#ml-papers 2019

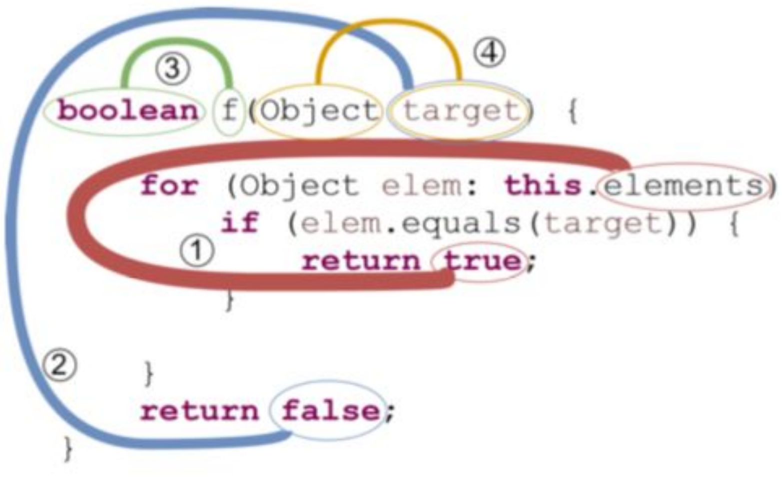
[code2vec: Learning Distributed Representations of Code]

Overview

- The **Code2Vec** model maps **code methods** to **vectors**, in a way that captures subtle differences.
- The algorithm works as follows:
  a. Convert code to an **abstract syntax tree (AST)**, generated by the compiler, which reflects the codes syntax
  b. From the AST, generate a **bag of contexts**. Each context corresponds to a path between leaf nodes.
  c. Map each **context -> vector**
  d. Combine all the context vectors into a single **code vector,** using a weighted sum. The weights are determined by **attention,** which is learned by the neural network.

Section 2: Motivating Example

Code Method

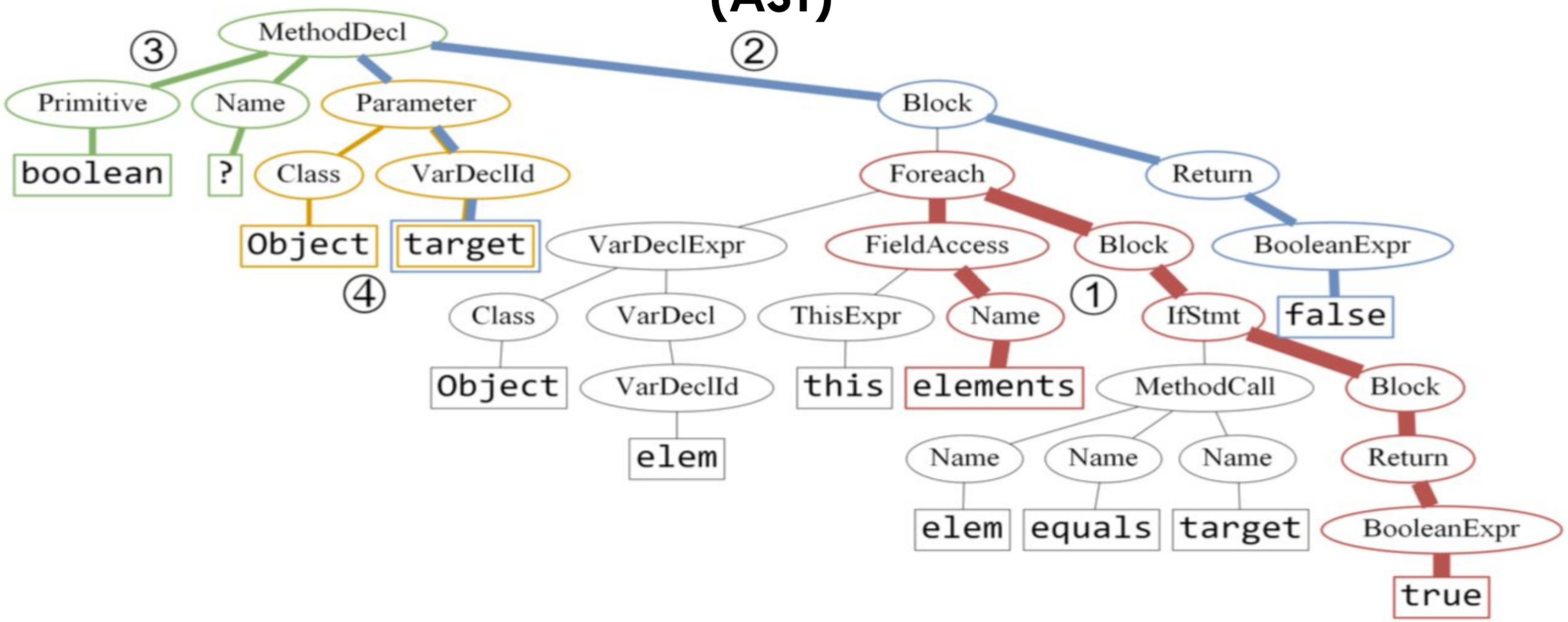


(a)

