A Survey of Representational Models of Source Code

Supervisor: Bryksin Timofey
Student: Murycheva Natalia

31/10/2018

Introduction

- with the rise of machine learning, there is a great deal of interest in treating **programs** as data **to be fed to learning algorithms**

- thus, the goal is to built models to learn intermediate, not necessarily human-interpretable, encodings of code

- they are called **representational code models**

Outline

- Types of representation code models
  - distributed representation
    - Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces
  - structured prediction
    - Predicting program properties from “big code”
  - hybrid
    - Deep Learning Similarities from Different Representations of Source Code

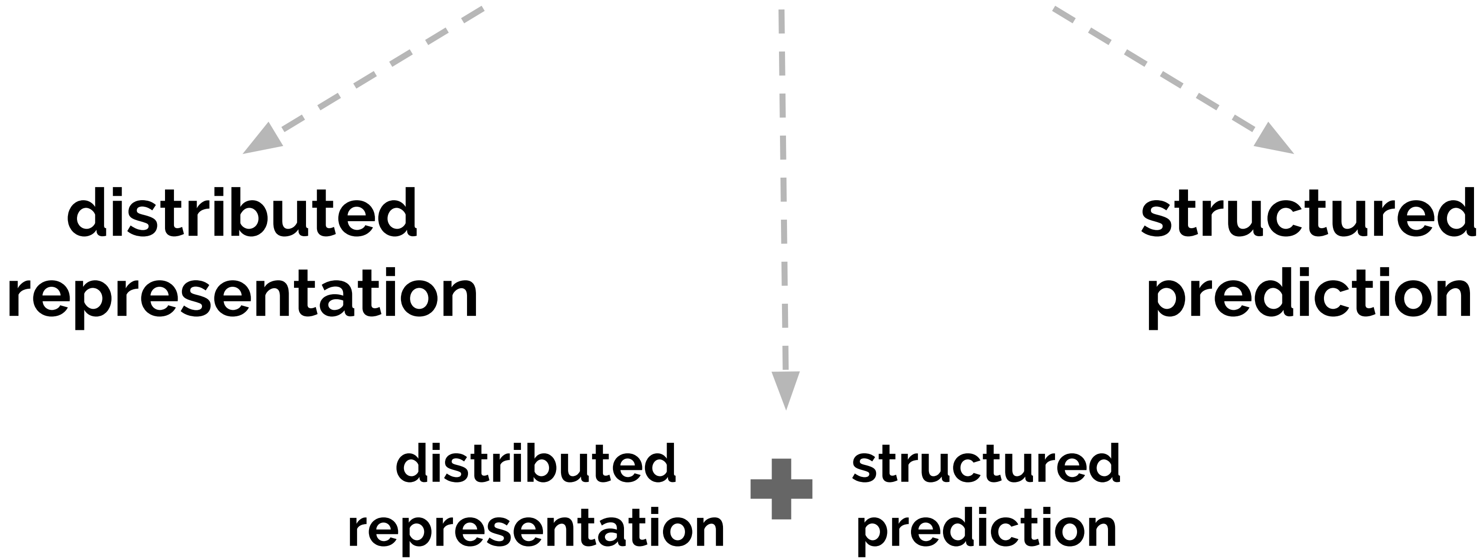
- List of input code representation
  - tokens
  - token context
  - syntax
  - linearized AST
  - ...

Articles’ Confidence Ratio

by citation

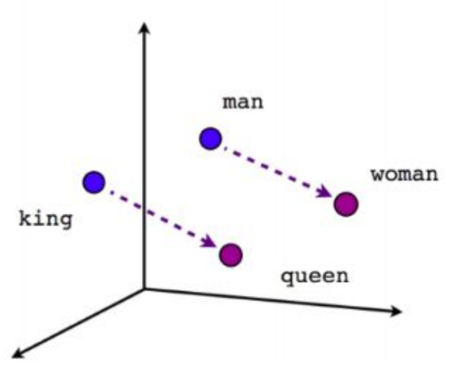
🟩	> 50
🟩	> 20 & ≤ 50
🟨	> 10 & ≤ 20
🟧	> 5 & ≤ 10
🟥	≤ 5

Representational Code Models


distributed representation
structured prediction
distributed representation + structured prediction

Distributed Representations

Definition



- $c \rightarrow R^D$
- the “meaning” of a vector is distributed in its components
- the relation (e.g., similarity) between two representations can be measured within this space

Distributed Representations

Example of its application

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces (2018, August)

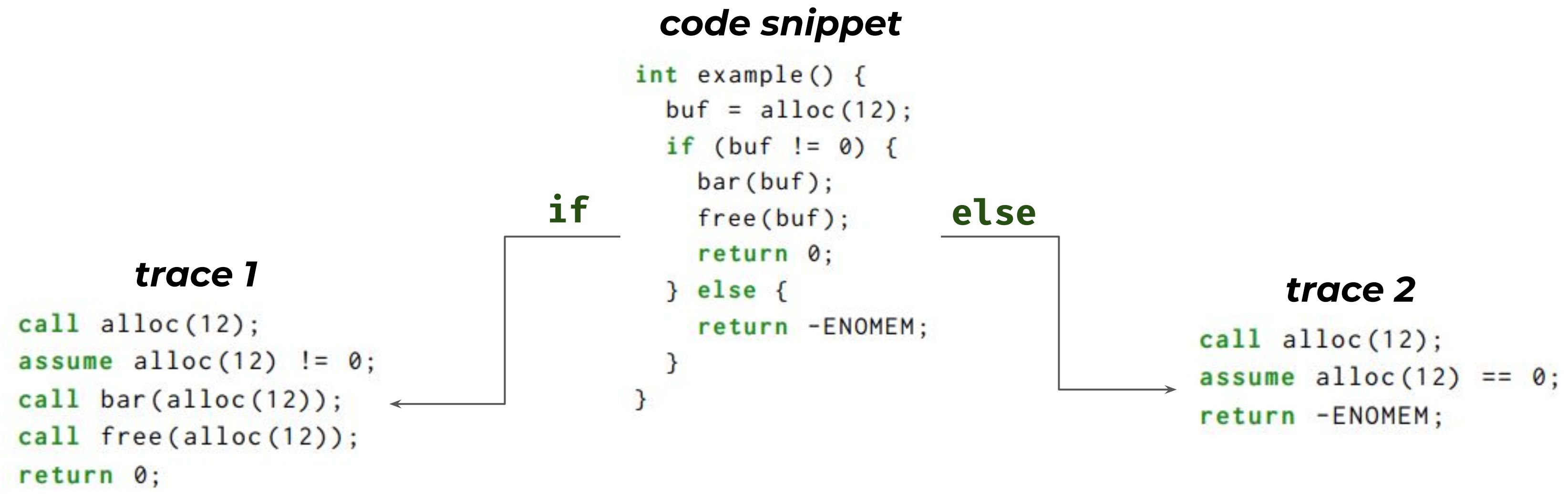
The toolchain consists of three phases:

