THE UNIVERSITY OF
CHICAGO

Bus pending

Final project for CAPP 30122

Michael Rosenbaum (mrosenbaum)   Regina Isabel Medina Rosales (rmedina)
Daniel Muñoz (dmunozbatista)   Keling Yue (keling)

Winter 2024

For code, please visit the github repo.

1. Abstract

We visualize inequality in transit delays in Chicago based on CTA's data on bus positions. The Chicago Transit Authority (CTA) regularly reports on bus positions and bus schedules in the city of Chicago.

Our project scrapes these reports on a regular interval of one minute to gather bus and train locations and then calculate which stops are regularly delayed. For the analysis, we merge the bus locations with publicly available data on community areas to calculate which areas regularly experience delays. We then visualize which routes experience the most delays compared to data on income. These data are displayed using a Dash application.

This application also contains resources to schedule and request location data to continue the project. The data linked here represents one week of bus positions, from February 22nd to 28th, 2024.

2. Data Sources

2.1. Bus data

We collect data on bus positions and expected positions from the Chicago Transit Authority's public data. These data report high-frequency current data and schedules, but do not contain historical data that can be used to map delays.

- CTA bus locations The CTA bus tracker API provides bus locations each minute on their bus tracker API.

- CTA schedule: The CTA provides data in the General Transit Feed Specification (GTFS) that describes scheduled routes.

2.2. GeoData

We display these data on maps constructed using public data from the City of Chicago and the Chicago Transit Authority. These shapefiles are used to form the base of our visualizations:

- CTA Route GeoData: CTA routes are publicly accessible as shapefiles on the City of Chicago's Open Data portal.

- Chicago Community Area GeoData: Chicago community area boundaries are publicly accessible as shapefiles on the City of Chicago's Open Data portal.

2.3. Sociodemographic data

Finally, sociodemographic data on income is added to the map to contextualize where transit delays occur. We draw all income data from the US Census Bureau:

- American Community Survey: We use block-level income and poverty measures from the American Community Survey for our map backgrounds.

3. How to use the software

Bus Pending operates in two modes

1. Visualization: the application cleans scraped data, creates summary measures, and then passes that information to a Dash application to visualize on your local machine. We link one week of bus locations for use here.

2. Scraping: the application creates a database of current bus schedules and bus locations. We include a shell script to automate calls to the CTA API for bus locations.

We describe details on how to run both components below:

3.1 - Visualization

The Bus Pending project can be run from a machine with Python installed.

1. Download the Bus Pending Github Repository to your local machine by running `git clone` 'https://github.com/RMedina19/bus_pending.git' to you preferred directory.
2. Run `poetry install` from the top-level directory '/bus-pending' to set up the working environment in Python. Note: this requires installing Python and Poetry, a Python package manager.
3. Download our database of bus positions (826 mb) and place it in a data directory '/bus-pending/data' or scrape data yourself. See below for details on scraping data yourself.
4. Clean and analyze data by running `poetry run python -m bus-pending`
5. Launch the dashboard by `poetry run python -m bus_pending/app`

3.2 - Scraping

Requesting data from the CTA requires an API key. A key can be requested for free.

Once you have an API key, we include a shell script to regularly scrape data from the CTA. Although we scraped bus locations each minute, this will exceed the rate limit of 10,000 requests per day, as each minute requires 13 requests to the CTA API for 18,740 requests per day. To scrape the script, do the following once you have the working environment set-up as suggested in Instructions:

1. Put your working directory into the bus_pending file by running `cd/bus_pending/bus_pending`.
2. Create a database using `poetry run python3 -m scraping -makedb`. This will scrape the scheduled routes and provide users the ability to select which components to scrape.
3. Run each iteration of the scraper by running `poetry run python3 -m scraping -quiet`. The –quiet option automatically loads the database if /data/buses.db does not exist and then downloads the full list of routes from the CTA API to scrape without requiring user input.

We provide a shell script schedule.sh, in the `./bus-pending/` folder to automate this process.

