Applying Knowledge Discovery and Question/Answering to Intelligent Tutoring System

Belinda Copus, CS 5560 Knowledge Discovery and Management, Summer 2017

***Abstract*—Computer Science students often experience difficulties in introductory computer programming courses. This phenomenon is a world-wide issue. The individual characteristics and learning needs of the student is often considered in designing a remedy to this problem. Knowledge Discovery and Question/Answering systems are explored in this paper as a potential addition to Intelligent Tutoring Systems.**

I. Introduction

Many collegiate Computer Science programs have in common the issue of high failure rates in their introductory programming courses. Multiple studies have established that failure rates (composites of "drop," "fail," and "withdraw," or "DFW," in this paper referred to synonymously as "failure" or "DFW") are consistently around 33% for CS1, in virtually every studied geography, and have been near this level for decades [1]. Since CS1 is both the gateway course and a gating course for CS majors, failure in CS1 is an obvious and significant predictor of a student's departure from a CS degree program. Many have explored why this phenomenon is occurring and how to improve high failure rates in CS1. In the United States currently, there is a greater demand for computer science trained workers than computer science programs are able to supply [2]. It is also widely reported that record numbers of incoming freshmen are choosing to study computer science.

Some have applied Educational Data Mining (EDM) to discover knowledge concerning student characteristics, such as age, gender, income, class, status of discipline, etc. and are able to predict or identify students at higher risk of failure at the earliest possible time, to inform teachers [4]. Others have surveyed students regarding their perceived reason for failure. Responses often don't know how to start writing a program, or can't work through syntax issues [7]. Given this fact, perhaps an improvement in the tools used for learning should be considered. The purpose of this paper is to explore a new approach to applying theories and techniques from the field of Knowledge Discovery Management to create better tools for learning computer programming.

II. Related Work

Intelligent Tutoring Systems (ITS) are current technologies being applied to the problem of assisting introductory programming students. Sharada, Shashi, & Madhavi [5] conducted an extensive literature review on ITS and currently available systems. They found that "providing a personal training assistant for each learner is beyond training budgets," but an ITS could "provide personalized instructions to students according to their cognitive abilities" [5]. [5] reviewed ITS on four dimensions: ability to provide student feedback, ability to model a student, ability to predict student performance, and ability to provide academic knowledge. One system was able to predict a student's solution and find relevant examples to demonstrate a more correct solution. Another system relied on a tree structure to model content. Each leaf of the tree represented a topic. Prerequisite knowledge could be easily modeled, and repository or learning and testing materials were associated with each node. Rules were then created to make sure the student was ultimately navigated through the tree [5]. While this system has value, it does not provide any intervention to guide the student to success.

[3] created a "Wizard of Oz" system that considered the importance of students' emotions in motivation and learning. This ITS would choose a tutorial action according to the pedagogical and affective state of the student using a decision network that was able to measure both learning and affect. Their results showed that 90% of students considered that their pedagogical state improved after the tutorial action. The results appear amazing, but one must consider that the "Wizards" in this system were actually teachers rather than a computer. Students were unaware of this fact. This paper did demonstrate that targeted and focused feedback and direction can have a substantial impact. One would hope that the authors would seek to create an ITS that models the Wizard of Oz system.

[6] implemented an IT'S that was an effective tool for improving a students' problem-solving abilities and was capable of mimicking a human tutor. Students using this system were fully engaged in learning-by-doing, which is a strong method for learning in computer programming. The heart of the system was flowcharting. Students first had to create a flow chart to a problem, and the system could then provide feedback to the student on their solution. When a solution was incorrect, the student was presented with a quiz. If the student was unable to answer questions correctly on the quiz, a Bayesian Network

was used to suggest readings and other learning materials for further study. The authors found that their system had a significant impact on learning for students with the lowest levels of prior learning. Although this system was probably the most advanced of those studied in this literature review, it is lacking. The system is solely focused on creating coding solutions to a problem, and not useful in translating a proposed flow chart to code. Secondly, after resources were presented to a student when further study was required, there appeared to be no mechanism to ensure that the student finally mastered the lacking concept. Third, the resources provided were limited to lecture notes.

