

**TOWARDS A VIRTUAL TEACHING ASSISTANT TO  
ANSWER QUESTIONS ASKED BY STUDENTS  
IN INTRODUCTORY COMPUTER SCIENCE**

by

Cecily Heiner

A dissertation submitted to the faculty of  
The University of Utah  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2009

UMI Number: 3377899

Copyright 2009 by  
Heiner, Cecily

### INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

**UMI<sup>®</sup>**

---

UMI Microform 3377899  
Copyright 2009 by ProQuest LLC  
All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

---

ProQuest LLC  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

Copyright © Cecily Heiner 2009

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

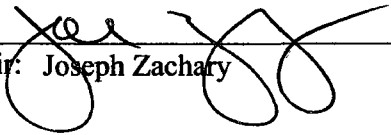
**SUPERVISORY COMMITTEE APPROVAL**

of a dissertation submitted by

Cecily Heiner

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.


9/13/09

  
Chair: Joseph Zachary

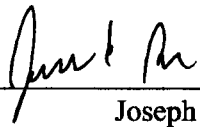
9/13/09

  
Hal Daume III

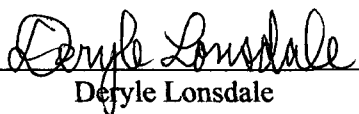
8/26/09

  
Ellen Riloff

7/2/2009

  
Joseph Beck

30 July '09

  
Deryle Lonsdale

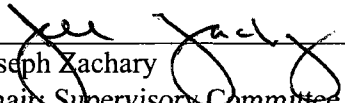
THE UNIVERSITY OF UTAH GRADUATE SCHOOL

**FINAL READING APPROVAL**

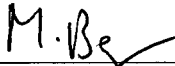
To the Graduate Council of the University of Utah:

I have read the dissertation of Cecily Heiner in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the supervisory committee and is ready for submission to The Graduate School.


9/13/09  
Date

  
Joseph Zachary  
Chair, Supervisory Committee

Approved for the Major Department

  
Martin Berzins  
Chair/Dean

Approved for the Graduate Council

  
Charles A. Wight  
Dean of The Graduate School

## **ABSTRACT**

Students in introductory programming classes often articulate their questions and information needs incompletely. Consequently, the automatic classification of student questions to provide automated tutorial responses is a challenging problem. This dissertation analyzes 411 questions from an introductory Java programming course by reducing the natural language of the questions to a vector space, and then utilizing cosine similarity to identify similar previous questions. I report classification accuracies between 23% and 56%, obtaining substantial improvements by exploiting domain knowledge (compiler error messages) and educational context (assignment name). My results are especially timely and relevant for online courses where students are completing the same set of assignments asynchronously and access to staff is limited.

*for my students*

And the King shall answer and say unto them, Verily I say unto you, Inasmuch as ye have done it unto one of the least of these my brethren, ye have done it unto me  
--Matthew 25:40



## TABLE OF CONTENTS

ABSTRACT.....	iv
ACKNOWLEDGEMENTS.....	ix
CHAPTER	
1. INTRODUCTION.....	1
1.1 Problem.....	3
1.2 Contributions.....	4
1.3 Overview.....	5
1.4 Data.....	6
1.5 Analysis.....	7
1.6 Results.....	8
1.7 Summary.....	11
1.8 Roadmap.....	13
2. PRIOR WORK.....	14
2.1 Information Retrieval.....	14
2.2 Classifying Questions from Novice Programmers.....	21
2.3 Tutorial Dialog.....	27
3. DATA.....	38
3.1 Historical Background.....	38
3.2 System Architecture Overview.....	38
3.3 Participants.....	44
3.4 Data Cleansing.....	46
3.5 Interrater Reliability.....	47
3.6 Dataset Statistics.....	48
3.7 Questions Per Assignment.....	50
3.8 Questions Per Student.....	52
3.9 Questions Per Category.....	52
4. ANALYSIS .....	57

4.1	Building a Matrix from Natural Language.....	58
4.2	Weighting the Vectors.....	60
4.3	Measure Similarity in an Online Learning Framework.....	61
4.4	Similarity Analysis.....	62
4.5	Compiler Output Processing.....	64
4.6	Answer Caching.....	68
4.7	Disaggregating by Assignment.....	69
4.8	Alternative Evaluations of Question Answering Systems.....	73
4.9	Synthesizing an Algorithm to Classify Questions.....	77
4.10	A Theoretical Cost and Benefit Analysis.....	79
5 .	PERIPHERAL ANALYSES.....	82
5.1	Latent Semantic Analysis.....	82
5.2	Negative Results with Source Code.....	84
5.3	Skipping Steps to Increase Accuracy.....	85
5.4	A Trained System to Classify Student Questions.....	86
6 .	CHALLENGES IN COLLECTING DATA.....	92
6.1	Dearth of Existing Data.....	92
6.2	Difficulties of Collecting Data.....	95
6.3	Requirements for a Software Tool to Collect Data.....	99
6.4	A Brief Description of the Data and its Uses.....	102
7 .	CONTRIBUTIONS, FUTURE WORK, AND CONCLUSIONS.....	105
7.1	Contributions.....	105
7.2	Future Work.....	108
7.3	Conclusions.....	117
	REFERENCES.....	118

## ACKNOWLEDGMENTS

My time during my master's degree was very influential in both my decision to pursue the PhD and the general direction of my PhD research. I am grateful to Jack Mostow who helped me learn how to write, Joseph Beck who helped me learn how to analyze data, and other Project LISTEN staff during 2003-2005 for many important research skills I learned from them. I am also grateful to several friends and classmates who stretched their social circles to include me.

My committee has been incredibly supportive. Joe Zachary, my PhD advisor, has been especially instrumental in nurturing my teaching skills. His kind concern for me and my students has been a blessing during my PhD years. I also gratefully acknowledge the many kind words he has written on my behalf in an effort to secure financial support. Deryle Lonsdale's generous recommendations have helped me get into graduate school twice, and his coursework continues to influence my teaching and research. Joseph Beck's commitment to mentoring me has endured beyond the master's degree, and discussions with him have been thought-provoking, action-stimulating, and generally useful. Hal Daume has a great sense of humor, fine quantitative capabilities, and excellent graduate level coursework. Ellen Riloff has helped me learn many things; I continue to have a deep respect for her many accomplishments and her zeal for life.

I have thoroughly enjoyed attending AIED, ITS, and SIGCSE as a graduate student, and my interactions with people at those conferences have helped me to understand what it means to be a colleague.

Peter Jensen and the entire CS1 staff at the University of Utah have been supportive of my research, even though I have crashed the server more than once. Brent Hosie's Undergraduate Research Opportunity Project enabled me to obtain inter-rater reliability measures for my dataset and gain valuable experience as a research advisor.

My parents have been very generous with me during this period of my life. Four months of living together after my master's degree has somehow turned into four years, and they have not complained. They've been generous with physical, mental, emotional, spiritual, and social support, and I'm grateful for the strong and loving family bonds we share. I thought about writing this dissertation for them, but I concluded that they would love me regardless of whether or not I wrote a dissertation for them.

I am dedicating this dissertation to my students past, present, and future. I am optimistic that their lives will be better because I have been able to have this PhD experience and learn how to be a better teacher, researcher, and person. As a mother gives birth out of love for her child, I write my dissertation out of love for my students.

## **CHAPTER 1**

### **INTRODUCTION**

Benjamin Franklin observed “The only thing more expensive than education is ignorance” [31]. Education has become even more expensive since Benjamin Franklin made this famous observation, with current college educations ranging in price from \$2,500 - \$25,000 per semester for tuition alone. The cost of an education is increasing faster than the rate of inflation, 2.9% per year after inflation [13]. An education often means a better paying job, more fulfilling work, and better social connections. In America, an education is critical to maintaining the democratic ideals of our forefathers. Research that can help control the cost of an education is a prerequisite to preserving the promise of equal opportunity for all.

The reasons for the high cost of a college education are numerous, but one of the greatest contributors is the human resources, and specifically the teaching staff. To help control costs, some colleges allow very large classes, especially for the introductory material. Class sizes of between 100 and 1000 are typical of many freshman and sophomore classes. One professor alone cannot hope to help that many students enrolled in a single course, and frequently they find themselves lecturing and relying upon teaching assistants (TAs) in order to evaluate the students, answer questions, and provide

human contact. Although cheaper than a professor, these teaching assistants still represent a major cost for the university. The School of Computing at the University of Utah spent \$320,671 for the salaries of teaching assistants in 2007-2008. In majors such as computer science where students have access to lucrative internships and easy access to financial aid, recruiting good TAs can be a significant challenge. Furthermore, students will typically work as a TA for a semester or two, so most universities are required to constantly recruit TAs, and they typically need to be retrained every semester. Although the direct, cash costs of human TAs are much lower than the cost of professors, the hidden, indirect costs are quite a bit higher including time to recruit and train them, benefits including health insurance and tuition, and others.

Teaching assistants generally perform a number of important roles in a traditional college course including lab or section leader, assignment grader, and personal tutor. Teaching assistants most commonly act as a personal tutor during consulting hours in which students are allowed to visit the TA at a predefined location at predefined hours. However, finding a time that is suitable for both the TA and the student to meet can be a challenge. Modern students have many more options when spending their time compared to previous generations, and many students in this generation want their courses and course materials available and accessible on their own time frame.

In recent years, the rise of electronic learning has provided a partial solution to this growing problem. Electronic learning includes both online courses and technology that supports learning such as e-mail and electronic bulletin boards. Compared to traditional higher education with an enrollment growth rate of 1.5%, online courses are

growing very rapidly; at 9.7%, the growth rate of online courses is more than six times the traditional course growth rate [6]. Almost 3.5 million or 20% of all U.S. higher education students were taking at least one online course in the fall of 2006 [6]. Unfortunately, online courses still have barriers, including a “lack of acceptance of online instruction by faculty” and “students need more discipline to succeed in online courses” [6]. E-mail and electronic bulletin boards are even more prevalent, but they also have problems. E-mailed answers to student questions frequently require students to wait until an instructor chooses to check e-mail and send a response (sometimes more than a day). Electronic bulletin boards can be difficult to navigate and make it difficult for professors to enforce a particular sequence in a tutorial dialog.

This dissertation describes a new approach to answering some student questions automatically that has the potential provide immediate answers, improve navigation, and enforce a particular sequence in a tutorial dialog. The ultimate goal is to create a system that can provide automated answers to common questions that are similar to other questions already in the system. This dissertation investigates the key component of such a system: the automatic classification of student questions.

### **1.1 Problem**

A comprehensive system for mediating student questions should have several features. The system should help the students recognize when they need to ask a question, classify the students' questions, provide automated interventions, and evaluate the effect of the interventions upon the student. This dissertation focuses on the central

problem in the pipeline, classifying student questions. Specifically, the dissertation focuses on classifying the questions that novice programmers ask in an introductory computer science course.

Classifying the questions that novice programmers ask in order to provide tutorial interventions is a challenging open problem. Students often articulate their questions and information needs incompletely. They also hedge, use colloquial language, misspell words, and write ungrammatically. For example, the following are information requests that novice programming students have made:

- “How do i return the file extension only?”
- “I need help extracting a file extension from a filename.”

Although phrased differently, both sentences indicate the same need, namely help with the file extension extraction problem; therefore, they should be classified the same way.

This research focuses on the problem of classifying student questions by matching them to previous questions with similar meanings but different phrasings.

## **1.2 Contributions**

This dissertation makes three contributions towards understanding the questions that novice programming students ask. First, the dissertation describes a unique software artifact called the Virtual Teaching Assistant system that mediates the question-answering process between students and staff and facilitates logging and mining the relevant data. This software system captures the natural language in the questions that students ask as well as the source code snapshots and context such as the date and



assignment that the student is working on. Second, the dissertation describes a dataset consisting of ecologically valid questions asked by students in an introductory programming class, including the dates and the assignments about which the questions were asked. This dataset enables investigations of patterns in question asking within a course. Third, the dissertation describes an analysis framework and an analysis showing that the additional context including the date and the assignment number can be leveraged to improve the classification of questions that students ask.

### **1.3 Overview**

To facilitate the study of student questions in an introductory programming course, I needed a corpus of the questions that students ask in that context. To collect ecologically valid data, I built the Virtual Teaching Assistant (VTA) system and deployed it in an introductory computer science course for a total of approximately one semester. The system mediated help requests between students and teaching assistants (TAs), capturing both the students' natural language and the corresponding code snapshots associated with a help request as well as timestamps, the assignment name, the answer, who answered it, and other relevant information.

Using this system, students typed input into a short form on a student software client, including login, machine name, Java class, Java method, and the actual question. Additionally, the students used a file chooser dialog to select the directory containing their source code. When the students clicked a button to submit their question, the system uploaded all of their code plus the accompanying information and question.

Questions and student source code submitted to the system appeared in a teaching staff's software client of the system, ordered by time of submission. A member of the teaching staff could answer the question in person or via the system with a text or URL response. The TA also indicated a category for each question. Then, the system logged the answer and the category to a database along with the question. The answer that the human TA gave was displayed in the interface of the student's software client. To facilitate research, after the data were collected, each question was manually categorized by one or more teaching assistants. This dissertation examines data collected during approximately one semester of system usage.

#### **1.4 Data**

Questions asked in Introduction to Computer Science 1 (CS1410) at the University of Utah form the dataset for this dissertation. Most students in Computer Science 1 are age 18-22. Computer Science 1 is the first required computer science course for computer science majors, with a strong emphasis on the Java programming language. The course has long hours for novice programmers and typically high dropout, fail, and withdrawal rates. The majority of students who take Computer Science 1 hope to major in computer science or a related field, but they must pass that class along with three other courses with sufficiently high grades to attain official status as a computer science major. Although approximately 233 students were active in the course during the study period, only 63 of them asked questions while using the study's logging software during the study period.

I tagged all of the data by associating all questions that could be answered with the same response to the same, unique tag. Then an undergraduate TA tagged approximately a third of the data, assigning tags from a set devised for that assignment. The TA did not recode the other two thirds of the data, but because the inter-rater reliability for the questions we sampled was high (Cohen's Kappa=0.872), I included all of the data in the final dataset. This left a dataset of 411 questions from 13 different assignments covering a total of 143 answer categories or information needs. Of the 411 questions, 268 of the questions (143 subtracted from 411) were repetitive in nature, and had a similar previous question. That means that 66% of the questions were repetitive.

### **1.5 Analysis**

Table 1.1 shows some sample student questions and the corresponding answer categories. The primary analysis utilizes an online learning framework to identify similar previous questions. Each question is compared to all previous questions, and the previous question with the highest cosine similarity score when compared to this question is considered the most similar. If the current question and the most similar question have the same answer category, the system earns a point for accuracy. For example, in Table 1.1, Q2 would only be compared to Q1, and the system would not earn a point for accuracy. However, Q5 would be compared to Q1, Q2, Q3, and Q4. Of these, Q2 would be the most similar, and since Q2 and Q5 share an answer category, the system would earn one point for accuracy.

**Table 1.1. Sample Questions, Vector Stems, and Answer Categories**

	Natural Language	Answer Category
Q1	How do i return the file extension only?	File extension extraction
Q2	my variable for rectSideOne is suppose to be 1/9, the program is returning a 0 for this calculation. I have no idea why.	Integer division
Q3	I need help extracting a file extension from a filename.	File extension extraction
Q4	program is not computing volume correctly	Integer division
Q5	Im having trouble understanding why (1/9) equals 0.0 instead of 0.111111	Integer division

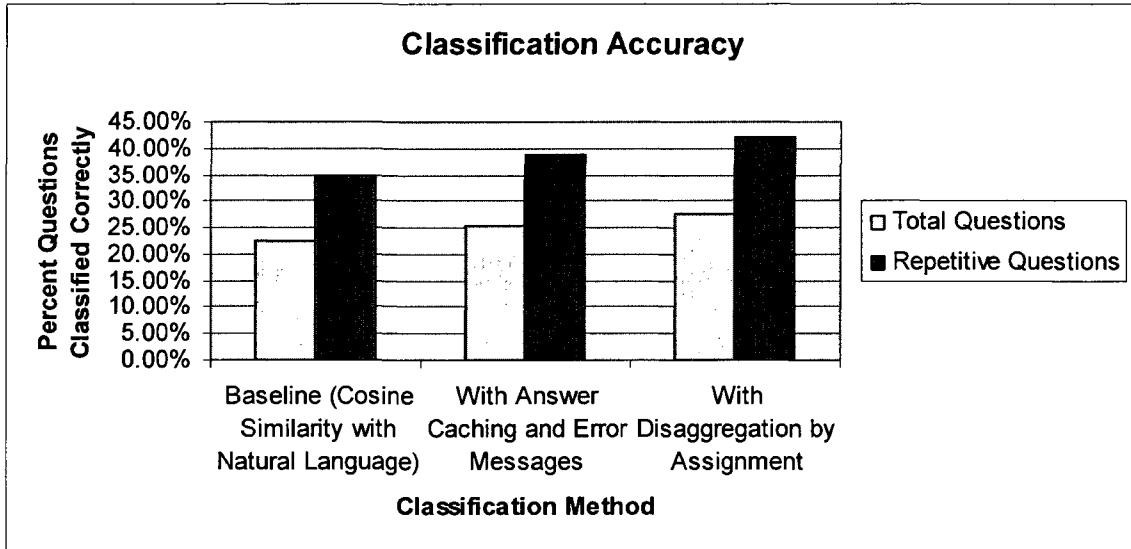
The analysis chapter also discusses new and novel methods for representing and incorporating educational context and compiler error messages in the vector space model. Specifically, I show that the compiler error messages must be processed in conjunction with source code and data from the Internet to extract underlying errors, and a technique called answer caching can be synergistically employed to improve classification accuracy. The following chapter also describes auxiliary analyses, including a comparison of the data across semesters and an investigation of a batched and supervised learning approach to classifying the questions that students ask.

## 1.6 Results

As shown in Table 1.2 and Figure 1.1, I report accuracy scores with two different denominators, total questions (411) and repetitive questions (268). Total questions includes the entire corpus of questions that the students asked using the VTA system. The repetitive questions are questions that were asked more than once. Of these, only the

**Table 1.2. Sample Questions, Vector Stems, and Answer Categories**

	Aggregated		Disaggregated	
	Total Questions	Repetitive Questions	Total Questions	Repetitive Questions
Baseline	93/411 (23%)	93/275 (35%)	113/411 (27%)	113/204 (55%)
With Error Msgs and Answer Cache	104/411 (25%)	104/275 (39%)	111/411 (27%)	111/204 (54%)



**Figure 1.1. Classification Accuracy**

repetitive questions bar could theoretically reach 100%. In all cases, the numerator is the number of correct similar questions found.

**1.6.1 Baseline**

As a baseline, cosine similarity is applied to the natural language of the students' questions. With that baseline, the algorithm can classify 96 questions or 35% of the repetitive questions and 23% the total questions. For those questions, an answer to a previous question could theoretically be recycled to answer that question.

### **1.6.2 With Error Messages and Answer Caching**

The low accuracy of question classification suggests room for substantial improvement. One possible way to improve classification is to leverage some domain specific knowledge, specifically the error messages from the compiler. Since more than 40% of the questions were submitted with code that did not compile, the compiler error messages represent a source of substantial unused data. To incorporate the error messages into the model, they were automatically processed to produce a term representing the underlying error, such as “capitalization” or “missingImport,” and those terms were incorporated into the model. To further boost accuracy, I leveraged a technique called answer caching [58] in which questions with the same answer category are merged to form a single vector. Without answer caching, the five questions in Table 1.1 are represented with five vectors. With answer caching, they are represented with two vectors, one for “File extension extraction” (the sum of the vectors for Q1 and Q3) and one for “Integer division” (the sum of the vectors for Q2, Q4, and Q5). The number of correctly classified questions (or numerator) for the “With Answer Caching and Error Messages” method is 104, and the denominators are the same as they were in the baseline conditions, 411 for total questions and 268 for repetitive questions. In this condition, the algorithm can classify 25% of the total questions and 39% of the repetitive questions.

### **1.6.3 Disaggregating by Assignment**

For a final improvement in classification accuracy, the data was disaggregated by assignment. Questions were compared only to other questions from the same assignment.

As shown in Figure 1.1, this technique improved the number of correctly classified questions (or numerator) to 113 or 28% of total questions and 42% of repetitive questions

With the data disaggregated by assignment, incorporating answer caching and error messages reduced accuracy (101 questions classified correctly). The lack of sufficient data to model different kinds of compiler errors is probably the cause of a drop in accuracy when answer caching and error messages are incorporated. Because compiler errors are being reduced to a single term, several of them are necessary to boost the compiler error terms to a heavy enough weight to influence the similarity algorithm. Excluding error messages and answer caching returns the classification algorithm to a domain independent state. Compiler error messages are a source of data that are only relevant in the computer science domain. By contrast, natural language and assignment numbers are a data source that is available in virtually every educational domain.

To facilitate comparison in the bar charts, we reuse the same denominators, 411 total questions and 268 repetitive questions. When only comparing questions from the same assignment, however, the number of repetitive questions is actually smaller (204), and that denominator gives a classification accuracy of 56% of repetitive questions.

### **1.7 Summary**

The central question of the dissertation is “Can domain knowledge and educational context improve classification of the questions that students ask in a novice programming class?” Using natural language and cosine similarity as a non-trivial baseline, I answer this question in the affirmative by improving accuracy by using

domain knowledge (processed compiler error messages) and educational context (assignment number). I replicate previous results showing that answer caching can improve accuracy by 1-3% and extend previous work on answer caching by achieving similar improvement on a more difficult dataset with ecologically valid tutorial dialogue.

Using domain knowledge, answer caching, and educational context, the algorithm can classify between 23% and 56% of the questions. Using just natural language, the algorithm can classify 96 questions or 35% of the repetitive questions and 23% the total questions. Using processed compiler error messages, the algorithm can classify 104 questions or 25% of the total questions and 39% of the repetitive questions. When disaggregating by assignment, the algorithm can classify 113 questions correctly or 28% of total questions and 56% of repetitive questions.

I also demonstrate that an online learning framework is a viable alternative for an automatic question answering system because it can classify almost as many questions as a similar algorithm in a batched setting. Many modern intelligent tutoring systems require extensive knowledge engineering and/or they must be or trained with data that has been harvested from a deployed system. The disadvantages to the former alternative are that the system designers must try to foresee every question that a student could ask, and the cost of engineering knowledge for the system is typically high. The disadvantage to the latter alternative is that the system does not benefit students as much in the year while data are being collected. The online learning framework reduces both of these disadvantages by waiting until a student has asked a question to engineer knowledge, and



then potentially exploiting that knowledge immediately after it has been added to the system.

