Foundations of Natural Language Processing
Lecture 6
Spelling correction, edit distance, and EM

Alex Lascarides
(Slides from Alex Lascarides and Sharon Goldwater)

31 January 2020

Recap: noisy channel model

A general probabilistic framework, which helps us estimate something hidden (e.g., for spelling correction, the intended word) via two distributions:

- $P(Y)$: Language model. The distribution over the words the user intended to type.
- $P(X|Y)$: Noise model. The distribution describing what user is **likely** to type, given what they **meant** to type.

Given some particular word(s) x (say, no much effert), we want to recover the most probable y that was intended.

Recap: noisy channel model

- Mathematically, what we want is

$$\operatorname*{argmax}_y P(y|x) \quad = \quad \operatorname*{argmax}_y P(x|y)P(y)$$

- Assume we have a way to compute $P(x|y)$ and $P(y)$.
  Can we do the following?
  - Consider all possible intended words y.
  - For each y, compute $P(x|y)P(y)$.
  - Return the y with highest $P(x|y)P(y)$ value.

Recap: noisy channel model

- Mathematically, what we want is

$$\operatorname*{argmax}_y P(y|x) \quad = \quad \operatorname*{argmax}_y P(x|y)P(y)$$

- Assume we have a way to compute $P(x|y)$ and $P(y)$.
  Can we do the following?
  - Consider all possible intended words y.
  - For each y, compute $P(x|y)P(y)$.
  - Return the y with highest $P(x|y)P(y)$ value.
- No! Without constraints, there are an infinite # of possible ys.

Algorithm sketch

- A very basic spelling correction system. Assume:
  - we have a large dictionary of real words;
  - we only correct non-word → word; and
  - we only consider corrections that differ by a single character (insertion, deletion, or substitution) from the non-word.
- Then we can do the following to correct each non-word x:
  - Generate a list of all words y that differ by 1 character from x.
  - Compute $P(x|y)P(y)$ for each y and return the y with highest value.

A simple noise model

- Suppose we have a corpus of **alignments** between actual and corrected spellings.

actual:	n	o	-		m	u	u	c	h		e	f	f	e	r	t
	|	|	|	|	|	|	|	|	|	|	|	|	|	|	|	|
intended:	n	o	t		m	-	u	c	h		e	f	f	o	r	t

- This example has
  - one **substitution** (o → e)
  - one **deletion** (t → -, where - is used to show the alignment, but nothing appears in the text)
  - one **insertion** (- → u)

A simple noise model

- Assume that typed character x_i depends only on intended character y_i (ignoring context).
- So, substitution o → e is equally probable regardless of whether the word is effort, spoon, or whatever.
- Then we have

$$P(x|y) = \prod_{i=1}^{n} P(x_i|y_i)$$

For example, $P(\text{no}|\text{not}) = P(\text{n}|\text{n})P(\text{o}|\text{o})P(\text{-}|\text{t})$

See Brill and Moore (2000) on course page for an example of a better model.

Estimating the probabilities

- Using our corpus of alignments, we can easily estimate $P(x_i|y_i)$ for each character pair.
- Simply count how many times each character (including empty character for del/ins) was used in place of each other character.
- The table of these counts is called a **confusion matrix**.
- Then use MLE or smoothing to estimate probabilities.

Example confusion matrix

$y\backslash x$	A	B	C	D	E	F	G	H	...
A	168	1	0	2	5	5	1	3	...
B	0	136	1	0	3	2	0	4	...
C	1	6	111	5	11	6	36	5	...
D	1	17	4	157	6	11	0	5	...
E	2	10	0	1	98	27	1	5	...
F	1	0	0	1	9	73	0	6	...
G	1	3	32	1	5	3	127	3	...
H	2	0	0	0	3	3	0	4	
...	...	...	...	...	...	...	...	...	...

- We saw G when the intended character was C 36 times.

Big picture again

- We now have a very simple spelling correction system, provided
  - we have a corpus of aligned examples, and
  - we can easily determine which real words are only one edit away from non-words.
- There are easy, fairly efficient, ways to do the latter (see http://norvig.com/spell-correct.html).
- But where do the alignments come from, and what if we want a more general algorithm that can compute edit distances between any two arbitrary words?

Alignments and edit distance

These two problems reduce to one: find the **optimal character alignment** between two words (the one with the fewest character changes: the **minimum edit distance** or MED).

- Example: if all changes count equally, MED(stall, table) is 3:

S	T	A	L	L		
	T	A	L	L		deletion
	T	A	B	L		substitution
	T	A	B	L	E	insertion

Alignments and edit distance

These two problems reduce to one: find the **optimal character alignment** between two words (the one with the fewest character changes: the **minimum edit distance** or MED).

