Experiments with Word Embeddings for Sequential Questioning

by

Emma McDonald

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

As a student learns to program, there will be gaps in the student's knowledge that must be addressed for the student to gain a full understanding of the material. A student's answer to a single question may provide some insight into the student's level of understanding. However, a well-chosen sequence of questions might more accurately identify any misunderstandings. For example, the popular 20 Questions game relies on a sequence of well-chosen questions for one player to guess what another player is thinking.

Inspired by the 20 Questions game, we suggest a method to select the next question in a sequence to identify gaps in a student's understanding. We model introductory computing science terms with word embeddings trained from a collection of Python course notes and textbooks. We also introduce a test suite of computing science concepts. Each of the 17 tests is an algebraic equation and each term in the equation is represented by one of our word embeddings. Thus, a test can be evaluated to produce a result that corresponds to another word embedding in the model.

The test suite represents a collection of concepts and skills that an introductory computing science student must learn. We demonstrate that we can represent a computing science concept by adding relevant substituent concepts and removing irrelevant concepts. We then posit that this ability can be used to diagnose the gap in understanding and recommend a relevant next question based on a student's answers so far.

Acknowledgements

There are many people to who I am extremely grateful for having helped me reach this milestone. In particular, I would like to mention the following.

My love and thanks to my family, always. My immeasurable gratitude to my supervisor, Dr Paul Lu, whose mentorship and patience have truly meant the world to me. My heartfelt gratitude to Dr Duane Szafron, whose kindness I will not forget. Thank you to Dr Lu and Dr Szafron for changing my life and making the dream of graduate school possible for me.

Thank you to Dr Carrie Demmans Epp for invaluable mentorship and advice. Thank you to my committee and to Dr Denilson Barbosa. Thank you to the friends, colleagues, and mentors who have provided feedback, support, and encouragement over the years. And thank you to Jacqueline Smith for everything from the beginning.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Imagine the scenario of an undergraduate university student enrolled in an introductory computing science class. For many students, this type of class is their first exposure to computing science and a corresponding new way of thinking. Students will make mistakes and ask questions, a time-honoured part of education. Imagine the student attending the instructor's office hours and asking questions. Using the student's questions, the instructor can determine what the student is stuck on. If need be, the instructor can ask more questions, to differentiate areas the student does and does not understand. For example, the student may ask about functions and demonstrate an inability to differentiate between function arguments and function parameters. These terms are carefully defined in the class material. The instructor has identified a gap in the student's knowledge.

The natural question is how to diagnose that gap, especially without the involvement of an instructor. For example, if a student is studying at home, the student may realize that they do not understand a concept. However, the root of the student's misunderstanding may not be easy for the student to self-diagnose.

For example, consider a failed loaf of bread. An ideal loaf of bread made with yeast will have a nice texture consisting of many small air pockets. The baker of the loaf must activate the yeast, combine the ingredients, knead the dough, let the dough rise, shape the loaf, rise the dough one more time, and then bake the loaf. If the bread does not turn out as expected, there are many clues that can indicate what went wrong. However, the answer is not necessarily straightforward, especially for an inexperienced baker. Even if one key problem is identified — for example, if the loaf did not rise — the solution still might not be straightforward. There are many possible culprits for a loaf of bread that did not rise: the yeast was expired; the yeast was not activated properly; the rising period was too short; the bread was overworked during the kneading step; etc. There are many further questions that must be asked to capture the exact issue at the correct layer of the problem.

A parallel with this idea of layered knowledge is the game of 20 Questions. The simplest version of 20 Questions is played with two players. One player secretly thinks of a concept. The concept could be a person, place, thing, or abstract idea. The other player then asks up to 20 questions that can be answered with a yes or a no, and uses the answers to these questions to guess the concept.

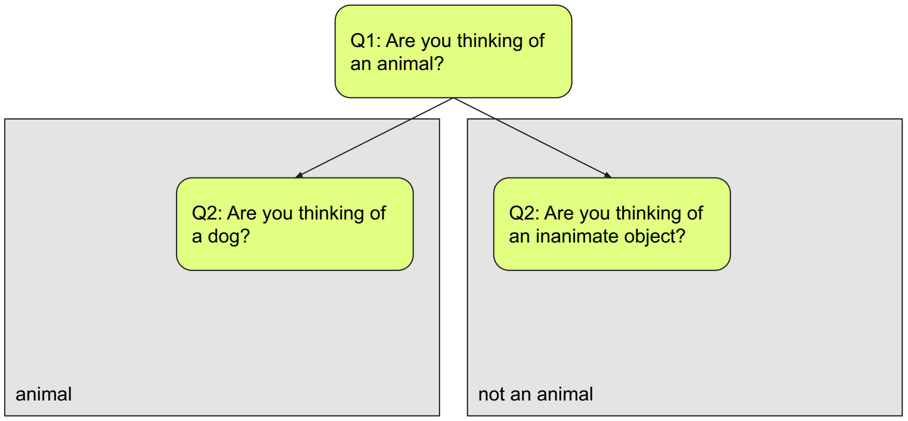
There are various adaptations of 20 Questions, but the overall structure is that one player thinks of a concept and the other player attempts to guess the concept by asking a sequence of questions. The key point of the game is that the questions form a sequence and the person asking questions uses the knowledge they gain from each answer to inform the next question.

To examine this idea of sequential questioning, consider a possible sequence of questions in a 20 Questions game, shown in Figure 1.1. The first question is:

Q1: Are you thinking of an animal?

A: No.

Because the answer to the first question is no, the concept lays in the **not an animal** category. It does not make sense to next ask **"Are you thinking of a dog?"**, because the entire category of animals has already been ruled out. Therefore, the person asking the questions knows not to ask any more animal-related questions. An entire block of possible questions has been ruled out and a piece of information has been acquired. That the concept is not an animal can be used to inform the next



Q1: Are you thinking of an animal?
Q2: Are you thinking of a dog?
Q2: Are you thinking of an inanimate object?
animal
not an animal

Figure 1.1: Example of a sequence of questions in a 20 Questions game. The answer to the first question, **Q1**, provides information that can be used to inform the second question, **Q2**. If the player answers yes to **Q1**, then **Q2** should be about something in the animal category.

question. For example, the second question could be:

Q2: Are you thinking of an inanimate object?

A: Yes.

Now there are two pieces of information: the concept is not an animal and the concept is an inanimate object. These pieces of information can be used to narrow down the possibilities for the third question. The third question can now be selected to help identify the inanimate object. In this way, the next question follows from the previous question.

Returning to the bread example, the baker may ask a sequence of questions to determine why the loaf of bread did not rise. For example, the baker may ask **"Was the yeast foamy and activated when I added it to the dough mixture?"** If the answer is no, this rules out that the yeast was dead and hence rules out a series of other questions, such as **"What is the expiration date on the yeast package?"** and **"When did I buy this yeast?"** Ruling out one section of questions narrows down the list of questions the baker should continue asking. Now they can ask **"How long was the first rising period?"** If the answer is less than an hour, then the baker now has a new line of questioning to follow, since it seems the rising period might not have been sufficient. For example, the baker can ask **"Was the dough doubled in size when the rising period was over?"**

This idea of a sequence of questions, where the answer to each question informs the next question to be asked, is similar to how an instructor attempts to diagnose a student's misunderstanding. Each answer the student gives provides insight into what the student does understand and what the student does not understand. Using this information, the instructor can select a next question that will garner a new piece of useful information.

However, in the absence of an instructor, it can be difficult for a student to ask their own self this sequence of leading questions. Originally, we thought to diagnose the gap automatically via hard-coded questions in the style of the popular 20 Questions game. However, this fell short in an attempt to automatically compute what the gap may be. The next natural question is how to learn the gap.

By using a corpus of computing science concepts, we can learn a word embedding for every term that a student must learn. The word embeddings represent the words in the corpus [21] and hence the relationships between the word embeddings may be used to represent the relationships between the computing science concepts in which we are interested. Combining the word embeddings with algebraic operators can reveal the way terms relate to each other [19]. The results of particular queries can show how multiple concepts relate. Hence, the word embedding model captures information about the terms that appear in introductory Python material.

We also define a test suite, where each test represents a concept or skill in computing science and introductory Python programming that a student needs to learn. The result of a test can be used to select the next question for a student to answer, based on the answers they have given so far.

For example, consider the following sequence of questions.

Q1: Is there a Python programming construct that can be used to remove non-adjacent duplicate code blocks?

Q2: Is it possible to define the value that a function can accept?

Q3: Is the state *argument* part of the function definition syntax diagram?

These questions are about, respectively, functions, parameters, and arguments. Based on the student's answers, we want to suggest a next question. If the student answers the first two questions correctly and the third question incorrectly, we can suggest that the student understands something about functions and parameters and misunderstands something about arguments. We can potentially use this information to tailor the next question to the student. For example, the following question is about function definitions and may be a suitable follow-up question:

Q4: Does the body of a function definition get evaluated no matter what if the function definition occurs in the program?

Representing computing science concepts by including and excluding particular substituent concepts is the idea behind the test suite. In the following chapters, we explore how we can represent computing science concepts in this manner using word embeddings and how we may then apply the word embeddings to suggest the next question.

Chapter 2

Background and Related Work

We build upon the history of analogy-style questions that word embeddings have demonstrated good performance in evaluating [21]. This section will explain the background of word embeddings and question answering.

First, to motivate this work, we will walk through an example of how we can formulate and apply a test to select the next question in a sequence. This will give relevant context when we then look at word embeddings and how they will be used.

Then, we will look at word embeddings. We will also look at related work in sequencing educational items, such as questions on a test.

2.1 Motivating example

Misunderstanding and subsequently correcting the misunderstanding is a common part of learning. Correcting a mistake is often an excellent learning opportunity in itself. What is less straightforward is how to identify the misunderstanding. Because concepts build on each other, there can be a layered effect to a misunderstanding. For example, recall the example of baking bread. If the bread does not rise, that is the top level of the problem. However, the root of the problem is further down in the layers of baking knowledge. It could be that there was a problem with the leavening agent or with the amount of time devoted to the rising stage of the bread-baking process. Understanding that the bread did not rise does not equal an understanding of why the bread did not rise. More questions are necessary to investigate the misunderstanding.

In the context of learning computing science, if a student answers a question incorrectly, the misunderstanding could lay in any of the concepts related to that question. For example, if a student does not understand that a local variable within a function cannot be accessed outside of that function, there are a number of possible misunderstandings that could cause this. It is possible the student misunderstands scope and namespaces, or return statements, or the motivation behind function definitions.

We want to identify where the misunderstanding lays by asking a sequence of well-chosen questions to triangulate the misunderstanding. To illustrate this idea, we will take a closer look at Test 1 from the test suite (explored in detail in Section 3.4):

+function +parameter -argument

Test 1 represents the idea of a **function definition**, because it includes the concept of **function** and **parameter** (part of a function definition) and excludes the concept of **argument** (part of a function call). Hence, this test addresses the concept of a **function definition** and its related concepts: the difference between defining and calling a function, the difference between parameters and arguments, the scope of a function, etc. As such, the expected results for this test should reflect those concepts, with words related to **function definition**, **definition**, **function body**, **function header**, or possibly even **scope** and **return statement**.

The top results for Test 1 are shown in Table 2.1. The results of Test 1 can now be used to suggest a next question to the student. Suggesting the next question will be explored more in Chapter 4 and Chapter 5.

We have seen what happens in the traditional 20 Questions game when a player answers a question: the answer gives direction to the line of questioning and rules out an entire series of questions that do not relate to the player's selected concept. However, what happens if the player makes a mistake? For example, if a player is thinking of a bumblebee and is asked **"Are you thinking of an animal?"**, the

Result	Cosine similarity
invocation	0.5922705392908736
define	0.550625758821478
definition	0.5391357447617813
inside	0.5223577835126354
value	0.5043148363693711

Table 2.1: Top 5 actual results for Test 1 (**+function +parameter -argument**). Higher cosine similarity is better.

player may mistakenly respond no, because they do not think of insects as being animals. If all the animal questions are ruled out at this point, then the questioner will never be able to guess the correct concept.

