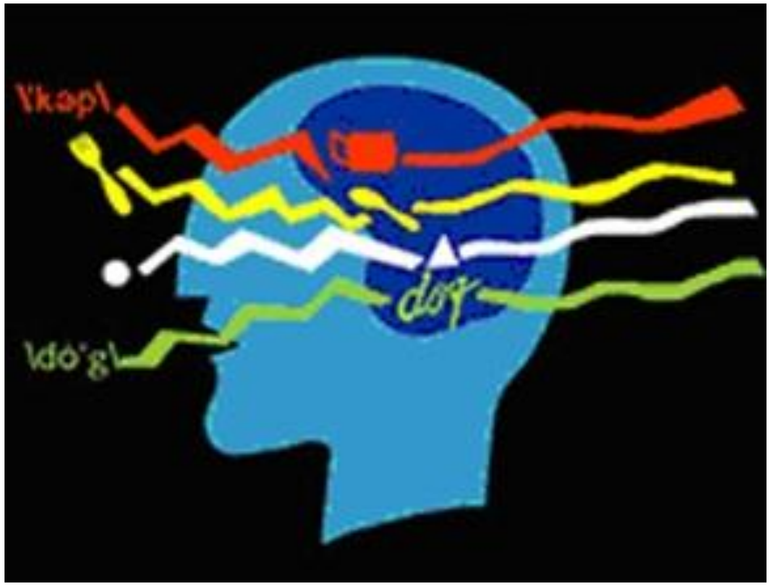
Topic 4 (Pt 2): Texts Pre-processing & String Matching: Stemming, Lemmatization, Segmentation & Edit Distance



Stemming

- A process of stripping off affixes to **find basic morphological structure** or reducing a word to its ***stem*** or ***root*** or ***base*** form
- Different variants of a term can be *conflated* to a single representative form – thus reduces the dictionary size (i.e., the no. of distinct terms)
- Can be implemented as an FST using a series of rules. Example:
  - relational relate
  - motoring motor

Root vs Stem vs Base

- ***Root***, ***stem*** and ***base*** are all terms used in the literature to designate that part of a word that remains when all affixes have been removed
  - **root**
    - a structure/form which is not further analysable when all inflectional and derivational affixes have been removed. E.g: un-**touch**-able, **ktb**(Arabic)
  - **stem**
    - concerned only when dealing with inflectional morphology. E.g: **untouchable**-s, **box**-es
  - **base**
    - any structure/form/morpheme to which affixes of any kind can be added, thus either a root or a stem can be considered as a 'base'

Porter Algorithm (Porter, 1980)

- A simple and efficient algorithm for stemming or stripping off affixes
- No lexicon (i.e., dictionary) is needed - a lexicon-free FST stemmer
- Based on a series of cascaded/nested rewrite rules such as the following:

1) **ATIONAL □ *ATE*** (e.g., rel**ational** □ rel**ate**)
2) **ING □ *ϵ*** if stem contains a vowel (e.g., motor**ing** □ motor*ϵ* or motor)

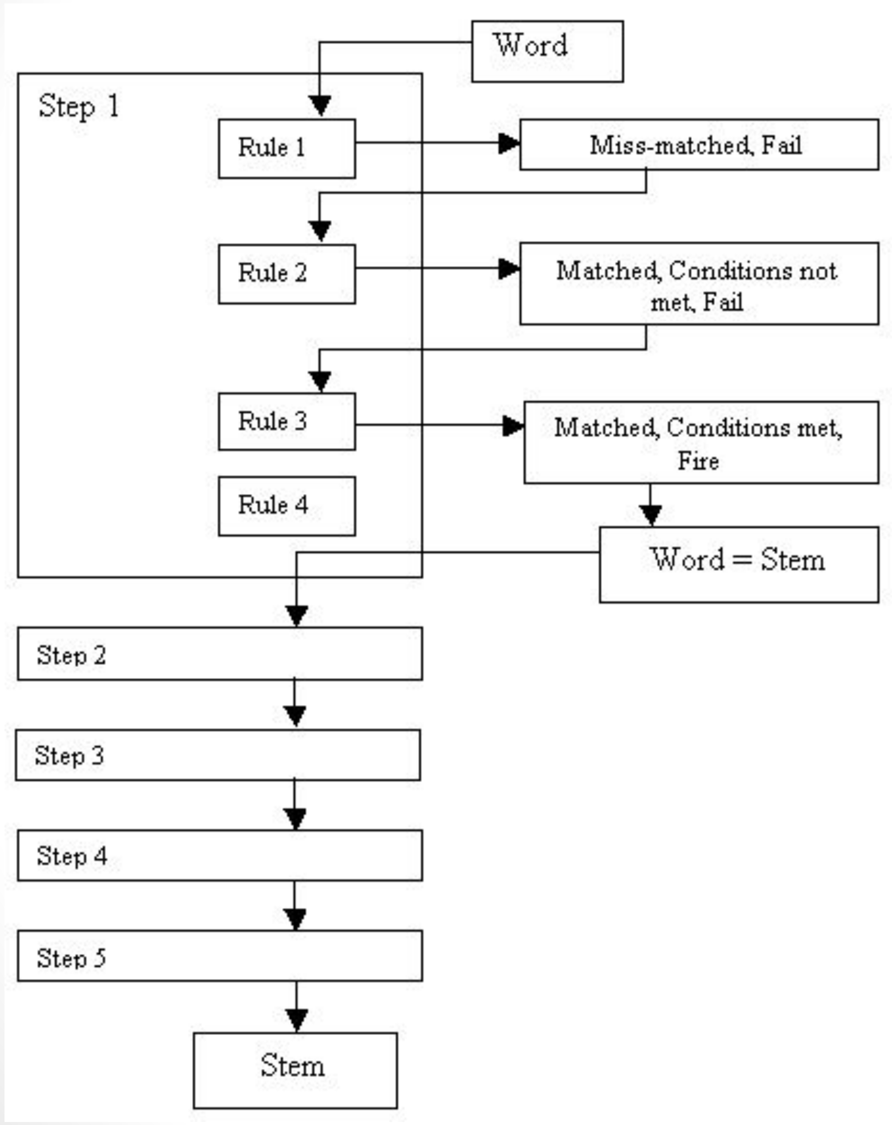
Porter Algorithm (Porter, 1980)

