YodaQA: A Modular Question Answering System Pipeline

Petr Baudíš
Dept. of Cybernetics, Czech Technical University, Technická 2, 166 27 Praha, Czech Republic
baudipet@fel.cvut.cz

Abstract. This is a preprint, submitted on 2015-03-22. Question Answering as a sub-field of information retrieval and information extraction is recently enjoying renewed popularity, triggered by the publicized success of IBM Watson in the Jeopardy! competition. But Question Answering research is now proceeding in several semi-independent tiers depending on the precise task formulation and constraints on the knowledge base, and new researchers entering the field can focus only on various restricted sub-tasks as no modern full-scale software system for QA has been openly available until recently.

By our YodaQA system that we introduce here, we seek to reunite and boost research efforts in Question Answering, providing a modular, open source pipeline for this task — allowing integration of various knowledge base paradigms, answer production and analysis strategies and using a machine learned models to rank the answers. Within this pipeline, we also supply a baseline QA system inspired by DeepQA with solid performance and propose a reference experimental setup for easy future performance comparisons.

In this paper, we review the available open QA platforms, present the architecture of our pipeline, the components of the baseline QA system, and also analyze the system performance on the reference dataset.

Keywords
Question answering, information retrieval, information extraction, linked data, natural language processing, Apache UIMA, software engineering.

1. Introduction

We consider the Question Answering problem — a function of unstructured user query that returns the information queried for. This is a harder problem than a linked data graph search (which requires a precisely structured user query) or a generic search engine (which returns whole documents or sets of passages instead of the specific information). The Question Answering task is however a natural extension of a search engine, as currently employed e.g. in Google Search [22] or personal assistants like Apple Siri, and with the high profile IBM Watson Jeopardy! matches [10] it has become a benchmark of progress in AI research. As we are interested in a general purpose QA system, we will consider an “open domain” general factoid question answering, rather than domain-specific applications, though we keep flexibility in this direction as one of our goals.

Diverse QA system architectures have been proposed in the last 15 years, applying different approaches to information retrieval. A full survey is beyond the scope of this paper, but let us outline at least the most basic choices we faced when designing our system.

Perhaps the most popular approach in QA research has been restricting the task to querying structured knowledge bases, typically using the RDF paradigm and accessible via SPARQL. The QA problem can be then rephrased as learning a function translating free-text user query to a structured lambda expression or SPARQL query. [3] [5] We prefer to focus on unstructured datasets as the coverage of the system as well as domain versatility increases dramatically; building a combined portfolio of structured and unstructured knowledge bases is then again an obvious extension.

When relying on unstructured knowledge bases, a common strategy is to offload the information retrieval on an external high-quality web search engine like Google or Bing (see e.g. the Mulder system [15] or many others). We make a point of relying purely on local information sources. While the task becomes noticeably harder, we believe the result is a more universal system that could be readily refocused on a specific domain or proprietary knowledge base, and also a system more appropriate as a scientific platform as the results are fully reproducible over time.

Finally, a common restriction of the QA problem concerns only selecting the most relevant answer-bearing passage, given a tuple of input question and set of pre-selected candidate passages [23]. This Answer Sentence Selection task is certainly worthwhile as a component of a QA system but does not form a full-fledged system by itself. It may be argued that returning a whole passage is more useful for the user than a direct narrow answer, but this precludes any reasoning or other indirect answer synthesis on the part of the system, while the context and supporting evidence can be still provided by the user interface. Direct answer output may be also used in a more general AI reasoning engine.

In this paper, we present our open source Question Answering system brmson™ YodaQA. This is not the
only open source QA framework currently available, but we found our goals not entirely compatible with any of the other systems we investigated. Specifically, we aim to build a system that (A) provides an end-to-end, universal pipeline integrating different knowledge base paradigms in a modular fashion; (B) is domain flexible and could cater even to the long tail of rarer question subjects, therefore has minimum of fixed categories and hand-coded rules.

In contrast, the classic QA system OpenEphyra [21] operates on the basis of fixed question categories with hand-crafted rules, and puts emphasis on querying web search engines. The OAQA initiative [12] has developed a basic QA framework, but does not provide an end-to-end pipeline and its usage of UIMA has in our opinion severe design limitations (see below). The WatsonSim system [13] has begun developing independently during the course of our own work and it works on Jeopardy! statements rather than questions.