4. Application Structure

We structure the dataflow in two parts, aligning with the scraping and visualization described in Section 3. This data flow is visualized below:
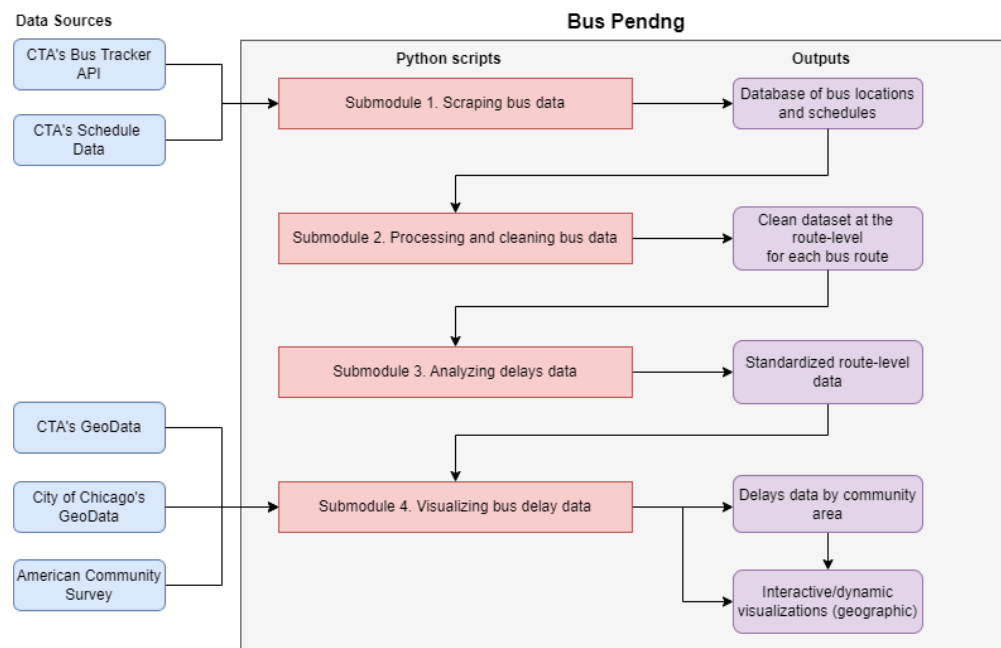
These lead to a structure:

```
bus_pending/
    analysis/
        analysis.py
        analyze_real_data.py
        compare_schedule_real_data.py
    app/
```

```
        app.py
    cleaning/
        clean.py
    scraping/
        __main__.py
        make_db.py
        scrape_buses.py
        scrape_routes.py
        scrape_schedules.py
    visualizations/
        acs.py
        app.py
        deck_app.py
        maps.py
    __init__.py
    __main__.py
    shcedule.sh
data/
    buses.db
tests/
.apikey
routes.txt
```

4.1 - Scraping Data Structure

The scraping elements are a standalone dataflow, meant to be collected once for visualization. These scripts conduct three tasks:

1. `scrape_routes.py` requests and store a list of routes locally from the CTA Bus Tracker API for future requests.
2. `make_db.py` creates a sqlite3 database in an ignored folder `./data` for schedule and bus location data.
3. `scrape_schedules.py` downloads GTFS data and load schedule and helper data into the database to produce expected transit delays. This is approximately ~250 MB of data and takes a few minutes.
4. `scrape_buses.py` scrapes bus location data from the CTA Bus Tracker API.

For projects that want to scrape delay data, (4) can be conducted regularly using scheduled calls to the schedule.sh shell script. The standard API key will allow calls every other minute.

4.2 - Visualization Data Structure

The visualization application is the primary component. Once data are downloaded, it:

1. `clean.py` queries the `./data/buses.db` database to extract bus-minute-level data and returns a route-level dataset with relevant columns as a CSV in the ignored `/data/` folder.
2. `analyze_real_data.py` and `analysis.py` Combines the route-level data with expected timing data from the `buses.db` data and determines delays.
3. !! Regina to add app structure

5. Members and Responsibilities

Our entire team jointly decided on the idea and approach of analyzing public transit delays. Work on software design and interfaces was decided as a group, including confirming which data would be used and the final units of analysis.

We split our coding responsibilities by each component of the application described above.

Author	GitHub Link	Lead
Michael Rosenbaum	GitHub	Scraping
Keling Yue	GitHub	Cleaning
Daniel Muñoz	GitHub	Analysis
Regina Isabel Medina	GitHub	Visualization and Server

These areas directly map to subfolders in the `./bus_pending` folder:

- Michael wrote all scripts in the `scraping/` subfolder and the schedule.sh script

- Keling wrote all scripts in the `cleaning/` subfolder

- Daniel wrote all scripts in the `analysis/` subfolder, including suggesting analytical decisions such as sample restrictions and pystest files in tests folder.

- Regina wrote all scripts in the `visualizations` and `app` subfolders.

Tests were written for each section as needed. Due to the nature of scraping and visualizing data, we chose to not include automated testing of scraping and visualization scripts, instead reviewing each product as it was produced for quality.

Finally, general documentation was decided on jointly and written by Michael.

6. Changes in Process

We initially set out to map administrative burden experienced in Chicago neighborhoods using train and bus access data to measure variation in wait time due to public transport delays. This was an ambitious goal. We accomplished a significant portion of that goal with this application, collecting one week of high-frequency data on bus positions that can be used to measure delays and then analyzing delays by route, and therefore location.

Our approach evolved based on some technical challenges and some focus changes based on interrogating the data more deeply. On the technical front, the CTA Bus Tracker API does not include the next stop, so we realized that we would need to determine when each bus passed each stop based on locating the next most likely stop on a route. This was not feasible given the time constraints of the course. In addition, reported bus data has a number of artifacts such as long duration routes, and short-length routes that suggest CTA bus reporting may not produce accurate data on bus delays at a granular enough level.

Instead, we focused on the route-level, determining how delays accrued over the course of the route. This data allowed us to better describe which buses experience delays regularly, and to plot those delays. Unsurprisingly, these are long local buses including the 4 (King Dr) and 8 (Halsted Ave) buses, which traverse the South Side of Chicago on the North/South axis to downtown.

We encourage you to explore the interactive plots, and better understand how often your local routes are delayed.