

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institute, Affiliated to VTU)

Bull Temple Road, Basavanagudi, Bengaluru - 560019



Technical Seminar Report on

“AUTOCOMPLETION AND SPELL CHECKER USING DYNAMIC PROGRAMMING”

Submitted in partial fulfillment of the requirements for the award of degree

**BACHELOR OF ENGINEERING IN
INFORMATION SCIENCE AND ENGINEERING**

By

ARYAN (1BM18IS018)
ADITYA A KAMAT (1BM18IS003)

Under the guidance of

MANJULA S
Assistant Professor

**Department of Information Science and Engineering
2021-2022**

B.M.S. COLLEGE OF ENGINEERING
B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institute, Affiliated to VTU)

**Bull Temple Road, Basavanagudi,
Bangalore – 560019**



C E R T I F I C A T E

Certified that the Technical Seminar has been successfully presented at **B.M.S. College of Engineering** by **Aryan** and **Aditya A Kamat** bearing USN: **1BM18IS018** and **1BM18IS003** respectively in partial fulfillment of the requirements for the VII Semester degree in **Bachelor of Engineering in Information Science & Engineering** of **Visvesvaraya Technological University, Belgaum** as a part of for the **course** Technical Seminar **Course Code – 20IS7SRTL5** during the academic year 2021-22.

Faculty Name: Manjula S

Designation: Assistant Professor

Signature:

Index

SN.	CONTENTS	PAGE NO.
1	Abstract	4
2	Problem Statement	5
3	Introduction	6
4	Relevance	7
5	Detailed Literature Survey	8
6	Literature Survey Summary Table	23
7	Methodology & Implementation	26
8	Results	37
9	Conclusion	38
10	References	40

Abstract

- Autocomplete is a feature in which an application predicts the rest of a word a user is typing. Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user.
- A spell checker (or spelling checker or spell check) is a software feature that checks for misspellings in a text. Spell-checking features are often embedded in software or services, such as a word processor, email client, electronic dictionary, or search engine.

Problem Statement

Aim: Implement Autocomplete and Spell checking using Levenshtein Distance Algorithm.

Motivation: Today's online users have high expectations for search. Thanks to Google and other sites, they've come to expect certain search patterns and functions, like autocomplete, in all of their search experiences. Autocomplete, or predictive search, is particularly important because it leads users to better search results, thus improving the user experience.

Objective: We use the Levenshtein Distance Algorithm to calculate the similarity between the input word and the target word in both spell-check and auto-complete programs.

Introduction

The Levenshtein distance algorithm is a dynamic programming algorithm that is a text similarity measure that compares two words and returns a numeric value representing the distance between them. The more similar the two words are, the less the distance between them, and vice versa.

The Web has a tremendous collection of useful information however, extracting accurate information from the web is extremely difficult, because the current search engines are restricted to keyword-based search techniques. Search engines generally find the data based on keywords they entered, therefore a lot of cases when the user cannot find the data that they need because there is an error while entering a keyword.

The web has a tremendous collection of useful information however, extracting accurate information from the web is extremely difficult, because the current search engines are restricted to keyword-based search techniques. Thus, the interpretation of information contained in web documents is left to the human user to be done manually.

Autocomplete is a feature provided by many web browsers such as a search engines interface, word processor, and command-line interpreter. It's able to advise users without writing the whole word completely.

string matching (ASM) is a well-known computational problem with important applications in database searching, plagiarism detection, spelling correction, and bioinformatics. Levenshtein distance algorithm is one of the Approximate string matching algorithms used in search strings based on the estimation approach.

This algorithm is a weighting approach to appoint a cost of 1 to every edit operation (Insertion, deletion, and substitution). This distance is known as Levenshtein distance, a special case of edit distance where unit costs apply.

Automatic spell checker systems aim to verify and correct erroneous words through a suggested set of words that are the nearest lexically to the erroneous ones. The spell-check can be divided into two main subproblems: error verification and the correction of errors found. The first one is simply to determine whether the word belongs to the target language.

Relevance

- Autocompletion is one of the most important concepts applied in smartphone keyboards and is also used in software such as Microsoft Word and Google Slides.
- Spell Checker is the core application of software such as Grammarly, which performs spell checking while writing emails or drafting reports

Literature Survey

[2018] [Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library] [Muhammad Maulana Yulianto, Riza Arifudin, Alamsyah]

Nowadays internet technology provides more convenience for searching information on a daily. Users are allowed to find and publish their resources on the internet using a search engine. A search engine is a computer program designed to facilitate a user to find the information or data that they need. Search engines generally find the data based on keywords they entered, therefore a lot of cases when the user can't find the data that they need because there is an error while entering a keyword. That's why a search engine with the ability to detect the entered words is required so the error can be avoided while we search the data. The feature that is used to provide the text suggestion is autocomplete and spell checking using the Levenshtein distance algorithm. The purpose of this research is to apply the autocomplete feature and spell checking with the Levenshtein distance algorithm to get text suggestions in an error data searching in the library and determine the level of accuracy on data search trials. This research uses 1155 data obtained from UNNES Library. The variables are the input process and the classification of books. The accuracy of the Levenshtein algorithm is 86% based on 1055 source cases and 100 target case.

The process of implementing the autocomplete and spell-checking of data contained in the search process. In the search process data using the Levenshtein distance algorithm that has three string matching operations includes deletion, insertion, and substitution. Research results obtained suggest is the appearance of text on the search system includes autocomplete and autocorrect. Spell checking accuracy rate obtained from the system by 86%, calculated using 1055 data as a source case (training data) and 100 data as a target of a case (test case).

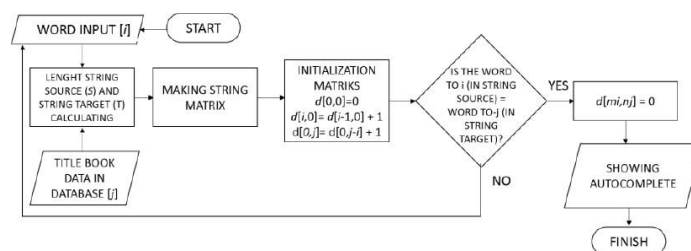


Figure 3. Flowchart autocomplete

		k	i	t	t	e	n				S	a	t	u	r	d	a	y			
		0	1	2	3	4	5	6			0	1	2	3	4	5	6	7	8		
s	1	1	2	3	4	5	6		S	1	0	1	2	3	4	5	6	7			
i	2	2	1	2	3	4	5		u	2	1	1	2	2	3	4	5	6			
t	3	3	2	1	2	3	4		n	3	2	2	2	3	3	4	5	6			
t	4	4	3	2	1	2	3		d	4	3	3	3	3	4	3	4	5			
i	5	5	4	3	2	2	3		a	5	4	3	4	4	4	4	3	4			
n	6	6	5	4	3	3	2		y	6	5	4	4	5	5	5	4	3			
g	7	7	6	5	4	4	3														

Figure 1. Levenshtein distance calculation table

[2017] [Research on String Similarity Algorithm based on Levenshtein Distance] [Shengnan Zhang, Yan Hu, Guangrong Bian]

The application of string similarity is very extensive, and the algorithm based on Levenshtein Distance is particularly classic, but it is still insufficient in the aspect of universal applicability and accuracy of results. Combined with the Longest Common Subsequence (LCS) and Longest Common Substring (LCCS), similarity algorithm based on Levenshtein Distance is improved, and the string similarity result of the improved algorithm is more distinct, reasonable, and accurate, and also has better universal applicability. What's more in the process of similarity calculation, the Solving algorithm of the LD and LCS has been optimized in the data structure, reduce the space complexity of the algorithm from the order of magnitude. And the experimental results are analyzed in detail, which proves the feasibility and correctness of the results.

The calculation of string similarity is further improved according to the idea of raising, analyzing and solving problems. Taking into account the important influence of LCS and LCCS on the computation of string similarity to improve the algorithm, and extensive experiments show that an optimized algorithm improves the general applicability of the algorithm and the accuracy of the results, moreover, a better discriminative similarity result can also be obtained for that string with very small differences. In addition, the traditional calculation process of LD and LCS is optimized in terms of data structure, which further reduces the space complexity of the algorithm in the case that the time complexity is not affected.

[2015] [Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection]
[Mrs.Shital Gaikwad, Prof.Nagaraju Bogiri]

Electronic Data Processing used automated methods for processing commercial data. There is a big amount of work on discovering duplicates in relational data; merely elite findings concentrate on duplication in additional multifaceted hierarchical structures Electronic information is one of the key factors in several business operations, applications, and determinations, at the same time as an outcome, guarantee its superiority is necessary. Duplicate finding a little assignment because of the actuality that duplicates are not accurately equivalent, frequently because of the errors in the information. Accordingly, many data processing techniques never apply widespread assessment algorithms which identify precise duplicates. As an alternative, evaluate all objective representations, using a probably compound identical approach, to identify whether the object is real world or not. Duplicate detection is applicable in data clean-up and data incorporation applications and which considered comprehensively for relational data or XML documents. It provides the person who reads with the groundwork for research in Efficient and Effective Duplicate Detection in Hierarchical Data or XML data.

Levenshtein distance algorithm uses a Bayesian Network to determine the probability of two XML objects being duplicates. The Bayesian Network model is composed of the structure of the objects being compared, thus probabilities of all objects are computed considering not only the information the objects contain, but also how such information is structured. Levenshtein distance algorithm is very flexible, XMLDup requires little user intervention user only needs to provide the dataset and a similarity threshold. Using the Levenshtein Distance algorithm gives better results than the Normalized Edit Distance algorithm. To improve the runtime efficiency of the XMLDup process, a network pruning strategy is also presented. The experiments performed on the mentioned data achieve high precision and recall scores.

[2020] [Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application] [Sugiarto, I Gede Susrama Mas Diyasa, Ilvi Nur Diana]

Archives are recorded records of various activities or organizations in various forms, such as letters, books, and others, stored systematically in the place provided to facilitate searches if needed again. Archived data is recorded based on date, letter number, and other things related to digital filing. Digital filing of letters by storing softcopy documents, accompanied by incoming and outgoing mail reports, and monitoring mail disposition. To make it easier for agencies to process correspondence and letter archiving and speed up searching for letters, the Levenshtein distance algorithm is used. This research results in the form of a letter filing and disposition application that is easy in managing letters such as letter filing, letter disposing, and others by applying the Levenshtein distance algorithm in letter search to speed up the letter search process.

In searching for letters, the Levenshtein distance algorithm is implemented by using several processes, namely inputting words in the letter search. The word is obtained from the subject of the letter that has been stored in the API, and then the word input has calculated the distance of the word that is close to the desired word to display the word being searched for in the letter search. Comparison of letter searches without using algorithms and letter searches using the Levenshtein distance algorithm in Archiving design and letter disposition can display data by the word being searched even though there are some errors when writing the word you want to search for, such as writing the word "kbersihan" the result that appears from the process is "kebersihan".

