Question Quality Improvement: Deep Question Understanding for Incident Management in Technical Support Domain

Anupama Ray,[1] **Csaba Hadhazi,**[2] **Pooja Aggarwal,**[1] **Gargi Dasgupta,**[1] **Amit Paradkar**[3]
[1]IBM Research, India, [2]IBM Hungary, [3]IBM T. J. Watson Research Center, Hawthorne

Abstract

Technical support domain involves solving problems from user queries through various channels: voice, web and chat, and is both time-consuming and labour intensive. The textual queries in web or chat mode are unstructured and often incomplete. This affects information retrieval and increases the difficulty level for agents to solve it. Such cases require multiple rounds of interaction between user and agent/chatbot in order to better understand the user query. This paper presents a deployed system called Question Quality Improvement (QQI), that aims to improve the quality of user utterance by understanding and extracting important parts of an utterance and gamifying the user interface, prompting them to enter the remaining relevant information. QQI is guided by an ontology designed for the technical support domain and uses co-reference resolution and deep parsing to understand the sentences. Using the syntactics and semantics in the deep parse tree structure various attributes in the ontology are extracted. The system has been in production for over two years supporting around 800 products resulting in a reduction in the *time-to-resolve* cases by around 29%, leading to huge cost savings. QQI being a core natural language understanding and metadata extraction technology, directly affects more than 8K tickets everyday. These cases are submitted after 50K edits done on the case based on QQI feedback. QQI outputs are used by other technologies such as search and retrieval, case routing for automated dispatch, case-difficulty-prediction, and by the chatbots supported in each product page.

1 Introduction

Problem solving in technical support domain requires understanding of user queries, so that agents can resolve the case via debugging at their end. For many debugging scenarios, agents rely on search tools (Gupta et al. 2018) for finding appropriate answers from internal and external knowledge sources. In the quest to improve the search results, the quality of the input question plays a very important role. Studies show that poor quality of user questions affects the downstream search accuracy. One way to guarantee the question quality is to mandate filling in pre-defined forms. However, this impacts the user experience of a free-text box where the expectation is to describe in free natural language the problem at hand. A more intelligent way would be to automatically discover what the user has already typed in and nudge them to enter the next most relevant information to add. The challenge here is to understand the relevant pieces of information supplied and those missing from the numerous human edits (text addition/deletion) made on a free text box. Instead, we focus on prompting the user to enter relevant next info, thereby creating a unique user experience.

In this paper we present a system that understands the user query and reflects the understanding as extractions of important attributes on a gamified User Interface (UI). This creates inquisitiveness in users that an intelligent system is trying to understand their query, in turn, motivating them to help the system. Thus, they add or edit their query to make it complete or more informative. The system relies on grammatical structure based extractions on deep-parse based slot grammars and predicate argument structures (McCord 1990). The extracted attributes appear as check-marks on the UI with a progress bar indicating the quality or strength of the question as shown in Figure 5. This linguistic based gamification happens in real-time to provide users instant feedback and improves their experience without the burden of mandatory form-filling.

1.1 Challenges

Understanding user queries have various challenges such as finding entities and important domain specific attributes to be able to solve the issue. The main attributes that we are trying to extract from query or prompt the user to enter in the query are: Symptom, Activity, Action, and Advise. These attributes could often be overlapping, for example: symptom and activity co-occur several times. The other challenges are presence of long blocks of machine generated text (error messages/logs pasted, or stack traces etc) which are very difficult to parse given the structure. Bad parse leads to no output annotation for these machine generated text snippets. For understanding an unstructured text query and extracting structured attributes, it is important to be able to resolve co-references, extract relationships between entities and attributes, understand text in colloquial or chat language with

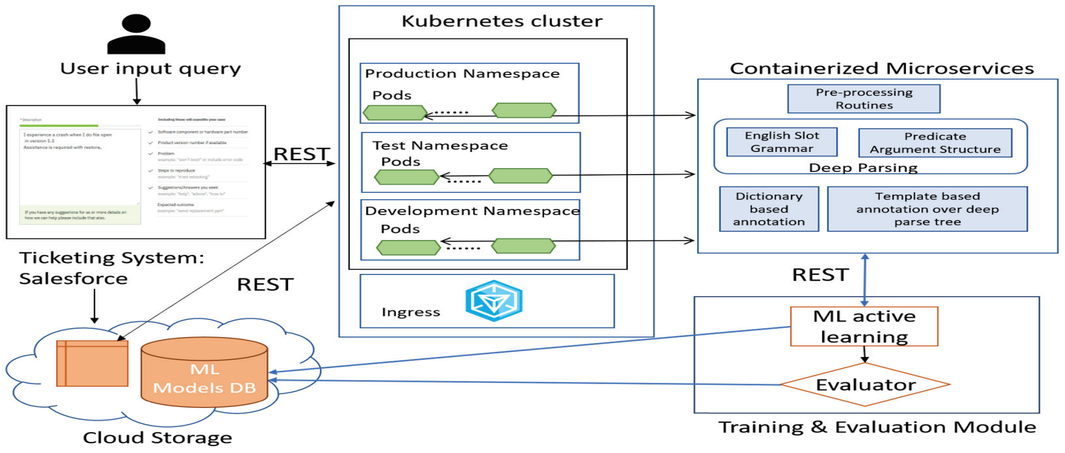


Figure 1: Architecture of proposed system

use of aliases or short-forms etc.

