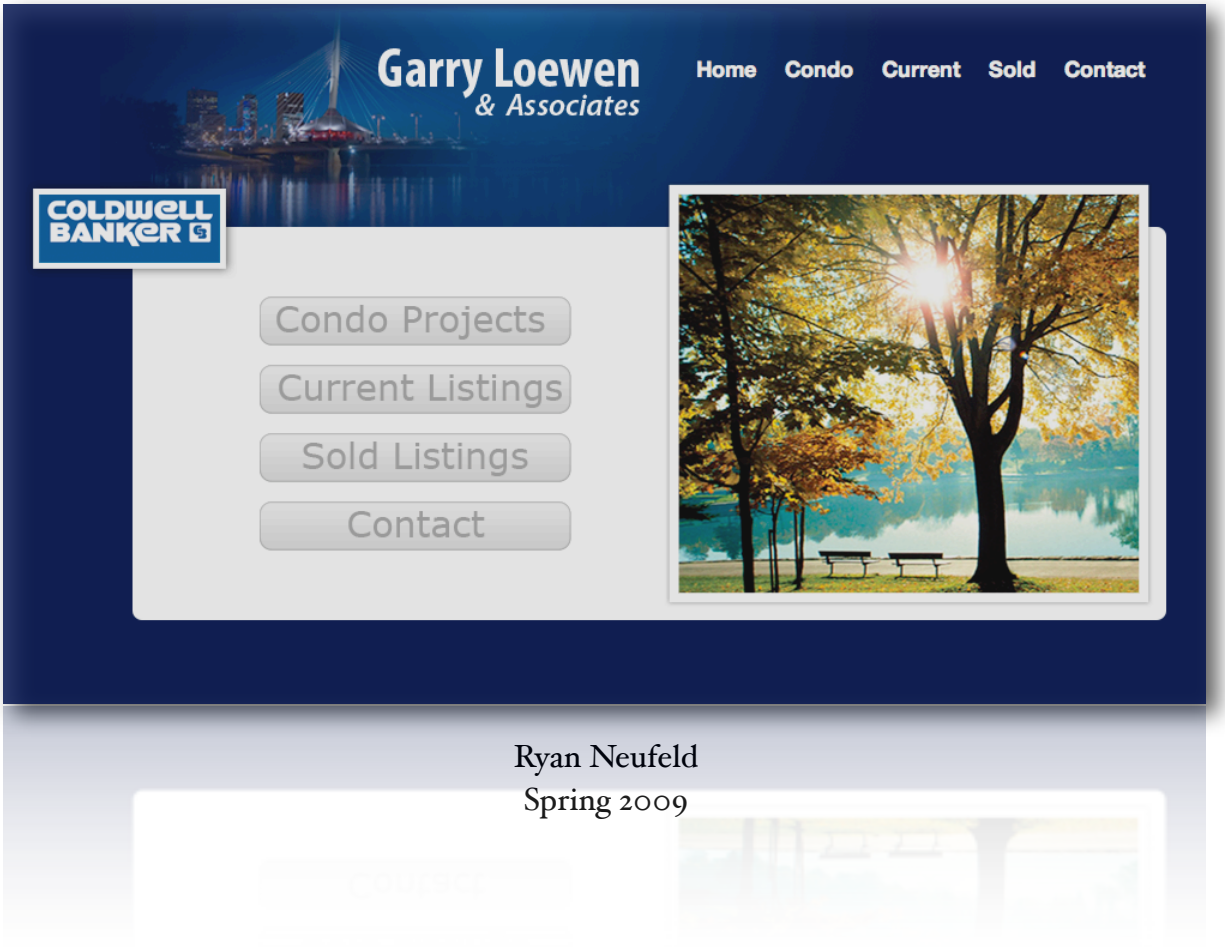
FUZZY-REALTY

Implementing a web-based fuzzy-search expert system



Ryan Neufeld

Spring 2009

Expert Systems Report

FUZZY-REALTY

Implementing a web-based fuzzy-search expert system

Ryan Neufeld
Spring 2009

Outline

For Stefan Penner's and my Expert Systems (ES) project we set out to create a website that utilized some sort of ES to complete a novel task. We ultimately decided to implement a Real Estate website that used a Fuzzy ES to assist in searching for realty listings that most closely matched the customer's parameters.

