CSC 448 Lecture Notes

December 9, 2019

Contents

1 Perceptron for Part-Of-Speech (POS) Tagging 4
- 1.1 POS Tagging 4
- 1.2 Viterbi Decoder 4
- 1.3 Perceptron for POS Tagging 5
  - 1.3.1 Perceptron Background 5
  - 1.3.2 Perceptron Algorithm for POS tagging 6
  - 1.3.3 Interpretation 6
  - 1.3.4 Averaged Perceptron 6

2 Hidden Markov Model 6
- 2.1 Maximizing the tags using dynamic programming 7
- 2.2 Conditions that allow dynamic programming 8
- 2.3 A more general look at the optimization problem 9
- 2.4 Forward/Backward algorithm 9

3 Expectation Maximization (EM) 10
- 3.1 Variational Bayes 14

4 Conditional Random Fields (CRF) 17
- 4.1 CRF Summary 19
- 4.2 Neural CRF 20
- 4.3 Extension: Latent CRF 20
- 4.4 Extension: Loss-Aware CRF 22

5 N-Gram Language Model 23
- 5.1 Alternative 1: Good-Turing 24
- 5.2 Alternative 2: Kneser-Ney 25

6 Deep Learning 26
- 6.1 Initialization (Restricted Boltzman Machine) 27
  - 6.1.1 Contrastive Divergence 28
  - 6.1.2 Stochastic Gradient Descent (SGD) 28
  - 6.1.3 Modeling Real-valued Data 28
- 6.2 Back Propagation 29

7 Fast Decoding for HMMs 29
- 7.1 Fast Decoding 29
  - 7.1.1 Dijkstra's algorithm 29
- 7.2 A* algorithm 30
  - 7.2.1 Admissible heuristic function 30
  - 7.2.2 A* in HMM decoding 30
  - 7.2.3 Alternatives 31

7.3 Multistack decoding 31
7.3.1 TOP K 31
7.3.2 Beam Search 32

8 Structured SVM **32**
8.1 Training 33
8.2 Decoding 33
8.3 Dealing with certain problems 34
8.4 Dual representation for structured SVM 34

9 Context Free Grammars **35**
9.1 Regular Languages 35
9.2 Finite Languages 35
9.3 Probabilistic CFG 36
9.4 Probabilistic CFGs again 36

10 Probabilistic Parsing **37**
10.1 Viterbi Decoding For Parsing 37
10.2 Posterior Decoding For Parsing 39
10.3 Posterior Decoding in HMM 42
10.4 Posterior Decoding in Parsing 42
10.5 Applying dynamic programming 42
10.6 Extracting data from sentences 43
10.7 Lexicalization 44
10.8 Parsing with features 45
10.9 Better Lexicalization 46
10.10Binarization or Markovization 46

11 Charniak Parser **47**
11.1 Coarse-grained Parser 47
11.2 Fine-grained Parser 48
11.3 Reranker 49

12 Dependency Parsing **49**
12.1 Shift Reduce Parsing 50

13 Machine Translation **52**
13.1 Word Alignment 53
13.2 Expectation Maximization 54
13.3 IBM Model 2 56
13.4 HMM Model for Machine Translation 57
13.5 IBM Model 3 58
13.6 IBM Model 4 59

14 Phrase Based Machine Translation **59**
14.1 Symmetrization 59
14.2 Rule Extraction 60
14.3 Decoding 61
14.4 Reordering 64
14.5 Decoder with Distortion Model 65
14.6 Lexicalized Reordering 65

15 Syntaxed Based Machine Translation **67**
15.1 Synchronous Context-Free Grammar(SCFG) 67
15.2 SCFG in Machine Translation 67

16 Decoding with SCFG **68**
16.1 Standard CKY Parsing . . . 69
16.2 Cube Pruning . . . 70

17 Hiero (Hierarchical Phrase-Based Machine Translation) **71**
17.1 SCFG for Hiero . . . 71
17.2 Rule Extraction . . . 71

18 String to Tree Machine Translation **72**
18.1 GHKM Rules Extraction . . . 72
18.2 Decoding . . . 73

19 Tree to String Machine Translation **75**

20 Evaluation and Parameter Tuning in Machine Translation **76**
20.1 BLEU: Bilingual Evaluation Understudy . . . 76
20.2 Tuning Parameters . . . 77
20.2.1 MERT: Minimum Error Rate Training . . . 77
20.2.2 PRO: Pairwise Ranking Optimization . . . 79
20.3 Final Remarks . . . 79

21 Sentiment Analysis **80**

22 Compositional Semantics **81**

23 Lexical Semantics **83**
23.1 Word Senses . . . 83
23.2 Word Sense Disambiguation . . . 83
23.2.1 Bootstrapping . . . 83
23.2.2 Lesk . . . 84
23.2.3 Co-training . . . 85
23.2.4 Using Wikipedia for Word Sense Disambiguation . . . 85
23.3 Word Sense Clustering . . . 85
23.4 DIRT - Discovering Inference Rules from Text . . . 87

1 Perceptron for Part-Of-Speech (POS) Tagging

1.1 POS Tagging

POS Tagging is a standard NLP procedure to give a tag for each word in a sentence. For example, "Time/NN flies/VBZ like/IN an/DT arrow/NN", in which words are accompanied with a POS tag. See the textbook for a full list of the standard set of tags and their meanings.

Features can be extracted for POS tagging, given a word sequence (sentence) and a tagging candidate (tag sequence), denoted by $x_{1...N}$ and $y_{1...N}$ respectively. The feature extraction procedure can be denoted by a function mapping $\Phi : (x, y) \mapsto \mathbb{R}^J$, in which J denotes the size of the feature set. This is better explained by example. Define the indicator function as Eq. (1).

$$I(expr) = \begin{cases} 1 & \text{if } expr \text{ is True} \\ 0 & \text{if } expr \text{ is False} \end{cases} \tag{1}$$

An example of unary feature bin is $\Phi_{100}(X, Y) = \sum_i I(x_i = time \wedge y_i = NN)$ and an example of pairwise feature bin is $\Phi_{200}(x, y) = \sum_i I(y_i = NN \wedge y_{i+1} = VBZ)$. Note that Φ_{100} depends on X, while Φ_{200} doesn't, which are all about design. Also note that *100* and *200* here are just random numbers for notation. The size of feature set J depends on vocabulary size and the set of tags.

For example, let the word sequence $x = (time, flies, like, an, arrow)$ and the tagging candidate $y = (NN, VBZ, IN, DT, NN)$. It is easily seen that $\Phi_{100}(x, y) = 1$, $\Phi_{100}(x, y) = 0$, $\Phi_{200}(x, y) = 1$ and $\Phi_{200}(X, Y) = 0$.

1.2 Viterbi Decoder

Given a set of weights and a word sequence, we define a objective function, a.k.a. score, as a weighed sum of features, as in Equation (2).

$$\operatorname*{argmax}_y w^T \Phi(x, y) \tag{2}$$

Evaluating this by considering all sequence y is impractical because there are N^T values of y for a sequence of length N and tag set of size T.

Assume that all features can be written as a sum over local features ϕ that consider only a sliding window of tag values:

$$\Phi_j(x, y) = \sum_{i=1}^{N} \phi_j(x, y_{i-1}, y_i, i)$$

For convenience, set $y_0 = \text{START}$. The maximization problem of Equation (2) can be rewritten as:

$$\max_{y_1,\ldots,y_N} \sum_{i=1}^{N} w^T \phi(x, y_{i-1}, y_i, i) \tag{3}$$

$$= \max_{y_N, y_{N-1}} w^T \phi(x, y_{N-1}, y_N, N) + \max_{y_{N-2}} w^T \phi(x, y_{N-2}, y_{N-1}, N-1) + \cdots + \max_{y_0} w^T \phi(x, y_0, y_1, 1) \tag{4}$$

The Viterbi decoder builds a table of values δ corresponding to terms of the above equation:

$$\delta(i, t) = \max_{y_1,\ldots,y_{i-1}} \sum_{i'=1}^{i} w^T \phi(x, y_{i'-1}, y'_i, i') \qquad \text{where } y_i = t \tag{5}$$

$$= \max_{t'} \delta(i-1, t') + w^T \phi(x, t', t, i) \tag{6}$$

Based on Equation (6), the dynamic programming algorithm to maximize $w^T \Phi(x, y)$ can be derived as Algorithm (1). Table entry $\delta(i, t)$ denotes the partial solution with tag at position i set to t. After computing δ with Algorithm (1), one can get the optimal sequence $Y_{1...N}$ either by setting a similar table of backpointers to record the choiced used to computer δ, or by runnign backwards through the table.

Algorithm 1: Viterbi Decoding

Require: w (vector of size J), x (sequence of length N)
1: allocate a table δ with size TN
2: **for** $i = 1$ to N **do**
3: **for** $t \in 1 \dots T$ **do**
4: $\delta(i,t) \leftarrow -\infty$
5: **for** $t' \in T$ **do**
6: $\delta(i,t) \leftarrow \max\left\{\delta(i,t), \delta(i-1,t') + w^T\phi(x,t',t,i)\right\}$
7: **end for**
8: **end for**
9: **end for**
10: **return** δ

1.3 Perceptron for POS Tagging

With a learning algorithm, we will be able to train feature weights, which is required by Viterbi decoding. Perceptron algorithm is one way to do so.

1.3.1 Perceptron Background

The standard perceptron algorithm is used for training a binary classifier. Given data points $i = 1 \dots S$, associated with features $f^i \in R^D$ and a binary label $y^i \in \{-1,+1\}$, perceptron algorithm tries to find solution for the optimization problem, $\mathrm{argmax}_{w,b}(\sum_i \mathbb{I}(y^i = t^i))$, in which $t^i = \mathrm{sign}(w^T f^i + b)$ is the predicted label. People usually remove the bias term b, by extending the feature f^i with a constant column, and we denote $\hat{f}^i = (f^i, 1)$ and $\hat{w} = (w, b)$, so that t^i becomes $\mathrm{sign}(\hat{w}^T\hat{f}^i)$. By iteratively adjusting feature weights for wrong decision points, perceptron algorithm, shown in Algorithm 2, guarantees to converge to an w that achieves perfect separation, if the data is linearly separable.

Algorithm 2: Perceptron Algorithm for Binary Classifier

Require: Features $f^{1\dots S} \in R^D$ and binary labels $y^{1\dots S}$
make extended feature matrix $\hat{f} \leftarrow (f, 1)$
initialize $(\hat{w}) \leftarrow \mathbf{0}$
repeat
for $i = 1 \dots S$ **do**
$t^i = \mathrm{sign}(\hat{w}^T\hat{f}^i)$
if $t^i \neq y^i$ **then**
$\hat{w} \leftarrow \hat{w} + y^i\hat{f}^i$
end if
return $\hat{w}$
end for
until Convergence or maximum number of iteration

There are some fundamental problems to use Algorithm 2 for POS tagging. Suppose that for all data instances $i = 1 \dots S$, length of the word sequences are fixed to N, then the number of all possible tagging sequence is T^N, which means naively, we need to construct a T^N-class classifier. To extend binary classifiers, such as the one shown above, to multiple class classifiers, people usually construct multiple one-vs-all binary classifiers. Basically, we'll need C binary classifiers for C-class classifiers. Therefore, it's not tractable to use the binary perceptron algorithm directly in POS tagging problem. An implication for this problem is that we will need exponentially large number of data instances to achieve decent error bound.

1.3.2 Perceptron Algorithm for POS tagging

The structured perceptron algorithm extends the standard perceptron algorithm to handle the large set of possible labels. By iteratively adjusting weights for incorrectly decoded sequences, the algorithm, shown in Algorithm 3, guarantees to converge to an w that achieves perfect separation, if the data is linearly separable.

Algorithm 3: Perceptron Algorithm for POS Tagging

Require: Word sequences $x^{(1)}, \ldots, x^{(S)}$
Require: POS tagging $y^{(1)}, \ldots, y^{(S)}$
Require: the feature extraction function Φ
  initialize $w \leftarrow \mathbf{0}$
  repeat
    for $s = 1 \ldots S$ **do**
      $\hat{y} = \mathrm{argmax}_y\, w^T \Phi(x^{(s)}, y)$ {Using Viterbi Decoder}
      if $\hat{y} \neq y^{(s)}$ **then**
        $\hat{w} \leftarrow \hat{w} + \Phi(x^{(s)}, y^{(s)}) - \Phi(x^{(s)}, \hat{y})$
      end if
    end for
  until Convergence or maximum number of iteration

1.3.3 Interpretation

The essence of perceptron algorithm, both for binary classifier and POS tagging, is to adjust weights when there is error. The way to adjust weights is to make sure the new weights will be better than the old ones, in terms of correcting prediction locally.

1.3.4 Averaged Perceptron

To reduce sensitivity to the point at which training stops, it is common to use the average of the weight vectors computed after each update.

2 Hidden Markov Model

A Hidden Markov Model (HMM) is another technique to identify the parts of speech. The most general problem consists of assigning a probability $P(y_1^N, x_1^N)$ to a sequence of N tags y and words x. This form for the probability to too complicated to model directly, so we approximate it by

$$P(y_1^N, x_1^N) = \prod_{i=1}^{n} P(y_i|y_{i-1}) P(x_i|y_i)$$

$P(y_1^N, x_1^N)$ is the set of all possible sequences that the words and the tags may have. $P(y_i|y_{i-1})$ are the transition probabilities between adjacent tags. $P(x_i|y_i)$ are the emission probability of words conditioned on the tags.

For the first time step at $i = 1$ there is no previous tag at $i = 0$. One can say that at the first time step there is some special tag y_0=START. The is the same as writing the probability as

$$P(y_1) \prod_{i=2}^{n} P(y_i|y_{i-1}) \prod_{i=1}^{n} P(x_i|y_i)$$

where

$$P(y_1) = P(y|START)$$

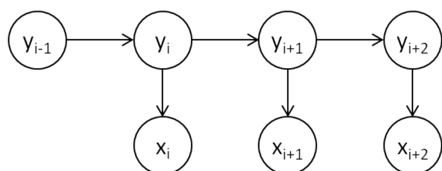
It is easier to implement the HMM using with a START state. The conditional probabilities are expressed in terms as a matrix of probabilities, where we define

$$A_{y_{i-1},y_i} = P(y_i|y_{i-1})$$

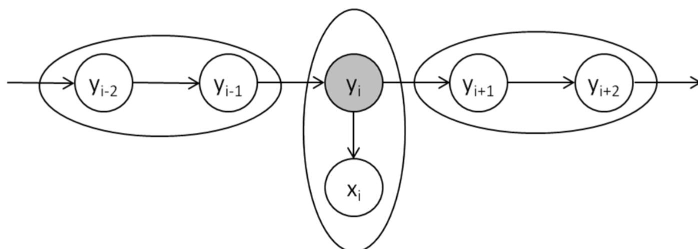
$$O_{y_i,x_i} = P(x_i|y_i)$$

If one chooses to use the formalism of having an initial state $P(y_1)$, it is defined by the vector π_{y1}.

The conditional probabilities can be represented as a Bayes network, where one can read off the dependencies.



If one observes y_i, the graph is separated into 3 pieces, and this separation makes dynamic programming possible.



All the states up to i are conditionally independent of all the states after i given y_i.

$$y_1 \dots y_{i-1} \perp\!\!\!\perp y_{i+1} \dots y_N | y_i$$

2.1 Maximizing the tags using dynamic programming

The best sequence is the one that maximizes the set of tags conditioned on the set of words, which is expressed as

$$y^* = \operatorname*{argmax}_{y_1^N} P(y_1^N|x_1^N)$$

Using the definition of conditional probability, y^* can be expressed as

$$y^* = \operatorname*{argmax}_{y_1^N} \frac{P(y_1^N, x_1^N)}{P(x_1^N)}$$

The denominator only depends on the observations of the words, so it is independent of varying the tags and it can be eliminated from the maximization. Although we ultimately want the list of tags, which is the argmax, we solve for the maximum of the expression to be able to use dynamic programming. Using the HMM approximation, one now solves for

$$y^* = \operatorname*{argmax}_{y_1^N} \prod_{i=1}^{N} P(y_i|y_{i-1})P(x_i|y_i)$$

We now can start to apply the techniques of dynamic programming. We pull the last time step outside of the product while all the other terms remain inside the product. The term outside the product can be maximized independently of the rest of the terms, which is expressed as

$$y^* = \operatorname*{argmax}_{y_{N-1}, y_N} P(y_N | y_{N-1}) P(x_N | y_N) \max_{y_1^N} \prod_{i=1}^{N-1} P(y_i | y_{i-1}) P(x_i | y_i)$$

The dynamic programming strategy can continue to be applied to the remaining product. At the last step, we only need to know the best set of tags up to y_{N-1} to find the complete set of tags up to y_N. The concepts described by the above equations can also be expressed graphically as a lattice of nodes. Each node represents a particular tag, and each column corresponds to a particular word in the sequence. A path through the lattice represents a choice of a particular set of tags. Using dynamic programming, one starts in the first column and selects the edge to a node in the next column based on what gives the maximum score for the path up to this point. The algorithm for the maximization using dynamic programming is given by

$\delta = -\infty$ (initialization)
for $i = 1 \ldots N$
  for $t = 1 \ldots T$
    for $t' = 1 \ldots T$
      $\delta(i, t) = \max \{\delta(i, t), \delta(i-1, t') P(t|t') P(x_i|t)\}$

This algorithm shows that as we progress through the lattice, we either keep the current path or replace it with a better scoring path if found. The running time of the algorithm is $O(NT^2)$. Keeping backpointers allows the path to be reconstructed. The multiplier of $\delta(i, t)$ is the cost function. For the HMM, the cost function is $P(t|t')P(x_i|t)$. For the perceptron algorithm, discussed in the last lecture, we added the following cost function instead of multiplying

$$\delta(i-1, t') + \sum_j w_j f_j(x, y, i)$$

2.2 Conditions that allow dynamic programming

The difference between the perceptron algorithm and the HMM algorithm is the choice of the cost function. In general, dynamic programming is possible because the maximization step can be pushed insude a product. This is possible when the following condition holds

$$\max [ab, ac] = a \max [b, c]$$

which is true only when a is positive. However, for the perceptron algorithm the weights may be negative. However, the perceptron algorithm works because it is additive and the distributive law takes the form $ab + ac = a(b + c)$. In general dynamic programming works if you have two operators and you can push one inside the other. See Table 1 for 3 examples of operations and the space in which dynamic programming holds.

$\otimes$	$\oplus$	
+	max	$\Re$
$\cdot$	max	$\Re^+$
$\cdot$	+	$\Re$

Table 1: Semiring Operations

Suppose you wrote Viterbi for perceptron and now you want to perform an HMM. All you need to do is figure out the features and weights and then solve the same problem. Taking the logarithm of the HMM

probabilities converts a multiplication to an addition. The change in the score when moving from one column of the lattice to the next column of the lattice becomes

$$\delta(i-1,t') + \log P(t|t') + \log(P(x_i|t))$$

Everything should be zero in the weighted sum over the features $\sum_j f_j(x,y,i)$ except

$$I(y_{i-1} = t' \wedge y_i = t) \;\text{ and }\; I(y_i = t \wedge x_i = w)$$

where the weights should be the log of the probability.

2.3 A more general look at the optimization problem

Lets now look at the problem in a more general way. We first run a decoder and the determine the accuracy. If we define the accuracy as how many tags are correct, then the accuracy is given by

$$\frac{1}{N}\sum_{i=1}^{N} I(\hat{y}_i = y_i)$$

where $\hat{y}_i$ is the output and y_i is the ground truth. You want to make this number as high as possible. HMM doesn't necessarily make this metric as high as possible. One can look at the problem another way and obtain the solution to this expression.

$$\max_{\hat{y}_1^N} E_{P(y_1^N|x_1^N)}\left[\frac{1}{N}\sum_{i=1}^{N} I(\hat{y}_i = y_i)\right]$$

Maximizing over the conditional probabilities $P(y_1^N|x_1^N)$ is how HMM appears in the model. This approach to setting up the problm is known by three terms: (1) Maximum Bayes Risk, (2) Maximum expected score, or (3) Minimum Expected Loss. The loss is the negative of the score. Because expectation is linear, we can pull the sum outside

$$\frac{1}{N}\max_{\hat{y}_1^N}\sum_{i=1}^{N} E_{P(y_1^N|x_1^N)}[I(\hat{y}_i = y_i)]$$

Pushing the maximum inside the sum gives

$$\sum_{i=1}^{N}\max_{\hat{y}_1^N} P(\hat{y}_i = y_i|x_1^N)$$

This can be interpretted as maximizing over one $\hat{y}_i$ at a time. In other words, for each position, one figures out which tag has the highest probability. However, there is another figure of merit to optimize, that is, the best sequence of tags that matches the correct sentence. In other words, you don't care what the number correct are, but the most probably sequence. If this is what you are optimizing over, then the expression to give it is

$$\left[\operatorname*{argmax}_{\hat{y}_i} P(\hat{y}_i = y_i|x_1^N)\right]_{i=1}^{N}$$

where the large brackets are defined as concatenation.

2.4 Forward/Backward algorithm

As discussed earlier in the notes, the conditional probability in the above equation can be written as the ratio of the full probability over the probability of the words sequence, and since the word sequence is fixed it does not affect the maximization. We pick a particular i and break the maximization into two parts: one

which marginalizes over all the y's up to the ith term, and one which marginalizes over all the y's after the ith term.

$$\left(\sum_{y_1\cdots y_{i-1}} P(y_1^i, x_1^i)\right)\left(\sum_{y_{i+1}\cdots y_N} P(y_{i+1}^N, x_{i+1}^N \mid y_i)\right)$$
$$= P(x_1^i, y_i)P(x_{i+1}^N|y_i)$$

The first term is known as the forward probability α and the second term as the backwards probability β. Lets start computing the forward probability α. It's a big sum

$$\alpha(i,t) = P(x_1^i, y_i = t) = \sum_{y_{i-1}} P(y_i = t|y_{i-1})P(x_i|y_i) \sum_{y_1\cdots y_{i-2}} \prod_{j=1}^{i-1} P(y_j|y_{j-1})P(x_j|y_j)$$

The second summation is just the forward probability again, but starting at the term $i-1$ rather than the term i. Therefore, one can use dynamic programming to calculate the forward probabilities with the following algorithm

$\alpha = 0$
for $i = 1 \dots N$
    for $t = 1 \dots T$
        for $t' = 1 \dots T$
            $\alpha(i,t) = \alpha(i,t) + P(y_i = t|y_{i-1} = t')P(x_i|y_i)\alpha(i-1,t')$

There are no need for backpointers in this algorithm because we just want the sum, not the path. Using the algorithm we obtain all the α's. The backwards probability is given by $\beta(i,t) = P(x_{i+1}^N|y_i)$. Note the β is a conditional probability. Why is this? The reason is that you have to pay the cost of generating y_i in either the α term or the β term, but not both. The dynamic programming relation and algorithm for calculating β is given by

$$\beta(i,t) = P(x_{i+1}^N|y_i = t) = \sum_{y_{i+1}} P(y_{i+1}|y_i)P(x_{i+1}|y_{i+1})\beta(i+1, y_{i+1})$$

for $i = N \dots 1$
    for $t = 1 \dots T$
        for $t' = 1 \dots T$
            $\beta(i,t) = \beta(i,t) + P(y_{i+1} = t'|y_i = t)P(x_{i+1}|y_{i+1} = t')\beta(i+1,t')$

What is the initialization for the dynamic programming algorithm? For α, if we use the convention of a start state, we initialize at 1. For the β, the initial condition is the probability of nothing occurring. The probability of nothing conditioned on something equals 1. After the α's and β's have been computing by running the dynamic programming loops, we can find the tags by just looking them up in the tables

$$\left[\underset{\hat{y}_i}{\text{argmax}}\, \alpha(i,\hat{y}_i)\beta(i,\hat{y}_i)\right]_{i=1}^N$$

This expression is not the same as $\text{argmax}_{\hat{y}} P(\hat{y}_1^N, x_1^N)$. The expression is also a Minimum Bayes risk rule, but for a different score function.

3 Expectation Maximization (EM)

E - Step: Guess what the labels should be at each position in the training sequence given parameters
M - Step: Guess what the parameters should be given labels

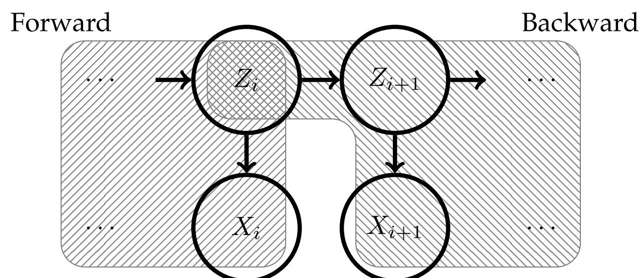
Repeat both steps until convergence.

At each step have some current estimate of emissions and transitions and some guess for the label t at each position i in the training sequence.

E-Step is done using forward and backward algorithm and probabilities.

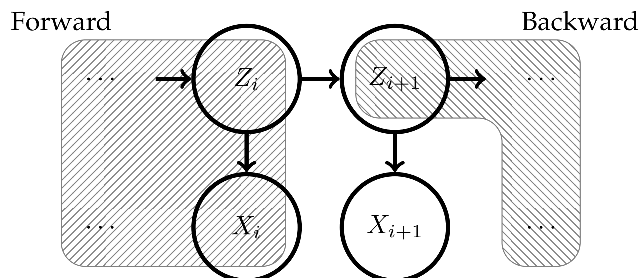
$$\alpha(i,t)\beta(i,t) = P(Z_i = t, X_1^N)$$

What we think happened at each time point: $P(Z_i = t \mid X_1^N) = \frac{1}{Z}\alpha(i,t)\beta(i,t)$



To compute expected counts for transitions, we need to fix the state at two adjacent time steps:

$$P(Z_i = t \wedge Z_{i+1} = t' \mid X_1^N) = \frac{1}{Z}\alpha(i,t)P(t' \mid t)P(X_{i+1} \mid t')\beta(i+1,t')$$



Expected Counts for Emissions:

$$ec(t,x) = \sum_{i \text{ s.t. } X_i = x} P(Z_i = t \mid X_1^N)$$

Expected Counts for Transitions:

$$ec(t,t') = \sum_i P(Z_i = t \wedge Z_{i+1} = t' \mid X_1^N)$$

First compute the forward and backward probabilities for all the sequences in your data. Then make another pass over the data and collect from the forward backward probabilities whether you think you saw the relevant emission and transition.