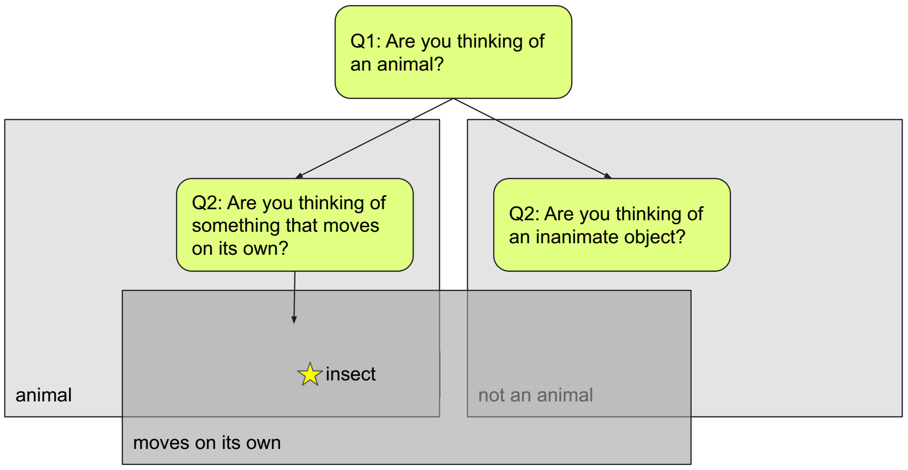

Q1: Are you thinking of an animal?
Q2: Are you thinking of something that moves on its own?
Q2: Are you thinking of an inanimate object?
insect
animal
not an animal
moves on its own

Figure 2.1: Example of a sequence of questions in a 20 Questions game. If the player makes a mistake answering **Q1**, then **Q2** can be used to identify the mistake if the possible concepts identified by **Q2** overlap with **Q1**.

In an educational setting, we must account for mistakes. This brings us to the idea of triangulation, which is illustrated in Figure 2.1. Instead of ruling out the subset of questions, we can ask additional questions that overlap with other questions, so that if a student makes a mistake, we can correct it. In the insect example, the questioner could ask: **"Does it move on its own?"** The player will answer yes, because bumblebees move on their own. Even though the **animal** question rules out bumblebees, the **moves on its own** question rules bumblebees back in. This inconsistency indicates that the player has made a mistake. Now the questioner can ask an additional question to clarify which answer was the incorrect one, thereby "triangulating" the mistake.

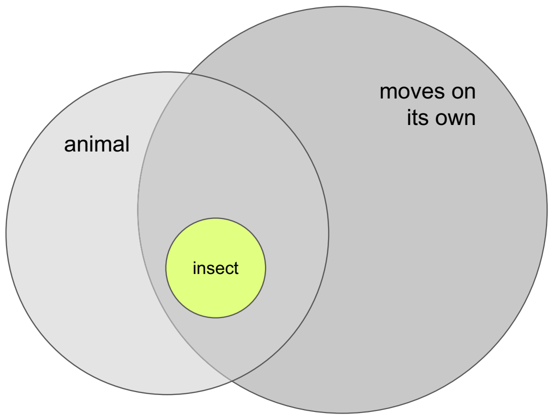


Figure 2.2: The **insect** category is represented by a combination of other categories that describe it. The category of **insect** answers lays in the intersection of **animal** and **moves on its own**.

We can also represent this sequence of questions as a Venn diagram, illustrated in Figure 2.2. The correct answer, **bumblebee**, lies in the middle circle, labeled **insect**. The **insect** circle is in the overlap of the **animal** and **moves on its own** circles. Hence, the concept **bumblebee** is represented by the intersection of relevant concepts that describe **bumblebee**.

This ability to correct or account for student mistakes is important in an educational setting. After all, it is common for a student to make a mistake, especially

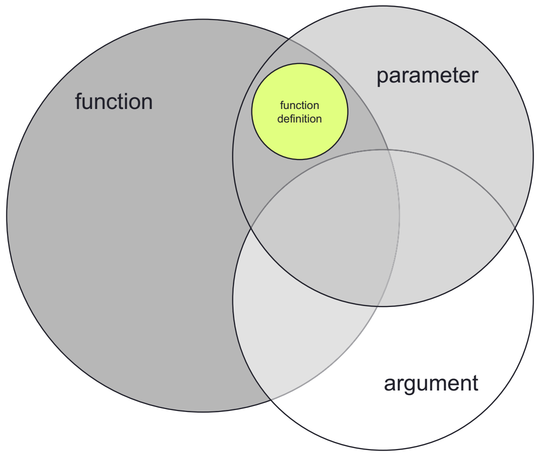


Figure 2.3: **Function definition** can be represented as the intersection of various relevant concepts. **Function definition** is in the intersection of **function** and **parameter** that *does not* overlap with **argument**.

when they are learning the topic. It is often said that mistakes are part of learning! If the student has made a mistake, we cannot necessarily rule out an entire section of questions with confidence. We must ask additional questions that triangulate a student's misunderstanding, by finding concepts that lie withing the various spheres of the student's answers.

We return to the programming example again. Just as an insect can be represented as the intersection of relevant categories, we can represent Test 1 as a collection of relevant and irrelevant concepts as well. In Figure 2.3, **function definition** can be represented as the intersection of various relevant and irrelevant concepts. **Function** and **parameter** both describe a **function definition** in Python, whereas **argument** is about function calls. Therefore, **function definition** can be represented as the intersection of **function** and **parameter** that *does not* overlap with **argument**. In other words, **function definition** can be represented by including **function** and **parameter** and excluding **argument**, as captured by Test 1.

This motivating example demonstrates how we can represent a concept by including and excluding appropriate concepts. In the next sections, we will look at related work in word embeddings and item sequencing.

2.2 Word embeddings

A language model allows us to represent and work with language [20]. There are many ways to model language, varying in detail and scope. Many models have a notion of a vocabulary that is represented by the model. The vocabulary is a finite list of words. Each vocabulary word is represented in the language model. Examples of language are then constructed from these vocabulary words, which can be combined in infinite ways. How you represent the words in the vocabulary also varies. For example, you could simply number each word in the vocabulary. This is an easy way to represent each word in the vocabulary and guarantees a unique identifier for each word. However, this model does not capture any information about the relationships between words. It is possible to capture more information with the way we represent the vocabulary words.

One such possibility is word embeddings [20]. A word embedding is a vector that represents a word. Each word in the vocabulary is represented by a unique vector [11]. All the vectors are the same length. The embeddings are learned from input text [15]. Each place in the vector captures some "quality" of the word, though what the specific qualities are is not human understandable. As such, words that are more similar will have more similar word embeddings [21]. Words that are less similar will have less similar word embeddings.

Word embeddings can be composed with algebraic operations [21]. The answer to an algebraic expression with word embedding terms is the word embedding that is closest to the result vector of the algebraic expression. To measure this similarity, we must consider the two properties of a vector: magnitude and direction. We must compare both the magnitude or length of the vectors and the direction or position of the vectors. If we imagine that all vectors have the same length (i.e., the same size), then the similarity between two vectors is determined by how similarly they are positioned.

In the context of word embeddings, it is common to use cosine similarity [12]. Cosine similarity measures the cosine of the angle between two vectors that are normalized, which accounts for difference in size.

The equation for cosine similarity between vector **A** and vector **B** is:

$$\text{cosine similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||} \tag{2.1}$$

Cosine distance is the compliment of cosine similarity.

$$\text{cosine distance} = 1 - \text{cosine similarity} \tag{2.2}$$

It should be noted that the terms cosine similarity and cosine distance are often used to refer to the same metric in the literature. We will use cosine similarity as defined above. Using this metric, the closer the cosine similarity of two word embeddings is to 1, the more similar the word embeddings are. A word embedding will have a cosine similarity of 1 with itself. The closer the cosine similarity is to 0 for two word embeddings, the less similar the embeddings are. Hence, higher cosine similarity is better when looking for similar words.

Computationally, the main form of information that a corpus offers is the statistical information about the words within the corpus [7]. Of course, this can be broken down further into statistics about the letters or morphemes that occur in the corpus, but we will deal generally with words.

Word embeddings are trained from a corpus of written material by a learning algorithm. The relationships between the words in the training corpus form the basis for the relationships between the word embeddings. For example, if two words co-occur often in the training corpus, these words necessarily have some sort of relationship

[12], whether it is grammatical (for example, an article precedes a noun, such as in the phrase “a computer”; the “ly” ending modifies an an adjective into an adverb, such as the adjective “quick” and the corresponding adverb “quickly”), geographical (for example, a province succeeds a capital city, such as in the phrase “Edmonton, Alberta”), social (for example, “sister” and “brother” are parallel roles that are talked about similarly, such as in the phrase “my sister and brother are both older than I am”), or otherwise [21]. There are many ways that words relate.

Each word in the training corpus is eligible to be included in the final vocabulary. A vocabulary is the complete list of words that are included in the language model. In this case, the vocabulary is the complete list of words for which each word has a corresponding word embedding. Usually, the vocabulary is truncated from the complete list of words that occurs in the training corpus. This is because certain words may be rare in the training corpus and may not be useful to the final product or the corresponding word embedding may not be good, because there was not enough available information to refine it usefully. Any word which is not in the vocabulary is called “out of vocabulary” or OOV for short. The case of OOV words must always be dealt with, since the entire vocabulary of a language almost never makes it into the final language model. In the case of English, the language used here, it would be both impossible and useless to include the entirety of the language in the word embedding model. There are more English words than are feasible to include in the model and a good number of them are not relevant to the work. As well, typos, mistakes, slang, abbreviations, grammatical errors, formatting issues, and other sundry errors or human choices create inconsistencies in a written text that create OOV words (even if the intended word, in the case of a spelling error, for instance, is in the vocabulary). Preprocessing can help or eliminate many of these cases, but unless the input or training material is rigidly structured, there may still be unfamiliar tokens in the preprocessed corpus. In some instances, any OOV word is simply replaced with the token OOV during the preprocessing phase.

The word embedding training algorithm used in this work is GloVe. Word embeddings rely on a training corpus. The corpus provides the meaning for each word and the relationships between words. Essentially, the only meaning that a corpus provides to an unsupervised learning algorithm is the frequency and occurrence of the words [7]. Between words, this means the main statistic is co-occurrence. Co-occurrence is when a word occurs in the proximity or context of another word [2]. Here, context refers to a context window. If the word occurs within the context window around the other word, then it is counted as co-occurring with that word [8].

Word embeddings have a few useful properties. Similar words have similar embeddings [5]. As well, word embeddings encode multiple degrees of similarity [12]. We will use GloVe word embeddings, explored in Section 3.3.

2.3 Item sequencing

We experiment with using word embeddings to select the next question in a sequence. How to select the next item based on a given sequence is closely related to the problem of how to order the items in a sequence. The notion of sequencing applies to both assessment items (such as questions on a test) and instruction items (such as topics in a lesson). We will focus on the ordering of assessment items, since that is most relevant to our work.

While we do not explore how to order the questions in the sequence, we introduce a method to select the next question in a sequence. Selecting the next question is also a matter of ordering and can be compared and contrasted with other methods of item sequencing. However, these are not equivalent tasks. For comparison, broadly speaking, our method to select the next question can be thought of as a 1-step problem (i.e., going from step n to step n+1; use information about the existing sequence to compute the next question). Ordering the sequence is an n-step problem (i.e., going from step 1 to step n; use various factors about the questions, the student answering the questions, and the goal of the system to compute the optimal order). Though these are related problems, performing the n steps to order the sequence is not the same as performing the one step to select the next question, even if this one step is performed multiple times to continue selecting next items. Some methods of ordering a sequence do work by continuously selecting the next question, which will be explored further in this section.

