L2: Basic Text Processing Problems

Frederik Evenepoel & Piotr Bonar - Pair K

2/03/2025

Exercise 1: You are given the Regular Expression /.+(.+)/. Assert whether it is valid. If the answer is affirmative, explain what would be captured and why.

Explanation:

The regular expression */.+(.+)/* is valid because it follows the correct syntax and matches any string with at least two characters. It captures only the last character of the text due to the greedy behavior of the first .+, which consumes most of the text before the capture group.

Working principle of regular expression:

1. . + → Matches one or more of any character.
2. *(. +)* → A capturing group that also matches one or more of any character, storing it in the first numbered capture group.
3. Greedy matching behavior of .+ → The first .+ initially consumes the entire input string. However, since *(.+)* must also match at least one character, the first .+ gives back just enough for *(.+)* to capture the last possible portion of the input string.

But there is a limitation: Since '.+' requires at least one character, and '*(.+)*' also requires at least one, the regex will only match strings with at least two characters. A single-character string or an empty string will not match at all.

Code example:

```
import re

pattern = r".+(.+)"
test_strings = ["hello", "JohnDoe", "hello world"]

for test in test_strings:
    match = re.match(pattern, test)
    if match:
        print(f"Input: {test}\nFull Match: {match.group(0)}\nCaptured Group: {match.group(1)}")
```

```
output:

Input: hello, Full Match: hello, Captured Group: o

Input: JohnDoe, Full Match: JohnDoe, Captured Group: e

Input: hello world, Full Match: hello world, Captured Group: d
```

Exercise 2: Correcting repeated keystroke errors using regular expressions

Explanation:

In the following exercise, we are required to construct a regular expression and apply it within a substitution to clean the distorted text. The pattern used in the substitution to find the repeated text patterns is:

([A-Za-z]{2,})\1

- *()* → Capturing group that stores a match for later reference.
- *[A-Za-z]* → Matches any uppercase or lowercase letter.
- *{2,}* → Ensures the match contains at least two consecutive letters, preventing single-letter repetitions from being mistakenly detected, like in ‘aggravates’ for instance.
- *\1* → Check if the captured sequence is immediately repeated in the text. We can tell right away from the distorted text that this is the case and what we are looking for.

We then use this pattern in a substitution:

- The first argument is the regular expression *([A-Za-z]{2,})\1*
- The second argument in the substitution, *\1*, refers to the first capturing group, replacing the duplicated segment..
- The third argument is the input text.

Code example

```
import re
text = """Noththing so aggravavatestes an earnestst person as a passissive resistancece. If
thethe indivividual so resisteded be ofof a notot inhumane temperer, andnd the
resistingng onone perfectctly harmless in his passissivityity; then, in the bettetter
moodsds of the formerer, he willwill endeavoror chacharitablyly to conconstrueue to
his imaginanationon whatat provesves impmpossible to be solvlved by his judgmentnt"""

result = re.sub(r'([A-Za-z]{2,})\1', r'\1', text)

print("Corrected text:", result)
```

```
Corrected text: 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
```

String edit distance

We can now apply the Levenshtein algorithm to compare the corrected text with the given correct text

```
import re
import Levenshtein

correct = """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.
"""

modified_words = re.findall(r'\b\w+\b', result)
correct_words = re.findall(r'\b\w+\b', correct)

total_distance = sum(Levenshtein.distance(m, c) for m, c in zip(modified_words, correct_words))

print("Total Levenshtein distance (word by word):", total_distance)
Total Levenshtein distance (word by word): 0
```

The cumulative Levenshtein distance of 0 between corresponding word pairs in two texts means that every word pair is identical. This shows that the texts are syntactically equivalent, having the same words in the same order, without considering factors like whitespace or punctuation. The substitution pattern therefore does his job on this text.

Instead of importing Levenshtein, we can implement the algorithm itself in the following format (like given on page 25 of the handbook), assuming a cost of 1. Applying this algorithm on our problem, we get again a cost of 0.

```
def minedit_distance(edited_word, target_word):
    n = len(edited_word)
    m = len(target_word)

    D = [[0] * (m + 1) for _ in range(n + 1)]

    D[0][0] = 0
    for i in range(1, n + 1):
        D[i][0] = D[i - 1][0] + 1
    for j in range(1, m + 1):
        D[0][j] = D[0][j - 1] + 1

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            D[i][j] = min(
                D[i - 1][j] + 1,
                D[i][j - 1] + 1,
                D[i - 1][j - 1] + (0 if edited_word[i - 1] == edited_word[j - 1] else 1)
            )

    return D[n][m]
```

Exercise 3: 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.