This completes the E-step.
M-step:

$$O_{t,x} = \frac{ec(t,x)}{\sum_{x'} ec(t,x')}$$

$$A_{t,t'} = \frac{ec(t,t')}{\sum\limits_{t''} ec(t,t'')}$$

$$\theta = \{O, A\}$$

Where O represents emissions and A represents transitions. This completes the EM algorithm, now we'll discuss why this is the right thing to do. Want to find good parameters for data so need parameters that maximize:

$$\underset{\theta}{\operatorname{argmax}}\, P(X;\theta)$$

MLE:

$$\underset{\theta}{\operatorname{argmax}} \sum_Z P(XZ;\theta)$$

$$\max_{A,O}\ \log \prod_{i=1}^{N} A_{Z_i Z_{i+1}} O_{Z_i X_i}$$

$$\max_{A,O}\ \sum_i \log A_{Z_i Z_{i+1}} + \log O_{Z_i X_i}$$

$$\max_A\ \sum_{t,t'} c(t,t') \log A_{t,t'} + \max_O \sum_{t,X} c(t,X) \log O_{t,X}$$

$$\forall t\ \max_{A_t,O} \sum_{t'} c(t,t') \log A_{t,t'} \text{ s.t. } \sum_{t'} A_{t,t'} = 1$$

Where: $c(t,t') = \sum_i I(Z_i = t \wedge Z_{i+1} = t')$

Count and normalize as a constrained maximization problem. Lagrangian of vector we are predicting,

$$g(A_{t,t'}) = \sum_{t'} c(t,t') \log A_{t,t'} + \lambda \sum_{t'} A_{t,t'}$$

$$0 = \frac{\partial g}{\partial A_{t,t'}} = \frac{c(t,t')}{A_{t,t'}} + \lambda$$

$$A_{t,t'} = \frac{c(t,t')}{-\lambda}$$

Big HMM Model for kth iteration of the EM loop

$$\boxed{\max_{\theta} \underset{P(Z|X\theta^k)}{\mathrm{E}} \log P(XZ;\theta)}$$

$$\mathrm{E}_{P(Z|X\theta^k)} \sum_i \log A_{Z_i,Z_{i+1}} + \log O_{Z_i,X_i}$$

Push expectation into sum:

$$\mathrm{E}\left[\sum_{t,t'} c(t,t') \log A_{t,t'} + \sum_{t,X} c(t,X) \log O_{t,X}\right]$$

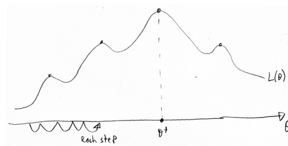
Update rule for M-step:

$$\sum_{t,t'} ec(t,t') \log A_{t,t'} + \sum_{t,X} ec(t,X) \log O_{t,X}$$

We get this update rule because A doesn't appear inside the right sum and O doesn't appear inside the left sum. The reason A doesn't appear there is because the expectation depends on frozen old parameters θ from iteration k. This enables us to do the math in the update rule and the max for the HMM model. θ is embedded in $ec(t, t')$ and $ec(t, X)$

Use old guess, looking for $argmax_\theta$ but no guarantee that will actually find it. This is hard because this is a complicated function with many local maxima.

$$\theta = \{A, O\} \quad L(\theta) = \log P(X; \theta)$$



The approximation at each step does not get worse because:

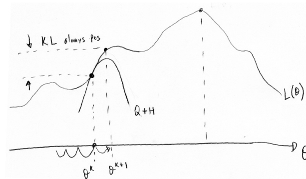
$$\boxed{\max_\theta \mathop{\mathrm{E}}_{P(Z|X\theta^k)} \log P(XZ;\theta)}$$

$$\max_\theta \mathrm{E}\, P(Z \mid X\theta^K)[\log P(XZ;\theta)] = Q$$

$$\mathrm{E}[\log P(X;\theta) + \log P(Z \mid X;\theta)]$$

$$\log P(X;\theta) + \mathrm{E}\left[\log \frac{P(Z \mid X;\theta)}{P(Z \mid X;\theta^k)} + \log P(Z \mid X;\theta^{(k)})\right]$$

Here is what we really want:
$L(\theta) - KL(Z \mid X; \theta^k \parallel Z \mid X; \theta) - H(Z \mid X; \theta^k)$
$Q = L - KL - H$
$Q + H = L - KL$



Can maximize Q directly because it is the same as maximizing likelihood over θ which means we jump directly to the top of the $Q + H$ curve. This is EM on hidden variables; solve parameters, update and increase P but risk not finding global optimum.

3.1 Variational Bayes

Change update rule very slightly, by adding exponent of digamma function in front of everything. This prevent EM from overfitting.

$$O_{t,X} = \frac{\exp(\Psi(ec(t,X)))}{\exp(\Psi(\sum_{X'} ec(t,X'))}$$

$$A_{t,t'} = \frac{\exp(\Psi(ec(t,t')))}{\exp(\Psi(\sum_{t^n} ec(t,t^n))}$$

$$\exp(\Psi(X)) \simeq \begin{cases} X - \frac{1}{2} & \text{if } X > 1 \\ \frac{X^2}{2} & \text{if } X \in [0,1] \end{cases}$$

Dirichlet Prior:

$$\theta = \{A, O\}$$

$$P(\theta;\alpha) = Dir(\alpha) = \prod_j \frac{\Gamma(\sum_i \alpha_{ji})}{\prod_i \Gamma(\alpha_{ji})} \prod_i \theta_{ji}^{\alpha_{ji}-1}$$

Alpha is in range 0 to 1. For variational bayes typically $\alpha_{ji} = 0.0001$

Updating the Emission and Transistion equations with addition of dirichlet alpha,

$$O_{t,x} = \frac{\exp(\Psi(ec(t,x) + \alpha_{t,x}))}{\exp(\Psi(\sum_{x'} ec(t,X') + \alpha_{t',x})}$$

$$A_{t,t'} = \frac{\exp(\Psi(ec(t,t') + \alpha_{t,t'}))}{\exp(\Psi(\sum_{t'} ec(t,t') + \alpha_{t,t'})}$$

Dirichlet prior implies not trusting small counts too much.

$$p(X_{N+1} = i \mid X_1^N \alpha) = \frac{1}{Z}(C(i) + \alpha_i)$$

Going back:

$$\max_\theta \mathrm{E}_{\underbrace{q(z)}}[\log P(XZ;\theta)] = Q$$

Choose any distribution over the hidden data such as $q(z)$, $q(z)$ is some distribution that we pick and takes place of θ_k from the last iteration. In our case, $q(z)$ is table of numbers.

$$\log P(X;\theta) + \mathrm{E}\left[\log \frac{P(Z \mid X;\theta)}{q(z)} + \log\, q(z)\right]$$

$$L(\theta) - KL(q(Z) \parallel P(Z \mid X;\theta)) - H(q(Z))$$

$$Q + H = L - KL$$

Push $Q + H$ as high as possible to meet L by using old parameters and then guess.

$$q(Z) = P(Z \mid X;\theta^k)$$

Really interested in $Q + H$ not just Q. Call $Q + H = F(q(Z), \theta)$

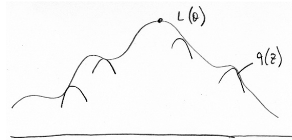
E-step:

$$\max_{q(Z)} F$$

M-step:

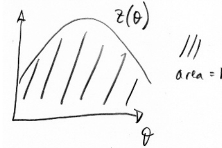
$$\max_{q(\theta)} F$$

Variational Optimization: F is lower bound on function (L) that we want to maximize. As we vary $q(z)$ it touches and traces out the function we're trying to optimize.



$$F(q(Z), q(\theta)) = \mathrm{E}_{q(Z)q(\theta)}[\log \frac{P(XZ\theta;\alpha)}{q(Z)q(\theta)}]$$

Now θ is a variable like q. Treat θ like we treated Z before.



Let's maximize with respect to θ (calculus of variations)

$$0 = \frac{\partial F}{\partial q(\theta)} = \frac{\partial}{\partial q(\theta)} \int q(\theta) \sum_Z q(Z)[\log P(XZ\theta;\alpha) - \log\, q(Z) - \log\, q(\theta)]\mathrm{d}\theta$$

$$0 = \sum_Z q(Z)\,[\log[P(XZ \mid \theta)P(\theta;\alpha)] - \log\, q(Z) - \log\, q(\theta)] - 1$$

Solve for $q(\theta)$

$$\log\, q(\theta) = \sum_Z q(Z) \left[\sum_{ij} c(i,j)\log\theta_{ji} + \sum_{ji}(\alpha_{ji} - 1)\log\theta_{ji}\right] + C$$

$$q(\theta) = \exp\left[\sum_{ij}(ec(j,i) + \alpha_{ji} - 1)\log\theta_{ji} + C\right]$$

$$q(\theta_j) \propto \prod_i \theta_{ji}^{\sum_{ij} ec(j,i)+\alpha_{ji}-1} = Dir(ec + \alpha)$$

This completes $q(\theta)$ now $q(Z)$

$$F(q(Z), q(\theta)) = \mathrm{E}_{q(Z)q(\theta)}\left[\log \frac{P(XZ\theta;\alpha)}{q(Z)q(\theta)}\right]$$
$$0 = \frac{\partial F}{q(Z)} = \int q(\theta)[\log P(XZ \mid \theta) + \log P(\theta;\alpha) - \log\ q(Z)]\mathrm{d}\theta + C$$
$$\log q(Z) = \mathrm{E}_{q(\theta)}\left[\sum_{ji} c(j,i)\log\theta_{ji} + (\alpha_{ji}-1)\log\theta_{ji}\right] + C$$
$$\log\ q(Z) = \sum_{ji} c(j,i)\,\mathrm{E}_{q(\theta)}[\log\theta_{ji}] + C'$$
$$q(Z) \propto \exp\left[\sum_{ji} c(j,i)\,\mathrm{E}_{q(\theta)}[\log\theta_{ji}]\right]$$
$$q(Z) \propto \prod_{ji} \exp\left(\mathrm{E}_{q(\theta)}[\log\theta_{ji}]\right)^{c(j,i)}$$

This means that you can compute $q(Z)$ by using forward backward with exponentiated expected log probabilities instead of real probabilities.

To find $\mathrm{E}_{q(\theta)}[\log\theta_{ji}]$:

Lemma: For probability distribution of the form:

$$P(X;\eta) = \frac{h(X)e^{X^T\eta}}{Z(\eta)}$$

it follows that:

$$\mathrm{E}[X] = \frac{\partial}{\partial\eta}\log Z(\eta)$$

To apply this to the the Dirichlet distribution that we found above for $q(\theta)$, chose $X = \log\theta, \eta = \alpha - 1$. The digamma function is defined as the derivative of the log gamma function:

$$\frac{\partial}{\partial X}\log\Gamma(X) = \Psi(X)$$

So differentiating the log of the normalization constant of the Dirichlet distribution results in:

$$\mathrm{E}_{q(\theta)}[\log\theta_{ji}] = \Psi(ec(j,i) + \alpha_{ji}) - \Psi\left(\sum_{i'} ec(j,i') + \alpha_{ji'}\right)$$

$$q(z) = \exp\left(\sum_{ji} c(j,i)E[\log(\theta_{ji})] + C\right)$$

can be expressed as

$$q(z) = \frac{1}{Z}\prod_{ji} e^{E[\log(\theta_{ji})]c_{ji}}$$

which can be readily compared to the canonical form for probability of a sequence of outcomes: $\prod_{ji}\theta_{ji}^{c_{ji}}$.

From properties of exponential family of functions, we can write $E[\log(\theta_{ji})] = \frac{\partial}{\partial\theta_{ji}}\log(Z(\theta))$. Note that $Z(\theta)$ is the normalization constant in the Dirichlet distribution, and thus includes Γ functions. The derivative of $\log(\Gamma)$ is the Ψ (digamma) function. $q(z)$ can be written as

$$q(z) = \frac{\exp(\Psi(ec(j,i) + \alpha(j,i))}{\exp(\Psi(\sum_i(ec(j,i) + \alpha(j,i)))}$$

$q(z)$ is computed using dynamic programming (forward-backward algorithm).

Note that in general, probabilities(P) have to add up to 1. Expectation of probabilities (E[P]) have to add up to 1. But $e^{E[\log P]}$ does not need to add up to 1.

Use of Variational Bayes (VB) with Expectation-Maximization (EM) usually improves the performance. EM is too general, and VB avoids over-fitting. Table below shows some results reported in Johnson's paper[1] for unlabeled POS tagging:

Estimator	#Tags	Accuracy
EM	50	0.40
VB	50	0.47
EM	25	0.46

Note that VB improves the accuracy. Also EM run with with fewer number of tags (than in the data) appears to improve accuracy as well. But the accuracies are much lower compared to those observed in supervised learning such as Perceptron.

Another approach is *Partially Supervised Tagging*, such as

- EM (on unlabeled data)+Fixed Labels (for labeled data).
- EM with many clusters, to generate parts of speech clusters and using supervised training to label the clusters (NN, DT etc..)

4 Conditional Random Fields (CRF)

CRF=Log linear model + Dynamic Programming

Conditional probability $P(y \mid x)$ where y represents the labels and x represents the data is written as

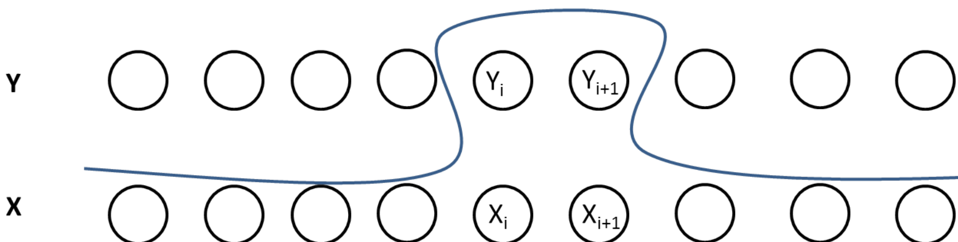
$$P(y \mid x) = \frac{1}{Z} e^{w^\top \Phi(y,x)}$$

Z is normalization factor given by $Z(X) = \sum_Y e^{w^\top \Phi(y,x)}$.

Decoding problem: $Y^* = \text{argmax}_Y \, P(Y \mid X)$. Log-linear models are also referred to as *logistic regression* or *maximum entropy*. The HMM is a *Generative Model* because it generates the joint probability distribution of the data and the labels $P(y \mid x) = \frac{P(y,x)}{P(x)}$. Max Entropy, like the perceptron, is a *Discriminative Model* because it generates only the conditional dependence of the labels on the data. $P(y_1^N \mid z_1^N)$ represents the conditional probability of an entire sequence.

Here Φ is built out of local features ϕ which look at two adjacent labels in the entire sequence.

$$\Phi_j(y_1^N, x_1^N) = \sum_{i=1}^{N} \phi_j(x, y_i, y_{i+1}, i)$$



[1] Mark Johnson, Why EM doesn't find good HMM POS tags?, *Proc. 2007 Joint Conf. on Empirical Methods in Natural Language Processing and Computational Natural Language*, pp. 296–305, June 2007.

Testing=Decoding. $\hat{y} = \text{argmax}_{y_1^N} P(y_1^N \mid x_1^N) = \text{argmax}_{y_1^N} w^T \Phi(y, x)$ because exponential does not alter the result of argmax. This is almost identical to the perceptron.

The weights w are found during training. $w^* = \text{argmax}_w \underbrace{\log P(y \mid x)}_{L}$. A typical way to do this would be to solve $\frac{\partial L}{\partial w} = 0$ for w. However, this cannot be solved easily. Instead, we use *gradient ascent* where w is updated iteratively: $w+ = \eta \frac{\partial L}{\partial w}$, where η is the step size (learning rate).

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}(-\log(Z(x)) + w^T \Phi) = \Phi(y, x) - \frac{\partial}{\partial w} \log(Z(x))$$

Recall, for exponential familiy of functions, if $P(X \mid \eta) = \frac{1}{Z(\eta)} h(X) e^{X^T \eta}$, then $\frac{\partial}{\partial \eta} \log(Z(\eta)) = E[X]$. In our case, $\eta = w$ and $X = \Phi$. Thus,

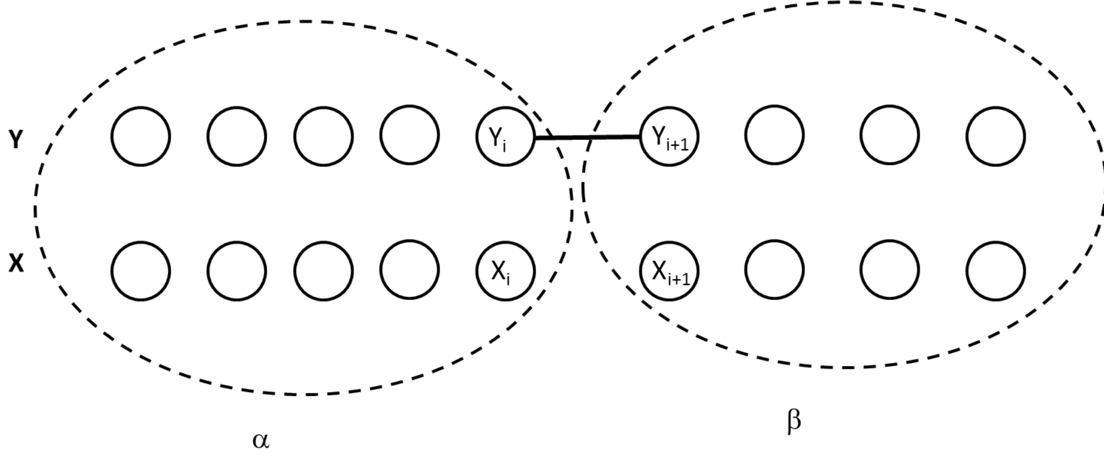
$$\frac{\partial L}{\partial w} = \Phi(y, x) - E_{P_w(y|x)}[\Phi]$$

In the above equation $E_{P_w(y|x)}[\Phi]$ are the expected features from the current estimate of the label sequence ($\hat{y}$) using the current w.

Note that this update rule is similar to the perceptron: $\Delta w = \Phi(Y, X) - \Phi(\hat{Y}, X)$. The difference is that perceptron looks at one sequence at a time, where as logistic regression looks at all the sequences in the update rule. Dynamic programming (forward-backward) is used for evaluating the expectations.

$$E_{P_w(y|x)}[\Phi_j] = \sum_i \sum_{t,t'} \phi_j(x, y_i = t, y_{i+1} = t', i) \underbrace{\alpha(i, t)\beta(i + 1, t') \frac{1}{Z} e^{w^T \phi(x, y_i, y_{i+1}, i)}}_{P_w(y_i = t, y_{i+1} = t' | x_1^N)} \tag{7}$$

Here t and t′ represent possible labels for y_i and y_{i+1}.



The probability of adjacent pairs of tags can be computed by adapting the forward-backward algorithm.

$$P_w(y_i, y_{i+1} \mid x_1^N) = \sum_{y_1, \ldots, y_{i-1}, y_{i+2}, \ldots y_N} P_w(y_i, \ldots, y_N \mid x_1^N) \tag{8}$$

$$= \sum_{y_1, \ldots, y_{i-1}, y_{i+2}, \ldots y_N} \frac{1}{Z} \prod_{i'=1}^{N} e^{w^\intercal \phi(x, y_{i'}, y_{i'+1}, i')} \tag{9}$$

$$= \frac{1}{Z} \left(\sum_{y_1, \ldots, y_{i-1}} \prod_{i'=1}^{i-1} e^{w^\intercal \phi(x, y_{i'}, y_{i'+1}, i')} \right) e^{w^\intercal \phi(x, y_i, y_{i+1}, i)} \left(\sum_{y_{i+2}, \ldots, y_N} \prod_{i'=i+1}^{N} e^{w^\intercal \phi(x, y_{i'}, y_{i'+1}, i')} \right) \tag{10}$$

$$= \frac{1}{Z} \alpha(i, y_i) e^{w^\intercal \phi(x, y_i, y_{i+1}, i)} \beta(i + 1, y_{i+1}) \tag{11}$$

$$\tag{12}$$

Here α is a dynamic programming table defined as:

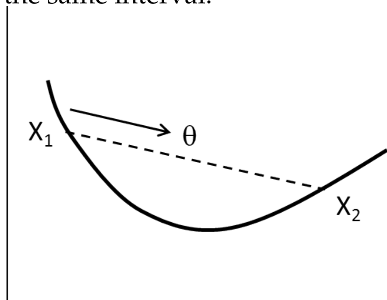
$$\alpha(i,t) = \sum_{t'} \alpha(i-1,t') e^{w^\intercal \phi(x,t',t,i-1)} \tag{13}$$

and β is analogous.

Unlike EM, CRF training does not get trapped in local optima because L_{CRF} is concave. Can we prove that L_{CRF} is concave?

Consider the log-likelihood function: $L = \log P(y \mid x) = w^T\Phi - \log(Z(x))$. Since $w^T\Phi$ is linear in w, we need to show that $\log(Z(x))$ is convex.

Definition of convex function: A function f is convex if for all x_1, x_2 and θ where $0 \le \theta \le 1$, $f(\theta x_1 + (1-\theta)x_2) \le \theta f(x_1) + (1-\theta) f(x_2)$. This is saying that any chord on the function is above the function itself on the same interval.



i.e. we need to show that Hessian $\nabla^2 f(x) \ge 0$ (positive semi definite). Or, $v^T \nabla^2 f(x) v \ge 0$.

Here $f(x) = \log \sum_i e^{x_i}$. $\nabla f = \frac{1}{\sum_i e^{x_i}} e^{x_i} = \frac{1}{\sum_i z_i} z_i$, where $z_i = e^{x_i}$.

We can express $\sum_i z_i$ in vector form as $1^T z$, where 1 is a column vector of ones.

$$\nabla^2 f = -\frac{1}{(1^T z)^2} z z^T + \frac{1}{1^T z} diag(z)$$

where $diag(z)$ is a diagonal matrix with z_i along the diagonal.

$$\begin{aligned}
(v^T \nabla^2 f v)(1^T z)^2 &= v^T(-zz^T + (1^T z) diag(z))v \\
&= -v^T z z^T v + (1^T z) v^T diag(z) v \\
&= -(v^T z)^2 + (1^T z) v^T diag(z) v
\end{aligned}$$

Therefore, we need to show that $(1^T z) v^T diag(z) v \ge (v^T z)^2$. The left hand side of this inequality, which is equal to $(\sum_i z_i)(\sum_i v_i z_i v_i)$, can be expressed as $(\sqrt{z}^T \sqrt{z})(v\sqrt{z})^T(v\sqrt{z})$.

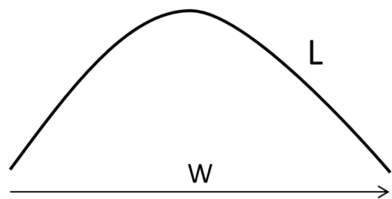
Let $a = \sqrt{z}$ and $b = v\sqrt{z}$. Then the inequality becomes $(a^T a)(b^T b) \ge (a^T b)^2$, which is the Cauchy Schwarz inequality and is always true for any vectors a and b. The equality holds only when vectors a and b are linearly dependent.

Thus we have shown that the log-likelihood L is concave and has a single maximum, which the gradient ascent algorithm attempts to converge to.

4.1 CRF Summary

The log likelihood function of CRF is concave and thus CRF training does not get trapped in local optima. Here, L is a log likelihood of CRF, w is a weight vector and f is a feature.

$$L = w^T \Phi - \log Z(w)$$



Since $w^T\Phi$ is linear in w, $\log \sum_y e^{w^T\Phi(x,y)}$ is convex. It doesn't matter if you scale by factors in form of $\log \sum_i e^{\lambda_i}$. Using Cauchy Schwarz theorem if you add convex functions we get convex. Therefore, e^{λ_i} is convex and the sum of convex is always convex. The fact that the function is convex is nothing to do with dynamic programming. We use CRF with HMM structure and use dynamic programming to calculate sum. Once you are at a local maximum you are done. For homework 4 you will implement this. We need to capture $\partial L/\partial w$. We should use the regularization term, $(\lambda/2)w^T w$ to push the weights towards zero. This term prevents overfitting.

$$L = w^T\Phi - \log Z(w) - \frac{\lambda}{2}w^T w$$

$$\frac{\partial L}{\partial w} = \Phi - E[\Phi] - \lambda w$$

Add the weights to the gradient before update. This is called L2 regularization and $e^{(\lambda/2)w^T w}$ is Gaussian prior.

$$\exp(L) = P(y|x)e^{(\lambda/2)w^T w}$$

4.2 Neural CRF

We can generalize the CRF equation

$$P(y|x) = \frac{1}{Z}e^{\sum_i w^T\phi(x,y_i,y_{i+1},i)} \tag{14}$$

as

$$P(y|x) = \frac{1}{Z}e^{\sum_i E(x,y_i,y_{i+1},i,w)} \tag{15}$$

where E is a general "energy" function. Generalizing the gradient derivation of the CRF, we have

$$\begin{aligned}
\frac{\partial L}{\partial w} &= \sum_i \frac{\partial}{\partial w}E(x,y_i,y_{i+1},i,w) - E_{P(y|x)}\left[\sum_i \frac{\partial}{\partial w}E(x,y_i,y_{i+1},i,w)\right] && (16)\\
&= \sum_i \frac{\partial}{\partial w}E(x,y_i,y_{i+1},i,w) - \sum_{i,t,t'} P(y_i = t, y_{i+1} = t' \mid x)\frac{\partial}{\partial w}E(x,t,t',i,w) && (17)\\
&= \sum_i \frac{\partial}{\partial w}E(x,y_i,y_{i+1},i,w) - \sum_{i,t,t'} \alpha(i,t)e^{E(x,t,t',i,w)}\beta(i+1,t')\frac{\partial}{\partial w}E(x,t,t',i,w) && (18)
\end{aligned}$$

We can model $E(x,y_i,y_{i+1},i,w)$ as a neural net parameterized by w and taking $\phi(x,y_i,y_{i+1},i)$ as input. Then the gradient $\frac{\partial E}{\partial w}$ in the formula above can be computed with backpropagation.

4.3 Extension: Latent CRF

Now for each label sequence y, we consider a refined label sequence with hidden variable z. For example, for word sequence $x =$"time flies", the label sequence would be $y =$"NN VV". We also have a refined label sequence $z =$"NN-2 VV-1", where the hidden variables 2 and 1 represent "time" is labeled with the second refined label of "NN" and "flies" is labeled with the first refined label of "VV". Now with latent variable z, We have

$$P(z,y \mid x) = \frac{1}{Z(x)}e^{w^\mathsf{T}\Phi(z,y,x)}$$

where $Z(x)$ is a normalization factor given by $Z(x) = \sum_{z,y} e^{w^\mathsf{T}\Phi(z,y,x)}$.

In order to guess the latent labels z from data label with y, we will use:

$$P(z|x,y) = \frac{P(y,z|x)}{\sum_z P(y,z|x)} = \frac{\frac{1}{Z(x)}\exp(w^\mathsf{T}\Phi)}{\sum_z \frac{1}{Z(x)}\exp(w^\mathsf{T}\Phi)} = \frac{\exp(w^\mathsf{T}\Phi)}{\sum_z \exp(w^\mathsf{T}\Phi)} \tag{19}$$

The log likelihood of the data is:

$$L = \log \prod_i p(y_i|x_i) = \sum_i \log \sum_z p(y_i, z_i|x_i) \tag{20}$$

$$= \sum_i \log \frac{\sum_z \exp(w^\mathsf{T}\Phi(x_i, y_i, z_i))}{\sum_{y,z} \exp(w^\mathsf{T}\Phi(x_i, y_i, z_i))} \tag{21}$$

For SGD we want to take the gradient:

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left[\log \sum_z \exp(w^\mathsf{T}\Phi) - \log \sum_{y,z} \exp(w^\mathsf{T}\Phi)\right] \tag{22}$$

$$= \sum_z \frac{\exp(w^\mathsf{T}\Phi)\Phi}{\sum_z \exp(w^\mathsf{T}\Phi)} - \frac{1}{Z(x)} \sum_{y,z} \exp(w^\mathsf{T}\Phi)\Phi \tag{23}$$

$$= \sum_z p(z|x, y)\Phi - \sum_{y,z} p(y, z|x)\Phi \qquad \text{using eq. 19} \tag{24}$$

$$= E_{p(z|x,y)}\Phi - E_{p(y,z|x)}\Phi \tag{25}$$

This is the expected feature when we saw the correct labels minus the expected feature using labels we expect to see. Similar to CRF without latent variables, $E_{P(Z|Y,X)}[\Phi]$ and $E_{P(Z,Y|X)}[\Phi]$ are calculated using forward-backward algorithm.

$$E_{P(Z,Y|X)}[\Phi_j] = \sum_i \sum_{t,t'} f_j(Z_i = t, Z_{i+1} = t', Y_i = y(t), Y_{i+1} = y(t'), X_1^N) \underbrace{\alpha(i,t)\beta(i+1,t')\frac{1}{Z}e^{w^T\phi(Z_i, Z_{i+1}, X_1^N)}}_{P(Z_i=t, Z_{i+1}=t', Y_i=y(t), Y_{i+1}=y(t')|X_1^N)}$$

$$E_{P(Z|Y,X)}[\Phi_j] = \sum_i \sum_{t,t' s.t. y(t)=y_i, y(t')=y_{i+1}} f_j(Z_i = t, Z_{i+1} = t', Y_i = y_i, Y_{i+1} = y_{i+1}, X_1^N) \underbrace{\alpha(i,t)\beta(i+1,t')\frac{1}{Z}e^{w^T\Phi(Z_i, Z_{i+1}, X_1^N)}}_{P(Z_i=t, Z_{i+1}=t'|Y_1^N, X_1^N)}$$

and the corresponding forward pass is

$$\alpha(i+1, t) = \sum_s \alpha(i, s) e^{w^T\phi(Z_i=s, Z_{i+1}=t, Y_i=y(s), Y_{i+1}=y(t), X_1^N)}$$

$$\alpha(i+1, t) = \sum_{s:y(s)=y_i} \alpha(i, s) e^{w^T\phi(Z_i=s, Z_{i+1}=t, Y_i=y_i, Y_{i+1}=y_{i+1}, X_1^N)}$$

note that when the refined label z is given, we can deterministically derive the non-refined label y by getting rid of the latent symbol, such as from "NN-2" to "NN".

