NationBrowse

Making sense of civic data

CS4970 Group 2

Graham Greenfield
Jeremy Howard
Nick Roma
Mike Tigas

Live URL

http://nationbrowse.com/

Source Repository

http://github.com/mtigas/cs4970_capstone

Table of Contents

List of figures & tables

Executive summary

NationBrowse is a Web application that provides a browsable front-end to civic data provided by the United States Government. The primary goal of the Web application is to not only provide an easy-to-use system for browsing and compiling statistics, but to also offer statistical analysis and comparison functionality.

The sheer amount of data available from the 2000 United States Census — among other datasets — presents a significant problem to those who may be interested in demographic information. (Note that the 2000 Census represents only a fraction of available datasets available through the U.S. Government's Data.gov directory.) Providing an application interface to assist in analyzing the data would be extremely useful to research organizations, marketers, investigative reporters, and the public at-large.

Beyond tables of numbers and statistics, the application will support dynamic generation of charts and graphs of the stored information. By presenting data in graphical form, it is hoped that the application will be usable to a wider variety of non-research audiences. In addition, map-based browsing of the data will be implemented; for example, a dataset such as population size could rendered as a color-coded (choropleth) map of the respective geographic areas — national, state, county, etc.

For those performing research on the data, a graphical user interface to "build" custom tables of data beyond those found via browsing, will also be provided. The ability to customize the acquisition of data, generate statistical analyses, and generate graphical representations of the data should prove useful in a research environment and to curious non-researchers alike.

Problem definition

The overwhelming amount of information available via the Data.gov directory makes it difficult for people and organizations to use. While releasing large amounts of government data presents a step toward transparency, making the data useful actualizes the goal of the government's Data.gov project.

Civic data and census data are not new — the United States Government digitally first released Census data in 2001 (for the 2000 National Census) and these datasets are widely used. However, only recently have Web technologies evolved to the point where a user-centric browsable presentation of the data is possible.

Few dynamic applications to civic data have existed until very recently. Since the launch of the Data.gov project in 2008, Sunlight Labs, a pro-transparency think tank, has sponsored an annual contest for applications (Apps For America) that use civic data from the Data.gov Web site.[1][2] Apps For America recognizes applications of Data.gov datasets that provide a useful tool to the public. Several apps from the Apps For America 2 contest appear to have goals that fall in-line with those of NationBrowse:

- ThisWeKnow[3] provides a browsable interface of many Data.gov datasets, but does not perform any analyses or provide any dynamic element to the browsing experience. There are no maps or alternative perspectives on the data, information is simply presented as a list of trivia regarding the user's selected location.
- DataMasher[4] provides a way to compare states against one another by "mashing up" two disparate datasets. For example, one may divide crime rate by high school graduation rate, and create a color-coded map that displays that relationship in each state. DataMasher admits that the Web site's output is not statistically rigorous, but is simply a tool to drive conversations regarding the data.

In addition to Web sites featured on Apps For America, several other Web sites have inspired this project either through use of civic data or the use of specific open-source technologies:

- The Los Angeles Times launched a "neighborhood mapping" project[7] that provides lists of demographic information for each L.A. neighborhood, in a very similar vein to the ThisWeKnow project, above.
- EveryBlock[5] is a Web site developed by several former programmers in the news industry, which provides aggregation of local civic data (police records, building permits) in addition to news stories and other information. EveryBlock provides a custom map interface to much of it's data. Paul Smith, a primary developer for EveryBlock's mapping infrastructure, recently wrote about the open-source GIS stack that EveryBlock uses.[6] The EveryBlock "stack" has directly inspired portions of the mapping and aggregation features of this project's scope.
- FiveThrityEight[10] is an poll-tracking Web site that follows state and national elections. During the 2008 U.S. Presidential Election, the site provided maps with projections of election results, based on data acquired from a large variety of pollsters. The current blog often displays charts and graphs of relevant data. It appears that these charts and graphs are manually created and not a result of dynamic generation from a database or data source.
- The *New York Times* releases a variety of mapping projects every few months, much of it based on civic data covering New York City. A current example of such a project is a map of homicide locations within New York City via a database updated from New York Police Department statements. [11]

