

Big Code Analysis

Wei Le

April 17, 2019

Big code

- ▶ open source software systems: such as Linux, MYSQL, Django, Ant and OpenEJB
- ▶ source code and meta data available: changes, review processes
- ▶ *big code*: billions of tokens of code and millions of instances of data
- ▶ data-driven: we can thus calculate statistical distributional properties, probabilistic models of source code

Who Cares?

- ▶ Darpa: Mining and Understanding Software Enclaves (MUSE)
- ▶ Startup: Deep Code
- ▶ Microsoft and ABB: using data to drive software development decisions

Topics

- ▶ Theme: how to map the code and the goal to machine learning models, then apply existing learning methods
- ▶ In this lecture, we will focus on how to map the problem rather the learning part
 - ▶ machine learning models used with different code representations
 - ▶ goals and applications
 - ▶ future directions

Applications

Applications: recommender system

Based on the language model built from copra and based on the current context of code, comments ...

- ▶ code completion: API sequence, any code
- ▶ comment completion

Applications: translations

- ▶ Java programs to C and vice versa
- ▶ code to text/pseudo code
- ▶ text to code

Applications: mining patterns

- ▶ loop idioms
- ▶ code convention, anomaly code
- ▶ code clone and repetitive code

Applications: information retrieval

- ▶ code search
- ▶ traceability

High level thoughts

Program Analysis and Machine Learning

- ▶ *Probabilistic Models of Code*: a probability distribution over code artifacts
- ▶ Code artifacts: file, method, tokens, variables, types, variable interactions ... (anything done by program analysis to extract a representation from code)
- ▶ Goal: dependencies between elements in the code that can represent the developers intention and semantics of code (e.g., name to explain a function)
- ▶ More program analysis applied, the representation more represents the semantics of code, less data needed and less pressure for learning, e.g., given tokens, machine learning needs to identify the syntactic constraints between tokens; however if we provide AST, such constraints are given in the input, no need to learn.

Program Analysis and Machine Learning

- ▶ there is exploitable regularity across human-written code that can be “absorbed” and generalized by a learning component which then transfers its knowledge to probabilistically reason about the new code
- ▶ machine learning models “compress” the information and then “decompress” with additional information when applying to the new code
- ▶ machine learning approaches want to capture the (long) dependencies between code entity (structure)
- ▶ machine learning: good at handling uncertainty, ambiguity, noise, but code is deterministic, so when to apply the probabilistic method? fusing multiple sources of information, ambiguous information like comments

Source Code and Natural Language

- ▶ *The Naturalness Hypothesis: software is a form of human communication (Knuth); software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.*
- ▶ Intuition: developers prefer to write and read code that is conventional, idiomatic, and familiar because it helps understanding and maintaining software systems.
- ▶ So the statistical methods work for natural language processing can be potentially applicable to the source code as they hold similar properties
- ▶ In fact, the evidence has show that the models for natural language work effectively for source code

Big code analysis: general approach

- ▶ Step 1: using machine learning to create probabilistic source code models that learn how developers naturally write and use code. (learning)
- ▶ Step 2: using these models to augment the tools for recommendations or program analysis (inference/prediction)
- ▶ statistical method: make hypotheses along with probabilistic confidence value to predict what developers might want to do next or what properties might be true for a trunk of code
- ▶ another direction is to learn mapping between code and meta-data such as commit messages, blog post, bug report

How do we do it?

Probabilistic models of code

- ▶ *generative models*: defines a probabilistic distribution over code by stochastically modeling the generation of smaller and simpler parts of code, e.g., what is the probability of deriving to this sub-tree as the left child of the root
- ▶ *representational models*: conditional probability distribution over code element properties, e.g., what is the probability of types X, Y and Z for this variable a (based on, e.g., surrounding context of a)
- ▶ *pattern mining models*: unsupervised learning to infer a latent (not observable) structure within code

Generative Models

- ▶ Machine learning approaches have worked for generating other structure data, e.g., images, chemical structure
- ▶ Modeling how code is written
- ▶ assign a non-zero probability to every possible snippet of code
- ▶ three types of models:
 - ▶ *language models*
 - ▶ *code-generative multimodal model*
 - ▶ *transducer model*

Generative Models

How to generate code in different context:

- ▶ Input: training data D
 - ▶ A output code representation: c
 - ▶ A possibly empty context: $C(c)$
 - ▶ Probabilistic distribution: $P_D(c|C(c))$
-
- ▶ language model: $C(c) = \emptyset$
 - ▶ code-generative multimodal model: $C(c)$ is a non-code modality, e.g., natural language
 - ▶ transducer model: $C(c)$ is also code

Generative Models

How to represent the code in generative models

- ▶ "a bag of words": a set of tokens
- ▶ a sequence of tokens: n-gram
- ▶ abstract syntax trees
- ▶ graphs

Reference	Type	Representation	P_D	Application
Aggarwal et al. [2]	$P_D(c s)$	Token	Phrase	Migration
Allamanis and Sutton [11]	$P_D(c)$	Token	n -gram	—
Allamanis et al. [5]	$P_D(c)$	Token + Location	n -gram	Coding Conventions
Allamanis and Sutton [12]*	$P_D(c)$	Syntax	Grammar (pTSG)	—
Allamanis et al. [13]*	$P_D(c m)$	Syntax	Grammar (NN-LBL)	Code Search/Synthesis
Amodio et al. [16]	$P_D(c)$	Syntax + Constraints	RNN	—
Barone and Sennrich [21]	$P_D(c m)$	Token	Neural SMT	Documentation
Beltramelli [23]	$P_D(c m)$	Token	NN (Encoder-Decoder)	GUI Code Synthesis
Bhatia and Singh [25]	$P_D(c)$	Token	RNN (LSTM)	Syntax Error Correction
Bhoopchand et al. [26]	$P_D(c)$	Token	NN (Pointer Net)	Code Completion
Bielik et al. [29]	$P_D(c)$	Syntax	PCFG + annotations	Code Completion
Campbell et al. [35]	$P_D(c)$	Token	n -gram	Syntax Error Detection
Cerulo et al. [37]	$P_D(c)$	Token	Graphical Model (HMM)	Information Extraction
Cummins et al. [48]	$P_D(c)$	Character	RNN (LSTM)	Benchmark Synthesis
Dam et al. [49]	$P_D(c)$	Token	RNN (LSTM)	—
Gulwani and Marron [77]	$P_D(c m)$	Syntax	Phrase Model	Text-to-Code
Gvero and Kuncak [82]	$P_D(c)$	Syntax	PCFG + Search	Code Synthesis
Hellendoorn et al. [85]	$P_D(c)$	Token	n -gram	Code Review
Hellendoorn and Devanbu [84]	$P_D(c)$	token	n -gram (cache)	—
Hindle et al. [88]	$P_D(c)$	Token	n -gram	Code Completion
Hsiao et al. [93]	$P_D(c)$	PDG	n -gram	Program Analysis
Lin et al. [118]	$P_D(c m)$	Tokens	NN (Seq2seq)	Synthesis
Ling et al. [120]	$P_D(c m)$	Token	RNN + Attention	Code Synthesis
Liu [121]	$P_D(c)$	Token	n -gram	Obfuscation
Karaivanov et al. [102]	$P_D(c s)$	Token	Phrase	Migration
Karpathy et al. [103]	$P_D(c)$	Characters	RNN (LSTM)	—
Kushman and Barzilay [110]	$P_D(c m)$	Token	Grammar (CCG)	Code Synthesis
Maddison and Tarlow [126]	$P_D(c)$	Syntax with scope	NN	—
Menon et al. [129]	$P_D(c m)$	Syntax	PCFG + annotations	Code Synthesis
Nguyen et al. [140]	$P_D(c s)$	Token	Phrase	Migration
Nguyen et al. [144]	$P_D(c)$	Token + parse info	n -gram	Code Completion
Nguyen et al. [141]	$P_D(c s)$	Token + parse info	Phrase SMT	Migration
Nguyen and Nguyen [139]	$P_D(c)$	Partial PDG	n -gram	Code Completion
Oda et al. [146]	$P_D(c s)$	Syntax + Token	Tree-to-String + Phrase	Pseudocode Generation
Patra and Pradel [153]	$P_D(c)$	Syntax	Annotated PCFG	Fuzz Testing
Pham et al. [154]	$P_D(c)$	Bytecode	Graphical Model (HMM)	Code Completion
Pu et al. [160]	$P_D(c s)$	Token	NN (Seq2seq)	Code Fixing
Rabinovich et al. [162]*	$P_D(c m)$	Syntax	NN (LSTM-based)	Code Synthesis
Raychev et al. [166]	$P_D(c)$	Token + Constraints	n -gram/ RNN	Code Completion
Ray et al. [163]	$P_D(c)$	Token	n -gram (cache)	Bug Detection
Raychev et al. [164]	$P_D(c)$	Syntax	PCFG + annotations	Code Completion
Saraiva et al. [173]	$P_D(c)$	Token	n -gram	—
Sharma et al. [175]	$P_D(c)$	Token	n -gram	Information Extraction
Tu et al. [180]	$P_D(c)$	Token	n -gram (cache)	Code Completion
Vasilescu et al. [181]	$P_D(c s)$	Token	Phrase SMT	Deobfuscation
Wang et al. [185]	$P_D(c)$	Syntax	NN (LSTM)	Code Completion
White et al. [188]	$P_D(c)$	Token	NN (RNN)	—
Yadid and Yahav [194]	$P_D(c)$	Token	n -gram	Information Extraction
Yin and Neubig [196]	$P_D(c m)$	Syntax	NN (Seq2seq)	Synthesis