[2007] [Using the Levenshtein Edit Distance for Automatic Lemmatization: A Case Study for Modern Greek and English] [Dimitrios P. Lyras, Kyriakos N. Sgarbas, Nikolaos D. Fakotakis]

In the presented work they have implemented the Edit Distance (also known as Levenshtein Distance) on a dictionary-based algorithm to achieve the automatic induction of the normalized form (lemma) of regular and mildly irregular words with no direct supervision. The algorithm combines two alignment models based on the string similarity and the most frequent inflexional suffixes. In our experiments, we have also examined the language-independency (i.e. independence of the specific grammar and inflexional rules of the language) of the presented algorithm by evaluating its performance on the Modern Greek and English languages. The results were very promising as we achieved more than 95% of accuracy for the Greek language and more than 96% for the English language. This algorithm may be useful to various text mining and linguistic applications such as spell-checkers, electronic dictionaries, morphological analyzers, search engines, etc.

In this paper, they demonstrated a new language-independent lemmatization algorithm based on the Levenshtein distance and we evaluated its performance both in Modern Greek and English languages. By the term “language-independent”, we mean that the algorithm can perform sufficiently well for a variety of languages regardless of the specific grammar and inflectional rules that apply to them. The proposed algorithm may be useful to various text mining and natural language processing applications such as spell-checkers, morphological processors, electronic dictionaries, web search engines, stemmers, identifiers of lexical features for automatic document classification, etc. The only resource required for the algorithm to function is a database containing all the words of the examined language in a headword form (lemmas). Optionally, one could also use a database containing all the suffixes of the language in question, thus improving the performance of the proposed system.

[2020] [Levenshtein Distance, Sequence Comparison and Biological Database Search] [Bonnie Berger, Michael S. Waterman, and Yun William Yu]

Levenshtein edit distance has played a central role both past and present—in sequence alignment in particular and biological database similarity search in general. They started a review with a history of dynamic programming algorithms for computing Levenshtein distance and sequence alignments. Following, we describe how those algorithms led to heuristics employed in the most widely used software in bioinformatics, BLAST, a program to search DNA and protein databases for evolutionarily relevant similarities. More recently, the advent of modern genomic sequencing and the volume of data it generates has resulted in a return to the problem of local alignment. It concludes with how the mathematical formulation of Levenshtein distance as a metric made possible additional optimizations to similarity search in biological contexts. These modern optimizations are built around the low metric entropy and fractional dimensionality of biological databases, enabling orders of magnitude acceleration of biological similarity search.

As bioinformatics data continues to grow in volume and ease of acquisition, it is essential to develop more sophisticated algorithms for data processing. Levenshtein published his landmark paper over half a century ago, forming the foundation of sequence comparison search. Although some modern bioinformatics heuristics using k-mer matching have partially supplanted the direct optimization of Levenshtein distance through dynamic programming, his formulation of metric string distance remains relevant to this day and a source of inspiration for active research.

In many applications, it is necessary to determine the similarity of two strings. A widely-used notion of string similarity is the edit distance: The minimum number of insertions, deletions, and substitutions required to transform one string into the other. In this report, they provided a stochastic model for string-edit distance. The stochastic model allows to learn a string-edit distance function from a corpus of examples, illustrate the utility of the approach by applying it to the difficult problem of learning the pronunciation of words in conversational speech. In this application, they learn a string-edit distance with nearly one-fifth the error rate of the untrained Levenshtein distance. The approach applies to any string classification problem that may be solved using a similarity function against a database of labeled prototypes.

Firstly, an underlying pronouncing lexicon is constructed directly from the observed pronunciations, without any human intervention, while their underlying lexicon is obtained from a hand-built text-to-speech system. Secondly, probability model $p(y \vee |w)$ assigns a nonzero probability to infinitely many surface forms, while their “network” probability model assigns a nonzero probability to only finitely many surface forms. Thirdly, the use of the underlying form $x \ t$ as a hidden variable means that this model can represent arbitrary (nonlocal) dependencies in the surface forms, which their probability model cannot.

In some real-world applications, the sample could be described as a string of symbols rather than a vector of real numbers. It is necessary to determine the similarity or dissimilarity of two strings in many training algorithms. The widely used notion of similarity of two strings with different lengths is the weighted Levenshtein distance (WLD), which implies the minimum total weights of single symbol insertions, deletions, and substitutions required to transform one string into another. To incorporate prior knowledge of strings into kernels used in support vector machines and other kernel machines, we utilize variants of this distance to replace distance measure in the RBF and exponential kernels and inner product in polynomial and sigmoid kernels and form a new class of string kernels: Levenshtein kernels in this paper. Combining our new kernels with support vector machine, the error rate and variance on the UCI splice site recognition dataset over 20 runs is 5.88 ± 0.53 , which is better than the best result 9.5 ± 0.7 from the other five training algorithms.

In many applications, the sample could be described as a string of symbols rather than a vector with continuous value components. It is not always the proper way to convert the symbolic data into integers or real numbers and to measure the similarity or dissimilarity using distances notion in the real space. In this paper, they directly consider the symbolic data of strings and use the weighted Levenshtein distance (WLD) as the similarity or dissimilarity of two strings. The variants of this distance are utilized to replace the distance measure and inner product in the four common kernels. Thus a new class of kernels based on WLD is constructed, which can be referred to as Levenshtein kernels. In the splice site recognition, state-of-the-art performance has been achieved by combining SVM with our kernels. It is demonstrated that incorporating prior knowledge about problems at hand into kernels can improve the performance of kernel machines effectively.

This paper has only considered the strings with different symbols. If a single symbol is replaced by a single word or a proper subsequence, our work still holds. In addition, the new kernels can still be applied in other kernel machines for classification and clustering.

The edit distance (or Levenshtein distance) between two words is the smallest number of substitutions, insertions, and deletions of symbols that can be used to transform one of the words into the other. This paper has considered the problem of computing the edit distance of a regular language (also known as constraint system), that is, the set of words accepted by a given finite automaton. This quantity is the smallest edit distance between any pair of distinct words of the language. It shows that the problem is of polynomial time complexity. They distinguish two cases depending on whether the given automaton is deterministic or nondeterministic. In the latter case, the time complexity is higher. They have shown why the problem of computing the edit distance of a given regular language is of polynomial time complexity. They have restricted our attention to the case of the unweighted edit distance because this case is closely related to the concept of error detection as discussed in the introduction. However, the methods can be applied even in the case where the edit distance is weighted such that the weights on the errors are positive numbers with a slight increase in the time complexity of the algorithm.

- The first question that arises is whether the time complexity of the algorithms can be improved asymptotically. It appears that this is possible for certain special cases at least. For example, when the given automaton is a trellis, that is, an automaton with a single final state accepting only words of the same length, then and in typical applications is much smaller.
- The next question that arises is whether the results extend to the case of other difference measures for words, in particular, those defined by weighted automata. In this case, the edit strings that one can use to transform words are restricted to only those permitted by the weighted automaton.

**[2008] [Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm]
[Zhan Su, Byung-Ryul Ahn, Ki-yol Eom, Min-Koo Kang, Jin-Pyung Kim, Moon-Kyun Kim]**

Plagiarism in texts is an issue of increasing concern to the academic community. Now, most common text plagiarism occurs by making a variety of minor alterations that include the insertion, deletion, or substitution of words. Such simple changes, however, require excessive string comparisons. This paper presents a hybrid plagiarism detection method. They investigate the use of a diagonal line, which is derived from Levenshtein distance, and simplified the Smith-Waterman algorithm that is a classical tool in the identification and quantification of local similarities in sequences, with a view to the application in plagiarism detection. The approach avoids globally involved string comparisons and considers psychological factors, which can yield significant speed-up by experiment results. Based on the results, we indicate the practicality of such improvement using Levenshtein distance and Smith-Waterman algorithm and to illustrate the efficiency gains. In the future, it would be interesting to explore appropriate heuristics in the area of text comparison.

They have described how the Levenshtein distance can be used to change the likely scarcity, which can improve both time and space efficiency, and how the simplified Smith-Waterman algorithm can be realized for significant local similarities. From the experimental results, it can be proved that it is a powerful tool for the detection of plagiarism in practice. Further efficiency gains in adapting the algorithm to this context would be valuable. The implementation of the improvement here might not be fast enough for an open web-based detection service, which has lots of data. In the future, it would be interesting to explore appropriate heuristics in the area of text comparison, which will decrease the disadvantages that the Smith-Waterman is too slow for large-scale application. It is also the importance of artificial intelligence research.

Although several normalized edit distances presented so far may offer good performance in some applications, none of them can be regarded as a genuine metric between strings because they do not satisfy the triangle inequality. Given two strings X and Y over a finite alphabet, this paper defines a new normalized edit distance between X and Y as a simple function of their lengths ($|X|$ and $|Y|$) and the Generalized Levenshtein Distance (GLD) between them. The new distance can be easily computed through GLD with a complexity of $O(|X| \cdot |Y|)$ and it is a metric value in $[0, 1]$ under the condition that the weight function is a metric over the set of elementary edit operations with all costs of insertions/deletions having the same weight. Experiments using the AESA algorithm in handwritten digit recognition show that the new distance can generally provide similar results to some other normalized edit distances and may perform slightly better if the triangle inequality is violated in a particular data set.

In this paper, a new normalized edit distance has been presented as a simple function of the string lengths and the Generalized Levenshtein Distance. The main contribution of the paper is to prove that the new distance is a metric value in $[0, 1]$ under common conditions and demonstrate, by using AESA in handwritten digit recognition, that it can generally achieve similar accuracies to two other normalized edit distances, yielding slightly better results if the triangle inequality is violated to a certain degree. Since no other normalized edit distance has been shown to be a metric, this work is significant in that regard. As future work, we plan to identify situations where the new distance is appropriate and study its performance in applications such as phylogenetic tree construction, where all three basic properties of a distance metric between two sequences are usually required at the same time. Moreover, we will also consider the problems of how to use the presented techniques to normalize a distance between histograms or a local alignment score between strings in a provably more rigorous way.

In the string correction problem, to transform one string into another using a set of prescribed edit operations. In string correction using the Damerau-Levenshtein (DL) distance, the permissible edit operations are substitution, insertion, deletion, and transposition. Several algorithms for string correction using the DL distance have been proposed. The fastest and most space-efficient of these algorithms is due to Lowrance and Wagner. It computes the DL distance between strings of length m and n , respectively, in $O(mn)$ time and $O(mn)$ space. In this paper, we focus on the development of algorithms whose asymptotic space complexity is less and whose actual runtime and energy consumption are less than those of the algorithm of Lowrance and Wagner.

To develop space- and cache-efficient algorithms to compute the Damerau-Levenshtein (DL) distance between two strings as well as to find a sequence of edit operations of length equal to the DL distance. The algorithms require $O(s \min\{m, n\} + m + n)$ space, where s is the size of the alphabet and m and n are, respectively, the lengths of the two strings. Previously known algorithms require $O(mn)$ space. The space- and cache-efficient algorithms of this paper are demonstrated, experimentally, to be superior to earlier algorithms for the DL distance problem on time, space, and energy metrics using three different computational platforms.