- More cascaded rewrite rules :

  3) ***ING stem + e*** if stem is a **short word** (depends on definition)

  - making make (mak + e)
  - mutating mutate (mutat + e)
    - If Rule 2) is applied instead of 3), then mutating mutat**ϵ** (error)

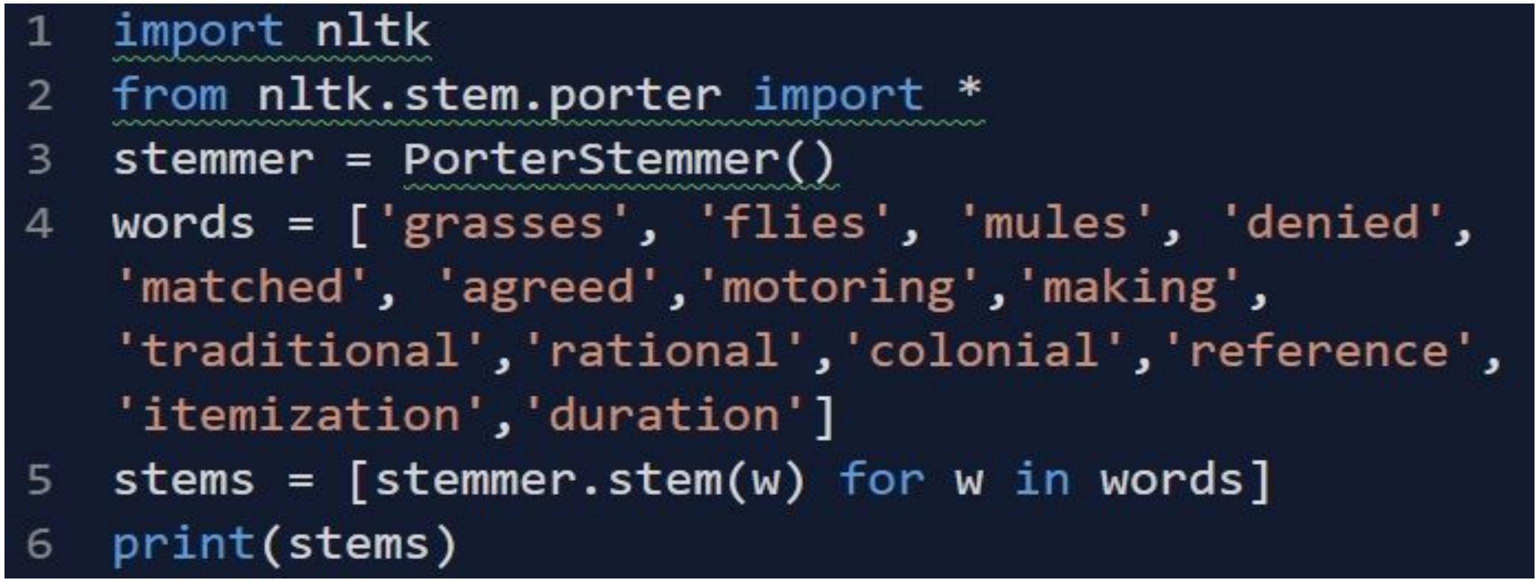
Rules in Porter Stemmer



- A simple and efficient algorithm for stemming or stripping off affixes
- No lexicon (i.e., dictionary) is needed - a lexicon-free FST stemmer
- Based on a series of cascaded rewrite rules

Stemming with NLTK

- Create a Porter stemmer



```
import nltk
from nltk.stem.porter import *
stemmer = PorterStemmer()
words = ['grasses', 'flies', 'mules', 'denied',
'matched', 'agreed','motoring','making',
'traditional','rational','colonial','reference',
'itemization','duration']
stems = [stemmer.stem(w) for w in words]
print(stems)
```