1.2 Main contributions

This paper presents a deployed system that uses intelligent linguistics within a UI to improve the question quality in technical support domain. The main contributions of the system are:

1. Deep Parsing based annotators: The annotators are the core component of the QQI system. QQI is guided by an ontology defined for technical support domain. The attributes in the ontology namely, *Symptom, Intent-Advise, Intent-Action, and Activity* are extracted from the deep parse tree structure based on a set of templates (grammatical structures with relationship) and dictionaries, (details presented in section 4.1). For each query the deep parsing trees are first generated and the grammatical structure based templates are used over the parse tree output to extract the attributes.
2. Machine Learning based continuous evaluation framework: To ensures the quality of the QQI system, we have an automated evaluation framework, which relies on a machine learning model and an evaluator. This system is a Conditional Random Forest based system, trained using manually annotated data and retrained using active learning from corrected QQI outputs.
3. Linguistics-based UI Gamification: The gamification on the user interface impacts the user psychology by comforting them that an intelligent technology is trying to understand their question. This interests and prompts them to improve their question quality without pressurizing them to enter any mandatory information. The UI shows the outputs of the extractions in realtime, along with a progress bar which is indicative of the importance of the annotations being extracted, indicating the quality of the question to the user.

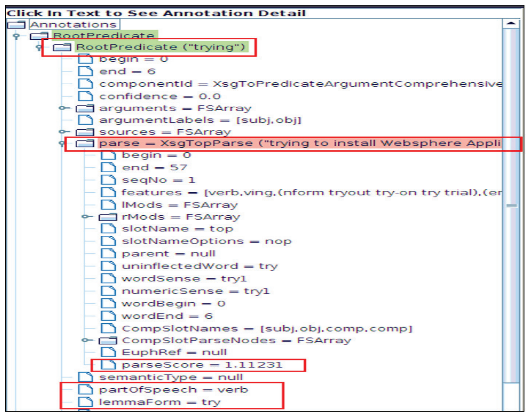
The results are presented with real customer data from 7000 different products being supported by QQI which reduces the *time-to-resolve* the cases by 29%. This leads to an overall cost savings of 25% annually.

In section 2 we present related work in this domain. In section 3, we briefly give an overview of the entire deployed system. In section 4 each component of QQI is explained in detail including the deployment, performance, scalability and maintenance. Section 5 explains the evaluation metrics for QQI, both from research and business perspectives. In section 6 we briefly mention the business impact that QQI had and continuous to bring in, followed by discussion and future work in section 7.

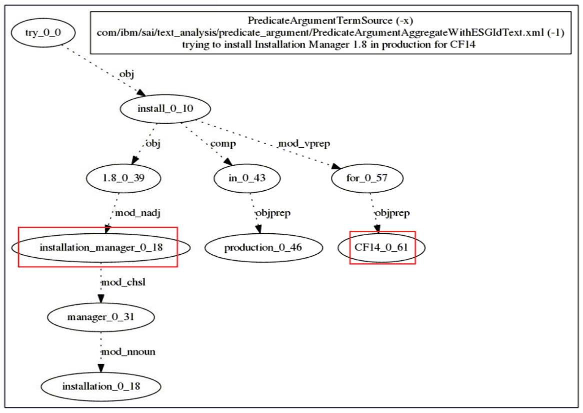
2 Related Work

There has been significant interest in natural language understanding from the research community leading to several technologies such as dependency parsers (Klein and Manning 2002), constituency parsing (Stern, Andreas, and Klein 2017), deep parsing (McCord, Murdock, and Boguraev 2012), Named entity recognition (Lafferty, McCallum, and Pereira 2001), Semantic Role Labeling (SRL) (Punyakanok, Roth, and Yih 2008) and different versions of them. The authors in (de Marneffe and Manning 2008) parse grammatical relations from text for automated understanding. Deep Parsing (McCord, Murdock, and Boguraev 2012) uses English Slot grammar (McCord 1990) that decomposes the natural language text to slots with sentence surface structure and deep grammatically structures to create a deep parse dependency tree. For Jeopardy, Watson used slot grammar parsing (McCord 1990) that decomposes the natural language text to slots, which is widely used for applications such as question decomposition, knowledge extraction for QA (Fan et al. 2012), relation extraction, (Wang et al. 2012), candidate generation (Lally et al. 2012), analysis and gathering of textual evidence (Murdock et al. 2012).

Deep Parsing is different from SRL (He et al. 2017) since the latter focuses on detection of words associated with the predicates and then classify the phrases into specific roles. SRL depends on a shallow parse, whereas deep parsing involves slot grammar which parses both structurally (connec-



```
Click In Text to See Annotation Detail
Annotations
  RootPredicate
    RootPredicate ("trying")
      begin = 0
      end = 6
      componentId = XsgToPredicateArgumentComprehensive
      confidence = 0.0
      arguments = FSArray
      argumentLabels = [subj,obj]
      sources = FSArray
      parse = XsgTopParse ("trying to install Websphere Appli
        begin = 0
        end = 57
        seqNo = 1
        features = [verb,ving,(nform tryout try-on try trial),(er
        lMods = FSArray
        rMods = FSArray
        slotName = top
        slotNameOptions = nop
        parent = null
        uninflectedWord = try
        wordSense = try1
        numericSense = try1
        wordBegin = 0
        wordEnd = 6
        CompSlotNames = [subj,obj,comp,comp]
        CompSlotParseNodes = FSArray
        EuphRef = null
        parseScore = 1.11231
      semanticType = null
      partOfSpeech = verb
      lemmaForm = try
```