Jacana [24] [25] is a promising set of loosely coupled QA-related methods and algorithms, focused on machine learning of textual entailment. It is not meant to be a full QA framework and using it as an end-to-end pipeline is not straightforward, but integration of the Jacana implementation as modules in YodaQA is our long-term plan.

OpenQA [1] is a recently introduced end-to-end QA pipeline platform also developed independently during the course of our work, and shares our goal of a common research platform in the field. However, the approach is very different, as OpenQA is more of a portfolio-style engine with mostly independent pipelines which have their candidate answers combined, while YodaQA emphasizes modularity on the pipeline stage level, with e.g. all answer producers sharing a common answer analysis stage.

The rest of the paper is structured as follows. We outline the general structure of our framework in Sec. 2. We then describe the current reference implementation of the pipeline components in Sec. 3. We propose a common experimental setup and analyze the system performance in Sec. 4. We conclude with a summary of our contributions and an outline of future extensions in Sec. 5.

2. YodaQA Pipeline Architecture

The goals for our system brmson™ YodaQA are to provide an open source Question Answering platform that can serve both as scientific research testbed and a practical system. The pipeline is implemented mainly in Java, using the popular Apache UIMA framework [11]. Extensive support tooling is included within the package.

Unlike OAQA, in YodaQA each artifact (question, search result, passage, candidate answer) is represented as a separate UIMA CAS, allowing easy parallelization and easy leverage of pre-existing NLP UIMA components; we also put emphasis on aggressive compartmentalization of different tasks to interchangeable annotators rather than using UIMA just for high level flow and annotation storage.

The framework is split in several Java packages: io package takes care of retrieving questions and returning answers, pipeline contains classes of the general pipeline stages, analysis contains modules for various analysis steps, provider has interfaces to various external resources and flow carries UIMA helper classes.

The system maps an input question to ordered list of answer candidates in a pipeline fashion, with the flow as in Fig. 1, encompassing the following stages:

- **Question Analysis** extracts natural language features from the question text and produces QA features based on them (clues, type, etc.).
- **Answer Production** generates a set of candidate answers based on the question. Typically, this happens by performing a Primary Search in the knowledge bases according to the question clues, and either directly using search results as candidate answers or filtering relevant passages from these (the Passage Extraction) and generating candidate answers from picked passages (the Passage Analysis).
- **Answer Analysis** generates various answer features based on detailed analysis (most importantly, type determination and coercion to question type).
- **Answer Merging and Scoring** consolidates the set of answers, removing duplicates and using a machine learned classifier to score answers by their features.
- **Successive Refining** (optional) prunes the set of questions in multiple phases, interjecting some extra tasks (evidence diffusion and gathering additional evidence).

The basic pipeline flow is much inspired by the DeepQA model of IBM Watson [9]. Throughout the flow, answer features are gradually accumulated and some results of early flow stages (especially the question analysis) are carried through the rest of the flow.

3. YodaQA Reference Pipeline

The particular Question Answering problem considered in the reference pipeline is finding a precise (narrowly phrased) answer to a naturally phrased English question, based on both unstructured (English Wikipedia, enwiki) and structured (DBpedia [17], Freebase [4]) knowledge bases.

In our pipeline, we build on existing third-party NLP analysis tools, in particular Stanford CoreNLP (Segmenter, POS-Tagger, Parser) [18] [6], OpenNLP (Segmenter, NER)
3.1. Question Analysis

During question analysis, produce a part-of-speech tagging and dependency parse of the question text, recognize named entities and, roughly inspired by the DeepQA system [16], heuristically generate several QA features: Clues, Focus, and LAT.

**Clues** represent keywords in the question that determine its content and are used to query for candidate answers. Clues based on different question components are assigned different weight (used in search retrieval and passage extraction, determined empirically) — in ascending order, all noun phrases, noun tokens and the selection verb (SV); the LAT (see below); named entities; the question sentence subject (determined by dependency parse). If the clue text corresponds to an enwiki article name or redirect alias, its weight is boosted and it is flagged as a concept clue.

**Focus** is the center point of the question sentence indicating the queried object. Six simple hand-crafted heuristics extract the focus based on the dependency parse. “name of —” constructions are traversed.

