Beyond Structured Prediction: Inverse Reinforcement Learning

Hal Daumé III
Computer Science
University of Maryland
me@hal3.name
A Tutorial at ACL 2011
Portland, Oregon
Sunday, 19 June 2011

Acknowledgements

Some slides:
Stuart Russell
Dan Klein
J. Drew Bagnell
Nathan Ratliff
Stephane Ross

Discussions/Feedback:
MLRG Spring 2010
Examples of structured problems

![Molecule diagram]

![Sentence diagram]

The man ate a tasty sandwich.

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.

Shaded Musco هجتها ضد إيران بشأن برنامجها النووي. ودعا وزير الخارجية الروسي طهران إلى اتخاذ خطوات إلمع لاستعادة الثقة مع المجتمع الدولي والتعاون الكامل مع الوكالة الذرية. بالمقابل أبد طهران استعدادها لاستدلال السماح بعمليات التفتيش الفاحش بشرط إسقاط مجلس الأمن منها النووي.
Examples of demonstrations
Examples of demonstrations
# NLP as transduction

<table>
<thead>
<tr>
<th>Task</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Machine Translation</strong></td>
<td>Ces deux principes se tiennent à la croisée de la philosophie, de la politique, de l'économie, de la sociologie et du droit.</td>
<td>Both principles lie at the crossroads of philosophy, politics, economics, sociology, and law.</td>
</tr>
<tr>
<td><strong>Document Summarization</strong></td>
<td>Argentina was still obsessed with the Falkland Islands even in 1994, 12 years after its defeat in the 74-day war with Britain. The country's overriding foreign policy aim continued to be winning sovereignty over the islands.</td>
<td>The Falkland islands war, in 1982, was fought between Britain and Argentina.</td>
</tr>
<tr>
<td><strong>Syntactic Analysis</strong></td>
<td>The man ate a big sandwich.</td>
<td>The man ate a big sandwich.</td>
</tr>
<tr>
<td>...many more...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Learn a function mapping inputs to complex outputs:

\[ f : X \rightarrow Y \]
Structured prediction 101

Learn a function mapping inputs to complex outputs:

\[ f : X \rightarrow Y \]
Why is structure important?

- Correlations among outputs
  - Determiners often precede nouns
  - Sentences usually have verbs

- Global coherence
  - It just *doesn't make sense* to have three determiners next to each other

- My objective (aka “loss function”) forces it
  - Translations should have good sequences of words
  - Summaries should be coherent
Outline: Part I

➢ What is Structured Prediction?
➢ Refresher on Binary Classification
  ➢ What does it mean to learn?
  ➢ Linear models for classification
  ➢ Batch versus stochastic optimization
➢ From Perceptron to Structured Perceptron
  ➢ Linear models for Structured Prediction
  ➢ The “argmax” problem
  ➢ From Perceptron to margins
➢ Structure without Structure
  ➢ Stacking
  ➢ Structure compilation
Outline: Part II

- Learning to Search
  - Incremental parsing
  - Learning to queue
- Refresher on Markov Decision Processes
- Inverse Reinforcement Learning
  - Determining rewards given policies
  - Maximum margin planning
- Learning by Demonstration
  - Searn
  - Dagger
- Discussion
Refresher on Binary Classification
What does it mean to learn?

➢ Informally:
  ➢ to predict the future based on the past

➢ Slightly-less-informally:
  ➢ to take labeled examples and construct a function that will label them as a human would

➢ Formally:
  ➢ Given:
    ➢ A fixed unknown distribution $D$ over $X*Y$
    ➢ A loss function over $Y*Y$
    ➢ A finite sample of $(x,y)$ pairs drawn i.i.d. from $D$
  ➢ Construct a function $f$ that has low expected loss with respect to $D
Feature extractors

- A feature extractor $\Phi$ maps examples to vectors

Dear Sir.

First, I must solicit your confidence in this transaction, this is by virtue of its nature as being utterly confidential and top secret. ...

$\Phi$

<table>
<thead>
<tr>
<th>Feature</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>W=dear</td>
<td>1</td>
</tr>
<tr>
<td>W=sir</td>
<td>1</td>
</tr>
<tr>
<td>W=this</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>W=wish</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>MISSPELLED</td>
<td>2</td>
</tr>
<tr>
<td>NAMELESS</td>
<td>1</td>
</tr>
<tr>
<td>ALLCAPS</td>
<td>0</td>
</tr>
<tr>
<td>NUM_URLS</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Linear models for binary classification

- Decision boundary is the set of “uncertain” points

- Linear decision boundaries are characterized by weight vectors

\[ \sum_i w_i \Phi_i(x) \]

<table>
<thead>
<tr>
<th>x</th>
<th>( \Phi(x) )</th>
<th>w</th>
<th>( \sum_i w_i \Phi_i(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIAS : 1</td>
<td></td>
<td>BIAS : -3</td>
<td>(1)(-3) +</td>
</tr>
<tr>
<td>free : 1</td>
<td></td>
<td>free : 4</td>
<td>(1)(4) +</td>
</tr>
<tr>
<td>money : 1</td>
<td></td>
<td>money : 2</td>
<td>(1)(2) +</td>
</tr>
<tr>
<td>the : 0</td>
<td></td>
<td>the : 0</td>
<td>(0)(0) +</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

“free money”
The perceptron

- Inputs = feature values
- Params = weights
- Sum is the response

- If the response is:
  - Positive, output +1
  - Negative, output -1

- When training, update on errors:

\[ w = w + y \phi(x) \]

“Error” when:

\[ yw \cdot \phi(x) \leq 0 \]
Why does that update work?

