Information Retrieval from Greek Documents: Evaluations and Improvements

Haralampos Varsamis

Spring Semester, 2022

Contents

1 Abstract

There are several search engines and they are used everyday. Because they are used so often there is a tendency to wonder which is preferable based on the needs of the user. Alongside the quality of a search engine to find relevant documents, an important indicator of its quality is its tolerance to errors and the ability to correct misspelled and mispronounced words that can negatively affect various tasks in Information Retrieval. For the Greek language there have been approaches with the usage of edit distance and phonetic algorithms but none with keyboard mistypes. In this paper, we start off by evaluating some search engines on a manually created dataset and we present an algorithm on keyboard mistypes correction. A pseudocode and its functionality are explained briefly and execution results are presented and discussed.

2 Introduction

In some manner, search engines are part of our day life. Many times, we find ourselves wondering which search engine is superior and why. The evaluation of a few search engines based on a manually created dataset will be covered in the sections that follow. To use a search engine, words are typed and searched within a corpus. While typing it is possible to make mistakes. These mistakes can occur due to incorrect knowledge of how the word is written or due to mistypes of some adjacent character.These problems can be handled with edit-related distances (e.g. the Levenstein distance Levenshtein (1966)),with phonetic matching algorithms(e.g. SoundexGR of Antrei Kavros and Yannis Tzitzikas (2022)).

Our approach is to take into consideration mistypes that may occur when typing a word. An initial input and a collection of potentially viable words are passed to the algorithm, which then attempts to reduce the Edit Distance between these words by modifying adjacent letters in the initial input.

3 Datasets

3.1 Dataset for Search Engines Evaluation

The dataset was not found online and had to be created by hand (could not find online datasets with queries,documents and relevance factors).

The queries were chosen from the TREC Fair Ranking 2020, which were translated to Greek. The documents were selected by searching data on each query and keeping 1-3 documents that appeared to be the most relevant(judged as relevant by human).Totally, 56 queries and 117 documents were collected to evaluate the search engines.

Table 1: Description of fields of a Document object

ID	Title	Body
Unique identifier of the Document	The title of the document	the body/abstract of the document

3.1.1 Document Format

3.1.2 Query Format

Table 2: Description of fields of a Query object

ID	Query	RelevantDocs
Unique identifier of the query	The query to be searched in the documents(one or more words)	A List of Document IDs and their Relevance Rank(each documents ranks at a position, 0 means most relevant,1 means second most relevant,etc...)

3.2 Dataset for Keyboard Mistype Correction Evaluation

The Dataset was taken from the resources SoundexGR−Resources repository and it was used to generate other files with different patterns such as substitution of 1,2 or 3 characters and combination of substitution and character addition that can be found at files with mistyped words. Specifically, each file consists of lines where the words are seperated by comma and the first word is the correct one and the following ones are the words that are misspelled/mistyped. Finally, the different versions that were used for the keyboard correction algorithm are:

Table 3: Description of Datasets used in the evaluation of Keyboard Mistype Correction

Filename	Correct	Incorrect	Description
OnePerWordSubstitutions	293	1758	One valid character is substituted with an adjacent(incorrect) one.Each word has the substitution applied to a different position(6 in total).
TwoPerWordSubstitutions	293	1465	Two valid characters are substituted with adjacent(incorrect) ones.Each word has the substitutions applied to a different position(5 in total).
ThreePerWordSubstitutions	293	1172	Three valid characters are substituted with adjacent(incorrect) ones.Each word has the substitutions applied to a different position(4 in total).
FourPerWordSubstitutions	293	1172	Four valid characters are substituted with adjacent(incorrect) ones.Each word has the substitutions applied to a different position(4 in total)
OneSubstitutionAndAddition	293	1758	One valid character is substituted with an adjacent(incorrect) one and a random character is added.Each word has the substitution applied to a different position and the addition at a random one(6 in total)
TwoSubstitutionsAndAddition	293	1465	Two valid characters are substituted with adjacent(incorrect) ones and a random character is added.Each word has the substitutions applied to a different position and the addition at a random one(5 in total)
TwoSubstitutionsAndTwoAdditions	293	1465	Two valid characters are substituted with adjacent(incorrect) ones and two random characters are added.Each word has the substitutions applied to a different position and the additions at a random one(5 in total)

Filename	Correct	Incorrect	Description
TwoSubstitutionsAndThreeAdditions	293	1465	Two valid characters are substituted with adjacent(incorrect) ones and three random characters are added.Each word has the substitutions applied to a different position and the additions at a random one(5 in total)
OneSubstitutionAndDeletion	293	1758	One valid character is substituted with an adjacent(incorrect) one and a random character(not the one that was substituted) is deleted.Each word has the substitution applied to a different position and the deletion at a random one(6 in total)
TwoSubstitutionsAndDeletion	293	1465	Two valid characters are substituted with adjacent(incorrect) ones and a random character(not the ones that were substituted) is deleted.Each word has the substitutions applied to different positions and the deletion at a random one(5 in total)

4 Search Engines Evaluations

Three search engines were chosen to be evaluated:

- Elasticsearch
- Solr
- Algolia

4.1 Evaluation Cases

To assess the search engines, stopword removal and stemming were used. In general, stopword removal refers to the process of simply removing the words that occur commonly across all the documents in the corpus. Typically, articles and pronouns are generally classified as stop words. Stemming refers to the process of reducing inflected (or derived) words to their word base or root form. As a result, the evaluation of each search engine will consist of comparing the various combinations of stopword removal and stemming.The possible cases are the following:

- Remove Stopwords and apply Stemming
- Remove Stopwords and do not apply Stemming
- Do not remove Stopwords and apply Stemming
- Do not remove Stopwords and do not apply Stemming

4.2 Evaluation Metrics

4.2.1 Precision

Precision is defined as the ratio of the number of relevant and retrieved documents(number of items retrieved that are actually useful to the user and match his search need) to the number of total retrieved documents from the query. Formally:

$$Precision = \frac{|\{relevant\,documents\} \cap \{retrieved\,documents\}|}{|retrieved\,documents|}$$

4.2.2 Recall

Recall is the fraction of the relevant documents that are successfully retrieved. For example, for a text search on a set of documents, recall is the number of correct results divided by the number of results that should have been returned. Formally:

$$Recall = \frac{|\{relevant\,documents\} \cap \{retrieved\,documents\}|}{|\{relevant\,documents\}|}$$

4.2.3 F-Measure

F-Measure or F-Score provides a way to combine both precision and recall into a single measure that captures both properties. This is the harmonic mean of the two fractions. Formally:

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

4.2.4 Fall-out

Fall-out is the proportion of non-relevant documents that are retrieved, out of all non-relevant documents available. Formally:

$$Fall - out = \frac{|\{non - relevant\,documents\} \cap \{retrieved\,documents\}|}{|\{non - relevant\,documents\}|}$$

4.3 Results

In this section, the results collected for each search engine are displayed and analyzed and then a comparison between the search engines follows.

It should be mentioned that each search engine evaluates document relevancy score differently, but for our purposes, the only thing that matters is that the search engine returns the proper documents. It is also important to note that the results are highly affected by the dataset, which implies that the number of stopwords within the query or the document has an impact on the final results.

4.3.1 Elasticsearch

Table 4: Results of Elasticsearch evaluation

	Precision	Recall	F-Score	Fallout
Remove Stopwords and apply Stemming	0.73	0.99	0.79	0.015
Remove Stopwords and do not apply Stemming	0.78	0.89	0.80	0.006
Do not remove Stopwords and apply Stemming	0.62	0.99	0.70	0.026
Do not remove Stopwords and do not apply Stemming	0.65	0.89	0.70	0.020

Between these cases, we can see that the case with the highest Precision is *Remove Stopwords and do not apply Stemming*, with a value of 0.78, and the cases with the highest Recall are *Remove Stopwords and apply Stemming* and *Do not remove Stopwords and apply stemming*, with a value of 0.99. We use F-Score to combine the results since Precision and Recall are inversely proportional. With a value of 0.80, the case with the highest F-Score value is *Remove Stopword and Do Not Apply Stemming*. Finally, we can see that the case with the highest F-Score has also the lowest Fall-Out value of 0.006.

