RC25356 (WAT1302-021) February 15, 2013
Computer Science

IBM Research Report

Tools and Methods for Building Watson

Eric Brown, Eddie Epstein, J. William Murdock, Tong-Haing Fin
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Tools and Methods for Building Watson

Authors: Eric Brown, Eddie Epstein, J. William Murdock, Tong-Haing Fin

Abstract

The DeepQA team built the Watson QA system for Jeopardy! in under four years by adopting a metrics-driven research and development methodology. This methodology relies heavily on disciplined integration of new and improved components, extensive experimentation at the component and end-to-end system level, and informative error analysis. To support this methodology, we adopted a formal protocol for integrating components into the overall system and running end-to-end integration tests, assembled a powerful computing environment for running a large volume of high-throughput experiments, and built several tools for deploying experiments and evaluating results. We describe our software development and integration protocol, the DeepQA computing environment for development, and the tools we built and used to create Watson for Jeopardy!. We also briefly allude to some of the more recent enhancements to these tools and methods as we extend Watson to operate in commercial applications.

Introduction

The Watson question answering system was created in less than four years by approximately two dozen researchers using a highly empirical metrics-driven methodology. In this methodology many ideas for finding and evaluating answers to questions were considered but only the most promising ideas were fully implemented and integrated into the final system. This approach was critical for guiding investment of limited resources and making rapid progress. To support this research and development methodology, the team required several protocols, resources, and tools.

One of the most important protocols the team adopted was a formal procedure for periodic system integration and testing. System integration and testing is critical for tracking overall end-to-end system performance and ensuring that the ongoing component level research and development activities are contributing to consistently improving performance at the overall system level. A similarly important protocol is tracking code changes and versioning components and system releases. Not only is this a common sense software engineering practice, but it is essential to support a metrics-driven development approach where experimental comparison of different component and system versions is required to track progress and guide future research and development investment. The DeepQA team addressed these aspects of the project by adopting source code versioning and tracking protocols and, in particular, a regular

system release and test process where all of the system components are tagged and a periodic system integration test, dubbed the "weekly run," was conducted to evaluate the system on a suite of test sets.

With appropriate protocols in place, the next key to metrics-driven research and development is the ability to rapidly evaluate ideas with empirical experiments and performance measurements. Performance results, however, are meaningful only if they are collected over a large enough data set. From the beginning of the project the team adopted a methodology of evaluating Watson over very large test sets containing several thousand questions. Dealing with such large data sets creates two problems. First, running large scale experiments is expensive and time consuming, especially since Watson can take nearly two hours to evaluate a single clue on a single high-end processing core. Second, the logging information from a single experiment is enormous (tens of gigabytes) and impossible to understand in the raw.

To address the first problem, the team assembled a computational cluster with hundreds of high-end, large main memory, multi-core servers, called the *DeepQA Cluster*. The DeepQA Cluster is able to run very large scale experiments, but it is a finite resource that must be carefully managed and shared by the research team. To achieve maximum utilization of the DeepQA Cluster, we implemented a cluster management and load balancing system that schedules experiments on the cluster and manages resource sharing among the users. The DeepQA Cluster enables a single experiment of several thousand clues to run in just a few hours, and for the entire DeepQA team to share the resources fairly with high resource utilization.

To address the second problem of how to manage, interpret, and analyze the output from experimental runs, we created the *Watson Error Analysis Tool*, or WEAT. The WEAT stores and catalogues all of our experiments, calculates summary metrics for an experiment containing thousands of questions, and supports drilling down into the details of how each result was produced. The WEAT uses DB2 to store all of the data associated with experimental results, and a web front end that allows quick access to summary performance metrics, a wide variety of ways to filter and compare experiments, and the ability to dive down into the details of how Watson came up with its final answer list.

Ferrucci (2012) uses the term AdaptWatson to characterize our methodology for rapid advancement and integration of algorithmic techniques. This paper describes the concrete instantiation of AdaptWatson that was employed in creating the original Jeopardy! version of the Watson question answering system. We describe in detail our software integration and testing protocol, the hardware cluster we built to support a large volume of high-throughput experiments, and some of the tools we created to enable system performance measurement and error analysis.

Software Integration and Testing Protocol

The DeepQA team wrote over 500,000 lines of Java code organized into more than 130 component projects to create the final Watson system that competed in the Jeopardy! Grand Challenge exhibition match against Ken Jennings and Brad Rutter. With such a large amount of code contributed by over two dozen researchers, instituting appropriate protocols for tracking code changes and integrating and evaluating components was critical. The DeepQA team took two main steps to manage this process.

Source Code and System Versioning

The first step was the adoption of a source code control and tracking system that supports file change tracking and logging, the management and versioning of groups of files organized into components, and the ability to associate a specific version of each component with an overall integrated system release. We met all of these requirements by using Subversion (SVN, http://subversion.apache.org/) for version control in combination with Eclipse (http://eclipse.org/) as our integrated development environment.

SVN provides source code control and versioning for individual files using a collaborative model where users check out copies of files (without locking) from the SVN repository, modify those files, and then commit those changes to a new revision of the file in the repository with appropriate log comments. SVN saves all revisions of files and their associated log entries, can retrieve arbitrary revisions at any time, and provides tools for comparing and merging changes across different revisions. SVN can also manage hierarchical directories of files and, in particular, supports tagging and branching at arbitrary granularities.

For Watson, we chose to organize the code into component projects where each project corresponds to an Eclipse project, and each project is a folder in SVN. For each SVN folder we then created *trunk*, *branch*, and *tag* sub-folders. The trunk folder always contains the mainline development version of the project, while the branch folder contains experimental or special versions of the project, and the tag folder tracks "tagged" versions of the project. A tag represents a version of the component and associates a specific revision of every file in the component project with the component version.

The end-to-end integrated Watson system is represented by an Eclipse *workspace* that contains all of the component projects that make up the system. A *release* of the end-to-end integrated system is the collection of specific versions (tags) of each component project that makes up the end-to-end system release. To track system releases, we used the rather informal mechanism of creating an Eclipse *project set file* (PSF) for each release. A PSF file is a list of Eclipse projects and, for each project, a specific version of that project in the source code repository (e.g., trunk, or more usefully, a specific tagged version of the project). Given a PSF file, Eclipse can check out the identified version of each project listed in the file and populate the workspace for the corresponding system release. We tracked our system releases by creating a new PSF file for

each system release, which in turn identified the tagged versions of each component project in that release.

System Development and Integration

With a protocol in place for managing the source code, component projects, and releases, the second step in establishing our software development and integration protocol was to formalize how modified or new components are integrated into the system. Consistent with our metrics-driven research and development process, we adopted the system development lifecycle depicted in **Error! Reference source not found.**. The development lifecycle starts with a baseline version of the system in Step 0. Of course, early on in the life of the project we did not have a baseline Watson system, so prior to formally adopting this protocol there was a bootstrapping phase that involved white board design and development of the first version. Once we had built an end-to-end system with sufficient functionality, we pursued the protocol described here.

