Natural Language Processing

Artificial Intelligence

Jacopo Parretti

I Semester 2025-2026

Indice

Parte I

String Similarity and Edit Distance

1 Minimum Edit Distance

We are going to deal with this main driving point: the definition of Minimum Edit Distance.

The question: are this 2 texts the same?

When is the case when 2 texts are the same? Of course when every single character is the same. What if we would like to understand if 2 texts are pretty close (not the same)?

Single characters in the text could be different in position.

1.1 How similar are two strings?

The fundamental question in edit distance is: how can we measure the similarity between two strings? This problem appears in many different applications across various domains of computer science and computational linguistics.

1.1.1 Spell Correction

In spell correction systems, we need to find the closest valid word to a misspelled input. For example, if the user typed "graffe", we need to determine which word in our dictionary is most similar.

Possible candidates:

- graf
- graft
- grail
- giraffe

By computing the edit distance between "graffe" and each candidate, we can identify that "giraffe" is likely the intended word (requiring only one deletion).

1.1.2 Computational Biology

In bioinformatics, sequence alignment is crucial for comparing DNA, RNA, or protein sequences. By aligning sequences, we can identify regions of similarity that may indicate functional, structural, or evolutionary relationships.

Example: Align two sequences of nucleotides:

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
```

```
TAGCTATCACGACCGCGGTCGATTTGCCCGAC
```

Resulting alignment (dashes represent insertions/deletions):

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

The alignment reveals matching regions (shown in the same positions) and differences between the sequences.

1.1.3 Other Applications

String similarity and edit distance algorithms are fundamental tools in many NLP tasks:

- **Machine Translation**: Comparing source and target language phrases, finding similar translations in translation memories
- **Information Extraction**: Matching entity names with variations (e.g., "IBM" vs "I.B.M." vs "International Business Machines")
- **Speech Recognition**: Correcting recognition errors by finding the closest valid word or phrase to the acoustic model output

1.2 Edit Distance

The **minimum edit distance** between two strings is defined as the minimum number of editing operations needed to transform one string into the other.

1.2.1 Edit Operations

There are three basic editing operations:

- **Insertion**: Add a character at any position in the string
  - Example: cat → cart (insert 'r')
- **Deletion**: Remove a character from any position in the string
  - Example: cart → cat (delete 'r')
- **Substitution**: Replace one character with another
  - Example: cat → bat (substitute 'c' with 'b')

1.2.2 Key Concept

The edit distance measures the *minimum* number of these operations required to transform one string into another. This metric provides a quantitative measure of string similarity: the smaller the edit distance, the more similar the strings are.

Important note: Each operation has a cost (typically 1), and we seek the sequence of operations that minimizes the total cost. Different variants of edit distance may assign different costs to different operations (e.g., Levenshtein distance uses uniform costs, while other variants may weight substitutions differently).

1.2.3 Example: Alignment of Two Strings

To better understand edit distance, let's examine how two strings can be aligned to show their differences. Consider the strings "INTENTION" and "EXECUTION":

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

In this alignment:

- The asterisk (*) represents a gap, indicating an insertion or deletion operation
- Vertical bars (|) connect corresponding positions between the two strings
- Characters that match are aligned vertically (E, T, I, O, N)

- Characters that differ indicate substitution operations

Operations needed to transform "INTENTION" to "EXECUTION":

1. Delete 'I' at position 1
2. Substitute 'N' with 'E' at position 2
3. Substitute 'T' with 'X' at position 3
4. Keep 'E' (match)
5. Insert 'C' at position 5
6. Substitute 'N' with 'U' at position 6
7. Keep 'T' (match)
8. Keep 'I' (match)
9. Keep 'O' (match)
10. Keep 'N' (match)

This gives us a total edit distance of **5 operations** (1 deletion + 3 substitutions + 1 insertion).

1.2.4 Cost Variants: Standard vs Levenshtein

The total distance depends on how we assign costs to each operation:

Standard Edit Distance (uniform costs):

- Each operation (insertion, deletion, substitution) costs 1
- Total distance for INTENTION $\rightarrow$ EXECUTION: **5**
- Calculation: 1 (deletion) + 3×1 (substitutions) + 1 (insertion) = 5

Levenshtein Distance (weighted substitution):

- Insertion costs 1
- Deletion costs 1
- Substitution costs 2 (considered as a deletion + insertion, hence a "double error")
- Total distance for INTENTION $\rightarrow$ EXECUTION: **8**
- Calculation: 1 (deletion) + 3×2 (substitutions) + 1 (insertion) = 8

Why the difference? In the Levenshtein variant, a substitution is viewed as conceptually equivalent to deleting a character and then inserting a different one, thus costing twice as much. This distinction is important when choosing which distance metric to use for a particular application.

1.3 Alignment in Computational Biology

In computational biology, sequence alignment is a fundamental technique for comparing DNA, RNA, or protein sequences. The goal is to identify regions of similarity and understand evolutionary relationships.

1.3.1 The Alignment Problem

Given a sequence of bases:

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC
```

```
TAGCTATCACGACCGCGGTCGATTTGCCCCGAC
```

An alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA---TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCCGAC
```

1.3.2 Alignment Objective

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

The alignment process involves:

- **Matching**: Aligning identical bases (e.g., A with A, G with G)
- **Mismatches**: Aligning different bases (substitutions)
- **Gaps**: Represented by dashes (-), indicating insertions or deletions (indels)

Key considerations:

- The alignment should maximize the number of matches
- Minimize the number of mismatches and gaps
- Gaps are penalized because insertions and deletions are relatively rare evolutionary events
- Different scoring schemes can be used: match scores, mismatch penalties, and gap penalties

This alignment problem is directly related to edit distance: finding the optimal alignment is equivalent to finding the minimum edit distance between the two sequences.

1.4 Other Uses of Edit Distance in NLP

Edit distance is a versatile tool used across many NLP applications beyond spell correction and sequence alignment.

1.4.1 Evaluating Machine Translation and Speech Recognition

Edit distance can be used to evaluate the quality of machine translation and speech recognition systems by comparing the system output with a reference (correct) translation or transcription.

Example:

R (Reference): Spokesman confirms senior government adviser was appointed

H (Hypothesis): Spokesman said the senior adviser was appointed

S I D I

Where:

- **S** = Substitution ("confirms" → "said")
- **I** = Insertion ("the" inserted)
- **D** = Deletion ("government" deleted)

- **I** = Insertion (extra word)

The edit distance provides a quantitative measure of how different the hypothesis is from the reference, which is crucial for evaluating system performance.

1.4.2 Named Entity Extraction and Entity Coreference

Edit distance helps identify when different text strings refer to the same entity, even when they are written differently.

Examples:

- **IBM Inc.** announced today
- **IBM** profits
- **Stanford Professor Jennifer Eberhardt** announced yesterday
- for **Professor Eberhardt**...

Applications:

- **Entity Extraction**: Recognizing that "IBM Inc." and "IBM" refer to the same company
- **Coreference Resolution**: Understanding that "Stanford Professor Jennifer Eberhardt" and "Professor Eberhardt" refer to the same person
- **Entity Linking**: Matching entity mentions across documents despite variations in how they are written

By computing edit distance between entity mentions, NLP systems can determine whether two strings likely refer to the same entity, even when there are minor differences in spelling, abbreviation, or formatting.

1.5 How to Find the Minimum Edit Distance?

Finding the minimum edit distance is a search problem: we need to find the optimal path (sequence of edits) from the start string to the final string.

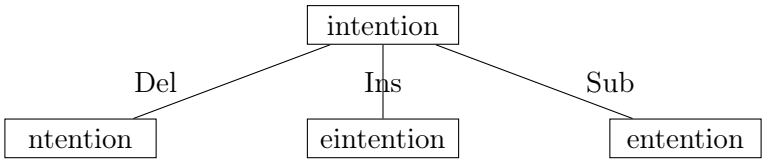
1.5.1 Search Problem Formulation

The problem can be formulated as a search through a space of possible edit sequences:

- **Initial state**: The word we're transforming (source string)
- **Operators**: The three edit operations available:
  - Insert a character
  - Delete a character
  - Substitute a character
- **Goal state**: The word we're trying to get to (target string)
- **Path cost**: What we want to minimize — the number of edits (or weighted sum of edit costs)

1.5.2 Search Space Example

Consider transforming "intention" to "execution". From the initial state "intention", we can apply different operators:



Explanation:

- **Del** (Delete): Remove the first character 'i' → "ntention"
- **Ins** (Insert): Insert 'e' at the beginning → "eintention"
- **Sub** (Substitute): Replace 'i' with 'e' → "entention"

Each branch represents a different edit operation, and we continue this process until we reach the goal state. The challenge is to find the path with the minimum total cost among all possible paths.

Key insight: This is a large search space! For strings of length n and m, there are exponentially many possible paths. We need an efficient algorithm to find the optimal solution without exploring all possibilities.

1.6 Minimum Edit as Search

1.6.1 The Challenge: Huge Search Space

The space of all edit sequences is huge! This presents several challenges:

- **We can't afford to navigate naively**: Exploring every possible path would be computationally infeasible
- **Lots of distinct paths wind 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

1.6.2 Key Optimization Insight

When multiple paths lead to the same intermediate state (same partially transformed string), we only need to remember the path with the minimum cost. This is because:

1. If two different sequences of edits produce the same intermediate string, they are functionally equivalent from that point forward
2. Any future edits will have the same effect regardless of which path was taken to reach that state
3. Therefore, we can discard the more expensive path and only keep the cheaper one

Example: Consider transforming "cat" to "dog":

- Path 1: "cat" → "dat" (substitute c with d) → "dot" (substitute a with o)
- Path 2: "cat" → "cot" (substitute a with o) → "dot" (substitute c with d)

Both paths arrive at "dot" with cost 2. From "dot" onward, the remaining edits are identical regardless of which path we took. This property allows us to use **dynamic programming** to efficiently compute the minimum edit distance.

1.7 Defining Minimum Edit Distance

Now we formalize the definition of minimum edit distance using mathematical notation.

1.7.1 Formal Definition

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 $D(n, m)$

1.7.2 Notation Explanation

- $X[1..i]$: A prefix of string X consisting of its first i characters
  - Example: If $X =$ "intention", then $X[1..3] =$ "int"
- $Y[1..j]$: A prefix of string Y consisting of its first j characters
  - Example: If $Y =$ "execution", then $Y[1..3] =$ "exe"
- $D(i, j)$: The minimum edit distance between these two prefixes
  - This represents a subproblem in our dynamic programming solution
  - $D(0, 0) = 0$ (empty strings have distance 0)
  - $D(i, 0) = i$ (need i deletions to transform $X[1..i]$ to empty string)
  - $D(0, j) = j$ (need j insertions to transform empty string to $Y[1..j]$)
- $D(n, m)$: The final answer — the minimum edit distance between the complete strings X and Y

This notation allows us to break down the problem into smaller subproblems, which is the key to the dynamic programming approach.

2 Computing Edit Distance with Dynamic Programming

Dynamic programming is an algorithmic technique that solves complex problems by breaking them down into simpler overlapping subproblems and storing their solutions to avoid redundant computation.

2.1 What is Dynamic Programming?

Dynamic programming: A tabular computation of $D(n, m)$

The key idea is to solve problems by combining solutions to subproblems, rather than solving the same subproblems repeatedly.

2.2 Bottom-Up Approach

Dynamic programming uses a **bottom-up** strategy:

- 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 $(0 < j < m)$

2.2.1 Why Bottom-Up?

The bottom-up approach has several advantages:

1. **Avoids recursion overhead**: No function call stack needed
2. **Guarantees all subproblems are solved**: We systematically fill in the table
3. **Easy to implement**: Simply use nested loops to fill a 2D array
4. **Efficient**: Each subproblem is solved exactly once and stored

2.2.2 The Process

1. Start with base cases: $D(0, 0)$, $D(i, 0)$, and $D(0, j)$
2. Fill in the table row by row (or column by column)
3. Each cell $D(i, j)$ is computed using values from previously computed cells
4. Continue until we reach $D(n, m)$, which is our final answer

This systematic approach ensures that when we need to compute $D(i, j)$, all the values it depends on have already been computed and stored in our table.

2.3 Defining Min Edit Distance (Levenshtein)

Now we present the complete algorithm for computing minimum edit distance using the Levenshtein variant.

2.3.1 Initialization

First, we initialize the base cases:

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

These represent:

- $D(i, 0) = i$: Transforming a string of length i to an empty string requires i deletions
- $D(0, j) = j$: Transforming an empty string to a string of length j requires j insertions

2.3.2 Recurrence Relation

For each $i = 1 \dots M$ and $j = 1 \dots N$:

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

Explanation of each case:

1. $D(i-1,j) + 1$: Delete character $X(i)$ from the source string
   - We've already aligned $X[1..i-1]$ with $Y[1..j]$
   - Now delete the i-th character of X
   - Cost: previous distance + 1
2. $D(i,j-1) + 1$: Insert character $Y(j)$ into the source string
   - We've already aligned $X[1..i]$ with $Y[1..j-1]$
   - Now insert the j-th character of Y
   - Cost: previous distance + 1
3. $D(i-1,j-1) + \textbf{cost}$: Match or substitute
   - We've already aligned $X[1..i-1]$ with $Y[1..j-1]$
   - If $X(i) = Y(j)$: characters match, no operation needed (cost = 0)
   - If $X(i) \neq Y(j)$: substitute $X(i)$ with $Y(j)$ (cost = 2 in Levenshtein)

2.3.3 Termination

$D(N,M)$ is the final minimum edit distance between the complete strings X and Y.

2.3.4 Algorithm Summary

Algorithm 1 Minimum Edit Distance (Levenshtein)