4.3.2 Solr

When it comes to querying, Solr offers two options. The first method is to obtain documents that include all query tokens (Logical AND operator, Table 5), whereas the second method is to retrieve documents that contain at least one of the query tokens (Logical OR operator, Table 6).

Table 5: Results of Solr evaluation with AND logical operator between query tokens

	Precision	Recall	F-Score	Fallout
Remove Stopwords and apply Stemming	0.77	0.82	0.77	0.005
Remove Stopwords and do not apply Stemming	0.71	0.66	0.65	0.003
Do not remove Stopwords and apply Stemming	0.77	0.82	0.77	0.008
Do not remove Stopwords and do not apply Stemming	0.71	0.67	0.66	0.006

For Table 5, we can see that the cases with the highest Precision are also the ones with the highest Recall and they are *Remove Stopwords and apply Stemming* and *Do not remove Stopwords and apply Stemming*, with values of 0.77 and 0.82 respectively. We use F-Score to combine the results since Precision and Recall are inversely proportional. With a value of 0.77, the cases with the highest F-Score values are *Remove Stopwords and apply Stemming* and *Do not remove Stopwords and apply Stemming*. Finally, we can see that here, the cases with the highest F-Score do not have the lowest Fall-Out values(0.005 and 0.008 where the lowest is that of *Remove Stopwords and do not apply Stemming* with 0.003).

Table 6: Results of Solr evaluation with OR logical operator between query tokens

	Precision	Recall	F-Score	Fallout
Remove Stopwords and apply Stemming	0.73	0.99	0.79	0.015
Remove Stopwords and do not apply Stemming	0.75	0.87	0.78	0.009
Do not remove Stopwords and apply Stemming	0.62	0.99	0.70	0.026
Do not remove Stopwords and do not apply Stemming	0.65	0.89	0.70	0.020

As for Table 6, we can see that the case with the highest Precision is *Remove Stopwords and do not apply Stemming*, with a value of 0.75, and the cases

with the highest Recall are *Remove Stopwords and apply Stemming* and *Do not remove Stopwords and apply stemming*, with a value of 0.99. We use F-Score to combine the results since Precision and Recall are inversely proportional. With a value of 0.79, the case with the highest F-Score value is *Remove Stopword and Apply Stemming*. Finally, we can see that the case with the highest F-Score does not have the lowest Fall-Out value(0.015 where the lowest is that of *Remove Stopwords and do not apply Stemming* with 0.009).

We can observe that Table 6's F-Score values are higher than Table 5's from comparing the two tables (Due to higher scores in Recall). Based on this, we could consider that the Logical OR operator between query tokens is a better approach to retrieve relevant documents.

4.3.3 Algolia

Algolia does not support stemming and they do not plan to implement it(see here). As a result, the evaluation will consist only of stopword removal.

Table 7: Results of Algolia evaluation without Stemming

	Precision	Recall	F-Score	Fallout
Remove Stopwords and do not apply Stemming	0.78	0.74	0.75	0.0006
Do not remove Stopwords and do not apply Stemming	0.81	0.78	0.78	0.0006

Removing the stopwords decreases the performance overall. The case *Do not remove Stopwords* has both the highest Precision and Recall scores with values of 0.81 and 0.78 respectively. With a value of 0.78, the case with the highest F-Score value is *Do not remove Stopwords*. Finally, we can see that both cases have lowest Fall-Out value 0.0006.

4.3.4 Search Engines Comparison

Because title cases are long they are simplified by converting them to their initials.So each case will be:

- Remove Stopwords and apply Stemming: RSAS
- Remove Stopwords and do not apply Stemming: RS
- Do not remove Stopwords and apply Stemming: AS
- Do not remove Stopwords and do not apply Stemming: NONE

Note that for each case the initials mentioned are only the ones that do not contain negation.

The results for different metrics are merged in a chart for all search engines in the figures that follow. As seen in Figure 1, filter *RS* achieved the

