CardGnome: Cards Suggestion Engine

CSM Field Session, 2012
Daria Tolmacheva, Mykal Cuin

Table of Contents

1. Introduction and Project Description 4
2. Requrements 4
   2.1 Functional
   2.2 Non-Functional
3. Detailed Design 5
   3.1 Architecture Design
   3.2 Fuzzy Logic Implementation Design
   3.3 Rules Table Design and Function
4. Implementation Details and Results 9
   4.1 Language and System Considerations
   4.2 Tool Usage
   4.3 Issues
5. Conclusions and Future Decisions 11
   5.1 Future Considerations
   5.2 Technical Lessons Learned

Table of Figures

Figure 1: Architecture Diagram 5
Figure 2: Database Schema 6
Figure 3**:** Fuzzy Logic Graph 8

1. Introduction and Project Description

CardGnome is a start up company founded in 2010. The company is a web-based seller of greeting cards that strives to provide excellent internet customer service to all the customers as well as the artists that provide products for the company. CardGnome works in a fast paced environment with only two individuals working on the technical aspect of the company's website and two individuals working on management and marketing side of the company.

Improvement of customer service lead to the idea of card suggestion function. This project has been a vision of the company since the very start of CardGnome. The card suggesting module would allow the client to reach out to each customer as well as making purchasing cards a faster and more satisfying process. The scope of this project includes creating a ruby function that would calculate the statistics of previous purchasing preferences of the users. Currently all the cards in the company's database are rated against different categories based on fuzzy logic. The project also requires creating the rules table in ruby on rails that would use fuzzy logic in order to suggest cards for the user. Later the module would be expended to also being able to suggest cards from preferences of facebook and twitter and user votes.

2. Requirements

2.1 Functional

- Create purchase_hist function that would query the database and calculate the percentages of different cards with different taxonomies purchased and would call return the array of those statistics
- Create a module function that would use the calculated statistics and based on them run the corresponding rules function that would return the list of suggested cards

2.2 Non-functional

- Create the rules table in the mysql database in ruby on rails that would store the list of rule functions
- Create the functions as ruby files
- Use mysql server to query the client's database