## **1.8 Roadmap**

This chapter has proposed that a system to automatically answer the questions that students ask has the potential to reduce the rapidly rising costs of a college education, and it has given a brief overview of the research; the remainder of the dissertation gives much more technical detail. Chapter 2 discusses related research from the fields of information retrieval, computer science education, and intelligent tutoring. Chapter 3 describes the software system that was used to collect ecologically valid data and it quantitatively and qualitatively describes the data. Chapter 4 presents the main analysis showing that domain knowledge and educational context can improve the classification of student questions as well as additional analyses that compare the results to standardized question answering results and argue that mine are comparable, and better for my dataset. Chapter 5 explains why some analyses were omitted and presents additional preliminary analyses with the goal of predicting how training data obtained in the first semester might be exploited to improve accuracy in future semesters. Chapter 6 describes why the data are an important contribution, including a discussion of why this type of data is difficult to collect and a description of the dataset itself. Chapter 7 explores limitations of the presented research and opportunities for future work; it also reiterates the contributions of the dissertation and concludes.

## **CHAPTER 2**

### **PRIOR WORK**

The research described in this dissertation is broadly interdisciplinary, drawing on ideas from both commercial and academic systems for information retrieval, computer science education, and intelligent tutoring systems. This chapter will focus on laying the intellectual foundation necessary to understand how research from those fields has influenced my work. This chapter examines three threads of prior work, one on information retrieval and automated question answering, one on learning to program in Java, and one on intelligent tutoring systems with an emphasis on relevant work on tutorial dialog.

This section begins with a discussion of the more widely recognized commercial information retrieval systems, and then it discusses a number of academic systems. Some of the academic systems are less well known but are important for their contributions in the areas of information retrieval, computer science education, and intelligent tutoring.

### **2.1 Information Retrieval**

#### **2.1.1 Industrial Solutions**

Google has created the world's most successful search engine. The founders viewed the Internet as a graph, and leveraged the linking structure to design an algorithm to

rank the relative value of the information in the graph [19]. This is one of the most recognized examples of using information beyond the text in a document to improve classification. Unfortunately, even Google requires keywords, words that users must supply to unlock the information of the internet. For savvy users, providing keywords is generally not problematic, but for novices and students who are just acquiring a technical vocabulary, even a search engine as good as Google can be a challenge to use.

Before the Internet was widely used, Microsoft created extensive context-sensitive help for people using their products. Their help system includes avatars to answer questions and perform keyword searches as well as automated infrastructure that recognizes common tasks. For example, when the user begins a common task such as writing a letter, an automated avatar offers to assist [12]. Microsoft's commitment to improving software usability is admirable, but it is limited in scope to their proprietary products.

How May I Help You is a commercially successful, partially automated system for routing telephone calls and resolving help requests [33]. A semi-autonomous model is well-suited for a question answering system in which the majority of the questions fall into a few major categories. In such a system, answers to the majority of questions can easily be selected automatically, and the small number of questions for which an automated response is not practical can be handled by humans. This dissertation proposes that a semi-autonomous model would also be good for answering student questions.

### 2.1.2 Question Answering- The TREC Track

Many more information retrieval systems have been the subject of academic research. Work in the information retrieval community has generally focused on the query or perhaps a question as the articulation of a user's information needs. A typical web query is between two and three words in length (e.g.,[14]). Although a typical factoid question is longer than two or three words, it is also quite short compared to other sources of language (e.g. documents, papers, etc.) used in natural language processing. The shortness of these articulations of a user's information needs can be problematic for statistical methods that rely on reasonably large numbers to compensate for noisy data.

A major meeting for the information retrieval community is the Text REtrieval Conference (TREC). The TREC competition solicits entries from various commercial and academic entities for a variety of information retrieval tasks. Of particular interest is the Question Answering Track, which ran between 1999 and 2007. A typical system entered in the corresponding contest might consist of several major software modules organized into a pipeline. One module might process the question, perhaps matching it against a template or reducing it to keywords in order to build a query. The next module would use the query as input and retrieve a batch of relevant documents. The third module would extract answers from the documents, and the fourth module would rank the resulting answers. Each annual contest usually attracted between 10 and 30 submissions from various academic and industrial groups working on the question answering problem.

A typical Question Answering Track task consisted of several hundred questions with the vast majority being factoid questions, such as "What is the capital of Texas?" The

remaining questions might be variants of the original question, request a list of data, not have an answer in the supplied corpus (indicated by Nil in Table 2.1), or involve some other minor variation of research interest. To keep the contest competitive, the question text as well as the text that was mined for answers came from increasingly difficult corpora each year, including blogs in one of the final years of the contest. Later versions of the TREC competition utilized more difficult datasets and more difficult tasks. Consequently, the scores in later years of the competition were often lower (e.g., [27]), and comparing TREC results across years is like comparing apples and oranges. Table 2.1 lists some of the major features of the TREC competition each year between 1999 and 2007.

Typically, the TREC competition requires each group to compute a variety of statistics describing their system for a given test set of data. One of these statistics is the F-score, a weighted average between precision (how many of the retrieved documents are relevant) and recall (how many of the relevant documents are retrieved). Another statistic is accuracy, the fraction of questions for which the system can return a correct answer. Sometimes the systems are evaluated using a measure called Reciprocal Answer Rank,

**Table 2.1. TREC competition features**

1999	Factoid questions (200), long+short answers
2000	User questions (500+ 193 variants)
2001	Nil answer, List task, context task
2002	Answers instead of snippets, confidence score
2003	Factoid, definition, lists, F-score
2004	Series of questions
2005	Series of Q's, document ranking, relationship
2006	Series of Q's, ciQA
2007	Series of Q's, ciQA, blogs

which allows a system to list several possible answers and receive more credit for listing the correct answer earlier in the list. Based on the answers that a system provides, additional scores are computed to rank the various question answering systems.

Within the question answering task, the easiest part is answering the “factoid” questions, and the best systems report answering just over half of these correctly (MRR=0.58). The other systems report accuracy substantially lower (MRR <0.5), and overall accuracy of all question types is also substantially lower [81]. . These results suggest that providing automated answers to questions is a difficult task for computers, even when the question is well-articulated by an expert and the corpus contains cohesive, coherent text with an appropriate answer.

Of all the TREC contests in the question answering track, the contest in TREC 9 (2000) is particularly related to my research. In that contest, the “question(s) were taken from a log of questions submitted to the Encarta system made available by Microsoft plus questions derived from an Excite query log” as well as “questions that were created by the assessors to be semantically identical but syntactically different from a question already in the test set” [81]. An examination of the test set suggests that 193 of the 693 questions (or more than 25%) fall into this category of questions that are semantically identical but syntactically different from a previous question in the test set.

Of the papers in the corresponding SIGIR conference (2003), one specifically mentions exploiting a technique called “answer caching” to provide answers to some of these questions. Answer caching is a technique that matches an incoming question to a semantically similar previous question or question category in order to recycle an answer

[58]. In the TREC contest referenced in the SIGIR paper, the questions were syntactically different, but lexically very similar. The SIGIR paper reports that the technique accounts for a modest 1-3% improvement which seems small considering that this technique alone could theoretically answer approximately a quarter of the questions. One weakness in the approach outlined in that paper is the measurement of similarity which relied on fairly strict lexical similarities, and the approach did not utilize term weightings in calculating similarity.

Unfortunately, the TREC competition does not have a track for computing the answer to student questions. Student questions are more challenging than typical factoid questions for several reasons. First, student questions are often incomplete and poorly articulated. For example, a student might simply say “I’m stuck.” Second, students often use colloquial language when requesting help. For example, they frequently use polite words such as please and thank you as well as slang words in their requests; typos, spelling errors, and grammar errors also indicate the colloquial nature of their typed text. Third, students frequently ask free form questions that are difficult to classify with a traditional grammar or information extraction frame. Fourth, student questions are often deeply rooted in a complex educational context that includes implied information about the course, text, instructor, assignments and various other details.

### **2.1.3 Passage Retrieval**

One of the major limitations of the research on answering factoid questions is a focus on finding an answer, usually a few words or perhaps a sentence long. However,

many users generate queries suggesting a need for longer answers. One alternative to satisfy these users is called passage retrieval. One approach to passage retrieval involves evaluating the similarity of the document to the query, and then finding the most relevant passages in sufficiently similar documents [69]. The ideal length of a passage appears to depend on the corpus from which it is extracted; however, 200-250 words seems to be a reasonable passage length for most topics [25]. These papers found that retrieval was more effective when documents were selected first, and then passages were selected from relevant documents. Another approach found relevant passages and then used the passages to identify relevant documents, and argued that for at least some corpora that approach was more effective [47].

#### **2.1.4 Frequently Asked Question (FAQ) Question Answering Systems**

A weakness of traditional statistical retrieval is the inability to leverage structural information. FAQ finding attempts to leverage the structural information inherent in a corpus of FAQ documents consisting of questions and answers. An early system achieved shallow language understanding with keyword matching and multiple lexicons[70]. Another system, the FAQ finder system, utilized shallow parsing and marker passing [38]. To improve upon previous work, one group combined the statistical information in the vectors with semantic information gleaned from WordNet [72]. More recent work has attempted to summarize FAQ's based on a user query [15]. Query logs have also been used to cluster similar questions and smooth models of question types, thus reducing the need to engineer knowledge [48].



## 2.2 Classifying Questions from Novice Programmers

Several classification schemes exist for classifying student questions and source code in introductory computer science, but many of these are too broad to be useful for classifying questions and giving automated answers. Garner, Haden, and Robins suggest a number of broad topical categories (e.g., loops or arrays) and present an analysis showing that the number of questions per topic varies week by week, with the majority of questions often clustered around one or two topics each week. They note that they were “not aware of any predefined criteria for validating any particular taxonomy of problem descriptions” so they provide a fairly detailed list of 27 categories that they used in the appendix of their paper [32, 64]. Kim, Shaw, Chen, and Herbert have demonstrated that approximately 80% of the speech acts on a class discussion board for an operating systems class are questions or answers, and the vast majority of those questions are about assignments and exams for the class; they suggest that the chapters of a textbook could be the foundation for an ontology for a computer science course [49]. Baffes and Mooney suggest two broad categories of students’ problems: incomplete student work and incorrect student work [10]. Spohrer, Soloway, and Pope list four major categories of novice programming errors including missing code, malformed code, spurious code, and misplaced code [74].

Unfortunately, all of these various classification schemes are too course grained to be useful for answering student questions. Most novice programming students recognize when they ask a question that they are missing code and/or they have malformed code, and many students recognize that they need help with a particular topic. Thus, correctly assigning a question to such a category does not increase a student’s knowledge, nor does it

close the gap in knowledge that triggered the question. One conspicuously absent classification scheme separates questions into categories based on whether or not the accompanying code compiles. Extensive prior work has examined both of these categories independently, and the remainder of this section gives substantial technical detail on prior work classifying Java compiler errors as well as a briefer overview of systems that aim to help novice Java programmers when their program does compile, e.g., with system design and run-time errors.

### **2.2.1 Classifying Compiler Errors**

Several recent projects have analyzed compiler errors made by students learning to program in Java. Typically, these papers involve a quantitative analysis of hundreds or thousands of compiler errors made by novice Java programmers during a study period as short as a month or two or as long as a semester.

#### *2.2.1.1 Gauntlet*

The Gauntlet project was launched by instructors at West Point. An early version of the Gauntlet system ran as a web application, and a later version was integrated with the local IDE. Both versions logged all the compiler errors for a version of student source code, and both provided additional feedback to help students fix their problems. Because every freshman at West Point is required to take an introductory programming class, the project was able to collect substantial amounts of data. The system collected a total of 559,419 errors in just one semester. The dataset is somewhat unusual because the majority of the

students were not planning to major in Computer Science. Collectively, the 10 most common compiler errors account for 51.8% of the errors collected [30, 41]; they are listed below:

- Cannot resolve symbol
- ; expected
- Illegal start of expression
- Class or interface expected
- <identifier> expected
- ) expected
- Incompatible types
- Int
- Not a statements
- } expected

#### 2.2.1.2 *BlueJ*

Matthew Jadud studied novice compilation behavior in the context of BlueJ, a Java interactive development environment (IDE) for novice programmers. BlueJ only displays and logs the first error that the compiler finds. In an introductory programming class of 206 students, 63 agreed to participate in the study of logged compiler errors during the weekly lab sections. A total of 1926 errors was logged belonging to 42 different types. Of these errors, more than half belonged to the five most common errors including missing semicolons (18%), unknown symbol variable (12%), bracket expected (12%), illegal start of expression (9%), and unknown symbol: class (7%). The next five most common error categories include unknown method (7%), incompatible types (4%), class-or-interface-expected (4%), identifier expected (4%), class expected (3%). Collectively, these ten error types represent approximately 80% of the errors in this set [42, 43].

### 2.2.1.3 GILD

Suzanne Thompson studied compilation errors from CS2 students using Eclipse and Java with the GILD plugin. Out of 115 students in the class, 10 agreed to participate in the study, and a total of 3535 error messages belonging to 88 compiler message types were identified. Of these errors, more than half belonged to the top five error categories. The top 10 categories are listed below:

- UndefinedName (20.2%)
- TypeMismatch (8.8%)
- UndefinedMethod (8.5%)
- ParsingErrorInsertToComplete (8.4% )
- ShouldReturnValue (4.9%)
- UndefinedType (4.8%)
- ParsingErrorDeleteToken (4.1%)
- PackageIsNotExpectedPackage ( 3.2%)
- UndefinedConstructor (2.9%)
- ParameterMismatch (2.6%)

Collectively, these 10 errors represent approximately two thirds of the errors in this set [76].

### 2.2.1.4 Jikes

Similar research carried out by some researchers using JCreator and Jikes (an open source compiler project that appears to produce error messages that are a little bit different from the javac compiler) included all of the assignments over the course of a module delivered to 192 students. In total 108,652 errors were collected. The authors report 226 distinct semantic messages with 6 errors (conditional, loop, method, array, class, string) constituting more than 50% of the errors in each concept [2].

### 2.2.1.5 Summary

In summary, several different groups have independently studied novice compiler errors and concluded that majority of the bugs come from a few major categories. However, as even the authors acknowledge, it may be somewhat misleading to suggest that a single error message category has a single cause or that the cause of the error message is adequately conveyed by the error message. For example, a Gauntlet paper notes:

“[The] same message (cannot find symbol) appears whenever one of the following three errors occur:

- The variable is declared properly, but is never initialized, and then is used.
- The variable is misspelled when it is used.
- The variable is declared, but its initialization occurs inside of a conditional block and is used outside of that conditional block” [30].

Additionally, a BlueJ paper spends approximately one fourth of the paper describing how one student spent approximately one fourth of an hour unsuccessfully attempting to resolve a missing semicolon error [43].

## 2.2.2 Run Time Errors

### 2.2.2.1 Propel

Other prior work has examined novice programming errors that occur when code compiles, for example those errors that occur before any code is written or alternatively when a program compiles and runs. ProPL is a natural language tutorial dialogue to help students design and plan their program [53]. ProPL demonstrated that natural language technology is effective in helping students learn how to compose Java programs, but did

not attempt to answer student questions, and it was limited in scope to one or two programming problems.

#### *2.2.2.2 FaultFinder*

At the other end of the programming pipeline, the FaultFinder system utilized a combination of scripts and machine learning techniques to perform static analysis on simple programs and identify run-time errors. The system was able to use a combination of machine learning, program slicing, and program comparison to automatically identify many errors that novices made. Unfortunately, the FaultFinder system depends on compiled student code, and it only identifies bugs [45].

#### *2.2.2.3 FindBugs*

Another system, FindBugs focused on more advanced programming issues such as threading and null pointer dereferencing. The FindBugs system contained a set of patterns that represented programming language idioms that were frequently associated with bugs. The system was able to automatically identify many bugs, but most of the bugs that it identified were beyond the scope of a novice programming class [40].

#### *2.2.2.4 Summary*

Like many of the preceding projects, the ultimate goal of this research is to provide quality automated help to novice programmers. Unfortunately, this dissertation will not actually bridge the gap of providing quality automated answers, but it does extend the prior

work on question classification by providing a deeper analysis of the source of the bug instead of looking only at compiler or program output.

## **2.3 Tutorial Dialog**

This section considers two general periods of progress in the general area of intelligent tutoring with extra emphasis on contributions related to tutorial dialog. The first period highlights historical contributions of early intelligent tutoring systems beginning with Jaime Carbonell's SCHOLAR and continuing through the seventies and eighties. The second period considers work done between roughly 1990 and the present, with an emphasis on work done more recently.

### **2.3.1 Early Intelligent Tutoring Work**

Intelligent tutoring first became an independent research area in the 1970s and 1980s. Early systems from that time period were extremely limited by memory, constraints, a lack of networking technology, and limited user interface choices (frequently text-only). Consequently, many of the contributions of these early systems are more theoretical in nature. They are worthy of review because many of the basic techniques introduced by these systems are still in use today, and the theoretical justification for this general area of research remains valid and popular in modern research.

### *2.3.1.1 SCHOLAR*

Summaries of early progress in intelligent tutorial dialog typically begin with a discussion of Jaime Carbonell's dissertation on SCHOLAR [53, 83]. SCHOLAR was a mixed-initiative tutorial dialogue system in which a student could pose questions to the system or the system could ask questions of the student [26]. The dialogue was based upon a semantic network, and the system could generate questions and evaluate answers by traversing the network. Wenger provides an overview of the system and provides some commentary; his writings include example dialog for SCHOLAR that occurs in the general domain of geography [83]. From a modern perspective, one of the major shortcomings of SCHOLAR is that it was not used by students.

### *2.3.1.2 WHY*

Another noteworthy early system called WHY conducted tutorial dialog in the domain of rainfall process. The WHY system incorporated a Socratic strategy for advancing the dialogue [75], and it inspired future work by other researchers on a system called WHY-2 [36]. The scripts of the WHY system were inadequate to pursue global goals such as pervasive misconceptions or extended explanations of complex concepts.

### *2.3.1.3 SOPHIE*

The SOPHIE projects (I-III) examined tutorial dialogue in conjunction with a tutoring system for designing electrical circuits [21]. The designers outlined the following four criteria for evaluating instructional interfaces [24]:



- Efficiency (the student should not have to wait)
- Habitability (the system should accommodate various phrasings)
- Robustness with ambiguities (the system should not expect independent systems to be unambiguous)
- Self-tutoring (the system should gracefully handle unacceptable inputs while teaching the student how to properly interact with itself)

The SOPHIE projects were also the first to consider tutorial dialog as a component of a larger intelligent tutoring process with other components, and they were the first to utilize information external to the dialog (e.g., from a game) in making dialog decisions. Again, many of the limitations of the system can be linked to representation. Specifically, some of the conceptual knowledge was difficult to represent with existing data structures and the system frequently fumbled when more than one error occurred in the circuit.

#### *2.3.1.4 BUGGY*

BUGGY, DEBUGGY, and IDEBUGGY are systems that helped students learn subtraction by exploiting a model of observable bugs. From a computational standpoint, they represent a major shift in tutoring systems towards representing learning and tutorial actions in a procedural network. BUGGY utilized an extensive procedural network to model subtraction with subprocedures [20]. DEBUGGY used the model introduced by BUGGY to diagnose student problems, and IDEBUGGY diagnosed student problems interactively [23]. Presumably data from these systems contributed to the REPAIR [22] and STEP [78] theories that introduce a generative model of bugs in terms of an underlying cognitive model. Representing tutoring and learning in a procedural network was an important advancement that eventually led to model tracing tutors and step tutoring.

### *2.3.1.5 MENO AND PROUST*

Programming was also a popular domain for intelligent tutoring during this time period. At the University of Massachusetts Amherst, the MENO project was an attempt to build an intelligent tutoring system for Pascal. MENO compared student program parse trees to template program parse trees. Differences in the parse trees were mapped to misconceptions in a database, allowing MENO to automatically retrieve misconceptions if the parse tree appeared to be incorrect [83]. Unfortunately, the complexity and variability of the student programs exceeded the expectations of the system. MENO-II reduced the size of the problem space by specializing in analyzing loops and variables [71], and MENO-TUTOR focused on remedial dialogue [89]. PROUST was another spin-off project that viewed program design as a hierarchical process consisting of agendas, goals, and code. PROUST ran on syntactically correct code and for one programming problem, it could comprehend 81% of questions, and detect 78% of the 795 total bugs. However, on a more complex problem, the system could only detect 64% of the bugs, and in both cases, a fair number of false alarms were generated [46].

### *2.3.1.6 LISP Tutor and ACT*

Meanwhile, at Carnegie Mellon University, a model tracing tutor called the LISP tutor help students learn the Lisp programming language while also serving as a test bed for John Anderson's ACT theory. John Anderson's involvement helped to ensure that early versions of the LISP tutor had a solid cognitive science foundation with each tutorial and learning action linked to specific fundamental behaviors in human learning [7, 8]. Many of

the lessons learned from the process of creating the LISP tutor eventually affected the creation of the cognitive tutors, a set of intelligent tutoring systems heavily influenced by cognitive science. Early versions of the cognitive tutors also linked learning and tutoring actions to behaviors of the human brain, but more recent versions have focused on learning gains, instructional science, and human-computer interaction and the focus has shifted away from cognitive science.

#### *2.3.1.7 PEA*

At Edinburgh, an independent project called the PEA (Program Enhancement Advisor) provided intelligent tutoring on Lisp programming style. The PEA project is relevant to this research because it focused specifically on programming style, an area that seems to be the topic of many novice programming questions [57]. Programming style is an interesting area for intelligent tutoring research because the domain consists of both objective elements and subjective elements, and subjective elements present potential problems for adoption by instructors with different philosophies than the tutoring system designers. The PEA project appears to not have progressed to the point where this became a problem. Probably the primary reason that the various Lisp tutors described in this section are not used today is that the Lisp programming language lost its luster.

#### *2.3.1.8 Summary*

Early intelligent tutoring research can be characterized as largely consisting of proof-of-concept systems that primarily demonstrated the capability of the computer in that

time period. Although some of the systems were used by humans (e.g., SOPHIE was used for a semester), they were rarely used for actual instruction. In 1984, Benjamin Bloom published a paper arguing that a two sigma difference separated one-to-one (human) tutoring and classroom instruction [16]. He further stated that one-to-one human tutoring was expensive and claimed that the challenge was to make it cost-effective. Believing that challenge was one that computers could solve and recognizing a need for more empirical studies, the intelligent tutoring research community began a new, more empirical era of research.

