EMPIRICAL METHODS IN AN OPEN DOMAIN QUESTION ANSWERING SYSTEM

by

Sean Tate Gallagher

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Science

Charlotte

2015

Approved by:

____
Dr. Wlodek Zadrozny

____
Dr. Bojan Cukic

____
Dr. Zbigniew Ras

ABSTRACT

SEAN GALLAGHER. Empirical methods in an open domain question answering system. (Under the direction of Dr. WŁODEK W. ZADROŻNY)

Answering questions effectively involves a complex interconnected set of linguistic and statistical analysis tools, which can be difficult to investigate and evaluate on their own. To evaluate any one of them, I developed two composable, recursive linguistic annotation pipelines and many segments associated therewith in order to facilitate the generation, transformation, analysis and ranking of answers to open domain English questions. The resulting pipeline is to the author's knowledge the first open-source analog to the Watson system, and has reached 41% precision in the first rank when answering trivia questions from *Jeopardy!*.

TABLE OF CONTENTS

INTRODUCTION

In order to evaluate natural language processing methods for question answering, I developed a novel recursive linguistic analysis system targeting open domain English questions. The question answering system we built on it generated and ranked answers for *Jeopardy!* questions, and in 41% of cases, the correct answer was the first guess, which is the highest ranked answer, and what we refer to as precision in the first rank.

An application such as this, built in the style of the DeepQA system from IBM, functions generally as the following steps, each of which can be implemented multiple times and run together:

- Generate answers for a question
- Aggregate answers with similar meaning into a single answer
- Find evidence, usually passages, supporting the hypothesis that a specific answer is correct for a question
- Score and rank the answers in the context of the question and each answer's evidence

Thus, any analog to Watson will necessarily consist of multiple parts, generating, merging, and describing aspects of the text. This yields large networks of dependent annotations and a "soup" of many analysis approaches. Hence, this one application reads as a collection of small parts, and it may be easier to refer occasionally to Figures 1, 2 and 3 for a sense of what effect a module may have on the whole. Finally, in the appendix, data is available to show that a system with any one of the features described will not perform well, but a system with all of them performs noticeably better.

The purpose of the system is to answer natural language questions, without relying on a predetermined knowledge base, and without unnecessarily restricting the domain beyond what the corpora would allow. Since comparing it to Watson is a priority, the performance of the apparatus is evaluated by its responses to Jeopardy! questions. While necessarily more conservative, the overall architecture and purpose of the system is the same as Watson and the intent of this system is to be as near an analog as possible to the original system featured on *Jeopardy!*, for better scientific inquiry into its processes, and to serve as a testbed for question answering research. To the knowledge of the authors, this implementation is the first one freely available to the public.

Evaluation by trivia is responsible for some quirks which can be found in the resulting system. For example, one might not otherwise expect an anagram of a word in the question to be the correct answer. Also, few questions written by the developers mention the lexical type of the answer but this is almost universally the case for *Jeopardy!*. Also, numerical answers are much less common in *Jeopardy!* as compared to TREC. While such quirks did change the nature of the problem to some extent, it also allowed us to use a much larger answered question set against which to measure the system. In practice, between consecutive runs on one thousand randomly selected questions there was typically a one percent change in accuracy, which indicated that at least that many samples were necessary to measure the changes most modules would have on the system.

ANNOTATION PIPELINES

Much of the system is developed on the concept of an annotation-transformation pipeline, which is used to extract structured features from the text, either by adding these features as tags or by modifying the original text to match it. The apparatus for connecting these feature generating annotators is quite general in practice, but the exact setup of the pipeline can be complex. Such is why the interactions between the components is rigid and limited, to avoid making the system more complex than necessary.

2.1 Linear Pipeline

The pipeline featured in the Watson computer was replete with features and was comprised of a collection of linear pipelines connected by expansion and reduction units at specific segments. The design of the pipeline architecture, named UIMA for Unstructured Information Management Architecture, placed weight on fluid interaction between components written in different languages, and also played a role in sharding these components in a distributed system so as to reduce the time to answer a single question.

The question answering system discussed here is named Watsonsim and as such was originally intended to follow the Watson layout exactly. However, the modularity of UIMA brought with it an excess of boilerplate code and features not useful on a small scale. It is reasonably easy to guarantee the modules will all be written in the same language and run on the same virtual machine simultaneously. It is also less of a priority to minimize answering latency, which required fairly fine grained parallelism, as compared to maximizing question throughput using very coarse grained and easier to maintain parallelism.

2.2 Recursive Pipeline

In the end, the solution developed for this system was a very simple linked list of text transformation segments combined with a network of functional annotators. As such, segments which modified the original text were separated from those which merely annotated it, allowing for parallelization and caching without extensive locking.

The annotation pipeline in Watsonsim differs from the answer transformation pipeline in that it can apply to text of any type, not only answers, and that it creates results separate

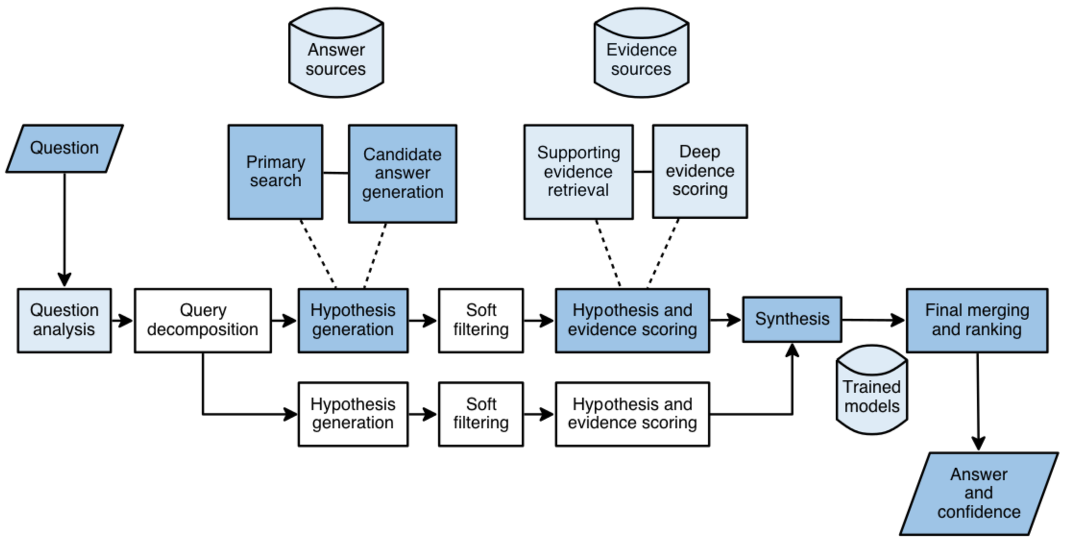


Figure 1: Linear pipeline model[1], as used in Watson [2][3]

Modules are shaded according to degree of completeness in Watsonsim, where darker shades indicate more complete. Arrows indicate the passage of a Common Analysis System (CAS) object, which is a collection of serialized annotations. Boxes, representing modules, may be realized by multiple implementations. For example, primary search is implemented five times in Watsonsim. Lines indicate some form of shared data other than a CAS. Notice that parts of the Watson pipeline could be selectively enabled and disabled at runtime. This feature (in pure white) was never implemented in Watsonsim, because recursion solved this issue in our case.

from the original answer rather than transforming the original. In practice, this pipeline is used for parsing, tokenization of a few varieties, for extracting lexical types, and any such things as do not change in the context of a question or another answer.

The sequence diagram of annotations in Figure 2 shows three levels of nested annotation. At any point, any annotation may call one another, even in a loop. Calls to the same annotators will be cached and reused to conserve time, and since caches follow the text rather than the annotator, the annotations are deleted (or at the least pended for garbage collection) promptly when the text is deleted.

