Parse rescoring

Mark Johnson
Brown University

November 2007
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescoring

Example of a feature class

Trees and sptrees
Linear models for parse rescoring

- Charniak $\ell$-best parser supplies parses $C = (x_1, \ldots, x_\ell)$ for each sentence
  - We typically use around $\ell = 50$ parses per sentence
- A feature $f$ is a function that maps a parse $x$ to real number $f(x)$
  - $f = (f_1, \ldots, f_m)$ is vector of features
  - $f(x) = (f_1(x), \ldots, f_m(x))$ is a vector of feature values
- A feature weight vector is a real-valued vector $w = (w_1, \ldots, w_m)$ that associates each feature $f_j$ with a weight $w_j$
- The score $s_w(x)$ of a parse $x$ is:
  \[
s_w(x) = w \cdot f(x) = \sum_{j=1}^{m} w_j f_j(x)
  \]
- The optimal parse $\hat{x} \in C$ is the one with the highest score:
  \[
  \hat{x} = \underset{x \in C}{\text{argmax}} s_w(x)
  \]
- Our goal: choose $f$ and $w$ to make $\hat{x}$ as accurate a parse as possible
What can features be?

- A feature can be *any real-valued function of the parse*
- By convention, $f_0(x)$ is the log probability of parse from Charniak’s parser
- Examples of useful features:
  - The number of times the tree fragment $(S (NP (DT) (NN)) (VP (VB)))$ occurs in the parse tree
  - The number of NPs in the parse tree beginning with a DT and ending with an NNS and followed by a punctuation symbol,
  - The number of nodes on the *right-most branch* of the parse tree
  - The number of VPs with less than 5 non-punctuation words between their right edge and the end of the sentence
- We typically have $m \approx 1,000,000$ features
- I don’t know how to identify the most useful features (if you can think of a good way, let me know!)
Supervised learning of feature weights

- All we know about the parses \( x \) are:
  - their feature vectors \( f(x) \), and
  - how accurate each parse \( x \in C \) is, so we can identify the best parse \( x^* \in C \)
- Choose feature weights \( w \) so that best parse \( x^* \) is optimal parse \( \hat{x} \)

<table>
<thead>
<tr>
<th>Best: ( x^* )</th>
<th>Rest: ( C \setminus {x^*} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0, 0, 0, 1, 2))</td>
<td>((0, 1, 0, 0, 2)) ((1, 0, 0, 0, 2)) ((0, 0, 1, 0, 2))</td>
</tr>
<tr>
<td>((0, 0, 0, 0, 2))</td>
<td>((0, 0, 0, 2, 0)) ((1, 0, 0, 0, 1))</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- The weight vector \( w = (-2, -2, -2, -1, 0) \) correctly classifies this data
- Supervised learning problem: given features and the \( \ell \)-best parses for \( n \) sentences, find \( w \) such that \( \hat{x} = x^* \) as often as possible
- A variety of methods can be used to do this, including:
  - \( \text{MaxEnt} \), which maximizes likelihood of \( P(x^* | C) \) under a log-linear model
  - \( \text{Boosting} \), which maximizes an approximate margin between \( x^* \) and \( \hat{x} \)
  - \( \text{Perceptron} \), which is a fast on-line learning algorithm
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescorer

Example of a feature class

Trees and sptrees
Why are they *Maximum Entropy* models?

- Goal: learn a probability distribution \( \hat{P} \) as close as possible to distribution \( P \) that generated training data \( D \).

- But what does “as close as possible” mean?
  - Require \( \hat{P} \) to have *same distribution of features* as \( D \)
  - As size of data \(|D| \to \infty\), feature distribution in \( D \) will approach feature distribution in \( P \)
  - so distribution of features in \( \hat{P} \) will approach distribution of features in \( P \)

- But there are many \( \hat{P} \) that have same feature distributions as \( D \). Which one should we choose?
  - The *entropy* measures the *amount of information* in a distribution
  - Higher entropy \( \Rightarrow \) less information
  - Choose the \( \hat{P} \) with *maximum entropy* that whose feature distributions agree with \( D \)
    \( \Rightarrow \) \( \hat{P} \) has the least extraneous information possible
Maximum Entropy models