1) **Transformation** - enumerates all paths, called traces, in each source procedure
2) **Abstraction** - reduction of the number of possible tokens that appear in the traces
3) **Learning** - words that appear in similar contexts close together in an embedding space

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Phase 1

Transformation - get all traces in each source procedure



code snippet

```
int example() {
  buf = alloc(12);
  if (buf != 0) {
    bar(buf);
    free(buf);
    return 0;
  } else {
    return -ENOMEM;
  }
}
```

trace 1

```
call alloc(12);
assume alloc(12) != 0;
call bar(alloc(12));
call free(alloc(12));
return 0;
```

trace 2

```
call alloc(12);
assume alloc(12) == 0;
return -ENOMEM;
```

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Phase 2

Abstraction - reduction of the number of various tokens

```
       call alloc(12);                  Called(alloc)
assume alloc(12) == 0;  ............>   RetEq(alloc, 0)
       return -ENOMEM;                  RetError(ENOMEM)
```

Example derivations for some abstractions:

`call foo()`	`Called(foo)`
`assume foo() == 0`	`RetEq(foo, 0)`
`return -C && C in ERR_CODES`	`RetError(ERR_CODES[C]), Error`
`return C && C not in ERR_CODES`	`RetConst(C)`

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Phase 2

Because of long context they introduced two additional abstractions:

These abstractions encode the flow of data in the trace to make relevant contextual information available without the need for arbitrarily large contexts

`call foo(obj)`
`call bar(obj)`
	`ParamShare(foo, bar)`

`call bar(foo())`	`ParamTo(bar, foo)`

These abstractions can be encoded in the following way:

(1) `RetNeq(alloc, 0)` $\Longrightarrow$ `alloc` , `$NEQ` , `0`

(2) `RetNeq(alloc, 0)` $\Longrightarrow$ `alloc` , `$NEQ_0`

(3) `RetNeq(alloc, 0)` $\Longrightarrow$ `alloc_$NEQ` , `0`

(4) `RetNeq(alloc, 0)` $\Longrightarrow$ `alloc_$NEQ_0`

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Phase 3

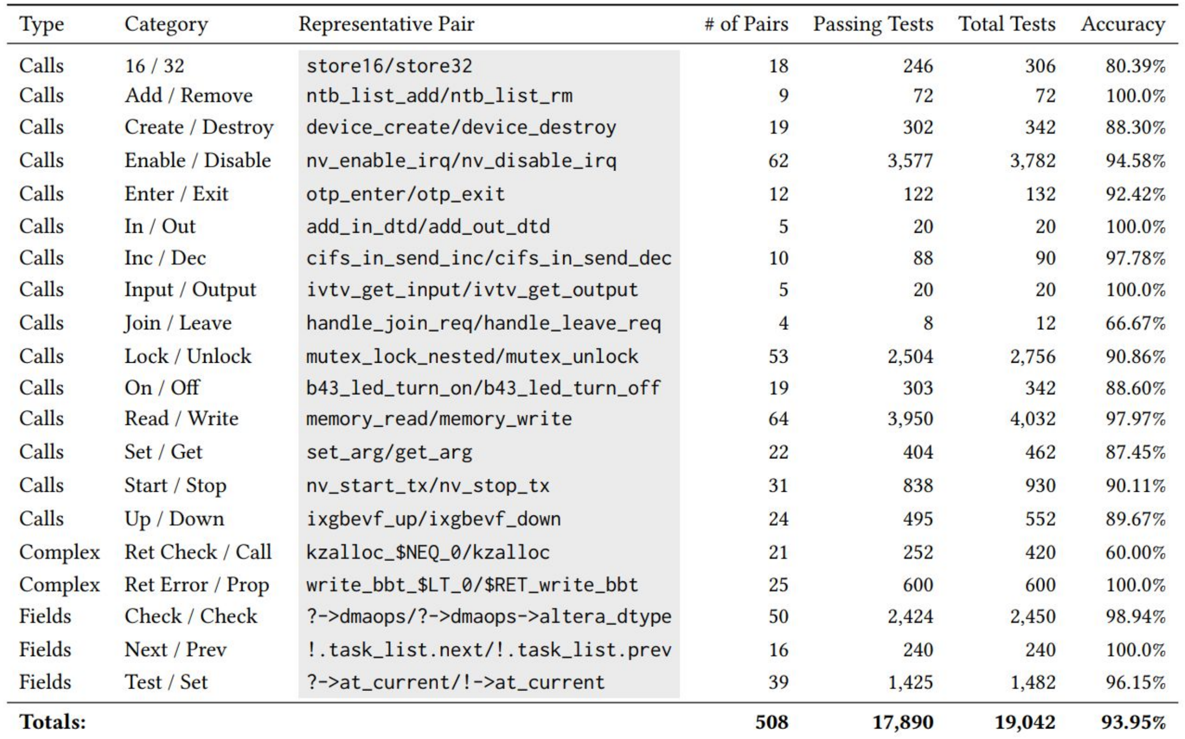
Learning

- GloVe: word-word co-occurrence probabilities encode some form of meaning

`300`	vectors dimension
`50`	window size
`1000`	vocabulary-minimum threshold

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Results

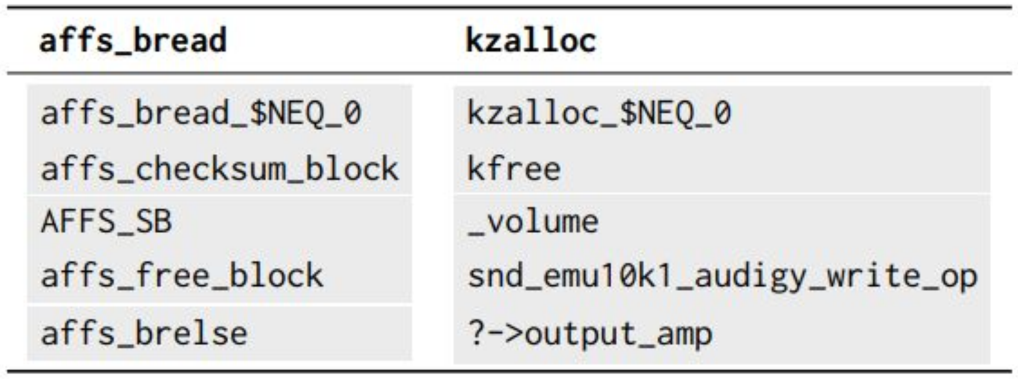


Type	Category	Representative Pair	# of Pairs	Passing Tests	Total Tests	Accuracy
Calls	16 / 32	`store16/store32`	18	246	306	80.39%
Calls	Add / Remove	`ntb_list_add/ntb_list_rm`	9	72	72	100.0%
Calls	Create / Destroy	`device_create/device_destroy`	19	302	342	88.30%
Calls	Enable / Disable	`nv_enable_irq/nv_disable_irq`	62	3,577	3,782	94.58%
Calls	Enter / Exit	`otp_enter/otp_exit`	12	122	132	92.42%
Calls	In / Out	`add_in_dtd/add_out_dtd`	5	20	20	100.0%
Calls	Inc / Dec	`cifs_in_send_inc/cifs_in_send_dec`	10	88	90	97.78%
Calls	Input / Output	`ivtv_get_input/ivtv_get_output`	5	20	20	100.0%
Calls	Join / Leave	`handle_join_req/handle_leave_req`	4	8	12	66.67%
Calls	Lock / Unlock	`mutex_lock_nested/mutex_unlock`	53	2,504	2,756	90.86%
Calls	On / Off	`b43_led_turn_on/b43_led_turn_off`	19	303	342	88.60%
Calls	Read / Write	`memory_read/memory_write`	64	3,950	4,032	97.97%
Calls	Set / Get	`set_arg/get_arg`	22	404	462	87.45%
Calls	Start / Stop	`nv_start_tx/nv_stop_tx`	31	838	930	90.11%
Calls	Up / Down	`ixgbevf_up/ixgbevf_down`	24	495	552	89.67%
Complex	Ret Check / Call	`kzalloc_$NEQ_0/kzalloc`	21	252	420	60.00%
Complex	Ret Error / Prop	`write_bbt_$LT_0/$RET_write_bbt`	25	600	600	100.0%
Fields	Check / Check	`?->dmaops/?->dmaops->altera_dtype`	50	2,424	2,450	98.94%
Fields	Next / Prev	`!.task_list.next/!.task_list.prev`	16	240	240	100.0%
Fields	Test / Set	`?->at_current/!->at_current`	39	1,425	1,482	96.15%
Totals:			**508**	**17,890**	**19,042**	**93.95%**

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

Results

Table 2: Top-5 closest words to affs_bread and kzalloc



affs_bread	kzalloc
affs_bread_$NEQ_0	kzalloc_$NEQ_0
affs_checksum_block	kfree
AFFS_SB	_volume
affs_free_block	snd_emu10k1_audigy_write_op
affs_brelse	?->output_amp

`affs_bread` is a function in the AFS file system that reads a block

`kzalloc` is a memory allocator

- **`affs_bread`**
  - **`affs_bread_$NEQ_0`** - checked to be non-null
  - and other similar functions
- **`kzalloc`**
  - **`kfree_$NEQ_0`** - checked to be non-null
  - last free functions seem out of place

Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces

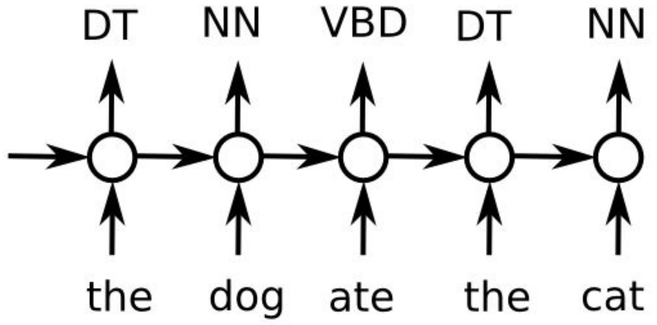
Results: using in downstream tasks

- trained a model to predict the error code that each trace should return
- used LSTM
- evaluated by two ways
  - bug finding
    - if the model change change error code, check this case
    - identified an incorrect error code in 57 of our 68 tests
  - repair / suggestion
    - predict the three most likely error codes for each trace in the test set
    - had a top-3 accuracy of 76.5%

Structured Prediction
Definition

- SP is an umbrella term for supervised machine learning techniques that involves **predicting structured objects**, rather than scalar discrete or real values

- Example: sequence tagging



- Well suited to code, because it can exploit the semantic and syntactic structure of code

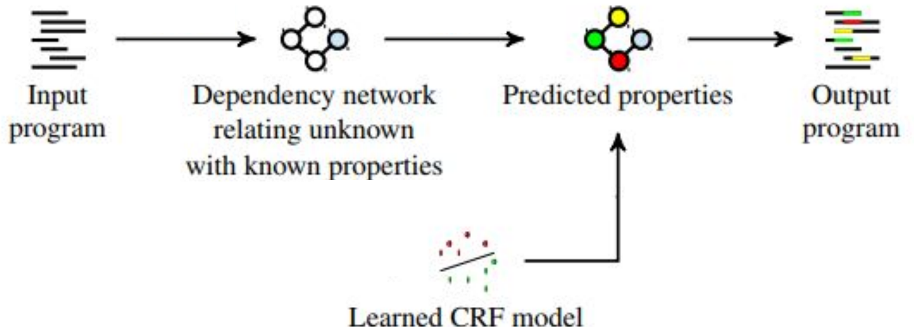
Structured Prediction
Definition

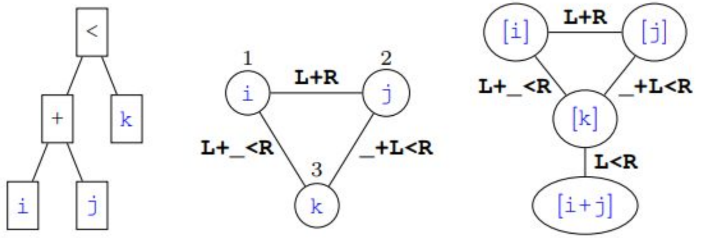
- Main techniques:
  - Conditional random field
    - CRFs is a type of discriminative undirected probabilistic graphical model
    - It is used to encode known relationships between observations and construct consistent interpretations
  - Structured support vector machine
  - Structured k-Nearest Neighbours
  - Recurrent neural network, in particular Elman network

