Unit - 5
Probabilistic Models of Pronunciation and Spelling

Dealing with Spelling Error

- Typing is a manual process and not everyone does it correctly. Spelling mistakes occur frequently
- The detection and correction of spelling errors is an integral part of modern word-processors
- Even in applications like Optical Character Recognition(OCR) and on-line handwriting recognition the individual letters are not accurately identified
- **Optical Character Recognition(OCR**) – Automatic recognition of machine or hand-printed characters.
  - Optical scanner converts a machine or hand-printed page into a bitmap which is then passed to OCR
- **On-line handwriting recognition** – is the recognition of human printed or cursive handwriting as the user writing
  - Takes advantage of dynamic information of the input such as number and order of strokes and the speed and direction of each stroke

- Optical Character Recognition (OCR) is the process that converts an image of text into a machine-readable text format.
- OCR have higher error rates than human typists, although they tend to make different errors than typists.
  - Example: OCR systems often misread “**D**” as “**O**” or “**ri**” as “**n**” producing mis-spelled words like ***dension*** for ***derision*** or ***POQ Bach*** for ***PDQ Bach***

- There are two spelling tasks in such scenarios
  - Error Detection
  - Error Correction

- Spelling Error Detection: It is a process of detecting and sometimes providing suggestions for incorrectly spelled words in a text.

  - **Example*:*** Spell Checker is an application program that flags words in a document that may not be spelled correctly.

Spelling error correction

Spelling correction can be broken down into three broader problems (Kukich (1992)):

- **Non-word error detection**: Detecting spelling errors that result in non-words

  Example: opportnity for opportunity , graffe for giraffe

- **Isolated-word error correction**: Check each word on its own for misspelling and correcting spelling errors that result in non-words

  Example: correcting **graffe to giraffe, from to form**

- **Context-dependent error detection and correction (real-word error):** Using the context to help detect and correct spelling errors
- Some of these may accidentally result in an actual word
  - Typographical errors (insertion, deletion, transposition) which accidently produce a real word **e.g. there for three**
  - Homophone or near-homophone **e.g. dessert for desert, or piece for peace**

Spelling Error Patterns

- The number and nature of spelling errors in human typed text differs from those caused by pattern-recognition devices like OCR and handwriting recognizers

- In an early study (Damerau, 1964), it is found that 80% of all misspelled words(non-word errors) were caused by **single-error misspellings**: a single one of the following errors:

  - **Insertion**: mistyping ***the*** as ***ther***
  - **Deletion**: mistyping ***the*** as ***th***
  - **Substitution**: mistyping ***the*** as ***thw***
  - **Transposition**: mistyping ***the*** as ***hte***

- Most of the research was focused on the correction of single-error misspellings

- Kukich in 1992, classified Human typing errors into two classes:
    - **Typographic Error** - Eg: Misspelling ***spell*** as ***speel***, generally related to the keyboard
    - **Cognitive Error** - Eg: Misspelling ***separate*** as ***seperate,*** are caused by writers who don’t know how to spell the word

- Typing errors are characterized as :
    - ***Substitutions, insertions ,deletions , transpositions***

- OCR errors are usually grouped into five classes:
    - ***Substitutions, multisubstitutions, space deletions or insertions , failures***

Detecting Non word errors

- Use a dictionary
- Use the models of morphology

- For other types of spelling correction, need a model of spelling variation.

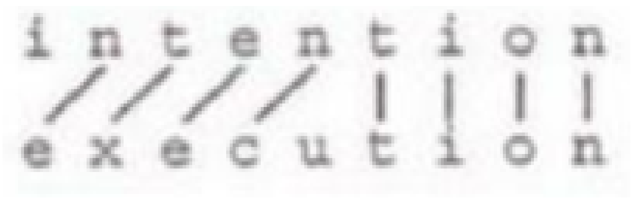
Minimum Edit Distance

- Much of natural language processing is concerned with measuring how similar two strings are.
- For example in spelling correction, the user typed some erroneous string—let’s say ***graffe***–and we want to know what the user meant.
- Which is closest?
  - ***graf***
  - ***graft***
  - ***grail***
  - ***giraffe***

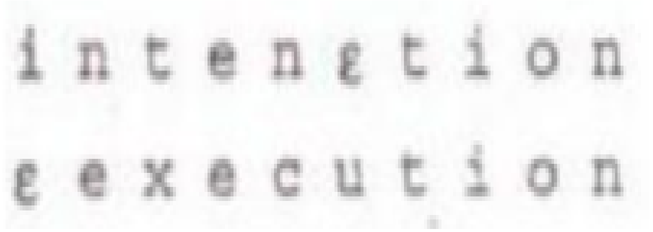
- Edit distance gives us a way to quantify both of these intuitions about string similarity.

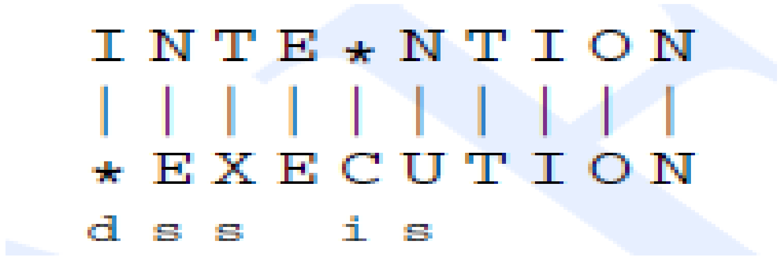
- More formally, the ***minimum edit distance*** between two strings is defined as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another.
  - ✓ **Substitution** – Change a single character from pattern s to a different character in text t, such as changing “shot” to “spot”.
  - ✓ **Insertion** – Insert a single character into pattern s to help it match text t, such as changing “ago” to “agog”.
  - ✓ **Deletion** – Delete a single character from pattern s to help it match text t, such as changing “hour” to “our”.
- For example the gap between ***intention*** and ***execution*** is five operations, which can be represented in three ways: as ***a trace, an alignment*** or ***an operation list*** as shown below:
- Many important algorithms for finding string distance rely on some version of the minimum edit distance algorithm, named by Wagner and Fischer (1974).