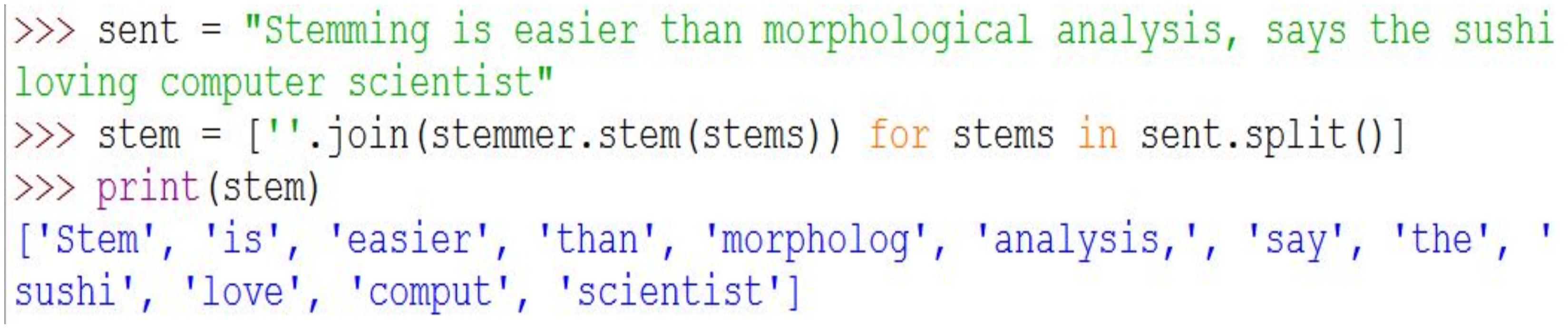
Errors in Stemming I

- **Commission**
  - Erroneously include affix when it should not have been: false positive
- **Omission**
  - Erroneously exclude affix when it should not have been : false negative

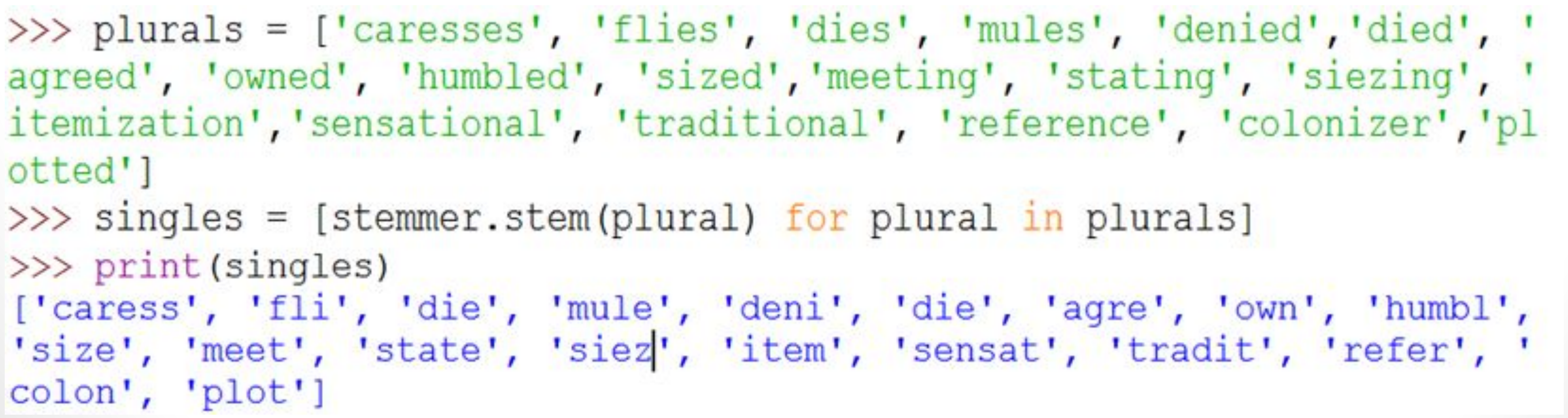
Commission errors	Omission errors
doing -> doe (**do**)	organization -> organ (**organize**)
generalization -> generic (**general**)	matrices -> matric **(matrice)**
numerical -> numerous (**numeric**)	noisy -> noisi **(noise)**
policy -> police (**policy**)	urgency -> urgenc **(urgent)**
European -> European **(Europe)**	

Commission and Omission Errors with Porter Stemmer

- Which words are stemmed with commission and omission errors?



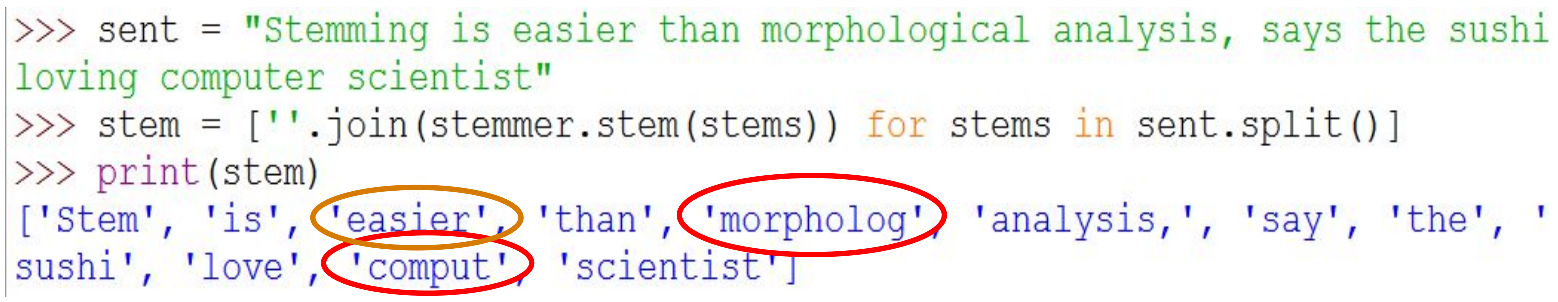
```
>>> sent = "Stemming is easier than morphological analysis, says the sushi
loving computer scientist"
>>> stem = [''.join(stemmer.stem(stems)) for stems in sent.split()]
>>> print(stem)
['Stem', 'is', 'easier', 'than', 'morpholog', 'analysis,', 'say', 'the', '
sushi', 'love', 'comput', 'scientist']
```



