Progress Evaluation - Milestone 2

Title: Customizable Analysis and Visualization Tool for COVID Cases

Team Members:

- Calvin Burns, cburns2017@my.fit.edu (Team Lead)
- Sam Hartle, shartle2017@my.fit.edu
- Stian Olsen, shagboeolsen2017@my.fit.edu
- Nicole Wright, nwright2017@my.fit.edu

Advisor: Dr. Philip Chan, pkc@cs.fit.edu

Client: Dr. Philip Chan, pkc@cs.fit.edu

Progress Matrix for Milestone 2:

Task	Completion %	Stian	Sam	Nicole	CJ	To do
1. Setup Django environments	100%	20%	20%	20%	40%	none
2. Create database models	100%	25%	25%	25%	25%	none
3. Import data to newly created database via CSVs	100%	25%	25%	25%	25%	none
4. Implement Feature 4.1 (Customizable Operations on Variables)	25%	0%	10%	0%	15%	UI and supporting backend code

Discussion of each accomplished task and obstacles for Milestone 2:

- Task 1:

  This task involved creating unique scripts for macOS and Windows that would allow each team member to run an instance of the project from their localhost, create an admin account for themselves within the Django project, manage migrations, and carry out many other features built into Django.

  Obstacles included running a Postgres database with non-default credentials on Windows, certain parsing issues with Windows batch scripts, and the initial learning curve of understanding the Django workflow, specifically migrations for each new database model.

- Task 2:

  We needed a total of 8 database models. These models were defined in the Django-provided file *models.py*. These database models were:

  - User
    - Represents a basic account
    - Model inherits the Django default use
    - See attributes and methods here:
      - https://docs.djangoproject.com/en/3.1/ref/contrib/auth/#user-model
    - Model is a proxy, additional methods can be added
  - Location
    - Represents a single location
    - Hierarchy of locations can be constructed using parent locations and fetching children
  - Metric
    - Represents a statistic that is not time related
    - Example is population data
    - Model will be linked to a *User*
    - Can also be linked to a *Location*
    - When saving data, we will save the type and encode the value to binary
    - When we fetch data, we will decode the value using the type
  - Dataset
    - Represents a collection of *Datapoints* as a whole
  - Datapoint
    - Represents a single data point associated with a *Dataset*
    - A database field for date and location is used
    - A JSONField stores the rest of the data
  - Operation
    - Represents an abstract, editable math operation or formula
    - Utilizing the SymPy library
  - Plot
    - Represents a saved plot
    - Model will be linked to a *User* and a *Dataset*
  - Dashboard
    - Allows a user to save a configuration of plots, tables, and maps
    - *display_json* will hold a JSON object with all necessary information

Obstacles included an effective way to import and query the large amounts of data involved with Covid. Our initial design was a 3-layer model hierarchy of a *DataSet*, sets of *DataPoints* within a *DataSet*, and a set of *DataFields* within a *DataPoint*. As you might expect, the time/space complexity of this structure was pushing n^3. This design was implemented and then used to store the import of the Florida DOH data. The import took a considerable amount of time (30-45 min) and the space expanded from a 110 MB CSV file to a 900 MB Postgres table.

Another issue encountered with this design was slow load times when running database queries. This seemed like a good, abstract solution but it lacked the scalability we need for the large amounts of data we will be working with.

The second design was a Django JSONField. In a recent version of PostgreSQL, the developers added support for a JSON field. This allows us to save any JSON object in a database model. In addition, you can query objects using fields inside the JSONField. Built-in lookups like *contains* work well. These changes were implemented and the CSV with 750,000+ Covid cases were imported again. This import was much quicker (10-15 mins) than the first design method. As far as space goes, the newly created Postgres tables were 512MB, about half the size of the previous implementation

An example that displays the power of this is the following query:

filters = {'gender': 'Male', 'age': '55'}

data_points.filter(data_json__contains=filters)

We create a JSON object named *filters*. This holds the filters that we want to apply to the *Datapoints*. Using the built-in *contains* function, we get a query set of all cases where the gender was male and age was 55. This design is particularly useful to the design and creation of *Plot* models.

- Task 3:

  Each team member wrote a script to import a unique CSV file with data relating to Covid. These included:

  1. FDOH case data
  2. Lockdown Dates by country
  3. Lab Testing throughout Florida
  4. Mask Mandate Dates

- Task 4:

  The model is complete for an operation. The model utilizes the SymPyCharField, which will allow a user to input any operation(s) that is supported by the SymPy library. This library will do all of the heavy-lifting for processing operations and generating results.

  We still need to implement the UI that will allow users to interact with the dashboard in order to customize operations on selected variables. In addition, the backend code that will interact with SymPy to process these custom operations needs to be implemented as well.

Discussion of contribution of each team member to Milestone 2:

- Stian: Completed setting up the Django environment on my local computer so that we all work on the same environment. Created a database model which will store a *location* and a *dashboard* database model which will allow the user to save a configuration of plots, tables and maps. Wrote a script which reads a csv file with lockdown data and stores the information using the DataSet model. Demonstrated a short demo on how we can filter out data from the dataset created with the scripts for task 3.

- CJ: Set up a Django project for the team, wrote an installation guide along with scripts to standardize development environments. Set up a Heroku server for deploying our application. Write issues on GitHub and distribute work. Created the *DataSet* and *DataPoint* models. Helped teammates with remaining work on their individual models. Chose a frontend Bootstrap template and abstracted the HTML files for use with Django. Wrote a script to import State Location data. Wrote a script to import County Location data. Wrote a script to import FDOH Covid Case data(750,000+ rows). Created a demo to display multiple dynamic charts: Created 2 custom *plots,* saved the plots to a custom *dashboard.*

- Sam: Setup the Django environment on my local computer to ensure a uniform development environment. Fixed a Postgres DB login issue with the original Django environment setup script for Windows. Wrote a Windows batch script for Nicole and myself to automate tasks such as activating the Django environment before each development session. Created a database model which will store a non-time related metric (population data) and an *operation* database model which will allow us to store and eventually process a custom operation done by the user on the frontend. Brainstormed with CJ regarding design choices for importing and querying individual *DataSets*. Wrote a script which reads a csv file with lab testing data and stores the information using the DataSet model. Wrote the majority of the progress evaluation and created the presentation.

- Nicole: Completed setup of the Django environment on my local computer (Windows) so we can all work in the same environment. Resolved issues with the version of Python conflicting with the version of Postgres I installed, and resolved several path issues by manually putting Postgres paths in my environment variables. I created a database model for both ‘User’ and ‘Plot’ in models.py. The user model is already built into Django, so by creating a proxy I was then able to add additional methods for the user. The methods get_plots, get_dashboards, and get_datasets were created. For the plot model I added a list of plot types along with several model fields. Finally, I wrote a script which reads a csv file with mask mandate data and stores the information using the Dataset model. Also I found a better data source for mask mandates that has better date/time information than the one I was previously using.

Task Matrix for Milestone 3:

Task	Stian	Sam	Nicole	CJ
1. Continue Feature 4.1 (Customizable Operations on Variables)	Design a frontend operation selection UI based off mock-up in Design Doc	Work on majority of the backend code for processing selections from fronted for custom operations	Assist Stian with UI for selecting custom operations on the fronted	Further investigation of SymPy library to see how it will integrate with Sam's writing of backend operations code
2. Small GUI demo that integrates lockdown and mask mandate data	Create a line graph demo which displays a timeline using the dates in the lockdown and mask mandate datasets	Support Stian and Nicole with specific operations they need for their demo	Create a line graph demo which displays a timeline using the dates in the lockdown and mask mandate datasets	Oversee development and assist as needed since demo will be similar to Male/Female pie chart demo
3. Consider different options for saving plots	Look into a solution which separates frequently used datasets from datasets that are not accessed as often	Look at past datasets for how often occurrences of newly added or changed previous data happens	Assist other team members with research options	Look into appending newly added data

Discussion of each planned task for Milestone 3:

- Task 1:

  This task will continue (and ideally complete) Feature 4.1. This feature was titled as “Customizable Operations on Variables” from the Requirements Document. This task will involve designing a frontend operation selection user interface that is based off the mock-up from the Design Document. This UI will allow users to select from a list of operations (drop-down menu?) and a list of variables and either plot the results or display the results in some meaningful way. For the backend, these selected operations and variables will need to be processed and stored properly in our database. We plan to utilize the SymPy library to process these operations.

- Task 2:

  This task will create a small GUI demo that integrates lockdown and mask mandate data. This demo will be similar to the male/female pie chart demo completed during Milestone 2. Any type of chart is acceptable. However, the demo should focus on various operations implemented in both the frontend and backend code and pay particular attention to the time required to carry out the queries involved in the demo’s operations.

- Task 3:

  This task will consider different options for saving plots that can be reused for later on and potentially modified. One option is to save computed results and only update the results(via an “append”-like operation) when a change is detected. It should also be investigated whether this is necessary or not. To do this, we will do some scalability tests, applying multiple operations on multiple plots.

Date(s) of meeting(s) with Client/Advisor (same) during Milestone 2:

- October 9th
- October 23rd

Client feedback on the current milestone: See Faculty Advisor Feedback below

Faculty Advisor feedback on each task for Milestone 2:

- Task 1:

  No comments or issues to date.

- Task 2:

  Dataset and *Datapoint* models should consider different options for saving and recalling data, operation results, and plots. The potential is there for the data to be modified after the fact and cause operation results and plots to be incorrect. However, this is infrequent at best. An alternative strategy may be to detect when changes have occurred and only reimport/update results when this happens. This would save time on importing and queries. This would also require the updating of previously saved plots.

- Task 3:

  Consider different options for saving plots:

  - Save all steps to create plot
    - Save dataset name
    - Save selected variable names
    - Save selected operations
    - Save plot types
  - Save the image of the plot
    - Plot can't be changed
  - Save data for plotting
    - Save results of operations
    - Disadvantage
      - Static data/plot
      - Large amounts of storage necessary
    - Advantage
      - Not having to rerun operations
  - Detecting/Recalculating data for infrequent dumps to older portions of the dataset
    - Better considers the amount of data we are starting to deal with

- Task 4:

  Task is still in progress. Comments will be more relevant at the 2-week checkpoint for Milestone 3.