Representational Models

- ▶ $P_D(\pi|f(c))$: conditional probability distribution of a code property π , where f is a function that transforms the code c into a target representation
- ▶ *Distributed representation*
- ▶ *Structured prediction*




Distributed Representation of Code

- ▶ *Local representation*: one to one mapping, map code to a value
- ▶ *Distributed representation*: many to many mapping, map many code artifacts to many values – input: code snippets, output: vector with continuous values
- ▶ advantage:
 - ▶ *embedding*: a mapping of discrete, categorical, variable to a vector of continuous numbers, low dimensional vector representation for objects
 - ▶ Semantically similar objects are mapped to close vectors (this effect is the same as local representation)
 - ▶ Word embedding example: $\text{vec}(\text{"king"}) - \text{vec}(\text{"man"}) + \text{vec}(\text{"woman"}) = \text{vec}(\text{"queen"})$

Application: Semantic labeling of code snippets

```
String[] f(final String[] array) {  
    final String[] newArray = new String[array.length];  
    for (int index = 0; index < array.length; index++) {  
        newArray[array.length - index - 1] = array[index];  
    }  
    return newArray;  
}
```

Predictions

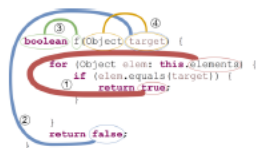
reverseArray		77.34%
reverse		18.18%
subArray		1.45%
copyArray		0.74%

- ▶ Training data: code snippets, and their labels
- ▶ Goal: predicting a name of a method (for easy understanding)
- ▶ code embedding: capture the semantic similarity of code snippets

Overall Approach

1. obtain a path from ast
2. paths are mapped to a real-value vector
3. learn the model
4. predict the name based on the model and the vector

Paths Used to Predict Labels



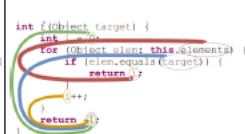
(a)

Predictions:		
contains	<div><div></div><div></div></div>	90.93%
matches	<div><div></div><div></div></div>	3.54%
canHandle	<div><div></div><div></div></div>	1.15%
equals	<div><div></div><div></div></div>	0.87%
containsExact	<div><div></div><div></div></div>	0.77%



(b)

Predictions		
get	<div><div></div><div></div></div>	31.09%
getProperty	<div><div></div><div></div></div>	20.25%
getValue	<div><div></div><div></div></div>	14.34%
getElement	<div><div></div><div></div></div>	14.00%
getObject	<div><div></div><div></div></div>	6.05%



(c)

Predictions		
indexOf	<div><div></div><div></div></div>	
getIndex	<div><div></div><div></div></div>	
findIndex	<div><div></div><div></div></div>	
indexOfNull	<div><div></div><div></div></div>	
getInstructionIndex	<div><div></div><div></div></div>	

Formal Definitions

Definition 2 (AST path). An AST-path of length k is a sequence of the form: $n_1 d_1 \dots n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminals, for $i \in [2..k]$: $n_i \in N$ are nonterminals and 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: $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \delta(n_i)$. 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.

Using this definition we define a *path-context* as a tuple of an AST path and the values associated with its terminals:

Definition 3 (Path-context). Given an AST Path p , its path-context is a triplet $\langle x_s, p, x_t \rangle$ where $x_s = \phi(start(p))$ and $x_t = \phi(end(p))$ 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.

