PSS:

A Phonetic Search System for Short Text Documents

by

Jerry Jiaer Zhang

B.Sc., Simon Fraser University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August, 2008

Abstract

Finding the right information from the increasing amount of data on the Internet is not easy. This is why most people use search engines because they make searching less difficult with a a variety of techniques. In this thesis, we address one of them called phonetic matching. The idea is to look for documents in a document set based on not only the spellings but their pronunciations as well. It is useful when a query contains spelling mistakes or a correctly spelled one does not return enough results. In these cases, phonetic matching can fix or tune up the original query by replacing some or all query words with the new ones that are phonetically similar, and hopefully achieve more hits. We propose the design of such a search system for short text documents. It allows for single- and multiple-word queries to be matched to sound-like words or phrases contained in a document set and sort the results in terms of their relevance to the original queries. Our design differs from many existing systems in that, instead of relying heavily on a set of extensive prior user query logs, our system makes search decisions mostly based on a relatively small dictionary consisting of organized metadata. Our goal is to make it suitable for start-up document sets to have the comparable phonetic search ability as those of bigger databases, without having to wait till enough historical user queries are accumulated.

Table of Contents

List of Tables

List of Figures

Acknowledgements

My deepest gratitude goes first and foremost to my supervisor, Dr. Son T. Vuong, for his constant encouragement and guidance. He has walked me through the very stages of the writing of this thesis. Without his consistent and illuminating instruction, this thesis could not have reached its present form. It is thanks to his continuous support that I could pursue whatever topic I found appealing. And it is thanks to his insightful suggestions and advice that made this thesis, hopefully, interesting.

Second, I would like to express my heartfelt gratitude to Dr. Eric Wohlstadter for his time and effort to help improve my thesis. Especially given his busy schedule and my short notice, his help is doubly appreciated.

Last my thanks would go to my beloved family for their loving considerations and support all through these years. I also owe my sincere gratitude to my friends Beier Cai and Karen Jiang for their help during the difficult course of my thesis work. Also thank Wei Li for his suggestions and tips to put this thesis together. A special thanks to Yvonne Chen, who has always inspired me and had great confidence in me over the past ten years. You are my poetry and light.

Dedication

To my parents.

Chapter 1

Introduction

1.1 Motivation

With the increasing amount of information available on the Internet, quickly finding what one needs among the overwhelming data is not easy. Search engines are being constantly improved in many ways to better serve this goal, and different venders have adopted various techniques to tweak their products towards the kind of search hits they see right. This is why the major general search engines sometimes behave very differently and return better results than others for certain queries. Despite of all the differences, however, they do share a lot of features in common. Google's " "Did you mean" is one example. It is the "We have included" function in Yahoo and "Were you looking for" in MSN. It addresses the spelling mistakes users often make by catching the most obvious errors and suggesting the correct spellings automatically. In fact, this feature has become so popular that it is now almost the standard function all the big players in the search engine field have to offer. On the other hand, as much as this feature is highly demanded, smaller websites do not seem to have it yet because of the limitations imposed by the current methodology - almost all web applications implemented their version of "Did you mean" based on the

query log approach[11], which is essentially a statistic model built upon an extensive set of prior user queries that smaller websites are not usually able to accumulate. In this thesis, we aim to address this gap through the introduction of PPS - a search system relying on a relatively small, self-contained dictionary to offer the similar feature with phonetic matching for short text documents.

1.2 Goal

The goal of this thesis is to propose the design of a phonetic search system for short text documents that only requires a relatively small data set to function correctly. Instead of counting on a set of extensive prior user queries, we focus on the correlations among different words, as well as the relationships between words and the documents containing them to create a dictionary to store such information for phonetic searches for single- and multiple-, correctly spelled and misspelled words. Last but not the least, we run various types of queries against a test data set of restaurant names to examine the accuracy and efficiency of the system.

