Populating a Knowledge Base for Product-Oriented Discussions

Pierre Colombo

EPFL - CentraleSupelec
colombo.pierre@gmail.com

Fearnley Martin

EPFL - CentraleSupelec
fearnley.martin@epfl.ch

Supervised by Claudiu Musat (Swisscom) & Robert West (EPFL - DLAB)

May 29, 2017

Abstract

In this report, we tackle the problem of question-answering systems in an end-to-end manner. First, we propose a framework for populating the knowledge bases that these systems use. To do this, we employ a systematic approach that allows us to extract content from a large variety of question-answer websites. Second, we study ways of storing and querying question-answer pairs in this knowledge base. We compare methods based on inverted index databases, graph databases, and neural networks.

Introduction

As chatbots are starting to find their place in business applications, many companies are now working on the underlying technology. The aim of these bots is to be able to have product-related discussions in natural language. A common approach is to use retrieval-based chatbots. This means that the bot tries to find the best answer to a query from a large knowledge base of question-answer pairs. It is in this context that we have been working on the following project, and we have sought to answer two main questions. Firstly, how do we build such knowledge bases? Secondly, how do we best store and retrieve data in such knowledge bases?

The first problem rests on the assumption that the chatbot's answers will only be as good as the quality of the knowledge base it gets them from. For the bot to be able to handle a large variety of questions and come up with helpful answers, it is essential to have a wide knowledge base in which to search. Our first task was thus building a tool to populate knowledge bases. The information sources we have targeted were forums and community question-answer sites. The idea was thus to build a systematic, generic crawler that will be able to crawl many different sites and extract the question-answer pairs for our knowledge base.

The second problem we set out to tackle is how we can best store this information in order to query it and get pertinent answers to users' questions. The guiding question was: can we use a graph database to better model the relationships between our data than the more traditional baseline model using Solr? To tackle this, we compare various models for retrieval-based question answering: a text-search model using Solr, a graph model using Neo4j and a

machine learning model using TensorFlow.

Part I
Systematic web crawling for building question-answer datasets

I. Introduction

i. Problem context

Our starting point was a web crawler built using Python's Scrapy library for extracting question-answer pairs from superuser.com. This was relatively easy to build as the content we wanted to extract and the links we wanted to follow always had the same html structure and css classes. We could thus hardcode these attributes into our crawler.

While this approach works well for individual sites, it is not scalable and cannot be generalised to other sites. Our goal, however, was to build a wide-ranging database using data from all over the web. The question is then how can we generalise such a solution so as to be able to crawl all potentially promising sites?

The approach followed for building the proposed solution can be summed up in a few steps. First, we identified and abstracted out all the hardcoded steps in our initial crawler for superuser.com. Then, we designed site-independent solutions for these functions via an empirical process. To do this, we used an evaluation set of around 10 different sites. We tried out various heuristics, iteratively improving them until the retained method was generic enough to work across all sites.

The problem can essentially by divided into two sub-problems. Firstly, given a web page, how do we identify if it contains interesting content, and how do we extract this content? We call this the parsing sub-problem. Secondly, given a website, how do we navigate through the website to find pertinent web pages to scrape in the first place? We call this the crawling sub-problem. In what follows, we will tackle these two sub-problems independently and evaluate different methods we can use to tackle them.

ii. What data are we working with?

The first steps involved understanding what we were trying to extract. Two questions underpinned this: what data was available to us to be scraped? And what data would be useful to us for the retrieval problem?

We came up with a list of fields that can be found in the GitHub repository, and which the most important are: question_body, question_date, question_author, answer_body, answer_date, answer_author. One of the ideas we will find later on, for example, is how we can use the fact that many answers might share the same author in order to help evaluate the best answer. Other metadata we might want to extract would be, for example, the number of upvotes a question has.

We also looked at how we can model question-answer pairs. Essentially, we can classify the sites we scrape into two categories. First, we have question-answer sites such as Stack Overflow. These sites are easy to model as we have only questions and answers on the page. Second, there are forums such as Mac Rumors or Reddit. These sites are typically much harder to model because the questions and answers we seek to extract are usually hidden amongst lengthy (and sometimes off-topic) user discussions. As we will see later, this problem has not yet been entirely solved.

II. The parsing sub-problem

i. Sub-problem description

Given any web page, how do we decide whether it contains valuable content and then how do we extract this content? At heart, we have a classification problem and a parsing

problem. As it turns out, we solve them in the same way, i.e. if we can correctly parse the page, then that means the page contains question-answers pairs and is thus what we qualify as a *results page*.

ii. Intuition behind the algorithm

Visually, it is easy for us to identify whether a page is a *results page* or not. We can see it straight away from the structure. These *results pages* have a question block at the top containing the body of the question. Then beneath this, we find blocks corresponding to the different answers. These question and answer blocks always contains the same elements: an author, a date and a body.

