DROSSER: A Searchable Data Structures Textbook

Katherine Krajovic
Department of Computer Science
Brandeis University
Waltham, MA, USA
krajovic@brandeis.edu

Abstract

This paper presents DROSSER, a queryable version of the *Open Data Structures*[1] textbook. DROSSER enables students to input a natural language question and have the most relevant section of the textbook, which will hopefully help to answer their question, displayed to them. A simple web app serves as the interface where students can pose questions and view the results. Users can also rate the effectiveness of each returned section thereby providing metrics that can be used to evaluate system improvements.

1 Introduction

The move to online learning in the last year and a half has required drastic adjustments to the format of office hours. Rather than sitting together in the computer science lounge, students and TAs gather in Zoom meetings. This virtual setting has made communication more difficult, with concepts often taking longer to convey, and students more reluctant to collaborate and discuss ideas together.

As a result of all of this, in the fall of 2020 we observed office hours that were taking longer than ever, with many students spending long periods of time either in virtual waiting rooms or sitting silently in a group meeting waiting for a chance to work one-on-one with a TA. TAs meanwhile, were often left answering the same few questions repeatedly, or spending long stretches of time trying to help one student with a specific problem while others were kept waiting.

To attempt to mitigate this, Julian Fernandez, Katie Krajovic, Chester Palen-Michel, and Vicky Steger created ATAM[2], a virtual TA chatbot with the idea that he would be deployed into office hours waiting rooms to chat with students, answer common questions, and refer them to a TA for more complicated or sensitive (grades, advice, etc.) questions. The goal of this system was to minimize student wait times, particularly for common or straight forward questions, and maximize the time that TAs could spend working on challenging debugging or conceptual questions.

One aspect of the ATAM project that left some room for improvement was the ability to search course materials for answers to content-based questions from students. That is where the inspiration for DROSSER began.

The idea of DROSSER is that a student should be able to ask any question about the course content that might occur to them while studying or working on an assignment, and be provided with course materials that help to answer that question. Without a system like this the student might have to resort to a ctrl-f exact string matching search of the course materials to attempt to find relevant information. Alternatively, they might turn to a web search engine which, while most likely highly accurate, might use language or reference concepts that are unfamiliar and therefore confusing to the student. This may be particularly misleading in more fundamental or introductory courses where students might be building skills or knowledge in a particular order and therefore not prepared to understand and sift through everything found online.

DROSSER can process natural language questions and return results based on more than just exact string matches. By querying course materials only, DROSSER should return results that are in line with what the students have been hearing in lectures or reading in assignments and course preparation. This should make the results easy for the students to understand and therefore more effective. Ultimately we hope that this system can be an additional educational resource for students in a time when learning seems harder than ever. Additionally, if DROSSER can consistently answer straightforward content questions, students should

[1] https://opendatastructures.org/ods-python/

[2] https://github.com/krajovic/ATAM

need to spend less time waiting to work with TAs, and TAs should have more time to devote to the most challenging problems.

2 DROSSER

This section describes the design of the DROSSER system. Many things mentioned here can be visualized in Figure 1.

2.1 The Content

Because this system is not currently being used in an actual course, an open source online textbook was used to simulate course materials. This section describes the text and how the information inside of it was organized and stored.

2.1.1 The Textbook

While ultimately this system should work with all the material from a given course, this initial version is mainly a proof of concept, and therefore uses a single textbook to represent the course material. As mentioned previously, the textbook used in the DROSSER system is *Open Data Structures*. Several different versions are available and for this project, the Python/pseudocode version was used. The textbook is also available in html format, which made for relatively convenient usage in the web-based user interface.

The textbook contains 14 chapters of content, with anywhere between 2-8 sections in each chapter. Each of these sections is represented as one webpage. Within each section, along with a few introductory paragraphs, there are usually between 0-5 subsections, which are numbered and titled portions of the webpage.