Structured Prediction

Example of its application

Predicting program properties from “big code” (2015) - JSNice





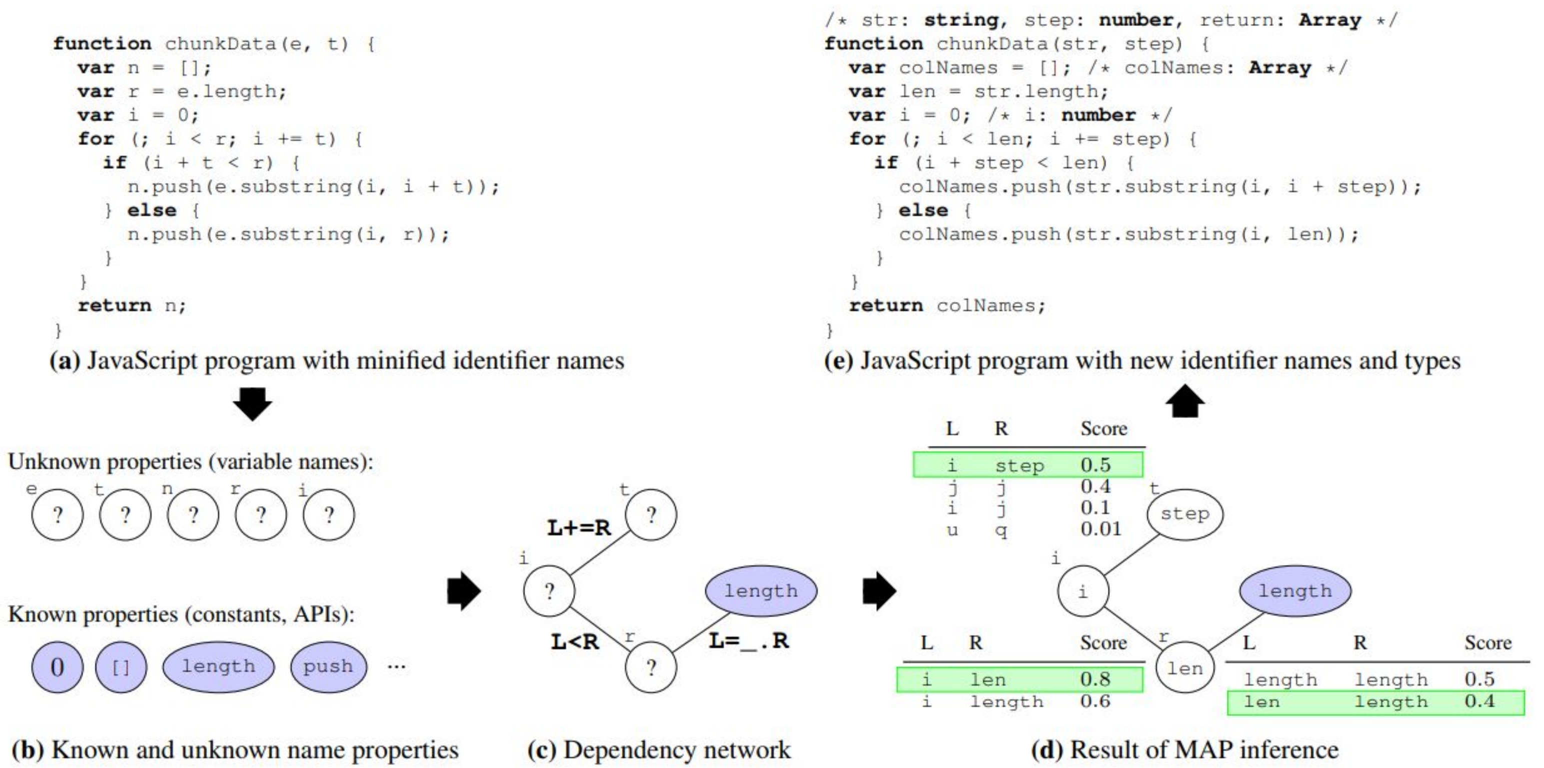
(a) (b) (c)

(a) the AST of expression i+j<k

two dependency networks built from the AST relations:
(b) for name predictions,
(c) for type predictions.

Predicting program properties from “big code”

Overview of the name inference procedure



```
function chunkData(e, t) {
  var n = [];
  var r = e.length;
  var i = 0;
  for (; i < r; i += t) {
    if (i + t < r) {
      n.push(e.substring(i, i + t));
    } else {
      n.push(e.substring(i, r));
    }
  }
  return n;
}
```

(a) JavaScript program with minified identifier names

```
/* str: string, step: number, return: Array */
function chunkData(str, step) {
  var colNames = []; /* colNames: Array */
  var len = str.length;
  var i = 0; /* i: number */
  for (; i < len; i += step) {
    if (i + step < len) {
      colNames.push(str.substring(i, i + step));
    } else {
      colNames.push(str.substring(i, len));
    }
  }
  return colNames;
}
```

(e) JavaScript program with new identifier names and types

Unknown properties (variable names):

Known properties (constants, APIs):

(b) Known and unknown name properties

(c) Dependency network

L	R	Score
i	step	0.5
j	j	0.4
i	j	0.1
u	q	0.01

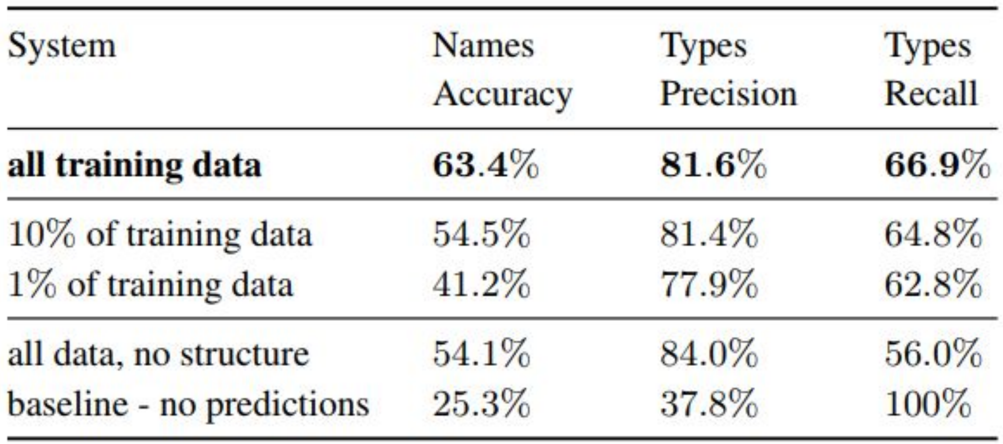
L	R	Score
i	len	0.8
i	length	0.6

L	R	Score
length	length	0.5
len	length	0.4

(d) Result of MAP inference

Predicting program properties from "big code"

Results



System	Names Accuracy	Types Precision	Types Recall
all training data	**63.4%**	**81.6%**	**66.9%**
10% of training data	54.5%	81.4%	64.8%
1% of training data	41.2%	77.9%	62.8%
all data, no structure	54.1%	84.0%	56.0%
baseline - no predictions	25.3%	37.8%	100%

- Names
  - the systems trained on less data have significantly lower precision showing the **importance of the amount of training data**
- Types
  - was evaluated on production JavaScript applications that typically have short methods with complex relationships, the recall for predicting program types is only 66.9%
  - *we note that none of the types we infer can be inferred by regular forward type analysis*
  - to increase the precision and recall **adding more (semantic) relationships between program elements will be of higher importance** than adding more training data

