Text Preprocessing in Natural Language Processing

Transforming Raw Text into Actionable Data

Outline

- What is text preprocessing?
- Why is it important?
- Preprocessing techniques

Quick recap of previous lecture

Why text preprocessing is important?

- **Noise Reduction:** Remove irrelevant characters (punctuation, HTML tags).
- **Consistency:** Lowercasing, standardizing formats (e.g., dates).
- **Efficiency:** Smaller vocabulary size = faster model training.
- **Accuracy:** Improves NLP task performance (e.g., sentiment analysis).

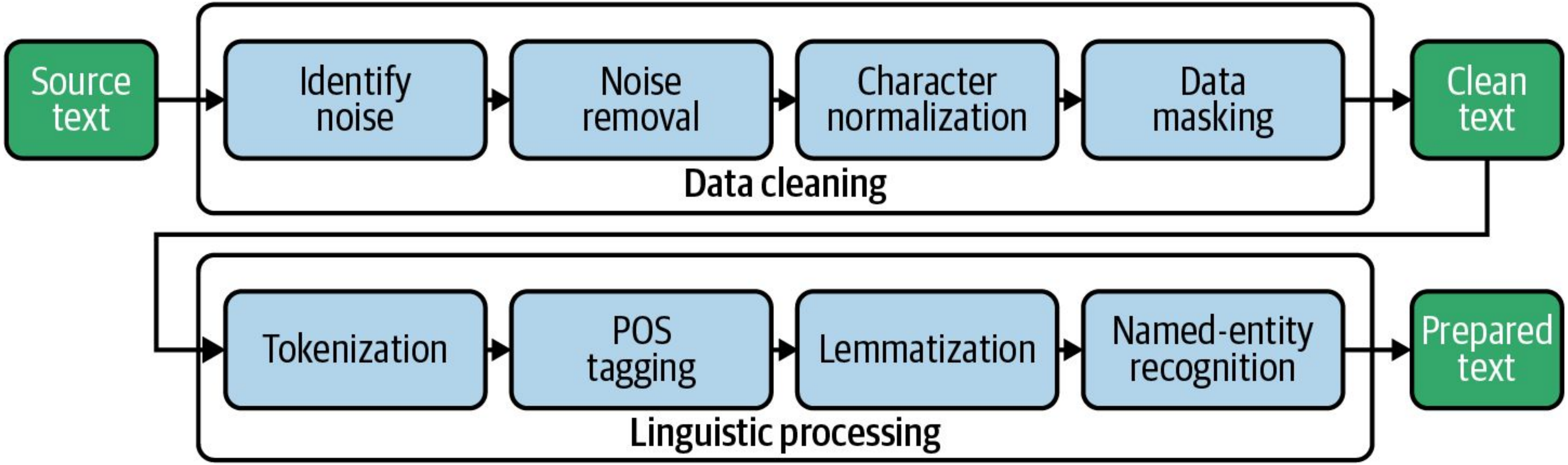
Challenges

- **Ambiguity:** “Apple” (fruit vs. company).
- **Language Differences:** Morphology in Arabic vs. English.
- **Resource Limits:** Lemmatization requires heavy dictionaries.