Trace



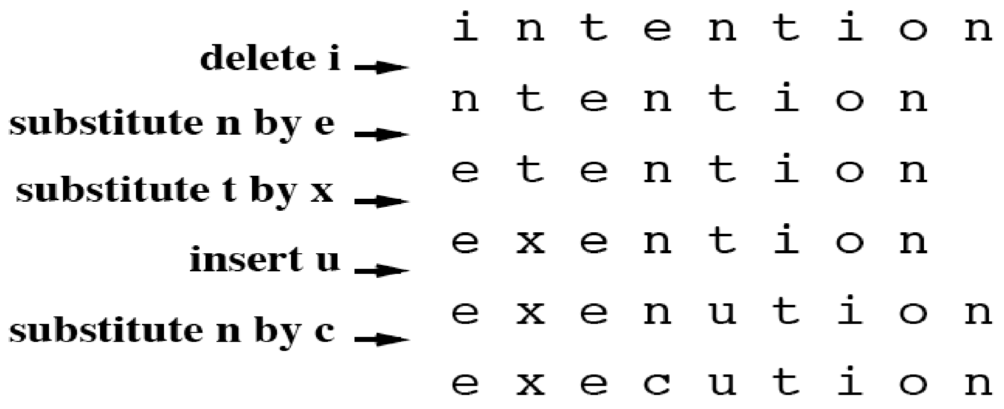
Alignment

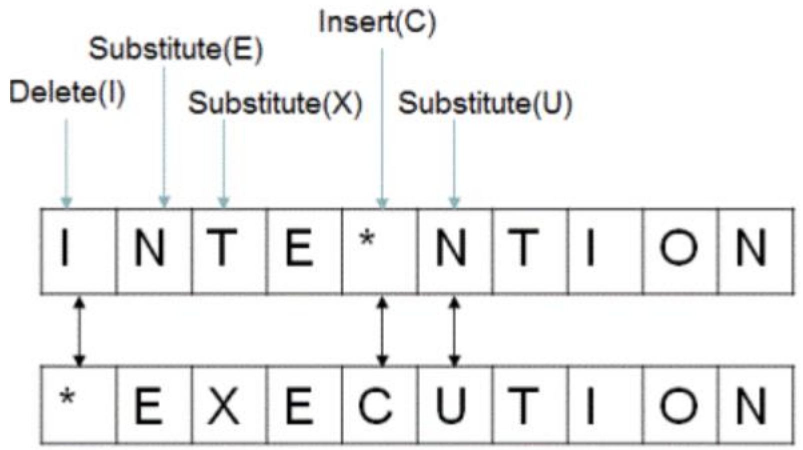




- Representing the minimum edit distance between two strings as an alignment.
- The final row gives the operation list for converting the top string into the bottom string; **d** for deletion, **s** for substitution, **i** for insertion

Operation List





- The gap between intention and execution, for example, is **5 (delete an i, substitute e for n, substitute x for t, insert c, substitute u for n)**

- Given two sequences, an alignment is a correspondence between the substrings of the two sequences

- Thus, we say ***I*** aligns with the ***empty string***, ***N*** with ***E***, ***T*** with ***X,*** and so on

- Beneath the aligned strings is another representation; a series of symbols expressing **an operation list** for converting the top string into the bottom string: ***d*** for deletion, ***s*** for substitution, ***i*** for insertion

Cost Calculation-Minimum Edit Distance

- Cost or weight can be assigned to each of these operations
- The ***Levenshtein distance*** between two sequences is the simplest weighting factor in which each of the three operations has a cost of **1**
- Assume that the substitution of a letter for itself, for example, ***t*** for ***t***, has ***zero cost***
- ***The Levenshtein distance between intention and execution is 5***

- Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of ***1*** and substitutions are not allowed.

- This is equivalent to allowing substitution, but giving each substitution a cost of ***2*** since any substitution can be represented by one insertion and one deletion)

- Using this version, ***the Levenshtein distance between intention and execution is 8.***

- Operations can be weighted by more complex functions like using confusion matrices (which assign a probability to each operation). Hence it becomes a ***maximum probability assignment*** instead of ***minimum edit distance***.

Defining Minimum Edit Distance

- Given two strings,
  - the source string ***X*** of length ***n***, and
  - the target string ***Y*** of length ***m***,

- **D[i ; j]** is defined as the edit distance between ***X[1::i]*** and ***Y[1:: j],***
  - i.e., the first ***i*** characters of ***X*** and the first ***j*** characters of ***Y***.

- The edit distance between ***X*** and ***Y*** is thus ***D[n;m]***.

Problem: Transform string **X[1…n]** into **Y[1…m]** by performing edit operations on string **X**.

Subproblem: Transform substring **X[1…i]** into **Y[1…j]** by performing edit operations on substring **X**.

Case 1: We have reached the end of either substring.

