Discriminative Sequence Labeling

Wu Ke

March 4, 2013

1 Basics

1.1 Linear Classifiers

In a classification problem, we are given the input x and want to find out which category it belongs to in a given label set Π. Information from the input x is often represented as a feature vector $\phi(x)$. The basic idea of linear classifiers, then, is to have a weight vector $\mathbf{w}_z$ for each class label z, in the same dimension as $\phi(x)$, to distinguish input from different categories. The label for an input is predicted to be the one whose weight vector gives the highest inner product value with $\phi(x)$, i.e. a linear classifier predicts the label $\hat{z}$ by letting

$$\hat{z} = \arg\max_{z} \mathbf{w}_z^{\intercal} \phi(x) \tag{1}$$

For example, recognizing a vowel given a snippet of recording can be formalized as a classification problem. If we are interested in recognizing [i, u, ɑ], then $\Pi = \{\text{i, u, ɑ}\}$. The feature vector is a 3-dimensional vector $(f_1, f_2, 1)^{\intercal}$, consisting of the first two formants and a bias term.[1] To see why this simple model works, note that,

- [i] exhibits low f_1 and high f_2;
- [u] exhibits low f_1 and low f_2;
- [ɑ] exhibits high f_1 and low f_2.

If we treat the first two formants as the X- and Y-axis of a plane, then the weights $\mathbf{w}_{\text{i}}, \mathbf{w}_{\text{u}}, \mathbf{w}_{\text{ɑ}}$ divide the plane into three parts, each part containing one vowel, as illustrated in Figure 1.

An alternative way of representing the model, which will be convenient in the following discussion, is to concatenate all weight vectors into a single vector $\mathbf{w}$ and define the following $\phi(x,z)$ to represent features: $\phi(x,z)$ is in the same dimension as $\mathbf{w}$ and the elements in $\phi(x,z)$ are all zeros except that those dimensions corresponding to $\mathbf{w}_z$ are set to $\phi(x)$. The classifier then predicts the label $\hat{z}$ by letting

$$\hat{z} = \arg\max_{z} \mathbf{w}^{\intercal} \phi(x,z) \tag{2}$$

Using our vowel recognition example, the new single weight vector is just

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}_{\text{i}} \\ \mathbf{w}_{\text{u}} \\ \mathbf{w}_{\text{ɑ}} \end{bmatrix} \tag{3}$$

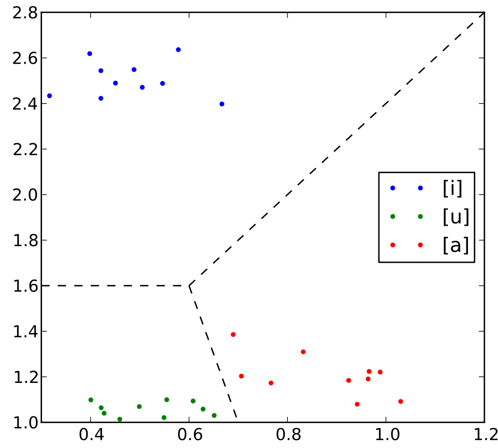


Figure 1: Vowels and their formants

1: Initialize **w** as a zero vector
2: **for** iteration i less than T **do**
3:     **for all** $x^{(j)}, z^{(j)}$ in the training data **do**
4:         $z^* = \arg\max_z \mathbf{w}^\intercal \phi(x^{(j)}, z)$
5:         **if** $z^* \neq z^{(j)}$ **then**
6:             $\mathbf{w} \leftarrow \mathbf{w} + \phi(x^{(j)}, z^{(j)}) - \phi(x^{(j)}, z^*)$
7:         **end if**
8:     **end for**
9: **end for**

Figure 2: The Perceptron Algorithm

$\phi(x, \text{i})$ is then $(f_1, f_2, 1, 0, 0, 0, 0, 0, 0)$; $\phi(x, \text{u})$ is then $(0, 0, 0, f_1, f_2, 1, 0, 0, 0)$; $\phi(x, \text{a})$ is then $(0, 0, 0, 0, 0, 0, f_1, f_2, 1)$.

A linear classifier is usually learned from a set of labeled data $\{(x^{(1)}, z^{(1)}), \ldots, (x^{(N)}, z^{(N)})\}$ (called *training data*). There are many algorithms for learning a linear classifier. The simplest one is the *perceptron algorithm* in Figure 2. The core of the algorithm is the perceptron update in Line 6. The intuition behind is to push the weights closer to the correct prediction $z^{(j)}$ and further from the current best prediction z^* whenever the current $\mathbf{w}$ leads to a mistake. To see how this is done, consider the k-th dimension of a feature vector,

- If $\phi(x^{(j)}, z^{(j)})_k = \phi(x^{(j)}, z^*)_k$, then changing w_k has no effect on the best prediction and thus we do not update w_k.
- If $\phi(x^{(j)}, z^{(j)})_k < \phi(x^{(j)}, z^*)_k$, then w_k will be decreased. $w_k \phi(x^{(j)}, z^*)_k$ shrinks more than $w_k \phi(x^{(j)}, z^{(j)})_k$ and thus $z^{(j)}$ is up-weighted while z^* is down-weighted.
- If $\phi(x^{(j)}, z^{(j)})_k > \phi(x^{(j)}, z^*)_k$, then w_k will be increased. $w_k \phi(x^{(j)}, z^*)_k$ increases less than $w_k \phi(x^{(j)}, z^{(j)})_k$ and thus $z^{(j)}$ is up-weighted while z^* is down-weighted.