- Example: if all changes count equally, MED(stall, table) is 3:

S	T	A	L	L		
	T	A	L	L		deletion
	T	A	B	L		substitution
	T	A	B	L	E	insertion

- Written as an alignment:

S	T	A	L	L	-
d	|	|	s	|	i
-	T	A	B	L	E

More alignments

- There may be multiple best alignments. In this case, two:

S	T	A	L	L	-		S	T	A	-	L	L
d	|	|	s	|	i		d	|	|	i	|	s
-	T	A	B	L	E		-	T	A	B	L	E

- And **lots** of non-optimal alignments, such as:

S	T	A	-	L	-	L		S	T	A	L	-	L	-
s	d	|	i	|	i	d		d	d	s	s	i	|	i
T	-	A	B	L	E	-		-	-	T	A	B	L	E

How to find an optimal alignment

Brute force: Consider all possibilities, score each one, pick best.

How many possibilities must we consider?

- First character could align to any of:

  - - - - - T A B L E -

- Next character can align anywhere to its right
- And so on... the number of alignments grows exponentially with the length of the sequences.

Maybe not such a good method...

A better idea

Instead we will use a **dynamic programming** algorithm.

- Other DP (or **memoization**) algorithms: Viterbi, CKY.
- Used to solve problems where brute force ends up **recomputing** the same information many times.
- Instead, we
  - Compute the solution to each subproblem **once**,
  - Store (memoize) the solution, and
  - Build up solutions to larger computations by combining the pre-computed parts.
- Strings of length n and m require $O(mn)$ time and $O(mn)$ space.

Intuition

- Minimum distance D(stall, table) must be the minimum of:
  - D(stall, tabl) + cost(ins)
  - D(stal, table) + cost(del)
  - D(stal, tabl) + cost(sub)
- Similarly for the smaller subproblems
- So proceed as follows:
  - solve smallest subproblems first
  - store solutions in a table (chart)
  - use these to solve and store larger subproblems until we get the full solution

A note about costs

- Our first example had cost(ins) = cost(del) = cost(sub) = 1.
- But we can choose whatever costs we want. They can even depend on the particular characters involved.
  - For example: choose cost(sub(c,c')) to be $P(c'|c)$ from our spelling correction noise model!
  - Then we end up computing the most probable way to change one word to the other.
- In the following example, we'll assume cost(ins) = cost(del)= 1 and cost(sub) = 2.

Chart: starting point

		T	A	B	L	E
	0					
S						
T						
A						
L						
L						?

- Chart[i, j] stores two things:
  - D(stall[$0..i$], table[$0..j$]): the MED of substrings of length i, j
  - **Backpointer(s):** which sub-alignment(s) used to create this one.

Deletion:	Move down	Cost =1
Insertion:	Move right	Cost=1
Substitution:	Move down and right	Cost=2 (or 0 if the same)

Sum costs as we expand out from cell (0,0) to populate the entire matrix

Filling first cell

		T	A	B	L	E
	0					
S	↑1					
T						
A						
L						
L						

- Moving down in chart: means we had a **deletion** (of S).
- That is, we've aligned (S) with (-).
- Add cost of deletion (1) and backpointer.

Rest of first column

		T	A	B	L	E
	0					
S	↑1					
T	↑2					
A						
L						
L						

- Each move down first column means another deletion.
  - D(ST, -) = D(S, -) + cost(del)

Rest of first column

		T	A	B	L	E
	0					
S	↑1					
T	↑2					
A	↑3					
L	↑4					
L	↑5					

- Each move down first column means another deletion.
  - D(ST, -) = D(S, -) + cost(del)
  - D(STA, -) = D(ST, -) + cost(del)
  - etc

Start of second column: insertion

		T	A	B	L	E
	0	←1				
S	↑1					
T	↑2					
A	↑3					
L	↑4					
L	↑5					

- Moving right in chart (from [0,0]): means we had an **insertion**.
- That is, we've aligned (-) with (T).
- Add cost of insertion (1) and backpointer.

Substitution

		T	A	B	L	E
	0	←1				
S	↑1	↖2				
T	↑2					
A	↑3					
L	↑4					
L	↑5					

- Moving down and right: either a **substitution** or **identity**.
- Here, a substitution: we aligned (S) to (T), so cost is 2.
- For identity (align letter to itself), cost is 0.

Multiple paths

		T	A	B	L	E
	0	←1				
S	↑1	↖↑2				
T	↑2					
A	↑3					
L	↑4					
L	↑5					

- However, we also need to consider other ways to get to this cell:
  - Move **down** from [0,1]: deletion of S, total cost is D(-, T) + cost(del) = 2.
  - Same cost, but add a new backpointer.

Multiple paths

		T	A	B	L	E
	0	←1				
S	↑1	←↖↑2				
T	↑2					
A	↑3					
L	↑4					
L	↑5					

- However, we also need to consider other ways to get to this cell:
  - Move **right** from [1,0]: insertion of T, total cost is D(S, -) + cost(ins) = 2.
  - Same cost, but add a new backpointer.

Single best path

		T	A	B	L	E
	0	←1				
S	↑1	←↖↑2				
T	↑2	↖1				
A	↑3					
L	↑4					
L	↑5					

- Now compute D(ST, T). Take the min of three possibilities:
  - D(ST, -) + cost(ins) = 2 + 1 = 3.
  - D(S, T) + cost(del) = 2 + 1 = 3.
  - D(S, -) + cost(ident) = 1 + 0 = 1.