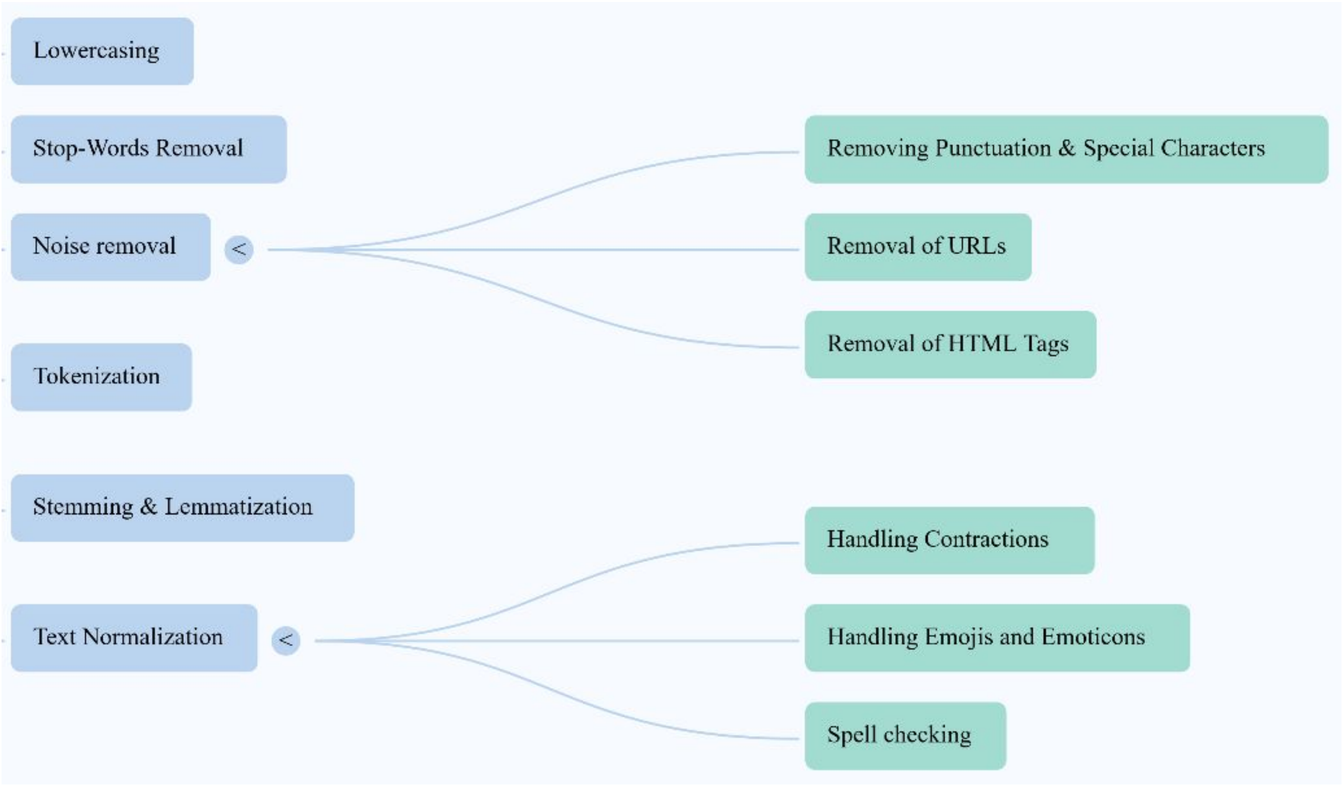
Domain-Specific Preprocessing

- **Social Media:** Emoji handling, slang normalization.
- **Scientific Texts:** Retain equations/symbols.
- **Medical Texts:** Protect sensitive terms (e.g., patient names).

Text preprocessing stages



Common preprocessing steps:



Tokenization

Tokenization is the process of breaking down text into smaller chunks such as words or subwords.

Importance

Foundation for most NLP tasks and models

Tools: NLTK and spaCy toolkits.

Word tokenization

- Splits text into individual words. For example:
- Input: "Tokenization is fun!"
- Output: ["Tokenization", "is", "fun", "!"]

Whitespace tokenization

- Splits text based on whitespace. For example:
- Input: "Tokenization is fun!"
- Output: ["Tokenization", "is", "fun!"]

Subword tokenization

- Breaks words into smaller subword units
- Input: "unbelievable"
- Output: ["un", "believ", "able"]

Character tokenization

- Splits text into individual characters. For example:
- Input: "Token"
- Output: ["T", "o", "k", "e", "n"]

Sentence segmentation

- Splits text into sentences. For example:
- Input: "**Tokenization is fun. Let's learn more!**"
- Output: ["**Tokenization is fun.**", "**Let's learn more!**"]

Regex-based tokenization

- Uses regular expressions to define custom tokenization rules.
- For example, splitting text based on punctuation or specific patterns.