PredicateArgumentTermSource (-x)
com/ibm/sai/text_analysis/predicate_argument/PredicateArgumentAggregateWithESGIdText.xml (-1)
trying to install Installation Manager 1.8 in production for CF14

Figure 2: Deep Parse Tree Structures: English Slot Grammar (left), Predicate Argument Structure (right)

tions between each node and slot: surface structure) and semantically (complement slots and adjunct slots). Complement slots play the role of grammatical and logical argument of word senses. Adjunct slots are associated with the Parts-of-speech (POS) of the headword sense. Deep parse tree has features such as POS, complement slot frame, morphosyntactic feature (semantic and syntactic), word-sense name, numerical score (likelihood of being correct parse tree structure), and generalized support verb. Thus it is much more elaborate than shallow parsing dependency trees and simple POS taggers.

In (Yang et al. 2017), authors present a system which claims to improve the state of the art for technical support question answering. They address the problem of knowledge representation by constructing a knowledge graph from well structured documents and other technical content. This work does not delve into the complexities of unstructured user questions in technical support such as incomplete or short sentences, combination of human and machine generated text (pasting error logs or stack traces while writing), or bad grammatical sentences with improper punctuations etc.

3 System Overview

Figure 1 shows the complete architecture of the proposed QQI system. The linguistic processing is based on the user query and is reflected on the UI while the user types in. For every one second pause or in presence of a sentence delimiter, QQI's Deep Parsing based Annotation APIs are fired to show annotations as results on the UI. This UI is created on the ticketing system front-end. At the backend caching and multi-threading helps make the QQI system real-time. Once the query is submitted, both QQI and learning based evaluator system get triggered. The query is appended with all the QQI outputs for the next set of applications such as chatbot, automation services, and search and retrieval. The evaluator service module continuously monitors the performance of QQI, by spraying the query to all the learning based annotators and comparing word-overlap between the output extractions by QQI versus learning modules.

4 Components of QQI System

This section describes in detail the different functional components of the system viz.: the deep parsing based annotator building, machine learning based continuous evaluation framework, and the UI functional component.

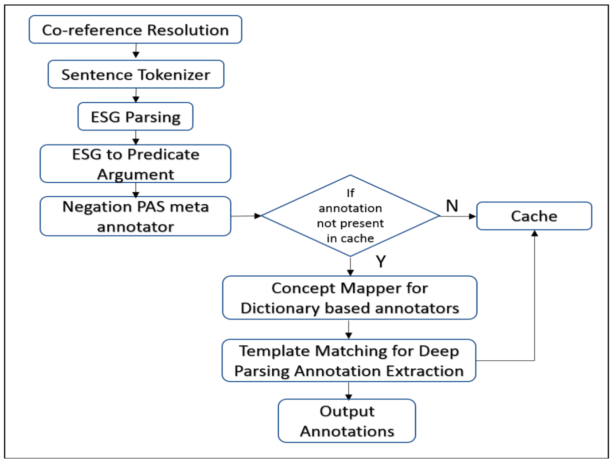
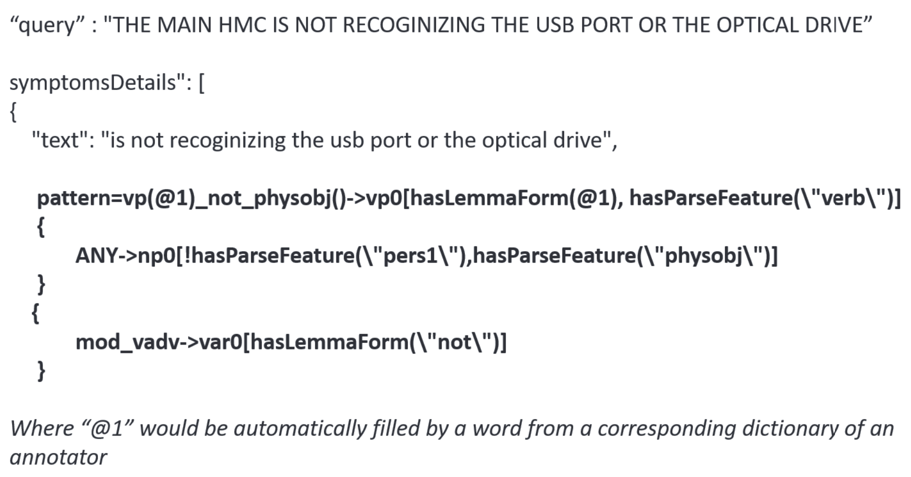

Co-reference Resolution
Sentence Tokenizer
ESG Parsing
ESG to Predicate Argument
Negation PAS meta annotator
If annotation not present in cache
N
Cache
Y
Concept Mapper for Dictionary based annotators
Template Matching for Deep Parsing Annotation Extraction
Output Annotations

Figure 3: Deep Parse Annotation Flow

4.1 Deep Parsing based QQI Annotators

In this section, we will first discuss about how the QQI annotators are build. The flow of deep parse annotator building is shown in figure 3.