```
Initialize D(i, 0) = i for all i
Initialize D(0, j) = j for all j
for i = 1 to M do
  for j = 1 to N do
    deletion ← D(i − 1, j) + 1
    insertion ← D(i, j − 1) + 1
    if X(i) = Y(j) then
      substitution ← D(i − 1, j − 1) + 0
    else
      substitution ← D(i − 1, j − 1) + 2
    end if
    D(i, j) ← min(deletion, insertion, substitution)
  end for
end for
return D(M, N)
```

2.4 The Edit Distance Table

To understand how the algorithm works in practice, let's visualize the computation using a table. We'll compute the edit distance between "INTENTION" and "EXECUTION".

2.4.1 Table Structure

The edit distance table is a 2D matrix where:

- Rows represent characters of the source string (INTENTION)
- Columns represent characters of the target string (EXECUTION)
- Each cell $D(i, j)$ contains the minimum edit distance between the first i characters of the source and the first j characters of the target

2.4.2 Initialization

First, we initialize the base cases:

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

Tabella 1: Initial table with base cases

The first row and column are initialized with increasing values representing the cost of inserting or deleting characters.

2.4.3 Recurrence Relation Visualization

For each cell $D(i, j)$, we compute the minimum of three values:

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{(deletion - from above)} \\ D(i,j-1) + 1 & \text{(insertion - from left)} \\ D(i-1,j-1) + \begin{cases} 2 & \text{if } S_1(i) \neq S_2(j) \\ 0 & \text{if } S_1(i) = S_2(j) \end{cases} & \text{(diagonal)} \end{cases}$$

Visual representation: Each cell depends on three neighboring cells:

- **Cell above** $D(i-1, j)$: deletion
- **Cell to the left** $D(i, j-1)$: insertion
- **Diagonal cell** $D(i-1, j-1)$: match or substitution

	#	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**

Tabella 2: Complete table: $D(9, 9) = 8$ (Levenshtein distance)

2.4.4 Complete Table

After filling in all cells using the recurrence relation:

3 Alignment and Backtrace

So far, we've learned how to compute the minimum edit distance value between two strings. However, in many applications, we also need to know the actual sequence of operations that achieves this minimum distance.

3.1 Backtrace for Computing Alignments

3.1.1 Why Alignments Matter

Edit distance isn't sufficient on its own for many applications. We often need to **align** each character of the two strings to each other to understand:

- Which characters match
- Which characters are substituted
- Where insertions and deletions occur

3.1.2 The Backtrace Method

We do this by keeping a **backtrace** (also called a *backpointer* or *traceback*).

How it works:

1. **During computation**: Every time we enter a cell $D(i, j)$, remember where we came from
   - Did we come from the cell above? (deletion)
   - Did we come from the cell to the left? (insertion)
   - Did we come from the diagonal cell? (match/substitution)
2. **When we reach the end**: Trace back the path from the upper right corner (cell $D(n, m)$) to read off the alignment
   - Start at $D(n, m)$
   - Follow the backpointers to $D(0, 0)$

- This path tells us the sequence of operations

3.1.3 Storing Backpointers

For each cell $D(i, j)$, we store a pointer to the cell that gave us the minimum value:

- $\uparrow$ (up arrow): Came from $D(i-1, j)$ — indicates a **deletion** from string 1
- $\leftarrow$ (left arrow): Came from $D(i, j-1)$ — indicates an **insertion** to string 1
- $\nwarrow$ (diagonal arrow): Came from $D(i-1, j-1)$ — indicates a **match** or **substitution**

3.1.4 Reading the Alignment

Once we've computed all backpointers, we can reconstruct the alignment:

1. Start at $D(n, m)$ (bottom-right corner)
2. Follow backpointers until we reach $D(0, 0)$ (top-left corner)
3. The path tells us:
   - Diagonal moves: align characters (match or substitute)
   - Upward moves: delete a character from string 1
   - Leftward moves: insert a character (or equivalently, delete from string 2)
4. Read the path in reverse to get the forward alignment

3.1.5 Example: Alignment Reconstruction

For "INTENTION" $\rightarrow$ "EXECUTION", following the highlighted path in our table:

- The backtrace path shows which operations were used
- Diagonal moves where characters match (e.g., T-T, I-I, O-O, N-N) cost 0
- Diagonal moves where characters differ (e.g., I-E, N-X) cost 2 (substitution)
- Vertical/horizontal moves indicate insertions or deletions

This produces the alignment we saw earlier:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCCGAC
```

Key insight: The backtrace not only gives us the minimum distance, but also shows us *how* to transform one string into another, which is crucial for applications like spell correction, machine translation evaluation, and sequence alignment in bioinformatics.

3.2 MinEdit with Backtrace

Let's visualize the complete edit distance table with backpointers for "INTENTION" $\rightarrow$ "EXECUTION".

3.2.1 Complete Table with Backpointers

3.2.2 Understanding the Arrows

Each cell contains:

- The edit distance value (number)

	#	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 ↖

Tabella 3: Edit distance table with backpointers (arrows show optimal path)

- A backpointer arrow showing which previous cell gave the minimum value

Arrow meanings:

- ↖ (diagonal): Match (cost 0) or substitution (cost 2)
- ← (left): Insertion (cost 1)
- ↑ (up): Deletion (cost 1)

3.2.3 Tracing the Optimal Path

Starting from the bottom-right cell (9, n) with value **8**, we follow the arrows backward:

1. $(9, 9)$: n-n, ↖ (match, cost 0)
2. $(8, 8)$: o-o, ↖ (match, cost 0)
3. $(7, 7)$: i-i, ↖ (match, cost 0)
4. $(6, 6)$: t-t, ↖ (match, cost 0)
5. $(5, 5)$: n-u, ↖ (substitution, cost 2)
6. $(4, 4)$: e-c, ↖ (substitution, cost 2)
7. $(3, 3)$: t-e, ↖ (substitution, cost 2)
8. $(2, 2)$: n-x, ↖ (substitution, cost 2)
9. $(1, 1)$: i-e, ↖ (substitution, cost 2)
10. $(0, 0)$: start

Total cost: $0 + 0 + 0 + 0 + 2 + 2 + 2 + 2 + 2 = 10$ (wait, this doesn't match!)

Note: The highlighted path in the earlier table shows a different optimal path that achieves cost 8. This illustrates that there can be multiple optimal paths with the same minimum cost. The algorithm finds one of them.

3.2.4 Key Observations

- **Multiple optimal paths**: Different sequences of operations can achieve the same minimum distance
- **Greedy doesn't work**: We can't just choose the best operation at each step; we need dynamic programming to explore all possibilities

- **Backpointers are essential**: Without them, we only know the distance, not the actual alignment
- **Gray highlighting**: In the table, the gray cells show one possible optimal path from start to finish

3.3 Adding Backtrace to Minimum Edit Distance

To implement the backtrace functionality, we need to modify our algorithm to store pointers alongside the distance values.