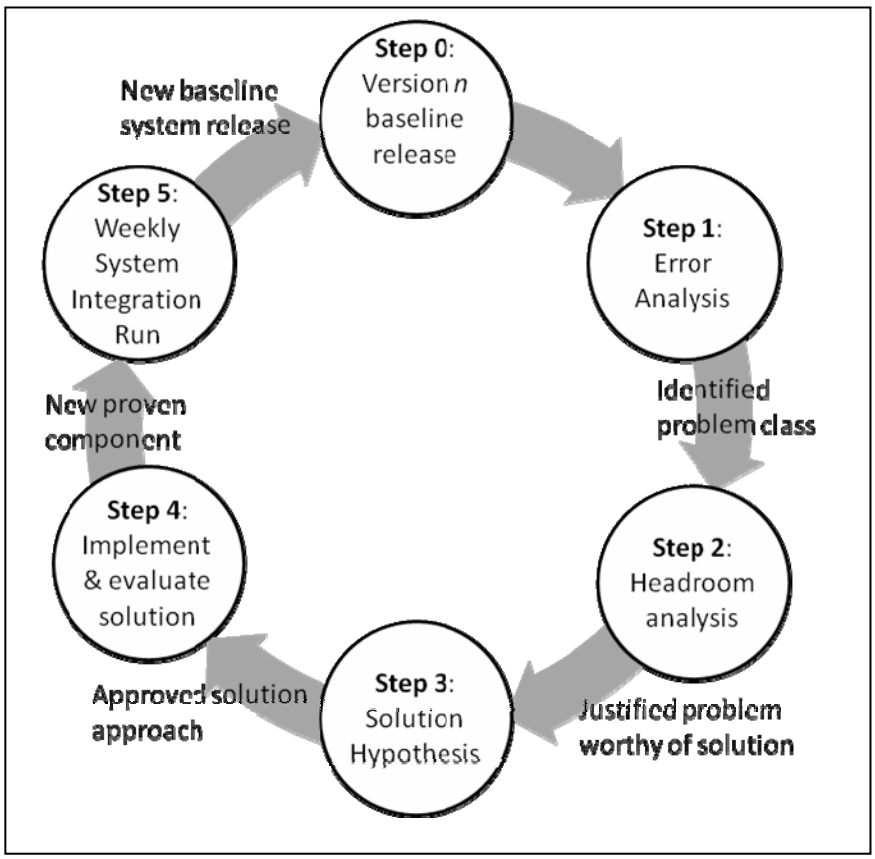


Figure 1: System development lifecycle.

The development process begins in Step 1 with running the baseline system on Jeopardy! clues and conducting error analysis on the results. We organize our Jeopardy! clues into three classes

of data: development, training, and test. Development data consists of clues analyzed by the DeepQA team as part of a general problem analysis – they provide insight into the problem and motivate the technical approaches and solutions pursued by the team. Training data consists of a random sample of clues that should be representative of the overall problem being solved. We use training data to train the machine learning models that weigh and combine the output of the components in the system (Gondek et al, 2012). Test data is also a representative random sample of clues assembled into a test set to measure the performance of the system. A key attribute of both training and test data is that they are kept "blind" (i.e., the system researchers and developers never look at these clues). This helps ensure that the system is not over fit to the training data and the evaluation results on test sets are meaningful.

There are special circumstances where test sets may be sampled from a selected subset of all clues. For example, Final Jeopardy! questions represent a particularly challenging class of Jeopardy! clues that demand special solution techniques. To evaluate these techniques, we have created test sets that consist of randomly sampled Final Jeopardy! questions.

The error analysis process in Step 1 involves a review of various aggregate metrics measured over the entire test set, as well as detailed analysis of specific success and failure cases. Ideally we can group the failure cases into classes and focus on addressing the problem represented by that class. For example, error analysis might reveal that recall is a problem (i.e., the correct answer is not being generated as a candidate answer for the question), indicating that the search and candidate answer generation components of the system (collectively called hypothesis generation) need improvement.

Once a class of problem has been identified, we perform an analysis of the impact of addressing that problem in Step 2. This analysis produces a "headroom" estimate, which shows the potential impact on overall system performance if the current problem class is solved. This is typically an ideal estimate, since creating a solution that solves the entire class is rare. Nevertheless, these estimates are useful for comparing the potential impact of different solutions competing for resources (mainly researcher time) and deciding which approaches are worthy of investment. This step also helps avoid "mole whacking," where a solution is built that addresses a very narrow problem, often in a very domain specific way. We always prefer solutions that scale to as large a class of problem as possible and try to avoid mole whacking. If the headroom analysis shows insufficient potential impact, development does not continue to the next step in the lifecycle. For example, consider the following example clue:

> *ENGLISH LITERATURE In "Ivanhoe" the crusading knight is disinherited by his father because of his love for her (Answer: Rowena)*

The focus of this clue is "her," implying that the clue is asking for a female person; this is obvious to a typical human reading the clue. However, a purely syntactic approach could just as

easily conclude that “his” is the focus of the clue and the question is asking for a male person. The human excludes this possibility presumably by knowing or assuming that the crusading knight in “Ivanhoe” is male and thus that “his” refers back to the knight and “her” is the only pronoun in the clue that does not refer back to some earlier noun. DeepQA does have pronoun anaphora resolution (Lally et al., 2012) but it does not make this assumption regarding the crusading knight and is not able to sort this example out. This is a particularly challenging example of a general challenge in which a Jeopardy! clue has both male and female pronouns and the QA system needs to decide which one is the question focus. This seems like a general NLP problem that could be a frequent source of error. However, we conducted a headroom study and found that such clues are actually extremely infrequent. Furthermore, the issue of determining which personal pronoun is the question focus among multiple alternatives is an idiosyncrasy of Jeopardy!’s declarative phrasing of questions. An interrogative version of this question would replace “her” with “whom”, making it trivial to identify the focus. Any work done to address clues of this sort would impact only a tiny fraction of Jeopardy! questions and would have little or no utility in other domains. Thus we did not continue to pursue this issue.

Step 3 of the system development lifecycle is solution hypothesis. In this step, one or more DeepQA team members will form a tactical sub-team to hypothesize a technical solution, sketch out a design, and possibly perform additional analysis or early prototyping and evaluation to justify the technical approach. The sub-team presents the solution hypothesis and analysis results to the larger team and the Principal Investigator for review and approval. If the approach is approved, development continues in Step 4.

Most of the team’s time is spent in Step 4 of the lifecycle, where we develop and evaluate new solutions. This step includes implementing the solution hypothesized in Step 3, evaluating the solution at the component level, and integrating the solution into the end-to-end baseline system to measure overall impact. While component-level performance is useful to determine if the solution is working as designed, ultimately end-to-end system impact is the most important measure of the value of the solution. Step 4 typically involves iteration within the step of modifying or extending the solution, testing the new component, and evaluating end-to-end system impact. We generally encourage a rapid prototyping approach here to get feedback as quickly as possible on the potential for the hypothesized solution. If the solution is not having the expected impact, we can catch this early on and redirect resources to more promising solutions. We do take care, however, to look for promising solutions that require more investment or more time, recognizing that some problems are more challenging and may take longer to solve. Once we have demonstrated the efficacy of a component solution in Step 4, the component is tagged with a new version and it is ready for full system integration and testing in Step 5.

In Step 5 we combine all integration-ready components into a new release of the end-to-end system and run a full integration test. We call this integration test a “weekly run,” which, if

successful, generates a new baseline version of the system for Step 0, and the lifecycle repeats. We use the term "weekly run" because early on in the project we ran Step 5 of the lifecycle on a weekly basis. As the system became more complex and the weekly run took longer to execute and analyze, the interval between weekly runs grew to two or three weeks, until eventually weekly runs were scheduled on demand when there was sufficient change at the component level to warrant a new integrated baseline release.

The weekly run process in Step 5 consists of the following tasks. First, all component owners that wish to include a new version of their component in the weekly run (after successful completion of Step 4) register their new tagged component in a weekly run log, along with documentation about the nature of the changes in the component. Next, a designated weekly run master creates a new PSF file with updated entries for the tagged versions of all of the components. This PSF file now represents the new baseline system release about to be created. The run master then uses the PSF file to check out a new Eclipse workspace with the specified version of each component. Once this workspace is checked out and compiled, the run master updates the system configuration as necessary to reflect new component parameters, system data, machine learning parameters, etc. The run master then configures the training data and test sets and launches the weekly run job, which trains the necessary machine learning models and runs the various evaluation test sets. Upon successful completion, the run master uploads the test results into the Watson Error Analysis Tool (described below), and the overall lifecycle continues with Step 1.