Thus, our strategy is to use the common html structure of these types of pages. In fact, the underlying structure we work with is a tree, due to the nature of html. In this tree, we can identify similar structures of elements. These elements correspond to the blocks we would like to extract. Essentially we search the html tree to find a html node (which we will call the *container*) which has as its children a list of html nodes (that we call *blocks)* that contain author, date and body elements.

iii. Implementing the algorithm

The actual implementation of the above idea can be boiled down to a few main steps:

1. Initialise and clean html
2. Identify container node and blocks
3. Extract the blocks from container
4. Filter the extracted blocks
5. Parse blocks to extract structured content
6. Filter blocks for Swisscom products

The design of the algorithm was somewhat a balancing act. We needed to identify elements that were common to all the question-answer pages, and distinguish them from elements that are characteristic to only individual sites. Adding an extra constraint may well help us parse a particular site, but may break the parsing of another. We thus aim to keep the algorithm as general as possible with respect to website characteristics.

Once the code has been cleaned (javascript removed etc.), we need to find the *container* html node which contains the posts (*blocks*) we want to extract. We recursively search through the html tree starting from the body node. We define a node to be valid if it contains an author element, a date element and a body element of minimum size. If a node is valid, we add it to the search queue. For each iteration, we take a node from the search queue and look at its children. If a node has at least two valid children blocks, we return it as the *container*. The intuition is that the returned *container* node contains at least a question and an answer. See algorithm 1.

Algorithm 1 Step 2: Identify *container* node

Input: html body node
Output: html *container* node

```
initialise empty queue
add html body node to queue
while queue not empty do:
    node ← queue.pop()
    valid_child_block_count = 0
    for child in node.children() do
        if child is valid_block then:
            valid_child_block_count + = 1
            queue.push(child)
        end if
    end for
    if valid_child_block_count >= 2 then:
        return node
    end if
end while
return None
```

In reality, the structure of the websites we would like to parse isn't quite so simple. Often we come across nested structures, where we might have two main blocks (question and answer) and then the answers nested under the answer block, for example. Or perhaps we might encounter nested comments under each answer. To handle this, we call the algorithm

again on the blocks we find (i.e. the valid children of the *container*). This way we find all the nested blocks of the main blocks, and then we flatten the structure into a simple list of blocks.

Once we have the list of blocks (i.e. the html node encompassing the posts we seek to extract), we notice that the html nodes corresponding to the blocks often contain a lot of boilerplate code surrounding the actual blocks we would like to extract. To deal with this, we recursively travel down the html tree hierarchy inside a block until we find the lowest level node which still qualifies as a valid block. We do this for each block and return the new list of child blocks. See algorithm 2.

A word on how we identify valid blocks. First, they must contain an author element. We search for nodes that have "author" or "user" in their css class. Then amongst these nodes, we search for one that has a link to a user profile. Second, they must contain a post date. We use a regex expression to detect and extract possible dates. Third, they must contain a text body. We impose some extra criteria on the body such as a min and a max length, and we exclude certain types of content such as long lists of links.

Algorithm 2 Step 3: Extract blocks from *container*

Input: list of initial child blocks
Output: list of final child blocks

```
initialise dict of form {child1:[], child2:[]...}
for child in initial_child_blocks do:
    while exists valid_grand_children do:
        for grandchild in child.children() do:
            if grand_child is valid then:
                dict[original_child] = grand_child
                child = grand_child
            end if
        end for
    end while
end for
```

Given our list of blocks (which should correspond to the different posts on the site), we now need to parse them and extract the content. The idea here is first identify the author node (with previous method), then search for the node containing the body text but not containing the author node. This helps us separate the body text from the other elements in the node. We also filter out blocks that match some empirical filtering criteria (for example, blocks that contain login forms).

If at the end of all this, we have managed to parse at least two valid blocks, we classify the page as a *results page*.

Accuracy	Recall	Precision
92%	88%	94%

Table 1: *Classification of results pages*

iv. Results

We would like to measure the performance of our algorithm for both the classification problem and the parsing problem. To do so, we prepared a data set of 180 different pages from 20 different domains. The urls were selected from the top Google search results for Swisscom products.

For the classification problem, half of the selected pages were *results pages* and the other half were not. For the parsing problem, we labeled each page with the number of question-answer pairs we expected to extract.

We generally manage to correctly classify pages as *results pages* or not, as shown in table 1. In general, the cases in which the algorithm fails are what we call *index pages*. These are pages which contain lists of links towards *results pages*, for example search results pages or forum pages. Often each link is accompanied by an author, a date and a preview of the question, therefore misleading the algorithm.