**LAT** (Lexical Answer Type) describes a type of the answer that would fit the question. This type is not of a predefined category but may be an arbitrary English noun, like in the DeepQA system. [20] The LAT is derived from the focus, except question words are mapped to nouns (“where” to “location”, etc.) and adverbs (like “hot”) are nominalized (to “temperature”) using WordNet relations.

3.2. Unstructured Answer Sources

The primary source of answers in our QA system is keyword search in free-text knowledge base, in our default setting the enwiki. While the information has no formal structure, we take advantage of the organization of the enwiki corpus where entity descriptions are stored in articles that bear the entity name as title and the first sentence is typically an informative short description of the entity. Our search strategies are analogous to basic DeepQA free-text information retrieval methods [7]. We use the Apache Solr3 search engine (frontend to Apache Lucene).

**Title-in-clue search** [7] looks for the question clues in the article titles, essentially aiming to find articles that describe the concepts touched in the question. The first sentence of the top six articles (which we assume is its summary) is then used in passage analysis (see below).

**Full-text search** [7] runs a full-text clue search in the article texts and titles, considering the top six results. The document texts are split to sentences which are treated as separate passages and scored based on sum of weights of

---


2Sometimes, different pipeline components default to different NLP backends to perform the same task, e.g. segmentation, based on empirically determined best fit.

clues occurring in each passage\(^4\); the top three passages from each document are picked for passage analysis.

**Document search** \([7]\) runs a full-text clue search in the article texts; top 20 article hits are then taken as potential responses, represented as candidate answers by their titles.

**Concept search** retrieves articles whose titles (or redirect aliases) exactly match a question clue. The first sentence and also passages extracted as in the full-text search are used for passage analysis.

Given a picked passage, the **passage analysis** process executes an NLP pipeline and generates candidate answers; currently, the answer extraction strategy entails simply converting all named entities and noun phrases to candidate answers. Also, object constituents in sentences where subject is the question LAT are converted to candidate answers.

### 3.3. Structured Answer Sources

Aside of full-text search, we also employ structured knowledge bases organized in RDF triples; for each concept clue, we query for predicates with this clue as a subject and generate candidate answers for each object in such a triple, with the predicate label seeded as one LAT of the answer.

In particular, we query the DBpedia **ontology** (curated) and **property** (raw infobox) namespaces and the Freebase RDF dump. For performance reasons, we limit the number of queried Freebase topics to 5 and retrieve only 40 properties per each; due to this limitation, we have manually compiled a blacklist of skipped “spammy” properties based on past system behavior analysis (e.g. location’s **people born here** or music artist’s **track**).

### 3.4. Answer Analysis

In the answer analysis, the system takes a closer look at the answer snippet and generates numerous features for each answer. The dominant task here is type coercion, i.e. checking whether the answer type matches the question LAT.

The answer LAT is produced by multiple strategies:

- Answers generated by a named entity recognizer have LAT corresponding to the triggering model; we use stock OpenNLP NER models date, location, money, organization, percentage, person and time.
- Answers containing a number have a generic **quantity** LAT generated.
- Answer focuses (the parse tree roots) are looked up in WordNet and **instance-of** pairs are used to generate LATs (e.g. Einstein is **instance-of** scientist).
- Answer focuses are looked up in DBpedia and its ontology is used to generate LATs.
- Answers originating from a structured knowledge base carry the property name as an LAT.

Type coercion between question and answer LATs is performed using the WordNet hypernymy relation — i.e. **scientist** may be generalized to person, or length to quantity. We term the type coercion score **WordNet specificity** and exponentially decrease it with the number of hypernymy traversals required. Answer LATs coming from named entity recognizer and quantity are not generalized. We never generalize further once within the noun. **Tops** WordNet domain and based on past behavior analysis, we have manually compiled a further blacklist of WordNet synsets that are never accepted as coercing generalizations (e.g. trait or social group).

The generated features describe the origin of the answer (data source, search result score, clues of which type matched in the passage, distance-based score of adjacent clue occurrences, etc.), syntactic overlaps with question clues and type coercion scores (what kind of LATs have been generated, if any type coercion succeeded, what is the WordNet specificity and whether either LAT had to be generalized).