All of these pages were saved locally for use in the user interface. Preprocessing was performed in which menu and navigation bars were removed from the html, as these would have enabled a user of the system to navigate to the Open Data Structures website rather than staying within DROSSER. A few specific hyperlinks were left in the html: a link to the main Open Data Structures website at the very bottom of each page, and any links to other sections or figures in a given section's content. In a future iteration these in-content links should be remapped to pages inside the DROSSER system, but that was not performed for this project.

2.1.2 Documents

The chunks of content that were considered documents for the purpose of this project were the aforementioned subsections. In order to obtain these, each subsection in a given section needed to be identified and then all content within that subsection extracted. In total there were 149 documents.

These documents were used to build an Elasticsearch index. They were indexed with an ID number, a title (representing the section title), a subsection number, a subsection title, all textual content in the subsection, and a vector representation which was generated using the Universal Sentence Encoder Embeddings (Cer et al., 2018).

These documents were the potential results in the system, against which user queries would be compared.

2.2 The Query

As previously mentioned, users interact with the system by providing natural language questions. Realistically, the user can input any text at all, a question, a command, a few search terms and the system will do its best to return a relevant result.

After a user provides a question, it is represented in our system as a Query object which contains three parts: the original text from the user, a search string, and a query embedding. The search string is generated by removing stop words, which include question words, and normalizing the tokens by lower casing and removing punctuation. The embedding is generated in the same way described for the documents in the previous section.

2.3 Elasticsearch

The search string portion of a Query object is used to query Elasticsearch to retrieve an initial set of relevant documents. Specifically, a simple multi-match query that compares the search string to the section title and subsection title is used. This query format was ultimately selected after substantial experimentation with different permutations of parts of the Query object as input matching against titles, texts, embeddings, or combinations of all of those.

Although this method of using just the query string to match against titles seems simple, it almost always returns relevant results. As will be discussed in the following sections, there are several additional steps in the process which help to refine the results obtained with this initial query. Searching simply based on title provides a solid list of candidate sections, and crucially helps us to avoid false positives which may have been returned simply due to the wording of a user's question or a single tangential sentence in a document.

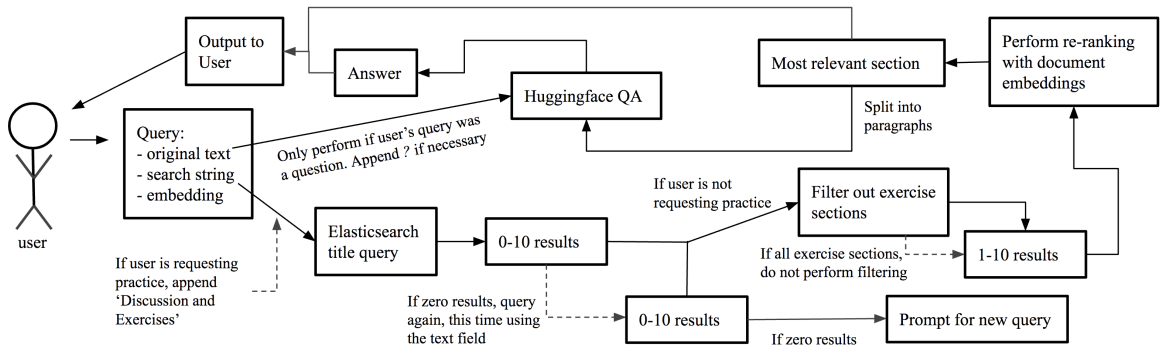

Figure 1: This figure displays a representation of the DROSSER system.

One case in which this approach does not work well is when a user's query is so specific that it does not contain any contentful words that are found in the subsection titles. As a simple example, imagine a question like *What is the efficiency of the pop operation?*. This would most likely be answered in a section about Stacks, but the token *stack* does not appear in the user's query. A title based Elasticsearch query in this case is unlikely to return the most relevant section.

To account for this, if it is ever the case that querying based on titles alone returns no matches, a second query is performed, this time on the text of each section. This substantially broadens the scope of terms that can be matched, and will hopefully help to solve things like the *pop* problem mentioned above. Querying based on text rather than embeddings is used in this case because this was seen to result in fewer false positives, for reasons similar to those discussed previously.