There is a large body of work called item response theory (IRT) that analyzes the properties of individual assessment items [1]. The particular properties of interest are difficulty and discrimination [1]. Difficulty is how difficult an assessment item is, in terms of how likely a student is to answer the question correctly. A low difficulty item will likely have many students answer it correctly. A high difficulty item will likely have few students answer it correctly. Discrimination is how well the assessment item delineates between students of lower and higher ability. An item with high discrimination will likely have lower-ability students answer it incorrectly and higher-ability students answer it correctly. An item with low discrimination will not clearly separate students by ability.

The purpose of using IRT is to assess a student's ability [1]. Ability is calculated from a student's answers to assessment items that have known difficulty and discrimination [1]. Calculating ability allows an instructor to classify and understand students by their level of mastery. In computerized assessment, the assessment can be adapted to the specific student taking the test. IRT is part of the process of selecting the next item in the computerized assessment, with calculations based on the student's responses and the properties of the possible items. By computing a student's ability using IRT, the probability that the student will answer a particular question correctly can be estimated [17]. As such, IRT is used to select the next item such that the item will help determine the student's ability [1], thus completing the goal of the assessment.

Our goal, rather than computing the student's ability, is to find a gap in their understanding. As explored later in Section 4.2 in Chapter 4, we use word embeddings to represent properties of the questions and use these properties as part of the process to suggest a next question. In our case, the property of the question we use is the topic of the question, rather than any properties concerning a question's difficulty or discrimination. This is demonstrated in the sample dialogue in Chapter 1, where we consider a sequence of questions. Each question is about a particular topic. We use a correct answer to signal understanding and an incorrect answer to signal misunderstanding. We can select a new topic that is related to what the student has understood and misunderstood. Then, we can select a next question based on that topic.

Usually, IRT is used in computerized assessment, but it can also be applied to an intelligent tutoring system (ITS). An ITS is an educational tool that provides a student with customized content [17]. If we implement a 20 Questions game in future work, the 20 Questions game could be an ITS. To examine IRT in the context of an ITS, we will look at a particular ITS called KidArn [17]. KidArn is an application for learning the Thai language. KidArn uses IRT to suggest the next activity for a student [17]. A student using KidArn answers a number of questions from a general pool of questions. The questions belong to a variety of different topic categories. The student answers questions until KidArn is able to calculate the student's ability for each topic via IRT with a certain degree of confidence. Then, the next activity is selected from a topic that the student has demonstrated low ability in. The activity can be administered within the same system (such as by asking another question from one of the low-ability topic categories) or outside the system (such as the instructor supplying a resource).

KidArn is an existing application that selects the next question through computation. The proposed word embedding system is not implemented in an ITS of any variety, but we provide a suggested method to select the next question. Both KidArn and our work compute the next category of question and then draw from a pool of questions that correspond to that category. A key difference is that KidArn selects a next question based on topics that the student has not demonstrated mastery of so far, as calculated by the student's ability on that topic [17]. Calculating ability for every topic requires a student to answer many questions to compute ability (and confidence in the computed ability) for every topic in the system. KidArn assumes a student does not understand a topic unless demonstrated otherwise by answering sufficiently many questions correctly. Our work selects the next question based on a diagnosed "gap" in the student's understanding. The gap is represented by the result of an equation created from a student's answers to a small number of questions in a sequence, and evaluated using word embeddings. We implicitly assume a student understands a topic unless they demonstrate otherwise by answering a question incorrectly. We then suggest a question related to something we infer they do not understand.

Since the ITSs discussed here, and the use of IRT, rely on a database of questions, it is important that the questions themselves are designed well, beyond the design of the system itself. The process of writing, editing, testing, and finally implementing test items is described by Spaan [3]. As explored in Chapter 1, our work is inspired by the 20 Questions game. Implementing a working 20 Questions game and developing the questions used in the game are beyond the scope of this work. We do not provide a database of questions. We focus on what could be done with a student's responses to a hypothetical set of questions. The sample dialogue at the end of Chapter 1 (and revisited in Chapter 5) illustrates what the questions could look like in a future iteration. Our word embedding system is intended to be used with a pool of true/false questions. As long as a potential set of questions is about introductory Python and computing science concepts — and hence can be classified or tagged with a selection of relevant topics — the questions could possibly be used in a 20 Questions style game. However, in future work, we can apply principles of good test item design when creating the question database to be used in implementation.

Sequencing educational items, whether these items are part of assessment or instruction, is an ongoing area of research that draws from, among other areas, computing science and psychology [14]. The next question on an assessment is often selected using IRT in computerized assessment and some intelligent tutoring systems, based on a student's demonstrated ability. We experiment with using word embeddings to instead suggest the next question based on an inferred gap in a student's understanding.

2.4 Concluding remarks

We have seen that word embeddings allow us to represent words with vectors and to perform algebraic operations on these vectors. These algebraic computations can give us some insight into the properties of the word embeddings. In particular, these computations may reveal relationships between related words.

We want to apply this property of word embeddings to computing science education. Through a motivating example, we explained our goal: when asking a student a sequence of questions, we want to determine a good next question to ask the student.

A related problem to suggesting a good next question is how to select a good next question in an assessment. Item response theory (IRT) is used to analyze a student's responses and select the questions in computerized assignments [1]. IRT can also be used in an intelligent tutoring system [17]. Instead of IRT, we propose a method that uses word embeddings to select the next question.

In the next chapter, we will explain how we can apply word embeddings to suggest a good next question in a sequence. We will also explore the test suite we will use to analyze whether the method of suggesting a next question produces successful results.

Chapter 3

Experimental Methodology

Word embeddings often demonstrate good results when answering analogy-style questions [19]. These analogy questions usually take the form of

A is to **B** as **C** is to ____

which can be translated into

B - **A** = ____ - **C**

B - **A** + **C** = ____

and evaluated to produce a result vector. The answer to the equation is the word embedding that is nearest to the result vector [21]. This answer fills in the blank in the analogy. For example, Test 1 is:

argument is to **function** as **parameter** is to ____

which can be translated to

function - argument + parameter = ____

and shows up in our test suite as

+function +parameter -argument

Using this analogy question technique, we create algebraic equations with an arbitrary number of positive and negative terms. Then, we can formulate an equation that represents a computing science concept.

There are two key parts to this work. The first is the word embedding model, trained from relevant computing science material (Section 3.3). The second is the test suite, which defines a collection of equations that represent various computing science concepts and skills (Section 3.4). The word embeddings are used to generate answers to the tests in the test suite.

In this chapter, we will explore in detail the development of the word embedding model. The corpus material is selected and preprocessed, before being used as training data for the GloVe algorithm. Then, we will explain the test suite and the expected results of the test suite.

3.1 Corpus

Word embeddings appear to represent the underlying semantic and syntactic structure of the corpus from which the embeddings are trained [5]. Hence, word embeddings can represent the relationships between words, as learned from the training corpus. These relationships can be illustrated by constructing algebraic expressions with the word embeddings as terms in the expressions [21].

The corpus used to train the word embedding model must contain enough training material to adequately represent these words and relationships [20]. If a word we are interested in does not appear in the training corpus, then there will not be a corresponding word embedding for that word. For example, if the word **function** does not appear in the corpus, then the word embedding model will not contain a word embedding for **function** and Test 1 can not be evaluated. This means the corpus must contain relevant domain knowledge.

However, just as the corpus must contain the words we are interested in representing, it is also important to consider how the words appear in the corpus. How the concepts are represented in the corpus and even the choice of what material is included in the corpus influences the word embedding model [20]. For example, as defined in the technical specifications for the Python programming language, function **parameters** and function **arguments** are two distinct, though related, concepts. However, these words may be used interchangeably in less formal sources, even such as textbooks. Depending on which sources are included in the corpus, this distinction may or may not be represented and hence the word embeddings for **parameter** and **argument** may or may not capture this distinction.

We include four sources in the corpus that is used to train the word embedding model. From now on, the complete corpus that is created from these four sources will be referred to as the full corpus. The four individual corpuses are all resources for Python programming and introductory computing science.

The first source is the complete course notes from *Problem Solving, Python Programming, and Video Games* (PVG) [25], which is based on CMPUT 174 - Introduction to the Foundations of Computation I, the general first year computing science course at the University of Alberta. One of the developers of PVG is the author of this dissertation. PVG is a massive open online course (MOOC) and is offered through Coursera. The PVG course notes are adapted from the video transcripts for the course.

PVG focuses on computational thinking and problem solving. The PVG videos are split into two distinct subsets: the lecture videos and the Python language videos. The lecture videos focus on problem solving and application. The Python language videos explain the particulars of Python and include small programming examples. These two video types do not share examples and it is possible to watch either subset on its own. In particular, the Python language videos are a stand-alone Python resource when separated from the rest of the course material.

The second source is the open-source online textbook *Foundations of Python Programming* (FOPP) [22]. The third source is the open-source online textbook *Problem*

Solving with Algorithms and Data Structures using Python (ADS) [23]. Both of these textbooks are hosted on the Runestone Interactive website and made available to various computing science courses. Both textbooks contain programming examples and interactive programming problems.

The fourth source is the documentation for version 3.9 of the Python programming language (PD) [24]. This documentation is a complete resource about Python, describing all aspects of the lexics, syntax, and semantics of the programming language. While a lot of the documentation is written in highly succinct, technical prose, there are also tutorials that go in-depth on specific topics. We have selected these tutorials and will from now on refer to this portion of the Python documentation as simply the Python documentation.

Corpus	**Number of tokens**
PVG	63857
FOPP	102394
ADS	50345
PD	57391
Full corpus	273987

Table 3.1: Size of each corpus by number of tokens after preprocessing (detailed in Section 3.2). The full corpus is composed of the four individual corpuses.

The size of the full corpus in tokens is shown in Table 3.1. For now, token count can be thought of as word count, but tokens will be explained in more depth in Section 3.2.3. The full corpus has 273987 tokens. Each individual corpus that makes up the full corpus is also shown by token count. The full corpus is relatively small [10, 13], as corpuses for training word embeddings often reach billions of tokens [20]. Expanding the full corpus is a possible direction for future work. However, there is evidence that small corpuses can still produce interesting results [13, 18].

3.2 Preprocessing

Once the corpus has been selected, it is preprocessed. The full corpus is preprocessed with a pipeline of common preprocessing steps. Preprocessing standardizes the text of the full corpus, which allows common patterns to be detected across similar items in the text [9]. As well, preprocessing removes inconsistent or unknown tokens and formatting from the full corpus. Each step in the preprocessing pipeline is described below in detail.

3.2.1 Cleaning

The first step in the preprocessing pipeline is cleaning. Cleaning is a general term that refers to standardizing text, in both formatting and writing style [4]. However, we use it to refer to cleaning up formatting in the full corpus. This step removes a set of text artifacts, such as horizontal section break lines, from the text. This set of artifacts is defined based on familiarity with the full corpus text.

3.2.2 Lowercasing

The next step in the preprocessing pipeline is lowercasing, which converts all the letters in the full corpus to lowercase letters. This is done to standardize the text and to ensure all instances of the same word are counted towards the same vocabulary total. For example, without the lowercasing step, the word **Function** and the word **function** would be counted as different words. We use the Python built-in *lower* function to lowercase the full corpus.

3.2.3 Tokenization

The next step in the preprocessing pipeline is tokenization, which splits the full corpus into standard tokens. A token is a basic unit of meaning. Each token is one standard unit, and may be a word or a punctuation mark. Tokens are distinct from each other and do not have relationships between each other, except for ordering. We use the spacy English tokenizer to tokenize the full corpus.

From now on, we will often refer to tokens as words, because it is natural to think of word embeddings as representing words from the full corpus, and the details of tokenization are not important for most of the discussion about the test suite.