Tokenization

Sentence: Segmentation and Word/Subword: Tokenization

- **Subword Tokenization:** Handle rare/compound words (e.g., BPE in GPT).

Tokenization is the process of breaking down text into smaller chunks such as words or subwords.

Importance

Foundation for most NLP tasks and models

Tools: NLTK and spaCy toolkits..

Word tokenization

- Splits text into individual words. For example:
- Input: "Tokenization is fun!"
- Output: ["Tokenization", "is", "fun", "!"]

Whitespace tokenization

- Splits text based on whitespace. For example:
- Input: "Tokenization is fun!"
- Output: ["Tokenization", "is", "fun!"]

Subword tokenization

- Breaks words into smaller subword units
- Input: "unbelievable"
- Output: ["un", "believ", "able"]

Character tokenization

- Splits text into individual characters. For example:
- Input: "Token"
- Output: ["T", "o", "k", "e", "n"]

Sentence segmentation

- Splits text into sentences. For example:
- Input: "**Tokenization is fun. Let's learn more!**"
- Output: ["**Tokenization is fun.**", "**Let's learn more!**"]

Regex-based tokenization

- Uses regular expressions to define custom tokenization rules.
- For example, splitting text based on punctuation or specific patterns.

Classes of tokenization algorithms

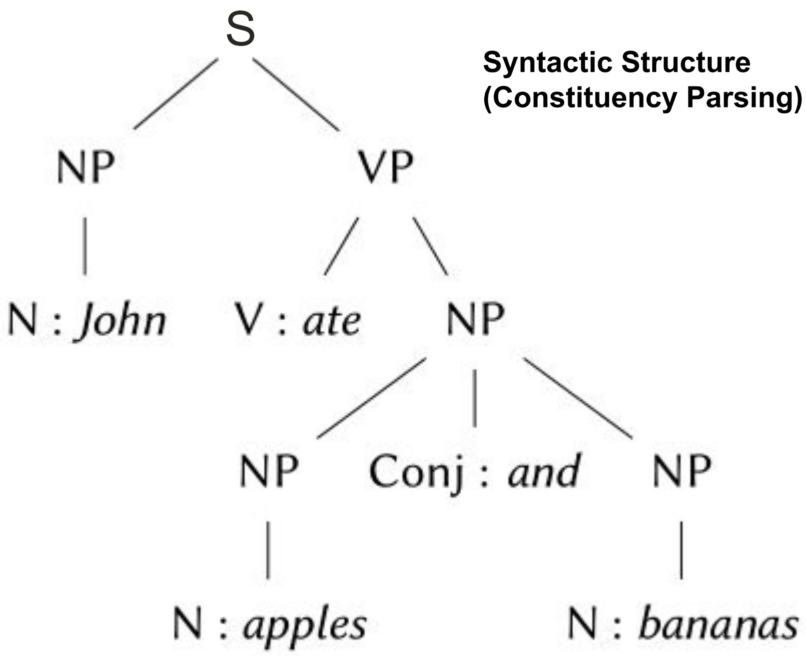
- Top-down tokenization
    - Rule-based (Penn Treebank Tokenizer)
- Bottom-up tokenization
    - Data-driven (Byte-Pair Encoding)

- Speed matters:
    - Tokenization is performed on every input before other language processing steps.
    - Large datasets and real-time systems need tokenizers that are highly optimized for speed.
    - Word tokenizers generally use deterministic (rule-based or algorithmic) logic for reliability and efficiency

Top-down Tokenization

- Involves splitting text based on predefined rules or patterns.
- Focuses on higher-level linguistic units, typically words and sentences.
- Penn Treebank Tokenizer

Syntactic Structure (Constituency Parsing)



Dependency Parsing

```
        ate
      /     \
  John   apple
             \
             and
               \
             bananas
```

