Probabilistic Model for Code with Decision Trees

Veselin Raychev, Pavol Bielik, Martin Vechev
Software Reliability Lab
Department of Computer Science
ETH Zurich

Big Data Revolution

Research area	Big Data	Application
Computer vision	IMAGENET COCO Common Objects in Context	Image labeling  A group of people shopping at an outdoor market. There are many vegetables at the fruit stand.
Programming languages	GitHub Bitbucket GitLab	?

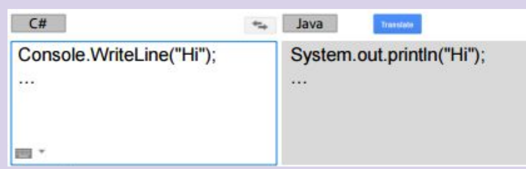
Statistical programming tools

SRL
SOFTWARE RELIABILITY LAB

Write new code [PLDI'14, POPL'16]:
Code Completion

```
Camera camera = Camera.open();
camera.SetDisplayOrientation(90);
                          ?
```

Port code [ONWARD'14]:
Programming Language Translation



Understand code/security [POPL'15]:
Deobfuscation
Type Prediction

JS NICE

www.jsnice.org

Other groups

- Natural language to code
- Automatic patch generation
- Bug finding
- Code completion
- others...

All of these benefit from a good probabilistic model for code.

Probabilistic model for code

Model is a key part of the Statistical Programming Tools

Goal: score programs

Select best among several candidates

Example: Which function is more likely?

(a)

```
function area(a) {
  return a.width * a.height
}
```

(b)

```
function area(a) {
  return a.width * a.close()
}
```

Statistical code completion

Model is a key part of Statistical Programming Tools

Goal: score programs

Select best among several candidates

Example:

```
function area(a) {
  return a.width * a.
}
```

height	Very likely
width	Less likely
open	impossible
close	impossible

Model usability

Directly applicable to code completion, but is a **key statistical component** for many others tasks: e.g. natural language to code, statistical bug localization

Existing works: naive models

Most common model: n-gram

Training (3-gram model):

3-gram

```
return a . width * a .
  + a . open (mode) ;
+ a . close () ;
      * a . height ;
  * a . close () ;
```

Count 3-grams	e.g `a.close` (2 times)

Prediction:

```
width + a .
```

a.close - 2 times
a.open - 1 times
a.width - 1 times
a.height - 1 times

	Probability
close	0.4
open	0.2
width	0.2
height	0.2

Existing works: naive models

Most common model: n-gram

Training (3-gram model):

Prediction:

Main problem:

Bad context leads to bad probability estimates

Count 3-grams | e.g `a.close` (2 times)

width + a .

Probability

`a.close` - 2 times

`a.open` - 1 times

`a.width` - 1 times

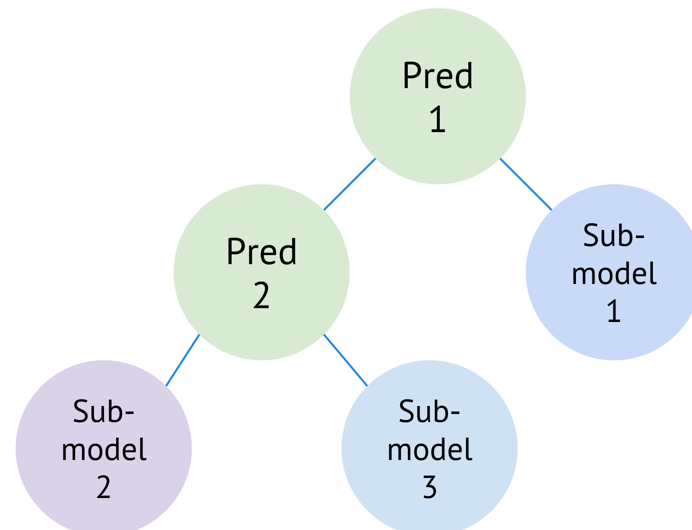
`a.height` - 1 times

close	0.4
open	0.2
width	0.2
height	0.2

1. Richer conditioning context

```
return a.width * a. ?
```

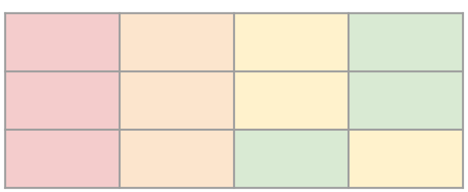
2. Domain Specific Language encoding Decision Trees



