Arabic Spell Checker with NLP Integration

Jalila Mouadi [1], Maryan Kassis [2], Obada Hattab [3]
Department of Electrical and Computer Engineering, Birzeit University [1,2,3]
1201611@student.birzeit.edu [1], 1200861@student.birzeit.edu [2], 1171616@student.birzeit.edu [3]

***Abstract*— The objective of this project is to create an Arabic spell checker that is more accurate and efficient by using natural language processing (NLP) techniques. Spell checkers are designed to detect misspelled words and offer recommendations based on context and linguistic patterns. To enhance correction recommendations, methods including tokenization, stemming, and part-of-speech tagging will be used. Utilizing well-known Arabic language corpora, performance will be examined and evaluated with current spell checkers.**

Keywords*— *Arabic Spell Checker, Natural Language Processing, Levenshtein Distance, n-gram, Tokenization, Stemming, Part-of-Speech Tagging

I. Introduction

With the increase in the usage of digital communication, particularly for languages as complicated as Arabic, the vital for effective spell checks are now essential. This project creates an Arabic spell checker by utilizing NLP methods. The n-gram technique and the Levenshtein distance are the two primary methods used. Tokenization, stemming, and part-of-speech tagging are some of the ways the spell checker uses to provide correction recommendations that are more precise and context-sensitive.

II. Methodology

A. Dataset Creation

A dictionary data set with more than 890000 words, was found for the Levenstein algorithm. And 8660 lines of corpora were found to use a data set for the n-gram model. Both data sets were found on the internet for different projects that had been published earlier on the web.

B. Levenshtein Distance

Levenshtein distance is a string metric that is used to calculate how different two sequences are from one another. The minimum number of single-character edits—that is, insertions, deletions, or substitutions—needed to transform one word into another is known as the Levenshtein distance between two words.
Lev a,b (| a | |, | b |) is the mathematical formula for the Levenshtein distance between two strings, a, b (of lengths | a | and | b |, respectively).

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise,} \end{cases}$$

C. N-gram Model

An n-gram is a simple and clear formalization neural language model, which means a probabilistic model that estimates the word probability given n-1 previous words, and consequently provides a probability to a whole sequence: a 2-gram (bigram) is a two-word sequence of words, and a 3-gram (a trigram) is a three-word sequence of words.

As an example, the sentence: "ذهب علي إلى الجامعة".
2-gram: "ذهب علي"، "علي إلى"، "إلى الجامعة"
3-gram: "ذهب علي إلى"، "علي إلى الجامعة"

D. Natural Language Processing Techniques

1. Tokenization

Tokenization allows one to examine the frequency of terms throughout the whole document, by dividing the text into separate tokens. Consequently, this serves as a basis for developing models that utilize these word frequencies. Tokenization also allows tagging tokens based on the sort of word they are, a feature we shall explore further in the discussion on Part of Speech Tagging.

2. Part-of-Speech Tagging

POS tagging is an essential linguistic process, where words are labeled according to their respective word types, such as nouns, verbs, adverbs, adjectives, and more, In a given text.

3. Stemming

In natural language processing (NLP), stemming is a text preprocessing approach that reduces words to their root or base form. Stemming algorithms generally function by replacing or removing word prefixes or suffixes according to a predetermined set of guidelines or heuristics. Simplifying and standardizing words is the aim of stemming, which enhances the efficiency of text categorization, information retrieval, and other NLP activities.

As an example, the terms "reading," "reader," and "reads," for instance, would all be reduced to their stem "read" by stemming. This enables the NLP model to identify that despite the differences in form, these words have a similar idea or meaning.

III. IMPLEMENTATION

A. Graphical User Interface

Tkinter was used in the development of a user-friendly GUI. Users may submit phrases, get their spelling checked, and get suggestions for corrections.

```
# Create the main window
root = tk.Tk()
root.title("Spell Checker Program")

# Calculate position to center the window on the screen
window_width = 400
window_height = 200
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()
center_x = int(screen_width / 2 - window_width / 2)
center_y = int(screen_height / 2 - window_height / 2)
root.geometry(f'{window_width}x{window_height}+{center_x}+{center_y}')

# Set the background color for the main window
root.configure(background='light blue')

# Define custom font style for the label
custom_font = tkFont.Font(family="Times New Roman", size=15, weight="bold")

# Create a label for the window with a different background color
welcome_label = tk.Label(root, text="Welcome to our Spell Checker program", bg='light blue', font=custom_font)
welcome_label.pack(pady=20)

# Define button dimensions and styles
button_width = 22
button_height = 2
button_color = 'light gray'
button_foreground = 'black'
custom_font1 = tkFont.Font(family="Times New Roman", size=11, weight="bold")

# Create a button for the Leven algorithm
leven_button = tk.Button(root, text="Levenshtein Distance", command=on_leven_click, width=button_width,
                         height=button_height, bg=button_color, fg=button_foreground, font=custom_font1)
leven_button.pack(pady=10)

# Create a button for n-gram
ngram_button = tk.Button(root, text="n-gram", command=on_ngram_click, width=button_width, height=button_height
                         bg=button_color, fg=button_foreground, font=custom_font1)
ngram_button.pack(pady=10)

# Start the GUI event loop
root.mainloop()
```