3. Detailed Design

3.1 Architecture

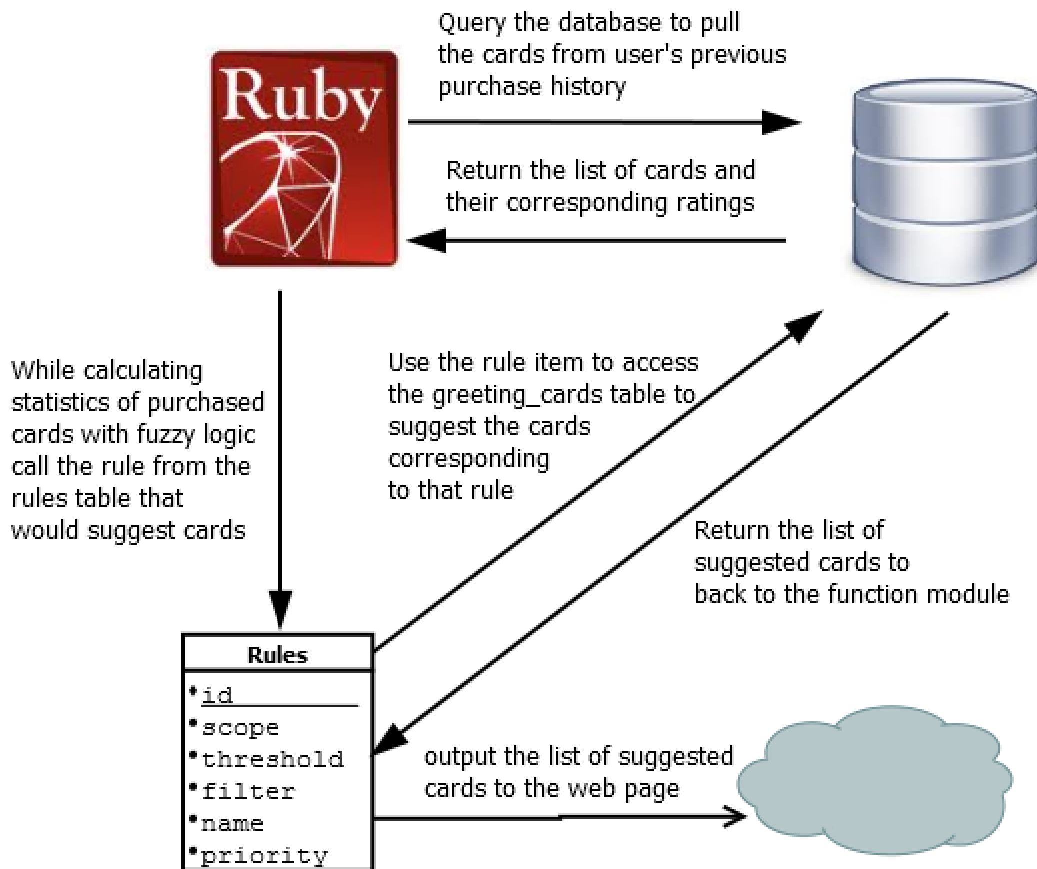

Figure 1: Architecture Diagram

(1)The architecture starts with the ruby on rails search_cards controller that calls the purchase_hist function that would take the inputs for username and the number of cards to suggest.

(2) Using the input the function purchase_hist would query the company's database on Mysql server to pull up the items from the Greeting_Card_Listing table and the corresponding ratings of ListingTaxonomiesAggregateRatings table.