- When $y w^{old} \cdot \phi(x) \leq 0$, update $w^{new} = w^{old} + y \phi(x)$.

\[
y w^{new} \phi(x) = y \left( w^{old} + y \phi(x) \right) \phi(x)
\]

\[
= y w^{old} \phi(x) + yy \phi(x) \phi(x)
\]

\[
< 0 \quad > \quad 0
\]
Support vector machines

- Explicitly optimize the margin
- Enforce that all training points are correctly classified

\[
\begin{align*}
\max_{\mathbf{w}} & \quad \text{margin} \quad \text{s.t.} \quad \text{all points are correctly classified} \\
\max_{\mathbf{w}} & \quad \text{margin} \quad \text{s.t.} \quad y_n \mathbf{w} \cdot \phi(x_n) \geq 1, \quad \forall n \\
\min_{\mathbf{w}} & \quad \|\mathbf{w}\|^2 \quad \text{s.t.} \quad y_n \mathbf{w} \cdot \phi(x_n) \geq 1, \quad \forall n
\end{align*}
\]
Support vector machines with \textit{slack}

- Explicitly optimize the \textit{margin}

- Allow some “noisy” points to be misclassified

\[
\begin{align*}
\min_{\mathbf{w}, \xi} & \quad \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_n \xi_n \\
\text{s.t.} & \quad y_n \mathbf{w} \cdot \phi(x_n) + \xi_n \geq 1, \quad \forall n \\
& \quad \xi_n \geq 0, \quad \forall n
\end{align*}
\]
Batch versus stochastic optimization

- Batch = read in all the data, then process it
- Stochastic = (roughly) process a bit at a time

\[
\begin{align*}
\text{min } & \quad \frac{1}{2} \| w \|^2 + C \sum_n \xi_n \\
\text{s.t. } & \quad y_n w \cdot \phi(x_n) + \xi_n \geq 1 \\
& \quad \xi_n \geq 0, \quad \forall n
\end{align*}
\]

- For \( n = 1 \ldots N \):
  - If \( y_n w \cdot \phi(x_n) \leq 0 \)
    - \( w = w + y_n \phi(x_n) \)
Stochastically optimized SVMs

SVM Objective

For $n=1..N$:
- If $y_n w \cdot \phi(x_n) \leq 1$
  - $w = w + y_n \phi(x_n)$
- $w = \left(1 - \frac{1}{CN}\right)w$

Implementation Note:
Weight shrinkage is *SLOW*. Implement it lazily, at the cost of double storage.

For $n=1..N$:
- If $y_n w \cdot \phi(x_n) \leq 0$
  - $w = w + y_n \phi(x_n)$
From Perceptron to Structured Perceptron
Perceptron with multiple classes

- Store separate weight vector for each class \( w_1, w_2, \ldots, w_K \)

- For \( n=1..N \):
  - Predict:
    \[
    \hat{y} = \text{arg} \max_k w_k \cdot \phi(x_n)
    \]
  - If \( \hat{y} \neq y_n \):
    \[
    w_{\hat{y}} = w_{\hat{y}} - \phi(x_n) \\
    w_{y_n} = w_{y_n} + \phi(x_n)
    \]

Why does this do the right thing?
Perceptron with multiple classes v2

- Originally:

  \[ w_1 \quad w_2 \quad w_3 \quad W \]

- For n=1..N:
  - Predict:
    \[ \hat{y} = \arg \max_k w_k \cdot \phi(x_n) \]
  - If \( \hat{y} \neq y_n \)
    \[ w_{\hat{y}} = w_{\hat{y}} - \phi(x_n) \]
    \[ w_{y_n} = w_{y_n} + \phi(x_n) \]

- For n=1..N:
  - Predict:
    \[ \hat{y} = \arg \max_k w \cdot \phi(x_n, k) \]
  - If \( \hat{y} \neq y_n \)
    \[ w = w - \phi(x_n, \hat{y}) + \phi(x_n, y_n) \]
Perceptron

- Originally:

```
<table>
<thead>
<tr>
<th></th>
<th>$w_1$</th>
</tr>
</thead>
</table>
```

“free money”

- For $n=1..N$:
  - Predict:
    
    $\hat{y} = \arg\max_k w_k \cdot \Phi(x_n)$
  
  - If $\hat{y} \neq y_n$
    
    $w_{\hat{y}} = w_{\hat{y}} - \Phi(x_n)$
    
    $w_{y_n} = w_{y_n} + \Phi(x_n)$

- For $n=1..N$:
  - Predict:
    
    $\hat{y} = \arg\max_k w \cdot \Phi(x_n, k)$
  
  - If $\hat{y} \neq y_n$
    
    $w = w - \Phi(x_n, \hat{y}) + \Phi(x_n, y_n)$
Features for structured prediction

- Allowed to encode *anything* you want

\[ \phi(x, y) = \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pro</td>
<td>Md</td>
<td>Vb</td>
<td>Dt</td>
<td>Nn</td>
</tr>
<tr>
<td>I</td>
<td>can</td>
<td>can</td>
<td>a</td>
<td>can</td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{align*}
\text{I} \_ \text{Pro} & : 1 & <s>\_\text{Pro} & : 1 & \text{has}_\_\text{verb} & : 1 \\
\text{can} \_ \text{Md} & : 1 & \text{Pro} \_ \text{Md} & : 1 & \text{has}_\_\text{nn}_\_\text{lft} & : 0 \\
\text{can} \_ \text{Vb} & : 1 & \text{Md} \_ \text{Vb} & : 1 & \text{has}_\_\text{n}_\_\text{lft} & : 1 \\
\text{a} \_ \text{Dt} & : 1 & \text{Vb} \_ \text{Dt} & : 1 & \text{has}_\_\text{nn}_\_\text{rgt} & : 1 \\
\text{can} \_ \text{Nn} & : 1 & \text{Dt} \_ \text{Nn} & : 1 & \text{has}_\_\text{n}_\_\text{rgt} & : 1 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\text{Nn} \_ \langle /s \rangle & : 1 & \ldots & \ldots & \ldots & \ldots \\
\end{align*} \]

- Output features, *Markov features*, other features
Structured perceptron

For $n=1..N$:

- Predict:

\[ \hat{y} = \arg \max_k w \cdot \phi \left( x_n, k \right) \]

- If $\hat{y} \neq y_n$:

\[ w = w - \phi \left( x_n, \hat{y} \right) + \phi \left( x_n, y_n \right) \]
Argmax for sequences

If we only have output and Markov features, we can use Viterbi algorithm:

(plus some work to account for boundary conditions)
Structured perceptron as ranking

- For n=1..N:
  - Run Viterbi: \( \hat{y} = \arg\max_k w \cdot \phi(x_n, k) \)
  - If \( \hat{y} \neq y_n \):
    \[
    w = w - \phi(x_n, \hat{y}) + \phi(x_n, y_n)
    \]

- When does this make an update?

