Searching a Database of Source Codes Using Contextualized Code Search

Rohan Mukherjee
Rice University
Houston, USA
rm38@rice.edu

Swarat Chaudhuri
Rice University
Houston, USA
swarat@rice.edu

Chris Jermaine
Rice University
Houston, USA
cmj4@rice.edu

ABSTRACT

Consider the case where a programmer has written some part of a program, but has left part of the program (such as a method or a function body) incomplete. The goal is to use the context surrounding the missing code to automatically "figure out" which of the codes in the database would be useful to the programmer in order to help complete the missing code. The search is "contextualized" in the sense that the search engine should use clues in the partially-completed code to figure out which database code is most useful. The user should not be required to formulate an explicit query.

We cast contextualized code search as a learning problem, where the goal is to learn a distribution function computing the likelihood that each database code completes the program, and propose a neural model for predicting which database code is likely to be most useful. Because it will be prohibitively expensive to apply a neural model to each code in a database of millions or billions of codes at search time, one of our key technical concerns is ensuring a speedy search. We address this by learning a "reverse encoder" that can be used to reduce the problem of evaluating each database code to computing a convolution of two normal distributions.

PVLDB Reference Format:

Rohan Mukherjee, Swarat Chaudhuri, and Chris Jermaine. Searching a Database of Source Codes Using Contextualized Code Search. *PVLDB*, 13(10): 1765-1778, 2020.

DOI: https://doi.org/10.14778/3401960.3401972

1 Introduction

An end-user has produced a partially completed computer program, where a piece of code (typically a body of a method or function) is wholly or partially missing. The goal is to search a large corpus of program fragments $D = \{\mathsf{Prog}_1, \mathsf{Prog}_2, ...\}$, and automatically choose the fragment from the database that is most likely to complete the program, without requiring an end-user to explicitly formulate a query. For example, consider the following code:

Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401972

```
class IO {
  public void readFully(InputStream fd,
    byte[] dst, int off, int len)
                    throws IOException {
      while (len > 0) {
         int r = fd.read(dst, off, len);
         off += r;
         len -= r; }}

  public void findMe (OutputStream out){
      __CODE_SEARCH__ }}
```

The goal is to find a code in a large database of codes that could replace the missing code indicator. The search is *contextualized* because the user need not describe the search using an explicit query; the context around the missing code—class member variables, comments, surrounding method signatures, and so on—is used to power the search. The goal is to provide search functionality "for free"; the user of an integrated development environment (IDE) need only click on a particular line where the missing code is to be inserted, and the system looks at the partially-completed code and figures out from the context what the answer should be. The user need not take the time to posit a query, composing keywords or writing an English description of the code that s/he wants. In this case, one of the top codes returned is:

```
/** Writes the contents of this byte array output
    stream to the specified output stream argument.*/
public void writeTo(OutputStream out)
                 throws IOException {
  ByteString[] cachedFlushBuffers;
  byte[] cachedBuffer; int cachedBufferPos;
  synchronized (this) {
    cachedFlushBuffers=flushedBuffers.toArray(
        new ByteString[flushedBuffers.size()]);
    cachedBuffer=buffer;
    cachedBufferPos=bufferPos; }
  for(ByteString byteString:cachedFlushBuffers){
    byteString.writeTo(out);
    }
   out.write(copyArray(
            cachedBuffer,cachedBufferPos));
}
```

The system was able to infer from the surrounding class—which included a method that reads from an input stream—that the user was looking for a `write` method.

Code search via neural embedding. Methods for learning neural embeddings have become widespread. The idea is to learn a neural function that is able to map objects to a position in a high-dimensional space, such that objects that are similar or related are positioned closely to one another. Methods for computing word embeddings such as Word2Vec [30] and BERT [14] are the best examples of this. Not surprisingly, such methods have been applied to code search, especially for powering natural language-based search [35, 19, 46]. The idea is to learn one neural function that embeds a query in a high-dimensional space, and another that embeds a code in the same space. Code search then reduces to nearest-neighbor search.

Unfortunately, there may be little reason to believe that such methods will work for contextualized code search (CCS). In CCS, both queries and codes are exceedingly complex objects, so that learning a generalizable embedding from millions of query-code co-occurrences in a training set seems hard. Queries in contextualized code search are inherently multi-modal, with a large number of disparate evidences as to what the query result should be: sets of types, method calls and keywords surrounding the missing code, natural language comments, sequences of formal parameters, and so on. Because of this, each query in a training corpus is likely unique, and will only be seen once in the training data. Compare this to the problem of learning a word embedding, where each word is seen many times in many different contexts. As a result, over-fitting may be a significant problem, leading to poor search performance.

Code search as program synthesis. We propose a unique approach to CCS, where we view CCS as a special case of statistical program synthesis. Statistical program synthesis [29, 10, 32] is the problem of learning how to automatically write programs. In particular, we view CCS as a variety of *conditional program generation* [32], where a learner learns to use the context X collected from the surrounding code to realize a posterior distribution function $P(\mathsf{Prog}|\mathsf{X}) = \int_{\mathsf{Z}} P(\mathsf{Prog}|\mathsf{Z})P(\mathsf{Z}|\mathsf{X})d\mathsf{Z}$, for a latent variable Z[1]. Z can be viewed as an unknown specification for the code to be generated. When treating CCS as an instance of conditional program generation, search is the task of finding the database program such that $\mathsf{Prog} = \text{argmax}_{\mathsf{Prog}' \in D} P(\mathsf{Prog}'|\mathsf{X})$.

In contrast to more traditional, embedding-based approaches that attempt to map both context and program to similar representations in a high-dimensional latent space, conditional program generation attempts to learn to generate the program from the context. This may be more resistant to over-fitting because the statistical program synthesizer must learn to accurately generate Prog, despite of the uncertainty in Z embodied by the distribution $P(Z|\mathsf{X})$.

The key difficulty with re-casting contextualized code search as synthesis is computational. When the goal is to synthesize a program by generating Prog so as to maximize $P(\mathsf{Prog}|\mathsf{X})$, seconds or even minutes of compute time can be devoted to solving the resulting optimization problem. During search, however, for a given query evidence X, $P(\mathsf{Prog}_i|\mathsf{X})$ must be evaluated for millions or billions of values of i very quickly, while the user waits. In a typical implementation, Prog_i will take the form of a parse tree, and evaluating $P(\mathsf{Prog}_i|\mathsf{X})$ requires repeatedly pushing productions in that parse three through a neural network. Doing this quickly for millions of different programs will not be feasible. To address this, we force the posterior $P(Z|\mathsf{X})$ distribution over the unknown specification Z to be multivariate Gaussian. When learning $P(\mathsf{Prog}|\mathsf{X})$, we concurrently learn an approximation $Q(Z|\mathsf{Prog}) \approx P(Z|\mathsf{Prog})$ (a so-called "reverse encoder") where $Q(Z|\mathsf{Prog})$ is also constrained to be normal. Computing $P(\mathsf{Prog}|\mathsf{X})$ then reduces to computing a convolution of $P(Z|\mathsf{X})$ and $Q(Z|\mathsf{Prog})$, which is computationally trivial when both distributions are multivariate Gaussian, leading to a very fast search.

[1]In the paper, we will use the convention that a mathematical object written in a sans-serif font such as X represents an observed value, while an italicized object such as X represents a random variable. Hence, $P(X)$ refers to the distribution of random variable X, while $P(\mathsf{X})$ refers to the likelihood of observing value X for random variable X.

Our contributions. Key contributions of our work are:

- We introduce the problem of contextualized code search. While code search has been studied for a long time (see Section 3 of the paper), prior efforts have typically been powered by user-supplied queries. In contrast, in CCS, the query is implicit, and inferred by the surrounding context. We are the first to study code search using this type of implicit query.
- We present a unified probabilistic framework in which a disparate, multi-modal set of contextual evidences X can be synthesized into a posterior distribution $P(\mathsf{Z}|\mathsf{X})$ over the unknown specification for the code being searched for. This distribution encodes the uncertainty inherent in search.
- We consider how to design the learning problem to ensure that search can happen quickly.
- Finally, we experimentally evaluate our tool for CCS (called CODEC) over a corpus consisting of nearly one billion lines of code. We show experimentally that a 16-GPU machine can be used to search our database size of 27.9 million Java methods in little over a second.

2 Example Application

In this section, we give a more detailed example of CCS, via a short case study that demonstrates our tool, called CODEC (Contextualized cODe sEarCh). We call our system CODEC to emphasize the synthesis-based approach to code search: the system learns to encode the context and decode that encoding into a program, rather than simply learning to encode contexts and programs into a latent space.

Consider the following unfinished user interface code:

```
import javax.swing.*;
class MyGuiAppl{

  /**
  create a new frame
  */
  public JFrame ?(? a){
    __CODE_SEARCH__; }}
```

CODEC extracts the class name `MyGuiAppl`, the Javadoc text for the method with the missing body ("`create a new frame`"), as well as the desired return type (`JFrame`) and the name of the formal parameter (`a`). Since no method name and no formal parameter type are given, these are ignored. CODEC searches a database of code fragments and returns the following code in its top few results:

```
/**
 * Creates a new UserInterface object.
 * @param title the title
 * @return the j frame
 */
public JFrame createFrame(final String
  title){
  JFrame frame=new JFrame(title);
  return frame; }
```

At this point, the programmer accepts this suggestion, and uses the search result to replace the incomplete code fragment. Next, the programmer adds the following to the method:

```
/**
 create button
*/
public ? ?(? a){
  __CODE_SEARCH__; }
```

CODEC now analyzes the entire class (including the new method just added), as well as the evidence present in the incomplete method (in this case, only the parameter name `a` is present) and returns the code among the top few results:

```
private JButton makeButton(String arg){
  JButton button=new JButton(arg);
  return button;
}
```

Again, the programmer accepts this result and uses it to replace the fragment. Now, the programmer adds the following method:

```
public void actionClose(JButton a,
  JFrame f){
  __CODE_SEARCH__; }
```

CODEC uses all of the code so far to power the search, including the header for the incomplete method. The top result is:

```
private void setCloseSurrogateButtonAction
  (JButton closeSurrogateButton,JFrame guiFrame){
  closeSurrogateButton.addActionListener
    (new ActionListener() {
    public void actionPerformed(ActionEvent event){
      closeString = CLOSE_UI_EXIT_SURROGATE;
      guiFrame.dispatchEvent(new WindowEvent
        (guiFrame,WindowEvent.WINDOW_CLOSING));
      if (closingWindow) {surrogate.userExit=true;}
      else {closeString=
        CLOSE_UI_SURROGATE_KEEPS_RUNNING;}
    }});
}
```

Throughout the process, the programmer expended little effort to use the tool. No explicit queries were formulated beyond the development of the skeleton of the class. CODEC exclusively uses the context to anticipate what sort of example codes might be useful for the developer to consider.

