Machine Learning in Compilers

Institute for Computing Systems Architecture
University of Edinburgh, UK

Hugh Leather
Machine learning in compilers

Start with compiler data structures
AST, RTL, SSA, CFG, DDG, etc.
Machine learning in compilers

Human expert determines a mapping to a feature vector

- number of instructions
- mean dependency depth
- branch count
- loop nest level
- ...
- trip count
Now collect many examples of programs, determining their feature values.

Execute the programs with different compilation strategies and find the best for each.

```
Features...

Best Optimisation Parameters
```
Machine learning in compilers

Now give these examples to a machine learner

It learns a model

Supervised Machine Learner

Examples

Features

... Best Heuristic Value

Model
Machine learning in compilers

This model can then be used to predict the best compiler strategy from the features of a new program.

Our heuristic is replaced.
Machine learning in compilers

- A model is really just a way of fitting a curve to data
A model is really just a way of fitting a curve to data
Machine learning in compilers

- Gives heuristic for unseen points
Example: Partitioning Stream Programs

Partitioning Stream Programs

- Map the input program graph to threads
- Need to find a good one from many possible partitions

3 possible partitions on a 2-core machine
Use a sequence of merging and splitting operations to generate a partition

Compact graph representation.
A Two Step Approach

1. Predict characteristics of the ideal partition

We do NOT run any of the generated partition for searching
A Two Step Approach

1. Predict characteristics of the ideal partition

Input Graph → Input Features → Model → Ideal Partition Features

- Pipeline depth
- Load balance
- #Comms channels
- Avg insns

Step 1

We do NOT run any of the generated partition for searching
A Two Step Approach

1. Predict characteristics of the ideal partition
2. Search for a partition with those characteristics

We do NOT run any of the generated partition for searching
Step 1: Prediction

- Nearest neighbour algorithm to predict the characteristics of the ideal structure of the input program

![Graph showing input program, training program, and characteristics of the best partition.](graph.png)
Step 2: Search

- Select a randomly generated partition whose structure is the most close to the predicted one.

We do not run the program!
Results

- ML significantly outperforms state-of-the-art
- Not far from Oracle ("Best") performance
Automatic Feature Generation (Removing the human expert)
Choosing Features

- **Problem**
  - ML relies on good features
  - Subtle interaction between features and ML
  - Infinite number of features to choose from

- **Solution**
  - *Automatically search for good features!*
The Problem

- The expert must do a good job of projecting down to features

amount of white space
average identifier length
age of programmer
name of program
...
...

The diagram illustrates the process of projecting down to features with a tree-like structure, where each node represents a feature or a sub-feature.
Machine learning works well when all examples associated with one feature value have the same type.
The Problem

- Machine learning doesn't work if the features don't distinguish the examples
The Problem

- Better features might allow classification

Feature 3 = 0

Feature 3 = 1
The Problem

- There are much more subtle interactions between features and ML algorithm
  - Sometimes adding a feature makes things worse
  - A feature might be copies of existing features
- There is an infinite number of possible features
An example – Loop unrolling

- Set up
  - 57 benchmarks from MiBench, MediaBench and UTDSP
  - Found best unroll factor for each loop in [0-16]
  - Exhaustive evaluation to find oracle
An example – Loop unrolling

Original Loop
for( i = 0; i < n; i = i ++ ) {
    c[i] = a[i] * b[i];
}

Unrolled 5 times
for( i = 0; i < n; i = i + k ) {
    c[i+0] = a[i+0] * b[i+0];
    c[i+1] = a[i+1] * b[i+1];
    c[i+2] = a[i+2] * b[i+2];
    c[i+3] = a[i+3] * b[i+3];
    c[i+4] = a[i+4] * b[i+4];
    c[i+5] = a[i+5] * b[i+5];
}
GCC vs Oracle

- GCC gets 3% of maximum
- On average mostly not worth unrolling
State of the art features

- Lots of good work with hand-built features
  - Dubach, Cavazos, etc
- Stephenson was state of the art
  - Tackled loop unrolling heuristic
  - Spent some months designing features
  - Multiple iterations to get right
GCC vs Stephenson

- Gets 59% of maximum!
- Machine learning does well
### GCC vs Stephenson

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>Stephenson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic</td>
<td>Months</td>
<td></td>
</tr>
<tr>
<td>Features</td>
<td>-</td>
<td>Months</td>
</tr>
<tr>
<td>Training</td>
<td>-</td>
<td>Days</td>
</tr>
<tr>
<td>Learning</td>
<td>-</td>
<td>Seconds</td>
</tr>
<tr>
<td>Results</td>
<td>3%</td>
<td>59%</td>
</tr>
</tbody>
</table>