Explanation:

The Porter Stemmer rules in the slides show a clear structure: it starts with handling plurals, then moves on to verb conjugations, followed by rules for longer stems, and finishes with more rules for even longer stems.

The substitution lines first look for a specific ending in a word. When a match is found, it gets replaced based on the predefined rule, like shown in the slides, changing the word's structure as required.

Code example:

```
import re
def porter(word): #Usage of rules in lesson
   # First step
   word = re.sub('sses$', 'ss', word)
   word = re.sub('ies$', 'i', word)
   word = re.sub('ss$', 'ss', word)
```

```
    word = re.sub('s$', '', word)
    # Second step
   word = re.sub('(\w*[aeiou]\w*)ing$', r'\1', word)
    word = re.sub('(\w*[aeiou]\w*)ed$', r'\1', word)
    # Third step
    word = re.sub('ational$', 'ate', word)
    word = re.sub('izer$', 'ize', word)
    word = re.sub('ator$', 'ate', word)
    # Final step
    word = re.sub('al$', '', word)
    word = re.sub('able$', '', word)
    word = re.sub('ate$', '', word)
    return word

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"""

words = re.findall(r"\b[a-zA-Z]+\b", text)
stemmed_words = [porter(word.lower()) for word in words]

print(" ".join(stemmed_words))
```

```
Output: 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
```

Exercise 4: Edit distance heuristic for bit-flipped keyboard error correction

Explanation:

The code below corrects text errors caused by random bit flips in characters, where a single binary digit change alters the ASCII code. Since these errors happen occasionally and is not a fault by the user, it uses a weighted edit distance approach that assigns lower costs to these.

To calculate the bit flip cost, we first compare characters at the binary level. If two characters differ by only a single bit in their ASCII representation, we count the substitution as a minor error with a reduced cost of 0.5. If multiple bits are different, we assign the standard cost of 1, while identical characters have no cost. For example, in the traditional Levenshtein distance, changing “Anna” to “Cnna” would have a cost of 1. But since ‘A’ and ‘C’ differ by just one bit in their ASCII codes, the cost is lowered to 0.5, treating it as a minor error from a bit flip. This approach improves the standard Levenshtein distance by distinguishing between regular substitutions and non intended bit-flip errors. If we treat user mistakes and random bit flip errors the same, the algorithm might overcorrect text too aggressively, even when the difference was just a tiny unintended bit flip.

The code creates a table using modified Levenshtein distance rules and then works backward from the bottom-right corner to find the right corrections. It focuses on diagonal moves for substitutions, only changing characters when a bit flip happens (cost 0.5), and ignores insertions and deletions to fix words like “hcllo” back to “hello” without messing up the structure.

Code example:

```
def bit_flip_cost(char1, char2):

    if char1 == char2:
        return 0
    xor = ord(char1) ^ ord(char2)
    return 0.5 if bin(xor).count('1') == 1 else 1

def weighted_edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    print(dp)

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = bit_flip_cost(s1[i - 1], s2[j - 1])

            dp[i][j] = min(
                dp[i - 1][j] + 1,
                dp[i][j - 1] + 1,
                dp[i - 1][j - 1] + cost
            )

    return dp

def backtrace_correction(s1, s2, dp):
    i, j = len(s1), len(s2)
    corrected_word = list(s1)

    while i > 0 and j > 0:
        if dp[i][j] == dp[i - 1][j - 1] + bit_flip_cost(s1[i - 1], s2[j - 1]):
            if bit_flip_cost(s1[i - 1], s2[j - 1]) == 0.5:
                corrected_word[i - 1] = s2[j - 1]
            i -= 1
            j -= 1
        elif dp[i][j] == dp[i - 1][j] + 1:
            i -= 1
        elif dp[i][j] == dp[i][j - 1] + 1:
            j -= 1

    return "".join(corrected_word)
```

```
def weighted_edit_corrector(s1, s2):
    dp = weighted_edit_distance(s1, s2)
    return backtrace_correction(s1, s2, dp)

# Test cases
s1 = "hello"
s2 = "hcllo"

corrected = weighted_edit_corrector(s1, s2)
print(f"Original: {s2}, Corrected: {corrected}")
```

```
Output: Original: hcllo, Corrected: hello
```