Josh Abel, Daniel DeFoe, Markelle Kelly, Samuel Verkruyse
CSC 466
6/13/2019

Project 3 Final Report

Data Sources

One of the primary sources for our database was the faculty directory. BeautifulSoup went over each of the entries and found the href links to the more detailed pages for each professor. These were stored in a set, and then we iterated over the set with BeautifulSoup in order to extract the additional information from these pages. Some pages did not have all of the fields, in which case we stored an empty string which the sustainer replaces with NULL.

The faculty paper submissions were important for the research aspect of the question. Each paper in the CS faculty Digital Commons website was scraped. Unfortunately, the website does not indicate the university a paper's authors are currently working at, so non-Cal Poly professors were temporarily included in the database and then removed by the sustainer.

Historical staff and their degrees were found from a GitHub wiki entry curated by Cal Poly professors. All tenure track historical professors were added to a table. Unfortunately, for some other professors, while it was clear they worked at Cal Poly, other information was too sparse to answer questions in a satisfactory manner and accordingly they were stored in an "other instructors" table. While our bot does not know much information about them, this table allows for it to recognize that these instructors exist and we just do not know much about them. Furthermore, these names were used for the canonical name table in order to ensure they were not removed from other tables.

The Cal Poly schedule page was particularly useful, as it allowed us to determine the location of any professor whenever they are teaching on campus. Furthermore, it gave us enough information to answer any course questions applicable to our topic, while leaving room for the actual courses group to do a more detailed analysis. However, it did lack each professor's office hours, which we had to obtain from the computer science office in the form of an Excel spreadsheet. This information was stored in a rather unstandardized format, so we found it easier to convert each entry into the form used by our Schedule table by manually curating insert statements, rather than attempting to create a script to do so.

The polyratings website was used to find overall ratings for instructors, as well as the average grades students received for each of their courses. At first, we also considered using the calpolyratings website because it includes newer professors, but due to a relatively small amount of entries we decided it was an unrepresentative data source and stuck to polyratings alone.

Finally, the CS Senior Projects page and the CPE Senior Projects page were used to find information about the senior projects each instructor has advised, including the titles and students involved. The CPE page was a particular challenge; while some CSSE professors have advised CPE projects, the vast majority of advisors are EE instructors. Again, the task of removing inapplicable professors was left to the sustainer.

Data Sustainer

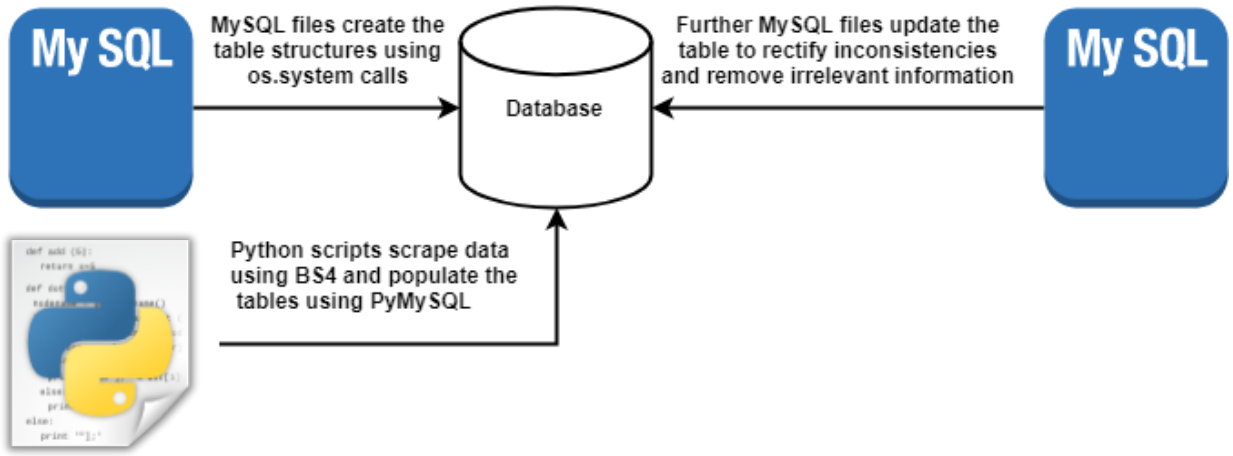
In order to implement the data sustainer, we first created scripts that gather all of the pertinent information from the applicable websites. After developing an understanding of the nature of our data and the kinds of questions we needed to answer, we designed the initial structure of the database and created a sql file to define the necessary tables. After BeautifulSoup extracts the relevant information, our data is stored in appropriate data structures before being inserted into their respective tables with a PyMySQL insert many call.