Sometimes even expanding the query in this way returns no matching documents. This is most often when the user's question is completely irrelevant to the content of the textbook or about relevant content that the textbook just doesn't cover. In these cases no section is returned to the user, and they are asked to try a different query.

2.4 Filtering

After the Elasticsearch querying process is completed, the system is left with a list of relevant documents which may contain anywhere between 1-10 results. At this point the results are filtered to remove any *Discussion and Exercises* sections unless the user appears to be specifically requesting something like practice problems or examples. There is one of these exercise sections at the end of each chapter. It was noticed very quickly while this system was being designed that in the final step of the document retrieval process, which will be described below, the system was identifying user questions as very similar to the discussion and exercises sections because they were full of questions similar to those that the users were asking. This caused them to be ranked as highly relevant very often, even though a user asking a question about content most likely isn't looking for similar questions as a response.

In order to avoid this problem, all of these sections are removed unless the user is asking to practice or see examples. Right now this practice non-practice distinction is made by checking the user's query for a list of practice related keywords including things like: *practice*, *problems*, *exercises*, *example*, etc. If it is ever the case that removing all exercise sections results in zero remaining sections, or, worded differently, that all of the candidate sections were exercise sections even though the system categorized the query as not specifically requesting practice, then this filtering is not performed. This is done for two reasons: first it is assumed that it is better to return something relevant but imperfect than nothing at all, and second, keyword look-up is a very brittle form of practice non-practice classification, so if all results returned by Elasticsearch are exercises, perhaps this is a sign that the system made an incorrect classification and the user was indeed requesting practice.

2.5 Section Ranking

Once the system has filtered the results down to a final list of candidate sections, the documents are ranked. This is done by computing the cosine similarity between the query embedding and document embedding for each candidate section. The document resulting in the highest cosine simialrity is returned.