### **2.3.2 Intelligent Tutoring Modern Work**

Modern intelligent tutoring research has seen the rise of several different tutoring systems that are currently deployed in classroom settings, and many of these systems have made at least partial progress on the two sigma challenge. The Cognitive Tutor Algebra Tutor has achieved a one sigma improvement over traditional classroom instruction in the domain of Algebra [51]. The Andes Tutor has achieved similar gains in the Physics domain, and they argue that they are the gold standard for modern intelligent tutoring systems [80]. Those gains are based on customized assessments. Using standardized assessments, the systems report smaller gains of 0.3 (Algebra) and 0.25 (Andes) [80]. The Project LISTEN group has achieved similar statistically significant effect sizes for their reading tutor that listens to children read aloud [55]. These systems have generally bypassed the problem of poorly articulated student language by focusing on “step tutoring” as opposed to “natural [language] tutoring” [77]. The general approach to step tutoring

consists of constraining the size of the information space that the system covers (e.g. a single algebra problem [51], a single physics problem [80], or a sentence to be read aloud [55]), dividing each problem into a set of skills or steps (usually modeled in a procedural network), and creating interventions or help messages for each skill, ending with a “bottom out hint” or answer for each step.

In modern intelligent tutoring research, two reasons are commonly cited for pursuing natural language tutoring. One popular reason is that natural language may help students to learn deeply and avoid shallow learning [36]. Others have suggested that natural language tutoring might close the gap between existing tutoring systems that report a one sigma improvement over classroom instruction and expert human tutors that are capable of a two sigma improvement [60]. Tutorial dialog has played both a peripheral role, as a plug-in to an existing tutoring system, and a central role, as the major focus of the tutoring system.

#### *2.3.2.1 Geometry Explanation Tutor*

Some systems have focused specifically on helping students to articulate their reasoning, as a plug-in module. For example, the Geometry Explanation Tutor is a plug-in to the Geometry Tutor that provides feedback on natural language justifications for steps in a geometry proof [5]. The Geometry Explanation Tutor was able to respond to and process student language well enough to provide some tutorial interventions[5], but it did not produce learning gains [4].

### 2.3.2.2 *CycleTalk*

Another example of a plug-in module is CycleTalk, a system that facilitated Wizard of Oz studies in the context of the CyclePad system that tutored students in the domain of thermodynamics [67]. The Wizard-of-Oz studies that the CycleTalk system facilitated allowed experimenters to conduct tutorial dialog with students. The collected dialogs reveal common characteristics of student dialog such as abbreviations, misspellings, and colloquial language. However, the presence of a human Wizard meant that the system never really had to classify student language or utterances or provide solutions to other technical challenges in dealing with student dialog.

### 2.3.2.3 *Atlas*

Atlas is a generic dialog plug-in for model tracing tutors. Model tracing tutors are intelligent tutoring systems that trace a graph of student skills, with edges representing the students' input and nodes representing the student's knowledge state. Atlas has two major components that can be extended with domain-specific knowledge [66]. The first component, the Atlas Planning Environment (APE) takes care of dialog management issues and plans tutorial strategies while monitoring the student's progress with model tracing techniques. The second component CARMEL takes care of the natural language understanding with a system that prefers well-formed, well-spelled dialog, but allows relaxations to accommodate the more colloquial language that students tend to use. A separate tool facilitates authoring Knowledge Construction Dialogues (KCD's), the domain specific pieces of knowledge that Atlas uses in constructing its dialogs. The Atlas system

was originally implemented as a plug-in for the Andes Physics tutoring system, and the system designers were able to show a 0.9 sigma improvement in learning gains over Andes without natural language tutoring [36]. Atlas later produced a couple of spin-off projects. One spin-off, WHY2, demonstrated the domain-independence of the system by deploying another plug-in in a different tutoring system (AutoTutor) in a different domain (computer literacy), and it explored opportunities to leverage the best aspects of both statistical language processing and symbolic language processing.

#### *2.3.2.4 AutoTutor*

The AutoTutor project, in contrast, considers dialogue to be an essential part of the tutoring process, and it has achieved a one half sigma improvement for their system that teaches computer literacy [36]. Generally, the AutoTutor project relies on statistical language processing techniques, treating student dialog as a bag of words and throwing away the syntactic information contained in the dialog. The AutoTutor project has researched a number of different analytical approaches for processing student language in response to tutorial prompts and published relevant papers. The first publication simply demonstrated that an information retrieval technique called Latent Semantic Analysis (LSA) [52] with natural language was a viable approach to selecting text for intelligent tutoring dialog with human raters as the gold standard [86]. Later, the same research group tuned the parameters for LSA and found slightly stronger performance by altering features such as the number of dimensions in the matrix and the amount of training data [87]. In another variation, the research group compared increasingly simple models, including

cosine similarity and concluding with a keyword model. The keyword model counts the number of words found in both a current question  $c$  and previous question  $p$  and divides by the maximum number of terms in either question. The much simpler keyword model is within 20% of the performance of the full LSA model for their dataset [85]. This line of research is also described in greater depth in a journal article [37].

#### *2.3.2.5 PedaBot*

The PedaBot project led to a similar line of research with a few fundamental differences. First, the PedaBot project aims to match student discussions to similar previous student discussion [50]. Because students are notoriously bad at articulating their discussion points, matching student input to student input is a more difficult problem than matching student input to expert-provided input. Second, although the PedaBot approach does not require expert-provided answers, it does require a list of expert-provided technical terms. The PedaBot project avoids generating these manually by automatically extracting them from a textbook or other authoritative, expert provided resource [50]. Like the AutoTutor group, the PedaBot group has examined various techniques for calculating similarity of the various discussions in the system, with the focus on LSA and cosine similarity [50]. Together, these groups have demonstrated convincingly that LSA and cosine similarity are a promising direction for processing tutorial dialogue.

The general approach still has a number of serious weaknesses. First, the research results are not as compelling as they could be. The AutoTutor group reports correlations with  $r < 0.5$  [85], and the PedaBot group reports finding discussions of “moderate



relevance” or discussions that rank 3 on a 4 point Likert scale [50]. Second, the approaches outlined require significant expert-authored resources, either in the form of a list of ideal answers in the case of AutoTutor or in the form of a list of technical terms for PedaBot, and matching these technical terms is critical to both approaches. However, students (especially novice programming students), often do not use technical vocabulary in articulating questions. Third, the approaches seem to rely on students being quite verbose in their interactions with the system. Literature in the information retrieval community has shown that longer queries are often more effective and robust [14], and LSA is known to be most effective with between 300 and 500 terms in the final matrix (after the size is reduced by the principle component analysis) [18, 85]. However, students (especially novice programming students) are not verbose when asking questions.

## **CHAPTER 3**

### **DATA**

#### **3.1 Historical Background**

When the original research project was designed, Joe Zachary taught Computer Science 1 at the University of Utah. In his course, he utilized a Java applet that he had written called the TA Call Queue where students entered their name and the location of their lab machine. This information then appeared in the TA interface, and a human TA was required to walk over to the student's machine in order to help them. The TA Call Queue did not allow students to type natural language describing their help request, help students obtain remote assistance, or log data to the server. To facilitate collecting data for my research and provide functionality not available with the TA Call Queue, I designed a new piece of software called the Virtual Teaching Assistant (VTA) system that logged student questions and allowed for remote feedback. With both systems, the human TAs periodically circulated through the lab and prompted the students to ask questions because some students are reluctant to ask questions without prompting.

#### **3.2 System Architecture Overview**

The VTA system consists of four major software components plus a database. All four software components are implemented in Java. Two of the software components are

client interfaces, one for students and another for TAs. Another major component performs various analytical procedures on the data as described in detail in the next chapter. The final major component is a set of Java servlets that run on a web server as middleware, processing data from the clients and storing it in a database.

The Virtual Teaching Assistant student software works as follows. The student decides to ask a question and launches the Virtual TA software by clicking on a button on the class webpage. The application already knows the student login, location, and current assignment number. However, the student can override that information. For example, students may choose indicate that they are using a personal laptop in the college computer lab or use their first name instead of their login name. The student can fill in the name of the Java method and class that they believe is relevant to their question, and any natural language they need to express their question. Additionally, the student must attach their source code folder using a standard open file dialog prior to submitting their question. When the student clicks the send button, the client connects to a server, logs the information it has collected from the student to a database, and passes the question to a human TA on duty via the TA client interface. Figure 3.1 shows the student interface.

The human TA sees the question, source code, and other educational context in a TA interface. In the TA interface, the upper left hand area displays the list of students who have questions on the queue; the lower left hand area lists the source code files associated with the selected student. The center area displays the student source code. The third panel to the right displays the student form and provides an area for the TA to type a response and/or an answer category. An answer category corresponds to a group

VTA-Student Interface	
CADE login	cecily
Location	lab1-7
Assignment	assignment4
Java Class	StringPractice
Java Method	fileExtension
<b>Browse</b>	te.CECILYMy Documents\ATestSuite\AVTABlueJ
How do I extract the extension of the filename?	
<b>Send</b>	
<b>Remove me from the queue; I answered my own question.</b>	
<b>Show me my history of questions and answers.</b>	

Figure 3.1. Student Interface

of questions that could be answered with the same answer. To facilitate the assignment of questions to answer categories, the TA interface has a button panel in the right-most pane with one button for each answer category for the current assignment; the TA can also add a new answer category if the current question does not fit into an existing category. Initially, the button panel for an assignment was empty, since no prior research existed to suggest what kinds of questions students might ask. However, as the students asked questions about an assignment, the number of labeled buttons for that assignment grew. The human TA can answer the question in person by walking to the student's machine in the computer lab, or the human TA can type a text response to the student that will appear in the student's client software in the lower text area. To remove the question from the queue, the TA must assign the question to an answer category. The answer category chosen by the human TA is logged to a database along with any text that the human TA has provided for the student. The TA interface is shown in Figure 3.2.

A set of middleware Java servlets connects the Student Interface and the TA Interface. These servlets transport data between the two interfaces, and they log data to the database. Currently, they are separate from the analysis software described in the next chapter, but they are designed to facilitate easy integration in the future when it will be possible to answer some questions automatically with the VTA system. The database stores the questions, the answers, and their educational context. Educational context includes information such the student's name/login, location, course, directory containing the student's source code, time the question was asked, time the question was answered, responding TA, and any other details relating to the question and the answer.

**Figure 3.2. TA Interface**

**VTA-TA Interface**

**Tasks**

Mon\_Mar\_16\_17\_19\_59\_MDT\_2009\_cecily

Mon\_Mar\_16\_17\_34\_24\_MDT\_2009\_John

Test.java

```

public class Test
{
    // instance variables - replace the example below with your
    private int x;

    /**
     * Constructor for objects of class Test
     */
    public Test()
    {
        // initialise instance variables
        ix = 0;
    }

    /**
     * An example of a method - replace this comment with you
     *
     * @param y a sample parameter for a method
     * @return the sum of x and y
     */
    public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}

```

**CADE login**

**Location**

**Assignment**

**StringPractice**

**fileExtension**

How do I extract the extension of the filename?

**Send**

**Answer Category:**

**Launch**

Moving the Pyramid

Sizing pyramids

Separating the pyramids

duplicate

Font

font object use

handin

Imports

Installing Bluej

Missing Java Files

Centering in a window

Chess coordinate system

Draw Chess Pieces

Drawing the Pyramid

New line escape sequence

Polygon versus DrawLine

### 3.3 Participants

The students who used the Virtual Teaching Assistant system were enrolled in Introduction to Computer Science 1 (CS1410) at the University of Utah. Approximately 150-160 students enroll in Computer Science 1 each fall, and approximately 80-90 students enroll in Computer Science 1 each spring. Most students in Computer Science 1 at the University of Utah are age 18-22, but there are also a few nontraditional students, such as students from local high schools or adults who have returned to school later in life. Computer Science 1 is the first required computer science course for computer science majors, and it is a gatekeeper course with a strong emphasis on the Java programming language as well as the traditionally long hours for novice programmers and the typically high dropout, failure, and withdrawal rates. The majority of students who take Computer Science 1 hope to major in computer science or a related field, but they must pass that class along with three others with sufficiently high grades to attain official status as a computer science major.

While the software was being designed, one of Joe Zachary's PhD students, Peter Jensen began teaching Computer Science 1. The course evolved substantially during his early years of teaching it. Peter's changes covered many aspects of the course from assignments, to the textbook, to the choice of integrated development environment (IDE). In this evolving educational environment, the data collection portion of the VTA software system was deployed three times. The first deployment occurred in Fall Semester of 2007 while the students were working on the ninth assignment. During this deployment, a small dataset of 71 questions was collected, but the data in that dataset was excluded



because the dataset was small, and the system was behaving unreliably. Unfortunately, resource leakage problems internal to the system (e.g., not closing files and buffers) caused the system to crash the server once or twice, and the system was taken offline for code reviews to improve stability. The data from that deployment are generally excluded unless specifically noted otherwise in the remainder of the dissertation. The second deployment occurred during Spring 2008 for assignments 3-12, and the majority of the data described in this dissertation come from that deployment. That deployment was stable and resulted in a dataset of approximately 300 questions included in the analysis. The third and final deployment occurred during Fall 2008 for the first four assignments and resulted in a dataset of approximately 100 questions included in the analysis; most of the data in that deployment comes from the third and fourth assignment. Assignments from Fall 2008 are indicated with an extra leading 0 between the assignment and the number, as in assignment03 and assignment04.

In the middle of the Fall 2008 semester, the instructor curtailed usage of the system for several reasons. First, he believed that students were not asking as many questions as they did with the TA Call Queue. Second, he believed that students did not like using the VTA system, although anecdotal evidence suggests that at least some students preferred the VTA system, especially because it allowed them to ask questions remotely. Third, he changed the primary IDE that students used in the course from BlueJ to Eclipse, and this switch mitigated some of the primary advantages of the VTA system because it made it much harder for TAs to remotely inspect students' source code.

### 3.4 Data Cleansing

When the data logging software was designed, every effort was made to facilitate rapid data analysis. For example, the software automatically prompted the TAs to enter the assignment name and number. Then the software automatically filled in the student client interfaces with assignment name to increase consistency. Similarly, the student clients automatically filled in the login and the name of the computer that the students were using. However, even though the data set was carefully designed, some cleansing was necessary. For example, some students used several computers away from the lab, and their logins needed to be recoded for consistency. In a few cases, the only way to obtain their login was to look in the comments in the source code.

Some students asked the same question twice because of a glitch in the software that caused a delay between question submission and system acknowledgement; the duplicate questions were removed from the dataset, but the original questions were left in the dataset. Most of the dataset deletions can be attributed to duplicate questions. If a human could not classify the question using the student's natural language or the status of the source code, then the question was excluded because the student probably relied on spoken dialog with a TA that the system was unable to capture. The instructors and developer occasionally tested the system, and questions generated for that purpose were also removed. Additionally, an occasional question related specifically to a laboratory assignment and not a weekly programming assignment, and those questions were also excluded. The weekly laboratory assignments were much shorter than the programming

assignments, and the vast majority of questions about the laboratory assignments were answered through spoken dialog not captured by the VTA system

While students were waiting for the TA, they might figure out the answer to their own question and remove themselves from the queue. Additionally, several students appeared to attempt to engage the TA in dialogue using the system for various reasons including expressions of gratitude (e.g., ""Thanks! That fixed it" or "ok I got it, thanks"), explaining location (e.g., "That would be great. Lab 4 - 20. Sorry is that 4th floor of the WEB?" "Sorry Im directionally challenged. On the server side or on the side close to the wall?"), requests for in person help (e.g., "Please come" or "anyone here?"), and complaints about the system (e.g., "i keep getting disconnected and it says i am doing it but i am not. Hopefully you still have me question."). Unless such dialog also included a new, novel question, it was also excluded from the dataset.

### **3.5 Interrater Reliability**

When is a question similar to a previous question? Similarity is an inherently continuous concept, with some questions being more similar and others being less similar. Unfortunately, to evaluate the accuracy of the algorithm, it is necessary to convert the continuous concept to a binary measurement. The original plan was to simply use the answer categories as recorded by the human TAs. Unfortunately, tagging the data appears to be a task that requires training (so that the TAs know what to aggregate and disaggregate), and it may be a task that is difficult to perform in real time while answering questions. Furthermore, not all of the answer categories that the TAs

chose are particularly descriptive of the student's question. As the prior work section emphasized, there are many possible classification schemes for compiler errors and student questions and none is a widely accepted standard. The human TAs participating in the study expressed concern that they didn't have time to tag data when several students were waiting for help, so they were told that they could tag questions with a tag such as "Answered in person" under those circumstances. Approximately a quarter of the data was tagged "Answered in person" by the human TAs. Finally, the human TAs could not easily recycle categories from previous assignments, so much of the data are tagged with redundant categories across assignments, such as "Testing," "Testing for integer," and "Tests". By the TAs original tags, excluding "self-resolution" and "Answered in person" categories, less than a third of the questions were repetitive in nature. This seemed unlikely and low, and the "Answered in person" category was problematic.

Consequently, after the data were collected, I coded all of the data. Each question that could conceivably be answered with the same response was assigned to a category for that response. Then an undergraduate TA coded approximately a third of the data, assigning tags from a set devised by me for that assignment. Because the interrater reliability was high (Cohen's Kappa=0.872), the other two thirds of the data were not coded by a second TA, but they were included in the dataset.

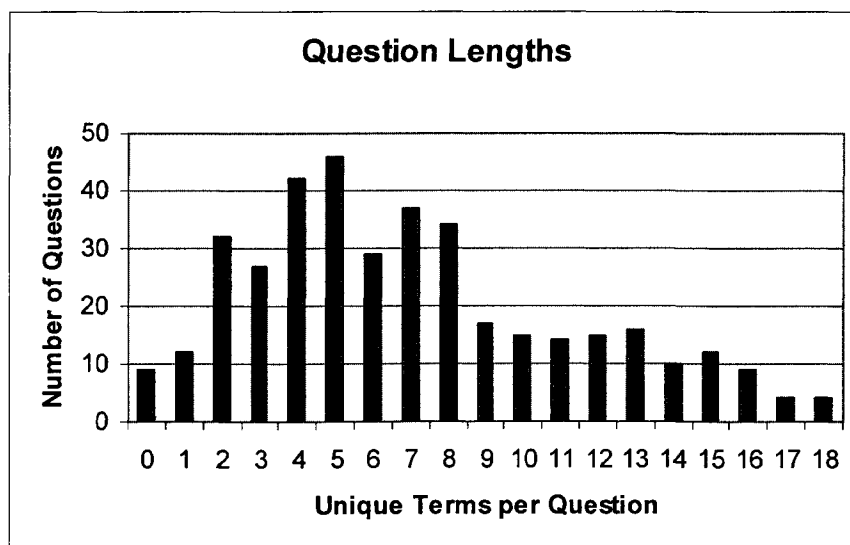
### **3.6 Dataset Statistics**

This left a dataset of 411 questions from 13 different assignments covering a total of 143 answer categories or information needs. Of the 411 questions, 268 of the

questions (143 subtracted from 411) were repetitive in nature, and had a similar previous question. That means that 65% of the questions were repetitive.

Excluding stop words, length of student questions ranged from 0 to 93 words, with a median of 7 words and a mode of 5 words. More than 2% of the questions had no words after stop words were excluded. More than 90% of the questions had 16 words or fewer. Of the 30 questions with more than 16 words, 12 contained source code mixed with natural language and 3 contained test result sets. Figure 3.3 shows the number of questions asked as a function of the number of unique words in the question.

The vast majority of the questions (75%) were asked on lab machines with the other 25% being asked on personal machines, from either home or the lab. Unfortunately, the logging software did not record a distinction. Although the logging software allowed



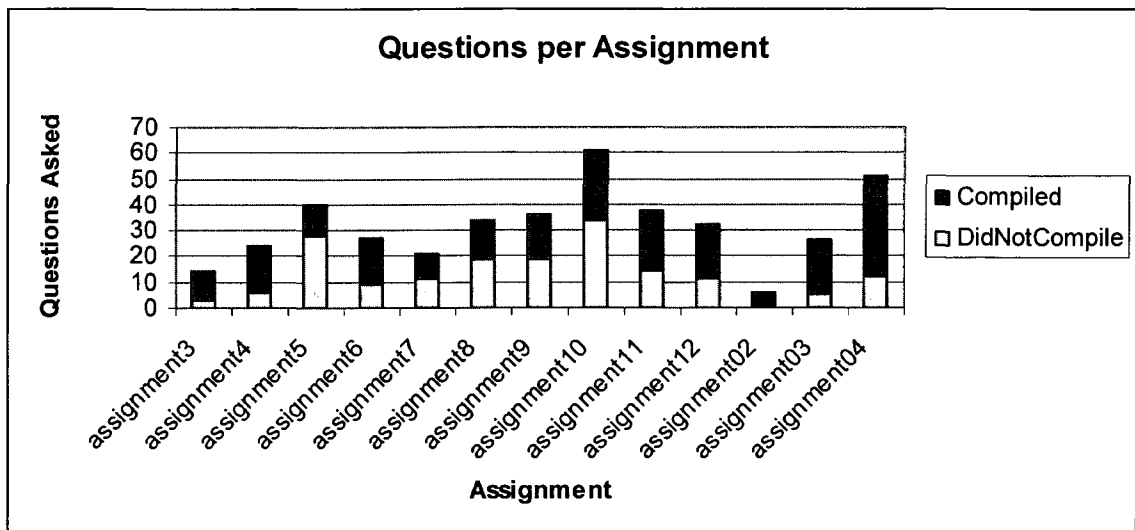
**Figure 3.3. Question Length**

students to indicate which Java class and method their question was about, less than a third actually did so, and of that third, many students only indicated a Java class, but not a Java method. Approximately half of the questions (43%) were submitted with code that did not compile.

### 3.7 Questions Per Assignment

With the assignment number as the independent variable and number of questions asked per assignment as the dependent variable, a bar chart shows which assignments generated the most questions. The number of questions is further disaggregated into the number of questions for which the code compiled and the number of questions for which the code did not compile. The results are shown in Figure 3.4.