3.2.4 Lemmatization

The next step in the preprocessing pipeline is lemmatization. Lemmatization is the practice of replacing each word with its root. For example, **programmed** and **programming** would both be replaced with **program**. The word **program** is already a root, so it is not modified. After lemmatization, all forms of the word **program** will be counted toward the vocabulary total for **program**. After extraneous words are removed from the full corpus during stopword removal (Section 3.2.6), the remaining words in the full corpus may offer valuable information. However, the natural variance of form and style in English writing prevents the meaning from being concentrated across all these forms. Lemmatization addresses this by standardizing the form of the words in the full corpus. We use the spacy English lemmatizer to lemmatize the full corpus.

3.2.5 Bigrams

Bigrams are two-word phrases that have meaning as a single semantic token. For example, **binary expression** has specific meaning, beyond just the meaning of its two constituent one word tokens **binary** and **expression**. Determining which bigrams are important in a corpus is an active area of research. In general, bigrams can be determined by the frequency with which they appear in a corpus. For example, **binary** and **expression** frequently co-occur. However, since the full corpus for this work is small, frequency may not be enough to determine the appropriate bigrams. As such, we define a custom bigram preprocessing step. We use the course glossary for the PVG MOOC to select the relevant bigrams. Any two-word term that occurs in the PVG glossary will be a bigram. We pass through the full corpus and turn each instance of these two words when they appear directly adjacent to one another into a bigram. In practice, this means turning, for example, **binary expression** into **binary_expression**, so that it is considered a single token.

3.2.6 Stopword removal

The next step in the preprocessing pipeline is stopword removal. A stopword is a predefined common word that adds little value to the meaning of a corpus. For example, the word **the** occurs so frequently in English text as to be useless in analysis, since the likelihood of co-occurrence is so high for so many words in the full corpus [6]. The complete stopword list that we remove from the full corpus is shown in Table 3.2.

However, a subtlety for a computing science corpus is that some stopwords are overloaded terms in the context of computing science. For example, the word **and** is included on many stopword lists, but **and** is also a Python keyword and one of the most common logical operators in introductory courses. Removing **and** from the full corpus is likely to remove important tokens from Python examples. Leaving **and** in the full corpus can muddy analysis, since the different semantic contexts of words are not considered when creating our word embeddings. We try to account for this problem with the bigram step (Section 3.2.5), by turning important stopwords into bigrams (such as **if_statement** and **and_operator**). Other overloaded stopwords are removed.

It is important that stopword removal is performed after the bigram preprocessing step. Otherwise, false bigrams may be created. For example, consider the sentence: **"if the statement is false ignore it"**. If stopword removal is performed first, the sentence becomes: **"if statement is false ignore"**. Then, the bigram step will detect **if statement** and turn it into **if_statement**. But this sentence does not

Stopword list						
a	by	hasn	just	other	their	when
about	can	hasn't	ll	our	theirs	where
above	couldn	have	m	ours	them	which
after	couldn't	haven	ma	ourselves	themselves	while
again	d	haven't	me	out	then	who
against	did	having	mightn	over	there	whom
ain	didn	he	mightn't	own	these	why
all	didn't	her	more	re	they	will
am	do	here	most	s	this	with
an	does	hers	mustn	same	those	won
and	doesn	herself	mustn't	shan	through	won't
any	doesn't	him	my	shan't	to	wouldn
are	doing	himself	myself	she	too	wouldn't
aren	don	his	needn	she's	under	y
aren't	don't	how	needn't	should	until	you
as	down	i	no	should've	up	you'd
at	during	if	nor	shouldn	ve	you'll
be	each	in	not	shouldn't	very	you're
because	few	into	now	so	was	you've
been	for	is	o	some	wasn	your
before	from	isn	of	such	wasn't	yours
being	further	isn't	off	t	we	yourself
below	had	it	on	than	were	yourselves
between	hadn	it's	once	that	weren	
both	hadn't	its	only	that'll	weren't	
but	has	itself	or	the	what	

Table 3.2: Complete stopword list. Stopwords are common words that are removed from a corpus during preprocessing (detailed in Section 3.2.6).

refer to an if statement to start with, so this bigram is incorrect. Hence, we perform stopword removal after the bigrams are already found and formatted.

3.3 GloVe word embedding model

We train word embeddings from the preprocessed full corpus using the GloVe script available from Stanford [7]. GloVe, which stands for Global Vectors for Word Representations, uses the statistics for the words across the entire corpus [7]. This key idea of using the global co-occurrence statistics, instead of limiting co-occurrence statistics to whatever falls within a particular window, is to better capture relationships across an entire document. For example, if **function calls** are explained in one chapter and **function definitions** are explained in another chapter, the relation between these three words — **function** and **call**; and **function** and **definition**; as well as the implied relationship between **call** and **definition** due to their dual proximity to **function** — will be captured.

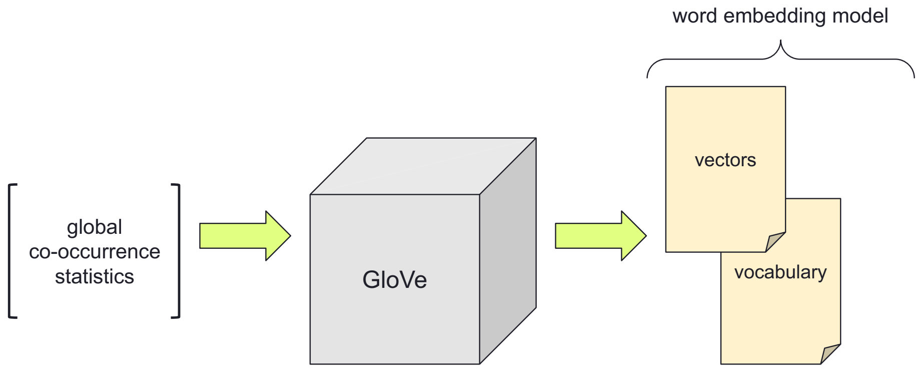


Figure 3.1: GloVe algorithm. Global word co-occurrence statistics from the full corpus are passed into the algorithm to produce a word embedding model consisting of a vocabulary and a corresponding word embedding for each word in the vocabulary.

The GloVe algorithm works by first creating a vocabulary by counting every occurrence of every word in the full corpus. The vocabulary is truncated according to a predefined threshold. We used 20 for the vocabulary threshold. Any word that has a count lower than 20 is removed from the vocabulary. This helps to discount the influence of uncommon words [9]. Then the word co-occurrence statistics are computed. These statistics are stored in a matrix that is the same dimension as the size of the vocabulary. Each entry x_{ij} in the matrix is the count for the number of times word$_i$ occurs in the context of word$_j$. For example, if the sentence **"the Python program was evaluated"** occurs in the full corpus and **Python** is word$_i$ and **program** is word$_j$, this sentence adds one to the x_{ij} count, since **Python** occurs in the context of **program**. Then the GloVe word embeddings are trained from the word co-occurrence statistics [7].

Our word embedding model has vectors of size 50. The word embeddings are learned from the full corpus via the GloVe script [7] using a context window of 15. The word embeddings represent the meanings of the terms via a common set of qualities that may or may not be shared by all the terms in the full corpus vocabulary. As such, the similarity between two terms can be thought of as how much of each quality they share. Words that are more similar have greater overlap in qualities and hence greater overlap in meaning.

3.4 Test suite

The test suite is a collection of tests that are intended to represent concepts from introductory computing science. Each test is an algebraic equation composed of terms relating to computing science and the Python programming language, connected by addition or subtraction operators. The terms are common terms that can be found in various introductory programming educational resources, as well as in the Python programming language documentation, which is a complete resource that explains all aspects of the Python programming language.

#	Test	Expected results
1	+function +parameter -argument	function_definition definition
2	+function -parameter +argument	function_call call
3	+binary +expression +operator -boolean	arithmetic addition
4	+binary +expression +operator -order	short_circuit logic or
5	+if_statement +boolean +elif -else	order condition clause
6	+str -concatenate	type
7	+str -concatenate +int	type convert
8	+global +function +call +local	scope namespace identifier
9	+global +function +call -local	scope namespace main main_function
10	-global +function +call +local	scope namespace function_call identifier
11	+local +identifier +function +call +return	scope namespace bind
12	+global +function +variable +identifier	scope namespace main main_function
13	+local +function +variable +identifier -global	scope namespace main main_function
14	+for_statement +list	sequence header
15	+while_statement +repetition -boolean	definite_repetition suite body
16	+while_statement +repetition -condition	suite body
17	+list +subscription	element index slice

Table 3.3: Test suite of introductory computing science concepts. Each test has a set of expected results, defined by expert knowledge. This table also appears in Chapter 4 as Table 4.5.

Each test in the test suite consists of these introductory computing science terms. Each test represents a particular high-level concept a student must learn. For example, students must understand the difference between function arguments and function parameters. These tests can then be represented as algebraic expressions using the word embeddings.

The test suite is detailed in Table 3.3. Each row of the table contains one test and the corresponding expected results for that test. Each test contains at least two terms, and each term is preceded by either a plus sign or a minus sign. The plus sign represents inclusion of that idea. The minus sign represents exclusion of that idea. Overall, each test represents a particular concept or skill that is important in introductory Python programming or computing science in general.

For example, Test 1 in Table 3.3 is **+function +parameter -argument**. The terms **function** and **parameter** both have plus signs, while the term **argument** has a minus sign. This means that **function** and **parameter** are both included and **argument** is excluded. Overall, Test 1 represents the concept of function definition, because parameters are part of function definitions and arguments are part of function calls, so this test is about the aspect of Python functions that *is* related to parameters and *is not* related to arguments.

The answer to each test can also be thought of as the concept that is both most similar to the plus terms and least similar to the minus terms [12]. To this end, a test may have more than one answer that is relevant or useful to suggest a follow-up question. Selecting only the top answer for each test may exclude useful results. Allowing more than one good result for each test also helps account for a range of difficulty across the test suite [12].

We consider a set of possible answers for each test. As such, each test has a set of expected results that contains at least two terms that are considered a good result for that test. This means, for example, that for Test 1 in Table 3.3, it would be reasonable to get back **definition** or **function_definition** as a result.

We use algebraic addition and subtraction because these are the operations used in word embedding composition [21] and because there is a natural parallel with a model of student understanding. Addition can be used to include concepts towards which a student has demonstrated understanding and subtraction can be used to exclude concepts towards which a student has demonstrated misunderstanding. Note that we use correct answers as evidence of understanding and incorrect answers as evidence of misunderstanding. This is not an entirely accurate model of a student's understanding. For example, a student might get a question wrong due to misreading the question. Alternately, a student might guess on a question they do not understand and get a correct answer. Nonetheless, this model is an acceptable abstraction in the context of this work. The results of these tests indicate missing or related concepts, which can then be used to suggest next questions for a student.

The tests and expected results were chosen according to expert knowledge. This project is an offshoot of a years-long redesign and assessment of the introductory Python programming course at the University of Alberta. The 20 Questions style pedagogical tool that inspired this work was initially conceived as an additional resource for remote students, such as those enrolled in a massive open online course (MOOC). As such, this work builds upon years of teaching and designing assessment for introductory computing science courses, particularly in the context of remote learning. The tests were chosen through iterative discussion between those experienced with the material and the Python programming language.

The expected results were chosen similarly. For each test, we selected terms that summarized the idea behind that test. Since the results for the tests depend so heavily on the full corpus, we sometimes had to modify the expected results depending on the available vocabulary. For example, all names in Python are called **identifiers**. However, it is common to refer to an identifier as a variable, as is done in other programming languages. As such, we may expect **variable** to come up in the full corpus, instead of or as well as **identifier**, and may have to adjust the results accordingly.

Each term in the test also corresponds to a word embedding. Since a word embedding is a vector, algebraic operations can be performed on the vectors. In the context of word embedding analogy questions, only addition and subtraction are used. Addition adds the meaning of the words together. For example, we expect adding **for_statement** and **list** to result in **header**. A list is often used in the header of a for statement. The combined meaning of these words represents an object that shares qualities of both terms. Subtraction removes the meaning of one word from another. For example, we expect subtracting **concatenation** from **str** to result in **type**. Concatenation is an operation for the specific type **str** in Python. Removing it represents removing qualities that are specifically related to **str** and leaves behind a generic **type**.

The following section is a detailed breakdown of the meaning of each test in the test suite, the expected results, and any related tests.

1. **+function +parameter -argument**

Main idea: Function definitions, compared to function calls. The difference between function parameters and function arguments.

Explanation: In Python, the terms passed into a function when the function is called are referred to as arguments. In the function definition that defines that function, the placeholders for the arguments are called parameters. When the function is called, the argument values (if any) are bound to the parameter names in the namespace of the function. This is an important concept for students to grasp, because misunderstandings in the difference between parameters and arguments can lead to misunderstandings about the scope of a function and unintended changes to values. Therefore, this test is about understanding which concepts are part of a function definition and how a function definition and function call differ.

Expected results: The expected results for this test are **definition** and **function_definition**, since this test is about function definitions.

Related tests: 2

2. **+function -parameter +argument**

   Main idea: Function calls, compared to function definitions. The difference between function arguments and function parameters.

   Explanation: As discussed for Test 1, the terms passed into a function call are called arguments and the corresponding placeholders in the function definition are called parameters. When we talk about a function and its arguments in Python, we are talking about a function call. Note that Test 1 and Test 2 can be thought of as the dual of each other. Together, they describe the difference between a function definition and a function call through the difference between parameters and arguments in Python. Opposite to Test 1, this test is about understanding which concepts are part of a function call, and how a function call differs from a function definition.

Expected results: The expected results are **call** and **function_call**, since this test is about function calls.

Related tests: 1

3. **+binary +expression +operator -boolean**

   Main idea: Arithmetic binary operators, compared to boolean binary operators.

   Explanation: A key type of statement in Python is a binary expression. This allows for the use and evaluation of binary operators. There are multiple categories of binary operators that can be used in a binary expression. The two categories that students are usually explicitly introduced to first are arithmetic binary operators (such as addition +, subtraction **-**, multiplication *****, etc) and boolean operators (such as **and**, **or**, etc). Students must learn different rules and applicability for these two categories. Differentiating between arithmetic and logical operators helps understand logic more generally.

   Expected results: Because introductory Python material focuses on arithmetic and boolean binary operators, we can expect that discussion of binary operators that excludes **boolean** should result in terms about arithmetic binary operators. The expected results for this test are **arithmetic** and **addition**.

   Related tests: 4

4. **+binary +expression +operator -order**

   Main idea: Commutativity of binary operators.

   Explanation: Whether or not the order of arguments in a binary expression matters depends on the binary operator used. It matters if the operator is commutative. For example, addition is commutative (the arguments can go in either order and will produce the same result), but subtraction is not. It also matters if the operator can be evaluated with short-circuit evaluation. When we get to boolean operators, the common **and** and **or** operators may seem commutative, but short-circuit evaluation means the order of the arguments matters.

Expected results: The expected results are **short_circuit** and **logic**, since this test is about short-circuit evaluation for logical operators.

Related tests: 3

5. **+if_statement +boolean +elif -else**

   Main idea: If statements with elif clauses but without else clauses.

   Explanation: An if statement must have an if clause. It may also have one or more elif clauses. It may also have an else clause. This test is about the if and elif clauses, excluding the else clause. Boolean is included because the if and elif clauses require a condition that necessarily evaluates to a boolean object. Understanding the interaction between the various clauses of an if statement is an important skill. Students often get confused about whether an if statement must have an else statement, and how the elif statements are evaluated.

   Expected results: The expected results for this test are **order**, **condition**, and **clause**, because this test is about the order of if statement clauses and about which clauses have conditions.

   Related tests: None

6. **+str -concatenate**

   Main idea: Types.

   Explanation: String concatenation is a basic Python programming concept, used to manipulate and format string objects. The type of a string object is

str. (The term **str** is often used instead of the phrase "string object", so **str** is an appropriate term to use in the test.) Learning string concatenation is useful practically, but is also useful on a deeper conceptual level: this is the beginning of understanding how context determines the meaning of symbols and terms. As well, string concatenation can be an introduction to mismatched types and type conversion.

Expected results: The expected result is **type**, because we are subtracting specifics about the **str** type, which we expect to result in the generic **type** concept.

Related tests: 7

7. **+str -concatenate +int**

   Main idea: Types and type errors.

   Explanation: Operators are an important part of computation. Students must learn about the various operators, including both the different types of operators (arithmetic, boolean, etc) as well as how different operator symbols may mean different things depending on the type of the operands. Related is attempting to combine disparate types with an incorrect operator and receiving a type error. This test is about string concatenation and the common type error that is returned when a student attempt to combine a str object and an int object.

   Expected results: The expected results are **type** and **convert**.

   Related tests: 6

8. **+global +function +call +local**

   Main idea: Scope and namespace of a function call.

   Explanation: This test is a dual of Test 9 and Test 10. During the evaluation of a function call, the global namespace and the function's local namespace are both available. This test is about how a function call has access to both namespaces. Learning to make the distinction between the various namespaces is key to understanding scope and hence what is accessible in memory at any given time in the program.

Expected results: The expected results for this test are **scope**, **namespace**, **main**, **main_function**, and **identifier**.

Related tests: 9, 10, 11, 12, 13

9. **+global +function +call -local**

   Main idea: Global namespace during a function call. Difference between local and global namespace.

   Explanation: This test is a dual of Test 8 and Test 10. During the evaluation of a function call, the global namespace and the function's local namespace are both available. This test is about the global namespace. A function call may modify objects that are accessible through the global namespace, such as when a list is passed as an argument.

   Expected results: The expected results for this test are **scope**, **namespace**, **main**, and **main_function**.

   Related tests: 8, 10, 11, 12, 13

10. **-global +function +call +local**

    Main idea: Local namespace during a function call. Difference between local and global namespace.

    Explanation: The test is a dual of Test 8 and Test 9. Test 9 is about the global namespace during a function call. This test is about the local namespace of a function call. It is important that students learn to differentiate between the local and global namespace, so that they understand what data they have access to at any given time. This helps avoid semantic errors, as well as unintentionally modifying objects in memory.

Expected results: The expected results are **scope**, **namespace**, **function_-call**, and **identifier**. We include **identifier** for this test but not for Test 9, because identifiers are the names that are in the namespace, which are often talked about when discussing what is accessible in a function's local namespace.

Related tests: 8, 9, 11, 12, 13

11. **+local +identifier +function +call +return**

    Main idea: Local namespace of a function call and return statements.

    Explanation: Students often struggle with the local nature of a function's namespace; that is, identifiers that are defined in a function's namespace cannot be accessed outside the scope of that function. As well, students struggle with how to gain access to the objects defined in the function's namespace and often attempt to use global identifiers to do this, when they should use return statements. Understanding this concept is important because it allows students to safely use functions to modify and create information.

    Expected results: The expected results for this test are **scope**, **namespace**, **main**, and **main_function**.

    Related tests: 8, 9, 10, 12, 13

12. **+global +function +variable +identifier**

    Main idea: Global identifiers.

    Explanation: Learning to differentiate between local and global identifiers is an important skill. Understanding the scope of identifiers helps to avoid modifying information by accident. As well, good Python style often dictates avoiding the use of global identifiers, especially in introductory courses. This is in part because it forces students to understand scope and reference, and to learn how to use return statements. While students should avoid accidentally modifying objects in memory, they do need to modify memory on purpose. Return statements are part of this.

Expected results: The expected results for this test are **scope**, **namespace**, **main**, and **main_function**.

Related tests: 8, 9, 10, 11, 13

13. **+local +function +variable +identifier -global**

    Main idea: Local identifiers.

    Explanation: This test is the dual of Test 12, focused on local identifiers. Students must learn to differentiate between the local and global namespace. This test is about the identifiers that are bound in the local namespace and accessible in a function call.

    Expected results: The expected results for this test are **scope**, **namespace**, **main**, and **main_function**.

    Related tests: 8, 9, 10, 11, 12

14. **+for_statement +list**

    Main idea: For statement headers that use lists.

    Explanation: The header of a for statement has both a target and a sequence. The sequence can be any iterable data structure. At each iteration, the target is bound to the next item in the sequence. A list is a common data structure to use for the sequence, particularly in introductory Python programming courses. Learning to traverse and operate on lists using a for statement is a foundational skill of introductory Python programming.

    Expected results: The expected results are **sequence** and **header**, because this test is about evaluating the header of a for statement, using a list as the sequence.

Related tests: None

15. **+while_statement +repetition -boolean**

    Main idea: Relationship between while statement conditions and repetition.

    Explanation: Understanding repetition is an important skill. Loops are a fundamental programming construct. Learning to differentiate which situations are best suited by definite or indefinite repetition helps chose the appropriate loop. Understanding that the condition in a while statement works the same as the condition in an if statement can help clarify how a while statement works. This test represents misunderstanding the condition of a while statement, which evaluates to a boolean just like an if statement condition, but understanding that a while statement is a form of repetition.

    Expected results: The expected results for this test are **definite_repetition**, **suite**, and **body**.

    Related tests: 16

16. **+while_statement +repetition -condition**

    Main idea: Relationship between while statement conditions and repetition.

    Explanation: Understanding repetition — both when to use it and how to implement it — is an important basic programming skill. This test is about how while statements and if statements both have conditions, but a while statement is used for repetition and an if statement is not.

    Expected results: The expected results for this test are **suite** and **body**, because this test is about a while statement without a condition.

    Related tests: 15

17. **+list +subscription**

    Main idea: The subscription operator on lists.

    Explanation: The subscription operator is a common and useful operator in introductory Python programming. Learning how to use subscription goes with understanding the structure of a list. As well, understanding subscription lets students iterate through a list and access one element at a time.

    Expected results: The expected results for this test are **element**, **index**, and **slice**, which are all terms related to using the subscription operator on a list.

    Related tests: None

3.5 Concluding remarks

In this chapter we examined the test suite and the word embedding model that will be used to evaluate the test suite. The word embedding model captures the relationships between the terms in the full corpus that was used to train the word embeddings, and hence captures the relationships between various introductory computing science concepts that are represented in the full corpus.

The test suite defines a collection of important introductory concepts and skills for a student learning Python programming and computing science. Each test is an equation that can be evaluated with the word embeddings. As well, each test has a set of expected results. In the next chapter, we will evaluate the test suite and compare the results to the expected results.

Chapter 4

Empirical Results

We have trained a domain-specific word embedding model and defined a test suite that can be evaluated with the word embedding model. The results of the tests in the test suite give us insight into how well the word embedding model represents relevant introductory computing science concepts. However, we can also look deeper into where the model succeeds and fails, to suggest improvements and additions to the training corpus. We will evaluate each test and score the results with various metrics. As well, we will evaluate the strength of the results that are not in the set of expected results for each test, which can be used to suggest improvements to the test suite.

As will be shown below, the initial results for the test suite are promising, but leave much room for improvement. 12 out of the 17 tests in the test suite return at least one of the expected results. Future work is needed to further develop the word embedding model and test suite to improve these results. However, we will also explore the results that are outside the expected results and see that some of these are also reasonable and may also be used to suggest the next question in a sequence.

4.1 Syntactic examples

Word embeddings are capable of capturing various linguistic regularities in a corpus [19]. This includes both syntactic regularities (about the grammar and structure of text) as well as semantic regularities (the way words relate to each other to create meaning). These regularities can be captured through algebraic equations [12]. To illustrate this, we will look at the type of syntactic information captured by our word embedding model. We do this to provide evidence that the word embedding model has captured various regularities from the full corpus. Note that for the following examples, we train and use a version of the word embedding model without the lemmatization preprocessing step applied to the full corpus (Section 3.2.4) so that syntactic variety remains in the final vocabulary.