Benchmarking shows that, our algorithms are able to handle much larger sequences than earlier algorithms due to the reduction in space requirements. On a single-core, they were able to compute the DL distance and an optimal edit sequence faster than known algorithms by as much as 73.1% and 63.5%, respectively. Further, to reduce energy consumption by as much as 68.5%. Multicore versions of the algorithms achieve a speedup of 23.2 on 24 cores.

[2015] [ADAPTATING THE LEVENSHTAIN DISTANCE TO CONTEXTUAL SPELLING CORRECTION] [AOURAGH SI LHOUSSAIN, GUEDDAH HICHAM, YOUSFI ABDELLAH]

In the last few years, computing environments for human learning have rapidly evolved due to the development of information and communication technologies. However, the use of information technology in the automatic correction of spelling errors has become increasingly essential. In this context, we have developed a system for correcting spelling errors in the Arabic language based on language models and the Levenshtein algorithm. The metric distance returned by the Levenshtein algorithm is often the same for multiple solutions in correcting a wrong word. To overcome this limitation we have added a weighting based on language models. This combination has helped us to screen and refine the results obtained in advance by the Levenshtein algorithm and applied to the errors of Arabic words. The results are encouraging and demonstrate the value of this approach.

In this article, they tried to introduce bi-grams language models in the Levenshtein metric correction method. From the results obtained, it is clear that this combination helps improve satisfactorily scheduling solutions returned by their method.

The major drawback of their approach lies in the fact that the test corpus size is limited. In the next contribution, they tried on a very large corpus and to get better estimates for bi-gram language models to enhance and increase the effectiveness of our correction approach.

[2003] [A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach] [Victoria J. Hodge and Jim Austin]

This proposes a simple, flexible, and efficient hybrid spell checking methodology based upon phonetic matching supervised learning, and associative matching in the AURA neural system. They integrated Hamming Distance and n-gram algorithms that have a high recall for typing errors and a phonetic spell-checking algorithm in a single novel architecture. The approach was suitable for any spell-checking application though aimed toward isolated word error correction, particularly spell checking user queries in a search engine. They used a novel scoring scheme to integrate the retrieved words from each spelling approach and calculate an overall score for each matched word. From the overall scores, they can rank the possible matches. This paper evaluated the approach against several benchmark spellchecking algorithms for recall accuracy. The proposed hybrid methodology has the highest recall rate of the techniques evaluated. The method has a high recall rate and low-computational cost.

Spell checkers are somewhat dependent on the words in the lexicon. Some words have very few words spelled similarly, so even multiple mistakes will retrieve the correct word.

Other words will have many similarly spelled words, so one mistake may make correction difficult or impossible. Of the techniques evaluated in Table 3, our hybrid approach had the joint highest recall rate at 93.9 percent tied with a spell that uses a much larger rule base. They maintained high recall accuracy for large lexicons and increase recall to Humans averaged 74 percent for isolated word-spelling correction (where no word context is included and the subject is just presented with a list of errors and must suggest a best guess correction for each word). Kukich posits that ideal isolated word error correctors should exceed 90 percent when multiple matches may be returned.

[2019] [Spelling Checker Algorithm Methods for Many Languages] [Novan Zukarnain, Bahtiar Saleh Abbas. Suparta Wayan, Agung Trisetyarso, Chul Ho Kang]

Spell checking plays an important role in improving document quality by identifying misspelled words in the document. The spelling check method aims to verify and correct misspelled words through a series of suggested words that are closer to the wrong word. Currently, Spell checkers for the English language are well established. This article analyses several studies conducted in various types of languages other than English. Like Africa, Arabia, China, Indonesia, India, Japan, Malaysia, and Thailand. We use the systematic literature Review approach to a paper published 2008 - 2018. there are 23 papers to represent each language and view methods used. The result shows that each language has a different Spelling Check Method. The Damerau-Levenshtein algorithm method is most often used for spelling checkers.

This study has two main advantages for theory and practice. As a theory, the results can be a reference for research into the technique of the spelling test method. For practice, these results can be used to identify which methods can be used in a particular language. It is known that every n language has a different method of spell checking. The Damerau-Levenshtein algorithm method is most frequently used for spell checkers, Authors, and Affiliations

To that end, we believe that many other methods still need to be checked to be used in spelling checking. In addition, there are still many languages in the world so that it becomes an interesting opportunity for IS scholars to further investigate the appropriate methods in each language. In conclusion, the findings of this review study provide preliminary insights into the current trends of the spell-checking method

Based on the spelling checker method identified, there is still much to be added for future research. Journal search engines, though added, such as IEEE, ACM, Springer, and others. But for the time being the number of journals is sufficient, and fully represents the facts. Therefore, empirical testing is needed which extensively uses formal statistics to validate these methods.

LITERATURE SURVEY SUMMARY TABLE

Sl.No	Title	Problem Addressed	Author's Approach	Results
1	Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library	To apply the autocomplete feature and spell checking with Levenshtein distance algorithm to get text suggestion in an error data searching in library and determine the level of accuracy on data search trials.	The feature that was used to provide autocomplete and spell checking was the Levenshtein distance algorithm.	Spell checking accuracy rate obtained from the system as 86%.
2	Research on String Similarity Algorithm based on Levenshtein Distance	Similarity algorithm based on Levenshtein Distance is improved.	Longest Common Subsequence (LCS) and Longest Continuous Common Substring (LCCS) are used	The calculation of string similarity is further improved taking into account the important influence of LCS and LCCS on the computation of string similarity.
3	Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection	The aim of this project is to present a novel algorithm to find duplicate objects in the hierarchical structures, like XML files.	Proposed system uses Levenshtein distance algorithm to find duplicates in structured data.	Using Levenshtein Distance algorithm gives better result than Normalized Edit Distance algorithm.
4	Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application	This project results in the form of a letter filing and disposition application by applying the Levenshtein distance algorithm in letter search to speed up the letter search process.	To make it easier to letter archiving and speed up searching for letters, the Levenshtein distance algorithm is used.	Archiving design and letter disposition can display data by the word being searched even though there are some errors when writing the word you want to search for.
5	Using the Levenshtein Edit Distance for Automatic Lemmatization: A	Performing an accurate lemmatization can be a quite difficult and time-consuming task especially for	The Levenshtein Distance is implemented on a dictionary-based algorithm in order to achieve the automatic	In this paper, a new language independent lemmatization algorithm based on the Levenshtein

	Case Study for Modern Greek and English	morphologically complex languages with highly inflexional structure, such as the Modern Greek language.	induction of the normalized form (lemma) of regular and mildly irregular words with no direct supervision.	distance was demonstrated and its performance was evaluated both on Modern Greek and English languages.
6	Levenshtein Distance, Sequence Comparison and Biological Database Search	The advent of modern genomic sequencing and the volume of data it generates has resulted in a return to the problem of local alignment.	Given two biological sequences of length n , the basic problem of biological sequence comparison can be recast as that of determining the Levenshtein distance between them.	Although some modern bioinformatics heuristics using k-mer matching have partially supplanted the direct optimization of Levenshtein distance, the algorithm is still relevant.
7	Learning String-Edit Distance	Determination of similarity of two strings	The authors implement a string-edit distance that reduces the error rate of the untrained Levenshtein distance by a factor of 4.7, to within 4 percent of the minimum error rate achievable by any classifier.	The authors demonstrate the efficacy of their techniques by correctly recognizing over 87 percent of the unseen pronunciations of syntactic words in conversational speech, which is within 4 percent of the maximum success rate achievable by any classifier.
8	Kernels Based on Weighted Levenshtein Distance	To determine the similarity or dissimilarity of two strings in many training algorithms.	The authors replace distance measure in the RBF and exponential kernels and inner product in polynomial and form a new class of string kernels: Levenshtein kernels.	It is demonstrated that incorporating prior knowledge about problems at hand into kernels can improve the performance of kernel machines effectively.
9	Computing the Levenshtein Distance of a Regular Language	In this paper, the problem of computing the edit distance of a regular language is considered.	The authors distinguish two cases depending on whether the given automaton is deterministic or nondeterministic.	The authors show why the problem of computing the edit distance of a given regular language is of polynomial time complexity.

10	Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm	Demonstrating the practicality of the Levenshtein algorithm for plagiarism detection.	Using the Levenshtein Distance algorithm and Smith-Waterman Algorithm.	The authors have described how the Levenshtein distance can be used to change the likely scarcity, which can improve both time and space efficiency.
11	A Normalized Levenshtein Distance Metric	To improve the metric between two strings and satisfy the triangle inequality.	The Generalised Levenshtein Distance algorithm is used.	The main contribution of the paper is to prove that the new distance is a metric valued in $[0, 1]$ and can generally achieve similar accuracies to two other normalized edit distances.
12	String correction using the Damerau-Levenshtein distance	In this paper, the authors focus on the development of algorithms whose space complexity is less and whose actual runtime and energy consumption are less than those of the algorithm of Lowrance and Wagner.	The authors develop space- and cache-efficient algorithms to compute the Damerau-Levenshtein (DL) distance between two strings.	The authors' algorithms are able to handle much larger sequences than earlier algorithms due to the reduction in space requirements.
13	Adapting the Levenshtein Distance to Contextual Spelling Correction	To screen and refine the results obtained by Levenshtein Distance and apply to errors in Arabic words.	The authors developed a system for correcting spelling errors in the Arabic language based on language models and Levenshtein algorithm.	From the results obtained, it is clear that this combination helps improve satisfactorily scheduling solutions returned by our method.

14	A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach	To create an approach that is aimed towards isolated word error correction.	The authors integrate Hamming Distance and n-gram algorithms that have high recall for typing errors and a phonetic spell-checking algorithm in a single novel architecture.	The proposed hybrid methodology has the highest recall rate of the techniques evaluated. This method has a high recall rate and low-computational cost.
15	Spelling Checker Algorithm Methods for Many Languages	To implement spell checking in languages such as Arabic, Chinese, Hindi, Japanese etc.	The approach is divided into several parts, which are: determine the sources of research, define the keyword model for the search process, start inclusion and exclusion criteria, extract data and analyze the result to answer a research question.	The approach is divided into several parts, which are: determine the sources of research, define the keyword model for the search process, start inclusion and exclusion criteria, extract data and analyze the result to answer a research question.

Methodology

We used Levenshtein Distance Algorithm to calculate the similarity between the input word and the target word in both spell check and auto-complete programs.