To compare CRF and LCRF, let's look at their update function
LCRF:

$$w+ = \eta(E_{P(z|y,x)}[\Phi] - E_{P(z,y|x)}[\Phi])$$

which moves towards the Z we expect to see given the Y we saw and moves away from the Y, Z we expect to see.
CRF:

$$w+ = \eta(\Phi - E_{P(y|x)}[\Phi])$$

which moves towards the y we saw and moves away from the y we expect to see.

4.4 Extension: Loss-Aware CRF

Suppose that we have a loss function $\ell(y, y^*)$ that measure the cost of predicting sequence y when the true sequence is y^*. We wish to minimize the expected loss:

$$
\begin{aligned}
L &= E[\ell(y, y^*)] \\
&= \sum_y P(y \mid x)\ell(y, y^*) \\
\frac{\partial L}{\partial w} &= \frac{\partial}{\partial w} \sum_y P(y \mid x)\ell(y, y^*) \\
&= \sum_y \ell(y, y^*) \frac{\partial}{\partial w} P(y \mid x) \\
&= \sum_y \ell(y, y^*) \frac{\partial}{\partial w} \frac{1}{Z} e^{w^\top \Phi(x,y)} \\
&= \sum_y \ell(y, y^*) \left[\frac{1}{Z} \Phi(x, y) e^{w^\top \Phi(x,y)} - \frac{1}{Z^2} e^{w^\top \Phi(x,y)} \sum_{y'} \Phi(x, y') e^{w^\top \Phi(x,y')} \right] \\
&= \sum_y \frac{1}{Z} e^{w^\top \Phi(x,y)} \ell(y, y^*) \left[\Phi(x, y) - \sum_{y'} \Phi(x, y') e^{w^\top \Phi(x,y')} \right] \\
&= E[\ell(y, y^*)\Phi(x, y)] - E[\ell(y, y^*)] E[\Phi(x, y)] \\
&= E[\ell\, \Phi] - E[\ell] E[\Phi] \qquad \text{for short}
\end{aligned}
$$

Thus the gradient of the expected loss is the covariance of the features with the loss. Peforming an iteration of gradient descent on this function will decrease weights of features that are highly correlated with loss.

We need to compute the three quantities in the update rule efficiently. Assume that the loss decomposes over positions in the sequence:

$$\ell(y, y^*) = \sum_i \ell'(y_i, y_i^*)$$

as is the case when loss is the number of incorrect tags:

$$\ell'(y_i, y_i^*) = I(y_i \neq y_i^*)$$

In this case, both $E[\ell]$ and $E[\Phi]$ can be computed using the same algorithm that was used to compute $E[\Phi]$ for CRF training (eq. 7).

The term $E[\ell\, \Phi]$ is more complex, as it involves multiplying ℓ and Φ for each sequence. Dropping x and y^* as they are fixed:

$$
\begin{aligned}
E[\ell\, \Phi] &= \sum_y P(y) \left(\sum_i \ell'(y_i) \right) \left(\sum_i \phi(y_i, y_{i+1}) \right) && (26) \\
&= \sum_y P(y) \sum_i \sum_{i'} \ell'(y_{i'}) \phi(y_i, y_{i+1}) && (27) \\
&= \sum_i \sum_{y_i, y_{i+1}} \phi(y_i, y_{i+1}) \sum_{y_1 \cdots y_{i-1}} \sum_{y_{i+2} \cdots y_n} P(y) \sum_{i'} \ell'(y_{i'}) && (28) \\
&= \sum_i \sum_{y_i, y_{i+1}} \phi(y_i, y_{i+1})\, \mathrm{el}(y_i, y_{i+1}, i) && (29)
\end{aligned}
$$

where $\mathrm{el}(y_i, y_{i+1}, i)$ represents the expected loss of all paths passing through the transition from y_i to y_{i+1} at step i.

Expected loss can be computed with an extended version of the forward backward algorithm. We compute a forward probability $\alpha(i,t)$ as in eq. 13, and compute a forward expected loss $\hat{\alpha}(i,t)$ with the recurrence:

$$\hat{\alpha}(i,t) = \sum_{t'} \hat{\alpha}(i-1,t')e^{w^\intercal \phi(t',t,i-1)} + \alpha(i-1,t')\ell'(t)e^{w^\intercal \phi(t',t,i-1)}$$

After computing $\hat{\beta}$ analogously,

$$\begin{aligned} \mathrm{el}(y_i, y_{i+1}, i) = \alpha(i,y_i)\beta(i+1,y_{i+1})\ell'(y_{i+1})e^{w^\intercal \phi(y_i,y_{i+1},i)} \\ + \hat{\alpha}(i,y_i)\beta(i+1,y_{i+1})e^{w^\intercal \phi(y_i,y_{i+1},i)} \\ + \alpha(i,y_i)\hat{\beta}(i+1,y_{i+1})e^{w^\intercal \phi(y_i,y_{i+1},i)} \end{aligned}$$

Thus, the overall procedure is to

1. compute α and $\hat{\alpha}$
2. compute β and $\hat{\beta}$
3. compute $E[\ell]$ and $E[\Phi]$ using α and β
4. compute $\mathrm{el}(y_i, y_{i+1}, i)$ using $\hat{\alpha}$ and $\hat{\beta}$
5. compute $E[\ell\,\Phi]$ with eq. 29
6. take an SGD update $w -= \eta(E[\ell\,\Phi] - E[\ell]E[\Phi])$

5 N-Gram Language Model

In language modeling we are interested in capturing the probability of a sequence of words. We are looking at the words themselves. N-gram means you look at a sentence with a sliding window.

$$P(w_1^N) = \prod_{i=1}^{N} P(w_i | w_{i-N+1}^{i-1})$$

Suppose trigrams for our example (N=3).

$$w1, w2, w3, w4, w5$$

w1 w2 w3 w4 w5

[P(w5|w3 w4)]

[P(w4|w2 w3)]

[P(w3|w1 w2)] ←

We pretend for w5 that earlier probability of the sentence doesn't matter. At the beginning we make up a special word that refer all sentences.

$$P(w_1 | \langle start \rangle \langle start \rangle)$$

The point of breaking down the probability, the independent assumption, is that given two previous words the rest of sentence doesn't matter. This reduces the number of parameters we need and we just need to count.

$$P(w_i | w_{i-2}, w_{i-1}) = \frac{c(w_{i-2}, w_{i-1}, w_i)}{c(w_{i-2}, w_{i-1})}$$

How many numbers in the table? Assume V is the size of vocabulary, therefore the number is V^3. This is too big and we cannot estimate this number accurately. We find most are zero because V^3 is a billion and we only have a million words to choose form. The big issue is what to do with the zero, estimate probability for things we've never seen. Use Dirchilet's prior.

$$P(X_{N+1}|X_1^N;\alpha) = \frac{(c(j)+\alpha_j)}{(\sum_j c(j)+\alpha_j)}$$

What does it mean for using Dirichelet with N-gram? We have the counts. Let's simplify this problem to bigram and then above equation becomes

$$P(w_i|w_{i-1}) = \frac{c(w_{i-1},w_i)}{c(w_{i-1})}$$

What should α be. Previously we used 1 and 1/2. Now we can be smarter. We can use the unigram probabilities with α.

$$\alpha_{w_i} = P(w_i)$$

$$P(w_i|w_{i-1}) = \frac{(c(w_{i-1},w_i)+C(w_i))}{(c(w_{i-1})+N)}$$

where N is the total number of words in training data. α is a vector but it does not have to be add up to 1. It's magnitude is a tradeoff between words we saw and words we didn't see. Therefore,

$$P(w_i|w_{i-1}) = \frac{(c(w_{i-1},w_i)+\alpha C(w_i))}{(c(w_{i-1})+\alpha N)}$$

What is the relationship of the conditional probability $P(w_i|w_{i-1})$ from just counting to the Dirichlet probability.

$$P_{Dir}(w_i|w_{i-1}) = (1-\lambda)P(w_i|w_{i-1}) + \lambda P(w_i)$$

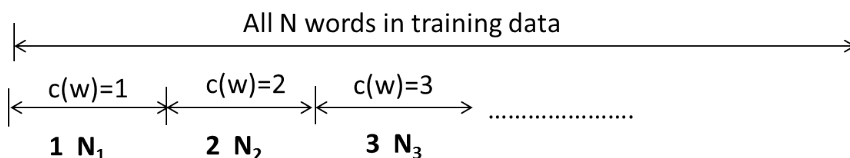
where

$$\lambda_{w_i} = \frac{\alpha}{c(w_{i-1})+\alpha N}$$

Even though we know how to calculate the probability in this way, this is not the efficient way to do it. Let's look at alternatives.

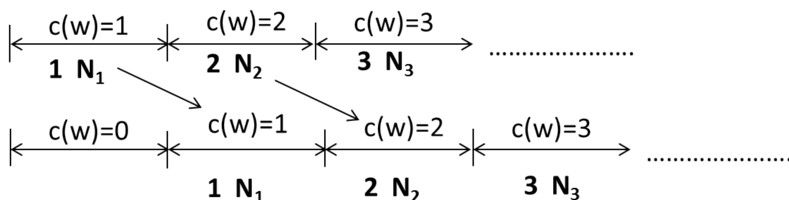
5.1 Alternative 1: Good-Turing

Imagine we have all of our words sorted to how often they occur.



where N_1 is the number of words we saw once and $N_r = \sum_w I(c(w)=r)$.

The idea of Good-Turing(GT) is to shift bars to the right. We equalize the probability masses, then we have space for words we haven't seen at all. We allocate space to these words.



$$P_{GT}(W) = ((r+1)N_{i+1})/(N\cdot N_i)$$

What happens for words you see frequently? There may be a word you've seen 1005 times. but there is nothing you've seen 1006 times. Usually you stop pretty soon, like r = 7. Then move on bar for 8 or higher. This example is for individual words. But we really want to do this for N-grams.

According to Katz Backoff

$$P_{Katz}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} P^*(w_i|w_{i-n+1}^{i-1}), & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \alpha(w_{i-n+1}^{i-1}) P_{Katz}(w_i|w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases}$$

Katz Backoff is recursive. You get the probability for droping the first word of the prior sentence. α is chosento make everything add up to one. What should α be exactly?

$$1 = \sum_{w_i} P_{Katz}(w_i|w_{i-n+1}^{i-1}) = \sum_{w_i, c>0} P^*(w_i|w_{i-n+1}^{i-1}) + \sum_{w_i, c=0} \alpha P_{Katz}$$

where $P^*(w_i|w_{i-n+1}^{i-1})$ is the discounted probability of words that we saw and $\sum_{w_i,c=0} \alpha P_{Katz}$ is the probability of words that we didn't see.

Solving for α:

$$\alpha(w_{i-n+1}^{i-1}) = \frac{1 - \sum_{w_i,c>0} P^*(w_i|w_{i-n+1}^{i-1})}{\sum_{w_i,c=0} P_{Katz}(w_i|w_{i-n+2}^{i-1})}$$

Katz Backoff frequently used with Good Turing as the discounted P^*. Informally if you have data with the big N-gram, use it, otherwise do it recursively. Katz Backoff makes everything to one. In practice you still get benefit from going up to every $N = 11$, but you can't write all the data to disk. Even if we see only one instance of N-gram, it still has a big probability. One good way to do this is by discounting estimates.

$$\max\{0, c(w_{i-1}, w_i) - D\}, D = \frac{1}{2}$$

This is called absolute discouting. It is not as fancy as Good Turing, but it is a basis of next technique.

5.2 Alternative 2: Kneser-Ney

$$P_{KN}(w_i|w_{i-1}) = \frac{\max\{0, c(w_{i-1}, w_i) - D\}}{c(w_{i-1})} + \beta(w_{i-1}) P_1(w_i)$$

β is chosen to make Kneser-Neys add up to one.

$$\beta(w_{i-1}) = \sum_{w_i : C(w_{i-1}, w_i) > 0} \frac{D}{C(w_{i-1})} = \frac{D}{C(w_{i-1})} N_{1+}(w_{i-1}\cdot)$$

where

$$N_{1+}(w_{i-1}\cdot) = \sum_{w_i} I(C(w_{i-1}, w_i) > 0)$$

We only sum over words that trigger the discount. β is big if we have a word that appears with a lot of different things after it. We still need to figure out P. It will be different from Katz Bakoff. It will be something we design for linear interpolation. The empirical probability $P(w_i)$ can be expressed as following.

$$P(w_i) = \sum_{w_{i-1}} P_{KN}(w_i|w_{i-1}) P(w_{i-1})$$

Kneser-Ney is combining absolute discounting with marginal constraint. Since P_1 appears in P_{KN}, we can determine P_1.

Soving for P_1

$$\frac{C(w_i)}{N} = \sum_{w_{i-1}:C(w_{i-1},w_i)>0} \frac{(C(w_{i-1},w_i)-D)C(w_{i-1})}{C(w_{i-1})N} + \sum_{w_{i-1}} \frac{D}{C(w_{i-1})} N_{1+}(w_{i-1}\cdot)P_1(w_i)\frac{C(w_{i-1})}{N}$$

If you sum over all w_{i-1} for all bigrams you get count of w_i. Some $C(w_{i-1})$ in the numerator and denominator cancel out.

$$C(w_i) = C(w_i) - DN_{i+}(\cdot w_i) + \sum_{w_{i-1}} DN_{1+}(w_i\cdot)P_1(w_i)$$

Now we can solve for P_1.

$$P_1(w_i) = \frac{N_{1+}(\cdot w_i)}{N_{1+}(\cdot\cdot)} \tag{30}$$

where $N_{i+}(\cdot\cdot)$ is the number of distinct positions in pairs of words. We've never seen the bigram before. Here we see the words in lots of other contacts. For example, 'San Francisco' we can say that this word is a common word, but Francisco is not common in other contexts.

Let's look at the estimation of N-grams. The exact formula for reference is

$$\frac{\max\{0, C(w_{i-1},w_i)-D\}}{C(w_{i-1})} + \frac{D}{C(w_{i-1})}\frac{N_{1+}(w_{i-1}\cdot)N_{1+}(\cdot w_i)}{N_{1+}(\cdot\cdot)}$$

Before we extend this formula, how do we code in real counts of bigrams, unigrams and unique occurrences? Easy to compute as going through data. Some terms are only counted the first time you see it. You can set them with one pass through data.

Extending the bigram formula above to n-grams yields:

$$P_{KN}(w_i \mid w_{i-n+1}^{i-1})\frac{\max\{0, C(w_{i-n+1}^{i})-D\}}{C(w_{i-n+1}^{i-1})} + \frac{D}{C(w_{i-n+1}^{i-1})}\frac{N_{1+}(w_{i-n+1}^{i-1}\cdot)N_{1+}(\cdot w_{i-n+2}^{i})}{N_{1+}(\cdot w_{i-n+2}^{i-1}\cdot)} \tag{31}$$

To applying Kneser-Ney smoothing recursively, we first rewrite the equation above in terms of a generic lower order distribution P_{n-1}:

$$P_{KN}(w_i \mid w_{i-n+1}^{i-1}) = \frac{\max\{0, C(w_{i-n+1}^{i})-D\}}{C(w_{i-n+1}^{i-1})} + \frac{DN_{1+}(w_{i-n+1}^{i-1}\cdot)}{C(w_{i-n+1}^{i-1})}P_{n-1}(w_i|w_{i-n+2}^{i-1})$$

This lower-order distribution is based on counts of unique n-grams (rather than raw n-gram counts) as in eq. 31, but these counts are themselves smoothed using absolute discounting and a further lower order distribution:

$$P_n(w_i|w_{i-n+1}^{i-1}) = \frac{\max\{0, N_{1+}(\cdot w_{i-n+1}^{i})-D\}}{N_{1+}(\cdot w_{i-n+1}^{i-1}\cdot)} + \frac{DN_{1+}(w_{i-n+1}^{i-1}\cdot)}{N_{1+}(\cdot w_{i-n+1}^{i-1}\cdot)}P_{n-1}(w_i|w_{i-n+2}^{i-1})$$

This process continues until we reach P_1, which is defined above in eq. 30.

Can we do Katz Backoff with absolute discounting? Sure no problem but we have to calculate the normalization constant, $\alpha(w_{i-n+1}^{i-1})$. If we do Kneser-Ney with anything other than absolute discounting, we get something harder to solve. Good Turing with marginal constraint requires a lot more math. Good Turing is really good at determining how much mass to allocate to zero. Katz Backoff is a good way of using that mass.

6 Deep Learning

Instead of using Hidden Markov Model to deal with the temporal variability of speech and Gaussian mixture model to determine how well each state of each HMM fits a short window of frames of coefficients that represents the acoustic input, we build a multi-layer perceptron (deep neural networks) with many hidden layers (usually three) as shown in Figure 1. Different from normal MLP of which the training process uses back propagation, the training step of deep learning contains two parts: initialization and back propagation.

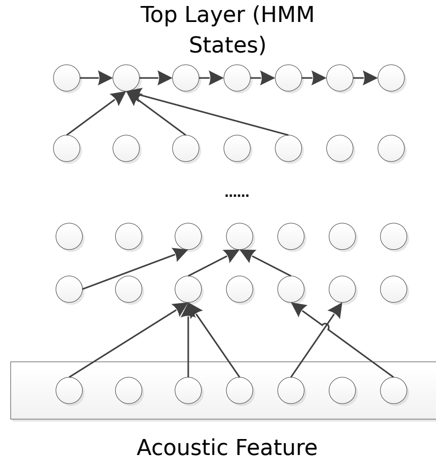


Figure 1: MLP Network

6.1 Initialization (Restricted Boltzman Machine)

According to the MLP network, we can compute the joint probability of hidden layer and visible layer:

$$P(v,h) = \frac{1}{Z}e^{-E(v,h)} \tag{32}$$

$$-E(v,h) = \sum_{i,j} h_i v_j w_{ij} + \sum_i a_i h_i + \sum_j b_j v_j \tag{33}$$

where h indicates the top layer/hidden layer/first level features, and v indicates bottom layer/visible layer/acoustic features. The normalization constant Z is given by Equation 34:

$$Z = \sum_{v,h} e^{\sum_{i,j} h_i v_j w_{ij} + \sum_i a_i h_i + \sum_j b_j v_j} \tag{34}$$

We want to maximize the presence of visible layer, shown in Equation 35:

$$\max_w \log P(v) = \prod_{n=1}^{N} P(v^n) \tag{35}$$

To solve this maximization problem, we do gradient ascend directly on the log of likelihood other than $Q+H$ of EM algorithm or CRF shown in Figure 2.

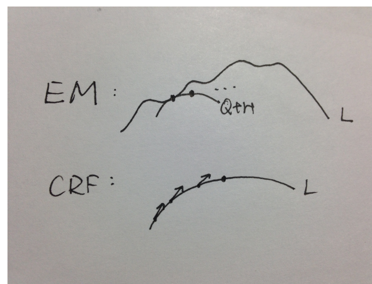


Figure 2: Compare to CRF and EM

$$
\begin{aligned}
\frac{\partial \log P(v^n)}{\partial w_{ij}} &= \frac{\partial[\log \sum_h \exp(-E(v,h)) - \log Z]}{\partial w_{ij}} \\
&= \frac{1}{\sum_h \exp(-E(v,h))} \sum_h h_i v_j \exp(-E(v,h)) - E_{P(h,v)}[h_i v_j] \\
&= E_{P(h|v)}[h_i v_j] - E_{P(h,v)}[h_i v_j]
\end{aligned} \tag{36}
$$

Equation 36is different from the gradient we have computed for CRF since in RBM we have v (acoustic) and h (first level features) while in CRF we have tags and words. The general term which contains all these three terms are show in Equation 37. Examples are shown in Table 2

$$
P(y|x) = \frac{1}{Z(x)} \sum_h \exp(w^T f(x,y,h)) \tag{37}
$$

Method	y	x	h
CRF	tags	words	–
RBM	v	–	h
speech recognition	words	acoustics	phone sequence

Table 2: Latent variable log-linear framework

6.1.1 Contrastive Divergence

The conditional probability of $P(h_i|v)$ is easy to compute, which makes it possible to get $E_{P(h|v)}[h_i vj]$ given the following: $P(h_i|v) \propto \exp(-E(v,h))$. However, the second term in Equation 36 is hard to compute, since we have to go through each combination of h and v. In this case, gibbs sampling is used to compute the conditional probability of $P(v_j|h)$. The process of Contrastive Divergence is as follows:

(1) Sample $P(h_i|v)$;

(2) Sample $P(v_j|h)$;

Our goal is to estimate $\frac{\partial \log L}{\partial w_{ij}}$. Even though this process is not completely accurate, it yields to be a fast and effective way to compute the gradient. Small differences would be made up in the following steps and process of the upper layers.

6.1.2 Stochastic Gradient Descent (SGD)

Stochastic gradient descent is a gradient descent optimization method for minimizing an objective function is written as a sum of differentiable functions. The process is taking one data at a time and then update all the parameters (also known as online learning).

$$
\begin{aligned}
\frac{\partial \log L}{\partial w} &= \frac{1}{N} \sum_n^N \frac{\partial \log P(x^n)}{\partial w} \\
&= E_{P(n)} \left[\frac{\partial \log P(x^n)}{\partial w} \right]
\end{aligned} \tag{38}
$$

6.1.3 Modeling Real-valued Data

Real-valued data such as MFCCs are more naturally modeled by linear variables with Gaussian noise and the RBM energy function can be modified to accommodate such variables given a Gaussian-Bernoulli RBM:

$$
E(v,n) = \sum_i \frac{(v_i - a_i)^2}{2\sigma_i^2} - \sum_j b_j h_j - \sum_{ij} \frac{v_i}{\sigma_i} h_j w_{ij} \tag{39}
$$

6.2 Back Propagation

Now we have trained with weight vectors between each layer indicated by γ, the objective function is:

$$P(l_{1:r}|v_{1:r}) = \frac{1}{Z(v_{1:r})} \exp\left(\sum_t \gamma^T \phi + \sum_t \sum_d \lambda_{l_t d} h_{td}\right) \tag{40}$$

Then we can compute the gradients:

$$\frac{\partial \log P}{\partial \gamma} = \phi - E[\phi] \tag{41}$$

$$\frac{\partial \log P}{\partial \lambda} = h_{l_t d} - E_{P(l|v)}[h_{ld}] \tag{42}$$

$$\frac{\partial \log P}{\partial w_{ij}} = \text{back-propagation} \tag{43}$$

7 Fast Decoding for HMMs

The problem with viterbi decoding is that it is too expensive. For instance, if we use n-gram language models, the number of states will be

$$T = V^{n-1} c, \tag{44}$$

- n: n-gram language model, n order
- V: vocabulary size
- c: number of states / pronunciation.

For HMM decoding, the complexity is NT^2, where N is the sequence length and T is the number of states. However, since $T = Vc$, so even for HMM, it's also impossible to decode efficiently.

7.1 Fast Decoding

Since decoding is equal to find the sequences with highest probabilities, there is jump probability from each node to another node. We can easily transform this decoding problem into a shortest path problem by defining the distance between the nodes as follows.
distance = cost = -$\log(Prob)$
In this section, we briefly discuss how to employ some of the shortest path algorithms for **fast** decoding.

7.1.1 Dijkstra's algorithm

Dijkstra's algorithm is one of the most famous algorithms for solving single-source shortest path on a graph. Algorithm 4 shows the general framework of Dijkstra's algorithm on a graph $G = (V, E)$, where Q is a

Algorithm 4: *Dijkstra's algorithm*

1: **repeat**
2: $v = pop(Q)$
3: **for** $v' : (v, v') \in E$ **do**
4: $push(Q, c(v) + c(v, v'), v')$
5: **end for**
6: **until** empty(Q)

priority queue (we can always initialize the Q by pushing the start node into this queue). The 4-th line of Algorithm 4 depends on the implementation of the priority queue. For most priority queue libraries, we cannot decrease the key directly due to the lack of support for keeping track of the position of the keys.

For the version of without decrease key, one node may have multiple entries in the stack. For the entries with larger cost than the current cost of the node, there is no need to check this entry again. Algorithm 5 avoids such unnecessary calculation by adding one checking condition for each entry.

Algorithm 5: *Dijkstra's algorithm (without decrease key)*

```
1: repeat
2:   v = pop(Q)
3:   if c > c[v] then
4:     then next
5:   end if
6:   for v' : (v, v') ∈ E do
7:     push(Q, c[v] + c(v, v'), v')
8:   end for
9: until empty(Q)
```

Complexity for Dijkstra's algorithm

- With decrease key: $O(E \log V)$ (Here we have $\log V$, because the update of priority queue typically takes $\log V$ operations (think about binary tree).)
- Without decrease key: $O(E \log E)$

By the way, if we implement a Fibonacci priority queue, then the complexity is $O(E + V \log(V))$. Do not use Fibonacci heap to implement a priority queue – it's not worth the savings.

The complexities for the two versions of the algorithm are indeed the same given that $E = O(V^2)$,

$$O(E \log(E)) = O(E \log(V^2)) = O(E \log(V))). \tag{45}$$

In HMM, $E = NT^2$, thus the complexity of Dijkstra's algorithm without decrease the key is $E \log(E) = NT^2 \log(NT^2)$. This complexity is even higher than Viterbi. However, Dijkstra's algorithm can handle positive loops in the graph. This may be useful in that some tasks may have loops. But not that, Dijkstra's algorithm cannot handle graphs with negative loops.

7.2 A* algorithm

In $A*$ algorithm, the cost function f is defined as

$$f(v) = c[v] + h(v), \tag{46}$$

where $h(v)$ is a heuristic function to estimate the cost from current node v to the destination node.

7.2.1 Admissible heuristic function

If the estimation heuristic function h never overestimates the cost to the goal, then we call this heuristic function admissible. the A* algorithm with an admissible heuristic function guarantees finding the optimal path. The design of heuristic function is a tradeoff between speed and accuracy.

7.2.2 A* in HMM decoding

In HMM decoding, we can always choose h to take the minimum of each column of the edges. That is

$$h(i, t) = \sum_{j=i}^{N} \min_{t,t'} c(j, t, j + 1, t'). \tag{47}$$

This can be precomputed and then used for the decoding algorithm. This heuristic function is definitely admissible. However, it does not take the states into consideration. So it may have little speedup for the decoding algorithm.

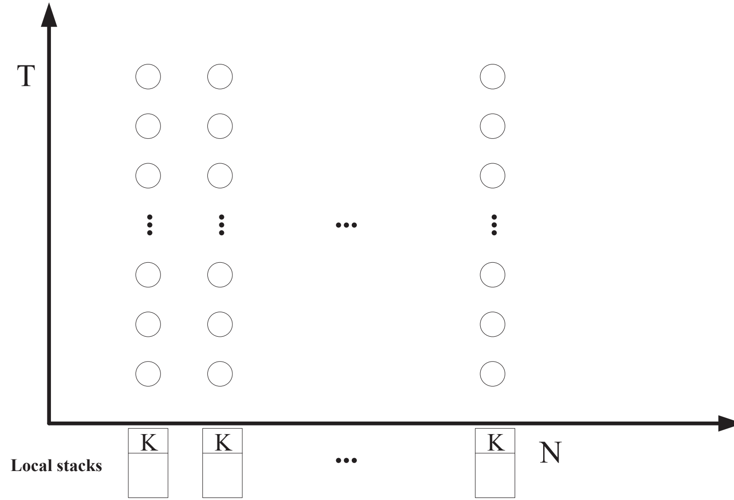


Figure 3: Multistack decoding

7.2.3 Alternatives

To speedup the decoding algorithm, we can use different strategies to make the heuristic function give some tolerable over-estimation of the cost.