Let us consider an example of pluralizing a noun. In English, a noun is a part of speech that describes a person, place, or thing. For example, **identifier** is a common noun in Python programming. An identifier is a name that is used to refer to an object of any type [24]. To pluralize a basic noun in English, the general rule is that you add an "s" to the end of the word. (This rule does not hold for every noun or in every case, but we will put aside the nuances of the English language for this example.) This pluralizing rule is true for **identifier**: the plural is **identifiers**.

Result	**Cosine similarity**
functions	0.7237300619081888
defined	0.6096466029647298
named	0.5531785797730547
calls	0.5358811194312112
methods	0.5237045536224422

Table 4.1: Top 5 results for **+identifiers -identifier +function** evaluated on the partially processed full corpus with GloVe word embeddings. The expected result is **functions**. Higher cosine similarity is better.

By subtracting the word embedding for **identifier** from the word embedding for **identifiers**, we expect to produce a vector that captures the "pluralizing s". To test this, we can add the "pluralizing s" result vector to the word embedding for a different noun and check if the result is closest to the pluralized version of that second noun.

For the expression **+identifiers -identifier +function**, we expect the result to be **functions**. After evaluating the expression, the top five results are shown in Table 4.1. The most similar result is **functions**, as expected.

Result	**Cosine similarity**
used	0.642538277899379
for	0.5917442810573474
example	0.5673556364570889
as	0.5661089248796551
this	0.5388184290800645

Table 4.2: Top 5 results for **+defined -define +use** evaluated on the partially processed full corpus with GloVe word embeddings. The expected result is **used**. Higher cosine similarity is better.

Another example is included in Table 4.2. This example captures the "ed" modifier for past tense verbs by subtracting the verb **define** from the past-tense version **defined**, and then adding another present tense verb **use**. We would expect the answer to be **used** and the top result in Table 4.2 is **used**.

Result	**Cosine similarity**
writing	0.5998810322828367
prompts	0.5675435808703901
functions	0.49740012134479533
named	0.4817549964810044
program	0.4678598190384271

Table 4.3: Top 5 results for **+reading -read +write** evaluated on the partially processed full corpus with GloVe word embeddings. The expected result is **writing**. Higher cosine similarity is better.

A final example is shown in Table 4.3. We subtract **read** from **reading** to get a vector that represents "ing" and add the word embedding for **write**. The expected answer is **writing**. The top result in Table 4.3 is **writing**.

These successful syntactic examples demonstrate that the word embedding model has captured syntactic information from the full corpus. In the next section, the test suite will be evaluated, which relies on semantic information from the full corpus.

4.2 Evaluating the test suite

We evaluate the test suite with the word embedding model trained from the full corpus. The results are shown in Table 4.4. Each test in the test suite (Table 4.4 column "Test") is evaluated according to this process:

1. The corresponding word embedding for each term in the test is added or subtracted from the total result to calculate a result vector.
2. The cosine similarity between the result vector and each word embedding in the model is calculated.
3. The similarity between the result vector and all terms from the test is updated to minus infinity.
4. The list of similarities is sorted from highest similarity to lowest similarity.
5. The top five results from the similarity list are returned.

The top five results are the five word embeddings that are most similar to the result vector. Note that step 3 removes the possibility of any term from the test appearing in the results [16]. This is because a word is most similar to itself [5] and the test words would otherwise often appear in the top results.

The top five results for each test are shown in Table 4.4 in column "Results". The results for each test are listed in descending cosine similarity (the first test is most similar and the fifth test is least similar). We call these the actual results for the test suite. Instead of only returning the top result for each test, we return the top five results for each test, because there may be multiple good results for a test [12].

#	Test	Results
1	+function +parameter -argument	invocation define definition inside return
2	+function -parameter +argument	build object instead function_call int
3	+binary +expression +operator -boolean	unary arithmetic tree right operation
4	+binary +expression +operator -order	unary arithmetic mathematical yield infix
5	+if_statement +boolean +elif -else	optional revise expression_statement suite condition
6	+str -concatenate	getitem dict int via repr
7	+str -concatenate +int	callable float type getitem annotation
8	+global +function +call +local	namespace main_function banner bind identifier
9	+global +function +call -local	use define return build write
10	-global +function +call +local	return define parameter definition method
11	+local +identifier +function +call +return	namespace bind main_function function_call name
12	+global +function +variable +identifier	local name namespace bind main_function
13	+local +function +variable +identifier -global	bind name main_function call namespace
14	+for_statement +list	element header contain word suite
15	+while_statement +repetition -boolean	control_structure definite_repetition debugger trace variation
16	+while_statement +repetition -condition	explanation compound_statement operator_token abstraction introduce
17	+list +subscription	element index sequence slice item

Table 4.4: Actual results for the test suite, evaluated on the full corpus using GloVe word embeddings.

For each test in the test suite, there is also an associated set of expected results, shown in Table 4.5 and explored in detail in Section 3.4. The expected results contain a set of terms that represent the idea of the test. For example, Test 1 in Table 4.5 is

+function +parameter -argument

and its corresponding expected results are **function_definition** and **definition**, because this test is about function definitions.

Every test does not have the same number of expected results, because there may not be the same number of reasonable results for each test. However, a single sufficient response for a test is enough to recommend a new question for a student. The difference in number of expected results per test will be accounted for in the various metrics used to evaluate the results of the test suite.

We define three metrics to assess the results for each test. The first metric is *total count* and it is the total number of expected results that occur in the actual results for each test. *Total count* is defined as:

$$S_i \tag{4.1}$$

#	Test	Expected results
1	+function +parameter -argument	function_definition definition
2	+function -parameter +argument	function_call call
3	+binary +expression +operator -boolean	arithmetic addition
4	+binary +expression +operator -order	short_circuit logic or
5	+if_statement +boolean +elif -else	order condition clause
6	+str -concatenate	type
7	+str -concatenate +int	type convert
8	+global +function +call +local	scope namespace identifier
9	+global +function +call -local	scope namespace main main_function
10	-global +function +call +local	scope namespace function_call identifier
11	+local +identifier +function +call +return	scope namespace bind
12	+global +function +variable +identifier	scope namespace main main_function
13	+local +function +variable +identifier -global	scope namespace main main_function
14	+for_statement +list	sequence header
15	+while_statement +repetition -boolean	definite_repetition suite body
16	+while_statement +repetition -condition	suite body
17	+list +subscription	element index slice

Table 4.5: Expected results for the test suite, as defined by expert knowledge. This table also appears in Chapter 3 as Table 3.3.

where S_i is the size of the intersection of the expected results and the actual results for a test. For example, the expected results for Test 1 are **function_definition** and **definition**. The actual results for Test 1 are **invocation**, **define**, **definition**, **inside**, and **return**. The intersection of these two sets is **definition**, which has size 1. Therefore, the *total count* for Test 1 is 1.

The second metric is *recall efficiency*. This is defined as:

$$S_i/S_a \tag{4.2}$$

where S_i is the size of the intersection of the expected results and the actual results for a test (Equation 4.1) and S_a is the number of actual results. This metric can be thought of as the efficiency with which the actual results recall the expected results or the percentage of actual results that are "good" results.

Returning again to Test 1, S_i is 1, as calculated above. S_a is 5, because there are 5 actual results. Thus:

$$S_i/S_a = 1/5 = 0.2 \tag{4.3}$$

The *recall efficiency* for Test 1 is 0.2

The third metric is *recall coverage*. This is defined as:

$$S_i/S_e \tag{4.4}$$

where S_i is the size of the intersection of the expected results and the actual results (Equation 4.1) and S_e is the number of expected results. This metric can be thought of as the percentage of expected results that are returned by the test. In other words, this is the coverage of the expected results by the actual results.

Returning again to Test 1, S_i is 1, as calculated above. S_e is 2, because there are 2 expected results for Test 1. Thus:

$$S_i/S_e = 1/2 = 0.5 \tag{4.5}$$

The recall coverage for Test 1 is 0.5.

#	Test	T	RE	RC	Best results
1	+function +parameter -argument	1	0.2	0.5	definition
2	+function -parameter +argument	1	0.2	0.5	function_call
3	+binary +expression +operator -boolean	2	0.4	1.0	arithmetic addition
4	+binary +expression +operator -order	0	0	0	
5	+if_statement +boolean +elif -else	1	0.2	0.333	condition
6	+str -concatenate	0	0	0	
7	+str -concatenate +int	1	0.2	0.5	type
8	+global +function +call +local	1	0.2	0.333	namespace
9	+global +function +call -local	0	0	0	
10	-global +function +call +local	0	0	0	
11	+local +identifier +function +call +return	2	0.4	0.667	bind namespace
12	+global +function +variable +identifier	2	0.4	0.5	namespace main_function
13	+local +function +variable +identifier -global	2	0.4	0.5	namespace main_function
14	+for_statement +list	1	0.2	0.5	header
15	+while_statement +repetition -boolean	1	0.2	0.333	definite_repetition
16	+while_statement +repetition -condition	0	0	0	
17	+list +subscription	3	0.6	1.0	element index slice

Table 4.6: Total count (T), recall efficiency (RE), and recall coverage (RC) for the test suite evaluated on the full corpus with GloVe word embeddings.

The results of these calculations for all the tests in the test suite are shown in Table 4.6. The terms from the expected results that are found in the actual results (the results that inform the total count) are shown in column "Best results".

The best result for *total count* is Test 17, which has a total count of 3. This means three of the expected results — **element**, **index**, and **slice** — are in the actual results.

The best result for *recall efficiency* is Test 17, which achieves 0.6 recall efficiency. This means that 60% of the actual results are expected results or 60% of the actual results are good results.

The best results for *recall coverage* are Test 3 and Test 17, which both achieve 1.0 on recall coverage. This means all of the expected results appear in the actual results or 100% of the expected results were found by the test.

In contrast to Test 3, Test 4 has 0 for all metrics. This is interesting because Test 3 and Test 4 are similar, differing only on the single minus term. Test 3 has **-boolean** and Test 4 has **-order**. This suggest that **order** is not necessarily a good term to use in the tests about binary operators.

Some dual tests (explored in Section 3.4) achieve similar results and some do not. Test 1 and Test 2 are dual tests and both have the same results on all three metrics. This suggests that there is a parallelism between the tests, which in turn suggests a balance between the concepts in the full corpus.

In contrast, Tests 8, 9, and 10 do not have similar results. Test 8 achieves good results, but the related Test 9 and Test 10 have 0 for all metrics. This suggests a relationship between the three concepts of **global**, **local**, and **function call** that does not hold up when one term is removed.

There are a number of tests that achieve 2 for the *total count* metric. This suggests at least two avenues upon which to suggest the next question for the student, which is promising.

Now we will examine the results in detail for each test.

1. **+function +parameter -argument**

   The best result for this test that occurs in the actual results is **definition**.

Expected results	**Actual results**
function_definition	invocation
definition	define
	definition
	inside
	return

Table 4.7: Expected and actual results for Test 1. These results come from Table 4.5 and Table 4.4 respectively.

Though **function_definition** would have been more specific, we can use the result of **definition** to ask about definitions within the context of functions or about definitions in general. The result **define** is also reasonable, since it is another form of the word **definition**.

The result **invocation** is related to function calls. Since we have already stated that Test 1 (which is about function definitions) and Test 2 (which is about function calls) are duals, it may be reasonable to ask a follow-up question about this related topic.

We could perhaps construct some meaning for the result **inside**, since a parameter is used inside the suite of a function definition. The result **return** is similarly related, in that return statements can be part of a function definition.

2. **+function -parameter +argument**

   The best result for this test that occurs in the actual results is **function_call**. This is also the ideal result for this test according to the expected results.

   The result **build** is interesting. In this context, **build** refers to built, as in built-in functions. (The word "built" is changed to "build" during the lemmatization preprocessing step discussed in Section 3.2.4.) Built-in Python functions are a particular category of function call that introductory computing science students will use and be familiar with. As such, a built-in function may be a good

Expected results	**Actual results**
function_call	build
call	object
	instead
	function_call
	int

Table 4.8: Expected and actual results for Test 2. These results come from Table 4.5 and Table 4.4 respectively.

example with which to explore function calls and function arguments.

