Makena Kong
Whitney Larsen
Pierre Lucas
Kevin Sanchez

CSC 466 Khosmood
Project 3 StaCIA

Report

Introduction

For Project 3, StaCIA, our group created a chat interface that answers questions about the Major, Minor and Curriculum of the Statistics Department. Our program and Slack bot are able to answer the questions you have in order to plan your schedule to graduate with a degree in Statistics at Cal Poly.

Data Source

Major

- http://catalog.calpoly.edu/collegesandprograms/collegeofsciencemathematics/statistics/bsstatistics/

Minor

- https://statistics.calpoly.edu/content/minors
- http://catalog.calpoly.edu/collegesandprograms/collegeofsciencemathematics/statistics/statisticsminor/

Data Sustainer

With the use of BeautifulSoup, we were able to scrape valuable information regarding Major, Minor and Curriculum of the Statistics Department. First, we had to find the web pages that would serve as data sources, which were determined after we wrote and categorized questions and what information will be required in the answers. Next, we had to inspect the web pages including their html files, so that we could determine if they contained necessary information and whether it was even plausible to scrape information. For example, we initially thought of scraping PolyFlows; however, the web page was not conducive to scraping as it required more than BeautifulSoup. Then, we had to analyze the page structure before writing functions that would methodically extract information from that single page. Although pages looked the same, they did not have the same html structure; therefore, the same function could not be used to scrape information from various pages. However, after writing a few functions, we were able to scrape necessary information.

Each time we gathered information from the web, we inserted our data into our MySQL database on Frank. Each time this program is executed, it drops all the tables in our

database, recreates them, and populates them with the necessary information. We understand that, in reality, the data sustainer would be running every day and it is not ideal to keep deleting and recreating our databases. However, for the sake of this project, we deemed it sufficient to recreate everything each time the program is executed.