- If substring **X** is empty, insert all remaining characters of substring **Y** into **X**. The cost of this operation is equal to the number of characters left in substring **Y**.
- **(‘ ', 'ABC') ——> ('ABC', 'ABC') (cost = 3)**
- If substring **Y** is empty, delete all remaining characters of substring **X**. The cost of this operation is equal to the number of characters left in substring **X**.
- **('ABC', ‘ ') ——> (‘ ', ‘ ') (cost = 3)**

Case 2: The last characters of substring **X** and **Y** are the same.

- If the last characters of substring **X** and substring **Y** match, nothing needs to be done – simply recur for the remaining substring **X[0…i-1], Y[0…j-1].** As no edit operation is involved, the cost will be **0**.
- **('ACC', 'ABC') ——> ('AC', 'AB') (cost = 0)**

Case 3: The last characters of substring **X** and **Y** are different.

- If the last characters of substring **X** and **Y** are different, return the minimum of the following operations:

- **Insert** the last character of **Y** into **X**. The size of **Y** reduces by **1**, and **X** remains the same.
- This accounts for **X[1…i], Y[1…j-1]** as we move in on the target substring, but not in the source substring.

- **('ABA', 'ABC') ——> ('ABAC', 'ABC') == ('ABA', 'AB')** (using case 2)

- **Delete** the last character of **X**. The size of **X** reduces by **1**, and **Y** remains the same.
- This accounts for **X[1…i-1], Y[1…j]** as we move in on the source string, but not in the target string.
- **('ABA', 'ABC') ——> ('AB', 'ABC')**

- **Substitute (Replace)** the current character of **X** by the current character of **Y**. The size of both **X** and **Y** reduces by **1**. This accounts for **X[1…i-1], Y[1…j-1]** as we move in both the source and target string.

- **('ABA', 'ABC') —> ('ABC', 'ABC') == ('AB', 'AB')** (using case 2)

  - It is basically the same as case 2, where the last two characters match, and we move in both the source and target string, except it costs an edit operation.

Defining Minimum Edit Distance(Levenshtein)

- Initialization

```
D(i,0) = i
D(0,j) = j
```

- Recurrence Relation:

```
For each  i = 1…M
      For each  j = 1…N
```

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

- Termination:

```
D(N,M) is distance
```

Dynamic Programming for Minimum Edit Distance

- We will compute D(n,m) **bottom up**, combining solutions to subproblems.
- Compute **base cases** first:
  - D(i,0) = i
    - a source substring of length i and an empty target string requires i deletes.
  - D(0,j) = j
    - a target substring of length j and an empty source string requires j inserts.
- Having computed D(i,j) for small i, j we then compute larger D(i,j) based on previously computed smaller values.
- The value of D(i, j) is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(\mathit{source}[i]) \\ D[i,j-1] + \text{ins-cost}(\mathit{target}[j]) \\ D[i-1,j-1] + \text{sub-cost}(\mathit{source}[i], \mathit{target}[j]) \end{cases}$$

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

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

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

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

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

The Final Edit Distance Table

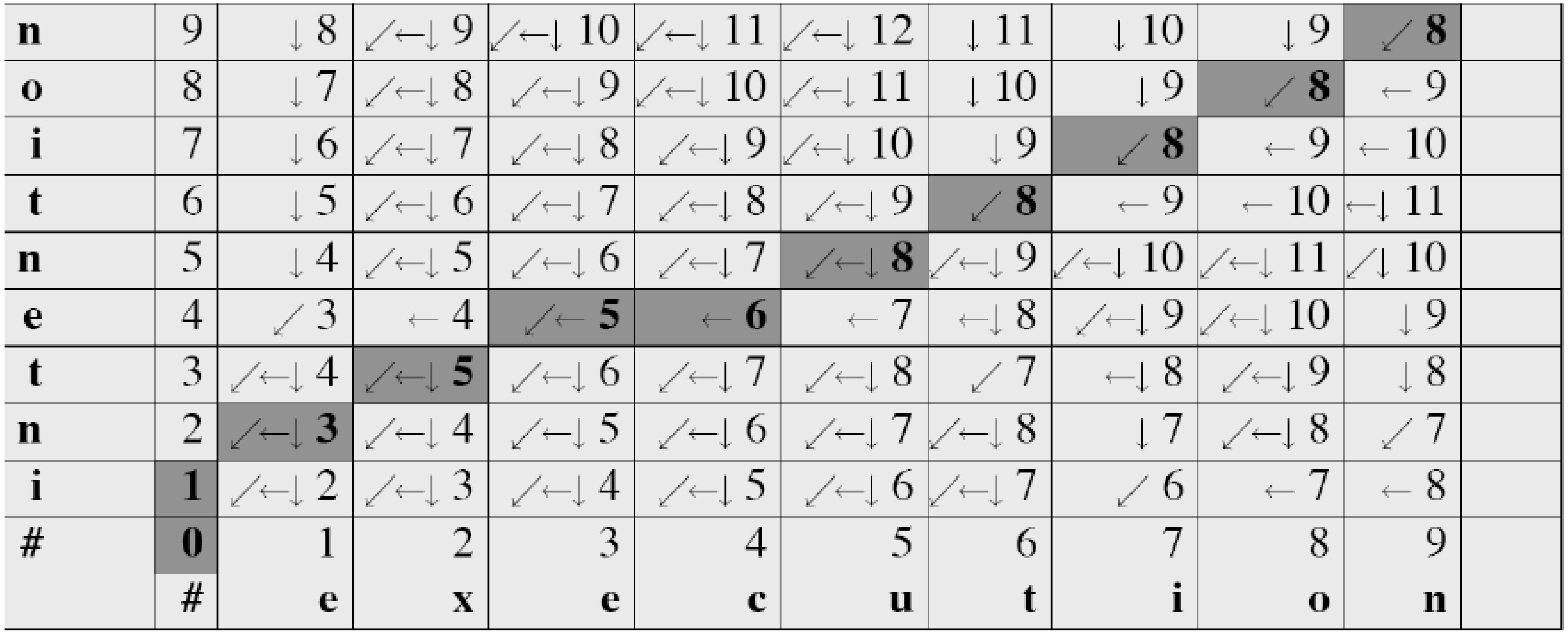
$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