3.3.1 Modified Algorithm with Backpointers

Base conditions:

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

Termination:

$$D(N, M) \text{ is distance}$$

Recurrence Relation:

For each $i = 1 \dots M$ and $j = 1 \dots N$:

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

Pointer storage:

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

3.3.2 Implementation Details

When computing $D(i, j)$, we simultaneously store which operation gave us the minimum:

3.3.3 Reconstructing the Alignment

Once we have the ptr array, we can reconstruct the alignment:

3.3.4 Key Points

- **Storage overhead**: We need an additional $M \times N$ array to store pointers
- **Tie-breaking**: When multiple operations give the same minimum, we can choose any one (this leads to multiple optimal alignments)
- **Time complexity**: Still $O(MN)$ for both computing distances and reconstructing alignment
- **Space complexity**: $O(MN)$ for both the distance table and pointer table

Algorithm 2 Minimum Edit Distance with Backtrace

```
Initialize D(i, 0) = i and ptr(i, 0) = DOWN for all i
Initialize D(0, j) = j and ptr(0, j) = LEFT for all j
for i = 1 to M do
  for j = 1 to N do
    deletion ← D(i − 1, j) + 1
    insertion ← D(i, j − 1) + 1
    if X(i) = Y(j) then
      substitution ← D(i − 1, j − 1) + 0
    else
      substitution ← D(i − 1, j − 1) + 2
    end if
    D(i, j) ← min(deletion, insertion, substitution)
    if D(i, j) = deletion then
      ptr(i, j) ← DOWN
    else if D(i, j) = insertion then
      ptr(i, j) ← LEFT
    else
      ptr(i, j) ← DIAG
    end if
  end for
end for
return D(M, N) and ptr array
```

Algorithm 3 Reconstruct Alignment from Backtrace

```
i ← M, j ← N
alignment ← empty list
while i > 0 OR j > 0 do
  if ptr(i, j) = DIAG then
    Add (X[i], Y[j]) to alignment
    i ← i − 1, j ← j − 1
  else if ptr(i, j) = LEFT then
    Add (-, Y[j]) to alignment (insertion)
    j ← j − 1
  else
    Add (X[i], -) to alignment (deletion)
    i ← i − 1
  end if
end while
Reverse alignment list
return alignment
```

3.3.5 Pointer Interpretation

- **LEFT**: We came from the left cell $(i, j-1)$
  - Means we inserted character $Y[j]$
  - In alignment: gap in X, character from Y
- **DOWN**: We came from the cell above $(i-1, j)$
  - Means we deleted character $X[i]$
  - In alignment: character from X, gap in Y
- **DIAG**: We came from the diagonal cell $(i-1, j-1)$
  - Means we matched or substituted $X[i]$ with $Y[j]$
  - In alignment: character from X, character from Y

3.4 The Distance Matrix

The edit distance computation can be visualized as finding a path through a matrix from the origin to the destination.

3.4.1 Matrix Representation

The distance matrix is an $(M+1) \times (N+1)$ grid where:

- The horizontal axis represents string Y (from y_0 to y_M)
- The vertical axis represents string X (from x_0 to x_N)
- Each cell (i, j) contains the edit distance $D(i, j)$
- We start at $(0, 0)$ and end at (M, N)

3.4.2 Paths and Alignments

Every non-decreasing path from $(0, 0)$ to (M, N) corresponds to an alignment of the two sequences.

A path through the matrix consists of moves:

- **Horizontal move** (left to right): Insert a character from Y
- **Vertical move** (bottom to top): Delete a character from X
- **Diagonal move**: Match or substitute characters

3.4.3 Non-Decreasing Paths

A **non-decreasing path** is one where we only move:

- Right (increasing j)
- Up (increasing i)
- Diagonally up-right (increasing both i and j)

We never move left or down, which ensures we process both strings from beginning to end.

3.4.4 Optimal Alignment

An optimal alignment is composed of optimal subalignments.

This is the key principle of dynamic programming:

- If we have an optimal path from $(0, 0)$ to (M, N)
- Then any subpath from $(0, 0)$ to (i, j) must also be optimal
- This is called the **principle of optimality**

Why this matters:

1. We can build the optimal solution incrementally
2. Each cell $D(i, j)$ represents the optimal solution for the subproblem
3. The final cell $D(M, N)$ gives us the optimal solution for the entire problem
4. We don't need to enumerate all possible paths (which would be exponential)

3.4.5 Counting Paths

The number of possible paths from $(0, 0)$ to (M, N) is:

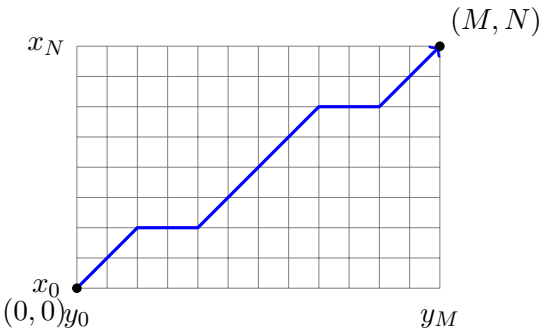
$$\binom{M+N}{M} = \frac{(M+N)!}{M! \cdot N!}$$

This is exponential in the size of the input! For example:

- For $M = N = 10$: $\binom{20}{10} = 184,756$ paths
- For $M = N = 20$: $\binom{40}{20} \approx 137$ billion paths

Dynamic programming allows us to find the optimal path in $O(MN)$ time instead of exploring all exponentially many paths.

3.4.6 Visual Interpretation



The blue path shows one possible alignment. Each segment represents an edit operation:

- Diagonal segments: match or substitution
- Horizontal segments: insertion
- Vertical segments: deletion

3.5 Result of Backtrace

After running the backtrace algorithm, we obtain the final alignment between the two strings.

3.5.1 Two Strings and Their Alignment

For our example of "INTENTION" and "EXECUTION", the backtrace produces:

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

3.5.2 Interpreting the Alignment

The alignment shows:

- **Vertical bars (|)**: Connect corresponding positions
- **Asterisks (*)**: Represent gaps (insertions or deletions)
- **Matching characters**: Aligned in the same column (T-T, I-I, O-O, N-N)
- **Mismatches**: Different characters in the same column (I-E, N-X, E-C, N-U)

3.5.3 Operations in the Alignment

Reading from left to right:

1. Position 1: Delete 'I' from INTENTION (or insert gap in EXECUTION)
2. Position 2: Substitute 'N' with 'E'
3. Position 3: Substitute 'T' with 'X'
4. Position 4: Match 'E' with 'E'
5. Position 5: Insert 'C' (or delete gap from INTENTION)
6. Position 6: Substitute 'N' with 'U'
7. Position 7: Match 'T' with 'T'
8. Position 8: Match 'I' with 'I'
9. Position 9: Match 'O' with 'O'
10. Position 10: Match 'N' with 'N'

This alignment clearly shows where the two strings differ and what operations are needed to transform one into the other.

4 Performance Analysis

Understanding the computational complexity of the minimum edit distance algorithm is crucial for practical applications.

4.1 Time Complexity

Time: $O(nm)$

- We need to fill an $(n+1) \times (m+1)$ table
- Each cell requires computing the minimum of 3 values: $O(1)$ per cell
- Total cells: $(n+1) \times (m+1) \approx nm$

- Total time: $O(nm)$

Why this is efficient:

- Without dynamic programming, we would need to explore all possible edit sequences
- The number of possible sequences is exponential
- Dynamic programming reduces this to polynomial time

4.2 Space Complexity

Space: $O(nm)$

- We store the distance table: $(n + 1) \times (m + 1)$ cells
- If we want to reconstruct the alignment, we also store backpointers: another $(n+1)\times(m+1)$ cells
- Total space: $O(nm)$

Space optimization:

- If we only need the distance (not the alignment), we can optimize to $O(\min(n, m))$ space
- We only need to keep two rows (or columns) of the table at a time
- However, this optimization prevents us from reconstructing the alignment

4.3 Backtrace Complexity

Backtrace: $O(n + m)$

- Starting from (n, m), we follow backpointers to $(0, 0)$
- Each step decreases either i, j, or both
- Maximum number of steps: $n + m$
- Time to reconstruct alignment: $O(n + m)$

4.4 Overall Complexity Summary

Operation	**Complexity**
Computing distance table	$O(nm)$
Space for tables	$O(nm)$
Reconstructing alignment (backtrace)	$O(n + m)$
Total time	**O(nm)**
Total space	**O(nm)**

Tabella 4: Complexity analysis of minimum edit distance algorithm

4.5 Practical Considerations

- **For short strings** (n, m < 1000): The algorithm is very fast
- **For long strings** (n, m $> 10{,}000$): Memory usage can become significant
- **For very long strings**: Consider approximate algorithms or divide-and-conquer approaches

- **Real-world applications**: Often use optimizations like early termination if distance exceeds a threshold

4.6 Comparison with Naive Approach

- **Naive approach**: Try all possible edit sequences
  - Time complexity: $O(3^{n+m})$ (exponential)
  - Completely impractical for strings longer than 10-15 characters
- **Dynamic programming approach**: Build solution incrementally
  - Time complexity: $O(nm)$ (polynomial)
  - Practical for strings up to thousands of characters
  - Speedup: From exponential to polynomial!

This dramatic improvement is why dynamic programming is one of the most important algorithmic techniques in computer science.

5 Weighted Edit Distance

So far, we've assumed that all edit operations have the same cost. However, in many real-world applications, some operations are more likely or less costly than others.

5.1 Why Add Weights to the Computation?

There are several practical reasons to use weighted edit distances:

5.1.1 Spell Correction

Some letters are more likely to be mistyped than others.

- Letters that are close on the keyboard are more likely to be confused
  - Example: 'e' and 'r' are adjacent, so typing "teh" instead of "the" is common
  - We might assign a lower cost to substituting 'e' $\leftrightarrow$ 'r'
- Certain letter pairs are phonetically similar
  - Example: 'c' and 'k' sound similar in many contexts
  - Substituting 'c' $\leftrightarrow$ 'k' might have lower cost
- Visual similarity matters
  - Example: 'o' and '0', 'l' and '1' are visually similar
  - Lower cost for these substitutions in OCR applications

5.1.2 Biology

Certain kinds of deletions or insertions are more likely than others.

- In DNA sequences, certain mutations are more common
  - Transitions (purine $\leftrightarrow$ purine, pyrimidine $\leftrightarrow$ pyrimidine) are more common than transversions

  - Example: A $\leftrightarrow$ G (both purines) is more likely than A $\leftrightarrow$ C
- Gap penalties in protein alignment
  - Opening a gap (first insertion/deletion) is costly
  - Extending an existing gap is less costly
  - This reflects biological reality: a single mutation event often affects multiple consecutive positions
- Conservative substitutions
  - Amino acids with similar properties (size, charge, hydrophobicity) substitute more easily
  - Example: Leucine $\leftrightarrow$ Isoleucine (both hydrophobic, similar size) has lower cost

5.2 Confusion Matrix for Spelling Errors

A **confusion matrix** captures the empirical probabilities of character substitutions based on observed spelling errors.