1.2 The Sequence Labeling Problem

Generally speaking, sequence labeling is a problem of predicting the state sequence $\mathbf{Z}$ given the observation sequence $\mathbf{X}$. Any single observation x_i in $\mathbf{X}$ comes from a known finite set Σ and any single state z_i in $\mathbf{Z}$ comes from a known finite set Π. A lot of problems can be formalized as a sequence labeling problem. For example, in part-of-speech (POS) tagging, we can treat each POS tag as a state, and each word as an observation. Then Σ is the set of all possible word types; and Π is the set of POS tags.

One can treat prediction at each single observation x_i as a classification problem and train a linear classifier to solve the problem. However, a linear classifier does not make use of the knowledge of any possible dependency between states and thus usually performs not that well as a sequence labeling model. On the other hand, the Hidden Markov Model (HMM) is a popular example of sequence labeling models that is capable of modeling the dependency of neighboring states. The advantage of using a classifier is the flexibility to use domain knowledge to engineer effective features. However, there is no such notion of "features" in HMM and observations are seen as unique, discrete symbols. It is therefore very difficult to incorporate domain knowledge into an HMM — one needs to translate such knowledge into a generative model, which requires much more care and effort than engineering features.

In the following sections, we will develop a family of models that combines the strength of both sides, a family of models that uses features rather than discrete symbols to represent an observation while is still capable of modeling dependency between states.

2 HMM as a Linear Model

Recall that an HMM defines a joint distribution of the observation sequence $\mathbf{X} = x_1 x_2 \ldots x_n$ and its hidden state sequence $\mathbf{Z} = z_1 z_2 \ldots z_n$. Let π be the initial state distribution where $\pi_i = P(z_1 = i)$, T be the transition matrix where $T_{ij} = P(z_{t+1} = i | z_t = j)$ and O be the observation matrix where $O_{ij} = P(x_t = i | z_t = j)$; then,

$$P(\mathbf{X}, \mathbf{Z}) = \pi_{z_1} \left(\prod_{i=1}^{n-1} O_{x_i z_i} T_{z_{i+1} z_i} \right) O_{x_n z_n} \tag{4}$$

[1] http://en.wikipedia.org/wiki/Formant

The log-probability is thus

$$\log P(\mathbf{X},\mathbf{Z}) = \log \pi_{z_1} + \sum_{i=1}^{n-1} \left(\log O_{x_i z_i} + \log T_{z_{i+1} z_i} \right) + \log O_{x_n z_n} \tag{5}$$

If the transition from state j to state i appears m times in $\mathbf{Z}$, then the term $\log T_{ij}$ also appears m times in the summation. Similarly, if the number of times that state j emits observation i in $(\mathbf{X},\mathbf{Z})$ is m, then the term $\log O_{ij}$ appears m times. This gives us another way to compute the log-probability. Given $\mathbf{X}$ and $\mathbf{Z}$, define the following two matrices $\Phi^{\mathbf{Z}}$ and $\Phi^{\mathbf{X},\mathbf{Z}}$, where $\Phi^{\mathbf{Z}}_{ij}$ is the number of times the transition from state j to state i is taken in $\mathbf{Z}$ and $\Phi^{\mathbf{X},\mathbf{Z}}_{ij}$ is the number of times symbol i is emitted from state j in $(\mathbf{X},\mathbf{Z})$. Then,

$$\log P(\mathbf{X},\mathbf{Z}) = \log \pi_{z_1} + \left(\sum_{i,j} \Phi^{\mathbf{Z}}_{ij} \log T_{ij} \right) + \left(\sum_{i,j} \Phi^{\mathbf{X},\mathbf{Z}}_{ij} \log O_{ij} \right) \tag{6}$$

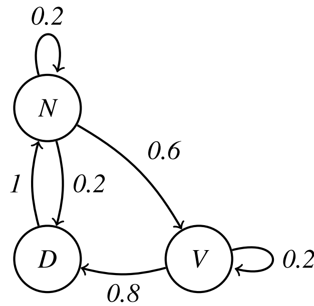
$\log \pi_{z_1}$ can be represented as the inner product of the z_1-th standard basis e_{z_1} and $\log \pi$. And by concatenating columns of the matrices into a single vector (for a matrix M, denote the operation as $\mathrm{vec}(M)$), the above can be written as a inner product of two vectors,

$$\log P(\mathbf{X},\mathbf{Z}) = \begin{bmatrix} \log(\pi) \\ \mathrm{vec}(\log T) \\ \mathrm{vec}(\log O) \end{bmatrix}^{\intercal} \begin{bmatrix} e_{z_1} \\ \mathrm{vec}(\Phi^{\mathbf{Z}}) \\ \mathrm{vec}(\Phi^{\mathbf{X},\mathbf{Z}}) \end{bmatrix} \tag{7}$$

The left vector is basically the log values of the parameters of the model and thus does not depend on $\mathbf{X}$ or $\mathbf{Z}$. The right vector describes $\mathbf{X}$ and $\mathbf{Z}$ but is agnostic about the model. This is like a linear classifier — $\begin{bmatrix} \log(\pi) \\ \mathrm{vec}(\log T) \\ \mathrm{vec}(\log O) \end{bmatrix}$ can be seen as the weight vector $\mathbf{w}$, and $\begin{bmatrix} e_{z_1} \\ \mathrm{vec}(\Phi^{\mathbf{Z}}) \\ \mathrm{vec}(\Phi^{\mathbf{X},\mathbf{Z}}) \end{bmatrix}$ can be seen as the feature vector $\Phi(\mathbf{X},\mathbf{Z})$ — with the only exception that what we are trying to predict here is not a simple class label but a sequence of labels.