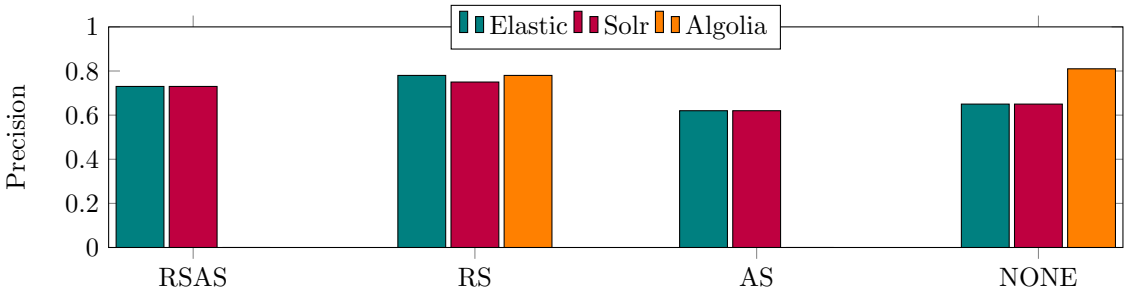

Figure 1: Search engine Precision for each different case

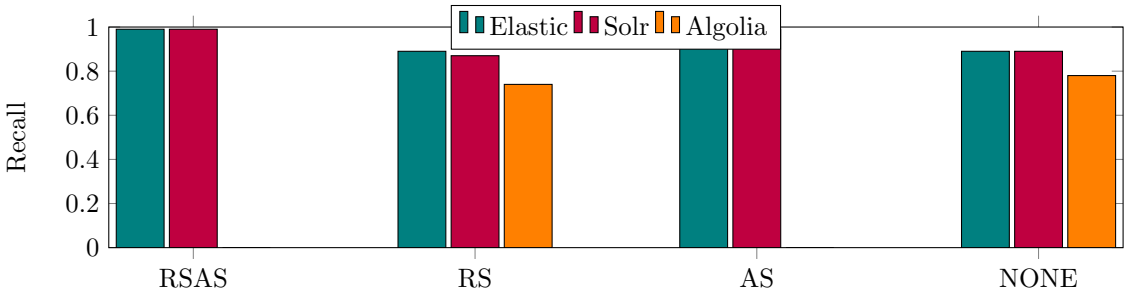

Figure 2: Recall of search engines for each different case

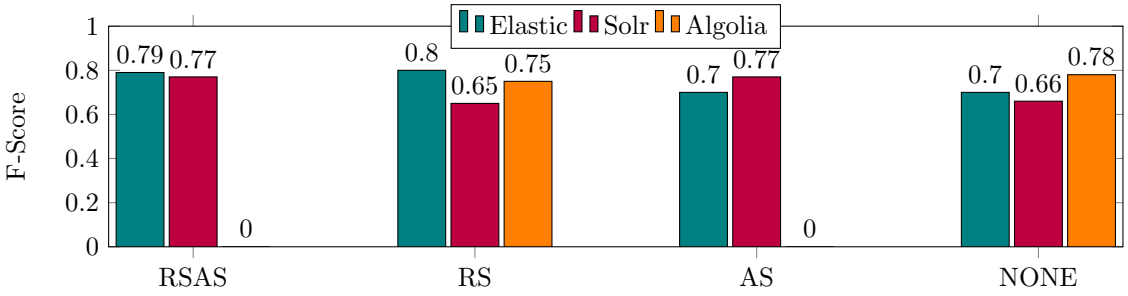

Figure 3: F-Score of search engines for each different case

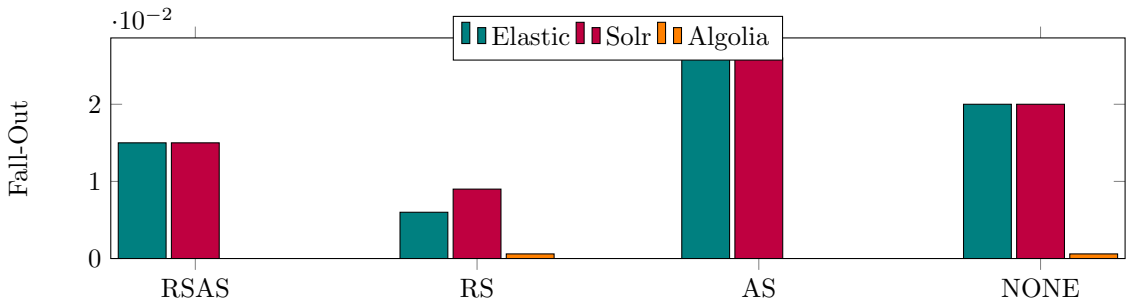

Figure 4: Fall-Out of search engines for each different case

highest Precision for all search engines(as a whole) with values of 0.78, 0.75, and 0.78. Individually, Algolia achieved the greatest Precision with no filters applied(*NONE*) with a value of 0.81, followed by Elasticsearch with 0.78 and Solr with 0.77. Figure 2 shows that for Elasticsearch and Solr, cases *RSAS* and *AS* earned the highest Recall with a value of 0.99; for Algolia, which does not allow stemming, the case *NONE* achieved the highest Recall with a value of 0.78. Figure 3 shows that Elasticsearch earned the greatest F-Score with *RS* filter with a value of 0.8, Solr achieved the highest F-Score with *RSAS* filter with a value of 0.77, and Algolia achieved the highest F-Score with filter *NONE* with a value of 0.78. Finally, Figure 4 shows that Algolia has the lowest fall-out values, with value 0.0006 , whereas Solr has the greatest fall-out values in all scenarios ranging between 0.009 and 0.026, followed by Elasticsearch with small differences.

By comparing the different F-Scores, it is feasible to claim that the best performance for this dataset was given by Elasticsearch.

5 Keyboard Mistype Correction

Keyboard typing may be challenging at times. When trying to write down a certain term, someone may be misled by their speed, their familiarity with the keyboard, their large fingers, or even the lighting in the area. When using a keyboard, individuals frequently input the wrong adjacent character. We introduce Keyboard Mistype Correction to address this issue. This algorithm looks for wrong letters to swap them for the proper ones using a collection of potentially legitimate words and a misspelled phrase as input. Doesn't Edit Distance handle scenarios like this, one would wonder. Does this actually help? Although not always, the answer is yes. This section will look at how this algorithm operates and when it is superior than other methods.

5.1 Keyboard Types

There are 4 main keyboard layout types, these are:

- AZERTY
- QWERTY
- QWERTZ
- QZERTY

The names are derived from the first 6 letters in the keyboard layout:

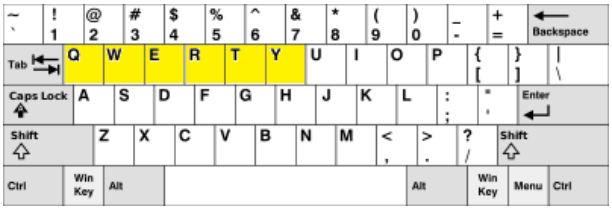

Figure 5: An image of a QWERTY Keyboard

The QWERTY keyboard layout is, by far, the most widespread layout in use. It is crucial to note that the Greek layout is based on the QWERTY layout, with glyphs being allocated to keys that carry glyphs that are as similar to those in QWERTY as feasible. Greek users who are already comfortable with the QWERTY keyboard may enter Latin characters more quickly as a result, saving time for those who are not.

5.2 The Algorithm Keyboard Mistype Correction

In this section the algorithm is presented and explained. Function *GetRowColumnAndUpper* is responsible to find the row, column of the character on the keyboard that was given as an argument and if it is an upper case character or a lower case one. Function *RemoveAccentuations* removes accentuations from the character given as argument or returns the character as given if it did not require any modification (ά - α, έ - ε, ΐ - ι, .., etc). Function *GetSurroundingCharacters* retrieves the surrounding characters from a keyboard key(in close distance)(character η has surrounding characters υ,ϑ,ξ,ν,β,γ). Function *FindMostExpectedAdjacent* receives a list of adjacentCharacters, a validQuery that is used as a comparison measure to check if any adjacent character is an improvement, an expected index that represents the current index that is being looked into for correction, a max index that represents the maximum number of characters that can be checked to try to find a valid replacement from the expected index and finally a depth number which is the number of valid replacements to skip before replacing a character(more than one valid replacements can

exist,only some of them actually decrease the Edit Distance). When more than one surrounding characters can replace a single character, the character closest to the expected index is selected because, given the distance between a realistic word index and some distant indexes, there may be a valid replacement. However, if a character is six positions away from the current index, it could be an incorrect replacement(hence the max index). Finally there is the main function that takes a list of pairs that contain words that were judged as possible matches and their respective distance with the initial query that is the incorrect word that is also passed as an argument. The algorithm starts off by finding the minimum length between the initial query and the correct query and then pads the one that has lower length. The main loop is run from *0 to minlength* and for each character that the two strings do not match it removes accentuations, finds its surrounding characters and then finds among the surrounding characters which is the most expected adjacent character. If a character is not found the procedure *FindMostExpectedAdjacent* will return -1(it means that no replacement was found, go on to the next character). If a valid replacement was found, the replacement is made and the Edit Distance of the new updated query is calculated. If the *Edit Distance* decreases then the updated string is stored, the query, *Edit Distance* are updated and the expected letter index is increased to point to the next character in the stream. This goes on until i has reached the value of *minLength*. When the loop is completed it is checked if the *query* has remaining characters to be investigated for replacement. If this statement is true, a similar procedure is followed just like the one in the previous *for* loop. The only difference is that now all surrounding characters are being checked one by one instead of finding the most expected one with *FindMostExpectedAdjacent*. Finally the results are stored (*correctQuery, updatedInitialQuery, CurrentEditDistance*).Lastly, there is the function *ConvertToValidWordsAndEditDistance* that takes a list of possible valid words and a query and creates the pairs that are required from the function CorrectKeyboardMistype. It acts as a preprocessing functionality to make the flow of execution simpler. To convert the words to the appropriate form the function runs a loop for each valid word, it calculates the edit distance between the valid word and the query and then the valid word and the edit distance are stored as a pair in a list of pairs.