There is one case in which this step is not per-

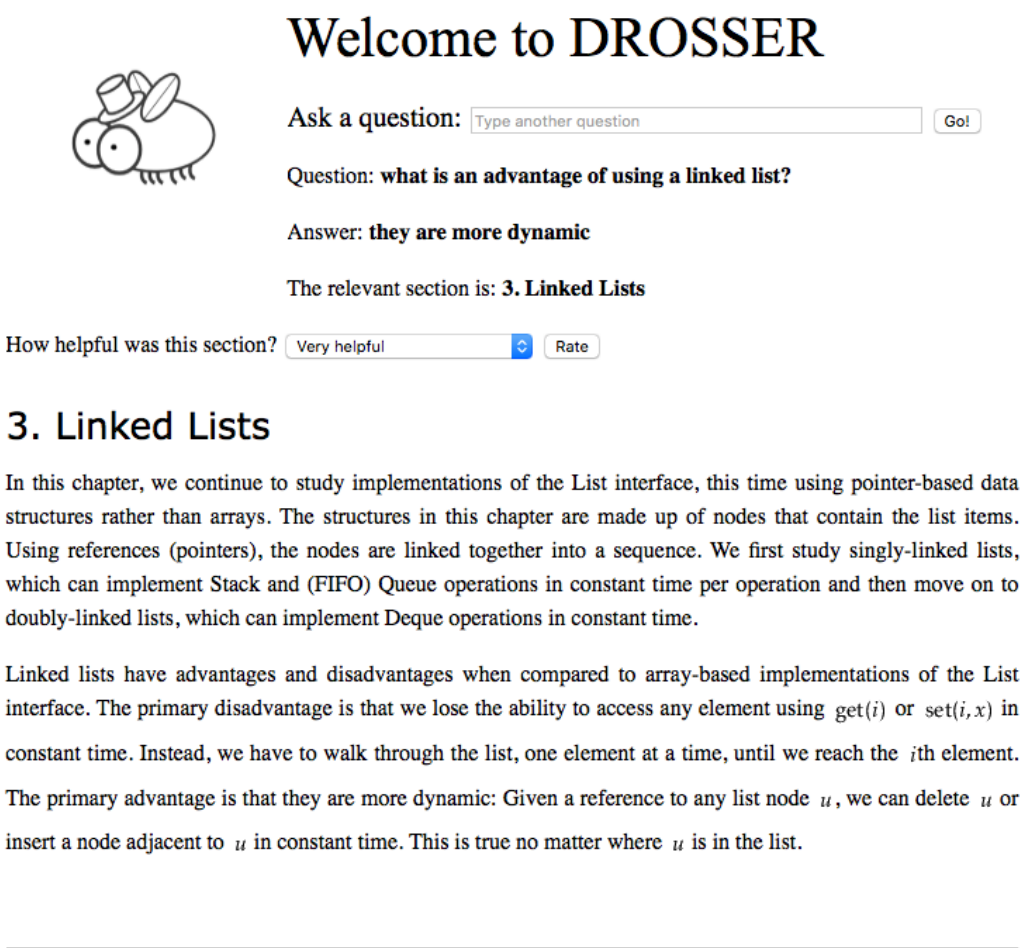

Figure 2: This figure displays the User Interface. The user can ask a question in the *Ask a question:* bar and see their question, the answer, and most relevant section that is returned to them.

formed. If the query string contains two or fewer tokens (recall that the query string has stop words removed) then the first result returned by Elasticsearch is selected and no document re-ranking based on the embeddings is performed. This is meant to account for the case where the user input one or two keyword search terms rather than a fully formed natural language question. In this case a sentence embedding of just a few terms will likely not be informative, and was observed in experimentation to produce less relevant results than the simple query of section and subsection titles.

2.6 Question Answering

At this point the single most relevant subsection has been identified and will be displayed to the user. Additionally, the query and text are run through a question answering system to attempt to provide an exact answer to the user's question if possible. The QA system currently used in DROSSER is Huggingface's Extractive Question Answering pipeline[3]. If the user's original query is not a question (if it does not contain any question words) then this question answering step is not performed because there is no question to provide an answer to.

Because the QA system works best with short texts, the content from the relevant subsection is split into paragraphs, simply by splitting on new lines. If any of the paragraphs resulting from this splitting have fewer than some threshold value of characters then they are not used. These are most

[3] https://huggingface.co/transformers/usage.html

commonly things like a description of an image, or a short mathematical formula that will likely not contain the answer, at least not without additional context. At the moment this threshold is set to 100 characters, because in experimentation this worked reasonably well. If the QA system changes in the future, this can be easily adjusted.

Each remaining paragraph is run through the QA pipeline along with the user's original query. Interestingly, while evaluating the system it was noticed that the presence or absence of *?* often impacted the results of the QA system, with *?* frequently producing better results than the lack thereof. Because of this, *?* is appended to any question that does not already end with a question mark before it is fed into the QA system. The answer that was returned with the highest confidence is the one that is ultimately displayed to the user.

2.7 The User Interface

The user interface for DROSSER is a web app built in Flask[4]. The home page displays a simple welcome message and prompt for the user to ask a question. Once the user asks their first question, the results are displayed in a format similar to that shown in Figure 2.

The user will always be able to see a copy of the question that they most recently asked, the *answer* provided by the QA system, and the title of the relevant section returned by the document retrieval system. In cases like the one shown in Figure 2 in which there is only one subsection to be displayed, the relevant section is guaranteed to be at the top of the page. In some cases when there are multiple subsections in the section that is displayed, the user may be required to scroll down to locate the relevant section.

The user may enter another question in the search bar, or they can provide feedback on how helpful the returned section was. The rating options are *Very helpful*, *Somewhat helpful*, *Relevant*, *Informative but not relevant*, and *Irrelevant*. Once the user submits this feedback using the *Rate* button, they are taken back to a page similar to the home page, and the feedback is stored. When the user terminates the session, the counts for each rating option are written to a JSON file. This file is then loaded on startup so that performance can be tracked across sessions.