One of the most interesting data points in the Spring 2008 data is assignment 5 which has many more questions asked than the other assignments in its neighborhood; it is also the only assignment for which the majority of the questions involve code that did not compile. A little background information may explain this spike. Immediately prior to assignment 5, the human TAs were given a rather stern lecture and told to make sure that all questions went through the logging system built into the Virtual TA software, even/especially the questions asked while they were circulating through the lab. If this is indeed what caused the spike in assignment 5, there are at least two interesting observations. First, the fact that many of the questions for assignment 5 involved code that did not compile suggests that students may be reluctant to pro-actively ask for help



**Figure 3.4. Questions Per Assignment**

resolving code that is obviously incorrect; perhaps they are embarrassed that their code does not compile? Second, if the spike can be attributed to prompted questions, then the data for assignment 5 suggest approximately 14 prompted questions (14 = 40 total questions for that assignment - 26 questions on average for the preceding and following assignments) out of 40 total questions or a prompted question rate of approximately 33%. Anthony et. al.[9] report approximately four unprompted questions for one prompted question or a prompted question rate of 20% for a tutorial system using chat, audio, and video. The difference in prompted question rates could be attributed to either the human doing the prompting or to the different experimental setups. Either way, tutorial strategies to increase the efficacy of prompting students to ask questions is worthy of further research, and results in this area are likely to be broadly applicable.

Two other observations from Figure 3.4 merit a little discussion. First, assignment3 and assignment4 have smaller question counts than assignment03 and assignment04. The difference can be attributed to enrollment differences in the Spring (about 80-90 students) versus the Fall (about 150-160 students). Second, many more questions were asked about assignment10 than the others. Assignment10 appears to have been an exceptionally difficult assignment for the students, but the reason is a mystery.

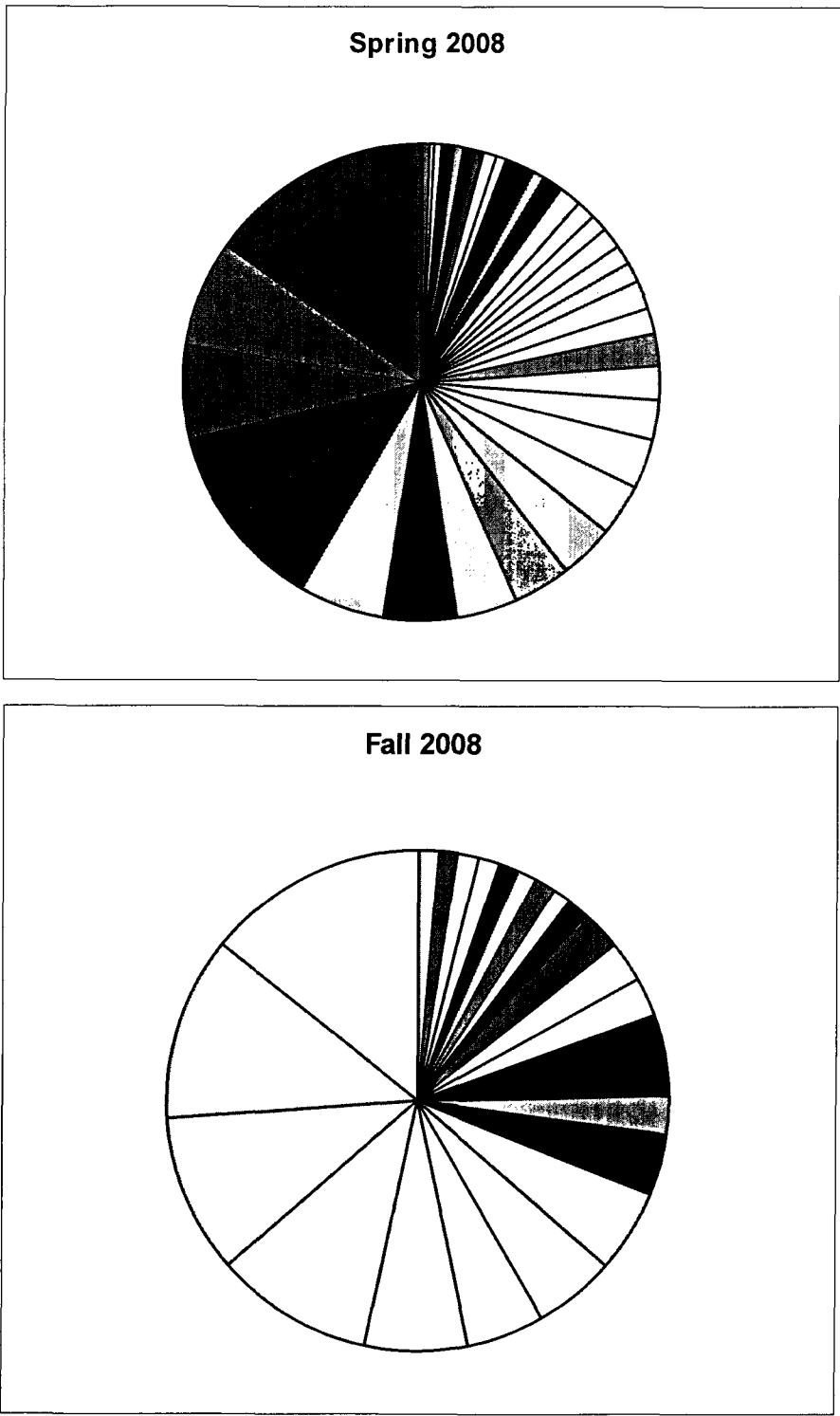
### **3.8 Questions Per Student**

During Spring 2008, a total of 39 students asked questions using the system software, and during Fall 2008, a total of 24 students asked questions using the system software. Figure 3.5 shows the percentage of questions asked per student. One observation from Figure 3.5 merits further discussion. One student asks significantly more questions than any of the other students in the Spring 2008 data. In both datasets, half of the questions can be attributed to five or fewer students. Personal experience and anecdotal evidence suggest that one student often contributes significantly more questions or dialogue than others, but many of the research papers on student questions report only a flat number, such as average number of student questions per hour [9, 34]; reporting flat numbers hides this behavior from the casual reader.

### **3.9 Questions Per Category**

How many questions occur in an answer category? To graph this, the x-axis represents the categories of answers for the top ten answer categories (categories with





**Figure 3.5. Percentage of Total Questions Asked by Each Student**

eight or more questions asked) ordered by the number of questions per category. The y-axis represents the number of questions asked for a particular category. Two series of data are presented, the total number of questions, and the number of questions with code that did not compile. Figure 3.6 shows the resulting graph.

Figures 3.7 and 3.8 shows similar graphs disaggregated by assignment, with a line for each assignment. The x-axis indicates the rank of the answer category with the most popular question for an assignment having a rank of 1. The y-axis indicates the number of questions asked in the category with that rank. Different assignments may assign different ranks to the same answer category, and the answer category of a particular rank generally differs across assignments. For example, the category with rank 1 for assignment3 is not the same category as the category with rank 1 for assignment4, although they are graphed in same position on the x-axis. The vast majority of the repetitive questions are contained in the top five categories per assignment. This suggests with very high likelihood that if the correct answer is not contained in the top five categories, it is not a repetitive question. Also, the number of repetitive questions is much higher in the data from Fall 2008 when more students were enrolled in the class. That suggests that this approach probably scales well, and it is probably more appropriate for larger classes.

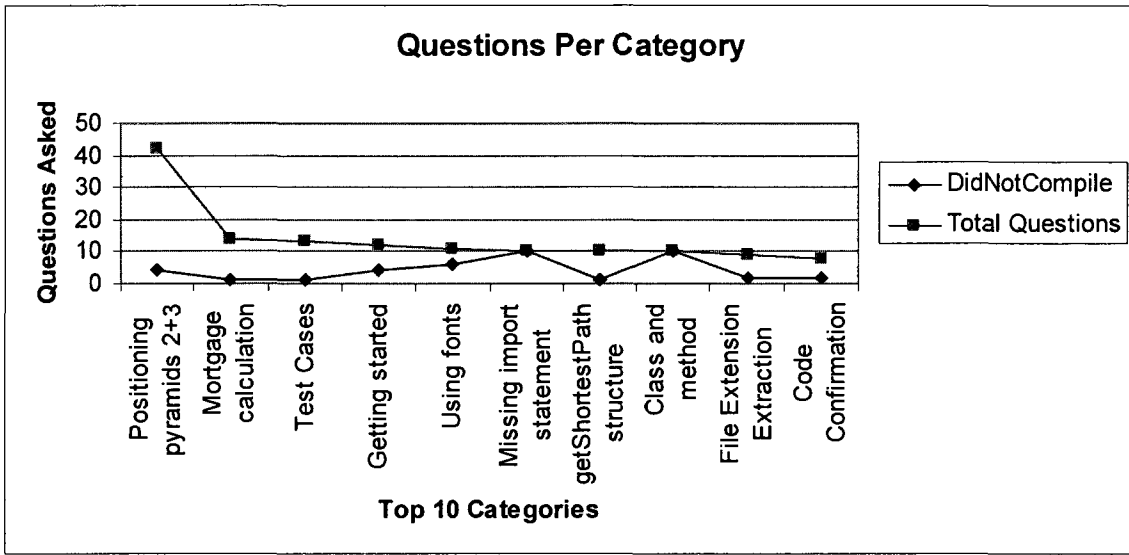


Figure 3.6. Questions Per Category

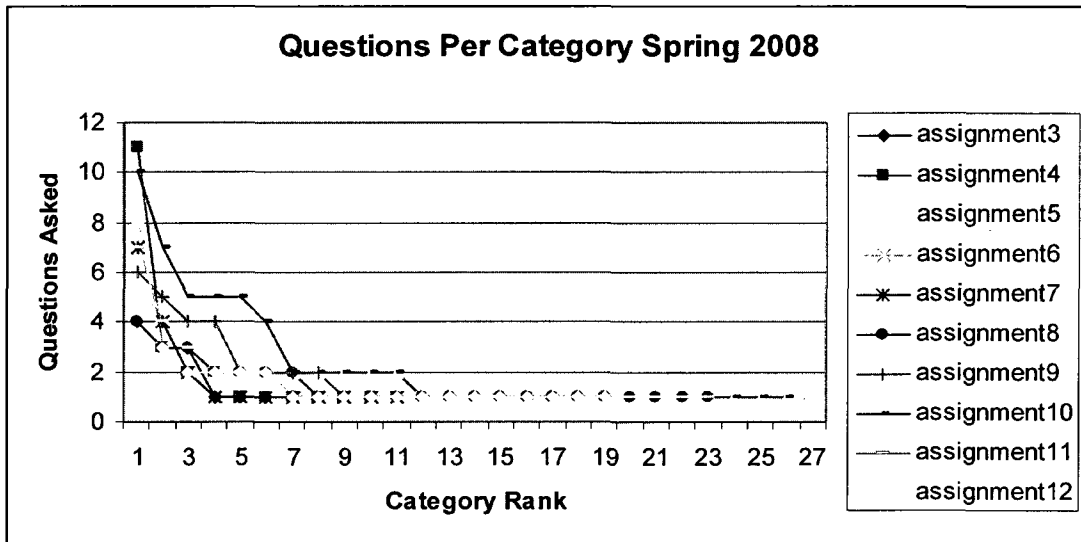
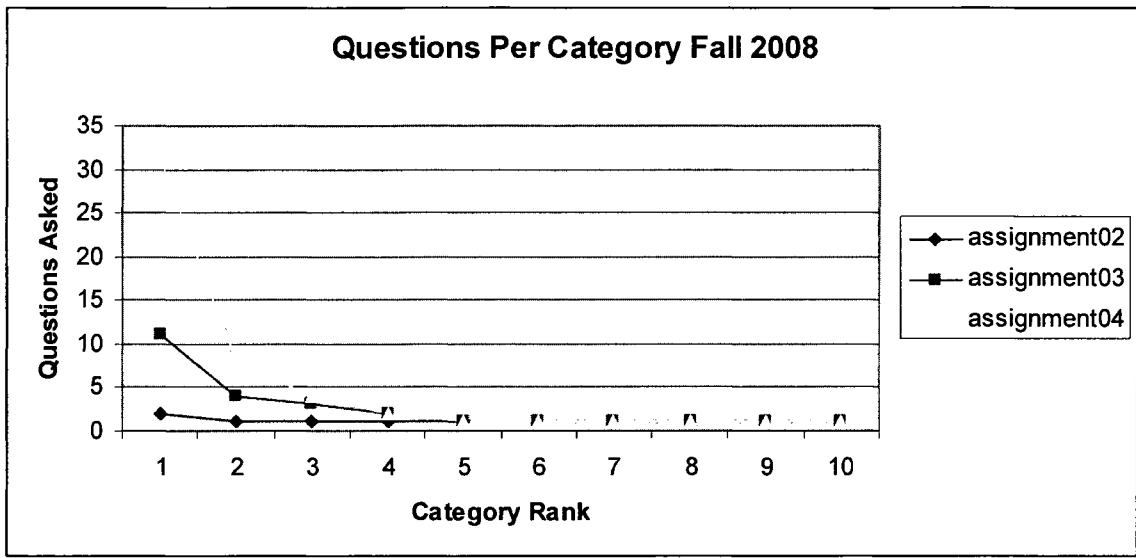


Figure 3.7. Questions Per Category Spring 2008



**Figure 3.8. Questions Per Category Fall 2008**

## **CHAPTER 4**

### **ANALYSIS**

The majority of the research described in this dissertation is focused on a particular research question, “Can domain knowledge and educational context improve the classification of student questions?” This chapter answers that question in the affirmative for introductory computer science.

Classification is a classic machine learning problem, and extensive prior work exists for classifying data. However, much of that research requires that the data be in vector or matrix form prior to the application of the machine learning algorithm, and reducing the compiler error messages, source code, and educational context to vector or matrix form is not a trivial problem. The remainder of this chapter describes and analyzes a general approach to classifying the questions in my dataset.

I frame question classification for introductory computer science as a process in which the data from the student questions, including the natural language, is reduced to a vector space and cosine similarity is applied to find similar questions. This dissertation analyzes 411 questions from an introductory Java programming course by reducing the natural language of the questions to a vector space, and then utilizing cosine similarity to identify similar previous questions. I report classification accuracies between 23% and

56%, obtaining substantial improvements by exploiting domain knowledge (compiler error messages) and educational context (assignment name). The mean reciprocal rank scores are comparable to and arguably better than most scores reported in a major information retrieval competition, even though the dataset consists of questions asked by students that are difficult to classify.

#### **4.1 Building a Matrix from Natural Language**

Reducing natural language to a vector space model is a relatively well-studied problem, and the techniques utilized in this research have been utilized in other similar systems (e.g., [37, 50]). This research performs the reduction in a four step process. First, the sentence is tokenized, and punctuation is removed. The resulting bag-of-words model is especially appropriate for the grammatically incorrect, difficult to parse language with which students express themselves. Second, any words found in a list of common stopwords are removed. The stopword list is the same one utilized in a similar tutorial dialog system [37] and is the stop word list in the original Bell-Core language processing distribution. Empirical comparisons with other stopword lists available from the internet suggested that this list was as effective or more effective than others for the task. I extended the stopword list with the following four words that behave as stopwords for this task: “im” (students’ shortened form of I’m), “problem,” “need,” and “help.” Third, the remaining tokens are stemmed with a Porter stemmer [61]. Stemming reduces words to their morphological roots, generally discarding suffixes. For example, although “communicate” and “communication” are different words, they share the same stem

“communicat.” Finally, the algorithm builds a vector that contains each remaining word stem in the original question and the number of times it occurs in the question. Table 4.1 shows five questions from students, their vector stems, and their answer category. As explained in the previous question, the answer category is a label created by a human TA to describe a question. A matching pair of answer category tags is considered the gold standard for establishing question similarity. Table 4.2 shows an abbreviated vector representation of the first three questions. Although the data in this example happen to have binary values, the value of data could be any whole number.

**Table 4.1. Sample Questions, Vector Stems, and Answer Categories**

	Natural Language	Vector Stems	Answer Category
Q1	How do i return the file extension only?	return file extens	File extension extraction
Q2	my variable for rectSideOne is suppose to be 1/9, the program is returning a 0 for this calculation. I have no idea why.	variabl rectsideon suppos 1/9 program return calcul idea	Integer division
Q3	I need help extracting a file extension from a filename.	need help extract file extens filename	File extension extraction
Q4	Program is not computing volume correctly	program comput volum correctly	Integer division
Q5	Im having trouble understanding why (1/9) equals 0.0 instead of 0.111111	im trouble understand 1/9 Equal	Integer division

**Table 4.2. Natural Language Representation of Questions**

	how	return	file	extens	only	variabl	rectSideOne	suppose	program	Calcul
Q1	1	1	1	1	1	0	0	0	0	0
Q2	0	1	0	0	0	1	1	1	1	1
Q3	0	0	1	1	0	0	0	0	0	0

## 4.2 Weighting the Vectors

### 4.2.1 Notation

At this point, a little notation will be useful in describing the data and the analysis. The data displayed in Table 4.2 suggest a matrix composed of vectors  $v_{1...m}$  where  $m$  is the total number of questions that have been asked, and  $v_i$  is the vector representing question  $i$ . Each vector  $v_i$  consists of entries for each of  $n$  stems where  $n$  is the total number of unique stems in all of the questions. The value  $v_{ij}$  is the number of times that stem  $j$  occurs in question  $v_i$ .

The next step in the typical analysis using cosine similarity is to weight the vectors. In this research, the vectors are weighted based on a common formula called term frequency inverse document frequency (tfidf score [1]) that gives more weight to rare words and less weight to common words. The tfidf score of a stem  $j$  in a question  $i$  is the product of the term (stem) frequency and the inverse document frequency. The term frequency is the number of times a stem  $j$  occurs in a question divided by the number of stems in the question  $i$ . The inverse document frequency is the logarithm of the total number of questions divided by the number of questions with the term (stem)  $j$ . Equation 4.1 shows the formula for calculating the weights for a stem ( $w_{ij}$ ) in a matrix  $w$  of weighted vectors  $w_{1...m}$  using tfidf given an initial set of vectors  $v_{1...m}$  where  $m$  is the total number of questions. Table 4.3 shows the weighted version of the data in Table 4.2.

$$\text{Equation 4.1. tfidf score}$$

$$w_{ij} = \left( \frac{v_{ij}}{\sum_{k=1}^n v_{ik}} \right) \ln \left( \frac{m}{\sum_{k=1}^m [v_{kj} > 0]} \right)$$



**Table 4.3. Weighted Natural Language Representation of Questions**

	How	return	File	extens	only	variabl	rectSideOne	suppose	program	calcul
Q1	.21	.13	.13	.13	.21	0	0	0	0	0
Q2	0	.11	0	0	0	.18	.18	.18	.18	.18
Q3	0	0	.34	.34	0	0	0	0	0	0

### 4.3 Measure Similarity in an Online Learning Framework

This section describes how cosine similarity is used in an online learning framework to identify similar questions. The most recent question that a student has submitted is considered the current question. Every question that occurred earlier in time than the current question is considered a previous question. Each of the previous questions is compared to the current question, and a similarity score is calculated as explained in the next paragraph. The previous question with the highest similarity score when compared to the current question is considered the most similar. For example, in Table 4.3, Q2 would only be compared to Q1. However, Q5 would be compared to Q1, Q2, Q3, and Q4. Of these, Q2 would be the most similar because it has the highest similarity score.

Several approaches could be utilized to measure the similarity of two vectors. Manning and Schutze list six different possible solutions for this problem [54]. I focus on the most commonly used approach in tutorial dialog systems, known as cosine similarity, as shown in Equation 4.2. Cosine similarity measures the cosine of the angle between two vectors,  $c$ , the current question vector and  $p$ , a previous question vector. The numerator of the cosine similarity is the dot product of the two weighted vectors. The denominator of the cosine similarity contains normalizing terms so that the magnitude of

both vectors is one. The resulting cosine similarity score is on a scale of 0 to 1. A cosine similarity score of 0 means that the pair has no common words, and 1 means that the questions are identical.

**Equation 4.2. Cosine Similarity**

$$\frac{\sum_{j=1}^n w_{c_j} w_{p_j}}{\sqrt{\sum_{j=1}^n (w_{c_j})^2} \sqrt{\sum_{j=1}^n (w_{p_j})^2}}$$

#### 4.4 Similarity Analyses

For each question, the similarity between that question and each previous question is calculated. The previous question that has the highest similarity score when paired with the current question is considered the “most similar.” If multiple previous questions have the same highest similarity score, the most recent new question is considered the “most similar,” because based on the principle of temporal locality more recent questions are more likely to be more similar. Once the “most similar” question has been identified, the answer category labels (as assigned by the expert human TA and verified by another TA) for the “most similar” question and the current question are compared. If they match, the system earns a point for accuracy, and if they do not match, the system does not earn a point for accuracy.

I report accuracy scores with two different denominators, all questions (411) and repetitive questions (268). Of these, only the repetitive questions bar could theoretically

reach 100%. In both scores, the numerator is the number of correct similar questions found (96). As shown in Figure 4.1, the trivial baselines include choosing a random answer, the most frequent answer, and the most recently used answer. Of those, the most recently used answer is the most effective algorithm, suggesting the importance of temporal locality in answering student questions. The cosine similarity algorithm can classify 35% of the repetitive questions or 23% the total questions. For those questions, an answer to a previous question could theoretically be recycled to answer that question. Another way of describing this result is that in a course with five human TAs supporting the professor, one of those TAs could be replaced by the software.

#### 4.4.1 On the Omission of Statistical Significance Tests

At this point, many readers may wonder whether or not the differences in accuracy reported by the various baselines are statistically significant. Statistical significance tests are

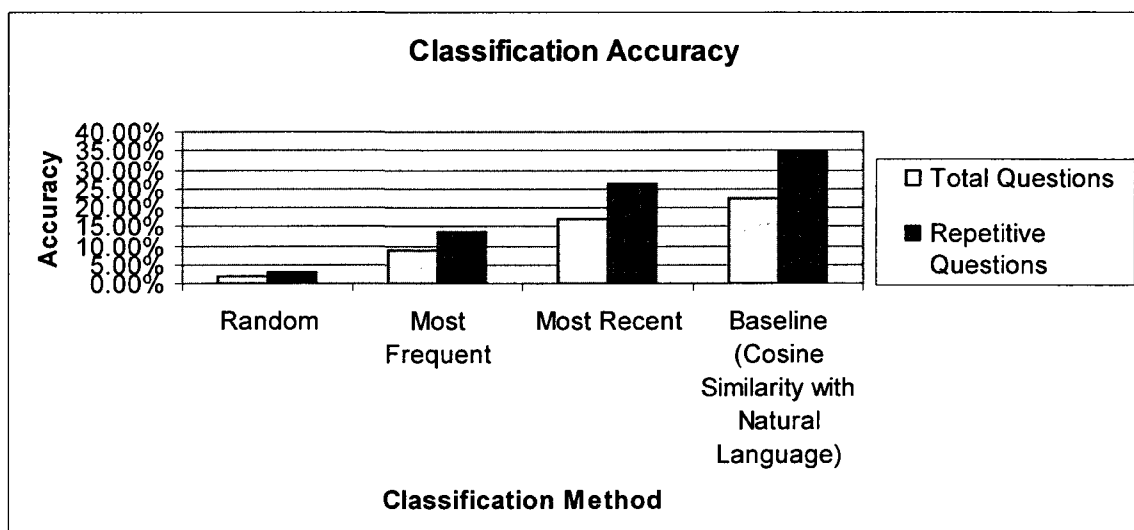


Figure 4.1. Classification Accuracy Baseline

meant to show whether differences in two distinct populations can be attributed to a variable that distinguishes those two populations or just chance. However, all of the data used in each condition in the experiments reported in my dissertation come from the same population, so any change in classification accuracy can be attributed to the algorithm. Because the experiments reuse data in the various conditions, statistical significance tests would be unlikely to produce accurate values, and in many cases would over-estimate the significance because existing statistics require that the data used in a test come from two distinct sets.