Example 1. *We are studying the Pidgin English on Gliese 581 d, where the vocabulary consists of only two words — "can", which can either be a noun or a verb; "the", which can only be a determiner. After an hour of field study, we find out the natives' language faculty can be more or less approximated as an HMM.*

A sentence starts with a determiner with probability 0.8 and with a verb with probability 0.2. The transition probabilities are illustrated in the following diagram,



The observation probabilities are,

	can	*the*
D	*0*	*1*
N	*1*	*0*
V	*1*	*0*

What does the model look like if we convert it into our newly discovered linear model form? What is $\Phi(\mathbf{X},\mathbf{Z})$ *when* $\mathbf{X}$ *= "the can can can the can" and* $\mathbf{Z}$ *= "D N V V D N"?*

Answer. Let us call "can" word 1 and "the" word 2; the 3 states "determiner", "noun" and "verb" states 1, 2, and 3 respectively. The initial state distribution is $\pi = (0.8, 0, 0.2)^\intercal$. The transition and the observation matrices are respectively,

$$T = \begin{bmatrix} 0 & 0.2 & 0.8 \\ 1 & 0.2 & 0 \\ 0 & 0.6 & 0.2 \end{bmatrix} \tag{8}$$

and

$$O = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{9}$$

Therefore, $\log \pi = (-0.22, -\infty, -1.61)^\intercal$,

$$\log T = \begin{bmatrix} -\infty & -1.61 & -0.22 \\ 0 & -1.61 & -\infty \\ -\infty & -0.51 & -1.61 \end{bmatrix} \tag{10}$$

and

$$\log O = \begin{bmatrix} -\infty & 0 & 0 \\ 0 & -\infty & -\infty \end{bmatrix} \tag{11}$$

Therefore,

$$\begin{aligned} w = (&-0.22, -\infty, -1.61, -\infty, 0, -\infty, -1.61, -1.61, \\ &-0.51, -0.22, -\infty, -1.61, -\infty, 0, 0, -\infty, 0, -\infty)^\intercal \end{aligned} \tag{12}$$

Because $\mathbf{X} = [2, 1, 1, 1, 2, 1]$ and $\mathbf{Z} = [1, 2, 3, 3, 1, 2]$, we know $e_{z_1} = (1, 0, 0)^\intercal$,

$$\Phi^{\mathbf{Z}} = \begin{bmatrix} 0 & 0 & 1 \\ 2 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{13}$$

and

$$\Phi^{\mathbf{X},\mathbf{Z}} = \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 0 \end{bmatrix} \tag{14}$$

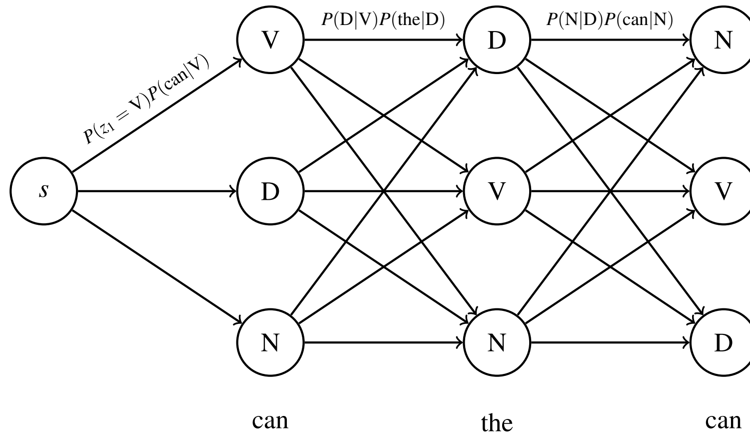
Therefore,

$$\Phi(\mathbf{X},\mathbf{Z}) = (1, 0, 0, 0, 2, 0, 0, 0, 1, 1, 0, 1, 0, 2, 2, 0, 2, 0)^\intercal \tag{15}$$

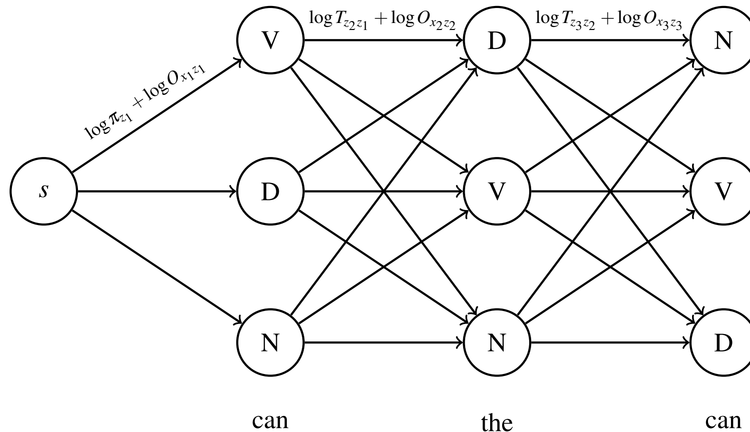
□

3 New Model, Old Viterbi

For sequence labeling, we are interested in labeling an unseen input sequence $\mathbf{X}$. In HMM this is achieved by finding $\arg\max_{\mathbf{Z}} \log P(\mathbf{Z}|\mathbf{X}) = \arg\max_{\mathbf{Z}} \log P(\mathbf{X}, \mathbf{Z})$ using the Viterbi algorithm. What does the Viterbi algorithm look like after we represent the HMM as a linear model? Let's use the example HMM from Section 2. Suppose we are tagging the sentence "can the can", the Viterbi algorithm constructs the following trellis and finds the path from s to one of the three right-most nodes with the highest probability,