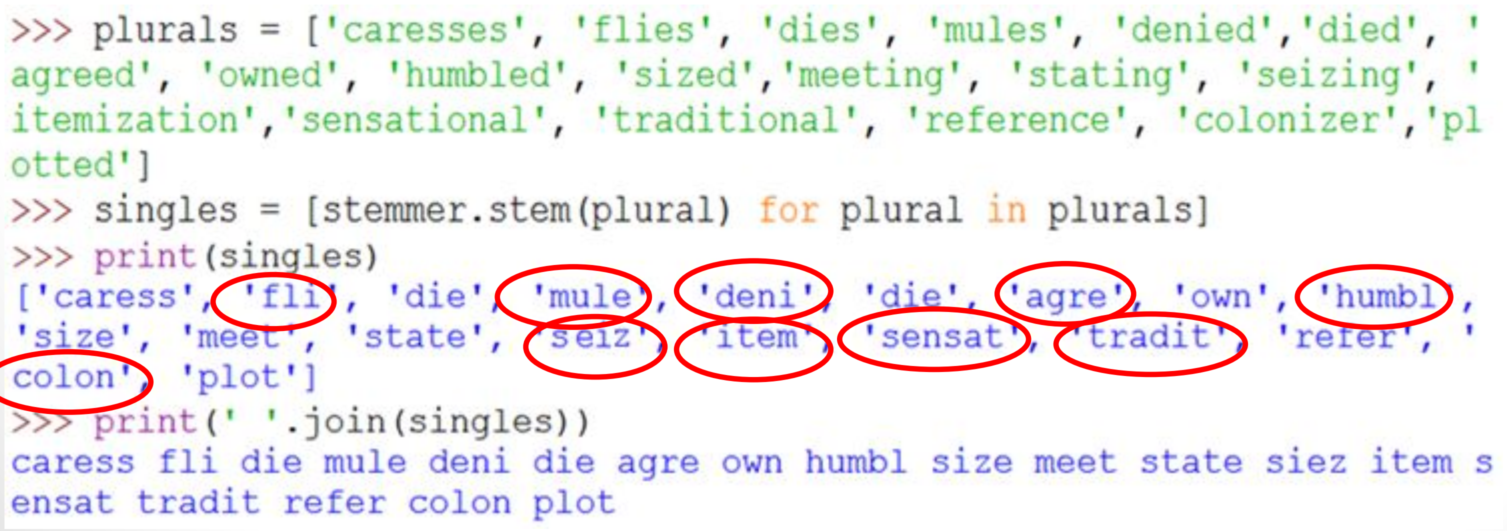
```
>>> plurals = ['caresses', 'flies', 'dies', 'mules', 'denied','died', '
agreed', 'owned', 'humbled', 'sized','meeting', 'stating', 'siezing', '
itemization','sensational', 'traditional', 'reference', 'colonizer','pl
otted']
>>> singles = [stemmer.stem(plural) for plural in plurals]
>>> print(singles)
['caress', 'fli', 'die', 'mule', 'deni', 'die', 'agre', 'own', 'humbl',
'size', 'meet', 'state', 'siez', 'item', 'sensat', 'tradit', 'refer', '
colon', 'plot']
```

Commission and Omission Errors with Porter Stemmer

- Which words are stemmed with commission or omission errors?



```
>>> sent = "Stemming is easier than morphological analysis, says the sushi
loving computer scientist"
>>> stem = [''.join(stemmer.stem(stems)) for stems in sent.split()]
>>> print(stem)
['Stem', 'is', 'easier', 'than', 'morpholog', 'analysis,', 'say', 'the', '
sushi', 'love', 'comput', 'scientist']
```



```
>>> plurals = ['caresses', 'flies', 'dies', 'mules', 'denied','died', '
agreed', 'owned', 'humbled', 'sized','meeting', 'stating', 'seizing', '
itemization','sensational', 'traditional', 'reference', 'colonizer','pl
otted']
>>> singles = [stemmer.stem(plural) for plural in plurals]
>>> print(singles)
['caress', 'fli', 'die', 'mule', 'deni', 'die', 'agre', 'own', 'humbl',
'size', 'meet', 'state', 'seiz', 'item', 'sensat', 'tradit', 'refer', '
colon', 'plot']
>>> print(' '.join(singles))
caress fli die mule deni die agre own humbl size meet state siez item s
ensat tradit refer colon plot
```

Errors in Stemming II

- **Understemming**
  - Two separate words that should be stemmed to the same root, but are not : false negative
- **Overstemming**
  - Two separate words that are stemmed to the same root, but should not have been : false positive

Understemming errors	Overstemming errors
dividing, divided -> divide division, divisor -> divise	dividing, divided -> divide divine, divination -> divide (divine)
alumnus -> alumnu alumnae -> alumna	university -> univers (university) universal, universe -> univers
adheres -> adhere adhesion -> adhes	numerous -> numer (number) numerical -> numer (numeric)