A quick look at the above sources shows the wide variety of tools available to browse demographic data. However, few of the above products creates any sort of statistical analysis regarding the wide variety of data supplied by the U.S. Government. Mapping and automated chart generation is also lacking — many maps and graphs are one-off creations in these products, while our goal is to provide coverage of these tools for *all* locations in the datasets. It is our hope

that NationBrowse allows a more thorough and useful (in a significant research sense) look into the information that the United States Government has released.

Requirements analysis

System constraints

As a data-driven Web application, NationBrowse is inherently constrained by the availability and usability of data supplied by the source(s). As of December 2009, the data will be manually imported into a database internal to the application. Because the data model for demographic information is generic, the opportunity for automated imports and updates exists, but relies on consistent formatting from the original data source. This is currently not the case as even the same dataset will vary in format and caveats from year-to-year — as can be seen by variations in Uniform Crime Reporting data from the FBI.[17]

The project is intended to be a publicly-accessible Web site. This implies a Web (HTTP) server implementation along with the programming interface to allow access to datasets stored in the application.

Performance requirements

The resources required to render data, chart images and maps could affect usability of the application — by the performance of server and database components. Aggressive caching and database design optimizations may be required when considering queries across large datasets — there are over 3000 counties and 60,000 ZIP codes tied to Census data.

NationBrowse will require statistical analyses to be performed over various datasets. Some functions (mean, standard deviation) are built-in to currently popular database servers, including

MySQL[18] and PostgreSQL[19]. Others are available in open-source software libraries (such as the SciPy Python library[1]), while other algorithms may have to be developed.

Image generation for charts could produce another significant stress on the application server. Therefore, "simple" charts for non-comparative data (pie charts or bar charts for demographics) will be delegated to the Google Charts API, while more complex charts dealing with comparative data (place-place or variable-variable correlations) will be rendered directly by the application.

Resource requirements

The application software requires an HTTP Web server with access to Python. The map rendering and GIS portions of the application are built toward the PostgreSQL database using PostGIS extensions, but may utilize any GIS-aware database.

The data that drives the application comes from the United States Government on Data.gov. Specifically, GIS data will be sourced from the Census Bureau's TIGER/Line shapefiles [23], which provide detailed shape/border data for the United States, all states and territories, all counties, and all ZIP codes within the United States. Other data (demographics) will be sourced from Census Bureau data and other datasets available on Data.gov.

The Django framework[12] includes a caching framework so that rendered pages and data do not have to be processed as frequently — this could aid the application from a performance standpoint (see "Performance requirements," above). For best performance, the Django documentation recommends the use of the memcached cache server software, although the Django caching framework may use a disk-based cache if this is not possible.

[1] SciPy provides a Matlab-like programming interface to many useful mathematic and statistical computations.

Alternative solutions

We chose Django[12] as the overall Web application framework, because it provides an ORM (object-relational mapping) API that handles much of the database work. In addition, Python provides a large standard library. Through Django and Python, many low-level tasks, such as interfacing with the database or serving templated pages, are made simpler. Using Django and Python allows us to concentrate our efforts on the statistical and mapping portions of the application. Further, Django's "GeoDjango" library natively provides tools to utilize GIS-enabled databases.

SciPy[13] and MatPlotLib[14] were chosen as the statistics backend and chart generation backend. These provide similar analysis and charting functionality to R and Matlab, but through the Python language. This provides excellent integration with the Django-powered front-end. MatPlotLib does not have the same notoriety as R or Matlab, but users and contributors include researchers at NASA's Jet Propulsion Laboratory and the National Oceanic and Atmospheric Administration (NOAA)[15].