Several websites, especially those external to Cal Poly (for example, Digital Commons) appeared to have a rate limit when utilizing BeautifulSoup. In order to ensure that this would not be a problem during future use, we catch this error and attempt to repeat the scraping upon error. If an excessive amount of errors are made, a note is logged to a file and the rest of the code is run.

After filling all of the initial database, several problems were discovered. Some professors, especially Dekhtyar, were referred to by different names across different sources, including multiple entries within polyratings alone. CPE senior projects and journal publications included professors who do not teach computer science or do not even teach at Cal Poly. We created a python script to find all distinct combinations of first and last names across our database, yielding around 900 names. We constructed a table with standard forms of professor names and compared the 900 total names to those. Names that were close to a canonical name were adjusted with an update statement and the rest were dropped from the database.

Other minor issues were fixed with update statements. These included discrepancies between the spelling of university names, the formatting of degree titles, the formatting of building locations and converting time from AM/PM into 24 hour time. A sql file for this type of fix was created and built upon throughout development. Further additions included incorporating information from other groups' tables and writing scripts to add useful columns, such as class names.

Once all of these sql files and python scripts were finished, they were stored in a single file. This file uses the os.system interface of python in order to either call the python programs or to input the sql files into the database. In combination with a cron job, this allows for the database to be updated periodically, maintaining the most recent information possible.



General Strategy

Our overall approach is to first predict a question's general topic and then parse information relevant to that topic, based on the potential questions and answers we generated. This is done in preparation for the next step of the program, which involves assembling a valid SQL statement from the extracted information. To achieve this, we implemented three phases: first, extracting the criteria the user specified (conditions for the WHERE statement), second, detecting what information was being requested (variables designated in the SELECT statement), and finally, using those to determine the table to indicate in the FROM statement.

Topic prediction was performed with scikit learn's MLPClassifier neural network. We divided question types into 7 topics - ratings, research, jobs, contact information, history, clubs, and courses - based on general question intent. Although these topics do not line up perfectly with our tables (of which there are 12), they group together questions that we want to parse similarly. Focusing on key words and phrases as well as proper nouns from our database, we iterated on an ideal set of features, ending up with 91% accuracy on our collection of training questions.

Once a question is categorized, it follows a topic-specific path for parsing out relevant information. First, to extract conditions for the WHERE statement, the query is run through several functions designed to detect values of variables in our table. For example, a question determined to be about courses is run through functions that look for professor names, course numbers, course names, days of the week, times of day, quarters, and years. These functions internally generate segments of the WHERE clause and return a shortened version of the query with the variable values and any related words removed. The benefit of removing these words is in simplifying the next step - generation of the SELECT statement. Any variable names not connected to a variable value are considered requested variables and added to the select statement. Because of this strategy, we are able to disregard the word "research" and thus avoid responding with research interests in the question "What professors are researching

artificial intelligence?", but add research interests to the SELECT clause when "research" is in the question "What is Dekhtyar researching?". We took this a step further by adding lots of synonyms for column names; "get ahold of" is connected to the phone number and email columns, "how many" corresponds to COUNT(*) if detected by spaCy as the question phrase, and other question words like "who", "when", and "where" are translated into column names based on the predicted topic. Finally, we take into account the variables used in both SELECT and WHERE, as well as our predicted topic, to determine which table to request the information from and combine all our clauses into a SQL statement.

English sentence generation revolves around a relatively simple schema, where sentences were designed for the scenario that every single column in a table was returned. These sentences were then simplified in order to allow them to make as much sense as possible no matter how many or which columns are returned. While this is a bit rigid, making the bot feel slightly less "human," and does not necessarily match the exact structure of the query, we always return a coherent answer that is representative of our information.

Our model's primary goal is to predict what topic the proposed question is about, but it also aims to detect if a question is not appropriate to ask the computer science faculty bot. It will generally predict this when a question is asked with language that does not lend itself to any features the model searches for. However, not all questions which do not contain any of these linguistic features are questions for another bot. This is seen in a simple question such as "Where is Foaad Khosmood?". This is certainly a question for the computer science bot, but it may be classified as relating to a different bot because professor name is not a feature for separating topic types. To address this, our program double checks questions that are classified as being for another bot by looking for the presence of a professor name or another affinitive qualifier of the query as a question for the professor bot. When one of these qualifiers is present, the program again will then try to detect a few "mini features" to re-assign the predicted topic and send it down the appropriate pipeline.