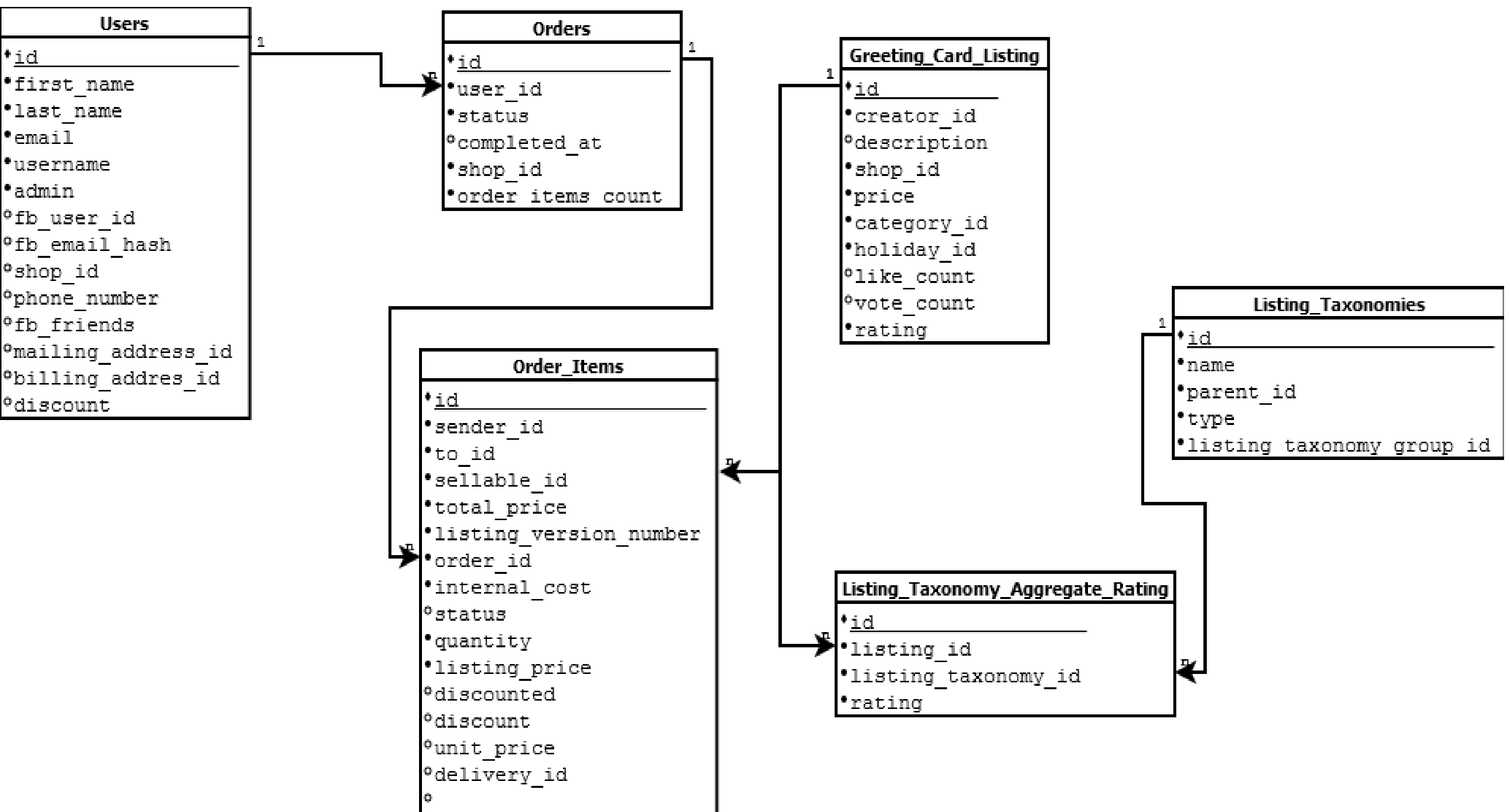

Figure 2: Database Schema

The function creates the following queries to access the data:

1. Access Users table to pull up the record with username equal to input username.
2. Access Orders table to pull all the records with the accessed user_id from the Users table.
3. Access OrderItems table to pull all the records with the accessed order_id's from the Orders table.
4. Access GreetingCardListing table to pull all the records with the accessed listing_id's from the OrderItems table.
5. Access ListingTaxonomyAggregateRatings table to pull all the records with the accessed id's from the GreetingCardListing table.
6. Access all the records from ListingTaxonomies table to have a list of all the taxonomy categories.

(3) The applied queries create result items that are returned from database to the function and stored in ruby variables. Then using those results the function uses fuzzy logic to calculate the averages of cards that are purchased for each rating. The function runs a loop for each card purchased with a loop within for each taxonomy id. Within the both loops the function populates the eval array with a boolean variable that represents whether the rating of the card is greater than or equal to the certain threshold for a specific taxonomy. Once the loop is finished the eval table stores the data for each card whether it can be qualified under a specific taxonomy category

for all taxonomies. Then using the eval array the function runs another loop that checks whether each card meets all, some, or only one taxonomy category by checking the number of true/false for each iteration. Based on the results the function can increment the counters for the number of cards that fall under a certain category of either fitting all taxonomies, some, or only one. An avg array is also created to store the average values of the cards that fit those three categories of either being all, some, or one taxonomy. The avg array is populated by taking the finished counter values and dividing them by the total number of cards that the user has purchased. Those values are then stored in the avg. array with corresponding labels of being either both, some, or a name of the only taxonomy that the card rating qualified.

(4) Using the avg statistics array the function would then call a eval_rule function with a specific id of the rule in the Rules table. The eval_rule would then query the database to search for the cards that match the requirements of a specific rule. In this project we implemented only several rules that search for cards that categorize under all the taxonomies, some taxonomies, or only one taxonomy.

(5) With the input from the rule's item data the database would populate the global array with the suggested cards.

(6) The global array $id, populated in the eval_rule, would then display its elements to the view of a controller in the web page suggesting id's of cards.

3.2 Fuzzy Logic Implementation Design

The fuzzy logic was used to suggest the cards with the taxonomy distribution based on the taxonomy distribution of the purchasing history of that user. The ListingTaxonomiesAggregateRatings table stores originally stores the cards listing_id that relates to the GreetingCardListing table ids and the average ratings for a specific taxonomy_id from the ListingTaxonomies table ids. These average ratings range between values of 0 to 10. There is also a certain threshold value which determines if the card can qualify under a certain taxonomy category.

