

Maximum Entropy Modeling Toolkit for Python and C++

Le Zhang
github.com/lzhang10

December 3, 2021

Contents

1	What is it	4
1.1	License	4
1.2	Known Problem	4
2	Introduction to Maximum Entropy Modeling	5
2.1	The Modeling Problem	5
2.2	Parameter Estimation	6
2.3	Further Reading	6
3	Tutorial	8
3.1	Representing Features	8
3.2	Create a Maxent Model Instance	8
3.3	Adding Events to Model	9
3.4	Training the Model	10
3.5	Using the Model	12
3.6	Case Study: Building a maxent Part-of-Speech Tagger	12
3.6.1	The Tagging Model	12
3.6.2	Feature Selection	13
3.6.3	Training The Model	13
3.6.4	Using The Tagger	15
4	Command Line Utility	18
4.1	The maxent Program	18
4.2	Data Format	18
4.3	Examples	19
5	API Reference	23
5.1	C++ API	23
5.2	maxent Namespace Reference	23
5.2.1	Detailed Description	24

5.2.2	Variable Documentation	24
5.2.2.1	int maxent::verbose = 1	24
5.3	maxent::MaxentModel Class Reference	24
5.3.1	Detailed Description	25
5.3.2	Constructor & Destructor Documentation	25
5.3.2.1	maxent::MaxentModel::MaxentModel ()	25
5.3.3	Member Function Documentation	25
5.3.3.1	void maxent::MaxentModel::add_event (const vector< string > & context, const outcome_type & outcome, size_t count = 1)	25
5.3.3.2	void maxent::MaxentModel::add_event (const context_type & context, const outcome_type & outcome, size_t count = 1)	26
5.3.3.3	template<typename Iterator> void maxent::MaxentModel::add_events (Iterator begin, Iterator end) [inline]	26
5.3.3.4	void maxent::MaxentModel::begin_add_event ()	26
5.3.3.5	void maxent::MaxentModel::end_add_event (size_t cutoff = 1)	26
5.3.3.6	double maxent::MaxentModel::eval (const vector< string > & context, const outcome_type & outcome) const	27
5.3.3.7	double maxent::MaxentModel::eval (const context_type & context, const outcome_type & outcome) const	27
5.3.3.8	void maxent::MaxentModel::eval_all (const vector< string > & context, std::vector< pair< outcome_type, double > > & outcomes, bool sort_result = true) const	27
5.3.3.9	void maxent::MaxentModel::eval_all (const context_type & context, std::vector< pair< outcome_type, double > > & outcomes, bool sort_result = true) const	28
5.3.3.10	void maxent::MaxentModel::load (const string & model)	28
5.3.3.11	MaxentModel::outcome_type maxent::MaxentModel::predict (const vector< string > & context) const	28
5.3.3.12	MaxentModel::outcome_type maxent::MaxentModel::predict (const context_type & context) const	29
5.3.3.13	void maxent::MaxentModel::save (const string & model, bool binary = false) const	29
5.3.3.14	void maxent::MaxentModel::train (size_t iter = 15, const std::string & method = "lbfgs", double sigma = 0.0, double tol = 1E-05)	29
5.4	maxent::Trainer Class Reference	30
5.4.1	Detailed Description	30
5.4.2	Member Function Documentation	30
5.4.2.1	void maxent::Trainer::set_training_data (shared_ptr< MEEEventSpace > events, shared_ptr< ParamsType > params, size_t n_theta, shared_array< double > theta, shared_array< double > sigma, size_t n_outcomes, shared_ptr< MEEEventSpace > heldout_events = shared_ptr<MEEEventSpace>())	30

5.5 Python API	31
6 Acknowledgment	32

Chapter 1

What is it

This package provides a Maximum Entropy Modeling toolkit written in C++ with Python binding. In particular, it includes:

- Conditional Maximum Entropy Model
- L-BFGS Parameter Estimation
- GIS Parameter Estimation
- Gaussian Prior Smoothing
- C++ API
- Python3 Extension module

If you do not know what Maximum Entropy Model (MaxEnt) is, please refer to chapter 2 for a brief introduction or section 2.3 for some recommended papers.

1.1 License

This software grew out of an attempt to port the Java maxent package (<http://maxent.sourceforge.net>) into C++ in the early 2000s. Therefore it follows the same license used by the java maxent package. The library is freeware and is licensed under the LGPL license (see LICENSE file for more detail, or visit <http://www.gnu.org/copyleft/lesser.html>). It is distributed with full source code and documentation.

1.2 Known Problem

Sometimes the L-BFGS training can stop with an error if you use a small Gaussian prior value (-g). It seems this problem only occurs on small training data.

Chapter 2

Introduction to Maximum Entropy Modeling

This section provides a brief introduction to Maximum Entropy Modeling. It is by no means complete, the reader is referred to “further reading” at the end of this section to learn more about this subject.

Maximum Entropy (ME, or maxent for short) model is a general purpose machine learning framework that has been successfully applied in various fields including spatial physics, computer vision, and Natural Language Processing (NLP). This introduction will focus on the application of maxent model to NLP tasks. However, it is straightforward to extend the technique described here to other domains.

2.1 The Modeling Problem

The goal of statistical modeling is to construct a model that best accounts for some training data. More specifically, for a given empirical probability distribution \tilde{p} , we want to build a model p as close to \tilde{p} as possible.

Of course, given a set of training data, there are numerous ways to choose a model p that accounts for the data. It can be shown that the probability distribution of the form 2.1.1 is the one that is closest to \tilde{p} in the sense of Kullback-Leibler divergence, when subjected to a set of feature constraints:

$$p(y | x) = \frac{1}{Z(x)} \exp \left[\sum_{i=1}^k \lambda_i f_i(x, y) \right] \quad (2.1.1)$$

here $p(y | x)$ denotes the conditional probability of predicting an *outcome* y on seeing the *context* x . $f_i(x, y)$'s are feature functions (described in detail later), λ_i 's are the weighting parameters for $f_i(x, y)$'s. k is the number of features and $Z(x)$ is a normalization factor (often called partition function) to ensure that $\sum_y p(y|x) = 1$.

ME model represents evidence with binary functions¹ known as *contextual predicates* in the form:

$$f_{cp,y'}(x, y) = \begin{cases} 1 & \text{if } y = y' \text{ and } cp(x) = \text{true} \\ 0 & \text{otherwise} \end{cases} \quad (2.1.2)$$

¹Actually, ME model can have non-negative integer or real feature values. We restrict our discussion to binary value feature here, which is the most common feature type used in NLP. This toolkit fully supports real feature values.

where cp is the *contextual predicate* that maps a pair of *outcome* y and *context* x to $\{true, false\}$.