3 Related Work

Code search has been long-studied. Early works were information retrieval (IR) based. Classic methods include CodeBroker [49], which assessed similarity using comments. Stratcona [21] used inheritance, method calls, and data types to compute similarity. XSnippet [36] used parent classes and type information, and Bajracharya et al. [9] implement keyword-based API pattern search. More modern IR-based code search engines such as Koders [4], Krugle [5], Codase [1], and Sourcerer [8] use text and graph-based search ranking.

Another line of work powers code search using semantic or syntactic constraints. Prospector [28] searches based on return types and types used in the code. JSearch [39], Little et al. [26], and PARSEWeb [44] search using ASTs and API call sequences. Reiss et al. [34] and CodeGenie [25] use test cases, contract specifications, and keywords from unfinished code to facilitate search. CodeHow [27] uses keywords from natural language description and reinforces the search with an additional API understanding phase by mapping the keywords to the descriptions available in an online API library. Facoy [23] proposes a code-to-code search methodology for detecting semantically similar code fragments by code alteration.

Modern code search tools use learned embeddings. Sachdev [35] suggests natural language based search using code embeddings and term frequency-inverse document frequency. Bajracharya [9] learns custom program and query embeddings mined from open-source data. [48, 19] map natural language code descriptions and programs to a shared latent space. Lili [31] propose a tree-based convolutional neural network to embed codes. Chen [13] propose a variational-autoencoder-based architecture for code retrieval and summarization. Cambronero [11] does a comparison between the different neural code engines that use natural language for program search. A recent work by Wan [47] proposes using abstract syntax trees (ASTs) for a more accurate representation of programs, to facilitate natural language search.

Recently there has also been interest in using deep learning-based techniques in other applications of software engineering. Neural-machine translation-based approaches have found application in detecting code clones in software repositories [12, 45, 50]. Our work has connections with recent attempts at using learning-based methods for program synthesis [20]. The idea of using deep neural models for code completion in IDEs has also been popular recently. Pythia [42] is an API recommendation engine that predicts likely API calls. Pythia is integrated as part of Visual Studio for Python. Other works [10, 33, 15] use ML to guide program synthesis. BAYOU [32] synthesizes programs into a high-level representation; the SKETCH language (see Figure 2) was proposed in that paper and much of our statistical model was borrowed from Bayou. Program context has also been used to guide synthesis [22]. One of our contributions is bridging the gap between synthesis and search.

The idea of using different program components (return types, API call sequences, parent class information, and so on) as context for judging programmer intent have been widespread [36, 28, 44, 25, 21]. Much recent work has applied deep learning for code search [19, 35, 48]. However, this latter category of methods has generally been restricted to search based upon natural language specifications (for example, using the information contained in a JavaDoc to learn how to relate text to code). To the best of our knowledge, our efforts are the first to use neural methods to power search using context.

4 The CODEC System

In this section, we describe the design and implementation of CODEC at a high level. A pictorial representation of the CODEC system is shown in Figure 1. The system has five components. Note that while the current implementation of CODEC is specific to Java, extension to other programming languages is straightforward.

(1) Context extractor. This component accepts a Java program, and uses the Eclipse compiler [2] to parse it into an AST. From the program AST, each program fragment and surrounding context are extracted. Processing a Java program with the context extractor results in a set of $(\mathsf{X}, \mathsf{Prog})$ pairs—that is, a set of (context, code fragment) pairs.

(2) Decompiler. Modern Java is an exceedingly complex language, and many of the lower-level details associated with a Java code fragment are likely unimportant for deciding whether the code fragment answers a particular query. Thus, CODEC decompiles each code fragment Prog into a simpler programming language called SKETCH that captures the essence of the Java fragment: API calls, types, general code "shape" (that is control flow and nesting) but ignores lower-level details such as variables and computation of expressions. The decompiler is realized by a function α, such that the SKETCH program $\mathsf{Y} = \alpha(\mathsf{Prog})$.

(3) Learner. Given a code corpus to be indexed for search, all of the programs are fed into the context extractor, and the resulting code fragments are decompiled into SKETCH codes. A subset of the decompiled codes is used to create the training set $D_{trn} = \{\mathsf{X}_i, \mathsf{Y}_i\}_{i=1...n}$. These pairs are then used to power a maximum likelihood estimation, where the goal is to choose the parameter set θ^* as

$$\theta^* = \text{argmax}_\theta \sum_i \log P(\mathsf{Y}_i|\mathsf{X}_i, \theta)$$

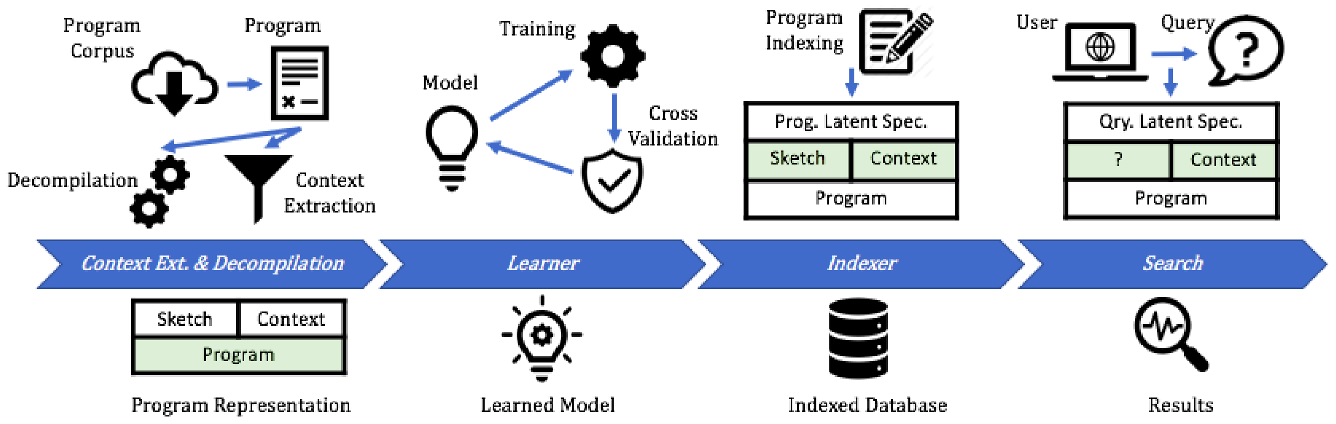


Figure 1: Schematic of the CODEC system.

(4) Indexer. Once the parameter set θ has been learned, each SKETCH code Y to be indexed is then transformed into an intermediate representation $\mathsf{Y}' = h(\mathsf{Y}, \theta)$ that makes it very fast to compute $P(\mathsf{Y}|\mathsf{X}, \theta)$ for a context query X. The set $D_{srch} = \{\mathsf{Y}'_i, \mathsf{Prog}_i\}_{i=1...n}$ where $\mathsf{Y}'_i = h(\alpha(\mathsf{Prog}_i))$ is then distributed across the set of servers used to power the search.

(5) Distributed search engine. Finally, at query time, query context X_q is automatically extracted from a partially-completed program, and D_{srch} is processed to compute the top K $(\mathsf{Y}'_i, \mathsf{Prog}_i)$ pairs from D_{srch} that maximize $P(\mathsf{Prog}_i|\mathsf{X}_q, \theta)$, which are presented to the user.

In the next few sections of the paper, we describe a few of these components in more detail, followed by an in-depth description of the statistical model $P(\mathsf{Prog}|\mathsf{X}, \theta)$ used to power the search, as well as how this model is learned from D_{trn}.

5 Context Extraction and Decompilation

In this section, we describe program context extraction and decompilation in a bit more detail.

5.1 Context Extraction

Given a Java program, the first step is to parse the program, and extract the various *evidences* that serve as the context for all of the code fragments that will be extracted from the code. In CODEC, some evidences are extracted on a per-class basis, so that all of the fragments from the same class will share the same evidence. The class-wide evidences extracted are:

(1) *Class Name*; name of the class split using camel-case; each is encoded with a one layer GRU-RNN network.
(2) *Class types*; types of the instance variables in the same class as the query method; each is encoded with a one-hidden-layer fully-connected network.
(3) *Surrounding Methods*; methods within the same class; each consisting of:
(a) *Return type*; encoded with a one-hidden-layer network.
(b) *Input parameter list*; sequence of (formal parameter, variable name) pairs encoded using a two-layer GRU-RNN, where the variable names are split using camel-case, encoded using a GRU-RNN, and concatenated with each formal parameter.
(c) *Method name*; name of the method split using camel-case; each is encoded with a one layer GRU-RNN network.
(d) *API call sequences*; API call sequences are extracted from methods within the same class; encoded with a one layer GRU-RNN network.

We also use four types of evidence from the header of the missing method (if available):

(1) *JavaDoc*; English text of JavaDoc associated with the method, lemmatized and stop words removed. Encoded using a bidirectional GRU-RNN.
(2) *Method name* for method containing the missing code; split using camel-case and encoded with a single-layer GRU-RNN.
(3) *Return type* of method with the missing code encoded with a one hidden-layer network.
(4) *Input parameter list*, of method with missing code; including formal parameter type and name, split using camel-case, encoded similarly to the input parameters from surrounding methods.

Finally, we use four types of evidence from *within* the (partial) code that is being searched for. These may be available if a fragment of the code has already been written. They are:

(1) *API calls*; these are the calls in the code; each is encoded with a one-layer network.
(2) *API call sequences*; these are extracted via symbolic execution of the code; each is encoded using a GRU-RNN.
(3) *Types*; these are the API types in the code; each is encoded with a one-layer network.
(4) *Keywords*; English-like words extracted from fully qualified name of the classes inside the method body, combined with English words appearing in types and API calls; each is encoded with a one-layer network.

5.2 Decompilation

As described previously, we believe that it is problematic to search for a code fragment in a complicated language such as Java directly. Every high-level language likely contains details (such as arithmetic operations) that are of little use during search, and likely make it difficult to learn how to relate queries with search results, obscuring the important facets of the code. A complicated and mature language such as Java is especially problematic. Consider the BAYOU program synthesis system [32]. Using a neural network to synthesize into a sketching language (and then using classical, AI-style search to complete the program) Bayou showed around 50% accuracy (in terms of being able to reproduce the "correct" result in the top-10 programs synthesized), whereas a version of Bayou that synthesized

$$
\begin{array}{lcl}
\mathsf{Y} & ::= & \mathsf{Y}_{api}; \mathsf{Y}_{ret}; \mathsf{Y}_{fp} \\
\mathsf{Y}_{ret} & ::= & \tau_r \\
\mathsf{Y}_{fp} & ::= & (\tau_{fp_0}, \ldots, \tau_{fp_n}) \\
\mathsf{Y}_{api} & ::= & \textbf{skip} \mid \textbf{call}\ \mathsf{Cexp} \mid \mathsf{Y}_1; \mathsf{Y}_2 \mid \\
 & & \textbf{if}\ \mathsf{Cseq}\ \textbf{then}\ \mathsf{Y}_1\ \textbf{else}\ \mathsf{Y}_2 \mid \\
 & & \textbf{while}\ \mathsf{Cseq}\ \textbf{do}\ \mathsf{Y}_1 \mid \textbf{try}\ \mathsf{Y}_2\ \mathsf{Catch} \\
\mathsf{Cexp} & ::= & \tau_{a_0}.\alpha(\tau_{a_1}, \ldots, \tau_{a_k}) \\
\mathsf{Cseq} & ::= & \text{List of } \mathsf{Cexp} \\
\mathsf{Catch} & ::= & \textbf{catch}(\tau_{a_1})\ \mathsf{Y}_1\ \ldots\ \textbf{catch}(\tau_{a_k})\ \mathsf{Y}_k
\end{array}
$$

Figure 2: Grammar for the SKETCH language. τ indicates a Java type, α is a method call name. SKETCH extends the BAYOU sketching language [32]

directly into a subset of Java was able to achieve less than 10% in terms of top-10 accuracy.

Hence, given a class, once we extract a code fragment Prog from the class, we *decompile* Prog into a simplified representation $\mathsf{Y} = \alpha(\mathsf{Prog})$. This representation is designed to retain the facets of the code that are likely to be important to a user during search: API calls and basic control flow, but ignores the rest. The grammar for the abstraction language SKETCH is given above in Figure 2. Translating from a parsed Java AST to a SKETCH AST is straightforward.

For an example of this, consider the following method:

```
void read(File file) {
  FileReader fr1;
  BufferedReader br1;
  fr1 = new FileReader(file);
  br1 = new BufferedReader(fr1);
  while ((br1.readLine()) = null) return;
```

This is decompiled into the following:

```
FileReader.FileReader (File)
BufferedReader.BufferedReader (FileReader)
while
  BufferedReader.readLine ()
do
  skip
```

Note the `skip`, which reflects the fact that the `while` loop has no body.

6 Statistical Model

At the heart of the CODEC system is the statistical model powering the search, embodied by the distribution function $P(\mathsf{Prog}|\mathsf{X}, \theta)$. During search, a set of evidences is extracted to represent the program context X, and then the few database fragments that maximize the value of this function are selected.

There are many possible choices for the model $P(\mathsf{Prog}|\mathsf{X}, \theta)$. We begin with the simple statistical model pictured in Figure 3, borrowed from conditional program generation [32] . Crucially, we assume a latent specification Z for the missing code fragment as well as the surrounding context. This specification captures the programmer intent, and conditioned upon Z, both the program context X as well as the sketch Y and the program fragment itself is generated.

As intimated in the introduction, there is a key benefit to using such a model to power code search. During search, the latent variable Z provides similar functionality to the latent-space embedding used in traditional, embedding-based methods [37]. However, there is a key difference. As Z is a true random variable, it has no single value. A learner, given a large number of $(\mathsf{X}, \mathsf{Prog})$ pairs from

$$\begin{tikzcd} & Z \arrow[dl] \arrow[dr] & & \\ X & & Y \arrow[r] & Prog \end{tikzcd}$$

Figure 3: Bayes net for X, Y, Z and $Prog$.

which to learn $P(\mathsf{Prog}|\mathsf{X}, \theta)$, must learn to accurately generate Prog, despite of the uncertainty in Z embodied by the distribution $P(Z)$. This may alleviate some of the problems with over-fitting one might except when using a more traditional neural encoding.

As depicted in Figure 3, Z is generated first, and based upon the programmer intent captured by Z, the evidences X in the surrounding context are generated, as well as the sketch Y. Once the sketch is generated, the program Prog is generated based on the sketch. Thus, the joint distribution $P(X, Y, Z, Prog|\theta)$ factorizes as $P(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \mathsf{Prog}|\theta) =$

$$P(\mathsf{Z})P(\mathsf{X}|\mathsf{Z})P(\mathsf{Y}|\mathsf{Z})P(\mathsf{Prog}|\mathsf{Y}).$$

(Note that we drop the parameter θ from each distribution function for simplicity).

7 Search Under the Model

7.1 Applying the Model for Search

During search, we are given a context X, and we wish to choose a database code fragment Prog_i to maximize $P(\mathsf{Prog}_i|\mathsf{X}, \theta) =$

$$\int_{\mathsf{Y}} P(\mathsf{Prog}_i|\mathsf{Y}, \theta) \int_{\mathsf{Z}} P(\mathsf{Y}|\mathsf{Z}, \theta)P(\mathsf{Z}|\mathsf{X}, \theta)d\mathsf{Z}d\mathsf{Y}$$

This looks difficult. However, we can simplify this expression by assuming that no code Prog_i is associated with more than one sketch; this is a fairly weak assumption, and is implied by the fact that we can decompile each code into its unique sketch using the decompilation function α. Given this assumption, the distribution function $P(\mathsf{Prog}_i|\mathsf{Y}, \theta)$ gives non-zero likelihood for only one Y value for a given program Prog_i. Hence, if we let $\mathsf{Y}_i = \alpha(\mathsf{Prog}_i)$, $P(\mathsf{Prog}_i|\mathsf{X}, \theta)$ can be re-written as:

$$P(\mathsf{Prog}_i|\mathsf{X}, \theta) = P(\mathsf{Prog}_i|\mathsf{Y}_i, \theta) \int_{\mathsf{Z}} P(\mathsf{Y}_i|\mathsf{Z}, \theta)P(\mathsf{Z}|\mathsf{X}, \theta)d\mathsf{Z}$$

In our implementation of CODEC we simplify this further by assuming that the process of creating a code from a sketch is deterministic, and that $P(\mathsf{Prog}|\mathsf{Y}, \theta) \neq 0$ if and only if $\mathsf{Y} = \alpha(\mathsf{Prog}_i)$, so that:

$$P(\mathsf{Prog}_i|\mathsf{X}, \theta) = P(\mathsf{Y}_i|\mathsf{X}, \theta) = \int_{\mathsf{Z}} P(\mathsf{Y}_i|\mathsf{Z}, \theta)P(\mathsf{Z}|\mathsf{X}, \theta)d\mathsf{Z}$$

Though this assumption is not necessary, it seems to give good results, and it means there is no need to define $P(Prog|\mathsf{Y}, \theta)$.

7.2 Making Search Fast

$P(\mathsf{Y}_i|\mathsf{X}, \theta)$ will need to be evaluated for millions or billions of Y_i values stored in a database, in response to a query X. For this reason, evaluating $P(\mathsf{Y}_i|\mathsf{X}, \theta)$ needs to be very, very fast. Without a careful choice of the various distribution functions to allow for a fast, closed-form evaluation of $P(\mathsf{Y}_i|\mathsf{X}, \theta)$, search will not be practical.

To ensure that we are able to have a closed-form evaluation of this function, we begin by expanding the function using Bayes' rule:

$$\int_{\mathsf{Z}} P(\mathsf{Y}|\mathsf{Z},\theta)P(\mathsf{Z}|\mathsf{X},\theta)d\mathsf{Z} \approx$$
$$P(\mathsf{Y}|\theta) \times \int_{\mathsf{Z}} \frac{P(\mathsf{Z}|\mathsf{X},\theta)P(\mathsf{Z}|\mathsf{Y},\theta)}{P(\mathsf{Z}|\theta)} d\mathsf{Z}$$

It can be shown that as long as all of the distribution functions within the integral ($P(\mathsf{Z}|\mathsf{X},\theta)$, $P(\mathsf{Z}|\mathsf{Y},\theta)$, and $P(\mathsf{Z}|\theta)$) are multivariate Gaussian, this can be integrated analytically, with very little computational cost.[2] For example, assume Z is scalar-valued,[3] and assume the mean and variance of $P(\mathsf{Z}|\mathsf{X},\theta)$ evaluate to μ_{X} and σ^2_{X}, respectively. Also assume that the mean and variance of $P(\mathsf{Z}|\mathsf{Y},\theta)$ evaluate to μ_{Y} and σ^2_{Y}, respectively. These can be computed offline, during database preparation, and stored in the database along with the programs to be searched.

Let $a_{\mathsf{X}} = -(2\sigma^2_{\mathsf{X}})^{-1}$ and define a_{Y} similarly. Likewise, let $b_{\mathsf{X}} = \mu_{\mathsf{X}}\sigma_{\mathsf{X}}^{-2}$ and define b_{Y} similarly. Finally, we let $P(\mathsf{Z}|\theta)$ be unit multivariate Normal. Then we have:

$$\log P(\mathsf{Y}|\mathsf{X},\theta) = \log P(\mathsf{Y}|\theta) + \frac{1}{2}\log\left(\frac{-2a_{\mathsf{X}}a_{\mathsf{Y}}}{a_{\mathsf{X}} + a_{\mathsf{Y}} + 1/2}\right) + \frac{b^2_{\mathsf{X}}}{4a_{\mathsf{X}}} + \frac{b^2_{\mathsf{Y}}}{4a_{\mathsf{Y}}} - \frac{(b_{\mathsf{X}} + b_{\mathsf{Y}})^2}{4(a_{\mathsf{X}} + a_{\mathsf{Y}} + 1/2)}$$

Except for the quantity $\log P(\mathsf{Y}|\theta)$, this is all trivial to evaluate quickly, at search time, and so it is computationally efficient to check $\log P(\mathsf{Y}_i|\mathsf{X},\theta)$ for each Y_i in the database as long as we have pre-computed and stored three values:

1. The term $\log P(\mathsf{Y}_i|\theta)$, which measures the bias towards returning a particular program, can be computed offline via Monte Carlo integration over the latent variable Z, using $P(\mathsf{Z}|\mathsf{Y}_i,\theta)$ as a proposal distribution for importance sampling [7].

2. The mean and variance of $P(\mathsf{Z}|\mathsf{Y}_i,\theta)$. In the univariate case, these will be scalar values, and in the multivariate case these will be vector-valued (assuming a diagonal covariance matrix for $P(\mathsf{Z}|\mathsf{Y}_i,\theta)$, as we will assume subsequently).

Together, these values constitute Y'_i. In parallel across multiple compute nodes, CODEC stores the set of programs to search $D_{srch} = \{\mathsf{Y}'_i, \mathsf{Prog}_i\}_{i=1...n}$. In response to a query context X, the few Prog_i values for which $Y_i = \alpha(\mathsf{Prog}_i)$ maximizes the $\log P(\mathsf{Y}|\mathsf{X},\theta)$ value are returned to the user.

8 Distribution Families Used

So far, we have not committed to any particular distribution functions, other than stating that our basic statistical model as shown in Figure 3, and stating that for practical reasons—we want the per-program computation time to be tiny during search—we will desire each of $P(Z|\theta)$, $P(Z|\mathsf{X},\theta)$ and $P(Z|\mathsf{Y},\theta)$ to be multivariate Gaussian. In this section, we discuss each of these distributions in more detail.

The prior on Z**:** $P(Z|\theta)$. Ensuring that $P(Z|\theta)$ is multivariate Gaussian is easy; since Z is a latent variable, we simply define $Z \sim$ Normal $(\vec{0}, \mathbf{I})$.

Z **conditioned on the sketch**: $P(Z|\mathsf{Y},\theta)$. Unfortunately, marrying the natural set of conditional dependencies depicted in Figure 3 with the desire for computational efficiency is not easy for the other distribution functions. In particular, ensuring Normality for $P(Z|\mathsf{Y},\theta)$ is, practically speaking, not possible. Since a given Y is a complex, tree-valued object (a set of recursive rule firings in a grammar) we wish to use a state-of-the-art neural tree decoder to realize $P(Y|\mathsf{Z},\theta)$. Several are available [43, 40, 18]; we use a top-down tree LSTM [51] to realize $P(Y|\mathsf{Z},\theta)$. However, using such a decoder almost assuredly means that $P(Z|\mathsf{Y},\theta)$ is *not* Gaussian. As we describe in the next section of the paper, we address this by using variational methods [16] to simultaneously learn a Gaussian approximation $Q(Z|\mathsf{Y},\theta)$ for $P(Z|\mathsf{Y},\theta)$, and to force $P(Z|\mathsf{Y},\theta)$ to be approximately Gaussian. Then $Q(Z|\mathsf{Y},\theta)$ can then be used in place of $P(Z|\mathsf{Y},\theta)$ to ensure fast search. We call $Q(Z|\mathsf{Y},\theta)$ a "reverse encoder" for a decompiled program Y, as it reverses the generative process to encode the program. In our implementation, $Q(Z|\mathsf{Y},\theta)$ is realized as a neural tree encoder [51], that is used to encode Y into a mean vector and covariance matrix of a Gaussian distribution.

Z **conditioned on the context**: $P(Z|\mathsf{X},\theta)$. This situation is a bit different. We could also use variational methods to allow for a Gaussian approximation to $P(Z|\mathsf{X},\theta)$, but instead we borrow the formulation from [32] to ensure that $P(Z|\mathsf{X},\theta)$ *is* Gaussian, at least under certain restricted circumstances.

Specifically, we assume that the context X is partitioned into a set of different sets of evidences, according to the type of evidence. Let $\mathsf{X}_{j,k}$ refer to the kth instance of the jth type of evidence in X. Various possible types of evidence are discussed in Section 5.1, and include: class variable types, other method signatures, documentation, etc. Assume a neural function $f_{j,\theta}$ for the jth evidence type that maps some representation of the evidence to some location in $\mathbb{R}^m$. Then let:

$$P(\mathsf{X}|\mathsf{Z},\theta) = \prod_{j,k} \text{Normal}\left(f_{j,\theta}(\mathsf{X}_{j,k})|\mathsf{Z}, \mathbf{I}\sigma^2_j\right)$$

Effectively, we assume that the encoded location of each piece of evidence has been sampled from a normal distribution with mean Z. Each evidence is sampled using a different variance. Higher variance corresponds to an evidence that is less closely related with the functionality of the code fragment being searched for.

If each mapping function is one-to-one and onto, then from Normal-Normal conjugacy, it follows that [32]:

$$P(\mathsf{Z}|\mathsf{X},\theta) = \mathcal{N}\left(\mathsf{Z}|\frac{\sum_{j,k}\sigma_j^{-2}f_{j,\theta}(\mathsf{X}_{j,k})}{1+\sum_j|\mathsf{X}_j|\sigma_j^{-2}}, \frac{1}{1+\sum_j|\mathsf{X}_j|\sigma_j^{-2}}\mathbf{I}\right)$$

Here, $|\mathsf{X}_j|$ refers to the size of the jth subset of evidence.

Note that Normal-Normal conjugacy will not hold if some mapping function is not one-to-one and onto. In practice, this will *not* hold. As an example, we may employ a bidirectional RNN [38] to encode English text in a JavaDoc comment into $\mathbb{R}^m$; such a function will not be one-to-one and onto. Still, in practice things seem to work well, and intuitively the fact that the function is not one-to-one

[2]Intuitively, this is not surprising, as the Gaussian distribution is simply an exponentiated quadratic function, and every exponentiated quadratic function is a Gaussian distribution, up to a constant factor. Hence, multiplying and dividing Gaussian distribution functions results in an exponentiated quadratic function, which is then also a Gaussian distribution function, up to a constant factor. Integrating over any distribution function results in a value of one, leaving only the constant factor. If we can compute that constant analytically, there is no need to integrate.

[3]An extension to multiple dimensions is straightforward and a full derivation with multivariate extension is available in the Appendix.

and onto may only be a problem if different evidences tend to be mapped to the same point in $\mathbb{R}^m$, which seems not to happen in practice. As intimated above, an alternative is to instead resort to a variational approximation for $P(\mathsf{Z}|\mathsf{X}, \theta)$ as well as $P(\mathsf{Z}|\mathsf{Y}, \theta)$, at the cost of making the learning problem somewhat more complex.

9 Training the Model

We have two goals. First, we wish to make sure that the reverse encoder $Q(Z|\mathsf{Y})$ is a reasonable proxy for $P(Z|\mathsf{Y})$. Second, we wish to ensure that the log-likelihood of our data set is maximized and that $\sum_i \log P(\mathsf{Y}_i|\mathsf{X}_i, \theta)$ has a large value. In the remainder of this section, we again drop the set of model parameters θ when convenient to simplify the presentation.

To begin, we note that we want the so-called "reverse encoder" $Q(Z|\mathsf{Y})$ to closely match the true posterior $P(Z|\mathsf{Y})$ for $\mathsf{Y} \sim P(Y)$. We can ensure this by minimizing the KL divergence between them. We begin our derivation of the learning problem by expanding this KL divergence. For a given Y:

$$D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{Y})\big) = \int_Z Q(\mathsf{Z}|\mathsf{Y})\big[\log Q(\mathsf{Z}|\mathsf{Y}) - \log P(\mathsf{Z}|\mathsf{Y})\big]d\mathsf{Z}$$

Note that $P(Z|\mathsf{Y})$ cannot be evaluated directly, as it is not one of the three distributions defined in the previous section. Hence, we expand it using Bayes' Rule:

$$\begin{aligned}D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{Y})\big) &= \int_Z Q(\mathsf{Z}|\mathsf{Y}) \times \big[\log Q(\mathsf{Z}|\mathsf{Y}) - \log P(Y|\mathsf{Z}) \\ &\qquad - \log P(\mathsf{Z}) + \log P(\mathsf{Y})\big]d\mathsf{Z} \\ &= \log P(\mathsf{Y}) + \int_Z Q(\mathsf{Z}|\mathsf{Y}) \times \big[\log Q(\mathsf{Z}|\mathsf{Y}) \\ &\qquad - \log P(\mathsf{Y}|\mathsf{Z}) - \log P(\mathsf{Z})\big]d\mathsf{Z}\end{aligned}$$

Expanding this further we have:

$$\begin{aligned}&D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{Y})\big) \\ &= \log P(\mathsf{Y}) + \int_\mathsf{Z} Q(\mathsf{Z}|\mathsf{Y}) \times \big[\log Q(\mathsf{Z}|\mathsf{Y}) - \log P(\mathsf{Z}|\mathsf{X})\big] \\ &+ \int_\mathsf{Z} Q(\mathsf{Z}|\mathsf{Y}) \times \big[\log P(\mathsf{Z}|\mathsf{X}) - \log P(\mathsf{Y}|\mathsf{Z}) - \log P(\mathsf{Z})\big]d\mathsf{Z} \\ &= \log P(\mathsf{Y}) + D_{KL}(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})) \\ &+ \int_\mathsf{Z} Q(Z|\mathsf{Y})\big[\log P(Z|\mathsf{X}) - \log P(Y|\mathsf{Z}) - \log P(Z)\big]d\mathsf{Z} \\ &= \log P(\mathsf{Y}) + D_{KL}(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})) + \int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \times \\ &\quad \frac{Q(\mathsf{Z}|\mathsf{Y})}{P(\mathsf{Z}|\mathsf{X})} \times \big[\log P(\mathsf{Z}|\mathsf{X}) - \log P(\mathsf{Y}|\mathsf{Z}) - \log P(\mathsf{Z})\big]d\mathsf{Z}\end{aligned}$$

Assume for a moment that $Q(Z|\mathsf{Y}) \approx P(Z|\mathsf{X})$ for $(\mathsf{X}, \mathsf{Y}) \sim P(X, Y)$ so their ratio is 1. While this is by no means guaranteed, we will reconsider this assumption later. Then we have:

$$\begin{aligned}D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{Y})\big) \approx \\ \log P(\mathsf{Y}) + D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})\big) \\ + \int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \times \big[\log P(\mathsf{Z}|\mathsf{X}) - \log P(\mathsf{Z})\big]d\mathsf{Z} \\ - \int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \times \log P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}\end{aligned}$$