Sentence Preprocessing Each input query is preprocessed via a co-reference resolution module and a sentence tokenizer module. Co-reference resolution is very important since the user might refer to the objects once and use several co-references to refer to it in subsequent lines. Co-reference resolution is done by the pairwise ranking of each mention. Sentences are tokenized, and prior to parsing the sentences, every reference to an entity is resolved, for the parse tree to be able to link structures to the correct entity.



```
“query” : "THE MAIN HMC IS NOT RECOGINIZING THE USB PORT OR THE OPTICAL DRIVE”

symptomsDetails": [
{
   "text": "is not recoginizing the usb port or the optical drive",

    pattern=vp(@1)_not_physobj()->vp0[hasLemmaForm(@1), hasParseFeature(\"verb\")]
    {
         ANY->np0[!hasParseFeature(\"pers1\"),hasParseFeature(\"physobj\")]
    }
   {
         mod_vadv->var0[hasLemmaForm(\"not\")]
    }
```

Where “@1” would be automatically filled by a word from a corresponding dictionary of an annotator

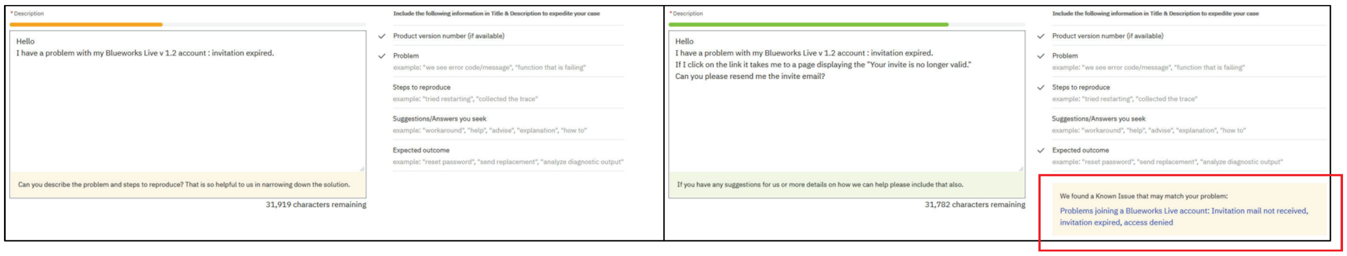
Figure 4: Sample template in semantic parsing model

Deep Parsing The main element in the QQI pipeline is the *Annotators* or the attribute extraction modules. These annotators have to interpret the text, and understand the structure to be able to identify and extract snippets from the text belonging to particular attributes. We want to extract the following attributes viz.: Product Name, Component, Version, PartNumber, ErrorCode, Symptom, Intent-Advise, Intent-Action, Activity. While the product name and component annotator are dictionary based, the version, part number and error codes annotator are regular expression based. The other annotators require deep understanding of text and thereby require deep parsing. These attributes are configurable and can vary depending on the ticket domain.

English Slot Grammar (ESG) There are several natural language parsers that allow tokenization and POS tagging (Klein and Manning 2002) but they provide a simple description of the grammatical relations in a sentence. This is not sufficient for several applications. Slot Grammar system has a lexical character and treats different grammatical structures in a language independent manner through different rule types. QQI uses two deep parsing components: English Slot Grammar (ESG) followed by Predicate Argument Structure (PAS) for linguistic analysis of text (McCord 1990). English Slot Grammar (ESG) works on input sentences to produce a deep parse tree with logical analysis and grammatical analysis. The parse trees for every segment is ranked based on the likelihood of the parse being correct and the highest ranked parse is used.

Predicate Argument Structure (PAS) Predicate Argument Structure (PAS) simplifies the ESG parse tree by combining the two dimensions of ESG by omitting nodes with auxiliary verbs, closed class nodes introducing verb phrases, determiners (except for high semantic determiners), forms of *be* with no predicate and with adjective predicate. This makes PAS generic and reflective of the core semantic meaning. PAS forms a labeled directed graph which is more flexible and requires less knowledge than ESG. To build a negation annotator which can mark a sense of negation within the parse structure, we modify the PAS by not dropping all the nodes. The negation annotator helps identify symptoms/problems like *does not work, cannot boot* from a question. Figure 2 shows an ESG parse tree and the corresponding PAS tree structure for a sample sentence. Unlike ESG, exact semantic meaning of a sentence is irrelevant to PAS as long as the core meaning is the same. Thus for semantically similar sentences, PAS structure remains same whereas ESG would be different. For example, active and passive sentences have different ESG parse tree but same PAS tree. Therefore, pattern matching on PAS tree is more efficient than on ESG parse tree.

