CHAPTER-1

1. ABSTRACT:

This implementation introduces a streamlined algorithm for identifying and correcting misspellings by finding the nearest match from a given word database. By employing traditional natural language processing techniques, the algorithm first preprocesses both the input search term and the entries in the database, ensuring consistency through text normalization and the removal of non-alphanumeric characters.

Next, it utilizes N-gram analysis to create sequences of characters for both the search term and each database word, which facilitates the quick identification of candidates with notable similarities. The algorithm then calculates the Levenshtein distance—a metric that measures the least number of single-character edits needed to transform one word into another—to pinpoint the closest match from the shortlisted candidates.

The result is a user-friendly output that presents the best matching word along with its Levenshtein distance, making it an effective solution for spelling correction and suggestion tasks. This method is especially useful in applications like text input systems and search functionalities, enhancing the user experience by effectively handling typographical errors without employing AI or machine learning techniques.

INTRODUCTION AND RELATED WORK:

In this project, the autocorrection and autospelling of a word are performed by providing an input word prompt from the user. The input word is then compared with words from a predefined list. Two encoding algorithms are employed for this purpose.

1. N-gram Indexing Algorithm: This algorithm compares adjacent characters of the incorrect text with the words from the predefined list. The more common N-grams between the input word and a word from the list, the closer that word is to the correct one.

2. Levenshtein Distance Algorithm: This algorithm calculates the number of additions, deletions, or substitutions needed to transform the incorrect word into a word from the predefined list. The lower the Levenshtein distance, the more accurate the word is considered.

The results from both algorithms are compared, and the best match is displayed as the output. The output should be presented using output console, where the user can enter the input word and receive the corrected text.

Example:
Input word: "Whatsap"
N = 2 (bigram comparison)

N- gram algorithm:

1. Incorrected word : “whatsap”
   Generated N-grams: [‘wh’, ‘ha’, ‘at’, ‘ts’, ‘sa’, ‘ap’]
2. Corrected word :
   Generated N-grams : [‘wh’, ‘ha’, ‘at’, ‘ts’, ‘sa’, ‘ap’, ‘pp’]
3. Common N-grams between “whatsap” and “whatsapp”: [wh’, ‘ha’, ‘at’, ‘ts’, ‘sa’,‘ap’] (6 common bigrams).
4. Filter/sort the list with more common bigrams

Levenshtein Distance algorithm:

Compare the string distance between the predefined word from the sorted list of n-gram indexing and incorrect word using distance the measuring in the matrix form

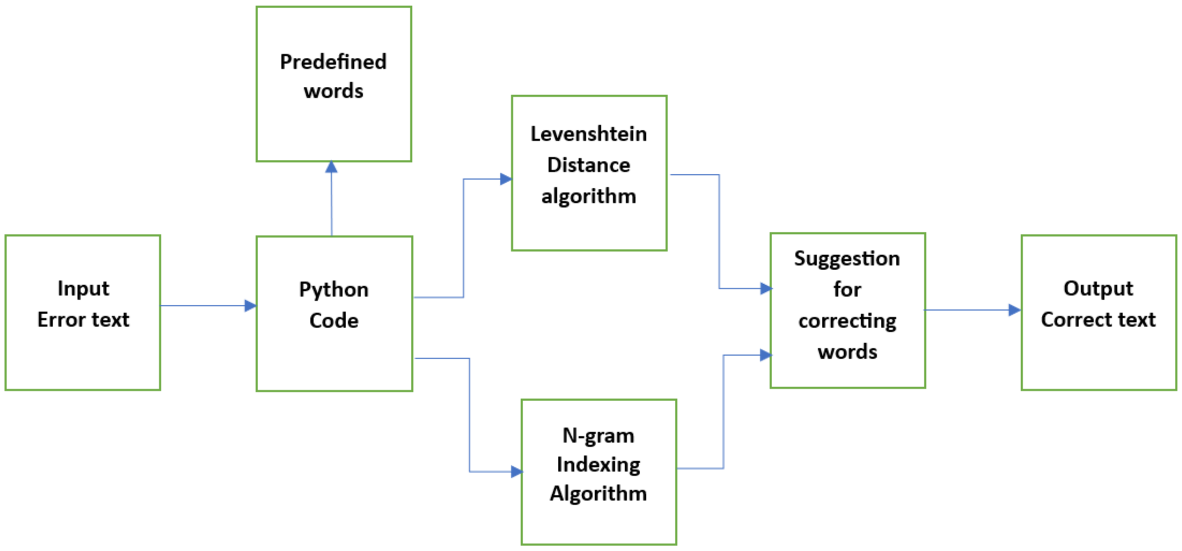
		w	**h**	**a**	**t**	**s**	**a**	**p**
	0	1	2	3	4	5	6	7
w	1	0	1	2	3	4	5	6
h	2	1	0	1	2	3	4	5
a	3	2	1	0	1	2	3	4
t	4	3	2	1	0	1	2	3
s	5	4	3	2	1	0	1	2
a	6	5	4	3	2	1	0	1
p	7	6	5	4	3	2	1	0
p	8	7	6	5	4	3	2	1

Here 1 is the Levenshtein distance length i.e., value one is the bottom right corner of the matrix (addition to the string). The final result is compared with incorrect text with all predefined text in CSV file and the best result is displayed as the suggested word/ matching words.