Top-down Tokenization

- Involves splitting text based on predefined rules or patterns.
- Focuses on higher-level linguistic units, typically words and sentences.
- Penn Treebank Tokenizer
  - Splits most punctuation from words.
    - Handles contractions and possessives (ownership)
    - e.g., "children’s" → "children" + "’s"; "can’t" → "ca" + "n’t".

```
Input:   "The San Francisco-based restaurant," they said,
         "doesn’t charge $10".
Output:  "␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣
         "␣does␣n’t␣charge␣$␣10␣"␣.
```

Top-down Tokenization

- Involves splitting text based on predefined rules or patterns.
- Focuses on higher-level linguistic units, typically words and sentences.
- Penn Treebank Tokenizer
  - Splits most punctuation from words.
    - Handles contractions and possessives (ownership)
    - e.g., "children's" → "children" + "'s"; "can't" → "ca" + "n't".
  - Advantages:
    - Fast and deterministic.
    - Designed for structured, well-formed text (e.g., newswire).
  - Deterministic output: Always produces the same tokenization for the same input

Bottom-up Tokenization

- Constructs tokens from lower-level units (characters or bytes), merging them based on data frequencies.
- This data-driven approach is especially useful for handling **rare words** and **diverse language** inputs. [**Finite vocabulary from training data**]
  - NLP algorithm learn some facts from one corpus (**training corpus**) and make decisions on unseen corpus (**test corpus**)
  - By splitting words into smaller, reusable pieces (***subwords***), the model can generalize.
- Most tokenization schemes have two parts: **a token learner**, and **a token segmenter**.
  - Token learner learns the subword vocabulary from the training corpus (e.g., BPE training process).
  - Token segmenter applies the learned vocabulary to segment new (test) data into tokens (possibly splitting unknown words into familiar subwords).

Subword Tokenization (Byte-Pair Encoding, BPE)

- Tokenizers automatically learn a set of tokens smaller than words (called ***subwords***), enabling flexible splitting of both known and unknown words.

Byte-Pair Encoding (BPE)

- Process:
  - Start with all characters as initial tokens.
  - Iteratively merge the most frequent adjacent token pairs in the corpus.
  - Build new tokens (subwords/words) with every merge step until ***k*** merges has been done.
    - *k = vocab_size - initial_vocab_size*
- Outcome:
  - Handles arbitrary and out-of-vocabulary words.
  - Produces a compact, efficient vocabulary for downstream models
- Used by: Modern language models like OpenAI’s GPT, Llama.

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

Vocabulary	#count
e	19
_	18
w	15
r	9
n	8
l	7
o	7
d	3
i	3
s	2
t	2

Vocabulary	#count
er	9
r_	9

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

Vocabulary	#count
e	19
_	18
w	15
r	9
n	8
l	7
o	7
d	3
i	3
s	2
t	2
er	9

Vocabulary	#count
er_	9
ne	9
lo	7
w_	7
ow	7
wer	6
wi	3
der	3
id	3
t_	2
we	2
st	2
t_	2

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_

Vocabulary	#count
e	19
_	18
w	15
r	9
n	8
l	7
o	7
d	3
i	3
s	2
t	2
er	9
er_	9

Vocabulary	#count
ne	8
w_	7
lo	7
ow	7
wer_	6
der_	3
id	3
we	2
st	2
t_	2

corpus		**vocabulary**
5	`l o w _`	`_, d, e, i, l, n, o, r, s, t, w, er, er_`
2	`l o w e s t _`	
6	`n e w er_`	
3	`w i d er_`	
2	`n e w _`	

corpus		**vocabulary**
5	`l o w _`	`_, d, e, i, l, n, o, r, s, t, w, er, er_, ne`
2	`l o w e s t _`	
6	`ne w er_`	
3	`w i d er_`	
2	`ne w _`	

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2	l o w e s t _	
6	ne w er_	
3	w i d er_	
2	ne w _	

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_

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 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

Figure 2.13 The token learner part of the BPE algorithm for taking a corpus broken up into individual characters or bytes, and learning a vocabulary by iteratively merging tokens. Figure adapted from Bostrom and Durrett (2020).

Wordpiece Tokenizer

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 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

$$\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

Exercise for you:

Token learner: Find possible subwords using Wordpiece algorithm

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

$$\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

Word normalization

Word normalization is the task of putting words or tokens in a standard format.