[4] https://flask.palletsprojects.com/en/1.1.x/

Question	Sect.	QA
whats the runtime of binary search	4	2
what is an adjacency matrix	5	1
give me an example of a recursion problem	4	–
when do I need to rehash	1	1
how does a heap work?	4	–
what is a balanced binary tree?	3	1
is deleting a node faster in a linked list or an array?	3	–
how do I sort a stack?	2	1
what is big o notation?	5	2
how do I traverse a tree level by level?	3	1
what is a double linked list	5	1
how do I compute a logarithm?	5	1
what is the runtime of quick-sort?	5	4
how do I search for an element in a binary tree?	5	1
how do I find an element in a doubly linked list ?	3	1
can you help me practice with stacks?	5	–
how long does it take to rehash?	2	1
what is the runtime of depth first search?	5	5
what does FIFO mean?	5	2
what are advantages and disadvantages of linked lists ?	5	3
Averages	**3.95**	**1.75**
Improved scores after adjustments	**4.05**	**2.06**

Table 1: This table shows the rating for the returned section (Sect.) and the answer provided by the QA system (QA) for each of the 20 sample questions.

3 Evaluation

In order to evaluate DROSSER, two people asked the system 10 questions each and then rated the returned sections as well as the QA-provided answers according to the scale defined above. The scale was mapped to integer values so that some basic averaging could be performed with *Very helpful* being worth 5 points and *Irrelevant* being worth 1. These questions along with the corresponding scores can be seen in Table 1. Note that only the raw scores from the initial test are included to save space, but the averages from before and after system tweaking are listed for comparison.

3.1 Results

After an initial set of 20 sample questions which can be seen in Table 1, the average score for the section returned was 3.95 and the average score for the QA-provided answer was 1.75. This was

considered to be reasonable performance for the document retrieval system, and very poor performance for the QA system, but this was expected.

The poorest performing queries were analyzed in particular to see if the system could be improved. Many small adjustments were made, but the system was not radically altered because the document retrieval process was working relatively well.

One example of a query that performed badly was *when do I need to rehash?*. The section returned was *1.1 The Need for Efficiency* which does not mention anything about hashing, earning this section a 1. This motivated the addition of a few stop words (*need*, *to*, and several other prepositions) to the stop word list that is used to form the query string. After this change the best matching section was *5.3 Hash Codes*. This is still not a perfect match, but it is much closer than the original section, and earned a score of 3.

Another interesting query to examine was *Give me an example of a recursion problem*. In this query the user is requesting a practice problem, but the section that is returned is *6.1.1 Recursive Algorithms*. This section is relevant to recursion, but has nothing to do with practice problems. Because of this, if a user's query is categorized as requesting practice problems, the phrase *Discussion and Exercises*, which is included in the title of all sections containing practice problems, is appended to the end of the query string to bias the results towards sections with practice problems. After this change the returned section contained practice problems with binary trees and recursive algorithms.

With the addition of these and several other small tweaks, the questions were rerun and the responses reevaluated. The average score for the relevant sections was 4.05.

For the QA portion of the system, the only change that was made was to append *?* to the end of a query if it was not already present. The average score for the QA-provided answers after this change was 2.06.

3.2 Discussion

Overall, the relevant section retrieval portion of the system, which was the main purpose of this work, is performing relatively well. Through testing even with just two beta users, several edge cases were identified and parameters and thresholds could be adjusted to produce improvement on a small set of test queries. Post tuning, the average rating for relevant section retrieval can be interpreted as on average at least somewhat helpful. It is worth noting that many of these tweaks, like those mentioned above, were very specific, and because of this it was not the case that every result improved or stayed the same. A few results got worse, but none by more than one point. With more users testing the system with new questions it is likely that more adjustments, fixes, and tweaks would be identified. Hopefully these would lead to slow improvements over time, but this is a system where optimizing for one kind of query might hurt performance for another.