- Weighted A*
  The cost function f can be estimated as $f(v) = c[v] + wh(v)$, where $w > 1$. In this way, the solution you find will satisfy $f_{sol} \leq w f_{opt}$.
- Inadmissible heuristic
  For inadmissible heuristic, A* cannot guarantee to find the optimal solution. For HMM decoding, we can define $h(i,t) = c(N-i)$, where i is the position of the sequence. If c is large enough, this may not give good results.

7.3 Multistack decoding

7.3.1 TOP K

Figure 3 shows the multistack decoding framework. For each position i in the sequence, we have a local stack for that position. The decoding proceeds over the local stacks sequentially. For each local stack, we only look the top K items. The pseudo-code is shown in Algorithm 6.

Algorithm 6: *Multistacking Decoding (Top K)*

1: **for** $i = 1$ to N **do**
2:   **for** $j = 1$ to K **do**
3:     $(c, v) = stack[i].pop()$
4:     $v = (i, t)$
5:     **for** t' **do**
6:       $stack[i+1].push(c + c(i, t, i+1, t'), (i+1, t'))$
7:     **end for**
8:   **end for**
9: **end for**

7.3.2 Beam Search

For multistack decoding with top K iterations, it may not work for some complicated NLP tasks. For instance, in speech recognition, we have to identify the start and end position of each word. For each start position, we may have several different choices of end positions. Therefore, we need to update several different stacks. In this case, we can update the stacks in an topological order. In this way, one stack may get updated several times, in different topological sequences. Algorithm 7 gives the pseudocode of multistack with beam search.

c^* in line 9 is the minimum cost in stack i. With a greater beam_width, we will have to finish more iterations.

Algorithm 7: *Multistacking Decoding (Beam Search)*

```
1:  partition into stacks
2:  for stacks in topological order do
3:    repeat
4:      (c, v) = stack[i].pop()
5:      v = (i, t)
6:      for t' do
7:        stack[i + 1].push(c + c(i, t, i + 1, t'), (i + 1, t'))
8:      end for
9:    until c > c* · beam_width
10: end for
```

8 Structured SVM

Structured SVM uses Max Margin as a metric for structured prediction. This note mainly talks about using Structured SVM for sequence labeling

In SVM for binary classication we mainly minimize the following term:

$$\min \frac{1}{2}\|w\|^2 + C\sum_n \xi_n \tag{48}$$

subject to the constraints:

$$y_n(w^T f(x_n) - b) \geq 1 - \xi_n \tag{49}$$

$$\xi_n \geq 0 \tag{50}$$

ξ_n are slack variables which describe the distance between data points and the margin. $\frac{1}{2}\|w\|^2$ is used for controlling model complexity and $\sum_n \xi_n$ is used for minimizing errors. In structured SVM for sequence labeling, we use the same optimization function, but subject to a different constraint:

$$w^T(f(x_n, y_n) - f(x_n, y)) \geq 1 - \xi_n \ \forall n, y \tag{51}$$

$$\xi_n \geq 0 \tag{52}$$

In fact, for equation (4) we usually use indicator function $I(y_n \neq y)$ to replace 1. If $y = y_n$, the left side will be 0, the right side will be $0 - \xi_n$, so we have $\xi_n \geq 0$ However, we are dealing with a sequence, so we need to add up the indicator value of each position, so we have:

$$w^T(f(x_n, y_n) - f(x_n, y)) \geq L(y_n, y) - \xi_n \ \forall n, y \tag{53}$$

$$L(y_n, y) = \sum_{i=1}^{|x|} I(y_{n,i} \neq y_i) \tag{54}$$

L is called loss function, which tells us how many positions have been tagged wrong. (Note that this is the same loss function that we used for MBR decoding for HMMs.)

At solution, we will get:

$$\xi_n = \max_y L(y_n, y) - w^T(f(x_n, y_n) - f(x_n, y)) \tag{55}$$

We just search for all possible y, and pick up the biggest value for ξ_n

8.1 Training

When we are doing structured SVM, we usually have many sentences in the training set, so we have to sum up the loss function of all sentences:

$$\min \frac{1}{2}\|w\|^2 + \frac{C}{N}\sum_{n=1}^{N}\xi_n \tag{56}$$

N is used for regularize the derivative, just like the $\frac{1}{2}$ before $\|w\|^2$. This does not matter because C is a constant whose value can update according to N

We take derivative with respect to w:

$$\frac{\partial F}{\partial w} = w + CE\left[\frac{\partial}{\partial w}\xi_n\right] \tag{57}$$

Substituting the value of ξ_n from eqn. (55) and differentiating, we get:

$$\frac{\partial F}{\partial w} = w + C(f(x_n, \hat{y}) - f(x_n, y_n)) \tag{58}$$

where $\hat{y}$ is the prediction sequence. So from this we can make some update on the weight, so given each sequence y_n and the prediction sequence $\hat{y}$, we make some update on weight w:

$$w \leftarrow w - \eta(w + C(f(x_n, \hat{y}) - f(x_n, y_n))) \tag{59}$$

η is the learning rate, which we can tune on the development set.

8.2 Decoding

Like in perceptron, we want to find the prediction sequence which would maximize the following score:

$$w^T f(x, y) \tag{60}$$

In structured SVM, we use a Loss Integrated Decoding method, which is different from perceptron in that it uses the loss function value as a part of the score. We assume that:

$$f_j(x, y) = \sum_{n=1}^{|x|} f_j(x, y_n, y_{n-1}) \tag{61}$$

$$L(y_n, y) = \sum_{i=1}^{|x|} l(x, y_{n,i}, y_i) \tag{62}$$

so we use dynamic programming to decode, for t, t':

$$\delta(i, t) = \max\{\delta(i, t), \delta(i-1, t') + l(x_n, y_{n,i}, t) - w^T(f(x, y_{n,i-1}, y_{n,i}) - f(x_n, t', t))\} \tag{63}$$

8.3 Dealing with certain problems

Corners of structured SVM are not differenciable, we can adopt a sub-gradient for max margin, when it comes to the corner, we can use the gradient between the gradient of intersecting hyperplane.

There are certain situations where the hyperplane for the new data will be outside the activating area, so this kind of hyperplane will never get activated. Also, there are situations where the parameters will bounce back and forth, we can fix this by make the update steps smaller and smaller, e.g., we can adopt a learn rate:

$$\eta = C\frac{1}{t} \tag{64}$$

It will reach strictly convex if the step size $\eta <$ concavity, which makes sure that it will converge. We go back to optimization problem:

$$\min_{w,\xi,b} \frac{1}{2}\|w\|^2 + C\sum_n \xi_n \tag{65}$$

Subject to the following constraint:

$$y_n(w^T f(x_n) - b) \geq 1 - \xi_n \; \forall n \tag{66}$$

$$\xi_n \geq 0 \tag{67}$$

So at solution we get:

$$\xi_n = \max\{0, 1 - y_n(w^T f(x_n) - b)\} \tag{68}$$

There will be one corner for each data point. Let's take a look at the different encoding and decoding metrics adopted by different models:

Table 3: Different metrics

	0/1	Loss function
$P(y\|x)$	CRF/HMM,Viterbi	CRF/HMM,MBR (decoding)
$\max w^T f$	perceptron	Max Margin (training)

8.4 Dual representation for structured SVM

When dealing with binary SVM, we can solve parameter w using Lagrange multiplier, the result is as follows:

$$w = \sum_n \alpha_n f(x_n) \tag{69}$$

By eliminating w from the optimization problem we can get:

$$\min \sum_n \alpha_n - \frac{1}{2}\sum_{n,m} \alpha_n \alpha_m f(x_n)^T f(x_m) \tag{70}$$

subject to the constraint

$$0 < \alpha_n < C \tag{71}$$

When calculating $f(x_n)^T f(x_m)$, we can use $k(x_n, x_m)$ to replace it and make use of kernel functions. Accordingly, the dual representation for the structured SVM is as follows:

$$\min \sum_{n,y} \alpha_{n,y} L(y_n, y) - \frac{1}{2}\sum_{n,m,y,y'} \alpha_{ny}\alpha_{my'} \Delta f(x_n, y)^T \Delta f(x_m, y') \tag{72}$$

subject to the constraint

$$0 < \alpha_{ny} < C \tag{73}$$

So from the left side of the constraint we can get:

$$\alpha_{ny} > 0 \Leftrightarrow w^T \Delta f(x_n, y) = L(y_n, y) - \xi_n \tag{74}$$

In the dual representation of structured SVM, there are too many constraints on variables, so it is difficult to calculate the parameter directly, we use the following method to calculate the parameter α, for each n:

$$\hat{y} = LossIntegratedDecoding(x) \tag{75}$$

$$Update\ \alpha_{n\hat{y}} \tag{76}$$

However, in NLP, we generally do not use kernel functions to map the input into a higher dimension since we already have enough good features.

9 Context Free Grammars

$G = (S, N, T, P)$
S - start symbol
N - non-terminals
T - terminals
P - production rules
Language(L) - set of strings the grammar generates ($L(G) \subseteq T^*$)
example:

$S = \{S\}$
$N = \{A, B, C\}$
$T = \{b, c\}$
$P = \{S \rightarrow A,$
$A \rightarrow BC,$
$B \rightarrow bB,$
$B \rightarrow \epsilon,$
$C \rightarrow cC,$
$C \rightarrow \epsilon\}$
$L = b^* c^*$

9.1 Regular Languages

Regular languages are context free grammars with an additional constraint that rules always have the form:

$A \rightarrow aB$ or
$A \rightarrow \epsilon$

We can see that HMMs can be modeled like a regular language where the current state is the non-terminal on the left side of the production rule. The next state is the non-terminal on the right side, and the emission is the terminal on the right side.

9.2 Finite Languages

Finite languages are regular languages that contain a finite number of strings.

An example of a regular language that isn't finite: a^*
An example of a CFL that isn't regular: $a^n b^n\ \forall n$
Are finite/regular/context-free languages countable? answer is yes to all three.
Why? All languages must have a finite number of production rules.
Is the number of languages countable? No, see diagonalization.

9.3 Probabilistic CFG

each rule has a probability.
ex.

.7	$S \to A$
.3	$S \to a$

$\sum_B P(A \to B) = 1.\ \forall A$ (sum over the left hand side = 1)

HMMs are probabilistic regular languages. The transition probabilities are the rule probabilities.

$A \to aA$
$A \to bA$
$A \to \epsilon$
$P(A \to aA) = P(a|A)P(A|A)$

9.4 Probabilistic CFGs again

$$P(derivation) = \prod_{r \in d} P(r)$$

$$P(string) = \sum_d P(d) \text{ where } d \text{ results in } s$$

$$= \sum_d \prod_r P(r)$$

$\sum_s P(s) = 1$ The sum of the probability of all the strings in the language is 1

This is always true for finite languages.

What about regular languages?

p	$S \to aS$
$1-p$	$S \to a$

$P(a^n) = p^n(1-p)$
$\sum_n p^n(1-p) = 1$

What about CFLs?

p	$S \to SS$
$1-p$	$S \to a$

E[# of $S's$ after n steps]

n	$E[\#S]$
1	1
2	$2p$
3	$4p^2$
...	...
n	$2^{n-1}p^{n-1}$

in general:
let N be a vector of the number of each nonterminal
what is $E[N]$ after n steps?
C is our transition matrix where $C_{ij} = E[\text{count of } N_j \text{ in } B, N_i \to B]$
Let S be the start vector $([1, 0, 0, ...])$

n	$E[N]$
1	S
2	$C^T S$
...	...
n	$(C^T)^{n-1} S$

if $\lambda(C) < 1$ (if the largest eigenvalue is less than 1) $\Rightarrow \sum_s P(s) = 1$

$P(w, ...) =$ probability that the first word in a string $= w$.
$P(w, ...) = \sum_s P(s)$. s.t. $s_1 = w$
$P(w, ...) = \sum_{S'} P(S \to wS')$ doesn't work for CFLs.
$L =$ non-terminals to non-terminals
$U =$ non-terminals to terminals

$P(W \to w) = U^T S + U^T L^T S + ... + U^T (L^T)^n S$
$L^* = I + L^T + ... = (I - L^T)^{-1}$
$P(W \to w) = U^T L^* S$

10 Probabilistic Parsing

We assume that natural language has a Context Free Grammar (CFG). Given an input sentence $x_1 \ldots x_N$, we want to find a parse tree for it. Are Markov assumptions, Viterbi or forward-backward applicable to trees as well? We will see that anything we do with Hidden Markov Models (HMM) we could also do with CFGs.

10.1 Viterbi Decoding For Parsing

If you recall from previous lectures, one thing we did was Viterbi decoding for finding the most probable tagging for a given sequence of words:

$$y^* = \operatorname*{argmax}_{\vec{y}} P(\vec{x}, \vec{y}) \tag{77}$$

Here the counterpart is:

$$tree^* = \operatorname*{argmax}_{tree} P(sentence, tree) = \prod_{r \in tree} P(r) \tag{78}$$

Where $tree^*$ is the optimal parse tree and r is a production rule used in building the tree. So a parse tree is derived as a result of sequence of rule applications, and its probability would be the product of the probability of all such rules applied. For example, in Figure 4 the probability of the generated parse tree can be computed by multiplication of the production rules used: P(S $\to$ NP VP) P(VP $\to$ VBZ PP) P(NP $\to$ NN) P(NN $\to$ Time) . . .

The way we used to solve the equation 77 was by dynamic programming, where we filled in a 2D lattice in the form of $\delta(i, t)$ in which i was the index in $\vec{x}$ input sequence and t was the tag assigned to $\vec{x}[i]$. Here we solve equation 78 by dynamic programming likewise, by filling in a 3D lattice as follows:

$$\delta(i, j, t) = \max_{s \in all\ subtrees} P(s) \tag{79}$$

Where i is the begin index, j is the end index of the generated substring of input $\vec{x}$, and t is a non-terminal symbol. $root(s) = t$ and yeilds(s)=$\vec{x}_i^j$, which means the root of s subtree is t and s yields the i up to j index of $\vec{x}$. Figure 5 shows the subtree s, rooted at t.

δ is filled with the probability of the best subtrees with attributes i, j, and t and it is a max product, so we actually can use dynamic programming, as shown in Algorithm 8.

This algorithm is $O(N^3T^3)$, where N is the length of the input sentence and T is the number of non-terminal symbols in the grammar. So decoding for CFG has worse time complexity than HMM.

Now the question is how to get the probabilities of the production rules which we used in the dynamic programming algorithm. One simple idea is to compute them by counting all the occurrences of certain

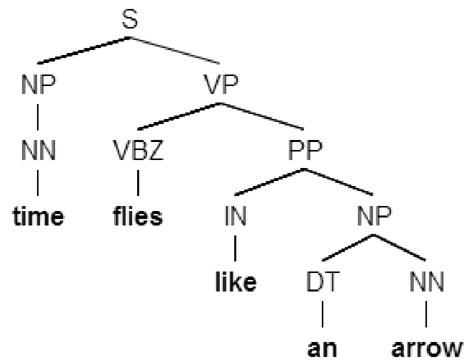


Figure 4: The parse tree of the sentence “Time flies like an arrow”

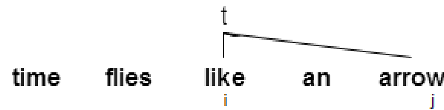


Figure 5: What δ means

Algorithm 8: Viterbi Decoding for Parsing

Input: grammar G with probabilities for production rules, $X_{1...N}$

1: allocate δ with size $|T| * (N-1)^2$ //T is the set of non-terminals and N is the number of words in the sentence.
2: $\delta = -\infty$
3: **for** $i = 1$ to N **do**
4:     **for** $t \in T$ **do**
5:         $\delta(i, i+1, t) = P(t \to x_i)$ //the probability for production rule $t \to x_i$
6:     **end for**
7: **end for**
8: **for** $span = 2$ to N **do**
9:     **for** $i = 0$ to $N - span$ **do**
10:         $k = i + span$
11:         **for** $j = i + 1$ to $k - 1$ **do**
12:             **for** A, B, C $\in$ non-terminal symbols in G **do**
13:                 $\delta(i, k, A) = \max\{\delta(i, k, A)$, P(A $\to$ BC) $\delta(i, j, B)\ \delta(j, k, C)\}$
14:             **end for**
15:         **end for**
16:     **end for**
17: **end for**
18: **return** $\delta(1, N+1, S)$ //S is the grammar symbol for the sentence.

---

rule in our corpus. Another way is to use Perceptron algorithm for learning the weights associated with features and use the following equation:

$$\delta(i,k,A) = \max\{\delta(i,k,A), f(A \to BC, i, j, k) + \delta(i,j,B) + \delta(j,k,C)\}$$

where $f_1(A \to BC, i, j, k) = \log P(A \to BC)$.

In the above dynamic programming, we have made a simplification: we have assumed our grammar is in Chomsky Normal Form (CNF). The production rules in a CNF should be one of the two following forms:

$$\begin{cases} A \to BC \\ A \to a \end{cases}$$

However, we know that the Penn TreeBank's grammar is not in CNF form, e.g., we have VP → VBP NP PP rule which is in the form $A \to BCD$. Good thing is that every CFG can be transformed into an equivalent CNF grammar. Here is how:

1. Binarize: In this step, we aim at removing any rules which have more than two non-terminals on the right hand side. If the rule is $A \to BCD$, we replace it with $A \to BX$ and $X \to CD$
2. ϵ-removal: No rules going to ϵ should be in the grammar. If we have $A \to epsilon$ and $B \to CAD$, we will replace them with $B \to CD$. And we keep moving with this step, until no more rules going to ϵ is remained.
3. Unary propagation: No rule which goes to a single non-terminal symbol should remain in the grammar. If we have $X \to Y$ and $Y \to z$, we replace them with $X \to z$ –where z is a terminal symbol. This is also called chain collapsing in the grammar.

What would happen to the rule probabilities if we convert the grammar to CNF? In NLP, we do ϵ-removal before counting the rules. For binarizing assume we have $P(A \to BCD) = p$, then we'll have $P(A \to BX)$=p and $P(X \to CD) = 1$ (since multiplication of the probabilities of replacement rules should match the original rule's probability). For unary propagation, we insert this line before the for loop over j in the algorithm 8 : $\delta(i,k,A) = \max\{\delta(i,k,A), P(A \to B)\delta(i,k,B)\}$.

10.2 Posterior Decoding For Parsing

Another thing we did with HMMs was posterior decoding, in which at each position –independently– we computed the most likely state that gave us the i'th index of the sequence: $\text{argmax}_{P(x,y_i)}$. We solved this by forward-backward algorithm. Here is how we define α, β, and ξ for posterior decoding in CFG.

$$\alpha(i,k,t) = P(S \Rightarrow^* x_1 \dots x_{i-1} t x_k \dots x_N) \tag{80}$$

$$\beta(i,k,t) = P(t \Rightarrow^* x_i \dots x_{k-1}) \tag{81}$$

$$\xi(A \to BC, i, j, k) = \frac{1}{Z}\alpha(i,k,A)\beta(i,j,B)\beta(j,k,C)P(A \to BC) \tag{82}$$

Figure 6 depicts what α and β mean for us in a subtree. As you can see, α is the outside and β is the inside probabilities. The forward-backward algorithm in CFGs is also called outside-inside algorithm. Figure 7 shows β subtrees of $\beta(i,j,B)$ and $\beta(j,k,C)$, with outer $\alpha(i,k,A)$ corresponding to equation 82.

Note that in CFG $\sum\limits_{A,B,C} \xi \leq 1$, however, in HMM we had $\forall i : \sum_{t,t'} \xi(i,t,t') = 1$. This is because of the fact that in CFG we do not have to have a production rule going from say i to k. Algorithm 9 shows how we fill in α and β.

How is the α (outside probability) related to forward probability we had in HMM? And how is β (inside probability) related to backward probability in HMM? The answer reveals by converting a HMM into a CFG: if we are at a state A going to state B with emission a, we could write a production rule like $A \to aB$. Figure 8 is an example CFG for a HMM, constructed for each state and transition as explained earlier.

Now considering this CFG, outside probability rooted at B would be the selected part on the tree. If we go back to the HMM counterpart, this part is obviously the forward probability.

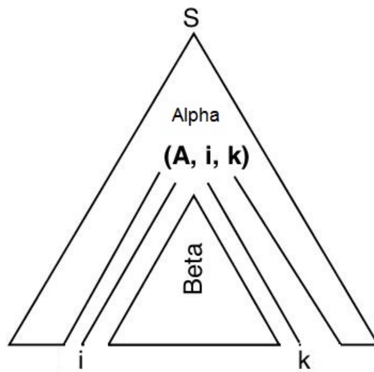


Figure 6: What alpha and beta mean in a parse tree

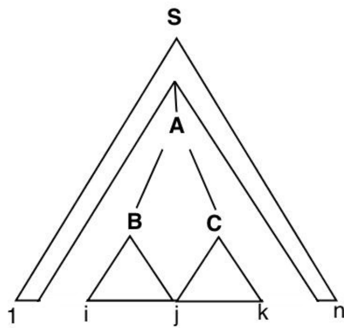


Figure 7: Alpha-beta can be viewed as considering all ways of drawing a tree structure like this.

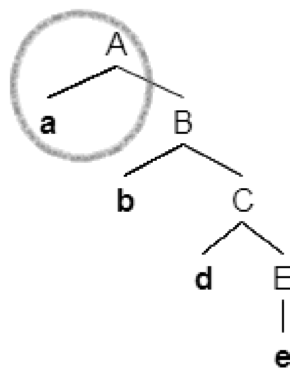


Figure 8: Conversion of a HMM into CFG equivalent

Algorithm 9: Inside-Outside Algorithm

Input: grammar G with probabilities for production rules, $X_{1...N}$

1: allocate α and β with size $|T| * (N-1)^2$
2: $\alpha(S, 1, N) = 1$
3: **for** $i = 1$ to N **do**
4:     **for** $t \in T$ **do**
5:         $\beta(i, i+1, t) = P(t \rightarrow x_i)$
6:     **end for**
7: **end for**
8: **for** $span = 2$ to N **do**
9:     **for** $i = 0$ to $N - span$ **do**
10:         $k = i + span$
11:         **for** $j = i + 1$ to $k - 1$ **do**
12:             **for** $A, B, C \in$ non-terminal symbols of G **do**
13:                 $\beta(i, k, A)$ += $\beta(i, j, B)\beta(j, k, C)P(A \rightarrow BC)$
14:             **end for**
15:         **end for**
16:     **end for**
17: **end for**
18: **for** $span = N$ to 1 **do**
19:     **for** $i = 0$ to $N - span$ **do**
20:         $k = i + span$
21:         **for** $j = i + 1$ to $k - 1$ **do**
22:             **for** $A, B, C \in$ non-terminal symbols of G **do**
23:                 $\alpha(i, j, B)$ += $\alpha(i, k, A)\beta(j, k, C)P(A \rightarrow BC)$
24:                 $\alpha(j, k, C)$ += $\alpha(i, k, A)\beta(i, j, B)P(A \rightarrow BC)$
25:             **end for**
26:         **end for**
27:     **end for**
28: **end for**
29: **return** α and β

10.3 Posterior Decoding in HMM

Recall how we solved MBR in HMM. We were trying to solve

$$\max_{\hat{y}} E_{P(y|x)} \left[\sum_{i=1}^{N} I(\hat{y}_i = y_i) \right]$$

where $\hat{y}$ was our solution, and y was the true tags. We pushed the expectation and the max inside of the sum, and reduced this to

$$\left[\operatorname*{argmax}_{\hat{y}_i} P(\hat{y}_i \mid x) \right]_{i=1}^{N}$$

Then, we calculated $P(\hat{y}_i \mid x)$ using forward-backward probabilities, and finally got

$$\left[\operatorname*{argmax}_{\hat{y}_i} \alpha(i, \hat{y}_i)\beta(i, \hat{y}_i) \right]_{i=1}^{N}$$

10.4 Posterior Decoding in Parsing

We use a similar formula when solving MBR in Parsing. Now, we try to maximize

$$\max_{\hat{t}} E_{P(t|x)} \left[\sum_{n \in \hat{t}} I(n \in t) \right]$$

Here, we look at all possible trees $\hat{t}$, and give points if the nodes n are in the real tree, t. We can again simplify by pushing the expectation inside of the sum:

$$\max_{\hat{t}} \sum_{n \in \hat{t}} P(n \in t)$$

We see that this situation is more complicated than in HMM, because we don't know what the structure of the tree will be, so we can't simplify it as much as we did before.

In addition, what we are maximizing is the precision of the data, and not the F-measure. Unfortunately, it is impossible to maximize F-measure using dynamic programming.

Now, we look at the definition of our nodes. Each node looks like $n = (A, i, j)$, where A is the non-terminal at the beginning of the node, i is the starting position of the node, and j is the ending position of the node.

Using this, we can calculate our $P(n \in t)$ as

$$P(n \in t) = \frac{\alpha(A, i, j)\beta(A, i, j)}{\beta(S, 1, N)}$$

where α and β are the outside and inside probabilities of our tree. Note that we normalize using $\beta(S, 1, N)$ becuase

$$P(n \in t \mid x) = \frac{P(n \in t, x)}{P(x)}$$

10.5 Applying dynamic programming

From here, how do we build the most likely tree? In HMM, we could simply look at each position and decide on the most likely tag using forward-backwards. Here, it is more complicated, because we need to end up with something that has a valid tree structure. So, we use dynamic programming, and the same method we used to calculate δ. Then, we used probabilistic rules, so we had a form that looked like

$$\max_{t} P(t) = \max_{t} \prod_{r \in t} P(r)$$

Algorithm 10: δ Calculation in Parsing

Input: $w_{1...I}$, $X_{1...N}$

1: allocate a table δ with size $|T| * N * N$
2: set $\delta = -\infty$
3: **for** span $= 2$ to N **do**
4:   **for** $i = 0$ to $N -$ span **do**
5:     $k = i +$ span
6:     **for** $j = i + 1$ to $k - 1$ **do**
7:       **for** A, B, C **do**
8:         $\delta(A, i, k) = \max\{\delta(A, i, k), \delta(B, i, j) + \delta(C, j, k) + \alpha(A, i, k)\beta(A, i, k)\}$
9:       **end for**
10:     **end for**
11:   **end for**
12: **end for**
13: **return** δ

This time, we want to sum up probabilities of nodes, but the algorithm is very similar.

Using this algorithm does a very good job of increasing F-measure, even though it doesn't directly maximize it because what it is maximizing is very closely related to F-measure. In fact, if the grammar is in Chomsky normal form, F-measure, Precision, and Recall will all be equal, and this algorithm will maximize it.

This method of parsing is called CYK Parsing (named after its creators, Cocke, Younger, and Kasami). It is also called bottom-up parsing, chart parsing, and forest parsing. This is because the calculated δ is sometimes called a chart, a forest, a packed forest, a hypergraph, or an and-or graph, due to the way that two lower nodes are connected to a single upper node at each step.

This is also called a Weighted Deduction System, because we can write the way two lower nodes combine into a higher node as a deduction:

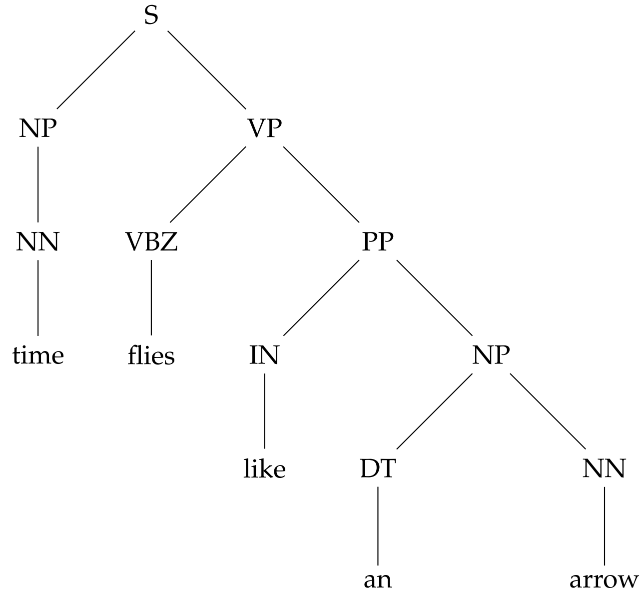
$$\frac{(B, i, j), (C, j, k)}{(A, i, k)}$$

We can also represent the way probabilities are combined:

$$\frac{P_1 : (B, i, j), P_2 : (C, j, k)}{P_1 + P_2 + \alpha\beta : (A, i, k)}$$

10.6 Extracting data from sentences

We need a way to extract our probabilities from a set of data. First, consider our example tree:



Given a large number of these sentences from a dataset, we could simply count up the number of each rule, and calculate probabilities that way. For example, for the rule S $\rightarrow$ NP VP, we would get

$$P(\text{S} \rightarrow \text{NP} \;\; \text{VP}) = \frac{\#(\text{S} \rightarrow \text{NP} \;\; \text{VP})}{\sum_{\beta} \#(\text{S} \rightarrow \beta)}$$

We can use this, but it will probably give us an F-measure of about 75%, where state-of-the-art parsers get about 91%. Obviously, we need to do something more.