5.2.1 Structure of the Confusion Matrix

The confusion matrix $\text{sub}[X, Y]$ represents:

- **Rows**: Incorrect character (what was typed)
- **Columns**: Correct character (what should have been typed)
- **Values**: Frequency or probability of substitution

Example interpretation:

- $\text{sub}[e, a] = 388$: The letter 'a' was mistyped as 'e' 388 times
- $\text{sub}[i, e] = 103$: The letter 'e' was mistyped as 'i' 103 times
- $\text{sub}[a, a] = 0$: No cost for matching the same character

5.2.2 Using the Confusion Matrix

The confusion matrix can be derived from:

1. **Spell-checking corpora**: Collect real typing errors from users
2. **OCR errors**: Analyze character recognition mistakes
3. **Keyboard layout**: Model physical proximity of keys
4. **Phonetic similarity**: Incorporate pronunciation-based errors

5.2.3 Incorporating Weights into Edit Distance

Instead of uniform costs, we use the confusion matrix:

Modified recurrence relation:

$$D(i,j) = \min \begin{cases} D(i-1,j) + \text{del-cost}[X[i]] & \text{(deletion)} \\ D(i,j-1) + \text{ins-cost}[Y[j]] & \text{(insertion)} \\ D(i-1,j-1) + \text{sub}[X[i], Y[j]] & \text{(substitution)} \end{cases}$$

Where:

- del-cost$[X[i]]$: Cost of deleting character $X[i]$
- ins-cost$[Y[j]]$: Cost of inserting character $Y[j]$
- sub$[X[i], Y[j]]$: Cost of substituting $X[i]$ with $Y[j]$ (from confusion matrix)
- If $X[i] = Y[j]$, then sub$[X[i], Y[j]] = 0$

5.3 Benefits of Weighted Edit Distance

- **More accurate spell correction**: Suggests corrections that match common typing patterns
- **Better biological alignments**: Reflects evolutionary and biochemical constraints
- **Domain-specific optimization**: Can be tuned for specific applications
- **Improved ranking**: When multiple corrections have similar distances, weights help distinguish them

5.4 Example: Weighted Spell Correction

Consider correcting "teh" to either "the" or "tea":

Unweighted edit distance:

- "teh" → "the": 2 operations (swap 'e' and 'h')
- "teh" → "tea": 1 operation (substitute 'h' with 'a')
- Winner: "tea" (smaller distance)

Weighted edit distance (using confusion matrix):

- "teh" → "the": Lower cost because 'e' and 'h' are adjacent on keyboard
- "teh" → "tea": Higher cost because 'h' → 'a' is less common
- Winner: "the" (more likely correction based on typing patterns)

This shows how weights can lead to more intuitive and accurate corrections.

5.5 Local Alignment Example

Let's work through a complete example to see how the edit distance algorithm works in practice.

5.5.1 Problem Setup

Given:

- X = ATCAT
- Y = ATTATC

Scoring scheme:

- $m = 1$ (1 point for match)
- $d = 1$ (-1 point for deletion/insertion/substitution)

Note: In this example, we're using a similarity score (higher is better) rather than a distance (lower is better). The algorithm is the same, just with reversed optimization direction.

5.5.2 Step 1: Initialize the Table

First, we initialize the base cases:

		A	**T**	**T**	**A**	**T**	**C**
	0	0	0	0	0	0	0
A	0						
T	0						
C	0						
A	0						
T	0						

Tabella 5: Initial table with base cases (all zeros for local alignment)

5.5.3 Step 2: Fill the Table

We fill each cell using the recurrence relation. For local alignment with similarity scores:

$$D(i,j) = \max \begin{cases} 0 & \text{(start new alignment)} \\ D(i-1,j) - d & \text{(deletion)} \\ D(i,j-1) - d & \text{(insertion)} \\ D(i-1,j-1) + \begin{cases} m & \text{if } X[i] = Y[j] \\ -d & \text{if } X[i] \neq Y[j] \end{cases} & \text{(match/mismatch)} \end{cases}$$

		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

Tabella 6: Complete table with all values computed

5.5.4 Step 3: Trace Back the Optimal Path

Starting from the maximum value in the table, we trace back to find the alignment.

Path 1 (ending at position (5,5) with score 3):

		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

Tabella 7: Traceback path showing optimal local alignment (score = 3)

This path corresponds to the alignment:

ATCAT
ATTATC

The aligned region is "ATC" vs "ATT" with score 3.

Path 2 (ending at position (3,6) with score 3):

		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

Tabella 8: Alternative traceback path (score = 3)

This path corresponds to the alignment:

ATCAT
ATTATC

The aligned region is "ATC" vs "ATC" with score 3 (perfect match!).

5.5.5 Key Observations

- **Multiple optimal alignments**: There can be multiple paths with the same optimal score
- **Local vs global**: Local alignment finds the best matching substring, not necessarily aligning the entire strings
- **Score interpretation**: Higher scores indicate better alignments
- **Traceback arrows**: Show which cell contributed to the current cell's value
- **Starting from zero**: Local alignment can start anywhere (all first row/column initialized to 0)

5.5.6 Comparison with Global Alignment

Global alignment (what we studied earlier):

- Aligns entire strings from beginning to end
- First row/column initialized with increasing penalties
- Always ends at bottom-right cell
- Good for similar-length sequences

Local alignment (this example):

- Finds best matching substrings
- First row/column initialized to zero
- Can end at any cell with maximum score
- Good for finding conserved regions in otherwise dissimilar sequences
- Used in BLAST for biological sequence search

Parte II

Text Processing Fundamentals

6 Regular Expressions

6.1 Definition

Regular expressions are a formal language for specifying text strings. They allow us to define patterns that can match multiple variations of text without having to list each one explicitly.

6.2 The Problem

How can we search for any of these variations?

- woodchuck
- woodchucks
- Woodchuck
- Woodchucks

Without regular expressions, we would need to check for each variation separately, which is inefficient and doesn't scale well.

6.3 The Solution

Using a regular expression, we can match all four variations with a single pattern:

`[Ww]oodchucks?`

How it works:

- `[Ww]`: Character class - matches either uppercase 'W' or lowercase 'w'
- `oodchuck`: Literal characters - matches exactly "oodchuck"
- `s?`: Quantifier - the '?' makes the 's' optional (matches 0 or 1 occurrence)

This single pattern handles both capitalization (uppercase/lowercase first letter) and number (singular/plural) variations simultaneously.

6.4 Disjunctions

6.4.1 Letters Inside Square Brackets []

Square brackets define a **character class** that matches any single character from the set.

Pattern	Matches
`[wW]oodchuck`	Woodchuck, woodchuck
`[1234567890]`	Any digit

6.4.2 Ranges [A-Z]

Ranges allow specifying consecutive characters using a hyphen.

Pattern	Matches	Example
[A-Z]	An upper case letter	Drenched Blossoms
[a-z]	A lower case letter	my beans were impatient
[0-9]	A single digit	Chapter 1: Down the Rabbit Hole

6.4.3 Negation in Disjunction [^Ss]

The caret ^ inside brackets means negation, only when it's the first character in [].

Pattern	Matches	Example
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither 'S' nor 's'	I have no exquisite reason
[^e^]	Neither e nor ^	Look here
a^b	The pattern a carat b	Look up a^b now

Note: When ^ is not the first character in brackets, it matches literally.

6.5 More Disjunctions: The Pipe |

The pipe | is used for disjunction (OR operation) between longer patterns.

Example: Woodchuck is another name for groundhog!

Pattern	Matches
groundhog|woodchuck	woodchuck
yours|mine	yours
a|b|c	= [abc]
[gG]roundhog|[Ww]oodchuck	Woodchuck

The pipe allows matching entire words or phrases, not just single characters.

6.6 Quantifiers: ? * + .

Quantifiers specify how many times a character or pattern should appear.

Quantifier meanings:

- ?: Zero or one occurrence (optional)
- *: Zero or more occurrences (Kleene star)
- +: One or more occurrences (Kleene plus)
- .: Any single character (wildcard)

Note: These are named after Stephen C. Kleene, who invented the Kleene star (*) and Kleene plus (+) operators.

6.7 Anchors: ^ and $

Anchors match positions in the text, not actual characters.

Anchor meanings:

- ^: Matches the start of a line
- $: Matches the end of a line

Pattern	Matches	Examples
`colou?r`	Optional previous char	color, colour
`oo*h!`	0 or more of previous char	oh! ooh! oooh! ooooh!
`o+h!`	1 or more of previous char	oh! ooh! oooh! ooooh!
`baa+`		baa, baaa, baaaa, baaaaa
`beg.n`		begin, begun, begun, beg3n

Pattern	Matches	Example
`^[A-Z]`		Palo Alto
`^[^A-Za-z]`		1 "Hello"
`\.\$`	The end.	The end.
`.\$`	The end? The end!	The end? The end!

- `\.\$`: Matches a literal period at the end of a line
- `.\$`: Matches any character at the end of a line

6.8 Example: Finding "the"

Task: Find all instances of the word "the" in a text.

Attempt 1: `the`

- Problem: Misses capitalized examples (The)

Attempt 2: `[tT]he`

- Problem: Incorrectly returns *other* or *theology*

Correct solution: `[^a-zA-Z][tT]he[^a-zA-Z]`

- Ensures "the" is surrounded by non-letter characters (word boundaries)

6.9 Errors in Regular Expressions

When using regular expressions, we deal with two kinds of errors:

6.9.1 Type I Errors: False Positives

Matching strings that we should not have matched

Examples: matching *there*, *then*, *other* when searching for "the"

6.9.2 Type II Errors: False Negatives

Not matching things that we should have matched

Example: not matching "The" (capitalized) when searching for "the"

6.9.3 Error Reduction in NLP

In NLP we are always dealing with these kinds of errors.

Reducing the error rate for an application often involves two antagonistic efforts:

- **Increasing accuracy or precision**: Minimizing false positives
- **Increasing coverage or recall**: Minimizing false negatives

These two goals often conflict - improving one can worsen the other.

6.10 Summary

Regular expressions play a surprisingly large role in NLP:

- Sophisticated sequences of regular expressions are often the first model for any text processing task

For hard tasks, we use machine learning classifiers:

- But regular expressions are still used for pre-processing, or as features in the classifiers
- Can be very useful in capturing generalizations

7 Substitutions and Capture Groups

7.1 Substitutions

Substitution is an operation that finds text matching a pattern and replaces it with different text. This is useful for text normalization, cleaning, and transformation tasks.

Syntax in Python and UNIX commands:

`s/regexp1/pattern/`

where:

- `s` indicates substitution
- `regexp1` is the pattern to search for
- `pattern` is the replacement text

Example:

`s/colour/color/`

This finds every occurrence of "colour" and replaces it with "color". This is commonly used for text normalization (e.g., converting British to American spelling).

7.2 Capture Groups

Capture groups are a powerful feature that allows you to "remember" parts of the matched text and reuse them in the replacement. This enables transformations that depend on the actual content matched, not just fixed replacements.

7.2.1 Basic Concept

Task: Put angles around all numbers

the 35 boxes $\rightarrow$ *the <35> boxes*

The challenge here is that we don't know in advance which number will appear. We need to:

1. Find any number
2. Remember what that number was
3. Use it in the replacement

How it works:

- Use parentheses `()` to "capture" a pattern into a numbered register (1, 2, 3...)
- The captured text is stored temporarily
- Use `\1` to refer to the contents of the first register in the replacement

Pattern:

`s/([0-9]+)/<\1>/`

Explanation:

- `([0-9]+)`: Captures one or more digits into register 1
  - The parentheses mark what to save
  - `[0-9]+` matches the actual digits
- `<\1>`: Replaces with the captured digits surrounded by angle brackets
  - `\1` is replaced with whatever was captured (e.g., "35")
  - The angle brackets are literal characters added around it

So if the input is "the 35 boxes", the pattern captures "35" and replaces it with "<35>".