This can be rewritten as,

$$\begin{aligned}D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{Y})\big) \approx \\ \log P(\mathsf{Y}) + D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})\big) \\ + D_{KL}\big(P(Z|\mathsf{X})\|P(Z)\big) - \int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \log P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}\end{aligned}$$

And so,

$$\begin{aligned}\log P(\mathsf{Y}) - D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{Y})\big) \approx \\ - D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})\big) \\ - D_{KL}\big(P(Z|\mathsf{X})\|P(Z)\big) + \int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \log P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}\end{aligned}$$

This implies that if we maximize the expected value of the RHS of the above approximation with respect to $(\mathsf{X}, \mathsf{Y}) \sim D_{trn}$ we will simultaneously perform a maximum likelihood estimation (maximizing the data log-likelihood $\log P(Y)$) and maximize the quality of the reverse encoder $Q(Z|\mathsf{Y})$ by making it a good approximation for $P(Z|\mathsf{Y})$. Then, in the end, to learn the model, we choose θ so as to maximize the following:

$$\begin{aligned}\mathrm{E}_{(\mathsf{X},\mathsf{Y})\sim D_{trn}}\Big[&- D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})\big) \\ &- D_{KL}\big(P(Z|\mathsf{X})\|P(Z)\big) \\ &+ \int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \log P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}\Big]\end{aligned}$$

This is easily possible via gradient descent. We sample (X, Y) pairs from D_{trn}, and for each pair, take a gradient step to minimize the value of the expression. Fortunately, since each $Q(Z|\mathsf{Y})$, $P(Z|\mathsf{X})$, and $P(Z)$ is multivariate Gaussian, there is a closed form for the pairwise KL divergence between them for which the gradient is easily computed using a platform such as TensorFlow.

One more complicated issue is to compute the gradient of the integral $\int_\mathsf{Z} P(\mathsf{Z}|\mathsf{X}) \log P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}$. Note that this can be re-written as $\mathrm{E}_{(\mathsf{Z})\sim P(Z|\mathsf{X})} \log P(\mathsf{Y}|\mathsf{Z})$. Maximization of this quantity is amenable to the standard "reparameterization trick" [16] used when training variational autoencoders. That is, we may sample Z from a standard Normal distribution, and then push the transformations represented by $P(Z|\mathsf{X}, \theta)$ and $Q(Z|\mathsf{Y}, \theta)$ into the quantity we are taking the expectation of, in order to back-propagate through the transformations.

Finally, we re-visit our assumption that $\frac{Q(\mathsf{Z}|\mathsf{Y})}{P(\mathsf{Z}|\mathsf{X})} \approx 1$. While not guaranteed, the argument for the validity of this simplifying assumption rests on the fact that the resulting maximization problem explicitly attempts to minimize the KL divergence term in our derivation, $D_{KL}\big(Q(Z|\mathsf{Y})\|P(Z|\mathsf{X})\big)$ for $(\mathsf{X}, \mathsf{Y}) \sim D_{trn}$. As this divergence is minimized during learning, the approximation will become increasingly valid.

Relationship to variational autoencoders. There is a resemblance between the material in this Section and the methods used to train variational autoencoders (VAEs) [16]. However, there are key differences. Given a data Y, a VAE is meant to learn a model of the form $P(Y = \mathsf{Y}) = \int P(\mathsf{Z})P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}$, whereas our goal is to learn a conditional model of the form $P(Y = \mathsf{Y}|\mathsf{X}) = \int P(\mathsf{Z}|\mathsf{X})P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z}$. In the case of a VAE, a variational distribution $Q(Z|\mathsf{Y})$ is used to approximate $P(Z|\mathsf{Y})$ to make training possible. In our case, because of the availability of the evidence set X (as in conditional program generation [32]) we do not need this approximation; it would be possible to learn $P(Y = \mathsf{Y}|\mathsf{X})$ directly without any variational approximation, using a bound based on Jensen's inequality. However, we also have to simultaneously learn an approximation

$Q(Z|\mathsf{Y})$ for $P(Z|\mathsf{Y})$ to use during code search. This results in a learning problem formulation that differs in important ways from VAEs, conditional VAEs [41], and conditional program generation.

10 Evaluation

There are three parts to our experimental study of CODEC.

In the first, we perform a quantitative study where "holes" are created in a large number of real-life programs, downloaded from GitHub, by removing a method body from each program. The method bodies are then mixed into a large database of method bodies, and we measure CODEC's ability to retrieve the correct method body from the database

In the second, we perform a qualitative user study where we ask 13 programmers to rate the quality of the results returned by CODEC and a number of competitive methods.

In the third, we examine the runtime efficiency of our so-called "reverse encoder," that enables a fast, analytic approximation to the likelihood that each program was generated by the search context.

10.1 Quantitative Study

10.1.1 Experimental Setup

Data used. We collect all the public, licensed projects available in Github [3]. This is a total of 8.71M java files. We use the Eclipse Java DOM Driver [2] to extract abstract syntax trees for a total of 27.9M methods with at-least one API call to the Java JDK. Out of these, we trained our statistical model (as well as the competitive models described below) on 2.32M randomly-sampled Java files, amounting to 6.7M methods. 21M Java methods were extracted from the remaining 6.39M files, and the methods extracted were indexed using the learned model.

Model training details. The latent space occupied by Z in our implementation is 256 dimensions. We have used 256 units in each of our neural architectures and a single hidden layer for our tree encoder and decoder. We used a batch-size of 128 methods during training and a learning rate 0.0001 for the Adam gradient descent algorithm [24]. Our deep model was trained on top of Tensorflow [6] using an Amazon EC2 `p2.xlarge` machine powered with an NVIDIA K80 GPU. Training required 200 hours.

Competitive methods. We compare CODEC with three baseline methods. The first two are CodeHow [27] and Deep-Code search [19]. Both of these methods were developed to support code search using natural language. CodeHow finds programs based on keyword matching, using API understanding to reformulate the query for higher accuracy. We modify CodeHow to use keywords from method headers (types, formal parameter names, method names) along with JavaDoc comments. Deep-Code search encodes the JavaDoc and the program (represented as a triplet of method name, sequence of API calls and keywords) into a shared latent space and attempts to minimize the cosine distance between them. We also implement a non-probabilistic version of CODEC that uses the same encoders for various evidences as CODEC, encoding the entire context as a weighted average of the various evidences. This non-probabilistic version uses the same neural architecture as CODEC's reverse encoder to encode the sketch into the latent space, and attempts to maximize the cosine similarity between the encoded context and the encoded sketch.

Retrieval task. We test each method using a set of 100 retrieval tasks. To construct a task, we randomly select a Java file having at least two method bodies from among the 6.4M Java files not used for training. The files selected represent a wide variety of Java applications, from computer networking applications to MySQL database application development to simple file I/O. We remove a random method body containing JavaDoc documentation from the file, and use the context in the remainder of the file to power search using the four different search techniques. The search is considered to be accurate if one or more method bodies that is "equivalent" to the removed-and-searched-for method body are among the top results returned.

Measuring "equivalence." Defining the notion of two codes being equivalent to one another is not straightforward. Determining if two codes produce the same output on all inputs is, in general, undecidable, and a real-life, GitHub-derived Java corpus presents many challenges. For example, a popular code may be replicated any times during its lifetime on GitHub. If the retrieval task returns an older version of the correct method, it is unclear whether this is "equivalent."

In the end, we came up with four different definitions of method equivalence, each of which we examine experimentally: (1) *API match*; two codes are equivalent if two use the same set of JDK API calls. (2) *Sequence match*; two codes are equivalent if the sets of all possible sequences of API calls, extracted using symbolic execution, are the same. (3) *Sketch match*; two codes are equivalent if decompilation into a SKETCH program (see Section 5.2), results in the same code. (4) *Exact match*; two programs are considered to be equivalent if the Java parse tree for the entire method body matches exactly.

Measuring search accuracy. We also consider multiple ways in which these various definitions of equivalence can be used to measure search accuracy; some of our ideas follow related work [19, 27]. Let us assume that we have a total of Q independent queries and **res** be a vector of size Q containing the identities of the method bodies we are searching in the database. Let us assume **Ans** be a matrix of size $Q \times K$, which denotes the identity of the results returned by our system for each of those queries within a pre-defined rank K. We consider three metrics, out of which two metrics are based of the notion of FRank. For a particular search query q, FRank$_q$ is the smallest rank k at which the user finds a code equivalent to the desired result among the top programs.

$$\text{FRank}(\mathbf{res}_q, \mathbf{Ans}_q) = \text{argmin}_k[I(\mathsf{Prog}_{\mathbf{res}_q} \equiv \mathsf{Prog}_{\mathbf{Ans}_{q,k}})]$$

Here, I accepts a boolean value and returns one if it is true, zero if false. Given this, our metrics are:

(1) *SuccessRate@K*, which estimates the probability of finding the intended result within a pre-defined rank K:

$$\text{SuccessRate@K} = \frac{1}{Q}\sum_q [I(\text{FRank}(\mathbf{res}_q, \mathbf{Ans}_q) \leq K)].$$

(2) *Precision@K*, which estimates the fraction of the top K results that are correct. Then:

$$\text{Precision@K} = \frac{1}{KQ}\sum_q \sum_k [I(\mathsf{Prog}_{\mathbf{res}_q} \equiv \mathsf{Prog}_{\mathbf{Ans}_{q,k}})]$$

(3) *MRR* or Mean Reciprocal Ratio, which is simply the average inverse FRank:

$$\text{MRR} = \frac{1}{Q}\sum_q [\frac{1}{\text{FRank}(\mathbf{res}_q, \mathbf{Ans}_q)}]$$

For each metric, a larger value means higher search accuracy.

Table 1: Prediction accuracy comparison from program context.

	SuccessRate@1				SuccessRate@10			
	API Match	Seq Match	Sk. Match	Exact Match	API Match	Seq Match	Sk. Match	Exact Match
CodeHow	0.03	0.02	0.02	0.00	0.12	0.09	0.09	0.06
Deep-Code	0.04	0.02	0.00	0.00	0.11	0.06	0.03	0.02
Non-Prob	0.01	0.01	0.01	0.00	0.02	0.01	0.01	0.00
CODEC	**0.27**	**0.24**	**0.23**	**0.11**	**0.35**	**0.32**	**0.32**	**0.15**

	Precision@10				MRR			
	API Match	Seq Match	Sk. Match	Exact Match	API Match	Seq Match	Sk. Match	Exact Match
CodeHow	0.04	0.03	0.03	0.01	0.13	0.12	0.12	0.10
Deep-Code	0.04	0.02	0.00	0.00	0.15	0.12	0.10	0.10
Non-Prob	0.01	0.00	0.00	0.001	0.11	0.11	0.11	0.09
CODEC	**0.26**	**0.24**	**0.22**	**0.10**	**0.35**	**0.33**	**0.32**	**0.20**

Table 2: Search problems considered in user study.

Id	Program Class	Programming Task
1	Socket	Send data using Client
2	Crypto	Encrypt data using MD5 hash
3	FileUtils	Copy a file to a different location
4	Generic list	Remove item from list
5	GUI-Swing	Add closing button to a frame
6	Conversion	Convert a list of string to HashMap
7	IO	Write data using `OutputStream`
8	String Operations	Check if `String` is palindrome
9	Parser	Parse JSON `String`; put into hash
10	Peeking Iterator	Advance iterator if next exists
11	SQL	Execute a select statement
12	Stopwatch	Return time recorded in milli-seconds
13	ThreadQueue	Get the collection of queued threads
14	WordCount	Split a texttString by delimiter
15	XML Utils	Serialize a XML node into a string

Table 3: p-value at which the hypothesis $H_0^{A,B}$ was rejected during the user study.

	CodeHow	Deep-Code	Non-Prob	CODEC
CodeHow	N/A	0.7160	0.0749	**0.9584**
Deep-Code	0.2477	N/A	0.0224	**0.9378**
Non-Prob	0.9058	0.9699	N/A	**0.9871**
CODEC	**0.0306**	**0.0457**	**0.0084**	N/A

10.1.2 Results and Discussion

Results for each of the four equivalence metrics - SuccessRate@1, SucccessRate@10, Precision@10, and MRR, for each of the four search methods tested, are shown in Table 1. The results suggest that, if the correct method body is available, using CODEC we can expect to obtain an exact match 11% of the time. We feel that this is a quite impressive result, given that CODEC is able to select the correct method body from among 27.9M candidate methods, using only contextual information as well as the JavaDoc comment. The chance of obtaining a "correct" program increases to 23% if success is measured in terms of returning a code that can produce the same set of API call sequences, and to 27% if success is measured in terms of obtaining a program with the same set of API calls.

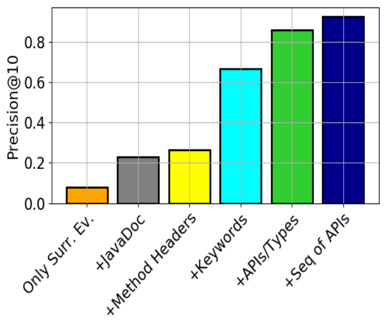
It is interesting that the competitive methods fared so poorly compared to CODEC. No other search methodology had a non-zero exact-match success rate for the top result returned. But even beyond exact-match, CODEC dominates all other methods, over all of the various metrics.

How useful is external context? In a sense, it may not be surprising that CODEC outperforms both CodeHow and Deep-Code search, as both of these are natural-language based, and hence they focus mostly on the JavaDoc comments (though as described, we did try to augment these methods to take into account contextual information as well, at least in a cursory manner). However, it turns out that the explanation for why CODEC has much higher accuracy is not as simple as "it uses more information to power search." To examine this, we repeated the experiment using *only* contextual information external to the method we are searching for (that is, no JavaDoc, and no header for the method being searched for). We then added in JavaDoc comments, the method header, and continued to add information about the internals of the method, such as the API calls present (those internals are not used in the results of Table 1). Results are shown in Figure 4. What we find is that although context and the method header do seem to add significantly to the search quality, unless one uses method internals, JavaDoc comments provide for the bulk of the accuracy. Note that JavaDoc comments are available to all competitive methods.

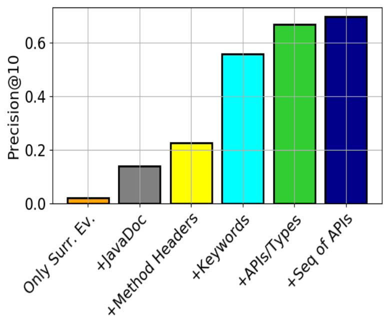
Influence of method internals. We see a major jump in accuracy while including evidences such keywords, API calls, and types from within the body of the method, which may be available in a partially-completed or completed method whose body is used to power search. Figure 4 shows that if such evidences are available, we see a significant increase in accuracy, from about 20% (in terms of sketch match) for only external evidences, up to nearly 70% if a complete set of within-method evidences are supplied.

Comparison with non-probabilistic CODEC. The results so far suggest that the presence of more data is not the only reason for CODEC's success. To dive deeper into this, we consider in detail how CODEC compares with its non-probabilistic version. Going into our experiments, we suspected that the regularization provided by the prior on Z would be useful; because Z is not known during training, any Z value in a neighborhood must have a reasonable likelihood of decoding into the correct sketch Y. Intuitively, this will force programs that are embedded close to one another to have a reasonable similarity in terms of SKETCH syntax, as they must share likely Z values. This should help boost generalization ability, and hence accuracy. In comparison, the non-probabilistic version of CODEC simply attempts to co-locate embedded sketches and context, which may result in very weak generalization ability. However, we did not anticipate the extent to which this distinction was crucial.

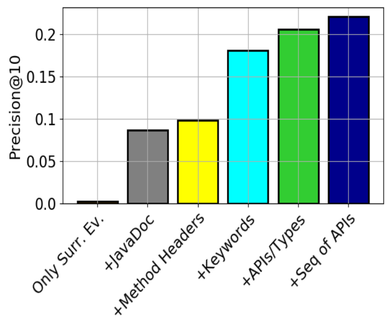
To examine this in a bit more detail, we randomly selected 10,000 methods and encoded the sketch for each of the methods into the latent space using the learned reverse encoder for CODEC, as well as the equivalent encoder for the non-probabilistic version of CODEC. We then clustered those 10,000 embeddings for both CODEC and



(a) API Match



(b) Sketch Match



(c) Exact Match

Figure 4: Contributions of various types of evidence to CODEC's retreival accuracy.

its non-probabilistic equivalent using k-means, with $k = 10$. For each cluster, we measured the average Jaccard similarity of the API calls made by the methods within the cluster to the calls made by the methods within the other clusters.

As shown in Figure 5, what we find is that the methods within the clusters formed by CODEC show much greater similarity compared to the clusters formed by its non-probabilistic version. The self-similarity within each cluster tops out at around 16% for the non-probabilistic version, whereas it tops out at around 24% for CODEC. This provides strong evidence that the embeddings learned by CODEC are high-quality precisely because of CODEC's synthesis-based code search.

10.2 Qualitative User Study

Clearly, CODEC demonstrates utility when a method body is removed, and a search methodology is able to find the method body (or an equivalent method body) in a database of millions of alternatives. However, in reality, a correct method body is not typically available in a database, and questions of how useful a returned code is are likely best answered by human programmers. Thus, we conducted a user study where programmers were asked to grade the utility of the programs returned by the various methods

10.2.1 Experimental Setup

Search problems tested. We constructed a set of 15, carefully created search problems, as shown in Table 2. Each search problem consists of a well-documented, hand-written class with two or more methods where one of the method bodies is missing.

Volunteers and rating instructions. We recruited 13 volunteers to rate search results. Each volunteer was a Rice Computer Science graduate student, and could be described as an expert programmer.

For each search task, and for each of the four competitive methods tested, each volunteer was shown the top three search results. For each search task, users were asked to rate the set of three results as a group on a scale of one-to-five in terms of retrieved code's perceived utility for helping a programmer to fill in the missing method body (five being "perfect match" and one being "poor match"). Beyond that, each volunteer was asked to develop his/her own interpretation of search result quality by examining the incomplete Java class. This amounted to 60 search result rating tasks in all, per volunteer. The complete set of rating tasks was designed to be completed in one hour, but volunteers were not given a time limit.

Statistical analysis. An average search result rating was computed for each of the four methods, across each of the 15 search problems $\times$ 13 volunteer $=$ 195 search results.

We were also interested in the statistical significance of comparisons of the average ratings across search methodologies: If one method has a higher rating on average, is the difference statistically significant? We designed a bootstrap-based pairwise hypothesis test [17] comparing the ratings given by a user for predictions on the same problem from different algorithms. We consider a null hypothesis of the following form. For two search methods A and B, define:

$H_0^{A,B}$ = "The average score for search strategy A is worse than the score for search strategy B."

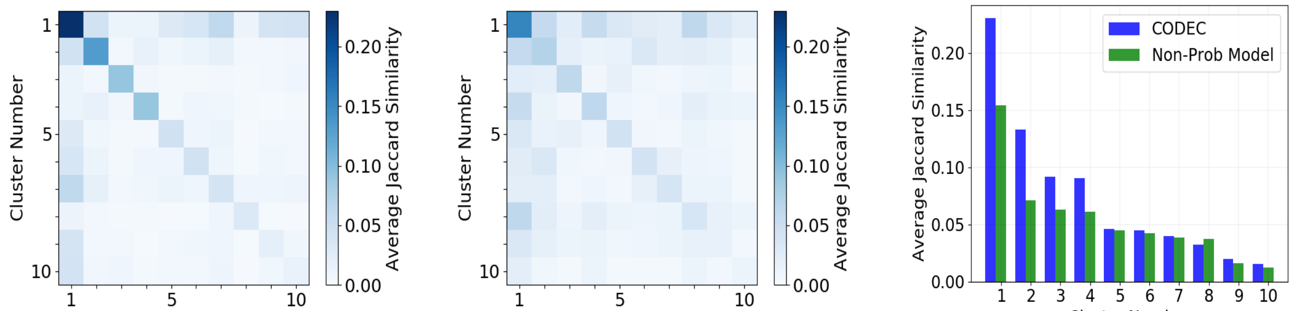
Our goal is to see if we can reject this null hypothesis for various combinations of A and B. Unfortunately, our experimental setup is rather complex, as there are two sources of variability in our experimental setup: (1) the set of participants chosen, as well as (2) the set of search tasks selected. With a different set of participants and a different set of search tasks, we may have obtained different results.