In general, the main problem encountered is that the algorithm classifies non-*result pages* as *results pages* (i.e. false positives). The opposite problem is less frequent, i.e. we don't miss that many true positives. The consequence of this is that we tend to extract more "garbage" question-answer pairs than we would like to.

We display the results for the parsing prob-

Correctly parsed pages	Correctly parsed posts
50%	85%

Table 2: *Parsing of results pages*

lem in table 2. We define a correctly parsed page as a page where we correctly extracted all of the posts on the page. The correctly parsed posts statistic shows the number of posts we recovered out of the total number of posts we expected to recover. The results show that even though we often miss some posts on a page, we manage to extract most of the posts overall. As for the quality of the parsing, we shall look at this later.

All in all the algorithm performs pretty well, and manages for a large variety of sites to extract most of the content we would like it to extract.

III. The crawler sub-problem

i. Sub-problem description

Given a website, how can we crawl it in a targeted way to navigate to the pages containing the information we would like to extract? This is important for both crawling efficiency and for making sure the crawler stays on on-topic pages.

In the following, we describe various methods we have used to achieve this goal. To measure the efficiency of the crawling, we look at how many *results pages* we encounter out of the total number of pages crawled. To measure the pertinence, we manually evaluate the extracted data over randomly selected samples.

ii. Baseline

The baseline strategy is to crawl the whole website. This is done by following all the links we encounter on the pages we scrape.

On the one hand, we are sure to eventually get to the content we are looking for. However, we are probably going to go through a lot of irrelevant pages. This is both slow, and runs the risk of extracting content which is completely off-topic.

Site	Ratio
community.dynamics.com	1.8%
forums.macrumors.com	2.2%
community.spiceworks.com	28.8%
forum.eset.com	7.2%

Table 3: *Efficiency of baseline crawling strategy*

In table 3, we look at the ratio of *results pages* scraped to the total number of pages crawled. This helps us evaluate how efficient the crawler is. For the following evaluation, we start from an *index page* in each case and let the crawler run over 500 pages.

We see that in general, we very quickly venture into parts of the site which don't contain any pages that are valuable to us, as the ratios tend to be small. The ideas presented in the following parts aim to make this process more efficient.

iii. Automated crawling strategy via *index pages*

This strategy aims to replicate the idea used in the hard-coded crawler for superuser.com. The premise is to start from an *index page* and follow only links towards *results pages* and pagination links. This way, we only visit relevant pages that contain question-answer pairs. In theory, this should be both efficient and targeted.

To do this, we designed an algorithm similar to the *results page* parsing algorithm. We try to detect structural elements that are characteristic of *index pages*, notably lists of links containing questions and pagination links. We only follow these extracted links during the crawling.

Unfortunately, if the crawler fails to extract the right links from the *index pages*, then it can never get to the pages we would like it to

Site	Ratio
community.dynamics.com	43.7%
forums.macrumors.com	77.4%
community.spiceworks.com	57.7%
forum.eset.com	74.9%

Table 4: *Efficiency of url-guiding crawling strategy*

scrape. In practice this is what happens, as it is difficult to correctly identify the right links to follow. Sometimes the crawler runs out of links to follow and terminates early. Essentially, we have restricted the crawler's link extraction criteria too much. The implementation of the described algorithm would need to be revised in order to actually be of use for the crawler. A possible improvement would be to loosen the restriction when selecting the links to follow in order to lessen the chances of missing the links we aim to extract.

iv. Crawling strategy via url-guiding strategy

Taking inspiration from the hard-coded crawler, we can specify which urls to whitelist and blacklist, as well as restrict the xpaths of the links to follow. N.B. that this is a manual operation, which must be carried out for each site we wish to crawl. However, we would like to evaluate the strategy to see how much it improves on the baseline strategy.

We run the same evaluation as the baseline strategy, only this time we manually specify some attributes. For example, for a given site, we might restrain the urls to follow to urls containing the following fragments */topic/*, */forum/*, and restrict the xpaths to the following classes *DataItem*, *Pagination*. As we will see, simply adding these parameters greatly improves the crawling. We display the results in table 4.

There is a very large improvement in the crawling efficiency, that is to say we only visit the pages we want to. Considering that the *results page* algorithm is not that fast, and that we often have to throttle the request rate to avoid over-working the receiving server, it is important to target the crawling.

IV. Overall performance

We would now like to evaluate the tool overall. To do this, we repeat the previous crawling tests with an upper limit of 50 000 pages. We use the url-guiding technique here. We look first at some general crawling statistics, then we plunge into the data recovered to evaluate the quality.