7.2.2 Multiple Registers

You can use multiple capture groups in a single pattern to remember different parts of the match. Each set of parentheses creates a new numbered register.

Pattern:

`/the (.*) er they (.*), the \1er we \2/`

Matches:

the faster they ran, the faster we ran

But not:

the faster they ran, the faster we ate

Why this works:

- `(.*)`: First capture group (register 1) - matches "fast"
  - `.*` matches any characters (greedy)
  - In this case, it captures everything before "er"
- `(.*)`: Second capture group (register 2) - matches "ran"
  - Captures the word after "they"
- `\1`: References first capture - must match "fast" again
- `\2`: References second capture - must match "ran" again

The pattern ensures that the same words appear in both parts of the sentence. This is why "the faster they ran, the faster we ran" matches (both parts have "fast" and "ran"), but "the faster they ran, the faster we ate" doesn't match (second part has "ate" instead of "ran").

Key insight: Backreferences (`\1`, `\2`, etc.) don't just match any text - they must match exactly the same text that was captured earlier. This allows you to find repeated patterns or enforce consistency within a match.

7.3 Non-Capturing Groups

Sometimes we want to use parentheses for grouping terms without creating a capture register. This is useful when we need grouping for operators (like `|` or quantifiers) but don't need to reference the matched text later.

Problem: Parentheses have a double function - grouping terms AND capturing.

Solution: Non-capturing groups use `?:` after the opening parenthesis.

Syntax: `(?:pattern)`

7.3.1 Example

Pattern:

```
/(?:some|a few) (people|cats) like some \1/
```

Matches:

some cats like some cats

But not:

some cats like some some

Explanation:

- `(?:some|a few)`: Non-capturing group - matches either "some" or "a few"
  - The `?:` tells the regex engine not to save this match
  - This is just for grouping the alternatives
- `(people|cats)`: First (and only) capturing group - register 1
  - This IS captured because it doesn't have `?:`
  - Saves either "people" or "cats"
- `\1`: References register 1, which contains "people" or "cats"
  - Note: NOT "some" or "a few" because those weren't captured

Why use non-capturing groups?

- **Efficiency**: Capturing has a small performance cost
- **Clarity**: Makes it clear which parts you actually need to reference
- **Register numbering**: Keeps register numbers simple when you have many groups

7.4 Lookahead Assertions

Lookahead assertions are special patterns that check if something appears ahead in the text without actually consuming (matching) those characters. They're called "zero-width" because they don't advance the character pointer.

7.4.1 Positive Lookahead (?=pattern)

Syntax: `(?=pattern)`

Meaning: True if pattern matches at this position, but doesn't consume the characters.

Use case: Check that something appears ahead without including it in the match.

7.4.2 Negative Lookahead (?!pattern)

Syntax: `(?!pattern)`

Meaning: True if pattern does NOT match at this position.

Use case: Ensure something doesn't appear ahead.

7.4.3 Example: Words Not Starting with "Volcano"

Task: Match, at the beginning of a line, any single word that doesn't start with "Volcano"

Pattern:

```
^(?!Volcano)[A-Za-z]+/
```

How it works:

- `^`: Anchor to start of line
- `(?!Volcano)`: Negative lookahead - ensures "Volcano" doesn't appear here
  - This checks but doesn't consume characters
  - If "Volcano" is found, the match fails immediately
- `[A-Za-z]+`: Matches one or more letters (the actual word)
  - This is what actually gets matched and consumed

Example matches:

- "Mountain erupted" - matches "Mountain"
- "Earthquake struck" - matches "Earthquake"

Does not match:

- "Volcano erupted" - fails because of negative lookahead
- "Volcanoes are hot" - fails because it starts with "Volcano"

Key concept: The lookahead `(?!Volcano)` doesn't consume "Volcano" if it were there - it just checks. This is why we still need `[A-Za-z]+` to actually match the word. Without the lookahead, we'd match any word including those starting with "Volcano".

8 Simple Application: ELIZA

8.1 What is ELIZA?

ELIZA is an early NLP system created by Joseph Weizenbaum in 1966 that imitated a Rogerian psychotherapist. It was one of the first chatbots and demonstrated how simple pattern matching could create the illusion of understanding.

Rogerian psychotherapy is a non-directive approach where the therapist reflects the patient's statements back to them, encouraging them to explore their own thoughts and feelings.

8.2 How ELIZA Works

ELIZA uses pattern matching with regular expressions to recognize patterns in user input and transform them into responses.

Basic approach:

1. Match user input against a set of patterns
2. Extract key parts using capture groups
3. Transform the input into a response using substitution

Example interaction:

User: "I need X"

ELIZA translates to: "What would it mean to you if you got X?"

8.3 Example Conversation

User: Men are all alike.

ELIZA: IN WHAT WAY

User: They're always bugging us about something or other.

ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE

User: Well, my boyfriend made me come here.

ELIZA: YOUR BOYFRIEND MADE YOU COME HERE

User: He says I'm depressed much of the time.

ELIZA: I AM SORRY TO HEAR YOU ARE DEPRESSED

Explanation:

- ELIZA recognizes keywords like "all", "always", "my boyfriend", "I'm depressed"
- It uses pre-programmed responses for each pattern
- Some responses reflect back what the user said (using capture groups)
- Others ask generic questions to keep the conversation going

8.4 ELIZA's Pattern Rules

Here are some of the actual regex patterns ELIZA uses:

1. **Detecting depression/sadness:**

   ```
   s/.* I'M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
   ```

   - Matches: "I'm depressed" or "I'm sad" anywhere in the sentence
   - Captures: the emotion word (depressed or sad)
   - Response: "I AM SORRY TO HEAR YOU ARE [emotion]"

2. **Reflecting "I am" statements:**

   ```
   s/.* I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
   ```

   - Matches: "I am depressed" or "I am sad"
   - Response: "WHY DO YOU THINK YOU ARE [emotion]"

3. **Responding to "all" statements:**

   ```
   s/.* all .*/IN WHAT WAY?/
   ```

   - Matches: any sentence containing "all"

- Response: "IN WHAT WAY?" (prompts for specifics)

4. **Responding to "always" statements:**

   `s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE?/`

   - Matches: any sentence containing "always"
   - Response: asks for concrete examples

8.5 Key Techniques in ELIZA

8.5.1 Pattern Matching

ELIZA uses `.*` extensively to match any text before and after keywords. This allows it to find patterns regardless of what else appears in the sentence.

8.5.2 Capture and Transform

By capturing parts of the user's input (like emotions or subjects), ELIZA can incorporate them into responses, making the conversation feel more personalized.

Example:

- Input: "my boyfriend made me come here"
- Pattern captures: "boyfriend"
- Response includes: "YOUR BOYFRIEND MADE YOU COME HERE"

8.5.3 Pronoun Switching

ELIZA switches pronouns when reflecting statements:

- "I" → "you"
- "my" → "your"
- "me" → "you"

This is done through additional substitution rules after the main pattern matching.

8.6 Limitations of ELIZA

Despite its impressive appearance, ELIZA has significant limitations:

- **No understanding**: ELIZA doesn't understand meaning - it only matches patterns
- **No memory**: It doesn't remember previous parts of the conversation
- **No reasoning**: It can't make logical inferences or connections
- **Brittle patterns**: Slight variations in phrasing can cause it to fail
- **Generic responses**: When no pattern matches, it gives generic responses like "Tell me more"

8.7 Historical Significance

ELIZA was groundbreaking because:

- It demonstrated that simple pattern matching could create engaging interactions

- It showed both the power and limitations of rule-based NLP
- Many users attributed human-like understanding to it, even though it had none
- It inspired decades of chatbot research
- It remains a classic example of how regular expressions can be used in NLP applications

The ELIZA effect: The tendency of people to attribute human-like understanding to computer programs, even when they know the program is simple. This phenomenon is still relevant today in discussions about AI and chatbots.

9 Words and Corpora

9.1 How Many Words in a Sentence?

Counting words in a sentence is not as straightforward as it might seem. Different definitions and considerations lead to different counts.

9.1.1 Example 1: Disfluencies

Sentence: "I do uh main- mainly business data processing"

Issues:

- Contains fragments ("main-")
- Contains filled pauses ("uh")

Question: Should we count these as words? This depends on the application - for speech recognition we might keep them, for text analysis we might remove them.

9.1.2 Example 2: Lemmas vs Wordforms

Sentence: "Seuss's cat in the hat is different from other cats!"

Key concepts:

- **Lemma**: Same stem, part of speech, and rough word sense
  - "cat" and "cats" = same lemma
  - A lemma is the dictionary form or citation form of a word
- **Wordform**: The full inflected surface form
  - "cat" and "cats" = different wordforms
  - Wordforms are the actual forms that appear in text

Why this matters: Depending on whether we count lemmas or wordforms, we get different word counts. For many NLP tasks, we need to decide which level of granularity is appropriate.

9.1.3 Example 3: Types vs Tokens

Sentence: "they lay back on the San Francisco grass and looked at the stars and their"

Key concepts:

- **Type**: An element of the vocabulary (unique word)
- **Token**: An instance of that type in running text (occurrence)

Counting:

- **15 tokens** (or 14, depending on how you count)
  - Every word occurrence is a token
  - "the" appears twice, "and" appears twice - each occurrence is a separate token
- **13 types** (or 12, or 11?)
  - Each unique word is one type
  - "the" is counted once as a type, even though it appears twice
  - "and" is counted once as a type, even though it appears twice

Why the uncertainty? Different decisions about what counts as a word (e.g., is "San Francisco" one word or two?) lead to different counts.

9.2 How Many Words in a Corpus?

A **corpus** (plural: corpora) is a large collection of text used for linguistic analysis and NLP tasks.

9.2.1 Notation

- N = number of tokens (total word occurrences)
- V = vocabulary = set of types
- $|V|$ = size of vocabulary (number of unique words)

9.2.2 Herdan's Law (Heap's Law)

The relationship between vocabulary size and corpus size follows a power law:

$$|V| = kN^{\beta}$$

where:

- k is a constant
- β is typically between 0.67 and 0.75

Key insight: Vocabulary size grows with more than the square root of the number of tokens, but less than linearly. This means:

- As you add more text, you keep encountering new words
- But the rate of new words decreases as the corpus grows
- You never stop finding new words entirely

9.2.3 Examples from Real Corpora

Observations:

- **Switchboard**: Spoken language has relatively small vocabulary for its size (lots of repetition)
- **Shakespeare**: Rich vocabulary despite relatively small corpus

Corpus	Tokens (N)	Types (|V|)
Switchboard phone conversations	2.4 million	20 thousand
Shakespeare	884,000	31 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13+ million

Tabella 9: Token and type counts in various corpora

- **COCA** (Corpus of Contemporary American English): Large modern corpus with extensive vocabulary
- **Google N-grams**: Massive web-scale corpus with millions of unique words

Practical implications:

- Larger corpora are needed to capture rare words and phenomena
- The type/token ratio varies significantly across domains and genres
- Spoken language tends to have lower type/token ratios than written language
- Web-scale corpora contain many rare words, misspellings, and proper nouns

9.3 Understanding Corpora

9.3.1 Words Don't Appear Out of Nowhere

A fundamental principle in corpus linguistics: **words don't appear out of nowhere**. Every text is produced in a specific context.