Distributed Representations + Structured Prediction

Definition

- two first types of representational code models are not mutually exclusive

- for example, structured prediction, such as predicting a sequence of elements, can be combined with distributed representations

Distributed Representations + Structured Prediction

Example of its application

- **Suggesting accurate method and class names** (2015)
  - use distributed representations to predict sequences of identifier sub-tokens to build a single token

- **Gated graph sequence neural networks** (2016)
  - learn distributed representations for the nodes of a fixed heap graph by considering its structure and the interdependencies among the nodes

- **Learning to represent programs with graphs** (2018, May)
  - predict the data flow graph of code by learning to paste snippets of code into existing code and adapting the variables used
  - this article will be discussed later

Distributed Representations + Structured Prediction

Example of its application

Deep Learning Similarities from Different Representations of Source Code (2018, May)

For each code snippet four different representations are extracted and normalized:

	extraction	**normalization**
Identifiers	leaf node in AST	replace it with its type (`<int>`)
AST	a pre-order visit of this code snippet sub-tree	remove two types of nodes: simpleName and qualifiedName
CFG	Soot - framework; method - graph, class - forest	N/A
bytecode	a code fragment is expressed as a stream of bytecode mnemonic opcodes	remove const

Deep Learning Similarities from Different Representations of Source Code

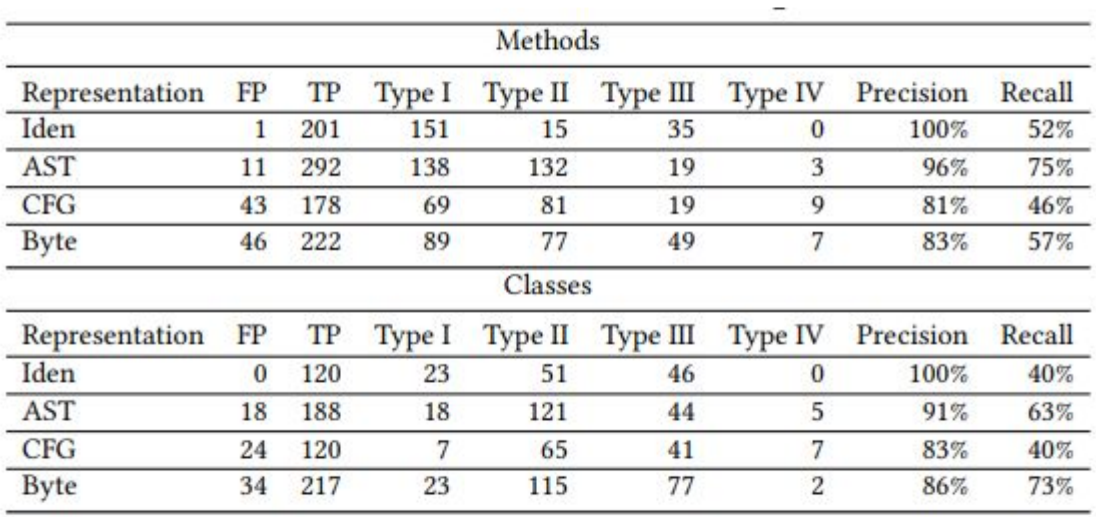
Embedding Learning Strategy

- Learn a single embedding for each representation:
  - Ident, AST, bytecode
    - `r = w1, w2, .., wj`
      - learn an embedding for each term `wi` (RtNN)
      - recursively combine the word embeddings to learn an encoding for the entire sentence r (Recursive Autoencoder)
  - CFG
    - employ the Graph Embedding Technique HOPE

- These four models are combined using Ensemble Learning
  - each single-representation model expresses its own vote about the similarity of two code fragments and these decisions are combined in a single label

Deep Learning Similarities from Different Representations of Source Code

Results



Methods								
Representation	FP	TP	Type I	Type II	Type III	Type IV	Precision	Recall
Iden	1	201	151	15	35	0	100%	52%
AST	11	292	138	132	19	3	96%	75%
CFG	43	178	69	81	19	9	81%	46%
Byte	46	222	89	77	49	7	83%	57%

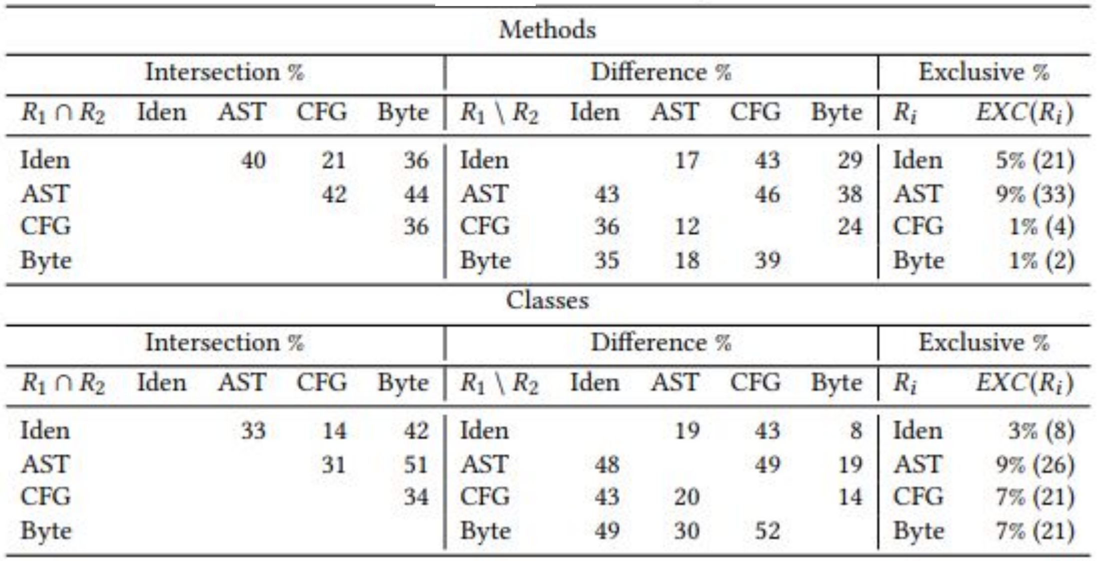
Classes								
Representation	FP	TP	Type I	Type II	Type III	Type IV	Precision	Recall
Iden	0	120	23	51	46	0	100%	40%
AST	18	188	18	121	44	5	91%	63%
CFG	24	120	7	65	41	7	83%	40%
Byte	34	217	23	115	77	2	86%	73%

- Iden
  - fails to detect a significant percentage of clones
- CFG & Byte
  - a lot of FP
  - in case when iden and AST models do not detect clone these models show good results
- AST
  - have the best overall balance between precision and recall

