The Basics of Edit Distance

COSI 114 – Computational Linguistics

James Pustejovsky

January 20, 2015

Brandeis University

Spelling Correction

- We can detect spelling errors (spell check) by building an FST-based lexicon and noting any strings that are rejected.
- But how do I fix "graffe"? That is, how do I come up with suggested corrections?
  - Search through all words in my lexicon
    - Graft, craft, grail, giraffe, crafted, etc.
  - Pick the one that's closest to graffe
  - But what does "closest" mean?
    - We need a distance metric.
    - The simplest one: minimum edit distance
      - As in the Unix diff command

Dynamic programming

- Algorithm design technique often used for optimization problems
- Generally usable for recursive approaches if the same partial solutions are required more than once
- Approach: store partial results in a table
- Advantage: improvement of complexity, often polynomial instead of exponential

Two different approaches

Bottom-up:

+ controlled efficient table management, saves time
+ special optimized order of computation, saves space
- requires extensive recoding of the original program
- possible computation of unnecessary values

Top-down: (Note-pad method)

+ original program changed only marginally or not at all
+ computes only those values that are actually required
- separate table management takes additional time
- table size often not optimal

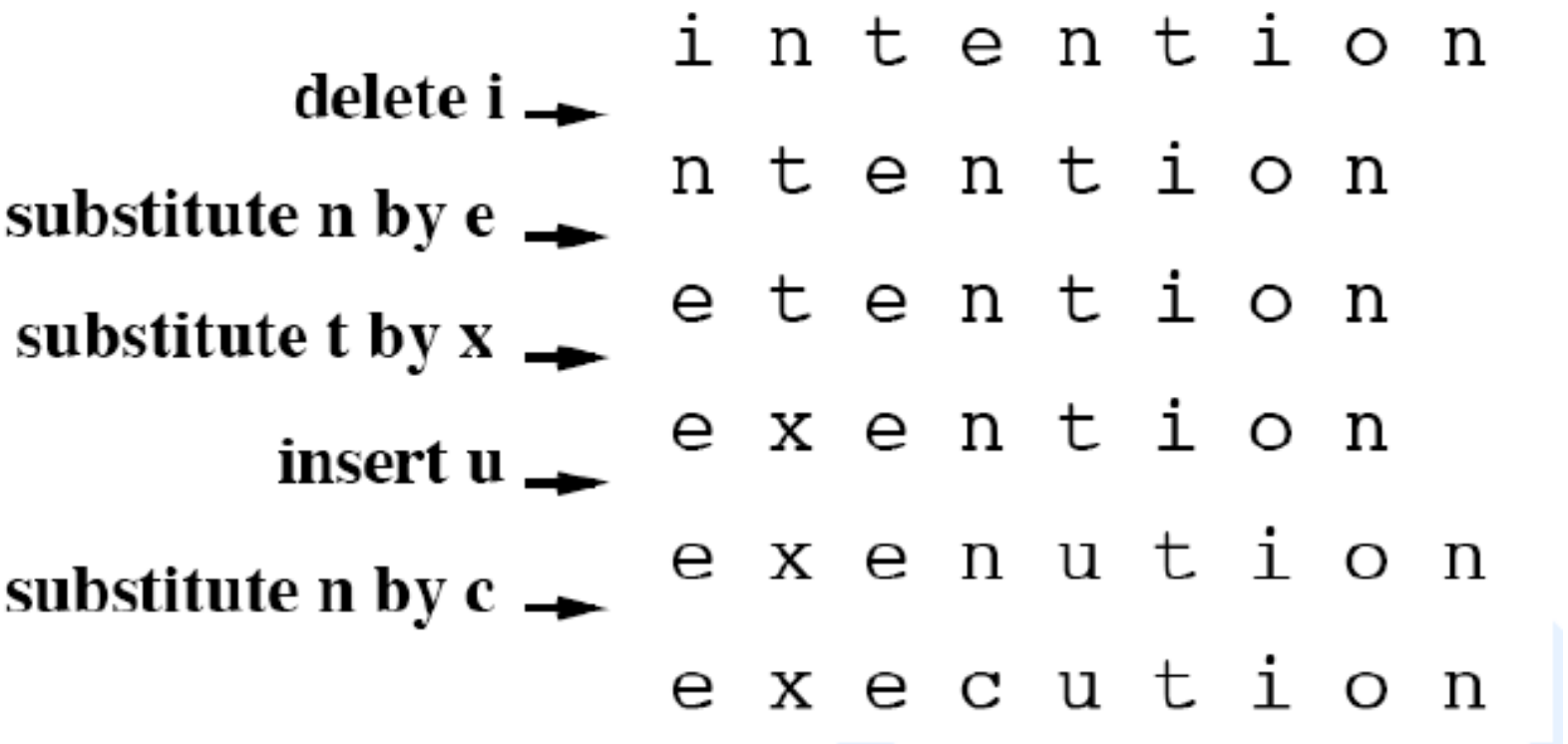
Edit Distance

- The minimum edit distance between two strings is the minimum number of editing operations
  - Insertion
  - Deletion
  - Substitution
- that one would need to transform one string into the other

Note

- The following discussion has 2 goals
  1. Learn the minimum edit distance computation and algorithm
  2. Introduce dynamic programming