### 3.5. Answer Merge-and-Score

The merging and scoring process also basically follows a simplified DeepQA approach \([14]\). Candidate answers of the same text (up to basic normalization, like the- removal) are merged; element-wise maximum is taken as the resulting answer feature vector (except for the #occurrences feature, where a sum is taken). To reduce overfitting, too rare features are excluded (when they occur in less than 1% questions and 0.1% answers).

Supplementary features are produced for each logical feature — aside of the original value, a binary feature denoting whether a feature has not been generated and a value normalized over the full set of answers so that the distribution of the feature values over the answer has mean 0 and standard deviation 1. The extended feature vectors are converted to a score \(s \in [0, 1]\) using a logistic regression classifier. The weight vector is trained on the gold standard of a training dataset, employing L2 regularization objective. To strike a good precision-recall balance, positive answers (which are about \(p = 0.03\) portion of the total) are weighed by \(0.5/p\).

### 3.6. Successive Refining

The pipeline contains support for additional refining and scoring phases. By default, after initial answer scoring, only the top 25 answers are kept with the intent of reducing noise for the next answer scoring classifier. Answers are compared and those overlapping syntactically (prefix, suffix, or substring aligned with sub-phrase boundaries) are subject
to evidence diffusion where their scores are used as features of the overlapping answers. Another answer scoring would be then performed, and the answer with the highest score is then finally output by the system.\(^6\)

However, while we have found these extra scoring steps beneficial with weaker pipelines (in particular without the clue overlap features), in the final pipeline configuration the re-scoring triggers significant overfitting on the training set and we therefore ignore the successive refining stage in the benchmarked pipeline.

4. Performance Analysis

As we present performance analysis of our system, we shall first detail our experimental setup; this also includes discussion of our question dataset.

Then, we proceed with the actual results — we measure the recall of the system (whether a correct answer has been generated and considered, without regard to its score) and accuracy-at-one (whether the correct answer has been returned as the top answer by the system). We find this preferable to typical information retrieval measures like MRR or MAP since in many applications, eventually only the single top answer output by the system matters; however, we also show the mean reciprocal rank for each configuration and discuss the rank distribution of correct answers.

Aside of the performance of the default configuration, we also discuss scaling of the system (extending the allotted answer time) and performance impact of its various components (hold-out testing).

4.1. Experimental Setup

Our code is version tracked in a public GitHub repository \(\text{https://github.com/brmson/yodaqa,}\) and the experiments presented here are based on commit \(\text{f6c0cf6} \) (tagged as v1.0). The quality of full-text search is co-determined by Solr version (we use 4.6.0) and models of the various NLP components which are brought in by DKPro version 1.7.0. As for the knowledge bases, we use enwiki-20150112, DBpedia 2014, Freebase RDF dump from Jan 11 2015, and WordNet 3.1. Detailed instructions on setting up the same state locally (including download of the particular dump versions and configuration files) are distributed along the source code.

As a benchmark of the system performance, we use a dataset of 430 training and 430 testing open domain factoid questions. (For system development, exclusively questions from the training set are used.) This dataset is based on the public question answering benchmark from the main tasks of the TREC 2001 and 2002 QA tracks with regular expression answer patterns\(^7\) and extended by questions asked to a YodaQA predecessor by internet users via an IRC interface. This dataset was further manually reviewed by the author, ambiguous or outdated questions were removed and the regex patterns were updated based on current data. We refer to the resulting 867 question dataset as curated and randomly split it to the training and testing sets.\(^8\)

An automatic benchmark evaluation system is distributed as part of the YodaQA software package. The system evaluates the training and test questions in parallel and re-trains the machine learning models before scoring the answers. Therefore, in all the modified system versions considered below, a model trained specifically for that version is used for scoring answers.

Our benchmark is influenced by two sources of noise. First, the answer correctness is determined automatically by matching a predefined regex, but this may yield both false positives and false negatives.\(^9\) Second, during training the models are randomly initialized and therefore their final performance on a testing set flutters a little.

4.2. Benchmark Results

Benchmark results over various pipeline configurations are laid out in Fig. 2. Aside of the general performance of the system, it is instructive to look at the histogram of answer ranks for the default pipeline, shown in Fig. 3. We can observe that while accuracy-at-one is 32.6\%, accuracy-at-five is already at 52.7\% of test questions.