<table>
<thead>
<tr>
<th>Pro</th>
<th>Md</th>
<th>Vb</th>
<th>Dt</th>
<th>Nn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pro</td>
<td>Md</td>
<td>Md</td>
<td>Dt</td>
<td>Vb</td>
</tr>
<tr>
<td>Pro</td>
<td>Md</td>
<td>Md</td>
<td>Dt</td>
<td>Nn</td>
</tr>
<tr>
<td>Pro</td>
<td>Md</td>
<td>Nn</td>
<td>Dt</td>
<td>Md</td>
</tr>
<tr>
<td>Pro</td>
<td>Md</td>
<td>Nn</td>
<td>Dt</td>
<td>Nn</td>
</tr>
<tr>
<td>Pro</td>
<td>Md</td>
<td>Vb</td>
<td>Dt</td>
<td>Md</td>
</tr>
<tr>
<td>Pro</td>
<td>Md</td>
<td>Vb</td>
<td>Dt</td>
<td>Vb</td>
</tr>
</tbody>
</table>

I can can a can
From perceptron to margins

Maximize Margin

$$\min_{w, \xi} \frac{1}{2} \|w\|^2 + C \sum_n \xi_n$$

Minimize Errors

s.t. $$y_n w \cdot \phi(x_n) + \xi_n \geq 1$$, \(\forall n\)

Each point is correctly classified, modulo \(\xi\)

Response for truth

$$\min_{w, \xi} \frac{1}{2} \|w\|^2 + C \sum_n \xi_{n, \hat{y}}$$

Response for other

s.t. $$w \cdot \phi(x_n, y_n) - w \cdot \phi(x_n, \hat{y}) + \xi_n \geq 1$$, \(\forall n, \hat{y} \neq y_n\)

Each true output is more highly ranked, modulo \(\xi\)
From perceptron to margins

\[ \min \begin{align*} 
& w, \xi \\
& \frac{1}{2} \|w\|^2 + C \sum_n \xi_n, \hat{y} 
\end{align*} \]

s.t. \( w \cdot \phi(x_n, y_n) \)

\( -w \cdot \phi(x_n, \hat{y}) \)

\( + \xi_n \geq 1, \forall n, \hat{y} \neq y_n \)

Each true output is more highly ranked, modulo \( \xi \)
Ranking margins

- Some errors are worse than others...

```
Pro  Md  Vb  Dt  Nn
Pro  Md  Md  Dt  Vb
Pro  Md  Nn  Dt  Md
Pro  Md  Nn  Dt  Nn
Pro  Md  Vb  Dt  Md
Pro  Md  Vb  Dt  Vb
I can can a can
```

Margin of one
Accounting for a loss function

➢ Some errors are worse than others...

Margin of $l(y,y')$
Accounting for a loss function

\[ w \cdot \phi(x_n, y_n) - w \cdot \phi(x_n, \hat{y}) + \xi_n \geq l(y_n, \hat{y}) \]

is equivalent to

\[ \max_{\hat{y}} w \cdot \phi(x_n, y_n) - w \cdot \phi(x_n, \hat{y}) + \xi_n \geq l(y_n, \hat{y}) \]

\[ w \cdot \phi(x_n, y_n) - w \cdot \phi(x_n, \hat{y}) + \xi_n \geq 1 \quad \text{and} \quad w \cdot \phi(x_n, y_n) - w \cdot \phi(x_n, \hat{y}) + \xi_n \geq l(y_n, \hat{y}) \]
Augmented argmax for sequences

➢ Add “loss” to each wrong node!

What are we assuming here?
Stochastically optimizing Markov nets

For $n=1..N$:
- Augmented Viterbi:
  \[
  \hat{y} = \arg\max_k w \cdot \phi(x_n, k) + l(y_n, k)
  \]
  \[
  \text{If } \hat{y} \neq y_n:
  w = w - \phi(x_n, \hat{y}) + \phi(x_n, y_n)
  \]
  \[
  w = \left(1 - \frac{1}{CN}\right) w
  \]
- Viterbi:
  \[
  \hat{y} = \arg\max_k w \cdot \phi(x_n, k)
  \]
  \[
  \text{If } \hat{y} \neq y_n:
  w = w - \phi(x_n, \hat{y}) + \phi(x_n, y_n)
  \]
Stacking

- **Structured models**: accurate but slow
  - Output Labels
  - Input Features

- **Independent models**: less accurate but fast
  - Output Labels
  - Input Features

- **Stacking**: multiple independent models
  - Output Labels 2
  - Output Labels
  - Input Features
Training a stacked model

- Train independent classifier $f_1$ on input features
- Train independent classifier $f_2$ on input features + $f_1$'s output
- **Danger:** overfitting!
- **Solution:** cross-validation
Do we really need structure?

- **Structured models**: accurate but slow
  
  ![Structured model diagram]

- **Independent models**: less accurate but fast
  
  ![Independent model diagram]

- **Goal**: transfer power to get fast+accurate
  
  ![Transfer model diagram]

- **Questions**: are independent models...
  