The modeler can choose arbitrary feature functions in order to reflect the characteristic of the problem domain as faithfully as possible. The ability of freely incorporating various problem-specific knowledge in terms of feature functions gives ME models the obvious advantage over other learn paradigms, which often suffer from strong feature independence assumption (such as naive bayes classifier).

For instance, in part-of-speech tagging, a process that assigns part-of-speech tags to words in a sentence, a useful feature may be:

$$f_{previous_tag_is_DETERMINER,NOUN}(x,y) = \begin{cases} 1 & \text{if } y = NOUN \text{ and } previous_tag_is_DETERMINER(x) = true \\ 0 & \text{otherwise} \end{cases}$$

which is *activated* when previous tag is DETERMINER and current word's tag is NOUN.

In Text Categorization task, a feature may look like:

$$f_{document_has_ROMANTIC,love_story}(x,y) = \begin{cases} 1 & \text{if } y = love_story \text{ and } document_contains_ROMANTIC(x) = true \\ 0 & \text{otherwise} \end{cases}$$

which is *activated* when the term ROMANTIC is found in a document labeled as type:love_story.

Once a set of features is chosen by the modeler, we can construct the corresponding maxent model by adding features as constraints to the model and adjust weights of these features. Formally, We require that:

$$E_{\tilde{p}} \langle f_i \rangle = E_p \langle f_i \rangle$$

Where $E_{\tilde{p}} \langle f_i \rangle = \sum_x \tilde{p}(x,y) f_i(x,y)$ is the empirical expectation of feature $f_i(x,y)$ in the training data and $E_p \langle f_i \rangle = \sum_x p(x,y) f_i(x,y)$ is the feature expectation with respect to the model distribution p . Among all the models subjected to these constraints there is one with the Maximum Entropy, usually called the *Maximum Entropy Solution*.

2.2 Parameter Estimation

Given an exponential model with n features and a set of training data (empirical distribution), we need to find the associated real-value weights for each of the n features which maximize the model's log-likelihood:

$$L(p) = \sum_{x,y} \tilde{p}(x,y) \log p(y | x) \quad (2.2.1)$$

Selecting an optimal model subjected to given constraints from the exponential (log-linear) family is not a trivial task. There are two popular iterative scaling algorithms specially designed to estimate parameters of ME models of the form 2.1.1: Generalized Iterative Scaling [Darroch and Ratcliff, 1972] and Improved Iterative Scaling [Della Pietra et al., 1997].

Recently, another general purpose optimize method *Limited-Memory Variable Metric* (L-BFGS for short) method has been found to be especially effective for maxent parameters estimating problem [Malouf, 2003]. L-BFGS is the default parameter estimating method used by this toolkit.

2.3 Further Reading

This section lists some recommended papers for your further reference.

- Maximum Entropy Approach to Natural Language Processing [Berger et al., 1996]

A must read paper on applying maxent technique to Natural Language Processing. This paper describes maxent in detail and presents an Increment Feature Selection algorithm for increasingly construct a maxent model as well as several example in statistical Machine Translation.

- Inducing Features of Random Fields [Della Pietra et al., 1997]

Another must read paper on maxent. It deals with a more general frame work: *Random Fields* and proposes an *Improved Iterative Scaling* algorithm for estimating parameters of Random Fields. This paper gives theoretical background to Random Fields (and hence Maxent model). A greedy *Field Induction* method is presented to automatically construct a detail random fields from a set of atomic features. An word morphology application for English is developed.

- Adaptive Statistical Language Modeling: A Maximum Entropy Approach [Rosenfeld, 1996]

This paper applied ME technique to statistical language modeling task. More specifically, it built a conditional Maximum Entropy model that incorporated traditional N-gram, distant N-gram and trigger pair features. Significantly perplexity reduction over baseline trigram model was reported. Later, Rosenfeld and his group proposed a *Whole Sentence Exponential Model* that overcome the computation bottleneck of conditional ME model.

- Maximum Entropy Models For Natural Language Ambiguity Resolution [Ratnaparkhi, 1998]

This dissertation discussed the application of maxent model to various Natural Language Dis-ambiguity tasks in detail. Several problems were attacked within the ME framework: sentence boundary detection, part-of-speech tagging, shallow parsing and text categorization. Comparison with other machine learning technique (Naive Bayes, Transform Based Learning, Decision Tree etc.) are given.

- The Improved Iterative Scaling Algorithm: A Gentle Introduction [Berger, 1997]

This paper describes IIS algorithm in detail. The description is easier to understand than [Della Pietra et al., 1997], which involves more mathematical notations.

- Stochastic Attribute-Value Grammars (Abney, 1997)

Abney applied Improved Iterative Scaling algorithm to parameters estimation of Attribute-Value grammars, which can not be corrected calculated by ERF method (though it works on PCFG). Random Fields is the model of choice here with a general Metropolis-Hasting Sampling on calculating feature expectation under newly constructed model.

- A comparison of algorithms for maximum entropy parameter estimation [Malouf, 2003]

Four iterative parameter estimation algorithms were compared on several NLP tasks. L-BFGS was observed to be the most effective parameter estimation method for Maximum Entropy model, much better than IIS and GIS. [Wallach, 2002] reported similar results on parameter estimation of Conditional Random Fields.

Chapter 3

Tutorial

The purpose of this section is twofold: first, it covers the basic steps required to build and use a Conditional Maximum Entropy Model with this toolkit. Second, it demonstrates the powerfulness of maxent modeling technique by building an English part-of-speech tagger with the Python *maxent* extension.

3.1 Representing Features

Follow the description of [Ratnaparkhi, 1998], the mathematical representation of a feature used in a Conditional Maximum Entropy Model can be written as:

$$f_{cp,y'}(x,y) = \begin{cases} 1 & \text{if } y = y' \text{ and } cp(x) = true \\ 0 & \text{otherwise} \end{cases} \quad (3.1.1)$$

where *cp* is the *contextual predicate* which maps a pair of *outcome* *y* and *context* *x* into $\{true, false\}$.