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

Backtrace for computing alignments

- Edit distance isn’t sufficient
  - We often need to **align** each character of the two strings to each other
- We do this by keeping a “backtrace”
- Every time we enter a cell, remember where we came from
- When we reach the end,
  - Trace back the path from the upper right corner to read off the alignment

Backtrace for computing alignments



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**

Backtrace for computing alignments

- Base conditions: Termination:

  `D(i,0) = i` `D(0,j) = j` `D(N,M) is distance`

- Recurrence Relation:

```
For each  i = 1…M
    For each  j = 1…N
```

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{deletion} \\ D(i,j-1) + 1 & \text{insertion} \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} & \text{substitution} \end{cases}$$

$$ptr(i,j) = \begin{cases} \text{LEFT} & \text{insertion} \\ \text{DOWN} & \text{deletion} \\ \text{DIAG} & \text{substitution} \end{cases}$$

Str1=abcdef
str2=azced

f	6	5	6	5	4	5
e	5	4	5	4	3	4
d	4	3	4	3	4	4
c	3	2	3	2	4	5
b	2	1	2	3	4	5
a	1	0	1	2	3	4
#	0	1	2	3	4	5
	#	a	z	c	e	d

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  $n \leftarrow$ LENGTH(source)
  $m \leftarrow$ LENGTH(target)
  Create a distance matrix distance[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
      $D[0,0] = 0$
      for each row i from 1 to n do
         $D[i,0] \leftarrow D[i\text{-}1,0]$ + del-cost(source[i])
      for each column j from 1 to m do
         $D[0,j] \leftarrow D[0,j\text{-}1]$ + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
      for each column j from 1 to m do
         $D[i,j] \leftarrow$ MIN( $D[i-1,j]$ + del-cost(source[i]),
                        $D[i-1,j-1]$ + sub-cost(source[i], target[j]),
                        $D[i,j-1]$ + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Noisy Channel Model

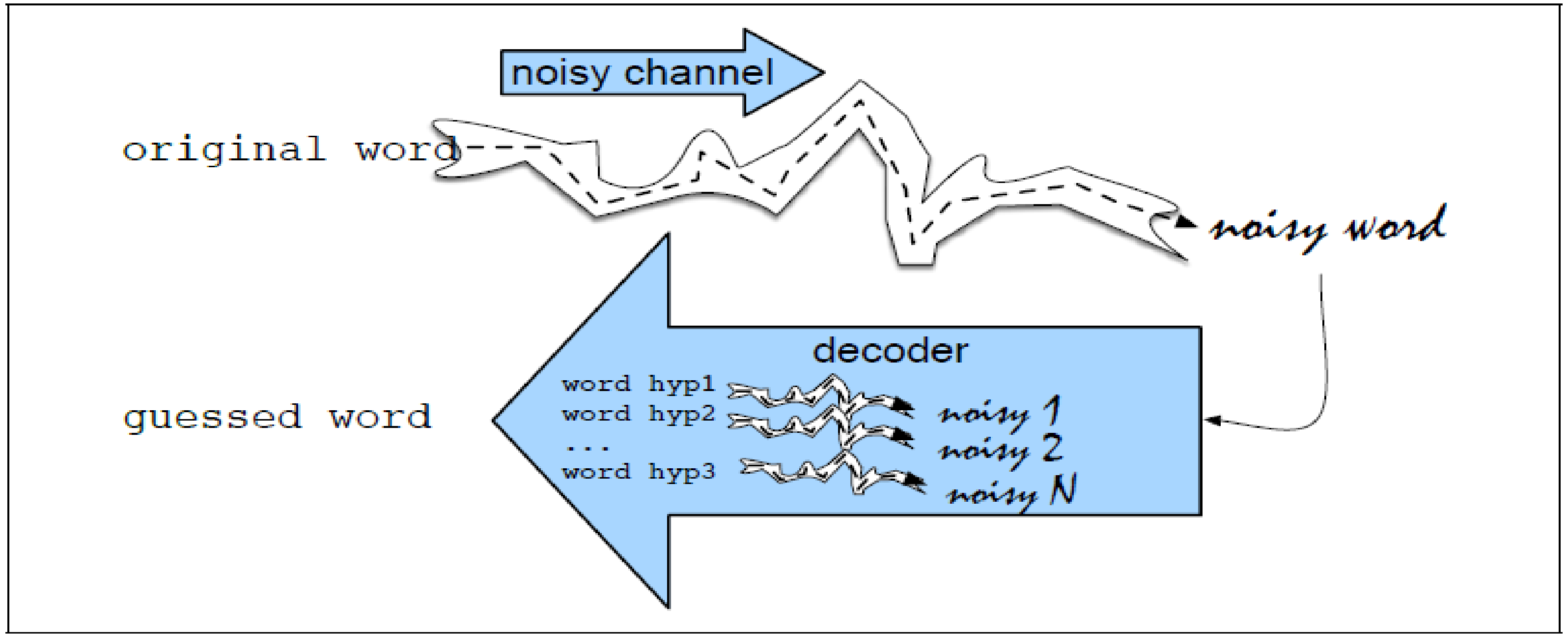
- The **noisy channel model** is a framework that computers use to check spelling, question answering, recognize speech, and perform machine translation.