A text is produced by:

- A specific writer(s)
- At a specific time
- In a specific variety (of language)
- Of a specific language
- For a specific function

Why this matters: Understanding the context of text production is crucial for interpreting corpus data and building NLP systems. The same word can have different meanings, frequencies, and usage patterns depending on these factors.

9.4 Corpora Vary Along Multiple Dimensions

Corpora are not uniform - they vary along several important dimensions that affect the language they contain.

9.4.1 Language

There are approximately **7,097 languages in the world**. Each has its own:

- Vocabulary and grammar
- Writing system (or lack thereof)
- Cultural context

- Computational resources available

Implication: NLP systems trained on one language don't automatically work for others. Multilingual NLP is a major research area.

9.4.2 Variety

Even within a single language, there are many varieties. For example:

- **African American English (AAE)** varieties
  - AAE Twitter posts might include forms like "*iont*" (I don't)
  - Different phonological, grammatical, and lexical features

Implication: Systems trained on standard varieties may perform poorly on other varieties, potentially causing bias and fairness issues.

9.4.3 Code Switching

Code switching occurs when speakers alternate between two or more languages within a conversation or even within a sentence.

Examples:

Spanish/English:

- "Por primera vez veo a @username actually being hateful! It was beautiful:)"
- Translation: "For the first time I get to see @username actually being hateful! it was beautiful:)"

Hindi/English:

- "dost tha or ra- hega ... dont wory ... but dherya rakhe"
- Translation: "he was and will remain a friend ... don't worry ... but have faith"

Challenges:

- Difficult for monolingual NLP systems
- Requires understanding of multiple languages simultaneously
- Common in multilingual communities but often ignored in NLP datasets

9.4.4 Genre

Different genres have distinct linguistic characteristics:

- **Newswire**: Formal, objective, structured
- **Fiction**: Narrative, dialogue, descriptive
- **Scientific articles**: Technical vocabulary, formal structure
- **Wikipedia**: Encyclopedic, informative, hyperlinked

Implication: A model trained on news articles may not work well on social media or scientific papers.

9.4.5 Author Demographics

The characteristics of the author affect the text:

- **Age**: Younger writers may use different slang and references
- **Gender**: Can influence topic choice and writing style
- **Ethnicity**: Affects cultural references and language variety
- **Socioeconomic status (SES)**: Influences vocabulary and topics

Implication: Demographic bias in training data can lead to biased NLP systems that work better for some groups than others.

9.5 Corpus Datasheets

To address the complexity of corpora, researchers have proposed **corpus datasheets** - standardized documentation for datasets.

Key references: Gebru et al (2020), Bender and Friedman (2018)

9.5.1 Motivation

Questions about why and how the corpus was created:

- **Why was the corpus collected?** What research question or application?
- **By whom?** Which researchers or organizations?
- **Who funded it?** Funding sources can influence corpus design

9.5.2 Situation

In what situation was the text written?

Understanding the context of text production:

- Social media posts vs. published articles
- Spontaneous speech vs. prepared presentations
- Private communications vs. public documents

9.5.3 Collection Process

If it is a subsample, how was it sampled? Was there consent? Pre-processing?

Important methodological details:

- **Sampling method**: Random, stratified, convenience?
- **Consent**: Did authors/speakers consent to their text being used?
- **Pre-processing**: What cleaning or normalization was applied?
- **Ethical considerations**: Privacy, representation, potential harms

9.5.4 Additional Information

Datasheets should also document:

- **Annotation process**: How was the data labeled? By whom?
- **Language variety**: Which dialect or variety is represented?
- **Demographics**: Who are the authors/speakers?
- **Limitations**: Known biases or gaps in the corpus

9.6 Why Corpus Datasheets Matter

Transparency: Researchers and practitioners need to understand what data they're working with.

Reproducibility: Proper documentation enables others to replicate and build on research.

Fairness: Understanding corpus composition helps identify and mitigate biases.

Appropriate use: Knowing the context and limitations prevents misuse of corpora and models trained on them.

Example concern: A sentiment analysis system trained on formal news text may fail on informal social media text, or work well for one demographic group but poorly for others. Datasheets help identify these issues before deployment.

Parte III

Text Normalization and Tokenization

10 Text Normalization

Text normalization is a fundamental preprocessing step required by every NLP task. It involves transforming raw text into a standardized format that can be processed by computational methods.

10.1 The Need for Text Normalization

Every NLP task requires text normalization.

When working with corpora, we need to consider several factors:

- **Corpus homogeneity**: A corpus is considered homogeneous when certain parameters are consistent (e.g., same language, same author, same genre)
- **Morphological complexity**: Due to morphology, it's often more reasonable to work with types rather than tokens
- **Language-specific properties**: Different languages require different normalization approaches

10.2 The Three Main Steps of Text Normalization

According to the slides, every NLP task requires three fundamental normalization steps:

1. **Tokenizing (segmenting) words**
    - Breaking text into individual word units
    - Particularly important for agglutinating languages
    - Determines the basic units of analysis
2. **Normalizing word formats**
    - Standardizing variations of the same word
    - Handling case differences, punctuation, etc.
    - Converting to canonical forms
3. **Segmenting sentences**
    - Dividing text into sentence units
    - Identifying sentence boundaries
    - Important for syntactic and semantic analysis

10.3 Linguistic Properties and Tokenization

10.3.1 Isolating vs. Agglutinating Languages

Different languages have different morphological properties that affect tokenization:

- **Isolating languages**: Languages where typically 1 token corresponds to 1 type

  - Minimal morphology (no conjugation, plurals, genders, pronouns)
  - Example: Japanese (highly isolated)
  - Each word tends to be a single, unchanging unit
- **Agglutinating languages**: Languages that combine multiple morphemes into single words
  - Complex morphology with many affixes
  - Single words can express what requires multiple words in other languages
  - Example: Japanese (also has agglutinating properties)

Key insight: The property of being isolating is important to recognize because it affects how we approach tokenization and normalization.

10.3.2 Tokens and Multi-tokens

In practice, we encounter:

- **Simple tokens**: Single word units
- **Multi-tokens**: Compound expressions that function as single units
  - Example: "New York", "machine learning"
  - Should these be treated as one token or multiple?
  - Decision depends on the application

10.4 Space-Based Tokenization

10.4.1 The Simple Approach

The most straightforward tokenization method:

Segment off a token between instances of spaces

This approach works for languages that use space characters to separate words:

- Arabic, Cyrillic, Greek, Latin, etc., based writing systems
- These languages explicitly mark word boundaries with spaces

10.4.2 Language Bias

Important limitation: Space-based tokenization has a significant language bias.

- **Works well for**: European languages and others using space-separated writing systems
- **Fails for**: Languages without spaces between words
  - Chinese, Japanese, Thai: No spaces between words
  - Requires different tokenization approaches
- **Directional bias**: Assumes left-to-right reading
  - Doesn't account for right-to-left languages (Arabic, Hebrew)
  - May need adjustment for bidirectional text

10.5 Unix Tools for Space-Based Tokenization

Unix provides powerful command-line tools for text processing, particularly suited for space-based tokenization.

10.5.1 The `tr` Command

The `tr` (translate) command is fundamental for tokenization:

Purpose: Transform characters in text, commonly used to separate words

Basic operation:

- Takes a file with words separated by spaces
- Generates a list of tokens (one per line)
- Pure tokenization without additional processing

10.5.2 Simple Tokenization in UNIX

Following Ken Church's "UNIX for Poets" approach, here's a complete tokenization pipeline:

```
tr -sc 'A-Za-z' '\n' < shakes.txt
    | sort
    | uniq -c
```

Step-by-step explanation:

1. `tr -sc 'A-Za-z' '\n' < shakes.txt`
    - `-s`: Squeeze repeated characters
    - `-c`: Complement the set (everything NOT in A-Za-z)
    - `'A-Za-z'`: Keep only alphabetic characters
    - `'\n'`: Replace all non-alphabetic characters with newlines
    - **Effect**: Changes all non-alpha characters to newlines, creating one word per line
2. `| sort`
    - Sorts all words in alphabetical order
    - Groups identical words together
    - Necessary for the next step
3. `| uniq -c`
    - `uniq`: Removes duplicate adjacent lines
    - `-c`: Counts occurrences of each unique line
    - **Effect**: Merges identical words and counts each type

Output format:

```
1945 A
  72 AARON
  19 ABBESS
   5 ABBOT
  ...
```

Each line shows the frequency count followed by the word type.

10.5.3 Example: The First Step (Tokenizing)

From the slides, the first step of the pipeline:

```
tr -sc 'A-Za-z' '\n' < shakes.txt | head
```

Output:

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
We
...
```

Each word appears on its own line, ready for further processing.

10.5.4 Example: The Second Step (Sorting)

After sorting the tokenized words:

```
tr -sc 'A-Za-z' '\n' < shakes.txt | sort | head
```

Output:

```
A
A
A
A
A
A
A
A
A
...
```

All identical words are now grouped together, preparing for counting.

10.6 The Bag of Words Model

10.6.1 Definition

The **bag of words** model is a simplified representation of text that:

- Treats text as an unordered collection of words
- Counts word frequencies
- Ignores word order, grammar, and structure

Mathematical representation: Similar to multisets in set theory

- Example: "aabbba" becomes $\{a : 3, b : 3\}$

- Each unique word (type) is associated with its count

10.6.2 Limitations of Bag of Words

Despite its widespread use, the bag of words model has significant limitations:

- **Loss of word order**
  - "The dog bit the man" vs "The man bit the dog" are identical
  - Cannot capture syntactic relationships
- **Ignores morphology**
  - "run", "runs", "running" are treated as completely different words
  - No understanding of word relationships
- **Problems with agglutinating languages**
  - Complex morphology creates explosion of unique forms
  - Loses semantic relationships between related forms
- **Directional bias**
  - Assumes left-to-right processing
  - Doesn't work well for right-to-left languages (Arabic, Hebrew)
- **No semantic understanding**
  - Cannot distinguish between different senses of the same word
  - No understanding of context or meaning

Despite these limitations, bag of words remains useful for:

- Document classification
- Information retrieval
- Simple text analysis tasks
- Baseline models for comparison

10.7 Practical Considerations

10.7.1 When to Use Space-Based Tokenization

Space-based tokenization is appropriate when:

- Working with languages that use spaces (English, Spanish, etc.)
- Need quick, simple preprocessing
- Computational resources are limited
- Task doesn't require sophisticated linguistic analysis

10.7.2 When More Sophisticated Methods Are Needed

More advanced tokenization is required for:

- Languages without space separation (Chinese, Japanese, Thai)
- Handling contractions and compound words
- Preserving multi-word expressions
- Dealing with social media text (hashtags, mentions, emojis)
- Subword tokenization for neural models (BPE, WordPiece)

11 Advanced Tokenization Techniques

11.1 More Counting: Merging Upper and Lower Case

When performing frequency analysis, we often want to treat uppercase and lowercase versions of the same word as identical.

Command:

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' | sort | uniq -c
```

Explanation:

- **First** `tr 'A-Z' 'a-z'`: Converts all uppercase letters to lowercase
  - This merges "The" and "the" into the same token
  - Reduces vocabulary size by eliminating case variations
- **Then tokenize, sort, and count** as before

Result: Words that differ only in capitalization are counted together.