#### **4.5 Compiler Output Processing**

The classification techniques described so far in this chapter have been used previously for a variety of tutorial dialog tasks, and they are inherently domain independent. However, the low accuracy of question classification suggests room for substantial improvement. One possible way to improve classification is to leverage some domain specific knowledge, specifically the output of the compiler. Since more than 40% of the questions were submitted with code that did not compile, the compiler error messages represent a source of substantial unused data.

Previous work has examined compiler errors from novices learning to program in Java. Different research groups have reported widely varying numbers for the different kinds of compiler errors. Table 4.4 reports on the number of kinds of compiler errors reported by three projects that studied novice Java programmers.

**Table 4.4. Compiler Error Type Messages**

<b>Project</b>	<b>Error Type Messages</b>
BlueJ [42]	42
GILD [76]	88
Jikes [2]	226

Because all three groups worked with similar data, the reports with numbers that differ by an order of magnitude are somewhat surprising. However, a more careful investigation shows that these groups are aggregating and disaggregating the kinds of compiler errors differently. For example, the common novice compiler errors of type “cannot find symbol” are aggregated together to form a single error type message in the BlueJ project. However, in the GILD and Jikes projects, “cannot find symbol” errors are disaggregated into four or more types including “cannot find symbol-constructor,” “cannot find symbol-class,” “cannot find symbol-variable,” and “cannot find symbol-method.” Similar aggregations and disaggregations for other compiler error messages probably explain the widely varying number of error type messages, although the limited documentation makes verification impossible.

Unfortunately, even the groups that have disaggregated error type messages may not have been as comprehensive as they could have been in designing their disaggregations. Consider for example the “cannot find symbol-class” error type message. If it is followed by the word “string,” a knowledgeable instructor might suspect that the underlying error is in fact a capitalization problem. However, if the error type message is “cannot find symbol-class” and it is followed by the word “Scanner,” the likely error is not a capitalization problem, but rather a missing import statement.

Similarly, the compiler will frequently report a “missing semicolon” when the student has imbalanced parentheses. In this research, the term “underlying error” refers to problems such as capitalization and missing import or missing parentheses. No other research that we are aware of classifies student questions or code snapshots according to their underlying errors.

A naïve approach to incorporating compiler output into the vector space model would be to simply tokenize the error messages and include them just as the natural language was included. The problem is that errors such as missing import and capitalization will appear to be very similar because they contain four similar tokens (“cannot,” “find,” “symbol,” and “class”), and the algorithm will be unable to distinguish between them. To remediate this problem, some of the most common compiler errors and code snapshots are processed by Java code that generates a brief description of the underlying error based on the code snapshot and then the underlying error is incorporated into the model. To facilitate replication, the conversions from compiler errors to the underlying error representation used in this research are described in Table 4.5. Of the nine compiler errors that the system processes, five are ambiguous, and the system uses information from the Internet and the students’ source code to pinpoint the exact error.

Table 4.6 shows vector representations for three compiler errors. Again, these vectors are based on the same general approach used for the natural language, and once again, these rows would extend the existing matrix with natural language.

- cannot find symbol class string (CE1)
- cannot find symbol class Scanner (CE2)
- variable foo is already defined (CE3)

**Table 4.5. Conversion of Compiler Errors to Underlying Error Terms**

<b>Compiler Error</b>	<b>Natural Language Representation</b>
Contains “already defined”	alreadyDefined
Contains “incompatible types”	incompatibleTypes
Contains “not have been initialized”	notHaveBeenInitialized
Contains “cannot find symbol- constructor”	cfsConstructor
Contains “cannot find symbol-class” AND search engine returns more than 100,000 hits when searching for the specific class AND the first result is capitalized differently than the specific class	cfsCapitalization
Contains “cannot find symbol-class” AND the source code does not contain the pertinent import statement	cfsMissingImport
Contains “cannot find symbol-class” OR “cannot find symbol-variable” AND the source code contains another symbol that is capitalized differently	cfsCapitalization
Contains “cannot find symbol- variable”AND the source code contains that symbol followed by a parenthesis	cfsMissingParenthesis
Contains “cannot find symbol- method” AND and the source code contains that symbol followed by a parenthesis	cfsMethodMismatch

**Table 4.6. Database Representation of Compiler Errors**

	Capitalizat	missingImport	alreadyDefined
CE1	1	0	0
CE2	0	1	0
CE2	0	1	1

## 4.6 Answer Caching

Previous work has exploited a technique called answer caching to provide answers to some questions that utilize different wordings to express the same information need [58]. Answer caching matches an incoming question to a similar previous question in order to recycle an answer. The answer caching technique then leverages the additional language in the similar question to build a more robust language model of that information need. Specifically, answer caching merges the data from vectors that indicate a similar information need to form a single vector. Without answer caching, the five questions in Table 4.1 are modeled with five vectors. With answer caching, they are represented with two vectors, one for “File extension extraction” (the sum of the vectors for Q1 and Q3) and one for “Integer division” (the sum of the vectors for Q2, Q4, and Q5). The original paper on answer caching reports a 1-3% improvement on a dataset with well-formed, grammatical, well-spelled questions. Figure 4.2 demonstrates a similar improvement when incorporating both answer caching and the processed error messages. Interestingly, the processed error messages alone do not improve classification, and answer caching alone only produces minor improvements ( $< 1\%$ ), but the combination of the techniques improves accuracy by 3% of the total questions. As shown in Table 4.7 answer caching does not improve classification for the compiler errors by themselves. This is probably because the recency algorithm chooses the question added to the model most recently, not the question asked most recently. However, answer caching helps substantially when the natural language is included in the model. The number of correctly classified questions (or numerator) for the “With Answer Caching and Error



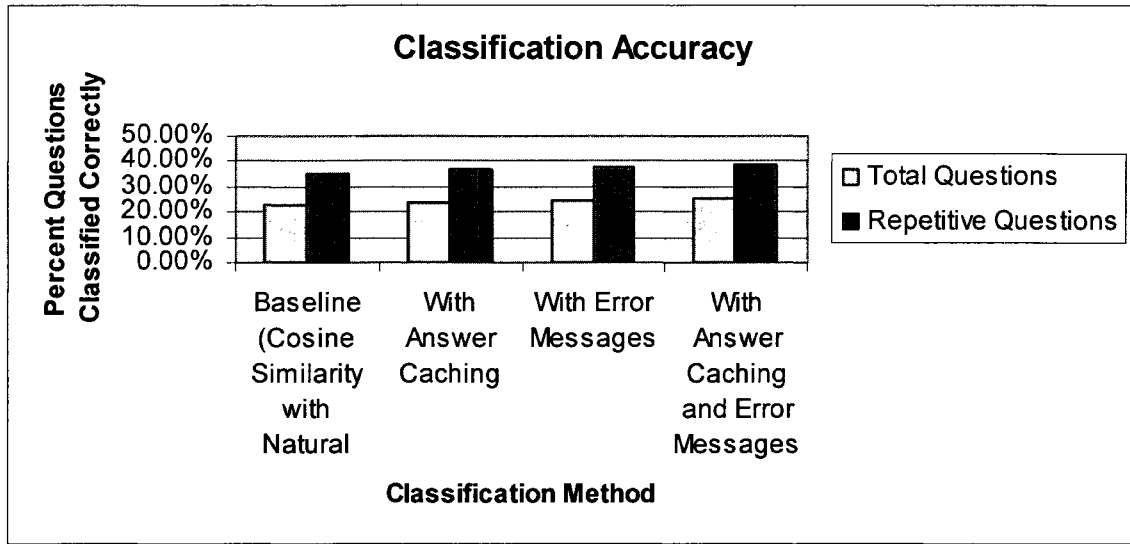


Figure 4.2. Classification Accuracy with Answer Caching and Error Messages

Table 4.7. Questions Classified Correctly with Compiler Error Messages

	Raw Compiler Error Messages		Processed Compiler Error Messages	
	Without Answer Caching	With Answer Caching	Without Answer Caching	With Answer Caching
Baseline	79	34	79	24
With Natural Language Terms	94	96	92	104

Messages” method is 104, and the denominators are the same as they was in the baseline conditions, 411 for total questions and 268 for repetitive questions.

#### 4.7 Disaggregating by Assignment

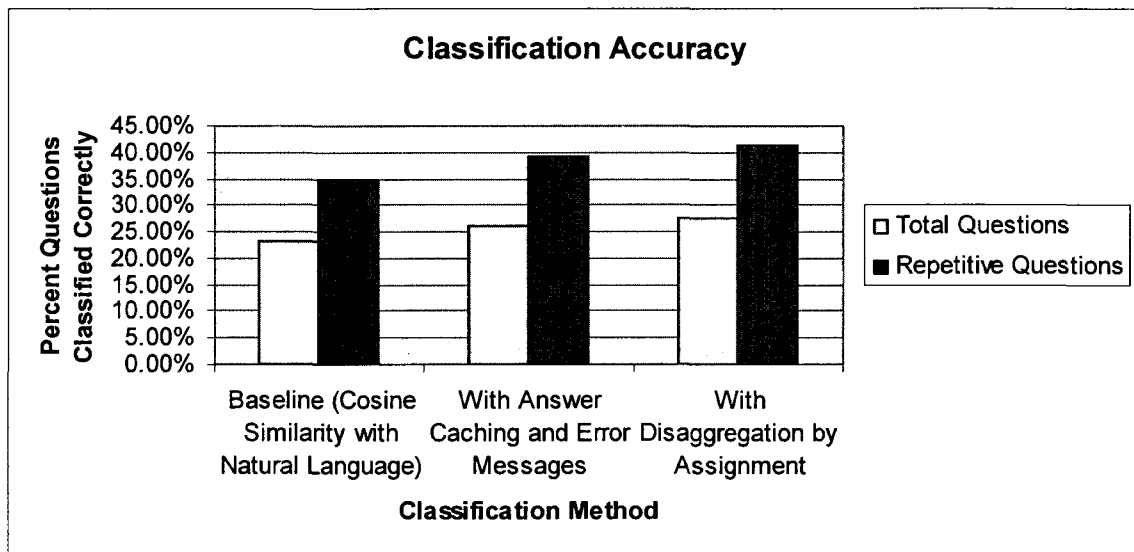
For a final improvement in classification accuracy, the data was disaggregated by assignment. For example, assignment1 questions were compared only to other assignment1 questions and assignment5 questions were compared only other assignment5

questions. As shown in Table 4.8 and Figure 4.3, this technique improved the number of correctly classified questions (or numerator) to 113. To facilitate comparison in the bar charts, we reuse the same denominators, 411 total questions and 268 repetitive questions. However, when only comparing questions from the same assignment, the number of repetitive questions is smaller (201) and that denominator gives a classification accuracy of 56% of repetitive questions. With the data disaggregated by assignment, incorporating answer caching and error messages reduced accuracy slightly (101 questions classified correctly or 50% when 201 is the denominator). Statistical significance tests are omitted as explained in 4.4.1.

The lack of sufficient data to model different kinds of compiler errors is probably the cause of a drop in accuracy when answer caching and error messages are incorporated. Because compiler errors are being reduced to a single term, several of them are necessary to boost the compiler error terms to a heavy enough weight to influence the similarity algorithm. However, excluding error messages and answer caching returns the classification algorithm to a domain independent state. Compiler error messages are a source of data that are relevant only in the computer science domain. By contrast, natural language and assignment numbers are a data source that is available in virtually every educational domain. The claim that the system is domain independent is strictly theoretical without empirical data from another domain, but it is a positive claim.

**Table 4.8. Classification Accuracies**

	Aggregated		Disaggregated	
	Total Questions	Repetitive Questions	Total Questions	Repetitive Questions
Baseline	93/411 (23%)	93/275 (35%)	113/411 (27%)	113/204 (55%)
With Error Msgs and Answer Cache	104/411 (25%)	104/275 (39%)	111/411 (27%)	111/204 (54%)

**Figure 4.3. Classification Accuracy with Disaggregation by Assignment**

#### 4.7.1 Analysis of Errors

Comparing individual instances of questions that the three major classification algorithms (baseline, domain knowledge, and compiler errors) classified differently is difficult because many elements are changing in each condition, including both the number of questions that a question is compared to and the weights of the terms in a question. Furthermore, the quantity of these changes is substantial enough that trying to manually understand them will probably cause cognitive overload. However, for

comparison, this section presents an analysis of some of the questions that were classified correctly and incorrectly by various algorithms.

#### *4.7.1.1 Baseline vs. Domain Knowledge (Processed Compiler Errors)*

Of the 14 instances that the baseline condition classified correctly and the domain knowledge condition classified incorrectly, only 3 had compiler errors. Many of the questions in this category appeared to be broad questions expressed in few words. For example:

- I need help with while loop and linked list.
- I need help with the maze.
- Is there any help on how to draw a pyramid?
- TA\_NAME i need your help when you get a chance.

Of the 25 instances that the domain knowledge condition classified correctly and the baseline classified incorrectly, 8 had compiler errors. Surprisingly, the majority of these were submitted with code that compiled. However, incorporating the information from error messages would have changed the weights in the model, and that may have caused the algorithm to make fewer mis-matches. Some examples of questions in this category include:

- Im having trouble understanding why  $(1/9)$  equals 0.0 instead of 0.1111....
- I am trying to fix my count CorrectChars method- i know what i want to do but not sure how to fix it
- my program cannot find the class file
- my compiler is saying it cant find setPrevious method even though i am passing it the right parameters

#### 4.7.1.2 Baseline vs. Educational Context (Assignment)

Of the 16 instances that the baseline condition classified correctly and the educational context condition classified incorrectly, only 3 had compiler errors. Many of these questions demonstrated at least some mastery of domain vocabulary. For example:

- Hello, I have no idea how to scale these pyramids. Can you point me in the right direction? Yeah, I have that code, but in the lab friday... The smiley DID NOT scale correctly, and TA\_NAME said he didnt have time to fix it. So he told us to just worry about the x,y position
- Getting some dumb null pointer nonsense
- So we dont use g.fillpolygon right? we are suppose to use g.drawLine ?

Of the 36 instances that the educational context condition classified correctly and the baseline classified incorrectly, 2 had compiler errors. Many of these questions appear to have come from student seeking reassurance that they were on the right path and making progresss towards completing their assignment. Some examples of questions in this category include:

- I am wondering if my class looks good or i should change a few things- i think it looks good. i am looking at the documentation
- Im not really sure if Im doing this correctly or not, I cant find any examples of contstructor stubs in the book.
- Is there any way I can prevent an "index out of bouds exception" on my array?  

```
String [][] mazeArray = new String [cols][rows]; mazeArray [cols - 1][rows] = "S";
```

Obviously cols - 1 is out of bounds, but canI somehow make it null or something along those lines?
- Alright I think I got it all. How does that look? Oh and any suggestions on code formating or commenting?

### 4.8 Alternative Evaluations of Question Answering Systems

The most accurate classification algorithm found similar previous questions for 27% of the questions or 42% of repetitive questions. Unfortunately, automatically

determining whether or not there is a similar previous question (or if this question is repetitive) is a prerequisite to exploiting similar questions in a tutorial intervention. One possible method of distinguishing between true positives and false positives utilizes the similarity score of the “most similar” question. The similarity score of the “most similar” question is considered the “maximum similarity score”. With an ideal threshold, all of the true positives (questions with correct similar previous questions) have a maximum similarity score above threshold, and the false positives (questions without similar previous questions) have a maximum similarity score below threshold. Figure 4.4 is an attempt to find an appropriate threshold. The independent variable is the maximum similarity score for a question, and the dependent variable is 0 if the algorithm did not find a similar previous question and 1 if the algorithm correctly identified a similar question. If Figure 4.4 were a step function, then the threshold would be at the step.

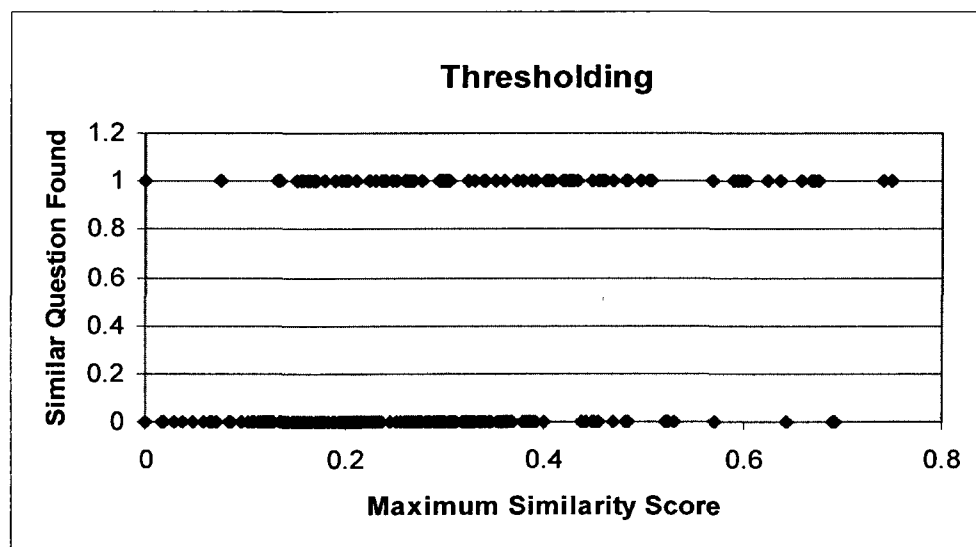


Figure 4.4. Thresholding

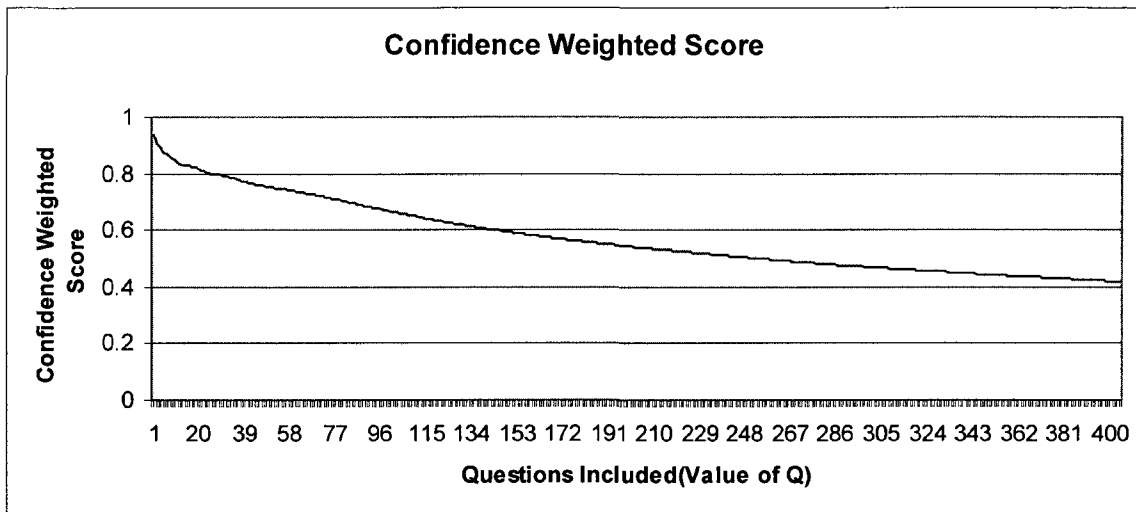
Figure 4.4 is a great way to visualize data while searching for thresholds, but cannot quantitatively measure the trade-offs between confidence and accuracy. To measure this trade-off, the TREC 2002 Question Answering Track of the Text REtrieval Conference (TREC) utilized confidence weighted scores as shown in Equation 4.3 [82]. The maximum similarity score for a question is also a measure of confidence; a higher maximum similarity score suggests that the system is more confident that the answer is correct. Consequently, the maximum similarity scores can be used to rank the questions from most to least confident.

In mathematical notation, a sequence of  $Q$  questions is sorted based on the maximum similarity score. The rank  $i$  of a question is the index of that question in the sorted sequence. A low value of  $i$  indicates a high maximum similarity score and a high value of  $i$  indicates a low maximum similarity score.

The confidence weighted score is shown in Figure 4.5 as a function of the number of questions. The x axis indicates the number of questions included sorted by descending order of maximum similarity score. The y axis indicates the confidence similarity score. The VTA system has a confidence weighted score of 0.41. This is a reasonable score for this task and dataset.

**Equation 4.3. Confidence Weighted Score**

$$\frac{1}{Q} \sum_{i=1}^Q \frac{\text{number correct in first } i \text{ ranks}}{i}$$



**Figure 4.5. Confidence Weighted Score**

Although identifying the appropriate information on the first try is desirable, it is not an essential characteristic of a good information retrieval algorithm. Search engines typically return thousands of results for a single query, and it is not unusual for users to consult multiple links before finding the desired information. One measurement that considers multiple possibilities is mean reciprocal rank. Using mean reciprocal rank, a system considers a ranked list of possible answers for each query or question. The score that a system receives for each question is the reciprocal of the rank of the possible answer that contains the actual answer. If the rank of the possible answer that contains the actual answer is greater than 5, the system receives a score of 0 for that question. Because the system is using answer caching, at most one of the possible answers will match the actual answer. Using mean reciprocal rank, 40% of the questions had a score greater than or equal to 0.2. The number of questions asked is shown as a function of



mean reciprocal rank for the 40% of questions with a score greater than or equal to 0.2 in Figure 4.6.