B. Levenshtein Distance Function

The implementation of a Levenshtein distance calculation function made it less complicated to find the dictionary word matches that are the closest.

This function calculates the Levenshtein distance between two strings s1 and s2. The Levenshtein distance is a metric for measuring the difference between two sequences by counting the minimum number of operations (insertions, deletions, substitutions) required to transform one string into another.

```
def levenshtein_distance(s1, s2):
    if len(s1) < len(s2):
        return levenshtein_distance(s2, s1)

    if len(s2) == 0:
        return len(s1)

    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row

    return previous_row[-1]
```

C. N-gram Frequency Distribution

This technique was generated with a pre-trained Bert model language model, to enhance the coding efficiency, and to obtain more accurate results.

```
def sequence_score(seq):
    score = 0
    for i in range(len(seq)):
        if i == 0:
            # Add a significant weight to the frequency of the word in the dictionar
            score += NWORDS[seq[i]] * 10
        elif i == 1:
            score += bigram_freq[seq[i - 1]][seq[i]]
        else:
            score += trigram_freq[seq[i - 2], seq[i - 1]][seq[i]]
    return score
```

This function used a unigram, bigram, and trigram frequencies to calculate the score of a word sequence.

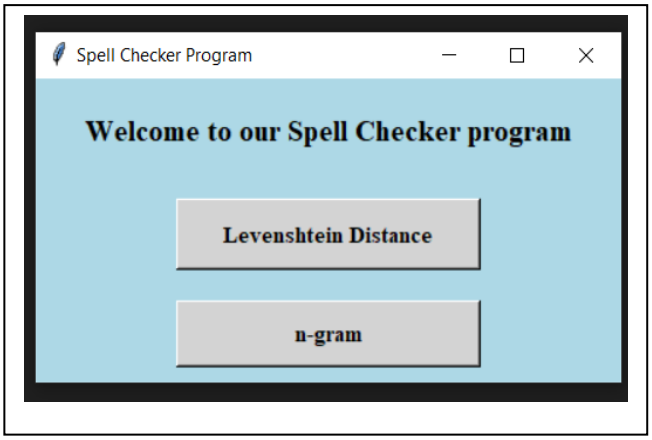
D. Beam Search Algorithm

```
def beam_search(words, beam_width=3):
    sequences = [([], 0)]
    for word in words:
        all_candidates = []
        for seq, score in sequences:
            candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
            for candidate in candidates:
                # Incorporate the frequency of the candidate into the score
                candidate_score = score + sequence_score(seq + [candidate]) + NWORDS[candidate] * 1
                all_candidates.append((seq + [candidate], candidate_score))
        # Sort the candidates by the updated score which includes frequency
        ordered = sorted(all_candidates, key=lambda x: x[1], reverse=True)
        sequences = ordered[:beam_width]
    return sequences[0][0]
```

The “Beam_search” function corrects a word sequence by combining edit distance and frequency-based scoring. This method guarantees that the spell checker can efficiently manage sequence probabilities and context.

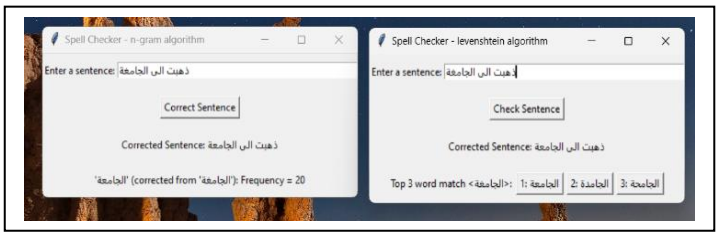
IV. RESULTS

After running the code, a list is shown to choose between the two discussed techniques, whether the Levenstein distance or the n-gram frequency model.



After running some tests,

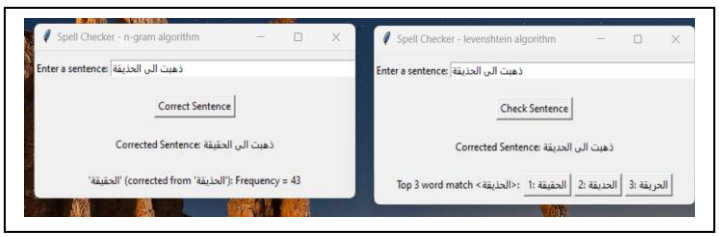
1. First result



The input: "ذهبت إلى الجامغة"

As we observed, both techniques get the corrected message right. The n-gram provides the indicated correct word commonly used in the training corpus with frequency 20, adding an extra layer of reliability. Whereas, Levenshtein provides multiple options for users to select the most appropriate correction based on context, which is more useful in case of an incorrect first result or ambiguous answer. It is particularly effective in correcting minor simple typographical errors such as insertion, deletions, and substitutions. However, the n-gram can ensure that corrections are contextually appropriate, by leveraging the context of the surrounding words.

2. Second result



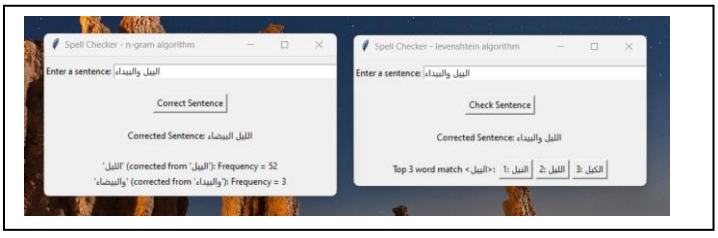
The input: "ذهبت إلى الحذيقة"

For spell checker based on the Levenshtein algorithm it suggested that a top matching word is "الحقيقة", but also the correct word was suggested in the top 3-word match: " 1: الحقيقة، 2:الحديقة،3:الحرقية", which is the word "الحديقة", providing multiple options allows the user to select the most appropriate word according to the context, were it was effective because, the algorithm here focuses on the phonetic similarity and does not consider the broader context, which led to incorrect suggestions that may sound similar but they are not accurate contextually.

On the other hand, the n-gram model also gave us the same results, where the frequency of the word sequence to the given result was 43, which is high, so this is a possibility for the incorrect result whereas maybe the correct sentence may have less frequency than the given result.

Neither of the methods produced the correct contextually appropriate correction. However, due to the Levenshtein multiple options features, allowed the correct result to appear within the top match result, so the user can choose the correct word manually.

3. Third result



The input: "البيل والبيداء"

As we can see, different results appear, for the n-gram corrected both given words, the first word corrected it right, considering it has high frequency with 52. But the other word "البيداء" was correct originally, due to the lack of this word in the dataset, it considered it a wrong word, and corrected it to the closest result with frequency 3.

On the contrary, the Levenstein algorithm, did have the second word in its dictionary, so it has no problem with it, but the first word was corrected to another word from what we meant to achieve, and it may have a deal with the order of the dictionary, but due to multiple option feature, the correct contextual result appeared in the top 3-word match, so it allows the user to chose the correct appropriate contextual word.

The results show that although each approach has advantages, the n-gram model outperformed the other by giving the right correct immediately. Along with less relevant suggestions, the Levenshtein algorithm provided the correct adjustment as the top match.

V. Conclusion

This study proposes an Arabic spell checker that blends modern NLP models with traditional spell-checking methods. Contextual awareness is absent in the Levenshtein distance method, which corrects basic typos successfully and provides several suggestions. Through the analysis of word sequence frequencies, the n-gram model improves overall accuracy and offers contextually relevant adjustments.

These approaches work much better when combined with contemporary NLP strategies like BERT, which guarantees that fixes are contextually and phonetically correct. According to experimental findings, this combined method produces an Arabic spell checker that is trustworthy and strong.

VI. Future Enhancements

In an effort to improve the user experience and refine corrective recommendations, future work will concentrate on increasing computing performance, growing the training dataset, and using advanced NLP approaches. This breakthrough opens the door for more developments in Arabic NLP applications and represents a major step toward enhancing text correctness and readability in Arabic language processing.

References

- *2024 Elsevier B.V., its licensors, and contributors.Levenshtein Distance. ScienceDirect.*
- *Daniel Jurafsky & James H. Martin. (2024). N-gram Language Models. Stanford University.*
- *2023 UBIAI Web Services. NLP Techniques: Tokenization, POS Tagging and NER. Ubiai*
- *2024, Saturn Cloud. Stemming in Natural Language Processing. Saturn Cloud*