The crawling statistics displayed in table 5 show us a few points. Firstly, we manage to crawl a large number of pages, of which a high percentage are *results pages*. Secondly, on average we manage to extract a reasonable amount of question-answer pairs. Thirdly, the crawling is very slow. This is principally due to request rate throttling, and the fact that the *results page* parsing algorithm is slow.

Now to actually evaluate the quality of the data scraped, we adopt the following methodology. For each of the five sites, we randomly sample 8 question-answer pairs. We then evaluate each pair on the following criteria:

1. Pertinence: is the question-answer pair on topic? (yes/no)
2. Question quality: is the question an actual question or problem, and is it understandable when taken out of the context of the page (yes/no)
3. Answer quality: does the answer actually respond to the question? (yes/no)
4. Parsing quality: has the page been correctly parsed and have the right elements been extracted? (graded out of five)

We aggregate the grades over the eight questions which gives us table 6:

Firstly, in general all the question-answer pairs extracted are on topic. We can probably credit this to the url-guiding strategy which helped us only stay on pages reachable from the chosen index page (e.g. search results for iPhone). Also in general, the questions are pertinent and well formulated. The parsing is generally acceptable, and we manage to get the essential elements out of the page, even

Site	# pages	questions/*results page*	*results pages* /crawled pages	pages/sec
community.dynamics.com	12123	1.88	39%	0.14
reddit.com	3301	1.19	56%	0.75
forums.macrumors.com	19738	4.53	76%	0.22
community.spiceworks.com	29656	6.94	73%	0.33
forum.eset.com	4666	6.4	82%	0.35

Table 5: *Crawling statistics*

Site	Pertinence	Question quality	Answer quality	Parsing quality
community.dynamics.com	8/8	7/8	8/8	27/40
reddit.com	5/8	5/8	3/8	28/40
forums.macrumors.com	8/8	6/8	5/8	32/40
community.spiceworks.com	7/8	5/8	3/8	14/40
forum.eset.com	8/8	7/8	2/8	21/40
superuser.com(*)	8/8	8/8	8/8	40/40

Table 6: *Quality evaluation of extracted data*

though there are parsing defaults. Sometimes we don't manage to perfectly isolate the question body, sometimes the date is not right, and there are other minor problems that only occur on individual sites.

The main problem encountered concerns the answer quality. This stems from a problem we mentioned right at the beginning. Often in forums there is often a fair amount of discussion which is either off-topic or where users don't answer the question. Currently, to form question-answer pairs from a list of posts taken from a given page, we consider the first post to be the question of each pair, and we complete the pairs by taking each subsequent post as an answer to this question. This works well on question-answer sites such as Stack Overflow, but less well on forums. We currently have no implemented means of dealing with this, but we discuss an envisioned solution idea further on in the report.

Note that in general, there tend to be minor parsing problems specific to each site, but overall we manage to extract decent question-answer pairs for our data set.

V. Improvements

In this section we will present some ideas that could be implemented to improve the performance of the crawler based on the shortcomings we have seen thus far.

i. Automatic learning for url-guiding

The first shortcoming is that to get an efficient crawling solution, we still need to enter a manual input when using the url-guiding method. There are two possible ideas we might want to pursue to overcome this. The first would be to continue improving the *index page* crawling method by improving the heuristics. In practice though, this has so far proved tricky because we don't have that much to go on for extracting the correct links from the *index pages*. The second would be trying to emulate the manual input via a learning solution. In the following paragraph, we will outline the main idea.

We would like to try to learn what the user would need to have manually input in an automated fashion. Given that we can detect whether a page is a *results page* or not, we can track which links lead us to *results pages* and which don't. From this we can firstly extract

the url structure of these links. By storing the urls of *results pages*, we can extract a common url structure by using the longest common url prefix which recurs in these urls and which doesn't occur in other pages. Similarly we can keep track of the xpaths which lead us to *results pages*. We can also go a level further and keep track of pages which lead us to *results pages*, and we can classify these pages as *index pages*.

Essentially we start with a general crawling strategy, and bit by bit, we build up a whitelist and a blacklist of links and xpaths to follow. Then, when we have gone over a certain threshold of crawled pages, we can start restricting the crawler to the learned urls and xpaths. This would allow us to learn what we would have needed to enter as manual input and thus greatly improve the efficiency of the crawling.

ii. Machine learning solution for filtering question-answer pairs

Another shortcoming we encounter is that when we extract question-answer pairs, we are unable to filter them based on their content. As we saw earlier, the answers we extract do not always answer the question extracted. This is because in forums, there is often a discussion before a correct answer is given. The difficulty then is extracting this answer from the rest of the discussion. The current solution is not yet able to differentiate between general discussion and actual answers.