Algorithm 1 Keyboard Mistype Correction

```
1: procedure GetRowColumnAndUpper(letter)
2:     for row ← 0 to greekKeyboardLayout.length do
3:         for col ← 0 to greekKeyboardLayout[row].length do
4:             if greekKeyboardLayout[row][col] not NULL then
5:                 if greekKeyboardLayout[row][col][0] == letter then
6:                     upperCase ← False
7:                     return < row, col, upperCase >
8:                 else
9:                     if greekKeyboardLayout[row][col][1] == letter then
10:                        upperCase ← True
11:                        return < row, col, upperCase >
12:                    end if
13:                end if
14:            end if
15:        end for
16:    end for
17:    return Null
18: end procedure
```

```
1: procedure RemoveAccentuations(input)
2:     for all row in greekVowelAccentuation do
3:         for all letter in row do
4:             if input == letter then
5:                 return row[0]
6:             end if
7:         end for
8:     end for
9:     return input
10: end procedure
```

```
1: procedure GetSurroundingCharacters(incorrectCharacter,distance)
2:     surroundingCharacters ← newlist()
3:     < row, col, upperCase >← GetRowColumnAndUpper(RemoveAccentuations(incorrectCharacter))
4:     if row == −1 then
5:         return NULL
6:     end if
7:     surroundingCharacters ← Add all surrounding characters from <row,col,upperCase>
8:     return surroundingCharacters
9: end procedure
```

```
1: procedure FindMostExpectedAdjacent(adjacentCharacters, validQuery, expectedIn-
   dex, maxIndex,depth)
2:     mostExpectedIndex ← −1
3:     for all adjacent in adjacentCharacters do
4:         for i ← expectedIndex to maxIndex do
5:             if RemoveAccentuations(validQuery[i] == adjacent) then
6:                 if i < mostExpectedIndex||mostExpectedIndex == −1 then
7:                     if depth ≠ 0 then
8:                         depth ← depth − 1
9:                         continue
10:                    end if
11:                    mostExpectedIndex = i
12:                end if
13:            end if
14:        end for
15:    end for
16:    return mostExpectedIndex
17: end procedure
```

```
1: procedure CorrectKeyboardMistype(WordsAndEditDistance, InitialQuery)
2:     Input
3:         A list of pairs that have LEFT:words that may be correct(compared with InitialQuery) and
4:         RIGHT:their Edit Distance with InitialQuery. The InitialQuery that contains an incorrect character
5:     Output
6:          list of Triplets where LEFT:the correct word, MID: the corrected query after going through keyboard
7:         mistype correction(with aim the word on the LEFT side) and RIGHT: the number of keyboard
8:         changes that were applied on the InitialQuery characters to reduce Edit Distance

9:     results ← new list of Triplets
10:    for all pair in WordsAndEditDistance do
11:        result ← newTriplet()
12:        correctString ← ""
13:        validQuery ← pair.LEFT
14:        query ← InitialQuery
15:        currentQueryEditDistance ← pair.RIGHT
16:        if query.length < validQuery.length then
17:            minLength ← query.length
18:            paddedQuery ← PadString(query)
19:            paddedValidQuery ← validQuery
20:        else
21:            minLength ← validQuery.length
22:            paddedQuery ← query
23:            paddedValidQuery ← padString(validQuery)
24:        end if
25:        expectedLetterIndex ← 0
```

```
        for i ← 0 to minLength do
            if paddedQuery[i] ≠ paddedValidQuery[i] then
                CharWithoutAccentuation ← RemoveAccentuations(paddedQuery[i])
                surroundingCharacters ← GetSurroundingCharacters(CharWithoutAccentuation,
                                                                 1)
                for depth ← 0 to 3 do
                    tmpExpectedLetterIndex ← FindMostExpectedAdjacent(surroundingCharacters,
                                                                      paddedValidQuery,
                                                                      expectedLetterIndex,i,depth)
                    if tmpExpectedLetterIndex == −1 then
                        continue
                    else
                        expectedLetterIndex ← tmpExpectedLetterIndex
                    end if
                    adjacentCharacter ← paddedValidQuery[expectedLetterIndex]
                    updatedString ← replaceChar(query, adjacentCharacter, i)
                    updatedQueryEditDistance ← calculateEditDistance(validQuery, updatedString)
                    if (updatedQueryEditDistanceltcurrentQueryEditDistance) then
                        correctString ← updatedString
                        query ← updatedString
                        currentQueryEditDistance ← updatedQueryEditDistance
                        expectedLetterIndex ← expectedLetterIndex + 1
                    end if
                end for
            else
                expectedLetterIndex ← expectedLetterIndex + 1
                continue
            end if
        end for
        if validQuery.length < query.length then
            for i = minLength to query.length do
                if paddedQuery[i] ≠ paddedValidQuery[i] then
                    surroundingCharacters ← GetSurroundingCharacters(paddedQuery[i], 1)
                    for all adjacentCharacter in surroundingCharacters do
                        updatedString ← replaceChar(query, adjacentCharacter, i)
                        CharAtPVQExpInd ← paddedValidQuery[expectedLetterIndex]
                        removedAccentuationCharacter ← RemoveAccentuations(CharAtPVQExpInd)
                        if adjacentCharacter == removedAccentuationsCharacter then
                            CharAtPVQExpInd ← paddedValidQuery[expectedLetterIndex]
                            updatedString ← replaceChar(updatedString, CharAtPVQExpInd, i)
                        end if
                        updatedQueryEditDistance ← calculateEditDistance(validQuery, updatedString)
                        if updatedQueryEditDistance < currentQueryEditDistance then
                            correctString ← updatedString
                            CharAtPVQExpInd ← paddedValidQuery[expectedLetterIndex]
```

```
                    removedAccentuationChar ← RemoveAccentuations(CharAtPVQExpInd)
                    if adjacentCharacter == removedAccentuationChar then
                        query ← updatedString
                        currentQueryEditDistance ← updatedQueryEditDistance
                        expectedLetterIndex ← expectedLetterIndex + 1
                        break
                    end if
                end if
            end for
        else
            expectedLetterIndex ← expectedLetterIndex + 1
            continue
        end if
    end for
end if
if correctString.isEmpty() then
    correctStirng ← InitialQuery
end if
result.Left ← pair.LEFT
result.Mid ← correctString
result.Right ← pair.RIGHT − currentQueryEditDistance
results.add(result)
end for
return results
end procedure

procedure ConvertToValidWordsAndEditDistance(validWords, Query)
    pairsForEachWord ← newList()
    for all word in validWords do
        editDistance ← EditDistance.calculate(word, Query)
        pair ← newPair(word, editDistance)
        pairsForEachWord.add(pair)
    end for
    return pairsForEachWord
end procedure
```

The following tables show examples of what the functions do to reach a better understanding.

Table 8 shows how each letter is mapped to a keyboard position. Based on a keyboard that looks like the one in the image in Figure 5, the letter ɑ is on the 3rd row, 2nd column and is not an upper case(indexing starts from 0).

Table 9 shows the mapped characters that can have an accentuation and the character they result in after the accentuation has been removed.

Notice how some characters have a lower number of surrounding characters (Table 10). This is because some characters are close to the edges of the

Table 8: Results of
GetRowColumnAndUpper execution

Input	Output
α	(2,1,0)
ω	(3,4,0)
Ω	(3,4,1)
υ	(1,6,0)
ϑ	(1,7,0)

Table 9: Results of
RemoveAccentuations execution

Input	Output
ά	α
έ	ε
ή	η
ί, ϊ, ΐ	ι
ό	ο
ύ, ϋ, ΰ	υ
ώ	ω

Table 10: Results of
GetSurroundingCharacters execution

Input	Output
α	·, ς, σ, ζ
ω	φ, γ, β, ψ
π	0, -, [, ′, λ, ο
η	υ, ϑ, ξ, ν, β, γ

keyboard and have fewer surrounding characters.

The value returned in Table 11 corresponds to the expected index for the valid query, which has to be substituted by the initialQuery's wrong character at the expected index. To be specific, in word μήνυμα, there is expected index 3 and value returned 3. Meaning, that in valid query index 3 is the character υ and in initialQuery it is the character η which is incorrect. By replacing initialQuery[3]:η with validQuery[3]:υ the edit distance will be decreased to 0 from 1.

Table 12 shows some examples of the CorrectKeyboardMistype execution. On row 2 of the table it is shown how a word that would probably never be appointed as a valid word(due to Edit Distance 4) can be a valid recommendation after executing the function that takes into account the mistypes. The word

Table 11: Results of
FindMostExpectedAdjacent execution

Input	Output
(αοτημα,αίτημα,1,1)	1
(αυτημα,αίτημα,1,1)	1
(μήνημα,μήνυμα,3,3)	3

ισχύ has Edit Distance 2 and is way closer than βάζω that has Edit Distance 4. But the moment, keyboard shifting/mistype is taken into account,word βάζω becomes a valid recommendation.