Here 3 techniques can be used for editing the input word into target word:

-Insertion

-Deletion

-Replacement

THE ALGORITHM:

Step 1: Initialization

I. Set a is the length of document 1 say $d1$, set b is length of document 2 say $d2$.

II. Create a matrix that consists $0 - b$ rows and $0 - a$ columns.

III. Initialize the first row from 0 to a .

IV. Initialize the first column from 0 to b .

Step2: Processing

I. Observe the value of $d2$ (i from 1 to a).

II. Observe the value of $d1$ (j from 1 to b).

III. If the value at $d2[i]$ is equals to value at $d1[j]$, the cost becomes 0.

IV. If the value at $d2[i]$ does not equal $d1[j]$, the cost becomes 1.

V. Set block of matrix $M[d1, d2]$ of the matrix equal to the minimum of:

I. the block immediately above add 1: $M[d2-1, d1]$

II. the block immediately to the left add 1: $M[d2, d1-1] + 1$.

III. The block is diagonally above and to the left adds the cost: $M[d2-1, d1-1] + \text{cost}$.

Step 2 is repeated till the distance $M[a, b]$ value is found.

Step 3: Result [1, 5].

2.1 Computing Techniques

$\text{Dis}(i, j)$ = score of best alignment from $d1_1..d1_i$ to $d2_1.....d2_j$

$\text{Dis}(i-1, j-1) + d(d1_i, d2_j)$ //copy $\text{Dis}(i-1, j) + 1$ //insert

Dis (i, j-1) + 1 //delete

Cost depend upon following factors:

Dis (0, 0) = 0 cost // if both strings are same

Dis (i, 0) = dis (i-1, 0) + 1 = 0 // if source string is empty

Dis (0, j) = dis (0, j-1) + 1 = 0 // if target string is empty [6, 7]

For example:

- Given two words, hello and hello, the Levenshtein distance is zero because the words are identical.
- For the two words helo and hello, it is obvious that there is a missing character "l". Thus to transform the word helo to hello all we need to do is insert that character. The distance, in this case, is 1 because there is only one edit needed.
- On the other hand, for the two words kelo and hello more than just inserting the character "l" is needed. We also need to substitute the character "k" with "h". After such edits, the word kelo is converted into hello. The distance is therefore 2, because there are two operations applied: substitution and insertion.
- For the two words kel and hello, we must first replace "k" with "h", then add a missing "l" followed by an "o" at the end. As a result, the distance is 3 because there are three operations applied.

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
l	3	3	2	1	2	2
m	4	4	3	2	2	3

Here "hello" is input word that has to be transformed into "kelm" which is target word. By this method we can find the levenshtein distance.

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

Implementation

We used Levenshtein Distance Algorithm to calculate the similarity between the input word and the target word in both spell-check and auto-complete programs.

Here 3 techniques can be used for editing the input word into target word:

- Insertion
- Deletion
- Replacement

For example:

- Given two words, **hello** and **hello**, the Levenshtein distance is zero because the words are identical.
- For the two words **helo** and **hello**, it is obvious that there is a missing character "l". Thus to transform the word **helo** to **hello** all we need to do is **insert** that character. The distance, in this case, is 1 because there is only one edit needed.
- On the other hand, for the two words, **kelo** and **hello** more than just inserting the character "l" is needed. We also need to substitute the character "k" with "h". After such edits, the word **kelo** is converted into **hello**. The distance is therefore 2 because there are two operations applied: substitution and insertion.
- For the two words **kel** and **hello**, we must first replace "k" with "h", then add a missing "l" followed by an "o" at the end. As a result, the distance is 3 because there are three operations applied.

The previous discussion is not strategic. We're following predefined steps that could be applied to any two words to transform one word into another. The strategy that we are going to discuss now is how to calculate a distance matrix using dynamic programming. Given two words *A* and *B*, the distance matrix holds the distances between all prefixes of the word *A* and all prefixes of the word *B*.

Here "hello" is input word that has to be transformed into "kelm" which is the target word. By this method, we can find the Levenshtein distance.

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
l	3	3	2	1	2	2
m	4	4	3	2	2	3

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

Spell Checker Code:-

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class SpellChecker {

    private ArrayList words;

    public SpellChecker() {
        words = new ArrayList();
    }

    public void loadDictionary(String fileName) {
        BufferedReader reader;
        FileReader freader;
        String holder;

        try {
            freader = new FileReader(fileName);
            reader = new BufferedReader(freader);

            holder = reader.readLine();

            while (holder != null) {
                words.add(holder);
                holder = reader.readLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("ERROR: That file cannot be found.");
        } catch (IOException e) {
            System.out.println("ERROR: The file may be locked. It
cannot be read from properly.");
        }
    }

    private int findTheMinimum(int a, int b, int c) {
        int minimum = a;

        if (b < minimum) {
            minimum = b;
        }

        if (c < minimum) {
            minimum = c;
        }

        return minimum;
    }

    private int findDifference(String inputWord, String testWord) {
        int n, m;
        char word1Holder, word2Holder;
        int above, left, diagonal, cost;

        n = inputWord.length();
        m = testWord.length();
```

```

        if (n == 0) {
            return m;
        }

        if (m == 0) {
            return n;
        }

        int[][] testMatrix = new int[n + 1][m + 1];

        for (int x = 0; x <= n; x++) {
            testMatrix[x][0] = x;
        }
        for (int x = 0; x <= m; x++) {
            testMatrix[0][x] = x;
        }

        for (int i = 1; i <= n; i++) {
            word1Holder = inputWord.charAt (i - 1);

            for (int j = 1; j <= m; j++) {
                word2Holder = testWord.charAt (j - 1);
                above = testMatrix[i - 1][j] + 1;
                left = testMatrix[i][j - 1] + 1;
                if (word1Holder == word2Holder) {
                    diagonal = testMatrix[i - 1][j - 1];
                } else {
                    diagonal = testMatrix[i - 1][j - 1] + 1;
                }

                testMatrix[i][j] = findTheMinimum(above, left,
diagonal);
            }
        }
    }
}

```

```

return testMatrix[n][m];
    }

    public String findClosestMatch(String inputWord) {
        int smallestWord = -1, smallestDistance = 100, holder;

        for (int x = 0; x < words.size(); x++) {
            holder = findDifference(inputWord,
words.get(x).toString());

            if (holder < smallestDistance) {
                smallestDistance = holder;
                smallestWord = x;
            }
        }

        if (smallestDistance == 0) {
            return "SPELLED CORRECTLY";
        } else {
            return words.get(smallestWord).toString();
        }
    }
    public void addToDictionary(String word) {
        words.add(word);
    }

    public static void main(String args[]) {
        BufferedReader input;
        String wantToContinue = "yes", word, suggestion;
        SpellChecker check = new SpellChecker();

        input = new BufferedReader(new InputStreamReader(System.in));

        try {

            System.out.print("Please enter a dictionary file. >> ");
            check.loadDictionary(input.readLine());

            while(wantToContinue.equalsIgnoreCase("yes")) {

                System.out.print("Please enter a word. >> ");
                word = input.readLine();

                suggestion = check.findClosestMatch(word);

                if (suggestion.equals("SPELLED CORRECTLY")) {
                    System.out.println("That word is correctly
spelled.");
                } else {

                    System.out.print("Did you mean " + suggestion + "? YES/NO >> ");
                    wantToContinue = input.readLine();

                    if (wantToContinue.equalsIgnoreCase("NO"))
                    {
                        System.out.print("Do you want to
add this word to the dictionary? YES/NO >> ");
                        wantToContinue = input.readLine();

                        {
                            if(wantToContinue.equalsIgnoreCase("YES"))

```



```

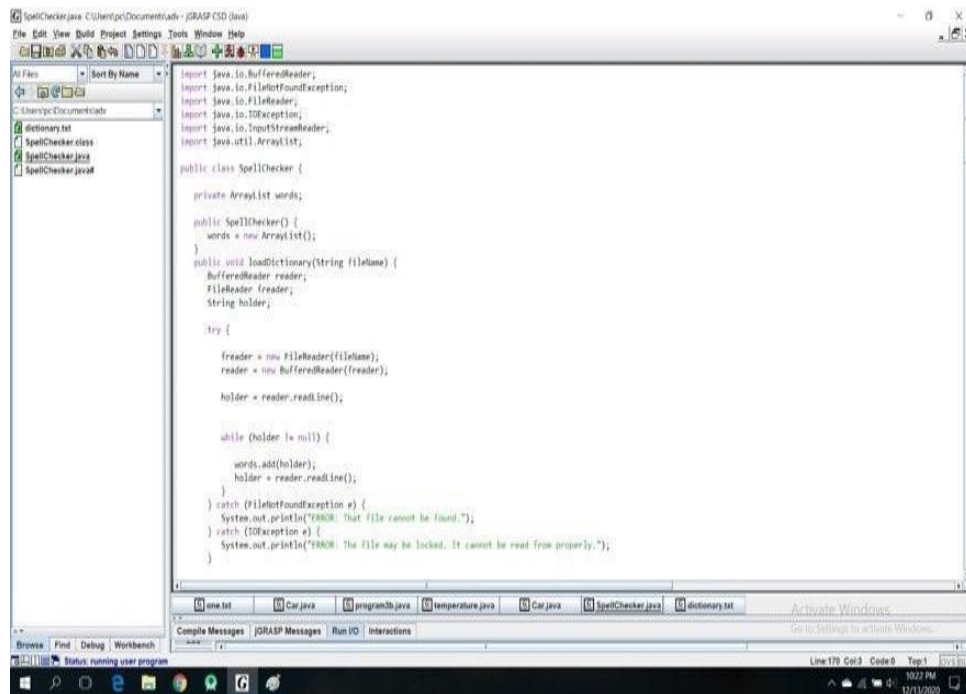
        }
    }
    }

    check.addToDictionary(word);

    system.out.print("Do you want to try another word?
YES/NO >> ");

    wantToContinue = input.readLine();
    }
} catch (IOException e) {
    System.out.println("ERROR: Cannot take keyboard input.");
}
}
}

```



The screenshot shows a Java IDE with the following code in the SpellChecker class:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class SpellChecker {

    private ArrayList words;

    public SpellChecker() {
        words = new ArrayList();
    }

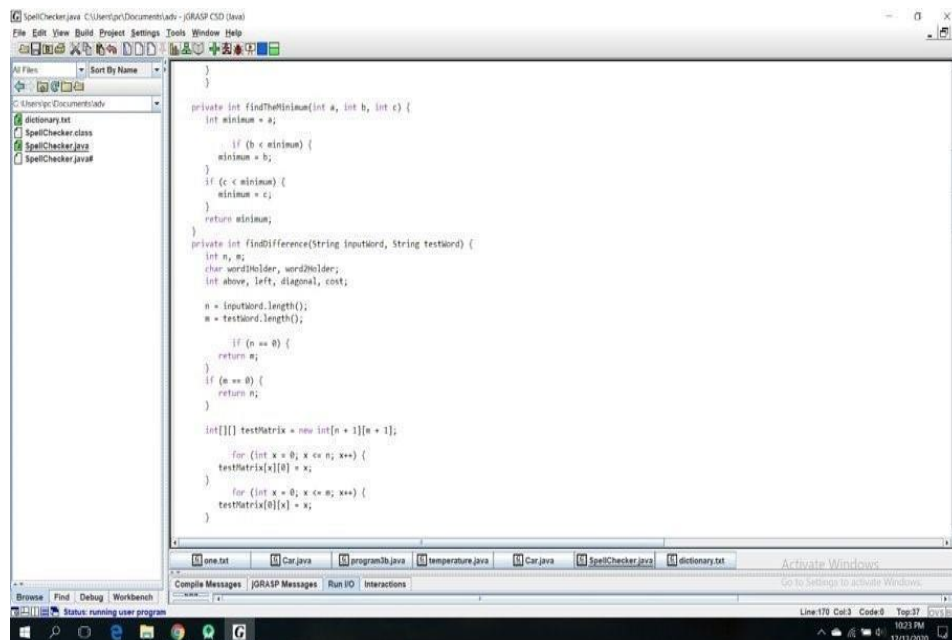
    public void loadDictionary(String filename) {
        BufferedReader reader;
        FileReader freader;
        String holder;

        try {
            freader = new FileReader(filename);
            reader = new BufferedReader(freader);

            holder = reader.readLine();

            while (holder != null) {

                words.add(holder);
                holder = reader.readLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("ERROR: that file cannot be found.");
        } catch (IOException e) {
            System.out.println("ERROR: the file may be locked. It cannot be read from properly.");
        }
    }
}
```



