Deep Learning over “Big Code” for Program Analysis and Synthesis

Swarat Chaudhuri, Vijay Murali, Chris Jermaine
Programming is hard

Program synthesis, debugging, verification, repair…

Can we automate these processes?
Decades of prior work

• Synthesis
  • [Pnueli & Rosner 1989]: temporal constraints
  • [Solar-Lezama 2008, Alur 2013]: partially written program (sketch)
  • [Gulwani 2011]: input-output examples

• Debugging
  • [Weiser 1981, Korel-Laski 1988]: slicing criterion
  • [Ball-Rajamani 2002, Godefroid 2005]: model checking property

• Most prior works require formal specifications!
Specifications

• Practical tasks
  • Reading/writing an XML document
  • Displaying an Android dialog box
  • Connecting to an SQL server
  … How to specify formally? …

• Bayou: a statistical approach that lets us break out of the reliance on formal specifications
  • Built to handle “uncertain” specifications
  • Applicable to various problems in formal methods
  • In this tutorial: program synthesis, debugging (bug-finding)
“Big Code”

• Online code corpora offer great opportunities for specification learning

• Especially useful for learning about broadly shared facets of programs

• Advances in machine learning (ML) can be leveraged
  • “Deep learning”
  • But not enough by itself…”

19.4 million active projects on GitHub (Oct. 2016)
Synergy of ML & FM

• ML has been highly successful in learning patterns from text, images, audio, etc.
  • We are dealing with programs – semantics is key!
  • Throwing ML at “big code” is not sufficient

• Bayou = Machine Learning Formal Methods

Good at handling uncertainty

Machine Learning

Bayou

Formal Methods

Good at handling semantics
Related Work

• “Big Code” is a very active area of research
  • [Raychev et al. 2014, Raychev et al. 2015]: data-driven code completion, analysis
  • [Gu et al. 2016]: predicting API sequences from natural language
  • [Yaghmazadeh et al. 2017]: SQL query synthesis
  • [Balog et al. 2017, Parisotto et al. 2017]: faster IO-example-based synthesis

• How is Bayou different?
  • Generic probabilistic framework
  • Interaction with formal methods (e.g., sketch learning)
  • Real general-purpose programming language
  • Deep models
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Outline

• **Introduction to the Bayou framework**
  • BayouSynth
    • Underlying probabilistic model
    • BayouDebug

• **Implementing BayouSynth with deep neural networks**
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• **Implementing BayouDebug**
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Program Synthesis

“Find a program that fits a specification.”

Many kinds of (formal) specifications: Input-output examples, traces, constraints, types...

Flash-fill (Microsoft Excel) can synthesize macros from a few examples [Gulwani 2011]

“One of the shock-and-awe features of Excel 2013.”
— Ars Technica
Program Synthesis

• What about the typical programmer?
  • Read from a JSON/XML document
  • Connect to an Android Bluetooth socket
  • Query an SQL database
  • Displaying a dialog box in UI
  • ...

BayouSynth
1. Java: works with a general purpose PL
2. APIs: synthesizes code involving APIs which are needed for most common tasks
3. Uncertainty: no need of a full formal specification
What do human programmers have that synthesizers don’t?
1. Ability to handle uncertainty

In formal methods and synthesis, specification is a Boolean property. A solution is valid iff it satisfies this property.

- Formal specifications are too costly
- Underspecifications can lead to meaningless output.

Humans can generalize imprecise and incomplete specifications.
2. Much better search heuristics

The space of programs of size $n$ grows exponentially in $n$. This is a fundamental bottleneck for program synthesis.

Current solutions assume simple syntactic program model,

- either a detailed program sketch as part of the problem,
- or a narrow domain-specific language.

Humans can zoom in on the relevant parts of this search space.
Lesson from other areas of AI: Use data!
Human programmers use data

OK, so I need to open this text file, parse it, and...

- Textbooks, documentation
- Forums, chats
- Other people’s code
- Personal experience

Programmers use this data to build mental models of *how to design programs*.

This model lets them interpret programmer intent and “guess” the structure of solutions.
Probabilistic models let us mimic this process inside a synthesizer.
Data-driven program synthesis

An idealized program

Evidence about what the program does

Candidate implementations, based on posterior

Prior distribution over syntax of programs and their associated evidence

Learned from data

Synthesizer

Posterior distribution over program syntax
BayouSynth

• Data-driven synthesis of API usage idioms
• Web demo: www.askbayou.com
• Source: github.com/capergroup/bayou

Neural Sketch Learning for Conditional Program Generation.
https://arxiv.org/abs/1703.05698
Outline

• **Introduction to the Bayou framework**
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
1. Programs

- The source code of a program

- General purpose programming language
  - Imperative
  - Rich control structure (loops and branches)
  - Exception handling
  - API method calls
  - ...

- We have a large amount of data on
1. Programs

Language capturing essence of API usage in Java

\[
\text{Prog} ::= \text{skip} \mid \text{Prog}_1; \text{Prog}_2 \mid \text{call} \ \text{Call} \mid \\
\text{let } x = \text{Call} \mid \\
\text{if Exp then Prog}_1 \text{ else Prog}_2 \mid \\
\text{while Exp do Prog}_1 \mid \text{try Prog}_1 \ \text{Catch}
\]

\[
\text{Exp} ::= \text{Sexp} \mid \text{Call} \mid \text{let } x = \text{Call} : \text{Exp}_1
\]

\[
\text{Sexp} ::= c \mid x
\]

\[
\text{Call} ::= \text{Sexp}_0.a(\text{Sexp}_1, \ldots, \text{Sexp}_k)
\]

\[
\text{Catch} ::= \text{catch}(x_1) \ \text{Prog}_1 \ldots \text{catch}(x_k) \ \text{Prog}_k
\]
2. Evidence