Example 3.1. A possible path-context that represents the statement: “ $x = 7$;” would be:

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

To limit the size of the training data and reduce sparsity, it is possible to limit different parameters of the paths. Following earlier works, we limit the paths by maximum *length* — the maximal value of k , and limit the maximum *width* — the maximal difference in child index between two child nodes of the same intermediate node. These values are determined empirically as hyperparameters of our model.

Distributed Representation of a Method

- ▶ two vectors: V - each row contains token values, P - A bag of path-context: $C = \{(x_s, p, x_t)\}$
- ▶ the weight is initialized with random values, then learned from the training data

Structure prediction: Predicting program properties from big code

```
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

Predicting program properties from big code

- ▶ code as a variable dependence network
- ▶ represent each variable as a node, the property of some node we know while the property of some node we don't know
- ▶ represent variable interaction as *conditional random field (CRF)* – a type of probabilistic graphical models that represent the conditional probability of labels y given observations of x $P(y|x)$
- ▶ train CRF to predict names and types of variables for Javascripts

Predicting program properties from big code

```
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;  
}
```

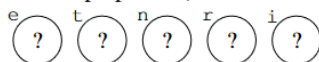
(a) JavaScript program with minified identifier names

```
/* 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

Predicting program properties from big code

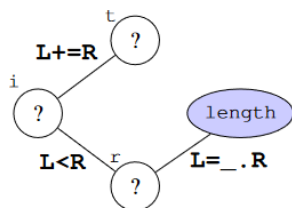
Unknown properties (variable names):



Known properties (constants, APIs):