Annotation Extraction After constructing the ESG and PAS tree, we add sub-tree patterns which match the parse tree structure, to extract the attribute annotations. Subtree patterns are written on the basis of sentence grammatical structure and certain keywords belonging to each attribute. For example: for *symptom* attribute, typical dictionary words would be ”fail”, ”problem”, ”issue” etc. For each attribute, there are different set of words and these words are placed in different dictionaries (noun, verb or usage). Manually creating such dictionaries and rules over them is both time-consuming and requires a lot of domain knowledge. There could be as many as 500 dictionary words for each attribute under different word-forms or grammar usage. Thus we write a set of templates that encode a grammatical structure and can automatically create rules using all words in a particular dictionary (specified as a wildcard). This makes the rule writing significantly scalable as we need to write 50 templates per attribute (10-15 templates per usage type within each attribute). The dictionaries are mined in semi-automated fashion by matching the ”template-only” on a large corpus of text and finding word suggestions which may be added to dictionary. Such word suggestions are then manually filtered and added to the dictionaries. Pattern based matching has been used in the Jeopardy Watson system (McCord, Murdock, and Boguraev 2012). Figure 4 shows a user


* Description

Hello
I have a problem with my Blueworks Live v 1.2 account : invitation expired.

Can you describe the problem and steps to reproduce? That is so helpful to us in narrowing down the solution.

31,919 characters remaining

Include the following information in Title & Description to expedite your case

✓ Product version number (if available)

✓ Problem
example: "we see error code/message", "function that is failing"

Steps to reproduce
example: "tried restarting", "collected the trace"

Suggestions/Answers you seek
example: "workaround", "help", "advise", "explanation", "how to"

Expected outcome
example: "reset password", "send replacement", "analyze diagnostic output"

* Description

Hello
I have a problem with my Blueworks Live v 1.2 account : invitation expired.
If I click on the link it takes me to a page displaying the "Your invite is no longer valid."
Can you please resend me the invite email?

If you have any suggestions for us or more details on how we can help please include that also.

31,782 characters remaining

Include the following information in Title & Description to expedite your case

✓ Product version number (if available)

✓ Problem
example: "we see error code/message", "function that is failing"

✓ Steps to reproduce
example: "tried restarting", "collected the trace"

Suggestions/Answers you seek
example: "workaround", "help", "advise", "explanation", "how to"

✓ Expected outcome
example: "reset password", "send replacement", "analyze diagnostic output"

We found a Known Issue that may match your problem:
Problems joining a Blueworks Live account: Invitation mail not received, invitation expired, access denied

Figure 5: Linguistics on User Interface

query with the extracted symptom and the pattern that led to the symptom extracted.

4.2 ML based Continuous Evaluation pipeline

Even after deployment of QQI in August 2017, for first six months, Subject Matter Experts (SME) from each product randomly verified QQI extractions on 100 randomly selected cases from logs and provided feedback on the quality of QQI. Although manual verification is best for constant improvement, with these SME cycles we had two major learnings. Firstly, QQI was quite stable and generic, thus this SME cycle was just a metric for QQI, and not very important for its improvements. Secondly, SME verification takes up a lot of time and manually a person cannot examine and provide metrics for more than a 100-500 cases a week. Thus there is a need to build an automated framework for evaluation of QQI to ensure its quality. We created an automated evaluation pipeline that will definitely be much less accurate than human annotation, but atleast provides a lower bound and raises a flag if QQI is near the lower bound.

The evaluation pipeline relies on a Conditional Random Fields (CRF) model, followed by a sequence comparator that evaluates the precision, recall and f-score QQI considering CRF as the groundtruth generator.

Conditional Random Fields (CRF) CRFs are graphical models that can capture such dependencies among input observations (Lafferty, McCallum, and Pereira 2001). A CRF model defines a conditional distribution $p(y|x)$ where *y* is the corresponding label sequence and *x* is the input sequence. The input *x* can be dependent on the current hidden label *y*, previous *n* hidden labels and on any of the other inputs in an *n* order CRF. In this problem, the input sequence *x* is the original user query which could be a set of sentences, and the label sequence *y* corresponds to the symptom, activity, intent or other assigned to the input sequence. The probabilistic model of a label sequence given some sequence of words is mediated in this model through a set of weighted functions f_i:

$$p(y|x) = \frac{exp(\sum_i \sum_t w_i f_i(y_{t-1}, y_t, x, t))}{Z(x)}$$

where the w_i are the weights assigned by the learning algorithm, and $Z(x)$ is a normalization factor over all label sequences.

For implementation, we created a ensemble model by creating two different CRFs from two different codebases. First we used the linear-chain Stanford CRF implementation (Finkel, Grenager, and Manning 2005) . The current word, previous word, next word, are used a context with a current word character n-gram ($n \leq 6$). We set presence of word in left window (size = 4), presence of word in right window (size = 4), position of word in the sentence, and current POS tag as features computed within the Stanford NER toolkit. These features were used to train and test CRF model which we call as CRF_{sn}. The second toolkit that we use is IBM Statistical Information and Relation Extraction (SIRE), (Florian et al. 2004) which has the capabilities to use dictionaries along with the CRF training examples for sequence extraction. All other features remained same for both CRF model trainings.

The CRF models are initially trained with 3000 manually annotated tickets. CRF_{sn} is retrained using active learning with different sampling strategies to pick samples from QQI corrected data (from SMEs). For SIRE, we bootstrapped SIRE training with a lot of QQI outputs with the aim of achieving similar performance of that of QQI, without any rule-writing or domain understanding. For both CRF models, B-I-O encoding is used, where each word in the query is annotated to one of the following cases (labels): the beginning of a symptom (B-S), inside of a symptom (I-S), the beginning of an activity (B-A), inside of an activity (I-A), the beginning of an intent (B-I), inside of an intent (I-I) or other (O). Apart from the 4 deep parse annotators we also extract regions (machine generated text) within the query.