Caching annotations leads to several interesting consequences. Firstly, reentrant functions cannot be cached until at least one of the functions returns successfully, so there are exceptions where an annotator can be called multiple times. Furthermore, on account of some flexibility in the Java specification on lambda expressions[4], it is necessary to create static aliases of annotators in order to guarantee caching. Otherwise, it is a compiler optimization, because the annotators might be reinstantiated in every call, leading to poor cache performance. Thus, our system uses only idempotent annotators so that performance may be lost but correctness will not. In return for this limitation, requesting annotations is a lazy process; no pipelines need be designed beforehand, and time is not wasted on unneeded or precomputed annotations.

Furthermore, because we use generic types and pass annotators instead of the desired annotation, as is done in UIMA[5], we reduce the number of new types in the program, and eliminate some wrapper types altogether. Hence, aliases included, functional annotations significantly reduce boilerplate code necessary to begin development by reducing the annotator to a single function with no mutable state.

Recursive transformations used a different abstraction, since their modifications were generally not idempotent, and would generate new or modified answers in the process, which prevents them from being trivially composable. To overcome this, transformers were chained as a linked list, and were directed so as to know only the previous components in the chain. By doing this, we retained the ability to freely reorganize the transformer stack without modifying its components, and to allow cycles, aggregration and expansion; in payment for this freedom we can only exchange questions and lists of answers between

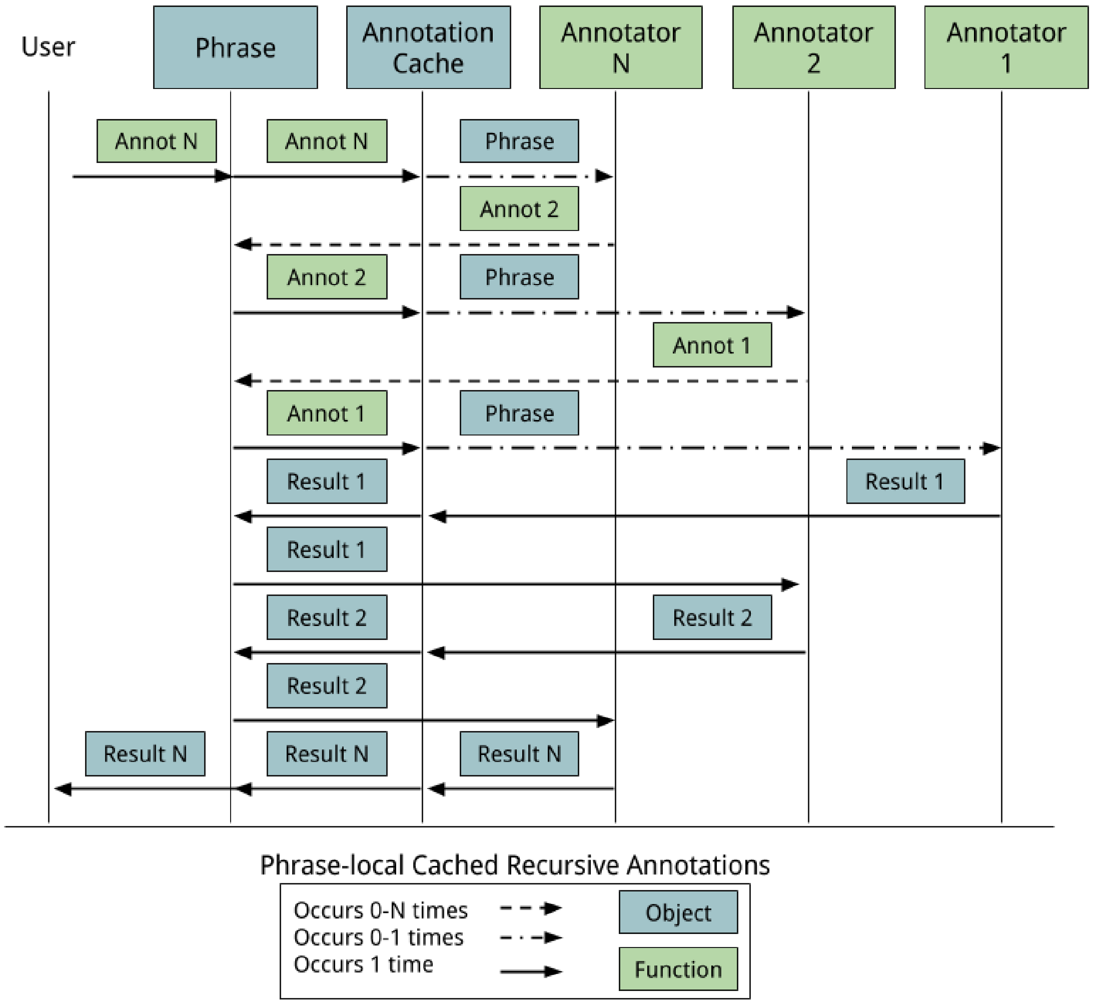


Figure 2: Watsonsim annotation sequence
In this diagram, we show three layers of recursion in our novel annotation process as a compressed sequence diagram. Notice that annotators can call each other or themselves, and can set or retrieve cached results using Phrase. As such, many annotation calls are prevented, but without having to predefine a pipeline.

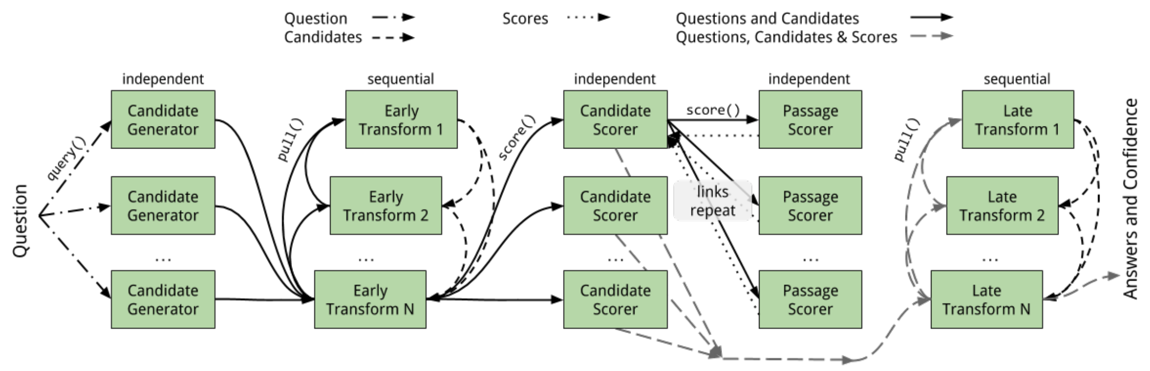


Figure 3: Recursive Watsonsim pipeline
In the novel recursive approach we developed, and now depict above, modules (in green) are connected by calls (arrows), and in the case of the transformation pipelines, are arranged in call stacks. Passage scores are called at the discretion of their corresponding answer scorers, usually answer scorers call passage scorers on all the passages in parallel and then perform aggregations on the result. Notice that transformers are aware of each previous step in the pipeline so they can ‘pull’ collections of answers through any earlier stage as necessary. Transformers could also easily implement their own transformation stacks, forming transformation trees.

transformers. We also have a fair amount of parameter and return value passing, as can be seen by comparing the density of the networks in Figures 1 and 3; this allows us to be much more specific about what can or cannot be changed by each stage. Transforms, which take questions and answers but return answers, can effect only answers. Passage scorers effect nothing; they merely return numeric values. Answer scorers can aggregate passage scores to modify answers, or they can make changes of their own.