At the most high level a customer inputs what they want then and then the website presents the listings that most closely match those specifications. Digging deeper the application takes the customers parameters and uses them to compute a score for each listing which it then uses to present appropriate listings to the user.

Switching Architectures

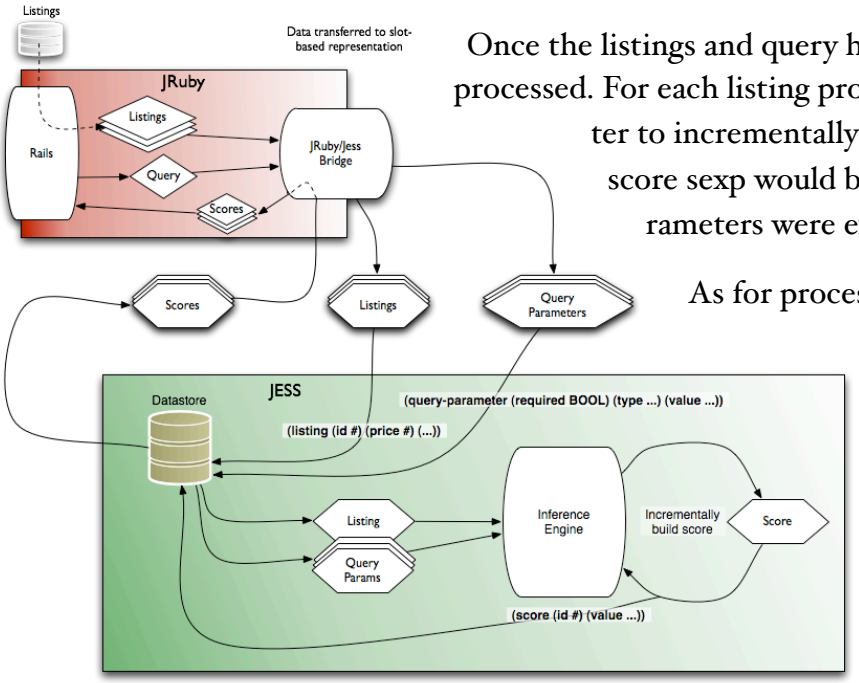
At conception the project was going to be implemented using a trifecta of languages – Ruby, Java, and JESS – tied together by JRuby. As the project matured and stronger notions were made as to its functionality it was decided to create a highly specialized ES that lived purely in Ruby (more specially its syntax; the project can still run in Java through JRuby). In our opinions the JESS ES we were aiming at was far to imperative, and consequently very obtrusive. Thus Ruby.

I'll quickly present for consideration the original and current architecture to hopefully shed more light on this decision.

The Original Architecture

In its original capacity the web application, running on the rails stack, would receive user input on traits it desired in a listing and form a Query object with that information. Each Query exists as a collection of Parameter objects that specify the type of parameter, whether or not it is required, and the value of the parameter.

Once the query was well-formed it would be forwarded, along with the contents of the Listings database, to the JRuby/JESS bridge, whose job it was to convert JRuby objects into appropriate JESS S-expressions (sexp).



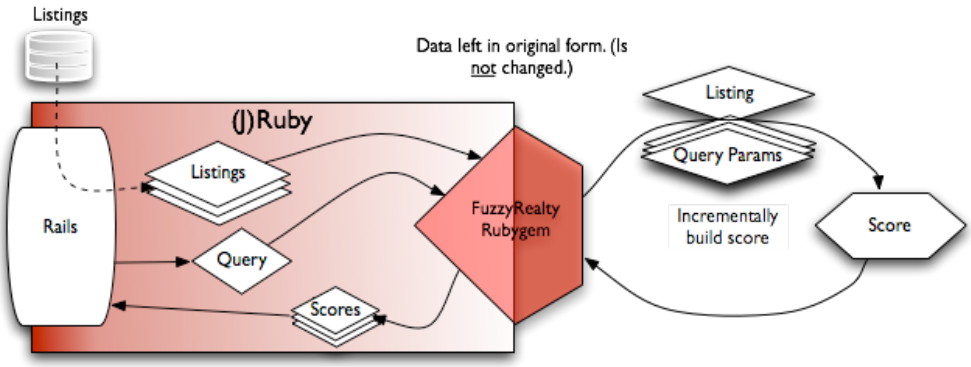
Once the listings and query had entered the ES they would then be processed. For each listing provided the ES would use each parameter to incrementally build a score for that listing. The score sexp would be added or subtracted to until all parameters were exhausted.

As for processing parameters the ES would fire the appropriate rule or function with the parameters value and the listing's value for that type to generate a basic score value. This basic score would then be multiplied by that type's weight (and possibly further modified if a said parameter was indicated as required).

Once all listings have been processed the ES returns control to the JRuby/JESS Bridge, which pulls score information from the Score sexp and attaches it to its Listing object (for the current query).

The New Architecture

The new architecture differs from the original in that it is implemented exclusively in Ruby. As a corollary the ES itself implements only the required subset of ES functionality that it requires. Seeing that Ruby is a dynamic imperative language it has little trouble emulating a highly specific rule-base while still providing the imperative flow of control the original architecture attempts to emulate.



At the highest level the ES is packaged into a Rubygem package called *FuzzyRealty* and need only be imported to be used. Being written in Ruby the new ES requires no conversion to and from sexp as was required in the original ES.

The ES quickly iterates over each listing and parameter to computer the score, as before. Unlike the JESS ES, however, the Ruby ES doesn't use an inference engine to do this work.

Since the ES is using a subset of ES functionality it emulates the actions of the intended JESS ES and dynamically calls the appropriate method to calculate scores.

– More information in the following section –

The Expert System

Fuzzy_Realty.rb

The primary point of entry into the ES, after importing it, is the `FuzzyRealty::ExpertSystem#scores/2` method that takes an array of `Listings` and a singular `Query` object. For each listing this method iterates through each parameter in the query to calculate the listings score. Like the original architecture parameters are made up of three attributes: type, value and a required flag. The `#scores` method uses a hash of parameter types to anonymous functions as its rule-base to call off of. After the appropriate function is selected it is called with the parameter's value and the listing. This call returns a score from zero to one indicating the closeness of the match (one being perfect). If the parameter was required and the score is poor the score is further reduced. After this the listing and score are added to an array of scores to be returned on conclusion.

Rulebase.rb

As mentioned the rule-base is constituted by a hash of parameter types and their respective computation functions. In our opinions this fairly closely emulates how the JESS system might have behaved. Having a hash like we do makes it simple to add additional business logic, while keeping these details separate from the application logic. Changes to the rule-base need only be supported by an additional weight for the particular type to be fully implemented and type-checked by the remainder of the system.

Weights.rb

The hash of types and numeric multipliers for weight is probably the smallest file in the project, but likely has the biggest influence on the system. We consulted with Garry Loewan on these values. I found it particularly difficult to pick good numbers for the weights. I can imagine determining near-perfect numbers would be a task fitting for a masters thesis.

Classes.rb

The classes file provides implementations for the `Query` and `Parameter` classes. In addition it provides a stub class `Listing` for testing without the Ruby on Rails model present. The `Query` class acts as a simple wrapper for collecting `Parameters`; it checks only that what it

receives is actually a `Parameter`. `Parameter`, as mentioned earlier, is simple type, value, and a required flag. `Parameter#initialize` checks that any type given exists in the weights table.

SCORES_TABLE.RB

The scores table file consists of two hashes used to represent a matrix of scores. What we decided when implementing the "style" and "location" types is that we would use a subset of reality. We laid out the values for each type and arbitrarily decided what score each relationship should yield. For example, a person seeking to live in location 'A', an upper-class suburb, will not want to live in location 'D', the slums, hence we assign it a score of zero. These tables degrade nicely in that they provide default response of zero, meaning no match.

Performance

SUBJECTIVE

Looking at the results for 100 random listings it is plain to see we may have stumbled upon some decent weights and score functions. Obviously there is a lot of time to be spent and things to be said on optimizing the performance of the ES, but we are in the ballpark. Presented below is a printout

```
Running "fuzzy_realty.rb"...                    ruby 1.8.7 (2008-08-11 patchlevel 72) [i686-darwin9.6.1]

Query 1, $250k Condominium in the prestiguous 'A' suburbs. 1575 sq. ft.
Top 10 Listings:
(1)     62.60   #<FuzzyRealty::Listing:0x18e430 @style="Bungalow", @location="A", @sqft=1768, @price=250401>
(2)     58.85   #<FuzzyRealty::Listing:0x1932f0 @style="Condominium", @location="A", @sqft=2036, @price=130519>
(3)     58.70   #<FuzzyRealty::Listing:0x191bf8 @style="Bungalow", @location="A", @sqft=2156, @price=179893>
(4)     55.09   #<FuzzyRealty::Listing:0x18f2f4 @style="Condominium", @location="A", @sqft=2101, @price=109127>
(5)     45.42   #<FuzzyRealty::Listing:0x18ac68 @style="Condominium", @location="B", @sqft=2298, @price=94940>
(6)     39.25   #<FuzzyRealty::Listing:0x18a77c @style="Condominium", @location="B", @sqft=1401, @price=260234>
(7)     38.00   #<FuzzyRealty::Listing:0x18e5d4 @style="Condominium", @location="A", @sqft=1252, @price=247653>
(8)     35.37   #<FuzzyRealty::Listing:0x18efac @style="Condominium", @location="B", @sqft=1488, @price=74891>
(9)     32.60   #<FuzzyRealty::Listing:0x192e04 @style="Bungalow", @location="A", @sqft=552, @price=254311>
(10)    32.00   #<FuzzyRealty::Listing:0x18dbfc @style="Split-level", @location="A", @sqft=2068, @price=233105>
```

Whilst the listings were random, we do see a fair spread across the scores. For the most part a higher score does seem to indicate a better match for the query. Sitting down and spending some time with the Real Estate agent could likely improve the weights and score functions to a manageable level. This would likely be done using a set of listings and some defined queries, having the agent judge where certain scores are lacking. We would take this information and try to correlate positive or negative matches with certain weights or score functions.

Technical

Benchmarks of the existing ES have proven to be quite promising. Calculating the score for 100,000 listings is only ~5 seconds, and 10,000 listings takes less than a second. Here are the results of the benchmark:

Benchmarking search through random listings				
	user	system	total	real
100,000 listings	4.670000	0.020000	4.690000	(4.721698)
10,000 listings	0.550000	0.000000	0.550000	(0.562986)
1,000 listings	0.030000	0.000000	0.030000	(0.026346)
100 listings	0.000000	0.000000	0.000000	(0.002347)
10 listings	0.000000	0.000000	0.000000	(0.000285)
1 listings	0.000000	0.000000	0.000000	(0.000108)

While promising it is important to note that this does not show time spent loading listings (as it should in a production application). I can say, however; depending on the performance of the database that this ES could be used in a production system with tweaking to the results.

I also took it upon myself to profile the ES. I won't present that data here, but it is available in `fuzzy-realty/docs/profile.html`. The biggest observation to come out of the profile is that, for the most part, method calls are primitive. This is to say we aren't spending a great deal of time in the system itself, but in language level calls. Not a frightening observation, but it indicates to me that the library itself is not overly complex.

Here are a few statistics and observations from the profile:

- 35% on calling items in the rulebase (and subsequent calls).
- Of the 35% in the rulebase the majority of time is spent on hash access (15%)
- Of the sampled profile, a large amount of time is spent sorting. Something to be noted for the web-application developer
- Aside from hash access the next big use of language features is float comparison (10%) and other operations

Conclusion

While it was a big risk I don't regret choosing to use Ruby to implement the ES. It both taught me new paradigms of use for Ruby, as well as more about how JESS works. I think my strong footing in Ruby helped me to get more out of this experience than I could imagine my less stable footing in JESS could have.

The project itself also has promise. It's performance, both technically and subjectively, is within acceptable bounds. I imagine some more time with an expert could do a lot for the project. All-in-all I think this was a good experience; I likely take more away from this than I did Software Engineering.

– Any further questions can be forwarded to rkneufeld@gmail.com –