Min Edit Example



Minimum Edit Distance

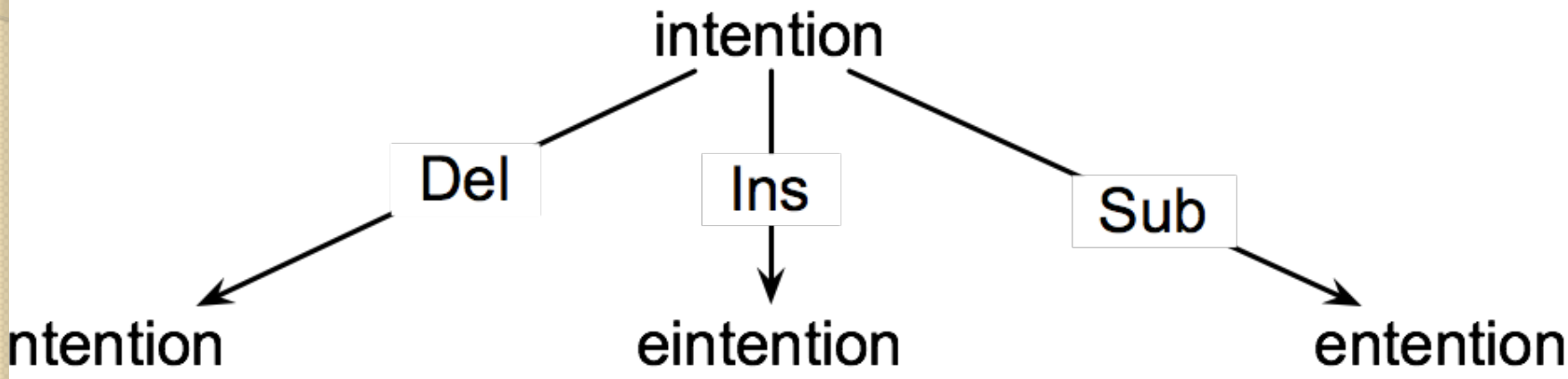
```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

- If each operation has cost of 1 distance between these is 5
- If substitutions cost 2 (Levenshtein) distance between these is 8

Min Edit As Search

- That's all well and good but how did we find that particular (minimum) set of operations for those two strings?
- We can view edit distance as a search for a path (a sequence of edits) that gets us from the start string to the final string
  - Initial state is the word we're transforming
  - Operators are insert, delete, substitute
  - Goal state is the word we're trying to get to
  - Path cost is what we're trying to minimize: the number of edits

Min Edit as Search



Min Edit As Search

- But that generates a huge search space
- Navigating that space in a naïve backtracking fashion would be incredibly wasteful
- Why?

Lots of distinct paths wind up at the same state. But there is no need to keep track of the them all. We only care about the shortest path to each of those revisited states.

Defining Min Edit Distance

- For two strings S_1 of len *n*, S_2 of len *m*
  - distance(*i,j*) or D(*i,j*)
    - Is the min edit distance of S_1[1..*i*] and S_2[1..*j*]
      - That is, the minimum number of edit operations need to transform the first *i* characters of S_1 into the first *j* characters of S_2
    - The edit distance of S_1, S_2 is D(*n,m*)
- We compute D(*n,m*) by computing D(*i,j*) for all *i* (0 < *i* < n) and

  j (0 < j < m)

Edit Distance

R	I	G	H	T				
					R	I	T	E
D	D	D	D	D	I	I	I	I
I	I	I	I	I	I	I	I	I

Edit Distance

9

R	I	G	H	T
R	I	T	E	
		S	S	D
0	0	2	2	I

5

R	I	G	H	T	
R	I			T	E
		D	D		I
0	0	I	I	0	I

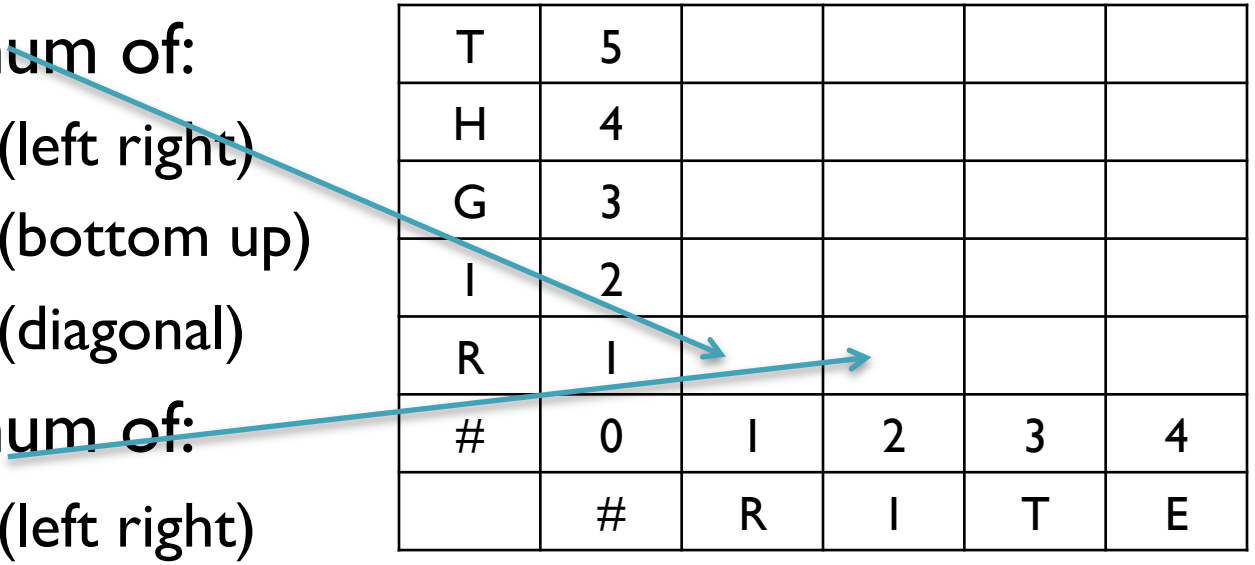
3

Minimum Edit Distance Algorithm

- Create Matrix
- Initialize 1 – length in LH column and bottom row
- For each cell
  - Take the minimum of:
    - Deletion: +1 from left cell
    - Insertion: +1 from cell below
    - Substitution: Diagonal +0 if same +2 if different
  - Keep track of where you came from

Example

- Minimum of:
  - 1+1 (left right)
  - 1+1 (bottom up)
  - 0+0 (diagonal)
- Minimum of:
  - 0+1 (left right)
  - 2+1 (bottom up)
  - 1+2 (diagonal)



T	5				
H	4				
G	3				
I	2				
R	I				
#	0	I	2	3	4
	#	R	I	T	E

Answer to Right-Rite

T	5	6, 6, 4	5, 5, 5	6, 2, 4	3, 5, 5
H	4	5, 5, 3	4, 4, 2	3, 3, 3	4, 4, 4
G	3	4, 4, 2	3, 3, 1	2, 2, 2	3, 3, 3
I	2	3, 3, 1	2, 0, 2	1, 3, 3	2, 4, 4
R	1	2, 0, 2	1, 3, 3	2, 4, 4	3, 5, 5
#	0	1	2	3	4
	#	R	I	T	E

In each box X, Y, Z values are
- X: From left: Insert-add one from left box
- Y: Diagonal, Compare-0 if same, 2 if different
- Z: From below: Delete-add one from lower box

Minimum is highlighted
in red with arrow to source
NOTE: All boxes will have arrows.
I didn't show them all.
Only one path back to root.

Defining Min Edit Distance

- Base conditions:
  - D(*i*,0) = *i*
  - D(0,*j*) = *j*

  - Recurrence Relation*:*
  - $D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$

Dynamic Programming

- A tabular computation of D(n,m)
- Bottom-up
  - We compute D(i,j) for small i,j
  - And compute larger D(i,j) based on previously computed smaller values

The Edit Distance Table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

N	9	8	9	10	11	12	11	10	9	**8**
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

Min Edit Distance

- Note that the result isn't all that informative
  - For a pair of strings we get back a single number