A possible solution to solve this would be by deploying a machine-learning solution to classify posts as either answers or general discussion. The envisioned method would be to train a model using FastText to distinguish between the two. As training data, we would use question-answer pairs from Stack Overflow as positive labels and general forum discussions taken from a chat forum as negative labels. Once the model has been trained, for each post extracted by the crawler, we would then filter by only selecting those that the FastText model classifies as answers.

If the solution were to work well enough, this would greatly help us filter the content we scrape in order to provide better quality answers in the database.

iii. Complete automation

The final missing step would be to integrate all the steps together to get a fully automated crawler. The end-to-end solution would start with a Swisscom product list, extract a possible list of sites to crawl from an automated Google search and then crawl these sites to extract the question-answer pairs they contain. Such a solution would be able to run in an automated and independent fashion to build a large scale database of question-answer pairs.

VI. Conclusion

In this part of the paper we have presented our work on building a systematic crawler for extracting question-answer pairs. We have divided the problem into two sub-problems for which we have provided solutions and evaluated their performance.

Firstly, given a webpage, we have designed a solution to detect whether it contains question-answer pairs, and extract them if so. Overall the proposed algorithm performs well and can handle a variety of different page types. We have a classification accuracy of 92% and a parsing accuracy of 85%. This self-contained problem is already a big step towards solving the general problem.

Secondly, given a website, we have explored various methods for crawling in a targeted manner. Our baseline simply involves crawling the entire site, which works but takes time and can lead us to off-topic pages. By manually inputting only three parameters (url-guiding method), we can improve the crawling efficiency by a factor 10.

Our solution still presents some shortcomings. Most notably, we would like to avoid any manual input. For this, we envisage implementing a learning system to progressively learn which urls and xpaths to follow. Sec-

ondly, our system doesn't understand the content it extracts and thus cannot distinguish real answers from general discussion. We envision a machine learning solution to detect and retain only proper answers to questions.

When we put everything together, we have a working solution for extracting question-answer pairs from the web which can be run on any website. The basic building blocks have been implemented, and it is now possible to continue building upon them to improve the quality of the extracted question-answer pairs.

Part II
Information Retrieval Systems: Models & Performance

VII. Introduction

i. Introduction

In the previous section, we described a website-independent method to collect question-answer pairs using web scraping in order to build a database of documents. In what follows, we will use the word "document" to designate a question-answer pair. This includes additional metadata pertaining to the question-answer pair such as the author, the date, the number of views etc. A question is composed of two parts: a title and a body.

In this section, we describe several information retrieval system models, optimise their parameters and analyse their performance.

ii. Problem formulation

We start by formalising the problem and the hypothesis we will use.

One of our ideas was to find relationships between questions and answers. However, this gave us very bad results. In reality, questions look like other questions and answers look like other answers. Hence the following models will essentially work by finding the closest questions (in the database) to the input query. In what follows, we will thus look at the problem of finding questions that are similar to the input query. The final question-answering system will then output the answers to these similar questions.

iii. Model description

To design our information retrieval systems, we used different approaches including machine learning, different databases structures and a weighted version of the PageRank algorithm. The first models were built on a database system based on inverted indexes (via Solr). Then we moved to a graph database (Neo4j) to improve the performance via the use of metadata.

VIII. Metrics

We first introduce the evaluation metric and the test set we will use to evaluate the performance of our systems.

i. Evaluation metric

Our metric should measure how close the input query is to the output question.

To do so, we have built a dataset of documents composed of original and duplicate questions. The idea is the following: we query the system with an original question and we expect to get the duplicate questions among the first results. Indeed, duplicate questions are just another way to formulate the same problem. A good system should have this property.

A cheap way to build such a dataset is to use Stack Overflow: duplicate questions are marked by online users with a reference to the original question.

Our approach is the following:

We define two ways of counting the duplicate questions in the top k results. These two metrics correspond to two different types of

Algorithm 3 Evaluation method

for each original question q : **do**
query the system
keep the first k results and count the duplicate questions
end for

user profile: the one who clicks on the first relevant result, and the one who considers all the top k search results and selects the best.

ii. Best rank

In the best-ranking approach, we only consider the rank of the first duplicate question. If the rank of the first duplicate question is bigger than k, then we consider the query to be a miss. If a query is considered to be a miss, then we do not take the result into account when we compute the average best rank.

The best possible rank the system can get for a question is 0.

iii. Recall

In this approach, we count the number of duplicate questions in the k first results. Then we use the definition of the recall:

$$\text{recall} = \frac{|\{\text{relevant docs}\} \cap \{\text{retrieved docs}\}|}{|\{\text{relevant docs}\}|}$$

If there are no duplicate questions in the k first documents the recall is 0.

An ideal recall would be 1.