This kind of math notation must be expressed as features of literal string in order to be used in this toolkit. So a feature in part-of-speech tagger which has the form:

$$f_{previous_tag_is_DETERMINER,NOUN}(x,y) = \begin{cases} 1 & \text{if } y = NOUN \text{ and } previous_tag_is_DETERMINER(x) = true \\ 0 & \text{otherwise} \end{cases} \quad (3.1.2)$$

can be written as a literal string: “tag-1=DETERMINER_NOUN”. You will see more concrete examples in the Case Study section.

3.2 Create a Maxent Model Instance

A *maxent* instance can be created by calling its constructor:
In C++:

```
#include <maxent/maxentmodel.hpp>
using namespace maxent;
MaxentModel m;
```

This will create an instance of MaxentModel class called *m*. Please note that all classes and functions are in the namespace *maxent*. For illustration purpose, the include and using statements will be ignored intentionally for the rest of this tutorial.

In Python:

```
from maxent import MaxentModel
m = MaxentModel()
```

The first statement *import* imports our main class `MaxentModel` from *maxent* module into current scope. The second statement creates an instance of `MaxentModel` class.

3.3 Adding Events to Model

Typically, training data consists of a set of events (samples). Each event has a *context*, an *outcome*, and a *count* indicating how many times this event occurs in training data.

Remember that a *context* is just a group of *context predicates*. Thus an event will have the form:

$$[(predicate_1, predicate_2, \dots, predicate_n), outcome, count]$$

Suppose we want to add the following event to our model:

$$[(predicate_1, predicate_2, predicate_3), outcome1, 1]$$

We need to first create a *context*:¹

In C++:

```
std::vector<std::string> context
context.append("predicate1");
context.append("predicate2");
context.append("predicate3");
. . .
```

In Python:

```
context = ['predicate1', 'predicate2', 'predicate3']
```

Before any event can be added, one must call *begin_add_event()* to inform the model the beginning of training.

In C++:

```
m.begin_add_event();
```

¹It's possible to specify feature value when creating a context:

In C++:

```
std::vector<pair<std::string, float> > context
context.append(make_pair("predicate1", 2.0));
context.append(make_pair("predicate2", 3.0));
context.append(make_pair("predicate3", 4.0));
. . .
```

This is simpler in Python:

```
context = [('predicate1', 2.0), ('predicate2', 3.0), ('predicate3', 4.0)]
```

For illustration purpose, we will only cover binary cases (which is more common). You can find more information on specifying real feature value in the API section.

In Python:

```
m.begin_add_event()
```

Now we are ready to add events: In C++:

```
m.add_event(context, "outcome1", 1);
```

In Python:

```
m.add_event(context, "outcome1", 1)
```

The third argument of *add_event()* is the count of the event and can be ignored if the count is 1.

One can repeatedly call *add_event()* until all events are added to the model.

After adding the last event, *end_add_event()* must be called to inform the model the ending of adding events. In C++:

```
m.end_add_event();
```

In Python:

```
m.end_add_event()
```

Additional arguments for *end_add_event()* are discussed in the API Reference.

In addition to this manual, you can get online help from a Python interpreter using the `help` command:

```
>>> help(MaxentModel)           # get help for all available functions
>>> help(MaxentModel.predict)  # get help for a specific function
```

3.4 Training the Model

Train a Maximum Entropy Model is relatively easy. Here are some examples:

For C++ and Python:

```
m.train(); // train the model with default training method
m.train(30, "lbfgs"); // train the model with 30 iterations of L-BFGS method
m.train(100, "gis", 2); // train the model with 100 iterations of GIS method
and apply Gaussian Prior smoothing with a global variance of 2
m.train(30, "lbfgs", 2, 1E-03); // set terminate tolerance to 1E-03
```

The training methods can be either “gis” or “lbfgs” (default). The Gaussian prior σ^2 is used to regularize the model by seeking a Maximum a Posteriori (MAP) solution.

Also, if `m.verbose` is set to 1 (default is 0)², training progress will be printed to stdout. So you will see something like this on your screen:

²The verbose flag can be turned on by setting `maxent.verbose = 1` in C++, and using `maxent.set_verbose(1)` in python.)

```

Total 125997 training events added
Total 0 heldout events added
Reducing events (cutoff is 1)...
Reduced to 65232 training events

Starting L-BFGS iterations...
Number of Predicates: 5827
Number of Outcomes: 34
Number of Parameters: 8202
Number of Corrections: 5
Tolerance: 1.000000E-05
Gaussian Penalty: on
Optimized version
iter  eval      log-likelihood  training accuracy  heldout accuracy
=====
 0    1-3.526361E+00  0.008%           N/A
 0    1-3.387460E+00  40.380%          N/A
 1    3-2.907289E+00  40.380%          N/A
 2    4-2.266155E+00  44.352%          N/A
 3    5-2.112264E+00  47.233%          N/A
 4    6-1.946646E+00  51.902%          N/A
 5    7-1.832639E+00  52.944%          N/A
 6    8-1.718746E+00  53.109%          N/A
 7    9-1.612014E+00  56.934%          N/A
 8   10-1.467009E+00  62.744%          N/A
 9   11-1.346299E+00  65.729%          N/A
10   12-1.265980E+00  67.696%          N/A
11   13-1.203896E+00  69.463%          N/A
12   14-1.150394E+00  71.434%          N/A
13   15-1.081878E+00  71.901%          N/A
14   16-1.069843E+00  70.638%          N/A
15   17-9.904556E-01  76.113%          N/A

Maximum numbers of 15 iterations reached in 183.195 seconds
Highest log-likelihood: -9.904556E-01

```

You can save a trained model to a file and load it back later: In C++ and Python:

```

m.save("new_model");
m.load("new_model");

```

A file named `new_model` will be created. The model contains the definition of context predicates, outcomes, mapping between features and feature ids and the optimal parameter weight for each feature.

If the optional parameter `binary` is true and the library is compiled with zlib support, a compressed binary model file will be saved which is both faster to load and smaller than a plain text model. The format of model file will be detected automatically during loading:

```

m.save("new_model", true); //save a (compressed) binary model
m.load("new_model");      //load it from disk

```

<i>Word:</i>	the	stories	about	well-heeled	communities	and	developers
<i>Tag:</i>	DT	NNS	IN	JJ	NNS	CC	NNS
<i>Position:</i>	1	2	3	4	5	6	7

3.5 Using the Model

The use of the model is straightforward. The *eval()* function will return the probability $p(y|x)$ of an outcome y given some context x : In C++:

```
m.eval(context, outcome);
```

eval_all() is useful if we want to get the whole conditional distribution for a given context: In C++:

```
std::vector<pair<std::string, double> > probs;
m.eval_all(context, probs);
```

eval_all() will put the probability distribution into the vector `probs`. The items in `probs` are the outcome labels paired with their corresponding probabilities. If the third parameter *sort_result* is true (default) *eval_all()* will automatically sort the output distribution in descendant order: the first item will have the highest probability in the distribution.