- $X$: Evidence about the intended programming task
  - Names of API methods
  - Types
  - Keywords (Natural language)
  - Behaviors / execution traces
  - Coming up: shape of code, pictures 😊

- We have a large amount of data on
2. Evidence

API Calls: Set of the names of API methods called.
   “readLine”, “close”

Types: Set of types on which API methods are called.
   “FileReader”

Keywords: Textual description of programming task.
   “read from file”, “print list”
Problem Statement

• Conditional Program Generation
  • Assume $x$ and $y$ follow an unknown joint distribution

• Offline:
  • Given a dataset of samples from $x$, learn a function that maps evidence to programs.
  • Learning goal: maximize $\log p(x)$, where

• Online:
  • Given $x$, produce $y$
Problem Statement

• What we actually do
  • The map is probabilistic, i.e.,

• Learn through **maximum conditional likelihood**
  • We have data of the form pairs
  • Assume distribution parameterized on some
  • Find an optimal value that maximizes the (log) likelihood

• With optimal parameters, sample from learned distribution given, i.e.,
Challenge in Big Code setting

```java
void read() throws IOException {
    FileReader in = new FileReader("a.txt");
    BufferedReader br = new BufferedReader(in);
    String line;
    while ((line=br.readLine())!=null)
    {
        System.out.println(line);
    }
    br.close();
}
```

```java
int read(String name) {
    FileReader fr; BufferedReader r;
    String s; int n = 0;
    try {
        fr = new FileReader(name);
        r = new BufferedReader(fr);
        for (; (s=r.readLine()) != null; n++);
        r.close();
        return n;
    } catch (IOException e) { return -1; }
}
```

Both programs perform the task “reading from a file”
Data inherently contains noise

1. From superficial differences irrelevant for synthesis
   • Variable names
   • Intermediate expressions
   • Syntactic forms (for loop vs. while loop)

   **Superficial differences in programs make it hard for probabilistic model to learn patterns**

2. From knowledge already known
   • Type safety constraints
   • Language-level rules (e.g., exceptions must be caught)

   **Probabilistic model learned from data cannot guarantee type safety constraints and rules**
Key insight...

- **Probabilistic models** are adept at learning unknown patterns from data
- **Synthesizers** are adept at handling known semantic and syntactic constraints

*Learn to generate programs at a higher level of abstraction and use combinatorial synthesizer to produce final code*
3. Sketches

: sketch, a syntactic abstraction of a program
  • Sketches abstract away superficial differences and known knowledge

```
[  
call FileReader.new(String)
call BufferedReader.new(FileReader)
loop ([BufferedReader.readLine()]){
  skip
}
call BufferedReader.close()
]
```
3. Sketches

Program-Sketch relation is many-to-one
- Abstraction function
- Concretization distribution

\[ \alpha(\text{Prog}) \]

\[ P(\text{Prog} \mid Y) \]

is not learned from data
- Fixed and defined heuristically with domain knowledge
3. Sketches

New goal: “Sketch-learning”
- Learn to generate sketches from evidence

Learn distribution
- Data is now triplets
- parameterizes the distribution
- Find an optimal value
3. Sketches

Two-step synthesis

1. Sample sketch from learned distribution, i.e.,

2. Synthesize from
   • Implemented in a combinatorial synthesizer
   • Uses type-directed search to prune space
   • Incorporates the PL grammar, language-level rules, type-safety constraints, ...
3. Sketches

Sketches can be defined in many ways. But one has to be careful…

- Too concrete: patterns in training data would get lost, would suffer

- Too abstract: concretizing sketches to code would get too hard to compute, would suffer

Our sketch language designed for API-using Java programs
3. Sketches

\( Y ::= \text{skip} \mid \text{call} \text{ Cexp} \mid Y_1 ; Y_2 \mid \)

\( \text{if Cseq then } Y_1 \text{ else } Y_2 \mid \)

\( \text{while Cseq do } Y_1 \mid \text{try } Y_1 \text{ Catch} \)

\( \text{Cexp ::= } \tau_0 . a(\tau_1, \ldots, \tau_k) \)

\( \text{Cseq ::= List of Cexp} \)

\( \text{Catch ::= } \text{catch}(\tau_1) Y_1 \ldots \text{catch}(\tau_k) Y_k \)
Training Corpus of Programs

Feature extractor
- Evidences
- Sketches

Draft Program with Evidences
foo(File f) {
    // read file
}

Distribution over Evidences & Sketches

Synthesized Program
foo(File f) {
    f.read();
    f.close();
}

Combinatorial Search
- Type-based pruning

Statistical Learning (Deep Neural Network)

$P(Y \mid X)$
Outline

• **Introduction to the Bayou framework**
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• **Implementing BayouSynth with deep neural networks**
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• **Implementing BayouDebug**
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Data-driven Correctness Analysis

Underlying thesis: Bugs are anomalous behaviors. [Engler et al., 2002; Hangal & Lam, 2002]

A specification is a commonplace pattern in program behaviors seen in the real world.

Learn specifications from examples of program behavior. [Ammons et al., 2002; Raychev et al., 2014]
BayouDebug

• Statistical framework for simultaneously learning a wide range of specifications from a large, heterogeneous corpus

• Quantitatively estimating a program’s “anomalousness” as a measure of its correctness

• BayouDebug: a system for finding API usage errors in Java/Android code
  • Underlying probabilistic model similar to BayouSynth but “mirrored”
  • Program is given, need to predict likelihood of its behaviors
BayouDebug

• Originally called *Salento*
• Source: [github.com/capergroup/salento](https://github.com/capergroup/salento)

Bayesian Specification Learning for Finding API Usage Errors
Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine.
Foundations of Software Engineering [FSE] 2017
AlertDialog.Builder b =
    new AlertDialog.Builder(this);
b.setTitle(R.string.title_variable_to_insert);
if (focus.getId() == R.id.tmpl_item) {
    b.setItems(R.array.templatebodyvars,
                this);
}
else if (focus.getId() == R.id.tmpl_footer) {
    b.setItems(R.array.templateheaderfootervars,
                this);
}
b.show();

This dialog box cannot be closed
1. Evidence

• We have programs and

• Evidence as before
  • Set of API calls in program
  • Set of types in program
  • ...