The screenshot shows the continuation of the SpellChecker class with the following code:

```
}

private int findMinimum(int a, int b, int c) {
    int minimum = a;

    if (b < minimum) {
        minimum = b;
    }
    if (c < minimum) {
        minimum = c;
    }
    return minimum;
}

private int findDifference(String inputWord, String testWord) {
    int n, m;
    char wordHolder, wordHolder2;
    int above, left, diagonal, cost;

    n = inputWord.length();
    m = testWord.length();

    if (n == 0) {
        return m;
    }
    if (m == 0) {
        return n;
    }

    int[][] testMatrix = new int[n + 1][m + 1];

    for (int x = 0; x <= n; x++) {
        testMatrix[x][0] = x;
    }

    for (int x = 0; x <= m; x++) {
        testMatrix[0][x] = x;
    }
}
```

```

SpellChecker.java C:\Users\pc\Documents\adv - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
All Files | Sort By Name
C:\Users\pc\Documents\adv
  dictionary.txt
  SpellChecker.class
  SpellChecker.java
  SpellChecker.java~

}
for (int x = 0; x <= n; x++) {
    testMatrix[0][x] = x;
}
for (int i = 1; i <= n; i++) {
    wordHolder = inputWord.charAt(i - 1);
    for (int j = 1; j <= n; j++) {
        wordHolder = testWord.charAt(j - 1);
        above = testMatrix[i - 1][j] + 1;
        left = testMatrix[i][j - 1] + 1;
        if (wordHolder == wordHolder) {
            diagonal = testMatrix[i - 1][j - 1];
        } else {
            diagonal = testMatrix[i - 1][j - 1] + 1;
        }
        testMatrix[i][j] = findTheMinimum(above, left, diagonal);
    }
}
return testMatrix[n][n];
}

public String findClosestMatch(String inputWord) {
    int smallestWord = -1, smallestDistance = 100, holder;
    for (int x = 0; x < words.size(); x++) {
        holder = findDifference(inputWord, words.get(x).toString());
        if (holder < smallestDistance) {
            smallestDistance = holder;
            smallestWord = x;
        }
    }
}

```

one.txt Car.java program3b.java temperature.java Car.java SpellChecker.java dictionary.txt

Compile Messages jGRASP Messages Run IO Interactions

Status: running user program

Line:170 Col:3 Code:0 Top:70

10:27 PM 12/13/2020

```

SpellChecker.java C:\Users\pc\Documents\adv - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
All Files | Sort By Name
C:\Users\pc\Documents\adv
  dictionary.txt
  SpellChecker.class
  SpellChecker.java
  SpellChecker.java~

    if (holder < smallestDistance) {
        smallestDistance = holder;
        smallestWord = x;
    }
}
if (smallestDistance == 0) {
    return "SPILLED CORRECTLY";
} else {
    return words.get(smallestWord).toString();
}

public void addToDictionary(String word) {
    words.add(word);
}

public static void main(String args[]) {
    BufferedReader input;
    String wantToContinue = "yes", word, suggestion;
    SpellChecker check = new SpellChecker();
    input = new BufferedReader(new InputStreamReader(System.in));
    try {
        System.out.print("Please enter a dictionary file. >> ");
        check.loadDictionary(input.readLine());
        while(wantToContinue.equalsIgnoreCase("yes")) {
            System.out.print("Please enter a word. >> ");
            word = input.readLine();

```

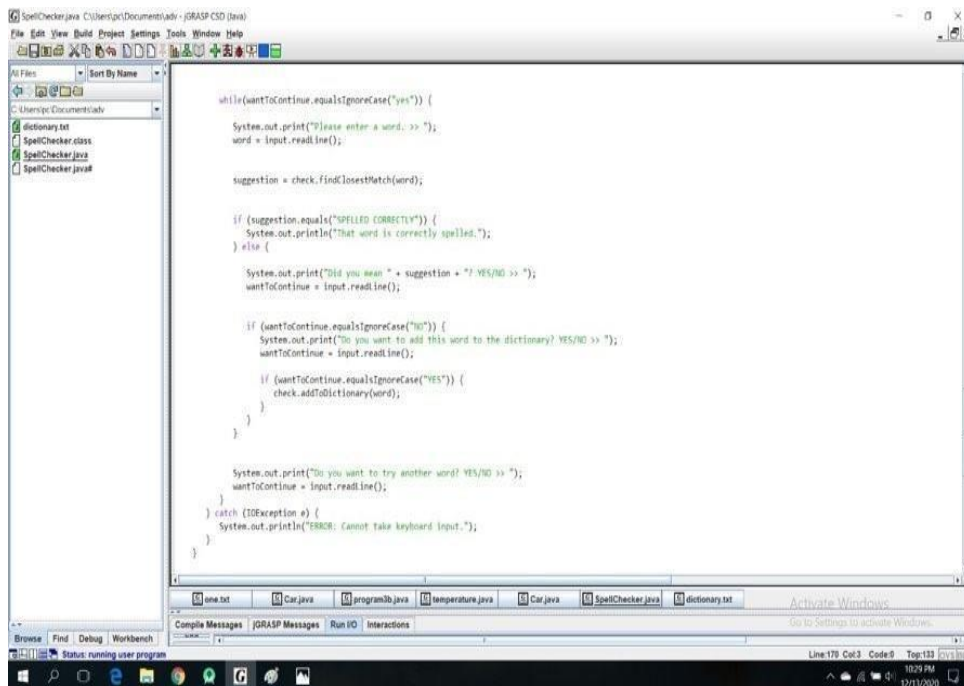
one.txt Car.java program3b.java temperature.java Car.java SpellChecker.java dictionary.txt

Compile Messages jGRASP Messages Run IO Interactions

Status: running user program

Line:170 Col:3 Code:0 Top:103

10:28 PM 12/13/2020



Auto-Complete Code:-

Import numpy

```

def levenshteinDistanceDP(token1, token2):
    distances = numpy.zeros((len(token1) + 1, len(token2) + 1))

    for t1 in range(len(token1) + 1):
        distances[t1][0] = t1

    for t2 in range(len(token2) + 1):
        distances[0][t2] = t2

    a = 0
    b = 0
    c = 0

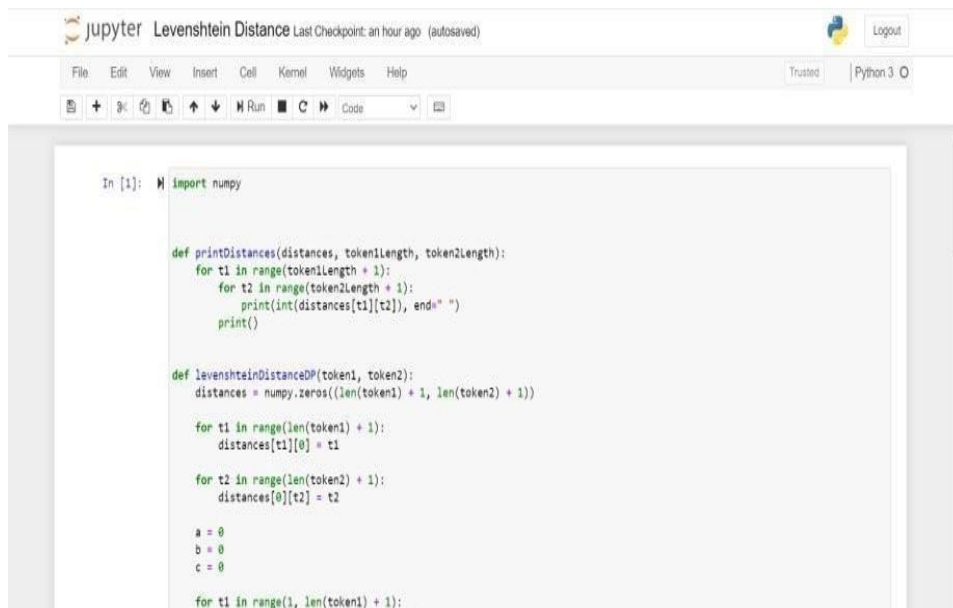
    for t1 in range(1, len(token1) + 1):
        for t2 in range(1, len(token2) + 1):
            if (token1[t1-1] == token2[t2-1]):
                distances[t1][t2] = distances[t1 - 1][t2 - 1]
            else:
                a = distances[t1][t2 - 1]
                b = distances[t1 - 1][t2]
                c = distances[t1 - 1][t2 - 1]

                if (a <= b and a <= c):
                    distances[t1][t2] = a + 1
                elif (b <= a and b <= c):
                    distances[t1][t2] = b + 1
                else:
                    distances[t1][t2] = c + 1

    printDistances(distances, len(token1), len(token2))

```

```
return distances[len(token1)][len(token2)]
```



Jupyter Levenshtein Distance Last Checkpoint: an hour ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [1]: import numpy

def printDistances(distances, token1length, token2length):
    for t1 in range(token1length + 1):
        for t2 in range(token2length + 1):
            print(int(distances[t1][t2]), end=" ")
        print()