If we take a log on every edge and replace probability product with sum, finding the path with the highest probability is equivalent to find the highest scoring path on the following graph,



The terms on the edges can all be found from the weight vector $\mathbf{w}$ and thus we can construct feature vectors for each edge, in the same way as we did for the whole sequence, i.e. we will count the number of times that a certain state is used as the initial state, that a certain transition takes place, and that a certain observation is emitted, but all within in a single edge. An edge can be represented as the tuple $\langle z, z', x \rangle$,[2] meaning coming from state z to state z' then emitting observation x, and let's denote the feature vector of

[2] For simplicity of notation, let's define $z_0 = s$.

this edge as $\phi(z,z',x)$. Each dimension of $\phi(z,z',x)$ has the same meaning as $\Phi(\mathbf{X},\mathbf{Z})$, i.e. they are counts of certain events. However, the dimensions of $\phi(z,z',x)$ can only be 0 or 1 and most of the dimensions will be 0. It is then easy to see, the following relation holds,

$$\Phi(\mathbf{X},\mathbf{Z}) = \sum_{i=1}^{n} \phi(z_{i-1}, z_i, x_i) \tag{16}$$

And naturally,

$$\mathbf{w}^\mathsf{T}\Phi(\mathbf{X},\mathbf{Z}) = \sum_{i=1}^{n} \mathbf{w}^\mathsf{T}\phi(z_{i-1}, z_i, x_i) \tag{17}$$

Therefore, for labeling with an HMM given as a linear model, all we need to do is to find $\arg\max_{\mathbf{Z}} \mathbf{w}^\mathsf{T}\Phi(\mathbf{X},\mathbf{Z})$. Since parts of the inner product can be distributed onto each edge, we can still use the Viterbi algorithm, simply by replacing each edge's weight with $\mathbf{w}^\mathsf{T}\phi(z_{i-1}, z_i, x_i)$.

4 Beyond HMM

So far we have seen that HMM can be represented in fancy algebra. But what we have got under the hood is still a plain-old HMM, which isn't that interesting. So what is all this fuss about, really?

Recall that what is nice about classifiers is that instead of worrying about how the data is generated from a certain distribution, one only needs to list out features that may be relevant in distinguishing data points from different categories. We now know HMMs are really a special case of a more general family of linear models and the Viterbi algorithm still works as long as the feature vector can be distributed onto each edge. As a result, we have a family of linear models where we can do efficient inference (i.e. labeling any input sequence), model state dependency and use expressive features at the same time.

From Section 3 we know, for any $(\mathbf{X},\mathbf{Z})$, $\Phi(\mathbf{X},\mathbf{Z})$ is essentially the sum of $\phi(\cdot)$ along transitions in $\mathbf{Z}$. In the case of HMM, $\phi(\cdot)$ uses the information of the previous hidden state, the current hidden state and the current observation. Can it use more information? Certainly. Since the whole $\mathbf{X}$ sequence is observed, at any location of the sequence we can extract information from $\mathbf{X}$, instead of only looking at a single observation. What about the states? Unfortunately because the only source of state information on an edge is its source and target states of the transition, looking further back or forward is very difficult if we would like stick to our Viterbi inference framework. To sum up,

> As long as $\phi(\cdot)$ gives a real-valued vector and is a function of $(z,z',\mathbf{X},i)$, where i is the current input location and z and z' are the previous and current states (i.e. z_{i-1} and z_i), it can be used to compute the feature vector over each edge.

Although the linear model has a single feature vector $\Phi(\mathbf{X},\mathbf{Z})$, but it is computed by applying the same $\phi(z,z',\mathbf{X},i)$ function at each location. Defining $\phi(z,z',\mathbf{X},i)$ becomes the sole task of defining features. Therefore $\phi(z,z',\mathbf{X},i)$ is often called the *feature template*. Since $\phi(z,z',\mathbf{X},i)$ tends to be sparse, it is convenient to describe it in a way that lists dimensions that are non-zero.

Example 2. *In an HMM, the feature template sets the following dimensions to 1,*

- *The dimension representing the transition from state z to state z';*
- *The dimension representing emission of observation x_i from state z'.*

Therefore, the feature template gives a sparse vector, which always consists of exactly two non-zero elements.

5 Examples of Sequence Labeling

With our generalized model, we are now able to incorporate richer features. When proper features are used and a proper weight vector is learned, we can often achieve better accuracy in labeling unseen data than a plain HMM.

Although there is no limitation of what value each dimension of $\phi(z,z',\mathbf{X},i)$ can take, most NLP applications use binary features. When not causing confusion, such features can be represented as tuples of predicates on $(z,z',\mathbf{X},i)$, meaning that the corresponding dimension is set to 1 if and only all the predicates are true.

Example 3. *Following Example 2, the feature templates can be equivalently represented as follows,*

- *Transition:* $[z=a,z'=b]$*, for all* $a,b \in \Pi$*;*
- *Emission:* $[z'=a,x_i=b]$*, for all* $a \in \Pi$ *and* $b \in \Sigma$*.*

Not all possible tuples are useful features. For example, features only using information from $\mathbf{X}$ and i will have the same value for any $\mathbf{Z}$, thus making no contribution in the actual prediction. Moreover, features not involving the current state z' do not have direct effect on predicting the label at position i.[3]

Now let's see a few examples of applications of sequence labeling.