Lang ::= BasicPart | BranchPart

BranchPart ::= if pred(x) then Lang else Lang

3. Evaluation



Conditioning context

Training (3-gram model):

```
return x.width * x.height

return y.width * y.height

area = s.width * s.height

s.width = s.width + 10

q.depth * q.width * q.height
```

Prediction:

```
return a.width * a. ?
```

Context: relevant for this prediction

height	0.8
width	0.2

width	→	height	4 times
width	→	width	1 time
width	→	close	0 times

Indirection

Query: x

```
return a.width * a. ?
```

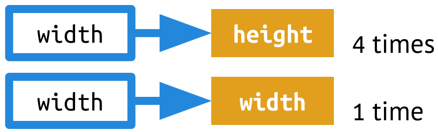
Completion: y

? = height

Context: ctx = f(x)

```
return a.width * a. ?
```

Model: P(y|f(x))



Example contexts

Query: x

```
return a.width * a. ?
```

ctx = f(x): last two tokens
3-gram language model

```
return a.width * a. ?
```

ctx = f(x): previous actions
on the same object [PLDI'14]

```
return a.width * a. ?
```

ctx = f(x): the third token
before the completion

Question:

Which context is the best?

Try many of them and evaluate

Evaluation metric: entropy

Average entropy:
~0.66667 bits

`console.` `info`	P(`info` | `console.`) = 1	0
`console.` `info`	P(`info` | `console.`) = 1	0
`a.width * a. height`	P(`height` | `a.`) = 0.5	1
`b.width * b. height`	P(`height` | `b.`) = 0.5	1
`a.right - a. left`	P(`left` | `a.`) = 0.5	1
`b.bottom - b. top`	P(`top` | `b.`) = 0.5	1

Evaluation metric: entropy

Average entropy:
~1.91 bits

`console.info`	P(`info` | `.`) = 1/3	~1.58
`console.info`	P(`info` | `.`) = 1/3	~1.58
`a.width * a.height`	P(`height` | `.`) = 1/3	~1.58
`b.width * b.height`	P(`height` | `.`) = 1/3	~1.58
`a.right - a.left`	P(`left` | `.`) = 1/6	~2.58
`b.bottom - b.top`	P(`top` | `.`) = 1/6	~2.58

Unconditional model

No conditioning in the probability distribution:
ctx=⊥

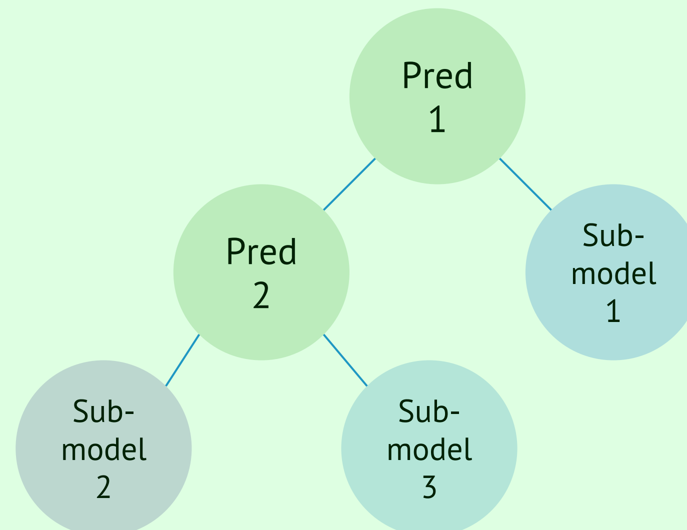
Average entropy:
~1.91 bits

`console.` $\boxed{\texttt{info}}$	P(`info`) = 1/3	~1.58
`console.` $\boxed{\texttt{info}}$	P(`info`) = 1/3	~1.58
`a.width * a.` $\boxed{\texttt{height}}$	P(`height`) = 1/3	~1.58
`b.width * b.` $\boxed{\texttt{height}}$	P(`height`) = 1/3	~1.58
`a.right - a.` $\boxed{\texttt{left}}$	P(`left`) = 1/6	~2.58
`b.bottom - b.` $\boxed{\texttt{top}}$	P(`top`) = 1/6	~2.58