- It aims to determine the correct word if you type its misspelled version or mispronounce it.

- The noisy channel model can correct several typing mistakes, including missing letters (changing “leter” to “letter”), accidental letter additions (replacing “misstake” with “mistake”), swapped letters (changing “recieve” to “receive”), and replacing incorrect letters (replacing “fimite” with “finite”).

- The noisy channel model was applied to the spelling correction task by researchers at AT&T Bell Laboratories (Kernighan et al. 1990, Church and Gale 1991) and IBM Watson Research (Mays et al., 1991)

Figure : Noisy Channel Model

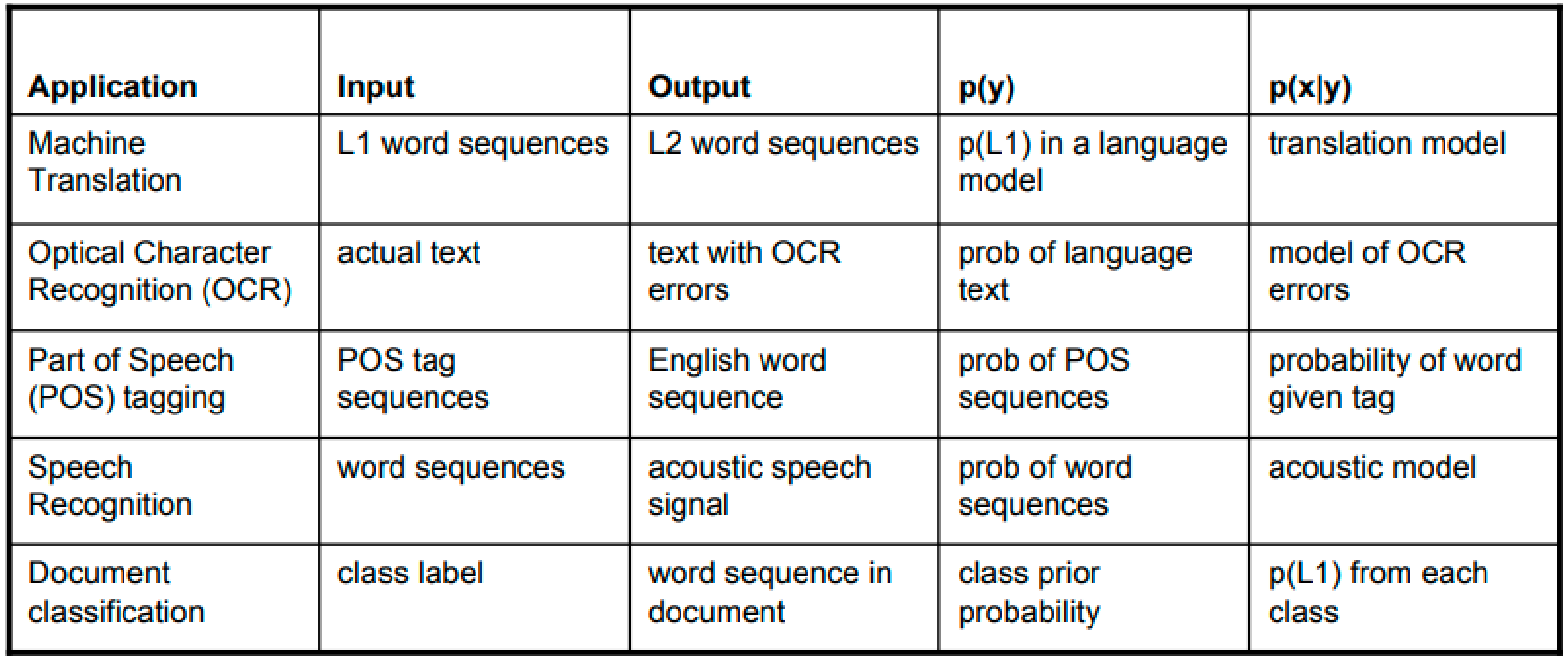


- In the noisy channel model, we imagine that the surface form we see is actually a "distorted" form of an original word passed through a noisy channel.
- Language is generated and passed through a noisy channel.
- Resulting noisy data are received
- Goal is to recover the original data from the noisy data.
- The **decoder passes each hypothesis** through a model of this channel and picks the word that best matches the surface noisy word.

- The intuition of the noisy channel model is to treat the misspelled word as, if a correctly spelled word had been “distorted” by being passed through a noisy communication channel.

- This channel introduces “noise” in the form of substitutions or other changes to the letters, making it hard to recognize the “true” word

- **Goal**: To build a model of the channel. Given this model, we then find the true word by passing every word of the language through the model of the noisy channel and seeing which one comes the closest to the misspelled word.