The information retrieval parameters of the default pipeline are selected so that answering a question takes about

\(^6\)There is also experimental support for additional evidence gathering phase, where the top 5 answers are looked up using the full-text search together with the question clues, and the number and score of hits are used as additional answer features and final answer rescoring is performed. Nevertheless, we have not found this approach effective.


\(^8\)The remaining 7 questions are left unused for now.

\(^9\)For example numerical quantities with varying formatting and units are notoriously tricky to match by a regular expression.
### 4. Comparison and Performance Evaluation

We also benchmarked performance with various components of the pipeline disabled. We can see that the full-text and structured knowledge bases are complementary to a degree, but the full-text base is eventually a much stronger answer source for our system. Type coercion and detection of the concept clues in the question are both important heuristics for our QA system.

Comparison of performance across multiple systems is currently non-trivial, unfortunately, as there is no universally agreed experimental setup so far and not even published results on the TREC datasets from the years we use are readily available. OpenEphyra seemed to typically achieve accuracy-at-one of “above 25%” on the TREC datasets including our years according to [21]. In the Answer Sentence Selection task [23], Jacana and similar textual entailment systems are reported\(^\text{10}\) to achieve MRR around 0.750 but this task represents a significant restriction upon the general end-to-end QA pipeline.

### 5. Conclusion

We have described a modular question answering system which can be used for effective mixing of both structured and unstructured knowledge bases, is domain-flexible and highly amenable to further extensions in various stages of its pipeline. We put emphasis on universal, machine learned methods and employ only a very limited amount of hand-crafted heuristics.

Meanwhile, the system is already demonstrating a reasonable open domain factoid question answering performance, being able to answer a third of the testing set questions correctly, over half of the questions in top five answers, and considering the correct answer for just about 80% of the questions; we have also shown a head-room for further performance scaling by extending the available computational resources.

Our system is made available as open source under the highly permissive Apache Software Licence\(^\text{11}\) and available for research collaboration on the GitHub social software hosting site. We hope for our system to become an universal research platform for testing of various question answering related strategies not only in isolation but also measuring their effect in a real-world high performance end-to-end pipeline. We also hope our work helps to clarify which of the numerous DeepQA contributions are most essential to a minimal working modern QA system.

### 5.1. Future Work

We present here just the first version of a system that could be improved in many desirable ways. The software platform itself would benefit in particular from a multi-threaded pipeline flow driver, a sophisticated user interface showing the generated answer in context and hypertext, and sped up benchmarking by caching information retrieval results (parsed picked passages) across runs.

Regarding algorithmic improvements, the most obvious candidates seem a more sophisticated answer extraction strategy (e.g. employing methods introduced in Jacana seems as a natural fit) and more reliable type coercion as a negative evidence source; we also hope that distributed rep-

---

\(^{10}\)See the ACL Wiki topic Question Answering (State of the art).

\(^{11}\)The GPL licence applies in case Stanford CoreNLP components are employed.
resentations might improve both areas. We feel that without further large effort in feature engineering, logistic regression is inadequate for scoring answers and we are seeing promising preliminary results from employing random forests instead.

Analysis and model training would be improved with larger benchmark datasets with more sophisticated correct answer verification. Some sub-tasks like type coercion would benefit from specialized datasets, and passage extraction scoring could be tuned on the Answer Sentence Selection dataset.

A robust heuristic for additional evidencing of most promising answers remains as an open problem in our system. While the natural idea of additional fulltext search combining the question and answer has been beneficial with a less sophisticated pipeline, it does not improve performance with our current featureful pipeline.

Acknowledgements

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/194/OKH3/3T/13. The research was co-supervised by Dr. Jan Šedivý and Dr. Petr Pošik.

References


About Author...

Petr Baudiš has received his Masters degree in Theoretical Computer Science at the Charles University in Prague. So far, he has been working chiefly in the fields of Computer Go and Continuous Black-box Optimization. Currently, he is a PhD student at the Czech Technical University and his interests are extending also to the topics related to information extraction from unstructured text corpora.
Fig. 4. A sample of the pipeline process when (correctly) answering a training question. ! indicates particularly distinguishing features.

Fig. 5. A sample of the pipeline process when (not quite correctly) answering a training question. ! indicates particularly distinguishing features.