Predictions:	
contains	90.93%
matches	3.54%
canHandle	1.15%
equals	0.87%
containsExact	0.77%

Top predictions

Abstract Syntax Tree (AST)



- **Attention** is illustrated by the line thickness for the top contexts (1), (2), (3), (4)

Section 2: Key Ideas

- Code can be represented as a **bag of path-contexts**
- Neural network based **attention** is needed to identify the important contexts.
- The embedding vectors can be used for a lot more than predicting method names
- We can interpret the model by **visualizing the attention** of each context

Section 3: Representing Code through AST Paths

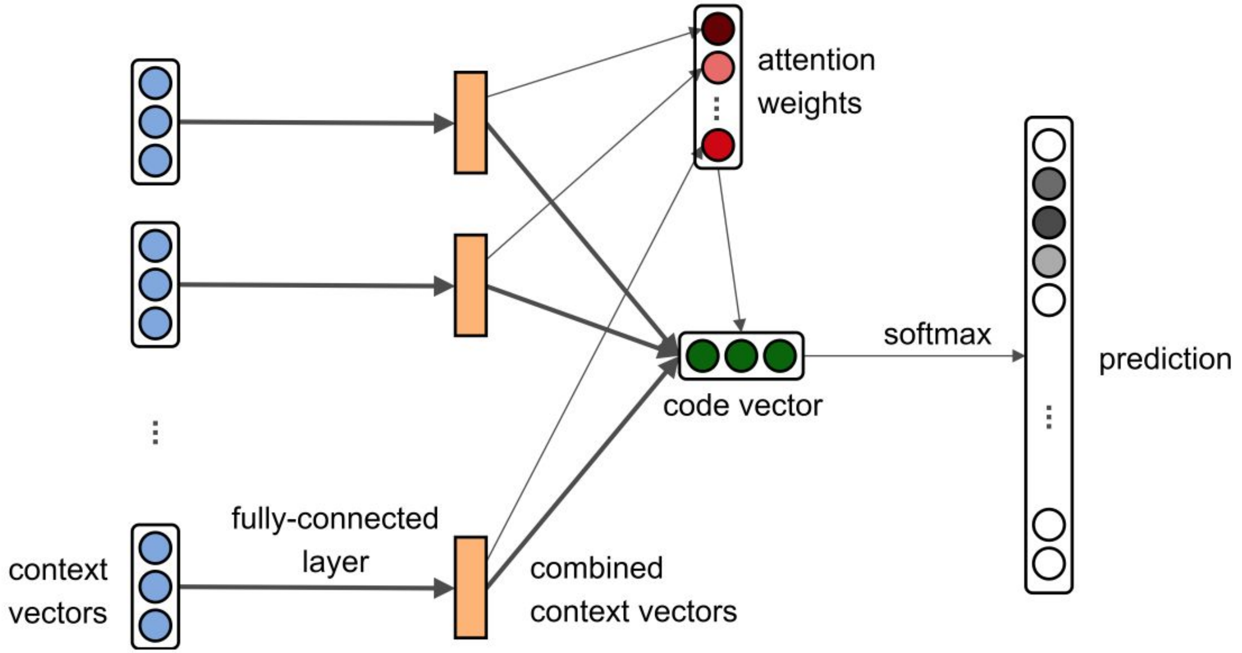
- An **abstract syntax tree (AST)** is a **compiler-generated tree** representing aspects of a program's syntax. The AST, in addition, **assigns a value to every terminal node.**
- An **AST path** of length k is a path consisting of k+1 nodes **n_1, ..., n_k+1** in the graph, each connected by and edge, where the first and last nodes are **leaf** nodes, and the others are non-leaves. In addition, we include information about the direction (up, down) of each edge.
- A **path context** is defined to be a **triple**:
  - **start node value**
  - **p** (the path context)
  - **end node value**