  - … expressive enough? (approximation error)
  
  - … easy to learn? (estimation error)
“Compiling” structure out

Labeled Data

CRF($f_1$)  POS: 95.0%  NER: 75.3%

IRL($f_1$)  POS: 91.7%  NER: 69.1%

$f_1 = \text{words/prefixes/suffixes/forms}$

[Image of a diagram showing the relationship between labeled data and CRF, IRL models with respective POS and NER accuracies.]
“Compiling” structure out

Labeled Data

CRF($f_1$) POS: 95.0% NER: 75.3%

$f_1 = \text{words/prefixes/suffixes/forms}$

IRL($f_1$) POS: 91.7% NER: 69.1%

IRL($f_2$) POS: 94.4% NER: 66.2%

CompIRL($f_2$) POS: 95.0% NER: 72.7%

f_2 = f_1 \text{ applied to a larger window}
Decomposition of errors

CRF($f_1$): $p_C$

- Coherence

marginalized CRF

nonlinearities

IRL($f_\infty$): $p_{A^*}$

global information

IRL($f_2$): $p_{1^*}$

Theorem:

$KL(p_C \parallel p_{1^*}) = KL(p_C \parallel p_{MC}) + KL(p_{MC} \parallel p_{A^*}) + KL(p_{A^*} \parallel p_{1^*})$

Sum of MI on edges

POS=.003 (95.0% → 95.0%)

NER=.009 (76.3% → 76.0%)

Train a truncated CRF

NER: 76.0% → 72.7%

Train a marginalized CRF

NER: 76.0% → 76.0%
Structure compilation results

Part of speech

Named Entity

Parsing

- Structured
- Independent
Coffee Break!!!
Outline: Part I

- What is Structured Prediction?
- Refresher on Binary Classification
  - What does it mean to learn?
  - Linear models for classification
  - Batch versus stochastic optimization
- From Perceptron to Structured Perceptron
  - Linear models for Structured Prediction
  - The “argmax” problem
  - From Perceptron to margins
- Structure without Structure
  - Stacking
  - Structure compilation
Outline: Part II

- Learning to Search
  - Incremental parsing
  - Learning to queue
- Refresher on Markov Decision Processes
- Inverse Reinforcement Learning
  - Determining rewards given policies
  - Maximum margin planning
- Learning by Demonstration
  - Searn
  - Dagger
- Discussion
Learning to Search
Argmax is *hard*!

- Classic formulation of structured prediction:

\[ score(x, y) = \text{something we learn to make “good” } x, y \text{ pairs score highly} \]

- At test time:

\[ f(x) = \operatorname{argmax}_{y \in Y} score(x, y) \]

- Combinatorial optimization problem
  - Efficient only in very limiting cases
  - Solved by heuristic search: beam + A* + local search
Argmax is \textit{hard}!

- Classic formulation of structured prediction:
  - something we learn to make $x,y$ pairs score highly
- At test time:
  - Combinatorial optimization problem
    - Efficient only in very limiting cases
    - Solved by heuristic search: beam + A* + local search

\begin{equation}
  f(x) = \arg\max_{y \in Y} \text{score}(x,y)
\end{equation}

Order these words: bart better I madonna say than ,

Argmax is hard!

- Classic formulation of structured prediction: something we learn to make "good" x,y pairs score highly.

- At test time:
  - Combinatorial optimization problem
    - Efficient only in very limiting cases
    - Solved by heuristic search: beam + A* + local search

\[
f(x) = \arg\max_{y \in Y} \text{score}(x, y)
\]

Order these words: bart better I madonna say than ,
Best search (32.3): I say better than bart madonna ,
Original (41.6): better bart than madonna , I say

Argmax is **hard**!

- **Classic formulation of structured prediction:** something we learn to make "good" $x,y$ pairs score highly

- **At test time:**
  - Combinatorial optimization problem
  - Efficient only in very limiting cases
  - Solved by heuristic search: beam + A* + local search

$$f (x) = \text{argmax}_{y \in Y} \text{score} (x, y)$$

---

**Order these words:** bart better I madonna say than ,

- **Best search** (32.3): I say better than bart madonna ,
- **Original** (41.6): better bart than madonna , I say

**Best search** (51.6): and so could really be a neural apparently thought things as dissimilar firing two identical

Argmax is hard!

- Classic formulation of structured prediction:
  something we learn to make “good” x,y pairs score highly

- At test time:
  Combinatorial optimization problem
  Efficient only in very limiting cases
  Solved by heuristic search: beam + A* + local search

\[
f(x) = \text{argmax}_{y \in Y} \text{score}(x, y)
\]

Order these words: bart better I madonna say than ,
Best search (32.3): I say better than bart madonna ,
Original (41.6): better bart than madonna , I say

Best search (51.6): and so could really be a neural apparently thought things as dissimilar firing two identical
Original (64.3): could two things so apparently dissimilar as a thought and neural firing really be identical

Incremental parsing, early 90s style

Train a classifier to make decisions

S

Right

VP

Left

Unary

I / Pro

can / Md

can / Vb

can / Nn

Left

Up

Right

Left

a / Dt

Right

NP

NP
Incremental parsing, mid 2000s style

Train a classifier to make decisions
Learning to beam-search

- For n=1..N:
  - Viterbi:
    \[ \hat{y} = \arg \max_k w \cdot \phi(x_n, k) \]
  - If \( \hat{y} \neq y_n \)
    \[ w = w - \phi(x_n, \hat{y}) + \phi(x_n, y_n) \]
Learning to beam-search

For n=1..N:
- Run beam search until truth falls out of beam
- Update weights immediately!
Learning to beam-search

For n=1..N:
- Run beam search until truth falls out of beam
- Update weights immediately!
- Restart at truth

[D+Marcus, ICML05; Xu+al, JMLR09]
Incremental parsing results

![Graph showing incremental parsing accuracy over number of passes over training data. The graph compares different update strategies: no early update, no repeated use of examples; early update, no repeated use of examples; early update, repeated use of examples. The accuracy increases with more passes, with the early update strategy showing a slight advantage.]
Generic Search Formulation