Evaluator Module The outputs of CRF and QQI are compared to check word overlap and we report partial overlap (thresholded over 0.6) and exact overlap to match QQI annotations versus CRF annotation. These word overlap is used to compute the precision, recall and f-score metrics for QQI. We also compute the coverage of each of the CRF annotator as well as QQI as the evaluator matches the outputs of CRF and QQI only for cases wherein the CRF model has been able to make any extraction. Since the rule-based QQI has much larger recall, thus it is important to know how much evaluator has been able to compare while checking the automated f-score metrics.

4.3 User Interface

Figure 5 shows snapshots of the User Interface (UI), where on the left is the description box, where the user can key in the query. The UI triggers the annotator API calls for every sentence delimiters such as full-stop, comma etc or pause while typing. The outputs of the annotators are reflected back on the UI real-time, tick-marking the fields that got extractions as seen on the right of figure 5. Users can also see questions appearing at the bottom of the description box, relevant to the missing checkmarks. QQI outputs are used to search for the similar resolved tickets and get the corresponding resolution as highlighted in red over the UI screenshot on the right of Figure 5. The progress bar on top indicates the strength of the query, and its colour changes from red (weak), to yellow (medium) to green (strong) as more attributes are extracted. This motivates the user to strengthen the query and the user then adds details to see if other fields get checkmarked. For the progress bar, the product name, component. symptom and action annotators have maximum weights followed by the other annotators. The weights of these annotators were determined after several experiments with these annotations as metadata for search and retrieval (Gupta et al. 2018).

4.4 Deployment and Maintenance

QQI is currently deployed as a single tenant service hosted in a Kubernetes cluster with multiple REST APIs. The deployment architecture with blocks showing each microservice are shown in Figure 1. The system has been running on Salesforce successfully since last two years with several products being onboarded at different times. This has led to sufficient data getting annotated and is helping the machine learning models to become generic now. QQI scales well on new products thus minimal improvements in terms of template creation is required. The Kubernetes container has Development, Test and production namespaces each with dedicated worker nodes and pods as shown in Figure1.

4.5 Performance and Scalability

For performance we use a tool called concept-mapper for the dictionary based annotators and maintain a cache as the user query has lot of overlap between successive edits. Concept-Mapper is a highly configurable, high performance dictionary lookup tool, implemented as an Unstructured Information Management Architecture (UIMA) component (Tanenblatt, Coden, and Sominsky 2010). Each dictionary annotator has separate dictionary and Concept-Mapper tool is used to map entries from dictionary to query to produce annotations. Concept-Mapper performs fast in-memory lookup even with multi-million entry dictionaries.

A small cache memory is used where the outputs for the past 10k queries are saved. For each incoming request, a cache lookup is attempted to check for annotations and QQI pipeline is invoked for the remaining sentences and previous output is picked up from the cache itself. Typically a submit type ticket has on average 16 edit type requests, meaning user had done 16 edits based on QQI's feedback. Thus, maintaining a cache is very useful in reducing the overall response time.

Performance and response time is essential in any solution built for live feedback. The UI is generating an event with every keystroke of the user, however this is throttled to only send out a single request in every 3 seconds, this way we ensure the service is not getting overwhelmed with requests, and the user still gets relevant & fresh feedback about what they are typing. The goal was to be able to serve 10 to 20 requests per sec with varying content and maintain a response time less than 3 seconds. Figure 6 shows the response time of QQI is usually around 1.8 miliseconds. The assessment of the performance took place with so called ”stress tests” that are dedicated to emulate extraordinary user traffic to measure response times. This stress testing is performed using JMeter performance test conducted with a frequency of 20 requests/sec based on 500 random actual problem descriptions.

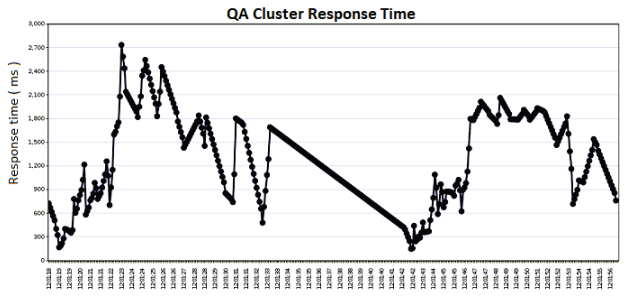


Figure 6: Stress Testing QQI: Time (x-axis) vs Response Time of QQI(Y-axis)

5 Evaluation Metrics and Results

In this section we describe the various metrics on which QQI is measured. From a research perspective we measure QQI's performance on the basis of the annotators: quantitatively and qualitatively. QQI annotators are evaluated quantitatively by using *Coverage* as the metric and qualitatively by *Precision, Recall, F-score*. From a business perspective, the biggest metric that QQI aims to minimise is the *Time-To-Resolve* a case. In Table 1, we present the statistics of tickets handled by QQI, and the impact of sentence preprocessing and negation annotation on QQI.