The results **object** and **int** are both too broad to offer immediate good follow-up questions. A function call is certainly an **object** in Python and a function may have **int** arguments, but neither are good results with which to explore the difference between function arguments and function parameters.

3. **+binary +expression +operator -boolean**

Expected results	**Actual results**
arithmetic	unary
addition	**arithmetic**
	tree
	right
	operation

Table 4.9: Expected and actual results for Test 3. These results come from Table 4.5 and Table 4.4 respectively.

The best result for this test that occurs in the actual results is **arithmetic**, since this test concerns non-boolean operators and arithmetic operators are the most discussed non-boolean operators in introductory programming.

The results **tree**, **right**, and **operation** are all related, but not specific enough to offer good follow-up questions.

The result **unary** is not what we were attempting to capture with this test, but is not a bad result. It is possible to have a unary operator as part of a binary expression. As such, we could ask a follow-up question about the arguments in a binary expression.

4. **+binary +expression +operator -order**

Expected results	**Actual results**
short_circuit	unary
logic	arithmetic
or	mathematical
	yield
	infix

Table 4.10: Expected and actual results for Test 4. These results come from Table 4.5 and Table 4.4 respectively.

Arithmetic and **mathematical** are both relevant results. Asking another question about arithmetic operators would be an appropriate choice. The other results may be related terms but do not offer specific follow-up questions. For example, both **yield** and **infix** are related to operators, but neither are related to the idea behind this test.

Because the actual results of this test do not overlap with the expected results, this suggests that the test does not capture the concept of short-circuit evaluation and should be reformulated in future work.

5. **+if_statement +boolean +elif -else**

The best expected result for this test is **condition**, since both if and elif clauses have conditions, but else clauses (the excluded term in the test) do not. As well, a condition evaluates to a boolean object.

The result **suite** is good, since the evaluation of a suite is an important part

Expected results	**Actual results**
order	optional
condition	revise
clause	expression_statement
	suite
	condition

Table 4.11: Expected and actual results for Test 5. These results come from Table 4.5 and Table 4.4 respectively.

of understanding if statements. However, the other results for this test do not offer related follow-up questions.

6. **+str -concatenate**

Expected results	**Actual results**
type	getitem
	dict
	int
	via
	repr

Table 4.12: Expected and actual results for Test 6. These results come from Table 4.5 and Table 4.4 respectively.

This test takes a particular type and subtracts away defining characteristics of that type. As such, the results **dict** and **int** are both good results, because they are both other Python types. Asking a student to differentiate various Python types is a good follow-up question.

The other results, such as **getitem**, suggest particular examples from the full corpus that may be useful to refer to students as a resource.

7. **+str -concatenate +int**

Expected results	**Actual results**
type	callable
convert	float
	type
	getitem
	annotation

Table 4.13: Expected and actual results for Test 7. These results come from Table 4.5 and Table 4.4 respectively.

Similar to the previous test, both **float** and **type** make sense as results for this test, since they are other Python types. However, they also suggest this test should be reworded, since it is only capturing information about types and not information about type errors.

The results **callable** and **annotation** are not related to the idea of this test and hence do not offer good options for a follow-up question. Neither str nor int are **callable** in Python and **annotation** is too general of a verb.

8. **+global +function +call +local**

Expected results	**Actual results**
scope	**namespace**
namespace	main_function
identifier	banner
	bind
	identifier

Table 4.14: Expected and actual results for Test 8. These results come from Table 4.5 and Table 4.4 respectively.

Most of the actual results of this test are related to the concept of scope that the test captures. Though **banner** is not a good result on the surface, it actually refers to a specific function name from an example in the PVG corpus. The example is useful for students to refer to when trying to understand the scope of a function call. Hence, referring students to this example could be useful, which suggest another possible use for the test results.

9. **+global +function +call -local**

Expected results	**Actual results**
scope	use
namespace	define
main	return
main_function	build
	write

Table 4.15: Expected and actual results for Test 9. These results come from Table 4.5 and Table 4.4 respectively.

Return is a great result for this test. It could even be added to the expected results in a future iteration of the test suite, since the return statement is an important part of accessing a value outside a function call.

10. **-global +function +call +local**

Expected results	**Actual results**
scope	return
namespace	define
function_call	parameter
identifier	definition
	method

Table 4.16: Expected and actual results for Test 10. These results come from Table 4.5 and Table 4.4 respectively.

This test has no expected results appear in the actual results, which suggests the test needs to be reworded or the expected results reconsidered. The result

parameter is related to the test, since in the local namespace of a function call, the parameter names are bound to the arguments that are passed.

The majority of the results for this test – **define**, **definition**, and **method** – are not helpful for differentiating between global and local scope for a function call.

11. **+local +identifier +function +call +return**

Expected results	**Actual results**
scope	**namespace**
namespace	**bind**
bind	main_function
	function_call
	name

Table 4.17: Expected and actual results for Test 11. These results come from Table 4.5 and Table 4.4 respectively.

The results **namespace**, **bind**, and **name** are good results for this test, since it is about returning values that are in the local namespace of a function call. Asking follow-up questions about the namespace is a good choice. The result **main_function** is also a good result, because many function calls happen in the main function in an introductory Python programming course.

12. **+global +function +variable +identifier**

Both this test and Test 13 have the same set of expected results, which suggests the need to differentiate these tests.

The result **main_function** is a good result, because a call to the main function is one of the only lines of code that is permitted outside of a function call in strict introductory programming style.

13. **+local +function +variable +identifier -global**

Expected results	**Actual results**
scope	local
namespace	name
main	**namespace**
main_function	bind
	main_function

Table 4.18: Expected and actual results for Test 12. These results come from Table 4.5 and Table 4.4 respectively.

Expected results	**Actual results**
scope	bind
namespace	name
main	**main_function**
main_function	call
	namespace

Table 4.19: Expected and actual results for Test 13. These results come from Table 4.5 and Table 4.4 respectively.

Both this test and Test 12 have the same set of expected results, which suggests the need to differentiate these tests.

The results **main_function** and **namespace** are good results for this test because it is about the local scope of a function call. The other results **bind**, **name**, and **call** are also good results, since these have to do with binding names in the namespace, which is part of the semantics of a function call.

14. **+for_statement +list**

    Similar to Test 8, the result **word** is a common variable name when using a list as the sequence in a for statement header, and could be used to suggest a specific example from the full corpus to a student. The rest of the results for this test are highly related to for statements.

Expected results	**Actual results**
sequence	element
header	**header**
	contain
	word
	suite

Table 4.20: Expected and actual results for Test 14. These results come from Table 4.5 and Table 4.4 respectively.

15. **+while_statement +repetition -boolean**

Expected results	**Actual results**
definite_repetition	control_structure
suite	**definite_repetition**
body	debugger
	trace
	variation

Table 4.21: Expected and actual results for Test 15. These results come from Table 4.5 and Table 4.4 respectively.

The result **definite_repetition** is a great result for this test, since it suggests that a while statement is a for statement plus a boolean condition. While the result **control_structure** is related, it is not specific. The other results are also too general to offer good specific follow-up questions.

16. **+while_statement +repetition -condition**

None of the results of this test are good. This suggests that the test must be reformulated and possibly that the word **condition** is not well represented in the full corpus. This may be addressed further in future work.

17. **+list +subscription**

Expected results	**Actual results**
suite	explanation
body	compound_statement
	operator_token
	abstraction
	introduce

Table 4.22: Expected and actual results for Test 16. These results come from Table 4.5 and Table 4.4 respectively.

Expected results	**Actual results**
element	**element**
index	**index**
slice	sequence
	slice
	item

Table 4.23: Expected and actual results for Test 17. These results come from Table 4.5 and Table 4.4 respectively.

All the results for this test are relevant. Since all five actual results are good, but there are only three terms in the expected results, this suggests that the expected results for this test should be modified.

In Table 4.24, each corpus that contributes to the full corpus is evaluated individually according to the three metrics. This allows us to examine the contribution each corpus makes to the overall results. Recall from Section 3.1 that the four individual sources that make up the full corpus are:

1. The course notes from the online course *Problem Solving, Python Programming, and Video Games* (PVG).

2. The textbook *Foundations of Python Programming* (FOPP).

3. The textbook *Problem Solving with Algorithms and Data Structures using Python* (ADS).

4. A selection of tutorials from the official Python documentation (PD).

		Total count					Recall efficiency					Recall coverage				
#	Test	PVG	FOPP	ADS	PD	Full	PVG	FOPP	ADS	PD	Full	PVG	FOPP	ADS	PD	Full
1	+function +parameter -argument	1	1	0	1	1	0.2	0.2	OOV	0.2	0.2	0.5	0.5	0	0.5	0.5
2	+function -parameter +argument	1	0	0	1	1	0.2	0	OOV	0.2	0.2	0.5	0	0	0.5	0.5
3	+binary +expression +operator -boolean	0	0	1	0	2	OOV	OOV	0.2	OOV	0.4	0	0	0.5	0	1.0
4	+binary +expression +operator -order	0	0	0	0	0	OOV	OOV	0	0	0	0	0	0	0	0
5	+if_statement +boolean +elif -else	2	0	0	0	1	0.4	OOV	OOV	OOV	0.2	0.667	0	0	0	0.333
6	+str -concatenate	1	0	0	0	0	0.2	0	OOV	OOV	0	1.0	0	0	0	0
7	+str -concatenate +int	1	0	0	0	1	0.2	0	OOV	OOV	0.2	0.5	0	0	0	0.5
8	+global +function +call +local	1	0	0	1	1	0.2	0	OOV	0.2	0.2	0.333	0	0	0.333	0.333
9	+global +function +call -local	0	0	0	0	0	0	0	OOV	0	0	0	0	0	0	0
10	-global +function +call +local	0	0	0	0	0	0	0	OOV	0	0	0	0	0	0	0
11	+local +identifier +function +call +return	2	0	0	0	2	0.4	OOV	OOV	0	0.4	0.667	0	0	0	0.667
12	+global +function +variable +identifier	0	0	0	0	2	OOV	OOV	OOV	0	0.4	0	0	0	0	0.5
13	+local +function +variable +identifier -global	0	0	0	0	2	OOV	OOV	OOV	0	0.4	0	0	0	0	0.5
14	+for_statement +list	1	0	0	0	1	0.2	OOV	OOV	OOV	0.2	0.5	0	0	0	0.5
15	+while_statement +repetition -boolean	0	0	0	0	1	0	OOV	OOV	OOV	0.2	0	0	0	0	0.333
16	+while_statement +repetition -condition	0	0	0	0	0	0	OOV	OOV	OOV	0	0	0	0	0	0
17	+list +subscription	2	0	0	0	3	0.4	OOV	OOV	OOV	0.6	0.667	0	0	0	1.0

Table 4.24: For each corpus that makes up the full corpus, total count (T), recall efficiency (RE), and recall coverage (RC). OOV indicates that a word in that test did not appear in the vocabulary for that corpus. Full corpus results included for comparison.

For each individual source, a word embedding model is trained, using the same preprocessing steps and parameters as the full model. Then the test suite is evaluated with the corpus-specific word embeddings. The result of each test for the full corpus is also included for comparison. If the result is listed as OOV, which stands for "out of vocabulary" as explained in Section 2.2, this means that the size of the actual results was 0 for that test. This occurs when one of the test terms does not appear in the vocabulary for that corpus and hence the test cannot be evaluated.

It is interesting to note that results across the individual corpuses are not additive. In some cases, there may not even be a result for any individual corpus, but the full corpus has a result. For example, Test 12 and Test 13 do not have a result for any individual corpus, but perform well on the full corpus with a total count of 2 for both tests.