➢ Search Problem:
  ➢ Search space
  ➢ Operators
  ➢ Goal-test function
  ➢ Path-cost function

➢ Search Variable:
  ➢ Enqueue function

Varying the **Enqueue** function can give us DFS, BFS, beam search, A* search, etc...

```python
nodes := MakeQueue(S0)

while nodes is not empty
  node := RemoveFront(nodes)
  if node is a goal state
    return node
  next := Operators(node)
  nodes := Enqueue(nodes, next)

fail
```
Online Learning Framework (LaSO)

- nodes := MakeQueue(S0)
- while nodes is not empty
  - node := RemoveFront(nodes)
  - if none of {node} ∪ nodes is y-good **or** node is a goal & not y-good
    - sibs := siblings(node, y)
    - w := update(w, x, sibs, {node} ∪ nodes)
    - nodes := MakeQueue(sibs)
  - else
    - if node is a goal
      - return w
    - next := Operators(node)
    - nodes := Enqueue(nodes, next)

**Monotonicity:** for any node, we can tell if it can lead to the correct solution or not

- If we erred...
  - Where should we have gone?
  - Update our weights based on the good and the bad choices
- Continue search...
- Monotonicity
Search-based Margin

➢ The *margin* is the amount by which we are correct:

Note that the *margin* and hence *linear separability* is also a function of the search algorithm!
Syntactic chunking Results

- Large Margin (beam 5): 4 min
- Large Margin (beam 25): 24 min
- Perceptron Search (Exact): 22 min
- Standard Perceptron Updates: 33 min

[Collins 2002]

[Zhang+Damerau+Johnson 2002]; timing unknown
Tagging+chunking results

![Graph showing joint tagging/chunking accuracy vs. training time (hours) on a log scale.](attachment:image.png)

Sutton+McCallum 2004

[D+Marcu, ICML05; Xu+al, JMLR09]
Variations on a beam

Observation:

- We needn't use the same beam size for training and decoding
- Varying these values independently yields:

<table>
<thead>
<tr>
<th>Training Beam</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.9</td>
<td>92.8</td>
<td>91.9</td>
<td>91.3</td>
<td>90.9</td>
</tr>
<tr>
<td>5</td>
<td>90.5</td>
<td>94.3</td>
<td>94.4</td>
<td>94.1</td>
<td>94.1</td>
</tr>
<tr>
<td>10</td>
<td>89.5</td>
<td>94.3</td>
<td>94.4</td>
<td>94.2</td>
<td>94.2</td>
</tr>
<tr>
<td>25</td>
<td>88.7</td>
<td>94.2</td>
<td>94.5</td>
<td>94.3</td>
<td>94.3</td>
</tr>
<tr>
<td>50</td>
<td>88.4</td>
<td>94.2</td>
<td>94.4</td>
<td>94.2</td>
<td>94.4</td>
</tr>
</tbody>
</table>
What if our model sucks?

- Sometimes our model *cannot* produce the “correct” output
  - canonical example: machine translation

![Diagram showing model outputs, current hypothesis, good outputs, and best achievable output.]
Local versus bold updating...

Machine Translation Performance (Bleu)

- **Monotonic**
  - Bold: 34.5
  - Local: 34.5
  - Pharoah: 35

- **Distortion**
  - Bold: 32.5
  - Local: 33
  - Pharoah: 33.5

Legend:
- Bold
- Local
- Pharoah
Refresher on Markov Decision Processes
Reinforcement learning

➢ Basic idea:
  ➢ Receive feedback in the form of rewards
  ➢ Agent’s utility is defined by the reward function
  ➢ Must learn to act to maximize expected rewards
  ➢ Change the rewards, change the learned behavior

➢ Examples:
  ➢ Playing a game, reward at the end for outcome
  ➢ Vacuuming, reward for each piece of dirt picked up
  ➢ Driving a taxi, reward for each passenger delivered
Markov decision processes

What are the values (expected future rewards) of states and actions?

\[ V(s)^* = 30 \]

\[ Q(s,a1)^* = 30 \]
\[ Q(s,a2)^* = 23 \]
\[ Q(s,a3)^* = 17 \]
Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s,a,s')$
    - Prob that $a$ from $s$ leads to $s'$
    - i.e., $P(s' | s,a)$
    - Also called the model
  - A reward function $R(s,a,s')$
    - Sometimes just $R(s)$ or $R(s')$
  - A start state (or distribution)
  - Maybe a terminal state

- MDPs are a family of non-deterministic search problems
- Total utility is one of:
  \[
  \sum_t r_t \quad \text{or} \quad \sum_t \gamma^t r_t
  \]
Solving MDPs

- In deterministic single-agent search problem, want an optimal plan, or sequence of actions, from start to a goal
- In an MDP, we want an optimal policy \( \pi(s) \)
- A policy gives an action for each state
- Optimal policy maximizes expected if followed
- Defines a reflex agent

Optimal policy when \( R(s, a, s') = -0.04 \) for all non-terminals \( s \)
Example Optimal Policies

R(s) = -0.01

R(s) = -0.03

R(s) = -0.4

R(s) = -2.0
Optimal Utilities

- Fundamental operation: compute the optimal utilities of states $s$ (all at once)

- Why? Optimal values define optimal policies!