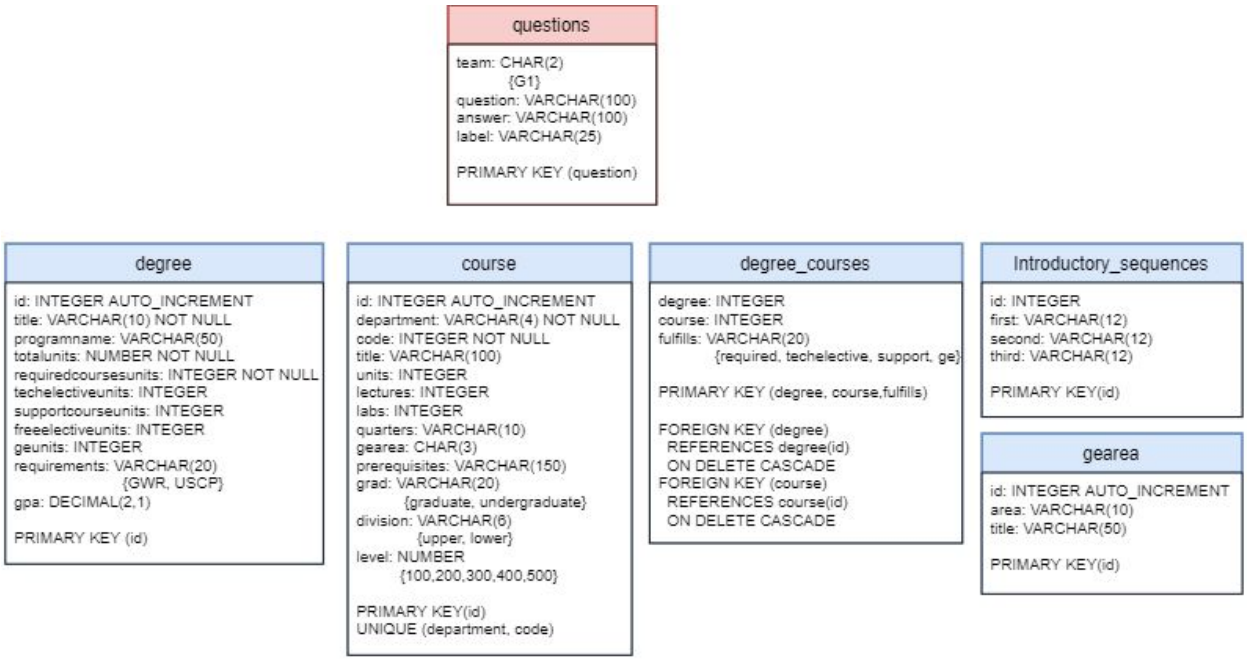

Figure 1: Database Schema

Query Parsing

We use regular expressions to parse the question for the variables and mark them. Each variable will be represented as a pair of the variable name and it's value. It is vaguely similar to NLTK's entity extraction tool, chunking. At first we considered using NLTK to parse the sentence but ultimately decided against it. There were 2 main reasons behind this. The first was that most of the variables we take in either have a specific format, like class codes, or are part of a small list of very specific phrases, like major/minor, graduate/undergraduate, etc. The second reason was that nltk relies on a corpus of the english language, which would not cover some of the phrases which are specific to college or specific to Cal Poly.

We wrote a parser which, using these regular expressions, recognizes the variables from the question and takes their values along with their corresponding variable type. For example, it recognizes "CSC 466" as referring to a course, so it stores "Course" and "CSC 466" in a tuple and adds it to the list of variables detected in the question. This list is then sent to the classifier. When the classifier returns the information retrieved from the database, that information and the user-input variables are formatted into a full sentence to answer the question.

Classifier

First, to design a coherent model, we grouped our questions by similar input and output and labeled each question with its group number.

Then, we designed a classification model based on multiple classifiers.

Data labellisation

Our classifier has been built to classify questions based on different criteria:

- The information asked by the question.
- The format of the answer.
- The number of parameters parsed in the question.

A class of questions has the same type of answer, but doesn't necessarily have the same variables within the answer.

For example, let's take the following question: "What are the [Department] courses for [Degree]?". In this question, the variable [Department] could be changed in the variable [Fulfills], that correspond to one value of the quadruple (GE, Required, Support, Tech Electives).

Moreover, whether it is [Department] or [Fulfills], the output would be the same: [Courses].

However, within a class, the questions need to have the same number of parameters. This has been constructed this way because of the SQL queries, written for each class.

Continuing with the same example, we couldn't classify the question"What are the [Level Number] level [Department] courses for [Degree]?", although it aimed for the same information.

Feature extraction

An important part of the classification is the feature extraction. It is hard to get features from questions because they can be either short, long and similar but not aiming for the same answer.

To extract the features from the questions, we followed the following method:

- First, we got rid of the stop words and everything that is not alphanumeric.
- Then, for each word, we stored up to 3 values:
    - His most common synonym.
    - His most common hyperword.
    - The word.

Below, you can see the features for the following sentence: "What courses do I have as a stat [Degree]?".

```
['courses', 'course', 'take', 'return', 'stat', 'degree',
 'degree', 'education', 'income', 'property']
```

Figure 2: Possible features

As shown, some words are repeated because the synonyms are identical to the word inputted. However, it won't be repeated in the final numerical vector. Indeed, we created a feature vector that includes all the features extracted from our questions. From this vector, we created a numerical vector for each question, composed of ones if the features are present in the question, and zeros if not.

The featured had to be vectorized this way as numerical values in order to be inputted in the classifiers provided by SciKit Learn.

Model

We tried different types of classifiers and parameters to finally decide to mix 5 classifiers: K-Nearest Neighbors, Decision Tree, Bagging classifier with K-Nearest Neighbors, Bagging classifier with Decision Tree, and Random Forest.

It can be computationally expensive to go through all of these classifiers. However, since the inputs are questions with a small number of features (and also since our data set is small), the delay for the answer was not noticable.

Each classifier would output one label, returning a total number of 5 labels. We would keep the label that has the most number of votes.