The Python binding has two slightly different `eval()` methods. The first is `eval()` which returns the most probable class label for a given context:

```
label = m.eval(context)
```

The second is `eval_all()`, which returns the whole conditional distributions in a list of (label, probability) pairs. This equals to the C++ `eval()` function:

```
result = m.eval_all(context)
```

Please consult the API Reference for a more detailed explanation of each class and function.

3.6 Case Study: Building a maxent Part-of-Speech Tagger

This section discusses the steps involved in building a Part-of-Speech (POS) tagger for English in detail. A faithful implementation of the tagger described in [Ratnaparkhi, 1998] will be constructed with this toolkit in Python language. When trained on 00-18 sections and tested on 19-24 sections of Wall Street Journal corpus, the final tagger achieves an accuracy of more than 96%.

3.6.1 The Tagging Model

The task of POS tag assignment is to assign correct POS tags to a word stream (typically a sentence). The following table lists a word sequence and its corresponding tags (taken from [Ratnaparkhi, 1998]):

To attack this problem with the Maximum Entropy Model, we can build a conditional model that calculates the probability of a tag y , given some contextual information x :

$$p(y|x) = \frac{1}{Z(x)} \exp \left[\sum_{i=1}^k \lambda_i f_i(x, y) \right]$$

Thus the possibility of a tag sequence $\{t_1, t_2, \dots, t_n\}$ over a sentence $\{w_1, w_2, \dots, w_n\}$ can be represented as the product of each $p(y|x)$ with the assumption that the probability of each tag y depends only on a limited context information x :

$$p(t_1, t_2, \dots, t_n | w_1, w_2, \dots, w_n) \approx \prod_{i=1}^n p(y_i | x_i)$$

Given a sentence $\{w_1, w_2, \dots, w_n\}$ we can generate the K best tag sequence candidates up to that point in the search and finally select the highest candidate as our tagging result.

3.6.2 Feature Selection

Following [Ratnaparkhi, 1998], we select features used in the tagging model by applying a set of feature templates to the training data.

Condition	Contextual Predicates
w_i is not rare	$w_i = X$
w_i is rare	X is prefix of w_i , $ X \leq 4$
	X is suffix of w_i , $ X \leq 4$
	X contains number
	X contains uppercase character
	X contains hyphen
$\forall w_i$	$t_{i-1} = X$
	$t_{i-w} t_{i-1} = XY$
	$w_{i-1} = X$
	$w_{i-2} = X$
	$w_{i+1} = X$
	$w_{i+2} = X$

Please note that if a word is rare (occurs less than 5 times in the training set (WSJ corpus 00-18)) several additional contextual predicates are used to help predict the tag based on the word's form. A useful feature might be:

$$f(x, y) = \begin{cases} 1 & \text{if } y = \text{VBG} \text{ and } \text{current_suffix_is_ing}(x) = \text{true} \\ 0 & \text{otherwise} \end{cases}$$

and is represented as a literal string: "suffix=ing_VBG".

Here is a list of some features gathered from training data (WSJ corpus):

Only features occur more than 10 times are preserved. Features for rare words are selected with a frequency cutoff of 5.

3.6.3 Training The Model

Once the feature set is defined, it is easy to train a maxent tagging model with this toolkit.

First, we need to create a MaxentModel instance and add events to it:

```
from maxent import MaxentModel
m = MaxentModel()
m.begin_add_event()
m.add_event("suffix=ing", "VBG", 1)
...
m.end_add_event()
```

curword=years
tag-1=CD
word-2=,
tag-1,2=,CD
word+1=old
curword=old
word-1=years
tag-1=NNS
tag-1,2=CD,NNS
word+2=will
word-1=old
prefix=E
prefix=El
suffix=r
suffix=er
suffix=ier

Next, let's call L-BFGS training routine to train a maxent model with 100 iterations:

```
m.train(100, "lbfgs")
```

```
Total 125997 training events added
Total 0 heldout events added
Reducing events (cutoff is 1)...
Reduced to 65232 training events
```

```
Starting L-BFGS iterations...
```

```
Number of Predicates: 5827
```

```
Number of Outcomes: 34
```

```
Number of Parameters: 8202
```

```
Number of Corrections: 5
```

```
Tolerance: 1.000000E-05
```

```
Gaussian Penalty: on
```

```
Optimized version
```

```
iter eval log-likelihood training accuracy heldout accuracy
```

```
=====
```

0	1-3.526361E+00	0.008%	N/A
0	1-3.387460E+00	40.380%	N/A
1	3-2.907289E+00	40.380%	N/A
2	4-2.266155E+00	44.352%	N/A
3	5-2.112264E+00	47.233%	N/A
4	6-1.946646E+00	51.902%	N/A
5	7-1.832639E+00	52.944%	N/A
6	8-1.718746E+00	53.109%	N/A
7	9-1.612014E+00	56.934%	N/A
8	10-1.467009E+00	62.744%	N/A
9	11-1.346299E+00	65.729%	N/A
10	12-1.265980E+00	67.696%	N/A
11	13-1.203896E+00	69.463%	N/A
12	14-1.150394E+00	71.434%	N/A
13	15-1.081878E+00	71.901%	N/A
14	16-1.069843E+00	70.638%	N/A
15	17-9.904556E-01	76.113%	N/A

Maximum numbers of 15 iterations reached in 183.195 seconds
 Highest log-likelihood: -9.904556E-01

After training finishes, save the model to a file:

```
m.save("tagger");
```

This will create a file called `tagger` on disk.

3.6.4 Using The Tagger

A state-of-the-art POS tagger that faithfully implements the search algorithm described in [Ratnaparkhi, 1998], page 43 is included in the toolkit under `example/postagger/` directory.

When trained on 00-18 sections of WSJ corpus and tested on 19-24 sections, this tagger boasts a word accuracy of 97.31% on known words and 87.39% on unknown words with a sentence accuracy of 57.95% and an overall 96.64% word accuracy.