- A conditional Maximum Entropy model $P_w$ consists of a vector of features $f$ and a vector of feature weights $w$.
- The probability $P_w(x|C)$ of an outcome $x \in C$ is:

$$P_w(x|C) = \frac{1}{Z_w(C)} \exp(s_w(x))$$

$$= \frac{1}{Z_w(C)} \exp \left( \sum_{j=1}^{m} w_j f_j(x) \right), \text{ where:}$$

$$Z_w(C) = \sum_{x' \in C} \exp(s_w(x'))$$

- $Z_w(C)$ is a normalization constant called the partition function
Feature dependence $\Rightarrow$ MaxEnt models

- Many probabilistic models assume that features are independently distributed (e.g., Hidden Markov Models, Probabilistic Context-Free Grammars)
  - Estimating feature weights is simple (relative frequency)
- But features in most linguistic theories interact in complex ways
  - Long-distance and local dependencies in syntax
  - Many markedness and faithfulness constraints interact to determine a single syllable’s shape
  - These features are not independently distributed
- MaxEnt models can handle these feature interactions
  - Estimating feature weights of MaxEnt models is more complicated
    - generally requires numerical optimization
A rose by any other name . . .

Like most other good ideas, Maximum Entropy models have been invented many times . . .

- In statistical mechanics (physics) as the Gibbs and Boltzmann distributions
- In probability theory, as Maximum Entropy models, log-linear models, Markov Random Fields and exponential families
- In statistics, as logistic regression
- In neural networks, as Boltzmann machines
A brief history of MaxEnt models in Computational Linguistics

- Logistic regression used in socio-linguistics to model “variable rules” (Sedergren and Sankoff 1974)
- Hinton and Sejnowski (1986) and Smolensky (1986) introduce the Boltzmann machine for neural networks
- Berger, Dell Pietra and Della Pietra (1996) propose Maximum Entropy Models for language models with non-independent features
- Abney (1997) proposes MaxEnt models for probabilistic syntactic grammars with non-independent features
- (Johnson, Geman, Canon, Chi and Riezler (1999) propose conditional estimation of regularized MaxEnt models)
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescoring

Example of a feature class

Trees and sptrees
Finding the MaxEnt model by maximizing likelihood

- Can prove that the MaxEnt model $P_{\hat{w}}$ for features $f$ and data $D = \{(C_1, x_1), \ldots, (C_n, x_n)\}$ is:

$$P_{\hat{w}}(x \mid C) = \frac{1}{Z_{\hat{w}}(C)} \exp\left(s_{\hat{w}}(x)\right) = \frac{1}{Z_{\hat{w}}(C)} \exp \sum_{j=1}^{m} \hat{w}_j f_j(x)$$

where $\hat{w}$ maximizes the likelihood $L_D(w)$ of the data $D$

$$\hat{w} = \arg\max_w L_D(w) = \arg\max_w \prod_{i=1}^{n} P_w(x_i \mid C_i)$$

I.e., choose $\hat{w}$ to make the winners $x_i$ as likely as possible compared to losers $C_i \setminus \{x_i\}$
Finding the feature weights $\hat{\mathbf{w}}$

- Standard method: use a gradient-based numerical optimizer to minimize the negative log likelihood $-\log L_D(\mathbf{w})$
  (Limited memory variable metric optimizers seem to be best)

\[-\log L_D(\mathbf{w}) = \sum_{i=1}^{n} -\log P_{\mathbf{w}}(x_i | C_i)\]
\[= \sum_{i=1}^{n} \left( \log Z_{\mathbf{w}}(C_i) - \sum_{j=1}^{m} w_j f_j(x_i) \right)\]

\[
\frac{\partial}{\partial w_j} -\log L_D(\mathbf{w}) = \sum_{j=1}^{n} \left( E_{\mathbf{w}}[f_j|C_i] - f_j(x_i) \right), \text{ where:}
\]
\[
E_{\mathbf{w}}[f_j|C_i] = \sum_{x' \in C_i} f_j(x') P_{\mathbf{w}}(x')
\]

- I.e., find feature weights $\hat{\mathbf{w}}$ that make the model’s distribution of features over $C_i$ equal distribution of features in winners $x_i$
Finding the optimal feature weights $\hat{w}$

- Numerically optimizing likelihood involves calculating $-\log L_D(w)$ and its derivatives.
- Need to calculate $Z_w(C_i)$ and $E_w[f_j|C_i]$, which are sums over $C_i$, the set of candidates for example $i$.
- If $C_i$ can be infinite:
  - depending on $f$ and $C$, might be possible to explicitly calculate $Z_w(C_i)$ and $E_w[f_j|C_i]$, or
  - may be able to approximate $Z_w(C_i)$ and $E_w[f_j|C_i]$, especially if $P_w(x|C)$ is concentrated on few $x$.
- Aside: using MaxEnt for unsupervised learning requires $Z_w$ and $E_w[f_j]$, but these are typically hard to compute.
- If feature weights $w_j$ should be negative (e.g., OT constraint violations can only “hurt” a candidate), then replace optimizer with a numerical optimizer/constraint solver (e.g., TAO package from Argonne labs).
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescoring

Example of a feature class

Trees and sptrees
Why regularize?

- MaxEnt selects $\hat{\mathbf{w}}$ so that winners are as likely as possible
- Might not want to do this with *noisy training data*
- *Pseudo-maximal or minimal* features cause numerical problems
  - A feature $f_j$ is *pseudo-minimal* iff for all $i = 1, \ldots, n$ and $x' \in C_i$,
    $f_j(x_i) \leq f(x')$ (i.e., $f_j(x_i)$ is the minimum value $f_j$ has in $C_i$)
  - If $f_j$ is *pseudo-minimal*, then $\hat{w}_j = -\infty$
- Example: Features 1, 2 and 3 are pseudo-minimal below:

<table>
<thead>
<tr>
<th>Winner $x_i$</th>
<th>Losers $C_i \setminus {x_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, 0, 1, 2)</td>
<td>(0, 1, 0, 0, 2) (1, 0, 0, 0, 2) (0, 0, 1, 0, 2)</td>
</tr>
<tr>
<td>(0, 0, 0, 0, 2)</td>
<td>(0, 0, 0, 2, 0) (1, 0, 0, 0, 1)</td>
</tr>
</tbody>
</table>

... ... 

so we can make (some of) the losers have arbitrarily low probability by setting the corresponding feature weights as negative as possible
Regularization, or “keep it simple”

- Slavishly optimizing likelihood leads to over-fitting or numerical problems
  \[ \Rightarrow \] Regularize or smooth, i.e., try to find a “good” \( \hat{\mathbf{w}} \) that is “not too complex”
- Minimize the \textit{penalized negative log likelihood}

\[
\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} - \log L_D(\mathbf{w}) + \alpha \sum_{j=1}^{m} |w_j|^k
\]

where \( \alpha \geq 0 \) is a parameter (often set by \textit{cross-validation on held-out training data}) controlling amount of regularization
Aside: Regularizers as Bayesian priors

- Bayes inversion formula

\[
P(w \mid D) \propto P(D \mid w) P(w)
\]

or in terms of log probabilities:

\[
- \log P(w \mid D) = - \log P(D \mid w) - \log P(w) + c
\]

⇒ The regularized estimate \( \hat{w} \) is also the Bayesian maximum a posteriori (MAP) estimate with prior

\[
P(w) \propto \exp \left( -\alpha \sum_{j=1}^{m} |w_j|^k \right)
\]

- When \( k = 2 \) this is a Gaussian prior
Understanding the effects of the priors

• The log penalty term for a **Gaussian prior** \((k = 2)\) is \(\alpha \sum_j w_j^2\) so its derivative \(2\alpha w_j \rightarrow 0\) as \(w_j \rightarrow 0\)

• Effect of Gaussian prior decreases as \(w_j\) is small

⇒ Gaussian prior prefers all \(w_j\) to be small but not necessarily zero

• The log penalty term for a **1-norm prior** \((k = 1)\) is \(\alpha \sum_j |w_j|\) so its derivative \(\alpha \text{sign}(w_j)\) is \(\alpha\) or \(-\alpha\) unless \(w_j = 0\)

• Effect of 1-norm prior is constant no matter how small \(w_j\) is

⇒ 1-norm prior prefers most \(w_j\) to be zero (**sparse solutions**)  

• My personal view: If most features in your problem are irrelevant, prefer a sparse feature vector.  
  But if most features are noisy and weakly correlated with the solution, prefer a dense feature vector (averaging is the solution to noise).
MaxEnt in syntactic parsing

- MaxEnt model used to pick correct parse from 50 parses produced by Charniak parser
  - $C_i$ is set of 50 parses from Charniak parser, $x_i$ is best parse in $C_i$
  - Charniak parser’s accuracy $\approx 0.898$ (picking tree it likes best)
  - Oracle accuracy is $\approx 0.968$
  - EM-like method for dealing with ties (training data $C_i$ contains several equally good “best parses” for a sentence $i$)