Table 1: Ticket Statistics

Total number of supported Products	793
Number of edit type tickets (per day)	50000
Number of submitted tickets (per day)	8000
Average number of words in submitted case	836
Average number of sentences in submitted case	13
Average time taken for processing (submit)	2.528 secs
Impact of negation-annotator on symptom	52.8
Impact of sentence pre-processing on symptom	25%
Impact of sentence pre-processing on Intent	31%
Average no. of edits per submit	16

The QQI annotations were tested with the help of Subject Matter Experts (SME) of each product for 22 products (results shown in table 2). For each product a random subset of 100 queries with their annotations are selected from the logs and SMEs manually checks whether the query has a particular attribute present and if it was correctly captured by QQI. *Coverage* is a measure of QQI's quantitative performance and indicates the percentage of query with extractions of an

Table 2: Quantitative and Qualitative Evaluation of QQI Deep-parse based Annotations on 22 products by SMEs

Annotator	Coverage	Precision	Recall	F-score
Symptom	70.48	78.66	89.39	83.68
Activity	57.96	56.45	83.33	67.30
Intent-Advise	53.79	88.63	95.54	91.95
Intent-Action	49.65	81.01	78.92	79.96

attribute. For example: 70% coverage of symptom means, 70% queries had one/more symptoms extracted out of the 100 queries in test. *Precision*, *Recall* and *F-score* are computed to evaluate the qualitative performance of QQI. Precision indicates the number of correct annotations out of the total number of extractions for each attribute. Recall indicates the number of correct extractions by QQI divided by the total number of expected extractions that were in the query (as marked by SME). F-score is the harmonic mean of Precision and Recall and is being used a standard measure of QQI's qualitative analysis.

The average F-score for all 4 deep parse annotation is usually more that 80%, across different hardware and software products with different type/format of user queries, thus assuring high recall generic annotators.

Since the CRF based evaluator is only able to compute f-score for queries with both CRF and QQI annotations, we do measure the coverage of CRF and consider the difference between coverages of CRF and QQI as an indicator of queries that did not get evaluated. Since $CRFF_{sn}$ is trained on only 3000 manually annotated tickets, the coverage and recall is quite low although the precision is high. SIRE performs better than $CRF-sn$ since it is bootstrapped with 30K tickets QQI outputs and is able to consume the QQI dictionaries used for rule writing. The Average f-score of SIRE and CRF_{sn} is 53.03% and 28.03% respectively. But the performance of the two CRF models being complimentary, an ensemble is created for the evaluation which has an average f-score of 61.8%.

Coverage guides the qualitative testing and for the products with coverage below 50% qualitative analysis is performed to improve the annotators. Coverage is computed for the non-deep parse annotators as we need to know if any additions are required in the dictionary or regular expressions. The coverage for Product Name, component, version, Part-Number and Errorcode are 31.79%, 88.44%, 43%, 0.10%, 10.78% respectively.

We use "Time-To-Resolve" (TTR) a case as a metric to measure the impact of QQI. The baseline against which the impact is measured are the cases which get no QQI annotations. Based on the QQI logs, we compare the reduction in TTR with the annotator output to measure contribution of each annotator on TTR as shown in Figure 7. We analyzed cases for a year and observed that when all the deep parse annotators are present in the case description, then reduction in TTR is 23.4%. The *Intent* annotator which is a combination of Intent-Action and Intent-Advise has the highest impact on TTR which is 33.6%. This can be explained as the agent has an exact understanding of what the user expects the agent to do in terms of advise or action required. Thus, Intent turns

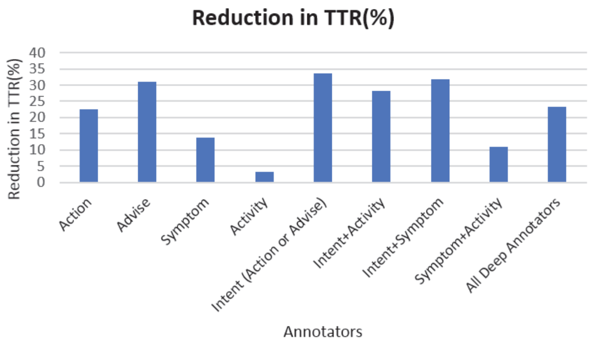


Figure 7: Contribution of Deep Parse Annotators in the Reduction of "Time to resolve" (TTR)

out to be the most important annotator for reduction in TTR. The effect of activity annotator in reducing the TTR is least (3.2%) in comparison with others since activity is indicative of steps user has already taken but not the actual problem or ask.

Table 3 shows an illustrative example of how QQI inspires user to edit the query with annotations. In the first case only the symptom got extracted and rest were not check-marked in UI. In the second API call, the query was edited so that more symptoms and activity could be extracted which checked the symptom and activity in the UI. Finally when the user is comfortable with the output of the extracted annotations they submit the case description as a ticket.

6 Business Impact

QQI impacts around 800 products that IBM supports on the Salesforce platform. QQI API is called 60K times everyday to support 8K tickets that are finally submitted after more than 50K edits done on them based on QQI feedback. As explained in metrics, several comparisons are done to compute the impact of time-to-resolve cases on tickets with and without QQI outputs, as well as influence of each annotator on the TTR reduction. The results show that the TTR decreased by 29% for tickets with QQI as compared to tickets without QQI, which leads to overall cost savings of 25% annually. QQI has also been packaged separately as a product offering to a different client thus leading to revenue generation.