While the desired result of performing steps 1 through 5 is to advance to the next step in the development lifecycle, it is possible that any given step is unsuccessful. Depending on the nature of the failure, the process may revert to any step earlier in the process. For example, if a component solution is failing to demonstrate the desired impact in Step 4, the team may abandon the approach and revert to Step 3 and propose a new solution approach.

Ablation Studies

The quantity and diversity of components for finding and evaluating answers in DeepQA makes our question answering system extraordinarily robust and flexible. Each of these components is fallible in some ways, but the system does not depend excessively on any of them. We have conducted many experiments in which we have ablated one or more components from the system, and we consistently find that all but the most dramatic of ablations have very little effect. In this sense, Watson is like a suspension bridge in which no single cable holding it up is indispensible.

Most of our evaluations are run on test sets of a few thousand questions. Such sets are typically useful for distinguishing among configurations that differ by at least 0.5% in accuracy. In our

experience, impact numbers less than this magnitude are rarely statistically significant for a test set of this size.

There are 50 evidence scoring components in the full Watson QA system. On a random sample of 3,508 previously unseen Jeopardy! clues, the full Watson QA system answers 71.1% correctly. If we ablate *all* of the evidence scorers, then the final merger must rely only on features produced during hypothesis generation; on the same set of clues, the Watson QA system with none of the evidence scorers answers 50.0% correctly. Thus the net impact of all of the evidence scorers together is 21.1%. Given 50 of these components with a net impact of 21.1%, it should not be surprising that removing any one of them has very little impact. When we run experiments in which we ablate a single evidence scorer from the full Watson system, we rarely see a statistically significant impact over a few thousand questions, and we never see an impact of 1% or greater. Consequently, we have taken other approaches to measuring the effectiveness of components.

For some blocks of related components, measure the impact of removing all of those components, and then further measure how any one of them performs relative to having all or none of them. We have taken this approach to measuring search strategies (Chu-Carroll et al., 2012), answer typing strategies (Murdock et al., 2012a), and evaluating passage evidence (Murdock et al., 2012b). Some blocks of components have substantial impact; for example, removing *all* of the 16 evidence scoring components that address answer typing reduces the accuracy of Watson by nearly 5%. When we then want to measure the effectiveness of a single one of these components, we can compare the performance of Watson with that one component to the performance with none of the components in that block. For example, Watson's accuracy with one selected answer typing component (the most effective one, by this measure) is 3% higher than its performance with no answer typing components (i.e., 2% lower than the performance with all answer typing components) (Murdock et al., 2012a).

Another approach to measuring impact of a component on question answering is to demonstrate how that component affects a much simpler version of DeepQA. One such configuration we have built is the Watson Answer Scoring Baseline (WASB) system. The WASB system includes all of Watson's Question Analysis and Search and Candidate Generation. It includes only one evidence scoring component: an answer typing component that uses a named-entity detector. The use of named-entity detection to determine if some answer has the semantic type that the question requires is a very popular technique in question answering (as discussed in detail in Murdock et a.l, 2012a), which is why we included it in our baseline. We have evaluated the impact of adding various components to the WASB system (Murdock et al., 2012a; 2012b; Kalyanpur et al., 2012; Chang et al., 2012), and found that we are able to examine and compare the individual effectiveness of components for which simple ablations from the full system do not provide statistically meaningful insights given the size of the test sets we use. We see many

individual components that provide an impact on accuracy in the range of 2%-5% when added to the WASB system.

Some components are designed to address specific issues that occur very infrequently, e.g., questions about a specific topic or questions that solve a specific type of puzzle (Kalyanpur et al., 2012; Prager, Brown & Chu-Carroll, 2012). For those components, impact on an sample drawn randomly from all known Jeopardy! questions is certain to be very small regardless of what baseline is used. In those cases, we typically evaluate performance on a random sample drawn from the pool of questions for which they are applicable. For example, we have an evidence scoring component that uses structured knowledge of geospatial entities and relations to address geography questions; that component has an impact of 1.9% on a random sample of questions for which it is applicable, but would have much smaller impact on a random sample of all questions.

In some cases, it is simply not possible to remove a component and still have the QA system run and answer questions; in those cases, it makes more sense to compare the impact of such a component to the impact of less sophisticated baseline versions of the component. For example, when we replace the final merging and ranking component (in the full Watson system) with a very simple baseline merging and ranking component, we see a 4.4% reduction in accuracy (Gondek et al., 2012).

DeepQA Cluster

The system development process described in the previous section is motivated by our general metrics-driven approach. A key element of this approach is running large scale experiments to evaluate component-level performance and, more importantly, end-to-end system performance. In particular, with a few exceptions, we judge the development of a new solution component successful only after we have demonstrated that the component improves end-to-end system performance.

One of the challenges with running many large scale experiments is the time it takes to run those experiments. The Watson system eventually grew in complexity to the point where a single question running on a single high-end processer would take 20 to 30 minutes on average, with some outliers requiring nearly two hours to complete (Epstein et al., 2012). Early in the project the DeepQA team recognized the growing costs associated with running experiments and the corresponding need for a computing environment that would enable several users to run many large scale experiments and obtain the results with reasonable turn-around time. We addressed this need by building the *DeepQA Cluster*. The DeepQA Cluster includes several hundred servers combined with a custom software system, called BlADE, for managing the servers.

Hardware

All of the servers in the DeepQA Cluster are IBM xSeries Blades running the Linux operating system. The DeepQA Cluster is not able to provide rapid response to individual questions; our solution for rapid interactive question answering is the "production" system, comprised of IBM Power 750 machines (Epstein et al., 2012). The DeepQA Cluster is primarily optimized, instead, for large batch jobs composed of many train and/or test questions.

The servers in the DeepQA Cluster are divided into several roles, including workstations, file servers, general purpose servers, and workers. Workstations are typically 4-core machines with 16GB of main memory. They provide the development environment for members of the DeepQA team, with one to four users assigned to each workstation. Users will log on to their workstation, start one of the standard Linux windowing environments (e.g., Gnome or KDE), and run command shells, editors, Eclipse, web browsers, and other tools and utilities. Users conduct most of their software development, debugging, and testing on their workstation, and launch all of their experimental runs from their workstation.

The file servers are higher end machines with 8 cores and 48GB of main memory each. They are dedicated to running NFS servers and providing access to the shared file systems in the cluster. The file servers mount storage from a Storage Area Network (SAN) using a 4Gb FibreChannel network, with access to over 100TB of storage.

The general purpose servers range from 4-core 16GB machines, to 8-core 48GB machines. We use these servers to run instances of the WEAT, DB2 (for the WEAT), and various development system components that are deployed as services for better throughput in experimental runs. These services typically include full-text search engines, large dictionary-based concept annotators, and knowledge base search servers.

The bulk of the servers in the cluster are worker machines, divided into two classes. The *small worker* class contains machines with 8 cores and 32GB of main memory, while the *large worker* class contains slightly higher end machines with 8 cores and 48GB to 80GB of main memory. These servers are dedicated to running experiments, debugging sessions, and other jobs requiring large computational resources.

Most of the servers in the DeepQA Cluster are connected by a 1Gb Ethernet network using a tree of high-end switches that support line-rate switching. The root switch in this tree also provides a 10Gb Ethernet port with line-rate switching. The 10Gb port attaches to a 10Gb Ethernet sub-network that connects all of the file servers and many of the general purpose servers, which generate most of the network traffic and benefit from the higher throughput of a 10Gb network.