The main executable script for training a tagger model is `postrainer.py`:

```
usage: postrainer.py [options] model

options:
  -h, --help                show this help message and exit
  -fFILE, --file=FILE      train a ME model with data from FILE
  --heldout=FILE           use heldout events from FILE
  --events_out=EVENTS_OUT
                           write training(heldout) events to file
  -mMETHOD, --method=METHOD
                           select training method [lbfgs,gis]
                           [default=lbfgs]
  -cCUTOFF, --cutoff=CUTOFF
                           discard feature with frequency < CUTOFF when training
                           [default=10]
  -rRARE, --rare=RARE      use special feature for rare word with frequency < RARE
                           [default=5]
  -gGAUSSIAN, --gaussian=GAUSSIAN
                           apply Gaussian penalty when training
                           [default=0.0]
  -b, --binary             save events in binary format for fast loading
                           [default=off]
  --ev_cutoff=EV_CUTOFF   discard event with frequency < CUTOFF when training
                           [default=1]
  --iters=ITERS            how many iterations are required for
                           training[default=15]
  --fast                   use psyco to speed up training if possible
  -TTYPE, --type=TYPE     choose context type [default for English]
```

To train 00-18 sections of WSJ corpus (in file `00_18.sent`, one sentence per line) with 100 iterations of L-BFGS, Gaussian coefficient 0.8 and save result model to “`wsj`”:

```
./postrainer.py -f 00_18.sent --iters 100 -g 0.8 wsj
```


The corresponding output during training is sent to stdout:

```

First pass: gather word frequency information
1000 lines
2000 lines
3000 lines
4000 lines
. . .
51000 lines
44520 words found in training data
Saving word frequency information to 00_18.sent.wordfreq

Second pass: gather features and tag dict to be used in tagger
feature cutoff:10
rare word freq:5
1000 lines
2000 lines
3000 lines
4000 lines
. . .
51000 lines
675386 features found
12092 words found in pos dict
Applying cutoff 10 to features
66519 features remained after cutoff
saving features to file wsj.features
Saving tag dict object to wsj.tagdict done
Third pass:training ME model...
1000 lines
2000 lines
3000 lines
4000 lines
. . .
51000 lines
Total 969825 training events added
Total 0 heldout events added
Reducing events (cutoff is 1)...
Reduced to 783427 training events

Starting L-BFGS iterations...
Number of Predicates: 28653
Number of Outcomes: 45
Number of Parameters: 66519
Number of Corrections: 5
Tolerance: 1.000000E-05
Gaussian Penalty: on
Optimized version
iter eval loglikelihood training accuracy heldout accuracy
=====
0 1-3.806662E+00 0.005% N/A
0 1-3.636210E+00 47.771% N/A
1 3-3.015621E+00 47.771% N/A
2 4-2.326449E+00 50.274% N/A
3 5-1.750152E+00 56.182% N/A

```

```

 4      6-1.497112E+00  61.177%   N/A
 5      7-1.373379E+00  64.895%   N/A
 . . .
94     96-1.990776E-01  97.584%   N/A
95     97-1.984520E-01  97.602%   N/A
96     98-1.976996E-01  97.612%   N/A
97     99-1.968460E-01  97.665%   N/A
98    100-1.961286E-01  97.675%   N/A
99    101-1.951691E-01  97.704%   N/A
100   102-1.946537E-01  97.689%   N/A
Maximum numbers of 100 iterations reached in 3817.37 seconds
Highest loglikelihood: -1.946537E-01
training finished
saving tagger model to wsj done

```

A script *maxent_tagger.py* is provided to tag new sentences using previously trained tagger model:

To tag new sentences using wsj model:

```

usage: maxent_tagger.py [options] -m model file

options:
  -h, --help            show this help message and exit
  -oOUTPUT, --output=OUTPUT
                        write tagged result to OUTPUT
  -mMODEL, --model=MODEL
                        load trained model from MODEL
  -t, --test            test mode, include original tag in output
  -v, --verbose
  -q, --quiet
  -TTYPE, --type=TYPE  choose context type

```

The tagging result will be sent to stdout, one sentence per line.

Chapter 4

Command Line Utility

4.1 The maxent Program

For convenience, a command line program `maxent` is provided to carry out some common operations like constructing ME model from a data file, predicting labels of unseen data and performing N-fold cross validation. The source code `src/maxent.cpp` also demonstrates the use of the C++ interface.

4.2 Data Format

`maxent` uses a data format similar to other classifiers:

```
(BNF-like representation)
<event>  .=. <label> <feature>[:<fvalue>] <feature>[:<fvalue>] ...
<feature> .=. string
<fvalue>  .=. float
<label>   .=. string
<line>    .=. <event>
```

Where label and feature are treated as literal “string”s. If a feature¹ is followed with a ‘:’ and a float value (must be non-negative), that number is regarded as feature value. Otherwise, the feature values are assumed to be 1 (binary feature).

Important: You must either specify all feature values or omit all of them. You can not mix them in a data file.

Here’s a sample data file:

```
Outdoor Sunny Happy
Outdoor Sunny Happy Dry
Outdoor Sunny Happy Humid
Outdoor Sunny Sad Dry
Indoor Rainy Happy Humid
Indoor Rainy Happy Dry
Indoor Rainy Sad Dry
. . .
```

¹Stickily speaking, this is context predicate.

Here `Outdoor` and `Indoor` are both labels (outcomes) and all other strings are features (contextual predicates).

If numeric features are present, they are treated as feature values (must be non-negative). This format is compatible with that used by other classifiers such as `libsvm` or `svm-light` where feature must be real value. For example, the following data is taken from a Text Categorization task in `libsvm` file format:

```
+1 4:1.0 6:2 9:7 14:1 20:12 25:1 27:0.37 31:1
+1 4:8 6:91 14:1 20:1 29:1 30:13 31:1 39:1
+1 6:1 9:7 14:1 20:111 24:1 25:1 28:1 29:0.21
-1 6:6 9:1 14:1 23:1 35:1 39:1 46:1 49:1
-1 6:1 49:1 53:1 55:1 80:1 86:1 102:1
```

4.3 Examples

Now assuming we have training data in `train.txt` and testing data in `test.txt`. The following commands illustrate the typical usage of the `maxent` utility:

Create an ME model named `modell` from `train.txt` with 30 iterations of L-BFGS (default)²:

```
maxent train.txt -m modell -i 30
```

If `-b` flag is present then the model file is saved in binary format, which is much faster to load/save than plain text format. The format of model file is automatically detected when loading. You need not to specify `-b` to load a binary model. If the library is compiled with `zlib`, binary model will be saved in gzip compressed format.

```
save a binary model:
maxent train.txt -b -m modell -i 30
```

```
then predict new samples with the newly created model:
maxent -p test.txt -m modell
```

By default, `maxent` will try to read data through `mmap()` system call if available. If this causes problems, `--nommap` option will disable `mmap()` call and use the slower standard I/O instead.