1. Richer conditioning context

```
return a.width * a.    ?
```

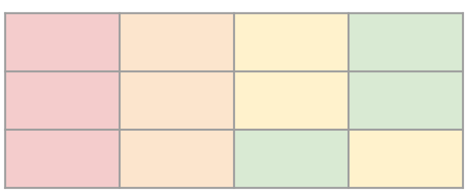
2. Domain Specific Language encoding Decision Trees



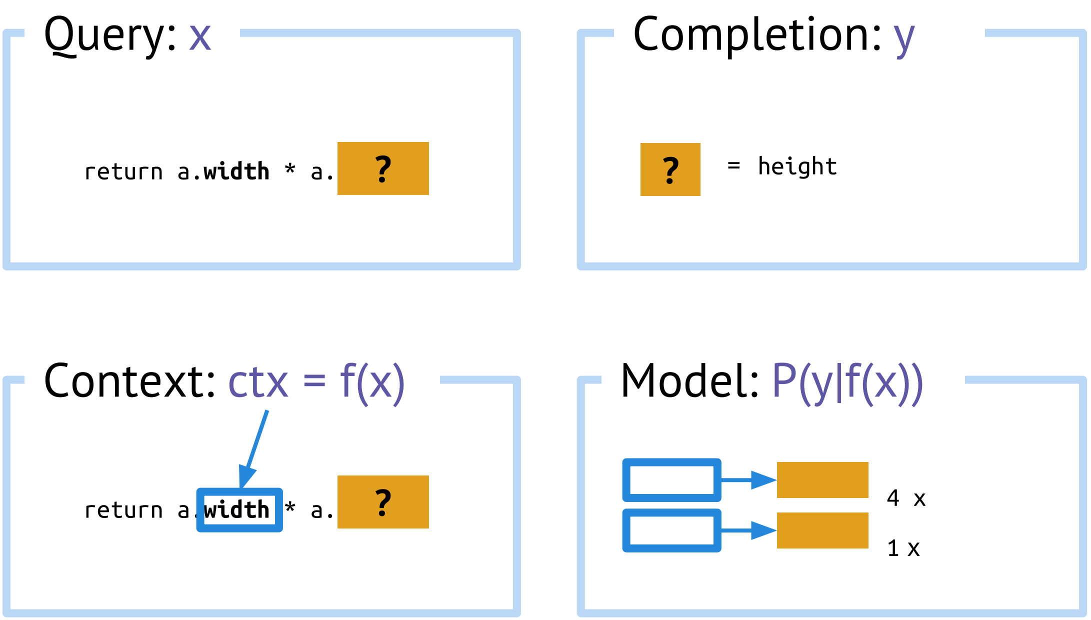
Lang ::= BasicPart | BranchPart

BranchPart ::= if pred(x) then Lang else Lang

3. Evaluation



Synthesize the best model



Find function f

From a domain specific language

Same basic idea in
Learning Programs from Noisy Data [POPL'16]

Main DSL requirement

Program ::= BasicPart | **BranchPart**

BranchPart ::= if pred(x) then **Program** else **Program**

Basic part

Includes models from a simple DSL

```
return a.width * a. ?
```

ctx = f(x): last two tokens
3-gram language model

```
return a.width * a. ?
```

ctx = f(x): previous actions on the same object [PLDI'14]

```
return a.width * a. ?
```

ctx = f(x): empty
Unconditional model

Branch part

Combines programs based on predicates. Language of predicates pred

e.g. are we completing a field name?
Is there another operation on the same object?

Special empty program ε

Best program

If there is no prior field access, then use 3-gram model, otherwise prior field access

Code	Probability	Entropy (bits)
`console.info`	P(`info` | `console.`) = 1	0
`console.info`	P(`info` | `console.`) = 1	0
`a.width * a. height`	P(`height` | `a.`) = 1	0
`b.width * b. height`	P(`height` | `b.`) = 1	0
`a.right - a. left`	P(`left` | `a.`) = 1	0
`b.bottom - b. top`	P(`top` | `b.`) = 1	0

Synthesis using basic part of DSL

Involved
See POPL'16

At high level:
Search through **thousands** of candidate programs that describe conditioning.