Knowledge Discovery involves processing data; and for the purpose of this project, unstructured data. The data is processed to determine topics the text represents. The final artifact produced in the process is a knowledge graph. This graph can then be used to discover new knowledge or existing knowledge presented by the data. Knowledge Discovery has a key strength in being able to discover information or relationships that are beyond factoid. The Knowledge Discovery process has been the foundation of Question Answering (QA) systems.

III. Proposed work

One major component required of an ITS is the need to provide the student with targeted information in order to correct and deepen her knowledge. It is proposed that a system be developed based on Knowledge Discovery and a Question and Answering system that could be used as a tool to improve any ITS, but particularly an ITS focused on introductory computer programming concepts. This QA system would be integrated into the ITS. For example, a student would create a programming solution to an assigned problem within the ITS. Errors would be detected and suggestions for correction would be made. Additionally, the student would have the ability to ask a question to the QA system. The system would respond with a correct answer and also an option to work practice problems on the particular concept. There would be other features included in the ITS (i.e. feedback to the instructor, prediction and early intervention), but this writing is focused on the Knowledge Discovery, QA portion of the system. The corpus for this system will be derived from freely available programming language textbooks.

IV. Implementation and Evaluation

Please refer to figure 1, QA Dataflow. The raw corpus is a PDF of a textbook on a specific programming language. For a proof of concept, a Python textbook was chosen. There are some manual operations involved in converting it to a text file, then breaking that up into topical pieces. The author broke it by chapter, so each chapter is a document. There were various challenges with this first step. Obvious tools like Adobe Acrobat ought to do this conversion well, but resulted in lots of artifacts, like words strung together without spaces in between. It turns out that Microsoft Word does a better job of pdf-to-text. Breaking it up was done manually, but could be automated—that's a natural language project of its own.

After we have text documents, we process them through three pipelines at the document level (the chapters), and then the sentence level, and then the concept level. Each of these also gets TF-IDF data at that level, so relevance determinations can be made using TF-IDF at the document level and at the sentence level.

Adding manually-constructed concepts, which are a kind of digest of the contents of the document, was an idea developed late in the project, and is the center pipeline in the figure. This pipeline is not yet fully implemented, but the documents have been tagged with a digest of relevant nouns on the first line of each document. The idea is to extract concepts—especially multiword noun phrases—in the question, and then match those to the concepts extracted from the document to get a second measure of document relevance. This would be combined with the TF-IDF information to get a better score at the document level. Only when the document first scores highly do we move to the sentence level—this is essential to scale the system in a meaningful way.

One of the interesting aspects of the concept-based scoring is that it captures relevance that may be hidden by assumption. For example, the initial corpus consists of a text that covers the Python language. But the word, Python, appears rarely in the text, since it's obviously a Python book. Some chapters never mention the word, Python—so the chapter-level documents may show zero relevance to the word, Python.

If the corpus were extended to a more broad selection of books—some Java books, Python books, Scala books—and a user asks, "Does Python have array slices?", the system should match the chapter documents based on the fact that the user wants to know about Python, and this is a Python book. So the Python concept is hand-