Final completed chart

		T	A	B	L	E
	0	←1	←2	←3	←4	←5
S	↑1	←↖↑2	←↖↑3	←4	←5	←6
T	↑2	↖1	←2	←3	←4	←5
A	↑3	↑2	↖1	←2	←3	←4
L	↑4	↑3	↑2	←↖↑3	↖2	←3
L	↑5	↑4	↑3	←↖↑4	↖↑3	←↖↑4

- Exercises for you:
  - How many different optimal alignments are there?
  - Reconstruct all the optimal alignments.
  - Redo the chart with all costs = 1 (Levenshtein distance)

Alignment and MED: uses?

Computing distances and/or alignments between arbitrary strings can be used for

- Spelling correction (as here)
- Morphological analysis: which words are likely to be related?
- Other fields entirely: e.g., comparing DNA sequences in biology.
- Related algorithms are also used in speech recognition and timeseries data mining.

Getting rid of hand alignments

Using MED algorithm, we can now produce the character alignments we need to estimate our error model, given only corrected words.

- Previously, we needed hand annotations like:

actual:	n	o	-		m	u	u	c	h		e	f	f	e	r	t
	|	|	|	|	|	|	|	|	|	|	|	|	|	|	|	|
intended:	n	o	t		m	-	u	c	h		e	f	f	o	r	t

- Now, our annotation requires less effort:

actual:	no	muuch	effert
intended:	not	much	effort

Catch-22

- But wait! In my example, we used costs of 1 and 2 to compute alignments.
- We actually want to compute our alignments using the costs from our noise model: the most probable alignment under that model.
- But until we have the alignments, we can't estimate the noise model...

General formulation

This sort of problem actually happens a lot in NLP (and ML):

- We have some probabilistic model and want to estimate its **parameters** (here, the character rewrite probabilities: prob of each typed character given each intended character).
- The model also contains variables whose value is unknown (here: the correct character alignments).
- We would be able to estimate the parameters if we knew the values of the variables...
- ...and conversely, we would be able to infer the values of the variables if we knew the values of the parameters.

EM to the rescue

Problems of this type can often be solved using a version of **Expectation-Maximization** (EM), a general algorithm schema:

1. Initialize parameters to arbitrary values (e.g., set all costs = 1).
2. Using these parameters, compute optimal values for variables (run MED to get alignments).
3. Now, using those alignments, **recompute** the parameters (just pretend the alignments are hand annotations; estimate parameters as from annotated corpus).
4. Repeat steps 2 and 3 until parameters stop changing.

EM vs. hard EM

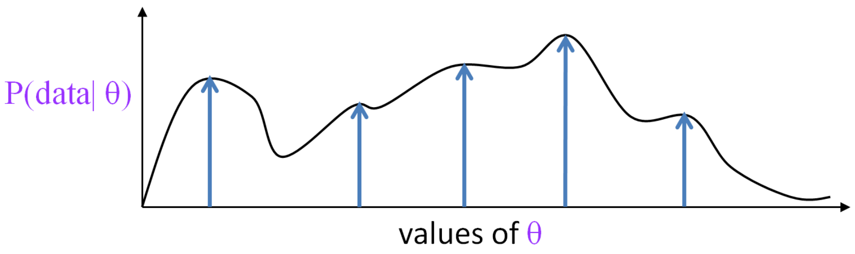
- The algorithm on the previous slide is actually "hard EM" (meaning: no soft/fuzzy decisions)
- Step 2 of true EM does not choose **optimal** values for variables, instead computes **expected** values (we'll see this for HMMs).
- True EM is guaranteed to converge to a local optimum of the **likelihood function**.
- Hard EM also converges but not to anything nicely defined mathematically. However it's usually easier to compute and may work fine in practice.

Likelihood function

- Let's call the parameters of our model θ.
  - So for our spelling error model, θ is the set of all character rewrite probabilities $P(x_i|y_i)$.
- For any value of θ, we can compute the probability of our dataset $P(\text{data}|\theta)$. This is the **likelihood**.
  - If our data includes hand-annotated character alignments, then $P(\text{data}|\theta) = \prod_{i=1}^{n} P(x_i|y_i)$
  - If the alignments a are latent, sum over possible alignments: $P(\text{data}|\theta) = \sum_a \prod_{i=1}^{n} P(x_i|y_i, a)$

Likelihood function

- The likelihood $P(\text{data}|\theta)$ is a function of θ, and can have multiple local optima. Schematically (but θ is really multidimensional):



- EM will converge to one of these; hard EM won't necessarily.
- Neither is guarateed to find the global optimum!

Summary

Our simple spelling corrector illustrated several important concepts:

- Example of a noise model in a noisy channel model.
- Difference between model definition and algorithm to perform inference.
- Confusion matrix: used here to estimate parameters of noise model, but can also be used as a form of error analysis.
- Minimum edit distance algorithm as an example of dynamic programming.
- (Hard) EM as a way to "bootstrap" better parameter values when we don't have complete annotation.