• Can be easily extracted from programs
2. Behaviors

• represents behaviors
  • Traces of API calls
  • Program state during execution (abstraction)

• Can also be extracted from programs
  • Dynamic/Symbolic Execution
  • Assuming a behavior model for a program
  • Behavior model derived from input distribution (dynamic) or static analysis (symbolic)
Generative probabilistic automaton
[Murawski & Ouaknine, 2005]

AlertDialog.Builder b =
    new AlertDialog.Builder(...);
b.setTitle(...);
if (...) {
    b.setItems (...);
} else if (...) {
    b.setItems(...);
}
else if (...) {
    b.setItems(...);
}
b.show();

Produced using static analysis
Specification Learning

From data, learn a distribution over program behaviors given evidence, i.e.,

- Data is in the form of pairs
- As before, assume parameterizes the distribution
- Find an optimal value using max-CLE
Correctness Analysis

- **Goal**: check if test program is correct
- Look at two distributions
  - how programs that look like tend to behave
  - how behaves

- Cast correctness analysis as statistical distance computation
  - Kullback-Leibler (KL) divergence:
    - High KL-divergence $\Rightarrow$ Prog is anomalous
Training Corpus of Programs

Features & Behaviors from corpus

Inference

Feature extractor
- API calls
- Behaviors

Features & Behaviors of test program

Test Program F

foo(File f)
{
    f.read();
    f.close();
}

Distribution over Features & Behaviors

Statistical Learning (Deep Neural Network)

P(Y | Prog)

Anomaly Score (Aggregate)

P(Y | X)
What we have covered

• Formal methods have always relied on formal specifications

• **Uncertainty** in specifications is an important consideration

• ML models learned from Big Code are a new and hot way of dealing with uncertainty

• PL ideas are still key
  • Syntactic abstractions are necessary for data-driven synthesis
  • Static/Dynamic analysis is necessary for data-driven debugging

• How to implement all of this? Coming up next…
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
What is a Neural Network?

- A logical circuit transforms binary input signals into binary outputs through logical operations.

- A neural network is a circuit where
  - Input and outputs can be smooth (continuous)
  - Operations are differentiable (matrix multiply, exponentiate, …)

\[
\text{output} = \text{softmax}(W x_1 + \ldots)
\]
Code Snippets

• Most common machine learning libraries

• We will use TensorFlow in this talk
  • Build a computation graph of neural network in Python
  • Statically compile graph into C++/CUDA
  • Setup training data for each input/output variable
  • Execute graph with data
  • [Abadi et al. 2016]

```python
import tensorflow as tf
```
Encodings

• Neural networks work on various kinds of inputs and outputs
  • Differentiable operations work on real numbers
  • Transform raw inputs into a suitable representation

I am a student  ➔  Je suis un étudiant

5 0 4 1  ➔  5 0 4 1

• Fixed vocabulary: , encode words uniquely
  • Naïve encoding – each word is its index (, , …)
  • Problem?

• One-Hot encoding: typical encoding for categorical data
One-Hot Encoding

- One-hot encoding of a word is an $n$-length vector where all elements are 0 except a 1 at index $i$.

### Word | One-hot encoding
--- | ---
| [ 1, 0, 0, ... 0 ] |
| [ 0, 1, 0, ... 0 ] |
| [ 0, 0, 0, ... 1 ] |

- **Pros/Cons**
  + Easy to encode, no unintended relationships between words
  - Length of encoding affected by vocabulary size, infrequent words

- All input evidences are assumed to have been converted to their one-hot representations.
Feed-Forward Neural Network

• A simple architecture of a “cell” (Tensorflow term)
  • Signal flows from input to output
  • Real-valued weight and bias matrices and

where is an “activation function”

\[ y = \sigma(Wx + b) \]
Activation Functions

- **Non-linear functions** that decide the output format of cell
  - Sigmoid, output between 0 and 1:
  - output between -1 and 1
  - Rectified Linear Unit (ReLU), output between 0 and
Implementing a FFNN

\[ y = \sigma(W \cdot x + b) \]

\[ y = \sigma(W \cdot x + b) \]

\[ W \quad b \]

\[ \downarrow \quad \downarrow \]

\[ \rightarrow \quad * \quad \rightarrow \quad + \quad \rightarrow \quad \sigma \quad \rightarrow \]

# input_size: size of input vocabulary
# output_size: size of output as needed

```python
x = tf.placeholder(tf.int32, [1, input_size])
W = tf.get_variable('W', [input_size, output_size])
b = tf.get_variable('b', [output_size])
y = tf.sigmoid(tf.add(tf.multiply(W, x), b))
```
Hidden Layers

• Notion of “internal state” can be implemented through hidden layers

```python
# num_units: number of units in the hidden layer
...
W_h = tf.get_variable('W_h', [input_size, num_units])
b_h = tf.get_variable('b_h', [num_units])
h = tf.sigmoid(tf.add(tf.multiply(W_h, x), b_h))

W = tf.get_variable('W', [num_units, output_size])
b = tf.get_variable('b', [output_size])
y = tf.sigmoid(tf.add(tf.multiply(W, h), b))
```
Stacking hidden layers

