Stuff I did in the Spring while not Replying to Email
(aka “advances in structured prediction”)

Hal Daumé III | University of Maryland | me@hal3.name | @haldaume3
Examples of structured prediction
Sequence labeling

x = the monster ate the sandwich
y = Dt Nn Vb Dt Nn

x = Yesterday I traveled to Lille
y = - PER - - LOC
NLP algorithms use a kitchen sink of features
(Bipartite) matching
Machine translation

Moscow stressed tone against Iran on its nuclear program. He called Russian Foreign Minister Tehran to take concrete steps to restore confidence with the international community, to cooperate fully with the IAEA. Conversely Tehran expressed its willingness.
Protein secondary structure prediction
Outline

➢ Background: learning to search

➢ Stuff I did in the Spring
  ➢ Imperative DSL/library for learning to search
  ➢ SOTA examples for tagging, parsing, relation extraction, etc.
  ➢ Learning to search under bandit feedback
  ➢ Hardness results for learning to search
  ➢ Active learning for accelerating learning to search

➢ Stuff I'm trying to do now
  ➢ Distant supervision
  ➢ Mashups with recurrent neural networks

Isn't this kinda narrow?
My experience, 6 months in industry

- **Standard adage:** academia=freedom, industry=time
  - Number of responsibilities vs number of bosses

- **Aspects I didn't anticipate**
  - Breadth (academia) versus depth (industry)
  - Collaborating through students versus directly
  - Security through tenure versus security through $$

- **At the end of the day:** who are your colleagues and what do you have to do to pay the piper?

Major caveat: this is comparing a top ranked CS dept to top industry lab, in a time when there's tons of money in this area (more in industry)
Part of Speech Tagging

NN  NNS  VBP  DT  NN  NN  IN  NNS
NLP  algorithms  use  a  kitchen  sink  of  features

Dependency Parsing

*ROOT*
NLP  algorithms  use  a  kitchen  sink  of  features
Joint prediction via learning to search

Joint Prediction Haiku

A joint prediction
Across a single input
Loss measured jointly

*ROOT*

features
Back to the original problem...

- How to optimize a discrete, joint loss?

- **Input:** \( x \in X \)
- **Truth:** \( y \in Y(x) \)
- **Outputs:** \( Y(x) \)
- **Predicted:** \( \hat{y} \in Y(x) \)
- **Loss:** \( \text{loss}(y, \hat{y}) \)
- **Data:** \( (x,y) \sim D \)
Back to the original problem...

- How to optimize a discrete, joint loss?

- Input: \( x \in X \)
- Truth: \( y \in Y(x) \)
- Outputs: \( Y(x) \)
- Predicted: \( \hat{y} \in Y(x) \)
- Loss: \( \text{loss}(y, \hat{y}) \)
- Data: \( (x, y) \sim D \)

**Goal:**

\[
\begin{align*}
\text{find } & h \in H \\
\text{such that } & h(x) \in Y(x) \\
\text{minimizing } & \mathbb{E}_{(x, y) \sim D} \left[ \text{loss}(y, h(x)) \right] \\
\text{based on } & N \text{ samples}
\end{align*}
\]

\[
\begin{align*}
(x_n, y_n) \sim D
\end{align*}
\]
Search spaces

- When $y$ decomposes in an ordered manner, a sequential decision making process emerges.
Search spaces

- When $y$ decomposes in an ordered manner, a sequential decision making process emerges.

Encodes an output $\hat{y} = \hat{y}(e)$ from which $\text{loss}(y, \hat{y})$ can be computed (at training time).
Policies

- A policy maps observations to actions

\[ \pi(\text{obs.}, x, t, \tau, \text{... anything else}) = a \]
An analogy from playing Mario

From Mario AI competition 2009

Input:

Output:
Jump in \{0,1\}
Right in \{0,1\}
Left in \{0,1\}
Speed in \{0,1\}

High level goal:
Watch an expert play and learn to mimic her behavior
Training (expert)
Warm-up: Supervised learning

1. Collect trajectories from expert $\pi^{\text{ref}}$
2. Store as dataset $D = \{ (o, \pi^{\text{ref}}(o,y)) \mid o \sim \pi^{\text{ref}} \}$
3. Train classifier $\pi$ on $D$

• Let $\pi$ play the game!
Test-time execution (sup. learning)

Supervised Approach after 100K Training Samples
What's the (biggest) failure mode?

The expert never gets stuck next to pipes

⇒ Classifier doesn't learn to recover!
Warm-up II: Imitation learning

1. Collect trajectories from expert $\pi^{\text{ref}}$
2. Dataset $D_0 = \{ (o, \pi^{\text{ref}}(o,y)) | o \sim \pi^{\text{ref}} \}$
3. Train $\pi_1$ on $D_0$
4. Collect new trajectories from $\pi_1$
   - But let the expert steer!
5. Dataset $D_1 = \{ (o, \pi^{\text{ref}}(o,y)) | o \sim \pi_1 \}$
6. Train $\pi_2$ on $D_0 \cup D_1$

• In general:
  • $D_n = \{ (o, \pi^{\text{ref}}(o,y)) | o \sim \pi_n \}$
  • Train $\pi_{n+1}$ on $\cup_{i \leq n} D_i$

If $N = T \log T$, $L(\pi_n) < T \epsilon_N + O(1)$ for some $n$
Test-time execution (DAgger)
What's the biggest failure mode?

Classifier only sees *right* versus *not-right*

- No notion of *better* or *worse*
- No *partial credit*
- Must have a single *target* answer
Learning to search: AggraVaTe

1. Let learned policy $\pi$ drive for $t$ timesteps to obs. $o$

2. For each possible action $a$:
   - Take action $a$, and let expert $\pi_{\text{ref}}$ drive the rest
   - Record the overall loss, $c_a$

3. Update $\pi$ based on example:
   $$(o, \langle c_1, c_2, \ldots, c_K \rangle)$$

4. Goto (1)
Training time versus test accuracy

POS Tagging (tuned hps)

Accuracy (per word)

Training time (minutes)

Accuracy:
- OAA: 96.6
- CRF++: 96.1
- StrPerc: 96.1
- StrSVM: 96.1
- StrSVM2: 95.9

Methods:
- OAA
- L2S
- L2S (ft)
- CRFsgd
- CRF++
- StrPerc
- StrSVM
- StrSVM2
Training time versus test accuracy
Test time speed

![Prediction (test-time) Speed](chart.png)
State of the art accuracy in....

- Part of speech tagging (1 million words)
  - US: 6 lines of code  
    10 seconds to train
  - CRFsgd: 1068 lines  
    30 minutes
  - CRF++: 777 lines  
    hours

- Named entity recognition (200 thousand words)
  - US: 30 lines of code  
    5 seconds to train
  - CRFsgd:  
    1 minute
  - CRF++:  
    10 minutes
  - SVM\textsuperscript{str}: 876 lines  
    30 minutes (suboptimal accuracy)
The Magic

- You write some greedy “test-time” code
  - In your favorite imperative language (C++/Python)
  - It makes arbitrary calls to a *Predict* function
  - And you add some minor decoration

- We will automatically:
  - Perform learning
  - Generate non-deterministic (beam) search
  - Run faster than specialized learning software
How to train?

1. Generate an initial trajectory using a rollin policy

2. Foreach state $R$ on that trajectory:
   a) Foreach possible action $a$ (one-step deviations)
      i. Take that action
      ii. Complete this trajectory using a rollout policy
      iii. Obtain a final loss
   b) Generate a cost-sensitive classification example:
      \[ (\Phi(R), \langle c_a \rangle_{a \in A}) \]
The magic in practice

void run(search& sch, vector<example*> ec) {
  for (size_t i=0; i<ec.size(); i++) {
    uint32_t y_true = get_example_label(ec[i]);
    uint32_t y_pred = sch.predict(ec[i], y_true);
    sch.loss(y_true != y_pred);
    if (sch.output().good())
      sch.output() << y_pred << ' ';
  }
}

A “hint” about the correct decision 😊 only at training time not hiding anything...
The illusion of control

- Execute run $O(T \times A)$ times, modifying Predict
- For each time step $myT = 1 .. T$:
  
  For each possible action $myA = 1 .. A$:
  
  define $Predict(\ldots) = \begin{cases} 
  myA & \text{if } t = myT \\
  \pi & \text{otherwise}
  \end{cases}$

  run your code in full
  set $cost_a = \text{result of } Loss$

  Make classification example on $x_{myT}$ with $<cost_a>$

```cpp
run(vector<example> ec)
  for i = 0 .. ec.size
    y_true = get_example_label(ec[i])
    y_pred = Predict(ec[i], y_true)
    Loss( # of y_true != y_pred )
```
Goal: find the Entities and then find their Relations

<table>
<thead>
<tr>
<th>Method</th>
<th>Entity F1</th>
<th>Relation F1</th>
<th>Train Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured SVM</td>
<td>88.00</td>
<td>50.04</td>
<td>300 seconds</td>
</tr>
<tr>
<td>L2S</td>
<td>92.51</td>
<td>52.03</td>
<td>13 seconds</td>
</tr>
</tbody>
</table>

L2S uses ~100 LOC.
Dependency parsing

L2S uses \( \sim 300 \) LOC.
Outline

➢ Background: learning to search

➢ Stuff I did in the Spring
  ➢ Imperative DSL/library for learning to search
  ➢ SOTA examples for tagging, parsing, relation extraction, etc.
  ➢ Learning to search under bandit feedback
  ➢ Hardness results for learning to search
  ➢ Active learning for accelerating learning to search

➢ Stuff I'm trying to do now
  ➢ Distant supervision
  ➢ Mashups with recurrent neural networks
Structured Contextual Bandit

Font size

Color

Position

- Loss of a single structured label can be observed.
A Search Problem
A Search Problem

Convert SCB into search problem (search space + actions).
Define structured features over each state.

- Learn policy mapping features to actions.
A Search Problem

Use status quo system as reference policy (Ref).
- Goal is to improve on Ref.
<table>
<thead>
<tr>
<th>L2S</th>
<th>SCB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Connection with Learning to Search

<table>
<thead>
<tr>
<th></th>
<th>L2S</th>
<th>SCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Space</td>
<td>States, Actions, Features</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2S</td>
<td>SCB</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Search Space</td>
<td>States, Actions, Features</td>
<td></td>
</tr>
<tr>
<td>Feedback</td>
<td>Labels</td>
<td>Single Loss</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Connection with Learning to Search

<table>
<thead>
<tr>
<th></th>
<th>L2S</th>
<th>SCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Space</td>
<td>States, Actions, Features</td>
<td></td>
</tr>
<tr>
<td>Feedback</td>
<td>Labels</td>
<td>Single Loss</td>
</tr>
<tr>
<td><strong>Ref</strong> Quality</td>
<td>Optimal</td>
<td>Can be bad</td>
</tr>
</tbody>
</table>
### Connection with Learning to Search

<table>
<thead>
<tr>
<th></th>
<th>L2S</th>
<th>SCB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search Space</strong></td>
<td>States, Actions, Features</td>
<td></td>
</tr>
<tr>
<td><strong>Feedback</strong></td>
<td>Labels</td>
<td>Single Loss</td>
</tr>
<tr>
<td><strong>Ref Quality</strong></td>
<td>Optimal</td>
<td>Can be bad</td>
</tr>
<tr>
<td><strong>Ref Availability</strong></td>
<td>Training only</td>
<td>Train and Test</td>
</tr>
</tbody>
</table>
## Connection with Learning to Search

<table>
<thead>
<tr>
<th></th>
<th>L2S</th>
<th>SCB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search Space</strong></td>
<td>States, Actions, Features</td>
<td>Single Loss</td>
</tr>
<tr>
<td><strong>Feedback</strong></td>
<td>Labels</td>
<td>Single Loss</td>
</tr>
<tr>
<td><strong>Ref Quality</strong></td>
<td>Optimal</td>
<td>Can be bad</td>
</tr>
<tr>
<td><strong>Ref Availability</strong></td>
<td>Training only</td>
<td>Train and Test</td>
</tr>
<tr>
<td><strong>Goal</strong></td>
<td>Imitate Ref</td>
<td>Improve upon Ref</td>
</tr>
</tbody>
</table>
### Connection with Learning to Search

<table>
<thead>
<tr>
<th></th>
<th>L2S</th>
<th>SCB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search Space</strong></td>
<td>States, Actions, Features</td>
<td>Single Loss</td>
</tr>
<tr>
<td><strong>Feedback</strong></td>
<td>Labels</td>
<td>Single Loss</td>
</tr>
<tr>
<td><strong>Ref Quality</strong></td>
<td>Optimal</td>
<td>Can be bad</td>
</tr>
<tr>
<td><strong>Ref Availability</strong></td>
<td>Training only</td>
<td>Train and Test</td>
</tr>
<tr>
<td><strong>Goal</strong></td>
<td>Imitate Ref</td>
<td>Improve upon Ref</td>
</tr>
</tbody>
</table>

Existing L2S algorithms give:

\[
L(\pi) \leq L(\pi^{\text{ref}}) + o(1)
\]
Main Goal

Learning to Search with:

1. A suboptimal reference $\Rightarrow$ Learn policy that improves on Ref.
2. Partial feedback.
Compete with Ref.
  - Global optimality if Ref is optimal and realizable.
  - Local optimality.
    - Compete with your own one-step deviations.

\[ \ell(y_e) = 0.2 \]
\[ \ell(y_e) = 0.1 \]
Exploration in Search Space

- Roll-in choice affects what states we train on.
- Roll-out choice affects how we score actions.
# Effect of Roll-in and Roll-out Policies

<table>
<thead>
<tr>
<th>Roll-out → Roll-in</th>
<th>Reference</th>
<th>Half/Half</th>
<th>Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td></td>
<td>Inconsistent</td>
<td></td>
</tr>
<tr>
<td>Learned</td>
<td>No local opt</td>
<td>Good</td>
<td>RL</td>
</tr>
</tbody>
</table>
# Effect of Roll-in and Roll-out Policies

<table>
<thead>
<tr>
<th>roll-out →</th>
<th>Reference</th>
<th>Half/Half</th>
<th>Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ roll-in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td></td>
<td><strong>Inconsistent</strong></td>
<td></td>
</tr>
<tr>
<td>Learned</td>
<td>No local opt</td>
<td>Good</td>
<td>RL</td>
</tr>
</tbody>
</table>

**Theorem**

Roll-in with Ref can generate a model with unbounded structured regret but zero cost-sensitive regret.

- States trained on are not representative of those seen at prediction time.
### Effect of Roll-in and Roll-out Policies

<table>
<thead>
<tr>
<th>roll-out →</th>
<th>Reference</th>
<th>Half/Half</th>
<th>Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll-in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>Inconsistent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learned</td>
<td>No local opt</td>
<td>Good</td>
<td>RL</td>
</tr>
</tbody>
</table>

**Theorem**

Roll-out with Ref may cause the learned policy to fail to converge to a local optimum if the reference policy is suboptimal.

- Causes poor assessment of comparison policies.
### Effect of Roll-in and Roll-out Policies

<table>
<thead>
<tr>
<th>roll-out \rightarrow</th>
<th>Reference</th>
<th>Half/Half</th>
<th>Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll-in \downarrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td></td>
<td>Inconsistent</td>
<td></td>
</tr>
<tr>
<td>Learned</td>
<td>No local opt</td>
<td>Good</td>
<td>RL</td>
</tr>
</tbody>
</table>

**Theorem**

Roll-in and roll-out with the learned policy ignores Ref and is equivalent to reinforcement learning.
## Effect of Roll-in and Roll-out Policies

<table>
<thead>
<tr>
<th>roll-out →</th>
<th>Reference</th>
<th>Half/Half</th>
<th>Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ roll-in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>Inconsistent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learned</td>
<td>No local opt</td>
<td>Good</td>
<td>RL</td>
</tr>
</tbody>
</table>

### Theorem

LOLS minimizes a combination of regret to Ref and regret to its own one-step deviations.
LOLS Regret Bound

**Theorem**

LOLS minimizes a combination of regret to Ref and regret to its own one-step deviations.

\[
\frac{1}{2} \left( L(\hat{\pi}) - L(\pi_{\text{ref}}) \right) + \frac{1}{2} \left( L(\hat{\pi}) - L(\pi_{\text{dev}}) \right) \text{ is small.}
\]

- Regret to Ref
- Regret to devs
LOLS Regret Bound

**Theorem**

LOLS minimizes a combination of regret to Ref and regret to its own *one-step deviations*.

\[
\frac{1}{2} \left( L(\hat{\pi}) - L(\pi_{\text{ref}}) \right) + \frac{1}{2} \left( L(\hat{\pi}) - L(\pi_{\text{dev}}) \right) \geq 0
\]