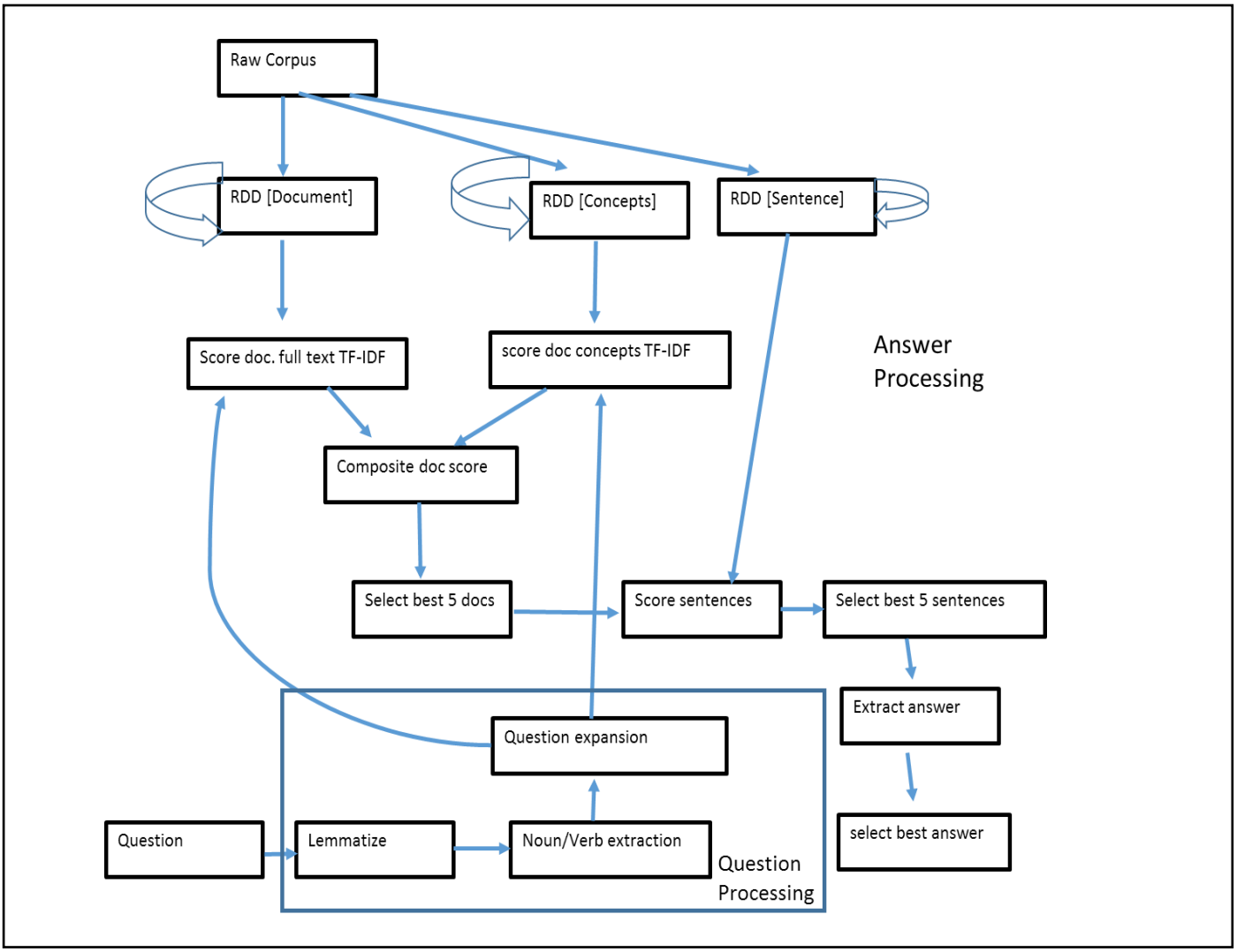

Fig 1. QA Dataflow

coded as a kind of tag, and the system scores the document on that in addition to relying on TF-IDF.

As for the RDDs themselves, the system extracts the lemmas, parts-of-speech, and named entity classes for all of the tokens. Documents are logically comprised of a sequence of sentences so that the system can become more sophisticated later, as we can implement proximity searching.

One other thing of note is that the system stores its processed corpus as Spark RDD distributed object file for processing in a distributed environment for better performance. The corpus prep has been separated from the actual QA system.