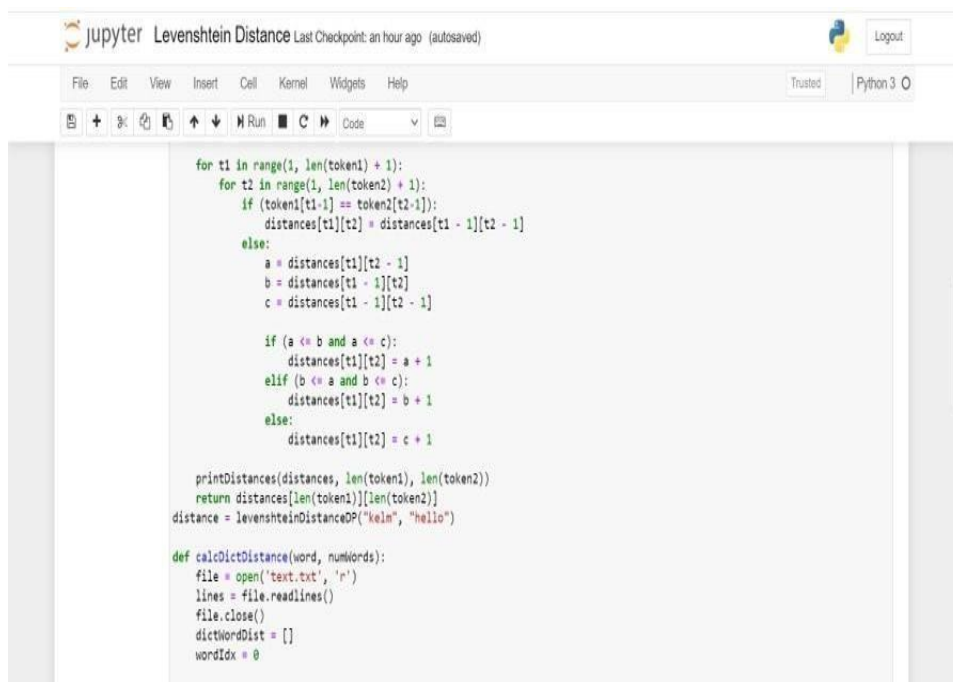
def levenshteinDistanceDP(token1, token2):
    distances = numpy.zeros((len(token1) + 1, len(token2) + 1))

    for t1 in range(len(token1) + 1):
        distances[t1][0] = t1

    for t2 in range(len(token2) + 1):
        distances[0][t2] = t2

    a = 0
    b = 0
    c = 0

    for t1 in range(1, len(token1) + 1):
```



Jupyter Levenshtein Distance Last Checkpoint: an hour ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
        for t1 in range(1, len(token1) + 1):
            for t2 in range(1, len(token2) + 1):
                if token1[t1-1] == token2[t2-1]:
                    distances[t1][t2] = distances[t1 - 1][t2 - 1]
                else:
                    a = distances[t1][t2 - 1]
                    b = distances[t1 - 1][t2]
                    c = distances[t1 - 1][t2 - 1]

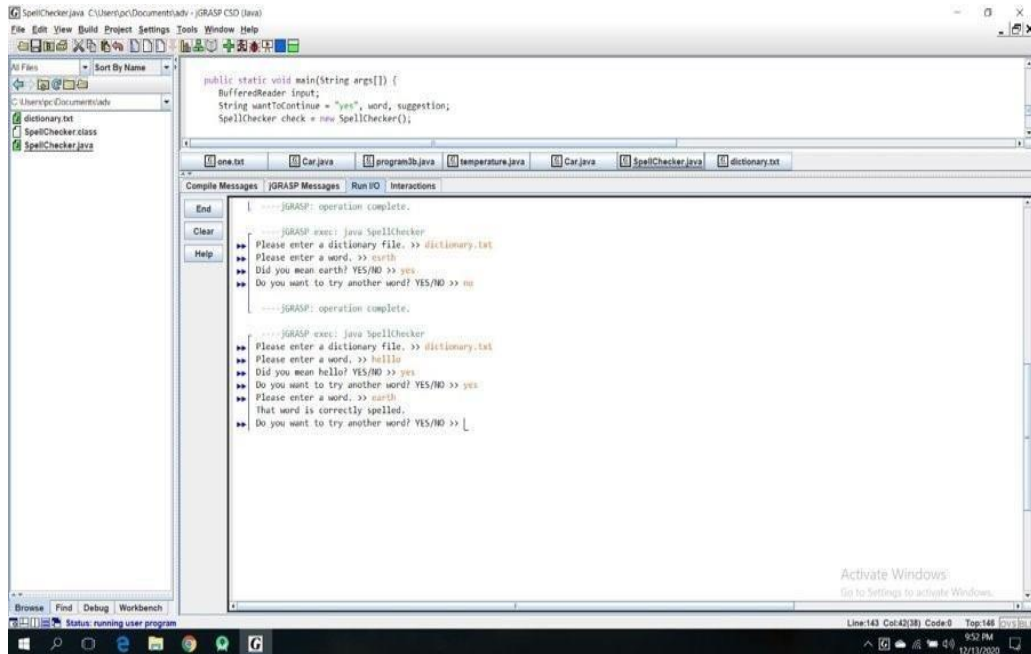
                    if (a <= b and a <= c):
                        distances[t1][t2] = a + 1
                    elif (b <= a and b <= c):
                        distances[t1][t2] = b + 1
                    else:
                        distances[t1][t2] = c + 1

    printDistances(distances, len(token1), len(token2))
    return distances[len(token1)][len(token2)]
distance = levenshteinDistanceDP("kern", "hello")

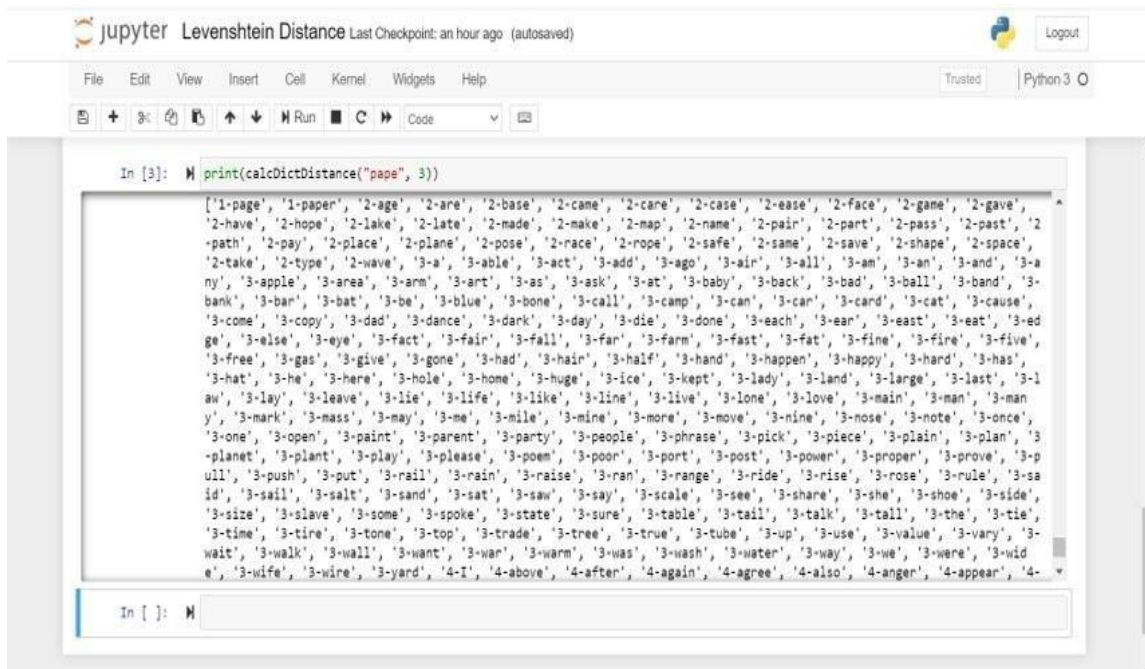
def calcDictDistance(word, numWords):
    file = open('text.txt', 'r')
    lines = file.readlines()
    file.close()
    dictWordDist = []
    wordIdx = 0
```


Results

Spell Checker Output



Autocomplete Output



Conclusion

Here we are solving two problems i.e. spell checking and auto-complete, these both can be solved using the Levenshtein Distance Algorithm where we are deciding the similarity of two words based on their Levenshtein distance.

With the help of similarity, we can auto-complete the words and also check if the word is spelled correctly.

Steps for finding Levenshtein Distance:

Step	Description
1	Set n to be the length of s. Set m to be the length of t. If n = 0, return m and exit. If m = 0, return n and exit. Construct a matrix containing 0..m rows and 0..n columns.
2	Initialize the first row to 0..n. Initialize the first column to 0..m.
3	Examine each character of s (i from 1 to n).
4	Examine each character of t (j from 1 to m).
5	If s[i] equals t[j], the cost is 0. If s[i] doesn't equal t[j], the cost is 1.
6	Set cell d[i,j] of the matrix equal to the minimum of: a. The cell immediately above plus 1: d[i-1,j] + 1. b. The cell immediately to the left plus 1: d[i,j-1] + 1. c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.
7	After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell d[n,m].

LIMITATIONS

It takes a lot of computational power to compare input words with the target words to provide spell checking and auto-complete due to the generation of a distance matrix.

Due to the usage of Dictionary and creating edit distance matrix for each comparison excess memory is used i.e. Space Complexity: $O(m*n)$ where 'm' is the length of the target word and 'n' is the length of the input word.

FURTHER ENHANCEMENTS

Some of the enhancements that we can implement are our usage of dictionary that is stored in unsorted sets where the time complexity to access the words are constant.

We can also instruct the program to not calculate Levenshtein distance if the words are the same which in turn reduces time.

REFERENCES

1. Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library Muhammad Maulana Yulianto, Riza Arifudin, Alamsyah
2. Research on String Similarity Algorithm based on Levenshtein Distance Shengnan Zhang, Yan Hu, Guangrong Bian
3. Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection Mrs.Shital Gaikwad, Prof.Nagaraju Bogiri
4. 4. Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application Sugiarto, I Gede Susrama Mas Diyasa, Ilvi Nur Diana
5. Using the Levenshtein Edit Distance for Automatic Lemmatization: A Case Study for Modern Greek and English Dimitrios P. Lyras, Kyriakos N. Sgarbas, Nikolaos D. Fakotakis
6. Levenshtein Distance, Sequence Comparison and Biological Database Search Bonnie Berger, Michael S. Waterman, and Yun William Yu
7. Learning String-Edit Distance Eric Sven Ristad and Peter N. Yianilos
8. Kernels Based on Weighted Levenshtein Distance Jianhua XU and Xuegong ZHANG
9. Computing the Levenshtein Distance of a Regular Language Stavros Konstantinidis
10. Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm Zhan Su, Byung-Ryul Ahn, Ki-yol Eom, Min-Koo Kang, Jin-Pyung Kim, Moon-Kyun Kim
11. A Normalized Levenshtein Distance Metric Li Yujian and Liu Bo
12. String correction using the Damerau-Levenshtein distance Chunchun Zhao and Sartaj Sahni
13. Adapting the Levenshtein Distance to Contextual Spelling Correction Si Lhoussain Aouragh, Hicham Gueddah, and Abdellah Yousfi
14. A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach Victoria J. Hodge and Jim Austin
15. Spelling Checker Algorithm Methods for Many Languages Novan Zukarnain, Agung Trisetyarso, Bahtiar Saleh Abbas, Chul Ho Kang, and Suparta

PPT Handouts



BMS College of Engineering
Department of Information Science and Engineering

"Autocompletion and spell checker using dynamic programming"

By :-

ADITYA A KAMAT (1BM18IS003)
ARYAN (1BM18IS018)

Abstract

- Autocomplete is a feature in which an application predicts the rest of a word a user is typing. Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user.
- a spell checker (or spelling checker or spell check) is a software feature that checks for misspellings in a text. Spell-checking features are often embedded in software or services, such as a word processor, email client, electronic dictionary, or search engine.