Alternatively, PHP-based solutions could have been used to create a similar project. (Similar Web application frameworks in PHP include Drupal and Doctrine.) However, the ability to interface with GIS-enabled databases and the high availability of scientific/statistic calculation libraries and graphing tools made Python a solid decision.

The database layer will be implemented in PostgreSQL, with the PostGIS extensions installed. This is the recommended GIS-aware database implementation for the GeoDjango project.[20]

Maps will be implemented in the OpenLayers Javascript framework. In addition, basic "base layer" maps will use the Google Maps API in OpenLayers. OpenLayers will allow us to render shapes and data within the user's browser, rather than the server — this method will allow interactivity within the map without resorting to less-accessible methods such as Adobe Flash.

Alternatively, mapping could be performed by a server-side GIS solution, such as Mapnik — which was originally planned for the project. Browser interactivity and server resources were the primary reasons a client-side interface was chosen. Mapnik, MapServer — another free and open-source project — and ESRI's ArcGIS Server — a commercial product — provide similar levels of map image generation, but vary in levels of database backend support. Mapnik utilizes Python, the primary language for this project, while the other products are provided in either C(++) or binary form.

In terms of full implementation of color-coded maps, ESRI also provides a commercial product, InstantAtlas Server. This product, which implements a front-end in Adobe Flash, generates both maps and charts of given data. It is marketed primarily as a "visual presentation tool" and does not provide any statistical analysis nor the ability to interface with and compare multiple datasets at once.

Evaluation metrics

As a Web application, the project will mostly be evaluated on feature-completeness and performance metrics. Django and Python offer various methods to test performance: the primary performance metrics would be server response time (or page load time), CPU load, and database load (as measured in number of queries).

Django provides notification e-mails for all server exceptions, which would allow up-to-the-minute debugging if the application were to fail in a launched, production environment.[22]

In terms of user experience, a click tracking tool such as Crazy Egg[21] could be used to further understand the usefulness and popularity of our feature set.

Statistical analyses should also prove "sound" based on accuracy and correct use of statistical algorithms. Initial evaluation meetings with Dr. Lori Thombs — director of the Social Science Statistics Center at the University of Missouri — and Seung Bin Cho — a graduate

student working with Dr. Thombs — led to the removal of some overly-ambitious statistical features including hypothesis testing. The recommendations from Thombs and Cho were to provide more focus on descriptive statistics — especially in the form of boxplots and other charts — than automated analysis. This is due to the fact that the civic data provided are already derived from samples that our application does not have access to — creating automated analyses from this information is inherently flawed and "dangerous" to make statements from.

Design Specifications

Overview

The project will provide an interface and tools for people to browse and analyze civic data. The project will generate graphical representations of data, in the form of maps and charts, and output descriptive statistics (in the form of tables and charts) on aggregations of the data.

Software design

The program will be built in the Python programming language and use the Django application framework. The SciPy library will also be used to provide mathematical/statistical features in the application. MatPlotLib will be used to implement application-rendered graphs, while the Google Charts API will be used (via generated URLs) for simpler, non-comparative graph types.

The front-end will be HTML/CSS-based and use several Javascript-based libraries, including jQuery (for basic effects and HTML manipulation) and OpenLayers (for generating maps).

Data requirements

The project will use PostgreSQL as the database server backend, because of the availability of the GIS extension, PostGIS, which allows the database to be queried based on

geographic information. Additionally, PostGIS is the recommended GIS backend for GeoDjango. [20]

The data is sourced from Data.gov and Web sites related to that catalog, including the United States Census and the FBI Uniform Crime Reports. Data will have to be imported manually — usually through writing one-off importers for each dataset. This is required solely because Data.gov datasets are not currently consistent in file format, data formatting, or organization methods.

Geospatial data will also come from the United States government, in the form of Census TIGER/Line data[23]. PostGIS provides tools for importing the data into the database, from the format that the TIGER/Line data is provided in (ArcGIS .shp files).