The QA system has several components. First, the question is lemmatized. The question is percolated through the NLP pipeline, which is already initialized and resident in memory, so it's nearly instantaneous from the user's perspective. Extraction of the nouns and verbs from the question are performed. These are referred to as "target terms." The TF-IDF hashes for the target terms are found and are saved for later TF-IDF comparison. The type of the interrogative is also extracted. Currently the system supports who, what, what is, when, where, and why question types, with varying levels of sophistication. If the question type isn't one of those, an answer response will be attempted, but the quality is more likely to be poor.

The next step is the last bit of functionality in the current revision of this system. Target terms are expanded through a custom synonym expansion. The reason behind this latest addition is that if a user uses the word "function," we might add the words "method" and "procedure" to the target terms, because they are near synonyms. This technique is somewhat naïve at present, because all added terms are assumed to have a uniform degree of sameness. A whole semantic graph would be better. The problem with the generally-available semantic graphs is that they are not specific enough for the domain—so really, a domain-specific one needs to be constructed.

The expanded query gets scored twice at the document level—once for the whole document text, and once for the concepts—then gets scored at the sentence level for those documents that are most relevant. The five best sentences are selected for further processing.

Processing at the sentence level proceeds as described: There is a default answer, which is to just return the whole sentence. This might or might not be the answer to the proposed question. If the question is a known interrogative type, we try to find it. For example, a "WHY" interrogative means we want to find a clause that begins with "because," and then return from the word "because" to the end of the

sentence. If we can't find such a clause, we return the whole sentence text.

In returning five answers, it is hoped that there will be increased chances of giving the user a useful answer. If the answers are limited it to just the *"best"* one, it is more likely that it will not be the correct answer.

Please refer to Fig. 2 Class Diagrams. The following classes were implemented in this system. Note that some of the classes, Document, Sentence, and Token, are not the Stanford CoreNLP classes of these names. Instead, this system implements new object definitions for those entities that are stored in the object files and distributed as RDDs. This was done because the CoreNLP classes maintain a lot of data as strings—for example, the part-of-speech and named entity class tags are strings, requiring extensive use of string equality operations throughout the question answering code. In order to make the system more performant, the lexically-oriented classes were implemented to use integers where possible, including two enumerated types: Named Entity Class, and PartOfSpeech. In this way, inner loop code was optimized in many places.

Another architectural enhancement was to extract the abstract class AbstractDocument. This class contains the code common to handling Documents and Sentences. All of the TF-IDF scoring is handled identically for sentence-level and document0level scoring, making the code clean and easy to maintain.