Training/Testing

For the testing our model, we choose to use 80% of the data as training and the remaining as testing. In the *figure 3* below, the accuracy of each classifier calculated on 20 different training sets is shown.

The best results were found for the following parameters of each classifier:

- K-NN: it works with 5 nearest neighbors and the euclidean distance.
- Decision Tree: it has a depth of 30.
- Bagging K-NN: based on the K-NN classifier described above, it is composed of 20 classifiers that are trained on 90% of the training data.
- Bagging Decision Tree: based on the Decision Tree classifier described above, it is composed of 20 classifiers that are trained on 90% of the training data.
- Random Forest: it is composed of 20 classifiers that are trained on 90% of the training data.

```
>>> classifier(5,2,20, 0.9,0.9,30)
Accuracy K-NN:  0.6354838709677742
Accuracy TREE:  0.6629032258064517
Accuracy Bag_KNN: 0.614516129032258
Accuracy Bag_Tree: 0.664516129032258
Accuracy RF:  0.6774193548387096
```

Figure 3: Accuracy from each classifier

The accuracy of the final model, which regroups the five classifiers, has not been computed. Nevertheless, while testing the outputs, this melted classifier had better results overall.

Label prediction

After parsing the inputted question and determining its variables, we use our trained model to predict which question group it belongs to.

Answer formatting

Answer selection

Once we have the duet [[Variable_Name], variable_value] — for instance [[Department], CSC]) — and the label of the question, we sat a set of rules to give the appropriate answer.

First, to get the content of the question, we wrote one SQL query per label, including variables, such as:

```
"SELECT c.department,c.code,c.title FROM course AS c JOIN degree_courses
AS dc ON c.id = dc.course JOIN degree AS d ON dc.degree = d.id WHERE
location1 = var1 AND location2 = var2 AND location3 = var3 ;"
```

In the query above, we can see 2 types of parameters.

- Location that include table and column name;
- The variable parsed.

The location of the variable in the database was deduced from the type of the variable (i.e. [Department] = course.department). Doing this allowed us to easily manage variables from different tables.

To be sure the SQL query would work, we implemented two rules:

- Did the parser extracted the right amount of variables, no more, no less?
- Are the feature types the one expected by the query?

To make these rule words, we wrote a SQL file containing 4 columns: the label, the query, the number of parameters expected, and the types of variables expected.

However, doing this implied a right match between the classifier and the parser, which was not always the case.

Answer replied

In the case arguments were missing, we outputted the following answer: "Could you be more precise? It seems that *x* arguments are missing.", *x* being the number of missing arguments (positive if too many, negative if not enough).

In the case the right amount of arguments was here but didn't match the one expected, the following answer was replied: "Could you reformulate the question?".

In the case the query SQL was accepted and launched, we finally had to insert the variables inside the answer and reply it to the user.

Tools, Packages, Libraries

- BeautifulSoup: We used this library to parse html and read the information needed to answer questions about our topic.
- Pymysql: We use this library to connect to, manipulate, and query from our MySQL database on frank.
- SlackClient: We used this library to create our bot for Slack.
- SciKit Learn: We used this library for its classification functions.

Testing and performance

Our program started at 50% accuracy with the K-Nearest Neighbors Classifier. We decided to try multiple classifiers at once for efficiency, so we also tried the Decision Tree Classifier, the Bagging Classifier (with KNN and DTree), and the Random Forest Classifier. We tested different parameter values, like n_estimators, distance metrics, weights, and such when they were applicable. In the end, we implemented our own Bagging Classifier with the aforementioned classifiers and with an average accuracy of 66%. More information can be found in the Model section of this paper.

Weaknesses/Problems

We should have used nltk rather than creating our own query parsing functions. There were tools in the library that had essentially the same functionality as the things we were trying to write.

Concerning the SQL query part, our set of rules to avoid inappropriate answers or wrong answer might have been to strict. An improvement could have been to create scalables SQL queries concerning the number of variables (which is currently fixed for each label).