Deep Learning Similarities from Different Representations of Source Code

Results

Complementarity Metrics (for TP)



Methods											
Intersection %					Difference %					Exclusive %	
$R_1 \cap R_2$	Iden	AST	CFG	Byte	$R_1 \setminus R_2$	Iden	AST	CFG	Byte	R_i	$EXC(R_i)$
Iden		40	21	36	Iden		17	43	29	Iden	5% (21)
AST			42	44	AST	43		46	38	AST	9% (33)
CFG				36	CFG	36	12		24	CFG	1% (4)
Byte					Byte	35	18	39		Byte	1% (2)

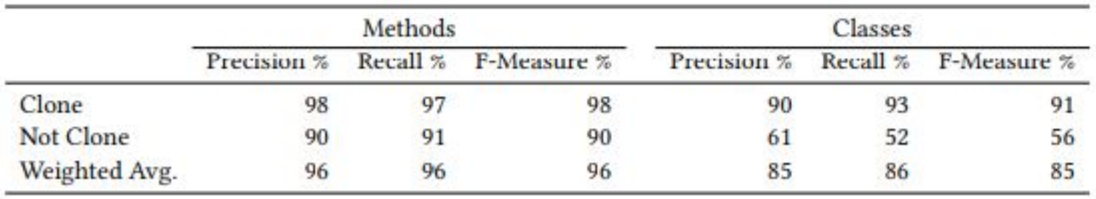
Classes											
Intersection %					Difference %					Exclusive %	
$R_1 \cap R_2$	Iden	AST	CFG	Byte	$R_1 \setminus R_2$	Iden	AST	CFG	Byte	R_i	$EXC(R_i)$
Iden		33	14	42	Iden		19	43	8	Iden	3% (8)
AST			31	51	AST	48		49	19	AST	9% (26)
CFG				34	CFG	43	20		14	CFG	7% (21)
Byte					Byte	49	30	52		Byte	7% (21)

- Intersection
  - relatively small overlap among the candidate sets suggests that these representations complement each other
- Difference
  - detected by a certain representation and missed by the other
- Exclusive
  - detected by a certain representation and missed by all the others

Deep Learning Similarities from Different Representations of Source Code

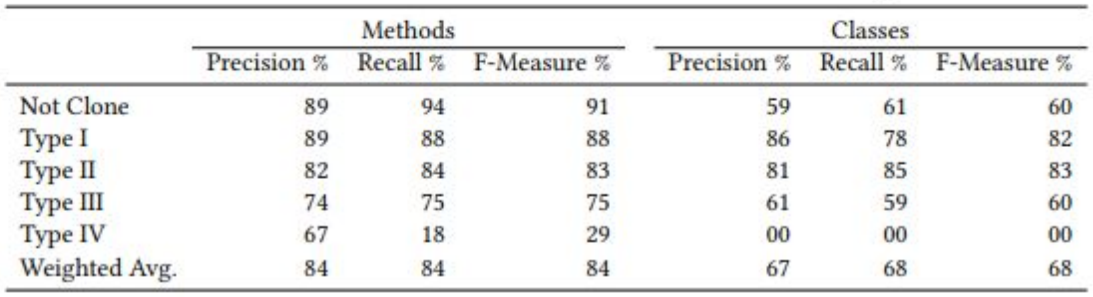
Results

Performance of the *CloneDetector*



	Methods			Classes		
	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Clone	98	97	98	90	93	91
Not Clone	90	91	90	61	52	56
Weighted Avg.	96	96	96	85	86	85

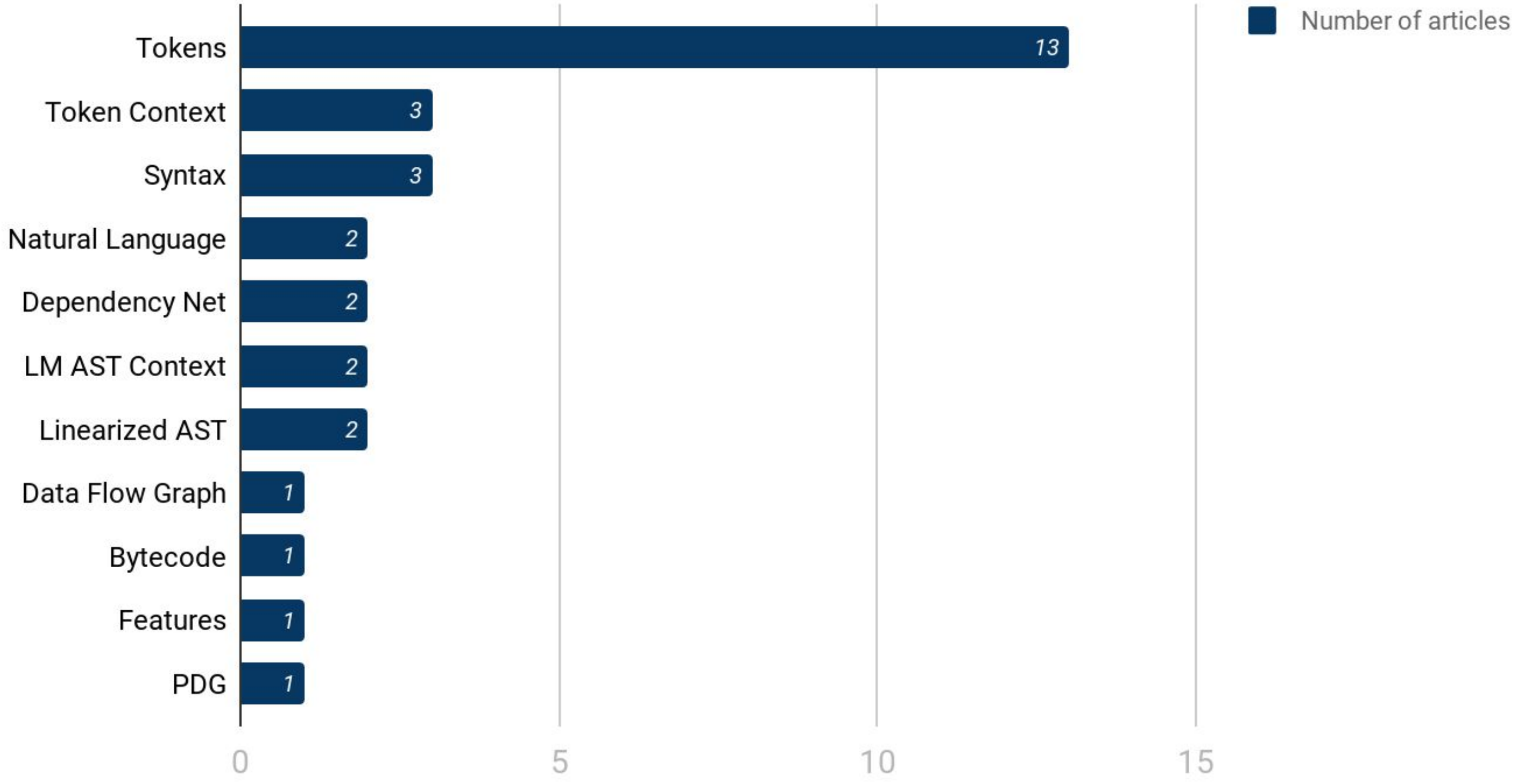
Performance of the *CloneClassifier*



	Methods			Classes		
	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Not Clone	89	94	91	59	61	60
Type I	89	88	88	86	78	82
Type II	82	84	83	81	85	83
Type III	74	75	75	61	59	60
Type IV	67	18	29	00	00	00
Weighted Avg.	84	84	84	67	68	68

