Code Embedding: Representation of Code Similarity in the Vector Space

Li Dinghong

May 22, 2019

Code Embedding: continuous vectors for representing snippets of code
Motivating applications:

- Semantic labeling
- Recommendation for search engine
- Vulnerability detection and plagiarism detection

Outline

- *Reverse Compilation Techniques (1995)*: raise concepts and general pipelines of decompilers, and several important intermediate representations
- *code2vec: Learning Distributed Representations of Code (2018)*: Learn semantic imformation using path-attention models on AST
- *Neural Code Comprehension: A Learnable Representation of Code Semantics (2018)*: Learn statement embeddings on XFG, which combines data- and control-flow graphs

Decompiler

- **Compiler:** a program that translates source code from a high-level programming language to a lower level language
- **Decompiler:** a program that translates source code from a low-level language into a high-level language

Phases of Decompilers

- **Frontend:**
  Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Control Flow Graph Generation
- **UDM (Universal Decompiling Machine):**
  Data Flow Analysis, Control Flow Analysis
- **Backend:**
  Code Generation

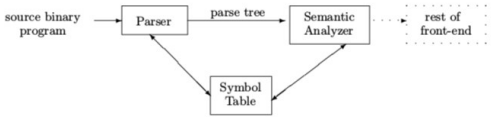
Phases of Decompilers

Compare: Phases of Compilers

- **Frontend:**
  Lexical Analysis, Syntax Analysis, Semantic Analysis
- **Intermediate Representation (ex: llvm)**
- **Backend:**
  Code Generation

Decompilers do not contain lexical anlyzers. Why?

Syntax Analysis



Interaction between the Parser and Semantic Analyzer

- Group a sequence of bytes into a phrase or sentence of the language; Output the parsing tree
- Grammar: Regular Expression (Context-free grammar is not necessary)
- Main Difficulty: To separate code from data. That is, to determine which bytes in memory represent code and which ones represent data

Semantic Analysis

- Determine the meaning of a group of machine instructions, collect information on the individual instructions of a subroutine, and propagate this information across the instructions of the subroutine
- base data types such as integers and long integers are propagated across the subroutine in this phase

Semantic Analysis

- **Definition:** An identifier ($< ident >$) is either a register, local variable (negative offset from the stack), parameter (positive offset from the stack), or a global variable (location in memory)
- **Definition:** An idiom is a sequence of instructions that has a logical meaning which cannot be derived from the individual instructions
- **Examples of idiom:** Subroutine Idioms, Calling Conventions, Long Variable Operations
- **Type Propagation**

Intermediate Code Generation

- **Low-level Intermediate Code**
  ex: opcode dest src1 src2 $\rightarrow$ add bx bx 3, push sp cx
- **High-level Intermediate Code**
  Three-address code: a linearized representation of an abstract syntax tree (AST) of the program

Control Flow Graph Generation

- **Assumption:** No self-modifying code, and make no use of data as instructions or vice versa
- **Definition:** Let
  - P be a program
  - $I = \{i_1, \cdots, i_n\}$ be the instructions of P
  - $D = \{d_1, \cdots, d_m\}$ be the data of P

  Then, $P = I \cup D$ and $I(P) \cap D(P) = \emptyset$

Control Flow Graph Generation

Intermediate instructions are classified in two sets for the purposes of control flow graph generation

- **Transfer Instructions (TI):** the set of instructions that transfer flow of control to an address in memory different from the address of the next instruction
  - Unconditional jumps
  - Conditional jumps
  - Indexed jumps
  - Subroutine call
  - Subroutine return
  - End of program
- **Non transfer instructions (NTI):** Instructions that do not belong to the TI set

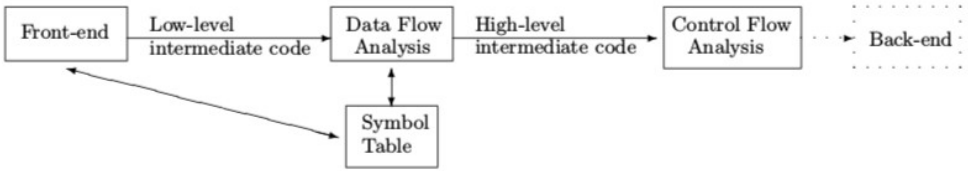
Control Flow Graph Generation

- **Definition:** A basic block $b = [i_1, \cdots, i_{n-1}, i_n], n \geq 1$ is an instruction sequence that satisfies the following conditions:
  - $[i_1, \cdots, i_{n-1}] \in NTI$ and $i_n \in TI$
    or
  - $[i_1, \cdots, i_n] \in NTI$ and i_{n+1} is the first instruction of another basic block