What is wrong with this approach? The words in the sentence affect the way that different rules are applied. For example, in this sentence, the word “flies” is intransitive, so of the two choices:

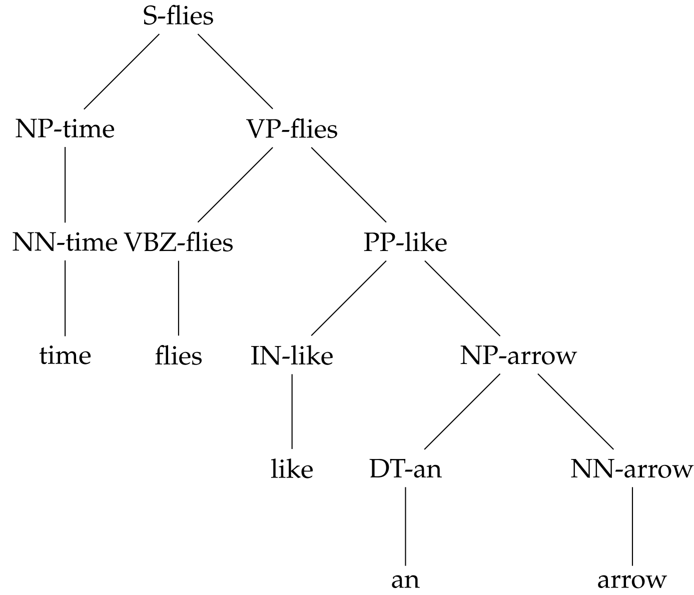
$$P(\text{VP} \rightarrow \text{VBZ} \;\; \text{NP} \;\; \text{PP})$$

$$P(\text{VP} \rightarrow \text{VBZ} \;\; \text{PP})$$

the second one will be higher.

10.7 Lexicalization

To account for this, we do something called Lexicalization. This marks each node in our tree with a specific word. For each rule, one of the child nodes is chosen as the parent node’s “head child”, and the word associated with that child node is chosen as the word associated with the parent. So, when we lexicalize our tree we end up with:



Now, we look at our probabilities. We can involve these new lexicalized nonterminals in two ways. First, we can add the head words to the left side of the rule:

$$P(\text{VP-flies} \rightarrow \text{VBZ}\;\; \text{PP})$$

Going from no head words to this results in $|V|$ times as many rules as before. We can also add the head words to the right side of the rule:

$$P(\text{VP-flies} \rightarrow \text{VBZ-flies}\;\; \text{PP-like})$$

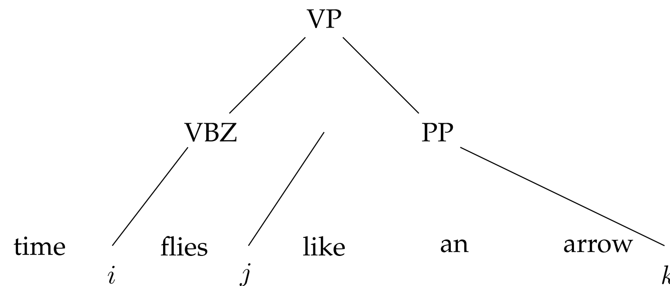
Now, we end up with $|V|$ times the length of the rule times as many rules. This ends up being too specific. However, we would like to use those words, so we add in the words on the right hand side by doing something like

$$P(\text{VP-flies} \rightarrow \text{VBZ}\;\; \text{PP})P(\text{PP} = \text{like} \mid \text{VP} = \text{flies})$$

(note that since the VBZ is where the VP gets its head word from, we don't have a probability involving that) In this case, we have $|V|^2$ times as many rules. This is also called bilexicalization.

10.8 Parsing with features

When we are parsing, our features can depend on several parts. They can depend on the sentence, the three nonterminals A, B, and C, and the start, split and end positions, i, j, and k:



So, we might have features that look like

$$I(\text{VP} \rightarrow \text{VBZ}\;\; \text{PP} \wedge X_i = \text{flies})$$

or

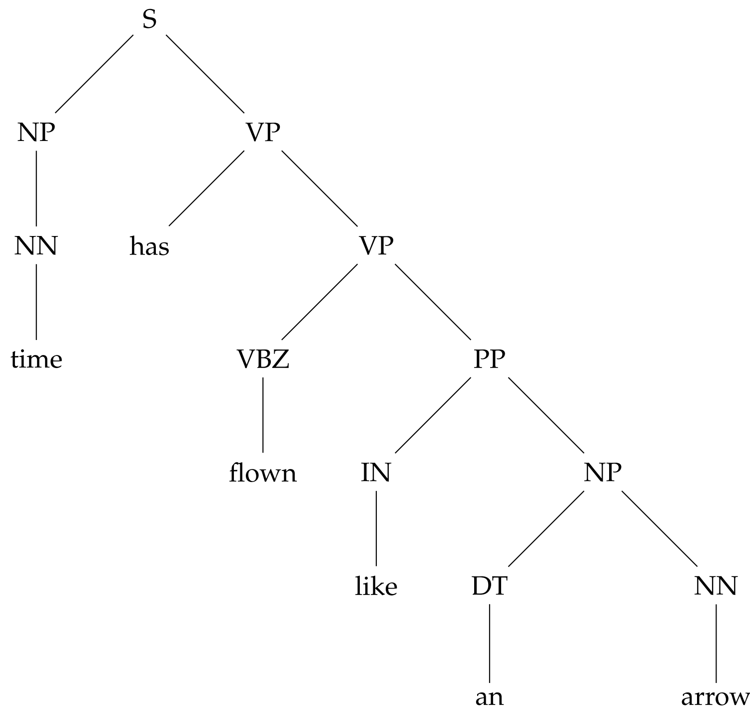
$$I(\text{VP} \to \text{VBZ PP} \land X_{i-1} = \text{time})$$

However, our features cannot depend on the head words of the nodes, just by calculating using i, j, and k. Instead, we simply add more nonterminals and parse using those instead.

We know that parsing is $O(N^3T^3)$. When we lexicalize, we end up with $T' = TV$ non-terminals (the number of previous non-terminals times the size of our vocabulary). So, we might think that the time to parse would become $O(N^3T^3V^3)$, but we only have to look at the vocabulary once when we lexicalize, so it becomes $O(N^3T^3V)$, and if we only consider the words that are in the sentence itself, $V = N$, so we end up with $O(N^4T^3)$. Likewise, if we do bilexicalization, we get $O(N^5T^3)$.

10.9 Better Lexicalization

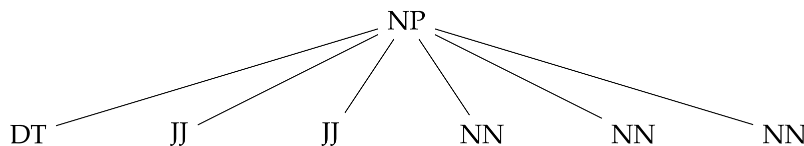
What if we change the sentence to



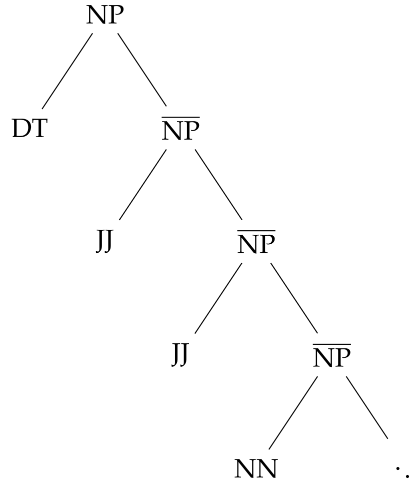
The head word of the sentence ends up being "has" now, which is the conjugated word in the sentence. This is useful for determining how to match singluar/plural with the nouns, but it no longer lets us know what the main "action" verb of the sentence is. So, sometimes people have multiple head words, such as a syntactic and a semantic head word.

10.10 Binarization or Markovization

Some rules are very spread out:



This is a problem, because these rules would have too many probabilities associated with them. So, we binarize the trees before counting rules, adding intermediate nodes that make the tree binary:



We then parse with the resulting grammar, and then remove the intermediate nodes as a postprocessing step.

Splitting the production into several steps results in smoothing, allowing us to recognize large, flat productions that have not been seen in the training data.

Note that binarizing before counting rules is different from converting to Chomsky Normal Form after counting rules. In the latter case, we would convert

$$P \qquad \text{A} \rightarrow \text{B C D}$$

into two rules

$$\begin{aligned} P &\qquad \text{A} \rightarrow \text{X D} \\ 1 &\qquad \text{X} \rightarrow \text{B C} \end{aligned}$$

here, the overall probability remains the same, and there is no smoothing effect.

11 Charniak Parser

The Charniak [2] parser consists of 3 stages:

1. Coarse-grained parser: filters out low probability constituents (A, i, j) using a crude (non-lexicalized) grammar.
2. Fine-grained parser: splits the promising coarse grained constituents into more fine-grained bilexicalized constituents, and generates a list of K-best parse trees using the fine-grained grammar.
3. Discriminative Reranker: reranks the K-best list using a MaxEnt model, and attempts to find the best parse tree.

The details of these 3 stages are described below.

11.1 Coarse-grained Parser

The computational complexity of parsing a sentence using a bilexicalized PCFG is $O(T^3N^5)$, which is significantly higher than that for non-lexicalized grammars: $O(T^3N^3)$. Charniak parser first applies a coarse-grained non-lexicalized parser, where the features are simple PCFG rules (e.g. S→ NP VP). For each constituent (A, i, j), the coarse parser estimates the marginal probability $P[(A, i, j)|X)]$, given the sentence X using the inside-outside algorithm:

$$P[(A, i, j)|X)] = \frac{\alpha(A, i, j)\beta(A, i, j)}{\beta(S, 1, N)} \tag{83}$$

[2] The software can be downloaded from http://bllip.cs.brown.edu/resources.shtml

The parser removes all the constituents having marginal probability $P[(A,i,j)|X)]$ below a threshold. The fine grained parser only considers the states that are refinements of the remaining coarse-grained states. Therefore, the coarse-grained pruning drastically reduces the dynamic programming state space for the fine-grained parser.

11.2 Fine-grained Parser

A fine-grained parser is applied on the pruned forest to generate the K-best list. The dynamic programming algorithm for 1-best parse tree can be easily extended to return K-best parse trees by remembering the K best ways to generate each constituent (A,i,j). So each entry in our dynamic programming chart $\delta(A,i,j)$ can be considered as a list of size K. The naive algorithm for general K-best parsing is presented in Algorithm (1), which requires $O(T^3N^3K^2)$ operations. We can reduce it to $O(T^3N^3K\log K)$ using Dijkstra's algorithm and maintaining a heap data structure at each chart entry to sort the K best ways to generate that constituent.

Algorithm 11: The dynamic programming for K-best parsing (naive implementation).

Input: Grammar G with probabilities for production rules, Input sentence $X_{1\ldots N}$
1: Allocate a table δ with size $|T|*N^2*K$, where T is the set of non-terminals.
2: Initialize $\delta[A,i,j][m]=-\infty$, $1\le i<j\le N, 1\ \le m\le K$.
3: **for** $i=1$ to $N-1$ **do**
4:   **for** $A\in T$ **do**
5:     $\delta[A,i,i+1][1]=P(A\to x_i)$
6:   **end for**
7: **end for**
8: **for** i,j,k **do**
9:   **for** $A,B,C\in T$ **do**
10:     **for** $m,n\in 1$ to K **do**
11:       $\delta[A,i,k]$.push $[\delta[B,i,j][m].\delta[C,j,k][n].P(A\to BC)]$
12:     **end for**
13:   **end for**
14: **end for**
15: **return** δ

Charniak's fine-grained parser is similar to the K-best parser in Algorithm (1), except it ignores the low probability coarse grained states. Each chart entry corresponds to the coarse-grained states (A,i,j). The algorithm is presented in Algorithm (2).

Algorithm 12: The fine-grained parsing (naive, no priority queues).

1: **for** i,j,k **do**
2:   **for** $A,B,C\in T$ **do**
3:     **if** $P(A,i,k)<\tau$ **then**
4:       next;
5:     **end if**
6:     **for** $m,n\in 1$ to K **do**
7:       $\text{item}_1\leftarrow \text{stack}[B,i,j][m]$
8:       $\text{item}_2\leftarrow \text{stack}[C,j,k][n]$
9:       stack$[A,i,k]$.push[combine(item_1, item_2, $A\to BC$)]
10:     **end for**
11:   **end for**
12: **end for**
13: **return** stack

Similar to the K-best parser, we can apply Dijkstra's algorithm to speed up (Algorithm (3)).

Algorithm 13: The fine-grained parsing (Dijkstra).

```
1:  for i, j, k do
2:    for A, B, C ∈ T do
3:      if P(A, i, k) < τ then
4:        next;
5:      end if
6:      for m ∈ 1 to K do
7:        (item_1, item_2) ← PRIO.pop()
8:        stack[A, i, k].push[combine(item_1, item_2, A → BC]
9:      end for
10:   end for
11: end for
12: return stack
```

11.3 Reranker

Finally the parse trees in the K-best list are reranked using a MaxEnt reranker and more sophisticated features. Let f be the feature vector and w be the associated weights that we want to learn. The training data consists of a sequence of sentences $\{x_i\}$ and corresponding correct parse trees $\{y_i\}$. The optimization problem to solve:

$$\hat{w} = \max_{w} \prod_{i} \frac{1}{Z_i} e^{w^T f(y_i, x_i)} \tag{84}$$

There is no guarantee that the K-best list contains the correct parse, so we use the set of highest scoring trees in the K-best list. Let $y_+(x)$ be the set of trees in the K-best list that have maximum score: $y_+(x) = \{y : \text{score}(y) = \max_{y' \in y(x)} \text{score}(y')\}_n$. Here, $y(x)$ is the set of all possible parse trees with yield x. The gradient for this loss function is:

$$\frac{\partial L}{\partial w} = E_{p(y|y_+(x)}[f(y, x)] - E_{p(y|y(x))}[f(y, x)] \tag{85}$$

Since the reranking step is executed only on the K-best list, we can afford to use more complex features (even some non-local features that do not work with dynamic programming). For example, for each constituent, we consider two types of head words: lexical heads and functional heads. The lexical head of a VP is the main verb, while the functional head is the auxiliary verb. For example, for the VP "is going", the lexical head "going" that tells us that the subject is a person, and the functional head "is" tells us that the subject is third person singular. These features allow dynamic programming, but computationally expensive and require $O(T^3 N^7)$ operations to parse each sentence. There are features that tells the reranker to prefer right-branching parse trees. These features are non-local and do not work with dynamic programming.

12 Dependency Parsing

By far the parsing algorithms we have learned parse a sentence based on phrase structure rules (in terms of constituency and adjacency) which is called phrase-structured parsing. For example VP → VBZ PP. Figure 9 depicts phrase-structured lexicalized parsing for the sentence "time flies like an arrow". With dependency parsing, we want to be able to parse with non-adjacency and not rely on constituents.

Given a sentence, assume each word to be a node and then draw an edge from each node to its head. Figure 10 is an example of such a graph for the sentence "time flies like an arrow". It is apparent that this graph is a connected, single-headed, and acyclic; thus, it makes a tree which we call dependency tree.

The dependency tree representation for the same sentence is shown in Figure 11. As you can see, some additional information about the relation between nodes has been provided on the edges. This is because

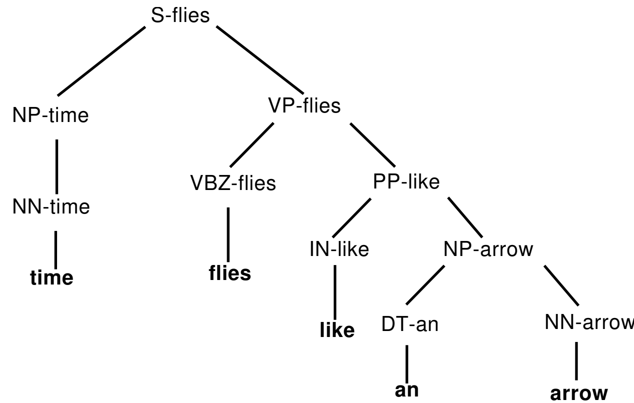


Figure 9: Phrase-structured parsing for the sentence "time flies like an arrow"

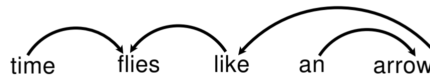


Figure 10: Dependency graph representation for the sentence "time flies like an arrow"

of the fact that going from phrase-structured parsing to dependency parsing results in losing such information. However, the information reflected via edges is not enough for going the other way back exactly to the phrase-structured tree. Dependency parsing is a lot easier than phrase-structure parsing. Since constructing a dependency tree does not require building a tree with arbitrarily many non-terminal nodes.

12.1 Shift Reduce Parsing

Given a sentence, we want to find the best dependency graph for it. One approach is shift reduce parsing (aka, transition-based parsing) which is parameterized over parser transitions –instead of substructures of dependency tree– in a way like that of an LR parser or a shift-reduce parser for formal languages.

In shift reduce parsing we work with something like a push down automata and choose to go either left or right in the sentence: shifting is for pushing something on stack and reduce is for making a dependency arrow. A parse configuration in this algorithm has three components: stack, buffer, and the set of dependency arrows. Buffer initially contains all words in the sentence and the algorithm terminates when the

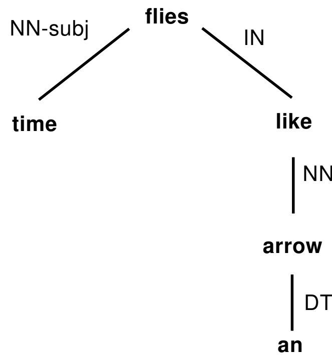


Figure 11: Dependency tree representation for the sentence "time flies like an arrow"

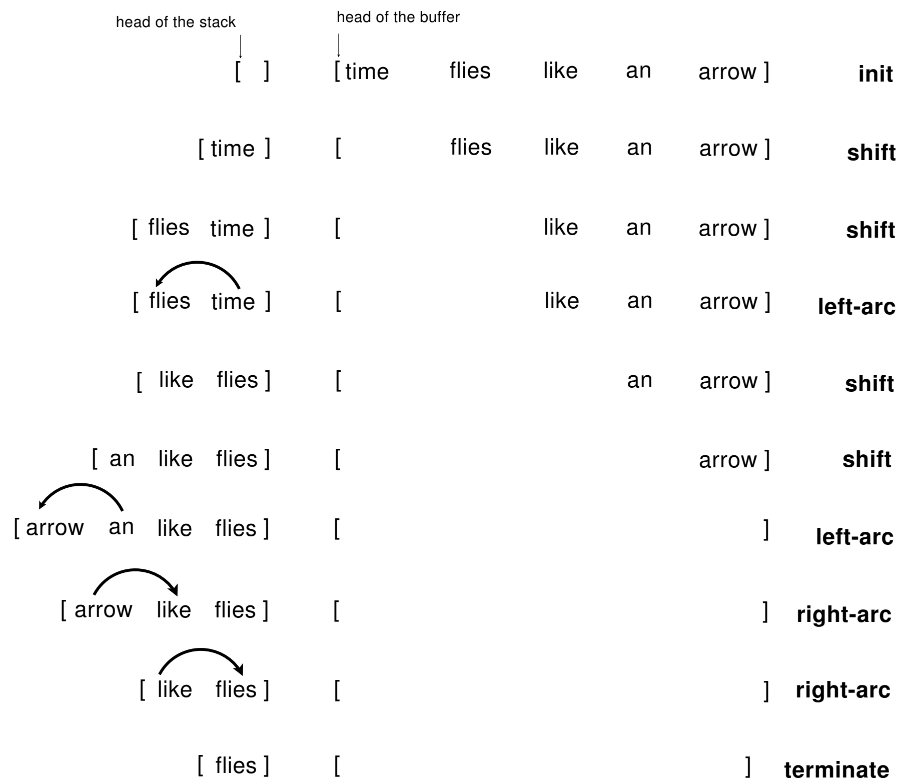


Figure 12: Shift-reduce algorithm executed on sentence “time flies like an arrow”

buffer is empty. We always go from one configuration to another by choosing one of the three actions (i.e., transitions):

- Right-arc: Add an arrow from the topmost word on the stack, w1, to the second-topmost word, w2, and pop w2
- Left-arc: Add an arrow from the second-topmost word on the stack, w2, to the topmost word, w1, and pop w1
- Shift: move top word in the buffer to the stack

A word is relaxed (i.e., we are done with it) if it gets poped out of the stack. Decision about which of the three transitions to make, is based on a trained classifier which tells us what to do at each time step. In this greedy algorithm we do not maximize any score, so we are not guaranteed to output the best dependency tree. Figure 12 represents this algorithm executed on input sentence "time flies like an arrow". In this figure the stack is on the left, the buffer is on the right, and each row corresponds to a step in which either of the three actions has been taken.

Algorithm 14: Early update algorithm

```
1: for i as step do
2:   if y_n not in stack then
3:     ỹ = stack[i][1]
4:     Δf(x, y_n, ỹ, i) = f(x, y_n, i) − f(x, ỹ, i)
5:     w = w + Δf(x, y_n, ỹ, i)
6:   end if
7: end for
```

As mentioned the actions taken at each step of the shift reduce parsing is learned via a classifier. Such a classifier aims at

$$\max_y w^T f(x, y) = \max_y w^T f(x, y, i) \tag{86}$$

assuming that the features decompose with time step i. Taking into account the transition made at each time step we would have the following:

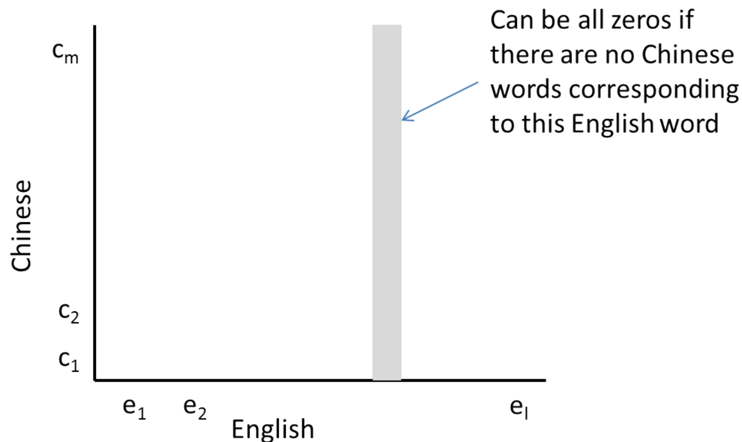
$$[\max_{action} w^T f(x, y, i, action)]_i \tag{87}$$

In the main paper on dependency parsing, the feature weight vector learning is based on "multi-stack decoding" where we keep the top-K ways of getting to each position in the sentence. In this way, we move (shift) from position i to $i + 1$ in sentence and then reduce everything in top-K. Here we want to perform this and learn the weight vector by Perceptron algorithm. Perceptron will not work for a greedy algorithm as it is. We should do an early update as demonstrated in algorithm 14 for it to work.

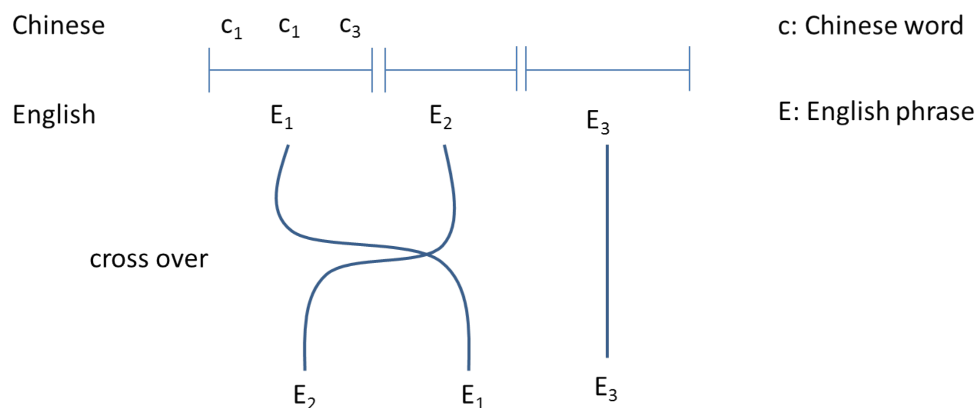
13 Machine Translation

Start with a parallel corpus, a document in both languages that has been translated sentence by sentence. The steps involved in machine translation are

- Word Alignment: For example, which Chinese word corresponds to which English word. Represented as a binary matrix $\{0, 1\}^{l \times m}$.



- Expectation Maximization: Maximize expectation of the alignment matrix
- Viterbi to get the best single matrix
- Rules Extraction: Phrase based or syntax based. Phrase table: Pair of phrases, e.g. English phrase corresponding to a Chinese phrase. Submatrix in the alignment matrix.
- Decoding: $\text{argmax}_e(score(phrase)P_{LM}(e))$.



$P_{LM}(e) = \prod_{i=1}^{l} P(e_i \mid e_{i-2}, e_{i-1})$: N-gram language model. Similar to speech recognition but more complicated because of cross over. Typical phrases are ≈ 7 words. Reordering can occur at the syntax tree level.

13.1 Word Alignment

Reference IBM Model 1 (Brown et. al. 1993). Brown invented statistical machine translation. French to English translation without dictionary or grammar rules. Long paper which discusses a sequence of models, most based on EM. Tutorial by Kevin Knight is a more conversational approach.

$$P(f \mid e) = \sum_a P(fa \mid e)$$

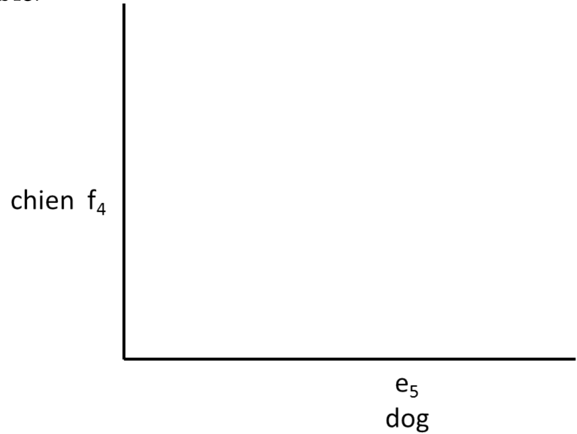
Here f is French, e is English, and a, the alignment matrix, is a hidden variable

Independence assumption: Assume each French word corresponds to exactly one English word. However, each English word can correspond to more that one French word. a_j is the index of the English word

corresponding to the French word f_j. e_0 corresponds to a NULL (special symbol).

$$P(fa \mid e) = P(|f| \mid |e|) \prod_{j=1}^{M} P(a_j \mid e) P(f_j \mid a_j e)$$

$|f| = M$ is the length of the French sentence. Note that $P(f_j \mid a_j e) = P(f_j \mid e_{a_j})$. In the example below, $a_4 = 5$. $P(f_4 = chien \mid a_4 = 5, e) = P(chien \mid dog)$. This a big table called (lexical) Translation table or T-table.



