 311 service requests typically have most days with low-to-moderate number of requests. Once in a while, there are major storms or disturbances which can cause a community to have high requests. This phenomenon creates long-tailed data distributions for requests.

For machine learning predictive models, like XGBoost, extreme target values can have a disproportionate influence on the model, pulling predictions toward them. By performing a log transformation, the long tail at the end of the distribution is squashed and squeezed. This helps make the overall target distribution more symmetric, as variance between observations is stabilized.

Training the Model

With my swath of historical Chicago service request training data, I now aimed to split the data for cross-fold validation. Cross-fold validation is a technique used to test how well a machine learning model is performing on unseen data. The data is shuffled and first split into groups, better known as folds. If there are K numbers of folds, the model is trained using K-1 number of these folds, and tested on the last fold of data. This process repeats so that every fold to be tested, while performance is measured for each fold. Eventually, the performance scores are averaged into a single, robust metric for the model.

To split the time-series data for cross-fold validation, I use the '`TimeSeriesSplit`' technique found within the Sci-kit learn Python package. This technique ensures that during data shuffling and fold creation, the validation data always comes chronologically after the training data.

i. **Applying XGBoost**

The XGBoost model creates an endless ensemble of decision trees in a sequential process, evaluating the training data on its tree to generate predictions and adjust itself.

The first decision tree makes very simple predictions for each community area by taking the average of the target variable across all communities. Then, the residual errors are calculated by subtracting the predicted values of each community's service requests from their actual values.

These errors become the new target variable for the next decision tree to predict. A second tree is created to predict the residual errors from the first tree by using all the features in the service requests dataset. Once this tree is created, a new prediction is calculated by `'New Prediction' = 'Old Prediction' + ('learning_rate' × 'error_prediction_from_new_tree')`. This small update to the prediction is governed by the learning rate, which is a number typically between 0.01 and 0.3.

After the new prediction is made, the entire process described above repeats. The process will aim to predict smaller errors that still remain in the model. A new tree is created that once again attempts to predict the residual errors based on the current predictions, and another tree is made to determine the individual feature sets that best explain the new errors.

After iteratively lowering its errors and creating numerous sets of decision trees, the XGBoost model ends once a predefined maximum number of trees is met. My model uses '`GridSearchCV`' to test out XGBoost with varying numbers of maximum trees, and chooses the model that delivers the best performance.

ii. Measuring Performance

'`GridSearchCV`' searches across many different models, with varying hyperparameters, and measures these against one another using their negative mean square error (NMSE). NMSE measures the average of the squares of the errors, and GridSearchCV aims to maximize the NMSE (bring it close to 0 as much as possible).

After the best model is selected, my pipeline evaluates it using the root mean square error (RMSE). The absolute value of the NMSE from the best model is calculated and the square root is taken, resulting in the root mean square error (RMSE). The RMSE that the model initially predicts is log-transformed, since the target variable is log-transformed. The final step of the pipeline is to convert back onto the data's original scale to make it interpretable.

Since my XGBoost model runs every day, it is constantly training on data that is changing on a rolling window. Therefore, there is a new RMSE that the model arrives at every day. Through building the model and testing over various days in early August, 2025, my model arrived at RMSEs on the data's original scale between about 1-9. Interpreting this, we can say that the model's margin of error for community-level service request predictions is between 1 and 9 requests.

Data Visualization

Building the Visualization

To build the visualization to show the predictions for each community area, I utilized libraries like ‘`geopandas`’ for geographic data wrangling and ‘`bokeh`’ for plotting. I loaded three data files into my visualization script:

- ‘`Boundaries - Community Areas_20250802.geojson `’
- ‘`Test_latest_counts_graphed_with_predictions.csv`’
- ‘`top_services_list.csv`’.

i. Data Processing

First, I merged the three data sources into one file, ‘`GeoDataFrame`’. For each community area, predictions are merged alongside the top service request types. To highlight the top five communities that have predicted increases in requests, I create a new column ‘`increase`’, by subtracting the previous request count from the day before from the predicted.

ii. Building the UI

Building the dashboard, I created text elements like the main title and an overview paragraph. I drew the community areas in a map as colored patches, with higher values of service requests depicted by a bright yellow and lower values of requests depicted by a dark purple.

Using the ‘`HoverTool`’ feature from the ‘`bokeh`’ package, I created the ability to use the mouse to hover over an area and bring up a tooltip. This tooltip shows the community name, predicted requests, and change from the previous day.

