

Natural Language Processing (AI615)

Prepared by Dr. Ashraf Elsayed

Lecture 2





مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY

Basic Text Processing

Lecture 2

Prepared by
Dr. Ashraf Elsayed

www.midocean.ae

The objectives of this lecture is to:

- Understand Basic Test Processing
- Define and explain the basic concepts of Byte Pair Encoding
- Understand Word Normalization, Lemmatization and Stemming
- Understand the Minimum Edit Distance

Basic Text Processing

Basic Text Processing

Byte Pair Encoding

Another option for text tokenization

Instead of

- white-space segmentation
- single-character segmentation

Use the data to tell us how to tokenize.

Subword tokenization (because tokens can be parts of words as well as whole words)

Subword tokenization

- Three common algorithms:
 - **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
 - **Unigram language modeling tokenization** (Kudo, 2018)
 - **WordPiece** (Schuster and Nakajima, 2012)
- All have 2 parts:
 - A token **learner** that takes a raw training corpus and induces a vocabulary (a set of tokens).
 - A token **segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary

Byte Pair Encoding (BPE) token learner

Let vocabulary be the set of all individual characters

= {A, B, C, D,..., a, b, c, d....}

- Repeat:
 - Choose the two symbols that are most frequently adjacent in the training corpus (say 'A', 'B')
 - Add a new merged symbol 'AB' to the vocabulary
 - Replace every adjacent 'A' 'B' in the corpus with 'AB'.
- Until k merges have been done.

BPE token learner algorithm

```
function BYTE-PAIR ENCODING(strings  $C$ , number of merges  $k$ ) returns vocab  $V$   
  
   $V \leftarrow$  all unique characters in  $C$            # initial set of tokens is characters  
  for  $i = 1$  to  $k$  do                           # merge tokens til  $k$  times  
     $t_L, t_R \leftarrow$  Most frequent pair of adjacent tokens in  $C$   
     $t_{NEW} \leftarrow t_L + t_R$                  # make new token by concatenating  
     $V \leftarrow V + t_{NEW}$                        # update the vocabulary  
    Replace each occurrence of  $t_L, t_R$  in  $C$  with  $t_{NEW}$  # and update the corpus  
  return  $V$ 
```

Byte Pair Encoding (BPE) Addendum

Most subword algorithms are run inside space-separated tokens.

So we commonly first add a special end-of-word symbol '___' before space in training corpus

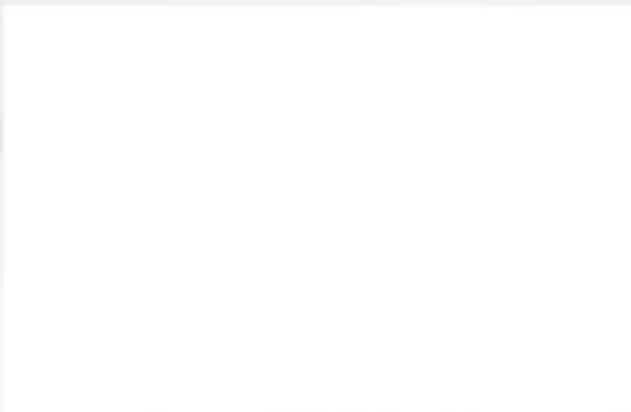
Next, separate into letters.

BPE token learner

Original (very fascinating 🤨) corpus:

low low low low low lowest lowest newer newer newer newer newer
newer wider wider wider new new

Add end-of-word tokens, resulting in this vocabulary:



vocabulary

—, d, e, i, l, n, o, r, s, t, w

BPE token learner

corpus

```
5   l o w _  
2   l o w e s t _  
6   n e w e r _  
3   w i d e r _  
2   n e w _
```

vocabulary

```
_, d, e, i, l, n, o, r, s, t, w
```

Merge **e r** to **er**

corpus

```
5   l o w _  
2   l o w e s t _  
6   n e w e r _  
3   w i d e r _  
2   n e w _
```

vocabulary

```
_, d, e, i, l, n, o, r, s, t, w, er
```

BPE

corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

Merge **er _** to **er_**

corpus

5 l o w _
2 l o w e s t _
6 n e w e r_
3 w i d e r_
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, e r

vocabulary

, d, e, i, l, n, o, r, s, t, w, e r, e r

BPE

corpus

5 l o w _
2 l o w e s t _
6 n e w er_
3 w i d er_
2 n e w _

vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er

Merge **n** **e** to **ne**

corpus

5 l o w _
2 l o w e s t _
6 ne w er_
3 w i d er_
2 ne w _

vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er, ne

BPE

The next merges are:

Merge	Current Vocabulary
(ne, w)	—, d, e, i, l, n, o, r, s, t, w, er, er—, ne, new
(l, o)	—, d, e, i, l, n, o, r, s, t, w, er, er—, ne, new, lo
(lo, w)	—, d, e, i, l, n, o, r, s, t, w, er, er—, ne, new, lo, low
(new, er—)	—, d, e, i, l, n, o, r, s, t, w, er, er—, ne, new, lo, low, newer—
(low, —)	—, d, e, i, l, n, o, r, s, t, w, er, er—, ne, new, lo, low, newer—, low—

BPE token **segmenter** algorithm

On the test data, run each merge learned from the training data:

- Greedily
- In the order we learned them
- (test frequencies don't play a role)

So: merge every **e r** to **er**, then merge **er _** to **er_**, etc.

- Result:
 - Test set "n e w e r _" would be tokenized as a full word
 - Test set "l o w e r _" would be two tokens: "low er_"

Properties of BPE tokens

Usually include frequent words

And frequent subwords

- Which are often morphemes like *-est* or *-er*

A **morpheme** is the smallest meaning-bearing unit of a language

- *unlikeliest* has 3 morphemes *un-*, *likely*, and *-est*

Basic Text Processing

Word Normalization,
Lemmatization and Stemming

Word Normalization

- Putting words/tokens in a standard format
 - U.S.A. or USA
 - uhhuh or uh-huh
 - Fed or fed
 - am, is, be, are

Case folding

- Applications like IR: reduce all letters to lower case
 - Since users tend to use lower case
 - Possible exception: upper case in mid-sentence?
 - e.g., **General Motors**
 - **Fed** vs. **fed**
 - **SAIL** vs. **sail**
- For sentiment analysis, MT, Information extraction
 - Case is helpful (**US** versus **us** is important)

Lemmatization

Represent all words as their lemma, their shared root
= dictionary headword form:

- *am, are, is* → *be*
- *car, cars, car's, cars'* → *car*
- Spanish **quiero** ('I want'), **quieres** ('you want')
→ **querer** 'want'
- *He is reading detective stories*
→ *He be read detective story*

Lemmatization is done by Morphological Parsing

- Morphemes:
 - The small meaningful units that make up words
 - **Stems**: The core meaning-bearing units
 - **Affixes**: Parts that adhere to stems, often with grammatical functions
- Morphological Parsers:
 - Ex: Parse *cats* into two morphemes *cat* and *s*

Stemming (simplified form of lemmatization)

- Reduce terms to stems, chopping off affixes crudely

This was not the map we found in Billy Bones's chest, but an accurate copy, complete in all things-names and heights and soundings-with the single exception of the red crosses and the written notes.



Thi wa not the map we found in Billi Bone s chest but an accur copi complet in all thing name and height and sound with the singl except of the red cross and the written note

.

Porter Stemmer

- Based on a series of rewrite rules run in series
 - A cascade, in which output of each pass fed to next pass
- Some sample rules:

ATIONAL → ATE (e.g., relational → relate)

ING → ε if stem contains vowel (e.g., motoring → motor)

SSES → SS (e.g., grasses → grass)

Dealing with complex morphology is necessary for many languages

- e.g., the Turkish word:
- **Uygarlastiramadiklarimizdanmissinizcasina**
- `(behaving) as if you are among those whom we could not civilize'
- **Uygar** `civilized' + **las** `become'
 - + **tir** `cause' + **ama** `not able'
 - + **dik** `past' + **lar** `plural'
 - + **imiz** `p1pl' + **dan** `abl'
 - + **mis** `past' + **siniz** `2pl' + **casina** `as if'

Complex
Morphology

Definitely require a richer
morphological parsing
algorithms

Sentence Segmentation

!, ? mostly unambiguous but **period** “.” is very ambiguous

- Sentence boundary
- Abbreviations like Inc. or Dr.
- Numbers like .02% or 4.3

Common algorithm: Tokenize first: use rules or ML to classify a period as either (a) part of the word or (b) a sentence-boundary.

- An abbreviation dictionary can help

Sentence segmentation can then often be done by rules based on this tokenization.

Computational Linguistics

Minimum Edit Distance

Definition of Minimum Edit Distance

How similar are two strings?

- Spell correction

- The user typed “graffe”

Which is closest?

- graf
 - graft
 - grail
 - giraffe

- Computational Biology

- Align two sequences of nucleotides

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC  
TAGCTATCACGACCGCGGTCGATTGCCCCGAC
```

- Resulting alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---  
TAG-CTATCAC--GACCGC--GGTCGATTGCCCCGAC
```

- Also for Machine Translation, Information Extraction, Speech Recognition

Edit Distance

- The minimum edit distance between two strings
- Is the minimum number of editing operations
 - Insertion
 - Deletion
 - Substitution
- Needed to transform one into the other

Minimum Edit Distance

- Two strings and their **alignment**:

I	N	T	E	*	N	T	I	O	N
*	E	X	E	C	U	T	I	O	N

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 them is 8

Alignment in Computational Biology

- Given a sequence of bases

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGGTCGATTGCCCCGAC

- An alignment:

–AGGCTATCACCTGACCTCCAGGCCGA–TGCCC–
TAG–CTATCAC–GACCGC–GGTCGATTGCCCCGAC

- Given two sequences, align each letter to a letter or gap

Other uses of Edit Distance in NLP

- Evaluating Machine Translation and speech recognition

R Spokesman confirms senior government adviser was shot

H Spokesman said the senior adviser was shot dead

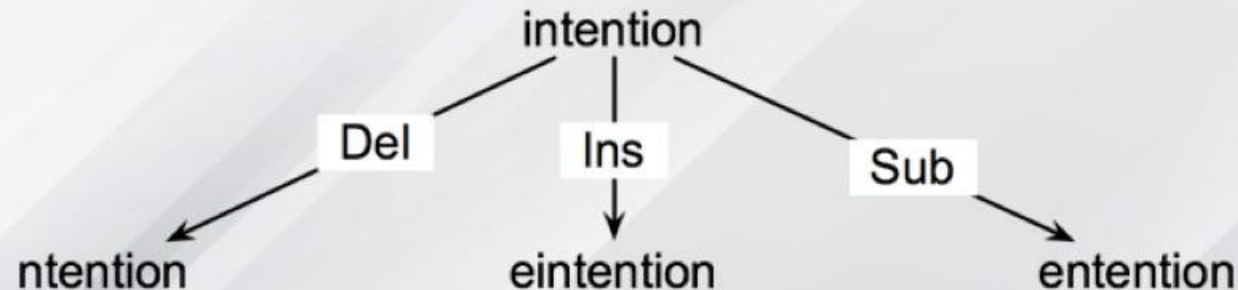
S I D I

- Named Entity Extraction and Entity Co-reference

- IBM Inc. announced today
- IBM profits
- Stanford President John Hennessy announced yesterday
- for Stanford University President John Hennessy

How to find the Min Edit Distance?

- 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



Minimum Edit as Search

- But the space of all edit sequences is huge!
 - We can't afford to navigate naïvely
 - Lots of distinct paths ends up at the same state.
 - We don't have to keep track of all of them
 - Just the shortest path to each of those revisited states.

Defining Min Edit Distance

- For two strings
 - X of length n
 - Y of length m
- We define $D(i,j)$
 - 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 $D(n,m)$

Minimum Edit Distance

Computing Minimum Edit Distance

Dynamic Programming for Minimum Edit Distance

- **Dynamic programming:** A tabular computation of $D(n,m)$
- Solving problems by combining solutions to subproblems.
- Bottom-up
 - We compute $D(i,j)$ for small i,j
 - And compute larger $D(i,j)$ based on previously computed smaller values
 - i.e., compute $D(i,j)$ for all i ($0 < i < n$) and j ($0 < j < m$)

Defining Min Edit Distance (Levenshtein)

- Initialization

$$D(i, 0) = i$$

$$D(0, j) = j$$

- Recurrence Relation:

For each $i = 1 \dots M$

For each $j = 1 \dots N$

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

- Termination:

$D(N, M)$ is distance

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
Src/Tar	#	E	X	E	C	U	T	I	O	N

The Edit Distance Table

i

Del-cost(source[i])

Ins-cost(target[j])

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

N	9										
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	
Src/Tar	#	E	X	E	C	U	T	I	O	N	

j

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

Edit Distance

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	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

Minimum Edit Distance

Backtrace for Computing Alignments

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

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

Edit Distance

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

MinEdit with 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 Minimum Edit Distance

- Base conditions:

$$D(i, 0) = i$$

$$D(0, j) = j$$

Termination:

$D(N, M)$ is distance

- Recurrence Relation:

For each $i = 1 \dots M$

For each $j = 1 \dots N$

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

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

The Distance Matrix



Every non-decreasing path

from $(0,0)$ to (M, N)

corresponds to
an alignment
of the two sequences

An optimal alignment is composed
of optimal subalignments

Result of Backtrace

- Two strings and their **alignment**:

I	N	T	E	*	N	T	I	O	N
*	E	X	E	C	U	T	I	O	N

Performance

- Time:

$O(nm)$

- Space:

$O(nm)$

- Backtrace

$O(n+m)$



مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY



Thank You