From this, both 2 algorithm approaches are going to be employed in this project, and it has many advantages, like N-gram indexing is efficient for initial filtering, scalable to large datasets, and language-independent, making it highly effective in reducing the search space for candidate words. Levenshtein distance provides a precise, flexible, and highly accurate measure of word similarity by handling all types of errors, making it one of the most reliable methods for autocorrection. Autocorrection of the text can be done fast. This project is going to be implemented in Python IDE and expected result should be in the form of G.U.I of the corrected/suggested text.

CHAPTER-2

2. BLOCK DIAGRAM AND SYSTEM OVERVIEW:



Predefined words
Levenshtein Distance algorithm
Input Error text
Python Code
Suggestion for correcting words
Output Correct text
N-gram Indexing Algorithm

This block diagram represents the workflow of a text correction system. The process begins with the input of error-prone text. This input is processed by a Python program, which uses two main algorithms to identify potential corrections: the Levenshtein Distance algorithm and the N-gram Indexing algorithm. The Levenshtein Distance algorithm calculates the similarity between the input words and a list of predefined words, identifying closely matching words. Simultaneously, the N-gram Indexing algorithm suggests words based on patterns in the input text. Both algorithms provide a list of possible corrections, and the system suggests the most appropriate corrected words, outputting a correctly formatted text as the final result.

TOOLS/ SOFTWARE REQUIRED:

1. Windows 8 or latest
2. Python IDE (Pycharm community edition)
3. Python version 3.11

CHAPTER-3

3. RESULT:

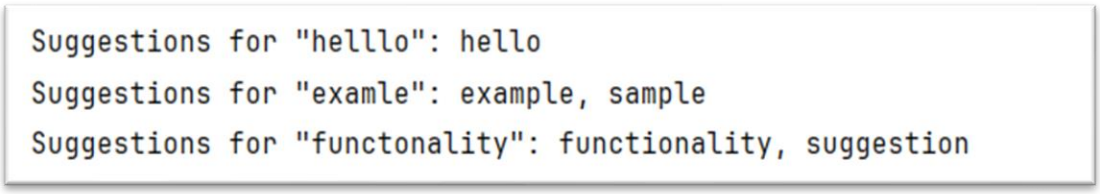
Python code:

```
import re
from collections import Counter


# Function to preprocess a word
def preprocess_word(word):
    # Normalize case and remove non-alphanumeric characters
    word = word.lower()
    word = re.sub(r'\W+', '', word)
    return word


# Function to generate N-grams from a word
def generate_ngrams(word, n):
    return [word[i:i + n] for i in range(len(word) - n + 1)]


# Function to create an ASCII list from a word
def createList(word):
    return [ord(char) for char in word]


# Levenshtein Distance Calculation (Edit Distance)
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]


# Function to process the search term with N-gram and Levenshtein Distance
```

```
def process_search_with_ngrams(searchTerm, database, n=2, threshold_min=4,
max_suggestions=1):
    # Preprocess the search term
    searchTerm = preprocess_word(searchTerm)
    search_ngrams = generate_ngrams(searchTerm, n)  # Generate N-grams for search term

    # Preprocess the database words and create ASCII lists
    processed_database = [preprocess_word(word) for word in database]
    lstAscii = [createList(word) for word in processed_database]

    candidate_indices = []

    # Step 2: Iterate through each string in the database
    for i in range(len(processed_database)):
        word = processed_database[i]
        word_ngrams = generate_ngrams(word, n)  # Generate N-grams for each word in the
database

        # Compare the overlap in N-grams between the search term and the database word
        common_ngrams = set(search_ngrams) & set(word_ngrams)

        # If there are significant common N-grams, proceed with length difference check
        if len(common_ngrams) > 0:
            diff = abs(len(lstAscii[i]) - len(createList(searchTerm)))
            if diff < threshold_min:
                candidate_indices.append(i)

    # Calculate Levenshtein Distance and store results
    distances = []
    for i in candidate_indices:
        current_word = processed_database[i]
        cost = levenshtein_distance(current_word, searchTerm)
        distances.append((current_word, cost))

    # Sort by distance and select top max_suggestions
    distances.sort(key=lambda x: x[1])
    suggestions = [word for word, dist in distances[:max_suggestions]]

    return suggestions


# Expanded dictionary of valid words (database)
database = ["sample","hello", "world", "whatsapp", "feature", "suggestion", "test", "typing",
            "application", "weather", "system", "functionality", "development", "python",
            "programming", "code", "example", "reference", "suggestions"]

# List of input words with their corresponding number of suggestions
input_words = [("helllo", 1), ("examle", 3), ("functonality", 0)]

# Process each input word and get suggestions
```

```
for word, num_suggestions in input_words:
    suggestions = process_search_with_ngrams(word, database,
max_suggestions=num_suggestions)
    print(f'Suggestions for "{word}": {", ".join(suggestions)}')
```

OUTPUT CONSOLE:



```
Suggestions for "helllo": hello
Suggestions for "examle": example, sample
Suggestions for "functonality": functionality, suggestion
```

CHAPTER-4

4. CONCLUSION:

This project serves as a robust text correction system that combines multiple techniques to suggest accurate word matches from a predefined database. It preprocesses both the input word and the database words by normalizing their case and removing non-alphanumeric characters. Using N-grams, it identifies candidates with overlapping character sequences and filters them based on length differences. To further refine the results, the code calculates the Levenshtein distance between the input word and the candidates, prioritizing words with the smallest edit distances. Finally, it returns a ranked list of suggestions, demonstrating the integration of N-grams and edit distance for effective spell-checking and text correction.

FUTURE SCOPE:

Future scope of this project can be implemented by this -
Enhanced Contextual Accuracy: Incorporating a natural language processing (NLP) model, such as a neural network or transformer-based architecture, could help analyze the context of the input word and provide more accurate suggestions.
Multilingual Support: Expanding the system to support multiple languages by adapting N-grams and Levenshtein distance calculations for language-specific rules, such as accents, diacritics, and word structure.
Incorporating Phonetic Algorithms: Integrating phonetic similarity algorithms, like Soundex or Metaphone, could improve the system's ability to suggest words based on their pronunciation, especially for misspellings that sound similar to the intended word.

CHAPTER-5

5. LITERATURE REVIEW:

The current literature on speech autocorrect and auto spell features in a text are –

[1] The paper proposes an algorithm that reduces the number of words for which the Levenshtein distance needs to be calculated by filtering irrelevant words using N-gram indexing. The proposed approach significantly reduces execution time, making it more suitable for large datasets. Case studies show that the algorithm runs almost three times faster than using Levenshtein distance alone.

[2] The document "Survey of Automatic Spelling Correction" provides a comprehensive review of automatic spelling correction (ASC) systems, their methods, and the evolution of approaches from 1991 to 2019.

Types of ASC Systems: Three main groups of ASC systems are described:

- A priori systems: Use predefined rules to detect and correct errors.
- Contextual systems: Analyze surrounding text to improve correction accuracy.
- Learning-based systems: Adapt error correction models from training data.

[3] The paper titled "Spell Correction and Suggestion using Levenshtein Distance" discusses the implementation of a spell correction system using the Levenshtein distance algorithm. The study focuses on correcting and suggesting movie names based on user input. The authors chose this approach due to the increasing importance of Natural Language Processing (NLP) in AI and data science, particularly in applications such as search engines.

[4] The paper "Spelling Correction Using Recurrent Neural Networks and Character Level N-gram" discusses a method for correcting misspelled words using a combination of recurrent neural networks (RNNs) and character-level n-grams. Key points include:

- A recurrent neural network (RNN) is trained with dictionary words to act as an oracle for misspelled words.
- Character-level bigrams (2-grams) are used to generate new query words from the misspelled input, improving the chances of accurate correction.
- The system is designed to suggest corrections rather than automatically correcting errors.
- The method is tested using Turkish words, and three operations (adding, deleting, or changing a character) are used to create test data.
- The results show improved accuracy when combining misspelled words with the generated query words, particularly for longer words.

[5] The paper titled "Predictive and Corrective Text Input for Desktop Editor using N-grams and Suffix Trees" explores a smart text editor designed for desktop computers. The system uses n-grams and suffix trees to implement predictive and corrective text input, improving typing efficiency, especially for groups like senior citizens or people with disabilities.

[6] The paper discusses a study on techniques for correcting intentional grammatical errors on Twitter/X during the COVID-19 pandemic. It evaluates methods like Levenshtein Distance, N-Gram, and Soundex phonetic algorithms. The main findings indicate that the Levenshtein Distance method was the most accurate (100% in several categories), while the N-Gram

technique also showed promise. Soundex was less consistent but improved when the first letter was excluded from encoding. The study highlights how correcting these intentional misspellings can improve natural language processing (NLP) tool performance.

REFERENCES:

[1] Lalwani, Mahesh, Nitesh Bagmar, and Saurin Parikh. "Efficient Algorithm for Auto Correction Using ngram Indexing." *International Journal of Computer & Communication Technology (IJCCT)* 3.3 (2014): 23-27.
[2] Hládek, Daniel, Ján Staš, and Matúš Pleva. "Survey of automatic spelling correction." *Electronics* 9.10 (2020): 1670.
[3] Mehta, Ansh, et al. "Spell correction and suggestion using Levenshtein distance." *Int Res J Eng Technol* 8.8 (2021): 1977-1981.
[4] Kinaci, A. Cumhur. "Spelling correction using recurrent neural networks and character level n-gram." *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*. IEEE, 2018.
[5] Bhatia, Akshay, et al. "Predictive and corrective text input for desktop editor using n-grams and suffix trees." *2016 International Conference on Advances in Human Machine Interaction (HMI)*. IEEE, 2016.
[6] Marini, Thainá, and Taffarel Brant-Ribeiro. "Comparative Analysis of Intentional Gramatical Error Correction Techniques on Twitter/X." *Proceedings of the 16th International Conference on Computational Processing of Portuguese*. 2024.