    - The min number of edits to get from here to there
- That's like a map routing program that tells you the distance from here to Denver but doesn't tell you how to get there.

Paths

- Keep a back pointer
  - Every time we fill a cell add a pointer back to the cell that was used to create it (the min cell that lead to it)
  - To get the sequence of operations follow the backpointer from the final cell

Backtrace

N	9	8	9	10	11	12	11	10	9	**8**
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

Adding Backtrace to MinEdit

- Base conditions:
  - D(*i*,0) = *i*
  - D(0,*j*) = *j*
- Recurrence Relation*:*
  - $D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{Case 1} \\ D(i,j-1) + 1 & \text{Case 2} \\ D(i-1,j-1) + \begin{cases} 1; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} & \text{Case 3} \end{cases}$

$$ptr(i,j) \begin{cases} \text{LEFT} & \text{Case 1} \\ \text{DOWN} & \text{Case 2} \\ \text{DIAG} & \text{Case 3} \end{cases}$$

Complexity

- Time:

  O(nm)

- Space:

  O(nm)

- Backtrace

  O(n+m)

Alignments

- An alignment is a 1 to 1 pairing of each element in a sequence with a corresponding element in the other sequence or with a gap...

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

Weighted Edit Distance

- Why would we want to add weights to the minimum edit distance computation?

Confusion matrix

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

X \ Y (correct)	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	7	1	342	0	0	2	118	0	1	0	0	3	76	0	0	1	35	9	9	0	1	0	5	0
b	0	0	9	9	2	2	3	1	0	0	0	5	11	5	0	10	0	0	2	1	0	0	8	0	0	0
c	6	5	0	16	0	9	5	0	0	0	1	0	7	9	1	10	2	5	39	40	1	3	7	1	1	0
d	1	10	13	0	12	0	5	5	0	0	2	3	7	3	0	1	0	43	30	22	0	0	4	0	2	0
e	388	0	3	11	0	2	2	0	89	0	0	3	0	5	93	0	0	14	12	6	15	0	1	0	18	0
f	0	15	0	3	1	0	5	2	0	0	0	3	4	1	0	0	0	6	4	12	0	0	2	0	0	0
g	4	1	11	11	9	2	0	0	0	1	1	3	0	0	2	1	3	5	13	21	0	0	1	0	3	0
h	1	8	0	3	0	0	0	0	0	0	2	0	12	14	2	3	0	3	1	11	0	0	2	0	0	0
i	103	0	0	0	146	0	1	0	0	0	0	6	0	0	49	0	0	0	2	1	47	0	2	1	15	0
j	0	1	1	9	0	0	1	0	0	0	0	2	1	0	0	0	0	0	5	0	0	0	0	0	0	0
k	1	2	8	4	1	1	2	5	0	0	0	0	5	0	2	0	0	0	6	0	0	0	4	0	0	3
l	2	10	1	4	0	4	5	6	13	0	1	0	0	14	2	5	0	11	10	2	0	0	0	0	0	0
m	1	3	7	8	0	2	0	6	0	0	4	4	0	180	0	6	0	0	9	15	13	3	2	2	3	0
n	2	7	6	5	3	0	1	19	1	0	4	35	78	0	0	7	0	28	5	7	0	0	1	2	0	2
o	91	1	1	3	116	0	0	0	25	0	2	0	0	0	0	14	0	2	4	14	39	0	0	0	18	0
p	0	11	1	2	0	6	5	0	2	9	0	2	7	6	15	0	0	1	3	6	0	4	1	0	0	0
q	0	0	1	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	14	0	30	12	2	2	8	2	0	5	8	4	20	1	14	0	0	12	22	4	0	0	1	0	0
s	11	8	27	33	35	4	0	1	0	1	0	27	0	6	1	7	0	14	0	15	0	0	5	3	20	1
t	3	4	9	42	7	5	19	5	0	1	0	14	9	5	5	6	0	11	37	0	0	2	19	0	7	6
u	20	0	0	0	44	0	0	0	64	0	0	0	0	2	43	0	0	4	0	0	0	0	2	0	8	0
v	0	0	7	0	0	3	0	0	0	0	0	1	0	0	1	0	0	0	8	3	0	0	0	0	0	0
w	2	2	1	0	1	0	0	2	0	0	1	0	0	0	0	7	0	6	3	3	1	0	0	0	0	0
x	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0
y	0	0	2	0	15	0	1	7	15	0	0	0	2	0	6	1	0	7	36	8	5	0	0	1	0	0
z	0	0	0	7	0	0	0	0	0	0	0	7	5	0	0	0	0	2	21	3	0	0	0	0	3	0

Problem: similarity of strings

Edit distance

For two given *A* and *B*, compute, as efficiently as possible, the edit distance *D(A,B)* and a minimal sequence of edit operations which transforms *A* into *B*.

i n f - - - o r m a t i k
-
i n t e r p o l - a t i o n

Problem: similarity of strings

Approximate string matching

For a given text T, a pattern P, and a distance d, find all substrings P' in T with $D(P,P') \leq d$

Sequence alignment

Find optimal alignments of DNA sequences