Input Code Representation

Application Statistics



source: A Survey of Machine Learning for Big Code and Naturalness

Input Code Representation

Tokens

- Natural language models for predicting programming comments (2013)
- Toward deep learning software repositories (2015)
- Exploring the Use of Deep Learning for Feature Location (2015)
- A convolutional attention network for extreme summarization of source code (2016)
- Summarizing source code using a neural attention model (2016)
- Bug detection with n-gram language models (2016)
- End-to-end Deep Learning of Optimization Heuristics (2017)
- Semantically enhanced software traceability using deep learning techniques (2017)
- DeepFix: Fixing common C language errors by deep learning (2017)
- Automatically generating commit messages from diffs using neural machine translation (2017)
- A neural architecture for generating natural language descriptions from source code changes (2017)
- Deep reinforcement learning for programming language correction (2018, January)

Input Code Representation

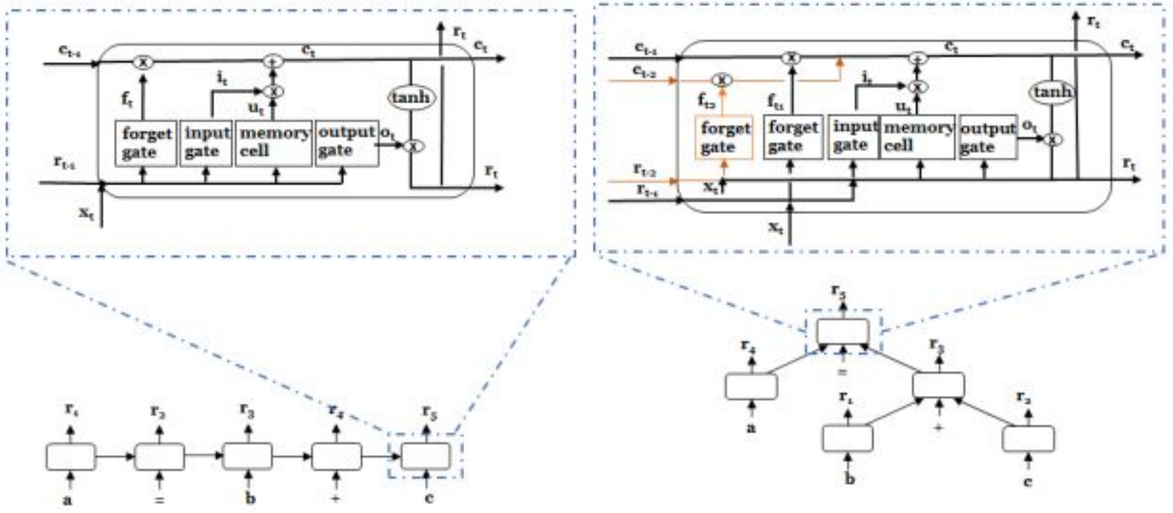
Token Context

- Suggesting accurate method and class names (2015)
  - local (surrounding tokens) & global (set of features) contexts
- A deep language model for software code (2016)
  - use LSTM
- Context2Name: A deep learning-based approach to infer natural variable names from usage contexts (2018, August)
  - summary usage (all contexts are concatenated)

Input Code Representation

Token Context

- **Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code** (2017)
    - it leverages the AST to capture structure information of code fragments
    - use LSTM to extract the semantic information carried by lexical tokens of source codes



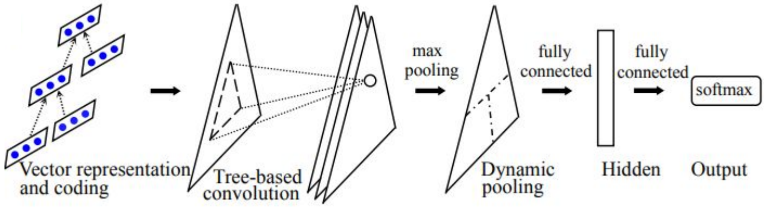
(a) Traditional LSTM for expression 'a=b+c;'

(b) AST-based LSTM for expression 'a=b+c;'

Input Code Representation

Syntax

- Learning program embeddings to propagate feedback on student code (2015)
  - Hoare Triples
- Convolutional neural networks over tree structures for programming language processing (2015)

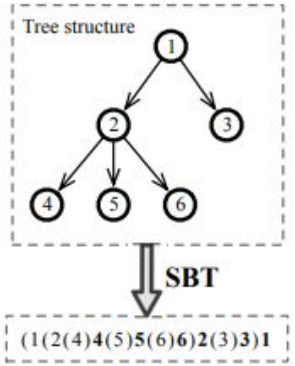


- Deep learning to find bugs (2017)
  - Egor performed

Input Code Representation

Linearized AST

- CodeSum: Translate program language to natural language (2017)
  - Structure-based Traversal of AST