As can be seen in Figure 3, recursive transformations account for only two stages of the question answering process. Annotations are not included in this process because they can and are called anywhere. Only search and scoring occupy special placement in the process, and this is on account of parallelization: searches have very high latency compared to most transformations, so maintaining parallelism is of greater importance than flexibility. Similarly, scoring involves sufficiently many independent modules as to be worthwhile to operate in parallel.

Recursive annotation was most useful for steps involving evidence gathering, where background information on a topic was retrieved; when new information directed much evidence toward an answer not already considered as a candidate, that new answer would be pushed back in the pipeline for examination and then included along with all the candidate answers that were considered up to that point. Most transforms were performed before scoring but transforms after scoring still served several purposes. They involved saving experiment run data and generating statistics, final confidence scoring using answer scores as features, and while it was not used as such, in principle such transforms could be used to convert the answer into a different format. For example, they could transform answers into JSON in the case of a web frontend, or phrased as an answer if it played an actual game of *Jeopardy!*.

THEMES OF ANALYSIS

Later sections describe lines of inquiry followed during the course of the experiment, but several aspects of these investigations are held in common. For example, entropy methods, keyword search, and semantic analysis will all require some form of preprocessing. Even retrieval of keyword search, which can often be reduced to a method call, is also a matter of some discussion since nearly most engines have tunable hyperparameters.

3.1 Source Acquisition

Our most significant corpora were the Wikimedia projects. The earliest incarnation of this software, Demo 1, included only a terse four megabyte snippet of Wikipedia and source acquisition consisted of reading lines in sequence. Later incarnations included several tens of thousands of articles, parsed with many errors using a few regular expressions. Later the full text of Wikipedia was loaded into a relational database and regular expressions were applied to incrementally remove irrelevant data. So many expressions were run that it was a time savings to develop a small regular expression extension for the benefit of faster evaluation.[1] Subsequent updates were completed with the WikiExtractor module from Tanl[6], which generated sufficiently clean and much more easily reproduced results.

Answered *Jeopardy!* questions were also necessary for the project, and were easily scraped from the *J! Archive* with the blessing of the administrator. Over one hundred thousand answered questions were available for querying.

3.2 Indexing

In many text analysis steps, it is either necessary or beneficial to find the units of information that can either be used as answers, such as in the case of the lexical type tables; or used as evidence to support answers, as in the case of keyword and document-topic model indices, or used to evaluate the degree to which evidence supports an answer, as in the case of word vector and entropy tables.

After several systems showed a pattern of iterating through the full corpus as a pre-processing step, an indexing pipeline was developed to complete multiple indexes in each

[1]Available at https://github.com/SeanTater/sqlite3-reutil

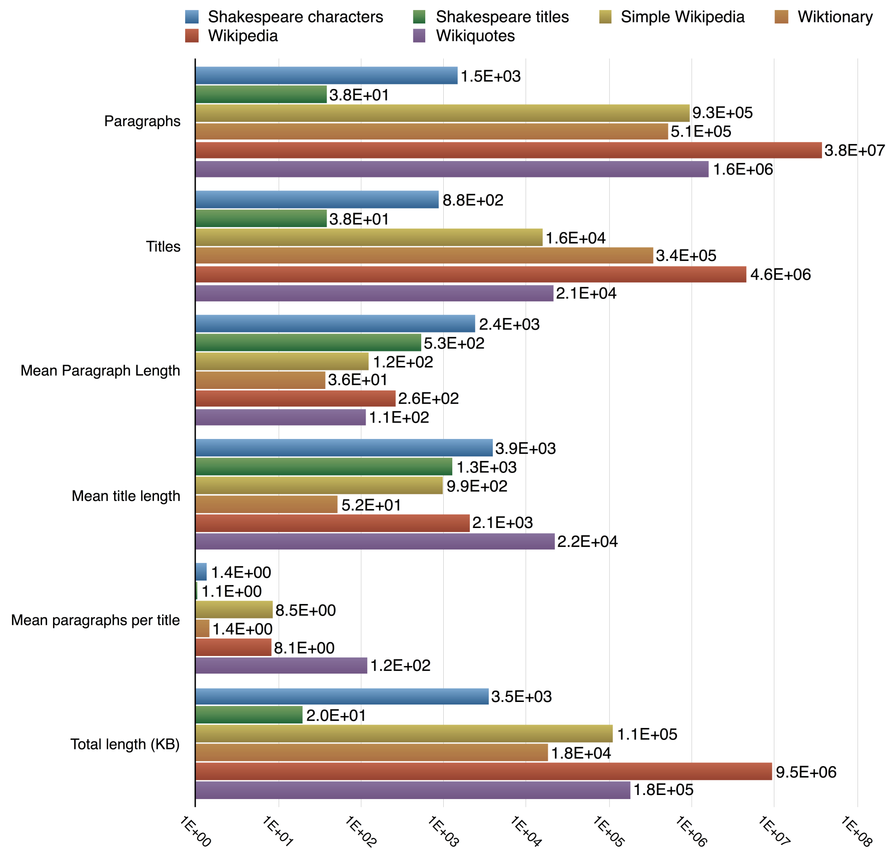


Figure 4: Corpora ingested in Watsonsim for candidate answer generation
Notice that some corpora are a great deal larger than others, and that many data sources in Watsonsim are not used directly for candidate generation, such as word embeddings or type annotations.

pass. In this way, keyword indexes with Lucene[7], search indexes with Indri[8], corpus-wide semantic dependency frequency tables with CoreNLP[9], phrase-lexical type tables, lemmatized and unlemmatized unigram and bigram frequency tables were generated. Document-topic model tables[10][11] and word vector tables[12] were generated with Gensim[13] in a separate process, as were the triplestore indices and keyword indices for DBPedia[14] entries.

There are some engineering concerns regarding indexing in practice. Indexes operating on term-document matrices, such as online Latent Semantic Indexing[10] and online Latent Dirchlet Allocation[11], can be both expensive to create and slow to transfer, which makes sharing the models awkward.

3.3 Generation

Once an index has been built, and a question has been supplied, modules are run to generate the initial candidate answers for a question. In the case of information retrieval modules such as Lucene and Index, this involves constructing and executing a query, and developing answers from the titles of the most relevant passages. In the case of an anagram solver, this involves creating likely anagrams from the question words. External searches are also implemented, to contact search engines with far larger indexes and hence higher recall.

3.4 Merging

Since multiple engines provide candidate answers for each question, and since documents in the collections are not guaranteed to have unique titles, very often multiple sources provide evidence for a single candidate answer. However, candidates with similar or identical semantics often have different surface representations. (In particular, longer answers and proper nouns often have typographical errors.) The evidence for the candidate answers should only be merged if the answers can be shown to have the same meaning. A number of methods exist, and the same techniques as were used in initial searches can be useful for determining the semantic similarity of answers. Examples include string matching; keyword matching; edit distances; weighted keyword similarity, such as by TF-IDF or by shared entropy; topical similarity such as by cosines of LSA[10] or LDA[11] vectors; or by contextual similarity with GloVe[15] or Word2Vec[12].

Deciding whether to merge two candidates is only half of the problem. When displayed to the user as the final answer, only one canonical representation of the candidate can be shown. But it is not always clear which representation of the candidates semantic to show. Each representation may be misspelled (Kerrmit), too specific (leptodactylidae) or too general (frog). Further, each score has its own weakness. Keyword matching is typically very pessimistic about relatedness, especially for single word answers which share no keywords. Edit distances may merge candidate answers, again often single words, which vary by one character but are otherwise unrelated.

In truth, the system implementation has only scratched the surface of the merging problem. For example, when multiple approaches yield different responses, the apparatus relies on heuristics where statistical analysis and large labeled corpora may be more accurate.

3.5 Transformation

Answers can be subsets of the originally suggested text for a number of reasons, and they can also be filtered on grounds of accuracy or to avoid including testing data. Also, some answers can be created from the answers themselves or from the passages that support them.

Filter	Example
Remove extraneous attribution	Leptodactylus- Wikipedia, the free encyclopedia
Remove lists	List of Amphibians
Remove URLs	https://en.wikipedia.org/wiki/Leptodactylus
Remove non-Latin text	:-)
Remove testing data	J! Archive - Show #7124, aired 2015-07-30
Remove long answers	You have the right to remain silent. Anything you say ...
Select associated lexical types	"Marilyn Monroe ... was an American actress"
Select named entities from passages	"With second husband Joe DiMaggio"

3.6 Evidence Gathering

Evidence gathering generally consists of searches based on the candidate text of an answer, and while the queries differ in that they contain a transformed answer in the query, the methods of search are otherwise unaffected and may consist of keyword searches, bayesian models, topic or context similarities.

3.6.1 Lexical Types

Most answers have a lexical type. Determining such types, as well as their equivalance classes, is a matter of much debate. The lexical type of any item is commonly given in the leading sentence of a Wikipedia article, which with low likelihood may match the surface form posed in the question. Better matching can be conducted, once again, with semantic analysis. But at some point the referent may either not be found or may be difficult to construct given the existing information.

In most cases, but particularly with organisms, types form a neatly directed acyclic graph with readily understood outlines by human authors, such as the following:

Referrer	Syntactic Referent	Semantic Referent
Leptodactylus	leptodactylid frogs	Leptodactylidae (1)
Leptodactylidae	family of frogs	Frog (2)
Frogs	group of short-bodied, tailless amphibians	Amphibian (2)
Amphibians	ectothermic, tetrapod vertebrates of the class Amphibia	Vertebrate
Vertebrates	species of animals (3)	Animal
Animals	multicellular, eukaryotic organisms	Organism
Organism	contiguous living system	System
System	set of interacting or interdependent component parts (2)	Part
Part	*does not exist* (4)	-

The existance of such structures, however, does not imply that the system can immediately make use of such constructions, for several reasons.

1. The term Leptodactylidae is a top result for a keyword search on leptodactylid frogs, but is not precisely the form given in the statement, thus incurring many searches for uncertain referents. This type hunt is done using the recursive nature of the Watsonsim search pipeline.

2. “Family”, “set”, “group”, “kind”, “type”, and “class” function as generics from which constraints can be inferred. As a matter of simple syllogism:

   $(Leptodactylus \in Leptodactylidae) \wedge (x \in Leptodactylidae \implies x \in Frogs)$
   $\implies Leptodactylus \in Frogs$

   Curiously, the surface forms are regular enough that it often suffices simply to choose the last noun of the generic lexical type’s surface form as the type of its contained elements, which is what has been implemented in Watsonsim.

3. In many cases, such as the definition of animals, types contain many important constraints, such as “multicellular,” “eukaryotic,” and “of the kingdom Animalia.” To be logically complete, any system which is to make the greatest usage of this available data should also at some point be able to infer that Leptodactylus is therefore multicellular.

4. Lexical type searches end implicitly when the referent is not found. Furthermore, it may already be of very limited use to know that Leptodactylus is a ‘Part.’

3.7 Scoring

There are three types of scorers, two of which may run in parallel. Scores computed from candidate answers render a facet or quality of the source text as a scalar value. For example, the length of the answer in characters, or in words, or perhaps the information entropy of an answer modeled as unigrams. Evidence scores, rather than measuring the text by itself, give a scalar value in the context of the candidate answer. Examples may include the number of unigrams shared between the answer and the evidence.

Confidence scoring does not involve processing any text. Rather, it concatenates the answer and evidence scores generated in the previous steps and models the likelihood of an answer being correct given the measurements made.

Two retrieval engines were used in the project: Lucene, which is an efficient Java implementation of a TF-IDF scored boolean query processing system operating on stemmed, stopword-removed bag of words models; and Indri, which is a research C++ implementation of a Bayesian language model of latent variables of documents given their terms as a bag of words. One can easily examine in the final scoring logistic regression coefficient figure

4.1 Performance

With the exception of the anagram module, all candidate answer generation eventually involves information retrieval on the main corpus, and answers that pass the first stages of merging and transformation will incur searches of their own, for supporting evidence retrieval. While filtering and merging prunes around a third to half of the candidate answers (varying widely), this still creates a ‘n+1’ query pattern, which is typically the performance bottleneck for the entire system.

After some profiling, we found that (1) our queries have a subjectively high number of terms, at 13.93 terms per query on average; that (2) all long term storage media were idle during the experiments, so the problem is not optimizing disk access, and that (3) over half of the time of

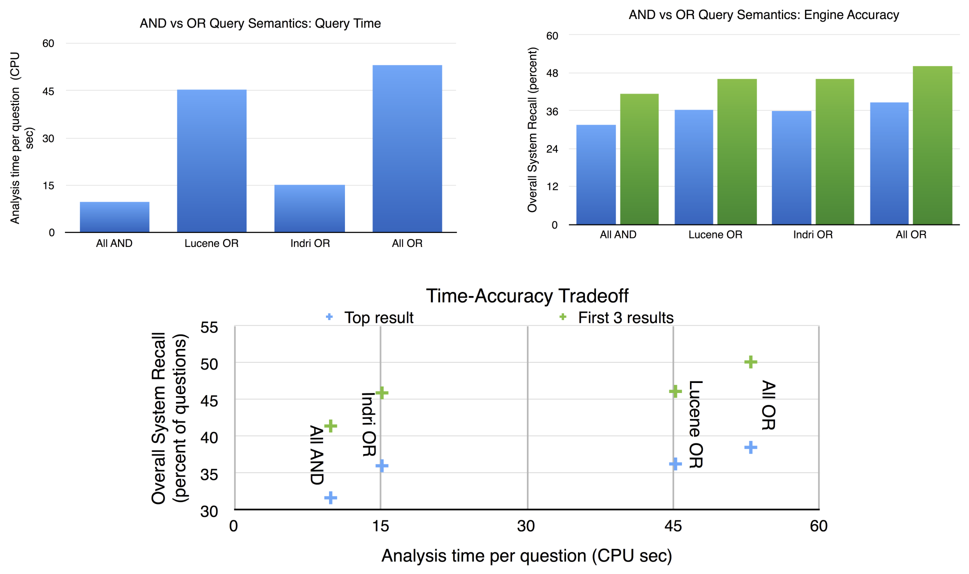


Figure 5: Tradeoff between information retrieval query time versus system accuracy The system gains approximately ten percent in both rank-one and rank-three precision when queries have the opportunity to include documents which do not contain all the required keywords, should they not all be available. However, this greatly increases the number of documents to be considered and hence causes nearly a five-fold slowdown of the system.

execution was spent in boolean scoring functions.

So to alleviate this bottleneck to a degree, we examined whether it makes sense in the context of question answering to use boolean AND instead of boolean OR. This would improve search performance since AND queries yield intersections of document sets which dwindle with increasing query length, but document sets in OR queries grow rather than shrink in the same circumstance. Furthermore, since we have subjectively long queries, we expect the trend as queries grow longer to be representative of our performance.

It is clear that the AND approach will have fewer results, although it was not clear whether the restriction that all non-stopword terms must be held in common was so strict as to eliminate correct answers and pertinent evidence as well. To test this, we examined both configurations in each of Indri (using the `band` directive) and Lucene (using the `MUST` clause type). The results had a significant negative impact on the recall of the system and was hence reverted.

CONFIDENCE SCORING

Confidence scoring is among the last steps of the question answering setup, where the numerous features discovered by answer and passage scoring modules are merged into a single measure of confidence in each candidate answer. Confidence scoring plays a major role in the effectiveness of the system, and making it work effectively is important for overall accuracy.

5.1 Representation

Representation plays an important role in defining a good machine learning solution, and in this case defining a good representation hinges on defining the learned and output data formats more specifically and making distinctions about precisely what the system inputs and outputs.

- Question level training data consists of example questions and one correct answer with the (debatable) understanding that all other answers are wrong. Question level data is from *J! Archive*.
- Answer level input data consists of many human-developed features calculated about answers generated by the system, for a particular question. The target variable, answer correctness, is determined by a test against the question level correct answer. Answer level data is normalized to a mean of 0 and standard deviation of 1 when aggregrated by question, as part of optimizing SVM accuracy[16], as mentioned later.
- Passage level input data consists of human-developed features calculated about the passages generated for a single candidate answer. There is no target variable on passage level data, and such data is simply aggregated using mean, median, max and min, where the aggregates are concatenated into the answer level input data at the point passage level data becomes available.
- System output data consists of a set of answers and their respective confidences, ranked by confidence in descending order.

Ranking alone is not sufficient in this case on the account that confidence is expected as part of the output stream. (Confidence is included so that in principle one could refrain from answering questions whose best candidate answers are uncertain.) Furthermore, in order to represent the resulting confidences as probabilities of each answer, the answer level output is processed using the softmax function aggregating across each question.

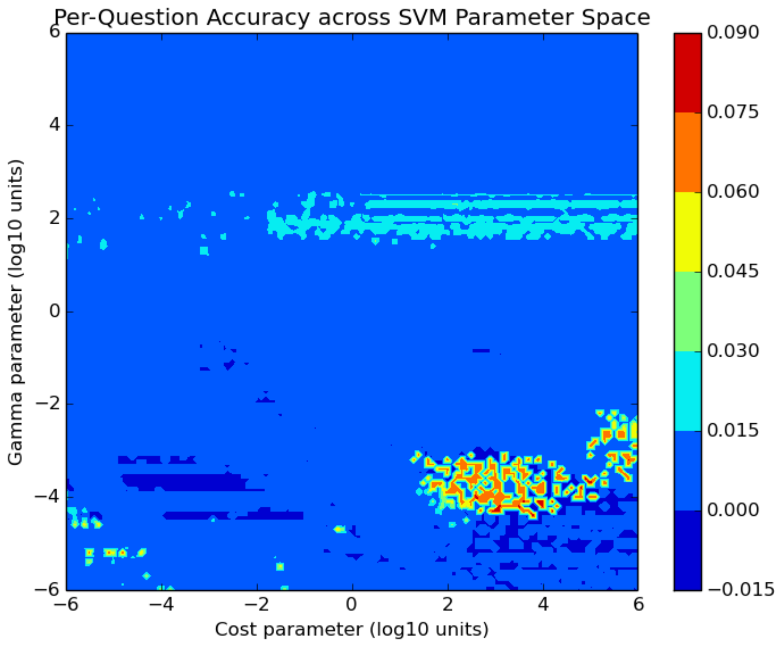


Figure 6: Per-question rank-1 recall over SVM (C, γ) parameter space on 10,000 randomly selected answers

Notice the archipelago of local maxima in the lower right of the diagram. Only two of the 463 'islands' have maximal accuracy. This instability makes finding optimal settings inconvenient. (Generating fields like these on the full set takes over two months.) The reason for this instability is uncertain and the subject of further research.

5.2 Algorithm and Parameter Tuning

Most statistical classification algorithms require some parameter tuning, but some have more complicated parameter spaces than others. This is particularly represented in the case of SVM, where we sampled the (C, γ) parameter space at $log(10)/10$ intervals covering 12 orders of magnitude in each dimension. For each sample, we trained a probability-output SVM model using 5-fold cross validation on a 66% subsample of a 10,000 answer subsample of feature-annotated candidate answers from a previous run of WatsonSim. (The probability-output SVM was implemented in `sklearn`[17], which uses libSVM[18] as a backend.) When we measured these models using 5-fold cross validation for probability outputs, against the independent answer ranking task, accurate models form an archipelago of sorts across the (C, γ) parameter space, with two global maxima and 463 local maxima, ranging across three orders of magnitude of γ, and six orders of magnitude for C. Compare this to the graph shown in Hsu at al[16], which shows around half a dozen local optima.

In practice, this means that finding a global maximum on the independent answer ranking is a very time-consuming task. Even for a ten percent subsample of a single run, representing only 100,000 answers, or around 1000 questions, one spends over four processor days to compute one (C, γ) pair of 14,400 (as in the previous experiment). Typical development may incur more than one feature change in a day, so keeping the parameters optimal is impractical. Nonetheless, peak SVM accuracies could routinely surpass those of any other measured model, and were responsible for the peak measured test accuracy of 41% precision in the top rank for randomly selected *Jeopardy!* trivia.

SHORT SEGMENTS

Many of the features necessary for an effective question answering system are actually quite simple and a great deal of their effectiveness is only appreciated in the context of the other modules. Some of the more notable subsystems that follow this pattern are entropy-based scores, length-based scores, anagram candidate answer generation, and relatedness measurement using Wikipedia redirects.

6.1 Entropy

As previously mentioned in the themes of analysis, we found while observing generated candidate answers that many highly ranking answers were either too common or too rare. (Where in the previous example "leptodactylus" is too rare, but "frog" is too common.) In order to inform the learning phase of the distinction between these answers, we generated a table of information entropy for each term in Wikipedia, and score each candidate answer by the total bits of information on a unigram model of the sentence. In logistic regression, this was found to have a slight negative coefficient, but the best model for this feature should be non-linear on the grounds that both high and low extremes are indicators of poor answer quality.

6.2 Length

In the entire system, the simplest feature generated for a candidate answer is its length in unicode codepoints. As can be seen in Appendix A, longer answers are associated with incorrect responses, though a system using logistic regression without this feature would not be far less accurate

6.3 Anagrams

Most primary answer generation modules are based on information retrieval, but one module is not: the anagram generation module. It simply suggests the set of words in a common spell-checking dictionary which are anagrams with either a single word in the question, or with a set of words in quotes. As one might expect, whitespace and nonalphabetic characters, and case are all ignored when matching anagrams.

6.4 Redirects

While extracting articles from Wikipedia, the extraction and indexing modules also make records of all redirects, organized by the titles of the source and target, all intra-wiki links by source, target, link name, and count, as well as all semantic dependency parse relations by source, target and count. The latter two returned far too many results to be used for suggestions of synonymous candidate answers, but redirects, while more sparse, gave synonyms with less noise. Unfortunately, even when redirect-generated candidate answers were tagged and then recorded by a feature (a scorer), redirects were found to be more of a hindrance than a help to accurate question answering.