Table 12: Results of *CorrectKeyboardMistype* execution

Input	Output
([(βάζω,1),(βάζο,1)],βαζχ)	[(βάζω,βάζω,1),(βάζο,βαζχ,0)]
([(βάζω,4),(ισχύ,2)],νσχβ)	[(βάζω,βάζω,4),(ισχύ,ισχύ,2)]
([(αίτημα,2),(έτοιμα,4)],αουημα)	[(αίτημα,αίτημα,2),(έτοιμα,αουημα,0)]

5.2.1 Execution Process

The execution requires the correct data to call the function *CorrectKeyboardMistype* that was presented in the previous section. As mentioned, the parameters are a query that is mistyped and a list of pairs of possible correct words and their edit distance with the query. The steps to reach that format and execute the algorithm are(it is taken as granted that the query and a set of possibly valid words(a vocabulary) exist):

1. Load the Vocabulary
2. Convert the vocabulary to a List of Strings
3. Take a query as input
4. Call the function *ConvertToValidWordsAndEditDistance* with parameters the vocabulary(as List) and the query
   (a) Create a list of pairs(String,Integer)
   (b) for each valid word calculate the edit distance between the valid word and the query
   (c) store it as a pair(String,Integer) and add it in the list of pairs mentioned in step (a)
   (d) Return the list of pairs

5. After the execution of step 4, all the required parameters will be available to execute the final function *CorrectKeyboardMistype* with parameters the list from step 4 and the query from step 3.

   (a) Briefly explained, for each pair

   (b) find minimum length between valid word and query

   (c) pad the side that has a lower number of characters(to avoid out of bounds during comparisons)

   (d) start a loop that will iterate for minimum length times(minLength from step (b))

   (e) compare the characters(between valid word and query) at each index and if they are not identical then

   (f) Remove Accentuations and find the surrounding characters of the character that did not match with the valid word

   (g) Find if any of the surrounding characters is a valid replacement to reduce the edit distance(through *FindMostExpectedAdjacent*)

   - The function *FindMostExpectedAdjacent* finds which character is the best replacement when more than one characters can replace the character that did not match.An example of the importance of this is the word μοοούνυμα. We can see that there are 3 extra characters added.When the index of the word reaches the first ύ how will it know based on the index if υ is for the νυ or for the μή? This function takes that into consideration and checks left to right if the υ or η would be a better replacement based on precedence and current number of correct characters that were found(which is the variable expectedLetterIndex that is passed as an argument and it is increased at every replacement or at every comparison that validWord.character == query.character at index i)

   (h) if a possible valid replacement is found,do the replacement and check if the edit distance is decreased. If the edit distance is decreased, updated the query to the new one and move on to the next character

   (i) After the completion of the for loop for up to minLength, check if there are remaining characters that can be checked(if query length greater than valid word length(due to extra characters)). If there are extra characters then

   (j) Iterate over these characters and check for each surrounding character if any replacement reduces the edit distance of the query.

   (k) If any of the surrounding characters reduces the edit distance then update the query

   (l) finally,store the valid word,the updated query and the updated edit distance and return the results.

6. The results are returned as Triplets with (ValidWord,UpdatedQuery, UpdatedEditDistance) as mentioned in step 5.(l). These results can be used to find out if the result matches any of the correct words and recommend it to the user(if edit distance is 0 then it means they are the same(which is the third parameter))

5.3 Evaluation

5.3.1 Evaluation Cases

There are 4 main cases that could be checked:

- Character Addition
- Character Deletion
- Character Substitution
- Character Transposition

The first two and the last bullet points won't be helpful because this algorithm relies on character substitution. Because the algorithm won't produce any useful results in certain cases, they won't be a significant factor in the evaluation. However, some examples of how the algorithm manages some of these situations when paired with character substitution will be shown.

5.3.2 Evaluation Metrics

The evaluation metrics are the ones mentioned in section 4.2 Evaluation Metrics with only difference the exclusion of Fall-Out.

5.3.3 Evaluation Results

Specific samples

Table 13 demonstrates how the algorithm swaps out incorrect characters with the correct ones. The significance of these results comes from the instances where the final word is affected by a mistyped character. Take note of how the words ψηλός and ψιλός have the exact same sound yet indicate something entirely different. Imagine that the second letter is the sole modification to the original utilized. Due to the fact that both words have an Edit Distance of 1(changing ψυλός/ψολός to ψηλός/ψιλός requires 1 change), the search engine would therefore be unsure on which to suggest. The algorithm that was created is suitable for these situations since, upon application, it demonstrates the ability to replace each individual missing letter that modifies the meaning of the word. Another case that is worth discussing are the words σήκω and σύκο. The letters υ and η are adjacent and that causes σήκω to have Edit Distance 2 after 2 substitutions, on the other hand the word σύκο is helped from the substitution

since ύ is a correct substitution which causes it to have 1 Edit Distance after 2 substitutions. Anyway, after executing the algorithm, the word σήκω that has greater Edit Distance becomes more relevant than σύκο.

Table 13: Results of Algorithm execution with simple substitutions

Correct Word	Initial Input	Modified Input	Old Edit Distance	New Edit Distance
ψηλός	ψυλός	ψηλός	1	0
ψιλός	ψυλός	ψυλός	1	1
ψηλός	ψολός	ψιλός	1	1
ψιλός	ψολός	ψιλός	1	0
ίσος	ίσπς	ίσος	1	0
ίσως	ίσπς	ίσπς	1	1
ίσος	ίσβς	ίσβς	1	1
ίσως	ίσβς	ίσως	1	0
μίλα	μθλα	μίλα	1	0
μήλα	μθλα	μθλα	1	1
μίλα	μγλα	μγλα	1	1
μήλα	μγλα	μήλα	1	0
σήκω	σύκβ	σήκω	2	0
σύκο	σύκβ	σύκβ	1	1
σήκω	στκο	στκο	1	1
σύκο	στκο	σύκο	1	0

Table 14 is a collection of the combined cases that were mentioned earlier. The first two cases of the word ακόλλητος consist of character addition and character substitution. In both cases the algorithm manages to find mistypes that can be replaced to decrease the Edit Distance. Following, there is an example with character deletion and a character substitution that the algorithm fails to recognize. Generally, the algorithm does not work well with cases that contain character deletion.

Then, there are four more cases that demonstrate how four seemingly random characters can, in fact, form a valid word that has been shifted to the right (due to unfamiliarity with the keyboard or a bad hand placement). Observe how, with an Edit Distance 2 the letters νσχβ are closer to the word ισχύ than to the words βάζο and βάζω with Edit Distance 4. These four random letters affect the significance of the words once the algorithm has been performed. Now the words βάζο and βάζω have an Edit Distance 1 and 0 respectively, while the word ισχύ remains at 2. Another thing that is worth observing, is how this algorithm can play an important part in the correct word recommendation. Both words βάζο and βάζω have an Edit Distance 4,with only one letter different and yet a complete different meaning. This algorithm takes that into consideration and based on the Table 14 results, it is shown that on rows 4 and 6 it would manage to recommend the correct word to the user.

Lastly there are two cases with initial input απλότημα and potentially cor-

rect words αίτημα and απλότητα. In these situations, the algorithm succeeds to identify and fix one of the substitutions, lowering the Edit Distance from 3 to 2, while the other word fails to identify any neighboring mistypes, resulting in no change and a remaining Edit Distance of 1. It is very obvious that the term approaches closer to απλότητα instead of αίτημα in this situation, showing that modifications do not always result in something.

Table 14: Results of Algorithm execution with combined actions and interesting cases

Correct Word	Initial Input	Modified Input	Old Edit Distance	New Edit Distance
ακόλλητος	αοοοόλλητος	ακοοόλλητος	3	2
ακόλλητος	αοοοόλοηυος	ακοοόλλητος	5	2
ακόλλητος	ακλλυτος	ακλλυτος	2	2
βάζο	νσχπ	βάζο	4	0
βάζο	νσχβ	βάζβ	4	1
βάζω	νσχβ	βάζω	4	0
ισχύ	νσχβ	νσχβ	2	2
αίτημα	απλότημα	απλίτημα	3	2
απλότητα	απλότημα	απλότημα	1	1

Bulk Evaluation with Datasets from section 3.2 - DKMC

First, it's important to remember that the algorithm should produce excellent results for datasets that contain substitutions, results of lower quality for datasets that combine addition and deletion with substitution, and results of even lower quality, if any, for datasets that contain no substitutions at all. Based on the expected difficulty to retrieve any results when trying to combine substitutions with addition and deletion, a solution is to use Edit Distance after executing the algorithm to include some cases where addition or deletion is present.

Because of this, we present a simple and direct variant called *Keyboard Mistype Correction And Edit Distance*, which merely runs the algorithm of Keyboard Mistype Correction and then calculates the Edit Distance between the final output and the supplied valid word. If the value returned is less than a value X (that is predetermined), the word is marked as relevant and is retrieved.

Finally, the results that will be presented in this section are for the Algorithm *KeyboardMistypeCorrection* and its variations *KeyboardMistypeCorrectionAndEditDistance1* and *KeyboardMistypeCorrectionAndEditDistance2* (Edit Distance <=1 and 2 after execution).

For simplicity, the dataset names will be reduced to:

- OnePerWordSubstitutions: 1PWS
- TwoPerWordSubstitutions: 2PWS
- ThreePerWordSubstitutions: 3PWS

- FourPerWordSubstitutions: 4PWS
- OneSubstitutionAndAddition: 1SnA
- TwoSubstitutionsAndAddition: 2SnA
- TwoSubstitutionsAndTwoAdditions: 2Sn2A
- TwoSubstitutionsAndThreeAdditions: 2Sn3A
- OneSubstitutionAndDeletion: 1SnD
- TwoSubstitutionsAndDeletion: 2SnD

and the Algorithm names will be reduced to:

- KeyboardMistypeCorrection: KMC
- KeyboardMistypeCorrectionAndEditDistance1: KMCnED1
- KeyboardMistypeCorrectionAndEditDistance2: KMCnED2

Figure 6 illustrates that *KeyboardMistypeCorrection*, compared to *KeyboardMistypeCorrectionAndEditDistance1* and *KeyboardMistypeCorrectionAndEditDistance2*, has the highest Precision. This is because *KeyboardMistypeCorrection* only returns exact matches, while the other 2 return exact matches plus words that fall within an Edit Distance range(1 and 2)(the denominator is increased and so the Precision falls). It should be noted that the *KeyboardMistypeCorrection* Precision values range from 0.896 to 0.97, with the 1PWS having the greatest value and the 2SnD having the lowest.

Following, figure 7 shows that *KeyboardMistypeCorrectionAndEditDistance2*, compared to *KeyboardMistypeCorrectionAndEditDistance1* and *KeyboardMistypeCorrection*, has the highest Recall. This is the expected result since Edit Distance increases the number of entries that are retrieved, meaning that more documents are retrieved and therefore more relevant ones come along(therefore the low Precision values). In the datasets where only substitution is applied all 3 algorithms achieved about 1.0 Recall. The impact is shown in the combined cases where the edit distance plays a big part. When addition and deletion is tested, *KeyboardMistypeCorrection* and *KeyboardMistypeCorrectionAndEditDistance1* Recall levels fall sharply(to ranges 0.143 - 0.167). Important to observe how none of the algorithms fail to retrieve relevant documents in the dataset *2Sn3A* and this because the edit distance is greater than 2 so none of the algorithms can catch this case(the test is also a bit extreme).

Lastly, figure 8 shows that *KeyboardMistypeCorrection* achieved the highest F-Measure scores when the tests contain only substitution(up to four substitutions) with values 0.985, 0.968, 0.959, 0.956 respectively for each dataset as shown in the figure. In the datasets where substitution is combined with addition or deletion of a character the algorithm *KeyboardMistypeCorrection* fails to identify the word(since it works with exact match) and the values fall

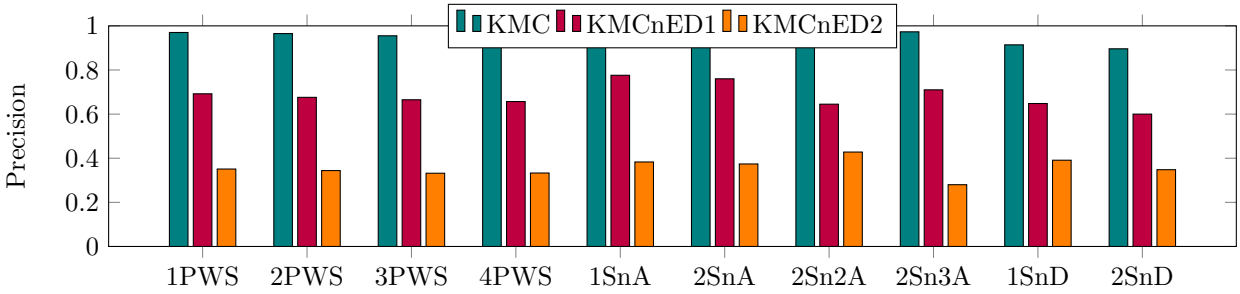

Figure 6: Precision of the algorithms on the datasets mentioned above

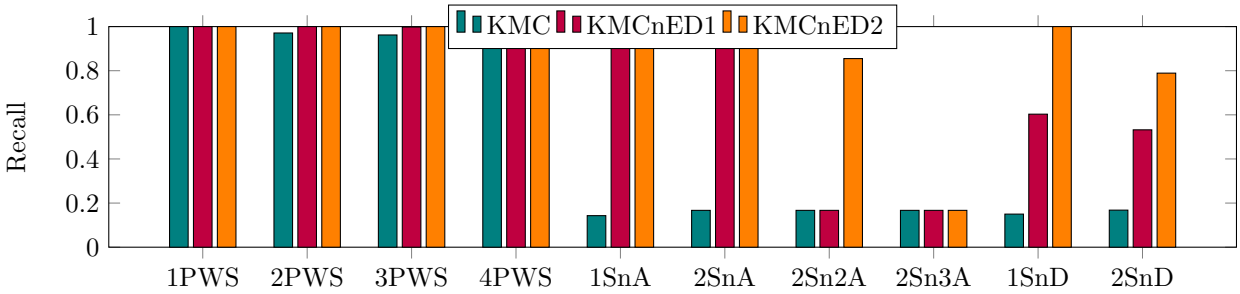

Figure 7: Recall of the algorithms on the datasets mentioned above

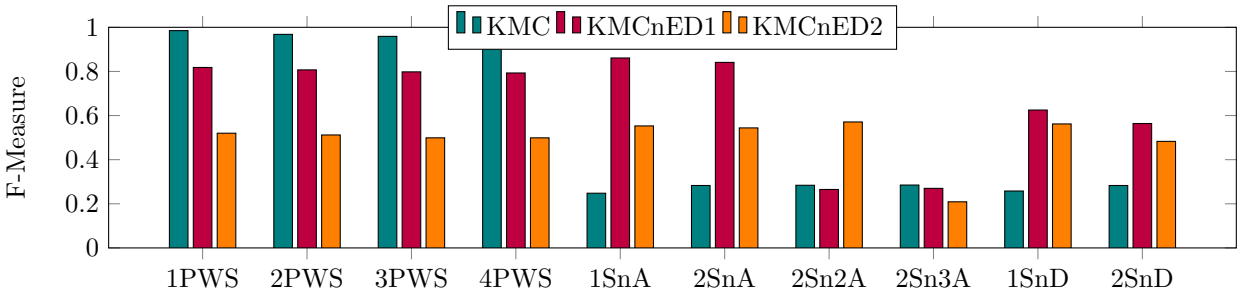

Figure 8: F-Measure of the algorithms on the datasets mentioned above

to a range of 0.143 - 0.167. On the other hand, *KeyboardMistypeCorrectionAndEditDistance1* produces good results but not the best when the tests contain only substitutions with values 0.818, 0.807, 0.798, 0.79 respectively for each dataset. In the datasets that combine additions and deletion it produces the best results compared to both *KeyboardMistypeCorrection* and *KeyboardMistypeCorrectionAndEditDistance2*.In *1SnA*, *2SnA*, *1SnD*, *2SnD*, it achieves F-Measure of 0.861,0.841,0.625, 0.564 respectively, whereas the other two are to a range of 0.143 - 0.167 for *KeyboardMistypeCorrection* and 0.5 - 0.55 for *KeyboardMistypeCorrectionAndEditDistance2*. *KeyboardMistypeCorrectionAndEd-*

itDistance2 achieves the values in the range 0.48 - 0.57 in 9/10 datasets and it is the one that reaches the lowest F-Measure value in the dataset *2Sn3A*(0.21) and the highest of all in the *2Sn2A*(0.57).

5.3.4 Comparison between several algorithms and their variations

The first two tables(15 and 16) are to present the numeric results for Precision and Recall for those who are interested. The main comparison will be upon the F-Measure results that were achieved.

For simplicity, the dataset names will be reduced to:

- OnePerWordSubstitutions: 1PWS
- TwoPerWordSubstitutions: 2PWS
- ThreePerWordSubstitutions: 3PWS
- FourPerWordSubstitutions: 4PWS
- OneSubstitutionAndAddition: 1SnA
- TwoSubstitutionsAndAddition: 2SnA
- TwoSubstitutionsAndTwoAdditions: 2Sn2A
- TwoSubstitutionsAndThreeAdditions: 2Sn3A
- OneSubstitutionAndDeletion: 1SnD
- TwoSubstitutionsAndDeletion: 2SnD

and the Algorithm names will be reduced to:

- KeyboardMistypeCorrection: KMC
- KeyboardMistypeCorrectionAndEditDistance1: KMCnED1
- KeyboardMistypeCorrectionAndEditDistance2: KMCnED2
- SoundexGR will remain as is
- SoundexGR$_{comp}$ will remain as is
- Stemmer will remain as is
- Phonemic will remain as is
- Edit Distance 1-4: ED1, ED2, ED3 and ED4

Table 15: Precision of different Algorithms

	1PWS	2PWS	3PWS	4PWS	1SnA	2SnA	2Sn2A	2Sn3A	1SnD	2SnD
KMC	0.97	0.97	0.96	0.95	0.94	0.94	0.97	0.97	0.91	0.90
KMCnED1	0.69	0.68	0.67	0.66	0.78	0.76	0.65	0.71	0.65	0.60
KMCnED2	0.35	0.34	0.33	0.33	0.38	0.37	0.43	0.28	0.39	0.35
SoundexGR	0.82	0.83	0.83	0.83	0.81	0.82	0.84	0.83	0.79	0.83
SoundexGR$_{comp}$	0.69	0.67	0.66	0.66	0.67	0.67	0.71	0.70	0.62	0.64
Stemmer	0.91	0.93	0.93	0.93	0.94	0.94	0.94	0.94	0.91	0.93
Phonemic	0.98	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
ED 1	0.95	0.91	0.91	0.91	0.90	0.91	0.90	0.90	0.87	0.90
ED 2	0.69	0.75	0.64	0.64	0.82	0.66	0.63	0.63	0.73	0.63
ED 3	0.38	0.42	0.45	0.45	0.48	0.51	0.37	0.33	0.40	0.42
ED 4	0.17	0.19	0.21	0.21	0.22	0.23	0.29	0.16	0.18	0.19

Table 16: Recall of different Algorithms

	1PWS	2PWS	3PWS	4PWS	1SnA	2SnA	2Sn2A	2Sn3A	1SnD	2SnD
KMC	**1.0**	0.97	0.96	0.96	0.143	0.17	0.17	0.17	0.15	0.17
KMCnED1	**1.0**	**1.0**	**1.0**	**1.0**	0.97	0.94	0.17	0.17	0.60	0.53
KMCnED2	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.86	0.17	1.0	0.79
SoundexGR	0.39	0.29	0.22	0.22	0.25	0.22	0.19	0.19	0.24	0.22
SoundexGR$_{comp}$	0.49	0.35	0.25	0.24	0.30	0.25	0.20	0.19	0.32	0.27
Stemmer	0.16	0.18	0.20	0.20	0.14	0.17	0.17	0.17	0.16	0.18
Phonemic	0.16	0.18	0.20	0.20	0.14	0.17	0.17	0.17	0.15	0.17
ED 1	**1.0**	0.23	0.24	0.24	0.15	0.18	0.17	0.17	0.16	0.20
ED 2	**1.0**	**1.0**	0.29	0.29	**1.0**	0.27	0.18	0.17	**1.0**	0.35
ED 3	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.28	0.18	**1.0**	**1.0**
ED 4	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.30	**1.0**	**1.0**

The F-Measure findings for each method and its variants on each individual dataset are shown in Table 17. It's crucial to note how the *KMC* algorithm correctly retrieves words when the mistyped word just involves substitutions. It achieves F-Measures of 0.99, 0.97, 0.96, and 0.96 in datasets *1PWS* through *4PWS*. It is noteworthy to say that the reason *KMC* achieves higher F-Measure score in *1PWS* compared to *ED1* is because in cases like βαζψ,βαζπ it will retrieve the words βάζω and βάζο(respectively) whereas *ED1* will retrieve both βάζο and βάζω in both cases(therefore *KMC* achieves higher score). As for the algorithms *SoundexGR*, *Stemmer* and *Phonemic*, the results are as expected, since they depend on characters that appear in the word and substitutions work as a character modification and not as a typo due to lack of knowledge(e.g. μινιμα, μηνημα, etc..) that retains the way it is pronounced.

Moving on to the *1SnA* dataset, ED2 achieves the best result with F-Measure score of 0.90 while KMCnED1 achieves 0.86. All the other algorithms achieve

Table 17: F-Measure of different Algorithms

	1PWS	2PWS	3PWS	4PWS	1SnA	2SnA	2Sn2A	2Sn3A	1SnD	2SnD
KMC	**0.99**	**0.97**	**0.96**	**0.96**	0.25	0.28	0.28	0.29	0.26	0.28
KMCnED1	0.82	0.81	0.80	0.79	**0.86**	**0.84**	0.27	0.27	0.63	**0.56**
KMCnED2	0.52	0.51	0.50	0.50	0.55	0.54	**0.57**	0.21	0.56	0.48
SoundexGR	0.53	0.44	0.35	0.34	0.38	0.35	0.31	**0.30**	0.37	0.35
SoundexGR$_{comp}$	0.58	0.46	0.36	0.35	0.41	0.37	0.31	**0.30**	0.42	0.37
Stemmer	0.27	0.30	0.33	0.34	0.25	0.29	0.28	0.28	0.27	0.30
Phonemic	0.28	0.30	0.34	0.34	0.25	0.29	0.29	0.29	0.26	0.29
ED 1	**0.98**	0.36	0.37	0.37	0.26	0.30	0.28	0.28	0.27	0.32
ED 2	0.82	0.86	0.40	0.39	**0.90**	0.38	0.28	0.26	**0.84**	0.45
ED 3	0.55	0.59	0.62	0.62	0.65	0.68	0.32	0.23	0.57	**0.60**
ED 4	0.29	0.32	0.34	0.34	0.36	0.37	0.44	0.21	0.30	0.32

scores that are lower than 0.65. As mentioned before,when character additions and deletions are taken into account, the quality of the algorithm deteriorates since it fails to retrieve the correct words. For this reason KMCnED1 was proposed and it actually increases F-Measure from 0.25 to 0.86 and thats because it handles the case of 1 character addition or deletion(supposedly).

For dataset *2SnA* the algorithm *KMCnEd1* has a greater impact with F-Measure value of 0.84 compared to the other algorithms that range around 0.2 - 0.4 with the exception of *ED3* and *KMCnED2* that have values 0.68 and 0.54. In this case too the algorithm manages to raise F-Measure of *KMC* from 0.28 to 0.84 which is a great difference.

For dataset *2Sn2A* the algorithm *KMCnED2* manages to achieve F-Measure score of 0.57 which is low but compared to the other ones that range to 0.25 - 0.44 is an indication of some success.

As for the complex dataset *2Sn3A*, it is simply shown that *KMCnED1* achieved good scores with addition only because *Edit Distance* was used and after a point it fails to identify the correct words(due to the increased number of additions). Also, the highest F-Measure score is achieved from *SoundexGR*, *SoundexGR*$_{comp}$ with value of 0.30, but this value only shows that none of the algorithms managed to find the correct words in this dataset.