Application	**Input**	**Output**	**p(y)**	**p(x|y)**
Machine Translation	L1 word sequences	L2 word sequences	p(L1) in a language model	translation model
Optical Character Recognition (OCR)	actual text	text with OCR errors	prob of language text	model of OCR errors
Part of Speech (POS) tagging	POS tag sequences	English word sequence	prob of POS sequences	probability of word given tag
Speech Recognition	word sequences	acoustic speech signal	prob of word sequences	acoustic model
Document classification	class label	word sequence in document	class prior probability	p(L1) from each class

Using Bayesian Model in Noisy Channel:

- This noisy channel model is a kind of **Bayesian inference**.
- We see an observation **x (a misspelled word)** and our job is to find the word **w** that generated this misspelled word
- Out of all possible words in the **vocabulary V** we want to find the word **w** such that **P(w/x)** is highest.
- We use the hat notation ˆ to mean "our estimate of the correct word".

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(w \mid x) \qquad 5.1$$

- The function **argmax$_x$ f (x)** means "the **x** such that **f (x)** is maximized".
- Equation 5.1 thus means, that out of all words in the vocabulary, we want the particular word that maximizes the right-hand side **P(w|x)**.

- The channel has a probabilistic interpretation, whereby you assume that the original pristine version has some ***probability (prior),*** and the addition of specific noise has some ***probability (conditional).***

- These probabilities can be used to find the most likely original version given the actually observed signal.

Bayesian Classification

- The intuition of Bayesian classification is to use Bayes' rule to transform Eq. 5.1 into a set of other probabilities
- Bayes' rule is presented in Eq. 5.2; it gives us a way- to break down any conditional probability ***P(a|b)*** into three other probabilities:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)} \qquad (5.2)$$

- Substitute Eq. 5.2 into Eq. 5.1 to get Eq. 5.3:

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} \frac{P(x \mid w)P(w)}{P(x)} \qquad (5.3)$$

- We can conveniently simplify Eq. 5.3 by dropping the denominator P(x)
- Since we are choosing a potential correction word out of all words, we need to compute for each word.

$$\frac{P(x|w)P(w)}{P(x)}$$

- But ***P(x)*** doesn't change for each word; we always ask about the most likely word for the same observed error ***x***, which must have the same probability ***P(x)***.
- Thus, we can choose the word that maximizes this simpler formula

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(x \mid w)P(w) \qquad (5.4)$$

- To summarize, the noisy channel model says that we have some true underlying word ***w***, and we have a noisy channel that modifies the word into some possible likelihood of misspelled observed surface form.

- The likelihood (or channel model) of the noisy channel model producing any particular observation sequence **x** is modeled by **P(x|w)**

- The prior probability of a hidden word is modeled by **P(w)**

- We can compute the most probable word given that we've seen some observed misspelling **x** by multiplying the prior probability **P(w)** and the likelihood **P(x|w)** and choosing the word for which this product is greatest.

- We apply the noisy channel approach to correct non-word spelling errors by taking any word not in our spell dictionary, generating a list of candidate words, ranking them according to Eq. 5.4, and picking the highest-ranked one.

- We can modify Eq. 5.4 to refer to this list of candidate words instead of the full vocabulary V as follows:

$$\hat{w} = \operatorname*{argmax}_{w \in C} \overbrace{P(x|w)}^{\text{channel model}} \overbrace{P(w)}^{\text{prior}} \tag{5.5}$$