4.3 Concluding remarks

In this chapter, we evaluated the test suite with the word embedding model and analyzed the results. On average, the test suite achieves 21% recall efficiency and 39% recall coverage. As well, 12 out of the 17 tests in the test suite have at least one good result, as determined by the expected results. That is, at least one of the expected results occurs in the actual results for 12 of the tests.

These results suggest there is a possibility to recommend an appropriate follow-up question using the word embedding method for 12 of our tests. We have also seen that some of the actual results that are not included in the expected results still have value, which suggests possible improvements to the test suite and expected results in the future.

Chapter 5

Concluding Remarks

Asking a well-chosen sequence of questions is the key to successfully playing the game 20 Questions, but it is also useful in education. When an instructor speaks to a student, the instructor can deduce the student's misunderstandings by asking a sequence of questions. This process allows the instructor to identify any gaps in the student's understanding.

We attempt to compute the gap in a student's understanding and suggest an appropriate next question in a sequence. To this end, we create a word embedding model trained using the GloVe algorithm from a corpus of introductory computing science and Python material. Word embeddings have long been used to attempt to capture the similarities and hence the relationships between words from a corpus [12]. By training word embeddings from a relevant computing science corpus, we attempt to capture the structure of introductory computing science, including how various concepts are related and build on each other.

A common approach to revealing the relationships between word embeddings is to use an analogy question, which can then be adapted into an algebraic equation [21]. These equations are evaluated with word embeddings as the terms. We adapt the idea of the analogy question into an algebraic equation with arbitrarily many positive and negative terms. We define a test suite, where each test represents a concept or skill that an introductory computing science student should understand. Each test is an algebraic equation that is evaluated using our word embedding model. As well, we define a set of expected results for each test.

Then, the tests are evaluated by computing a result vector and finding the word embeddings in the model that are closest to the result vector. Instead of only taking the top result for each test, we consider the top five results, which we call the actual results. We then compare the actual results with the expected results for each test.

The results for the test suite show promise, but also much room for improvement and future work. 12 out of 17 tests have at least one expected result occur in the set of actual results, which demonstrates that this method does produce at least some appropriate results. The test suite provides evidence that a sequence of questions may be translated into an algebraic equation and evaluated with the word embedding model.

We will look at Test 1 again in the context of a 20 Questions style game. Test 1 is:

+function +parameter -argument

A possible sequence of questions, as first introduced in Chapter 1, might look like the following sequence, which is also illustrated in Figure 5.1.

Q1: Is there a Python programming construct that can be used to remove non-adjacent duplicate code blocks?

The answer is yes, because removing non-adjacent duplicate code blocks is a use for functions. This question is about functions and hence this test can be catalogued or tagged with the term **function**.

Q2: Is it possible to define the value that a function can accept?

The answer is yes, because a parameter is how the input to a Python function is defined. This question is about parameters and can be tagged with **parameter**.

Q3: Is the state *argument* part of the function definition syntax diagram?

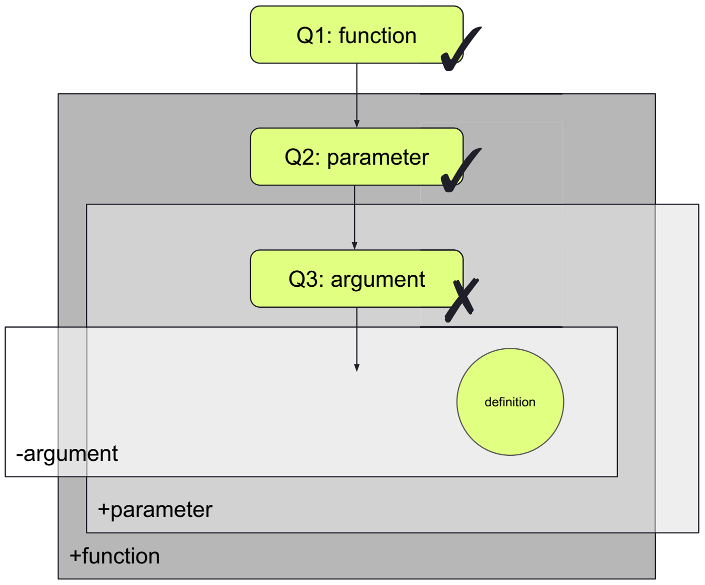


Figure 5.1: Sequence of Python programming questions being used to suggest a next question based on the student's answers. A correct answer from the student is indicated with a checkmark and an incorrect answer is indicated with an x. The next question is suggested by the term, **definition**, that appears in the intersection of the information gained from the sequence of **Q1**, **Q2**, **Q3**.

The answer is no, because there is no **argument** state in the function definition syntax diagram. There is a **parameter list** state. This question can be tagged with **argument**.

If the student gets **Q1** and **Q2** correct, but **Q3** incorrect, as represented by two checkmarks and an x in Figure 5.1, we can build the following representation of the student's answers so far: **+function +parameter -argument**. This is exactly Test 1. As detailed in Chapter 4, the results for Test 1 are: **invocation**, **define**, **definition**, **inside**, and **return**. These results can be used to select a next question, by selecting a question tagged with one of these terms from the bank of questions used by the game. For example, a question tagged with **definition** might be:

Q4: Does the body of a function definition get evaluated no matter what if the function definition occurs in the program?

The answer to this question is no, because the body of a Python function definition is not evaluated until the function is called. This question is about function definitions, and could be a good follow-up question for a student who misunderstands something about the difference between Python function definitions and function calls, as demonstrated by a misunderstanding in the difference between parameters and arguments.

As illustrated by this example, the sequence of added and subtracted terms built by the student's answers results in an equation that can be used to suggest a reasonable follow-up question to a sequence of questions.

Bibliography

[1] F. B. Baker, *The basics of item response theory*, en, 2nd ed. College Park, Md.: ERIC Clearinghouse on Assessment and Evaluation, 2001, ISBN: 978-1-886047-03-7.

[2] D. L. T. Rohde, L. M. Gonnerman, and D. C. Plaut, "An improved model of semantic similarity based on lexical co-occurence," *Communications of the Acm*, vol. 8, pp. 627–633, 2006.

[3] M. Spaan, "Evolution of a Test Item," *Language Assessment Quarterly*, vol. 4, no. 3, pp. 279–293, Aug. 2007, Publisher: Routledge, ISSN: 1543-4303. DOI: 10.1080/15434300701462937. [Online]. Available: https://doi.org/10.1080/15434300701462937 (visited on 12/24/2021).

[4] J. Lin and D. Ryaboy, "Scaling big data mining infrastructure: The twitter experience," *ACM SIGKDD Explorations Newsletter*, vol. 14, no. 2, pp. 6–19, Apr. 2013, ISSN: 1931-0145. DOI: 10.1145/2481244.2481247. [Online]. Available: http://doi.org/10.1145/2481244.2481247 (visited on 03/13/2020).

[5] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," en, *arXiv:1301.3781 [cs]*, Sep. 2013, arXiv: 1301.3781. [Online]. Available: http://arxiv.org/abs/1301.3781 (visited on 01/14/2021).

[6] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," en, p. 9, 2013.

[7] J. Pennington, R. Socher, and C. Manning, "Glove: Global Vectors for Word Representation," en, in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. [Online]. Available: http://aclweb.org/anthology/D14-1162 (visited on 01/28/2021).

[8] O. Levy, Y. Goldberg, and I. Dagan, "Improving Distributional Similarity with Lessons Learned from Word Embeddings," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015. DOI: 10.1162/tacl_a_00134. [Online]. Available: https://aclanthology.org/Q15-1016 (visited on 11/19/2021).

[9] V. Mohan, "Preprocessing Techniques for Text Mining - An Overview," Feb. 2015.

[10] M. Sahlgren and A. Lenci, “The Effects of Data Size and Frequency Range on Distributional Semantic Models,” en, in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Austin, Texas: Association for Computational Linguistics, 2016, pp. 975–980. DOI: 10.18653/v1/D16-1099. [Online]. Available: http://aclweb.org/anthology/D16-1099 (visited on 12/06/2021).

[11] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching Word Vectors with Subword Information,” *arXiv:1607.04606 [cs]*, Jun. 2017, arXiv: 1607.04606. [Online]. Available: http://arxiv.org/abs/1607.04606 (visited on 05/26/2021).

[12] G. Finley, S. Farmer, and S. Pakhomov, “What Analogies Reveal about Word Vectors and their Compositionality,” in *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017)*, Vancouver, Canada: Association for Computational Linguistics, Aug. 2017, pp. 1–11. DOI: 10.18653/v1/S17-1001. [Online]. Available: https://www.aclweb.org/anthology/S17-1001 (visited on 01/19/2021).

[13] A. Herbelot and M. Baroni, “High-risk learning: Acquiring new word vectors from tiny data,” en, in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark: Association for Computational Linguistics, 2017, pp. 304–309. DOI: 10.18653/v1/D17-1030. [Online]. Available: http://aclweb.org/anthology/D17-1030 (visited on 12/06/2021).

[14] S. Doroudi, V. Aleven, and E. Brunskill, “Where’s the Reward?” en, *International Journal of Artificial Intelligence in Education*, vol. 29, no. 4, pp. 568–620, Dec. 2019, ISSN: 1560-4306. DOI: 10.1007/s40593-019-00187-x. [Online]. Available: https://doi.org/10.1007/s40593-019-00187-x (visited on 12/24/2021).

[15] P. Molino, Y. Wang, and J. Zhang, “Parallax: Visualizing and Understanding the Semantics of Embedding Spaces via Algebraic Formulae,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 165–180. DOI: 10.18653/v1/P19-3028. [Online]. Available: https://www.aclweb.org/anthology/P19-3028 (visited on 03/22/2021).

[16] M. Nissim, R. van Noord, and R. van der Goot, “Fair is Better than Sensational:Man is to Doctor as Woman is to Doctor,” *arXiv:1905.09866 [cs]*, Nov. 2019, arXiv: 1905.09866. [Online]. Available: http://arxiv.org/abs/1905.09866 (visited on 11/19/2021).

[17] S. Rungrat, A. Harfield, and S. Charoensiriwath, “Applying Item Response Theory in Adaptive Tutoring Systems for Thai Language Learners,” in *2019 11th International Conference on Knowledge and Smart Technology (KST)*, ISSN: 2374-314X, Jan. 2019, pp. 67–71. DOI: 10.1109/KST.2019.8687462.

[18] G. Wohlgenannt, A. Barinova, D. Ilvovsky, and E. Chernyak, “Creation and Evaluation of Datasets for Distributional Semantics Tasks in the Digital Humanities Domain,” *ArXiv*, 2019.

[19] L. Fournier, E. Dupoux, and E. Dunbar, “Analogies minus analogy test: Measuring regularities in word embeddings,” in *CONLL*, 2020. DOI: 10.18653/v1/2020.conll-1.29.

[20] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, “On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?” en, p. 14, 2021.

[21] T. Mikolov, W.-t. Yih, and G. Zweig, “Linguistic Regularities in Continuous Space Word Representations,” en, p. 6,

[22] B. Miller, P. Resnick, L. Murphy, J. Elkner, P. Wentworth, A. B. Downey, C. Meyers, and D. Mitchell, *Foundations of Python Programming*. [Online]. Available: https://runestone.academy/runestone/books/published/fopp/index.html (visited on 12/06/2021).

[23] B. N. Miller and D. L. Ranum, *Problem Solving with Algorithms and Data Structures using Python — Problem Solving with Algorithms and Data Structures*. [Online]. Available: https://runestone.academy/runestone/books/published/pythonds/index.html (visited on 12/06/2021).

[24] *Python 3.9 Documentation*. [Online]. Available: https://docs.python.org/3.9/ (visited on 11/25/2021).

[25] D. Szafron, P. Lu, E. McDonald, and E. Hill, *Problem Solving, Python Programming, and Video Games*, en. [Online]. Available: https://www.coursera.org/learn/problem-solving-programming-video-games (visited on 12/06/2021).