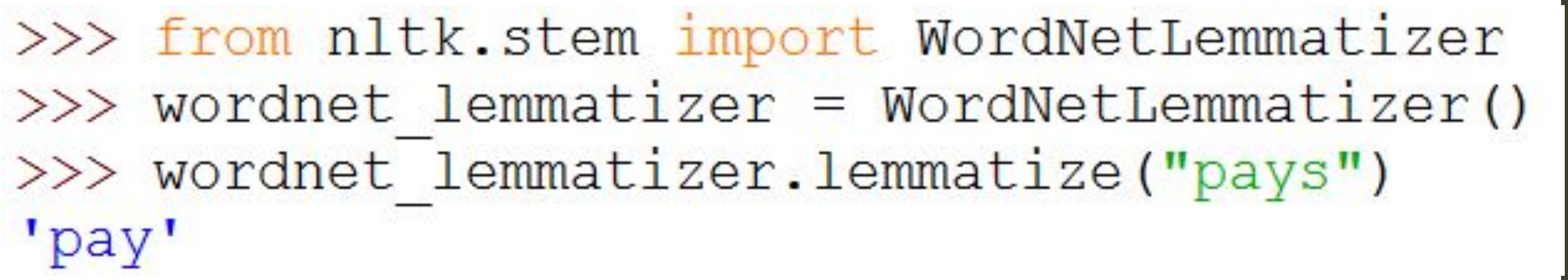
Lemmatization

- A *lemma* is the canonical or dictionary form of a set of related words.
  - ***pay*** – is the *lemma* for ***paying***, ***paid*** and ***pays***
- A *lemma* usually, but <u>not necessarily</u> resembles the words it is related to:
  - ***be*** – is the *lemma* of ***is***, ***was*** and ***am***
- Unlike *stemming*, *lemmatisation* not only tries to group related words together, but also group words by their *word sense* or *meaning*.

Lemmatization (cnt…)

- The same word may represent two different meanings. Example:
  - **wake** – means **“to wake up”** or **“a funeral”**
- *Lemmatization* requires the <u>understanding of context</u>, thus is a more complicated and expensive process as compared to stemming

Lemmatization with NLTK



```
>>> from nltk.stem import WordNetLemmatizer
>>> wordnet_lemmatizer = WordNetLemmatizer()
>>> wordnet_lemmatizer.lemmatize("pays")
'pay'
```

Word Segmentation

- The process of **segmenting/tokenizing text into words**
- Common languages like English using Latin alphabet easily separate words by spaces:

  Mr John said that …
- What about words ending with special characters?

  cents. said, positive.” crazy? google.com
- Segmenting purely on white spaces is not enough, has to address errors by treating punctuations (i.e., stop words) as **word boundary**.

Word Segmentation

- Languages with special characters such as Chinese, Japanese and Thai cannot be easily separated. Example:
  - English sentence :   Enter the room
  - Chinese sentence: **进入房间** (Jìnrù fángjiān)
- Segmentation may also involve tokenizing multiple expressions
  - Example : houseboat -> house boat

Sentence Segmentation

- A crucial step in text processing, **segmenting/tokenizing text (i.e., paragraphs) into individual sentences**.
- Usually based on punctuations commonly used to mark sentence boundaries (i.e., . ? , !)
- The function of the (.) however is ambiguous as it can serve various purposes

  RM56.56   Mr.   Co.m.p.h
- A tokenization algorithm based on machine learning can be used for segmentation
- Minimal approach through regular expression

Example: Chinese Word Segmentation

- Chinese words are composed of characters known as ***hanzi***
- Each character represents a single morpheme and is pronounceable as a single syllable
- An average Chinese word is about 2.4 characters long
- A greedy search algorithm known as maximum matching are commonly used to segment Chinese words with the help of a list of dictionary containing all possible Chinese words

Chinese Word Segmentation (cont…)

Algorithm:

1. Start at beginning of string
2. Repeat
    a. Advance pointer past each character in word
    b. Advance one character at a time
3. Until word match is found

Analogy of algorithm (according to English dictionary)

- English phrase (with spaces removed):

  the table down there

  thetabledownthere

- Maximum word match : ***1) theta 2) bled 3) own 4) there***
- Final output: ***theta bled own here*** (there are also other possibilities)

String Matching: Measuring distance between words

- How similar are two strings?
  - **Spelling correction:**
    - If user typed "giraffe", which of the following is the closest?
      - graf
      - graft
      - gaffe
      - giraff
  - **Computational Biology:**
    - Align two Sequences of *nucleotides*

AGGCTATCACCTGACCTCCAGGCCGATGCCC

TAGCTATCACGACCGCGGTCGATTTGCCCGAC

-AGGCTATCACCTGACCTCCAGGCCGA -- TGCCC---

TAG –CTATCAC- - GACCGC--GGT-CGATTTGCCCGAC

Spelling Error Detection

- Detection and correction of spelling errors is an integral part of modern word processors and search engines
- Three different spelling problems:
  - **Non-word** error detection: spelling errors resulting in non-words (e.g., giraffe graffe)
    - Can use FST
  - **Isolated word** error detection: looking at individual error correction
  - **Context-dependent** error correction : using context to detect errors

Spelling Error Detection

- Correcting spelling error requires searching through all possible words, and pick the most likely source
- Choose among potential sources using a **distance metric** between the source and the surface error
- Can apply probabilistic and non-probabilistic methods to find the closest spelling
- Example of non-probabilistic method is **Minimum Edit Distance**

Minimum Edit Distance

- Deciding which of two words is closer to some third word in spelling is a special case of the general problem of **string distance**.
- The distance between two strings is a measure of **how alike two strings are** to each other
- The minimum edit distance between two strings is the **minimum number of editing operations** needed to transform one string into another
    - insertion
    - deletion
    - substitution