Example 3.1. A possible path-context that represents the statement: "`x = 7;`" would be:

$$\langle \mathrm{x}, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), 7 \rangle$$

Section 4.0: Model Introduction

- **Problem**: We need to represent a code snippet (conceptually a bag of contexts) as a single vector
    - *Insight*: The should use *all* the context vectors but be allowed to learn how much focus to apply to each (attention)
- **Solution**: The models performs a weighted average, i.e. performing a dot product with some global attention vector



Section 4.1-4.2: Path Contexts & Attention

- **First**, we have to turn our text code-snippets into a mathematical object
- Then, we represent C as the set of path-contexts that can be derived from it
  - Where x_s and x_t are values of AST leaves and *p* is the connecting path

$$TPairs\,(C) = \{(term_i, term_j)\,|\,term_i, term_j \in termNodes\,(C) \land i \neq j\}$$

$$Rep\,(C) = \left\{ (x_s, p, x_t) \;\middle|\; \begin{array}{l} \exists(term_s, term_t) \in TPairs\,(C) : \\ x_s = \phi\,(term_s) \land x_t = \phi\,(term_t) \\ \land\, start(p) = term_s \land end(p) = term_t \end{array} \right\}$$

- **Then,** we get our embedding c & on this we apply attention (c~ is post tanh activation function)

$$c_i = embedding\,(\langle x_s, p_j, x_t \rangle) = [\ value\_vocab_s \,;\, path\_vocab_j \,;\, value\_vocab_t\ ] \in \mathbb{R}^{3d}$$

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

- **Finally,** we get our code vector, *v* as the aforementioned dot product

$$\text{code vector } v = \sum_{i=1}^{n} \alpha_i \cdot \tilde{c}_i$$

Section 4.3-4.5: Training

- Use cross-entropy loss & gradient descent (innovative!)
- What to do with the network:
  - Create an embedding for an unseen piece of code
  - Map code into continuous space!!
- Predicting tags and names
  - way more boring
- Their **claims of contribution**:
  - Map multiple contexts into a fixed-length vector using an attention-based weighting
  - Lots of criticisms for the subtoken people (computationally inefficient, not generalizable)

Section 5: Distributed vs. Symbolic Representations

- We want to represent a qualitative concept (a word, a piece of code) quantitatively
- One approach would be to use symbolic (also referred to as localized) representations
    - each unit is uniquely represented with a single component
    - e.g one-hot-encoding
- One major downside to localized representations is high-dimensionality. For example, to represent 10-word sequences from a 100K word dictionary you need 1,000,000 dimensions
- In contrast *distributed* representations distribute the representation of an element (e.g. a word, a snippet of code) over multiple components.
    - each unit is represented by a vector describing various aspects of element
    - e.g. word embeddings
- In this model, the difference between symbolic and distributed representations of code is the difference between polynomial and linear time
    - symbolic representation => $O(|X|^2 * |P| * |Y|)$
    - distributed representation => $O(d * (|X| * |P| * |Y|)$
- Distributed representation makes training feasible!

STASH

Section 7: Model Limitations

- Closed Labels Vocabulary
  - Model is able to predict only labels that were observed as-is at training time
  - Works for the majority of targets, specifically ones which repeat across multiple programs
  - Model with struggle with more specific and diverse targets (e.g., **`findUserInfoByUserIdAndKey`**) and could only catch the main idea (for example: **`findUserInfo`**)
- Sparsity
  - Model consumes lots of trained parameters
    - large GPU memory consumption at training time
    - increases the size of the stored model
    - requires a lot of training data
  - Too much Data
    - Anything not observed in the data at training will not be represented in the model. To address this, a huge data set was used. Granularity boosts signal of less relevant data. Causes model to not perform as well as smaller datasets
- Dependency on variable names
  - When given uninformative, obfuscated or adversarial variable names, the prediction of the label is usually less accurate
  - Train the model on a mixed dataset of good and hidden variable names, hopefully reducing model dependency on variable names

Section 8: Related Work

- Bimodal modelling and natural language
  - *Bimodal*: Machines use it, humans can read it
  - They claim their way of representing the structure of code vs. other implementations treating code as an NLP problem is better
- Representation of code in ML models
  - Use syntactic relations to represent code
- Attention
  - Wide use in NLP, Speech Recognition, and image captioning
  - Attempt to make the model care about only the signal
    - E.g. white noise v. a person actually speaking
- Distributed Representations
  - Words in similar contexts have similar meanings
  - Motivated by word2vec