Control Flow Graph Generation

- **Definition:** A control flow graph $G = (N, E, h)$ for a program P is a connected, directed graph, that satisfies the following conditions:
  - $\forall n \in N$, n represents a basic blocks of P
  - $\forall e = (n_i, n_j) \in E$, e represents flow of control from one basic block to another and $n_i, n_j \in N$
  - $\exists f : B \rightarrow N \bullet \forall b_i \in B, f(b_i) = n_k$ for some $n_k \in N \wedge \exists b_j \in B \bullet f(b_j) = n_k$ *(Question: What does $\bullet$ here mean?)*
- **Types of basic blocks:**
  1-way, 2-way, n-way, call basic, return basic, fall basic

Data Flow Analysis



Context of the Data Flow Analysis Phase

- **Goal:** To improve the quality of the low-level intermediate code, and to reconstruct some of the information lost during the compilation process

Data Flow Analysis

- **Definition:** A register is **defined** if the content of the register is modified (i.e. it is assigned a new value). In a similar way, a flag is defined if it is modified by an instruction.
- **Definition:** A register is **used** if the register is referenced (i.e. the value of the register is used). In a similar way, a flag is used if it is referenced by an instruction.
- **Definition:** A **locally available definition** d in a basic block B_i is the last definition of d in B_i.

Taxonomy of Data Flow Problems

- A typical data flow equation for basic block B_i

$$out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i))$$

(the information at the end of basic block B_i is either the information generated on B_i, or the information that entered the basic block and was not killed within the basic block)

$$in(B_i) = \cup_{p \in Pred(B_i)} out(p)$$

Comparsion: Different Intermediate Representations

- AST: Intuitive understanding of codes / Miss important dependency information
- Data Flow Graph: Preserve Data dependency information / Unnecessarily redundant
- Control Flow Graph: Tackle important information / Far from human understanding

Challenges for AST: Representation

- NLP methods usually treat text as a linear sequence of tokens / programming languages can greatly benefit from representations that leverage the structured nature of their syntax
- *Representation*: we use paths in the program's abstract syntax tree (AST) as our representation
- *Advantages*: significantly lowers the learning effort (compared to learning over program text) / scalable and general
- *Challenges*: **how to aggregate a bag of contexts, and which paths to focus on for making a prediction**

Challenges for AST: Attention

- *Problems*: Representing each bag of path-contexts monolithically is going to suffer from sparsity – even similar methods will not have the exact same bag of path-contexts
- The attention mechanism is learned simultaneously with the embeddings. This optimizes both the atomic representations of paths and the ability to compose multiple contexts into a single code vector
- *Challenges*: **how to aggregate a bag of contexts, and which paths to focus on for making a prediction**

Challenges for AST: Soft and Hard Attention

- *Soft-attention*: weights are distributed "softly" over all path-contexts in a code snippet
- *Hard-attention*: selection of a single path-context to focus on at a time
- Performance of the model is greatly improved thanks to soft attention!

Representing Code Using AST-Paths

Definition: Abstract Syntax Tree

An Abstract Syntax Tree (AST) for a code snippet C is a tuple $\langle N, T, X, s, \sigma, \phi \rangle$

- N: a set of nonterminal nodes
- T: a set of terminal nodes
- X: a set of values
- $s \in N$: the root node
- $\sigma : N \to (N \cup T)^*$: a function that maps a nonterminal node to a list of its children
- $\phi : T \to X$: a function that maps a terminal node to an associated value
- Every node except the root appears exactly once in all the lists of children

Representing Code Using AST-Paths

Definition: AST path

An AST-path of length k is a sequence of the form:

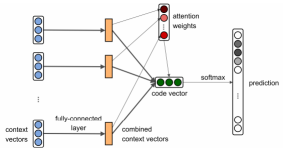
- $n_1 d_1 \cdots n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminals,
- for $i \in [2, k]$: $n_i \in N$ are nonterminals
- for $i \in [1, k]$: $d_i \in \{\uparrow, \downarrow\}$ are movement directions (either up or down in the tree) (If $d_i = \uparrow$, then: $n_i \in \sigma(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \sigma(n_i)$)
- For an AST-path p, we use $start(p)$ to denote n_1 - the starting terminal of p; and $end(p)$ to denote n_{k+1} - its final terminal

Representing Code Using AST-Paths

Definition: Path-context

- Given an AST Path p, its path-context is a triplet $\langle xs, p, xt \rangle$ where $xs = \phi(start(p))$ and $xt = \phi(end(p))$ are the values associated with the start and end terminals of p.
- That is, a path-context describes two actual tokens with the syntactic path between them
- A possible path-context that represents the statement: "x = 7;" $\langle x, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), 7 \rangle$
- We limit the paths by maximum length and maximum width as hyperparameters of our model