11.2 Sorting the Counts

To find the most frequent words, we can sort by frequency instead of alphabetically.

Command:

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' | sort | uniq -c | sort -n -r
```

New component:

- `sort -n -r`: Sort numerically in reverse order
  - `-n`: Numerical sort (treats numbers as numbers, not strings)
  - `-r`: Reverse order (highest to lowest)

Output example:

```
23243 the
22225 i
18618 and
16339 to
15687 of
12780 a
12163 you
10839 my
```

```
10005 in
 8954 d
```

Question from the slide: "What happened here?"

The output shows the most frequent words in Shakespeare's works, with "the" appearing 23,243 times. Notice that:

- Function words (the, i, and, to, of, a) dominate the frequency list
- The letter "d" appears as a separate token (likely from contractions like "I'd")
- This demonstrates Zipf's law: a small number of words account for most occurrences

12 Issues in Tokenization

Tokenization is more complex than simply splitting on spaces. Several challenging cases require careful consideration.

12.1 Punctuation Cannot Be Blindly Removed

Many tokens contain punctuation that carries meaning and should not be automatically stripped:

- **Abbreviations**: m.p.h., Ph.D., AT&T, cap'n
  - Periods are integral to the abbreviation
  - Removing them would destroy the meaning
- **Prices**: $45.55
  - The period separates dollars from cents
  - The dollar sign indicates currency
- **Dates**: 01/02/06
  - Slashes separate day, month, and year
  - Different formats exist (US vs European)
- **URLs**: http://www.stanford.edu
  - Periods, slashes, and colons are structural
  - Should be kept as a single token
- **Hashtags**: #nlproc
  - The hash symbol is part of the token
  - Common in social media text
- **Email addresses**: someone@cs.colorado.edu
  - Contains periods and @ symbol
  - Should remain as a single token

Key insight: Punctuation removal must be context-aware. Blindly stripping all punctuation destroys important information.

12.2 Clitics

Clitics are words that don't stand on their own grammatically and attach to other words.

Examples:

- **English**: "are" in *we're*, "je" in French *j'ai*, "le" in *l'honneur*
  - These are grammatically separate words but phonologically attached
  - Question: Should "we're" be one token or two ("we" + "are")?

Tokenization decisions:

- **Keep as one token**: Preserves the surface form as written
- **Split into two**: Normalizes to grammatical units
- **Choice depends on**:
  - The NLP task (parsing might prefer splitting)
  - The language (different languages have different conventions)
  - Consistency with training data

12.3 Multiword Expressions (MWEs)

When should multiword expressions be treated as single words?

Examples:

- **Proper nouns**: New York
  - Refers to a single entity (the city)
  - "New" and "York" separately have different meanings
- **Idiomatic expressions**: rock 'n' roll
  - The meaning is not compositional
  - Cannot be understood from individual words

Challenges:

- **Identification**: How do we automatically detect MWEs?
- **Ambiguity**: "New York" could also be "new" + "York" in other contexts
- **Variability**: Some MWEs allow internal modification ("brand new car" vs "brand car")
- **Language-specific**: Different languages have different MWE patterns

Approaches:

- Use a dictionary of known MWEs
- Statistical methods to identify frequent collocations
- Named entity recognition for proper nouns
- Task-specific decisions based on application needs

13 Tokenization in NLTK

The Natural Language Toolkit (NLTK) provides sophisticated tokenization tools that handle many of the issues discussed above.

13.1 Regular Expression Tokenization

NLTK's `regexp_tokenize` function allows flexible tokenization using regular expressions.

Example from Bird, Loper and Klein (2009), *Natural Language Processing with Python*:

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...     | \w+(-\w+)*      # words with optional internal hyphens
...     | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...     | \.\.\.          # ellipsis
...     | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Pattern breakdown:

1. `([A-Z]\.)+`: Abbreviations
   - Matches sequences like "U.S.A."
   - Keeps periods as part of the token
2. **Words with optional internal hyphens** (`\w+(-\w+)*`):
   - Matches "poster-print" as a single token
   - Handles compound words
3. **Currency and percentages** (`\$?\d+(\.\d+)?%?`):
   - Matches "$12.40" and "82%"
   - Keeps monetary amounts together
4. **Ellipsis** (`\.\.\.`):
   - Treats "..." as a single token
   - Distinguishes from sentence-ending periods
5. **Separate punctuation tokens** (`[][.,;"'?():-_`]`):
   - These punctuation marks become individual tokens
   - Includes brackets, parentheses, quotes, etc.

Advantages of regex tokenization:

- Highly customizable for specific domains
- Can handle complex patterns
- Explicit rules make behavior predictable

- Can be tuned for different languages and text types

Disadvantages:

- Requires careful pattern design
- May not generalize well to new domains
- Can become complex and hard to maintain
- Order of patterns matters (first match wins)

14 Tokenization in Languages Without Spaces

14.1 The Challenge

Many languages (like Chinese, Japanese, Thai) don't use spaces to separate words!

Fundamental question: How do we decide where the token boundaries should be?

Example (Chinese):

- Written text: 我爱自然语言处理
- Possible tokenizations:
  - 我 / 爱 / 自然 / 语言 / 处理 (I / love / natural / language / processing)
  - 我 / 爱 / 自然语言 / 处理 (I / love / natural-language / processing)
  - 我 / 爱 / 自然语言处理 (I / love / natural-language-processing)
- Different segmentations can be valid depending on the interpretation
- No spaces in the original text to guide segmentation

14.2 Approaches to Word Segmentation

1. Dictionary-based methods:

- Use a lexicon of known words
- Find the longest matching words in the text
- **Problem**: Ambiguity when multiple segmentations are possible
- **Problem**: Out-of-vocabulary words (new terms, names)

2. Statistical methods:

- Learn word boundaries from segmented training data
- Use probabilistic models (e.g., Hidden Markov Models)
- Choose the most likely segmentation based on learned patterns
- Better handles ambiguity and unknown words

3. Neural methods:

- Use deep learning models (LSTMs, Transformers)
- Learn character-level representations
- Predict word boundaries as a sequence labeling task

- State-of-the-art performance for many languages

4. **Subword tokenization:**
   - Break words into smaller units (characters or subwords)
   - Used in modern neural NLP (BPE, WordPiece, SentencePiece)
   - Avoids the word segmentation problem entirely
   - Works across languages, including those with and without spaces

14.3 Language-Specific Challenges

Chinese:

- No spaces between words
- Characters can be words or parts of words
- Ambiguous segmentation is common
- Proper nouns and new terms are challenging

14.4 Word Tokenization in Chinese: A Detailed Look

Chinese Character Structure

Chinese words are composed of characters called **"hanzi"** (or sometimes just **"zi"**).

Key properties:

- Each character represents a **meaning unit** called a **morpheme**
- Each word has on average **2.4 characters**
- Deciding what counts as a word is **complex and not agreed upon**

The ambiguity problem: Unlike English where word boundaries are marked by spaces, Chinese text is continuous, making it unclear where one word ends and another begins.

14.5 Example: Multiple Segmentation Possibilities

Consider the Chinese sentence: 姚明进入总决赛 ("Yao Ming reaches the finals")

This sentence can be segmented in multiple valid ways:

1. **Three words (3-word segmentation):**
   - 姚明 / 进入 / 总决赛
   - YaoMing / reaches / finals
   - This treats "Yao Ming" as a single name token
2. **Five words (5-word segmentation):**
   - 姚 / 明 / 进入 / 总 / 决赛
   - Yao / Ming / reaches / overall / finals
   - This splits the name and breaks down compound words
3. **Seven characters (character-level, don't use words at all):**

- 姚 / 明 / 进 / 入 / 总 / 决 / 赛
- Yao / Ming / enter / enter / overall / decision / game
- This treats each character as a separate token

Key insight: Each segmentation is linguistically defensible, but they lead to different analyses and different vocabulary sizes.

14.6 Practical Approach: Character-Based Tokenization

Common solution: In Chinese, it's common to just treat each character (zi) as a token.

Advantages:

- **Very simple**: The segmentation step is trivial - just split into characters
- **No ambiguity**: No need to decide word boundaries
- **Consistent**: Works uniformly across all texts
- **Handles unknowns**: New words are automatically handled as character sequences

Disadvantages:

- Loses word-level semantic information
- Larger sequence lengths (more tokens per sentence)
- May not capture multi-character word meanings effectively

14.7 Comparison with Other Languages

Chinese: Character-based tokenization is simple and effective

- So the segmentation step is very simple
- Just split on character boundaries

Thai and Japanese: More complex word segmentation is required

- Cannot simply use character-level tokenization
- Thai: Requires understanding of word boundaries within continuous script
- Japanese: Must handle three writing systems and grammatical particles

Modern approach: The standard algorithms are **neural sequence models trained by supervised machine learning**

- Use deep learning (LSTMs, Transformers)
- Learn from annotated training data
- Predict word boundaries as a sequence labeling task
- Achieve state-of-the-art performance

14.8 Summary: Chinese Tokenization

- **Hanzi/zi**: Chinese characters are meaning units (morphemes)
- **Average word length**: 2.4 characters per word
- **Ambiguity**: Multiple valid segmentations exist for the same text

- **Practical solution**: Character-level tokenization is common and simple
- **Alternative**: Neural sequence models for word-level segmentation
- **Trade-off**: Simplicity vs. linguistic accuracy

Japanese:

- Mixes three writing systems (Hiragana, Katakana, Kanji)
- No spaces between words
- Highly agglutinative (words combine with particles)
- Writing system can provide some segmentation clues

Thai:

- No spaces between words
- Spaces indicate phrase or sentence boundaries
- Requires specialized segmentation tools

14.9 Practical Tools

Several tools are available for segmenting languages without spaces:

- **Jieba** (Chinese): Popular open-source Chinese segmenter
- **MeCab** (Japanese): Morphological analyzer and tokenizer
- **PyThaiNLP** (Thai): Thai language processing toolkit
- **Stanford Word Segmenter**: Multi-language statistical segmenter
- **SentencePiece**: Language-agnostic subword tokenizer

15 Subword Tokenization

15.1 Motivation

Instead of white-space or single-character segmentation, we can **use the data to tell us how to tokenize**.

Subword tokenization: Tokens can be parts of words as well as whole words.

Advantages:

- Handles unknown words
- Reduces vocabulary size
- Captures morphology
- No out-of-vocabulary problem

15.2 Three Common Algorithms

1. **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016) - Used in GPT
2. **Unigram Language Modeling** (Kudo, 2018) - Used in T5
3. **WordPiece** (Schuster and Nakajima, 2012) - Used in BERT

All have 2 parts:

1. **Token learner**: Takes training corpus, induces vocabulary
2. **Token segmenter**: Tokenizes new text using that vocabulary

15.3 Byte-Pair Encoding (BPE)

Initial vocabulary: All individual characters $V = \{A, B, C, \ldots, a, b, c, \ldots\}$

Algorithm - Repeat k times:

1. Choose the two symbols most frequently adjacent in corpus (e.g., 'A', 'B')
2. Add merged symbol 'AB' to vocabulary: $V \leftarrow V \cup \{\text{AB}\}$
3. Replace every 'A' 'B' in corpus with 'AB'

Parameter k controls vocabulary size.

15.4 BPE Algorithm Pseudocode

Algorithm 4 Byte-Pair Encoding

function BPE(corpus C, merges k) **returns** vocab V
$V \leftarrow$ all unique characters in C
for $i = 1$ to k **do**
  $t_L, t_R \leftarrow$ Most frequent adjacent pair in C
  $t_{NEW} \leftarrow t_L + t_R$
  $V \leftarrow V + t_{NEW}$
  Replace t_L, t_R in C with t_{NEW}
end for
return V

15.5 BPE Practical Details

Most subword algorithms run inside space-separated tokens.

Common approach:

1. Add end-of-word symbol '__' before spaces
2. Separate into letters
3. Run BPE (merges don't cross word boundaries)

Example: "the cat" → "the__ cat__" → "t h e __ c a t __"

15.6 BPE Example: Step by Step

Original corpus:

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

After adding end-of-word tokens (__), initial vocabulary:

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

Initial corpus state:

```
Frequency  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 __
```

Iteration 1: Merge `e r` → `er`

Most frequent adjacent pair is 'e' 'r' (appears 9 times: 6 in "newer" + 3 in "wider")

```
Vocabulary: __, d, e, i, l, n, o, r, s, t, w, er
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 __
```

Iteration 2: Merge `er __` → `er__`

Most frequent adjacent pair is 'er' '__' (appears 9 times)

```
Vocabulary: __, d, e, i, l, n, o, r, s, t, w, er, er__
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 __
```

Iteration 3: Merge `n e` → `ne`

Most frequent adjacent pair is 'n' 'e' (appears 8 times: 6 in "newer" + 2 in "new")

```
Vocabulary: __, d, e, i, l, n, o, r, s, t, w, er, er__, 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 __
```

Subsequent merges:

- **(ne, w)**: → `new` - Vocabulary adds: `new`
- **(l, o)**: → `lo` - Vocabulary adds: `lo`
- **(lo, w)**: → `low` - Vocabulary adds: `low`
- **(new, er__)**: → `newer__` - Vocabulary adds: `newer__`
- **(low, __)**: → `low__` - Vocabulary adds: `low__`

Final vocabulary includes:

`__, d, e, i, l, n, o, r, s, t, w, er, er__, ne, new, lo, low, newer__, low__`

Final corpus:

```
5          low__
2          low e s t __
6          newer__
3          w i d er__
2          new __
```

Key observations:

- Frequent words become single tokens (`low__`, `newer__`)
- Rare words remain as character sequences (`w i d er__`)
- Common morphemes are learned (`er`, `new`)
- Vocabulary grows incrementally with each merge

15.7 BPE Token Segmenter Algorithm

Once we have learned the vocabulary, we use it to tokenize new test data.

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

- **Greedily**
- **In the order we learned them**
- **Test frequencies don't play a role**

Process: Apply merges sequentially - merge every `e r` to `er`, then merge `er __` to `er__`, etc.

Examples:

- Test word "`n e w e r __`" would be tokenized as a full word: `newer__`
  - Applies merges: `e r` → `er`, then `er __` → `er__`, then `n e` → `ne`, then `ne w` → `new`, finally `new er__` → `newer__`
- Test word "`l o w e r __`" would be two tokens: `low er__`
  - Applies merges: `e r` → `er`, then `er __` → `er__`, then `l o` → `lo`, then `lo w` → `low`
  - Result: `low` and `er__` remain as separate tokens (no merge learned for this combination)

Key point: The segmenter applies merges deterministically in the learned order, regardless of frequencies in the test data.

15.8 Properties of BPE Tokens

BPE tokens have useful linguistic properties:

Usually include:

- **Frequent words**: Common words become single tokens
- **Frequent subwords**: Often morphemes like *-est* or *-er*

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

Example: The word *unlikeliest* has 3 morphemes:

- *un-*: prefix meaning "not"
- *likely*: root word

- *-est*: suffix meaning "most"

Why this matters:

- BPE naturally discovers morphological structure
- Related words share subword components
- Example: *play*, *playing*, *played* share the *play* subword
- Enables better generalization across morphological variants

15.9 Modern Usage

Subword tokenization is standard in modern NLP:

- **GPT**: BPE
- **BERT**: WordPiece
- **T5**: Unigram
- Most transformer models use subword tokenization

Why it works: Balances vocabulary size and sequence length, handles morphology, no unknown words problem.

16 Word Normalization

After tokenization, we often need to normalize words to standard forms. Word normalization involves putting words/tokens in a standard, canonical form.

Common examples:

- **U.S.A.** or **USA** - Standardizing abbreviations
- **uhhuh** or **uh-huh** - Normalizing disfluencies
- **Fed** or **fed** - Case normalization
- **am, is, be, are** - Lemmatization to base form

16.1 Case Folding

Case folding means reducing all letters to lower case.

Applications like Information Retrieval (IR):

- Users tend to use lower case in queries
- Improves recall by matching regardless of case

Possible exceptions - Upper case in mid-sentence may be meaningful:

- *General Motors* - Proper noun (company name)
- *Fed* vs. *fed* - Federal Reserve vs. past tense of "feed"
- *SAIL* vs. *sail* - Acronym vs. common word

For sentiment analysis, MT, Information Extraction:

- Case is helpful (*US* versus *us* is important)

- Trade-off between generalization and precision

16.2 Lemmatization

Lemmatization represents all words as their lemma - their shared root or dictionary headword form.

The lemma is the canonical form of a word, typically the form you would look up in a dictionary.

Examples:

- *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*

Benefits: Reduces vocabulary size, groups related word forms together, useful for search and IR.

16.3 Lemmatization by Morphological Parsing

Lemmatization is typically done through **morphological parsing**.

Morphemes are the small meaningful units that make up words:

- **Stems**: The core meaning-bearing units
  - Example: *cat* in "cats", *walk* in "walking"
- **Affixes**: Parts that adhere to stems, often with grammatical functions
  - Prefixes: *un-*, *re-*, *pre-*
  - Suffixes: *-s*, *-ing*, *-ed*, *-er*

Morphological parsers analyze words into their component morphemes:

- Parse *cats* into: *cat* (stem) + *s* (plural suffix)
- Parse Spanish *amaren* ('if in the future they would love') into:
  - Morpheme: *amar* 'to love' (stem)
  - Features: 3PL and future subjunctive

Challenges: Irregular forms (*went* → *go*), ambiguity (*saw*), language-specific rules.

16.4 Stemming

Stemming reduces terms to stems, chopping off affixes crudely.

Difference from lemmatization:

- Stemming is simpler and faster
- May produce non-words
- Doesn't use morphological analysis
- Just applies rules to remove suffixes/prefixes

Original text	After stemming
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

Example comparison:

Notice: Words are reduced to stems that may not be valid words (*Billi*, *accur*, *copi*).

16.4.1 Porter Stemmer

The most common stemming algorithm for English.

Based on a series of rewrite rules run in series:

- A cascade, in which output of each pass fed to next pass
- Rules applied sequentially

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*)

Characteristics:

- Fast and simple
- Language-specific (different rules for each language)
- May over-stem or under-stem
- Widely used despite imperfections

16.5 Complex Morphology in Other Languages

Dealing with complex morphology is necessary for many languages.

Example: Turkish word

Uygarlastiramadiklarimizdanmissiniz

Meaning: '(behaving) as if you are among those whom we could not civilize'

Morphological breakdown:

- *Uygar* 'civilized' + *las* 'become'
- + *tir* 'cause' + *ama* 'not able'
- + *dik* 'past' + *lar* 'plural'
- + *imiz* 'p1pl' + *dan* 'abl'
- + *mis* 'past' + *siniz* '2pl' + *casina* 'as if'

Key insight: Languages like Turkish, Finnish, Hungarian have highly agglutinative morphology where many morphemes combine into single words. Simple stemming is insufficient - proper morphological analysis is essential.

17 Sentence Segmentation

The third step of text normalization (after tokenization and word normalization) is segmenting text into sentences.

17.1 The Challenge

!, ? mostly unambiguous - Usually mark sentence boundaries

Period "." is very ambiguous:

- **Sentence boundary**: "I went home. She stayed."
- **Abbreviations**: Inc., Dr., Mr., U.S.A.
- **Numbers**: .02%, 4.3

17.2 Common Algorithm

Tokenize first: Use rules or ML to classify a period as either:

1. **(a) Part of the word** (abbreviation, number)
2. **(b) A sentence boundary**

Helpful resource: An abbreviation dictionary can help identify periods that are part of abbreviations.

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

17.3 Rule-Based Approach

Common rules:

- Period followed by uppercase letter usually indicates sentence boundary
- Period at end of line is usually sentence boundary
- Period followed by known abbreviation is not sentence boundary
- Use whitelist of common abbreviations (Dr., Inc., etc.)

17.4 Machine Learning Approach

Features for classification:

- Word before period
- Word after period
- Capitalization of following word
- Length of words around period
- Presence in abbreviation dictionary

Models: Decision trees, logistic regression, or neural networks can be trained on annotated data.

17.5 Challenges

- **Abbreviations at end of sentence**: "The company is called Yahoo Inc." (period serves both functions)
- **Quotations**: "He said, 'I'm leaving.' Then he left."
- **Multiple punctuation**: "Really?! No way!"
- **Ellipsis**: "And then..."
- **Domain-specific**: Scientific text, legal documents have different conventions