A Survey of Machine Learning for Big Code and Naturalness

by M.Allamanis et al.

Daniel Perez

April 2, 2018

Overview

2017 survey paper covering machine learning applications to programming

1. Probabilistic models for code
2. Applications
3. Future directions

The naturalness hypothesis

> *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools*

Software and natural languages are both written by humans
$\rightarrow$ they share similar properties
$\rightarrow$ we can use these properties to model software

Natural and programming language differences (1/2)

Code has two channels: human and computer

Executability: code is executable. small changes can change meaning a lot. natural languages are more robust to change

Formality: Programming languages are formal languages. Top-down design vs bottom-up design for natural languages. Programming languages are unambiguous.

Natural and programming language differences (2/2)

Other differences

- Words carry more information than identifiers
- No clear relation between code units and textual units
- Functions are much longer than sentences
- Abstract syntax trees are much deeper than text parse trees
- Source code has more neologism: 70% of tokens are identifiers

Types of models

Models can be classified in 3 categories.
Some models enter multiple categories

Code-generating models: Probability distribution over smaller parts of code, e.g. token or AST nodes
NLP equivalent: generative models

Representational models: Conditional probability distribution over code element properties, e.g. variable types. Usually application specific
NLP equivalent: named entity recognition, text classification

Pattern mining models: Model of latent structure within code
NLP equivalent: topic model

Code-generating Probabilistic models

Describe *how* code is written.

Given training set $\mathcal{D}$, learn distribution $P_{\mathcal{D}}(\mathbb{c}|C(\mathbb{c}))$ to generate code

- If $C(\mathbb{c}) = \varnothing$, $P_{\mathcal{D}}$ is a language model
  e.g. assign probability to a token
- If $C(\mathbb{c})$ is non-code, $P_{\mathcal{D}}$ is a code-generative multimodal model
  e.g. correlate code and comments or specifications
- If $C(\mathbb{c})$ is code, $P_{\mathcal{D}}$ is a transducer model
  e.g. translate a code into another language

Sequence-based models

View code as a sequence $\mathbb{c} = t_1 \dots t_M$, model a distribution

$$P(t_m | t_1 \dots t_{m-1}, C(\mathbb{c}))$$

Simplest model is n-gram, where

$$P(t_1 \dots t_M | C(\mathbb{c})) = \prod_{m=1}^{M} P(t_m | t_{m-1} \dots t_{m-n+1}, C(\mathbb{c}))$$

Approximate probability by counting the number of times t_m occurs after $t_{m-1} \dots t_{m-n+1}$

$P(t_m | t_{m-1} \dots t_{m-n+1})$ is not always known, so use of smoothing techniques is common

Sequence-based models with RNN

Same objective of modeling a distribution

$$P(t_m | t_1 \dots t_{m-1}, C(\mathbb{C}))$$

Usually use an LSTM trained to predict the next token

Pros

- Capture longer range dependencies
- Can encode similarities better, e.g.

```
for(int i = 0; i <= N; i++) vs  for(int j = 0; j <= N; j++)
```

Cons

- Computationally expensive
- Harder to interpret

Models using other representations

Syntactic Models
Code-generating model working with AST instead of sequence

- Usually start from root node and generate using DFS or BFS
- No consensus for modeling tree using more complex structure
- PCFG does not seem to work well (??)

Semantic language models
Code-generating model using graph (e.g. CFG)

- Not well studied in literature
- Some tries have been made to suggest API completions

Representational models

Model to predict properties of source code

- Application-specific model
- Can use representations of generative models or not

Given a property π and a function f to transform code $\mathbb{c}$, goal is to learn

$$P_{\mathcal{D}}\left(\pi | f(\mathbb{c})\right)$$

Distributed representations

Given code $\mathbb{c}$, learn

$$m : \mathbb{c} \to \mathbb{R}^d$$

usually at the token level. Mainly useful for name prediction or as feature to use in other algorithms. Usually implemented at the token level.

Can be extended to learn sequences, i.e.

$$m : \mathbb{c}^n \to \mathbb{R}^d$$

Has been successfully used to classify code and as a context of a generative model to synthesize code.

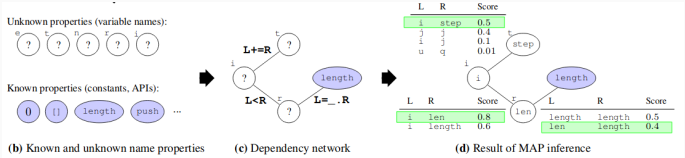
Structured prediction

Joint prediction on a set of variables

- Usually uses a graph representation
- Conditional random field (CRF) seems to be a promising model

Has been successfully used to predict variable names and for auto-completion

```
function (e, t) {
  var r = e.length;
  var i = 0;
  for (; i < r; i += t) {
    ...
  }
}
```



(b) Known and unknown name properties (c) Dependency network (d) Result of MAP inference

Figure 1: Name inference with CRF

Recommender systems

Main task is code completion

- Has been studied quite extensively
- Usually uses language models (n-gram)
- Many different evaluation methods: Accuracy for rank, mean reciprocal rank, Cross entropy, keystrokes saved

Coding conventions

Coding conventions can be divided in naming convention and formatting

Naming convention: variable/class names

Often seems to end up doing something very close to code completion

Formatting: where to put spaces, brackets, etc

Most approaches learn from a predefined set of rules

Code defects

Often anomaly detection applied to programming languages

- Often uses relatively simple models (e.g. n-grams)
- Works well for certain classes of defects (e.g. copy-paste bug finding)
- Other work focus on bug isolation using traces
- Main problem is that most model cannot differentiate between rare and incorrect code

Code Migration, Porting and Clones

Code translation

- Only works for very similar languages (e.g. Java and C#)
- Even Java to C is currently impossible because of memory allocation

Code migration
Seems not to have been studied much.

Some tries have been made to enable “smart” copy-pasting

Code to Text Translation and Traceability

Main tasks

- Generate text from code (e.g. auto documentation)
- Allow to search code using text

Current state

- Generating pseudo-code from Python works well
- Simple queries work well (e.g. how to get first digit of int?)
- Documentation generation seems to only handle trivial cases

Program Synthesis

Mainly divided into two types of research

- Programming by example
- Generating code from specification

- Programming by example works well for simple tasks (e.g. dedup, count, split)
- Generating code from text is still brittle
  - Language generation approach does not work
  - Generating using a production rules seems to kind of work?

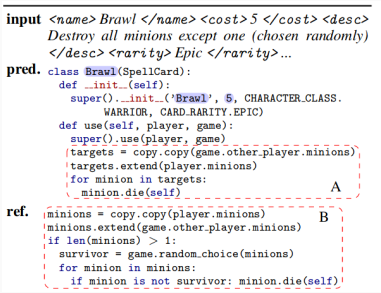


Figure 2: Code generation

Program Analysis

Main tasks

- Use ML to perform program analysis
- Use ML to check program analysis results

Current state

- ML based program analysis can learn interesting rules
- Checking program analysis results seems difficult, mainly because of lack of dataset
- Some interesting results with user-guided program analysis

Waves of machine Learning

We are currently at wave two

- First wave: hand-extracted features with existing ML methods
- Second wave: features auto extracted from source code
- Third wave: use of semantics in machine learning models?

We still lack ML models capable of including semantics. We at least need to

- Represent neologism
- Capture relation between objects
- Express compositionality
- Find good metrics

New domains

Possible research tasks

Debugging

- Finding root cause of bugs
- Reconstruct complete traces from partial traces

Traceability

- Link code to specifications
- Link bugs reports to code