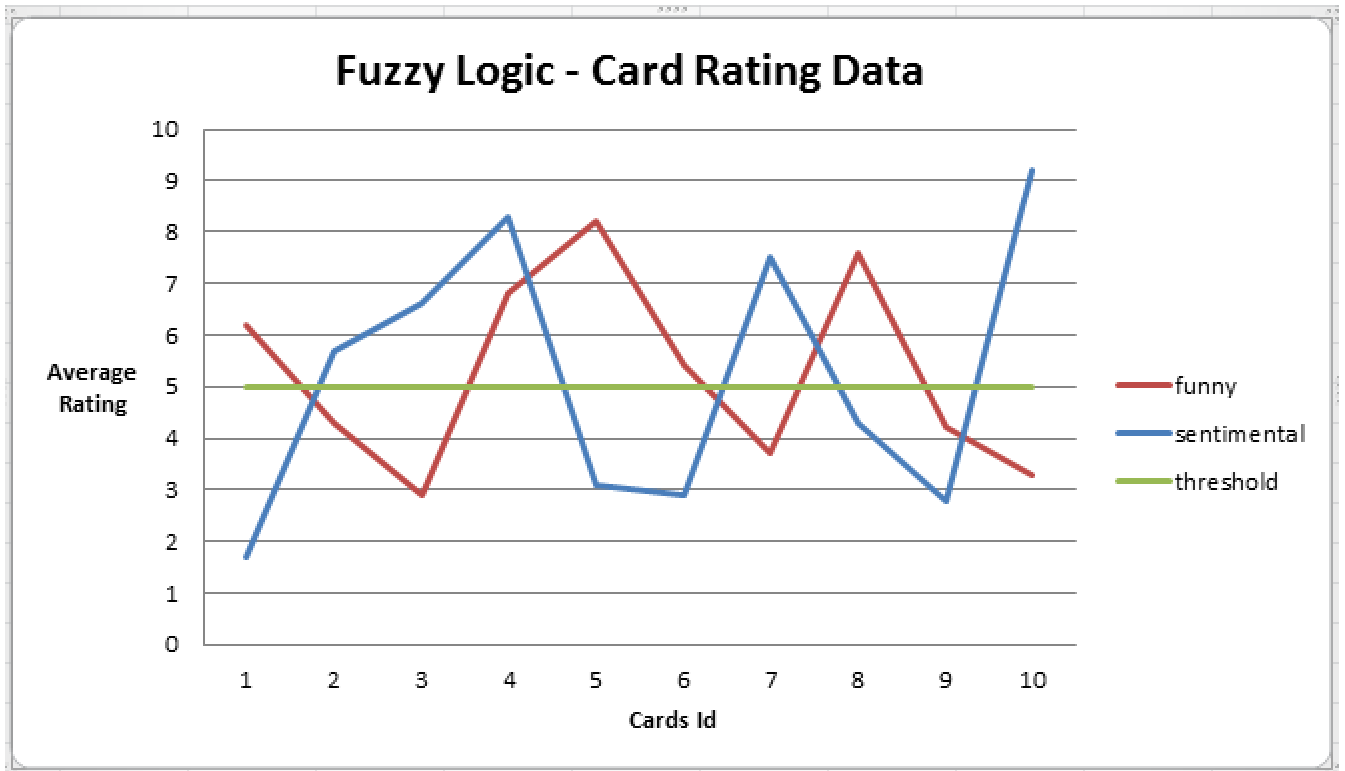

Figure 3: Fuzzy Logic Graph

Considering the Figure3 graph for the sample data the following cards with ids 1-10 have the following ratings for taxonomies of "funny" and "sentimental". The threshold is set at 5. Therefore we can determine which cards fit both taxonomies (eg. 5) and which are only "funny" or only "sentimental". From this data we can determine the average sets of the cards that fit both taxonomies or only "funny" or only "sentimental". Determining those averages (eg. percentages) we can suggest cards that contain the same percentages that fit the same sets of taxonomies. For example, if 30% of the user's purchasing history were the cards that were only "funny" then if we would suggest 10 cards to the user, 3 of those cards would belong to the only "funny" set.

3.3 Rules Table Design and Function

The client requested us to build a table of rules that would hold all of the rules and the fields that the rules required. The fields of the table are id, scope, threshold, filter, name, and priority. The client also requested a function to use the rules table, that would look through the list of all of the greeting cards currently in the the GreetingCardListing database, and use a "funny", "sentimental", or any type of rule that would be needed to use to sort the suggestions cards. The rule that would be used would be based on the id given to the eval_rule function, and how many cards the rule can return would be based on the past_purchase output. Each rule would have a threshold number, most likely a decimal number from one to ten, to compare against the current rating of the card in each category. The rule method would then take in data from the Rules table and ListingTaxonomyAggregateRating table to use for comparisons. The Rules table would provide the threshold, the id of the rule, and the scope of the rule. The ListingTaxonomyAggregateRating table would provide the actual rating of the card when