Software

To effectively utilize the hardware resources in the DeepQA cluster, the DeepQA team developed the BlADE software system. The BlADE system is a custom server resource management and scheduling tool that provides several key functions:

1. Automatic assignment of workers to jobs
2. Distribution of a single job across multiple workers for high throughput
3. Load balancing across available workers
4. Fair distribution of resources among multiple users, with support for priorities
5. Ability to reserve workers for dedicated tasks
6. Overall cluster utilization status

The core BlADE system runs on a single management server. The BlADE management server keeps track of all of the worker machines that participate in its cluster and all of the jobs currently running in the cluster. The management server is also responsible for scheduling jobs, rebalancing worker resources across the jobs, managing worker reservations, and providing a web-based monitoring tool.

The most common task on the DeepQA Cluster is running a large batch experiment on several thousand questions. A user launches this kind of job by running the BlADE client program from their workstation and passing the BlADE client a job configuration. The BlADE client starts by contacting the BlADE management server to allocate workers for the job. The BlADE client then creates launch scripts for each of the remote workers and sets up a queue of questions to be run by the workers. BlADE then launches a remote shell command on each of the allocated worker machines to execute the launch scripts, and each remote job enters the initialization stage. Once the a remote job has initialized (where a remote job is typically most of the Watson question answering pipeline, which can take several minutes to initialize), the job starts pulling questions off of the job question queue and executes those questions.

If there is just one job running on the cluster, it will be allocated all of the non-reserved resources (or as many resources as are required to run all of the questions simultaneously, should there be enough resources). As soon as more than one job is submitted, BlADE will allocate the non-reserved resources between the multiple jobs according to the priorities associated with the jobs. This may require rebalancing, where a worker is revoked from one job and given to a different job. In this instance, the question being run for the victim job will be terminated and placed back on the question queue for that job, and ownership of the worker will transfer to the new job. As more jobs are simultaneously submitted, BlADE continuously rebalances the resources to ensure fair distribution and high utilization across the jobs. We assign a higher priority to certain tasks, such as the weekly run jobs, which tells BlADE to allocate a larger share of resources to those tasks and enable them to finish sooner.

A screen shot of the BlADE monitoring tool is shown in **Error! Reference source not found.**. The tool shows a summary status of all of the workers in the cluster, including how many are unavailable (i.e., down), reserved, migrating to user tasks, starting (i.e., initializing), running, and free. The tool also shows how many jobs are currently running and for each job gives details about the job (including owner, task description, priority, number of workers from the cluster in use, how many questions are running and queued for the job, average question run time, and an estimate of the job completion time given the resources currently allocated to the job). The tool also provides controls to kill a job, adjust job priority, or reserve a machine from the cluster for dedicated use. The BlADE monitoring tool supports drilling down into individual jobs to view logging information, any exceptions that have been thrown in the job, and worker utilization for workers assigned to the task.

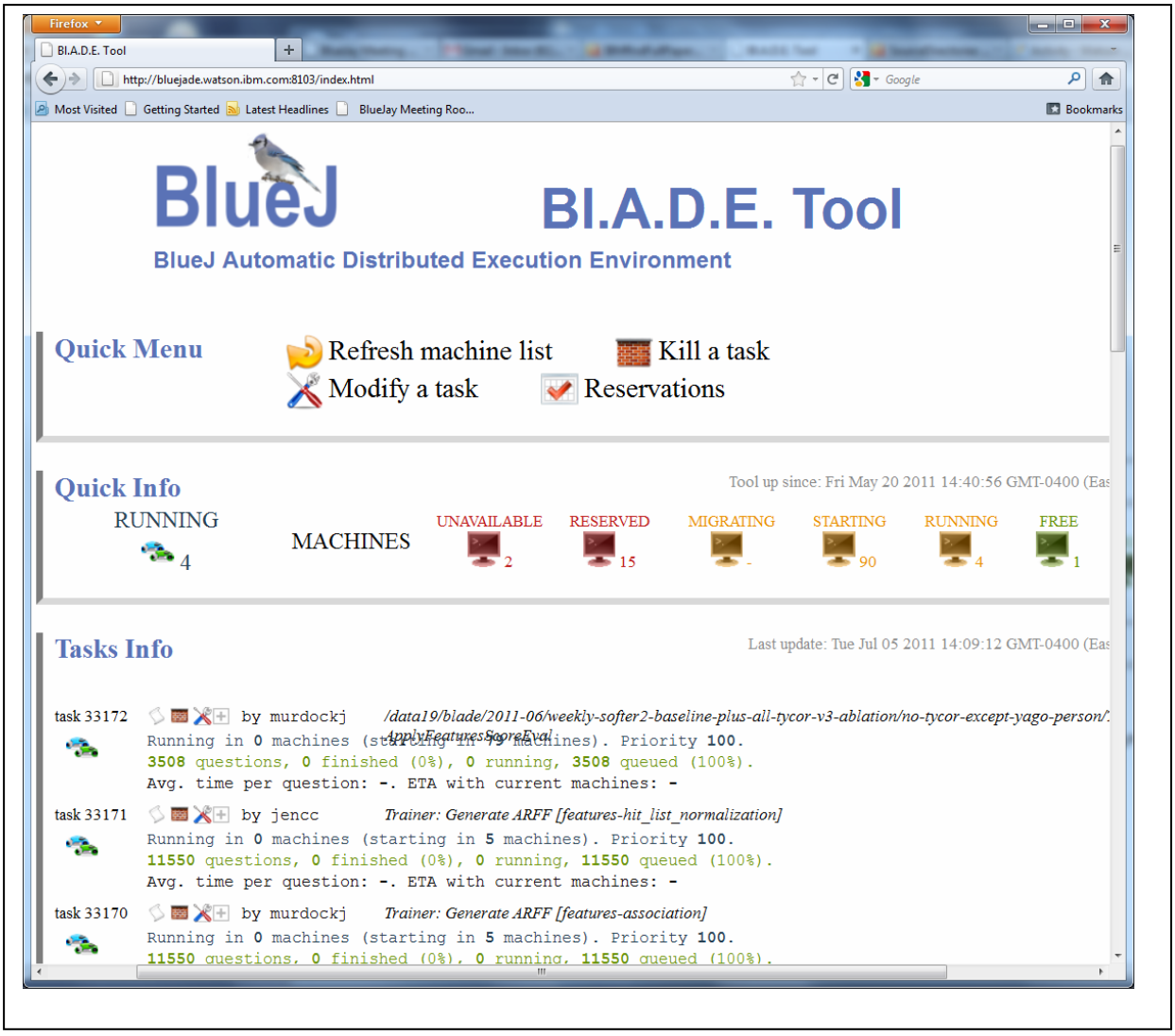


Figure 2: BlADE web-based monitoring tool

Worker utilization information includes details such as average load over recent time periods, memory use, I/O load, CPU states, and network traffic. This level of hardware resource utilization is provided by an optional hardware monitoring tool, e.g., Ganglia (http://ganglia.info/), which typically requires the installation of a monitoring program on every worker in the cluster.

Outside of this hardware monitoring tool, the BlADE system does not have any software installed locally on the workers. Rather BlADE relies on a shared file system that provides a common file system view across all of the worker and workstation machines, and the ability to execute remote shell commands to launch user jobs on the workers.

Error Analysis Tools

During the approximately four years it took to develop Watson, the DeepQA team ran over 7,000 experiments. In a typical experiment, each question generates approximately 25MB of output. For a typical test set of 3500 questions, this is nearly 90GB of output data. A full weekly run with over 14,000 training questions and several test sets can easily generate over 750GB of output. With so much output to analyze and the central role of error analysis in our metrics driven research and development process, the DeepQA team had a clear requirement for powerful error analysis tools.

The core tool we developed to meet this requirement is the Watson Error Analysis Tool, or WEAT. The WEAT is a web-based application with a rich front end that runs in a browser and a powerful back end Java servlet that supports a wide variety of queries and filters and uses DB2 to store all of its data. From the beginning we wanted the WEAT to be a shared resource where users could easily share and reference experimental results. A web front end with a central database repository in the back end satisfies this requirement and, in particular, allows a WEAT user to run the WEAT using any browser client with network access to the WEAT servers.