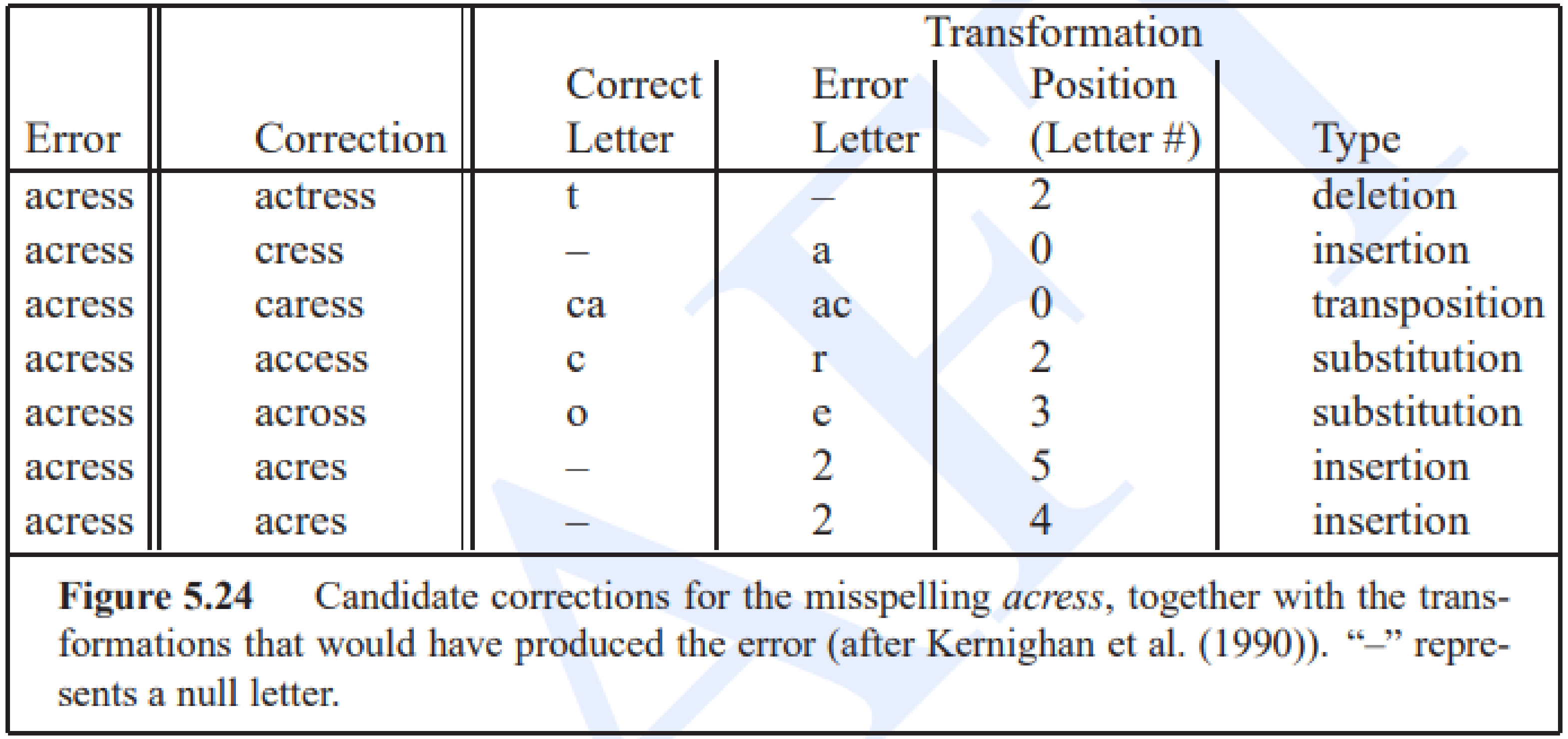
Non--word spelling error example

- Two stages of Algorithm are:
    - Propose candidate corrections by finding words that have a similar spelling to the input word.
    - Score the candidates

- The **first stage of the algorithm proposes candidate corrections** by finding words that have a similar spelling to the input word.
- Analysis of spelling error data has shown that the majority of spelling errors consist of a ***single-letter change***. Hence, make the simplifying assumption that these candidates have an edit distance of 1 from the error word

- To find this list of candidates minimum edit distance algorithm is used. It is extended so that in addition to insertions, deletions, and substitutions, we’ll add a fourth type of edit, **transpositions**, in which two letters are swapped
- The version of edit distance with transposition is called **Damerau-Levenshtein edit distance.**

- Applying all such single transformations to ***acress*** yields the list of candidate words:

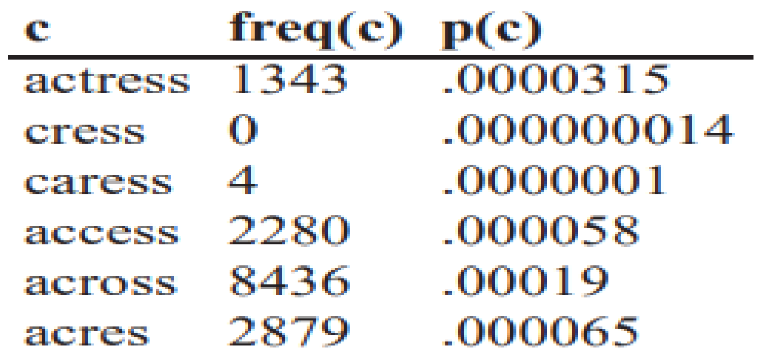


		Transformation			
Error	Correction	Correct Letter	Error Letter	Position (Letter #)	Type
acress	actress	t	–	2	deletion
acress	cress	–	a	0	insertion
acress	caress	ca	ac	0	transposition
acress	access	c	r	2	substitution
acress	across	o	e	3	substitution
acress	acres	–	2	5	insertion
acress	acres	–	2	4	insertion

Figure 5.24 Candidate corrections for the misspelling *acress*, together with the transformations that would have produced the error (after Kernighan et al. (1990)). "–" represents a null letter.

- The second stage of the algorithm is to **Score the candidates.**
- Once we have a set of candidates, score each one using Eq. 5.5 requires that we compute the **prior and the channel model**
- The prior probability of each correction **P(w)** is the language model probability of the word **w** in context, which can be computed using any language model.
- So the probability of a particular correction word **w** is computed by dividing the count of **w** by the number **N** of words in the corpus

  P(w)=Count(w)/N
- For this example let's start in the following table by assuming a unigram language model
- Let's use the corpus of Kernighan et al. (1990), which is the 1988 AP newswire corpus of 44 million words. Since in this corpus the word actress occurs 1343 times out of 44 million, the word acres 2879 times, and so on, the resulting unigram prior probabilities are:



c	freq(c)	p(c)
actress	1343	.0000315
cress	0	.000000014
caress	4	.0000001
access	2280	.000058
across	8436	.00019
acres	2879	.000065

Estimate the likelihood P(x|w)

- A simple model might estimate, for example, **p(acress|across)** just using the number of times that the letter **e** was substituted for the letter **o** in some large corpus of errors.
- To compute the probability for each edit in this way we'll need a confusion matrix that contains counts of errors.
- In general, a confusion matrix lists the matrix number of times one thing was confused with another.
- Thus for example a confusion matrix will be a square matrix of size **26x26** (or more generally **|A| x |A|,** for an alphabet **A**) that represents the number of times one letter was incorrectly used instead of another.
- The cell labeled **[*t, s*]** in an insertion confusion matrix would give the count of times that ***t*** was inserted after ***s***.
- Four confusion matrices will be used:

$\text{del}[x, y]$: count(xy typed as x)

$\text{ins}[x, y]$: count(x typed as xy)

$\text{sub}[x, y]$: count(x typed as y)

$\text{trans}[x, y]$: count(xy typed as yx)