```
G A G C A - C T T G G A T T C T C G G
- - - C A C G T G G - - - - - - - - -
```

Edit distance

Given: two strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$

Wanted: minimal cost $D(A,B)$ for a sequence of edit operations to transform A into B.

Edit operations:

1. Replace one character in A by a character from B
2. Delete one character from A
3. Insert one character from B

Edit distance

Cost model:

$$c(a,b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases}$$

$a = \varepsilon,\ b = \varepsilon$ possible

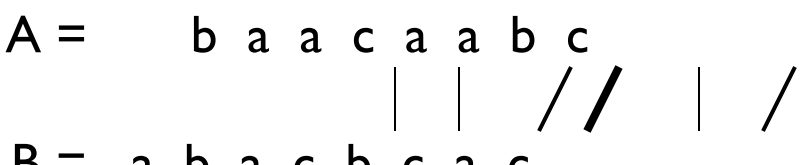
We assume the triangle inequality holds for *c*:

$$c(a,c) \leq c(a,b) + c(b,c)$$

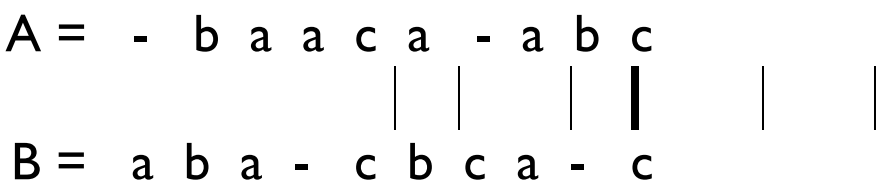
→ Each character is changed at most once

Edit distance

Trace as representation of edit sequences



or using indels



Edit distance (cost): 5

Division of an optimal trace results in two optimal sub-traces
→ dynamic programming can be used

Computation of the edit distance

Let $A_i = a_1 \ldots a_i$ and $B_j = b_1 \ldots b_j$

$D_{i,j} = D(A_i, B_j)$

Computation of the edit distance

Three possibilities of ending a trace:

1. a_m is replaced by b_n :

   $D_{m,n} = D_{m-1,n-1} + c(a_m, b_n)$

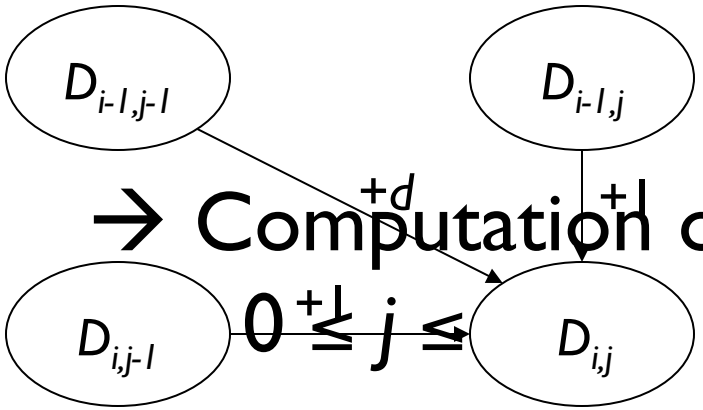
2. a_m is deleted: $D_{m,n} = D_{m-1,n} + 1$

3. b_n is inserted: $D_{m,n} = D_{m,n-1} + 1$

Computation of the edit distance

Recurrence relation, if $m,n \geq 1$:

$$D_{m,n} = \min \begin{cases} D_{m-1,n-1} + c(a_m, b_n), \\ D_{m-1,n} + 1, \\ D_{m,n-1} + 1 \end{cases}$$



→ Computation of all $D_{i,j}$ is required, $0 \leq i \leq$ $0 \leq j \leq$

Recurrence relation for the edit distance

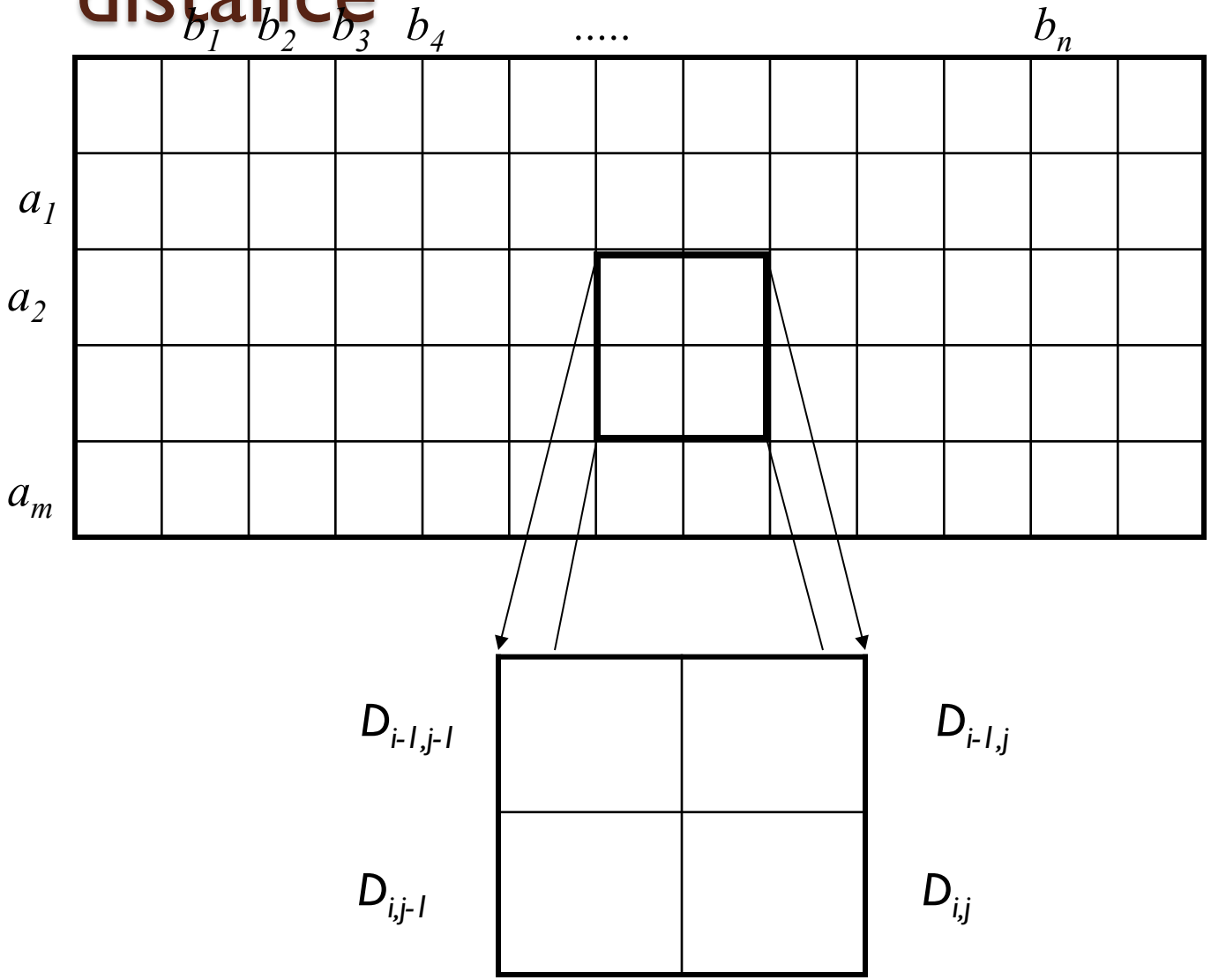
Base cases:

$$D_{0,0} = D(\varepsilon, \varepsilon) = 0$$
$$D_{0,j} = D(\varepsilon, B_j) = j$$
$$D_{i,0} = D(A_i, \varepsilon) = i$$

Recurrence equation:

$$D_{i,j} = \min \begin{Bmatrix} D_{i-1,j-1} + c(a_i, b_j) \\ D_{i-1,j} + 1, \\ D_{i,j-1} + 1 \end{Bmatrix}$$

Order of computation for the edit distance



Algorithm for the edit distance

Algorithm edit_distance
Input: two strings $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$
Output: the matrix $D = (D_{ij})$