5.1 POS Tagging

We begin with the classic problem of POS tagging. The two main challenges in this task are ambiguity, i.e. a single word type may have different possible tags, and out-of-vocabulary (OOV) words, i.e. tagging new words that do not appear in the training data.

To resolve ambiguity, we need broader context, which is why we are consulting the previous state z_{i-1} when scoring the current state z_i. In the simplest approach, we simply use the set of POS tags as our state set, and thus the broadest context we can get in terms of states is at the state bigram level by the transition feature $[z=a,z'=b]$. Further, since we have access to the whole input sequence $\mathbf{X}$, instead of just x_i, we can also look at neighboring words to help resolve ambiguity, making use of the fact that many common words have selectional preferences over certain categories in natural language. The following word features are often used in practice based on this intuition,

- Previous word: $[z'=a,x_{i-1}=b]$;
- Next word: $[z'=a,x_{i+1}=b]$;
- Word two words back: $[z'=a,x_{i-2}=b]$;
- Word two words ahead: $[z'=a,x_{i+2}=b]$;
- Neighboring bigrams: $[z'=a,x_{i-2}=b,x_{i-1}=c]$, $[z'=a,x_{i-1}=b,x_i=c]$, or $[z'=a,x_i=b,x_{i+1}=c]$.

One could of course also include the previous state z in the features above. However, this will increase the number of features by a factor of $|\Pi|$ times. With much more features, if there is not a sufficient amount of training data, the learning algorithms tend to overfit the training data, in a way of simply "remembering" predictions of snippets in the training data. Similarly, if we try to increase the size of the window where words

[3] A feature involving z but not z' does affect the prediction at position i, by affecting the prediction at the previous position $i-1$. Therefore such feature can be equivalently represented as a feature involving z' but not z.

are considered neighbors, or include higher order neighboring n-grams, the size of our feature space grows very quickly and the problem of overfitting may again prevent any improvement in prediction accuracy.

Usually the above features are enough for resolving ambiguity in POS tagging without OOVs. However, to tackle the OOV problem, more domain knowledge needs to be put into feature engineering. Although the contextual information gives useful hints to deciding the POS tag of an OOV word, some slightly deeper knowledge of the problem (such as morphology) is often much more effective. For example, if we are working on English, then we know certain prefices and suffices are very useful in guessing the POS tag of an unknown word. One can either manually build a list of prefices and suffices or simply enumerate all possible sequence of letters up to a certain (and small) length. Another example is numbers and dates — treated naively, most dates will be regarded as OOVs; but a simple rule based system is able to recognize most of the unseen instances in these two classes and we can include such information as features as well.

5.2 Syntactic Chunking

The task of syntactic chunking, sometimes also called "shallow parsing", is to find all base constituents (chunks) of certain categories in a sentence. A base constituent of category XP is a phrase that does not contain any XP inside it. For example, the base NPs in the following sentences are all marked,

> [$_{\text{NP}}$He] reckons [$_{\text{NP}}$the current account deficit] will narrow to [$_{\text{NP}}$only # 1.8 billion] in [$_{\text{NP}}$September].

One does not need to go as far as syntactic parsing to find chunks. In fact, all we need for NP chunking is three labels, $\{\text{B}, \text{I}, \text{O}\}$. "B" means the current word begins a base NP; "I" means the current word is inside a base NP; and "O" means the current word is outside any based NP. The above example can then be labeled as,

> He/B reckons/O the/B current/I account/I deficit/I will/O narrow/O to/O only/B #/I 1.8/I billion/I in/O September/B ./O

Now chunking becomes similar to POS tagging, with the only superficial difference being the state set. Usually in addition to words, POS tags produced by an automatic tagger are also often used as features. One can do the same thing by looking at neighbors for tags as we did in POS tagging for words.

However, simply incorporating the additional POS information is not enough to obtain a reasonably high accuracy at chunking. The problem is our small state set. Note inside a base NP, most of the words are labeled as I, except the first word. As a result, after the first few words in the chunk, all words will have I as the previous state, and thus transition features no longer provide useful information. In other words, we are facing the problem of ambiguity, due to the lack of context.

One solution is to enlarge the context of the transition to trigrams of labels. This can be achieved by using a bigram of chunk labels $z = \langle p(z), n(z) \rangle$ as the state, where $p(z)$ is the chunk label of the previous word and $n(z)$ is the chunk label of the current word. States are now bigrams such as $\langle \text{B}, \text{I} \rangle$ or $\langle \text{I}, \text{O} \rangle$. We can still use our old transition feature $[z = a, z' = b]$, only now a and b are label bigrams. The following property always holds for legitimate transitions between the previous state z and the current state z',

$$n(z) = p(z') \tag{18}$$

This ensures the number of possible transitions is at most $|\Pi|^3$ instead of $|\Pi|^4$. Also if we still want to keep the number of word and POS features small, we can let them depend only on the current chunk label, which is now $n(z')$.

1: Initialize $\mathbf{w}$ as a zero vector
2: **for** iteration i less than T **do**
3: **for all** $\mathbf{X}^{(j)}, \mathbf{Z}^{(j)}$ in the training data **do**
4: $\mathbf{Z}^* = \arg\max_{\mathbf{Z}} \mathbf{w}^\intercal \Phi(\mathbf{X}^{(j)}, \mathbf{Z})$
5: **if** $\mathbf{Z}^* \neq \mathbf{Z}^{(j)}$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} + \Phi(\mathbf{X}^{(j)}, \mathbf{Z}^{(j)}) - \Phi(\mathbf{X}^{(j)}, \mathbf{Z}^*)$
7: **end if**
8: **end for**
9: **end for**

Figure 3: The Structured Perceptron Algorithm

5.3 Many More

In addition to POS tagging and chunking, many other problems in NLP can be seen as a sequence labeling problem. With proper feature engineering, the generalized model discussed here can often achieve state-of-the-art accuracy in such tasks.

