Kevin Chau, Jose Moreno, Ify Aniefuna (Haram-Bae's)

BENG/CSE/BIMM 182

Professor Bafna

10 June 2017

Implementation of a Web-Based Genome Annotation Table

The goal of our project was to create a web server that is able to display the genome annotation results of the Project 1 teams. Project 1 teams used several different tools like Prosite, Pfam, and Blast in order to determine the various functions of a select number of different proteins from the *Acinetobacter baumanni* bacteria. Our web server was designed to programmatically take as input their genome annotation table files, along with optional raw annotation outputs, and create a fully searchable genome annotation table. The table allows users to easily find keywords for the many different genome annotations, either globally or by tool. We also implemented a download functionality that allows the user to download the entire table in CSV format. Furthermore, the table rows are clickable links that redirect, upon click, to the raw annotation file. The Project 1 teams we worked with were Triple A, JEM, Gas Pedal, JKT Inc. and w/e.

The backend of our web application was implemented with Django Python web framework. We chose this framework based on the recommendations provided to us as well as the easy-to-follow guides provided by the developers of Django. This web framework follows a model-view-controller format for application building, in which the information is organized into models, our representations of the data; the views constitute the presentation layer of the model,

or what the user actually sees; the controller controls the flow of information and interactions between the models and the views.

The first step toward the creation of our genome annotation web application was the creation of the information models that provide structure to the given Project 1 data. These annotation models were written as Python Annotation class objects and were created in correspondence with the possible annotation fields; these fields were “Sequence,” “BLAST,” “Pfam,” “Prosite,” “KEGG Pathway,” “NucPloc,” “Gene Ontology,” and “Comments. Next, the functionality of the web application was implemented. The data was first preprocessed with a user-defined data loading script. First, each genome annotation provided by each Project 1 team was read, either from CSV or TSV, into a Pandas DataFrame in memory. Next, the statistics calculations were performed; a dictionary of annotation tool keys mapped to hit count and total tool count was created. For each DataFrame, and for each tool, the number of hits was added to the total hit count and the total number of records was added to the total tool count; by doing this for each DataFrame, our calculations account for whether or not a Project 1 team used a particular tool. As an aside, not all Project 1 teams gave comments for their annotation files. These statistics were written to a static flat file for later reading by our Django functions.

Once the statistics were calculated, all DataFrames were concatenated into a single DataFrame, which was subsequently written out to a data CSV file. Since not all Project 1 teams provided Gene Ontology terms, which were used to standardize the vocabulary of the annotations, we implemented a term mapping script in order to populate that field. Dictionary mapping files, which map Uniprot, Pfam, and Prosite accession numbers (among others) to their corresponding Gene Ontology IDs. These IDs were then written to the final data CSV file.

The full data file was then converted to a JSON-format datafile. The Python subprocess module was used to call Django's 'loaddata' function, which allows for the conversion of a fixture (our JSON data file) into the specified models and loading of those models into the local SQLite3 database. This data loading script also has an optional "--clear" flag that the user may specify if the local Django database needs to be cleared. If this flag is not given, all data is appended to the preexisting table.

Several functions were implemented in order to manage and actually display the information now that the data has been loaded into the SQLite3 database. We wrote a "base" function that is called when the user navigates to the home page. The Django URL configuration file uses regular expressions to capture the extension of the current URL and calls the appropriate function; the "base" function is thus mapped to the "^$" empty string regular expression. In this function, all Annotation objects are loaded into memory, the statistics file is read, and all objects are rendered along with the corresponding "base" HTML file, which displays the information. Said information is loaded into the HTML table with a for-loop that loops through all of the Annotation objects. This HTML static table is then converted into a more feature-rich table using the DataTables jQuery plugin. Global table searching is implemented by default; field-specific searching was possible through writing a text-input box for each column writing a Javascript search function within the DataTables jQuery script.

The download functionality was made to allow the user to download all of the information presented on the genome annotation table. The download function was written such that it returns an HTTP response that serves the full genome annotation CSV file. This function was linked to the regular expression "^download$" in the URL configuration. Upon the user

clicking the “Download” button, which itself is written in the “base” HTML file with Javascript, the “download” string is appended to the current URL, which links to the “download” function as specified by the URL configuration file.

The clickable rows were implemented in order to display the raw alignment file for the specific protein in the row, given that such a raw annotation was supplied by the Project 1 team in charge of annotating that protein. Again, Javascript was written such that upon clicking a row, the protein sequence name is appended to the URL, which then points to the “raw_display” function, which in turn reads all annotation files into memory and dumps the text into a “raw” HTML file.