Minimum Edit Distance (cont…)

- Useful in NLP applications like machine translation, information extraction, speech recognition
- Example: two strings and their **alignment**:

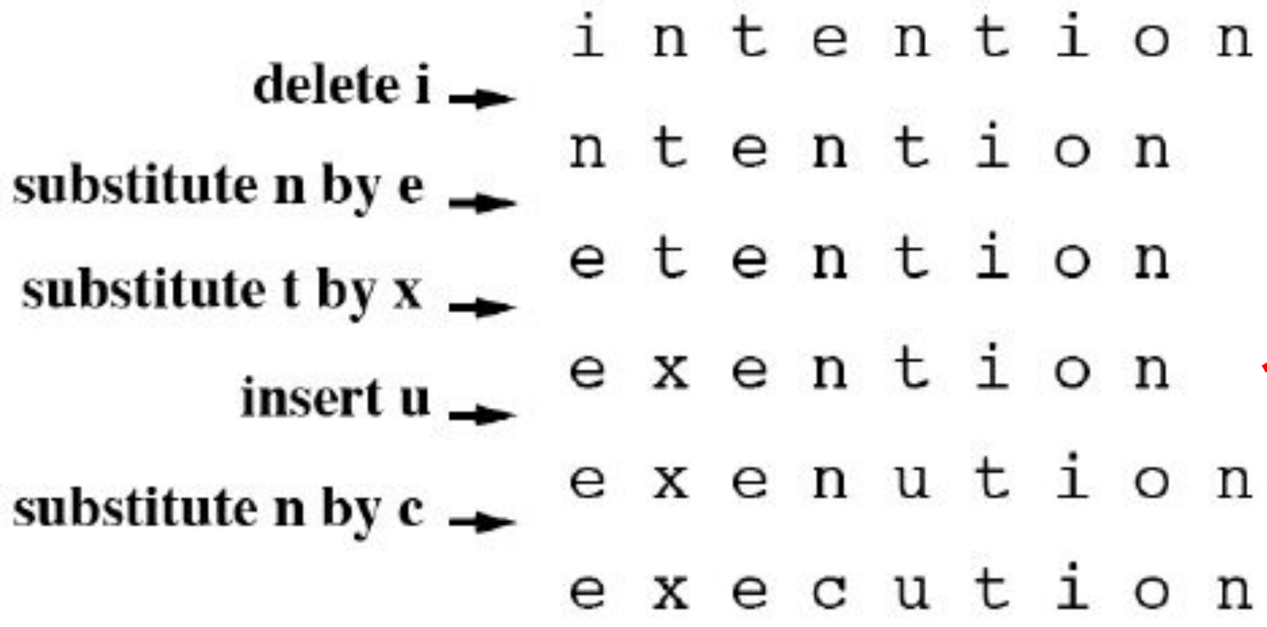
```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

1 + 1 + 1 + 1 + 1

1 + 2 + 2 + 1 + 2

- If each operation has a cost of 1
  - Distance between these = **5**
- If substitutions cost is 2 (Levenshtein)
  - Distance between them = **8**

Minimum Edit Distance (cont…)



i n t e n t i o n

delete i →

n t e n t i o n

substitute n by e →

e t e n t i o n

substitute t by x →

e x e n t i o n

insert u →

e x e n u t i o n

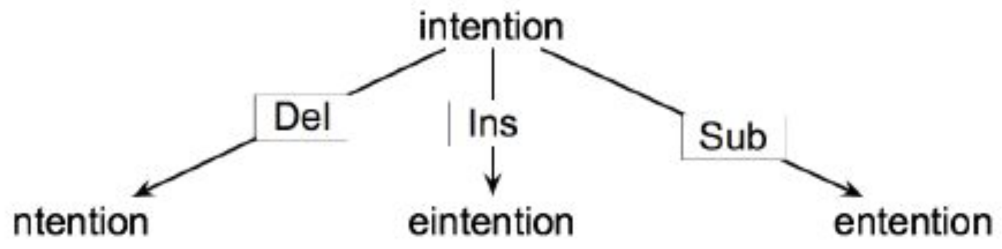
substitute n by c →

e x e c u t i o n

is there a shorter path???

Finding Minimum Edit Distance (cont…)

- Searching for a path (sequence of edits) from the start string to the final string:
  - **Initial state**: the word we're transforming
  - **Operators**: insert, delete, substitute
  - **Goal state**: the word we're trying to get to
  - **Path cost**: what we want to minimize, the number of edits
- But search space is huge!!!



Definition of Minimum Edit Distance (Levenshtein Algorithm)

- Given two strings, X of length *m* and Y of length *n*
- We define D(*i*,*j*) 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*)**
- **Example:**
  - X = i m p o s s i b l e (m = 10)
  - Y = r e s p o n s i b l e (n = 11)

Dynamic Programming for Minimum Edit Distance

- Dynamic programming applies a table-driven method (tabular computation) of D(*n*,*m*)
- Solve problems by **combining solutions to sub-problems**
- Bottom-up
  - Compute D(*i*,*j*) for small *i, j*
  - Compute larger D(*i, j*) based on previously computed smaller values
  - Compute D(*i*,*j*) for all *i(0 < i < m)* and *j*(0 < *j* < n*)*

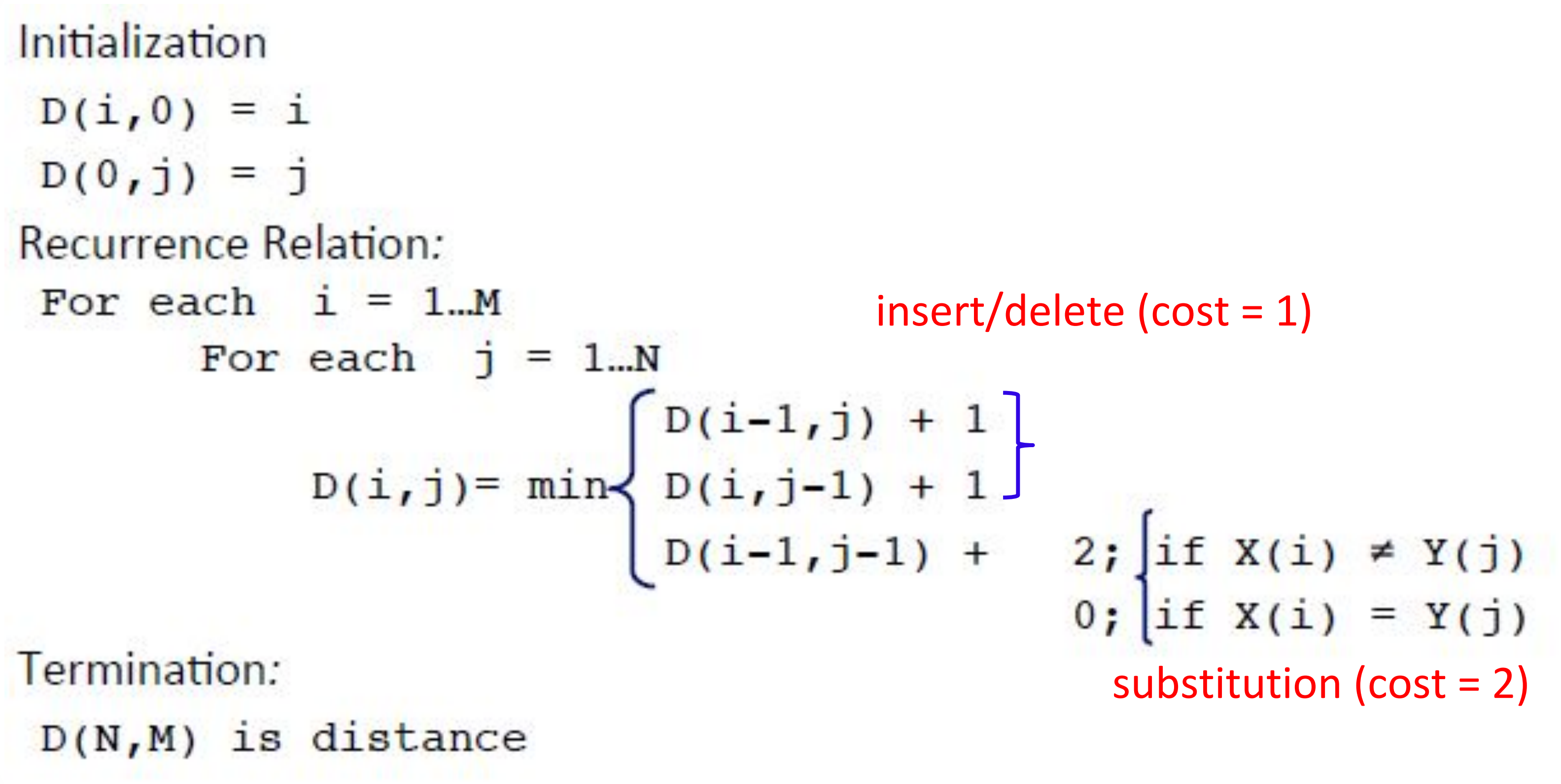
Minimum Edit Distance Algorithm