Introduction

- The levenshtein distance algorithm is a dynamic programming algorithm which is a text similarity measure that compares two words and returns a numeric value representing the distance between them.
- The more similar the two words are, the less the distance between them, and vice versa.

Problem Statement

Implement Autocomplete and Spell checking using Levenshtein Distance algorithm

Relevance

- Autocompletion is one of the most important concept applied in smartphone keyboards and is also used in softwares such as Microsoft Word and Google Slides.
- Spell Checker is the core application of softwares such as Grammarly, which performs spell checking while writing emails or drafting reports.

Literature Survey

Sl-NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
1	Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library	To apply the autocomplete feature and spell checking with Levenshtein distance algorithm to get text suggestion in an error data searching in library and determine the level of accuracy on data search trials.	The feature that was used to provide autocomplete and spell checking was the Levenshtein distance algorithm.	Spell checking accuracy rate obtained from the system as 86%.
2	Research on String Similarity Algorithm based on Levenshtein Distance	Similarity algorithm based on Levenshtein Distance is improved.	Longest Common Subsequence (LCS) and Longest Continuous Common Substring (LCCS) are used.	The calculation of string similarity is further improved taking into account the important influence of LCS and LCCS on the computation of string similarity.

Literature Survey

Sl-NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
3	Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection	The aim of this project is to present a novel algorithm to find duplicate objects in the hierarchical structures, like XML files.	Proposed system uses Levenshtein distance algorithm to find duplicates in structured data.	Using Levenshtein Distance algorithm gives better result than Normalized Edit Distance algorithm.
4	Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application	This project results in the form of a letter filing and disposition application by applying the Levenshtein distance algorithm in letter search to speed up the letter search process.	To make it easier to letter archiving and speed up searching for letters, the Levenshtein distance algorithm is used.	Archiving design and letter disposition can display data by the word being searched even though there are some errors when writing the word you want to search for.

Literature Survey

Sl-NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
5	Using the Levenshtein Edit Distance for Automatic Lemmatization: A Case Study for Modern Greek and English	Performing an accurate lemmatization can be a quite difficult and time-consuming task especially for morphologically complex languages with highly inflexional structure, such as the Modern Greek language.	The Levenshtein Distance is implemented on a dictionary-based algorithm in order to achieve the automatic induction of the normalized form (lemma) of regular and mildly irregular words with no direct supervision.	In this paper, a new language independent lemmatization algorithm based on the Levenshtein distance was demonstrated and its performance was evaluated both on Modern Greek and English languages.
6	Levenshtein Distance, Sequence Comparison and Biological Database Search	The advent of modern genomic sequencing and the volume of data it generates has resulted in a return to the problem of local alignment.	Given two biological sequences of length n , the basic problem of biological sequence comparison can be recast as that of determining the Levenshtein distance between them.	Although some modern bioinformatics heuristics using k-mer matching have partially supplanted the direct optimization of Levenshtein distance, the algorithm is still relevant.

Literature Survey

SL_NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
7	Learning String-Edit Distance	Determination of similarity of two strings	The authors implement a string-edit distance that reduces the error rate of the untrained Levenshtein distance by a factor of 4.7, to within 4 percent of the minimum error rate achievable by any classifier.	The authors demonstrate the efficacy of their techniques by correctly recognizing over 87 percent of the unseen pronunciations of syntactic words in conversational speech, which is within 4 percent of the maximum success rate achievable by any classifier.
8	Kernels Based on Weighted Levenshtein Distance	To determine the similarity or dissimilarity of two strings in many training algorithms.	The authors replace distance measure in the RBF and exponential kernels and inner product in polynomial and form a new class of string kernels: Levenshtein kernels.	It is demonstrated that incorporating prior knowledge about problems at hand into kernels can improve the performance of kernel machines effectively.

Literature Survey

SL_NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
9	Computing the Levenshtein Distance of a Regular Language	In this paper, the problem of computing the edit distance of a regular language is considered.	The authors distinguish two cases depending on whether the given automaton is deterministic or nondeterministic.	The authors show why the problem of computing the edit distance of a given regular language is of polynomial time complexity.
10	Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm	Demonstrating the practicality of the Levenshtein algorithm for plagiarism detection.	Using the Levenshtein Distance algorithm and Smith-Waterman Algorithm.	The authors have described how the Levenshtein distance can be used to change the likely scarcity, which can improve both time and space efficiency.

Literature Survey

SL_NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
11	A Normalized Levenshtein Distance Metric	To improve the metric between two strings and satisfy the triangle inequality.	The Generalised Levenshtein Distance algorithm is used.	The main contribution of the paper is to prove that the new distance is a metric valued in $[0, 1]$ and can generally achieve similar accuracies to two other normalized edit distances.
12	String correction using the Damerau-Levenshtein distance	In this paper, the authors focus on the development of algorithms whose space complexity is less and whose actual runtime and energy consumption are less than those of the algorithm of Lowrance and Wagner.	The authors develop space- and cache-efficient algorithms to compute the Damerau-Levenshtein (DL) distance between two strings.	The authors' algorithms are able to handle much larger sequences than earlier algorithms due to the reduction in space requirements.

Literature Survey

SL_NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
13	Adapting the Levenshtein Distance to Contextual Spelling Correction	To screen and refine the results obtained by Levenshtein Distance and apply to errors in Arabic words.	The authors developed a system for correcting spelling errors in the Arabic language based on language models and Levenshtein algorithm.	From the results obtained, it is clear that this combination helps improve satisfactorily scheduling solutions returned by our method.
14	A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach	To create an approach that is aimed towards isolated word error correction.	The authors integrate Hamming Distance and n-gram algorithms that have high recall for typing errors and a phonetic spell-checking algorithm in a single novel architecture.	The proposed hybrid methodology has the highest recall rate of the techniques evaluated. This method has a high recall rate and low computational cost.

Literature Survey

SL_NO	Title of the Paper	Problem Addressed	Authors Approach / Method	Results
15	Spelling Checker Algorithm Methods for Many Languages	To implement spell checking in languages such as Arabic, Chinese, Hindi, Japanese etc.	The approach is divided into several parts, which are: determine the sources of research, define the keyword model for the search process, start inclusion and exclusion criteria, extract data and analyze the result to answer a research question.	It is known that every language has a different method of spell checking. The Damerau-Levenshtein algorithm method is most frequently used for spell checkers.