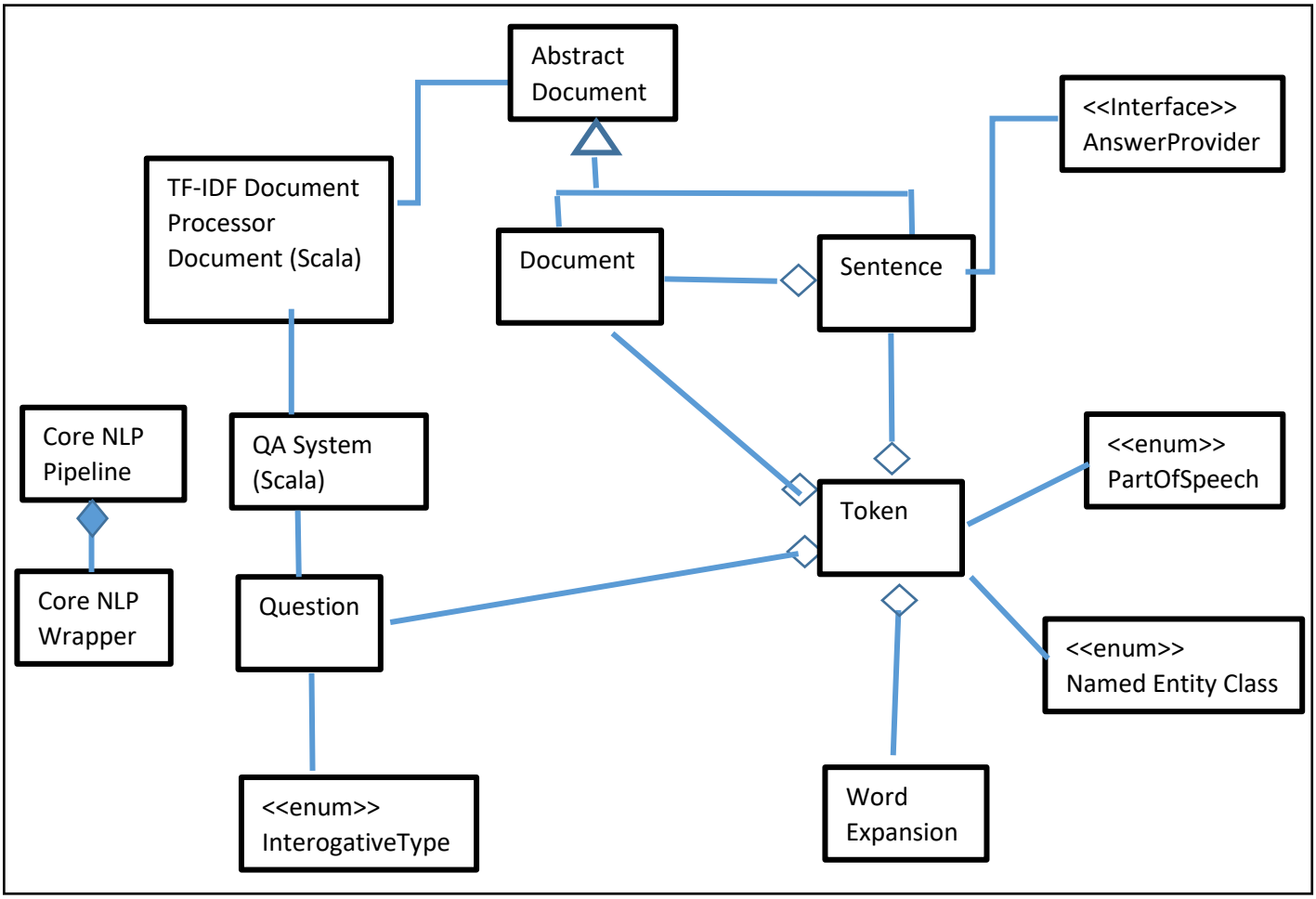

Fig. 2 Class Diagrams

V. Discussion and Limitations

1. Why is it a good idea to specify object invariants in Python?
2. What makes an abstract data type abstract?
3. Are there any advantages to using postfix?
4. Does python provide a split method for tokenizing strings?
5. Can a module contain more than one function with the same name?
6. What is a string slice?
7. What is an accumulator?
8. What is a type conversion?
9. Can I make a subarray from a string?

Fig. 3 Questions for testing QA System

Version One			
Question	Top 5	Extracted	
1	0	0	Why is it a good idea to specify object invariants in Python?
2	0	0	What makes an abstract data type abstract?
3	1	0	Are there any advantages to using postfix?
4	1	0	Does python provide a split method for tokenizing strings?
5	0	0	Can a module contain more than one function with the same name?
6	1	0	What is a string slice?
7	1	1	What is an accumulator?
8	1	1	What is a type conversion?
9	0	0	Can I make a subarray from a string?
	56%	22%	

Version Two			
Question	Top 5	Extracted	
1	**1**	**1**	Why is it a good idea to specify object invariants in Python?
2	0	0	What makes an abstract data type abstract?
3	1	0	Are there any advantages to using postfix?
4	1	0	Does python provide a split method for tokenizing strings?
5	0	0	Can a module contain more than one function with the same name?
6	**0**	0	What is a string slice?
7	1	1	What is an accumulator?
8	1	1	What is a type conversion?
9	**1**	0	Can I make a subarray from a string?
	67%	33%	

Fig. 4 Results

Two versions of this system were tested. The first version did not have the custom synonym feature. The synonym feature was added to the second version. From Version One to Version Two, the principle changes were question expansion by closely-related terms and the move from document-level TF-IDF matching to sentence-level TF-IDF matching. In addition, the new version recognizes "what is" questions as a specific form of "what" interrogatives, and attempts to extract definitions accordingly.