```
function MIN-EDIT-DIST(target, source) returns min-dist

  m <- LENGTH(target)
  n <- LENGTH(source)
  Create a distance matrix dist[m+1,n+1]
  Initialize 0th row and col to be distance from empty
  string
    dist[0,0]= 0
    for each column i from 1 to m do
      dist[i,0] <- dist[i-1,0]+ins-cost(target[i])
    for each row j from 1 to n
      dist[0,j] <- dist[0,j-1]+del-cost(source[j])

  for each column i from 1 to m do
    for each row j from 1 to n do
      dist[i,j] <- MIN(dist[i-1,j] + ins-cost(target$_{i-1}$),
                 dist[i-1,j-1] + subst-cost(source$_{j-1}$,
           target$_{i-1}$),dist[i,j-1] +
           del-cost(source$_{j-1}$))
  return dist[m,n]
```

Minimum Edit Distance Algorithm



Initialization

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

Recurrence Relation:

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

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

insert/delete (cost = 1)

substitution (cost = 2)

Termination:

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

Example 1: Levenshtein Algorithm

Levenshtein distance between “brake” and “break” (m = n)

	#	B	R	E	A	K
#	0	1	2	3	4	5
B	1	0	1	2	3	4
R	2	1	0	1	2	3
A	3	2	1	2	1	2
K	4	3	2	3	2	3
E	5	4	3	2	3	$\boxed{2}$

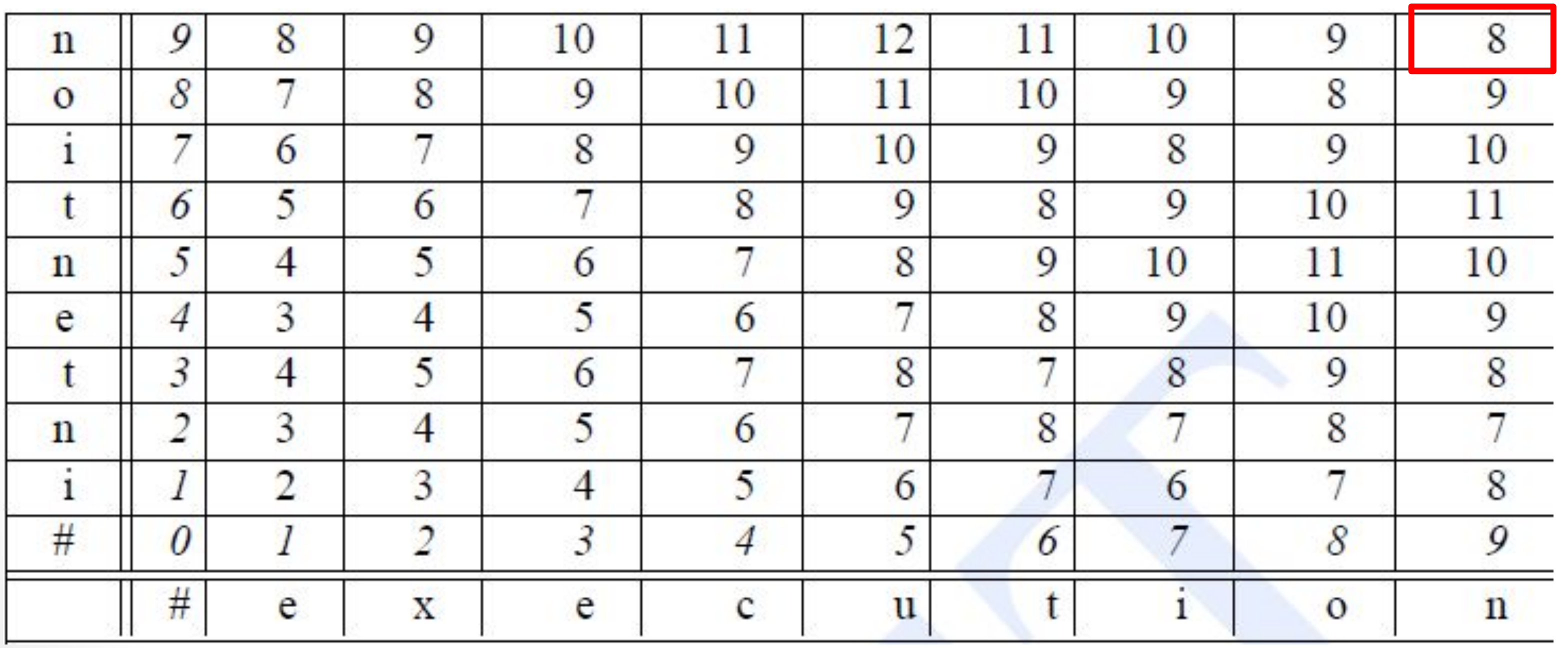
Example 2: Levenshtein Algorithm

Levenshtein distance between “HONDA” and “HYUNDAI” (m < n)

	#	H	Y	U	N	D	A	I
#	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	2	3	4	5	6	7
N	3	2	3	4	3	4	5	6
D	4	3	4	5	4	3	4	5
A	5	4	5	6	5	4	3	**4**

Example 3: Levenshtein Algorithm

Levenshtein distance between “intention” and “execution” (m = n)



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

Local Alignment Problem

- Given two strings:

  $\mathbf{x = x_1 \ldots . x_M ,}$

  $\mathbf{y = y_1 \ldots . y_N}$

- Find substrings x', y' whose similarity (optimal local alignment value) is maximum

  x = aaaacccccggggtta

  y = ttcccgggaaccaacc

Local Alignment Example (Smith-Waterman)

X = ATCAT

Y = ATTATC

Let:

m = 1(1 point for match)

d = 1(--1 point for del/ins/sub)

		A	T	T	A	T	C
	0	0	0	0	0	0	0
A	0	1	0	0	1	0	0
T	0	0	2	1	0	2	0
C	0	0	1	1	0	1	3
A	0	1	0	0	2	1	2
T	0	0	2	0	1	3	2

Local Alignment Example

X = **ATCAT**

Y = **ATTATC**

		A	T	T	A	T	C
	0	0	0	0	0	0	0
A	0	1	0	0	1	0	0
T	0	0	2	1	0	2	0
C	0	0	1	1	0	1	3
A	0	1	0	0	2	1	2
T	0	0	2	0	1	$\boxed{3}$	2

Local Alignment Example

X = ATCAT

Y = ATTATC

		A	T	T	A	T	C
	0	0	0	0	0	0	0
A	0	1	0	0	1	0	0
T	0	0	2	1	0	2	0
C	0	0	1	1	0	1	3
A	0	1	0	0	2	1	2
T	0	0	2	0	1	3	2

Edit Distance Exercise 6 (Levenshtein)

1) Using the distance.py, compute the distance between the following words:
   - stemming vs stamping
   - imputation vs importation
   - stability vs solidity

2) For each of the word pairs above, show your manual calculation of the operations costs for changing from one word to another using the **Levenshtein algorithm**

* Work in pairs