IX. Information retrieval systems based on inverted index databases

In this section, we present our first two information retrieval systems based on an inverted index database (Solr). The baseline model is a simple query. The second model consists of a re-ranking of the results given by the query. The re-ranking is performed by a Dual Encoder LSTM network on top of the database. The re-ranking tries to take into account the patterns existing between the query and the questions in the documents.

i. Building & querying Solr

Solr performs natural language preprocessing when a document is added to the database. Solr inverts the classical page-centric data structure (document->words) to a keyword-centric data structure (word->documents), hence each field (the question body and the question title) is preprocessed using stop words, a word delimiter filter, a lower-case filter and a Snowball stemmer for the English language.

The same filters are applied to the input when a query is performed.

ii. Description of the models

ii.1 Baseline

The Baseline Model is obtained by only querying the question title and question body fields of the query.

Solr will then do its own scoring based on the following formula:

$$\text{score}(d,q) = \sum_{\text{t in q}} \text{tf(t) idf(t)}^2 \text{norm(t,d)}$$

where

$$\text{tf(t)} = \sqrt{\text{frequency}}$$

$$\text{idf}(t) = 1 + \log\ (\ \text{numDocs}/\ (\text{docFreq+1})\ )$$

Note that this formula is very close to the TF-IDF approach.

ii.2 Dual encoder LSTM on top of Solr

The second model consists of a Dual Encoder LSTM used to re-rank Solr's first k results. The LSTM approach gives very good results on the Ubuntu dataset. Keep in mind though that the original problem in the paper that presents the LSTM model is not quite the same as our problem. Indeed we are dealing with documents that are question-answer pairs whereas the Ubuntu Corpus is a dialogue corpus.

The structure of the LSTM is the same as in the paper [1].

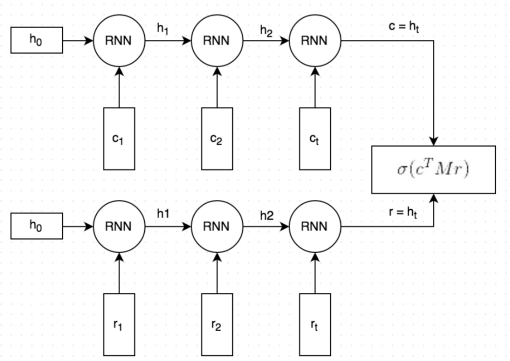

Figure 1: *Structure of the Dual Long Short Term Memory Encoder*

- Both the question title and the question body are split into words and embedded into a vector using GloVe.
- The words are given to a recurrent neural network
- The RNN captures patterns between the question title and the question body.
- M is learned during the process (see figure 1)

While the LSTM Dual Encoder may not be the best known method that implements neural networks for question answering, it has good results when trained with the Ubuntu corpus on other problems (see [1]).

We have trained this LSTM Dual Encoder on two datasets: the StackOverflow dataset we created (about 120 000 documents) and on the Ubuntu dataset. The StackOverflow dataset is closer to our problem. This dataset was built with the crawler described in the previous part of this report.

In our case, the context is the body of the question and the title is the utterance. The LSTM is trained to determine whether the question title is linked with the question body. To do so, we feed the LSTM a training dataset where each item is composed of a question title, a question body and nine distractors taken randomly from the dataset. The LSTM detects patterns and learns to distinguish whether a question body is associated with a given question title.

iii. Scores on test set

The evaluation was performed on a set of 200 questions that are disjoint from the validation set (100 documents) used to perform parameter optimisation. For each model, we return the top 10 results. The model called Stack Overflow Top 10 refers to the LSTM trained using the Stack Overflow corpus and was used to re-rank the first 10 answers provided by Solr. The model called Stack Overflow (resp. Ubuntu) refers to the LSTM trained using the Stack Overflow (resp. Ubuntu) corpus and was used to re-rank the first 100 answers provided by Solr.

iii.1 Best rank

We perform the evaluation using the best-ranking metric. We get the results displayed in table 7. As a reminder, the column "Rank" refers to the average rank of the best ranking duplicate question returned over 200 original question queries. The column "Misses" shows the percentage of times no duplicate questions were found in the top 10 returned questions.

Model	Rank	Misses
Baseline	2.8	16.5%
Ubuntu	6.9	87.0%
Stack Overflow	4.6	70.5%
Stack Overflow Top 10	4.1	16.5%

Table 7: *Solr based models: best rank*

Here the baseline is the model using just Solr. The Ubuntu and Stack Overflow models involve using the LSTM model trained on the corresponding datasets to re-rank the top 100 results returned by Solr. The StackOverflow Top 10 uses the LSTM trained on Stack Overflow to re-rank the top ten results returned by the Solr model.

