

## ✓ Basic Text Processing Problems

### Fundamentals of Natural Language

#### Authors:

Sonia Espinilla, 1708919

Agustina Inés Lazzati, 1707964

*Autonomous University of Barcelona*

*March 2, 2025*

---

#### ✓ Graded Exercise 1 (1 point)

*You are given the Regular Expression `/.+(.+)/`. Assert whether it is valid. If the answer is affirmative, explain what would be captured and why?*

To determine the validity of a Regular Expression (regex), we need to analyze its structure and behavior in depth. As we know, the dot `.` symbol matches any character except for line terminators. The plus `+` symbol, as a quantifier, consumes as many characters as possible within the input pattern, making it greedy. In Speech and Language Processing, it is stated that this symbol matches "one or more occurrences of the previous character or expression." [1] Accordingly, the first greedy component of the expression, `.+`, will match one or more occurrences of any character, capturing the entire text.

The expression also includes parentheses, which store a pattern in memory, forming a capture group. So, `(.+)` attempts to capture any character one or more times. However, since the initial greedy segment `.+` has already matched the entire input, no characters remain for the capture group. Consequently, this expression captures only the last character of the sentence.

However, a problem arises when this expression is applied to a single-character input. Since both `.+` and `(.+)` require at least one character, there are no remaining characters for the capture group. As a result, the regex fails to match the string.

---

#### ✓ Graded Exercise 2 (3 points)

*Bartleby is a Scrivener whose keyboard is acting up. Every so often, the keyboard duplicates some of his last keystrokes, threatening to render an entire day of work*

*unusable. Write a regular expression substitution pattern that can clean up this text of errors:*

*Noththing so aggravavatestes an earnestst person as a passissive resistancece. If thethe individual so resisteded be ofof a notot inhumane temperer, andnd the resistingng onone perfectctly harmless in his passissivityity; then, in the bettetter moodsd of the formerer, he willwill endeavoror chacharitablyly to conconstrueue to his imaginanationon whatat provesves impmpossible to be solvled by his judgmentnt.*

*You may use the clean text as reference and you can compute the String Edit Distance between both texts to evaluate your progress.*

*Nothing so aggravates an earnest person as a passive resistance. If the individual so resisted be of a not inhumane temper, and the resisting one perfectly harmless in his passivity; then, in the better moods of the former, he will endeavor charitably to construe to his imagination what proves impossible to be solved by his judgment.*

Aiming to solve this issue, where the last keystrokes are often duplicated by the keyboard, we tried to find a regular expression substitution pattern that matches these duplicated tokens and eliminates them. After attempting to solve this problem, we realized that our pattern matching must be strict enough to exclude the repetition of single characters, as in the case of "harmless," where the double "s" must not be removed. This is why we ended up with the following regular expression: `s/(\w*?)(\w{2,})\1\2+/\1\2/`.

With respect to the pattern matching regex, our expression captures two groups. The first, `(\w*?)`, captures as few "word characters" as possible due to the non-greedy nature of the `?` symbol. The second capturing group, `(\w{2,})`, matches at least two word characters. Finally, we add `\1\2+` to match the word from the first capturing group, followed by one or more occurrences of the word captured in the second group.

On the other hand, the replacement pattern is composed of `\1\2`, which concatenates the first and second capturing groups, removing the repeated occurrences of the word.

```
import re
from Levenshtein import distance as levenshtein_distance

text = """
Noththing so aggravavatestes an earnestst person as a passissive resistancece. If thethe
"""

final_text = """
Nothing so aggravates an earnest person as a passive resistance. If the individual so res
"""

pattern = r'(\w*?)(\w{2,})\1\2'
```

```

replacement = r'\1\2'

cleaned = re.sub(pattern, replacement, text)

edit_distance = levenshtein_distance(cleaned, final_text)

print("Cleaned Text:")
print(cleaned)
print("\nEdit Distance:", edit_distance)

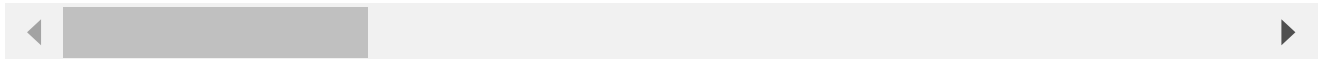
```



Cleaned Text:

Nothing so aggravates an earnest person as a passive resistance. If the individual so

Edit Distance: 0



### ✓ Graded Exercise 3 (3 points)

*Implement the Porter Stemmer rules seen on the slides using regular expressions. Apply this stemmer on the following text and show the results. Show the code in the submission.*

*When I was young, it seemed that life was so wonderful*

*A miracle, oh, it was beautiful, magical*

*And all the birds in the trees, well they'd be singing so happily Oh, joyfully, oh, playfully watching me*

*But then they sent me away to teach me how to be sensible*

*Logical, oh, responsible, practical*

*Then they showed me a world where I could be so dependable*

*\*Oh, clinical, oh, intellectual, cynical \**

```

import re

def apply_rules(text, rules):
    for reg_expression, substitution in rules:
        text = re.sub(reg_expression, substitution , text)
    return text

# Step 1a: Plural
step_1a = [
    (r'sses\b', 'ss'),
    (r'ies\b', 'i'),
    (r'(?<!s)s\b', '')
]