These changes produced modest improvements across the board, and also allowed the system to answer certain questions that were previously problematic. For example, the question, "Can I make a subarray from a string?" was not adequately answered by the first version of the system, since the question does not use the proper vocabulary for Python as expressed in the corpus—in Python, such subarrays are called "slices." Query expansion resulted in the following proposed answer among the top choices: "7.4 String slices A segment of a string is called a slice," which at least tacitly contains the answer.

It is important to note that, although the system performs better when the wrong vocabulary is used, it performs more poorly when the correct vocabulary is used. Question 6, which was adequately handled by Version One, is inadequately handled by Version Two. It is believed that this is a result of the question expansion, in which additional tokens are simply added to the question terms as a "bag of words." This naïve approach needs improvement. A better method would likely be to expand by producing more questions, rather than by simply increasing the number of target terms in the question.

Another interesting observation concerns Question 1, which was not answered adequately by Version 1, but was answered perfectly by Version 2. As there was no change made to extraction of answers for "why" interrogatives, the improved performance is believed to be attributable to the simplification of using sentence-level statistical (TF-IDF) matching exclusively, instead of the two-step matching against documents first, followed by sentences; i.e. the document that contained the correct answer was screened out by the first version of the system.

It is clear that the current state of this QA System is not very effective. Even with the second revision, a 33% success rate was calculated on the test questions. It will be critical to improve this number to at least 95% success. The reasoning is that if this system is to be used by novice programming students, inaccuracy and failure to provide clear responses will only introduce further frustration to the student.

Increasing the size of the corpus would likely help discovery of correct answers. Also, relating programming terms is challenging. An ontology related to Computer Science, similar to the Medical world, would also improve this system.

VI. Conclusion

Knowledge Discovery of unstructured text could be effectively applied to the computer program learning systems used by introductory computer programming students. The addition of this technology to many of the aforementioned Intelligent Tutoring Systems brings additional advantages such as allowing student assistance with questions outside of the context of writing code, the ability to provide a vast amount of content or knowledge, since many textbooks have been written on any chosen programming language, returning answers that seek to educate rather than provide a solution to a coding problem.

A future direction for this project could be to use the QA system for a new augmentation to an Intelligent Tutoring System. This system would have knowledge of common programming errors in addition to textbook knowledge. The system would also track errors not previously unrecognized so as to add to the corpus.

References

[1] C. Watson, & F. Li, "Failure rates in in introductory programmng revisited," ITiCSE'14 Proceedings in Computer Science Education., pp. 39-44, 2014.

[2] Bureau of Labor Statistics. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

[3] Y. Hernandez, E. Sucar, & G. Arroyo-Figueroa, "Building an Affective Model for Intelligent Tutoring Systems with Base on Teachers' Expertise", MICAI 2008, pp. 754-764, 2008.

[4] E. Costa, B. Fonseca, M. Santana, F. Araujo, & J. Rego, "Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses", Computers in Human Behavior, pp. 247-256, 73, 2017.

[5] N. Sharada, M. Shashi, & D. Madhavi, “A Comprehensive Study on Intelligent Tutoring Systems”, International Journal of advanced studies in Computer Science and Engineering, vol. 4, no. 8, pp. 1-7, 2015.
[6] D. Hooshyar, R. Ahmad, M. Yousefi, F. Yusop, & J. Horng, “A flowchart-based intelligent tutoring system for improving problem-solving skills of novice programmers”, Journal of Computer Assisted Learning, vol 31, pp. 345-361, 2015.
[7] B. Wilson, & S. Shrock, “Contributing to success in an introductory computer science course: a study of twelve factors”, *SIGCSE technical symposium on Computer Science Education*. 33(1), 184-188, 2001.