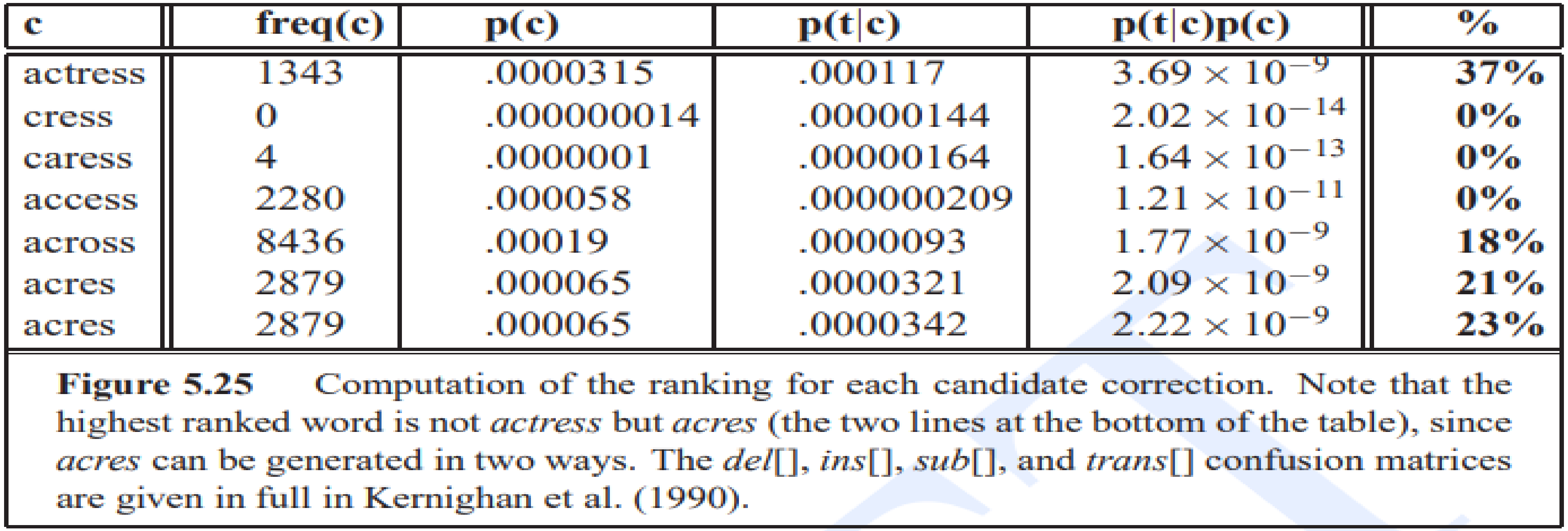
- Four confusion matrix will be used:

del[x,y]: count(**xy** typed as **x**)
ins[x,y]: count(**x** typed as **xy**)
sub[x,y]: count(**x** typed as **y**)
trans[x,y]: count(**xy** typed as **yx**)

- del[x,y] contains the number of times in the training set that the characters xy in the correct word were typed as x.
- ins[x,y] contains the number of times in the training set that the character x in the correct word was typed as xy.
- sub[x,y] the number of times that x was typed as y.
- trans[x,y] the number of times that xy was typed as yx.

- Once we have the confusion matrices, we can estimate **P(x|w)** as follows where (**w**$_i$ is the **i**th character of the correct word **w**) and **x**$_i$ is the **i**th character of the typo **x**:

$$P(x|w) = \begin{cases} \dfrac{\text{del}[x_{i-1}, w_i]}{\text{count}[x_{i-1} w_i]}, \text{ if deletion} \\ \dfrac{\text{ins}[x_{i-1}, w_i]}{\text{count}[w_{i-1}]}, \text{ if insertion} \\ \dfrac{\text{sub}[x_i, w_i]}{\text{count}[w_i]}, \text{ if substitution} \\ \dfrac{\text{trans}[w_i, w_{i+1}]}{\text{count}[w_i w_{i+1}]}, \text{ if transposition} \end{cases} \tag{5.6}$$



c	freq(c)	p(c)	p(t|c)	p(t|c)p(c)	%
actress	1343	.0000315	.000117	3.69×10^{-9}	**37%**
cress	0	.000000014	.00000144	2.02×10^{-14}	**0%**
caress	4	.0000001	.00000164	1.64×10^{-13}	**0%**
access	2280	.000058	.000000209	1.21×10^{-11}	**0%**
across	8436	.00019	.0000093	1.77×10^{-9}	**18%**
acres	2879	.000065	.0000321	2.09×10^{-9}	**21%**
acres	2879	.000065	.0000342	2.22×10^{-9}	**23%**

Figure 5.25 Computation of the ranking for each candidate correction. Note that the highest ranked word is not *actress* but *acres* (the two lines at the bottom of the table), since *acres* can be generated in two ways. The *del*[], *ins*[], *sub*[], and *trans*[] confusion matrices are given in full in Kernighan et al. (1990).

- This implementation of the Bayesian algorithm predicts acres as the correct word (at a total normalized percentage of 44%), and **actress** as the second most likely word.
- Unfortunately, the algorithm was wrong in a text string here: The writer's intention becomes clear from the context: . . .***was called a "stellar and versatile*** **acress** ***whose combination of sass and glamour has defined her. . . "***. The surrounding words make it clear that *actress* and not *acres* was the intended word.
- **This is the reason that in practice we use trigram (or larger) language models in the noisy channel model, rather than unigrams.**

END