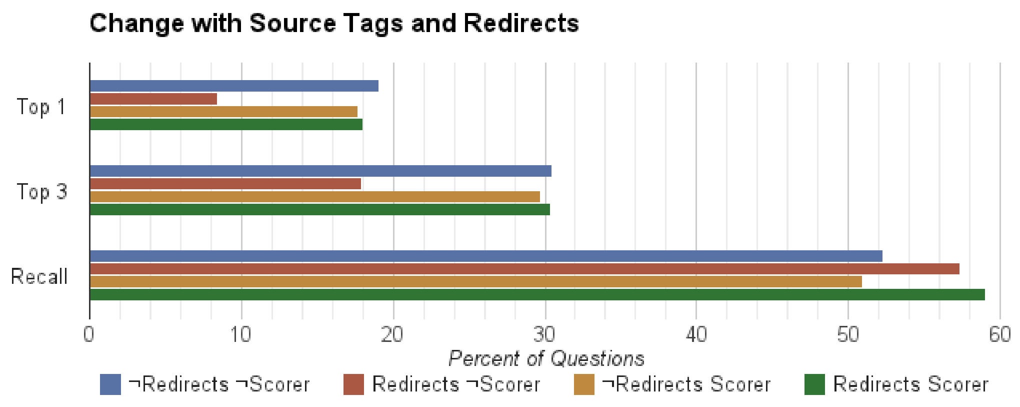


Figure 7: Effect of redirects on answering precision[19]
Redirects were found to have an overall negative effect on rank-one and rank-three precision. They introduced very much noise, and when tagged as redirects they did little harm but were not often scored favorably even when they were correct.

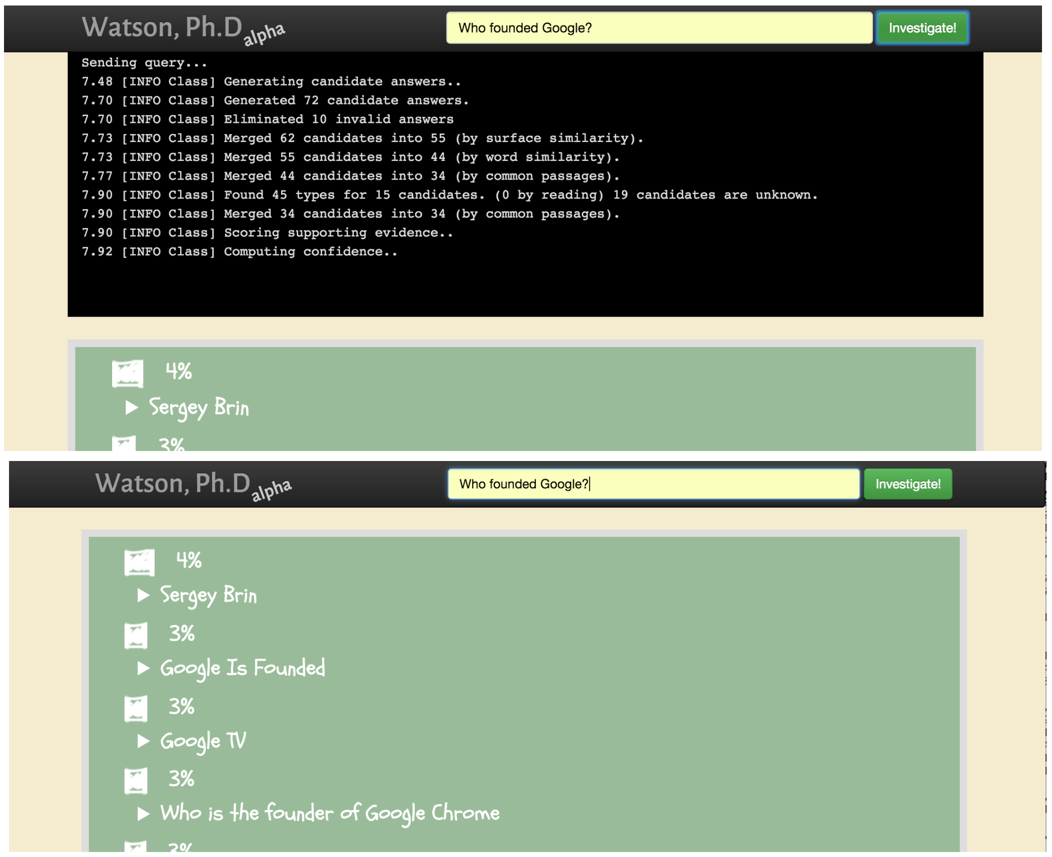
There are a number of supporting modules in the question answering framework which were surprisingly necessary for developing of the system but were not directly related to answering questions.

Web frontend

The system can take a while to setup, so we developed an online interface for the application, where users can interactively see the processing steps, resulting answers, and the sources the system reviewed.

Centralized statistics collection

The framework runs on many platforms and in multiple locations, so tests were run on the contributor's laptops, in the shared lab, and can in principle run anywhere. To account for



```
Sending query...
7.48 [INFO Class] Generating candidate answers..
7.70 [INFO Class] Generated 72 candidate answers.
7.70 [INFO Class] Eliminated 10 invalid answers
7.73 [INFO Class] Merged 62 candidates into 55 (by surface similarity).
7.73 [INFO Class] Merged 55 candidates into 44 (by word similarity).
7.77 [INFO Class] Merged 44 candidates into 34 (by common passages).
7.90 [INFO Class] Found 45 types for 15 candidates. (0 by reading) 19 candidates are unknown.
7.90 [INFO Class] Merged 34 candidates into 34 (by common passages).
7.90 [INFO Class] Scoring supporting evidence..
7.92 [INFO Class] Computing confidence..
```

Figure 8: Interactive question answering using a web application
At the time of this publication the system is available publicly online. The system also gives a great deal of information about the process both while it is being computed, and in an aggregated, detailed format when complete.

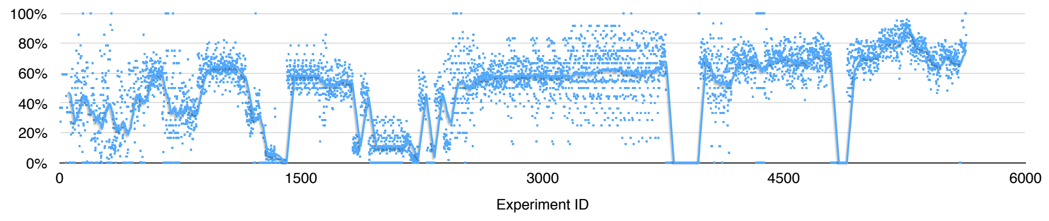


Figure 9: Precision at rank three, over time
Dots represent single experiments, and the line represents a 50-experiment moving average. Notice that the size of each experiment increased by three orders of magnitude over time, which decreased the between-experiment variance. Experiments are run on random subsets of the full *Jeopardy!* dataset, where testing and training data are chosen separately without replacement.

this decentralization we developed a small web application for users to pool their caches (to save unnecessary queries) and to report the results of their experiments.

Parallel Execution

Some aspects of the framework are designed to run in parallel, in order to improve overall question throughput in the case of large batched experiments. Firstly, any number of questions can be answered simultaneously by using separate pipelines. This is reasonably efficient because most of the memory a pipeline consumes is shared, immutable, and hence thread-safe. Multiple computers can also share the same database. The user chooses the database framework at runtime and that choice determines what configurations are safe to run. The laboratory uses SQLite over a shared network file system, which is fast but not robust against simultaneous writes, so it is shared read-only in a master-slave fashion.

Experiment Scale

The data sources involved in the project to date are not by any means “big data”; the largest data source was Wikipedia, which can be easily stored on a common laptop. However, we have taken care that our solutions would be reasonably scalable, avoiding using algorithms with complexity quadratic or worse as a function of corpus size. The only exception to such issue is topic modeling based search, which can be approximated through the use of predictive models instead of counting and matrix factorization.

That said, there is still certain impetus to mention straightforwardly the scale of the project as it stands. It involved running over 5600 experiments on an 40-core university cluster; it answered over one million questions; it generated over 44 million candidate answers; and examined just under one billion supporting passages for those answers. In addition, 145 thousand lines of code were added and removed by 17 contributors in 776 commits across two years.

