

## Lecture 16: Applications Using Ordered Alignment Models

Many of the problems we have seen so far can be seen as the translation of some source stream of data into a target stream. For instance, in part of speech tagging we have a source stream of words and an output stream of POS tags. In the tagging models we have seen there has been a one to one correspondence between elements in the two streams (i.e., there is one tag for each word). Additional complications arise when this restriction is relaxed and many elements in the source stream might correspond to one element in the target stream (e.g., a many acoustic events in a speech stream correspond to one word), or the corresponding elements may appear in different orders in the two streams (e.g., *the red house* corresponds to *la maison rouge* in French, but the second word in English, *red*, corresponds to the third word, *rouge*, in the French sentence. In this lecture we will look at techniques for aligning two streams of data, i.e., identifying the correspondences. We will assume, however, that the alignments preserve order, and discussion of more general alignment models until we discuss Machine Translation. This will give us another opportunity to look at the EM algorithm for learning probabilistic models from data when the alignment is not known.

### 1. Min-Edit Distance Models

One particular class of problems that use alignments involve comparing two streams and determining how “different” they are. Typically, the distance measure is formalized in terms of the cost of the editing operations required to transform one stream into the other. For example, in a spelling correction application say we are given the input SPECH and the dictionary includes the words SPEECH, SPEC, SPECK and SPECS. Which of these is closest? If we look at each of these cases, we see different editing operations that we would need.

Insertion: the difference between SPECH and SPEECH is that we have to insert a letter (namely E).

Deletion: The difference between SPECH and SPEC is we have to delete a letter (namely H).

Substitution: The difference between SPECH and SPECK is that we have to substitute one letter (H) for another (K).

We could, of course, model a substitution operation as one delete and an insert, but by keeping it as a separate operation allows to assign a cost to the operation independently of insertions and deletions. We might, for instance, look at a corpus of spelling errors and find that insertion errors occur most frequently, followed by substitutions, and finally deletions. We could set the costs of these operations so that we would prefer the common types of errors over the rarely ones when finding the word with the minimum edit-distance cost. Note that for spelling, we might have other edit operations as well, such as transpositions (e.g., typing RG rather than GR), as we may want to model this as an independent operation rather than as two substitution errors.

Note that we can represent deletion, insertion and substitution errors in terms of an alignment between the two words that show how the letters correspond, using a special symbol e where there is no corresponding letter. For example, consider the edit distance between SPEECH and SPECKS as shown here:

```

S P E E C H □
| | | | | | |
S P E □ C K S

```

In the fourth position we have a deletion operation, in the sixth a substitution operation and in the seventh an insertion operation. If all these operations have the same cost, say 1, then the distance between these words would be 3. But note that there are other possible alignments between these words, not only

```

S P E E C H □
| | | | | | |
S P □ E C K S

```

which has the same distance, but also strange alignments such as

```

S P □ E E C H □
| | | | | | |
S □ P □ E C K S

```

which has a cost of 5! So, when we talk of the difference between two streams we always want to know the minimum edit distance. This means that we need an algorithm to find the minimum edit difference. It might seem that this is a simple algorithm, but it can be complex. A decision that looks good early on might lead to problems later on. For instance consider finding the minimum cost alignment between SPEECH and SPSPEECH. Working left to right, it might seem we can align the two initial S's, followed by the two P's, but this leads us to the alignments like

```

S P E E C H □ □
| | | | | | |
S P S P E E C H

```

which contains 6 errors. When we have two letters that differ, how do we know if it would be best to treat it as a substitution, insertion or deletion error?

This turns out to be another problem for which a dynamic programming algorithm (like the shortest-distance graph search, and the Viterbi and Forward algorithms) is useful. As in the Viterbi search of an HMM, this problem has the property that the shortest path to a state S is always part of any solution of which state S is part. In fact, we can see this quite directly by formalizing the algorithm as a search through a graph. The states of the search are a pair of indices (i, j), which say we are currently at the i'th position in the source stream, and j'th position in the target stream. For example, the state (1,3) with the SPEECH and SPSPEECH example above would mean we are at the S in SPEECH and the second S in SPSPEECH. There are three

Let  $\text{MinCost}(i,j)$  be the minimum cost to state  $(i,j)$ . Then we can compute the costs for all the states as follows:

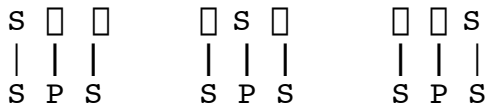
For  $i = 1, N$

For  $j = 1, M$

$$\text{MinCost}(i, j) = \text{Minimum}\{ \text{Insertcost} + \text{MinCost}(i-1, j); \\ \text{deleteCost} + \text{MinCost}(i, j-1); \\ \text{substituteCost}(i,j) + \text{MinCost}(i-1, j-1)\}$$

Figure 1: The Min Edit Distance Algorithm

ways we could have got to this state, captured by the following alignments, where errors are marked by a thicker alignment marker:



If we assign a cost of 1 to all errors, then the first possibility costs 2, the second 3 and the third 2. So the minimum cost to state  $(1,3)$  is 2, and we could have got there using the first or third alignment. Which one we used to get to  $(1,3)$  does not affect anything that can happen after, so we can just remember one of these. The Min-Edit Distance Algorithm is show in Figure 1. Note that the substitution cost would be 0 if the  $i$ 'th letter in the source equals the  $j$ 'th letter in the target, and 1 (or whatever we set) if they are different. In addition, when find the minimum length at each stage, we must remember which edit operation was used to produce extend the search to the current state. This maintains a record of the search the same way that the Viterbi algorithm saves a back pointer to the previous state on the the maximum probability path. Essentially, this algorithm is simply finding the minimum cost path through the graph shown as Figure 2.

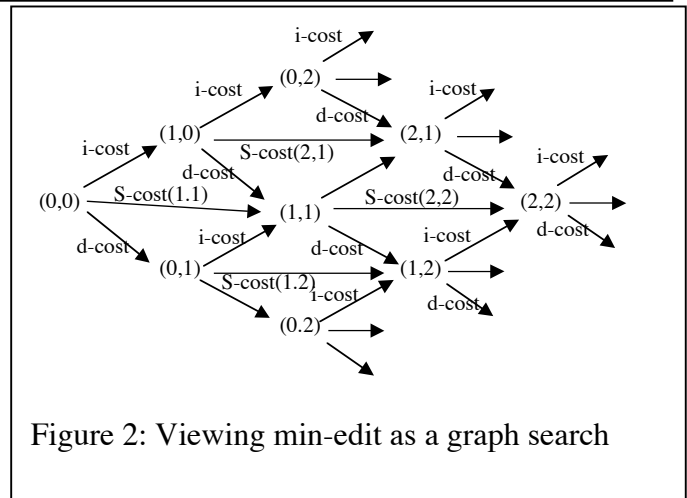


Figure 2: Viewing min-edit as a graph search

We could of course generalize this model further. The substitution cost could depend on how often the two letters are confused. The cost for an insertion or deletion might depend on the symbol being deleted (say some letters are more commonly omitted than others), In any event, we a fast method to computing edit distance, we can easily build a spelling correction algorithm. When faced with an unknown word, we simply substitute the known word that is closest to it.

## 2. An Application of Min-Edit Distance: Accuracy Scores for Speech Recognition

Besides spelling correction, min-edit algorithms are useful for other tasks. Here we look at one more example, determining the accuracy of speech recognition systems. This maps to a min-edit

distance problem easily: the source stream is the output of the speech recognizer, and the target is an accurate transcription of the speech. Thus, the SR may have recognized

HE WORE TIES IN THE OLD STORE

whereas what was actually said was

HE SAW THE PIE IN THE STORE

We can measure the difference between these two streams using the edit distance idea, finding the minimum cost sequence of insertions, deletions and substitutions to make the strings identical. Assuming an equal cost of 1 for each error type, a minimum cost alignment is

HE	WORE	□	TIES	IN	THE	OLD	STORE
HE	SAW	THE	PIE	IN	THE	□	STORE

There is one insertion error, one deletion and one substitution error in this minimum cost alignment. The usual way to combine these numbers into a single score is called the **word error rate** (WER) and it is defined as follows:

$$\text{WER} = 100 * (\#\text{insertions} + \#\text{deletions} + \#\text{substitution errors}) / \text{Length of transcript}$$

So the word error rate on the above example is  $300/7 = 42\%$ . Note that while results are usually reported in terms of percent, this is not technically accurate for WER can exceed 100! For example, if the sentence was really HELLO, and the SR recognized HE WROTE, the WER would be 200 “percent”.