Hence, a classical statistical test such as a t-test is not directly applicable, as it assumes the ratings are sampled (identically and independently distributed) from a single population. In our case, this assumption does not hold as the scores obtained by a single volunteer are conditioned upon the volunteer selected. Thus our use of a bootstrap-based method to attempt to reject the null hypothesis. For a particular pair of search methods, we re-sample with replacement from among the volunteers, and we re-sample with replacement from among the rating tasks, and compute the mean score. If A has a better average than B, then for that re-sampled data set instance, the null hypothesis has been rejected. This process is repeated many times, and the fraction of the time that the null hypothesis is not rejected is the p-value of the test.

10.2.2 Results and Discussion

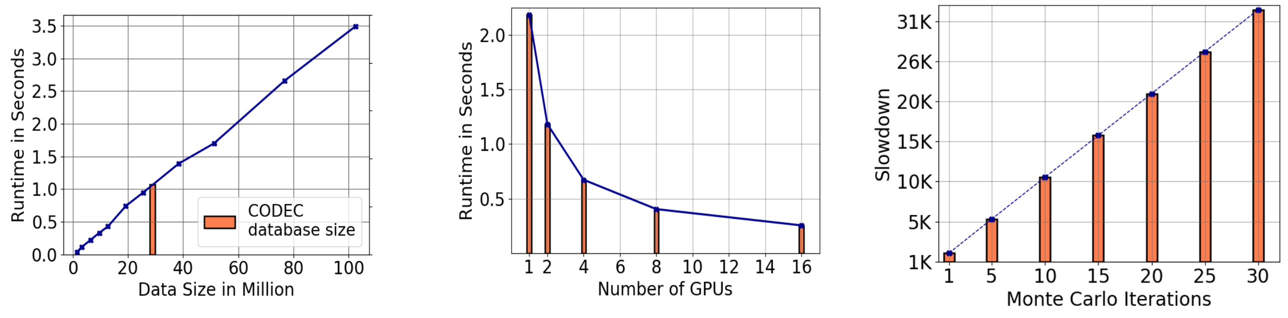
Across all rating tasks and volunteers, we see that CODEC receives an average rating of 3.89, Deep-Code search and CodeHow receive an average rating of 2.81 and 2.53 respectively, while the non-probabilistic version of CODEC receives a rating of 1.71.

Qualitatively, it seems that these results show a very large spread, with CODEC more than a full rating point higher than Deep-Code search and CodeHow, and the non-probabilistic version of CODEC a full rating point behind Deep-Code search. We would venture to say that for a real code-search application, these gaps would translate to significant differences in user satisfaction with the various methods, and perhaps even to gaps in programmer productivity as a user needs to spend extra time and effort with a search where she/he is not happy with the results.



(a) CODEC inter/intra-cluster similarity. (b) Non-prob inter/intra-cluster similarity. (c) Comparison of inter-cluster similarity.

Figure 5: Comparing inter-cluster similarity (measured via API call similarity) for CODEC and its non-probabilistic variant.



(a) Runtime variation with increasing data size for a 16-GPU machine. CODEC uses an indexed database of 27.9M programs collected from Github.

(b) Runtime variation with increasing number of GPUs for a dataset of 6.4M indexed programs.

(c) Slowdown using Monte-Carlo sampling, compared to CODEC's reverse encoder.

Figure 6: Performance characteristics of CODEC.

Statistically, there are significant differences among the methods, as shown in Table 3. CODEC's average score is, statistically speaking, significantly higher than the average score of the other three competitive methods—the only possible exception is CODEC compared to Deep-Code search, where the null hypothesis is rejected at a p-value of 0.04. There is not a statistically-significant difference between Deep-Code search and CodeHow, but all methods are better than the non-probabilistic version of CODEC.

10.3 Runtime Performance

One of the key technical innovations of our approach is the introduction of the reverse encoder $Q(Z|\mathsf{Y},\theta)$ which makes it possible to evaluate $P(\mathsf{Y}_i|\mathsf{X},\theta)$ for a particular sketch Y_i analytically, using a closed-form formula. This is crucial as it allows a large database to be searched quickly. In this subsection, we benchmark our CODEC implementation, running it on an Amazon AWS `p2.16xlarge` machine with 16 NVIDIA K80 GPUs.

CODEC uses this approximation to perform parallel/distributed search using multiple GPUs, possibly spread over multiple machines. The implementation is fairly simple. b_{X} is pre-computed for each database program and stored in GPU RAM. In the case of a multi-dimensional latent space, each b_{X} is a vector, whose dimensionality is equivalent to the dimensionality of the latent space (256 dimensions). Then, in response to a query, the computations of Section 7 are performed on each GPU, resulting in an approximation of $P(\mathsf{Y}_i|\mathsf{X})$ for each Y_i in the database. Assuming the goal is to return the top K programs, the top K $P(\mathsf{Y}_i|\mathsf{X})$ values are sent from each GPU to a central server, where the top K $P(\mathsf{Y}_i|\mathsf{X})$ values overall are computed, and the associated codes are returned.

Runtime with varying database size. We start our analysis with a small database of 1.6M Java methods, where each GPU is assigned a data size of 0.1M methods, and increase the database size to a total of 102.4M synthetic programs, and measure the time to compute the top K programs for $K = 100$. Note that CODEC uses a set of 27.9 million Java methods collected from Github, which takes approximately 1.14 seconds to search. Results are shown in Figure 6(a). Runtime increases linearly with data size, and for a dataset of 102.4M programs, we take around 3.5 seconds per query using 16 GPUs. Considering that an Amazon `p2.16xlarge` costs $14.40 per hour, this equates to a cost of around 1.4 cents per 100M programs searched.

Runtime with varying hardware. Since CODEC search is embarrassingly parallel, it should be possible to push down the runtime by simply increasing the number of GPUs available. For this experiment, we fix the dataset size to be 6.4M programs, small enough so that the associated vectors can all stored on a single GPU. We then increase the number of GPUs by repeatedly doubling from 1 until 16 GPUs, and notice that the runtime decreases approximately by half with each doubling, as shown in Figure 6(b). This implies that it should be possible to push the search time for a large database down nearly arbitrarily, by simply using more GPUs.

Efficiency and accuracy of the reverse encoder. An alternative to using reverse encoder (evaluated in all of the experiments this far) is to use Monte-Carlo (MC) simulation. The obvious, MC method for evaluating $P(\mathsf{Y}_i|\mathsf{X},\theta)$ and estimating this value, is to draw N samples from $P(Z|\mathsf{X},\theta)$, and then use the estimator $P(\mathsf{Y}_i|\mathsf{X},\theta) \approx \frac{1}{N}\sum_{\mathsf{Z}_j\sim P(Z|\mathsf{X},\theta)} P(\mathsf{Y}_i|\mathsf{Z}_j,\theta)$. Note that each sample requires a probability calculation, $P(\mathsf{Y}_i|\mathsf{Z}_j, \theta)$, which will be quite expensive (even on a GPU) as this probability calculation makes use of a tree-based, recurrent neural network.

We compare CODEC's reverse encoder-based implementation with this MC alternative, running on a single GPU, using 0.1M programs on a `p2.xlarge` Amazon machine.

Results are plotted in Figure 6(c), where the ratio of the MC time to CODEC's reverse encoder-based time is shown as a function of the number of MC iterations performed. We find that after around 30 MC iterations, the Jaccard similarity between the top 100 programs returned by both computations for an arbitrary query converges to around 0.91, and that further MC iterations do not increase this similarity further (at this stage, the mean co-efficient of variation for the MC estimate is around 1%). This indicates that running 30 MC iterations is a good rule-of-thumb for the MC approach, at least for our database. At this point, the MC method is around $\sim 31,000$ times slower than CODEC's reverse encoder.

11 Conclusions

We have proposed the problem of contextualized code search, where a database of code fragments is searched for a match to a query composed of various evidences extracted from the surrounding program. The benefit of contextualized code search compared to other code search methods is that search happens "for free" using the surrounding context; the user need not specify the parameters for search. We have proposed a general, probabilistic framework that allows the inclusion of various types of evidence (sets of types that appear in the surrounding code, lists of formal parameters, English comments, etc.). Virtually any evidence can be used, as long as a suitable encoder for the evidence can be developed. A key technical innovation is the learning of a "reverse encoder" that allows for fast search, by allowing the framework to compute a simple, closed-form version of the posterior probability of generating a code from the evidence at query time. We have shown that the resulting search engine gives high-quality results.

We end the paper by asking, could CODEC be extended past Java? In terms of engineering effort, adding an additional language would require (a) designing a new intermediate language (similar to SKETCH) for the target language, (b) re-implementing the context extractor and decompiler, and (c) updating the learner and search engine to incorporate any changes in evidence types and in the intermediate language. For most modern imperative languages (Python and C++ come to mind), the engineering effort would be minimal, as the evidences would stay the same, and SKETCH could be used with small changes. However, even a functional language such as Scala should require relatively little effort.

Perhaps a more interesting question is: would CODEC give good results with other languages? We anticipate it would, with one caveat: our CODEC prototype relies heavily on the fact that Java has a widely-used set of standard types and methods. These are important evidence types for CODEC. In a language such as C for which there is arguably less uniformity in terms of the types and libraries used, the CODEC approach might be more successful for searching a more limited code base (say, the code produced by a corporation or open-source project), as opposed to searching a more general database such as GitHub.

12 Acknowledgements

We thank the anonymous reviewers for their constructive suggestions. We thank the volunteers for their time in helping us with the qualitative user study experiment. Work presented in this paper has been supported by NSF under grant numbers 1910803 and 1918651.