Named entity recognition Finding names of people, organizations and places in the text. Much like chunking, we can use labels to mark spans of named entities. Spelling features such as capitalization are very useful in this task.

Acronym prediction Predicting acronyms of long words or phrases, such as DNA for "deoxyribonucleic acid", and TB for "tuberculosis". This can be done be labeling each character as either in (I) or not in (O) the acronym. Because of the extremely small label set, techniques for broadening the context is very important to improving accuracy. One such technique is to put the number of immediately preceding O labels as part of the O states.

6 Learning the Weights

Now we know if we define our feature template $\phi(z, z', \mathbf{X}, i)$, then we can label any new observation sequence, as long as we also have a good weight vector $\mathbf{w}$. But where do we get $\mathbf{w}$? If we have a lot of labeled data, for example sentences that are already POS tagged, then there are many ways of learning the weights.

6.1 Structured Perceptron

One way to learn the weights is to adopt the idea of the perceptron algorithm. The resulting *structured perceptron algorithm* is also fairly simple, as illustrated in Figure 3.

Almost identical to the perceptron algorithm in Figure 2, the core of the algorithm is the perceptron update in Line 6. The intuition behind is to push the weights closer to the correct prediction $\mathbf{Z}^{(j)}$ and further from the current best prediction $\mathbf{Z}^*$ whenever the current $\mathbf{w}$ leads to a mistake. To see how this is done, consider the k-th dimension of a feature vector,

- If $\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^{(j)})_k = \Phi(\mathbf{X}^{(j)}, \mathbf{Z}^*)_k$, then changing w_k has no effect on the best prediction and thus we do not update w_k.
- If $\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^{(j)})_k < \Phi(\mathbf{X}^{(j)}, \mathbf{Z}^*)_k$, then w_k will be decreased. $w_k\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^*)_k$ shrinks more than $w_k\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^{(j)})_k$ and thus $\mathbf{Z}^{(j)}$ is up-weighted while $\mathbf{Z}^*$ is down-weighted.

- If $\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^{(j)})_k > \Phi(\mathbf{X}^{(j)}, \mathbf{Z}^*)_k$, then w_k will be increased. $w_k\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^*)_k$ increases less than $w_k\Phi(\mathbf{X}^{(j)}, \mathbf{Z}^{(j)})_k$ and thus $\mathbf{Z}^{(j)}$ is up-weighted while $\mathbf{Z}^*$ is down-weighted.

In practice, $\mathbf{w}$ directly obtained from the vanilla perceptron algorithm generalizes not very well. One possible improvement is to keep a history of $\mathbf{w}$ at each data point in each iteration and average them to obtain the final weights. This technique, called *averaged perceptron*, often improves the accuracy of the learned model.

6.2 Conditional Random Field

Another possible way to learn the weights is to give a probabilistic interpretation to the score $\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z})$ so that when given a collection of labeled data $\{(\mathbf{X}^{(1)}, \mathbf{Z}^{(1)}), \ldots, (\mathbf{X}^{(N)}, \mathbf{Z}^{(N)})\}$, we can look for the weight vector $\hat{\mathbf{w}}$ that maximizes the conditional probability of obtaining the labeled data, namely,

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \prod_{i=1}^{N} P(\mathbf{Z}^{(i)} | \mathbf{X}^{(i)}; \mathbf{w}) \tag{19}$$

In Section 2, we transformed an HMM into a linear model by taking a log on $P(\mathbf{X}, \mathbf{Z})$. Given score $\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z})$ for some weight vector $\mathbf{w}$ and labeled data $(\mathbf{X}, \mathbf{Z})$, we can simply go in the reverse direction by putting an exponential back. However, there is no guarantee that $\exp(\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z}))$ is a probability. Fortunately, when $\mathbf{X}$ is known, the number of possible assignments to $\mathbf{Z}$ is finite, which means we only need to normalize to obtain $P(\mathbf{Z}|\mathbf{X})$. If we define,

$$P(\mathbf{Z}|\mathbf{X}; \mathbf{w}) \propto \exp(\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z})) \tag{20}$$

Then the normalizing factor is $\mathscr{Z}(\mathbf{X}; \mathbf{w}) = \sum_{\mathbf{Z}} \exp(\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z}))$, and therefore,

$$P(\mathbf{Z}|\mathbf{X}; \mathbf{w}) = \frac{\exp(\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z}))}{\mathscr{Z}(\mathbf{X}; \mathbf{w})} \tag{21}$$