The mean reciprocal rank score of an entire system is the average of the mean reciprocal rank scores of all the questions asked using the system. Using all of the questions, the VTA system has a mean reciprocal rank of 0.31. Using only the repetitive question, the VTA system has a mean reciprocal rank of 0.46. These scores are reasonable.

#### 4.9 Synthesizing an Algorithm to Classify Questions

The majority of this chapter is focused on analysis of the data and various measurements. One point of concern is that incorporating compiler errors does not improve accuracy when the assignment is disaggregated by assignment. Table 4.9 shows the accuracies for the three major classification algorithms when the code is

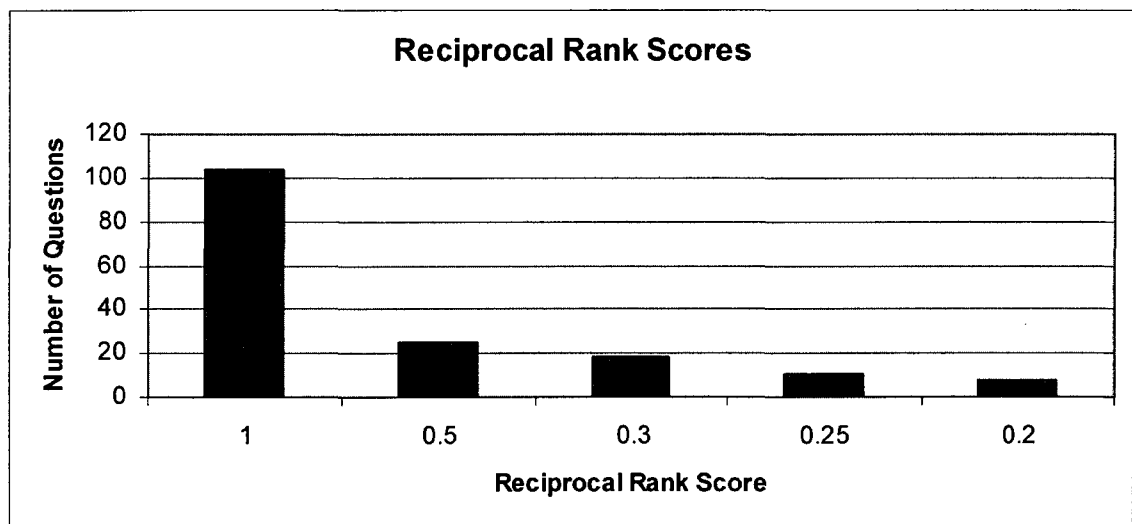


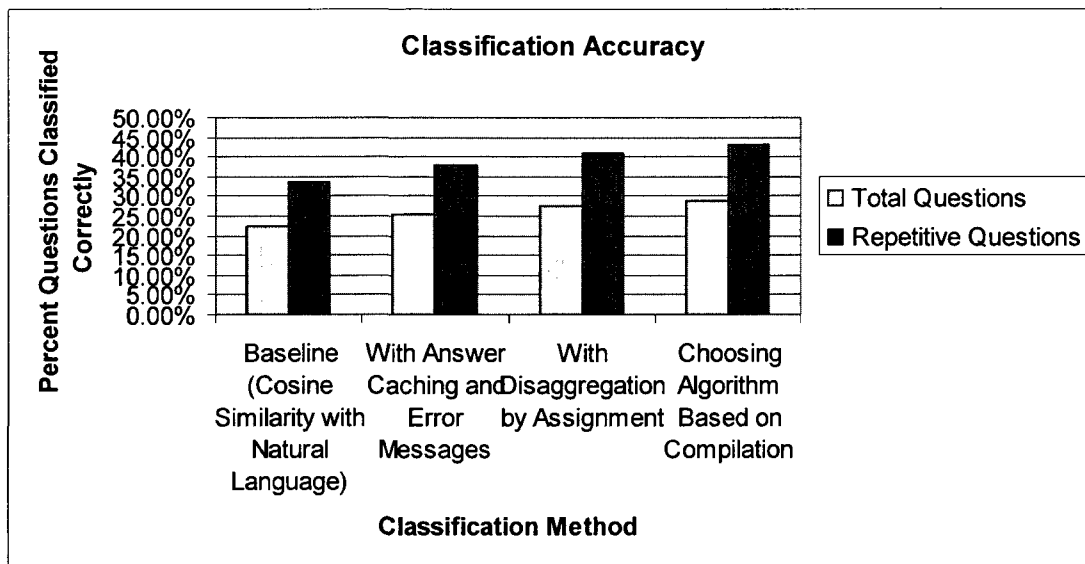
Figure 4.6. Reciprocal Rank Score

**Table 4.9. Classification Accuracies when Disaggregated by Compiler Error**

<b>Algorithm</b>	<b>Compiler Status</b>	<b>Accuracy</b>
baseline	didNotCompile	0.146199
domain	didNotCompile	0.210526
context	didNotCompile	0.169591
baseline	Compiled	0.283333
domain	Compiled	0.283333
context	compiled	0.35

disaggregated based on whether or not it compiled. Not surprisingly, the best algorithm when the code does not compile is the domain knowledge algorithm that includes compiler errors. The best algorithm when the code does compile is the context algorithm that only compares to other questions from the same assignment. Whether or not the code compiles is easy to automatically compute, and using the results, the system can automatically decide to use the appropriate algorithm, similar to previous work on choosing a correct intervention for students learning to read [39]. By using the information about whether or not the code compiles, the system can automatically select the more accurate classification algorithm to classify 119 questions or 29% of the total questions and 44% of the repetitive questions correctly as shown in Figure 4.7.

Finally, if the cosine similarity score is below .15, the likelihood is very great that the system does not have an answer for the question, and approximately 20% of the data falls into this bin. For these questions, the system would not even attempt an answer. For the questions where the system thinks it might have an answer, 40% of the time the correct answer is one of the top five ranked answers. Assume that the system returns five possible matching questions and answers for each query, and assume that the system does



**Figure 4.7. Classification Accuracies**

not even try to answer the 20% of the questions with the lowest cosine similarity, so that the system is only attempting to answer 80% of the questions. Then the system will return a relevant answer for 40% of all questions divided by 80% of questions attempted or 50% of the time. That is probably good enough to be potentially useful in a real classroom setting.

#### **4.10 A Theoretical Cost and Benefit Analysis**

The introduction began by suggesting that human resources for courses represent a significant cost in education that could be reduced by the clever application of technology. An empirical analysis of this issue is beyond the scope of the dissertation; however, a theoretical analysis is presented here in the hopes that it may inform future studies.

At least four major input variables would influence the value of this system; they are the quality of the instructing staff, the availability of the instructing staff, the quality of the answers in the system, and the quality of the classification algorithm. For now, it is reasonable to suppose that the quality of the classification algorithm is going to be somewhat bad, between 25% and 50% accurate. Because a quality instructor could be hired to write answers for the system, it is reasonable to assume that the quality of the answers in the system will be quite good. The quality and the availability of the teaching staff are much more difficult to measure and evaluate, and will vary greatly depending on the courses in which the system is deployed. Large introductory courses typically employ several teaching assistants, and the quality of the teaching assistants in the introductory courses usually varies greatly including some of the best and some of the worst within the department. The best teaching assistants and the professor are probably available for a maximum of ten hours a week combined. The remaining teaching assistants may be available for as little as ten hours a week or as much as the whole week, but they may or may not be able to provide useful help and feedback. Since many novice programming students plan to program at midnight, and most instructors are not available then, it is reasonable to suppose that the availability of the instructional staff is not good, and may even be bad.

Given a question, the system can do one of three things; the system can answer correctly, the system can answer incorrectly, and the system can decide to wait for a human to answer the question. If the system answers correctly by recycling an answer, it saves human instructional staff time, and depending on the availability of the

instructional staff (which is probably not good), it saves substantial student time. If the system answers incorrectly, it may cause student frustration and/or confusion; however, if the instructional staff was unavailable (which is probable), then student frustration and/or confusion may have been inevitable anyway. If the system waits for a human to answer the question, then the system does not fundamentally change the outcome. Thus, the chief benefit of the system is the potential to save human time for both the student and the instructional staff. The chief costs of the system are potential frustration for students (which may be inevitable) and the time to develop the system, which should be close to a constant assuming a stable portfolio of assignments.

## CHAPTER 5

### PERIPHERAL ANALYSES

The process of completing the research for a dissertation often allows a researcher to explore several possibilities, including some that are not empirically impressive. This chapter includes justification for excluding some analysis techniques as well as some miscellaneous results that are not directly related to those already presented. They are included because they may be useful to other researchers with similar but different research interests.

#### 5.1 Latent Semantic Analysis

In Latent Semantic Analysis (LSA) [52], a form of principle component analysis called singular value decomposition reduces the dimensionality of the matrix. Then, cosine similarity is calculated on the rows of the matrix. LSA has a strong theoretical foundation including a link to human cognition and learning that provides a very appealing foundation [52]. Additionally, LSA has previously been implemented in the AutoTutor intelligent tutoring system as described by Hastings<sup>1</sup>, Graesser, and colleagues. That research group tuned the parameters for LSA and found slightly stronger performance by altering features such as the amount of training data and the

---

<sup>1</sup> Hastings has recently dropped the Wiemer portion of his last name.

number of dimensions in the matrix [87]. They found that 400 terms in the final matrix was an ideal number for LSA, a result that has been recently replicated in other literature [18].

Those results combined with measurements of the data used for this research suggest that LSA is probably not appropriate for the task of classifying novice programmer questions. Since the assignments have an average of 30 questions each, and each question has an average of 6 terms after stop words are removed, and many of those words are repetitive across questions, theoretically, the total number of terms for a typical assignment is less than 200. Figure 5.1 confirms this empirically. Both the theoretical and empirical analyses confirm that this data are well out of the range in which LSA would produce optimal performance according to existing theory. Consequently, LSA was not investigated as an empirical technique for this dataset.

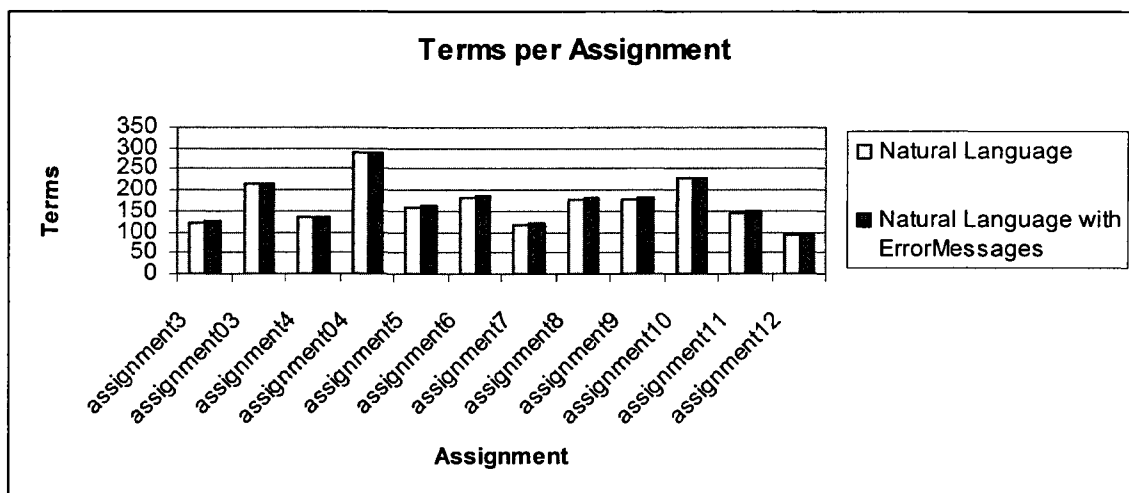


Figure 5.1. Terms per Assignment

## 5.2 Negative Results with Source Code

Designing abstract representations of student source code that are conducive to classification remains a difficult open problem. Simply tokenizing source code and extending vector representations with token types was not effective at all. A marginally effective alternative is to use a sequence of three tokens to create a trigram, and then build a vector representation of trigrams. For example, consider the following line of code:

```
(4+5)/(6+7));
```

That line of code can be broken into the following trigram sequences

```
(4+ 4+5 +5) 5)/ )/( /(6 (6+ 6+7 +7) 7)) );
```

These trigrams can then be used to extend the vector space models in a manner similar to the natural language additions. That approach was somewhat effective for one assignment, assignment6. In that dataset of 27 questions, six were repetitive, and the algorithm found four similar previous questions including extra semicolon and integer division errors and two on the structure of a method. However, when the data for that assignment was mixed with other assignments, any improvement in classification accuracy was lost. Creating partial parse trees or alignments of student code that does not compile remains an open problem.

Another alternative to incorporating student source code that may be more fruitful is applying a different measurement of similarity. This research utilizes cosine similarity because it has become the standard for tutorial dialog; however, that measurement was



originally used in the information retrieval literature. Part of the justification for using cosine similarity in traditional information retrieval is that neither the query nor the document is more important in determining a match. However, a query is often quite short (typically two words, e.g., [44]), and a document is typically much longer. By normalizing the vectors before comparing them, the weight given to the documents and queries is distributed evenly. Similarly, when comparing two student questions, a verbose question is not necessarily more important than a less verbose question. However, when comparing answer strings or student programs or responses, the fact that a program is longer often indicates that a student has made more progress. In such a scenario, using distances of vectors that have not been normalized may be more appropriate than cosine similarity.

### **5.3 Skipping Steps to Increase Accuracy**

Calculating the cosine similarity of two vectors is essentially a two step process. To calculate the numerator, the dot product of the two vectors is calculated. To calculate the denominator, the square root of the sum of the squares is calculated in order to normalize the vectors. In the process of writing and debugging the analysis code, I inadvertently discovered that skipping the square root step in the normalization step of calculating the denominator produces minor improvements in classification accuracy. In the condition with the highest classification accuracy (disaggregation by assignment), this minor modification improved classification accuracy by one question. In some of the other conditions, it improved accuracy by a few more questions, but not enough to

compete with the disaaggregation by assignment condition. Since the weights representing the words are all decimal values between 0 and 1, taking the square root of the values shrinks the spectrum and redistribute the weight. Consequently, I speculate that the square root step gives more weight to less important words and less weight to more important words.

I reverted to the traditional definition of cosine similarity (and I have used it throughout the dissertation) to facilitate scientific comparison, but I note here that skipping the square root step does appear to slightly improve accuracy, and it would be less computationally expensive to skip that step.

#### **5.4 A Trained System to Classify Student Questions**

The research described in previous chapters of the dissertation employs an online learning framework to analyze and classify the questions that students ask. This approach is most appropriate while the system is learning and training on initial data; it models how a system would perform in its first semester. However, many courses recycle assignments across semesters, providing an extra source of redundancy that could be leveraged to potentially improve accuracy. The dataset collected for this dissertation facilitates two analyses in this area.

##### **5.4.1 Exploiting Redundancy Across Semesters**

The first analysis examines whether or not including the data from Spring 2008 can improve the accuracy when classifying data from Fall 2008, using data from

assignment3 and assignment4 in Spring 2008 and assignment03 and assignment04 in Fall 2008. Assignment3 and assignment03 are the same assignment, given in different semesters, and assignment4 and assignment04 are also the same assignment, given in different semesters. Table 5.1 shows data from individual assignments as well as similar questions aggregated across semesters. Similar questions found indicates how many similar questions the online learning analysis algorithm found. Repetitive questions and total questions are self explanatory. Unique answer categories indicates how many different answer categories assignment(s) had.

As Table 5.1 shows, aggregating across assignments was less effective for assignment3 and assignment03 because disaggregated the algorithm could classify 18 (5+13) questions, but aggregated the algorithm could classify only 13 questions, for a loss of five questions. However, for assignment4 and assignment04, aggregating across assignments was effective, classifying 48 questions accurately compared to 46 (9+37) when the assignments are disaggregated. These results are divergent and inconclusive.

The unique answer categories column is somewhat revealing. In assignment3 and assignment03, merging categories across assignments reduces the total number of

**Table 5.1. Redundancy Across Semesters**

	<b>Similar Questions Found</b>	<b>Repetitive Questions</b>	<b>Total Questions</b>	<b>Unique Answer Categories</b>
assignment3	5	7	14	7
assignment03	13	17	26	9
assignments3 & 03	13	25	40	15
assignment4	9	14	24	10
assignment04	37	41	51	10
assignments4 & 04	48	57	75	18

categories by 1  $((7+9)-15)$ ; however, in assignment4 and assignment04, merging across assignments reduces the total number of categories by 2  $((10+10)-2)$ . In both cases, it appears that questions that are asked only once in a semester are unlikely to be asked again in future semesters.

#### 5.4.2 A Batched Classification Approach

Another way to study redundancy across semesters is to apply a batched learning approach instead of an online learning approach. With an online learning approach, the first question of an answer category will never be similar to another question. With a batched approach, the first question of an answer category could be found to be similar to another question if there is more than one question in that answer category. Additionally, with an online learning approach, each question is compared only to previous questions, but in a batched approach, each question is compared to all other questions, including questions asked after it. In the batched setting, the language model is completely trained before a question is classified.

More questions would be considered repetitive with a batched approach. Table 5.2 compares the number of questions classified correctly in the online version and a

**Table 5.2. Counts and Percentages of Correctly Classified Total Questions**

	<b>Online</b>	<b>Batched</b>
Aggregated Without Caching	98 (24%)	117 (28%)
Aggregated With Caching	104 (25%)	111 (27%)
Disaggregated Without Caching	111 (27%)	135 (33%)
Disaggregated With Caching	101 (25%)	126 (31%)

batched version, both when the data are aggregated and disaggregated across assignment and with and without answer caching. In all four major settings, the error messages are included in the model. As expected, the batched version classified more questions correctly, and a larger percentage of the total questions. The numbers in parentheses indicate the percentage of total questions classified correctly, with 411 as the denominator, since there are 411 total questions.

Table 5.3 is similar to Table 5.2 with one major exception, the denominators used in calculating the percentages are based on repetitive questions instead of total questions. In the aggregated online setting, the denominator is 268, and in the disaggregated online setting, the denominator is 201. In the batched aggregated setting, the denominator is 329, and in the batched disaggregated setting the denominator is 273. As before, the batched version still classified more questions correctly by counts, but a smaller percentage of the repetitive questions. One major problem with the supervised paradigm from a pedagogical standpoint is that to be useful, an instructor must commit to recycling assignments year after year to leverage improvements over the online setting. The online approach by contrast gives the instructor more freedom to change and adapt assignments, while still preserving relatively good classification accuracy.

**Table 5.3. Counts and Percentages of Correctly Classified Repetitive Questions**

	<b>Online</b>	<b>Batched</b>
Aggregated Without Caching	98 (37%)	117 (36%)
Aggregated With Caching	104 (39%)	111 (34%)
Disaggregated Without Caching	114 (55%)	135 (49%)
Disaggregated With Caching	111 (50%)	126 (46%)

### 5.4.3 A Supervised Learning Approach

For comparison purposes, I ran a supervised machine learning analysis using decision trees with 10-fold cross validation in Weka [88] on the natural language and the processed compiler errors. As shown in Figure 5.2, the top level decision word was “pyramid” indicating a large number of questions about the pyramid assignment (assignment4 and assignment04). The final tree had 27 nodes and a size of 53. With a classification accuracy of 96/398 or 24%, it is actually less accurate than the same data run through the online learning framework, probably because the online learning framework can leverage recency. Slightly fewer questions were included in the supervised setting because questions with no terms were excluded to facilitate using existing software.

```

pyramid < 0.5
| font < 0.5
| | payment < 0.5
| | | extens < 0.5
| | | | cfsmissingimport < 0.5
| | | | | insertaft < 0.5
| | | | | | pointer < 0.5
| | | | | | | quarter < 0.5
| | | | | | | | amount < 0.5
| | | | | | | | | namedcompon < 0.5
| | | | | | | | | | start < 0.5
| | | | | | | | | | | document < 0.5
| | | | | | | | | | | | arrai < 0.5
| | | | | | | | | | | | | cfsmethodmismatch < 0.5
| | | | | | | | | | | | | | nothavebeeniniti < 0.5
| | | | | | | | | | | | | | | dot < 0.5
| | | | | | | | | | | | | | | | monthypay < 0.5
| | | | | | | | | | | | | | | | | part < 0.5
| | | | | | | | | | | | | | | | | | equal < 1.5
| | | | | | | | | | | | | | | | | | | test < 0.5: Positioning pyramids 2+3(12.0/212.0)
| | | | | | | | | | | | | | | | | | | | test >= 0.5: Test Cases(4.0/7.0)
| | | | | | | | | | | | | | | | | | | | equal >= 1.5: Gameword equals method(2.0/0.0)
| | | | | | | | | | | | | | | | | | | | part >= 0.5: Constructor stubs(2.0/0.0)
| | | | | | | | | | | | | | | | | | | | monthypay >= 0.5: Mortgage calculation(2.0/0.0)
| | | | | | | | | | | | | | | | | | | | dot >= 0.5: File extension extraction(2.0/0.0)
| | | | | | | | | | | | | | | | | | | | nothavebeeniniti >= 0.5: Variable initialization(2.0/0.0)
| | | | | | | | | | | | | | | | | | | | cfsmethodmismatch >= 0.5: Class and method mismatch(6.0/13.0)
| | | | | | | | | | | | | | | | | | | | arrai >= 0.5
| | | | | | | | | | | | | | | | | | | | | rectangl < 0.5
| | | | | | | | | | | | | | | | | | | | | file < 0.5
| | | | | | | | | | | | | | | | | | | | | | revers < 0.5: Array index out of bounds exception(3.0/8.0)
| | | | | | | | | | | | | | | | | | | | | | revers >= 0.5: Incompatible types(2.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | file >= 0.5: Unique files structure(3.0/2.0)
| | | | | | | | | | | | | | | | | | | | | | rectangl >= 0.5: findSmallest structure(3.0/1.0)
| | | | | | | | | | | | | | | | | | | | | | document >= 0.5: JavaDoc documentation generation(3.0/1.0)
| | | | | | | | | | | | | | | | | | | | | | start >= 0.5
| | | | | | | | | | | | | | | | | | | | | | assign < 0.5: get shortest path structure(3.0/9.0)
| | | | | | | | | | | | | | | | | | | | | | assign >= 0.5: Getting started(5.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | namedcompon >= 0.5: Named Component(4.0/2.0)
| | | | | | | | | | | | | | | | | | | | | | amount >= 0.5: Mortgage calculation(3.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | quarter >= 0.5: ChangeToDollars(3.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | pointer >= 0.5: null pointer exception(3.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | insertaft >= 0.5: Insert after node structure(5.0/1.0)
| | | | | | | | | | | | | | | | | | | | | | cfsmissingimport >= 0.5: Missing import statement(6.0/2.0)
| | | | | | | | | | | | | | | | | | | | | | extens >= 0.5: File extension extraction(6.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | payment >= 0.5
| | | | | | | | | | | | | | | | | | | | | | mortag < 0.5: Mortgage calculation(8.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | mortag >= 0.5: Test Cases(3.0/0.0)
| | | | | | | | | | | | | | | | | | | | | | font >= 0.5: Using fonts(10.0/1.0)
pyramid >= 0.5
| method < 1.0: Positioning pyramids 2+3(29.0/3.0)

```

**Figure 5.2. Decision Tree**

## **CHAPTER 6**

### **CHALLENGES IN COLLECTING DATA**

The data collection tool (software artifact) and the analysis framework are both contributions that are described in great detail in previous chapters of the dissertation. The data are also described in a previous chapter, but without much discussion of why it is an important contribution and why it is challenging to acquire the kind of data required. This chapter describes why it is difficult to do the kind of research described in the dissertation and explains in greater detail why the data are a valuable contribution in their own right. The final section speculates about how the data might be used in future research.