```
1 D[0,0] := 0
2 for i := 1 to m do D[i,0] = i
3 for j := 1 to n do D[0,j] = j
4 for i := 1 to m do
5   for j := 1 to n do
6       D[i,j] := min( D[i - 1,j] + 1,
7                      D[i,j - 1] + 1,
8                      D[i –1, j – 1] + c(a_i,b_j))
```

Example

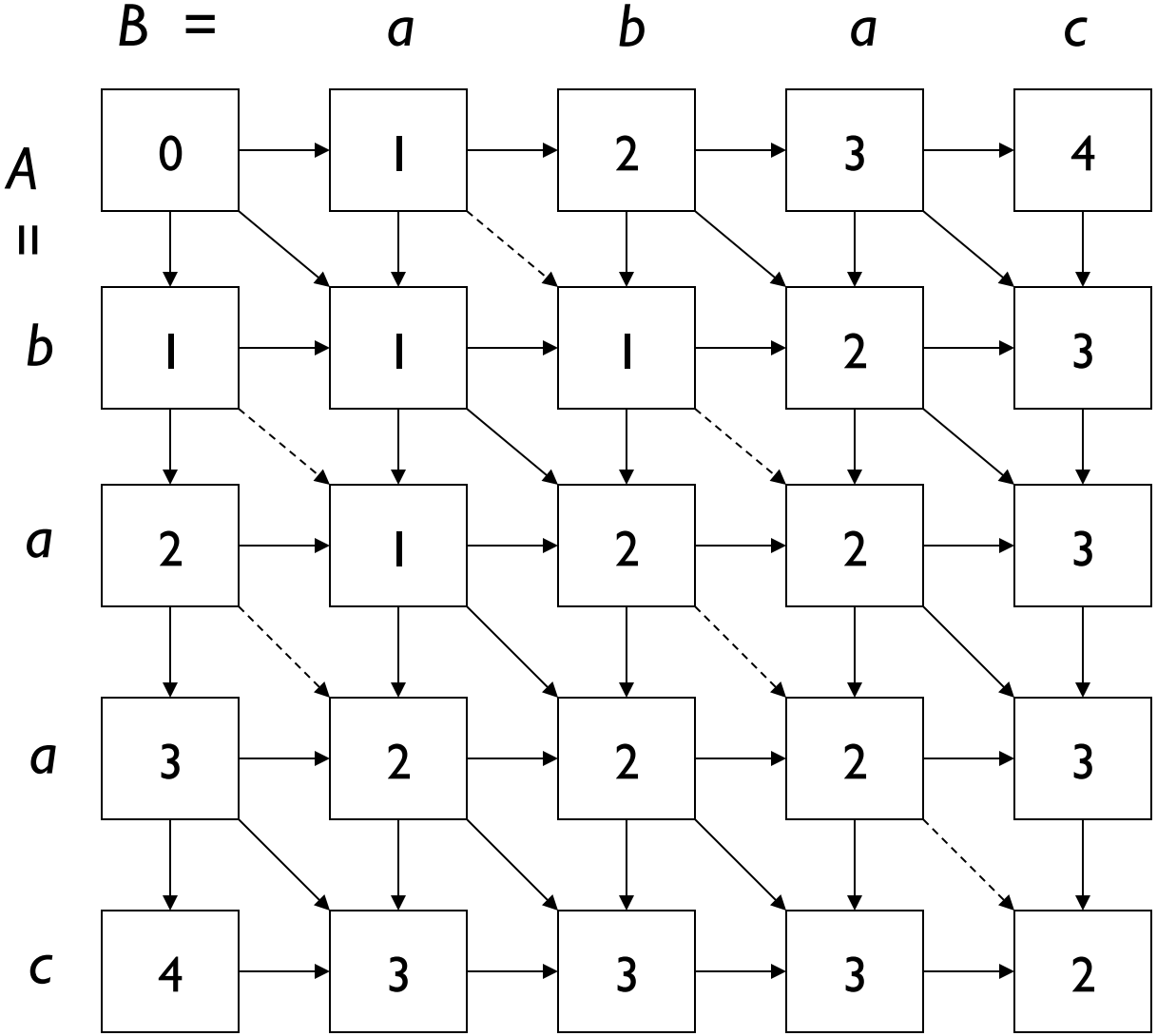
		a	b	a	c
	0	1	2	3	4
b	1				
a	2				
a	3				
c	4				

Computation of the edit operations

```
Algorithm edit_operations (i,j)
Input:  matrix D (computed)
1   if i = 0 and j = 0 then return
2   if i ≠ 0 and D[i,j] = D[i – 1 , j] + 1
3     then „delete a[i]“
4             edit_operations (i – 1, j)
5   else if j ≠ 0 and D[i,j] = D[i, j – 1] + 1
6      then „insert b[j]“
7             edit_operations (i, j – 1)
8   else
      /* D[i,j] = D[i – 1, j – 1 ] + c(a[i], b[j])  */
9           „replace a[i] by b[j] “
10           edit_operations (i – 1, j – 1)

Initial call: edit_operations(m,n)
```

Trace graph of the edit operations



Sub-graph of the edit operations

Trace graph: Overview of all possible traces for the transformation

of A into B, directed edges from vertex (i, j) to $(i + 1, j)$, $(i, j + 1)$ and

$(i + 1, j + 1)$.

Weights of the edges represent the edit costs.

Costs are monotonic increasing along an optimal path.

Each path with monotonic increasing cost from the upper left corner

to the lower right corner represents an optimal trace.

Approximate string matching

Given: two strings $P = p_1 p_2 \ldots p_m$ (pattern) and

$T = t_1 t_2 \ldots t_n$ (text)

Wanted: an interval $[j', j]$, $1 \le j' \le j \le n$, such that the substring $T_{j',j} = t_{j'} \ldots t_j$ of T is the one with the greatest similarity to pattern P, i.e. for all other intervals $[k', k]$, $1 \le k' \le k \le n$:

$$D(P, T_{j',j}) \le D(P, T_{k',k})$$

Approximate string matching

Naïve approach:

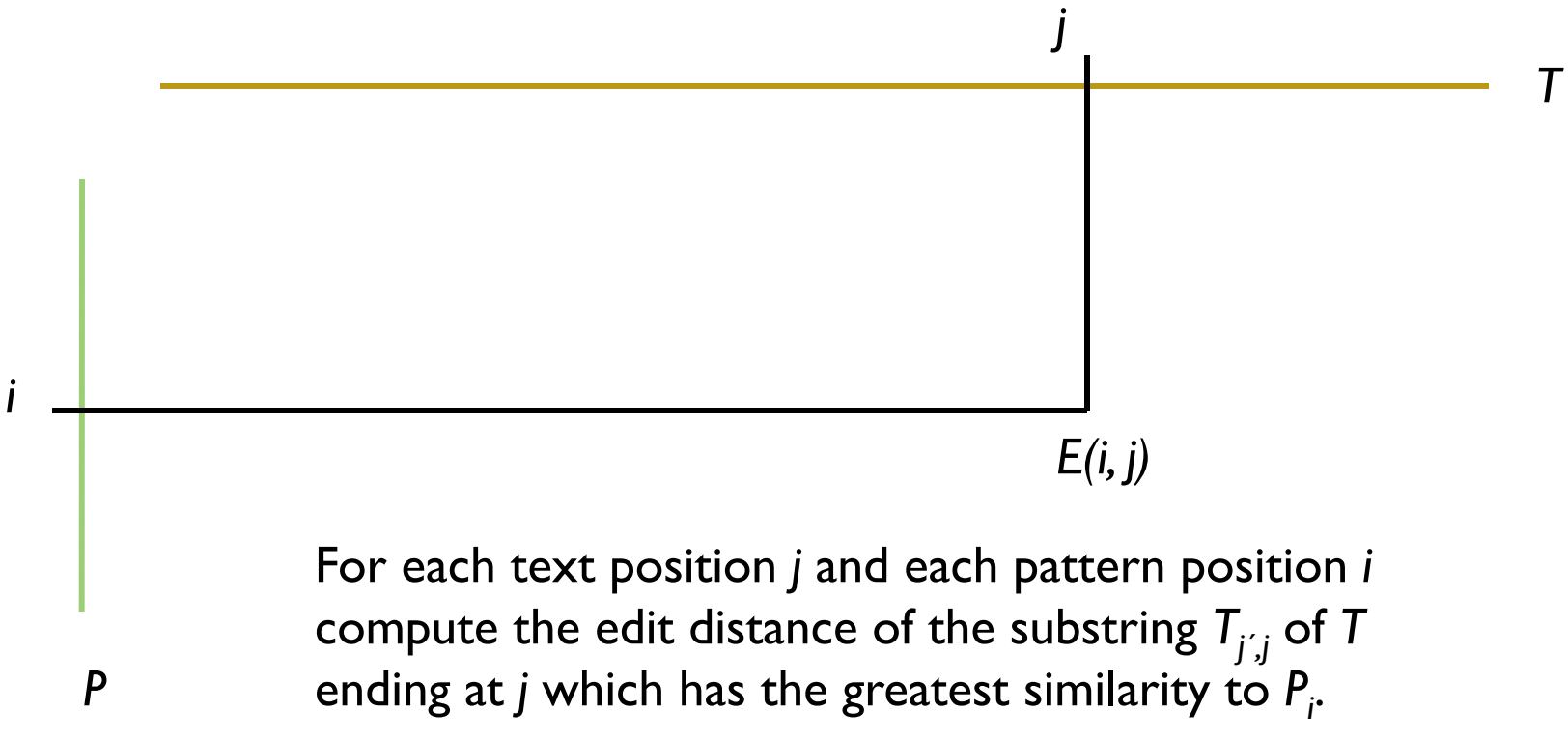
for all $1 \leq j' \leq j \leq n$ **do**

compute $D(P, T_{j',j})$

choose minimum

Approximate string matching

Consider a related problem:



For each text position j and each pattern position i compute the edit distance of the substring $T_{j',j}$ of T ending at j which has the greatest similarity to P_i.

Approximate string matching

Method:
for all $1 \le j \le n$ ***do***
compute j' such that $D(P,T_{j',j})$ is minimal

For $1 \le i \le m$ and $0 \le j \le n$ let:

$$E_{i,j} = \min_{1 \le j' \le' j+1} D(P_i, T_{j',j})$$

Optimal trace:

P_i = b a a c a a b c
$T_{j',j}$ = b a c b c a c

Approximate string matching

Recurrence relation:

$$E_{i,j} = \min\begin{Bmatrix} E_{i-1,j-1} + c(p_i, t_j), \\ E_{i-1,j} + 1, \\ E_{i,j-1} + 1 \end{Bmatrix}$$

Remark:

j' can be completely different for $E_{i-1,j-1}$, $E_{i-1,j}$ and $E_{i,j-1}$.

A subtrace of an optimal trace is an optimal subtrace.

Approximate string matching

Base cases:

$$E_{0,0} = E(\varepsilon, \varepsilon) = 0$$
$$E_{i,0} = E(P_j, \varepsilon) = i$$

but

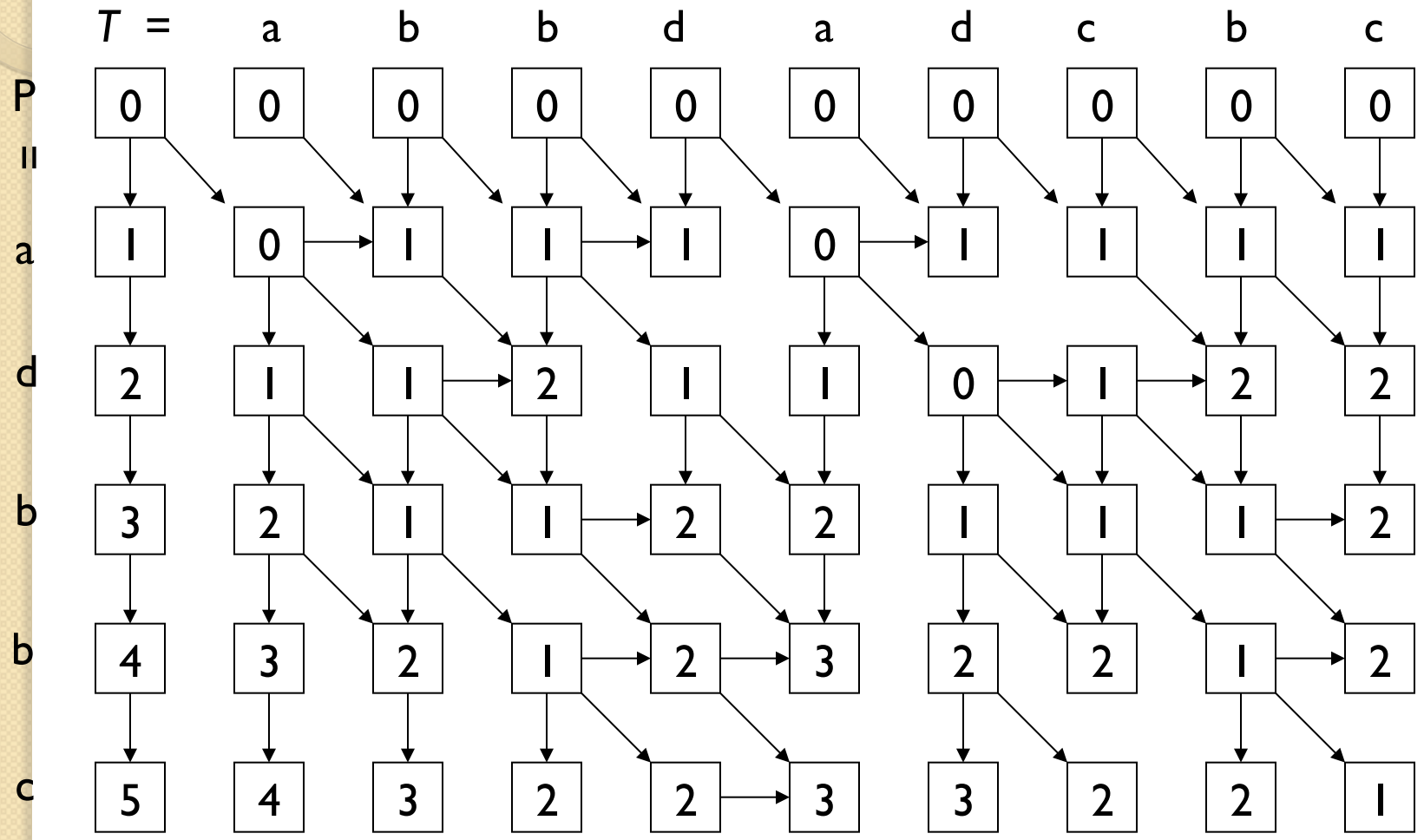
$$E_{0,j} = E(\varepsilon, T_j) = 0$$

Observation:

The optimal edit sequence from P to $T_{j',j}$ does not start with an insertion of $t_{j'}$.

Approximate string matching

Dependency graph



Approximate string matching

Theorem

If there is a path from $E_{0,j'-1}$ to $E_{i,j}$ in the dependency graph, then

$T_{j',j}$ is a substring of T ending in j with the greatest similarity to P_i

and

$$D(P_i, T_{j',j}) = E_{i,j}$$

Back To Spelling

- Remember graffe…
- We can compute the score/distance between graffe and assorted candidates
- MS Word gives
  - giraffe
  - gaffe
  - giraffes
  - graft

How does it do that?

DP Search

- In the context of language processing (and signal processing) this kind of algorithm is often referred to as a DP search
  - Min edit distance
  - Viterbi and Forward algorithms
  - CKY and Earley
  - MT decoding

Word Prediction

- Guess the next word…
  - *So I notice three guys standing on the ???*

What are some of the knowledge sources you used to come up with those predictions?

Word Prediction

- We can formalize this task using what are called *N*-gram models
  - *N*-grams are token sequences of length *N*
    - -gram means "written"
- Our earlier example contains the following 2-grams (aka bigrams)
  - (So I), (I notice), (notice three), (three guys), (guys standing), (standing on), (on the)
- Given knowledge of counts of N-grams such as these, we can guess likely next words in a sequence.

N-Gram Models

- More formally, we can use knowledge of the counts of *N*-grams to assess the conditional probability of candidate words as the next word in a sequence.
- Or, we can use them to assess the probability of an entire sequence of words.
  - Pretty much the same thing as we'll see...

Applications

- It turns out that being able to predict the next word (or any linguistic unit) in a sequence is an extremely useful thing to be able to do.
- As we'll see, it lies at the core of the following applications
  - Automatic speech recognition
  - Handwriting and character recognition
  - Spelling correction
  - Machine translation
  - And many more

Counting

- Simple counting lies at the core of any probabilistic approach. So let's first take a look at what we're counting.
  - *He stepped out into the hall, was delighted to encounter a water brother.*
    - 13 tokens, 15 if we include "," and "." as separate tokens.
    - Assuming we include the comma and period as tokens, how many bigrams are there?

Counting

- Not always that simple
  - *I do uh main- mainly business data processing*
- Spoken language poses various challenges.
  - Should we count "uh" and other fillers as tokens?
  - What about the repetition of "mainly"? Should such do-overs count twice or just once?
  - The answers depend on the application.
    - If we're focusing on something like ASR to support indexing for search, then "uh" isn't helpful (it's not likely to occur as a query).
    - But filled pauses are very useful in dialog management, so we might want them there
- Tokenization of text raises the same kinds of issues

Counting: Types and Tokens

- How about
  - *They picnicked by the pool, then lay back on the grass and looked at the stars.*
    - 18 tokens (again counting punctuation)
- But we might also note that "*the*" is used 3 times, so there are only 16 unique types (as opposed to tokens).