- To scale up, must reduce feature development time
A feature space for a motivating example

- Simple language the compiler accepts:
  - Variables, integers, '+', '*', parentheses

- Examples:
  - a = 10
  - b = 20
  - c = a * b + 12
  - d = a * ((b + c * c) * (2 + 3))
A feature space for a motivating example

- What type of features might we want?

\[ a = ((b+c)*2 + d) * 9 + (b+2)*4 \]
A feature space for a motivating example

- What type of features might we want?

\[ a = ((b+c) \times 2 + d) \times 9 + (b+2) \times 4 \]
A feature space for a motivating example

- What type of features might we want?

\[ a = ((b+c)^2 + d) \times 9 + (b+2)^4 \]
A feature space for a motivating example

- What type of features might we want?

\[
\text{Value} = 3
\]

\[
a = ((b+c)*2 + d) * 9 + (b+2)*4
\]
A feature space for a motivating example

- Define a simple feature language:

  \[
  \text{<feature>} ::= \text{"count-nodes-matching("
  \text{<matches>"")
  
  \text{<matches>} ::= \text{"is-constant"}
  | \text{"is-variable"}
  | \text{"is-any-type"}
  | ( \text{"is-plus" | "is-times" }
  ( \text{"&& left-child-matches(" <matches> "")" }
  ( \text{"&& right-child-matches(" <matches> "")" })
  
  \text{GCC grammar is huge >160kb}
  
  \text{Genetic search for features that improve machine learning prediction}
Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Start with the root non-terminal

Grammar

\[
\langle A \rangle ::= \langle A \rangle \langle A \rangle \langle A \rangle \\
| \quad \text{“b”}
\]

Sentence

\begin{align*}
A
\end{align*}
Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Choose randomly among productions and replace

**Grammar**

```plaintext
<A> ::= <A><A><A> | “b”
```

**Sentence**

```
AAA
```
Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Repeat for each non-terminal still in the sentence

**Grammar**

\[
\text{<A>} ::= \text{<A><A><A>} \mid \text{“b”}
\]

**Sentence**

\[
b\text{AAAb}
\]
Generate a feature from a grammar

- Now generate sentences from the grammar to give features
- Continue until there are no more non-terminals

Grammar

\[<A> ::= <A><A><A> | “b”\]

Sentence

“bbbbbb”
Genetic search over features

- Search space is parse trees of features
- Genetic programming searches over feature parse trees
- Features which help machine learning are better
Results

- GCC 3%  Stephenson 59%  Ours 75%
- Automated features outperform human ones
Results

- Top Features Found

39%  ▪  get-attr(@num-iter)
Results

- **Top Features Found**

  39%  
  - `get-attr(@num-iter)`

  14%  
  - `count(filter(/*, !((is-type(wide-int) || (is-type(float extend) && [is-type(reg)]/count(filter(/*, is-type(int)))))) || is-type(union type))))`
Results

- Top Features Found

39%  - get-attr(@num-iter)

14%  - count(filter(/*, !(is-type(wide-int) || (is-type(float extend) && ![is-type(reg)]/count(filter(/*,is-type(int)))) || is-type(union type))))

8%   - count(filter(/*, (is-type(basic-block) && (@loop-depth==2 || (0.0 > (count(filter(/*, is-type(var decl)))) -
(\text{count(filter(/*, (is-type(xor) \&\& \text{@mode==HI}))} +
s\text{sum(filter(/*, (is-type(call insn) \&\& has-attr(@unchanging))))},
c\text{count(filter(/*, is-type(real type))))})))) /
c\text{count(filter(/*, is-type(code label))))}}))))
GCC vs Stephenson vs Ours

<table>
<thead>
<tr>
<th></th>
<th>GCC</th>
<th>Stephenson</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heuristic</td>
<td>Months</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Features</td>
<td>-</td>
<td>Months</td>
<td>-</td>
</tr>
<tr>
<td>Training</td>
<td>-</td>
<td>Days</td>
<td>Days</td>
</tr>
<tr>
<td>Learning</td>
<td>-</td>
<td>Seconds</td>
<td>Hours</td>
</tr>
<tr>
<td>Results</td>
<td>3%</td>
<td>59%</td>
<td>75%</td>
</tr>
</tbody>
</table>
Conclusion

- Analytic approaches no longer working
- Iterative compilation
  - Empirical and good but too slow
- Machine learning here to stay
  - Outperforming human heuristics
  - Very fast development time
- Now used for many things
  - Multi-core, GPGPU, Mobile, JIT, SQL, etc.