### ***3. Simple Translation Problems Using Alignment: Grapheme to Phoneme Conversion***

Alignment models are also useful in translation problems where there is not a one-to-one correspondence between the words in the **source** language and words in the **target** language. For example, say we are trying to build a program to derive phonetic pronunciations from English word. The word KEY maps to the phoneme sequence /k/ /i/, and SEEK maps to /s/ /i/ /k/. We like to develop a probabilistic model that provides us with the most likely translation, i.e., how often does the letter E map to /i/ and how often to some other phoneme like /e/ as in GET. In other words, we are trying to solve a problem of the form

$$\text{argmax}_T P(T_{1,N} | S_{1,M})$$

where  $T_{1,N}$  is any sequence in the target language of length N and  $S_{1,M}$  is the sequence in the source language that we want to translate. As we have done many times before, we use Bayes rule to rewrite it and then delete the denominator to produce

$$\operatorname{argmax}_T P(T_{1,N}) * P(S_{1,M} | T_{1,N})$$

$P(T_{1,N})$  is the language model (for pronunciations), and  $P(S_{1,M} | T_{1,N})$  is called the translation model. To apply this model, we need to estimate the two probability distributions from a pronunciation dictionary that lists words and their pronunciations. Estimating  $P(T_{1,N})$  is straightforward - we could use a n-gram model trained off all the pronunciations in the corpus. So the problem remains to estimate

$$P(S_{1,M} | T_{1,N})$$

We have to develop some method of handling the sequences of different length, and determining the alignment so we know what letters correspond to what phonemes. We will do this by introducing the notion of an **alignment function**  $a_i$ . This function maps a position in the source sequence into a position in the target, i.e., if  $a_i(k) = l$ , then this means the k'th letter of the word corresponds to the l'th letter of the sequence. We use a special value  $\square$  that is used to signify letters that don't correspond to any phoneme. For the pronunciation task, we have a number of constraints on what the alignment function can be which greatly reduce the number of possibilities we have to consider.

1. Ordering constraint: For if  $j > k$ , and neither  $a_i(j)$  nor  $a_i(k)$  equal  $\square$  then  $a_i(j) > a_i(k)$  - i.e., the correspondences move through the sequences in a left to right manner;
2. Coverage constraint: For all  $k$  from 1 to  $N$ , there is an  $j$  such that  $a_i(j)=k$  (i.e., every phoneme is generated from some letter).

Note that these are strong constraints and imply that only single letters map to phonemes, and all the extra letters are ignored. This is not very accurate as we all know that sequences of letters often correspond to a single phoneme. For instance, rather than saying that EE can map to /i/ in the word seek, we will have to pick one E to map to /i/ and map the other to  $\square$ . This means we will have difficulty modeling some examples well, but such models are often used because of their simplicity.

We now can try estimating  $P(S_{1,M} | T_{1,N})$  using the unigram approximation  $\prod_j P(S_j | T_{a_i(j)})$ . For example, given an alignment function that maps 1 to 1, 2 to 2 and 3 to  $\square$  the probability of  $P(K E Y | /k/ /i/)$  would be

$$P(K | /k/) * P(E | /i/) * P(Y | \square)$$

But this is only one possible alignment. A better estimate, given we don't know what the right alignment is, would be the sum over all possible alignments. In other words, we estimate  $P(S_{1,M} | T_{1,N})$  with the formula

$$\sum_i \prod_j P(S_j | T_{a_i(j)})$$

where  $A=\{a_i\}$  is the set of all possible alignment functions. Thus, our better estimate of  $P(K E Y | /k/ /i/)$  sums over the three possible alignments

$$\begin{aligned}
& P(K | /k/) * P(E | /i/) * P(Y | \square) + \\
& P(K | /k/) * P(E | \square) * P(Y | /i/) + \\
& P(K | \square) * P(E | /k/) * P(Y | /i/)
\end{aligned}$$