Tools, Packages, and Libraries

Our program relies on several packages for model creation. With pandas and numpy for dataframe and matrix operations on our features, we use scikit-learn to create and train our topic prediction model and bot prediction model. Most significantly, we employed the "train_test_split", "MLPClassifier", and "cross_val_score" applications. Once we assessed our models with scikit-learn's metrics, we saved and exported them with pickle.

We also use PyMySQL to link python to SQL on the Cal Poly frank server. This package allowed us to set up the infrastructure of our database, insert our data, and make corrections and adjustments to improve consistency.

For natural language processing purposes, we relied on spaCy and nltk. Nltk's PorterStemmer function was useful for feature extraction. Our variable parsing functions also employ ntlk's synsets and spaCy's noun chunks to detect words and phrases that are too complex for regular expressions to handle. In simpler cases, we use the "Re" package for this

purpose. For our more specific variable types, names and times, we use “nameparser” from HumanName and the datetime package.

Testing and Performance

An important preliminary step in testing our program was ensuring the accuracy of our models. Based on the question structures we had written, as well as those posted by other groups, we generated a large set of possible queries. We used this for training and testing our model, ending up with an F1 score of about 0.81.

After our models had been trained and our program was working from start to finish, we improved the quality of our answers by simply asking as many questions as we could think of and taking note of ones our bot handled poorly. With this strategy, we were able to find synonyms we needed to check, variables we needed to expand parsing for, and English generation we needed to refine. We handled as many of these issues as we could, enhancing the performance of our model.

In terms of overall performance, although our bot is not perfect, it is quite robust to various wordings of questions because of our method of parsing queries. By choosing not to use distances from a predefined list of questions, we are generally able to handle questions with unnecessary phrases and even wordings we never thought of.

Weaknesses and Unsolved Problems

The program covered a wide variety of questions about Cal Poly computer science professors, but this does not mean that it is without room for improvement. One disadvantage of dividing questions by topic is that our bot does not handle compound questions gracefully. For example, a question like “What is Tim Kearns researching and when are his office hours?” is not currently possible to answer, since our program is designed to formulate queries for a single SQL table. The way we handle questions like this is to simply choose one topic and answer that part of the question, completely disregarding the rest of it. While one could argue that just returning Kearns’ research topics is better than nothing at all, it would make our bot much more practical if it handled slightly more complex cases like this one. A possible solution we discussed for this issue is simply dividing compound queries into separate questions before running them through our algorithm (in this example, “What is Tim Kearns researching?” and “When are Tim Kearns’ office hours?”).

One other shortcoming of our program is its topic prediction. We have a relatively high accuracy in our topic model, but when a question is classified incorrectly, the algorithm is quite unforgiving. Since we only parse variables related to the predicted topic, it is simply not possible to come back and get the correct answer if the query is sent down the incorrect pipeline. Furthermore, if a question is not intended for our bot, there is room for improvement in determining which other bot to recommend. One possible extension of the model would be to retrieve proper nouns, such as statistics professor names, from other databases.

Aside from failures and shortcomings, there are several next steps we would investigate if we were to continue working on this project. One important addition is the ability to match slightly misspelled qualifying words. If we implemented fuzzy matching, we would be more likely to accurately respond to the average user who may not be able to spell a professor's name perfectly. Additionally, the program would benefit from a session feature. Currently, our program is unable to ask clarifying questions or remember information about previously asked questions. By keeping track of a particular user, we could take into account more information when generating answers. If a user asked a question about "Foaad Khosmood", then we could retain this subject, so a subsequent question's reference to "he" or "his" would be recognized as referring to "Foaad Khosmood". To stay on the vein of communication, we also have room to grow in formulating fluent English responses. Currently, our answers are formulated in a naïve way that simply combines phrases, resulting in answers that are not convincingly human. By expanding the capacity of our English generation and extracting more information about the intent of the question, we could produce more specialized responses. Furthermore, the bot could make the chat feel more like a real conversation by considering information beyond the direct scope of the question. For example, if a user asks if a certain professor has published a paper on machine learning, instead of simply responding with "Yes, they have", the bot could explain, "Yes, they have published 2 papers on machine learning, (title 1) and (title 2)." By putting more consideration into what questions are actually asking and how people actually talk, the bot could become more advanced and user-friendly.