- In going forward, we'll have occasion to focus on counting both types and tokens of both words and *N*-grams.

Counting: Corpora

- What happens when we look at large bodies of text instead of single utterances

- Google Web Crawl
  - Crawl of 1,024,908,267,229 English tokens in Web text
  - 13,588,391 wordform types
    - That seems like a lot of types... After all, even large dictionaries of English have only around 500k types. Why so many here?

- Numbers
- Misspellings
- Names
- Acronyms
- etc

Language Modeling

- Now that we know how to count, back to word prediction
- We can model the word prediction task as the ability to assess the conditional probability of a word given the previous words in the sequence
  - $P(w_n|w_1,w_2...w_{n-1})$
- We'll call a statistical model that can assess this a *Language Model*

Language Modeling

- How might we go about calculating such a conditional probability?
  - One way is to use the definition of conditional probabilities and look for counts. So to get
  - P(*the* | *its water is so transparent that*)
- By definition, that is:

  $$\frac{P(\text{its water is so transparent that the})}{P(\text{its water is so transparent that})}$$

  We can get each of those from counts in a large corpus.

Very Easy Estimate

- How to estimate?
  - P(the | its water is so transparent that)

P(the | its water is so transparent that) =

$$\frac{\text{Count(its water is so transparent that the)}}{\text{Count(its water is so transparent that)}}$$

Very Easy Estimate

- According to Google those counts are 12000 and 19000 so the conditional probability of interest is...

  - P(the | its water is so transparent that) = 0.63

Language Modeling

- Unfortunately, for most sequences and for most text collections we won't get good estimates from this method.
  - What we're likely to get is 0. Or worse 0/0.

- Clearly, we'll have to be a little more clever.
  - Let's first use the chain rule of probability
  - And then apply a particularly useful independence assumption

The Chain Rule

- Recall the definition of conditional probabilities
- Rewriting:

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

$$P(A \wedge B) = P(A \mid B)P(B)$$

- For sequences...
  - P(A,B,C,D) = P(A)P(B|A)P(C|A,B)P(D|A,B,C)
- In general
  - $P(x_1,x_2,x_3,\ldots x_n) = P(x_1)P(x_2|x_1)P(x_3|x_1,x_2)\ldots P(x_n|x_1\ldots x_{n-1})$

The Chain Rule

$$P(w_1^n) = P(w_1)P(w_2|w_1)P(w_3|w_1^2)\ldots P(w_n|w_1^{n-1})$$
$$= \prod_{k=1}^{n} P(w_k|w_1^{k-1})$$

P(its water was so transparent)=

P(its)*

P(water|its)*

P(was|its water)*

P(so|its water was)*

P(transparent|its water was so)

Unfortunately

- There are still a lot of possible sequences in there
- In general, we'll never be able to get enough data to compute the statistics for those longer prefixes
  - Same problem we had for the strings themselves

Independence Assumption

- Make the simplifying assumption
  - P(lizard| the,other,day,I,was,walking,along,and,saw ,a) = P(lizard|a)
- Or maybe
  - P(lizard| the,other,day,I,was,walking,along,and,saw ,a) = P(lizard|saw,a)
- That is, the probability in question is to some degree *independent* of its earlier history.

Markov Assumption

So for each component in the product replace with the approximation (assuming a prefix of N - 1)

$$P(w_n \mid w_1^{n-1}) \approx P(w_n \mid w_{n-N+1}^{n-1})$$

Bigram version

$$P(w_n \mid w_1^{n-1}) \approx P(w_n \mid w_{n-1})$$

Estimating Bigram Probabilities

- The Maximum Likelihood Estimate (MLE)

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

An Example

- <s> I am Sam </s>
- <s> Sam I am </s>
- <s> I do not like green eggs and ham </s>

$$P(\text{I}\,|\,\text{<s>}) = \tfrac{2}{3} = .67 \qquad P(\text{Sam}\,|\,\text{<s>}) = \tfrac{1}{3} = .33 \qquad P(\text{am}\,|\,\text{I}) = \tfrac{2}{3} = .67$$
$$P(\text{</s>}\,|\,\text{Sam}) = \tfrac{1}{2} = 0.5 \qquad P(\text{Sam}\,|\,\text{am}) = \tfrac{1}{2} = .5 \qquad P(\text{do}\,|\,\text{I}) = \tfrac{1}{3} = .33$$

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})}$$

Maximum Likelihood Estimates

- The maximum likelihood estimate of some parameter of a model M from a training set T
  - Is the estimate that maximizes the likelihood of the training set T given the model M
- Suppose the word "*Chinese*" occurs 400 times in a corpus of a million words (Brown corpus)
- What is the probability that a random word from some other text from the same distribution will be "Chinese"
- MLE estimate is 400/1000000 = .004
  - This may be a bad estimate for some other corpus
- But it is the **estimate** that makes it **most likely** that "Chinese" will occur 400 times in a million word corpus.

Berkeley Restaurant Project Sentences

- *can you tell me about any good cantonese restaurants close by*
- *mid priced thai food is what i'm looking for*
- *tell me about chez panisse*
- *can you give me a listing of the kinds of food that are available*
- *i'm looking for a good place to eat breakfast*
- *when is caffe venezia open during the day*

Bigram Counts

- Out of 9222 sentences
  - Eg. "I want" occurred 827 times

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Bigram Probabilities

- Divide bigram counts by prefix unigram counts to get probabilities

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Bigram Estimates of Sentence Probabilities

- P(<s> I want english food </s>) =
  P(i|<s>)*
  P(want|I)*
  P(english|want)*
  P(food|english)*
  P(</s>|food)*
  =.000031

Kinds of Knowledge

- As crude as they are, *N*-gram probabilities capture a range of interesting facts about language.

- P(english|want) = .0011
- P(chinese|want) = .0065
- P(to|want) = .66
- P(eat | to) = .28
- P(food | to) = 0
- P(want | spend) = 0
- P (i | <s>) = .25

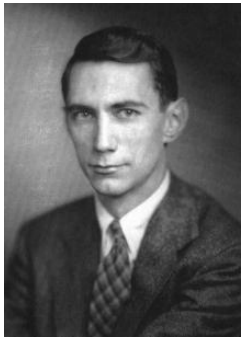
World knowledge

Syntax

Discourse

Shannon's Method