- MaxEnt model uses 1,219,273 features, encoding a wide variety of syntactic information
  - including the Charniak model’s log probability of the tree
  - trained on parse trees for 36,000 sentences
  - prior weight $\alpha$ set by cross-validation (don’t need to be accurate)

- Gaussian prior results in all feature weights non-zero
- L1 prior results in $\approx 25,000$ non-zero feature weights
- Accuracy with both Gaussian and L1 priors $\approx 0.916$ (Andrew and Gao, ICML 2007)
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescoring

Example of a feature class

Trees and sptrees
Stochastic gradient ascent

• MaxEnt: choose $\hat{w}$ to maximize log likelihood
• If $w \neq \hat{w}$ and $\delta$ is sufficiently small, then

$$\log L_D \left( w + \delta \frac{\partial \log L_D(w)}{\partial w} \right) > \log L_D(w)$$

i.e., small steps in direction of derivative increase likelihood

$$\frac{\partial \log L_D(w)}{\partial w_j} = \sum_{j=1}^{n} \left( f_j(x_i) - E_w[f_j | C_i] \right), \text{ where:}$$

$$E_w[f_j | C_i] = \sum_{x' \in C_i} f_j(x') P_w(x')$$

• Gradient ascent optimizes the log likelihood in this manner.
  • It is usually not an efficient optimization method
• Stochastic gradient ascent updates immediately in direction of contribution of training example $i$ to derivative
  • It is a simple and sometimes very efficient method
Perceptron updates as a MaxEnt approx

• Perceptron learning rule: Let $x_i^*$ be the model’s current prediction of the optimal candidate in $C_i$

$$x_i^* = \arg\max_{x' \in C_i} s_w(x')$$

If $x_i^* \neq x_i$, where $x_i$ is the correct candidate in $C_i$, then increment the current weights $w$ with:

$$\delta (f(x_i) - f(x_i^*))$$

• MaxEnt stochastic gradient ascent update:

$$\delta \frac{\partial \log L_D(w)}{\partial w} = \delta (f(x_i) - E_w[f | C_i])$$

If $P_w(x \mid C_i)$ is peaked around $x_i^*$, then $E_w[f \mid C_i] \approx f(x_i^*)$

$\Rightarrow$ The Perceptron rule approximates the MaxEnt stochastic gradient ascent update
Regularization as weight decay

- When we approximate regularized MaxEnt as either Stochastic Gradient Ascent or the Perceptron update, *regularization corresponds to weight decay* (a popular smoothing method for neural networks)

- Contribution of *Gaussian prior* to log likelihood is $-\alpha \sum_j w_j^2$
  so derivative of regularizer is $-2\alpha w_j$
  ⇒ weights *decay proportionally to their current value* each iteration

- Contribution of *1-norm prior* to log likelihood is $-\alpha \sum_j |w_j|$
  so derivative of regularizer is $-\alpha \text{sign}(w_j)$
  ⇒ non-zero weights *decay by a constant amount* each iteration
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescorer

Example of a feature class

Trees and sptrees
Overview of the parse rescorer

Training time

- Penn treebank
- Cross-validating $\ell$-best parser
- $\ell$-best parsed PTB
- extract-features
  - features/$FNN$/
    - train, dev, test
- weight estimator
  - cvlm, gavper

Run time

- Input sentence
- $\ell$-best parses for input sentence
- $\ell$-best parser
- best-parses
- best parse for input sentence

- models/$FNN$/
- $\circ$ENN-weights
- models/$FNN$/
- $\circ$ENN-weights
Pruning useless features with two-pass feature extraction

- There are too many features to store every feature for every parse
- The job of a feature is to distinguish the best parse from the rest of the parses

⇒ Only keep features whose value on the best parse differs from their value on at least one other parse in at least 5 sentences
  - A feature is pseudo-constant iff its value is the same for all parses of each sentence

⇒ Two passes over training data in feature extraction:
  - first pass counts how often each feature distinguishes the best from the rest, and only keeps useful features
  - second pass prints out how often each useful feature appears in each parse
Features are implemented by feature classes

• Groups of related features (e.g., all tree fragments up to a certain size) are implemented by the same feature class.