#### **6.1 Dearth of Existing Data**

Although many papers have been written both on the broader topic of tutorial dialog and on the narrower topic of automatically answering student questions, it is not at all clear that the results transfer to other domains, platforms, and educational settings.

##### **6.1.1 Single Domain Systems with Paid Subjects**

At least two of the longer lines of tutorial dialog research have focused on a single domain, and they have used paid subjects. The CIRCISM project [29] focused on



tutoring medical students on circulation, and the JavaTutor project [17] is focusing on tutoring students in Java. Both projects collect data by paying subjects to participate in tutorial dialogs, usually in a wizard-of-oz style study with the tutor and student in separate rooms. Such studies are very expensive to run because an expert must be paid to tutor the student and the student must be paid. Furthermore, recruiting students to participate can be expensive and difficult, and some psychology research suggests that paying subjects can alter results (e.g., [68]). While such a setting generates many more dialog turns than a traditional classroom or laboratory setting, it is not clear that the results of such studies generalize to more traditional and economical educational settings.

### **6.1.2 Tutorial Dialog Linked to Model Tracing Tutors**

Other longer lines of tutorial dialog research include the AutoTutor project (e.g. [84]) and the WHY-2 projects (e.g. [79]). Both of those have explored more than one domain, but the number of domains is still very small (approximately three). Most of their tutorial dialog appears to be tutor-initiated instead of student-initiated, and it appears that the tutoring prompts contain significant vocabulary to scaffold student responses. These systems also often depend on a relatively tight connection to a complicated tutoring system such as a model-tracing tutor, which are known to be expensive platforms in which to engineer educational content. As a result, their research is often based on only a few problems that might be covered in a week instead of being based on the spectrum of problems that would be covered over the course of a semester. The scalability of such a system over a larger curriculum is debatable at best.

### **6.1.3 Potential Problems with Processing Existing Data**

Data collection is full of trade-offs and design decisions, including grain size, raw feature set, and taxonomies for tagging. Much existing data on the questions that students ask is situated in extremely rich, but difficult to mine tutorial dialog, and at least some of the projects have not released their raw data to the larger research community. For example, the JavaTutor corpus consists of both free-form tutorial dialog with programming experts and keystroke data from beginning programmers. That corpus is not currently publicly available, but if it were, to be usable for a replication study on question classification, the questions and answers would have to be extracted from the dialog, probably manually. Then, the answers would have to be tagged according to the same taxonomy, again probably manually. Then, at least part of the data would have to be tagged again to establish interrater reliability. Only then would it be possible to attempt to replicate the automatic classification of questions.

### **6.1.4 General Problems with Extracting Questions from Tutorial Dialog**

The questions asked in a tutorial dialog may differ from the questions that students ask spontaneously in a classroom. The questions in tutorial dialog appear to suffer from a stream-of-consciousness problem that is typical of think-aloud protocols. Such a setting can create a social pressure to communicate, and any questions collected may not reflect a genuine information need. Questions from tutorial dialog often fail to reflect distinct points of difficulty in the learning process, making it difficult to isolate students' most challenging questions. In summary, the questions in tutorial dialog may

differ substantially from the questions that they ask spontaneously when they have a genuine information need.

## **6.2 Difficulties of Collecting Data**

### **6.2.1 The Quantity Problem**

Students do not ask questions very often. One estimate in the literature for elementary students is an average rate of “one question per hour” [35, 56]. This dissertation had a similar data collection rate for a much older student population. One on one tutoring and other settings in which the instructional staff prompts the student appears to increase the rate of question asking, but even with additional prompting students do not ask very many questions, and the quality of the questions is somewhat lower [35, 56]. Furthermore, as this dissertation has demonstrated, the majority of the questions are often asked by a few students, with the majority of students never asking any questions. The difficulties of studying the questions that students ask and generating scientifically valid results with meaningful measures of statistical significance have exacerbated the dearth of existing data.

### **6.2.2 The Quality Problem**

Not only do students ask questions infrequently, the questions that they ask are often poorly articulated and impossible to answer out of context. This makes automatically identifying and answering such questions difficult, and it also complicates research to accomplish these tasks automatically. For example, students might say “I’m

stuck.” Such a sentence clearly carries two implied questions, 1) “Did I do something wrong?” and 2) “What is the next step?”, but a computer program looking for simple, shallow features such as a question mark may fail to recognize that “I’m stuck” is even a kind of a question. Furthermore, a question such as “I’m stuck” requires some context to answer. At a minimum, the requirements are a model of an ideal solution, the student’s partial solution, and a method for aligning the two in order to identify deficiencies. In ill-defined domains, such as programming where there are multiple correct solutions, an additional step is necessary to choose the best solution for the current student.

### **6.2.3 Lack of Existing Tools**

The difficulty of collecting data is compounded by a lack of adequate existing tools. Little research funding exists to promote the creation of good tools, and most existing tools are hacked together compounds that rely on several existing pieces of software. For example, the Marmoset project captures snapshots of student code (e.g. [73]), Marmoset functions as a plugin in the Eclipse environment using CVS repositories. To use Marmoset, the end-user must install CVS, Eclipse, and Marmoset, three separate installations, and root access would probably also be necessary to deploy the software. The requirement of root access would make modifying the software difficult and induce an extra dependency in the software update cycle. Additionally, the software dependencies make it unsuitable for introductory classes that use another IDE such as BlueJ. At least Marmoset is an open source project which would facilitate obtaining the source code. Other projects, such as AutoTutor, are not open source, and the source code

is not available. Recent efforts have attempted to create an open source version of AutoTutor. Sometimes the tutoring system is available only as an executable, making it impossible to modify the functionality and run the desired experiment on a particular tutoring platform.

#### **6.2.4 Obtaining Approval**

One of the great difficulties in collecting data for any kind of human subjects research is simply obtaining the necessary approval. The Institutional Review Board (IRB) approved the experimental design. Collecting data in a traditional laboratory setting as part of traditional laboratory activities helped the study to qualify for an expedited review and also allowed me to avoid collecting consent forms, two additional obstacles that would have had to have been leaped with different protocols. I worked closely with both my advisor and his PhD student who were teaching the introductory course when I was designing the software to insure that it would have instructor approval. Although I needed instructor approval to deploy the software in a course, the instructor did not actually use the software much at all. Instead, the teaching assistants utilized the software. After an initial deployment or two it became obvious that the experimental protocol would need to be relaxed at least a little bit to meet their needs. For example, the original protocol required the TAs to tag the data and author answers and both of those requirements were removed. Also, the TAs were still allowed a high degree of interaction with students, including spoken dialog, that was not captured. Finally, the software needed to provide at least some benefits to students. The major benefit for

students was the ability to ask questions remotely, and there is some anecdotal evidence that it was effective for at least some students. By allowing all students in the class to use the software, ethical issues over access were avoided. To ensure that all students and TA's knew how to use the software, I ran training sessions as part of one of the required weekly labs at the beginning of both semesters in which the data was collected.

### **6.2.5 Authoring Answers and Tagging the Data**

One concern that the TAs had was that authoring answers to questions was time-consuming, and it was not practical when several students were waiting to answers for questions. Because I was not actually recycling answers, I told the TAs that they did not need to have the answer recorded, reasoning that I could author answers later when I was ready to deploy the system with automatic answers. The TAs also had a difficult time identifying when two questions were similar. Part of this was my fault; I did not explicitly train them or give them a specific taxonomy to be used when classifying similar questions. After researching the literature, I realized that compiler errors alone could be classified in many ways, and there were even more options for free form student questions. One key choice in developing a taxonomy is where to aggregate and where to disaggregate, and existing research does not agree on this matter. After spending many days studying student questions, I bailed on attempting to define a formal taxonomy, and I simply labeled each question with a word or short phrase describing how I would answer the question or describe the error, such as “capitalization” or “getNext method.”

Because I had tagged the questions in a somewhat subjective manner, I needed another human to tag at least some of the data so that I could calculate interrater reliability and ensure that the labels were capturing a real aspect of the data and not just my imagination. Because the tagging process involved matching a question to a previous question, tagging the data alone probably required several weeks of full-time human attention.

### **6.3 Requirements for a Software Tool to Collect Data**

This dissertation is based on a dataset of 411 questions. Although spontaneous student questions are rare, the cost of collecting the questions manually may have been less than the cost of creating a tool to collect the data. The chief advantage to developing a tool is that it will simplify future data collection efforts and future wider-scale deployments. This section explains the software requirements for such a tool and describes some of the difficulties in developing such a tool.

#### **6.3.1 Robust**

Data collection took place over more than 20 hours per week when TAs were on duty, and being available to support the software in person 20 hours a week was not realistic. That meant that the software needed to be robust enough that both TAs and students could use it without additional software support, and the software had to be available when I was not available. Because the software ran on the web and utilized the web-server, it was important that it be fairly stable and manage resources well. Early

versions of the software crashed the web server that was servicing online classes and that caused problems. Later versions of the software were subject to resource management reviews. Additionally, because several students may need to ask a question at once, the software needed to be tested by several students simultaneously. TAs helped with preliminary versions of multiuser testing, and I attended and led training sessions as part of a weekly laboratory at the beginning of each semester that provided additional opportunities to stress test the system for resource management.

### **6.3.2 Simple Interface**

One original goal of the system was to create software that could be used by a variety of users in a variety of domains. Because most educators and students cannot program, the interface could not require students to program or require teachers to program. The only operations that teachers or students have to be able to do to use the system are a limited amount of typing, a limited amount of file system browsing, and clicking a button on an interface. To accommodate advanced instructors who know how to program, the interface would accept the URL of a program or webpage with an applet. I also worked closely with several TAs to determine what features were useful and not useful for them.

A key functionality required for my research was the ability to transfer an entire folder of data easily. Students working on introductory programming projects often create several files in the same folder for a single assignment, and dependencies often exist between the files, making it necessary to collect all of them. While several options



exist for transferring a single file, no tools that I am aware of facilitate transferring an entire folder of data. The simplest existing option involved zipping a folder, transferring it, and then requiring it to be unzipped on the other side, a set of operations too laborious to expect of a student every time they needed to ask a question. I also needed a queue to track students and manage several students. While existing chat and personal messaging applications facilitate communication in a one-on-one or group setting, they do not generally include queuing functionality, and they could not be utilized in the very real-world setting of a typical introductory programming class where a half a dozen students all wish to ask a question and hope to receive an answer as quickly as possible because they are racing against the clock to finish their assignment before the deadline.

### **6.3.3 Modular**

A key design goal was to make the system modular and independent, and the system was at least partially successful in this goal. The analysis software has a pretty clean separation from the data collection software, and the analysis software can analyze the same dataset in several ways and compare them. Unfortunately, the data collection software could be more cleanly separated into separate modules, and the analysis software could also be more cleanly separated into separate modules. One problem with the TA interface is that it displays the students' code, but does not provide compiler error feedback or syntax highlighting. After serious thought, I have concluded that the VTA system is not meant to be a web-based interactive development environment (IDE), and the elements of the software that display student code and compiler error messages will

probably be factored out in future versions. One reason to try to make the system independent of other systems was to avoid issues in versioning. For example, I considered making the system a BlueJ plugin, but there are several versions of BlueJ, even just in the on-campus laboratory that students use at the University of Utah, and probably even more considering all of the personal laptop configurations. Different versions look for plugins in different locations, creating installation problems for a plugin system. Other software dependencies have potential for additional versioning problems, so the system was designed to avoid software dependencies to the extent that it is possible to do so. One major advantage to implementing the system in Java is that Java runs on Macintosh, Windows, and Linux platforms, increasing user choices.

## **6.4 A Brief Description of the Data and its Uses**

The dataset described in this dissertation consists of 411 spontaneous questions asked in a laboratory setting in an introductory computer science course. The majority of the questions were asked by only a few students in the course as shown in Chapter 3 along with other summary statistics. The remainder of this section describes two subsets of the data collected, the portion that will be publicly released and the portion that will remain proprietary for now along with an explanation of why not release it.

### **6.4.1 The Publicly Released Data**

One goal in publicly releasing as much data as possible is to facilitate replication experiments, and the data in the public release set has been carefully selected in order to

achieve that goal. The publicly released data include a set of 411 questions spontaneously asked by students while completing programming assignments. Specifically, the dataset includes the natural language of the student questions, the assignment, the timestamp, the original and the processed compiler errors used in the analysis, and the tags for the answer category. The vast majority of those questions were collected from students working in a laboratory setting as evidenced by the machine names. Additionally, the stopword list and symbol list are available to facilitate replication studies

#### **6.4.2 The Privately Retained Data**

In addition to the publicly released data, some data must be privately retained. For example, the identities of the students and the TAs cannot be released for ethical reasons. The source code is not being released because it would be difficult to anonymize it. (Anonymizing source code is probably an independent PhD project.) Although these data are not currently available to the public, I am still hoping to use at least some of them for future research projects such as clustering student code.

#### **6.4.3 Uses of the Data**

The data could potentially be used in at least two different kinds of experimental settings. One possibility is to use the data with existing analyses to close the gap between theoretical belief and empirical knowledge. For example, the analysis from a recent paper on the subjectivity of questions could be repeated with this data set to determine if

the same markers are valid for questions collected in a more traditional educational setting. The data could be used in future meta-analyses on tutorial dialog to discover commonalities and differences by comparing it with data from other tutorial dialog systems. Finally, the future will bring discoveries of new analysis techniques which may also be applied to this data. Having multiple data sets on which to test a theory or model decreases the risk of overfitting and broadens the set of claims that it is possible to make.

## **CHAPTER 7**

### **CONTRIBUTIONS, FUTURE WORK, AND CONCLUSIONS**

#### **7.1 Contributions**

##### **7.1.1 The Virtual Teaching Assistant System (Software Artifact)**

First, the dissertation describes a unique software artifact called the Virtual Teaching Assistant system that mediates the question-answering process between students and staff and facilitates logging and mining the relevant data. This software system captures the natural language in the questions that students ask as well as the source code snapshots and context such as the date and assignment that the student is working on. The artifact has been utilized by several dozen students, and it is a suitable platform for future research on the questions that novice programmers ask.

##### **7.1.2 A Set of Student Questions**

Second, the dissertation describes a dataset consisting of ecologically valid questions asked by students in an introductory programming class, including the dates and the assignments about which the questions were asked. This dataset enables investigations of patterns in question asking within a course. The dataset also contains source code and compiler errors. The data are stored in a combination of a database file system that are organized to facilitate mining the data. The ease of mining the data and

platform independence is a key distinguishing features of this dataset compared to many other datasets with source code snapshots. Also, the fact that it consists entirely of spontaneously asked student questions make it suitable for future studies comparing the language in various other tutorial dialog scenarios including human-computer dialog, human-human dialog, and other non-tutorial sets of questions and answers, such as corpora from corporations.

### **7.1.3 An Online Analysis Framework**

Third, the dissertation describes an analysis framework and an analysis showing that the additional context including the date and the assignment number can be leveraged to improve the classification of questions that students ask. An online learning framework is a viable alternative for an automatic question answering system, answering a similar number of questions as a batched or supervised setting when doing a similar analysis. Many modern intelligent tutoring systems require extensive knowledge engineering and/or they must be trained with data that has been harvested from a deployed system. The disadvantages to the former alternative are that the system designers must try to foresee every question that a student could ask, and the cost of engineering knowledge for the system is typically high. The disadvantage to the latter alternative is that the system does not benefit students as much in the year while data are being collected. The online learning framework reduces both of these disadvantages by waiting until a student has asked a question to engineer knowledge, and then potentially exploiting that knowledge immediately after it has been added to the system.

The central question of the dissertation is “Can domain knowledge and educational context improve classification of the questions that students ask in a novice programming class?” Using natural language and cosine similarity as a non-trivial baseline, I answer this question in the affirmative by improving accuracy by using domain knowledge (processed compiler error messages) and educational context (assignment number). I replicate previous results showing that answer caching can improve accuracy by 1-3% and extend previous work on answer caching by achieving similar improvement on a more difficult dataset with ecologically valid tutorial dialogue.

Using domain knowledge, answer caching, and educational context, the algorithm can classify between 23% and 56% of the questions. In the baseline condition with natural language and cosine similarity, the algorithm classifies 96 questions correctly or 35% of the repetitive questions and 23% the total questions. Using processed compiler error messages and answer caching, the algorithm classifies 104 questions or 25% of the total questions and 39% of the repetitive questions. When disaggregating by assignment, the algorithm classifies 113 questions correctly or 28% of total questions and 56% of repetitive questions. By exploiting strengths of both approaches, the algorithm can classify 119 questions or 29% of total questions and 36% of the repetitive questions.

The analysis shows that the natural language or the compiler errors alone are inadequate to classify student questions. Rather, it is the combination of those features plus other features such as temporal locality that improve classification accuracy. The dissertation shows that temporal locality is an important features in educational questions,

and that a least recently asked question is almost as likely to have the correct answer as a question with a high similarity score.

## 7.2 Future Work

The majority of the research described in this dissertation is focused on a particular research question, “Can domain knowledge and educational context improve the classification of student questions?” The dissertation answers that question in the affirmative for introductory computer science, but leaves many research questions unanswered.

For example, it is reasonable to suppose that if the VTA system were deployed in a mature course with a stable portfolio of instructional activities and assignments, the percentage of novel questions would eventually plateau, perhaps over the course of two or three years. A longitudinal study to determine where in time that plateau occurs in time would answer the questions “How long does it take to train the VTA system?” and “How many questions can it classify automatically when it is fully trained?” Furthermore, the extra data collected in such a setting might enable the system to leverage the benefits of both answer caching with error messages and disaggregation by assignment for an extra boost in accuracy. Deploying the system in other courses besides introductory computer science could help answer questions about how question asking behaviour varies across courses and domains, and also reveal whether or not there are differences in the number of repetitive questions asked.



In summary, this research raises many research questions about the questions that students ask. In the remainder of this section, I describe relevant areas of future work that pertain less directly to the questions that students ask, but are important for understanding their help seeking behaviour and opportunities for intelligent interventions to assist them.

### **7.2.1 Metacognition**

A good teaching assistant can provide answers to the questions that students are asking. A great teaching assistant anticipates the questions that a student should be asking and provides an answer even when the student fails to provide a prompt. Such a teacher can tutor a student not only in cognitive skills but also in meta-cognitive skills. Early work on metacognition has defined two categories of novice programming behavior. “Stoppers” are “unwilling to explore the problem further”, while “movers...[try] to repair code in ways that... will not work.”[59]. Chad Lane’s PhD thesis [53] showed that novice programmers using the Java language also fail to realize that they need help. His tagged data could be utilized in a machine-learning experiment to train a classifier that can distinguish between students who are productively engaged and on task and students who are in need of assistance from a teaching assistant, human or virtual. Jaime Spaaco has also examined novice programming and produced a dataset that could be utilized in metacognitive work for novice programmers [73]. Additionally, the dataset used for my own thesis could be analyzed to produce a set of indicators that students need help. Other related work in metacognition beyond the programming

domain includes papers by Roll [65], Alevan [3], and Rebolledo-Mendez [62, 63].

Collectively, these papers suggest that students in several domains are in need of metacognitive tutoring and research in this area is likely to be broadly applicable. However, many people in the Intelligent Tutoring Systems and Artificial Intelligence in Education communities consider learning gains the gold standard as demonstrated by an award-winning paper [80], and linking metacognitive tutoring to learning gains remains an elusive goal [65].

### **7.2.2 Intervention Evaluation**

There are several reasons that introductory computer science students may need an intervention. Students enter Computer Science 1 with varying levels of expertise; many need remedial help just to get caught up with their peers who have had more programming experience. Some students need multiple encounters with a skill to learn it well, but many assignments involve a single encounter with a skill or a single application of a skill. Students may need additional instruction to understand the motivation behind a skill. Finally, some students may benefit from new forms of user-targeted instruction (e.g., animation clips) that computers utilize to improve engagement in education.

Unfortunately, not all interventions were created equally, so there is a need to evaluate interventions. For example, previous research has shown that in one group of interventions, a couple of the interventions were substantially less effective than the majority [39]. Inspired by the randomized trials and embedded experiments of the Project LISTEN Reading Tutor [55], the architecture of the VTA system could easily be

modified to allow educational scientists to collect or create a number of interventions and measure their relative efficacy in answering student questions. Once interventions have been evaluated, the less effective interventions can be taken out of the pool and replaced with new interventions or interventions that are known to be more effective.

Extensions may include a general framework for organizing interventions and describing them based on several criteria, such as expected prior knowledge, ease of use, student satisfaction, student learning gains, and others. In some ways, this part of the system could work like traditional recommender systems used on websites for travel, movies, and other hobbies. An external mechanism for interventions that are text or URL based is expected to be much better than more traditional internal interventions because intervention designers will not need to be intimately familiar with the tutoring system, and they will be able to create interventions with the web-based tool of their choice on the development cycle of their choice. Finally, the approach outlined by the dissertation separates the diagnosis of the student's problem from the selection of intervention, so that intervention designers do not have to worry about breaking the system when they create interventions.