Note the parallel with the forward probability for HMMs. Just as the forward probability is the probability of obtaining the output and ending in a state (computed by summing over all possible paths to that state), this estimates the probability of the phoneme sequence for the word by summing over all possible alignments of the phoneme sequence to the word. In fact, we can design an min-edit analogue to the forward (or Viterbi) algorithms that incrementally compute the probability to alignment states (i, j), where i indicates we are at position i in the source (i.e.,  $S_i$ ) and j indicates we are at position j in the target (i.e.,  $T_j$ ). We can consider the possible transitions between states: We can get to state (i, j) only from states (i-1, j-1) (with “transition” probability  $P(S_i | T_j)$ ) or (i-1, j-1) (with “transition” probability  $P(S_i | \square)$ ). For the Viterbi analogue, we take the max probability sequence so far, and for the forward analogue, we sum the probabilities of the paths.

#### 4. Training Alignment Based Models

We have a problem similar to when we wanted to train an HMM from a corpus. We could find the maximum probability phoneme sequence if we could estimate the distribution  $P(\text{letter} | \text{phoneme})$ , but we can't learn this distribution unless we know the alignment function used. The solution to this problem is the same as before - we use the EM algorithm where the hidden data is the alignment function. It is instructive to observe we could try to rephrase this problem with an HMM as shown in Figure 3. Each phoneme is represented as a state, as well as  $\square$ . The output probabilities would be the probabilities of the letters for each phoneme  $P(L | \text{ph})$ . The problem with this model is with the transitions. For instance, given (SEEK, /s//i//k/), there are only two legal paths through the network, also shown in Figure 3. Because we aren't considering all paths,

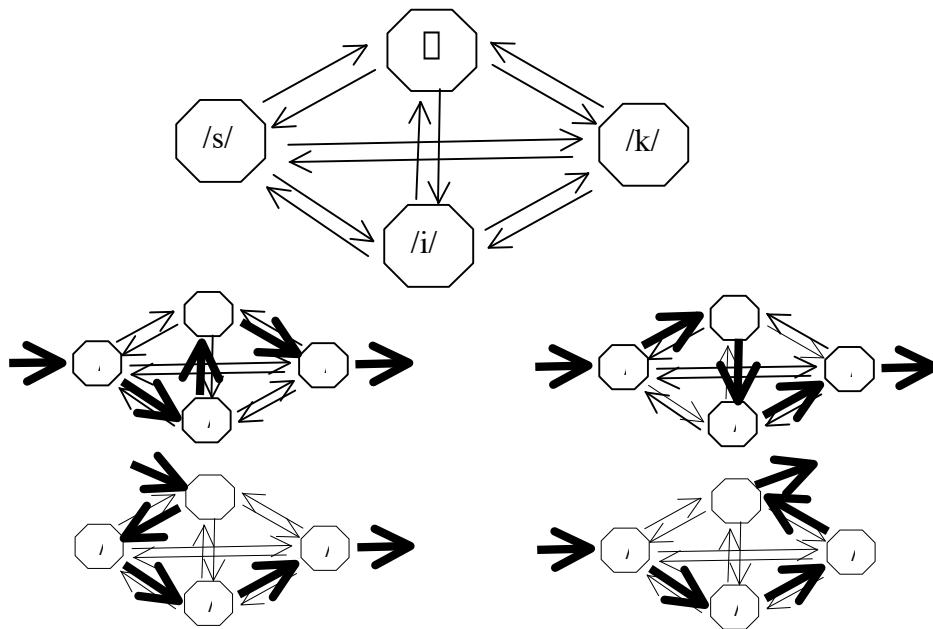


Figure 3: An HMM Approximation of the Alignment, and the possible paths allowed

we can't directly use the Baum-Walch method. Rather, we will resort back to applying the EM brute force.

Remember that the EM algorithm starts with some guess at a distribution and then improves it by iterating over two steps:

1. Use the current distribution to estimate the probabilities of each possible alignment
2. Use a count weighted by the alignment probabilities to re-estimate the distribution.