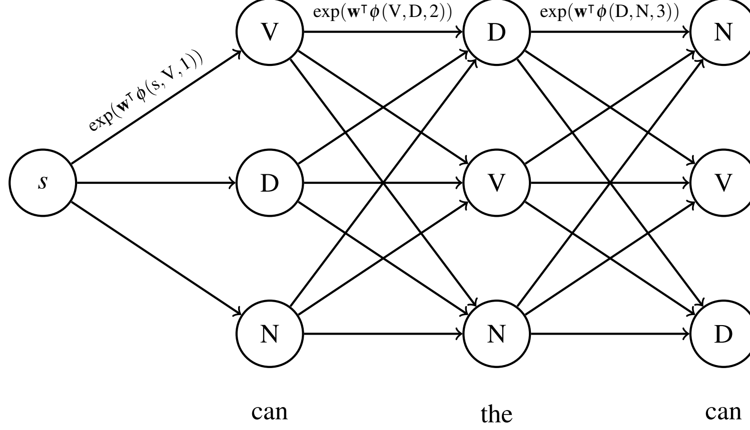
At a glance, one needs to sum over exponentially many possibilities in order to compute $\mathscr{Z}(\mathbf{X}; \mathbf{w})$. But things are not that bad if we take a closer look. If the linear model is an HMM, then we know $\mathscr{Z}(\mathbf{X}; \mathbf{w}) = P(\mathbf{X}; \mathbf{w})$. In an HMM, we compute $P(\mathbf{X}; \mathbf{w})$ by computing the forward or backward probabilities, using dynamic programming that is very similar to Viterbi. The same technique can be applied here as well. Note $\Phi(\mathbf{X}, \mathbf{Z}) = \sum_i \phi(z_{i-1}, z_i, \mathbf{X}, i)$, thus,

$$\mathscr{Z}(\mathbf{X}; \mathbf{w}) = \sum_{\mathbf{Z}} \exp\left(\mathbf{w}^\intercal\Phi(\mathbf{X}, \mathbf{Z})\right) \tag{22}$$

$$= \sum_{\mathbf{Z}} \exp\left(\sum_i \mathbf{w}^\intercal\phi(z_{i-1}, z_i, \mathbf{X}, i)\right) \tag{23}$$

$$= \sum_{\mathbf{Z}} \prod_i \exp\left(\mathbf{w}^\intercal\phi(z_{i-1}, z_i, \mathbf{X}, i)\right) \tag{24}$$

The term $\exp(\mathbf{w}^\intercal\phi(z_{i-1}, z_i, \mathbf{X}, i))$ here, can be seen as the contribution of the transition from z_{i-1} to z_i to the unnormalized probability mass. In fact in an HMM it is exactly the probability of following the particular edge. Since it only depends on one single transition, let's play our old trick of putting it onto the trellis in Section 3. For brevity, the observation sequence is omitted in the feature template,



Since we have spread all the pieces onto edges, we can compute the following forward and backward scores, the same way as in HMM. The forward score at position i and state z is the sum of exponential scores of all paths from the initial state,

$$\alpha_{\mathbf{X}}(i,z;\mathbf{w}) = \sum_{z_{1:i-1}} \left(\prod_{j=1}^{i-1} \exp(\mathbf{w}^\intercal \phi(z_{j-1}, z_j, \mathbf{X}, j)) \right) \exp(\mathbf{w}^\intercal \phi(z_{i-1}, z, \mathbf{X}, i)) \tag{25}$$

In an HMM, the above is exactly the forward probability. This can be efficiently computed using dynamic programming with the following equivalent recursive definition,

$$\alpha_{\mathbf{X}}(i,z;\mathbf{w}) = \begin{cases} 1 & i = 0 \\ \sum_{z'} \alpha_{\mathbf{X}}(i-1, z'; \mathbf{w}) \exp(\mathbf{w}^\intercal \phi(z', z, \mathbf{X}, i) & \text{otherwise} \end{cases} \tag{26}$$

Similarly, the backward score at position i and state z is the sum of exponential scores of all paths from $z_i = z$ to the end,

$$\beta_{\mathbf{X}}(i,z;\mathbf{w}) = \sum_{z_{i+1:n}} \exp(\mathbf{w}^\intercal \phi(z, z_{i+1}, \mathbf{X}, i+1)) \prod_{j=i+1}^{n-1} \exp(\mathbf{w}^\intercal \phi(z_j, z_{j+1}, \mathbf{X}, j+1)) \tag{27}$$

In an HMM, the above is exactly the backward probability. This can be efficiently computed using dynamic programming with the following equivalent recursive definition,

$$\beta_{\mathbf{X}}(i,z;\mathbf{w}) = \begin{cases} 1 & i = n \\ \sum_{z'} \exp(\mathbf{w}^\intercal \phi(z, z', \mathbf{X}, i+1)) \beta_{\mathbf{X}}(i+1, z'; \mathbf{w}) & \text{otherwise} \end{cases} \tag{28}$$

Then $\mathscr{Z}(\mathbf{X};\mathbf{w})$ is simply $\beta_{\mathbf{X}}(0,s)$ or $\sum_z \alpha_{\mathbf{X}}(n,z)$. Another interesting quantity that we will later need is the conditional probability of a certain transition from z to z' taking place between the $(j-1)$-th and j-th observation, namely,

$$P(z_{j-1} = z, z_j = z' | \mathbf{X}; \mathbf{w}) = \sum_{\mathbf{Z} \setminus \{z_{j-1}, z_j\}} P(\mathbf{Z}|\mathbf{X};\mathbf{w}) \tag{29}$$

Substitute $P(\mathbf{Z}|\mathbf{X};w)$ with our definition,

$$P(z_{j-1} = z, z_j = z' | \mathbf{X}; \mathbf{w}) = \sum_{\mathbf{Z} \setminus \{z_{j-1}, z_j\}} \frac{1}{\mathscr{Z}(\mathbf{X};\mathbf{w})} \exp(\mathbf{w}^\intercal \Phi(\mathbf{X}, \mathbf{Z})) \tag{30}$$