- Assigning probabilities to sentences is all well and good, but it's not terribly illuminating . A more entertaining task is to turn the model around and use it to generate random sentences that are *like* the sentences from which the model was derived.
- Generally attributed to Claude Shannon.



Shannon's Method

- Sample a random bigram (<s>, w) according to the probability distribution over bigrams
- Now sample a new random bigram (w, x) according to its probability
  - Where the prefix w matches the suffix of the first.
- And so on until we randomly choose a (y, </s>)
- Then string the words together
- <s> I
  I want
  want to
  to eat
  eat Chinese
  Chinese food
  food </s>

Shakespeare

Unigram	• To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have • Every enter now severally so, let • Hill he late speaks; or! a more to leg less first you enter • Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like
Bigram	• What means, sir. I confess she? then all sorts, he is trim, captain. •Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. •What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman? •Enter Menenius, if it so many good direction found'st thou art a strong upon com-mand of fear not a liberal largess given away, Falstaff! Exeunt
Trigram	• Sweet prince, Falstaff shall die. Harry of Monmouth's grave. • This shall forbid it should be branded, if renown made it empty. • Indeed the duke; and had a very good friend. • Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.
Quadrigram	• King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; • Will you not tell me who I am? • It cannot be but so. • Indeed the short and the long. Marry, 'tis a noble Lepidus.

Shakespeare as a Corpus

- N=884,647 tokens, V=29,066
- Shakespeare produced 300,000 bigram types out of V^2= 844 million possible bigrams...
  - So, 99.96% of the possible bigrams were never seen (have zero entries in the table)
  - This is the biggest problem in language modeling; we'll come back to it.
- Quadrigrams are worse: What's coming out looks like Shakespeare because it ***is*** Shakespeare

The Wall Street Journal is Not Shakespeare

unigram: Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives

bigram: Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her

trigram: They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Evaluating *N*-Gram Models

- Best evaluation for a language model
  - Put model A into an application
    - For example, a speech recognizer
  - Evaluate the performance of the application with model A
  - Put model B into the application and evaluate
  - Compare performance of the application with the two models
  - ***Extrinsic evaluation***

Difficulty of extrinsic (in-vivo) evaluation of N-gram models

- Extrinsic evaluation
  - This is really time-consuming
  - Can take days to run an experiment
- So
  - As a temporary solution, in order to run experiments
  - To evaluate N-grams we often use an **intrinsic** evaluation, an approximation called **perplexity**
  - But perplexity is a poor approximation unless the test data looks **just** like the training data
  - So is **generally only useful in pilot experiments (generally is not sufficient to publish)**
  - But is helpful to think about.

Model Evaluation

- How do we know if our models are any good?
  - And in particular, how do we know if one model is better than another.
- Well Shannon's game gives us an intuition.
  - The generated texts from the higher order models sure look better.
    - That is, they sound more like the text the model was obtained from.
  - The generated texts from the WSJ and Shakespeare models look different
    - That is, they look like they're based on different underlying models.
- But what does that mean? Can we make that notion operational?

Evaluation

- Standard method
  - Train parameters of our model on a **training set**.
  - Look at the models performance on some new data
    - This is exactly what happens in the real world; we want to know how our model performs on data we haven't seen
  - So use a **test set**. A dataset which is different than our training set, but is drawn from the same source
  - Then we need an **evaluation metric** to tell us how well our model is doing on the test set.
    - One such metric is **perplexity**

But First

- But once we start looking at test data, we'll run into words that we haven't seen before (pretty much regardless of how much training data you have.
- With an *Open Vocabulary* task
  - Create an unknown word token <UNK>
  - Training of <UNK> probabilities
    - Create a fixed lexicon L, of size V
      - From a dictionary or
      - A subset of terms from the training set
    - At text normalization phase, any training word not in L changed to <UNK>
    - Now we count that like a normal word
  - At test time
    - Use UNK counts for any word not in training

Perplexity

- The intuition behind perplexity as a measure is the notion of surprise.
  - How surprised is the language model when it sees the test set?
    - Where surprise is a measure of…
      - Gee, I didn't see that coming…
    - The more surprised the model is, the lower the probability it assigned to the test set
    - The higher the probability, the less surprised it was

Perplexity

- Perplexity is the probability of a test set (assigned by the language model), as normalized by the number of words:

$$\begin{aligned} \mathrm{PP}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \end{aligned}$$

- Chain rule:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

- For bigrams:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

- Minimizing perplexity is the same as maximizing probability
  - **The best language model is one that best predicts an unseen test set**

Lower perplexity means a better model

- Training 38 million words, test 1.5 million words, WSJ

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109