After crafting the map, I created a panel to host a list of community hotspots for increases in service requests. I also established a panel to host line graphs of service requests for each community area when selected.

iii. Adding Interactivity

Through using JavaScript, I enabled interactivity and dynamism into my web dashboard. By writing JavaScript instructions called callback code, visualizations are able to adjust whenever the user clicks on communities on the map.

First, the data in the trend graph is replaced with the data only for the selected community, and the title of the graph updates too. The top service types are also updated.

If the user clicks on an empty part of the map, the callback to JavaScript resets the visuals to default views.

iv. Final Arrangement

Through arranging the components of the webpage, I created an organized layout. I used the `'row'` and `'column'` functions of `'bokeh'`.

The Visualization

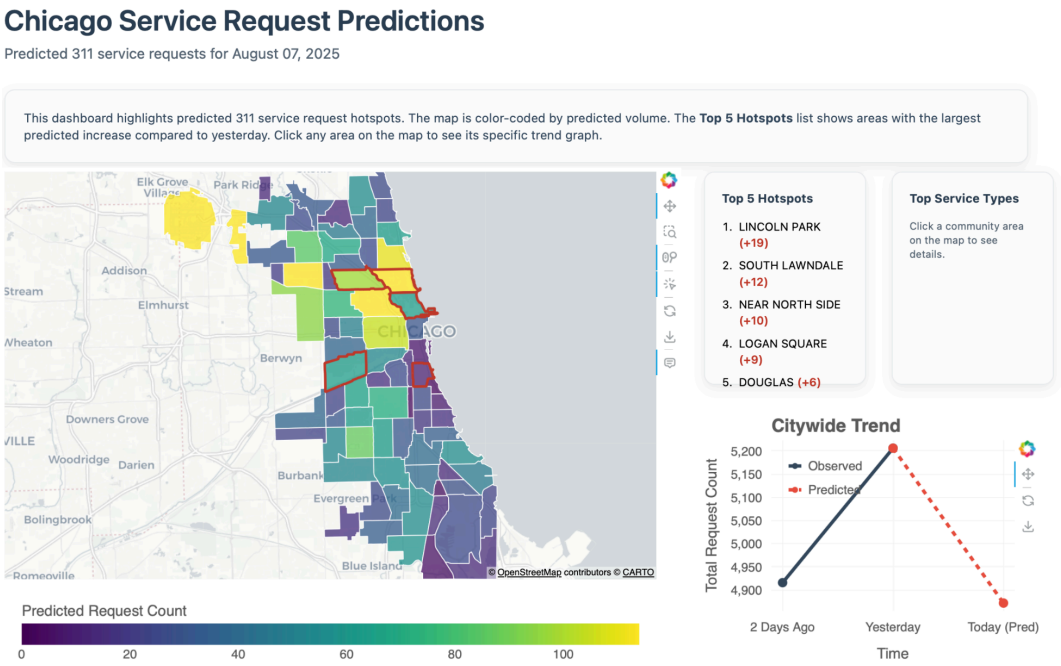

Figure 1. A choropleth map of Chicago displaying the predicted number of 311 service requests for August 7, 2025.