COMPARISON

Watson as it existed in *Jeopardy!* was an extensive QA system of many modules. One such module allowed the machine to choose whether or not to answer a question according to the confidence the machine has calculated for the top response. The typical measure of IBM Watson's performance was hence precision at 70% confidence. Watsonsim does not have this feature and instead tries to give some answer to every question even if the confidence is low, because for the purposes of this research there is no benefit in withholding a low-confidence answer; no bets are being made. As such, precision at rank 1 in measurements of Watsonsim are comparable to precision at 100% answering rate in Watson. In such a task, Watson gave a baseline of around 15 percent precision in 2007, with the new DeepQA pipeline beginning at 30% precision in December 2007 and improving as high as 68% precision in April 2010[2]. Watsonsim lies in the middle, with around 41% precision at a forced guess.

About a year after Watsonsim started, another very active question answering project based on UIMA began, YodaQA. Its objective is to answer TREC questions and it has achieved approximately 32.6% precision at rank one[20]. The differing datasets mean the resulting accuracies are not direcly comparable, but when the DeepQA system was adapted for TREC after it ran on *Jeopardy!*, it achieved similar results for both sets [2].

A technical report in 2014 covered the state of this project much earlier in it's development[21]. Many changes have been made since the report and hence are first documented here. These include, but are not limited to the following:

- Recursive pipeline
- Cached functional annotations
- Dense vector corpus search
- DBPedia based type recognition
- MLP, SVM, and Random Forest scoring models
- Candidate answer filters
- Web frontend
- Parsing based type recognition
- Anagram candidate answer generator
- Automated parallel training, test, and index execution
- Length and entropy scorers
- Fast parsing-based scorers
- Many experimental results, such as SVM parameter spaces and feature analysis.

TEAMS AND ACKNOWLEDGMENTS

The author was the largest contributor to the project, accounting for 69% of commits, and 86% of lines of code. However, over the two year project timeline, there were many other contributors:

Contributor	Nature of Contribution
Phani Rahul	wrote the initial Lucene search apparatus
Jagan Vujjini	defined the first CSV-based pipeline
Ken Overholt	wrote a pipeline solving fill-in-the-blank questions
Adarsh Avadhani	wrote the evidence retrieval stack
Walid Shalaby	wrote the WEKA interface
Varsha Devadas	wrote an interface to OpenNLP named entity recognition
Stephen Stanton	wrote a Bing web search client
Jonathan Shuman	integrated the Lucene search with UIMA
Matt Gibson	ingested Wiktionary and parallelized statistics collection
Ricky Sanders	developed heuristics for merging candidate answers
Yeshvant Bhavnasi	wrote an acronym solver and indexed Wikipedia with Gensim LSI
Hossein Hemati	generated some statistics from autogenerated logs
David Farthing	maintained the semantic relation database
Wlodek Zadrozny	wrote the first constituency parse scorer
Robert K.S.	permitted access to J! Archive for our training and testing data

REFERENCES

[1] W. W. Zadrozny, S. Gallagher, W. Shalaby, and A. Avadhani, "Simulating ibm watson in the classroom," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, (New York, NY, USA), pp. 72–77, ACM, 2015.

[2] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty, "Building Watson: An overview of the DeepQA project," *AI Magazine*, 2010.

[3] D. Ferrucci, "Introduction to "This is Watson"," *IBM Journal of Research and Development*, vol. 56, pp. 1:1–1:15, May-June 2012.

[4] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification.* Oracle, Java SE 8 ed., February 2015.

[5] D. Ferrucci and A. Lally, "UIMa: An architectural approach to unstructured information processing in the corporate research environment," *Nat. Lang. Eng.*, vol. 10, pp. 327–348, Sept. 2004.

[6] G. Attardi, A. Cisternino, F. Formica, M. Simi, A. Tommasi, and C. Zavattari, "PIQASso: Pisa question answering system," in *Proceedings of Text Retrieval Conference (Trec-10)*, pp. 599–607, November 2001.

[7] A. Biaecki, R. Muir, and G. Ingersoll, "Apache lucene 4," in *Proceedings of the 35th International ACM SIGIR Conference*, 2012.

[8] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft, "Indri: a language model based search engine for complex queries," in *Proceedings of the International Conference on Inteligence Analysis*, 2004.

[9] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60, 2014.

[10] R. Rehurek, "Fast and faster: A comparison of two streamed matrix decomposition algorithms," *CoRR*, vol. abs/1102.5597, 2011.

[11] M. D. Hoffman, D. M. Blei, and F. Bach, "Online learning for latent dirichlet allocation," *NIPS*, 2010.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *In Proceedings of Workshop at ICLR*, 2013.

[13] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, (Valletta, Malta), pp. 45–50, ELRA, May 2010. http://is.muni.cz/publication/884893/en.

[14] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "Dbpedia a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[15] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pp. 1532–1543, 2014.

[16] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, “A practical guide to support vector classification,” tech. rep., 2010.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[19] S. Gallagher, “Wikipedia redirects,” tech. rep., University of North Carolina at Charlotte, 2015.

[20] P. Baudiš, “YodaQA: A modular question answering system pipeline,” in *Proceedings of the 19th International Student Conference on Electrical Engineering*, POSTER 2015, CTU, 2015.

[21] S. Gallagher, W. W. Zadrozny, W. Shalaby, and A. Avadhani, “Watsonsim: Overview of a question answering engine,” tech. rep., University of North Carolina at Charlotte, December 2014.

APPENDIX: FEATURE ANALYSIS

Developing good features for text requires some degree of analysis on the quality of the results obtained from the features on a large set of texts. Non-linear models such as random forests and support vector machines are have the best classification accuracy, but the results would be difficult to interpret on a feature-by-feature basis so we chose to use logistic regression in scikit-learn and take the magnitudes of the learned coefficients as indicators of the descriminative quality of a feature.

Each feature is listed with its internal codename alongside its learned coefficient. In the middle is displayed the overall question answering accuracy of a system trained on all features except one (a so-called 'knockout') and the rightmost column shows a system trained on that feature alone. In all cases the result of a one-feature classification was comparable to or worse in accuracy to a system which simply guessed every answer was wrong (named 'all 0s') . However, many features contributed only a tiny amount on their own and a system without them is nearly indistinguishable from the best system developed at the point this experiment was made.

The codenames are explained below:

Answer Length

The length of the candidate answer in Unicode codepoints.

Answer POS

Whether the candidate answer is either a noun or a noun phrase, represented as 1 or 0.

Bing Answer Present

Whether the candidate answer was originally generated by Bing, represented a 1 or 0

Bing Answer Rank

The mean rank of the answer in Bing search results, if 'Bing Answer Present', counted from 0, where 0 is the first result of the first page of results. If not 'Bing Answer Present', then 'Bing Answer Rank' = -1. Keep in mind that the candidate ranks and scores are arithmetic means, and must be so because multiple candidate answers (and hence their associated search results) can be merged.

