The premise of the assessment scraper project is to gather the valuation the municipal government assigns to properties for tax evaluation from their web interface and compare this to the price the property has sold for. For various reasons that affect demand, the selling price will deviate from the assessed valuation, but it serves as a benchmark to compare these deviations across the city. Within these results, I hope to identify clusters showing areas of increased or lesser demand indicated by consistently larger discrepancies in price-to-assessment value.

While Montreal provides some options to access public data (although recent it’s been getting better <http://montrealouvert.net/>), there is no API or other source for this assessment data so I had to turn to screen scraping the web form interface. The motivation for this project should be abundantly clear to anyone who attempts using this to try to retrieve an assessment.

***Investigating the data source:***

The first thing to notice is that when the link is clicked to access the web application from the link on the city website (<http://ville.montreal.qc.ca/portal/page?_pageid=3137,3741723&_dad=portal&_schema=PORTAL>), it opens a new tab, flashes random code from <http://evalweb.ville.montreal.qc.ca/> and redirects to <http://evalweb.ville.montreal.qc.ca/Role2011actualise/recherche.asp>. However, if we try accessing the last address directly without having been directed from <http://evalweb.ville.montreal.qc.ca/>, the web form will claim that the session is expired (try it out by clearing out your browsers cookies if you’ve been through the expected way already). This is the first clue that there is something fishy going on with the cookies at that intermediary stage. Initially, it may appear to have something to do with the code that is briefly displayed the page redirects, but as this snippet is not loaded, it actually does nothing at all. Instead, the browser is redirected to <http://evalweb.ville.montreal.qc.ca/default.asp> where the session id (ASPSESSIONIDSSSTQASD) is set by some behind-the-scenes .ASP magic.

Now that we’ve reached the input form, it should be just a matter of “typing the name or part of the name” as per the instructions in the form…or so it may seem. It’s probably a good first stop to try the main street in town and so for Montréal, we’ll try “rue Ste-Catherine”. The web form doesn’t want a prefix like “rue” so dropping that and trying “Ste-Catherine” returns a list of choices as we would expect. But if adding “rue” breaks everything, we can other variations in notation will as well. It turns out neither “Ste. Catherine” or “Ste Catherine” yield any result, however, something interesting does occur during the first attempt. As the form is submitted, “Ste-Catherine” is altered to read “SAINTE Catherine”. Digging into the page’s *scripts.js* file, we find it’s doing some manipulation to remove accents from the input string, replace hyphens and apostrophes with spaces, and replace instances of “St-“ and “Ste-“ with “SAINT” and “SAINTE” respectively. Now we have the expected bounds for our input queries.

With our input search a success, we are presented at the next screen, a list of arrondissements that the street passes through. The URL in the address bar doesn’t give much indication as to how we’ve arrived here but digging into the cookie again, there is a new entry for our input street name. The results are displayed as *street\_name*/*arrondissement*/*city* which makes it simple to do a string split on but one might notice that with a longer name such as “Arrondissement de Villeray – Saint-Michel – Parc-Extension”, the “Extension” is cut short to “Extens”. Therefore it is necessary to find a way to match this neighborhood string non-exactly. Another bigger issue is encountered when an entered address is a subset of another address. This could often be because of two streets with the same specific name but a different generic name (e.g., rue Clark vs. blvd. Clark). Worse yet however is when a street like “av. 4e” is entered and results are returned from “av. 4”, “av. 14”, “av. 24” … “av. 134e”. If we dig deeper and inspect the source of one these list options, we notice each option has a unique identifier value.

After one of these options is selected, we are brought to a yet another new tab containing a list for every building address of the street in the selected arrondissement. Examining the cookie, we can see there’s nothing new added there but this time, the URL contains a unique identifier—the same one from the arrondissement list on the last page. Examining the source on this page, we can see that it follows the same system as before with each address having its own unique ID number. Clicking through, we will finally land at the results page for the data we are looking to scrape.