• Forms the “deep” in deep learning

\[ W_1 \quad b_1 \quad h_1 \quad W_2 \quad b_2 \quad h_2 \]

\[
\rightarrow \quad \ast \quad \rightarrow \quad + \quad \rightarrow \quad \sigma \quad \rightarrow \quad \ast \quad \rightarrow \quad + \quad \rightarrow \quad \sigma \quad \rightarrow \quad \ldots \quad \rightarrow
\]

• Weights/biases can be shared (all \( W \) and \( b \) are the same)
  • Design choice that leads to different architectures
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
    • The Encoder-Decoder architecture
    • Gaussian Encoder-Decoder
    • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Recurrent Neural Network

- RNNs model *sequences* of things
  - Assume input and output
  - RNNs have a notion of hidden state across “time steps”
  - Feedback loop updates hidden state at each step

$$\mathbf{x}_1 \rightarrow \mathbf{y}_1 \rightarrow \mathbf{x}_2 \rightarrow \mathbf{y}_2 \rightarrow \ldots \rightarrow \mathbf{x}_n$$
Recurrent Neural Network

- Model hidden state at time step as a function of input and hidden state

- Each hidden state encodes entire history (as permissible by memory) due to feedback loop

- Important property: weights for hidden state \((, , )\) are shared across time steps
  - Most often we do not know the number of time steps a priori
  - Shared weights model the same function being applied at each time step
  - Keeps model parameters tractable and mitigates overfitting
Implementing an RNN

- Tensorflow provides an API for RNN cells
  - Configure type of RNN cell (vanilla, LSTM, etc.)
  - Configure activation functions (sigmoid, tanh, etc.)

```python
# input: x = [x_1, x_2, ..., x_n]
# expected output: y_ = [y_1, y_2, ..., y_n]
# num_units: number of units in the hidden layer

rnn = tf.nn.rnn_cell.BasicRNNCell(num_units, activation=tf.sigmoid)
state = tf.zeros([1, rnn.state_size])
y = []
for i in range(len(x)):
    output, new_state = rnn(x[i], state)
    state = new_state
    logits = tf.add(tf.multiply(W_y, output), b_y)
    y.append(logits)
```

62
RNNs for Program Synthesis

• Consider a program as a sequence of tokens from a vocabulary of tokens

```java
void read() throws IOException {
    ...
}
```

• As data is noisy, we typically want to learn a distribution over programs
  • Output programs can be sampled from learned distribution

• For a program where each is a token,
  • Each token is obtained from a history of tokens
  • RNN hidden state is capable of handling history
RNNs for Program Synthesis

• If we train an RNN to learn we can use it to generate code token-by-token

• Synthesis strategy: sample token at time step and provide it back as input for time
  • No evidence: Unconditional program generation
  • No sketches: Learning would be difficult
  • Not optimal, still useful to introduce ML concepts
Output Distributions

• First, we need the RNN output to be a distribution

• **Softmax activation function**
  • Converts a -sized vector of real quantities into a categorical distribution over classes

• Advantages over standard normalization
  • Handles positive and negative values
  • Implies raw values are in log-space, which is common in MLE

```python
for i in range(len(x)):
    output, new_state = rnn(x[i], state)
    state = new_state
    logits = tf.add(tf.multiply(W_y, output), b_y)
    y.append(tf.nn.softmax(logits))
```
Loss Functions

• The RNN we have built would likely not produce expected outputs immediately
  • For training, define what it means for a model to be bad and reduce it

• Loss Functions define how bad a model is with respect to expected outputs in training data
  • Cross-entropy (categorical)
  • Mean-squared error (real-valued)

• Cross-entropy measures the distance between two distributions
  • : ground truth “distribution” (one-hot encoding)
  • : predicted distribution
Loss Functions

- Example: vocabulary size 4
  - Expected output is : , predicted distribution:
  - Cross-entropy loss
- Loss for output sequence is typically the average over sequence
- Tensorflow’s API has softmax and cross-entropy sequence loss built into a single call

```python
# expected output: y_ = [y_1, y_2, ..., y_n]
...
for i in range(len(x)):
    output, new_state = rnn(x[i], state)
    state = new_state
    logits = tf.add(tf.multiply(W_y, output), b_y)
    y.append(logits)

loss = tf.contrib.seq2seq.sequence_loss(y, y_, weights=tf.ones(...))
```
Loss Functions

- Tensorflow adds loss operation to computation graph

Targets

Outputs

Softmax Cross Entropy
Ingredients for Training

**Neural Network**
Complex architecture to model generation of outputs from inputs

**Loss Function**
High-dimensional function measuring error w.r.t. ground truth

**Training Data**
Ground truth inputs and outputs

**Gradient Descent**
Find the point where function value is minimal
Gradient Descent

- Optimization algorithm to compute (local) minimum
  - Iteratively move parameters in the direction of negative gradient
  - Need for differentiable operations

A single “step” of gradient descent

```python
given: function f(x), loss
for each parameter p of function
    p_grad = 0
    for each data point d in training data
        g = gradient of loss w.r.t. p
        for d
            p_grad += g
        p += -p_grad * learning_rate
```
Stochastic Gradient Descent

- **Stochastic Gradient Descent (SGD)** approximates GD
  - Considers only a single data point for each update
  - Takes advantage of redundancy often present in data
  - Requires more parameter updates, but each iteration is faster

- In practice, **mini-batch Gradient Descent**
  - Use a small number of data points (10-100)

A single “step” of gradient descent

```python
given: function f(x), loss
for each parameter p of function
  p_grad = 0
  for each data point d in batch
    g = gradient of loss w.r.t. p
  for d
    p_grad += g
  p += -p_grad * learning_rate
```
Backpropagation