Sometimes we only want to know the testing accuracy of a model trained from a given training data. The train/prediction steps can be combined into a single step without explicitly saving/loading the model file:

```
maxent train.txt test.txt
```

Performing 10-fold cross-validation on `train.txt` and report accuracy:

```
maxent -n 10 train.txt
```

When `-v` option is set, verbose messages will be printed to `stdout`:

²Cygwin users: due to a bug in Cygwin's implantation of `getopt_long()`, all options passed after training filename is discarded. You should specify all options *before* training filename: `maxent -m modell -i 30 train.txt`.

```

maxent train.txt -m modell -v
Total 180 training events added
Total 0 heldout events added
Reducing events (cutoff is 1)...
Reduced to 177 training events

Starting L-BFGS iterations...
Number of Predicates: 9757
Number of Outcomes: 2
Number of Parameters: 11883
Number of Corrections: 5
Tolerance: 1.000000E-05
Gaussian Penalty: off
Optimized version
iter  eval      loglikelihood  training accuracy  heldout accuracy
=====
 0    1-6.931472E-01  38.889%      N/A
 1    2-2.440559E-01  86.111%      N/A
 2    3-1.358731E-01  98.333%      N/A
 3    4-1.058029E-01  98.889%      N/A
 4    5-5.949606E-02  99.444%      N/A
 5    6-3.263124E-02  100.000%     N/A
 6    7-1.506045E-02  100.000%     N/A
 7    8-7.390649E-03  100.000%     N/A
 8    9-3.623262E-03  100.000%     N/A
 9   10-1.661110E-03  100.000%     N/A
10   11-6.882981E-04  100.000%     N/A
11   12-4.081801E-04  100.000%     N/A
12   13-1.907085E-04  100.000%     N/A
13   14-9.775213E-05  100.000%     N/A
14   15-4.831358E-05  100.000%     N/A
15   16-2.423319E-05  100.000%     N/A
16   17-1.666308E-05  100.000%     N/A
17   18-5.449101E-06  100.000%     N/A
18   19-3.448578E-06  100.000%     N/A
19   20-1.600556E-06  100.000%     N/A
20   21-8.334602E-07  100.000%     N/A
21   22-4.137602E-07  100.000%     N/A
Training terminats succesfully in 1.3125 seconds
Highest log-likelihood: -2.068951E-07

```

Predict data in test.txt with modell and save predicated labels (outcome label with highest probability) to output.txt, one label per line:

```
maxent -p -m modell -o output.txt test.txt
```

If the `-detail` flag is given in prediction mode, full distribution will be outputted:

```
<outcome1> <prob1> <outcome2> <prob2> ...
```

```
maxent -p -m modell --detail -o output.txt test.txt
```

It is possible to specify a set of *heldout* data to monitor the performance of model in each iteration of training: the decline of accuracy on heldout data may indicate some *overfitting*.

```

maxent -m model1 train.txt --heldout heldout.txt -v
Loading training events from train.txt
.
Loading heldout events from heldout.txt

Total 1000 training events added
Total 99 heldout events added
Reducing events (cutoff is 1)...
Reduced to 985 training events
Reduced to 99 heldout events

Starting L-BFGS iterations...
Number of Predicates: 24999
Number of Outcomes: 2
Number of Parameters: 30304
Number of Corrections: 5
Tolerance: 1.000000E-05
Gaussian Penalty: off
Optimized version
iter eval loglikelihood training accuracy heldout accuracy
=====
 0 1-6.931472E-01 43.300% 48.485%
 1 2-3.821936E-01 74.400% 71.717%
 2 3-1.723962E-01 95.600% 95.960%
 3 4-1.465401E-01 97.100% 97.980%
 4 5-1.196789E-01 97.600% 97.980%
 5 6-9.371452E-02 97.800% 97.980%
 6 7-6.035709E-02 98.700% 97.980%
 7 8-3.297382E-02 99.700% 98.990%
 8 9-1.777857E-02 99.800% 98.990%
 9 10-9.939370E-03 99.900% 100.000%
 9 10-8.610207E-02 95.900% 94.949%
10 12-8.881104E-03 99.900% 98.990%
11 13-4.874563E-03 99.900% 98.990%
12 14-2.780725E-03 99.900% 98.990%
13 15-1.139578E-03 100.000% 98.990%
14 16-5.539811E-04 100.000% 98.990%
15 17-2.344039E-04 100.000% 98.990%
16 18-1.371225E-04 100.000% 98.990%
Training terminats succesfully in 8.5625 seconds
Highest log-likelihood: -9.583916E-08

```

In this example, it seems performance peaks at iteration 9. Further training actually brings down the accuracy on the heldout data, although the training accuracy continues to increase. Applying a Gaussian prior can help avoid overfitting, just use `-g float` to specify the global Gaussian variance σ^2 .

Finally, `-h` option will bring up a short help screen:

```

maxent -h

Purpose:
  A command line utility to train (test) a maxent model from a file.

```

```
Usage: maxent [OPTIONS]... [FILES]...
-h          --help          Print help and exit
-V          --version       Print version and exit
-v          --verbose      verbose mode (default=off)
-mSTRING    --model=STRING  set model filename
-b          --binary       save model in binary format (default=off)
-oSTRING    --output=STRING prediction output filename
            --detail       output full distribution in prediction mode (default=off)
-iINT       --iter=INT     iterations for training algorithm (default='30')
-gFLOAT     --gaussian=FLOAT set Gaussian prior, disable if 0 (default='0.0')
-cINT       --cutoff=INT   set event cutoff (default='1')
            --heldout=STRING specify heldout data for training
-r          --random       randomizing data in cross validation (default=off)
            --nommap       do not use mmap() to read data (slow) (default=off)

Group: MODE
-p          --predict      prediction mode, default is training mode
-nINT      --cv=INT        N-fold cross-validation mode (default='0')

Group: Parameter Estimate Method
            --lbfgs       use L-BFGS parameter estimation (default)
            --gis         use GIS parameter estimation
```

Chapter 5

API Reference

5.1 C++ API

This section is generated automatically from C++ source code. Unfortunately, sometimes the generated interfaces are overly complex to be useful. I hope the Chapter 3 is enough for most people.

5.2 maxent Namespace Reference

All classes and functions are placed in the namespace maxent.