As we can see from the table, the baseline performs notably better than the subsequent

improvements. The LSTM models used to re-rank the Solr top 100 vastly increase the miss rate and lower the average rank. The LSTM used to re-rank Solr's top 10 results does not manage to improve the average rank either. To conclude the LSTM does not bring any improvement regarding our metric.

iii.2 Recall

We perform the same evaluation using the recall metric over 200 original questions.

Model	Average Recall
Baseline	0.1677
Ubuntu	0.0072
StackOverflow	0.0499

Table 8: *Solr based models: average recall over 200 questions*

With both metrics, changing the dataset from Ubuntu to Stack Overflow has improved the results (number of misses decrease, the average rank is better and the recall increases). However, although using a specialised dataset gives better results, the LSTM approach remains inferior to the baseline.

Conclusion: Although the LSTM re-ranking approach might give very good results on other problems (see [1]), in our case the re-ranking does not perform well with respect to both metrics. The baseline model performs much better.

X. Systems based on a graph database

i. Why use a graph database?

The web-crawler gives us documents with a lot of extra metadata. The idea of using a graph database is to try to take into account metadata such as the date, the author reputation and other information. The graph database can be used to model interactions between the words contained in the documents, the authors of the documents etc.

ii. Building the database

In the models we present, we add only a single feature: the document author.

We pre-process the text fields before storing them. To do this we apply the Snowball stemmer, remove stop words, remove urls, apply word splitting etc. We apply the same transformations to the query.

We use the following model for our graph database:

- one node for each document
- one node for each word
- one node for each author

We define the following relationships between words and documents:

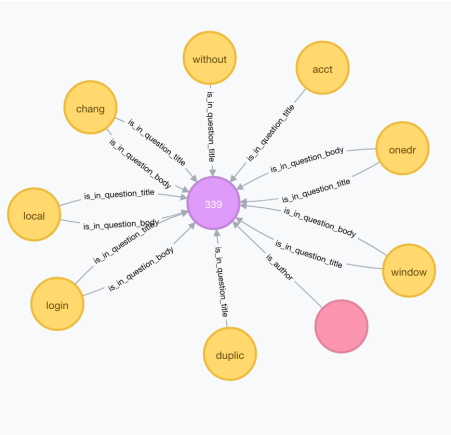

Figure 2: *Neo4j database model*

The yellow vertices represent the words, the purple vertex a document and the red vertex an author. Each document has a TF-IDF and a PageRank score, each relationship between an author (respectively a word) and a document has a PageRank weight.

iii. Description of the models

We have implemented two types of algorithms that run on top of the framework provided by the Neo4j database. The first approach is based on TF-IDF and does not take into account the context (authors). The second is based on

PageRank algorithms using one additional feature (the author). The second one could theoretically be scaled up and many other features could be added.

iii.1 Baseline: TF-IDF

The baseline model is based on the TF-IDF formula.

Algorithm 4 Baseline

```
for each document do
    for each word in the filtered query do
        compute TF-IDF(word,document)
    end for
end for
return : Top k documents
```

Baseline improvements

To optimise this model, we computed two different TF-IDFs: one using the question title and one using the question body. We then took a linear combination of the two TF-IDFs. Hence the score of a document is given by:

$$score = c_{title} * TF\_IDF_{title} + c_{body} * TF\_IDF_{body}$$

Using this new score, we maximise the best-ranking metric described above and find the best linear combination via a grid search. The results are described in table 9, where Average Rank corresponds to the average rank of the best ranking duplicate question on our evaluation set of 200 original questions.

c_{title}	c_{body}	Average Rank
0	1	4.0
1	1	3.5
1	10	3.7
1	0	3.6
10	1	3.3

Table 9: *Coefficient optimisation for TF-IDF over 100 questions*

Thus for our algorithm we will set: $c_{title} = 10 * c_{body}$

iii.2 Models based on the weighted PageRank algorithm

The PageRank algorithm is the famous method used by Google to rank website pages. The idea behind this algorithm is to give higher weights to pages that contain the words that appear in the query. The weighted algorithm is an adaptation of this paper [2].

PageRank in a weighted undirected graph (documents and words)

The second type of model is based on a weighted PageRank algorithm on a subgraph. The subgraph considered is composed of all words that appear in the query and all documents related to those words.

Algorithm 5 Weighted PageRank

```
Input : Query Q
Extract the subgraph
Initialise PageRank for all nodes to 1
for each word in the query do
    Set PageRank weights
    for each vertex v do
        compute PR(v)
    end for
end for
return : Top k documents
```

Where PR(v) is:

$$PR(v) \propto \sum_{\text{neighbour of v}} \frac{PR(u) * \text{weight}(u,v)}{\sum_{\text{neighbour of u}} \text{weight}(u,w)}$$

We tried out various methods to set the weights of the edges between documents and words. Firstly, we tried setting all the weights to one (unweighted PageRank). Secondly, we used the appearance frequency of each document. Thirdly, we tried using TF-IDF. We evaluated each model on a set of 100 original questions. The results are displayed in table 10.