Hardware

The primary back-end of the application should conceivably run on any server capable of running a Web server (Apache, lighttpd, nginx, IIS, etc.), Python, and PostgreSQL+PostGIS.

In a large-scale deployment, the Web application server, the database server, the map/tile servers, and the hosting for "static" files (CSS, images, etc.) could (and should) be separated.[16] In a small, prototype environment (i.e., the scope of this class project), this outward-scaling of resources is not necessary.

Testing methods

Django has built-in test suite support[9], based on Python's built-in unit testing support. Error checking should be built-in to necessary portions of the application; fatal errors or exceptions which are uncaught are normally caught by Django, which generates a friendly "500 Server Error" page and sends traceback information to e-mail addresses in the site configuration. [22]

Implementation resources & costs

The primary costs underlying NationBrowse relate to the operational costs of the Web server. These costs vary from under $10 per month for a low-end shared server setup, to several hundreds of dollars for more elaborate setups. This determination of server requirements will inevitably be based on the number of end users and server load. Additionally, domain name registration fees apply.

The prototype, http://nationbrowse.com/ is hosted on a SliceHost[8] Web server owned by one of the team members. For demonstration purposes, the application will operate on a local machine, provided the PostgreSQL+PostGIS database is installed; this utilizes a built-in Django development testing server.

Product implementation

Overview

NationBrowse will be implemented as a Web application, written in Python/Django. The URLs and interface will be simple — the browsing interface should be up-front and not buried underneath several pages of homepage "fluff" and categorization.

Data model

Django provides object-relational mapping (ORM), which represents database rows of data as Python objects. To provide an abstract interface for dataset information and location data, the data model consists of:

- **"Places" objects**, each representing one particular place — i.e., the United States as a whole, or the state of Indiana, or Warren County, Missouri.
  - **Nation** (see "Place," below)
  - **State** (see "Place," below)

- **County** (see "Place," below)
- **Place**: Each of the above types inherits from an "abstract" base class (Place) that provides extra GIS wrapper functions and wrappers to access datasets related to a given place.

  Each object type includes a "poly" field which stores the shape of the location, so the location may be rendered on a map later. In addition, generic information such as FIPS (Federal Information Processing Standard) code — which is generally used as the "primary key" of a location in civic data — and abbreviations are also stored within these object models.

  All of the "places" data is sourced from the U.S. Government's TIGER/Line data files.[23]

- **"Demographics" objects,** representing the information in one dataset for one place — i.e., the 2000 Census data for Boone County, Missouri, or the 2008 FBI Uniform Crime Reporting data for the state of South Dakota.
  - **DataSource** objects are categorizations that represent the *source* of data (including year) for one row of information (2000 Census, 2008 American Community Survey, etc). In addition to name and year, a source URL is stored so that researchers can return to the source of data, if they wish. The other demographic object types key against this.
  - **PlacePopulation** objects represent the demographic information for one place, from a specific dataset.

    Fields include total population, population in urban areas, population in rural areas, population of one race, population of mixed race, population within particular races, population within particular age brackets, and many more from either Census or American Community Survey population counts.
  - **CrimeData** objects provide totals of reported crimes in a given place.

    This includes the total number of violent crimes — further broken down into

murders, rapes, robberies, and aggravated assaults — and the total number of property crimes. Some currently-unimplemented fields include totals of law enforcement officers employed in a given place. This data is sourced from the FBI Uniform Crime Reporting program.

- **SocialCharacteristics** objects provide socio-economic data for a given place, from a specific dataset.

  Fields include the number of people enrolled in school (which is broken down into several fields representing various schooling levels), per-capita income, and median income. This information generally comes from the American Community Survey, as the 2000 Census did not contain such figures.

The above data models may be found within the project repository, under:

- django_server/server/nationbrowse/places/models.py
- django_server/server/nationbrowse/demographics/models.py

Data browsing & maps

Data pages will be specific to one location. For example, the site contains specific pages for: Boone County, Missouri (http://nationbrowse.com/places/county/mo/boone/), Missouri itself (http://natiobnrowse.com/places/state/missouri/), and etc. The "root" view (http://nationbrowse.com/) will consist of the national-level data page.

For the national view and the state views, the "detail" page contains a Javascript-based map view that displays locations contained within the current one (i.e., for state pages, a map and list of the state's counties is displayed). The map utilizes OpenLayers and the Google Maps API to render shaded state or county shapes on top of a simple base map. The map is shaded by population size by default, but links near the map dynamically change the shading value to key off of either: population density, crime rate, or per-capita income. Hovering over a location on the map will highlight the place's name in the list and display very basic data in a miniature data pane,

adjacent to the map. Clicking on the location in the list or the map will bring up the detail page for that location.

Below the map view, tables of demographic, socio-economic, and crime data related to the location are displayed. Charts and graphs relating to the data are displayed inline.

Charts and graphs

In all, seven types of charts are supported by the application, though not all of them are currently implemented through the site's interface.

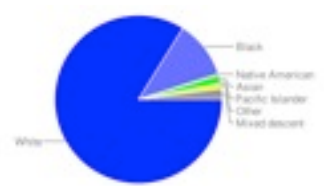

Figure A: Google Chart API, pie chart example
This is a race breakdown pie chart, generated from data for the state of Missouri. (Data: 2008 American Community Survey)

Three graph types are implemented via the Google Chart API: pie charts (Figure A, above), bar charts (histograms; Figure B, below), and "grouped" bar charts (Figure C, below).

Figure B: Google Chart API, bar chart example
This is a histogram of the population, by age bracket, within the state of Missouri. (Data: 2008 American Community Survey)

The charts are generated by passing data values, labels, and colors within a GET request to a Google Chart-specific URL. Within the application, each chart type is implemented in a wrapper function that accepts arrays of data, labels, colors, and size information and creates the Google Chart URL string.

Figure C: Google Chart API, grouped bar chart example
This "grouped" bar chart was generated via the wrapper functions,
from randomly generated dummy data.

Four other graph types are rendered directly within the application, using the Matplotlib Python library: boxplots (Figure D), scatterplots (Figure E), three-dimensional bar charts (Figure F), and scatter histograms (Figure G).

The API to the implementing functions appears the same as those wrapper functions for the Google Chart API: all of the take arrays of values, labels, and colors as input. Where these methods differ is that instead of returning a URL string, a Matplotlib "Canvas" is returned with the chart generated, but not yet rendered to any particular image file type.

Within the server, these graphs are utilized through URLs that contain some sort of variable aspect that contains the graph type and the name of the location that the graph is being generated for — implemented in Django's URL configuration system, this is similar to the URL hierarchy for place detail pages. It is up to the corresponding Django views (called via those particular URLs) to get the data from the respective Place objects (see "Data model," p. 10), call the Matplotlib wrapper function with the data, and render the returned Canvas as a PNG image

file. By directly rendering the image to the URL's response, the application does not need to store actual image files on the server filesystem. Additionally, because the PNG files are rendered through Django's request/response system, the result of a URL is cached using Django's built-in caching framework (see p. 5).

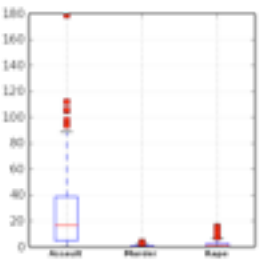

Figure D: Matplotlib boxplot example
Incidence boxplot for assault, murder, rape in Missouri
(Data: 2008 FBI Uniform Crime Reports)

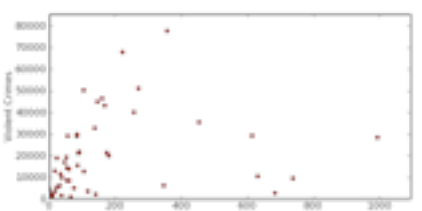

Figure E: Matplotlib scatterplot example
U.S. States, plotted by Population Density vs Violent Crimes Reported.
(Data: 2008 FBI Uniform Crime Reports)

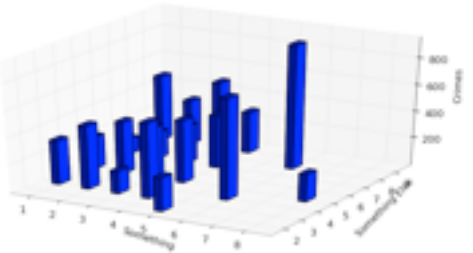

Figure F: Matplotlib 3D bar chart example
This three-dimensional bar chart was generated from randomly-generated dummy data. The 3D bar chart implementation, though complete, is not used in the current product due to lack of well-scaling data.

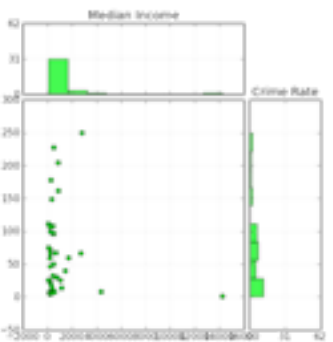

Figure G: Matplotlib scatter histogram example
Random sample of 100 U.S. Counties, plotted by Median Income vs Violent Crime Rate (per 100k residents).
(Data: 2008 FBI Uniform Crime Reports)

In prototype server testing, generating images in Matplotlib often overworked the server[2] for several seconds. By utilizing the Google Chart API to offload rendering of "basic" chart types, the application-side load required to generate images is significantly decreased and pages can be displayed to the user more quickly. (Or alternatively, more pages can be generated simultaneously.)

Charting code for the above features, including Django view code, can be located within the repository, in the following directory:

- django_server/server/nationbrowse/graphs/

The live examples for dynamically generated Matplotlib charts can be viewed at:

- http://nationbrowse.com/graphs/scatterhist_test.png
- http://nationbrowse.com/graphs/scatterplot_test.png
- http://nationbrowse.com/graphs/3d_test.png
- http://nationbrowse.com/graphs/boxplot_test.png

Custom data table interface

A custom "QueryBuilder" interface allows users to create their own table outputs of data. The interface utilizes an SQL-style approach of first "joining" the tables one is interested in, then "projecting" the columns one wants to see in the result table, and finally "filtering" the data (similar to SQL "where" clauses). The interface is implemented entirely in jQuery and jQuery UI.

The user interface queries the server to dynamically populate lists of which columns are available in the selected tables. The Django-powered server-side portion returns this data in a JSON-encoded array, which allows jQuery to directly import the data as a Javascript array object, without any further need for scraping or data parsing.

[2] This is to say, CPU load at or near 100%, and (UNIX) system load average over 1.00.

The final result is requested via an AJAX-like request to the server, which contains a list of the tables joined, the columns chosen, and the filters applied. In this final result, the server does not return raw data to the interface, but instead generates the HTML table itself and returns that. The jQuery-powered interface simply "injects" this HTML into the page to render the final output, an example of which can be seen here:

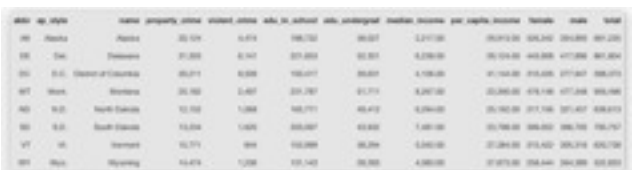

Table 1: Example of QueryBuilder results
Results generated from the QueryBuilder on the State table, joined with the three data type tables, with several columns projected. Filtered to only display if "placepopulation.total < 1000000" (states with population under one million.)

The QueryBuilder interface and code can be located within the repository at:

- sandbox/graphic query
  - graphic_query.html is the main page of the interface.
- django_server/server/nationbrowse/querybuilder/
  - views.py contains the server-side interface which provides data to the UI.

Evaluation of product performance

Regarding the objectives set about in pp. 2-10, the project is a partial success. This is measured mostly on the basis of feature-completeness and a basic understanding of performance speeds on normal Web sites.

In terms of feature-completeness, the product accomplished *most* of the initial objectives (data aggregation, mapping, graphing), but some initial ambitions did not make it into the final product. Each "branch" of the project — data output, mapping, graphing, and custom query building — were completed in (at least) a basic manner, to allow us to present a prototype with each disparate piece functioning. But several of our planned features were dropped from the current prototype. In particular:

- Matplotlib-based graphs are not currently dynamically created for every type of place, nor are they implemented in a readily-visible manner on "place detail" pages. This is due to server load issues, which are in fact, a two-fold problem:

  1. because data is stored in separate models (and therefore, separate tables) and because of limitations within the Django ORM, queries that require filtering on places and then aggregating data must join data *per place* and not via one all-encompassing aggregation query. In other words: queries to generate a simple scatterplot of several hundred counties must send hundreds or thousands of queries to the database, many of them running joins.
  2. rendering a chart from hundreds or thousands of input values takes quite some time and resources.

  Implementing the Matplotlib images within the templates would cause these images to be rendered immediately upon a page being displayed to a user. The resulting simultaneous load of multiple image requests would essentially overload the server and, in testing, was shown to lock up the server due to lack of RAM.

- Statistical analysis was originally planned for the project, but did not factor in largely due to advice from our primary statistical advisor, Dr. Lori Thombs. She and the graduate student that spoke to us (Seung Bin Cho) recommended we *not* perform automated analyses but instead display simple descriptive statistics (max, min, mean, standard deviation) and focus more heavily on visually representing the data with charts and graphs and maps.

  - Creating an output of "matrix of correlation" was recommended to us by Cho, but we were not able to come up with a suitable implementation.

- Display of the χ^2(chi squared) statistic was planned, but not implemented because direct variable-to-variable comparisons and testing were not implemented on the advice of Thombs and Cho.

- Mapping and graphs were not implemented in the QueryBuilder interface. However, we focused on completing the QueryBuilder feature at it's most basic form by December, rather than attempt to add more features to the application than we could handle. Given several more weeks, a dynamic method to choose graphs to display or choose more mapping variables would be possible to implement — and in fact, such a method could be used to alleviate some of the performance concerns regarding Matplotlib-based graphs in all pages (not just QueryBuilder-based results).

In terms of outright performance, the current application performs satisfactorily, at best. For the national and state-level pages, a noticeable delay of *several seconds* occurs if the particular page has not been requested before — mainly due to computations regarding map shape data and the large data tables that are displayed. The Django caching framework allows the application to simply retrieve a page from cache if it has been viewed before — resulting in near-instant load times — but "first run" performance is still poor.

Discussion, future work, and conclusions

The problem of unifying civic data and making it more useful is genuine. Government organizations could benefit from more consistently shared data. Researchers could more easily discover patterns and interesting trends regarding our nation.

Because Data.gov acts more like a directory or catalog of datasets, the overall aim of a project like NationBrowse is to act like a centralized datasource — much like the Data.gov site *should* operate. By aggregating the data and allowing easier browsing and custom querying, the project does meet some of these goals. In the future, features such as exporting custom tables to Excel, XML, or CSV files could make this "one datasource" goal much more useful.

However, data alone only provides *some* of the context required to make statements and make the information more relevant to average citizens. Currently — and even with a data source like NationBrowse — that job goes to researchers and reporters. Matt Waite, a Web developer and former reporter at the St. Petersburg Times (Florida) wrote on his blog last year, regarding "data ghettos,"[24] or, Web sites which attempt to super-aggregate datasets and datasources without context. Although speaking in terms of the news industry, his ideas apply to data aggregate sources in general.

Although this team was discouraged from making automated statistical analysis and claims, we still feel as though further context *could* be created within a project such as this.

While NationBrowse may fit the example of that "data ghetto," some of the tools designed in the project — a map-based browsing interface and dynamic generation of charts and graphs to provide alternative data visualizations — could be useful in future projects that try to make use of the ever-growing amount of civic data available.

References

[1] “Apps For America.” 1 April 2009. Retrieved 21 September 2009. <http://www.sunlightlabs.com/contests/appsforamerica/>

[2] “Apps For America 2.” 8 August 2009. Retrieved 21 September 2009. <http://www.sunlightlabs.com/contests/appsforamerica2/>

[3] “ThisWeKnow.” Retrieved 20 September 2009. <http://www.thisweknow.org/>

[4] “DataMasher.” Retrieved 20 September 2009. <http://www.datamasher.org/>

[5] “EveryBlock.” Retrieved 20 September 2009. <http://www.everyblock.com/>

[6] Smith, Paul. “Take Control of Your Maps.” *A List Apart*. 8 April 2008. Retrieved 20 September 2009. <http://www.alistapart.com/articles/takecontrolofyourmaps>

[7]”Mapping L.A. Neighborhoods.” *The Los Angeles Times*. Retrieved 20 September 2009. <http://projects.latimes.com/mapping-la/neighborhoods/>

[8] “Slicehost.” Retrieved 23 September 2009. <http://www.slicehost.com/>

[9] “Testing Django applications.” Retrieved 22 September 2009. <http://docs.djangoproject.com/en/dev/topics/testing/>

[10] “FiveThirtyEight: Politics Done Right.” Retrieved 5 October 2009. <http://www.fivethirtyeight.com/>

[11] “New York City Homicides Map.” Retrieved 6 October 2009. <http://projects.nytimes.com/crime/homicides/map>

[12] “Django.” Retrieved 25 September 2009. <http://www.djangoproject.com/>

[13] “SciPy.” Retrieved 8 October 2009. <http://www.scipy.org/>

[14] “matplotlib: python plotting.” Retrieved 8 October 2009. <http://matplotlib.sourceforge.net/>

[15] “Credits.” Retrieved 8 October 2009. <http://matplotlib.sourceforge.net/users/credits.html>

[16] Holovaty, Adrian and Jacob Kaplan-Moss. “Ch. 20: Deploying Django: Scaling.” *The Django Book*. <http://djangobook.com/en/1.0/chapter20/#s-scaling>

[17] “Uniform Crime Reports.” *Federal Bureau of Investigation*. Retrieved 24 November 2009. <http://www.fbi.gov/ucr/ucr.htm>

[18] “GROUP BY (Aggregate) Functions.” *MySQL 5.4 Reference*. Retrieved 10 December 2009. <http://dev.mysql.com/doc/refman/5.4/en/group-by-functions.html>

[19]”Aggregate Functions.” *PostgreSQL 8.4.1 Documentation*. Retrieved 10 December 2009. <http://www.postgresql.org/docs/8.4/interactive/functions-aggregate.html>

[20] “GeoDjango Installation” Retrieved 11 December 2009. <http://geodjango.org/docs/install.html#spatial-database>

[21] “Crazy Egg: Visualize your visitors.” Retrieved 11 December 2009. <http://crazyegg.com/>

[22] “Error Reporting.” *Django Documentation*. Retrieved 11 December 2009. <http://docs.djangoproject.com/en/dev/howto/error-reporting/>

[23] "TIGER®, TIGER/Line® and TIGER®-Related Products." Retrieved 11 December 2009. <http://www.census.gov/geo/www/tiger/>

[24] Waite, Matt. "Data ghettos." 3 January 2008. Retrieved 12 December 2009. <http://www.mattwaite.com/posts/2008/jan/03/data-ghettos/>