compared to the card's listing_id number and place that data into a variable called rating. The eval_rule method then uses a loop to check through all of the cards in the database and compare the rating of the card to the threshold of the rule that the card is under ,such as the threshold of id 1, the funny rule, being 4.2. The card will need to be rated 4.2 or higher in funny for the card to pass. If the card the does not meet the set threshold of the rule the card will be rejected and the listing_id will not be presented to the web page, and the client. If the rating of the card does pass the set threshold of the rule the card's listing_id will be returned to the web page, and viewable for the client.

Then next step of the rules table was to integrate it into the post purchase history search function. The rules table would take the output of the search function, which is the different spread of card taxonomies such as .333 being funny, .333 being sentimental. This distribution is used to determine how many cards each rule will be able to print out of a number given by the user. So if the user wants 10 cards printed, three of them will be funny, and three of them will be sentimental. The other 4 will be chosen from other rule distributions.

4. Implementation Details and Results

4.1 Language and System Considerations

Language/System	Why?
Ruby on Rails	The client’s language of choice and what all of their code and databases are already implemented on.
HTML	Used in testing database returns.
Ubuntu	The clients are avid Mac users, and are using gems that require a UNIX base.

4.2 Tool Usage

MySQL Viewer	Allowed the view of all the database structure and also being able to add sample data to the database
MySQL Administrator	Allowed any modifications of the database tables
RVM	Used in changing ruby versions

4.3 Issues

We had four major technical issues pop up throughout the coding process. A comparison issue, an active relation issues, troubles with trying to create a test GUI, and finally some design issues with functions in the controllers.

The first major issue popped up in the eval_rule function. We need the rating field of the ListingTaxonomyAggregateRating database and the threshold field of the Rule database to be able to compare, and check if a card will make it through on a certain rule or not. The issue was mainly that the rating field was not returned correctly, and that Ruby has an issue with comparing Symbols and floats. After trying many different options such as turning both of them into strings, we finally used the .sum option in Ruby to just make the rating add against itself, and would now return it as a float instead of a symbol. This allowed the comparison go through.

Another issue was when trying to test if the database was pulling from the fields needed. It would only return a an Active:Relation object, and would not show what it had pulled from the database. We figured out that by adding .inspect to the end of the pulling code it would transform the Active:Relation into a string of what was pulled from the database.

The third major issue was while trying to make a GUI for a demo. We had an html page that used a form to accept a username and the number of cards that the user wanted presented to them. The form would then link to an output page that would output the suggested cards, and a taxonomy distribution for the user. We ran into the problem that the form would not return the data on the output page or into the search_cards function. The data would pop up in the URL though. We still couldn't find a workaround for this one.

The final issue was with the eval_rule. The search controller needed to call the eval_rule which was in the rules controller. We tried making the search controller require the rules controller, and that didn't work. We tried forcing rules_eval into its own module and connecting the module to the search controller, but that didn't work either. Finally the only thing we could figure out to do was take the eval_rule and place it inside the search controller. It did work from there because it was now local.

Many issues popped up, but we were able to overcome most of them in a rather simple manner.

5. Conclusions and Future Decisions

5.1 Future Considerations

With all of the errors and bugs that popped up with us as well as some of the faults in communication we were unable to test our code against their production code. Therefore the company will need to test the engine in the future. In the future the company can improve the code by adding more rules and adding a more efficient function to implement those rules to suggest cards. Also the connection to facebook and twitter can be implemented to work with the rules table and the purchase history function.

5.2 Technical Lessons Learned

We learned that Ruby has to be tricked into pulling actual numbers from the databases such as the issues with the rating and needing to force the rating from a symbol into a float variable. We also learned that Ruby likes to more work in the realm of database relations instead of giving variables actual values out of the database such as needing the .inspect ending to make

them a string instead of a relation. Another technical lesson learned is that Ruby on Rails has a very steep learning curve, and doesn't take very well to being changed outside of how it wants to be changed. Finally, never push to a Github master branch without making sure you know exactly what it does. It's possible that it just might be the production code master branch.