In spite of the scary looking, we are in fact just summing over all paths going through a certain transitions. Therefore by factoring out the terms in the left and right of $(\mathbf{w}^\intercal \phi(z_{j-1}, z_j, \mathbf{X}, j))$, we have the following simple form,

$$P(z_{j-1} = z, z_j = z' | \mathbf{X}; \mathbf{w}) = \frac{1}{\mathscr{Z}(\mathbf{X}; \mathbf{w})} \alpha_{\mathbf{X}}(j-1, z_{j-1}; \mathbf{w}) \exp\left(\mathbf{w}^\intercal \phi(z_{j-1}, z_j, \mathbf{X}, j)\right) \beta_{\mathbf{X}}(j, z_j; \mathbf{w}) \tag{31}$$

Now we are ready to return to the problem of finding $\hat{\mathbf{w}}$ that maximizes the conditional probability,

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \prod_i P(\mathbf{Z}^{(i)} | \mathbf{X}^{(i)}; \mathbf{w}) \tag{32}$$

$$= \arg\max_{\mathbf{w}} \log \left(\prod_i P(\mathbf{Z}^{(i)} | \mathbf{X}^{(i)}; \mathbf{w}) \right) \tag{33}$$

$$= \arg\max_{\mathbf{w}} \sum_i \log P(\mathbf{Z}^{(i)} | \mathbf{X}^{(i)}; \mathbf{w}) \tag{34}$$

$$= \arg\max_{\mathbf{w}} \sum_i \log \left(\frac{\exp(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z}^{(i)}))}{\mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})} \right) \tag{35}$$

$$= \arg\max_{\mathbf{w}} \sum_i \left(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z}^{(i)}) - \log \mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w}) \right) \tag{36}$$

Since the target function in the last equation is concave , one can use many techniques from convex optimization to find the optimal $\mathbf{w}$ in the last equation. Many of such algorithms require the gradient with respect to $\mathbf{w}$ of the target function. For the $\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z}^{(i)})$ term, the gradient is simply $\Phi(\mathbf{X}^{(i)}, \mathbf{Z}^{(i)})$. For the $\log \mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})$ term,

$$\nabla \log \mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w}) = \frac{1}{\mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})} \sum_{\mathbf{Z}} \exp(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z})) \Phi(\mathbf{X}^{(i)}, \mathbf{Z}) \tag{37}$$

$$= \frac{1}{\mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})} \sum_{\mathbf{Z}} \exp(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z})) \sum_j \phi(z_{j-1}, z_j, \mathbf{X}^{(i)}, j) \tag{38}$$

$$= \frac{1}{\mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})} \sum_j \sum_{\mathbf{Z}} \exp(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z})) \phi(z_{j-1}, z_j, \mathbf{X}^{(i)}, j) \tag{39}$$

$$= \frac{1}{\mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})} \sum_j \sum_{z_{j-1}, z_j} \sum_{\mathbf{Z} \setminus \{z_{j-1}, z_j\}} \exp(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z})) \phi(z_{j-1}, z_j, \mathbf{X}^{(i)}, j) \tag{40}$$

$$= \frac{1}{\mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w})} \sum_j \sum_{z_{j-1}, z_j} \left(\sum_{\mathbf{Z} \setminus \{z_{j-1}, z_j\}} \exp(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z})) \right) \phi(z_{j-1}, z_j, \mathbf{X}^{(i)}, j) \tag{41}$$

$$= \sum_j \sum_{z_{j-1}, z_j} P(z_{j-1}, z_j | \mathbf{X}^{(i)}; \mathbf{w}) \phi(z_{j-1}, z_j, \mathbf{X}^{(i)}, j) \tag{42}$$

Therefore, the optimal weight vector $\hat{\mathbf{w}}$ is the one that makes the expected feature vector over all possible paths equal to the feature vector from the training data.

With the gradient one can now apply algorithms such as L-BFGS or stochastic gradient descent to find $\hat{\mathbf{w}}$. In practice, CRF tends to overfit when a lot of features are available. To get models that generalize better, one usually needs to add regularization during training. The basic idea of regularization is to penalize large

weights so that the model does not become over-confident as a result of simply remembering the data. This can be achieved by adding a (scaled) vector norm to the target function. L_2 norm is most widely used for regularization, with which the optimization problem becomes,

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_i \Big(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z}^{(i)}) - \log \mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w}) \Big) - C\mathbf{w}^\intercal \mathbf{w} \tag{43}$$

C here is a hyperparameter that indicates the strength of regularization; larger C poses higher penalty. The target function can still be optimized using gradient-based methods because the added term is smooth.

Another widely used choice is L_1 norm, where the target function becomes

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_i \Big(\mathbf{w}^\intercal \Phi(\mathbf{X}^{(i)}, \mathbf{Z}^{(i)}) - \log \mathscr{Z}(\mathbf{X}^{(i)}; \mathbf{w}) \Big) - C \sum_j |w_j| \tag{44}$$

C here is a hyperparameter that indicates the strength of regularization; larger C poses higher penalty. Because the target function is no longer smooth, sub-gradient based methods has to be applied for optimization, which we will not discuss here.

In general, L_1 norm tends to push more weights to 0, and results in compact models, at the cost of accuracy. L_2 gives better accuracy but produces large models. Therefore, sometimes one does a first-pass L_1-regularized training as feature selection and then obtains the final weights using L_2-regularization.

6.3 Comparison and Summary

Conditional random field and structured perceptron can be seen as optimizing the same 0/1-loss function under different approximations. In practice, a regularized CRF and an averaged structured perceptron give similar accuracy, if the same set of features are used. However, this does not mean both training paradigms result in the same learned weights. There are also other paradigms for learning the weight vector, e.g. more recently researchers have proposed various models for adopting the large margin learning paradigm to learning such models.