- Define the utility of a state $s$: $V^*(s) = \text{expected return starting in } s \text{ and acting optimally}$

- Define the utility of a q-state $(s,a)$: $Q^*(s,a) = \text{expected return starting in } s, \text{ taking action } a \text{ and thereafter acting optimally}$

- Define the optimal policy: $\pi^*(s) = \text{optimal action from state } s$
The Bellman Equations

Definition of utility leads to a simple one-step lookahead relationship amongst optimal utility values:

Optimal rewards = maximize over first action and then follow optimal policy

Formally:

\[ V^*(s) = \max_a Q^*(s, a) \]

\[ Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

\[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]
Solving MDPs / memoized recursion

- Recurrences:

\[ V_0^*(s) = 0 \]

\[ V_i^*(s) = \max_a Q_i^*(s, a) \]

\[ Q_i^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_{i-1}^*(s') \right] \]

\[ \pi_i(s) = \arg \max_a Q_i^*(s, a) \]

- Cache all function call results so you never repeat work
- What happened to the evaluation function?
Q-Value Iteration

➢ Value iteration: iterate approx optimal values
  ➢ Start with $V_0^*(s) = 0$, which we know is right (why?)
  ➢ Given $V_i^*$, calculate the values for all states for depth $i+1$:

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right]$$

➢ But Q-values are more useful!
  ➢ Start with $Q_0^*(s, a) = 0$, which we know is right (why?)
  ➢ Given $Q_i^*$, calculate the q-values for all q-states for depth $i+1$:

$$Q_{i+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_i(s', a') \right]$$
RL = Unknown MDPs

- If we knew the MDP (i.e., the reward function and transition function):
  - Value iteration leads to optimal values
  - Will always converge to the truth

- Reinforcement learning is what we do when we do not know the MDP
  - All we observe is a trajectory
    - \((s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, \ldots)\)

- Many algorithms exist for this problem; see Sutton+Barto's excellent book!
Q-Learning

- Learn $Q^*(s,a)$ values
  - Receive a sample $(s,a,s',r)$
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:
    \[
    Q^*(s,a) = \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]
    \]
  - Incorporate the new estimate into a running average:
    \[
    \text{sample} = R(s,a,s') + \gamma \max_{a'} Q(s',a')
    \]
    \[
    Q(s,a) \leftarrow (1 - \alpha)Q(s,a) + (\alpha) [\text{sample}]
    \]
Exploration / Exploitation

- Several schemes for forcing exploration
  - Simplest: random actions (\(\varepsilon\) greedy)
    - Every time step, flip a coin
    - With probability \(\varepsilon\), act randomly
    - With probability \(1-\varepsilon\), act according to current policy

- Problems with random actions?
  - You do explore the space, but keep thrashing around once learning is done
  - One solution: lower \(\varepsilon\) over time
  - Another solution: exploration functions
Q-Learning

➢ In realistic situations, we cannot possibly learn about every single state!
   ➢ Too many states to visit them all in training
   ➢ Too many states to hold the q-tables in memory

➢ Instead, we want to generalize:
   ➢ Learn about some small number of training states from experience
   ➢ Generalize that experience to new, similar states:

   \[
   Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a)
   \]

➢ Very simple stochastic updates:

   \[
   Q(s, a) \leftarrow Q(s, a) + \alpha [\text{error}]
   \]

   \[
   w_i \leftarrow w_i + \alpha [\text{error}] f_i(s, a)
   \]
Inverse Reinforcement Learning

(aka Inverse Optimal Control)
Inverse RL: Task

- **Given:**
  - measurements of an agent's behavior over time, in a variety of circumstances
  - if needed, measurements of the sensory inputs to that agent
  - if available, a model of the environment.

- **Determine:** the reward function being optimized

- Proposed by [Kalman68]
- First solution, by [Boyd94]
Why inverse RL?

- Computational models for animal learning
  - “In examining animal and human behavior we must consider the reward function as an unknown to be ascertained through empirical investigation.”

- Agent construction
  - “An agent designer [...] may only have a very rough idea of the reward function whose optimization would generate 'desirable' behavior.”
  - eg., “Driving well”

- Multi-agent systems and mechanism design
  - learning opponents’ reward functions that guide their actions to devise strategies against them
IRL from Sample Trajectories

- Optimal policy available through sampling (e.g., driving a car)

- Want to find Reward function that makes this policy look as good as possible

- Write $R_w(s) = w \phi(s)$ so the reward is linear

and $V^\pi_w(s_0)$ be the value of the starting state

$$\max_w \sum_{k=1}^K f\left(V^\pi^*_w(s_0) - V^\pi_k(s_0)\right)$$

How good does the “optimal policy” look?

How good does the some other policy look?

Warning: need to be careful to avoid trivial solutions!
Apprenticeship Learning via IRL

➢ For $t = 1,2,...$

➢ Inverse RL step:
  Estimate expert’s reward function $R(s) = w^T \phi(s)$ such that under $R(s)$ the expert performs better than all previously found policies $\{\pi_i\}$.

➢ RL step:
  Compute optimal policy $\pi_t$ for the estimated reward $w$
Car Driving Experiment

➢ No explicit reward function at all!
➢ Expert demonstrates proper policy via 2 min. of driving time on simulator (1200 data points).
➢ 5 different “driver types” tried.
➢ Features: which lane the car is in, distance to closest car in current lane.
➢ Algorithm run for 30 iterations, policy hand-picked.
➢ Movie Time! (Expert left, IRL right)
“Nice” driver
“Evil” driver
Maxent IRL

Distribution over trajectories:

\[ P(\zeta) \]

Match the reward of observed behavior:

\[ \sum_\zeta P(\zeta) f_\zeta = f_{\text{dem}} \]

Maximizing the \textit{causal entropy} over trajectories given stochastic outcomes:

\[ \max H(P(\zeta) \| O) \]

(Condition on random uncontrolled outcomes, but only \textbf{after} they happen)

As uniform as possible
Data collection

Length
Speed
Road Type
Lanes

Accidents
Construction
Congestion
Time of day

25 Taxi Drivers

Over 100,000 miles

Ziebart+al, AAAI08
Predicting destinations....
Planning as structured prediction

[Images of training data and learned models for different modes]
Maximum margin planning