13 Appendix: Reverse Encoder in Multi Dims

The goal is to be able to compute $P(\mathsf{Y}|\mathsf{X})$ in the case that Z, the embedding, is multi-dimensional. We begin with:

$$\begin{aligned}
P(\mathsf{Y}|\mathsf{X}) &= \int_{\mathsf{Z}} P(\mathsf{Z}|\mathsf{X})P(\mathsf{Y}|\mathsf{Z})d\mathsf{Z} \\
&= \int_{\mathsf{Z}} \frac{P(\mathsf{Z}|\mathsf{X})P(\mathsf{Z}|\mathsf{Y})P(Y)}{P(\mathsf{Z})} d\mathsf{Z} \\
&= \int_{\mathsf{Z}} \exp\big(\mathbf{a}_\mathsf{X}\cdot\mathsf{Z}^2 + \mathbf{b}_\mathsf{X}\cdot\mathsf{Z} + \mathbf{c}_\mathsf{X}\big) \\
&\quad \times \exp\big(\mathbf{a}_\mathsf{Y}\cdot\mathsf{Z}^2 + \mathbf{b}_\mathsf{Y}\cdot\mathsf{Z} + \mathbf{c}_\mathsf{Y}\big) \\
&\quad \times \exp\big(-\mathbf{a}_\mathsf{I}\cdot\mathsf{Z}^2 - \mathbf{b}_\mathsf{I}\cdot\mathsf{Z} - \mathbf{c}_\mathsf{I}\big) \times P(Y)d\mathsf{Z}
\end{aligned}$$

where $(\mathbf{a}_\mathsf{X}, \mathbf{b}_\mathsf{X}, \mathbf{c}_\mathsf{X})$, $(\mathbf{a}_\mathsf{Y}, \mathbf{b}_\mathsf{Y}, \mathbf{c}_\mathsf{Y})$, $(\mathbf{a}_\mathsf{I}, \mathbf{b}_\mathsf{I}, \mathbf{c}_\mathsf{I})$ are the parametarizations of multidimensional normal distributions $P(\mathsf{Z}|\mathsf{X})$, $P(\mathsf{Z}|\mathsf{Y})$ and $P(\mathsf{Z})$, respectively.

Note that each of these can be represented as follows, for each dimension i:

$$\begin{aligned}
&\mathbf{a}^i_\mathsf{X} = -\frac{1}{2\sigma^2_\mathsf{X}}, \mathbf{b}^i_\mathsf{X} = \frac{\mu^i_\mathsf{X}}{\sigma^2_\mathsf{X}}, \mathbf{c}^i_\mathsf{X} = -\frac{(\mu^i_\mathsf{X})^2}{2\sigma^2_\mathsf{X}} - \frac{1}{2}\ln(\sigma^2_\mathsf{X}) - \frac{1}{2}\ln 2\pi \\
&\mathbf{a}^i_\mathsf{Y} = -\frac{1}{2\sigma^2_\mathsf{Y}}, \mathbf{b}^i_\mathsf{Y} = \frac{\mu^i_\mathsf{Y}}{\sigma^2_\mathsf{Y}}, \mathbf{c}^i_\mathsf{Y} = -\frac{(\mu^i_\mathsf{Y})^2}{2\sigma^2_\mathsf{Y}} - \frac{1}{2}\ln(\sigma^2_\mathsf{Y}) - \frac{1}{2}\ln 2\pi \\
&\mathbf{a}^i_\mathsf{I} = -\frac{1}{2}, \quad \mathbf{b}^i_\mathsf{I} = 0, \quad \mathbf{c}^i_\mathsf{I} = -\frac{1}{2}\ln 2\pi
\end{aligned}$$

Continuing, we have:

$$\begin{aligned}
P(\mathsf{Y}|\mathsf{X}) &= P(\mathsf{Y}) \times \int_{\mathsf{Z}} \exp\big[(\mathbf{a}_\mathsf{X} + \mathbf{a}_\mathsf{Y} - \mathbf{a}_\mathsf{I})\cdot\mathsf{Z}^2 + \\
&\quad + (\mathbf{b}_\mathsf{X} + \mathbf{b}_\mathsf{Y})\cdot\mathsf{Z} + (\mathbf{c}_\mathsf{X} + \mathbf{c}_\mathsf{Y} - \mathbf{c}_\mathsf{I})\big]d\mathsf{Z} \\
&= P(\mathsf{Y}) \times \int_{\mathsf{Z}} \exp\big(\mathbf{a}^*\cdot\mathsf{Z}^2 + \mathbf{b}^*\cdot\mathsf{Z} + \mathbf{c}'\big)d\mathsf{Z}
\end{aligned}$$

Where: $\mathbf{a}^* = \mathbf{a}_\mathsf{X} + \mathbf{a}_\mathsf{Y} - \mathbf{a}_\mathsf{I}$, $\mathbf{b}^* = \mathbf{b}_\mathsf{X} + \mathbf{b}_\mathsf{Y}$ and $\mathbf{c}' = \mathbf{c}_\mathsf{X} + \mathbf{c}_\mathsf{Y} - \mathbf{c}_\mathsf{I}$. Simplifying, we have:

$$\begin{aligned}
P(\mathsf{Y}|\mathsf{X}) &= P(\mathsf{Y}) \times \exp(\mathbf{c}' - \mathbf{c}^*) \\
&\quad \times \int_{\mathsf{Z}} \exp(\mathbf{a}^*\cdot\mathsf{Z}^2 + \mathbf{b}^*\cdot\mathsf{Z} + \mathbf{c}^*)d\mathsf{Z} \\
&= P(\mathsf{Y}) \times \exp(\mathbf{c}' - \mathbf{c}^*)
\end{aligned}$$

where,

$$\begin{aligned}
\mathbf{c}^* &= \frac{\mathbf{b}^{*2}}{4\mathbf{a}^*} + \frac{1}{2}\ln(-\frac{\mathbf{a}^*}{\pi}) \\
\mathbf{c}' &= \mathbf{c}_\mathsf{X} + \mathbf{c}_\mathsf{Y} - \mathbf{c}_\mathsf{I} \\
&= \frac{\mathbf{b}^2_\mathsf{X}}{4\mathbf{a}_\mathsf{X}} + \frac{1}{2}\ln(-\frac{\mathbf{a}_\mathsf{X}}{\pi}) + \frac{\mathbf{b}^2_\mathsf{Y}}{4\mathbf{a}_\mathsf{Y}} + \frac{1}{2}\ln(-\frac{\mathbf{a}_\mathsf{Y}}{\pi}) - \frac{1}{2}\ln 2\pi \\
&= \frac{\mathbf{b}^2_\mathsf{X}}{4\mathbf{a}_\mathsf{X}} + \frac{\mathbf{b}^2_\mathsf{Y}}{4\mathbf{a}_\mathsf{Y}} + \frac{1}{2}\ln(-\frac{\mathbf{a}_\mathsf{X}}{\pi}) + \frac{1}{2}\ln(-\frac{\mathbf{a}_\mathsf{Y}}{\pi}) - \frac{1}{2}\ln 2\pi
\end{aligned}$$

Finally this reduces our final computation to,

$$\log P(\mathsf{Y}|\mathsf{X}) = \log P(\mathsf{Y}) + \sum_i (\mathbf{c}'_i - \mathbf{c}^*_i)$$

where,

$$\begin{aligned}
\mathbf{c}' - \mathbf{c}^* = &\frac{\mathbf{b}^2_\mathsf{X}}{4\mathbf{a}_\mathsf{X}} + \frac{\mathbf{b}^2_\mathsf{Y}}{4\mathbf{a}_\mathsf{Y}} - \frac{\mathbf{b}^{*^2}}{4\mathbf{a}^*} + \frac{1}{2}\ln(-\frac{\mathbf{a}_\mathsf{X}}{\pi}) + \\
&\frac{1}{2}\ln(-\frac{\mathbf{a}_\mathsf{Y}}{\pi}) + -\frac{1}{2}\ln(-\frac{\mathbf{a}^*}{\pi}) - \frac{1}{2}\ln 2\pi
\end{aligned}$$

Note that this computation is easily implemented to a matrix-vector style operation that can be parallelized to run on a GPU.

14 References

[1] Codase web site: http://www.codase.com.

[2] Eclipse jdt web site: https://help.eclipse.org/luna/.

[3] Github web site: http://www.github.com.

[4] Koders web site: http://www.koders.com.

[5] Krugle web site: http://www.krugle.com.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50(1):5–43, Jan 2003.

[8] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[9] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166, New York, NY, USA, 2010. ACM.

[10] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[11] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra. When deep learning met code search. *CoRR*, abs/1905.03813, 2019.

[12] S. Chakraborty, M. Allamanis, and B. Ray. Tree2tree neural translation model for learning source code changes. *CoRR*, abs/1810.00314, 2018.

[13] Q. Chen and M. Zhou. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 826–831, New York, NY, USA, 2018. ACM.

[14] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[15] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017.

[16] C. Doersch. Tutorial on variational autoencoders, 2016.

[17] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap*. Springer, 1993.

[18] J. Gu, H. Shavarani, and A. Sarkar. Top-down tree structured decoding with syntactic connections for neural machine translation and parsing. *CoRR*, abs/1809.01854, 2018.

[19] X. Gu, H. Zhang, and S. Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 933–944, New York, NY, USA, 2018. ACM.

[20] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[21] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 237–240, New York, NY, USA, 2005. ACM.

[22] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Mapping language to code in programmatic context. *CoRR*, abs/1808.09588, 2018.

[23] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon. F a c o y: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pages 946–957. ACM, 2018.

[24] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[25] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.*, 53(4):294–306, Apr. 2011.

[26] G. Little and R. C. Miller. Keyword programming in java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 84–93, New York, NY, USA, 2007. ACM.

[27] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270, Nov 2015.

[28] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[29] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, Mar. 1971.

[30] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[31] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293. AAAI Press, 2016.

[32] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017.

[33] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

[34] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[35] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra. Retrieval on source code: A neural code search. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*,

MAPL 2018, pages 31–41, New York, NY, USA, 2018. ACM.

[36] N. Sahavechaphan and K. T. Claypool. Xsnippet: Mining for sample code. volume 41, pages 413–430, 10 2006.

[37] R. Salakhutdinov and G. Hinton. Semantic hashing. *RBM*, 500(3):500, 2007.

[38] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45:2673 – 2681, 12 1997.

[39] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 905–908, New York, NY, USA, 2006. ACM.

[40] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 129–136, USA, 2011. Omnipress.

[41] K. Sohn, H. Lee, and X. Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3483–3491. Curran Associates, Inc., 2015.

[42] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2727–2735, New York, NY, USA, 2019. Association for Computing Machinery.

[43] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, Beijing, China, July 2015. Association for Computational Linguistics.

[44] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[45] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 25–36, Piscataway, NJ, USA, 2019. IEEE Press.

[46] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu. Multi-modal attention network learning for semantic source code retrieval, 2019.

[47] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu. Multi-modal attention network learning for semantic source code retrieval. *arXiv preprint arXiv:1909.13516*, 2019.

[48] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 404–415, May 2016.

[49] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 513–523, New York, NY, USA, 2002. ACM.

[50] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 70–80, Piscataway, NJ, USA, 2019. IEEE Press.

[51] X. Zhang, L. Lu, and M. Lapata. Tree recurrent neural networks with application to language modeling. *CoRR*, abs/1511.00060, 2015.