### **7.2.3 Missing Sigma**

One of the original justifications for much of the tutorial dialog work done in the last two decades was that perhaps tutorial dialog could close the gap (the so-called missing sigma problem) between traditional model tracing tutors and expert human tutors. Previous work has shown that a tutoring system that has been augmented with

natural language tutoring is more effective than the same system without natural language tutoring. However, based on effect sizes alone, model tracing tutors (e.g., Andes or the Algebra Tutor with learning gains of 1.2 [80]) have produced higher learning gains than natural language tutoring (e.g., AutoTutor with learning gains of 0.9 [36]). To further add to the confusion, recently VanLehn has argued that step tutoring (or model tracing tutoring) is equivalent to natural [language] tutoring, and that both are more effective than answer-based tutoring [77]. Although making comparisons between the two approaches can be thought-provoking, neither one has fully matured, and trying to definitively evaluate them is roughly comparable to predicting which of two third graders is going to be a taller adult based simply on their height in the third grade.

Modern tutoring systems are not bad, but they still make many silly mistakes compared to an expert human tutor. For example, they allow students to game the system and extract the answer without learning [11], and they frequently use sub-optimal strategies [39]. Their dialog classification skills are weak, and they don't process natural language as well as humans do. On the other hand, the computer has an amazing capacity to retrieve facts and execute algorithms consistently, and in these two areas computers are substantially better than humans. More research is needed to better understand how to leverage the strengths of the computer and compensate for its weaknesses specifically in the area of tutoring.

#### **7.2.4 Support for Peer Tutoring, Computer-Based Collaboration**

As computing has become more pervasive and computation has extended beyond the individual desktop, interest has increased in computer supported collaborative learning and peer tutoring. Early research in this growing field appears promising. Future research may include designing algorithms to assess student progress on a problem and pairing the student asking the question with another student who is likely to be able to help the student, perhaps by completing a peer tutoring script together. Additional support may even identify and adequately recognize or even reward the most helpful student peers. Another approach may allow the system to automatically identify groups of students who have the same concern, allowing the TA to teach the entire group at the same time.

#### **7.2.5 Usability Issues**

A number of usability questions on both the teacher and the student side must be answered before an automatic question answering system can be deployed in a classroom setting. On the teacher side, research is needed to determine if training the teacher can improve and speed up the necessary process of tagging an initial set of questions to train the classifier. Additional research may investigate if automatic measures such as question similarity can be exploited to facilitate that task. Once automated interventions are added, the teacher will need to determine if the student still needs human help because the system classified the question incorrectly or because the automated intervention was ineffective. On the student side, studies should investigate whether or

not a drop-down menu of frequently asked questions can help students who articulate their questions poorly. Additionally, students may not accept automated answers to their questions, especially if they know that a human TA is on duty and available.

### **7.2.6 Teaching and Grading Support**

Automated grading has been a popular research topic at SIGCSE for several years. One set of automatic grading tools has already been used in cheating detection [28]. Automated grading may allow students to receive real-time evaluations of their work and reduce the amount of time human TA's are required to spend grading course materials. The likelihood that such an automated tool could provide impartial, unbiased, fair evaluations to all students increases the appeal of automated grading. Another appealing possibility with automated grading is a teacher tool that can automatically highlight aspects of the assignment that students found most troubling, so that teachers can spend extra time reviewing these matters with the current class and provide extra teaching to future classes.

### **7.2.7 Classification Schemes for Questions that Novice Programmers Ask**

Given a set of categories, classifying questions appears to be relatively straightforward for humans. However, no widely accepted set of categories or taxonomy exists for the questions that novice programmers ask. Previous work has suggested either 42, 88, or 226 different categories for compiler errors [2, 42, 76], and compiler errors only account for less than half of the questions in the data set utilized for this dissertation.

Those papers are simply trying to classify compiler errors based on the compiler error message, not the underlying misconception the student has expressed. Furthermore, a single piece of code may have multiple issues. Consider a typical problem asking a student to write code to determine the max in an array of integers. The student may attempt the problem with partially complete code as follows:

```
public static int findMax(int [] a){
    int max=0;
    while(){
        if( <max){
```

Such a piece of code suggests that the student perhaps began the problem, reached a partial impasse while creating the loop, and a full impasse while working on the conditional. Ideally, the student would have requested help at the point of the partial impasse, but students frequently appear to wait until they have reached a full impasse before requesting help. The resulting code has multiple problems. From the perspective of the compiler, there is an “illegal start of expression.” From a syntactic perspective, the curly braces are not balanced. From a planning perspective, the student does not appear to know how to construct a loop. Furthermore, these various errors do not lend themselves to placement in a hierarchy. “Illegal start of expression” problems can occur in many kinds of methods.

To further complicate question classification, many students seem to avoid asking directly about their code, generating distracting natural language garbage. In this research, such questions would have probably been assigned a label such as “findMax structure,” but that is clearly a composite label that encompasses a broad range of

problems. Work on classification schemes that allow free-response student-input to be assigned multiple, more-fine-grained designations would be applicable for question classification as well as other problems. That research will probably also require work on partial parsing, and other approaches for handling poorly formed student input that cannot be parsed with readily available tools.

Dealing with student input that contains more than one error remains a difficult open research problem. The research described in this dissertation partially bypasses that problem by allowing “complex problems” composed of multiple smaller problems and focusing on the most urgent problem, e.g., the problem causing the first compiler error.

### **7.2.8 Student Retention and Efficiency**

The research to be completed for the thesis must be completed in a relatively short time period that makes longitudinal studies unrealistic. However, as automated answering systems and other forms of intelligent tutoring become a more integral part of the college course experience, it will be important to study the effect that these tools have on long-term student retention and student efficiency. Student retention and student programming efficiency may be strongly linked in introductory computer science where many students who drop out complain about long hours, often long hours of staring at a computer screen with little productivity. An extended version of the thesis work could be utilized in such a long-term study to determine if such automated tools help or hurt students in achieving short-term efficiency and proficiency and long-term success.



### 7.3 Conclusions

My classification methods work over half the time for repetitive student-generated questions, assuming that the questions can be separated by assignment. Thus, my methods would work particularly well in a course in which the same assignments are used over and over, and the long-term goal of using a classification-based approach to automatically answer questions would be especially valuable in a course in which students have low access to course staff. These two conditions are typical of online classes, which represent a fast growing segment of courses in higher education.

The future of education is a large puzzle with many pieces still in development. Video streaming technology makes it possible to broadcast lectures to a large audience, and the internet has the potential to reduce or eliminate the cost of textbooks. However, to encourage students to engage in large classes and online classes, we need more automatic tools to process their input in a scalable and timely manner, and these kinds of tools represent major missing pieces. This dissertation has helped to define one of those missing pieces, namely the elements of an online system to automatically answer student questions by recycling answers to previous questions.

## REFERENCES

1. tf-idf *Wikipedia*, 2009.
2. Ahmadzadeh, M., Elliman, D. and Higgins, C. An analysis of patterns of debugging among novice computer science students *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ACM, Caparica, Portugal, 2005.
3. Aleven, V., McLaren, B., Roll, I. and Koedinger, K., Toward tutoring help seeking Applying cognitive modeling to meta-cognitive skills. in *Intelligent Tutoring Systems*, (Maceio Brazil, 2004), Springer, 227-239.
4. Aleven, V., Ogan, A., Popescu, O., Torrey, C. and Koedinger, K. Evaluating the Effectiveness of a Tutorial Dialogue System for Self-Explanation. in *Intelligent Tutoring Systems*, 2004, 443-454.
5. Aleven, V., Popescu, O. and Koedinger, K. Pilot-Testing a Tutorial Dialogue System That Supports Self-Explanation. in *Intelligent Tutoring Systems : 6th International Conference, ITS 2002, Biarritz, France and San Sebastian, Spain, June 2-7, 2002. Proceedings*, 2002, 531-540.
6. Allen, I.E. and Seaman, J. Online Nation, 2007.
7. Anderson, J.R., Corbett, A.T., Koedinger, K.R. and Pelletier, R. Cognitive Tutors: Lessons Learned. *Journal of the Learning Sciences*, 4 (2). 167 - 207.
8. Anderson, J.R. and Reiser, B.J. The LISP tutor *Byte*, 1985, 159-175.
9. Anthony, L., Corbett, A.T., Wagner, A.Z., Stevens, S.M. and Koedinger, K.R., Student Question-Asking Patterns in an Intelligent Algebra Tutor. in *Intelligent Tutoring Systems*, (Maceio, Brazil, 2004), Springer, 455-467.
10. Baffes, P.T. and Mooney, R.J., A Novel Application of Theory Refinement to Student Modeling. in *American Association for Artificial Intelligence*, (Portland, Oregon, 1996), 403-408.
11. Baker, R.S., Corbett, A.T. and Koedinger, K.R., Detecting Student Misuse of Intelligent Tutoring Systems. in *Intelligent Tutoring Systems*, (Maceio, Brazil, 2004), Springer, 531-540.

12. Ball, G., Ling, D., Kurlander, D., Miller, J., Pugh, D., Skelly, T., Stankosky, A., Thiel, D., Dantzich, M.V. and Wax, T. Lifelike computer characters: the persona project at Microsoft. in *Software agents*, MIT Press, 1997, 191-222.
13. Baum, S., Brodigan, D., Ma, J. and Steele, P. Trends in College Pricing, 2007.
14. Belkin, N.J., Kelly, D., Kim, G., Kim, J.Y., Lee, H.J., Muresan, G., Tang, M.C., Yuan, X.J. and Cool, C. Query length in interactive information retrieval *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, Toronto, Canada, 2003.
15. Berger, A. and Mittal, V.O. Query-relevant summarization using FAQs *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, Association for Computational Linguistics, Hong Kong, 2000.
16. Bloom, B.S. The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher*, 13 (6). 4-16.
17. Boyer, K.E., Dwight, A.A., Fondren, R.T., Vouk, M.A. and Lester, J.C. A development environment for distributed synchronous collaborative programming *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ACM, Madrid, Spain, 2008.
18. Bradford, R.B. An empirical study of required dimensionality for large-scale latent semantic indexing applications *Proceeding of the 17th ACM conference on Information and knowledge management*, ACM, Napa Valley, California, USA, 2008.
19. Brin, S. and Page, L., The anatomy of a large-scale hypertextual Web search engine. in *Seventh International World Wide Web Conference*, (1998), 107-117.
20. Brown, J.S. and Burton, R.R. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2. 155-191.
21. Brown, J.S., Burton, R.R. and de Kleer, J. Pedagogical, natural language, and knowledge engineering techniques in SOPHIE I,II,and III. in Sleeman, D.H. and Brown, J.S. eds. *Intelligent Tutoring Systems*, Academic Press, London, 1982.
22. Brown, J.S. and VanLehn, K. Repair theory: a generative theory of bugs in procedural skills. *Cognitive Science*, 4. 379-426.
23. Burton, R.R., Diagnosing bugs in a simple procedural skill. in *Intelligent Tutoring Systems*, (1982), Academic Press.

24. Burton, R.R. and Brown, J.S. Toward a natural language capability for computer-assisted instruction. in O'Neil, H. ed. *Procedures for Instructional Systems Development*, Academic Press, New York, 1979.
25. Callan, J.P. Passage-level evidence in document retrieval *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, Springer-Verlag New York, Inc., Dublin, Ireland, 1994.
26. Carbonell, J.R. Mixed-Initiative Man-Computer Instructional Dialogues, Massachusetts Institute of Technology, Cambridge, MA, 1970.
27. Dang, H.T., Lin, J. and Kelly, D., Overview of the TREC 2006 Question Answering Track. in *Text REtrieval Conference*, (2006), 99-117.
28. Engels, S., Lakshmanan, V. and Craig, M., Plagiarism Detection Using Feature-Based Neural Networks. in *Special Interest Group in Computer Science Education*, (Covington, Kentucky, 2007), ACM, 34-38.
29. Evens, M. and Michael, J. *One-on-One Tutoring by Humans and Computers*. Lawrence Erlbaum Associates, 2005.
30. Flowers, T., Carver, C.A. and Jackson, J., Empowering Students and Building Confidence in Novice Programmers through Gauntlet. in *Frontiers in Education*, (2004).
31. Franklin, B.
32. Garner, S., Haden, P. and Robins, A., My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. in *Australian Computing Education Conference*, (Newcastle, Australia, 2005), Australian Research and Practice in Information Technology
33. Gorin, A.L., Riccardi, G. and Wright, J.H. How may I help you? *Speech Communication*, 23. 113-127.
34. Graesser, A. and Person, N. Question Asking During Tutoring. *American Educational Research Journal*, 31. 104-137.
35. Graesser, A.C., Person, N.K. and Huber, J. Mechanisms that Generate Questions. in Lauer, T.W., Peacock, E. and Graesser, A.C. eds. *Questions and Information Systems*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1992, 167-188.
36. Graesser, A.C., VanLehn, K., Rosé, C.P., Jordan, P.W. and Harter, D. Intelligent Tutoring Systems with Conversational Dialogue *AI Magazine*, 2001, 39-50.

37. Graesser, A.C., Wiemer-Hastings, P., Wiemer-Hastings, K., Harter, D. and Person, N. Using Latent Semantic Analysis to Evaluate the Contributions of Students in AutoTutor. *Interactive Learning Environments*, 8 (2). 129-147.
38. Hammond, K., Burke, R., Martin, C. and Lytinen, S., FAQ finder: a case-based approach to knowledge navigation. in *Artificial Intelligence for Applications*, (Los Angeles, CA, 1995), 80-86.
39. Heiner, C., Beck, J. and Mostow, J., Improving the Help Selection Policy in a Reading Tutor that Listens in *International Conference on Computer Aided Language Learning*, (Venice, Italy, 2004).
40. Hovemeyer, D. and Pugh, W. Finding bugs is easy *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ACM, Vancouver, BC, CANADA, 2004.
41. Jackson, J., Cobb, M. and Carver, C., Identifying Top Java Errors for Novice Programmers. in *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, (2005), T4C-24-T24C-27.
42. Jadud, M.C. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, 15 (1). 25 - 40.
43. Jadud, M.C. Methods and tools for exploring novice compilation behaviour *Proceedings of the 2006 international workshop on Computing education research*, ACM, Canterbury, United Kingdom, 2006.
44. Jansen, B.J., Spink, A. and Saracevic, T. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing & Management*, 36 (2). 207-227.
45. Jensen, P. Hybrid Automated Fault Localization in Programs written by Novice Programmers *School of Computing*, University of Utah, Salt Lake City, 2007, 175.
46. Johnson, L. PROUST *Byte*, 1985, 179-192.
47. Kaszkiel, M. and Zobel, J. Passage retrieval revisited *Proceedings of the 20th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, Philadelphia, Pennsylvania, United States, 1997.
48. Kim, H. and Seo, J. High-performance FAQ retrieval using an automatic clustering method of query logs. *Information Processing & Management*, 42 (3). 650-661.

49. Kim, J., Shaw, E., Chern, G. and Herbert, R., Novel tools for assessing student discussions: Modeling threads and participant roles using speech act and course topic analysis. in *Artificial Intelligence in Education*, (Los Angeles, 2007), IOS Press.
50. Kim, J., Shaw, E., Ravi, S., Tavano, E., Arromratana, A. and Sarda, P. Scaffolding On-Line Discussions with Past Discussions: An Analysis and Pilot Study of PedaBot. in *Intelligent Tutoring Systems*, 2008, 343-352.
51. Koedinger, K.R., Anderson, J.R., Hadley, W.H. and Mark, M.A. Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8. 30-43.
52. Landauer, T.K., Foltz, P.W. and Laham, D. An Introduction to Latent Semantic Analysis. *Discourse Processes*, 25 (2/3). 259-284.
53. Lane, H.C. Natural Language Programming and the Novice Programmer *Computer Science*, University of Pittsburgh, Pittsburgh, Pennsylvania, 2004, 193.
54. Manning, C.D. and Schütze, H. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.
55. Mostow, J., Beck, J.E., Bey, J., Cunneo, A., Sison, J. and Tobin, B., An Embedded Experiment to Evaluate the Effectiveness of Vocabulary Previews in an Automated Reading Tutor. in *Society of Scientific Studies of Reading*, (2003).
56. Neber, H. Training epistemic questioning behavior of elementary students. *Verhaltenstraining*. 360-374.
57. Neches, R., Swartout, W.R. and Moore, J.D. Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development *IEEE Transactions on Software Engineering*, 1985, 1337-1351.
58. Pasca, M.A. and Harabagiu, S.M. High performance question/answering *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, New Orleans, Louisiana, United States, 2001.
59. Perkins, D.N., Hancock, C., Hobbs, R., Martin, F. and Simmons, R. Conditions of Learning in Novice Programmers. in *Elliot Soloway, James C. Spohrer*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 261-279.
60. Pon-Barry, H. In Search of Bloom's Missing Sigma: Adding the conversational intelligence of human tutors to an intelligent tutoring system *Computer Science*, Stanford, Palo Alto, 2004, 45.

61. Porter, M.F. An algorithm for suffix stripping. *Program*, 14. 130-137.
62. Rebolledo-Mendez, G., Boulay, B.D. and Luckin, R., "Be bold and take a challenge": Could motivational strategies improve help-seeking? in *Artificial Intelligence in Education*, (Amsterdam, The Netherlands, 2005), IOS Press, 459-466.
63. Rebolledo-Mendez, G., Boulay, B.d. and Luckin, R., Motivating the Learner: An Empirical Evaluation. in *Intelligent Tutoring Systems*, (Jhongli, Taiwan, 2006), Springer Verlag, 545-554.
64. Robins, A., Haden, P. and Garner, S., Problem Distributions in a CS1 Course. in *Australian Computing Education Conference*, (Hobart, Tasmania, 2005), Conferences in Research in Practice in Information Technology.
65. Roll, I., Alevan, V., McLaren, B., Ryu, E., Baker, R.S.J.d. and Koedinger, K.R., The Help Tutor: Does Metacognitive Feedback Improve Students' Help Seeking Actions, Skills, and Learning? in *Intelligent Tutoring Systems*, (Jhongli, Taiwan, 2006), Springer Verlag, 360-369.
66. Rose, C.P., Jordan, P., Ringenberg, M., Siler, S., VanLehn, K. and Weinstein, A., Interactive Conceptual Tutoring in Atlas-Andes. in *Artificial Intelligence in Education*, (2001), IOS Press, 256-266.
67. Rosé, C.P., Torrey, C., Alevan, V., Wu, A.R.C. and Forbus, K. CycleTalk: Toward a Dialogue Agent That Guides Design with an Articulate Simulator. in *Intelligent Tutoring Systems*, 2004, 401-411.
68. Rush, M.C., Phillips, J.S. and Panek, P.E. Subject recruitment bias: The paid volunteer subject. . *Perceptual and Motor Skills*, 47 (2). 443-449.
69. Salton, G., Allan, J. and Buckley, C. Approaches to passage retrieval in full text information systems *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, Pittsburgh, Pennsylvania, United States, 1993.
70. Sneiders, E., Automated FAQ Answering: Continued Experience with Shallow Language Understanding. in *AAAI Fall Symposium*, (1999).
71. Soloway, E., Rubin, E., Woolf, B.P., Bonar, J. and Johnson, L. MENO-II: an AI-based programming tutor. *Journal of Computer-Based Instruction*, 10 (1). 20-34.
72. Song, W., Feng, M., Gu, N. and Wenyin, L., Question Similarity Calculation for FAQ Answering. in *Semantics, Knowledge and Grid, Third International Conference on*, (2007), 298-301.

73. Spacco, J., Hovemeyer, D. and Pugh, W., An eclipse-based course project snapshot and submission system. in *3rd Eclipse Technology Exchange Workshop* (Vancouver, BC, 2004).
74. Spohrer, J.C., Soloway, E. and Pope, E. A Goal/Plan Analysis of Buggy Pascal Programs. in Soloway, E. and Spohrer, J.C. eds. *Studying the Novice Programmer*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 355-399.
75. Stevens, A.L. and Collins, A., The goal structure of a Socratic tutor. in *the National ACM Conference*, (Seattle, WA, 1977), Association for Computing Machinery, 256-263.
76. Thompson, S.M. An Exploratory Study of Novice Programming Experiences and Errors, University of Victoria, 2006, 153.
77. VanLehn, K. The Interaction Plateau: Answer-Based Tutoring < Step-Based Tutoring = Natural Tutoring. in *Intelligent Tutoring Systems*, 2008, 7-7.
78. VanLehn, K. *Mind Bugs*. MIT Press, Cambridge, MA, 1990.
79. VanLehn, K., Jordan, P.W., Rose, C.P., Bhembe, D., Bottner, M., Gaydos, A., Makatchev, M., Pappuswamy, U., Ringenberg, M., Roque, A., Siler, S. and Srivastava, R., The Architecture of Why2-Atlas: A Coach for Qualitative Physics Essay Writing. in *Intelligent Tutoring Systems*, (Biarritz, France and San Sebastian, Spain, 2002), Springer, 158-167.
80. VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A. and Wintersgill, M., The Andes physics tutoring system: Five years of evaluations. in *Artificial Intelligence in Education*, (Amsterdam, Netherlands, 2005), IOS Press, 678-685.
81. Voorhees, E.M., Overview of the TREC-9 Question Answering Track. in *Text REtrieval Conference (TREC)*, (Gaithersburg, MD, 2000).
82. Voorhees, E.M., Overview of the TREC 2002 Question Answering Track. in *Text REtrieval Conference (TREC)*, (Gaithersburg, MD, 2002).
83. Wenger, E. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*. Morgan Kauffman Publishers, Los Altos, CA, 1987.
84. Wiemer-Hastings, P., Graesser, A., Harter, D. and Tutoring Research, G. The Foundations and Architecture of Autotutor. in *Intelligent Tutoring Systems*, 1998, 334-343.



85. Wiemer-Hastings, P., Wiemer-Hastings, K. and Graesser, A., How Latent is Latent Semantic Analysis? in *Proceedings of the 16th International Joint Congress on Artificial Intelligence*, (San Francisco, 1999), Morgan Kaufmann, 932–937.
86. Wiemer-Hastings, P., Wiemer-Hastings, K. and Graesser, A.C., Approximate natural language understanding for an intelligent tutor. in *12th International Florida Artificial Intelligence Research Conference*, (1999), AAAI Press, 172-176.
87. Wiemer-Hastings, P., Wiemer-Hastings, K. and Graesser, A.C., Improving an intelligent tutor's comprehension of students with Latent Semantic Analysis. in *Artificial Intelligence in Education*, (1999).
88. Witten, I.H. and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*.
89. Woolf, B.P. *Context-dependent Planning in a Machine Tutor* *University of Massachusetts*, University of Massachusetts, Amherst, Amherst, MA, 1984.