- Let $\mu(s,a)$ denote the probability of reaching q-state $(s,a)$ under current model $w$

\[
\begin{align*}
\max_w & \quad \text{margin} \quad \text{s.t.} \quad \text{planner run with } w \text{ yields human output} \\
\min_w & \quad \frac{1}{2} \|w\|^2 \quad \text{s.t.} \quad \mu(s,a)w \cdot \phi(x_n, s, a) - \hat{\mu}(s,a)w \cdot \phi(x_n, s, a) \geq 1 \\
& \quad , \quad \forall n, s, a
\end{align*}
\]

Q-state visitation frequency by human

Q-state visitation frequency by planner

All trajectories, and all q-states
Optimizing MMP

MMP Objective

➢ For n=1..N:
  ▶ Augmented planning:
    Run A* on current (augmented) cost map
to get q-state visitation frequencies \( \mu(s,a) \)

➢ Update: \( w = w + \sum_s \sum_a [\hat{\mu}(s,a) - \mu(s,a)] \phi(x_n, s, a) \)

➢ Shrink: \( w = \left(1 - \frac{1}{CN}\right)w \)
Maximum margin planning movies
Parsing via inverse optimal control

- State space = all partial parse trees over the full sentence labeled “S”
- Actions: take a partial parse and split it anywhere in the middle
- Transitions: obvious
- Terminal states: when there are no actions left
- Reward: parse score at completion
Parsing via inverse optimal control

- Maximum Likelihood
- Projection
- Perceptron
- Apprenticeship Learning
- Maximum Margin
- Maximum Entropy
- Policy Matching

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>65</td>
<td>85</td>
<td>90</td>
</tr>
<tr>
<td>Proj</td>
<td>60</td>
<td>80</td>
<td>95</td>
</tr>
<tr>
<td>Perceptron</td>
<td>70</td>
<td>85</td>
<td>90</td>
</tr>
<tr>
<td>AP</td>
<td>75</td>
<td>90</td>
<td>95</td>
</tr>
</tbody>
</table>

[Neu+Szepevari, MLJ09]
Learning by Demonstration
Integrating search and learning

Input: Le homme mange l' croissant.
Output: The man ate a croissant.

Hyp: The man ate
Cov: Le homme mange l' croissant.

Hyp: The man ate a croissant
Cov: Le homme mange l' croissant.

Hyp: The man ate a fox
Cov: Le homme mange l' croissant.

Hyp: The man ate happy
Cov: Le homme mange l' croissant.

Hyp: The man ate a
Cov: Le homme mange l' croissant.

Classifier 'h'
Reducing search to classification

- Natural chicken and egg problem:
  - Want $h$ to get low expected future loss
  - ... on future decisions made by $h$
  - ... and starting from states visited by $h$

- Iterative solution

Input: Le homme mange l' croissant.
Output: The man ate a croissant.

Hyp: The man ate a croissant
Cov: Le homme mange l' croissant.

Loss = 0

Hyp: The man ate a fox
Cov: Le homme mange l' croissant.

Loss = 1.8

Hyp: The man ate happy
Cov: Le homme mange l' croissant.

Loss = 1.2

Hyp: The man ate a croissant
Cov: Le homme mange l' croissant.

Loss = 0.5

Hyp: The man ate a croissant
Cov: Le homme mange l' croissant.

Loss = 0
Reduction for Structured Prediction

➢ Idea: view structured prediction in light of search

Each step here looks like it could be represented as a weighted multi-class problem.

Can we formalize this idea?

Loss function:
L([N V R], [N V R]) = 0
L([N V R], [N V V]) = 1/3
...

L([N V R], [N V V]) = 1/3
Reducing Structured Prediction

Desired: good *policy* on test data
(i.e., given only input string)

Key Assumption: *Optimal Policy for training data*

Given: input, true output and state;
Return: best successor state

Weak!
How to Learn in Search

**Idea:** Train based only on optimal path (ala MEMM)

**Better Idea:** Train based only on optimal policy, then train based on optimal policy + a little learned policy then train based on optimal policy + a little more learned policy then ... eventually only use learned policy
Translating $D^{SP}$ into Searn($D^{SP}$, loss, $\pi$):

- Draw $x \sim D^{SP}$
- Run $\pi$ on $x$, to get a path
- Pick position uniformly on path
- Generate example with costs given by expected (wrt $\pi$) completion costs for “loss”
Algorithm: Searn-Learn(A, D^{SP}, loss, \pi^*, \beta)

1: Initialize: \pi = \pi^*
2: while not converged do
3: Sample: D \sim \text{Searn}(D^{SP}, \text{loss}, \pi)
4: Learn: h \leftarrow A(D)
5: Update: \pi \leftarrow (1-\beta)\pi + \beta h
6: end while
7: return \pi without \pi^*

Ingredients for Searn:
Input space (X) and output space (Y), data from X
Loss function (loss(y, y')) and features
“Optimal” policy \pi^*(x, y_0)

Theorem: \[ L(\pi) \leq L(\pi^*) + \frac{2 \text{avg(loss)} T \ln T}{T} + c \frac{(1+\ln T)}{T} \]
But what about demonstrations?

➢ What did we assume before?

Key Assumption: **Optimal Policy for training data**

Given: input, true output and state;
Return: best successor state

➢ We can have a *human* (or system) demonstrate, thus giving us an *optimal policy*
3d racing game (TuxKart)

Input:

Resized to 25x19 pixels (1425 features)

Output:

Steering in [-1,1]
DAgger: Dataset Aggregation

- Collect trajectories from expert $\pi^*$
- Dataset $D_0 = \{(s, \pi^*(s)) | s \sim \pi^*\}$
- Train $\pi_1$ on $D_0$
- Collect new trajectories from $\pi_1$
  - But let the expert steer!
- Dataset $D_1 = \{(s, \pi^*(s)) | s \sim \pi_1\}$
- Train $\pi_2$ on $D_0 \cup D_1$

In general:

- $D_n = \{(s, \pi^*(s)) | s \sim \pi_n\}$
- Train $\pi_n$ on $\bigcup_{i<n} D_i$

If $N = T \log T$, $\mathcal{L}(\pi_n) < T \epsilon_N + O(1)$ for some $n$
Experiments: Racing Game

Input:

Resized to 25x19 pixels (1425 features)

Output:

Steering in [-1,1]
Bettete

Average falls per lap

![Graph showing the average falls per lap against the number of training data points. The graph compares different methods: DAgger, SMILe(0.1), and Supervised. The y-axis represents average falls per lap, while the x-axis shows the number of training data points.]
Super Mario Bros.