Common Constituents Max
Common Constituents Mean
Common Constituents Median
Common Constituents Min

How many subtrees the constituency parses of both the question and the answer have in

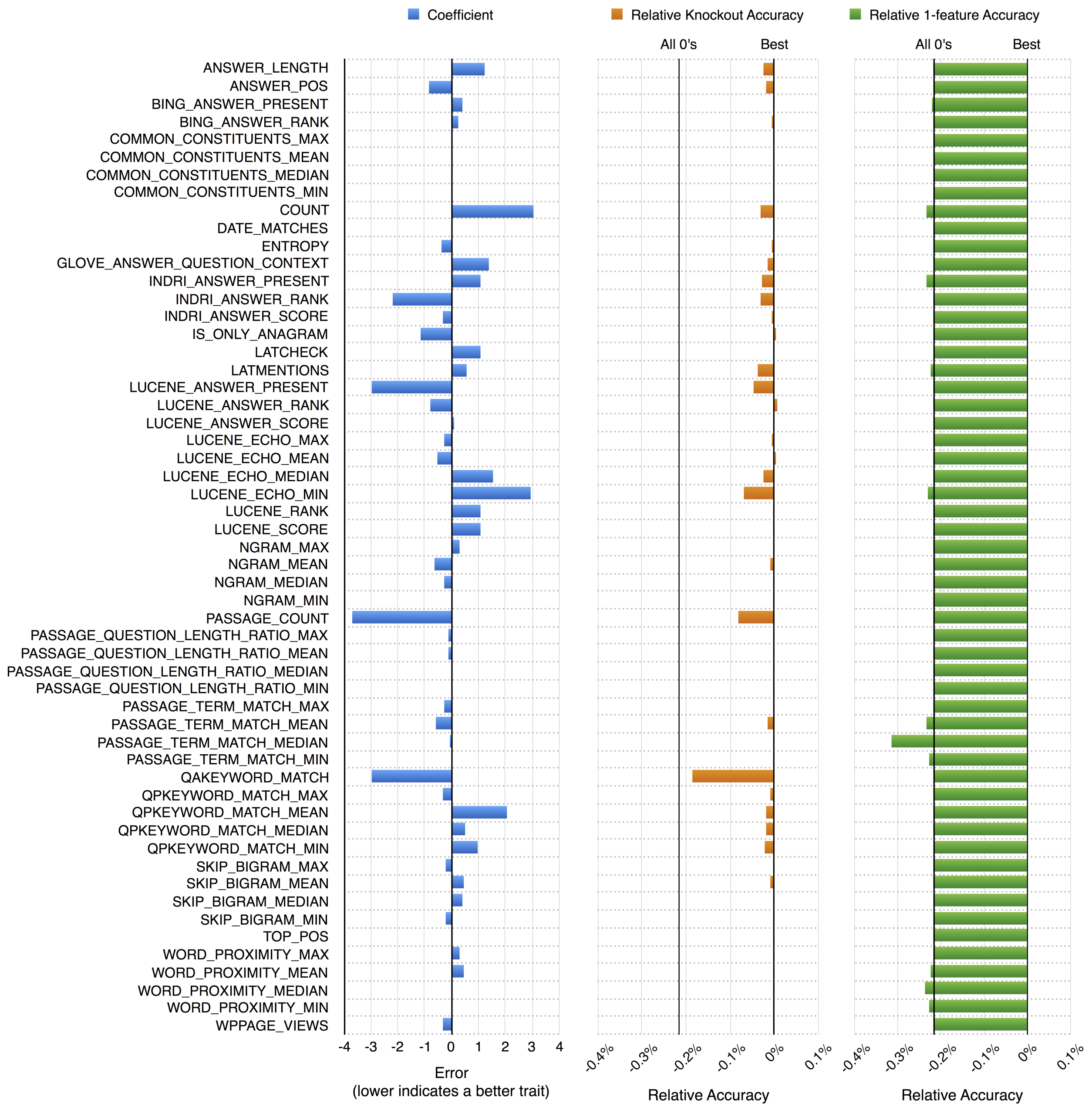


Figure 10: Feature performance using logistic regression
Blue columns depict the magnitude and sign of the coefficient associated with the feature in logistic regression, lower coefficients indicate that the presence or strength of a feature is greater evidence toward the correctness of the candidate answer. Orange columns depict how much of the per-answer binary accuracy of the system would be lost if the feature were removed. Green columns depict the same measure if all other features were removed.

common with each other, where equality is defined as the same keyword and the same part of speech.

Count

How many copies of the same candidate answer are present in one answer; this is not the same as the number of times the answer has been merged only because it possible to merge two answers which have each themselves already been merged.

Date Matches

Whether any date mentioned in the question and answer match each other, should a pair of such dates have been found.

Entropy

The information entropy of the candidate answer, measured in bits, according to the frequency of unigrams found in the entire source corpus. Punctuation, capitalization, and any other unknown words are ignored and do not contribute to the entropy.

Glove Answer Question Context

The cosine similarity between the question and the answer context vectors, where the context vectors are generated as the elementwise arithmetic mean, elementwise log-product, or elementwise geometric mean of the context vectors associated with the case-sensitive unigrams of the question or answer, ignoring punctuation and whitespace. In the case given above, the merging function was the arithmetic mean, and the source of context vectors was the GloVe 300 dimension Common Crawl corpus.

Indri Answer Present

Whether the candidate answer was originally generated by the Indri engine.

Indri Answer Rank

If 'Indri Answer Present', the mean rank of the Indri search results associated with the candidate answer, otherwise -1.

Indri Answer Score

If 'Indri Answer Present', the mean score Indri reports for the search results associated with the candidate answer, otherwise -1.

Is Only Anagram

Whether the candidate answer was originally generated by the anagram generator *and not* 'Bing Answer Present' *and not* 'Indri Answer Present' *and not* 'Lucene Answer Present'. This score was introduced to prevent the introduction of many low quality answers for questions containing terms with many anagrams.

LATCheck

Whether the lexical type of the candidate answer matches the type found in the question, if types have been associated with both the question and answer. Cases with unassociated questions and answers are represented as unmatched.

LATMentions

The number of distinct lexical types associated with a candidate answer

Lucene Answer Present

Whether the answer was originally generated by the Lucene engine.

Lucene Answer Rank

If 'Lucene Answer Present', the rank of the search result associated with this candidate answer, otherwise -1.

Lucene Answer Score

The score assigned by Lucene to the first search result associated with this candidate answer.

Lucene Echo Max
Lucene Echo Mean
Lucene Echo Median
Lucene Echo Min

The scores associated with the Lucene passages returned from the supporting evidence retrieval portion of the pipeline.

Lucene Rank
Lucene Score

Deleted scores replaced by 'Lucene Echo' variants, kept for compatibility of models.

NGram Max
NGram Mean
NGram Median
NGram Min

Count of stemmed, stopword-filtered trigrams in common between a candidate answer and its supporting evidence. (The codename stems from its flexibility for any N, however 2 was already provided by skip-bigrams and 4 was deemed too rare.)

Passage Count

Total supporting passages found for this candidate answer. Because passage searches are limited at a constant value, 'Passage Count' is often equal to a constant multiplied by 'Count'. However this is not the case if a query yields very few results (as in the case of AND queries).

Passage Question Length Ratio Max
Passage Question Length Ratio Mean
Passage Question Length Ratio Median
Passage Question Length Ratio Min

The ratio of lengths of the passage over the question

Passage Term Match Max
Passage Term Match Mean
Passage Term Match Median
Passage Term Match Min

The number of stemmed, stopword-filtered unigrams shared between the question and passage

QAKeyword Match

The number of stemmed, stopword-filtered unigrams shared between the question and answer, normalized to the length of the question.

QPKeyword Match Max
QPKeyword Match Mean
QPKeyword Match Median
QPKeyword Match Min

A replica of 'Passage Term Match' normalized to the length of the question.

Skip Bigram Max
Skip Bigram Mean
Skip Bigram Median
Skip Bigram Min

The number of shared bigrams, allowing one unmatched word between terms, where terms are stemmed and filtered for stopwords.

Top POS

A hash of the question's part of speech as determined by the root of its constituency parse.

Word Proximity Max
Word Proximity Mean
Word Proximity Median
Word Proximity Min

The average distance between stemmed, stopword-filtered unigram matches, when comparing questions to passages

WP Page Views

The number of pageviews of the Wikipedia article by the same name as the candidate answer, if such an article exists, where pageviews are averaged across 100 randomly selected hourly logs of Wikipedia traffic since 2007.