• Reverse-mode automatic differentiation
  • “Magic sauce” of gradient descent & deep learning
  • Automatically compute partial derivatives of every parameter in NN

• During optimization, compute gradients in almost the same order of complexity as evaluating the function
Backpropagation

- Each basic operation is associated with a gradient operation
  - Use chain rule to compute derivative of loss w.r.t. operation
  - Example:
    - Efficient by computing and reusing intermediate partial derivatives
- During SGD, all parameters can be updated in one swoop
  - Learning rate controls amount of update
Backpropagation

A single “step” of gradient descent

given: function $f(x)$, loss
grad = 0
for each data point $d$ in batch
    $g =$ gradient of loss w.r.t. each param
for $d$
    grad += $g$
backprop_gradients(grad)

• For RNNs – Backpropagation Through Time (BPTT)
  • “Indefinite length”, unroll into multi-layer FFNNs and backprop
  • Problem: Due to multiplication, run into either exploding (> 1) or vanishing (< 1) gradients
  • In practice, Truncated BPTT – build RNN with fixed-length and backprop till length
Training in Tensorflow

• Add training operation to loss function
  • Tensorflow automatically adds backpropagation operations
• Create a Tensorflow “session” to initialize variables
• Feed mini-batches for each iteration as dictionary

```python
...
y_ = tf.placeholder(tf.int32, [batch_size, rnn_length], ...)
step = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
with tf.Session() as sess:
  tf.global_variables_initializer().run()
  for epoch in range(50):
    batches = get_mini_batches()
    for (batch_x, batch_y) in batches:
      sess.run(step, feed_dict={x: batch_x, y_: batch_y})
```
Example: Character-level RNN

• Training an RNN on Linux source to generate code character-by-character

• Token level model may be easier or difficult
  + Character vocabulary (ASCII) is simpler than token vocabulary
  - Character model could generate malformed keywords (if, while, etc.) but token model would not

• Nevertheless, interesting model to consider as example

http://karpathy.github.io/2015/05/21/rnn-effectiveness
static void do_command(struct seq_file *m, void *v) {
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)((int_state ^ (in_8(&ch->ch_flags) & Cmd)) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << 1))
            pipe = (in_use & UMXTHREAD_UNCCA) +
            (((count & 0x00000000fffffff8) & 0x000000f) << 8);
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
    • Gaussian Encoder-Decoder
    • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Conditional Generative Model

- RNNs can learn to model generation of sequences of data
  - where Prog is a sequence of tokens/characters
  - For synthesis we need a **conditional** generative model

- Can we **condition** an RNN to generate sequences based on some input?
  - Specifically, can we make an RNN learn?
  - We can then condition the generation of code on evidence

**Encoder-Decoder architecture**

- Often used in Neural Machine Translation (NMT)
  - Google translate
**Encoder-Decoder Architecture**

- **Key insight:** To learn a conditional distribution
  - Use an encoder network to encode $x$ into a hidden state $h$.
  - Use a decoder network to generate $y$ from the encoded state.