Once all data had been uploaded and was displayable to the user, some analysis was performed on the resulting data. From the calculated statistics, it would appear that BLAST and Pfam had the highest rate of annotation hits, at approximately 85% and 86% annotation rate, respectively. Prosite and KEGG Pathway both had less than 50% annotation rates. NucPloc yielded a 100% annotation rate. However, one should be wary of these statistics; although all teams used the BLAST, Pfam, and Prosite tools for protein annotation, not all teams used the same metrics for annotation retention. That is, each team used different criteria for whether or not a hit for a protein sequence was significant; these annotation statistics would be better supported if annotation pipelines were standardized with the same value thresholds. A possible reason for why BLAST and Pfam had the highest rate of annotation hits could be due to the way these tools search their databases. Blast uses local alignment and Pfam uses HMM’s while tools like Prosite and Kegg use manually entered motifs and profiles in addition to regular expresssion

consensus matching to provide functional annotation to a given sequence. The specificity that Prosite and Kegg use to find hits is a possible reason as why they have fewer annotation hits.

We had also mined the highest frequency Gene Ontology terms in order to better understand the average functionality of the given proteins (Figure 1).

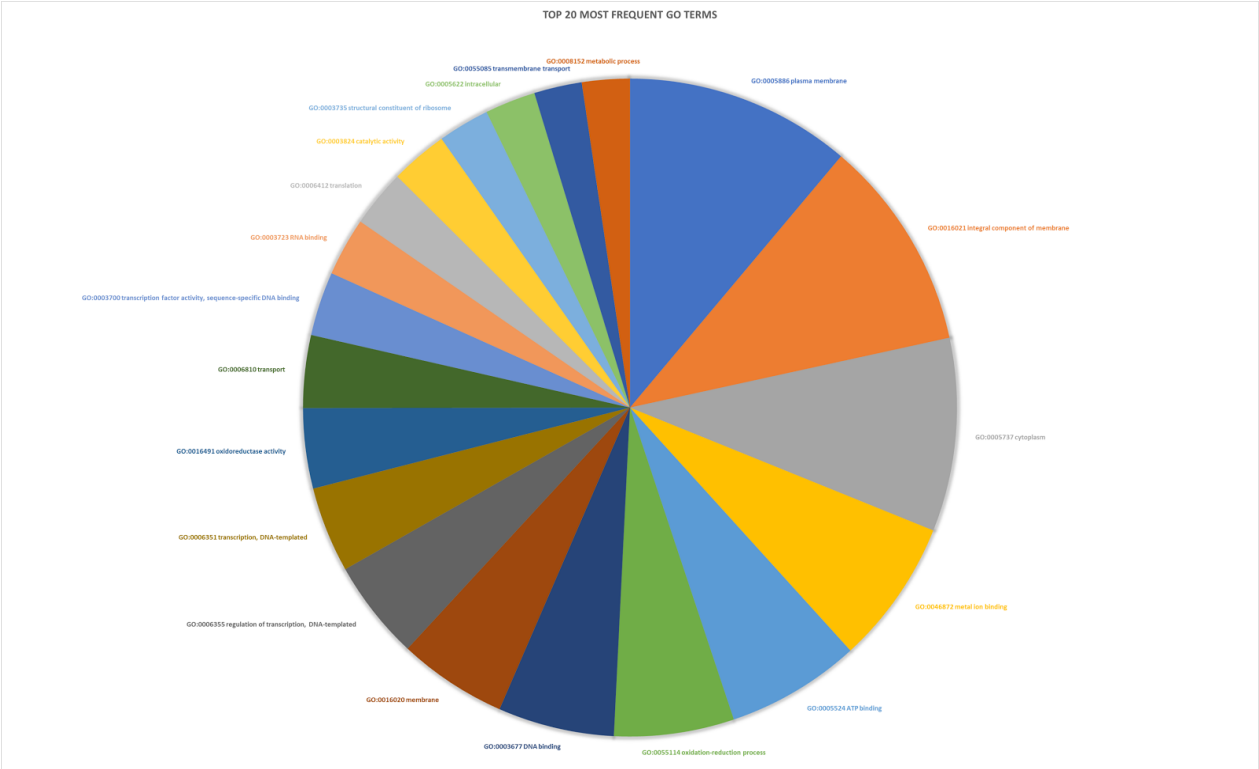

Figure 1. Pie chart showing the twenty most frequent Gene Ontology terms in the context of the 1450 annotations received from the Project 1 teams.

As expected, most of the identified protein domains correspond to some standard functionality, such as membrane structure, ATP binding, and transport.

All code, along with directions as to how to load data, can be found at the following GitHub page: https://github.com/kkchau/harambae