- Case folding [generalize text to same case (lower case)]
- Normalize [(USA, US), (uh-huh, uhhuh)]
- Morphological analysis [mapping to same root word - (Warsaw, Warszawa)]
  - Stems
  - Affixes
    - Prefix
    - Suffix

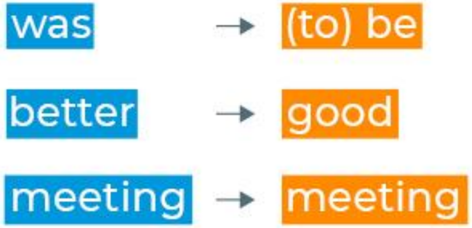
Eg.

- cats - ['cat', 's']
- fox - ['fox']

Lemmatization

Use linguistics to get valid root forms (e.g., “better” → “good”).

- Maps words to dictionary roots
- Produces valid words
- Requires POS tagging
- **The algorithm can be complex.**



Stemming

Reduce words to root form (e.g., “running” → “run”).

- Removes word suffixes aggressively
- May produce non-words
- Fast and simple

Text Normalization: Handling Contractions

Convert text to canonical form

Eg. Helloooo → Hello

Expand contractions

Convert shortened forms to full words

Standardize forms

Reduce variations in expression for consistency

Tools: Regex, custom dictionaries.

- can’t
- cannot
- won't
- will not
- isn’t
- is not

Text Normalization: Emojis, Emoticons, and Spell Checking

Handle emojis & emoticons

Convert to text descriptions or remove as needed

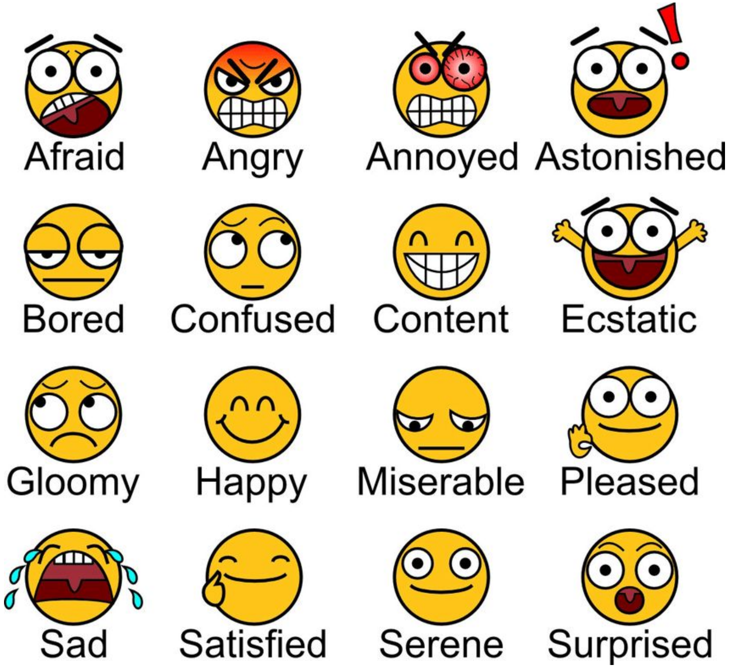
`Toolkit: demoji`

Spell checking

Correct typos to improve data quality

Use Case: Fix typos in social media/text messages.

`Toolkit: pyspellchecker`



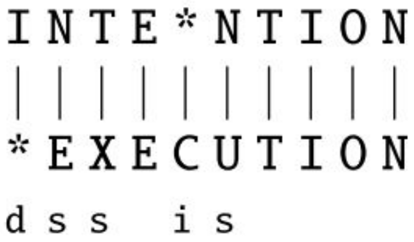
Minimum edit distance

Minimum Edit Distance is the **minimum number of operations** required to convert one string into another.

Edit Operations:

- **Insertion** (add a character)
- **Deletion** (remove a character)
- **Substitution** (replace one character with another)

The gap between **intention** and **execution**



Applications:

- Spell checking
- Machine translation evaluation
- Plagiarism detection
- Speech recognition

Why does it matter?:

- MED quantifies how “similar” two pieces of text are [critical in many NLP tasks].