A typical Watson experiment consists of a large number of questions run through the entire, end-to-end processing pipeline (Ferruci, 2012). Each question has results from the various processing stages, including question analysis, primary search, candidate generation, supporting evidence retrieval, answer and evidence scoring, and final merging and ranking. When we designed the WEAT we wanted to provide easy access to the results from each of these stages, including flexible mechanisms for filtering views over those results and comparing those results between experiments running different versions of the system. This led to a relational database design where the experimental results are split out into tables organized according to a schema that supports the most common filtering and query operations, as well as a certain amount of extensibility to accommodate changes in the Watson components and output being analyzed.

In addition to experimental run output, the WEAT stores metadata about each question, including how the questions are organized into the original Jeopardy! episodes from which they were drawn, category labels, dollar amounts, round information (i.e., Single Jeopardy!, Double Jeopardy!, and Final Jeopardy!), which questions are Daily Doubles, and the answer key for each question. This information enables the WEAT to compute a wide variety of evaluation metrics, including per-episode metrics. The full suite of metrics produced by the WEAT is generated by a pluggable module called *ScoreEval*. The ScoreEval susbsystem can be run as a stand-alone scoring tool to evaluate the results of an experiment, or it can be incorporated into another tool such as the WEAT. In the case of the WEAT, ScoreEval produces a number of question answering metrics useful for evaluating an experiment over a large number of individual questions. The question-answering metrics used in DeepQA are described in detail in the following section.

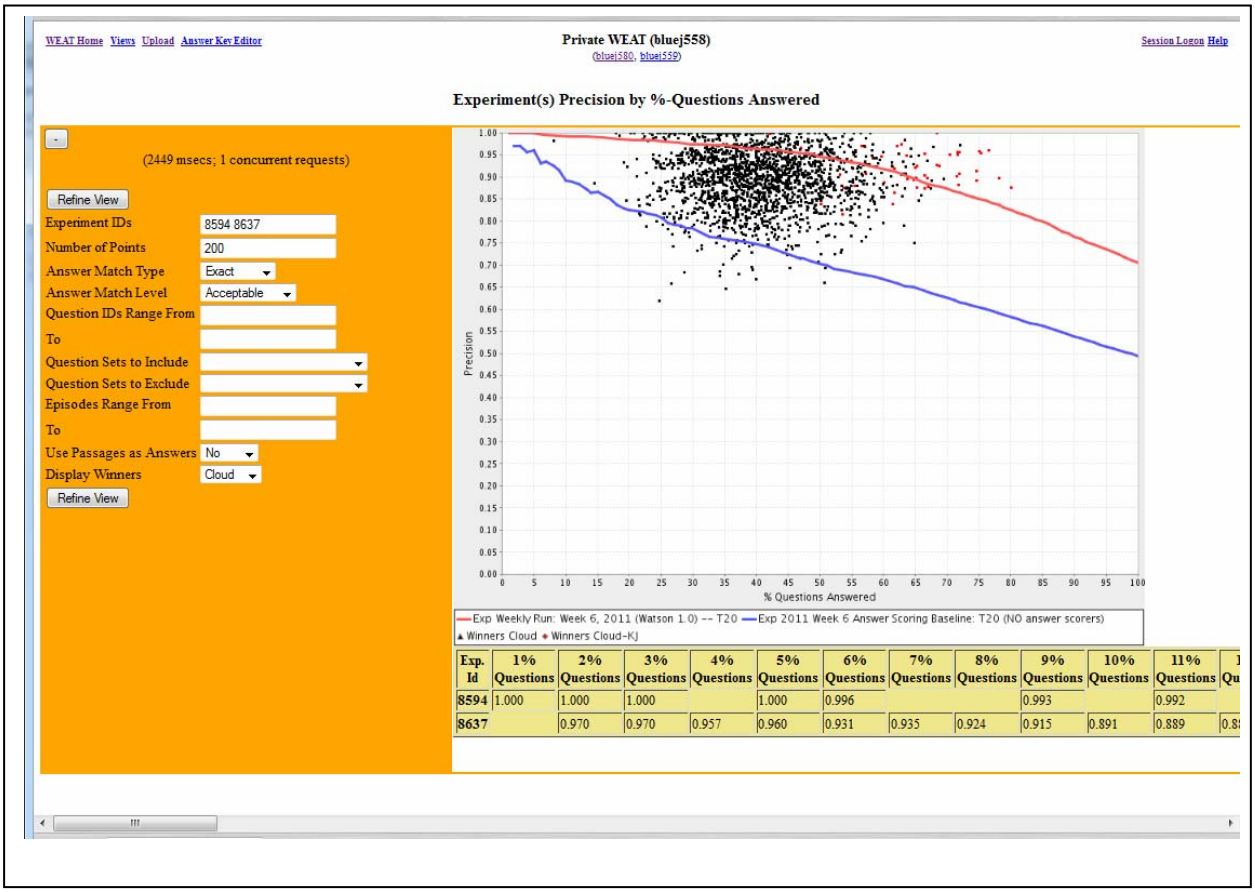


Figure 3: WEAT precision curves

Another important feature of the WEAT is the ability to plot a precision curve from an experiment against a reference "winners cloud". In this plot, the x-axis is the percent of questions attempted, and the y-axis is the precision for those questions attempted (assuming it

only attempts the clues for which it is most confident). The winners cloud plots the performance of actual Jeopardy! game winners for a large sample of real Jeopardy! games, allowing a convenient reference point for human performance. Figure 3 shows the plot we have used to visualize performance on a large blind data set for the complete and baseline versions of the QA system. From this plot it is easy to see how versions of the system compare and also to get a sense for the quality of the confidence estimation performed by the system.

WEAT is able to filter questions based on a wide variety of criteria and drill down into the details of a question. Figure 4 shows some of the filter criteria on the left hand side that the user can specify to select a subset of questions in the experiment. Common tasks include showing only those questions where the correct answer is not in first place (accuracy problem), the correct answer is not in the answer list (recall problem), or showing specific classes of questions based on values for any of the feature scores assigned to the answers by the DeepQA analytics. Another common filter task used when comparing two experiments is to view only those questions where the first experiment has the wrong answer and the second experiment has the correct answer in first place, or vice versa. This view, along with several variations based on whether or not and where the correct answer appears in the answer list, gives a convenient snapshot of the performance differences (at the question level) of two different versions of the system.

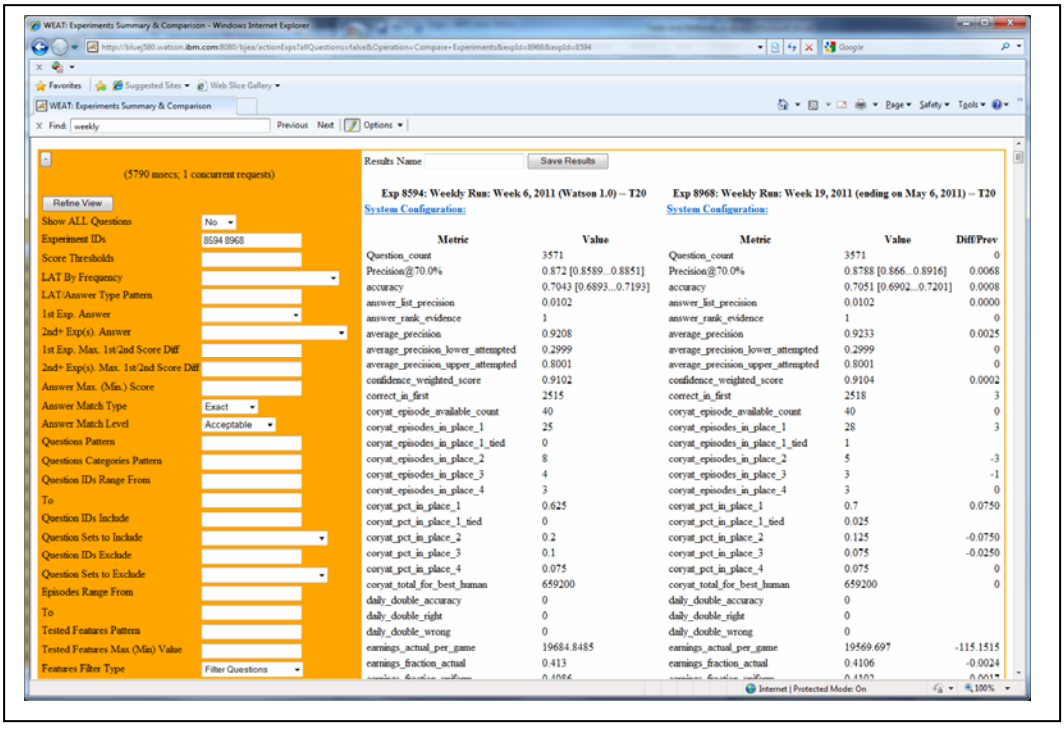
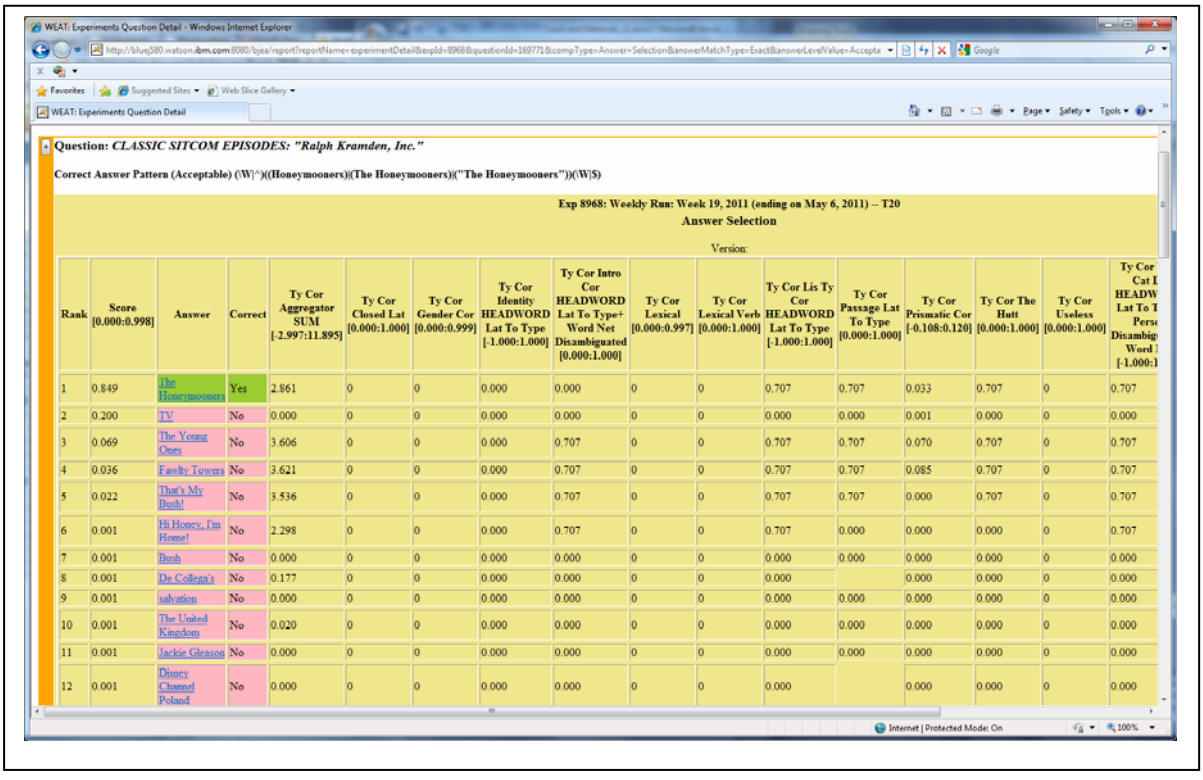


Figure 4: WEAT summary metrics

Once the user has identified a specific problem question to analyze, the WEAT provides the ability to drill down into the details of the question. The drill down starts by looking at the results from each of the overall processing pipeline stages, including the Focus and Lexical Answer Types identified during Question Analysis (Lally et al., 2012), the document and passage search results produced by the various primary search operations (Chu-Carroll et al., 2012), the candidate answers generated (Chu-Carroll et al., 2012), the feature scores produced by all of the analytics run throughout the entire Watson pipeline, and the rank and confidence score assigned to each answer by the Final Merging and Ranking component (Gondek et al., 2012).

Comparing and understanding the feature scores produced the by the Watson analytics for two different candidate answers in the same question, or the same candidate answers across two different versions of the system, is one of the most challenging error analysis tasks. The WEAT attempts to simplify this task by allowing the user to filter the feature score view by specifying a subset of the features using a regular expression. Figure 5 shows an answer list with feature scores for every answer, where the features have been filtered to show only the TyCor features (Murdock et al., 2012a). Without filtering this view to just the TyCor features, there would be several hundred columns of feature scores for every candidate answer. Filtering this view to a subset of features of interest greatly focuses the error analysis task for the user.



Rank	Score [0.000:0.998]	Answer	Correct	Ty Cor Aggregator SUM [-2.997:11.895]	Ty Cor Closed Lat [0.000:1.000]	Ty Cor Gender Cor [0.000:0.999]	Ty Cor Identity HEADWORD Lat To Type [-1.000:1.000]	Ty Cor Intro Cor HEADWORD Lat To Type+ Word Net Disambiguated [0.000:1.000]	Ty Cor Lexical [0.000:0.997]	Ty Cor Lexical Verb [0.000:1.000]	Ty Cor Lis Ty Cor HEADWORD Lat To Type [-1.000:1.000]	Ty Cor Passage Lat To Type [0.000:1.000]	Ty Cor Prismatic Cor [-0.108:0.120]	Ty Cor The Hutt [0.000:1.000]	Ty Cor Useless [0.000:1.000]	Ty Cor Cat I HEADW Lat To T Pers Disambig Word [-1.000:1
1	0.849	The Honeymooners	Yes	2.861	0	0	0.000	0.000	0	0	0.707	0.707	0.033	0.707	0	0.707
2	0.200	TV	No	0.000	0	0	0.000	0.000	0	0	0.000	0.000	0.001	0.000	0	0.000
3	0.069	The Young Ones	No	3.606	0	0	0.000	0.707	0	0	0.707	0.707	0.070	0.707	0	0.707
4	0.036	Fawlty Towers	No	3.621	0	0	0.000	0.707	0	0	0.707	0.707	0.085	0.707	0	0.707
5	0.022	That's My Bush!	No	3.536	0	0	0.000	0.707	0	0	0.707	0.707	0.000	0.707	0	0.707
6	0.001	Hi Honey, I'm Home!	No	2.298	0	0	0.000	0.707	0	0	0.707	0.000	0.000	0.000	0	0.707
7	0.001	Bush	No	0.000	0	0	0.000	0.000	0	0	0.000	0.000	0.000	0.000	0	0.000
8	0.001	De Collega's	No	0.177	0	0	0.000	0.000	0	0	0.000		0.000	0.000	0	0.000
9	0.001	salvation	No	0.000	0	0	0.000	0.000	0	0	0.000	0.000	0.000	0.000	0	0.000
10	0.001	The United Kingdom	No	0.020	0	0	0.000	0.000	0	0	0.000	0.000	0.000	0.000	0	0.000
11	0.001	Jackie Gleason	No	0.000	0	0	0.000	0.000	0	0	0.000	0.000	0.000	0.000	0	0.000
12	0.001	Disney Channel Poland	No	0.000	0	0	0.000	0.000	0	0	0.000		0.000	0.000	0	0.000

Figure 5: WEAT answer list details filtered to show TyCor features

Figure 6 shows a comparison of feature scores for the same answer, “Sam,” across two different experiments (i.e., versions of the system). In this example, the correct answer is “Sam”, but the system on the left placed that answer in 32nd place, while the system on the right placed that answer in first place. The feature score comparison view highlights in red the feature scores that differ across the two versions, allowing the user to quickly zoom in on the part of the system that accounts for the difference.

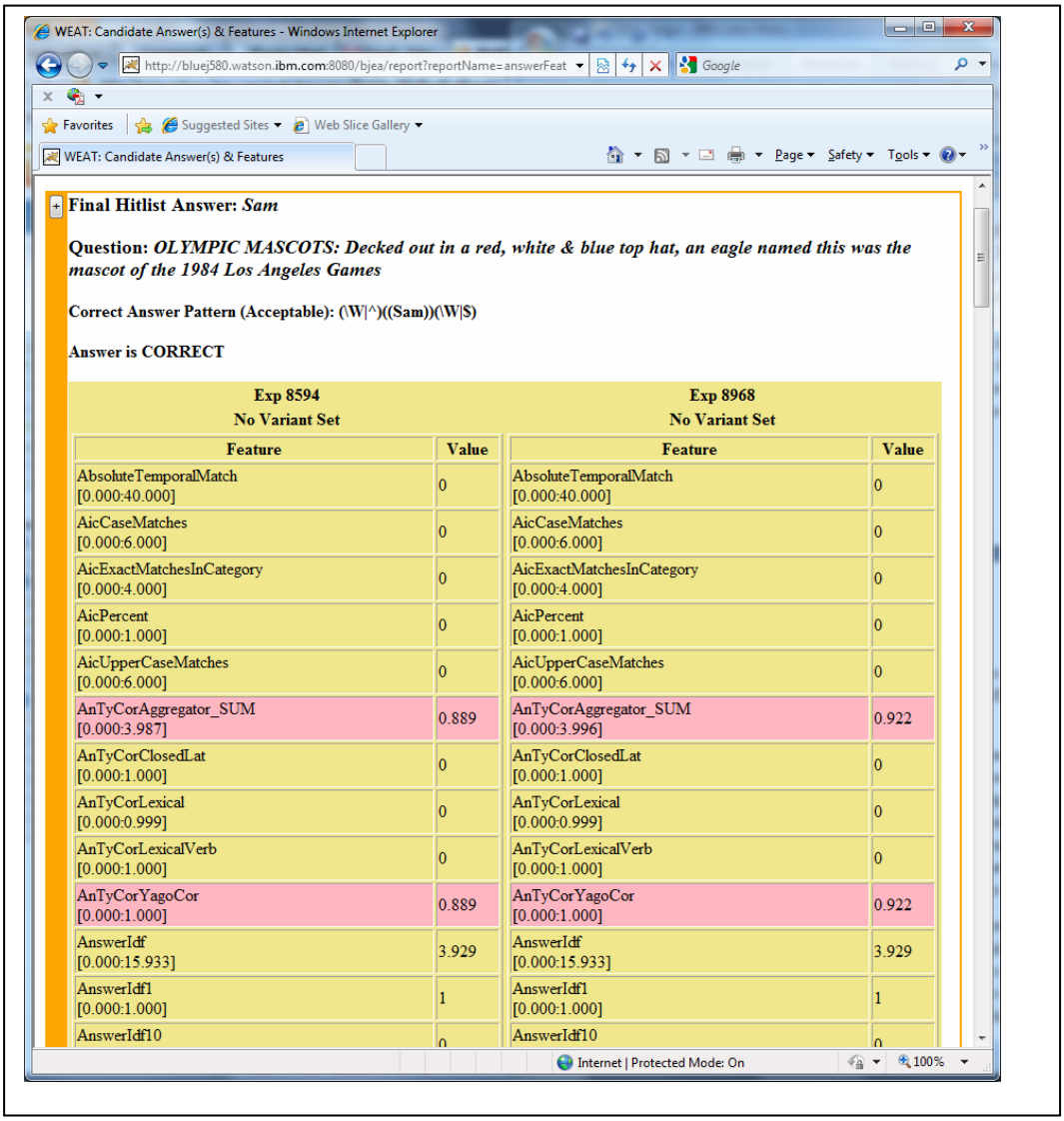


Figure 6: WEAT comparison of feature scores across experiments

Metrics

The full suite of metrics produced by the WEAT is generated by a pluggable module called *ScoreEval*. The ScoreEval susbsystem can be run as a stand-alone scoring tool to evaluate the

results of an experiment, or it can be incorporated into another tool such as the WEAT. In the case of the WEAT, ScoreEval produces a number of question answering metrics useful for evaluating an experiment over a large number of individual questions. Some of the most interesting metrics that ScoreEval generates are:

- Accuracy: Percent of questions with correct answer in first place
- Precision@70%: Percent of questions with correct answer in first place when the set is restricted the 70% with the highest confidence scores
- Average precision from 30% to 80%: Average value across the range from 30% to 80% of the percent of questions with correct answer in first place when the set is restricted the with the highest confidence scores
- Confidence weighted score: A weighted accuracy score, with higher weights to answers that have higher confidence (Voorhees, 2002).
- Ideal uniform earnings fraction: The maximum across all possible threshold values of the "uniform earnings fraction" at that threshold. Uniform earnings fraction at a threshold is defined as the number of questions for which the top ranked answer is correct and has a confidence greater than the threshold divided by the total number of questions for which the top ranked answer has a confidence greater than the threshold.
- Coryat Score: The sum of the Jeopardy! dollar value of clues answered correctly minus the Jeopardy! dollar value of clues answered incorrectly.
- Coryat percent in first place: The percentage of Jeopardy! games in the test set for which the system's Coryat Score was greater than the observed Coryat Score for the highest human participant in that game (when it was originally aired).
- Binary Recall: Percent of questions with the correct answer somewhere in the answer list)
- Binary Recall @ K (for K=5, 10, 25, 50, and 100): Percent of questions with the correct answer in the top K answers for that question.

Accuracy is a simple and intuitive metric that directly measures the effectiveness of the system at identifying the correct answer. It is very useful for component developers working on development data because changes in accuracy can be clearly and unambiguously associated with specific questions that were gained or lost.

However, accuracy does not measure all of the aspects of the system's capabilities that we would want to measure. One key issue that accuracy does not measure is how effective the system's confidence scores are at gauging whether a particular answer is right. Precision@70, average precision from 30% to 70%, and confidence weighted score, and ideal uniform earnings fraction are four metrics that do account for confidence. Of these, ideal uniform earnings fraction most directly measures what we want confidence to accomplish to win at Jeopardy!, since Jeopardy! provides the same amount of credit for answering a question correctly as the penalty it provides for answering a question incorrectly. Coryat score is even more specific to Jeopardy! because it

factors in the specific dollar values that Jeopardy! assigns to clues. Coryat percent in first place also takes into account historical human performance in Jeopardy! and is particularly useful for building intuitions about how competitive a system would be against a typical human Jeopardy! champion. Coryat does have limits, since it ignores issues like speed (Epstein et al., 2012) and game strategy (Tesauro et al., 2012).

An aspect of the system's effectiveness that none of the metrics discussed in the previous two paragraphs addresses is the ability of the system to provide correct answers that are not the top answer. This ability is not particularly important for playing Jeopardy! because a Jeopardy! player generally only provides the answer that they like best, not any other answer. However, in other applications, a question answering system may provide a list of answers with supporting evidence for each. For those applications, the ability to find the right answer and rank it near the top of the list can be extremely valuable.

Statistical Significance

A major advantage that accuracy, binary recall, and binary recall @ K have is that the analysis of statistical significance for accuracy is relatively tractable and well-understood. Specifically, comparing any of these metrics for two different configurations of Watson on the same test set of questions involves a set of paired binary outcomes (where each question constitutes a pair of outcomes: one for each configuration). We use McNemar's test to determine significance in accuracy differences; McNemar's test uses the fact that samples are paired to make much finer grained distinctions than a test (e.g., a t-test) that treats each configuration as an independent set of observations (Fleiss, 1981). If we evaluated configurations of our system by comparing the results of one configuration on one test set to a different configuration on a different test set, we would have no way of pairing questions and we would need to use an unpaired test such as a t-test. If we were doing such experiment, it would not be clear how much of any observed difference was attributable to the difference in the configurations and how much was due to the difference in the test sets; thus we would only be able to measure relatively large differences in behavior for a given test set size (or, equivalently, we would need a very large test set size to be able to measure small differences in behavior).

Comparing precision@70% between two experiments can also be viewed as two sets of binary outcomes. However, these outcomes cannot be completely paired since two configurations can result in different subsets of the questions being among the 70% for which the system has the highest confidence. We could apply a t-test to these subsets, but (as noted above) such a test provides substantially less discriminative power. Furthermore, it is not clear that these subsets would satisfy the assumptions of the t-test, which involve independent, randomly sampled observations. Presumably it would be possible to analyze the properties of this metric and construct a powerful test of significance that reflects the precision@70% sampling method and

takes into account the pairing of results for those questions that are in the top 70% for both experiments.

The remaining metrics (average precision, confidence weighted score, etc.) seem to be even more challenging to construct a custom significance test for, since they bear even less resemblance to a set of binary outcomes on independent, randomly distributed samples. Smucker, Allan, and Carterette (2007) observe that this issue arises for a variety of information retrieval metrics and propose a variety of alternatives, including Fischer's randomization test, which seems particularly well suited to computing significance across the many metrics we use. That test replaces an analytical, metric-specific approach with a statistical approach in which two samples are compared by constructing many random permutations of the two systems and seeing if the distribution of the metric for those permutations implies a significant difference between the systems. We intend to implement such a test in our ScoreEval module in future work.

Figure 4 shows a screen shot of a portion of the summary metrics produced by the WEAT, comparing the results across two different weekly runs.

More Recent Work

The methods and tools and tools described in this paper proved invaluable during the four years in which we developed the original Watson Jeopardy! system. We are continuing to pursue new improvements and enhancements. A detailed discussion of these recent developments is beyond the intended scope of this paper; we expect to describe some of them in more detail in future publications. Here we provide a just few brief examples to illustrate the directions we are going.

With the development of commercial applications based on Watson, our requirements for source code control and version release management have become more complex. To satisfy this need we have moved from SVN to the Rational Team Concert (http://www.ibm.com/software/rational/products/rtc/) lifecycle management solution, a commercial grade system with sophisticated support for issue tracking, system integration testing, version release building, and multiple work streams.

For the DeepQA Cluster, we have built a new tool called DUCC (Distributed UIMA Cluster Computing). DUCC is a generalization of the BlADE tool that handle a wider variety of jobs, moving well beyond the rather narrow domain of a batch processing questions. DUCC is now part of the Apache UIMA sandbox.

For error analysis, we are exploring richer filtering and query implementations that naturally extend with the data model and component architecture. In addition, we are extending the error analysis tools to provide a deeper look into the evidence that was used to draw some conclusion. End user applications often differ from Jeopardy! in that they require that evidence be presented to convince a user that some conclusion is correct. Since the team now needs to be more

effective in deciding what evidence to present, our developer tools and metrics are becoming better aligned with this goal.

We also constantly look for better ways to integrate and deploy all of the tools for improved ease of use and end user productivity. One clear lesson from our experience is that the methods and tools that support a research and development project must enjoy an appropriate level of investment and support if the entire project is to succeed.

Conclusions

To build Watson in under four years the DeepQA team had to build more than just Watson. We also had to develop tools and methodologies to support a metrics-driven research and development program based on many large scale experiments and detailed error analysis. In particular we had to adopt a formal system versioning and release protocol, assemble a powerful hardware environment for running many experiments from multiple users, and build powerful tools to deploy experiments and analyze results. We have described our solutions to each of these challenges, including a formal "weekly run" process for integrating new components into Watson, the DeepQA Cluster of hardware resources for running experiments with high throughput and rapid turn-around time, the BlADE cluster management tool for resource allocation and load balancing, and the WEAT error analysis tool for debugging and analyzing the gigabytes of output generated by each experiment.

Acknowledgements

The authors would like to acknowledge the contributions of all of the Watson project participants who have helped to develop and refine these tools and methods. We would particularly like to acknowledge David Ferrucci, who lead the work described here, Guillermo Averboch, who did extensive development of the error analysis tooling, and Pablo Dubuoe, who built the BlADE tool and drove much of our work on system integration and evaluation.

References

J. Chu-Carroll, J. Fan, B. K. Boguraev, D. Carmel, D. Sheinwald, C. Welty. 2012. Finding needles in the haystack: Search and candidate generation. IBM Journal of Research and Development 56(3/4).

E. A. Epstein, M. I. Schor, B. Iyer, A. Lally, E. W. Brown, J. Cwiklik. 2012. Making Watson fast. IBM Journal of Research and Development 56(3/4).

J. L. Fleiss. 1981. *Statistical Methods for Rates and Properties*. John Wiley & Sons.

D. C. Gondek, A. Lally, A. Kalyanpur, J. W. Murdock, P. Duboue, L. Zhang, Y. Pan, ZM Qiu, C. Welty. 2012. A framework for merging and ranking of answers in DeepQA. IBM Journal of Research and Development 56(3/4).

A. Kalyanpur, B. K. Boguraev, S. Patwardhan, J. W. Murdock, A. Lally, C. Welty, C., J. M. Prager, B. Coppola, A. Fokoue-Nkoutche, L. Zhang, Y. Pan, Z. M. Qiu. 2012. Structured data and inference in DeepQA. IBM Journal of Research and Development 56(3/4).

A. Lally, J. M. Prager, M. C. McCord, B. K. Boguraev, S. Patwardhan, J. Fan, P. Fodor, J. Chu-Carroll. 2012. Question analysis: How Watson reads a clue. IBM Journal of Research and Development 56(3/4).

J. W. Murdock, A. Kalyanpur, C. Welty, J. Fan, D. A. Ferrucci, D. C. Gondek, L. Zhang, H. Kanayama. 2012a. Typing candidate answers using type coercion. IBM Journal of Research and Development 56(3/4).

J. W. Murdock, J. Fan, A. Lally, H. Shima, B. K. Boguraev. 2012b. Textual evidence gathering and analysis. IBM Journal of Research and Development 56(3/4).

J. Prager, E. Brown, J. Chu-Carroll. 2012. Special Questions and Techniques. IBM Journal of Research and Development 56(3/4).

M. D. Smucker, J. Allan, and B. Carterette. 2007. A Comparison of Statistical Significance Tests for Information Retrieval Evaluation. CIKM'07.

G. Tesauro, D. C. Gondek, J. Lenchner, J. Fan, J. M. Prager. 2012. Simulation, learning, and optimization techniques in Watson's game strategies. IBM Journal of Research and Development 56(3/4).

E.M. Voorhees. 2002. Overview of the TREC 2002 Question Answering Track. TREC'02.

C. Wang, A. Kalyanpur, J. Fan, B. K. Boguraev, D. C. Gondek. 2012. Relation extraction and scoring in DeepQA. IBM Journal of Research and Development 56(3/4).