```

```

# Step 1b: Conjugation
step_1b = [
    (r'ing\b', ''),
    (r'ed\b', '')
]

# Step 2: Long Stems
step_2 = [
    (r'ational\b', 'ate'),
    (r'izer\b', 'ize'),
    (r'ator\b', 'ate')
]


# Step 3: Longer Stems
step_3 = [
    (r'al\b', ''),
    (r'able\b', ''),
    (r'ate\b', '')
]

text = """When I was young, it seemed that life was so wonderful
A miracle, oh, it was beautiful, magical
And all the birds in the trees, well they'd be singing so happily
Oh, joyfully, oh, playfully watching me
But then they sent me away to teach me how to be sensible
Logical, oh, responsible, practical
Then they showed me a world where I could be so dependable
Oh, clinical, oh, intellectual, cynical """

# Apply each step in order
text = apply_rules(text, step_1a)
text = apply_rules(text, step_1b)
text = apply_rules(text, step_2)
text = apply_rules(text, step_3)

print(text)

```

 When I wa young, it seem that life wa so wonderful  
 A miracle, oh, it wa beautiful, magic  
 And all the bird in the tree, well they'd be sing so happily  
 Oh, joyfully, oh, playfully watch me  
 But then they sent me away to teach me how to be sensible  
 Logic, oh, responsible, practic  
 Then they show me a world where I could be so depend  
 Oh, clinic, oh, intellectu, cynic

In our final result, we must consider that we have only applied the basic steps of the Porter Stemmer algorithm. As a result, some words, such as beautiful or joyfully, remain in their full form instead of being reduced to their root forms, that is the primary function of the algorithm, but this limitation highlights its basic nature.

Additionally, we have detected some issues. As it is said in the book *Speech and Language Processing*: "The algorithm is based on rewrite rules run in series, with the output of each pass fed as input to the next pass"[1]. That is why, for example, the word was appears as "wa" in our text because the algorithm mistakenly removed the final s, interpreting it as a plural suffix, over-generalizing. A more advanced stemming or lemmatization method should account for such irregularities in the English language. To improve precision, an advanced algorithm should incorporate a dataset or dictionary of irregular verbs providing also grammatical categories of words. This would allow the algorithm to recognize words like was as a verb and either reduce it to its base form (be) or retain the final "s" when appropriate, avoiding incorrect transformations.

---

### ✓ Graded Exercise 4 (3 points)

*Bartleby's keyboard is acting up again, this time in a much more unpredictable way. It turns out that this keyboard, which sends ASCII characters to the computer, will randomly flip one of the bits of the sent signal. It is such an annoying bug in fact that every time that it happens, it also switches the parity bit accordingly, so it cannot be detected easily.*

*We are trying to implement a spelling corrector for it which depends on a string edit distance algorithm. Implement a distance heuristic that can help correct this phenomenon. You may restrict yourself to alphabetic characters.*

*Help: You can check the binary representation of ASCII here . If you flip a bit from the letter A: 0b0100 0001 you can obtain @: 0b0100 0000 , C: 0b0100 0011 , ...*

The problem presents a case where some bits randomly flip, changing characters according to their ASCII values. To solve this issue, we should implement a heuristic based on string edit distance. It is said that: "The edit distance between two strings s and t is a measure of how many operations are required to transform one string into another. There are several ways to compute edit distance, but one of the most popular is the Levenshtein edit distance, which counts the minimum number of insertions, deletions, and substitutions." [2]

In this case, some characters will be more probable than others. Specifically, characters that differ by only a single bit will have a higher likelihood of being the correct character. This is why we should implement a weighted edit distance algorithm that penalizes characters whose ASCII values differ by more than one bit and those that, based on context, have a lower probability of appearing. This cost function will guide us in identifying the most likely characters.

In the first step of our algorithm, we apply a tokenization process to determine which tokens have a low probability of being correct based on the context of the sentence being read. This involves creating a dataset, like a dictionary, that maps words to their correct forms for error correction.

When we identify an issue with sentence structure or the incorrect usage of words, we can make the necessary corrections.

For the incorrect word, we separate it into individual letters and convert each character to binary. Restricting ourselves to alphabetic characters (from 01000001 to 01111010 in binary), we compute our heuristic by creating a matrix based on the edit distance, where characters that differ by only one flipped bit will have a heuristic value of 1, while characters that differ by more than one bit will be assigned an infinite heuristic. For example:

```
# If only one bit differs.

Incorrect Word = "Hnla"
Correct Word = "Hola"
o = 0110 1110      # Incorrect character

# Alternative characters
o = 0110 1111
m = 0110 1101
k = 0110 1011
g = 0110 0111
```

To determine the correct character, we will also penalize tokens that are less likely to appear based on context using a perplexity model, which evaluates the probability of a token's occurrence within the given sentence. In the example proposed, this will add more weight to the letters m, k and g, being "o" the character selected by our heuristic.

---

## Bibliography

- [1] Jurafsky, D., & Martin, J. H. (2023). Speech and Language Processing (3rd ed.). Recuperado de [https://web.stanford.edu/~jurafsky/slp3/ed3book\\_Jan25.pdf](https://web.stanford.edu/~jurafsky/slp3/ed3book_Jan25.pdf).
- [2] Eisenstein, J. (2018). Natural Language Processing. Recuperado de <https://cseweb.ucsd.edu/~nnakashole/teaching/eisenstein-nov18.pdf>.