From Mario AI competition 2009

Input:

Output:

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

Extracted 27K+ binary features from last 4 observations
(14 binary features for every cell)
Training (expert)
Test-time execution (classifier)
Test-time execution (Dagger)
Average distance per stage

![Graph showing average distance per stage](image)
Perceptron vs. LaSO vs. Searn

- Incremental perceptron
- LaSO
- Searn / DAgger

Un-learnable decision
Discussion
Relationship between SP and IRL

- Formally, they're (nearly) the same problem
  - See humans performing some task
  - Define some loss function
  - Try to mimic the humans

- Difference is in philosophy:
  - (I)RL has little notion of beam search or dynamic programming
  - SP doesn't think about separating reward estimation from solving the prediction problem
  - (I)RL has to deal with stochasticity in MDPs
Important Concepts

- Search and loss-augmented search for margin-based methods
- Bold versus local updates for approximate search
- Training on-path versus off-path
- Stochastic versus deterministic worlds
- Q-states / values
- Learning reward functions vs. matching behavior
Hal's Wager

➢ Give me a structured prediction problem where:
  ➢ Annotations are at the lexical level
  ➢ Humans can do the annotation with reasonable agreement
  ➢ You give me a few thousand labeled sentences

➢ Then I can learn reasonably well...
  ➢ ...using one of the algorithms we talked about

➢ Why do I say this?
  ➢ Lots of positive experience
  ➢ I'm an optimist
  ➢ I want your counter-examples!
Open problems

➢ How to do SP when argmax is intractable....
  ➢ Bad: simple algorithms diverge [Kulesza+Pereira, NIPS07]
  ➢ Good: some work well [Finley+Joachims, ICML08]
  ➢ And you can make it fast! [Meshi+al, ICML10]

➢ How to do SP with delayed feedback (credit assignment)
  ➢ Kinda just works sometimes [D, ICML09; Chang+al, ICML10]
  ➢ Generic RL also works [Branavan+al, ACL09; Liang+al, ACL09]

➢ What role does structure actually play?
  ➢ Little: only constraints outputs [Punyakanok+al, IJCAI05]
  ➢ Little: only introduces non-linearities [Liang+al, ICML08]

➢ Role of experts?
  ➢ what if your expert isn't actually optimal?
  ➢ what if you have more than one expert?
  ➢ what if you only have trajectories, not the expert?
Things I have no idea how to solve...

\[ \text{all} : (a \to \text{Bool}) \to [a] \to \text{Bool} \]

Applied to a predicate and a list, returns `True' if all elements of the list satisfy the predicate, and `False' otherwise.

```haskell
%module main:MyPrelude
%data main:MyPrelude.MyList aadj =
    {main:MyPrelude.Nil;
     main:MyPrelude.Cons aadj ((main:MyPrelude.MyList aadj))};
%rec
{main:MyPrelude.myzuall :: %forall tadA . (tadA ->
ghczmprim:GHCziBool.Bool)
->
(main:MyPrelude.MyList tadA) ->
ghczmprim:GHCziBool.Bool =
\ @ tadA
   (padk::tadA -> ghczmprim:GHCziBool.Bool)
 (dsddE::(main:MyPrelude.MyList tadA)) ->
   %case ghczmprim:GHCziBool.Bool dsddE
   %of (wildB1::(main:MyPrelude.MyList tadA))
   {main:MyPrelude.Nil ->
    ghczmprim:GHCziBool.True;
    main:MyPrelude.Cons
    (xadm::tadA) (xsadn::(main:MyPrelude.MyList tadA)) ->
    %case ghczmprim:GHCziBool.Bool (padk xadm)
    %of (wild1Xc::ghczmprim:GHCziBool.False)
    {ghczmprim:GHCziBool.False ->
     ghczmprim:GHCziBool.False;
     ghczmprim:GHCziBool.True ->
     main:MyPrelude.myzuall @ tadA padk xsadn}));
```
(s1) A father had a family of sons who were perpetually quarreling among themselves. (s2) When he failed to heal their disputes by his exhortations, he determined to give them a practical illustration of the evils of disunion; and for this purpose he one day told them to bring him a bundle of sticks. (s3) When they had done so, he placed the faggot into the hands of them to break it in strength, and when the faggot, took the sticks separately, one by one, and again put them in them easily. (s6) “My sons, if you are of one mind, and unite to assist each other, you will be as this faggot, uninjured by all the attempts of your enemies; but if you are divided among yourselves, you will be broken as easily as these sticks.”
Software

➢ Sequence labeling
  ➢ Mallet http://mallet.cs.umass.edu
  ➢ CRF++ http://crfpp.sourceforge.net

➢ Search-based structured prediction
  ➢ LaSO http://hal3.name/TagChunk
  ➢ Searn http://hal3.name/searn

➢ Higher-level “feature template” approaches
  ➢ Alchemy http://alchemy.cs.washington.edu
  ➢ Factorie http://code.google.com/p/factorie
Summary

- Structured prediction is easy if you can do argmax search (esp. loss-augmented!)
- Label-bias can kill you, so iterate (Searn/Dagger)
- Stochastic worlds modeled by MDPs
- IRL is all about learning reward functions
- IRL has fewer assumptions
  - More general
  - Less likely to work on easy problems
- We're a long way from a complete solution
- Hal's wager: we can learn pretty much anything

Thanks! Questions?
References

See also:

http://braque.cc/ShowChannel?handle=P5BVAC34
Stuff we talked about explicitly

Other good stuff