- Competes with Ref.
LOLS Regret Bound

**Theorem**

LOLS minimizes a combination of regret to Ref and regret to its own one-step deviations.

\[
\frac{1}{2} \left( L(\hat{\pi}) - L(\pi^{\text{ref}}) \right) + \frac{1}{2} \left( L(\hat{\pi}) - L(\pi^{\text{dev}}) \right) \geq 0
\]

Regret to devs

- Competes with Ref.
- Locally optimal when Ref is optimal (even if unrealizable).
LOLS Regret Bound

Theorem

LOLS minimizes a combination of regret to Ref and regret to its own one-step deviations.

\[
\frac{1}{2} \left( L(\hat{\pi}) - L(\pi^{\text{ref}}) \right) + \frac{1}{2} \left( L(\hat{\pi}) - L(\pi^{\text{dev}}) \right) \text{ is small.}
\]

- Competes with Ref.
- Locally optimal when Ref is optimal (even if unrealizable).
- If Ref suboptimal, either locally optimal or better than Ref.
Local Optimality is Hard

Finding local optimum could be hard without further assumptions.

**Theorem**

Can require $\Omega(2^T)$ cost-sensitive classification examples to reach local optimum.

$T$ is the number of decisions per example.
Find the dependency structure of words in a sentence.

<table>
<thead>
<tr>
<th>roll-out →</th>
<th>Reference</th>
<th>Mixture</th>
<th>Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ roll-in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>87.2</td>
<td>89.7</td>
<td>88.2</td>
</tr>
<tr>
<td>Learned</td>
<td>90.7</td>
<td>90.5</td>
<td>86.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference is optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
</tr>
<tr>
<td>Learned</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference is suboptimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
</tr>
<tr>
<td>Learned</td>
</tr>
</tbody>
</table>

LOLS always good, even with suboptimal Ref.
Structured Contextual Bandit

- Loss of a single structured label can be observed.
- Reference policy is not optimal under this setting.
We adapt an $\epsilon$-greedy algorithm
Upon receiving an example:

- $w$ with prob. $1 - \epsilon$: follow the current policy

\[ \ell(y_e) = 0.2 \]
We adapt an $\epsilon$-greedy algorithm.
Upon receiving an example:

- w/ prob. $1 - \epsilon$: follow the current policy
- w/ prob. $\epsilon$: perform a randomized LOLS update.
We adapt an $\epsilon$-greedy algorithm. Upon receiving an example:

- with prob. $1 - \epsilon$: follow the current policy
- with prob. $\epsilon$: perform a randomized LOLS update.
We adapt an $\epsilon$-greedy algorithm
Upon receiving an example:

- w/ prob. $1 - \epsilon$: follow the current policy
- w/ prob. $\epsilon$: perform a randomized LOLS update.
We adapt an $\varepsilon$-greedy algorithm
Upon receiving an example:

- w/ prob. $1 - \varepsilon$: follow the current policy
- w/ prob. $\varepsilon$: perform a randomized LOLS update.

Regret against ref and deviations is still bounded.
Outline

➢ Background: learning to search

➢ Stuff I did in the Spring
  ➢ Imperative DSL/library
  ➢ SOTA examples for tagging, parsing, relation extraction, etc.
  ➢ Learning to search under bandit feedback
  ➢ Hardness results for learning to search
  ➢ Active learning for accelerating learning to search

➢ Stuff I'm trying to do now
  ➢ Distant supervision
  ➢ Mashups with recurrent neural networks

Observation: rollouts at all time steps not equally useful

Solution: importance-weighted active learning selection of where to rollout vs skip

Hacky heuristic: 5* speedup, slightly increased accuracy

Training RNNs with LOLS yields drastic increases in performance on non-adversarial synthetic data
Distant supervision

- Learning with a human in the loop

- Repeat forever:
  - Information need
  - Machine makes complex prediction
  - Human is happy or unhappy, provides extra feedback
  - Machine learns
  - Human learns

- How to handle the last step?
• Novel programming paradigm for integrating ML into software
• State of the art results on many tasks, very quickly, little code
• New problems, new algorithms
• Positive results (notion of local optimality, and regret guarantees)
• Negative results (hardness of exact local optimality)
• Lots of places to go from here...

Thanks! Questions?