• A feature class is a C++ class that implements a group of features. It must:
  ▶ define the virtual function identifier, which returns a unique identifying string for this feature class, e.g., TreeFrag
  ▶ define the type Feature, which are the features belonging to this feature class
  ▶ define the function parse_featurecount, which maps each parse to the feature values for each feature in the feature class.

• Features can be any kind of object that:
  ▶ can be written to a single line with <<, and read back in with >>
  ▶ can be hashed with hash<Feature>()
  ▶ if you use (vectors of) the predefined symbols or trees, this is automatically done for you by templates
How feature classes communicate with the program

- The FeatureClassPtrs object is a vector of pointers to the feature classes used by the feature extractor. Its constructor usually calls function that pushes the feature classes to be used.

```cpp
inline void FeatureClassPtrs::features_conlll() {
    push_back(new NLogP());
    push_back(new Rule());
    push_back(new Rule(0, 1));
    push_back(new Rule(0, 0, true));
    push_back(new Rule(0, 0, false, true));
    ...
}
```

- The feature class is called with a parse and feature count map `feat_count`:
  - A feature count map is a “smart” map object that (appears to) map features to non-negative integers.

- If a feature $f$ has value $v$ in the parse, then the feature class should set

$$
\text{feat}_\text{count}[f] = v
$$

Since the value of many features is the number of times the feature appears in the parse tree, it can be easier to increment the feature $\text{++feat}_\text{count}[f]$
Types of feature classes

- A *parse* consists of a *parse tree* together with other information, e.g., Charniak parser probability, etc.

- Features are functions from parses to real numbers, but most features count how often specific configurations occur in the parse tree. The tree-walking code needed to do this is already encapsulated in a `NodeFeatureClass`.

- Feature classes inheriting directly from `FeatureClass` (e.g., `BinnedLogCondP`) define

  ```cpp
  template <typename FeatClass, typename Feat_Count> void
  parse_featurecount(FeatClass& fc, const sp_parse_type& parse,
                      Feat_Count& feat_count)
  ```

- Feature classes inheriting from `NodeFeatureClass` (e.g., `SubjVerbAgr`) define

  ```cpp
  template <typename FeatClass, typename Feat_Count> void
  node_featurecount(FeatClass& fc, const sptree* node,
                    Feat_Count& feat_count)
  ```
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescorer

Example of a feature class

Trees and sptrees
Example feature class: SubjVerbAgr

- Goal: add a feature that will (roughly) capture subject-verb agreement
- Penn POS tags distinguish singular and plural nouns and verbs
- Idea: create a feature consisting of the subject NP’s head’s POS and the VP’s head’s POS.
- Good (NP head POS, VP head POS) combinations will have positive weights, bad combinations will have negative weights (we hope)
class SubjVerbAgr : public NodeFeatureClass {
public:

    // Feature is vector of symbols
    typedef std::vector<symbol> Feature;

    template <typename FeatClass, typename Feat_Count>
    void node_featurecount(FeatClass& fc, const sptree* node, Feat_Count& feat_count);

    virtual const char * identifier() const {
        return "SubjVerbAgr";
    }

    // Macro defines functions that every feature class needs
    SPFEATURES_COMMON_DEFINITIONS;
};
template <typename FeatClass, typename Feat_Count>
void SubjVerbAgr::node_featurecount(FeatClass& fc, const sptree* node,
                                    Feat_Count& feat_count) {
    if ((node->label.cat != S() && node->label.cat != SINV())
        || node->label.syntactic_lexhead == NULL)
        return;
    const sptree* subject = NULL; // subject is last NP before VP
    for (const sptree* child = node->child; child != NULL;
         child = child->next)
        if (child->label.cat == NP())
            subject = child;
        else if (child->label.cat == VP())
            break;
    if (subject == NULL || subject->label.semantic_lexhead == NULL)
        return;
    Feature f;
    f.push_back(subject->label.semantic_lexhead->label.cat);
    f.push_back(node->label.syntactic_lexhead->label.cat);
    ++feat_count[f];
}
Outline

Linear models

Maximum Entropy models

Learning Maximum Entropy models from data

Regularization and Bayesian priors

Relationship to stochastic gradient ascent and Perceptron

Implementation of parse rescorer

Example of a feature class

Trees and sptrees
The representation of trees

- A tree includes a label and a pointer to next and child trees.

- The label is a template class argument to the tree_node class.

- A node’s label must include a category cat field, but it may include other fields as well.

- The labels of sptrees include pointers to syntactic and semantic lexical head nodes, string positions of left and right edges of this node, etc.