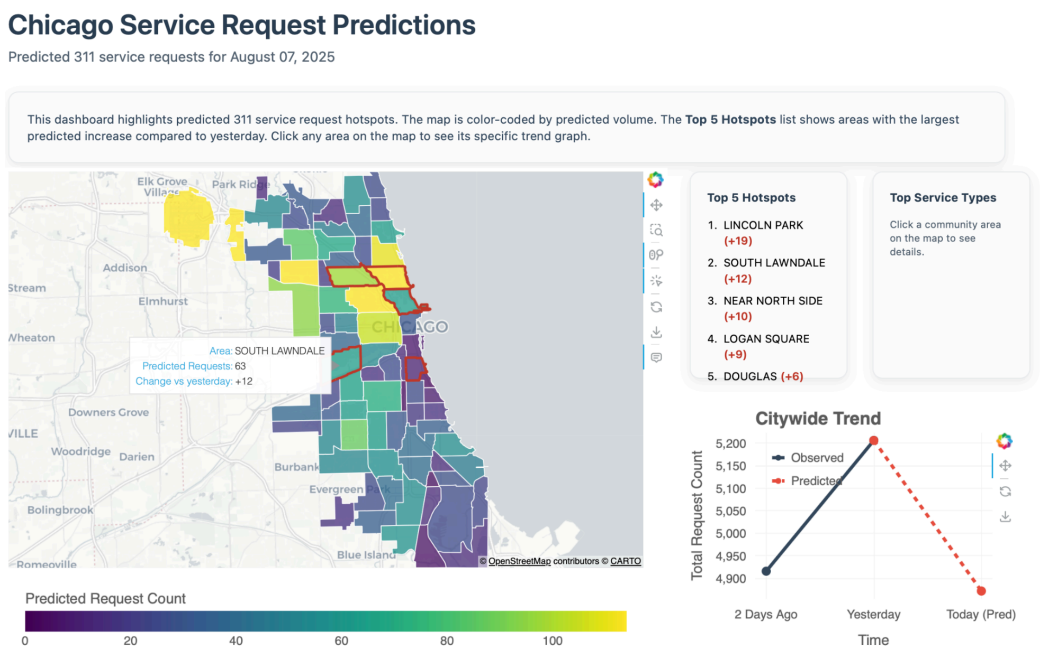

Figure 2. Hovering over a community, like South Lawndale, displays a tooltip.

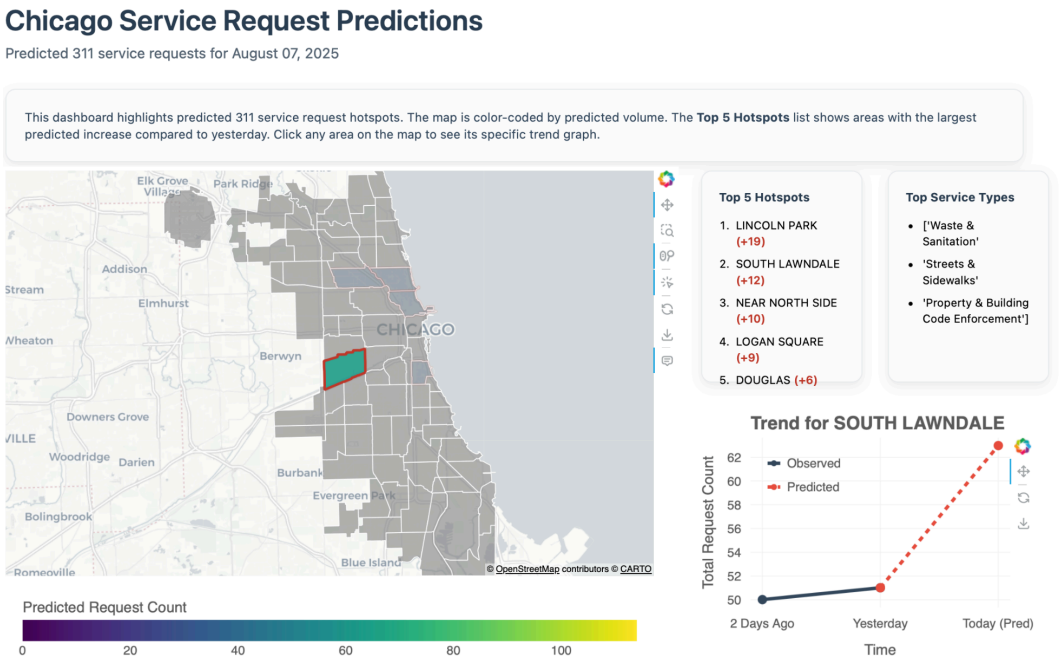

Figure 3. Selecting a specific community, like South Lawndale, filters down the trend line graph and the top service types to that community.

The final visualization is shown in Figure 1, as it appeared on August 7, 2025. On the left, a map of Chicago's community areas are shown. The user can easily hover over (Figure 2)

and select a community area (Figure 3) to filter the visualization and find more specific information.

Conclusion & Final Thoughts

The visualization achieves the project's goals by giving end-users a complete view of Chicago's communities and giving them accessible tools to understand service requests patterns across the metropolis. Users can easily view the top hotspots for predicted service request growth, as they are outlined in bright red on the map. Top service request types are displayed on the far right, giving users more insight into differences between communities. Lastly, the trend line graph provides a small, but important, historical look into the past 2 days of observations and how the predicted request counts measure up.

It's important to think about possible future avenues to take this project. Expansion on this project would provide an opportunity to generate greater insights into Chicago's various communities and their individual service needs.

A simple addition to the webpage would be an RSS feed of live news. This general news feed

Generative AI summaries that could conduct speedy Internet searches would be a great first step. By summarizing news headlines and presenting them to end-users, the model could explain spikes in service requests.

Another area of improvements would be adding more explainability to the predictions. The XGBoost model works in a closed-off, backend environment and does not show any of its predicting rationale to the end-user. Adding a list of most important community features could effectively achieve this goal.