The QA portion of this system currently does not work well, and this was anticipated. Even after adjustments, half of the responses provided by the system received a ranking of 1. This means that half the time the system is returning something completely irrelevant. The question answering currently in place is fully off the shelf. It is not fine tuned for this domain or this task. Even if it were performing more accurately, the kinds of answers that it returns are short spans from the text itself, which are useful for simple definitions or facts, but almost never useful for more open ended or complicated questions that might frequently be asked by students. This is an aspect of DROSSER in which future work could lead to substantial improvement.

4 Conclusion

This section contains some reflections and ideas for future work on this project.

4.1 Future Work

The most obvious opportunity for future work on DROSSER is the question answering system. I am very interested in writing a QA system from scratch. I have recently been doing a lot of reading and a bit of work in question classification for open domain question answering. I am interested in investigating how this, and all steps in a QA process might need to be changed to make them suitable for educational question answering where the best answer might not always be the shortest, and the cost of being wrong is potentially very high.

Credit for this next idea goes entirely to Chester, but if we were to get people using this system, as part of the feedback when a provided answer was irrelevant or unhelpful we could ask the user to select the span of text that did answer their question (assuming a relevant section had been returned).

The question answer pairs in this feedback could then be used in the training of a better QA system, or at least the fine tuning of an existing system to question answering in an educational setting.

One small thing that might improve the existing system would be to make use of content specific keywords. Keywords could be automatically extracted from the documents with a simple metric like tf-idf, and once these were identified it could be interesting to investigate how they could be used in the initial Elasticsearch query.

Another change to try would be to generate several embeddings from smaller sections of text rather than just one for each subsection. The subseciton text could be split into paragraphs (similar to what is done in the QA process) and an embedding could be generated for each one. Then the cosine similarity could be computed between the query and each paragraph of each section. This was not attempted in this iteration of the project because of runtime concerns.

A final small thing would be to make some changes to the user interface. I spent substantial time trying and failing to get the webpages to automatically scroll to the relevant subsection, but I was always a few hundred pixels off. I'm sure that there is a work around for this somewhere, and other general ways in which the UI could be improved.

Another loftier goal for this system is getting it to work with resources that are not just this textbook. Ideally, this should be a system where after each lecture, someone can upload a few new PDFs of content and have them incorporated into the searchable material. One big step in this would be processing PDFs, even just to extract the text let alone identify independent sections. This is not a trivial task and still oversimplifies the problem because realistically all different kinds of resources could be uploaded, like textbooks, slides, worksheets, assignments, and all of these would probably need to be handled differently. Even after the text is extracted, the different resources might need to be stored and queried in different ways. Additionally, certain tweaks that were made to improve this system would not be relevant for other resources. The end goal is transforming this into a generic architecture that can take in any kind of course content and make it queryable through natural language. I think this would be challenging to create, but also very useful if it existed.

4.2 Reflection

This project was a lot of fun and I learned many things along the way but I will emphasize three. 1. Web apps can be frustrating sometimes, but building something that you can actually interact with is extremely satisfying. 2. Just because we have a bunch of cool models does not mean every problem is solved. There is still room for creativity in how we use and apply them, and sometimes the fanciest newest thing isn't the best way to solve a problem. 3. Everything is worth documenting, even when it feels like you're flailing around in the dark trying to understand how something works.

DROSSER is far from perfect, it's probably not even helpful right now, but it's a real thing that someone could use. The fact that after two years at Brandeis I have the ability to see a problem and go build something to fix it, or just generally build something that might make learning easier for someone is the coolest thing ever. I am thankful for the opportunity that this class has provided to let me work on something like this. It's helped me see that I have tools now, I can make things, and hopefully I can go out and use my powers for good.

Acknowledgements

Inspiration for many of the ideas in this project should be credited to Julian, Chester, and Vicky. Without them I also most likely would never have been confident enough to attempt to build this system.

I also have to credit my brother Daniel for naming the fly shown in Figure 2 Drosser, thus inspiring the name of this system, as well as my mum for suggesting that my original design could be improved by adding a *jaunty hat*.

References

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*.