$$ h = \sigma(W_h x + b_h) $$
Implementing an Encoder-Decoder

• Simply compute RNN initial state using the output of FFNN

```python
# num_units,_enc,_dec: hidden state/encoder/decoder
dimensionality
...

h_enc = tf.sigmoid(tf.add(tf.multiply(W_enc, x), b_enc))

# transform into hidden state dimensions
W_h = tf.get_variable('W_h', [num_units_enc, num_units])
b_h = tf.get_variable('b_h', [num_units])
h = tf.sigmoid(tf.add(tf.multiply(W_h, h_enc), b_h))

rnn = tf.nn.rnn_cell.BasicRNNCell(num_units_dec, ...)
h_dec = tf.sigmoid(tf.add(tf.multiply(W_dec, h), b_dec))
for i in range(len(y)):
    output, new_h_dec = rnn(y[i], h_dec)
    h_dec = new_h_dec
...```
Encoder-Decoder Characteristics

1. Encoder and decoder must be trained together
   • Gradients from decoder passed all the way back to encoder

2. Low-dimensional hidden state
   • Compared to encoder inputs (one-hot) and decoder outputs (softmax)
Encoder-Decoder Characteristics

• “Bottleneck” due to (1) and (2)
  • Encoder learns to encode inputs in the most efficient way that is useful for decoder
  • Hidden state acts as a regularizer – captures the essence of inputs that is necessary to produce the right outputs
  • Mitigates overfitting

• For the synthesis problem
  • Encoding multiple inputs (evidence)
    • In sequence? Concatenate hidden states? Average?
  • Decoding into trees (sketches)
    • Representing structure using sequence?
    • Inferring the most likely sketch?

*Is there a principled way to do this?*
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Latent Intents

Each programming task has an intent
- Example (abstractly): “file reading”, “sorting”
- There is a distribution over ,
- Since we do not know anything about , it is latent
- Assume a prior

We have evidence about the intent:
- API calls, types, keywords. Example: `readLine`, `swap`

We have implementations of the intent:
- Sketches – abstractions of implementation

Given , and are conditionally independent:
: Intent from Evidence

• How should we define?
  • We can have multiple evidences
  • We want each evidence to independently shift our belief on

• Define a generative model of evidence from intent

  where \( f \) is the encoding function

• Models the assumption that encoded value of each evidence is a sample from a Normal centered on
  • prior
  • with some variance (learned)
from Normal-Normal conjugacy:

\[ f(x_2) \sim N(z_2, \sigma^2 I) \]

\[ z_2 \]

\[ Z \sim N(0, I) \]

\[ z_1 \]

FileReader

readLine
: Intent from Evidence

How the encoder maps evidence to latent space (posterior)
• Sketch is tree-structured, RNNs work with sequences
  • Deconstruct sketch into set of production paths
  • Based on production rules in sketch grammar

• Sequence of pairs where
  • is a node in sketch, i.e., a term in the grammar
  • , the type of edge between and
    • Sibling connects terms in the RHS of the same rule (sequential composition)
    • Child connects a term in the LHS with the RHS of a rule (loop condition with body)
Sketch from Intent

4 paths in sketch

1. `(try, FR.new(String), (FR.new(String),)), (BR.new(FR), ), (while, ),
   (BR.readLine(), ), (skip, )`

2. `(try, ), (catch, ), (FNFException, ), (printStackTrace(), )`
Sketch from Intent

- Generate sketch by recursively generating productions paths

**Basic step:** given and a history of fired rules, what is the distribution over the next rule?
  - where
  - Distribution dependent on history and – not context-free!
  - Sample a and recursively generate tree

- Implemented using an RNN
  - Neural hidden state can encode history
  - Top-down Tree-Structured RNNs (Zhang et al, 2016)
Production rule in sketch grammar

Distribution on rules that can be fired at a point, given history so far.

History encoded as a real vector.
Putting it all together…

• Originally, we were interested in

(from our probabilistic model)

(from the Monte-Carlo definition of expectation)

(from Jensen’s inequality)

(lower bound for CLE)
Putting it all together…

• In English
  • encodes evidence into distribution over
  • A value of is sampled from the distribution
  • decodes into a sketch

Problem?

Gradients cannot pass through stochastic operation!
Reparameterization

• **Key intuition**: all Normal distributions are scaled/balanced versions of
  • Sampling from $\mathcal{N}(\mu, \sigma^2)$ = sampling from $\mathcal{N}(0, 1)$, multiplying by $\sigma$ and adding $\mu$

• Instead of $\mathcal{N}(\mu, \sigma^2)$, get sample and compute
  • Encoder produces $\mu$ and $\log \sigma$ as the parameters of

• $\mu$ is an input to the network, not part of it
  • Gradients can flow through!
  • [Kingma 2014]
Reparameterization

Evidence

Encoder

Intent

Decoder

Sample from

Sketch

Gaussian Encoder Decoder (GED)
What we have covered…

• How to implement neural network architectures
  • Feedforward Neural Network
  • Recurrent Neural Network

• How to build an Encoder-Decoder network for program synthesis
  • GED is suited for synthesis but it is not the only architecture that can be instantiated from the Bayou framework

• How neural networks are trained
  • Gradient descent, backpropagation, reparameterization

• Coming up next
  • How the PL parts interact with the ML parts in BayouSynth and BayouDebug
What we have not covered...

- Multi-modal evidences with different modes
  - API calls, types, keywords, etc. may each have a different variance towards

- Getting a distribution over top-k likely sketches instead of sampling a single sketch
  - Beam search

- Top-Down Tree-Structured LSTM network
  - Architecture for learning tree-structured data

- Handling complex evidences such as Natural Language
  - One-hot encoding would blow up, need a more “dense” embedding
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
Draft Program with Evidences

foo(File f) {
    f.read();
    f.close();
}

Synthesized Program

foo(File f) {
    f.read();
    f.close();
}

Combinatorial Search
• Type-based pruning

Statistical Learning
(Deep Neural Network)

\[ P(Y \mid X) \]

Feature extractor
• Evidences
• Sketches

Evidences & Sketches from corpus

Training

Corpus of Programs

foo(File f) {
    // read file
}
The Problem

• Synthesize program from sketch using

• What do we aim to guarantee?
  • Program is syntactically correct
  • Program is type-safe
  • Program follows language-level rules (e.g., exceptions, imports)

• What is required to produce a program that can guarantee the above?
The Problem

1. Declaring & initializing variables

2. Finding expressions of the right type

3. Synthesizing code to handle language rules

Type-directed synthesis
Programs from Sketches

1. Given an **environment** of variables…