Second assumption:

$$P(a_j \mid e) = P(a_j) = \frac{1}{l+1}$$

Ignore the reordering between words. Bag of French words to bag of English words.

$$P(fa \mid e) = \underbrace{\frac{C}{(l+1)^m}}_{\text{constant}} \prod_j \underbrace{P(f_j \mid e_{a_j})}_{\text{learned using EM}}$$

13.2 Expectation Maximization

Recall EM:

$$\underset{\Theta}{\operatorname{argmax}} \log \sum_Z P(X, Z; \Theta)$$

$\Theta = P(f \mid e)$, i.e. T-table, $Z = a$ and $X = f$.

$$P(X, Z; \Theta) = \prod_s \frac{C}{(l+1)^m} \prod_j P(f_j \mid e_{a_j})$$

which can be rewritten by rerranging according to values instead of tokens as

$$P(X, Z; \Theta) = C \prod_{ef} P(f \mid e)^{c(f,e)}$$

$c(f, e)$ is the count of f and e appearing together,e.g. how many times did *dog* line up with *chien*.

$$c(f, e) = \sum_s \sum_j I(f_j = f \wedge e_{a_j} = e)$$

Taking logs we have

$$\log(P(X, Z; \Theta)) = \sum_{f,e} E[c(f, e)] \log(P(f \mid e))$$

In EM we have

$$\max_{\Theta} E_{P(Z|X;\Theta^{old})}[\log P(X,Z;\Theta)]$$

In each iteration of EM

- calculate expected counts $ec(f,e)$
- update $P(f \mid e) = \frac{ec(f,e)}{\sum_{f'} ec(f',e)}$

Compute expected counts from the sentences. $P(a_j \mid ef)$ is the probability of a given the two sentences. i.e. look at one row of a matrix at a time.

$$P(a_j = i \mid ef) = \frac{P(a_j = i, f \mid e)}{\sum_{i'} P(a_j = i', f \mid e)} = \frac{P(f_j \mid e_i)}{\sum_{i=0}^{l} P(f_j \mid e_i)}$$

The full algorithm may be written as
E:Step:

For s in sentences
  For j= 0 to $|f|$
    For i= 0 to $|e|$
      $t(j,i) = P(f_j \mid e_i)$
      $total(j) += P(f_j \mid e_i)$
    For i=0 to $|e|$
      $t(j,i) \mathrel{/}= total(j)$
      t-table$(j,i) += t(j,i)$

M:Step:

For e,f
  $P(f \mid e) = \frac{\textit{t-table}(f,e)}{\sum_{f'} \textit{t-table}(f',e)}$

Repeat E-step and M-step until convergence. The complexity is $O(Sn^2)$ for the E-step and $O(V)$ for the M-step, where $n = m = l$ is the number of words in a sentence. Note that the runtime of the M-step assumes a sparse matrix data structure for the T-table.

Model 1 is the only convex EM problem and the algorithm is guaranteed to converge to the global optima. To show this, we have from before,

$$P(X,Z;\Theta) = \prod_s \underbrace{\frac{C}{(l+1)^m}}_{C'} \prod_j P(f_j \mid e_{a_j})$$

Then,

$$\sum_Z P(X,Z;\Theta) = \sum_a \prod_s C' \prod_j P(f_j \mid e_{a_j})$$

The hidden variable $Z = a$ here. Due to the independence assumption, what happens in each row is independent of what happens in a different row. Therefore, we can push the summation in to get

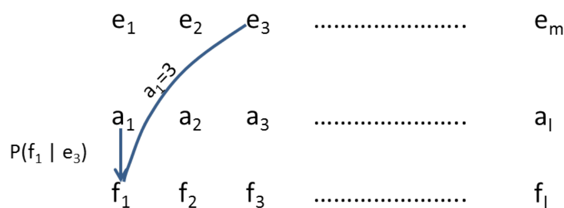
$$\sum_Z P(X,Z;\Theta) = \prod_s C' \prod_j \sum_{a_j=0}^{l} P(f_j \mid e_{a_j})$$

e_{a_j} are all the English words that the French word f_j could have come from. Taking logs, we can write the log-likelihood L as

$$L = \sum_s \sum_j \log(\sum_{a_j=0}^{l} P(f_j \mid e_{a_j})) + \log(C')$$

We observe that log is a concave, nondecreasing function and sum of logs is concave. Thus L must be concave. In HMM, the transitions and emissions interact with each other, and so L is not concave. However in reality, L is not strictly concave here. It can have flat spots, i.e. some combinations of parameters for which L is exactly the same.

Is the IBM Model 1 a graphical model? It can be represented as a graphical model. However, $P(f_j \mid e_{a_j})$ acts as a switch which controls the connectivity in the graph between the f (French) nodes and the e (English) nodes (see figure below).



13.3 IBM Model 2

IBM Model 2 relaxes the assumption that every alignment is equally probable. Specifically, Model 2 relaxes the constraints on the multiplier of the product equation (1). IBM Model 2 quantifies how reordering of the words affects the probability of obtaining a French translation from an English sentence. To accomplish this, we introduce a new function a, the reordering distribution. a characterizes that the position of the word in the French sentence depends on the corresponding word in the English sentence, but also on the length of the English and French sentences. The probability of the French translation for a specific alignment vector a is now given by

$$P(f, a|e) = \epsilon \prod_{j=1}^{m} P(f_j|e_{a_j})a(a_j|j, l, m) \tag{88}$$

The multiplier ϵ is equal to the probablity of the French sentence having m words for an English sentence have l words, or $P(m|l)$ which is just a constant. For model 1 we were able to apply the expectation maximization (E-M) algorithm and solve for the optimal alignment with dynamic programming. Will the generalization of alignment probabilities still enable the use of dynamic programming and the algorithm developed for IBM Model 1? The answer is yes. All the independence assumptions are the same. The dynamic programming algorithm is exactly the same. The E step is

for j = 1 to m
    for i = 0 to l
        count[j][i] = $P(f_j|e_i)a(i|j, l, m)$

normalize so the counts add up to 1

and the M step is

normalize the t-table
normalize the a-table

The computation of Model 2 is the same as Model 1, but there are more parameters. There are m^4 parameters, which creates a huge table. So in practice what one does is to train using Model 1 for a few iterations and use the result to train Model 2.

In the last lecture we looked at the likelihood function for Model 1 which is given in equation (3)

$$\sum_{s} \sum_{j=1}^{m} \log \left(\sum_{i=1}^{l} P(f_j|e_i) \right) \tag{89}$$

The sum over P is a linear function which is concave. The logarithm function is monotonically increasing and the result remains concave. Therefore when EM is applied to Model 1 the global maximum can be

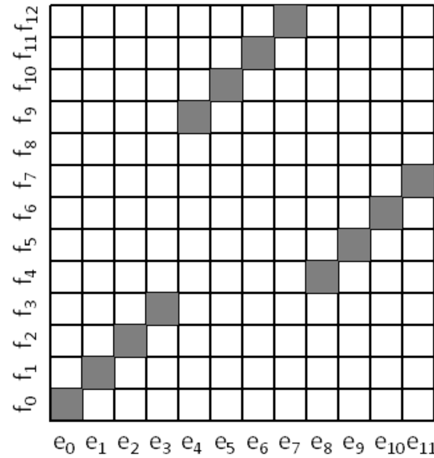


Figure 13: Alignment Matrix

reached. However the likelihood function for Model 2 contains an extra term

$$\sum_{s} \sum_{j=1}^{m} \log \left(\sum_{i=1}^{l} P(f_j|e_i) a(i|j,l,m) \right) \tag{90}$$

Because of the product of the conditional probability P for the French word on the English word, and the alignment term a, the log likelihood function is no longer convex. The consequence of losing this property is that without training on Model 1, Model 2 can become stuck in a local maximum that is grossly suboptimal. The reason that there are multiple levels of models in the IBM paper is that Model 1, which makes naive assumptions about the relationship between words within the sentences, is needed to train Model 2. Model 2 adds intelligence in the choice of word alignment.

When the transition is made between Model 1 and Model 2, the alignment matrix a is initialized with a uniform distribution. This initialize gives Model 2's performance to that of Model 1 at the transition.

13.4 HMM Model for Machine Translation

We now turn towards the HMM model that is not in the original IBM paper but really is the next step in adding complexity to better represent the relationship between the two languages. One should think of it as IBM Model $2\frac{1}{2}$. The HMM Model is similar to model 2, except that we condition the probability on the position of the previous word. Mathematically this is expressed as

$$P(f, a|e) = \prod_{j=1}^{m} P(f_j|e_{a_j}) P(a_j|a_{j-1}, l, m) \tag{91}$$

The conditioning of the alignment matrix on its previous term is saying graphically that the alignment matrix should consist of clusters of diagonal strips. In making an alignment, groups of English words generally follow the English sentence from the start of the sentence to the end of the sentence in the same way that the French words move along the sentence. Sometimes there are regions that don't align, but they correspond to entire phrases. Within a phrase, the words move in order. A graphical representation of this situation is shown in the figure 1. The English words are given on the x-axis and the French words are given on the y-axis. The gray square represents an aligned word. This is a typical pattern. Model 1 doesn't handle a pattern like this well. Model 1 is not able to recognize that a pair of words is likely to align if the cell below and to the left is aligned.

Graphically, one think of the alignment variables of forming the hidden states of a Markov chain. Each node in the chain has l states, one for each of the French words. The chain is m steps long. One can use the HMM dynamic programing algorithm to solve for the highest probability alignment.

The E-step of dynamic programing problem is solved with the forward algorithm
for j = 1 to m
    for i = 0 to l
        for i' = 0 to l
            count[j][i] += $\alpha(j-1,i')P(f_j|e_i)P(i|i',l,m)$
Normalize counts so it adds up to 1.

Creating a table that gives the probability of a misalignment by an arbitrary number of words gives the model too many parameters. There would be a tendency to overfit. It is preferable to model the probability of word separation with a distribution function S. S can be thought of as an expected count of jumping the number of steps of its argument. By using S, we decrease the number of parameters from N^4 to N.

$$P(a_j|a_{j-1},l,m) \propto \frac{S(a_j - a_{j-1})}{\sum_{i'} S(i')} \tag{92}$$

When we use S the pseudo-code of the HMM model now becomes
for j = 1 to m
    for i = 0 to l
        for i' = 0 to l
            count[j][i] += $\alpha(j-1,i')P(f_j|e_i)P(i|i',l,m)$
            $S(i-i')$ += count[j][i][i']
            t-table[f_j][e_i] += count[j][i][i']

In the example shown in figure 1, we see a jump of 6 from e_3 to e_4 and a jump of 12 from e_7 to e_8. The presence of the jump decreases the probability of having this particular transition, but the good alignment of future transitions still make this a preferred alignment.

The running time of the dynamic programming algorithm is ml^2 which we can just simplify to N^3. It is slower, but still satisfactory for use.

Before we move to the next model, lets consider NULL words in the HMM. There are words in a French sentence that are not associated with any words in an English sentence. For example, the French word *le* may come from no corresponding English word. If we only have one NULL column in the alignment graph, then the alignment matrix would have a lot of huge jumps to the null word, and then jumps back up to the closely spaced alignments. This would cause a large decrease in the score of a particular alignment.

One way to solve this problem is to put a NULL word between every English word in the alignment matrix. Then normal jumps, a jump that occurred for words that are adjacent in both English and Frency, would be a jump of two because the jump would have to be over the interlaced NULL words. If a NULL word needed to be inserted, there would be a jump of 1 to the NULL followed by a subsequent jump of 3. This would not significantly decrease the score. For a sequence of NULL words, one would stay in this NULL state and the algorithm would be able to "remember" where the last word alignment was when there is another word alignment.

13.5 IBM Model 3

IBM Model 3 completely breaks dynamic programming. It does this because we want to have a term of how many children each English word has (children here means the French words). In the previous models each column in the alignment matrix contained only one entry. However, an English word can come from any number of French words. In IBM Model 3 we can have only one element in each column of the alignment matrix, but we can have an arbitrary number of elements in each row. The number of entries in the row is defined as fertility ϕ

$$\phi_i = \sum_{j=1}^{m} I(a_j = i) \tag{93}$$

where I is the indicator function. The probability for an alignment of a French sentence with an Enclish sentence now becomes

$$P(f_i a|e) = \binom{m}{\phi_0} p_0^{m-\phi_0} p_1^{\phi_0} \prod_{i=1}^{l} \phi_i! n(\phi_i|e_i) \prod_{j=1}^{m} P(f_j|e,a_j) d(j|a_j,m,l) \tag{94}$$

The terms before the product over i quantify the fertility of the null word. They represent the number of insertions into the sentence. The number of insertions should depend on the sentence length. The factorial of the fertility inside the product over i occurs because it doesn't matter which order the group of words is found. Because the fertility ϕ is a function of the alignment vector a, the term n, which is outside of the product over j is what breaks dynamic programming.

Because it is not possible to perform E-M with dynamic programming, Gibbs sampling is used for IBM Model 3, although the IBM authors choose to search through all possible alignments of a sentence. For Gibbs sampling, one would train using model 2 to get an alignment matrix. From this alignment matrix, small changes would be made. If the model 3 probability is higher with this tweaked alignment matrix, then it would be accepted, otherwise it would be rejected. A successful application of the algorithm would converge in a local maximum. This is similar to hill climbing, a deterministic algorithm. In Gibbs sampling the acception of a change is made with some probability less than one, in this algorithm it is made deterministically.

What are the changes that can be made to the alignment matrix? One change could be moving an element within a column. Another change might be switching the order where two words came from.

13.6 IBM Model 4

The idea of IBM Model 4 is that it is kind of adding HMM to model 3. The position where one word ends up depends on the previous word. In equations this is expressed as

$$P(f_i a|e) = \binom{m}{\phi_0} p_0^{m-\phi_0} p_1^{\phi_0} \prod_{i=1}^{l} \prod_{k=1}^{\phi_i} \frac{1}{\phi_{0i}} t(\tau_{ik}|e_i) p_{ik}(\pi_{ik}) \prod_j n(\phi_i|e_i) \tag{95}$$

where τ_{ik} is the kth French word produced by e_i (that is, some f_j), and π_{ik} is the position of the kth French word produced by e_i (that is, the corresponding j). All the children of English word i get reordered as a group. There are two cases. The first child is conditioned on the words themselves

$$p_{ik}(j) = \begin{cases} d_1(j - c_{p_i}|A(e_i)B(\tau_{i1})) & \text{if } k = 1 \\ d_>(j - \pi_{i,k-1}|B(\tau_{ik})) & \text{o/w} \end{cases} \tag{96}$$

A and B are clusters of English and French words, such as English Nouns and French Nouns. Note that the position of the French words depends on the position of the previous French word produced by the same English words, which is similar to dependencies of the HMM alignment model.

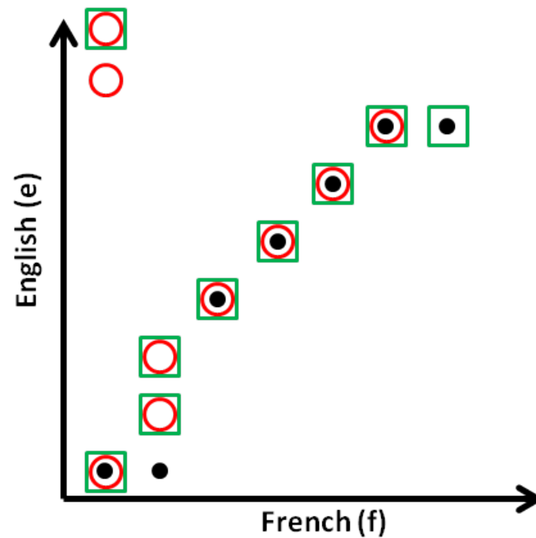
14 Phrase Based Machine Translation

14.1 Symmetrization

Up to this point, all of the models (IBM 1-4 and HMM) that we talked about are asymmetric; they are many to one in one direction where every French word comes from one English word. This assumption was in place to make the alignment easier but it isn't true in general since alignments can be many to many in either direction. Neither direction is really special so whatever we do should be symmetric. The way this is typically resolved is by running IBM M4 (EM and Viterbi) in both directions and combining the two alignment grids to form a single grid. The process of merging the two alignment grids is called symmetrization.

The first alignment grid will have one dot in each column and the other will have one circle in each row. The final symmetrized alignment is represented with green squares in the figure where many to many

alignments are allowed.



$$\text{Dots} = \underset{a}{\operatorname{argmax}}\, P(f, a|e)$$

$$\text{Circles} = \underset{a}{\operatorname{argmax}}\, P(e, a|f)$$

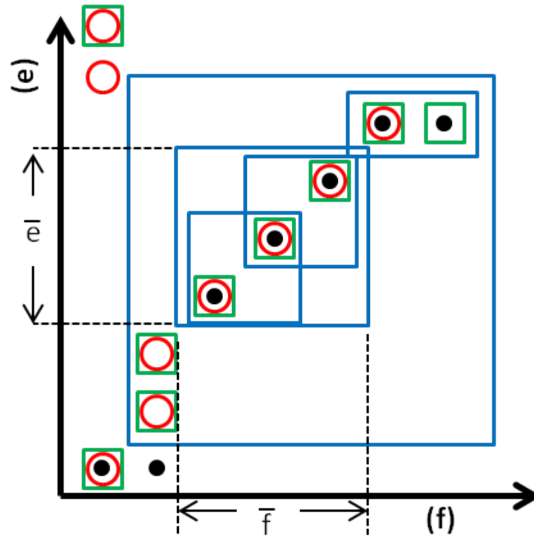
When merging the two alignment matrices, one can take the union, the intersection, or something in between. The most common heuristic to do this is,

1. Start with intersection points.
2. Add points from union: For each unaligned word that is adjacent to an existing point. Have more confidence that a point is good if it's adjacent to points that already captured.
3. Finally, if unaligned words remain, then add points even if they're not adjacent. Don't want to have any unaligned words left over.

14.2 Rule Extraction

Once we have a symmetric alignment, we can extract all consistent phrase pairs. A consistent phrase pair is represented as a block in which all words are aligned only with words inside the block and not with any outside the block. For a given alignment, many phrase pairs are possible and some are shown in the figure

below in the form of blue rectangles. An example of a phrase pair is: $(\bar{f}, \bar{e})$



How many valid consistent phrases can there be in a sentence of length n?

Depends on alignment. For a purely diagonal alignment, it would be n^2 because if you fix the English phrase, only one French phrase is consistent. An alignment that allows many possibilities, where the alignment matrix is fully populated would provide the lowest case since grabbing one English word would require grabbing the entire French sentence, leading to the entire English sentence and thus only a single phrase. The worst case, most phrases, comes from an alignment with nothing populated leading to n^4 phrases. In practice: $n <=$ phrases $<= n^2$

Once the phrases are extracted, the translation probabilities can be determined using MLE and we're now ready for decoding.

14.3 Decoding

We're translating into English. The decoding problem is framed as follows:

$$E = \underset{e}{\operatorname{argmax}}\, w^T f(e, f, d)$$

Where: $f()$ = feature function, d = derivation or sequence of phrase pairs applied to get the french from the english, $\bar{e}$ = english phrase, $\bar{f}$ = french phrase, and $\bullet$ represents any english side phrase

There are seven total features,

1. $\log\left(P(\bar{e}|\bar{f})\right) = \frac{C(\bar{e},\bar{f})}{C(\bullet,\bar{f})}$
2. $\log\left(P(\bar{f}|\bar{e})\right)$
3. $\log\left(P_{lex}(\bar{e}|\bar{f})\right)$
4. $\log\left(P_{lex}(\bar{f}|\bar{e})\right) = \prod_{j=1}^{|\bar{f}|} \frac{1}{|\{J:(i,j)\in a\}|} * \sum_{J:(i,j)\in a} P(f_i|e_j)$

5. Length $= |\bar{e}|$
6. N-Phrases = Total number of phrases used
7. Language Model $= \log \prod_{i=1}^{m} P(e_i|e_{i-2}, e_{i-1})$

The final feature is the language model which is a tri-gram model. This model biases to shorter sentences since probability reduces with the addition of each word.

Need the length feature to ensure that output sentences are not too short since the LM biases towards shorter sentences.

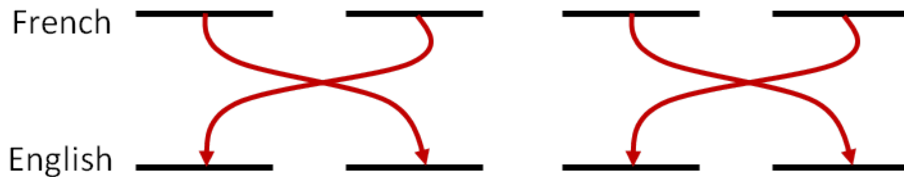
For lexical feature 4, the $P(f_i|e_j)$ comes from the IBM word model. Intuition is that if word pairs in a phrase are good, then this is an indicator that the phrase is good.

Some will say that speech recognition is based on Bayes rule,

$$P(e|f) = \frac{P_T(f|e)P_{LM}(e)}{P(f)}$$

$$P(e|a) = \frac{P(a|e)P(e)}{P(a)}$$

But this isn't true since the probabilities are mixed up. The reason why it works is not Bayes rule but because we're combining different sources of information. The features measure strength of the red arrows.



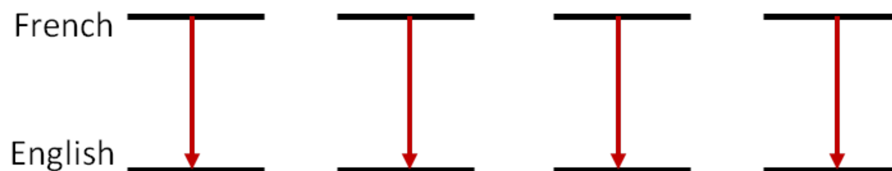
We're translating into English and thus want an English model. Rules 1-4 describe how well the French matches the English, and rules 5-7 describe the fidelity of the English model. There is no model for French because French is fixed.

Want to Decode:

$$\underset{e}{\text{argmax}}\, P(e|f) = \frac{P_T(f|e)P_{LM}(e)}{P(f)}$$

$$\underset{e}{\text{argmax}} \left[\sum_{r \in d} Score(r)\right] + w \sum_{i} \log \left[P(e_i|e_{i-2}, e_{i-1})\right]$$

We need an algorithm for decoding but it's complicated because phrases can be out of order. If we assume for the time being that the French is in the same order as the English, we can go left to right across the sentence just like in speech recognition. We don't need to worry about the re-ordering but do need to decide which phrase pair to apply at each point in the sentence. The choice of phrase pair is driven by it's score as well as how well it matches with the English that we have so far.



Not like parsing. There is no tree and no hierarchical structure since we're gluing phrases together and creating English as we go where the English is scored by the LM.

Hypothesis or partial solution = $(r_1, r_2, r_3, ...)$ st $f(r_1) * f(r_2) * ... * f(r_n) = f_1$

In order to do dynamic programming (DP), need to know when it is safe to discard a partial solution since something else is better. This happens when we have perfect alignment.

Algorithm 15: Partial dynamic programming decoder. $O(n^2)$

1: $state = [i]$
2: **for** $i = 1$ to $|f|$ **do**
3: **for** $j = 1$ to i **do**
4: **for** r st f(r)=f_j^i **do**
5: $\delta[i]$ max= $\delta[j] + score(r)$
6: **end for**
7: **end for**
8: **end for**
9: **return** δ

This algorithm would be perfect if objective function were only: $\text{argmax}_e \left[\sum_{r \in d} Score(r) \right]$
This algorithm solves the segmentation problem where we need to determine location of spaces in a French sentence; come up with a score for every word and look for segmentation that has the highest total score. But this is not enough for the language model. We need two more elements in $state$ since the LM is a sliding window that looks at 3 words at a time. The objective function has overlaps that include part of one phrase, the end of a previous phrase and the beginning of the next.

For two consecutive phrases, let e and e' represent the last two words of the first phrase and i, the first word of the second phrase. u and u' correspond to e and e' and are the last two words of hypothesis unless the English side is empty or has only one word. The relationship is defined as (':'= concatenate),

$$ee' : e(r) = *uu'$$

Algorithm 16 captures the complete decoding algorithm. If we assume that the French and English sentences have the same number of words then the big-O is: $O(n^2V^2)$. If we assume a constant number of possible translations for each word, then $V = O(n)$, giving $O(n^4)$ for trigrams, and $O(n^{2+(m-1)})$ for m-grams.

Algorithm 16: Monotonic Phrase-based Decoding.

1: $state = [i, e, e']$
2: **for** $i = 1$ to $|f|$ **do**
3: **for** $j = 1$ to i **do**
4: **for** r st f(r)=f_j^i **do**
5: **for** e, e' **do**
6: $\delta[i, u, u']$ max= $\delta[j, e, e'] + score(r) + P_{LM}(e(r)|e, e')$
7: **end for**
8: **end for**
9: **end for**
10: **end for**
11: **return** δ

14.4 Reordering

Build English hypothesis left to right. At each point will have a derivation which is a sequences of rules. The rules will be the first n rules in English which come from any subset of French - not necessarily continuous. Before the state was i and some LM state. Now the state includes how much of the French we've covered - not position 1..i but some bit vector of which French phrases we've covered. We're not moving left-to-right through the French sentence. Can still do DP and recombine hypothesis if they cover the same French words.

Hypothesis looks like this: $\left[[oo_j.......o_k o....], e, e'\right]$

Algorithm 17: Reordering

1: **for** $i = 0$ to $|f|$ **do**
2:   **for** h st i french words covered **do**
3:     **for** r **do**
4:       **for** j, k **do**
5:         $\delta[h \circ r]$ max= $\delta[h] + score(r) + P_{LM}$
6:       **end for**
7:     **end for**
8:   **end for**
9: **end for**
10: **return** δ

Space complexity: $O(2^n n^{m-1})$
Time complexity: $O(2^n n^{m-1} n^2)$

Can't do this - need buckets. Keep top k in buckets - a bucket for each number of French words that we've covered so far.

This reduces the time complexity to: $O(n^3 k)$

Recall that our hypothesis h is in the form of $h = [[0110 \cdots 11], e, e']$. Let $b = [0110 \cdots 11]$, we have $h = [b, e, e']$. Define $bin(h) = \sum_i b_i$ to denote how many words we have covered so far in the foreign language and let $h \in B[bin(h)]$. The decoding algorithm can thus be defined as in Algorithm 18.

Algorithm 18: *Phrase-Based Decoding*

1: **for** $i = 0$ to $|f|$ **do**
2:   **for** $h \in B[i]$ **do**
3:     **for** i, j **do**
4:       **for** r s.t. $f(r) = f_i^j$ **do**
5:         $h' = h \circ r$
6:         $cost[h'] = cost[h] + cost[r] + \log P_{LM}(e(h \circ r))$
7:       **end for**
8:     **end for**
9:   **end for**
10: **end for**

Number of hypothesis: $2^n n^2$.

Time complexity

- $O(2^n n^2 n^2)$ for full dynamic programming.

- $O(n^3k)$ for best-k.

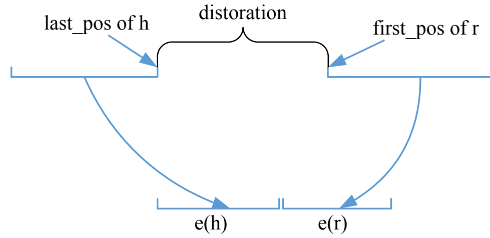


Figure 14: Illustration of Distortion.

14.5 Decoder with Distortion Model

The decoder typically generate the target language sequentially. While, with reordering, the decoder may jump forth and back on the foreign language. Previous decoding algorithm does not take the distortion cost into consideration. We can add the distortion into the model by calculating the distortion distance.

Figure 14 shows the concept of distortion. Distortion could be negative, positive or zero. The distortion distance is defined as $|last_pos(previous_rule) - first_pos(rule)|$. To take the distortion into consideration, we need to define our hypothesis as $h = [[0110 \cdots 11], l, e, e']$. Here l is an integer, which keeps track of the last position of the phrase in the foreign language of the previous rule you applied. Algorithm 19 summarizes the decoding algorithm with distortion distance taken into consideration.