Model



The architecture of our path-attention network

- A fully-connected layer learns to combine embeddings of each path-contexts with itself
- Attention weights are learned using the combined context vectors, and used to compute a code vector
- The code vector is used to predict the label

Model

- *Fully connected layer*: Since every context vector c_i is formed by a concatenation of three independent vectors, a fully connected layer learns to combine its component

$$\tilde{c_i} = tanh(W \cdot c_i)$$

- *Aggregating multiple contexts into a single vector representation with attention*: Given the combined context vectors: $\{\tilde{c_1}, \cdots, \tilde{c_n}\}$, the attention weight α_i of each $\tilde{c_i}$ is computed as the normalized inner product between the combined context vector and the global attention vector a

$$\text{attention weight } \alpha_i = \frac{exp(\tilde{c_i}^T \cdot a)}{\sum_{j=1}^{n} exp(\tilde{c_j}^T \cdot a)}$$

$$\text{code vector } v = \sum^{n} \alpha_i \cdot \tilde{c_i}$$

Model

Prediction:

- Prediction of the tag is performed using the code vector. We define a tags vocabulary which is learned as part of training:

$$tags\_vocab \in \mathbb{R}^{|Y| \times d}$$

- The predicted distribution of the model $q(y)$ is computed as the (softmax-normalized) dot product between the code vector v and each of the tag embeddings:

$$f \text{ or } y_i \in Y : q(y_i) = \frac{exp(v^T \cdot tags\_vocab_i)}{\sum_{j=1}^{n} exp(v^T \cdot tags\_vocab_j)}$$

Model

- *Training:* p is a distribution that assigns a value of 1 to the actual tag in the training example and 0 otherwise
- Loss function: cross entropy

$$H(p||q) = \sum_{y \in Y} p(y) \log q(y) = \log q(y_{true})$$

- *Predicting tags and names:*

$$prediction(C) = \arg\max_{\mathcal{L}} P(\mathcal{L}|C) = \arg\max_{\mathcal{L}} q_{v_C}(y_{\mathcal{L}})$$

where q_{v_C} is the predicted distribution of the model, given the code vector v_C

The Naturalness Hypothesis

- Software is a form of human communication
- Software corpora have similar statistical properties to natural language corpora
- These properties can be exploited to build better software engineering tools

Contributions

- Define an embedding space, inst2vec, based on an Intermediate Representation (IR) of the code that is independent of the source programming language
- Provide a novel definition of contextual flow for this IR, leveraging both the underlying data- and control-flow of the program

Contributions

- The LLVM IR is processed to a robust representation that we call conteXtual Flow Graphs (XFGs). XFGs are constructed from both the data- and control-flow of the code, thus inherently supporting loops and function calls
- We evaluate the representation using clustering, analogies, semantic tests, and three fundamentally different high-level code learning tasks

Related Work

- Distributed representations of code were first suggested by Allamanis et al.
- **Code Representation:** Previous research focuses on embedding high-level programming languages in the form of tokens or statements, as well as lower level representations such as object code
- **Automated Tasks on Code:** Restricted to the OpenCL language. Furthermore, the state-of-the-art in automatic tuning for program optimization uses surrogate performance models and active learning, and does not take code semantics into account
- **Embedding Evaluation:** We are the first to quantify the quality of a code embedding space itself in the form of clustering, syntactic analogies, semantic analogies, and categorical distance tests

A Robust Distributional Hypothesis of Code

- **Statements** that occur in the same **contexts** tend to have similar **semantics**
- **Statements:** Universality and uniformity
- **Contexts:** Statements whose execution directly depends on each other (In our representation, context is the union of data dependence and execution dependence, thereby capturing both relations)
- **Semantics:** We draw the definition of semantics from Operational Semantics in programming language theory, which refers to the effects (e.g., preconditions, postconditions) of each computational step in a given program. (e.g., two versions of the same algorithm with different variable types would be synonymous)

Contextual Flow Graphs (XFG)

- **XFG:** Directed multigraphs, where two nodes can be connected by more than one edge
- **Node:** Variables or label identifiers (e.g., basic block, function name)
- **Edge:** Either represents data-dependence (in black), carrying an LLVM IR statement; or execution dependence

XFG Construction

Generate XFGs incrementally from LLVM IR:

- Read LLVM IR statements once, storing function names and return statements
- Second pass over the statements, adding nodes and edges according to the following rule-set:
  - Data dependencies within a basic block are connected
  - Inter-block dependencies (e.g., $\emptyset$-expressions) are both connected directly and through the label identifier (statement-less edges)
  - Identifiers without a dataflow parent are connected to their root (label or program root)

inst2vec: Embedding Statements in Continuous Space

- **Preprocessing**
- **Dataset**
- **Setup and Training**