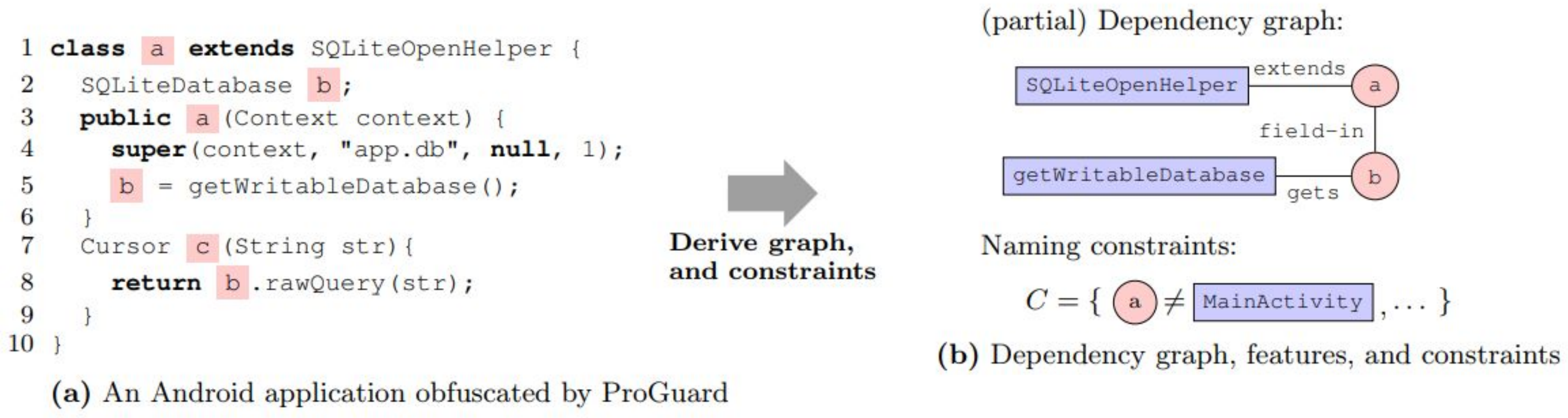
- From the root node, we first use a pair of brackets to represent the tree structure and put the root node itself behind the right bracket, that is **(1)1**
- Next, traverse the subtrees of the root node and put all root nodes of subtrees into the brackets, i.e., **(1(2)2(3)3)1**
- Recursively traverse each subtree until all nodes are traversed and get the final sequence **(1(2(4)4(5)5(6)6)2(3)3)1**

- code2vec: Learning Distributed Representations of Code (2018, April)
  - Zarina performed
- code2seq: Generating Sequences from Structured Representations of Code (2018, October)
  - as code2vec but for code summarization task

Input Code Representation

Dependency Net

- Statistical Deobfuscation of Android Applications (2016) - DeGuard



```
class a extends SQLiteOpenHelper {
  SQLiteDatabase b;
  public a(Context context) {
    super(context, "app.db", null, 1);
    b = getWritableDatabase();
  }
  Cursor c(String str){
    return b.rawQuery(str);
  }
}
```

$$C = \{ \text{a} \neq \texttt{MainActivity}, \ldots \}$$

(a) An Android application obfuscated by ProGuard

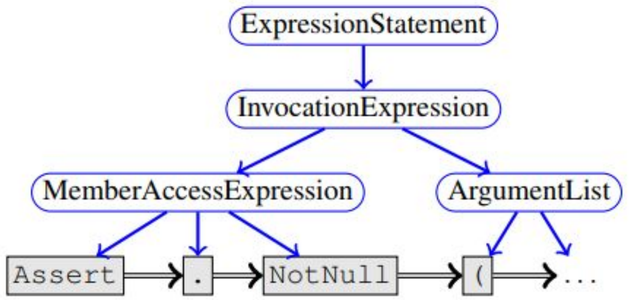
(b) Dependency graph, features, and constraints

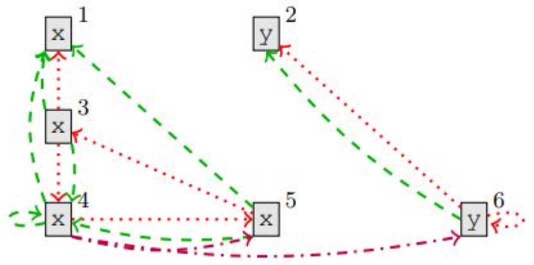
- Predicting program properties from "big code" (2015) - JSNice
  - have already reviewed

Input Code Representation

PDG

- [Learning to Represent Programs with Graphs](#) (2018, May)
  - the backbone of a program graph is AST
    - syntax nodes (non-terminals) & syntax tokens (terminals)
    - edges that connect them: **`child`** & **`nextToken`**
  - to capture the flow of control and data through a program, use additional edges
    - **`computedFrom:`** `v = expr;`
    - **`returnsTo:`** edge between `returnToken` and method declaration
    - **`lastWrite`**
    - **`lastLexicalUse:`** `if() {..x..} else {..x..}`
    - ...

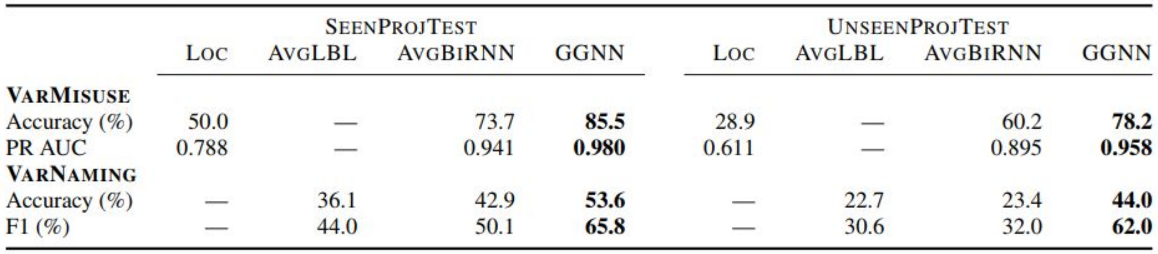




(b) Data flow edges for (x¹, y²) = `Foo();` `while` (x³ > 0) x⁴ = x⁵ + y⁶ (indices added for clarity), with red dotted LastUse edges, green dashed LastWrite edges and dashdotted purple ComputedFrom edges.

Learning to Represent Programs with Graphs

Results



	SEENPROJTEST				UNSEENPROJTEST			
	LOC	AVGLBL	AVGBIRNN	GGNN	LOC	AVGLBL	AVGBIRNN	GGNN
VARMISUSE								
Accuracy (%)	50.0	—	73.7	**85.5**	28.9	—	60.2	**78.2**
PR AUC	0.788	—	0.941	**0.980**	0.611	—	0.895	**0.958**
VARNAMING								
Accuracy (%)	—	36.1	42.9	**53.6**	—	22.7	23.4	**44.0**
F1 (%)	—	44.0	50.1	**65.8**	—	30.6	32.0	**62.0**

- Loc (simple two-layer bidirectional GRU)
  - this baseline allows to evaluate how important the usage context information is
- AVGBiRNN (an extension to Loc)
  - is a significantly stronger baseline that already takes some structural information into account
- AVGLBL (a log-bilinear model)

Input Code Representation

Data Flow Graph

- Automatically Generating Features for Learning Program Analysis Heuristics for C-Like Languages (2017)

LM AST Context

- Structured Generative Models of Natural Source Code (2014)

Natural Language

- Bimodal modelling of source code and natural language (2015)
- Deep API learning (2016)

A Survey of Machine Learning for Big Code and Naturalness (2017, v2 in 2018)

is the main resource for the second part, Input Code Representation

Contacts:

- TG - @natalymr