Algorithm 19: *Phrase-Based Decoding*

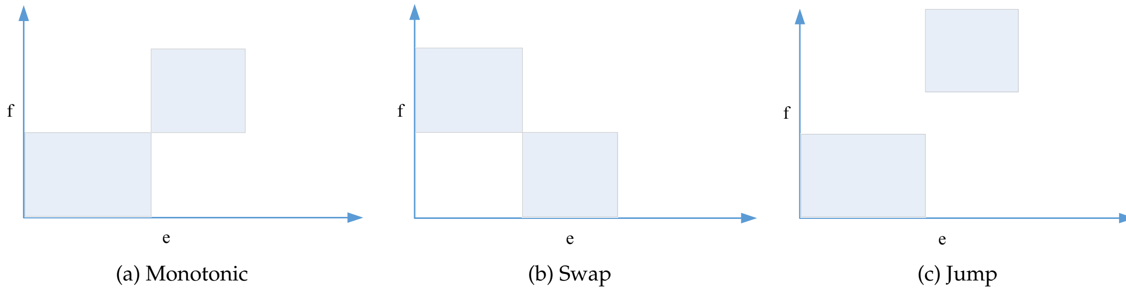
1: **for** $i = 0$ to $|f|$ **do**
2:   **for** $h \in B[i]$ **do**
3:     $h = [b, l, e, e']$
4:     **for** i, j **do**
5:       **for** r s.t. $f(r) = f_i^j$ **do**
6:         $h' = h \circ r$
7:         $cost[h'] = cost[h] + cost[r] + w_{LM} \log P_{LM}(e(h \circ r)) + w_{distortion}|l - i|$
8:       **end for**
9:     **end for**
10:   **end for**
11: **end for**

14.6 Lexicalized Reordering

Distance based reordering only considers the movement distance. Sometimes, it may be more helpful to define the reordering on the actual phrases. To simplify the reordering model, we only consider three types of reordering, namely monotonic, swap and jump. Figure 15 shows the three types of reordering.

To learn $p(reordering|\bar{e}, \bar{f})$ from the data, the reordering type can be detected from the word alignment. Figure 16 shows the possible types for monotonic and swap reordering given the current alignment block. With this, it is possible to count the number of different reordering types from the training data for different phrase pairs.

Phrase-based decoding is NP-complete. To prove this, we can reduce the classical Travelling Salesman Problem (TSP) to phrase-based decoding. The reordering in our decoding problem is reduced to the order of the cities visited in TSP. Let $v_1, v_2, dots, v_n$ be the cities in TSP. Define the pair of cities as (v_i, v_i), which is similar to the pair of foreign phrase and target phrase. However, here the mapping is deterministic, i.e. $(v_i \rightarrow v_i)$. Figure 17 shows the idea of reordering fro TSP.



(a) Monotonic (b) Swap (c) Jump

Figure 15: Different Types of Reordering.

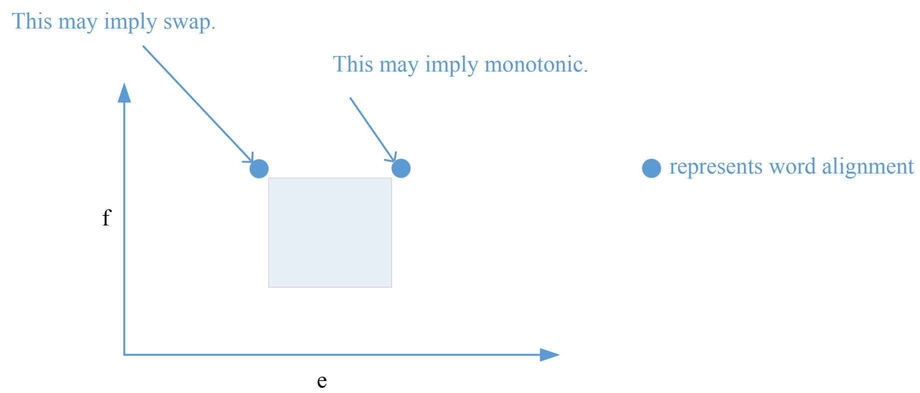


Figure 16: Estimation of different reordering types.

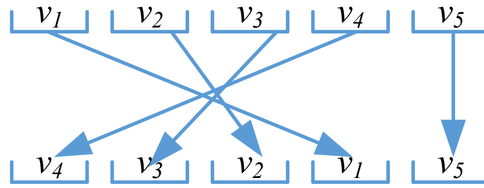


Figure 17: Order of visit cities in TSP.

The *language model* cost for TSP is $\log P_{LM}(v|u) = w(u, v)$, where $w(u, v)$ is the distance between city u and city v.

15 Syntaxed Based Machine Translation

15.1 Synchronous Context-Free Grammar(SCFG)

SCFG models the foreign language and the target language together.

- $A \rightarrow BC, CB$
  There are two right hand sets. Each of them must have the same two non-terminals with whatever orders they like.

- $C \rightarrow$ dog, Xiao Gou (Chinese pinyin)
  Do not need to have the same number of terminals.

People also add index to the non-terminals to distinguish them. For example, to distinguish B in the right hand set of rule $A \rightarrow BB, BB$, we can rewrite this rule as $A \leftarrow B^1B^2, B^2B^1$.

According to the above of SCFG, the two trees generated from SCFG have the same structure, but may have different orders at each node.

- $CFG \Rightarrow languages \subset \Sigma^*$

- $SCFG \Rightarrow Transition \subset \Sigma^* \times \Sigma^*$

SCFG generates transitions between foreign language and target language. We can define the transition as a language $Transition = Language \subset \Sigma^*\#\Sigma^*$. Each side of the rules themselves is CFG, i.e., the foreign language side rules and the target language side rules are CFGs respectively.

SCFG is not CFG Counter example, $a^nb^n\#b^na^n$ is from SCFG $S \rightarrow aS'b, bS'a,\ S' \rightarrow ,\ .$ However, $a^nb^n\#b^na^n$ is not CFG.

SCFG does not have corresponding CNF For instance, rule $S \rightarrow A^1B^2C^3D^4, C^3A^1D^4B^2$ cannot be decomposed to CNF for both the foreign and the target rules simultaneously.

15.2 SCFG in Machine Translation

Two steps

- First align words.

- Find rules consistent with word alignment.

For SCFG $A \rightarrow e_1Be_2Ce_3,\ f_1Bf_2Cf_3$, the number of possible phrase pairs is $O(n^{12})$ in worst case and $O(n^6)$ for monotonic alignment.

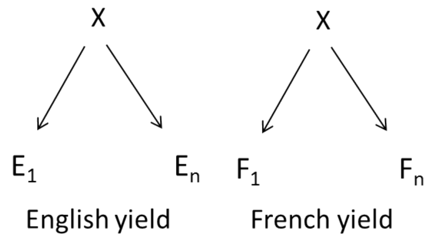
16 Decoding with SCFG

Definition of SCFG:

$$X \to e, f$$
$$X \to X^{[1]}X^{[2]}, X^{[2]}X^{[1]}$$

The bracketed numbers link up nonterminal symbols on the source side with nonterminal symbols on the target side: 1 links to 1 , 2 with 2 , and so on.

What does decoding mean here? We have French strings and want corresponding English strings with the highest probability. If we make SCFG probabilities, assign P to all rule expansions it gives us some distribution over trees and some tress as distribution over derivation. Every derivation is a pair of trees, some English string at bottom, French string at bottom as shown below.



Destribution of derivation, P(d):

$$P(d) = \prod_{r \in d} P(r)$$

Best English Translation, e^*:

$$e^* = \underset{e}{\text{argmax}} P(e \mid f) \qquad eq(1)$$

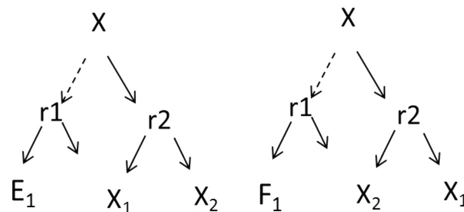
English we set as is

$$e(\underset{d:f(d)=F}{\text{argmax}} P(d))$$

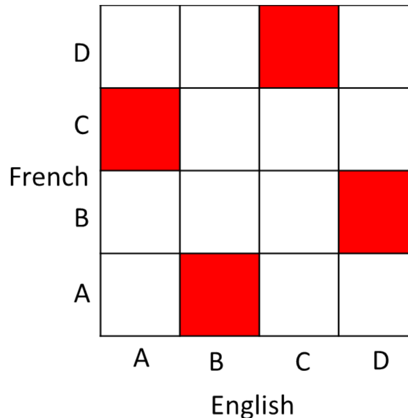
Find the best derivation, and see what comes out on the other side. This is parsing, parsing with French side of SCFG and read off from English side. There is an extra variable d that we don't have in eq (1). We can explain with

$$P(e \mid f) = \sum_{d:e(d)=e} P(e, d \mid f)$$

We are making an approximation and not summing over all trees. Rather we are using the simple tree with the highest probability. Parse with the French side of the grammar, read off the English. Each node tells us which rule is used, then translate to English by reading off rules.



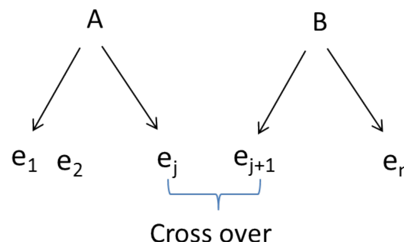
Reading off the English rule2 tells us X_1 and X_2 are flipped. What is the complexity of doing SCFG decoding? The complexity of parsing is $O(|G|n^3)$, where $|G|$ is the size of grammar and n is the length of sentence.



There is a catch. In last class we said there is no CNF for SCFG. For example $S \rightarrow ABCD; CADB$, no two symbols are discrete. No piece we can break off in English that would be adjacent in French. Something like this prevents $O(n^3)$ complexity.

16.1 Standard CKY Parsing

If we can binarize the rule we have to consider all combinations in the string that lie below each terminal. Simplify the rules with just nonterminals and there are K of them. The complexity is $O(n^{k+1})$. We can do better. We can break these particular rule down into steps. Let's just consider binary rules case for now and deal with complex case later. We want a n-gram language model. The n-gram language model will cross over subtrees.



Ultimately we want $\underset{d}{\operatorname{argmax}} P(d) P_{LM}(e(d))$. We want dynamic programming (DP) algorithm to do this. We will need to keep track of last two words as we did for phrased based machine translation (MT). We will do the same for parsing. Parse French and keep track of English words at boundary. In standard CKY parsing entries in chart are [A, i, j] where A is nonterminal, i is the beginning position and j is the end position. Now our entries need to be [A, i, j, l, r]. English string begins with something on right and something on left. We parse with the expected state for DP. The algorithm is for standard chart parsing we had. The complexity is $O(|G| n^3 V^{4(m-1)})$ for m-grams. Typically, V is $O(n)$ because each English word has approximately 5 French translations. Thus, the complexity becomes $O(|G| n^{3+4(m-1)})$.

DP Algorithm:

for i, j, k
  for all rules $A \rightarrow BC, CB$
    for l_1, r_1, l_2, r_2
      $\delta[A_j, i, k, l_2, r_1] \max = P(A \rightarrow BC, CB) \delta[B_j, j, k, l_2, r_2] \delta[C_i, i, j, l_1, r_1] P_{LM}(l_1 \mid r_2)$

Let's look at features.

$P(d) \sim W^T f(d)$
$f(d) = \sum_{r \in d} f(r)$

$$f(r) = \begin{cases} P(\bar{e} \mid \bar{f}) \\ P(\bar{f} \mid \bar{e}) \\ P_{lex}(\bar{e} \mid \bar{f}) \\ P_{lex}(\bar{f} \mid \bar{e}) \\ \#words \\ \#rules \end{cases}$$

Decoding:

If it was just parsing with French, we could binarize French, parse, and read off English. However it is not that simple because of language model. Let's say we binarize side of grammar. Maybe we grab C, A. We are combining in French a C and an A and make one new nonterminal X. We want to make the language model for X. We want to make the rule for X. We have two substrings in English, flattening in space. They have something between C and A but we don't know it. That is why we can't binarize French and do what we did. Now we have to combine them all. Take A, B, C, D combine all to set as S, and look in English for whatever A began with and whatever B ended with. The complexity is $O(n^{k+1+2k(n-1)})$, where k is nonterminals.

$[X_1, i_1, i_3, l, r, l_2, r_2]$
$[X_2, i_1, i_4, l_1, r_1, l, r]$

It is possible to improve on this. We have A, B in French. We want to combine into new nonterminal. But there are more pieces of English we need to keep track of. We do one step at a time for whole thing in French and whole thing in English then we are done. Now the complexity becomes $O(n^{3+6(n-1)})$ from $O(n^{5+8(n-1)})$ when k is 4. In French side is i, j, k like in normal parsing. In English side the most complicated step is the last step which 4 English nonterminals combined with one French. This is better than before.

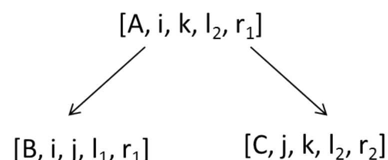
If we are given a SCFG rule with k, we can lower this, but still linear in k and it is still k in the exponent. For a given rule the complexity is $O(n^{k+1+2k(n-1)})$. For a given a French rule the complexity of the rule depends on its permutation.

$S \rightarrow A^1 B^2 C^3 D^4, C^3 A^1 D^4 B^2$

The permutation is $1, 2, 3, 4 \rightarrow 3, 1, 4, 2$. We want to find a best strategy for collecting the righthand nonterminals. These are $(4!)$ possible order in which we could collect them. This problem is NP-complete.

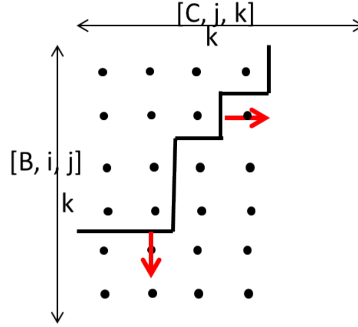
16.2 Cube Pruning

Here is what we really do. For binary case, first parse with French side of grammar and then do top k for options in chart. We find some good candidates of i,j,k for language model options.



Cube pruning is top-K parsing but with a variable combination cost. In top-K parsing we are combining top-K parses so they can be rescored later. This means we find the top-K items at each cell in the chart.

$A \rightarrow B, C, i, j, k$



K ways of matching B, and k ways of matching C. Top k out of the k^2 will be just another frontier. When we put one of these candidates into the chart we got below.

$$\delta[A, i, k] = \delta[B, i, j]\, \delta[C, j, k]\, (P \rightarrow BC) P_{LM}(l_2 \mid r)$$

This worked because it was constant. This whole product decreases when we go to the right, and when we go down.
Now we look at the top K ways in French. We are now likely at what is in a bin, some subject of the information we need. In our DP we only want the best but we will analyze it into the top K in each bin. We also have the language model cost. It is different for each part in the figure shown above. There is no longer a frontier. It is possible parts can get better when we go. We have a priority queue off to the side and we do a mini Dijkstra and explore the frontier. Anytime we look at the part in a matrix you add it to the frontier. The complexity is $O(|G|n^3(k \log k))$, where $k \log k$ term comes from priority queue.

17 Hiero (Hierarchical Phrase-Based Machine Translation)

17.1 SCFG for Hiero

In Hiero, SCFG has rules in one of the following forms:

- $X \rightarrow f_1 X^{\boxed{1}} f_2 X^{\boxed{2}} f_3, e_1 X^{\boxed{1}} e_2 X^{\boxed{2}} e_3$ *or* $X \rightarrow f_1 X^{\boxed{1}} f_2 X^{\boxed{2}} f_3, e_1 X^{\boxed{2}} e_2 X^{\boxed{1}} e_3$
- $X \rightarrow f_1 X^{\boxed{1}} f_2, e_1 X^{\boxed{1}} e_2$
- $X \rightarrow f_1, e_1$

From above, we can see that on the right side there are three types of # of nonterminals:
(1) two nonterminals (which can enable reordering)
(2) one nonterminal
(3) only terminals

17.2 Rule Extraction

When we want to extract rules that have two nonterminals on both the French and English sides from a sentence pair, we have to pick 6 indices (see Figure 1) for terminals on both sides, which would result in a grammar size of O(n^{12}). This grammar size is too big, so we need to reduce the number of rules.

We reduce the number of rules using the following restrictions:

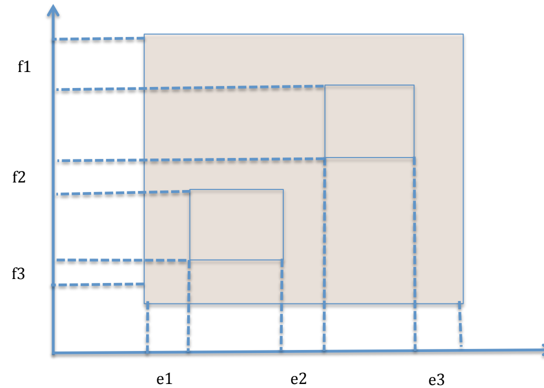


Figure 18: Alignment matrix for Hiero rules.

- Rules are limited to five nonterminals plus terminals on the French side
- Only keep "tight phrases"–If there are multiple phrase pairs containing the same set of alignments, only the smallest is kept. That is, unaligned words are not allowed at the boundaries of phrases
- It is prohibited for nonterminals to be adjacent on the French side, which is a major cause of spurious ambiguity.

  There are two reasons for this:
  (1) The terminals in the middle can provide information about when to reorder
  (2) The terminals in the middle also provide information about the right indice of the first nonterminal and the left indice of the second nonterminal

  For example (Chinese to English):
  $X \rightarrow X^{\boxed{1}}\ de\ X^{\boxed{2}}, X^{\boxed{2}}\ of\ X^{\boxed{1}}$
  $X \rightarrow X^{\boxed{1}}\ de\ X^{\boxed{2}}, X^{\boxed{1}}\ 's\ X^{\boxed{2}}$
  The "de" controls the reordering and also the speed of decoding

- Glue rules: allow the grammar to divide a French sentence into a sequence of chunks and translate one chunk at a time. Glue rules are formalized as follows:
  $(1) S \rightarrow SX$
  $(2) S \rightarrow X$

18 String to Tree Machine Translation

18.1 GHKM Rules Extraction

GHKM is short for "Gallery Hopkins Knight and Marcu" ("What's in a Translation Rule?." HLT-NAACL. 2004). GHKM is another way of rule extraction. It parses the English sentences first and then extract the rules.

Figure 2 shows a parse tree of the English side of a sentence pair:

We extract the rules based on the term "Frontier nodes". Frontier nodes are nodes (including leaf nodes) whose yield compose a phrase, namely are consistently aligned to French words (See the root node of each sub-tree in the shape area of Figure 2). From Figure 2, we can see that PRP is a frontier node because "He" is the leaf node and it consistently aligned to the French side "Il". RB is not a frontier node because the leaf node is "not", which aligned to the French side word "ne" and "pas". The "va" in the middle of "ne" and "pas" are not aligned to words within the leaves of RB, so it's not consitently aligned and not counted as a frontier node. However, the parent node of RB, VP, is a frontier node. It dominates the leaves of "not go", which consistently aligns to the French side "ne va pas". So it is an frontier node.

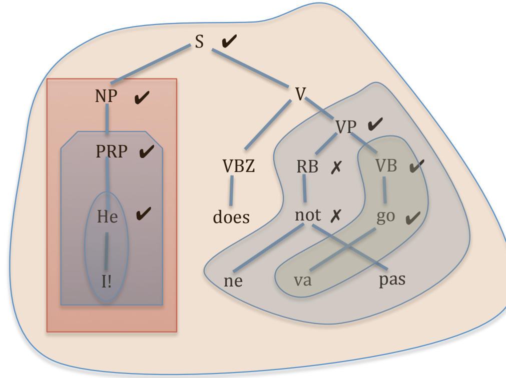


Figure 19: Rules Extraction using Frontier node.

We extract rules by cutting at the frontier nodes. The rules extracted are different if we put different restrictions on the extraction procedure.

- Minimal rules: we only extract the minimal fragment rooted at the frontier node that is subtree to any other rules rooted at the same node. We just extract one rule at each frontier node, so the total number is no more than the # of nodes in the parse tree. The number of rules extracted is O(n)

  For example:

  $S \rightarrow NPVP,\ NPVP \quad PRP \rightarrow He, I!$

  More interesting is for VP:

  $VP \rightarrow not\ VB, ne\ VB\ pas$

- General rules: we put no restriction extract all possible rules, we make a binary decision at every frontier node, so the total # of rules is exponential, which is O(2^n)

- Binary rules: the # of rules extracted is O(n^3)

The final strategy adopted is: extract minimal rules and combine two adjacent rules, the # of rules extracted is O(n^2)

Now we have a question, are all the minimal rules binarizable? The answer is no!

For a tree shown in Figure 3, we can extract the rule: $S \rightarrow CBA, ABC$, and we can binarize this rule to the following form:

$(1) S \rightarrow XA, AX$

$(2) X \rightarrow CB, BC$

However, with a parse tree provided in Figure 4, we cannot binarize the rules.

Even if the English parse tree is binary, the result synchronous grammar rule can still be not binarizable. Just like the example shown in Figure 5

18.2 Decoding

The decoding is the same with hiero, we choose the English sentence that has the highest score for the following model:

$$\underset{e}{\text{argmax}} \left(\prod_{r \in d} P(r) \right) P_{LM}(e) \tag{97}$$

We parse the French side, and keep the top K result at each chart. Then we apply LM at each node remained after the pruning. The cost of the beam search is $|G|n^3 K \log K$ (using better K best parsing of Liang Huang 05)

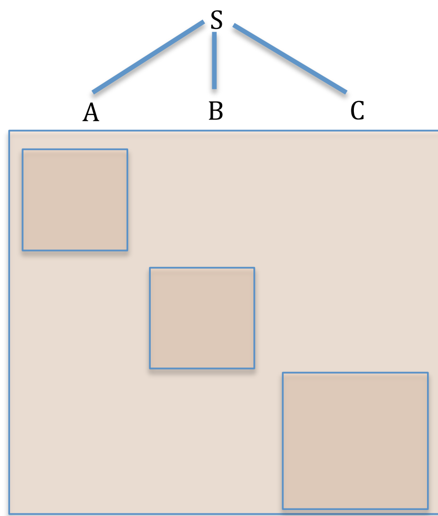


Figure 20: A binarizable example.

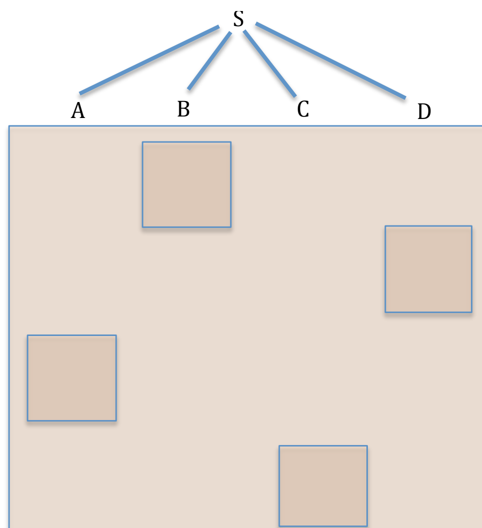


Figure 21: A non-binarizable example.

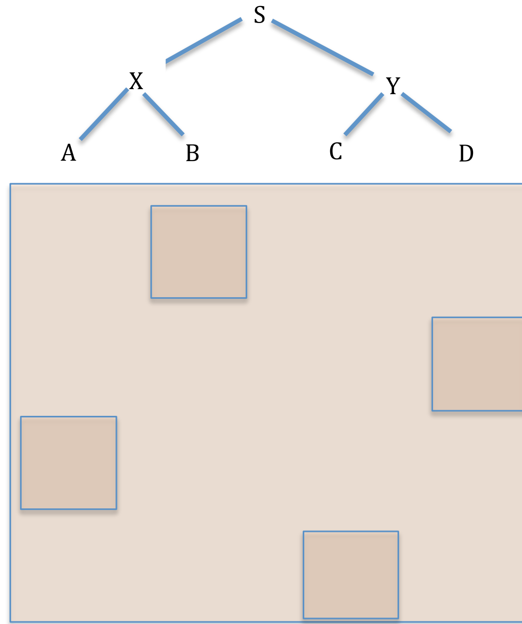


Figure 22: Binary but non-binarizable example.

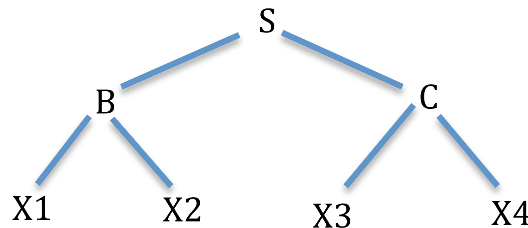


Figure 23: A simple parse tree.

If we include the LM, then the cost is $|G|n^{3+4(m-1)}K \log K$, which is not practical for machine translation. So the realistic way is to use cube pruning to prune the chart, and then use LM in the remaining nodes.

19 Tree to String Machine Translation

The rules are the same with string to tree machine translation.

As for decoding, we apply bottom up: we sort the nodes in topological order and apply rules at different cuts (frontier node) of the trees (We don't know French to guide the cut)

Suppose we have the rules shown in Figure 7. We can build a parse tree shown in Figure 6 using bottom-up and get the strings on the English side by applying the rules

The pseudo code for tree to string machine translation:

The time complexity of this algorithm is $|G|nK \log K$, which is better than $|G|n^3K \log K$ of string to tree machine translation. However, in decoding we have to rely on the parser, which might make it less accurate.

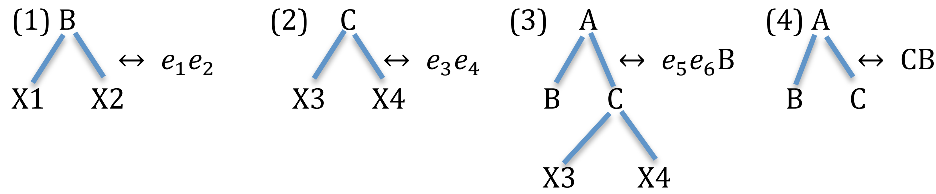


Figure 24: Tree to string rules.

Algorithm 20: *Decoding of Tree to String MT*

1: **for** n nodes in bottom-up **do**
2: **for** r rules that match n **do**
3: n_1, n_2 for r.h.s. nonterminals in r
4: **for** $h_1 \in stack[n1]$ **do**
5: **for** $h_2 \in stack[n2]$ **do**
6: Push(stack[n], r(h_1, h_2))
7: **end for**
8: **end for**
9: **end for**
10: **end for**

20 Evaluation and Parameter Tuning in Machine Translation

The weight vector of the features we introduced in the earlier sessions is usually set to maximize the evaluation metric. Human evaluation is very expensive and time-consuming. Evaluating the performance of a machine translator should be automatic. In this lecture note we firstly introduce evaluation metrics for MT and then discuss how we tune parameters with respect to them.

By far we have seen that in parallel corpus we have pairs of French and English sentences. But one English sentence is not enough, and different people could translate the same french sentence in a different way. So suppose we have multiple English sentences as the correct ”human reference” sentences and we call the candidate system's result the ”output”. Standard evaluation metric for MT is BLEU.

20.1 BLEU: Bilingual Evaluation Understudy

BLEU is one of the most popular automated and pretty simple metrics for evaluating a machine translated output. Main idea in BLEU metric is to count how many N-grams (up to 4-grams) are common between machine output and any of the human references, which is a precision score –roughly speaking we want to count how many correct N-grams are in the output and divide it to the number of all produced N-grams. One candidate system could have produced more than once sentences for which we will sum over. Following is the BLEU score formula:

$$BLEU = \prod_{n=1}^{4} P_n \tag{98}$$

Where P_n is:

$$P_n = \frac{\sum_{i \in sentences} \sum_{w \in output[i]} \min\{count(w, output[i]), count(w, references[i])\}}{\sum_{i \in sentences} \sum_{w \in output[i]} count(w, output[i])} \tag{99}$$

Where $\min\{count(w, output[i]), count(w, references[i])\}$ is called “clipped count”. Clipped count sets a minimum on number of repeating a word which takes care of the outputs such as “the the the ... the” in

which the candidate system has just ouput one correct word.