Initialisation Type	Number of Matches
Unweighted	0.3%
TF-IDF	0.27%
Frequency	0.54%

Table 10: *Weight initialisation of PageRank*

From these results, we decide to use the frequency as the weight for the edges.

Adding additional features

To take into account the authors, we ran a weighted PageRank where we took the number of documents each author has written to be the weight of an edge between an author and a document.

Now each document has two scores, one coming from the word PageRank algorithm and one coming from the author PageRank algorithm. To merge these two PageRank scores, we took a linear combination of the two normalised scores. We use the best-ranking metric described above to find the best linear combination via a grid search.

$$score = c_{word} * PR_{word} + c_{author} * PR_{author}$$

We optimise by taking the average best rank over 100 original questions:

c_{author}	c_{word}	Best rank
1	1	2.8
1	10	2.5
1	100	2.6
10	1	7.5
10	10	2.8
10	100	2.5
100	1	5.8

Table 11: *Coefficient optimisation for PageRank*

From these results, we decide to use $c_{word} = 10 * c_{author}$ as the linear coefficients.

iv. Scores on test set

In this section, we compare how well these algorithms perform on the test set. The evaluation was performed on a set of 200 questions that are disjoint from the validation set (used to perform parameter optimisation).

iv.1 Best rank

We perform the evaluation using the best-ranking metric over 200 original questions.

Model	Rank	Misses
TF-IDF	3.2	24.25%
TF-IDF opt	2.65	23.5%
PageRank	2.6	27%
PageRank opt	2.4	22.75%

Table 12: *Neo4j-based models: average best rank and percentage of misses*

We can observe two things:

- Optimising over the test set improves the results a lot (both in terms of rank and misses)
- The TF-IDF approach is beaten by the optimised weighted PageRank algorithm. This is a bit unexpected, as by adding the authors we have improved the score but our metric does not directly consider information that comes from the author.

iv.2 Recall

We perform the evaluation using the Recall metric over 200 original questions.

Model	Average Recall
TF IDF	0.076
TF IDF opt	0.0796
PageRank	0.070
PageRank opt	0.088

Table 13: *Neo4j Based Models: average recall over 200 questions*

Conclusion: The weighted PageRank performs better than the basic TF-IDF approaches using graph databases.

iv.3 Comparison

In table 14, we compare the baseline method (inverted index database) with the PageRank

opt method (graph database).

Model	Best rank	Misses
PageRank opt	2.4	22.75%
Baseline	2.8	16.5%

Table 14: *Model comparison: best rank and misses over 200 questions*

Model	Average Recall
PageRank opt	0.088
Baseline	0.1677

Table 15: *Model Comparison: average recall over 200 questions*

Conclusion: The weighted PageRank performs better than the baseline approaches regarding the average rank, but has more misses (22.75% vs 16.5%). The recall is also not as good as Solr's.

The weighted PageRank algorithm seems promising but probably need improvements.

Conclusion

In this report, we have studied the problem of building a question-answering system in an end-to-end fashion. This has involved two major tasks. Firstly, we looked at how to populate a knowledge base, and secondly, we looked at various models for storing and querying the data.

The first task involved building a generalised system that is able to crawl any question-answer site or forum and extract the information it contains. We broke the problem down into two sub-problems: parsing individual webpages, and efficiently crawling question-answer sites. We implemented solutions for these sub-problems and evaluated their performance.

The second task involved studying different ways of storing and querying the question-answer pairs. Our goal was to see if using a graph database we could take into account more metadata and thus improve the quality of the answers returned. Based on our implementation of the different methods, we see that the Solr-based approach outperforms the graph database approach for two of our metrics (miss rate and recall), but the graph database approach is promising with regards to the average rank metric. Note that Solr's good performance was to be expected as Solr is a state-of-the-art system used in production. Also, we do not claim to have exhaustively studied the graph database implementation and it is possible that further improvements concerning the implementation will be able to improve the system's performance with regard to the miss rate and the recall metrics.

All in all, we have accomplished two things. Firstly, we provide a basic framework for the systematic extraction of question-answer pairs from the web. Secondly, we have studied various implementations of question-answering systems and concluded that while the Solr-based implementation performs the best, the graph database approach has some promise and merits further research to continue improving the system.

References

[1] The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems. By Ryan Lowe , Nissan Pow , Iulian V. Serban and Joelle Pineau

[2] Weighted PageRank Algorithm. By Wenpu Xing and Ali Ghorbani

[3] GloVe: Global Vector for Word Representation. By Jeffrey Penningtonm Richard Socher, Christopher D.Manning.