Basic part

Includes models from a simple DSL

```
return a.width * a. ?
```

ctx = f(x): last two tokens
3-gram language model

```
return a.width * a. ?
```

ctx = f(x): previous actions on the same object [PLDI'14]

```
return a.width * a. ?
```

ctx = f(x): empty
Unconditional model

Synthesis of branch part

Space of possible programs:

BranchPart ::= if pred(x) then **Program** else **Program**

Search thousands

Search thousands

Search thousands

Billions even without nesting

Infinite with nesting

Goal: find program with best entropy

Idea 1: synthesis in parts

1. Synthesize branch with empty programs in leaves

Synthesize : if pred(x) then ε else ε

Search thousands

Until no predicate improves entropy over basic ε

Then synthesize basic part

Goal: find program with **approximately** best entropy

2. Synthesize leaves separately

Synthesize : if pred(x) then **Program** else **Program**

Recursively call 1 for branch

Recursively call 1 for branch

This is decision tree learning

Synthesis of branch part

Space of possible programs:

BranchPart ::= if pred(x) then **Program** else **Program**

Search thousands

Search thousands

Search thousands

Search for each component separately

Performance at expense of possible non-optimality

Goal: find program with **approximately** best entropy

Idea 2: synthesis in parts

1. Synthesize basic program

f' ∊ BasicPart

Search thousands

Goal: find program with **approximately** best entropy

2. Synthesize branch

Synthesize : if pred(x) then **f'** else **f'**

Search thousands

Until no predicate improves entropy over simply **f'**

If no, return **f'**

3. Synthesize leaves separately

Synthesize : if pred(x) then **Program** else **Program**

Recursively call 1 for branch

Recursively call 1 for branch

Main result

Synthesis procedure 1 is a new formulation of a known and popular algorithm for decision tree learning: ID3

In fact, we extended ID3 to support programs in decision tree leaves from the BasicPart DSL fragment

Synthesis procedure 2 is new
And also applicable to decision trees

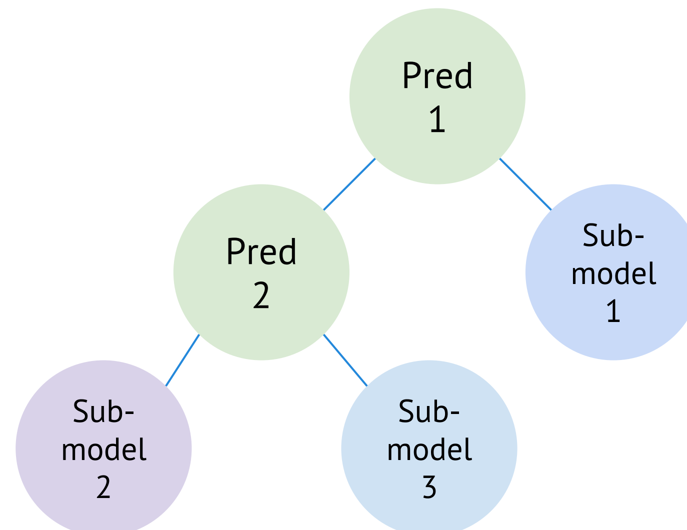
New name:

E13

1. Richer conditioning context

```
return a.width * a. ?
```

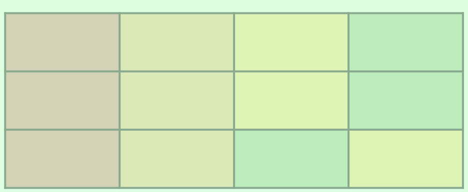
2. Domain Specific Language encoding Decision Trees



Lang ::= BasicPart | BranchPart

BranchPart ::= if pred(x) then Lang else Lang

3. Evaluation



Evaluation

150’000 JavaScript files, from GitHub.com, parsed into ASTs. Public datasets.

100’000 files
Training data for synthesis

Synthesis time: ~100 hours

50’000 files
Evaluation data

Evaluation files are not on the same projects as training

Question: How well can we predict program elements?

Accuracy: JavaScript

Query time for all models is basically the same
(>10K queries per second)

Task	**PCFG**	**3-gram**	**ID3+**	**E13**
API completion	0.04%	30.0%	54%	**66.6%**
Field access completion	3.2%	32.9%	52.5%	**67.0%**
Predicting loops	0%	37.5%	**65.0%**	28.3%

Many more evaluation results in the paper.... Also for Python. Easily applicable to all languages.

Big improvement over prior methods

Both ID3+ and E13 learn useful probabilistic models