From this we’ve gathered that we need four things in particular:

1. an input address cleanser
2. a method to select the correct arrondissement
3. a method to select the correct street
4. a mechanism for appropriately handling the cookie requests and behind-the-scenes javascript/ASP.NET shenanigans

***Writing the scraper:***

In the first post on building a scraper requiring user interaction with forms, I investigated the data source of Montreal’s tax assessment roll to get a look at what actions a scraper script would need to perform. I had a spreadsheet of 30,000 sales prices for properties in the city between 2005 and 2007 I wanted to compare to the city’s assigned property value. The Python libraries for manipulating .xls spreadsheets are a bit slow compared to any actual database, however, since respectful scraping requires a time delay between requests to not impact the service for others, it was fine to leave this as is. I divided this property sales spreadsheet into 15 smaller spreadsheet files of 2,000 rows each to make things more manageable and easier to debug (*link to split\_input.py*). Because the assessment roll’s web interface calls for selecting the arrondissement during the scrape lookup, I took the opportunity here to assign the written names from the geocoded code assigned by Canada Post using a lookup table in another spreadsheet.

With the input spreadsheets set up, next was writing the scraping scripts. Since I needed a way to debug and test addresses individually, the main script (*assessment\_scraper.py*) is designed to check if input is being supplied by the user or by a spreadsheet (this write-up discusses the process involved for being automated since the user acts to choose the correct addresses when run manually)*.* If a spreadsheet source is being used, *assessment\_scraper.py* is called via *scrape\_xls.py* where address formatting is fixed to match the input expected by the assessment roll’s web interface. However, because possessive phrases like “de la” and “du” can appear rather indiscriminately at either the start or end of the returned street names by the web app, these are parsed separately within the main *assessment\_scraper.py* script. The scraper’s design is that it should fail at every opportunity since bad data is worse than missing data and so it tries different variations for finding the best result. In order to do this, the script creates a list of strings of the street name as written and variations without these phrases if applicable. To communicate with the assessment roll web interface, I needed something that would make it easy to interact with forms, submit requests, and manage cookies…essentially a barebones browser which I found in *Mechanize* (link to mechanize site. The script iterates through this list of the street names by initializing a browser session using *Mechanize* (via *Browser.py*) and submitting requests until a list of street names is returned. In order to submit the request, Browser.py’s ‘*mimic\_cookie\_js*’ function is called since the search parameters to the server are submitted here (link to relevant section in post 1). When it finds a match in the results, it breaks from its loop greedily since this will be the closest found match.

When a result for the street is found, the script follows the same process for finding the correct arrondissement since this is required to see the list of street numbers. As is tradition for the assessment roll, this list of results often contains abbreviated strings for the names of the streets and arrondissements making exact string matching impossible. Instead, the scraper calls the function ‘*nbhd\_search*’ (via *MontrealAddressParser.py*) to perform 3 variations of fuzzy string matching (using *Fuzzy.py* which utilizes the *jellyfish* library in PyPi) to find the correct street and arrondissement. The proper arrondissement is selected from the form and a request is sent to the server by Mechanize to retrieve the street address list.

This list of street numbers contains the data to construct the final URL for the assessment by containing a unique unit ID number. By matching the input street number to a number in the list, the assessment data is fetched and then passed to the *TableParser.py* module. The table parser is a specific module to simply extract the assessment data from the HTML tables and puts this information in a dictionary. The dictionary is then returned to the calling script for outputting however is most suitable (typically a *.csv* with the spreadsheet or Flask interfaces) or returned as a raw text if the script is being called manually.

***Getting the data to GIS:***

After the data was scraped from the assessment roll, it needed to be spatially referenced so it could be analyzed. I found the most flexible but accurate tool to be Google’s geocoding API, the only caveat being its 2,000 free query limit per day. Due to this, I settled on outputting the geocoded data to an SQLite database so that it could quickly be resumed when the quote was reached (*link to geocode\_xls.py*). This also meant that if any request took longer than any predetermined timeout period, it could easily be tried again the next time it was run. By creating a tool that necessitated a few passes, I was reasonably sure that all the addresses capable of being geocoded were determined without the issue of network errors.

After the spreadsheet of input data had been scraped, I added a view with two additional columns in the database I thought would be useful in my analysis later using SQL functions. The first metric was the difference between the actual sales price and that of the city’s property assessment. The second was this price difference represented as a percent from the assessment price.

The data then needed to be imported to a format that can be easily manipulated using GIS tools. The AdressQuébec data is in ESPG:3798|NAD83/MTQ Lambert, the geocoded coordinates from Google must be reprojected. The easiest way to do this proved to be exporting the data to a tab-delineated .csv file and then using QGIS with its “Add Delimited Text Layer” plugin to import the .csv. Once loaded, the data should be projected as EPSG:4326/WGS84 using a coordinate system. This can then easily be reprojected to EPSG:3798 by saving a new shapefile.

The data in the shapefiles is good as a backup and for sharing the data to collaborate but for easier management with QGIS, importing the shapefiles to a PostGIS tables makes the data more accessible for queries later on (as discussed here <https://www.youtube.com/watch?v=oOhbbEkl4Kg>). In order to do this, a database is first created in a PostgreSQL server and PostGIS extension enabled to create the necessary geometry columns.

$ createdb gis\_table

$ psql gis\_table

> CREATE EXTENSION postgis;

BostonGIS has a useful cheatsheet (http://www.bostongis.com/pgsql2shp\_shp2pgsql\_quickguide\_20.bqg) for using the shp2pgsql utility to import the shapefile to Postgres. (Add shapefile to postgis database, define projection with –s, pipe to psql to import to PostGIS)

$ shp2pgsql -W “latin1” scraped\_data.shp | psql -h localhost -d *db* -U *gis*\_*user*

Because of timeouts and the assessment roll site being periodically unavailable, scraping the data from the input spreadsheet took the better part of a month. The scraper was designed to quit whenever it was presented with a potentially ambiguous situation since it would be much harder to later remove erroneous data.

My most common problem was that the address written in the spreadsheet would be for a single unit (e.g., 4937 rue St-Hubert) out of a development that had been assessed as a group (i.e., 4994-5000 rue St-Hubert). Furthermore, sometimes this individual sales price reasonable matched the price indicated on the assessment while others were wildly different indicating the mismatch of units. In addition, real estate is sold for personal or familial reasons at prices greatly different from their assessed value and these interfere with trying to distinguish the mismatch of units by searching for outliers in price alone.

In order to deal with this, I decided that scrubbing my assessments of those for a range of properties would account for this mismatch while retaining legitimate outliers and introduce the least bias (clean\_db\_data.py). With the data now in PostGIS to be used in QGIS or exported to ArcMap, I

***Analyzing the Data***

The first thing I wanted was to try to get a grasp of areas of the city where properties for greater or less than their assessed property value. Here I was looking for an indication of areas that has risen or fallen in value since the last property assessment. When I made the heat map using the QGIS plugin, I noticed that large areas like the airport, rail yards, or especially parks caused neighboring values to be stretched beyond a reasonable extent. I attempted to correct this by using the data for zoning and green space from OpenStreetMap. The heat map, however, was too fine of a grain and I decided it would be better to first compare

*Commands*:

shp2pgsql -s 4326 -W "latin1" nad83\_routes.shp | psql -h localhost -d *db\_name* -U *gis*\_*user*

Add shapefile to postgis database, define projection with –s, pipe to psql

***Database Wrangling***

Being able to get anything from the data means the data needs to be in a usable format first. Since right now, the scraped data is still dispersed amongst thousands of .csv files, my first step was adding them to a SQLite file since it’s easily manipulable and portable. This I accomplished through *csvs\_to\_db.py* which iterates through each .csv in the folder for the assigned year and inserts each to a database table in *mls\_sales\_nolatlng.sqlite*.

Since this data is to be studied spatially with GIS, the next task was to geocode each sale. The best publically available tool I found for this job was Google Map’s API, however it is limited to 2000 queries each day. This meant I had to implement a method to resume from the last place it left off while ignoring addresses it had previously been unable to successfully geocode to avoid using up the limited requests unnecessarily. Therefore, *geocode\_xls.py* creates tables for both successful and failed geocode queries on the GMaps API. For each address, the script first checks if the address has already been successfully geocoded and moves on if a match is found. Otherwise, the script will then check if the address is in the failed\_addresses table or finally attempt to geocode the address if a match is not found in either. (In addition, while writing *geocode\_xls.py*, the geopy library on PyPI didn’t function as expected because the *exactly\_one* variable needs to be tweaked so it checks for it correctly in the *googlev3.py* geocoder)