Compounds

- class **MaxentModel**
This class implements a conditional Maximun Entropy Model.
- class **RandomFieldModel**
This class implement a non-conditional Random Field Model.
- class **RandomFieldTrainer**
RandomFieldTrainer class provides an interface to various training algorithms such as L-BFGS and GIS methods with Gaussian smoothing.
- class **Trainer**
Trainer class provides an abstract interface to various training algorithms.

Variables

- int **verbose** = 1
verbose flag

5.2.1 Detailed Description

All classes and functions are placed in the namespace maxent.

5.2.2 Variable Documentation

5.2.2.1 int maxent::verbose = 1

verbose flag

If set to 1 (default) various verbose information will be printed on stdout. Set this flag to 0 to restrain verbose output.

5.3 maxent::MaxentModel Class Reference

This class implements a conditional Maximun Entropy Model.

```
#include <maxentmodel.hpp>
```

Public Member Functions

- **MaxentModel** ()
Default constructor for MaxentModel.
- void **load** (const string &model)
Load a MaxentModel from a file.
- void **save** (const string &model, bool binary=false) const
Save a MaxentModel to a file.
- double **eval** (const context_type &context, const outcome_type &outcome) const
Evaluates a context, return the conditional probability $p(y|x)$.
- void **eval_all** (const context_type &context, std::vector< pair< outcome_type, double > &outcomes, bool sort_result=true) const
Evaluates a context, return the conditional distribution of the context.
- outcome_type **predict** (const context_type &context) const
Evaluates a context, return the most possible outcome y for given context x.
- void **begin_add_event** ()
Signal the beginning of adding event (the start of training).
- void **add_event** (const context_type &context, const outcome_type &outcome, size_t count=1)
Add an event (context, outcome, count) to model for training later.
- void **add_event** (const vector< string > &context, const outcome_type &outcome, size_t count=1)
Add an event (context, outcome, count) to model for training later.

- double **eval** (const vector< string > &context, const outcome_type &outcome) const
Evaluates a context, return the conditional probability $p(y|x)$.
- void **eval_all** (const vector< string > &context, std::vector< pair< outcome_type, double > &outcomes, bool sort_result=true) const
Evaluates a context, return the conditional distribution of given context.
- outcome_type **predict** (const vector< string > &context) const
Evaluates a context, return the most possible outcome y for given context x .
- template<typename Iterator> void **add_events** (Iterator begin, Iterator end)
Add a set of events indicated by range [begin, end).
- void **end_add_event** (size_t cutoff=1)
Signal the ending of adding events.
- void **train** (size_t iter=15, const std::string &method="lbfgs", double sigma=0.0, double tol=1E-05)
Train a ME model using selected training method.

5.3.1 Detailed Description

This class implements a conditional Maximun Entropy Model.

A conditional Maximun Entropy Model (also called log-linear model)has the form: $p(y|x) = \frac{1}{Z(x)} \exp \left[\sum_{i=1}^k \lambda_i f_i(x, y) \right]$ Where x is a context and y is the outcome tag and $p(y|x)$ is the conditional probability.

Normally the context x is composed of a set of contextual predicates.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 maxent::MaxentModel::MaxentModel ()

Default constructor for **MaxentModel**.

Construct an empty **MaxentModel** instance

5.3.3 Member Function Documentation

5.3.3.1 void maxent::MaxentModel::add_event (const vector< string > & context, const outcome_type & outcome, size_t count = 1)

Add an event (context, outcome, count) to model for training later.

This function is a thin wrapper for the above

See also:

eval(), with all feature values omitted (default to 1.0, which is binary feature case).

`add_event()` should be called after calling

See also:

`begin_add_event()`.

Parameters:

context A list string names of the context predicates occur in the event. The feature value defaults to 1.0 (binary feature)

outcome A `std::string` indicates the outcome label.

count How many times this event occurs in training set. default = 1

5.3.3.2 void maxent::MaxentModel::add_event (const context_type & context, const outcome_type & outcome, size_t count = 1)

Add an event (context, outcome, count) to model for training later.

`add_event()` should be called after calling

See also:

`begin_add_event()`.

Parameters:

context A `std::vector` of `pair<std::string, float>` to indicate the context predicates and their values (must be ≥ 0) occurred in the event.

outcome A `std::string` indicates the outcome label.

count How many times this event occurs in training set. default = 1

5.3.3.3 template<typename Iterator> void maxent::MaxentModel::add_events (Iterator begin, Iterator end) [inline]

Add a set of events indicated by range [begin, end).

the value type of `Iterator` must be `pair<context_type, outcome_type>`

5.3.3.4 void maxent::MaxentModel::begin_add_event ()

Signal the beginning of adding event (the start of training).

This method must be called before adding any event to the model. It informs the model the beginning of training. After the last event is added

See also:

`end_add_event()` must be called to indicate the ending of adding events.

5.3.3.5 void maxent::MaxentModel::end_add_event (size_t cutoff = 1)

Signal the ending of adding events.

This method must be called after adding of the last event to inform the model the ending of the adding events.

Parameters:

cutoff Event cutoff, all events that occurs less than cutoff times will be discussed. Default = 1 (remain all events). Please this is different from the usual sense of *feature cutoff*

5.3.3.6 double maxent::MaxentModel::eval (const vector< string > & context, const outcome_type & outcome) const

Evaluates a context, return the conditional probability $p(y|x)$.

This method calculates the conditional probability $p(y|x)$ for given x and y.

This is a wrapper function for the above

See also:

`eval()`, omitting feature values in paramaters (default to 1.0, treated as binary case)

Parameters:

context A list of string names of the contextual predicates to be evaluated together.

outcome The outcome label for which the conditional probability is calculated.

Returns:

The conditional probability of $p(\text{outcome}|\text{context})$.

See also:

`eval_all()`

5.3.3.7 double maxent::MaxentModel::eval (const context_type & context, const outcome_type & outcome) const

Evaluates a context, return the conditional probability $p(y|x)$.

This method calculates the conditional probability $p(y|x)$ for given x and y.

Parameters:

context A list of pair<string, double> indicates names of the contextual predicates and their values which are to be evaluated together.

outcome The outcome label for which the conditional probability is calculated.

Returns:

The conditional probability of $p(\text{outcome}|\text{context})$.

See also:

`eval_all()`

5.3.3.8 void maxent::MaxentModel::eval_all (const vector< string > & context, std::vector< pair< outcome_type, double > > & outcomes, bool sort_result = true) const

Evaluates a context, return the conditional distribution of given context.