Finally, there are 2 datasets *1SnD* and *2SnD* that contain substitutions and deletion and the best results are given by edit distance 2 with value of 0.84 for *1SnD*, 0.60 for *2SnD* and *KMCnED1* with a value of 0.56 for *2SnD*. Other than the value of 0.84 in *1SnD*, it is shown that the algorithms failed to handle deletion and substitution as none produced great results.

5.4 Discussion on parameter difficulties

Although highly useful, the method may also be taxing. It needs a collection of *correct* words that will serve as the foundation for the algorithm comparisons

(due to dependency on the Edit Distance Algorithm). So what can be done to retrieve these words?

5.4.1 Through Edit Distance and Tries

Combining Edit Distance and Tries could create a set of words that may be valid. The idea is:

1. Each character on the keyboard has some surroundings. For each character in the input,find all the surrounding characters.
2. Remove combinations of letters that are redundant/do not exist(combinations like ξν, ξγ, γξ, ξ,κ σξ, σς, σκτ, νσχ,...)(optimization action)
3. Create all combinations containing one of the possible letters in each position(7^{length} possible combinations(worst case))
4. After doing steps 1 to 3 for the first 2 to 5 characters(first k characters based on the length of the word), select all words from the vocabulary that start with these characters and have the same length as the initial input
5. Filter the words that remain from step 4 by comparing if remaining characters match any of the surrounding characters that were found from step 1.

To reach a better understanding an example is presented below. This will be done for the input νσχβ that has correct input βάζω and close words ισχύ and βάζο. In table 18 each input character and its respective surrounding characters(Step 1) is presented.

Table 18: Step 1 execution,surrounding characters(plus initial character) from initial input

ν	σ	χ	β
ν, η, ξ, μ, β	σ, ς, ε, δ, χ, ζ, α	χ, σ, δ, ψ, ζ	β, γ, η, ν, ω

The combinations νς, ηα, ξσ, ξς, ξδ, ξχ, ξζ, μχ, μζ, βσ, βς, βχ are from the surrounding characters of the first two letters νσ(as shown in table 18) and have no words that contain these two at ANY position(based on e-dictionary Dictionary of Standart Modern Greek). By applying Step 2 on the combinations, the combinations are reduced to:

Notice how beneficial this optimization is, it manages to decrease the total combinations by 35%.

After the optimization, the combinations are created(Step 3) - νσ , νε, νδ, νχ, νζ, να, ησ, ης,...etc. As for Step 4, due to the length of the word only the first 2 characters were chosen to search in the vocabulary. Using the first two characters and the rule of same length, about 15 words are returned(νάζι,

Table 19: Step 2 execution,reduction from 35 possible combinations to 23

ν	η	ξ	μ	β
σ, ε, δ, χ, ζ, α	σ, ς, ε, δ, χ, ζ	ε, α	σ, ς, ε, δ, α	ε, δ, ζ, α

ναζί, ήχος, μέσα, μέση, μέσο, μάχη, μάσα, μαδώ, μάζα, μαζί, βάβα, βαβά, βάζο, βάζω). Step 5 filters out the words that do not contain on 3rd position any of the characters χ, σ, δ, ψ, ζ and on 4th position the characters β, γ, η, ν, ω.After the filtering is done,the remaining words are 4: μέση, μάχη, μαδώ, βάζω. These 4 words are the ones that will be used as the *correct* ones. These 4 words are exact matches and do not handle cases like βάζο that we presented in table 14. This can be captured by using Edit Distance to increase the range of the words that can be used as *correct* ones.

5.4.2 Through Length

Because the algorithm above tends to look like another approach to keyboard mistypes correction, we could simplify the way words are chosen by selecting words that are of the same length and in case we want to handle possible additions/deletions to also retrieve words that are of length plus or minus 1 of the initial input.

6 Conclusion

We created a dataset and compared 3 search engines(Elasticsearch, Solr, Algolia) on their ability to query data and we introduced a keyboard mistype correction algorithm for the Greek language by creating a QWERTY keyboard layout and mapping all surrounding buttons. Specifically, we compared their Precision, Recall, F-Measure/F-Score and Fallout values on 4 different cases 1)Remove stopwords and apply stemming, 2)Remove stopwords and do not apply stemming, 3) Do not remove stopwords and apply stemming and 4)Do not remove stopwords and do not apply stemming. Based on the results we concluded that Elasticsearch performed better overall for the dataset with a F-Measure value of 0.8 with filter Remove Stopwords and do not apply stemming, following Algolia with 0.78 with filter Do not remove stopwords and do not apply stemming and then Solr with 0.77 with filter Remove stopwords and apply stemming(should not forget that trade-offs between stemming and stopword removal matter when preprocessing and querying which means that these results present the best performance for each search engine among all filters only at value point). Following the search engines evaluation we presented the algorithm for the Keyboard Mistype Correction which we concluded that it has the ability to recommend words that contain only substitutions very efficiently with a F-Measure score of 0.99, 0.97, 0.96, 0.96 for datasets that contain 1 to

4 substitutions(respectively). Important to note that the Keyboard Mistype Correction algorithm is capable of recommending the correct consonant words more efficiently because the mistypes are usually adjacent characters that can be distinguished when checking thy keyboard layout(instead of retrieving all words that have Edit Distance 1 which decreases the Precision as observed in Table 15). Then, datasets that combine substitution with character addition were presented and because the algorithm does not do well with character additions and deletions, *KeyboardMistypeCorrectionAndEditDistance1* and *KeyboardMistypeCorrectionAndEditDistance2* were presented which supposedly retrieved words that had Edit Distance of 1 or 2 after the execution of Keyboard Mistype Correction. *KeyboardMistypeCorrectionAndEditDistance1* achieved F-Measure scores of 0.86 and 0.84 to datasets that contained 1-2 substitutions and 1-2 character additions. As for datasets that contained 3 additions or character deletions none of the algorithms achieved great scores.

References

Antrei Kavros and Yannis Tzitzikas. 2022. SoundexGR: An Algorithm for Phonetic Matching for the Greek Language.

Jesús Vilares, Miguel A. Alonso, Yerai Doval, Manuel Vilares. 2016. Studying the Effect and Treatment of Misspelled Queries in Cross-Language Information Retrieval, pp 3-10.

Wikipedia. List of QWERTY keyboard language variants

Alfavita Newsroom.Ομόηχες λέξεις: Ακούγονται το ίδιο, γράφονται διαφορετικά, μπερδεύουν συχνά

Guilherme Torresan Bazzo, Gustavo Acauan Lorentz, Danny Suarez Vargas and Viviane P. Moreira. 2020. Assessing the Impact of OCR Errors in Information Retrieval.In Springer Link,part of the Lecture Notes in Computer Science book series (LNISA,volume 12036)

Jesus Vilares, Manuel Vilares, Juan Otero. 2011. Managing Misspelled Queries in IR Applications

Gregdevogo. 2020. Autocorrect in Google, Amazon and Pinterest and how to write your own one. In TowardsDataScience.

Wolf Garbe. 2021.What are some algorithms of spelling correction that are used by search engines? For example, when I used Google to search "Google imeges", it prompted me, "Did you mean: Google images?". In quora.

Huizhong Duan, Bo-June Hsu (Microsoft). Online Spelling Correction for Query Completion

Yanen Li.. 2011. CloudSpeller: Spelling Correction for Search Queries by Using a Unified Hidden Markov Model with Web-scale Resources

Yasser Ganjisaffar.. 2011. qSpell: Spelling Correction of Web Search Queries using Ranking Models and Iterative Correction

Yoh Okuno (Yahoo). Spelling Generation based on Edit Distance

Yoh Okuno. 2011. Spelling Alteration for Web Search Workshop

Casey Whitelaw (Google). Using the Web for Language Independent Spellchecking and Autocorrection

Jo Adetunji. 2014.A dozen ways to avoid that $ 617bn 'fat finger' moment