How about recall score? Here we cannot have a recall score, as there are more than one correct translations. But still we can do something about outputs such as "the", which is a very short output but has precision of 1.0 if 'the' appears in any of the reference sentences. BP is brevity penalty (penalty for producing short outputs) which is defined as below:

$$BP = \min\{1, e^{1-\frac{r}{c}}\} \quad (100)$$

and r is length of the reference sentence and c is the length of the output. Using BP the new BLEU score will be:

$$BLEU = BP. \prod_{n=1}^{4} P_n \quad (101)$$

As you can tell, BLEU does not account for producing a good English (e.g., coherent or grammatically correct). Though, it has been observed that BLEU score is highly correlated with human evaluation score, which make BLEU a very good yet simple metric. There are many ways to make BLEU better. Here are a few of them:

- Run a part of speech tagger on both output and reference and use the tagged words for computing precision.
- Run a parser on both output and reference. Then give score points to syntactic constituents in reference that are aligned to acceptable constituents in output.
- Use the set of synonyms of a word, not only the exact words.

20.2 Tuning Parameters

20.2.1 MERT: Minimum Error Rate Training

Determining the weights for features of a translation system's decoding model is usually performed via MERT, which aims at maximizing the evaluation metric, i.e., minimizing the error rate.

Here is what we attempt to optimize:

$$\underset{w}{\text{argmax}}(BLEU(\underset{d}{\text{argmax}}\, w^T f(d), reference)) \quad (102)$$

Where d is a derivation with highest score.

Our weight vector as introduced in previous lectures is 11-dimensional, but for now let's assume we only have two features and the weight vector is 2D. In figure 25 you can see two weights and the blue score in the graph. This graph is piecewise constant (with respect to BLEU axis), not smooth. So we cannot apply gradient ascent for optimization because the surface is not a convex. Och (2003) suggests an efficient line optimization method for this. It computes all pieces and picks the max.

MERT strategy can be explained in the following steps:

1. Choose search direction: choose one of the weights.
2. Optimize exactly along search direction: change the weight vector along the selected direction –only works for up to 12 features.

As mentioned ealier, MERT wants to compute all pieces and find the max. What we want to reach (one weight –search direction– considered each time) looks like the graph in figure 26.c. Such graph can be drawn using a graph of model score vs. weight changes. Each line in 26.a is a hypothesis with slope $\Delta w^T f$. By adding hypotheses one at a time, we get piecewise linear convex function as a result of intersection of lines. This convex function is bolded in figure 26. After finding all intersection points, we can draw the objective graph as in 26.c. It should be noted that the weights are not independent. So doing this one weight at a time should be done iteratively, i.e., decision about the optimum w_1 could be changed after changing w_2 and so on.

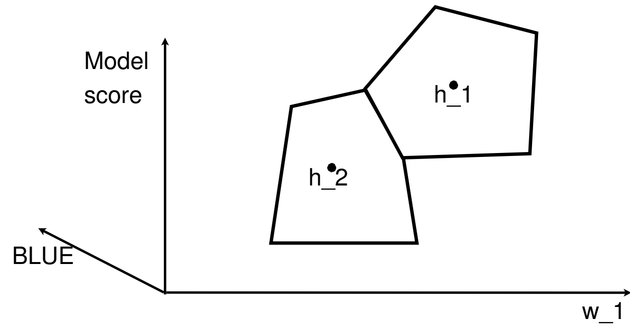


Figure 25: The surface made by weights. The points are hypothesis English sentences. Model score is $w^T f$.

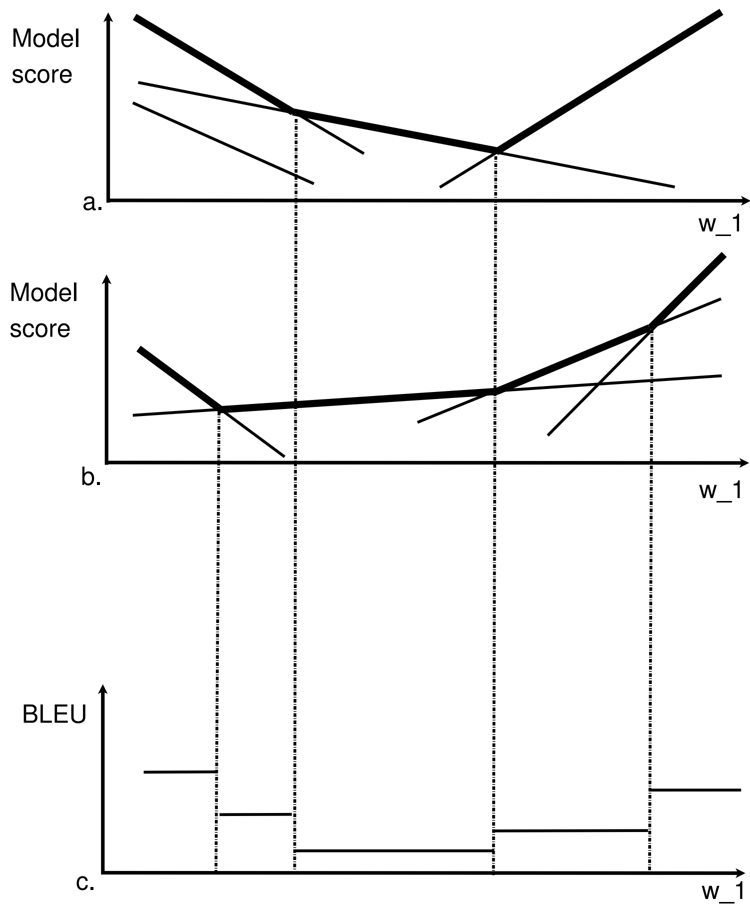


Figure 26: Visualizing MERT method for optimizing the parameters

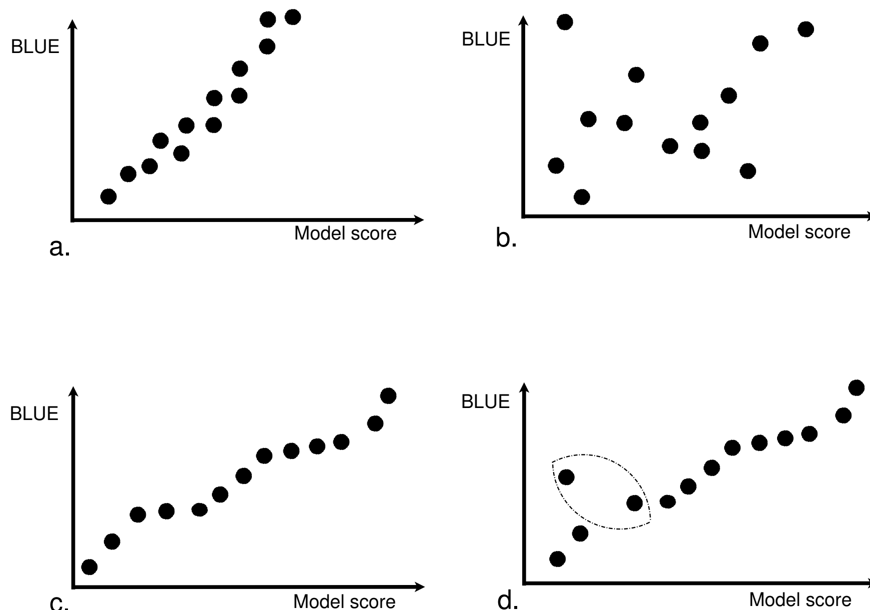


Figure 27: Correlation between BLEU and model score of hypothesis sentences

20.2.2 PRO: Pairwise Ranking Optimization

We expect the model score ($w^T f$) and BLEU score to be correlated, i.e., we expect to have a regression line among points (translation hypotheses) in the 2D BLEU vs model score graph (figure 27.a). However, the real relation between these looks like figure 27.b. In PRO we aim at making model and BLEU score correlated in two steps: ranking step and pairwise step. The goal of tuning in PRO is to learn a weight vector which results in assigning a high score to a good translations and a low score to a bad translations.

In ranking step we order the points according to model score, and expect it to be also ranked according to BLEU score. Figure 27.c depicts an example of a solved ranking. We are not looking for a straight regression line, only a monotonic function. We want to find weight vector which makes points look like this.

In pairwise part we go over each pair of points and see if they are ordered as expected, and make adjustment if not. An example of a pair which requires adjustment is shown in 27.d. If the point are too close (BLEU difference > 0.5), we do not care about ranking anymore. Algorithm 21 explains it more.

Algorithm 21: Pairwise part of PRO algorithm

```
1: for pair of points (i,j) do
2:   if BLEU difference > 0.5 then
3:     y_{i,j} = sign(BLEU(i) − BLEU(j))
4:     ...regular maxent classifier for training p(y_{i,j}|f_i − f_j)...
5:   end if
6: end for
```

In MERT, in decoder we return only one hypothesis: we are so much sensitive to one specific sentence, which could result in over-fitting. However, in PRO we look at best model score as well.

20.3 Final Remarks

Machine translation is a supervised task, as we have the correct translation paired with each test input. However, we have hidden variable (alignments) which require some unsupervised methods. MT is dis-

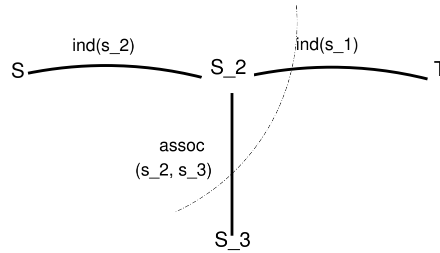


Figure 28: A sample graph for classifying two items. S is source node and T is sink node. A cut has been shown with dashed line.
After modeling the problem in this, finding the minimum cut will become the objective. Ford-Fulkerson algorithm could be used here for finding min-cut which equals max flow in the graph.

criminative, as we use MERT. However, the EM part for finding out hidden variables is generative.

21 Sentiment Analysis

In this lecture we will have a look at the paper "A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts" by Bo Pang and Lillian Lee. The problem at hand is a polarization problem in the context of movies: given a review of a user, determine if it is a negative or a positive review. Consider a website such as Rotten Tomatoes. A user can write a review together with giving a score out of 100 to the movies. So we already have the correct label (negative/positive) for each review based on the score that the user has given.

By simply training a SVM, one can easily do the label learning. The paper at hand adds another characteristic to the model which is discriminating between subjective and objective sentences. Objective sentences are the ones which describe the plot of the movie and subjective sentences are the ones the user has talked about his/her opinion and whether or not he/she liked the movie. The authors have tried both Naive Bayes and SVM For classifying the sentences based on objectivity and subjectivity. The question here is that how to do the training while we have no pre-annotated reviews with subjectivity and objectivity distinguished. The answer is that the authors have used online existing plot summaries (which you know is objective) and user snippets (subjective) from different movie databases for training the classifier.

In this paper, there is one more heuristic considered: objective sentences should be adjacent and subjective sentences should be adjacent. This heuristic gives us the following objective function:

$$min_{c_1,c_2}\{\sum_{x\in c_1} ind_2(x) + \sum_{x\in c_2} ind_1(x) + \sum_{x_1\in c_1, x_2\in c_2} assoc(x_1,x_2)\} \tag{103}$$

Here c_1 is the cluster denoting objective sentences and c_2 is the one denoting subjective sentences. The function $assoc$ gives a cost if two sentences are close but labeled differently. ind_x indicates the class to which the sentence tends to belong. The definition for $assoc$ is as follows:

$$assoc(s_i,s_j) = \begin{cases} f(j-i).c & \text{if } (j-i) < \tau \\ 0 & \text{if } o/w \end{cases} \tag{104}$$

$$f(d) = \begin{cases} 1 \\ e^{1-d} \\ 1/d^2 \end{cases} \tag{105}$$

How to solve the optimization problem presented in 103? This paper comes up with the idea of modeling the problem as a min-cut on a graph. In figure 28 you see this graph-based model.

Overall, this paper shows that using the minimum-cut approach results in an efficient algorithms for sentiment analysis.

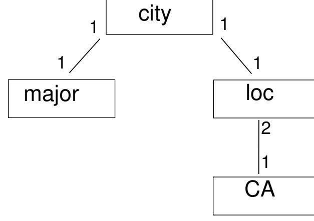


Figure 29: A sample DCS tree.

22 Compositional Semantics

In this section we will go over tha paper “Learning Dependency-Based Compositional Semantics” (Percy Liang et al.). This work learns to map questions to answers via latent logical forms, i.e., without building logical forms. The final goal is to be able to apply a predicate on the “world” database and get the update.

V is defined to be the set of all values such as ‘3’ and ‘CA’. P is defined to be a set of predicate such as ‘state’ and ‘count’. A world w is mapping from each predicate p to a set. For instance $w(state) = \{CA, NY, ...\}$. The representation model used here is called dependency-based compositional semantics (DCS). The logical forms in DCS are called DCS trees. In these trees nodes are labeled with predicates, and edges are labeled with the relations. Figure 29 shows a sample DCS tree. The written form of this tree is as follows:

$$z = \langle city;^1_1 :< major >;^1_1 :< loc;^2_1 :< CA >>\rangle \tag{106}$$

We denote this as $\langle z \rangle_w = \{SF, LA, ...\}$ We can compute the denotation $\langle z \rangle_w$ of a DCS tree z by exploiting dynamic programming:

$$\langle < p;^{j_1}_{j'_1} : ... > \rangle_w = w(p). \cap \bigcap_{i=1,...,m} \{v : v_{j_i} = t_{j'_i}, t \in \langle c_i \rangle_w\} \tag{107}$$

So we compute the set of tuples v consistent with the predicate at that node v at each node, then we intersect all of conditions up to its children.

Now the question is how to handle queries which require counting? For example if we want to know “how many major cities there are?”. The authors introduce a new aggregate relation, notated Σ. Assume a tree $\langle \Sigma : c \rangle_w = \{\langle c \rangle\}$, whose root is connected to a child c via Σ. If the denotation of c –as computed earlier– is a set s, then the parent’s denotation become a singleton set containing just s. Figure 30 shows an example.

How to handle quantifiers, such as the most or the least? If you have a comparison quantifier, the root x of the DCS tree is the entity which should be compared, the child c of a C relation is the comparative (or superlative), and then its parent p contains the information which is used for comparison. Figure 31 shows DCS tree for the query “state bordering the most states”. Here we have to delay the argmax, until reaching the node containing the answer. Here is how we denote such DCS tree:

$$\underset{x}{\operatorname{argmax}}\{|Y| \ s.t. \ state(x) \wedge border(x, y) \wedge y \in Y \wedge state(y)\} \tag{108}$$

How do they do semantic parsing in this paper? The task here is to map natural language queries to DCS trees. Given a query (utterance as called in the paper) $x = (x_1, ..., x_n)$, $Z_L(x) \subset Z$ will be the set of permissible DCS trees for x. The approach is a kind of dependency parsing, based on recursion: For each span i..j , we build a set of trees $C_{i,j}$ and set $Z_L(x) = C_{0,n}$. Each of the sets $C_{i,j}$ is built by combining the trees of its sub-spans $C_{i,k}$ and $C_{k',j}$ (split points are k and k’). $C_{i,j}$ is defined recursively as follows:

$$C_{i,j} = F\left(A\left(L(x_{i+1..j}) \cap \bigcap_{i \le k \le k' < j, a \in c_{i,k}, b \in c_{k',j}} T_1(a, b)\right)\right) \tag{109}$$

Here the combinations are augmented by a function A (which makes the tuples) and filtered by a function F.

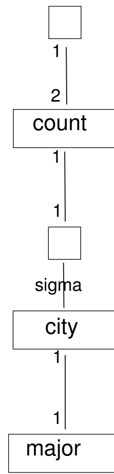


Figure 30: A sample DCS tree for counting.

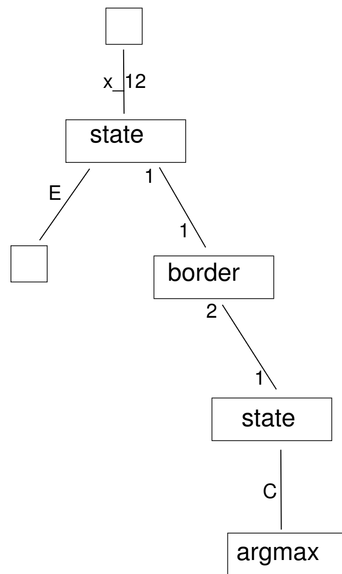


Figure 31: A sample DCS tree for comparative/superlative constructions.

The discriminative semantic parsing model (which is CRF) will have the following derivative:

$$\frac{\partial L}{\partial w} = E[F|xy] - E[F|x] \tag{110}$$

23 Lexical Semantics

23.1 Word Senses

WordNet

synsets = synonyms

For every word, there are many word senses. We can find what senses a word has by looking at WordNet, which basically contains every word in English. WordNet covers nouns, verbs, adjectives, etc. Every word has more than one sense, and senses may have more than one meaning. Example of two words with the same meaning: {large.1 big.2 ...} = "of great size". There are relationship between synsets and hypernyms. Hypernyms give a way of clustering words. For instance, tea is a beverage that you can drink. WordNet has senses, but there can be too many - as an example, say 3 out of 7 senses cannot be told apart. When WordNet has too much information, people often collapse the WordNet senses first to get rid of ones that are too similar. Senses are necessary to handle words like the noun *plant*, which can refer to the organic plant or a factory.

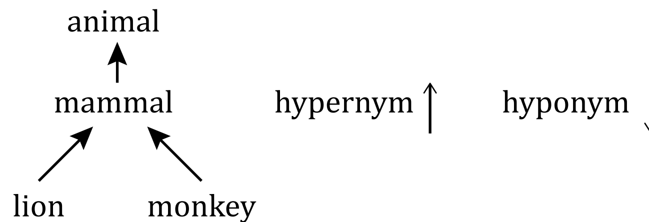


Figure 32: The relationships between synsets and hypernyms/hyponyms.

Is pink a hyponym of red? No, not really.

The hierarchy for WordNet is deepest for nouns. This information is meant to provide an understanding of what WordNet is and what is referred to when papers mention WordNet.

23.2 Word Sense Disambiguation

Paper: Yarowsky 1995.

23.2.1 Bootstrapping

Bootstrapping is most useful when there is a small amount of labelled data and a large amount of unlabelled data. You train a classifier on the small set of labelled data, find the data that you are most confident about, train again, then continue to find more data to retrain on. Sometimes this works, and sometimes this does not. The guesses on unlabelled data may be wrong, which would emphasize the wrong guesses. Word sense disambiguation may be the most successful example of bootstrapping.

Every single word is a separate problem. For example, say that after one million words are taken from WSJ and each sense that appears in it is annotated, you find that the word "plant" only appears seven times. Like this, it is difficult to get a significant amount of training data, which is why we need to use unlabelled data.

The most significant for word sense disambiguation can be the other words that are nearby in the same sentence or in a fixed width window. Bootstrapping is partially supervised. The features are words within some sentence or window. The decision list (or your favorite classifier) has one sense per discourse, and is a decision tree with one branch. There is sense per collocation. This is convenient since we can just add to the end of the list, and don't need to retrain. Decisions are made immediately.

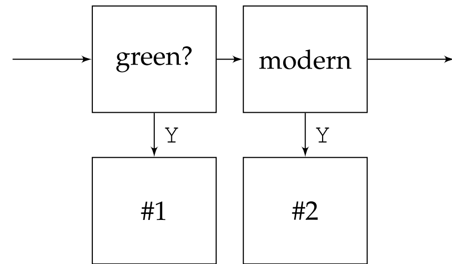


Figure 33: A decision list example.

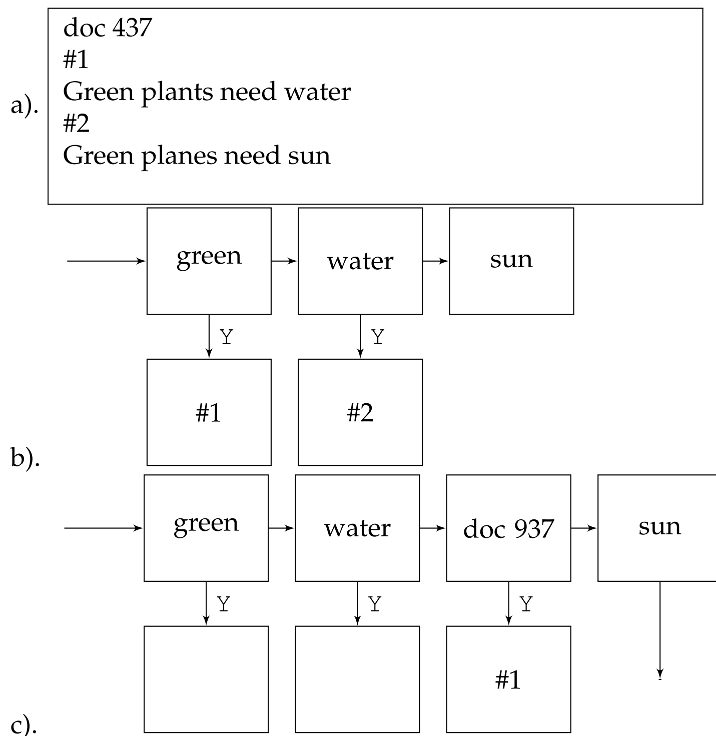


Figure 34: a). Example document used as a feature. b). A decision list. c) A decision list with a document added as a feature.

In this paper, he only used 2 senses, so random accuracy was 50 percent. WordNet was used as a seed to inform if an instance can be classified as #1. A specific document can be added as a feature (Figure 3). When bootstrapping is done, the document can be deleted.

23.2.2 Lesk

Lesk = compare with definition.

The Lesk algorithm takes a sentence where the word is used and compares it with the definition by counting overlapping words to see if they match. It is a knowledge-based approach.

23.2.3 Co-training

Co-training uses two classifiers for bootstrapping. For instance, if one classifier is confident on some points and the other is confident on other points, even if the classifiers do poorly individually, they can do much better together. However, bootstrapping algorithms can make mistakes easily and then feed off on their own mistakes, and are dependent on the seed for performance. If existing definitions are used as the seed, bootstrapping can be considered to be unsupervised, but they can also be considered semi-supervised.

23.2.4 Using Wikipedia for Word Sense Disambiguation

The Mihalcea paper describes how we can use Wikipedia for word sense disambiguation. The Wikipedia links such as /car_(vehicle) and [[car_(vehicle)—car]] can be used to disambiguate words. This is done by taking all Wikipedia articles with the same WordNet sense and using them as training sets. If something does not exist in the training set, the classifier will get it wrong, but that is acceptable. When run on Wikipedia, the classifier achieved 84% accuracy, but only achieved 68% accuracy when run on SENSEVAL. However, as SENSEVAL is harder, a performance of 68% accuracy on that data can be considered to be as good as higher accuracies on other systems. This approach will work even better in the future as Wikipedia is still growing and can provide even larger training sets. However, this method works best on nouns since Wikipedia generally doesn't have many articles on verbs or other parts of speech.

Wikipedia is also a great source for more than word sense disambiguation, such as for determining hypernyms and information extraction.

23.3 Word Sense Clustering

Lin 1998.

Given some words, we would like to find synonyms. Words are clustered together if they appear in the same context. This can be affected by how often the words are seen together, and if they are often seen together randomly.

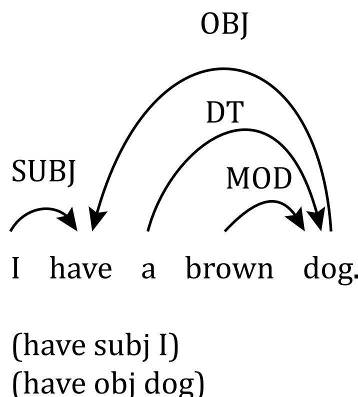


Figure 35: Step for word sense clustering.

$$I(w, r, w') = -\log \frac{P(r)P(w'|r)P(w|r)}{P(w, r, w')} \tag{111}$$

$I(w, r, w')$ denotes the amount of information contained in $(w, r, w') = c$, which denotes the frequency count of the dependency triple (w, r, w'). $I(w, r, w')$ is pointwise because individual values are used, and

denotes the mutual information between w and w'.

$$I(A,B) = D(P(A,B)||P(A)P(B)) \tag{112}$$

Where D is the divergence, and include P(A)P(B) as if A and B are independent.

$$I(A,B) = \sum_{a,b} P(a,b)\log(\frac{P(a,b)}{P(a)P(b)}) \tag{113}$$

The KL divergence between real joint distributions and something else, if really independent, is zero. The mutual relatioship between two words is taken in context. The goal is to find similarities between any two words. For I(w,r,w'), if the denominator P(w,r,w') is bigger than the numerator, then the value is negative. $T(w_1)$ contains the non-zero entries for w_1.

$$sim(w_1,w_2) = \frac{\sum_{r,w \in T(w_1)\cap T(w_2)} I(w_1,r,w) + I(w_2,r,w)}{\sum_{r,w\in T(w_1)} I(w_1,r,w) + \sum_{r,w\in T(w_2)} I(w_2,r,w)} \tag{114}$$

Here, similarity as computed by Equation 4 can never be greater than 1 or less than 0, and is a kind of vector similarity as the similarity of two vectors is measured. w_1 is a big vector of I values for different contexts and gives you the vector representation of a word.

$$w_1 = \begin{pmatrix} I(w_1,r,w) \\ I(w_1,r,w') \\ I(w_1,r,w'') \\ \vdots \end{pmatrix} = \overrightarrow{w_1}$$

Always keep only the positive entries of w_1, and throw away the negative entries:

$$\overrightarrow{w_1} = (w_1)_+ \tag{115}$$

$$\sum min(w_r,0) = \sum w'_i = |\overrightarrow{w'}|_1 \tag{116}$$

This is just one way to measure similarity between two vectors. Another way to check similarity would be like the dot product, except that we just want the cosine, which means we want to divide the dot product value. To calculate the cosine similarity:

$$cos(w_1,w_2) = \frac{\overrightarrow{w_1}^T\overrightarrow{w_2}}{\sqrt{\|\overrightarrow{w_1}\|_2\|\overrightarrow{w_2}\|_2}} \tag{117}$$

We can also measure the size of the set without caring what i is and just find the number of positive values.

$$sim_{cosine} = \frac{|T(w_1)\cap T(w_2)|}{\sqrt{|T(w_1)||T(w_2)|}} \tag{118}$$

This is the same as taking the vector from before and rounding all values to either 0 or 1.

$$sim_{Dice} = \frac{2|T(w_1)\cap T(w_2)|}{|T(w_1)|+|T(w_2)|} \tag{119}$$

$T(w_1)$ is the number of non-zero entries for w_1 and $T(w_2)$ is the number of non-zero entries for w_2.

Equations 7 to 9 above all return numbers between 0 and 1. We can also get pretty good word alignment from these equations. Instead of counting the number of non-zero entries, we can count the number of times we see an English word and the number of times we see a French word.

$$sim_{Jacard} = \frac{|T(w_1) \cap T(w_2)|}{|T(w_1)| + |T(w_2)| - |T(w_1) \cap T(w_2)|} \tag{120}$$

A problem with this approach is that sometimes we cannot differentiate between words with opposite meanings. For instance, the algorithm returns the result that *brief* and *lengthy* are similar.

23.4 DIRT - Discovering Inference Rules from Text

Lin and Patel 2001.

DIRT focuses on relations instead of on words, so that the goal is to find synonymous relations instead of words. *X solves* $Y \leftrightarrow X$ *finds a solution for* Y is an example of a relation that we are trying to extract.

Before, r was one jump in the dependency tree, but now r can be many jumps in the tree.

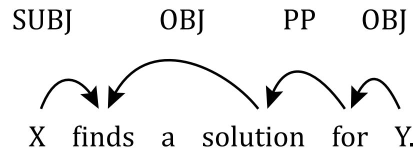


Figure 36: Example sentence.

The entire chain is a relation. For a sentence of length n, there can be n^2 relations. The similarity for two relations can be calculated using Equation 11.

$$sim(r_1, r_2) = \sum_{\substack{(x_1 r_1 y) \\ (x_2 r_2 y)}} sim(x_1 x_2) + \sum_{\substack{(x r_1 y_1) \\ (x r_2 y_2)}} sim(y_1 y_2) \tag{121}$$