Minimum edit distance

Minimum Edit Distance is the **minimum number of operations** required to convert one string into another.

Edit Operations:

- **Insertion** (add a character) [("cat" → "cart" by inserting 'r')]
- **Deletion** (remove a character) [("cart" → "cat" by deleting 'r')]
- **Substitution** (replace one character with another) [("cat" → "cut" by substituting 'a' with 'u')]

Applications:

- Spell checking [("recieve" → "receive") distance = 2]
- Machine translation evaluation [Compare translated output with a reference translation]
- Plagiarism detection [Find passages that are almost the same but with small edits.]
- Speech recognition [Align output text with the actual spoken words to score accuracy.]

Why does it matter?:

- MED quantifies how “similar” two pieces of text are [critical in many NLP tasks].

How to calculate MED?

- The space of all possible edits is enormous, so we can’t search naively.
- We could just remember the shortest path to a state each time we saw it.
- Dynamic programming - a table-driven method to solve problems by combining solutions to subproblems. (Eg. Viterbi algorithm)
- MED base cases:
  - D[i,j] = The edit distance between X[1..i] and Y[1.. j]
  - D[i,0] = i (requires i deletes)
  - D[0,j] = j (requires j deletes)

How to calculate MED?

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

Levenshtein Distance

$$D[i,j] = \min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

		0	1	2	3	4	5	6	7	8	9
	Src\Tar	**#**	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**
0	**#**	0	1	2	3	4	5	6	7	8	9
1	**i**	1	2	3	4	5	6	7	6	7	8
2	**n**	2	3	4	5	6	7	8	7	8	7
3	**t**	3	4	5	6	7	8	7	8	9	8
4	**e**	4	3	4	5	6	7	8	9	10	9
5	**n**	5	4	5	6	7	8	9	10	11	10
6	**t**	6	5	6	7	8	9	8	9	10	11
7	**i**	7	6	7	8	9	10	9	8	9	10
8	**o**	8	7	8	9	10	11	10	9	8	9
9	**n**	9	8	9	10	11	12	11	10	9	8

Figure 2.18 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.17, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions.

	#	e	x	e	c	u	t	i	o	n
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
i	↑ **1**	↖←↑ 2	↖←↑ 3	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖ 6	← 7	← 8
n	↑ 2	↖←↑ **3**	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↑ 7	↖←↑ 8	↖ 7
t	↑ 3	↖←↑ 4	↖←↑ **5**	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖ 7	←↑ 8	↖←↑ 9	↑ 8
e	↑ 4	↖ 3	← 4	↖← **5**	← **6**	← 7	←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9
n	↑ 5	↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ **8**	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖↑ 10
t	↑ 6	↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖ **8**	← 9	← 10	←↑ 11
i	↑ 7	↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9	↖ **8**	← 9	← 10
o	↑ 8	↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↑ 10	↑ 9	↖ **8**	← 9
n	↑ 9	↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖←↑ 12	↑ 11	↑ 10	↑ 9	↖ **8**

At each step:

- If you follow ↖ **(diagonal)**, you either substitute (if letters differ) or match (if same).
- If you follow ↑ **(top)**, you've deleted a character from the source.
- If you follow ← **(left)**, you've inserted a character into the source.

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

When to Skip Preprocessing?

- **Context:** Tasks needing case sensitivity (e.g., NER).
- **Models:** Modern LLMs (BERT, GPT) handle raw text better.

Impact on Model Performance

- Text preprocessing improves NLP system performance
- **Case Study:** Sentiment analysis accuracy improves by 15% after stop word removal.

Key Takeaway: Preprocessing tailors text for NLP tasks.

Wider reading

Tokenization:

- Wordpiece: https://huggingface.co/learn/llm-course/chapter6/6
- Unigram: https://huggingface.co/learn/llm-course/en/chapter6/7

References:

- Chapter 2: https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf