Introducing the IBM Watson Retrieve and Rank Service

An exciting recent development for Watson was the release of the Retrieve and Rank Service on Watson Developer Cloud. As described in the official documentation, the Retrieve and Rank (R&R) Service "helps users find the most relevant information for their query by using a combination of search and machine learning algorithms to detect 'signals' in the data. Built on top of Apache Solr, developers load their data ["corpus"] into the service, train a machine learning model based on known relevant results, then leverage this model to provide improved results to their end users based on their question or query."

When an end user asks a question of the R&R Service, the service first executes a carefully constructed Solr query that scours the corpus looking for relevant results. For each retrieved result, it then computes a number of numerical "feature scores" that assess the linguistic similarity between the question and the result. Finally, a learning to rank machine learning algorithm uses its previously trained model to transform the feature scores into a final relevance score for each result. This three step question-answering approach (search -> scoring -> ranking) is extremely powerful; the original Watson supercomputer that beat top-ranked human challengers on Jeopardy! used a generalization of this approach.

The original Watson Jeopardy! system leveraged a novel question-answering framework known as DeepQA. This framework treats the question-answering process as a pipeline consisting of a series of distinct steps. First, the question is analyzed to determine what is being asked. Next, candidate answers are generated by searching the corpus for relevant documents. Subsequently, supporting evidence is retrieved, and answers are scored based on a number of deep and shallow linguistic features. Finally, machine learning algorithms combine the various numerical answer scores to produce an overall confidence for each candidate answer.

The DeepQA framework is powerful for several reasons. First, it supports massive parallelism, allowing many answers to be considered simultaneously. Second, the framework is probabilistic in nature -- no component needs to firmly commit to an answer, but can instead output features or scores that reflect a degree of confidence in the answer. Finally, and most importantly, the framework is highly flexible, allowing the developer to integrate a wide range of components to customize the pipeline using logic that is specific to a particular use-case, domain, or application.

With this background in mind, we wanted to build an application that would show developers how they could embed the R&R service into the DeepQA architecture to build powerful, custom-tailored QA systems. This idea ultimately would become the R&R application that can be found here.

Since our goal was to make our entire application open-source, we needed to find a corpus that could be made publicly available. Fortunately, we found a tremendous resource in the Stack Exchange network: a family of sites with interesting content, rich metadata, tremendous volume, a Creative Commons license, and natural language questions that are long-tail in nature -- perfect for a question-answering system that would leverage the R&R service. After experimenting with several sites, we decided on the English Language and Usage site, inviting end users to ask our app questions about English.

With a corpus chosen, it was time to start building and benchmarking our system. As a baseline for performance, we used a very simple DeepQA pipeline that consisted solely of a primary search/candidate answer generation phase. The answers were generated by searching a local Lucene search index, with Lucene's score used to provide a rank-ordering of the search results.

This local Lucene ordering provided a useful performance baseline for our experiment. Since the R&R service is based on Apache Solr and also uses Lucene as its search library, we could incrementally bring in additional portions of the R&R solution to determine if they increased the accuracy of our search. In an effort to create a useful comparison, we started our exploration of the R&R service by tailoring its Solr configuration to match our local Lucene setup as closely as possible. Unfortunately, many of the advanced query mechanisms we had written with our specific corpus in mind were not possible to port directly to the R&R service. This resulted in a slight decrease in accuracy when comparing our local Lucene results with those of the "retrieve" portion of the R&R service.

It should be noted that the previous phase of our experiment was simply to: a) provide a rough comparison with our offline results and b) benchmark the usefulness of the "rank" portion of the R&R service. In practice, R&R's search capabilities are intended to be used alongside a ranker; after performing searches, R&R scores the potential results so that they may be compared later in the pipeline. While the number of scorers is subject to change with different releases of the R&R service, in our specific experiment 32 out-of-the-box linguistic scores were assigned to each of our candidate answers.

When creating the ranker, a training phase is required that includes giving the service a series of example questions, along with their known answers. A sophisticated machine learning algorithm uses the known answers to determine the optimal combination of feature scores that will best predict the relevancy of answers at runtime. At the end of our training phase, the ranker had detailed knowledge as to which scorers it should value in helping users find answers to their runtime queries.

In our experiment, the combined use of the "retrieve" and "rank" portions of the R&R service led to a massive boost in accuracy over our initial local Lucene results. Specifically, a 21% improvement was observed simply by training the ranker with the out-of-the-box scorers.

The previously described experiment solely leveraged R&R's linguistic similarity scores between the questions and answers. No knowledge of the underlying corpus was used to score the returned results. We hypothesized that by using knowledge unique to our Stack Exchange corpus, it would be possible to obtain even better results.

This hypothesis led us to switch to using a more advanced use-case of the R&R service. Instead of simply making a single REST call to have R&R search and return results, we now made an initial call to the "retrieve" portion of the R&R service to obtain candidate answers. Our application's code then scored each answer using customized answer scorers that are unique to our use-case. These scorers made use of the detailed metadata accompanying a Stack Exchange thread. In total, six scorers were written. The scorers took into account things like the number of upvotes of an accepted answer, the reputation of an answer’s author, and the number of page views of a thread -- all information specific to our corpus that the R&R service would be unable to account for by itself. Once the answers were scored, we sent them to the "rank" portion of the R&R service so that our custom answer scores could be considered during the final ranking of answers.

The resulting accuracy improvements were considerable. A 41% improvement in accuracy was observed over using the native capabilities of the R&R service. Overall, a sizeable 71% accuracy improvement was achieved when the results of using the R&R service with custom scorers were compared with the initial local Lucene approach. At this point, we made one final tweak to our configuration. Postulating that the scorers would work best when given additional candidate answers to score, we doubled the number of answers we requested from the "retrieve" portion of R&R from 50 to 100. Our hunch that these additional answers would improve our recall, and subsequently, our accuracy, proved to be right. An additional improvement of 9.3% was observed over the previous configuration for a final total of 87% improvement over the initial baseline Lucene results.

While we were pleased with our final results, raw results are not the only mechanism for determining the success of a project. Ease of, and time taken for, deployment should also be considered. To improve both of these factors we looked to the DeepQA framework. This framework allowed for a well defined path for asking a question and obtaining a set of answers. During each phase of this project, from issuing the query, to custom scoring the answers, to obtaining final ranked results, the DeepQA framework provided classes and methods that the user can extend to greatly decrease the time to project completion. Better yet, the DeepQA framework is flexible enough to be used not only with the R&R service, but with a variety of different algorithms and services, as mentioned earlier.