   - Map from variable names to types
   - Example:

2. and a set of **functions** …
   - Library methods associated with each type
   - Example, for String: `substring: Integer String, concat: String String,` …

3. **find an expression** of a target type
   - Example, String:
     `x, y.toString(), x.concat(y.toString()), x.substring(y)`
   - Search over space of function compositions
     - Type-based pruning and cost-based heuristic
Programs from Sketches

Invocation chain: a composition of method calls
• $a().b().c()...$ or $a(b(c(...)))$ or a mixture

Two-step enumerative search
1. Up to bounded breadth, gather all invocation chains

Target:

- `toString()`
- `concat(String)`
- `contains(String)`
2. For each invocation chain, recursively search for expressions for arguments in the chain
   • During search, prune invocation chain if return type of chain is such that is not a subtype of

Type-based Pruning

toString()

concat(String)

contains(String)

Target: CharSequence

Target: String

Recursive Search

Return type: Boolean
Not a subtype of CharSequence
Cost-based Heuristic

• How to order the search for expressions?
  • No definitive answer, use a heuristic cost function

1. Performance: expression should be found quickly
2. Parsimony: expression should be simple
3. Relevance: expression should use user-provided variables
4. …

• Cost function sorts a list of invocation chains or expressions according to heuristics
  • Implicitly controls the “distribution”
**Programs from Sketches**

Given

1. An **environment**, initially user-provided
2. Function that finds an expression of type from environment

Let be the function that synthesizes a sketch expression into code in environment

<table>
<thead>
<tr>
<th>Sketch expression</th>
<th>Code produced by</th>
<th>Update to</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>( x = e_1.a(e_2, ..., e_n) )</td>
<td>( \text{where is the return type of method } a )</td>
</tr>
<tr>
<td>( \text{loop (cond) } { \text{body} } )</td>
<td>( \text{while } ((b=) { \text{body} } )</td>
<td>( \text{...} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Programs from Sketches

Post-processing of code
- Add variable declarations from new variables in environment
- Add import declarations, try-catch for unhandled exceptions, etc.

Caveats
1. Some sketches may not be synthesizable into programs
   - Environment is not sufficient to find expressions
   - Neural model went crazy (e.g., void method in loop condition) – experiments show extremely unlikely

2. Many Java APIs utilize generic types
   - Requires search for types before search for expressions of type
   - Wildcard types, bounded types, …
Experiments

• **Corpus:** online repository of
  • 1500 Android apps
  • 100M lines of code
  • 150K methods, randomly selected 10K methods in test set

• **Data:** convert all Java code into canonical subset of Java without syntactic sugar
  • Each method is a “program”
  • From each method, extract evidence, sketch

• **Implementation:** *Bayou*
  • Refer to paper for hyper-parameters and training environment
Experiments

Statistics on Evidences

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Vocab</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_{\text{Calls}}$</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>2584</td>
</tr>
<tr>
<td>$X_{\text{Types}}$</td>
<td>1</td>
<td>15</td>
<td>3</td>
<td>1521</td>
</tr>
<tr>
<td>$X_{\text{Keys}}$</td>
<td>2</td>
<td>29</td>
<td>8</td>
<td>993</td>
</tr>
<tr>
<td>$X$</td>
<td>4</td>
<td>48</td>
<td>13</td>
<td>5098</td>
</tr>
</tbody>
</table>
Training

• Clustering of GED latent space () after training
Inference

• **Goal 1**: test accuracy of model in synthesizing programs
  • Problem: semantic equivalence is **undecidable**!
    • Approximately measure equivalence using
      1. Syntactic check – decidable
      2. Quantitative metrics – How similar are the sets/sequences of API calls, structures in code, etc.?

• **Goal 2**: measure the effect of sketch learning on accuracy
• **Goal 3**: how does the number of input evidences affect accuracy?
• **Goal 4**: how does the GED compare with related models?
• **Goal 5**: how well does it generalize to unseen data?
Inference

Ratio where expected test AST came in top-10

<table>
<thead>
<tr>
<th>Observability/Model</th>
<th>100%</th>
<th>75%</th>
<th>50%</th>
<th>25%</th>
</tr>
</thead>
<tbody>
<tr>
<td>GED-NoSkch</td>
<td>0.13</td>
<td>0.09</td>
<td>0.07</td>
<td>0.02</td>
</tr>
<tr>
<td>GSNN-NoSkch</td>
<td>0.07</td>
<td>0.04</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>GED-Sketch</td>
<td>0.59</td>
<td>0.51</td>
<td>0.44</td>
<td>0.21</td>
</tr>
<tr>
<td>GSNN-Sketch</td>
<td>0.57</td>
<td>0.48</td>
<td>0.41</td>
<td>0.18</td>
</tr>
</tbody>
</table>

- **Observability**: percentage of total input evidence that model was provided
- **GSNN**: Model related to GED, Gaussian Stochastic NN [Sohn 2016]
- **NoSkch**: model trained directly over AST of programs
Qualitative Evaluation

API calls: \{ \text{readLine} \}

Probability: 0.08

\[
\begin{array}{l}
\text{call}\ \\
\text{InputStreamReader.new(InputStream)} \\
\text{call}\ \text{BufferedReader.new(Reader)} \\
\text{call}\ \text{BufferedReader.readLine()} \\
\end{array}
\]

Probability: 0.06

\[
\begin{array}{l}
\text{call}\ \text{BufferedReader.new(Reader)} \\
\text{call}\ \text{BufferedReader.readLine()} \\
\end{array}
\]

Probability: 0.01

\[
\begin{array}{l}
\text{call}\ \text{FileReader.new(File)} \\
\text{call}\ \text{BufferedReader.new(Reader)} \\
\text{call}\ \text{BufferedReader.readLine()} \\
\end{array}
\]
Qualitative Evaluation

API calls: \{ \texttt{readLine} \} 
Types: \{ \texttt{File} \}

\textbf{Note}: did not explicitly specify \texttt{FileReader}
Conclusion

• A method for generating type-safe programs in a Java-like language from uncertain inputs.

• Key insight: learn over sketches (abstractions) of programs, then use combinatorial methods to generate final program.

• Implementation — Bayou — shows promise in generating complex method bodies from a few tokens.

• Future work:
  • Neural architecture for program generation from natural language.
  • Permit instance-specific constraints during program generation using semantic information.

Big Takeaway

To synthesize code:
1. Use machine learning to learn to generate sketches.
2. Use formal methods to synthesize final code.
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
BayouDebug: A recap

Two random variables:
• $\text{Evidence}$ (set of syntactic tokens)
• $\text{Behavior}$ (sequence of program actions).

During training, learn a distribution

While debugging a program with evidence, start with a distribution.

**Anomaly score for Prog:** compute a statistical distance.
Corpus of Programs

Feature extractor
- API calls
- Behaviors

Features & Behaviors from corpus

Inference

Features & Behaviors of test program

Test Program F
```java
foo(File f) {
    f.read();
    f.close();
}
```

Distribution over Features & Behaviors

Statistical Learning (Deep Neural Network)

$P(Y|Prog)$

$P(Y|X)$

Anomaly Score (Aggregate)
Implementing BayouDebug

• Can be implemented using the same model as BayouSynth.
  • Current efforts along these lines.

• Here we will show a different implementation used in the original conference paper.

• Example of how the Bayou framework can be implemented in multiple ways.
Example: Visual Idioms

AlertDialog.Builder b =
   new AlertDialog.Builder(this);
b.setTitle(R.string.title_variable_to_insert);
if (focus.getId() == R.id.tmpl_item) {
   b.setItems(R.array.templatebodyvars, this);
}
else if (focus.getId() == R.id.tmpl_footer) {
   b.setItems(R.array.templateheaderfootervars, this);
}
b.show();

This dialog box cannot be closed
Generative probabilistic automaton
[Murawski & Ouaknine, 2005]

AlertDialog.Builder b =
new AlertDialog.Builder(...);
b.setTitle(...);
if (...) {
    b.setItems (...);
}
else if (...) {
    b.setItems(...);
}
else if (...) {
    b.setItems(...);
}
b.show();

Produced using static analysis
Statistical model

- Introduce a latent variable
- Represents a program’s \textit{true specification}.
  - Controls syntactic features $X$ as well as behaviors $Y$.
  - Distribution captures frequency of different “types” of programs
    - GUI programs, low-level system programs, scientific programs,…

\textbf{Assumption:} $X$ and $Y$ conditionally independent given $Z$. 
Statistical Model

- $Z$ represented as a real vector.
- obtained from a topic model called Latent Dirichlet Allocation (LDA) [Blei, Ng, and Jordan, 2003]
- given by a topic-conditioned recurrent neural network [Mikolov and Zweig, 2012]
Latent Dirichlet Allocation (LDA)

Generative topic model, widely used in NLP

• Models a bag of symbols as a distribution over *topics*
  • A bag can be 50% ‘dog-related’, 30% ‘cat-related’, 20% ‘other’
• Models a topic as a distribution over symbols
  • ‘dog-related’ generates “woof” and “bark” with high probability
For us…

Symbols are API calls.

A specification is a topic distribution!

• Topics represent different APIs, or distinct ways of using the same API.
• A specification is a way of mixing different styles.

Algorithmically:

• Training process learns a full joint distribution $P(X, Z)$.
• During inference, use a sampling technique, for example Gibbs sampling, to estimate $P(Z | X)$. 
Top 5 symbols from a few topics in an Android corpus.
A is the DialogBox API. B and C are other APIs
Recurrent neural networks (RNNs) model a distribution $P(Y)$ over sequences.

A *topic-conditioned RNN* also takes in a topic distribution, and implements $P(Y | Z)$. 

![Diagram of topic-conditioned recurrent neural networks]
Tying it all together

To estimate

- Sample using Gibbs sampling.
- For each , sample using topic-conditioned RNN.
- The ’s follow
Anomaly detection

**Goal:** Compute the sum

where $Y$ ranges over paths in an automaton.

A problem in automata analysis!

We estimate sum by sampling.
AlertDialog.Builder b = new AlertDialog.Builder(...);
b.setTitle(...);
if (…) {
    b.setItems (...);
}
else if (…) {
    b.setItems(...);
}
b.show();

• The model assigns the buggy path a very low probability.
• Leads to a high anomaly score (3.16) for the program.
• Deleting the path from the automaton reduces score to 0.01.
BayouDebug*

- Detecting anomalous API usage in Java/Android code
  - Built using Tensorflow, scikit-learn, and Soot

- Evaluated on a corpus of 2500 Android apps, ~180 million lines of code

* Called Salento in original paper.
Sample bugs

- Showing dialog boxes without buttons
- Using improper encryption mode
- Single crypto object used to encrypt/decrypt multiple data
- Closing unopened Bluetooth socket
- Failed socket connection left unclosed
- Dialog displayed without message
- Unusual button text
- ...

Example behaviors of programs with top-10% anomaly scores
Distribution of anomaly scores

![Distribution of anomaly scores](image)

- **X-axis:** Program models
- **Y-axis:** Anomaly scores
- Color bar indicates anomaly score values from 5 to 50
- The graph shows the distribution of anomaly scores across program models.
Precision-recall plot
Effect of random mutations
Outline

• Introduction to the Bayou framework
  • BayouSynth
  • Underlying probabilistic model
  • BayouDebug

• Implementing BayouSynth with deep neural networks
  • Feed-forward Neural Network
  • Recurrent Neural Network
  • The Encoder-Decoder architecture
  • Gaussian Encoder-Decoder
  • Type-directed synthesis

• Implementing BayouDebug
  • Latent Dirichlet Allocation and Topic-Conditioned RNN

• Conclusion
1. Uncertainty matters

- Formal methods and programming systems research typically ignore uncertainty in intent and incompleteness of knowledge.

- This is unfortunate, as programming is a human process.
2. Big Code can help

• In formal methods and programming systems, one typically solves each problem from scratch.

• We can do better by exploiting common idioms and specifications.

• Statistical models trained on large code corpora can provide this knowledge.

• Bayou uses deep models. However, in some scenarios, non-deep models with explicitly represented features might work as well or better.
Many models in recent work

- Graphical models:

- Extensions of Probabilistic grammars

- Feature synthesis + probabilistic grammars:

- Graph neural networks:
Many models in recent work

• Graphical models:
  • Predicting program properties from Big Code. Raychev, Vechev, and Krause. POPL 2015.

• Extensions of Probabilistic grammars
  • Mining idioms from source code. Allamanis & Sutton. FSE 2014.
  • Structured generative models of natural source code. Maddison & Tarlow. ICML 2014.

• Feature synthesis + probabilistic grammars:

• Graph neural networks:
  • Learning to represent graphs with programs. Allamanis, Brockschmidt, Khademi. ICLR 2018.

Emerging wisdom: straightforward application of off-the-shelf ML models can only go so far
3. PL matters

• Programs are different from traditional ML domains:
  • More structured
  • Crisp requirements such as type safety.

• PL ideas such as types, logical deduction, compositionality are critical to handling discrete program structure and enforcing guarantees.

• **Needed**: a science of software that combines classic PL with statistical, data-driven ideas.
Thank You!