This method calculates the conditional probability $p(y|x)$ for each possible outcome tag y.

This function is a thin warpper for the above

See also:

`eval_all()` feature values are omitted (default to 1.0) for binary feature value case.

Parameters:

context A list of string names of the contextual predicates which are to be evaluated together.

outcomes an array of the outcomes paired with it's probability predicted by the model (the conditional distribution).

sort_result Whether or not the returned outcome array is sorted (larger probability first).
Default is true.

TODO: need optimized for large number of outcomes

See also:

`eval()`

5.3.3.9 `void maxent::MaxentModel::eval_all (const context_type & context, std::vector< pair< outcome_type, double > > & outcomes, bool sort_result = true) const`

Evaluates a context, return the conditional distribution of the context.

This method calculates the conditional probability $p(y|x)$ for each possible outcome tag y .

Parameters:

context A list of pair<string, double> indicates the contextual predicates and their values (must be ≥ 0) which are to be evaluated together.

outcomes An array of the outcomes paired with it's probability predicted by the model (the conditional distribution).

sort_result Whether or not the returned outcome array is sorted (larger probability first).
Default is true.

TODO: need optimized for large number of outcomes

See also:

`eval()`

5.3.3.10 `void maxent::MaxentModel::load (const string & model)`

Load a MaxentModel from a file.

Parameters:

model The name of the model to load

5.3.3.11 `MaxentModel::outcome_type maxent::MaxentModel::predict (const vector< string > & context) const`

Evaluates a context, return the most possible outcome y for given context x .

This function is a thin wrapper for

See also:

`predict()` for binary value case (omitting feature values which default to 1.0)

Parameters:

context A list of String names of the contextual predicates which are to be evaluated together.

Returns:

The most possible outcome label for given context.

See also:

`eval_all()`

5.3.3.12 MaxentModel::outcome_type maxent::MaxentModel::predict (const context_type & context) const

Evaluates a context, return the most possible outcome y for given context x.

This function is a thin wrapper for

See also:

`eval_all()`.

Parameters:

context A list of String names of the contextual predicates which are to be evaluated together.

Returns:

The most possible outcome label for given context.

See also:

`eval_all()`

5.3.3.13 void maxent::MaxentModel::save (const string & model, bool binary = false) const

Save a `MaxentModel` to a file.

Parameters:

model The name of the model to save.

binary If true, the file is saved in binary format, which is usually smaller (if compiled with libz) and much faster to load.

5.3.3.14 void maxent::MaxentModel::train (size_t iter = 15, const std::string & method = "lbfgs", double sigma = 0.0, double tol = 1E-05)

Train a ME model using selected training method.

This is a wrapper function for the underline `Trainer` class. It will create corresponding `Trainer` object to train a Conditional `MaxentModel`. Currently L-BFGS and GIS are implemented.

Parameters:

iter Specify how many iterations are need for iteration methods. Default is 15 iterations.

method The training method to use. Can be "lbfgs" or "gis". L-BFGS is used as default training method.

sigma coefficient in Gaussian Prior smoothing. Default is 0, which turns off Gaussian smoothing.

tol Tolerance for detecting model convergence. A model is regarded as convergence when $|\frac{\text{Log-likelihood}(\theta_2) - \text{Log-likelihood}(\theta_1)}{\text{Log-likelihood}(\theta_1)}| < \text{tol}$. Default tol = 1-E05

The documentation for this class was generated from the following files:

- maxentmodel.hpp
- maxentmodel.cpp

5.4 maxent::Trainer Class Reference

Trainer class provides an abstract interface to various training algorithms.

```
#include <trainer.hpp>
```

Public Member Functions

- `void set_training_data` (shared_ptr< MEEEventSpace > es, shared_ptr< ParamsType > params, size_t n_theta, shared_array< double > theta, shared_array< double > sigma, size_t n_outcomes, shared_ptr< MEEEventSpace > heldout_es=shared_ptr< MEEEventSpace >())
Setting training data directly.

5.4.1 Detailed Description

Trainer class provides an abstract interface to various training algorithms.

Usually you need not use this class explicitly. **MaxentModel::train()** provides a wrapper for the underline **Trainer** instances.

5.4.2 Member Function Documentation

- 5.4.2.1 void maxent::Trainer::set_training_data** (shared_ptr< MEEEventSpace > *events*, shared_ptr< ParamsType > *params*, size_t *n_theta*, shared_array< double > *theta*, shared_array< double > *sigma*, size_t *n_outcomes*, shared_ptr< MEEEventSpace > *heldout_events* = shared_ptr<MEEEventSpace>())

Setting training data directly.

Parameters:

- events* A vector of Event objects consist of training event space.
- params* The internal params.
- n_theta* The number of θ_i parameters.
- sigma* Coefficients in Gaussian Prior Smoothing.
- correct_constant* Correct constant used in GIS algorithm.
- n_outcomes* Number of outcomes.
- heldout_events* A vector of Event objects consist of heldout event space. this parameter can be safely ignored.

The documentation for this class was generated from the following files:

- trainer.hpp
- trainer.cpp

5.5 Python API

This section is under construction. . . .

Chapter 6

Acknowledgment

The author owns his thanks to:

- Developers of <http://maxent.sourceforge.net>, the java implementation of MaxEnt with GIS training algorithm. Actually, this toolkit evolves from an early attempt to port Java maxent to C++.
- Robert Malouf. Dr. Malouf kindly answered my questions on maxent and provides his excellent implementation on four maxent parameter estimation algorithms.
- Jorge Nocedal, for the excellent Fortran L-BFGS implementation.
- And of course all users who provided valuable feedback and bug reports.

Bibliography

- A. Berger. The improved iterative scaling algorithm: A gentle introduction, 1997.
- Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.
- J.N. Darroch and D. Ratcliff. Generalized iterative scaling for log-linear models. *The Annals of Mathematical Statistics*, Vol. 43:pp 1470–1480, 1972.
- Stephen Della Pietra, Vincent J. Della Pietra, and John D. Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393, 1997.
- Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation, 2003.
- A. Ratnaparkhi. Maximum entropy models for natural language ambiguity resolution, 1998.
- R. Rosenfeld. A maximum entropy approach to adaptive statistical language modeling. *Computer, Speech and Language* 1996, 10:187–228, 1996. Longe version: Carnegie Mellon Tech. Rep. CMU-CS-94-138.
- Hanna Wallach. Efficient training of conditional random fields. Master’s thesis, University of Edinburgh, 2002.