Let's consider an example. Say we want to estimate the distribution given a two word training corpus: KEY  $\rightarrow$  /k/ /i/ and SEEK  $\rightarrow$  /s/ /i/ /k/. Let us initially assume a uniform distribution for  $P(\text{Letter} \mid \text{Phoneme})$  - i.e., each letter has a probability of  $1/26 (= .038)$  given any phoneme. There are three possible alignments for KEY (each with a prob of  $5.6 * 10^{-5}$  of generating KEY) and four for SEEK (each with a prob. of  $2.1 * 10^{-6}$  of generating SEEK). We can now count all the correspondences present over the alignments and weight them by these probabilities. For example, the K - /k/ correspondence occurs twice in the alignments for KEY, each with a weight of  $5.6e-5$ , and 3 times in alignments for SEEK, each with a weight of  $2.1e-6$ , giving us a total of  $1.2e-4$ . All the counts are shown in Figure 4.

	Weight	S /s/	S $\square$	E /i/	E $\square$	E /k/	E /s/	K /k/	K $\square$	Y /i/	Y $\square$
SEEK, /s/ /i/ $\square$ /k/	$2.1e-6$	1		1	1			1			
SEEK, /s/ /i/ /k/ $\square$	$2.1e-6$	1		1		1		1			
SEEK, /s/ $\square$ /i/ /k/	$2.1e-6$	1		1	1			1			
SEEK, $\square$ /s/ /i/ /k/	$2.1e-6$		1	1			1		1		
KEY, /k/ /i/ $\square$	$5.6e-5$			1				1			1
KEY, /k/ $\square$ /i/	$5.6e-5$				1			1		1	
KEY, $\square$ /k/ /i/	$5.6e-5$					1			1	1	
Weighted Totals		$6.3e-6$	$2.1e-6$	$6.4e-5$	$6.0e-5$	$5.8e-5$	$2.1e-6$	$1.2e-4$	$5.8e-5$	$1.1e-4$	$5.6e-5$

Figure 4: Computing the weighted counts given the uniform distribution

From there, we can re-estimate the probabilities. For  $P(K \mid /k/)$  would be  $1.2e-4$  divided by the sum of all probabilities of form  $P(x \mid /k/)$ , which is  $1.2e-4 + 5.8e-5 = 1.6e-4$ , giving us a new estimate of .74. The complete set of numbers is shown in Figure 5 (as Iteration 1). As we continue to iterate, notice that the correspondence between K and /k/ continues to grow, as does the correspondence between E and /i/ and between S and /s/. Note also that we are training the probabilities for letters to be ignored, i.e.,  $P(\text{letter} \mid \square)$ , after 3 iterations  $P(K \mid \square) = .32$ ,  $P(E \mid \square) = .34$  and  $P(Y \mid \square) = .32$  and  $P(S \mid \square) = .02$ .

	Iteration 0	Iteration 1	Iteration 2	Iteration 3
$P(K \mid /k/)$	.038	.67	.78	.94
$P(E \mid /k/)$	.038	.33	.22	.06
$P(E \mid /i/)$	.038	.37	.52	.70
$P(S \mid /s/)$	.038	.75	.996	.999

Figure 4: Selected probabilities for the first three iterations

## 5. More General Models of Alignment

A more general approach to this problem, intuitively better suited to the grapheme to phone conversion problem, is to view alignment as associating subsequences together. The approach above is a specific instance of this in which all subsequences must be of length one. The advantage is that we don't need to use the  $\square$  value to ignore letters. For instance, our alignment of KEY would be (K, /k/) (EY, /i/), i.e., the letter sequence EY corresponds to the phoneme /i/. Similarly, SEEK would have the intuitively correct alignment of (S, /s/) (EE, /i/) (K /k/), associating the sequence EE with the phoneme /i/. Of course, when we don't know the alignment to start with, we would have to consider all possible alignments: i.e., for SEEK we'd also have the alignments (SE, /s/) (E, /i/) (K /k/) and (S, /s/) (E, /i/) (EK /k/). The alignment function in this case would identify a sequence of such pairs <l-seq, p-seq>, then we would estimate the probability of a letter sequence given a phoneme sequence as

$$P(L_{1,N} | Ph_{1,M}) = \sum_i \prod_j P(a_i^l(j) | a_i^{ph}(j))$$

where  $i$  ranges over all possible alignment functions and  $a_i^l(j)$  is the letter sequence of the  $j$ 'th element of the  $i$ 'th alignment function, and  $a_i^{ph}(j)$  is the phoneme sequence of the same element. Grapheme to Phoneme models based on this more general model outperform the simpler alignment functions described above.

We will see yet more complicated alignment functions when we consider Machine Translation.