Implementation

Autocomplete using Levenshtein Distance algorithm:

```

414 def isSquare(numpy):
415     for i in range(1, int(numpy)+1):
416         if numpy%i==0:
417             print(i)
418     return True
419
420 def isPrime(numpy):
421     for i in range(2, int(numpy)+1):
422         if numpy%i==0:
423             return False
424     return True
425
426 def isPrime(numpy):
427     for i in range(2, int(numpy)+1):
428         if numpy%i==0:
429             return False
430     return True
431
432 def isPrime(numpy):
433     for i in range(2, int(numpy)+1):
434         if numpy%i==0:
435             return False
436     return True
437
438 def isPrime(numpy):
439     for i in range(2, int(numpy)+1):
440         if numpy%i==0:
441             return False
442     return True
443
444 def isPrime(numpy):
445     for i in range(2, int(numpy)+1):
446         if numpy%i==0:
447             return False
448     return True
449
450 def isPrime(numpy):
451     for i in range(2, int(numpy)+1):
452         if numpy%i==0:
453             return False
454     return True
455
456 def isPrime(numpy):
457     for i in range(2, int(numpy)+1):
458         if numpy%i==0:
459             return False
460     return True
461
462 def isPrime(numpy):
463     for i in range(2, int(numpy)+1):
464         if numpy%i==0:
465             return False
466     return True
467
468 def isPrime(numpy):
469     for i in range(2, int(numpy)+1):
470         if numpy%i==0:
471             return False
472     return True
473
474 def isPrime(numpy):
475     for i in range(2, int(numpy)+1):
476         if numpy%i==0:
477             return False
478     return True
479
480 def isPrime(numpy):
481     for i in range(2, int(numpy)+1):
482         if numpy%i==0:
483             return False
484     return True
485
486 def isPrime(numpy):
487     for i in range(2, int(numpy)+1):
488         if numpy%i==0:
489             return False
490     return True
491
492 def isPrime(numpy):
493     for i in range(2, int(numpy)+1):
494         if numpy%i==0:
495             return False
496     return True
497
498 def isPrime(numpy):
499     for i in range(2, int(numpy)+1):
500         if numpy%i==0:
501             return False
502     return True
503
504 def isPrime(numpy):
505     for i in range(2, int(numpy)+1):
506         if numpy%i==0:
507             return False
508     return True
509
510 def isPrime(numpy):
511     for i in range(2, int(numpy)+1):
512         if numpy%i==0:
513             return False
514     return True
515
516 def isPrime(numpy):
517     for i in range(2, int(numpy)+1):
518         if numpy%i==0:
519             return False
520     return True
521
522 def isPrime(numpy):
523     for i in range(2, int(numpy)+1):
524         if numpy%i==0:
525             return False
526     return True
527
528 def isPrime(numpy):
529     for i in range(2, int(numpy)+1):
530         if numpy%i==0:
531             return False
532     return True
533
534 def isPrime(numpy):
535     for i in range(2, int(numpy)+1):
536         if numpy%i==0:
537             return False
538     return True
539
540 def isPrime(numpy):
541     for i in range(2, int(numpy)+1):
542         if numpy%i==0:
543             return False
544     return True
545
546 def isPrime(numpy):
547     for i in range(2, int(numpy)+1):
548         if numpy%i==0:
549             return False
550     return True
551
552 def isPrime(numpy):
553     for i in range(2, int(numpy)+1):
554         if numpy%i==0:
555             return False
556     return True
557
558 def isPrime(numpy):
559     for i in range(2, int(numpy)+1):
560         if numpy%i==0:
561             return False
562     return True
563
564 def isPrime(numpy):
565     for i in range(2, int(numpy)+1):
566         if numpy%i==0:
567             return False
568     return True
569
570 def isPrime(numpy):
571     for i in range(2, int(numpy)+1):
572         if numpy%i==0:
573             return False
574     return True
575
576 def isPrime(numpy):
577     for i in range(2, int(numpy)+1):
578         if numpy%i==0:
579             return False
580     return True
581
582 def isPrime(numpy):
583     for i in range(2, int(numpy)+1):
584         if numpy%i==0:
585             return False
586     return True
587
588 def isPrime(numpy):
589     for i in range(2, int(numpy)+1):
590         if numpy%i==0:
591             return False
592     return True
593
594 def isPrime(numpy):
595     for i in range(2, int(numpy)+1):
596         if numpy%i==0:
597             return False
598     return True
599
600 def isPrime(numpy):
601     for i in range(2, int(numpy)+1):
602         if numpy%i==0:
603             return False
604     return True
605
606 def isPrime(numpy):
607     for i in range(2, int(numpy)+1):
608         if numpy%i==0:
609             return False
610     return True
611
612 def isPrime(numpy):
613     for i in range(2, int(numpy)+1):
614         if numpy%i==0:
615             return False
616     return True
617
618 def isPrime(numpy):
619     for i in range(2, int(numpy)+1):
620         if numpy%i==0:
621             return False
622     return True
623
624 def isPrime(numpy):
625     for i in range(2, int(numpy)+1):
626         if numpy%i==0:
627             return False
628     return True
629
630 def isPrime(numpy):
631     for i in range(2, int(numpy)+1):
632         if numpy%i==0:
633             return False
634     return True
635
636 def isPrime(numpy):
637     for i in range(2, int(numpy)+1):
638         if numpy%i==0:
639             return False
640     return True
641
642 def isPrime(numpy):
643     for i in range(2, int(numpy)+1):
644         if numpy%i==0:
645             return False
646     return True
647
648 def isPrime(numpy):
649     for i in range(2, int(numpy)+1):
650         if numpy%i==0:
651             return False
652     return True
653
654 def isPrime(numpy):
655     for i in range(2, int(numpy)+1):
656         if numpy%i==0:
657             return False
658     return True
659
660 def isPrime(numpy):
661     for i in range(2, int(numpy)+1):
662         if numpy%i==0:
663             return False
664     return True
665
666 def isPrime(numpy):
667     for i in range(2, int(numpy)+1):
668         if numpy%i==0:
669             return False
670     return True
671
672 def isPrime(numpy):
673     for i in range(2, int(numpy)+1):
674         if numpy%i==0:
675             return False
676     return True
677
678 def isPrime(numpy):
679     for i in range(2, int(numpy)+1):
680         if numpy%i==0:
681             return False
682     return True
683
684 def isPrime(numpy):
685     for i in range(2, int(numpy)+1):
686         if numpy%i==0:
687             return False
688     return True
689
690 def isPrime(numpy):
691     for i in range(2, int(numpy)+1):
692         if numpy%i==0:
693             return False
694     return True
695
696 def isPrime(numpy):
697     for i in range(2, int(numpy)+1):
698         if numpy%i==0:
699             return False
700     return True
701
702 def isPrime(numpy):
703     for i in range(2, int(numpy)+1):
704         if numpy%i==0:
705             return False
706     return True
707
708 def isPrime(numpy):
709     for i in range(2, int(numpy)+1):
710         if numpy%i==0:
711             return False
712     return True
713
714 def isPrime(numpy):
715     for i in range(2, int(numpy)+1):
716         if numpy%i==0:
717             return False
718     return True
719
720 def isPrime(numpy):
721     for i in range(2, int(numpy)+1):
722         if numpy%i==0:
723             return False
724     return True
725
726 def isPrime(numpy):
727     for i in range(2, int(numpy)+1):
728         if numpy%i==0:
729             return False
730     return True
731
732 def isPrime(numpy):
733     for i in range(2, int(numpy)+1):
734         if numpy%i==0:
735             return False
736     return True
737
738 def isPrime(numpy):
739     for i in range(2, int(numpy)+1):
740         if numpy%i==0:
741             return False
742     return True
743
744 def isPrime(numpy):
745     for i in range(2, int(numpy)+1):
746         if numpy%i==0:
747             return False
748     return True
749
750 def isPrime(numpy):
751     for i in range(2, int(numpy)+1):
752         if numpy%i==0:
753             return False
754     return True
755
756 def isPrime(numpy):
757     for i in range(2, int(numpy)+1):
758         if numpy%i==0:
759             return False
760     return True
761
762 def isPrime(numpy):
763     for i in range(2, int(numpy)+1):
764         if numpy%i==0:
765             return False
766     return True
767
768 def isPrime(numpy):
769     for i in range(2, int(numpy)+1):
770         if numpy%i==0:
771             return False
772     return True
773
774 def isPrime(numpy):
775     for i in range(2, int(numpy)+1):
776         if numpy%i==0:
777             return False
778     return True
779
780 def isPrime(numpy):
781     for i in range(2, int(numpy)+1):
782         if numpy%i==0:
783             return False
784     return True
785
786 def isPrime(numpy):
787     for i in range(2, int(numpy)+1):
788         if numpy%i==0:
789             return False
790     return True
791
792 def isPrime(numpy):
793     for i in range(2, int(numpy)+1):
794         if numpy%i==0:
795             return False
796     return True
797
798 def isPrime(numpy):
799     for i in range(2, int(numpy)+1):
800         if numpy%i==0:
801             return False
802     return True
803
804 def isPrime(numpy):
805     for i in range(2, int(numpy)+1):
806         if numpy%i==0:
807             return False
808     return True
809
810 def isPrime(numpy):
811     for i in range(2, int(numpy)+1):
812         if numpy%i==0:
813             return False
814     return True
815
816 def isPrime(numpy):
817     for i in range(2, int(numpy)+1):
818         if numpy%i==0:
819             return False
820     return True
821
822 def isPrime(numpy):
823     for i in range(2, int(numpy)+1):
824         if numpy%i==0:
825             return False
826     return True
827
828 def isPrime(numpy):
829     for i in range(2, int(numpy)+1):
830         if numpy%i==0:
831             return False
832     return True
833
834 def isPrime(numpy):
835     for i in range(2, int(numpy)+1):
836         if numpy%i==0:
837             return False
838     return True
839
840 def isPrime(numpy):
841     for i in range(2, int(numpy)+1):
842         if numpy%i==0:
843             return False
844     return True
845
846 def isPrime(numpy):
847     for i in range(2, int(numpy)+1):
848         if numpy%i==0:
849             return False
850     return True
851
852 def
```

Implementation

Autocomplete using Levenshtein Distance algorithm:

OUTPUT:

[illegible]

Implementation

Spell check using Levenshtein Distance Algorithm

[illegible]

Implementation

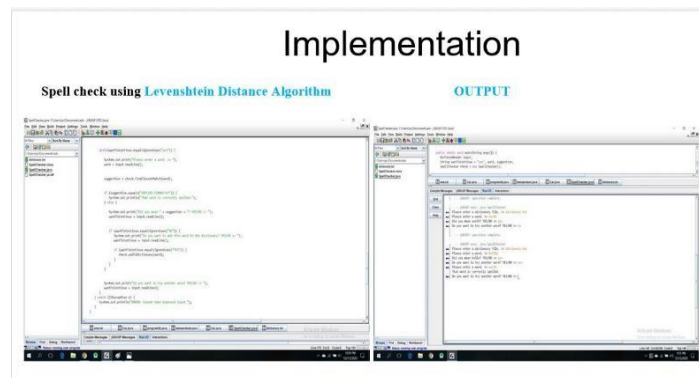
Spell check using Levenshtein Distance Algorithm

The screenshot shows the RStudio IDE with the following content:

R Console:

```
summary(mtcars)
      mpg   cyl  disp    hp  wt    qsec    vs    am  gear  carb
 14.700   6.0 160.0   113 3.51  16.99   0.00  0.01   4     4
 15.833   6.1 161.3   115 3.57  17.02   0.00  0.01   4     4
 19.700   4.4 146.7   125 3.57  16.46   0.01  0.01   4     4
 22.800   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 24.400   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 26.010   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 27.800   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 29.900   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 32.410   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 34.300   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 36.100   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 38.400   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 40.400   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 42.100   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 44.400   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 46.300   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 48.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 50.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 52.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 54.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 56.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 58.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 60.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 62.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 64.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 66.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 68.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 70.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 72.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 74.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 76.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 78.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 80.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 82.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 84.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 86.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 88.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 90.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 92.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 94.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 96.000   4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 98.000   4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 100.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 102.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 104.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 106.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 108.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 110.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 112.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 114.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 116.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 118.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 120.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 122.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 124.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 126.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 128.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 130.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 132.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 134.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 136.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 138.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 140.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 142.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 144.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 146.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 148.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 150.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 152.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 154.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 156.000  4.6 146.7   125 3.57  16.46   0.01  0.01   4     4
 158.000  4.6 161.3   122 3.59  16.86   0.01  0.01   4     4
 160.000  4.6 146.
```

Implementation



References

1. **Autocomplete and Spell Checking Levenshtein Distance Algorithm to Getting Text Suggest Error Data Searching in Library**
Muhammad Maulana Yulianto, Riza Arifudin, Alamsyah
2. **Research on String Similarity Algorithm based on Levenshtein Distance**
Shengnan Zhang, Yan Hu, Guangrong Bian
3. **Levenshtein Distance Algorithm for Efficient and Effective XML Duplicate Detection**
Mrs.Shital Gaikwad, Prof.Nagaraju Bogiri
4. **Levenshtein Distance Algorithm Analysis on Enrollment and Disposition of Letters Application**
Sugianto, I Gede Susrama Mas Diyasa, Iki Nur Diana
5. **Using the Levenshtein Edit Distance for Automatic Lemmatization: A Case Study for Modern Greek and English**
Dimitrios P. Lyras, Kyriakos N. Sgarbas, Nikolaos D. Fakotakis
6. **Levenshtein Distance, Sequence Comparison and Biological Database Search**
Bonnie Berger, Michael S. Waterman, and Yun William Yu

References

7. **Learning String-Edit Distance**
Eric Sven Ristad and Peter N. Yianilos
8. **Kernels Based on Weighted Levenshtein Distance**
Jianhua XU and Xuegong ZHANG
9. **Computing the Levenshtein Distance of a Regular Language**
Stavros Konstantinidis
10. **Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm**
Zhan Su, Byung-Ryul Ahn, Ki-yol Eom, Min-koo Kang, Jin-Pyung Kim, Moon-Kyun Kim
11. **A Normalized Levenshtein Distance Metric**
Li Yujian and Liu Bo
12. **String correction using the Damerau-Levenshtein distance**
Chunchun Zhao and Sartaj Sahni
13. **Adapting the Levenshtein Distance to Contextual Spelling Correction**
Si l'houssein Aouragh, Hicham Guendiah and Abdellah Youfi
14. **A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach**
Victoria J. Hodge and Jim Austin
15. **Spelling Checker Algorithm Methods for Many Languages**
Novan Zukarnain, Agung Trisetiyo, Bahtiar Saleh Abbas, Chul Ho Kang and Suparta Wayan