Final Project Documentation

Adam Biesenbach

DS GA 1007

Spring 2015

What Does the Program Do (What Output Does the Program Produce)?

The program analyzes the geo-spatial distribution of 311 complaints for the New York City area. In a nutshell, it pulls in a subset of the 311 data, cleans it, prompts the user for (1) the agency that they are interested in, (2) the complaint type they are interested in, and (3) the date range that they are interested in investigating for that agency and complaint type. If the user decides that they want to change their previous choice at the agency and complaint menus, they can return to these menus by typing '**back**'. At the date range menu, if the user would rather use the maximum date range available (as opposed to explicitly typing in a date range), they can do so by typing **'max'** at either the **'start date'** prompt or the **'end date'** prompt. If at any of the prompts the user would like to exit the program, they can do so by entering **'finish'**.

After this input is received and validated, the program produces a heat-map of NYC with different neighborhoods colored according to the density of the selected complaint within the neighborhood boundaries (as defined in the NYC shape file I found, detailed below). The buckets for the different levels of the heat map were generated by the 'Jenks breaks' classification method, which is an algorithm designed specifically to maximize the similarity of numbers in groups while maximizing the distance between the groups. More information on using Jenks breaks and other classification methods in Python can be found here.

In addition, the program produces a scatter plot of NYC dotted with the locations of the given complaint type, as well as a pie chart that provides the distribution of that complaint type for that date range across the five boroughs. All the figures are saved in the **/arb348/Output** subdirectory, with the figures saved as:

(1) **"Heatmap_[complaint_choice]_[start_date]_[end_date].png":** heat map for that complaint and date range.

(2) **"Scatter_[complaint_choice]_[start_date]_[end_date].png":** scatterplot for that complaint and date range.

(3) **"BoroughPiePlot_[start_date]_[end_date].pdf":** pie chart that provides the distribution of that complaint type for that date range across the five boroughs

After this is done, the program will prompt the user with a yes/no question, as to whether or not they would like the program to produce some additional output for the date range they selected. If they say '**yes**', the program will produce some additional summary charts, including:

(1) **"AgencyBarplot_ [start_date]_[end_date].pdf":** bar charts of the 10 departments with the most complaints.

(2) **"BoroughBarplot_ [start_date]_[end_date].pdf":** bar charts of the number of complaints by borough for the date range.

(3) **"ComplaintPieplot_[start_date]_[end_date].pdf"**: pie charts showing the distribution of the top 10 complaints over the time period.

(4) **"ZIPBarplot_[start_date]_[end_date].pdf"**: bar charts with the top ten ZIP codes of complaints over the period.

These figures are saved in the same output subdirectory: **arb348/Output**. If the users enter '**no**,' the program begins again by prompting the user for another agency/complaint/date range (this will also occur if they answered '**yes**', after the additional output is produced).

How Do You Run the Program?

The user only needs to run **main.py** to run the entire program; all the other processes will run from this module.

Are There any Dependancies that the Program Requires?

There are a number of specific modules that need to be installed before heatmaps can be created according to the method that I undertook. Below are the unusual packages used in my program that may need to be installed. I installed them all using the 'pip install *' package management system.

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors
from matplotlib.collections import PatchCollection
from mpl_toolkits.basemap import Basemap
from matplotlib.colors import Normalize
from shapely.geometry import Point, Polygon, MultiPoint, MultiPolygon
from shapely.prepared import prep
from pysal.esda.mapclassify import Map_Classifier
from descartes import PolygonPatch
import fiona
from itertools import chain
```

In particular:

1. **Fiona** is a package used is for reading geometries and attributes of shapefiles. In this program we use a NYC shapefile to create the heatmap images.

2. **Basemap** from **mpl_toolkits.basemap** is a module for plotting data on maps with matplotlib. We use it here to create an map class instance with attributes upon which we overlay our NYC-specific information.

3. **Map_Classifier:** this module from **pysal.esda.mapclassify** is a data clustering class designed to determine the best arrangement of values into different groupings. This is done by seeking to minimize each class's average deviation from the class mean,

while maximizing each class's deviation from the means of the other groups. In other words, the method seeks to reduce the variance within classes and maximize the variance between classes. This is used in the **ComputeJenksBreaks.py** module to help determine the proper buckets for the levels of the heatmap, by cutting up the values into discrete groupings.

4. **matplotlib:** the final heatmaps are created and saved using **matplotlib** functionality. The **cm, PatchCollection,** and **colors** functions are all also used to create the heatmap and scatterplot figures.

5. **shapely** is a module for manipulation and analysis of geometric objects in the Cartesian plane. Here we use it for many purposes, but its primary use is in creating objects in map coordinates from our DataFrame longitude and latitude values.

6. **PolygonPatch** from **descartes** allows us to take the individual community districts in our NYC shapefile (stored as polygon objects), and convert them to patches to make it easier to plot the information using matplotlib.

7. **Chain** is used to return elements from a first iterable until it is exhausted, then proceeds to another iterable, until all of the iterables are exhausted. This function is used in the **CreateBasemapInstance** function for creating a list of the shapefiles' coordinates.

8. **matplotlib.colors**: This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap. In particular, a data array is first mapped onto the range 0-1 using an instance of **Normalize** or of a subclass; then this number in the 0-1 range is mapped to a color using an instance of a subclass of **colormap**.

If External Data Sets are Used, How to Obtain Them?

The main raw data used in this project are 311 service requests in .csv format, imported into a pandas data frame (using class method) in the **DataClass311.py** module . The dataset can be downloaded here. I included a subset of the data for the 311 requests from March 2015 to the present (I couldn't upload a file any larger to github), and the program can be run using this data. If a file with more observations is desired, the user can obtain it from the site linked above. The variables used in the program are:

"Created Date", "Complaint Type", "Agency", "Agency Name", "Borough", "Longitude", "Latitude", "Incident Zip"

In addition to this dataset, shape files containing geo-spatial information on New York City communities are used to allow bucketing of the complaints along neighborhood lines. The original shapefiles I downloaded were from this source. The file that I used was labelled "Community Districts (Clipped to Shoreline)".

However, the shapefiles from this site are not formatted correctly for use with the **fiona,** since their coordinates are given in feet rather than degrees of longitude and latitude (which **fiona** needs). After some struggling, I solved the conversion issue using a free GIS program called QGIS, available here. To convert a shapefile to geographic coordinates, I first load in the shapefile into QGIS, and covered the geographic coordinate system to WGS 84, the global geodetic systems or datum used by all GPS equipment, and which is also used by **fiona.**

Luckily, the shapefiles are rather small and I was able to load them into **github** for the user. The files are stored in the **/arb348/Data** directory, and are named "**nyc_corrected**" with various file types. Since the program will read these files automatically, the user should not need to do anything to properly incorporate them into the analysis.

What is the Structure of the Package?

To make the program easier to understand, I tried to name individual modules and the functions and classes within them as clearly as possible. The modules that contain plotting functions are all within the **/Plotting** subdirectory, while the modules that deal with obtaining input from the user are contained in the **/UserInput** subdirectory. The main module (**main.py**), the test module (**test.py**), and the module housing the user-defined exceptions (**Exceptions.py**) are all in the main directory, to make them easier to find and/or execute. In addition, the data used in the project is stored in the **/Data** subdirectory, and the output files (the matplotlib plots) are stored in the **/Output** subdirectory.

What Else is There to Know About this Program?

1. Since the 311 dataset this program calls is rather large, I decided to design it so that the file only needs to be loaded and cleaned once. Multiple heatmaps and other charts are then created from this data set. In fact, the user will be repeatedly prompted for input until they enter '**finish**'.

2. The code written for **Natural_Breaks** class and the **natural_breaks** function are part of the **pysal.esda.mapclassify** package, and were not written by me. I needed to add versions of them explicitly to my module (as opposed to just importing them) because I wanted to remove some automatic print statements that the function produces when the number of observations is small. This code is contains in the **/arb348/Plotting** subdirectory, in the **ComputeJenksBreaks.py** module.

3. The way I've chosen to perform the analysis here differs somewhat from my original project proposal. My original plan was for the user to choose a specific summary statistic over a given time period, and that these numbers would be used in generating the heatmaps. In hindsight, I think it only makes sense to create the heatmaps by neighborhood for the total numbers of complaints themselves. As an alternative way of providing additional summary information to the user, I've included an optional subroutine that allows the user to choose whether they want additional information for the time period they requested. If they answer yes, the program produces the additional summary figures mentioned above, which provide breakdowns along Borough, ZIP code, Complaint Type, and other dimensions.

4. The creation of the heatmaps was a new process to me, and I used a number of resources online to help come up with a framework for working with them. Tom MacWright wrote a great overview of the tools used to create them on his webpage here. For help in creating the matplotlib Basemap instances, Thomas Lecocq's blog entry here was helpful. I also took inspiration from Tyler Hartley's example here, where he creates a heat map of his own location history in Seattle based on google maps data.

5. The **test.py** file contains tests to ensure that the necessary data is present before the program is run. In particular, this module should test (1) for the existence of the 311 data file (named '**311_Service_Requests.csv**'), (2) for the existence of the various shapefiles needed by the program, and (3) that the pandas dataframe created from the raw 311 data contains the necessary columns for the program to run.

6. To make exception handling a bit easier, many of the exceptions caught by the various try-catch statements in the program refer back to the **Exceptions.py** module, located in the main **/arb348** directory.

7. Only valid input should be accepted from the user when they make their choices. At the 'agency choice' and 'complaint type' menus, valid input means an integer in the list corresponding to the 'Department ID' or 'Complaint ID'. Other that these values, the only other acceptable inputs are '**back**' (to return to the 'agency choice' menu from the 'compliant menu' only) or '**finish**' (for either menu).

For the date range choice, the date must (1) be included in the date range for that specific complaint, and (2) have the ending date that does not precede the starting date. Other that these values, the only other acceptable inputs are '**back**' (to return to the 'complaint type' menu) or '**finish.'**