7 Discussion and Future Work

The QQI service touches every ticket be it on webpage or a chat interface, and the average number of edits before submitting a case clearly indicates that QQI enables user to provide a better question. The annotations of QQI are additionally used by chatbots to determine Product name,symptoms and intents before starting a chat session and by search services as metadata to improve retrieval results. Also, the reduction of TTR by 29% shows that a ticket improved upon by QQI helps agent resolve the case much faster and leads of huge cost savings. We are also building other annotators such as server name, IP address, diagnostic test, test results

Table 3: Deep parse annotations evolves with case description

Query	Query Type	Symptom	Activity	Intent
ACL's are not working for few users/group. Hello team,	edit	[ACL's are not working for few users/group []	[]	[]
ACL's are not working for few users/group. Hello team, We are getting inode error on couple hive databases when users belonging to group $app_1 1469$ tries to access the databases on $PROD_{BATCH}$ clusters. We have applied rwx permissions(ACL's) for group on HDFS path.	edit	[ACL's are not working for few users/group] [are getting inode error on couple hive databases]	[We have applied rwx permissions for group on HDFS path] [users belonging to group $app_1 1469$ tries to access the databases on $PROD_BATCH$ clusters]	[]
ACL's are not working for few users/group. Hello team, We are getting inode error on couple hive databases when users belonging to group $app_1 1469$ tries to access the databases on $PROD_{BATCH}$ clusters. We have applied rwx permissions(ACL's) for group on HDFS path.Even we tried providing 777 permission on hdfs path. Could you please have a look.	submit	[ACL's are not working for few users/group] [are getting inode error on couple hive databases] [777]	[We have applied rwx permissions for group on HDFS path] [users belonging to group $app_1 1469$ tries to access the databases on $PROD_{BATCH}$ clusters] [Even we tried providing 777 permission on hdfs path.]	[Could you please have a look]

and resolution. These new annotators are required to populate Knowledge graphs and use the Knowledge graphs for effective ticket resolution and improve retrieval results.

Acknowledgements

The authors acknowledge the time and efforts from everyone involved in building, discussing and giving feedback for this product, which enabled it to shape up and mature.

References

de Marneffe, M.-C., and Manning, C. D. 2008. The stanford typed dependencies representation. In *COLING: Workshop on Cross-Framework and Cross-Domain Parser Evaluation*.

Fan, J.; Kalyanpur, A.; Gondek, D. C.; and Ferrucci, D. A. 2012. Automatic knowledge extraction from documents. *IBM Journal of Research and Development* 56(3.4):5:1–5:10.

Finkel, J. R.; Grenager, T.; and Manning, C. D. 2005. Incorporating non-local information into information extraction systems by gibbs sampling. In *Association for Computational Linguistics*.

Florian, R.; Hassan, H.; Ittycheriah, A.; Jing, H.; Kambhatla, N.; Luo, X.; Nicolov, N.; and Roukos, S. 2004. A statistical model for multilingual entity detection and tracking. In *In NAACL/HLT*.

Gupta, A.; Ray, A.; Dasgupta, G.; Singh, G.; Aggarwal, P.; and Mohapatra, P. 2018. Semantic parsing for technical support questions. In *Proceedings of the 27th International Conference on Computational Linguistics, COLING*.

He, L.; Lee, K.; Lewis, M.; and Zettlemoyer, L. S. 2017. Deep semantic role labeling: What works and what's next. In *ACL*.

Klein, D., and Manning, C. D. 2002. Fast exact inference with a factored model for natural language parsing. In *NIPS*, 3–10. MIT Press.

Lafferty, J.; McCallum, A.; and Pereira, F. 2001. Conditional random fields: probabilistic models for segmenting and labeling sequence data. In *ICML*.

Lally, A.; Prager, J. M.; McCord, M. C.; Boguraev, B.; Patwardhan, S.; Fan, J.; Fodor, P.; and Chu-Carroll, J. 2012. Question analysis: How watson reads a clue. *IBM Journal of Research and Development* 56(3):2.

McCord, M. C.; Murdock, J. W.; and Boguraev, B. 2012. Deep parsing in watson. *IBM Journal of Research and Development* 56(3):3.

McCord, M. C. 1990. Slot grammar: A system for simpler construction of practical natural language grammars. In *Proceedings of the International Symposium on Natural Language and Logic*, 118–145.

Murdock, J. W.; Fan, J.; Lally, A.; Shima, H.; and Boguraev, B. 2012. Textual evidence gathering and analysis. *IBM Journal of Research and Development* 56.

Punyakanok, V.; Roth, D.; and Yih, W.-t. 2008. The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics* 34(2):257–287.

Stern, M.; Andreas, J.; and Klein, D. 2017. A minimal span-based neural constituency parser. In *Association for Computational Linguistics (Volume 1: Long Papers)*.

Tanenblatt, M. A.; Coden, A.; and Sominsky, I. L. 2010. The conceptmapper approach to named entity recognition. In *LREC*.

Wang, C.; Kalyanpur, A.; Fan, J.; Boguraev, B.; and Gondek, D. 2012. Relation extraction and scoring in deepqa. *IBM Journal of Research and Development* 56(3):9.

Yang, S.; Zou, L.; Wang, Z.; Yan, J.; and Wen, J.-R. 2017. Efficiently answering technical questions – a knowledge graph approach. In *AAAI Conference on Artificial Intelligence*.