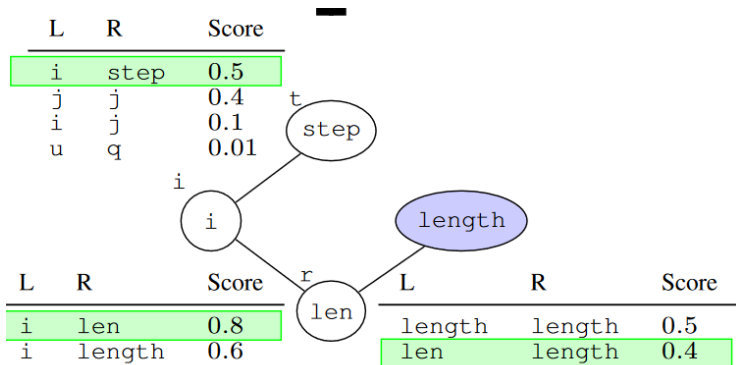


(b) Known and unknown name properties



(c) Dependency network

Predicting program properties from big code



Reference	Input Code Representation (c)	Target (π)	Intermediate Representation (f)	Application
Allamanis et al. [6]	Token Context	Identifier Name	Distributed	Naming
Allamanis et al. [13]*	Natural Language	LM (Syntax)	Distributed	Code Search
Allamanis et al. [10]	Tokens	Method Name	Distributed	Naming
Allamanis et al. [8]	PDG	Variable Use Bugs	Distributed	Program Analysis
Bavishi et al. [22]	Token Context	Identifier Name	Distributed	Naming
Bichsel et al. [27]	Dependency Net	Identifier Name	CRF (GM)	Deobfuscation
Bruch et al. [33]	Partial Object Use	Invoked Method	Localized	Code Completion
Chae et al. [38]	Data Flow Graph	Static Analysis	Localized	Program Analysis
Corley et al. [44]	Tokens	Feature Location	Distributed	Feature Location
Cummins et al. [47]	Tokens	Optimization Flags	Distributed	Optimization Heuristics
Dam et al. [49]*	Token Context	LM (Tokens)	Distributed	—
Gu et al. [76]	Natural Language	API Calls	Distributed	API Search
Guo et al. [79]	Tokens	Traceability link	Distributed	Traceability
Gupta et al. [81]	Tokens	Code Fix	Distributed	Code Fixing
Gupta et al. [80]	Tokens	Code Fix	Distributed	Code Fixing
Hu et al. [94]	Linearized AST	Natural Language	Distributed	Summarization
Iyer et al. [96]	Tokens	Natural Language	Distributed	Summarization
Jiang et al. [97]	Tokens (Diff)	Natural Language	Distributed	Commit Message
Koc et al. [106]	Bytecode	False Positives	Distributed	Program Analysis
Kremenek et al. [108]	Partial PDG	Ownership	Factor (GM)	Pointer Ownership
Levy and Wolf [114]	Statements	Alignment	Distributed	Decompiling
Li et al. [115]	Memory Heap	Separation Logic	Distributed	Verification
Loyola et al. [124]	Tokens (Diff)	Natural Language	Distributed	Explain code changes
Maddison and Tarlow [126]*	LM AST Context	LM (AST)	Distributed	—
Mangal et al. [127]	Logic + Feedback	Prob. Analysis	MaxSAT	Program Analysis
Movshovitz-Attias and Cohen [133]	Tokens	Code Comments	Directed GM	Comment Prediction
Mou et al. [132]	Syntax	Classification	Distributed	Task Classification
Nguyen et al. [142]	API Calls	API Calls	Distributed	Migration
Omar [148]	Syntactic Context	Expressions	Directed GM	Code Completion
Oh et al. [147]	Features	Analysis Params	Static Analysis	Program Analysis
Piech et al. [155]	Syntax + State	Student Feedback	Distributed	Student Feedback
Pradel and Sen [157]	Syntax	Bug Detection	Distributed	Program Analysis
Proksch et al. [159]	Inc. Object Usage	Object Usage	Directed GM	Code Completion
Rabinovich et al. [162]*	LM AST Context	LM (AST)	Distributed	Code Synthesis
Raychev et al. [165]	Dependency Net	Types + Names	CRF (GM)	Types + Names
Wang et al. [183]	Tokens	Defects	LM (n -gram)	Bug Detection
White et al. [188]*	Tokens	LM (Tokens)	Distributed	—
White et al. [187]*	Token + AST	—	Distributed	Clone Detection
Zaremba and Sutskever [197]	Characters	Execution Trace	Distributed	—

Pattern mining models

- ▶ Pattern mining models aim to discover a finite set of human-interpretable patterns from source code, without annotation or supervision
- ▶ Probabilistic pattern mining models of code infer the likely latent structure of a probability distribution

Pattern mining models

- ▶ Examples of pattern mining models: frequent pattern mining – it is not a probabilistic model but counting based
- ▶ Probabilistic model can find interesting models, sometimes frequent occurred patterns may not be interesting
- ▶ Most probably grouping of API elements
- ▶ TSGs (tree substitution grammar) learn to group commonly co-appearing grammar production (tree fragments) – finding "idioms" in code

Existing work on Pattern Mining Models

Reference	Code Representation (c)	Representation (g)	Application
Allamanis and Sutton [12]*	Syntax	Graphical Model	Idiom Mining
Allamanis et al. [4]	Abstracted AST	Graphical Model	Semantic Idiom Mining
Fowkes and Sutton [63]	API Call Sequences	Graphical Model	API Mining
Murali et al. [135]	Sketch Synthesis	Graphical Model	Sketch Mining
Murali et al. [136]	API Usage Errors	Graphical Model	Defect Prediction
Movshovitz-Attias and Cohen [134]	Tokens	Graphical Model	Knowledge-Base Mining
Nguyen et al. [143]	API Usage	Distributed	API Mining
Fowkes et al. [62]	Tokens	Graphical Model	Code Summarization
Wang et al. [184]	Serialized ASTs	Distributed	Defect Prediction
White et al. [187]*	Token & Syntax	Distributed	Clone Detection

Summary

- ▶ Program analysis: get information from code, represent that information
- ▶ Big code analysis: how this information and representation is used to modify, instantiate machine learning models/probabilistic models
 - ▶ generative models (language models, bimodal, transducer model dependent on information available)
 - ▶ representation models: CRF (graphical model inference), distribution vector(encoding used by deep learning)
 - ▶ pattern mining models: clustering algorithms
- ▶ Applications:
 - ▶ predicting developers intent: recommender system
 - ▶ predicting anomaly code
 - ▶ code translation
 - ▶ clone and idiom mining

Further Reading

- ▶ Predicting program properties from "big code"
- ▶ code2vec: Learning Distributed Representations of Code
- ▶ Deep Code
- ▶ Machine Learning for Programming
- ▶ A Survey of Machine Learning for Big Code and Naturalness (2018. May)
- ▶ Machine learning on source code
- ▶ statistics, information theory, machine learning