1.3 Thesis Contribution

The main contribution of this thesis is to allow for multi-word sound-based search to increase result accuracy if there are no exact matches. To achieve this, we:

- Introduce the system that makes novel use of several widely adapted al-

gorithms from different fields and incorporate them to form the search framework.

- Provide a high level design specifying the implementation of this system that is platform-independent.

- Offer an alternative approach to provide word or phrase correction that is not based on extensive query statistics, which many main stream search engines heavily rely on. Our system differs from them in that decisions are made mostly based on well-organized metadata and thus result in better accuracy for small document sets. On the other hand, it is still possible to allow additional filters, such as query statistics, to be incorporated into our system because of its staged search concerns.

Chapter 2

System Design

2.1 Dictionary Creation and Maintenance

All searching mechanisms eventually come down to comparing queries with the database. In one way, if we consider a query itself as a document that is typically shorter than those stored in a database, search is really just pulling documents out of a data collection in terms of their similarities to the original query. Even without taking accuracy or other factors into account, a brute-force, word-by-word lookup procedure in each document is obviously undesirable due to its inefficiency. Therefore, we need to organize the data in a way that allows fast access and easy creation and maintenance. We call it a dictionary because like what a real dictionary has to offer, our data structure for storing documents enables non-linear lookups. In addition, it builds meta-data that describes each document's properties for multiple word sound-based search.

2.1.1 Text Processing

A document consists of words, symbols and Arabic numerals. It is usually not difficult to break text into words. For this system, we identify words with regular expressions to match predefined regex patterns to text to extract a

list of unique words that contain only letters, digits, apostrophes, and special symbols like @, ., /, etc. This process of identifying words in a document is sometimes called tokenization.

2.1.2 Dictionary Creation

The tokenized text of a document can then be used to create the dictionary. A dictionary is a data structure that carries not only the original document text but also additional information that describes its properties. The following sections discuss the creation of these properties that are stored together with the original documents they are derived from as metadata.

Word List

Word List is a list of distinct words that appear in the document. It is directly from the list of words generated during the text processing phase. Additionally, it is sorted. Sorting is relatively expensive but there are two reasons we decided to do it. First, documents are likely to be static once they are created and stored in the database, so the number of sortings that needs to be performed is only linear - $O(n)$. Second, the time and space complexity of creating a dictionary does not add to the run time of searching, so it is preferable to organize the data in a way that facilitates search performance. By placing words in alphabetical order, we can implement binary search for word matching to reduce the time complexity to $O(lgn)$.

tf-idf Weight

tf-idf Weight is a statistical measure to evaluate the importance of a word to a document in a set of documents [13]. It is obtained by multiplying Term frequency and Inverse Document Frequency.

Term Frequency is the number of times a word appears in the document. Because a word tends to appear more frequently in a longer document than it does in a short one regardless of its real importance to it, we need to normalize this value to avoid bias in favour of longer documents [16]. The mathematical representation of this property for implementation is:

$$tf_i = \frac{n_i}{\sum_1^k n_k} \tag{2.1}$$

where tf_i is term frequency of the considered term i, n_i is the number of occurrences of I, and $\sum_1^k n_k$ is the total number of occurrences of all words in the document [13].

Inverse Document Frequency is a measure of the importance of a word or a term to the document. It is the logarithm of the quotient obtained by dividing the total number of documents by the number of documents that contain the search word or term. The mathematical representation of this property for implementation is:

$$idf_i = \log \frac{N}{\sum_{k \in N}\{d_k : t \in d_k\}} \tag{2.2}$$

where idf_i is the inverse document frequency of the considered term i, N is the total number of documents, and $\sum_{k \in N}\{d_k : t \in d_k\}$ is the sum of the number of documents that contain word i [16]. The natural logarithm is the most commonly used one in calculating inverse document frequencies.

Once we have term frequency and inverse document frequency calculated, the value of tf-idf is simply the product of the two:

$$tf - idf = tf \cdot idf \tag{2.3}$$

It shows that a high tf-idf weight is archived by a high term frequency in the given document and a low document term frequency in the whole set of documents. Therefore, terms appearing commonly in all documents or infrequently in a considered document tend to be given low weights and thus can be filtered out [14].

Double Metaphone Code

Double Metaphone is one of the most commonly adopted phonetic matching algorithms. It is the second generation of the Metaphone algorithm family that accounts for not only pronunciations of English words, but also their alternatives accumulated, preserved, or assimilated in English from other languages such as French, Italian, Spanish, and etc.[17] It indexes words by their pronunciations and generates two keys, primary and alternate, that represent the sound of the words [9]. To compare two words for a phonetic match, one takes the primary and alternate keys of the first word, com-

pare them with those of the second word. The two words are considered phonetically matching only if one of the following four cases is met:

1. Primary Key 1 = Primary Key2
2. Primary Key 1 = Alternate Key 2
3. Alternate Key 2 = Primary Key 1
4. Alternate Key 1 = Alternate Key 2

According to the author of this algorithm, the above four comparisons result in decreasing similarities of the sounds of the two words. If none of the keys are matched, the two words are considered phonetically dissimilar [9].

Local Phrase Frequency

The phrase frequency keeps track of the frequency of phrases that appear in a document. To the context of this paper, a phrase is a group of words which does not necessarily function as a single unit in the syntax of a sentence. Rather, it represents one or more consecutive words placed in the same order as they are in the contained sentence. For example, given the sentence "I live in San Jose", "I live", "live in" and "in San" are all considered as valid phrases. However, "I in San" is not a phrase because "I" and "in" are not consecutive. Neither is "live I San Jose" because the words are not in the same order as they appear in the original sentence.

The way we count phrase frequencies takes a bottom-up approach. Starting from the beginning of a document, it first groups every two consecutive words together and calculates the frequency. Next, it groups every three

consecutive words together and calculates the frequency. This process goes on and on till it groups all words of the document together and calculates the frequency, which is always 1. The following illustrates the order of the phrases derived from the document "Hi, I live in San Jose.":

1. Hi I
2. I live
3. live in
4. in San
5. San Jose
6. Hi I live
7. I live in
8. live in San
9. in San Jose
10. Hi I live in
11. I live in San
12. live in San Jose
13. Hi I live in San Jose

Because all we care about are the permutation of consecutive words, punctuation is neglected and removed from the sentence. Phrases derived from

the above list are searched through the whole document to count their occurrences in the document. Again, in order to prevent bias towards longer documents, the occurrences are divided by the word length of the document. The quotients thus serve as the phrase frequencies. Each phrase, together with its frequency, is then saved in a local phrase frequency table for each document. We call it local because this value is independent of the content of other documents in the document set.

Global Phrase Frequency

After the local phrase frequencies of a document are calculated, they are then copied to the global phrase frequency table. If the same phrase exists in the table, its frequency is increased by that of the same local phrase frequency. If the phrase does not yet exist in the table, the local phrase frequency is simply copied to the table as a new entry. The reason of having both local and global phrase frequencies is so that we know not only how often a set of words occur together, but also how frequently such a combination appears across documents.

2.1.3 Dictionary Maintenance

Because the system mainly relies on the additional document properties built during the creation phase, and some of the values of these properties are related to the whole document set, maintaining the integrities of the properties is essential for the system to function properly. As new documents are added to the document set, the dictionary is dynamically updated to adjust the relative term match strength of each document that is derived from

the original text of the new documents. The major work is to re-calculate the tf-idf weight property of each document. As shown in the Dictionary Creation section , such calculation is mathematically straightforward. A database script can just do the job. Furthermore, in most cases this process does not have to be performed each time a new document is added to the document set. A better approach is to periodically execute that same script but only look for those unprocessed documents since the last run, and re-adjust the whole document set at the end. This is possible because all that existing documents need is the size of the new document set and the lists of terms in the new documents. Therefore, the time complexity of dictionary maintenance by running a database script is constant, regardless of the size of the new documents processed. Last but not the least, the frequency of performing this batch maintenance depends on how often new documents are entered into the database and how well one wants to keep the system up-to-date.

2.2 Single Word Search

Searching for a single word involves finding all matching documents and sorting them in the order of relevance. If the number of results is lower than the predefined configurable Result Size threshold, the system starts the phonetic matching action to try to look for candidate documents containing words that sound like the queried one. Then these candidates are ranked based on their relevance to the query and only those that exceed the predefined configurable Sound-Like threshold are returned. Therefore, we

break down single-word search into two stages:

1. The system performs text matching search. If the queried word is found in more than the Result Size number of documents, the system sorts and returns all of them.

2. If the queries word is not found or only exists in the number of documents smaller than the Result Size threshold, the system performs phonetic matching search, sorts and returns the results.

The two stages can be further categorized into three steps: Word Matching, Resulting Sorting, and Phonetic Matching. Each of them is discussed in more detail below.

2.2.1 Word Matching

Word matching is the most straightforward case to deal with in our system. We use a combination of the Boolean Model and Vector Space Model on the metadata of the dictionary entries the system has built at creation time to not only find documents that contain the searched word, but also sort them in terms of relevance.

The Boolean Model is based on Boolean logic and sets theories. The idea is to conceive both the word query and the word list metadata of the searched documents as sets of terms [1]. Searching is purely based on whether or not the query word exists in the document word lists. Boolean Model is quite efficient in that it does not check for things such as term frequency, document length, etc. All it does is to make sure the retrieved documents

have all the queried terms appearing in them at least once [2]. On the other hand, despite of the mathematical simplicity of the Boolean Model, it has its limitation because if several documents all present the same terms but with different occurrences, different number of unrelated unique words, or even different document length, they are indistinguishable to the model. Therefore, we need to further process the result set so documents that are more related to the query are placed before the others.

2.2.2 Result Sorting

Vector Space Model can just do the job. It represents document text as vectors in an algebraic model where each non-zero dimension corresponds to a distinct word in that document [10][13]. For example, if we have "The quick brown fox jumps over the lazy dog" as a document, the dimension of "the" is two while that of the rest words are all one and any other dimensions are zero. If we build vectors for the respective documents, we can actually calculate the document similarities by comparing the angles between them. This is one of the assumptions made in the Document Similarities Theory [4]. Like we said at the beginning of this section, because a query can be considered as a short text document, search is really just pulling documents out of a data collection in terms of their similarities to the original query document. To calculate vector angles, they have to be in the same vector space with the same dimensions. Therefore, the system first takes the union of all unique terms of two documents. The size N of the union set is used to create two vectors of N-dimension. These two vectors are both initialized to zero. For the first vector, each of its dimensions is then changed from zero to

a non-negative integer indicating the frequency of the word that dimension represents in the first document. Zero means no word occurrence. Then the system does the same for the second document vector. Now that the two vectors in the same vector space contain each document's word frequency, we can find out the angle between them in order to see how closely related they are.

While the above approach is conceptually straightforward, there are a few things our system has been implemented differently to further improve its accuracy.

The first is the vector representation in the Vector Space Model. Instead of using term frequency as values for vector dimensions, we applied the tf-idf weights to evaluate how important a word is to the considered document [13]. The previous application is simple but fails to take into account the key factor that longer documents might have a low proportional term frequency even thought that term may have a higher occurrence than it does in a much shorter document. In such cases, it is obviously imprudent to simply take the longer one just because of its higher term frequency. What we need is to evaluate the relative importance of the term in a document to the whole document set. This is why we want to apply tf-idf weights. As stated in the previous section, the local tf parameter normalizes word frequencies in terms of the length of the document the words reside in. Furthermore, there is also a global parameter idf, which contributes to the result the frequency of the documents containing the searching word relative to the whole document set. The product of the two parameters, known as the tf-idf weight thus tends to represent the similarity of two documents with respect to the local term

frequency ratio (to the document itself) and the overall document frequency ratio (to the whole document set) [15]. In other words, rare terms are given more weight than common terms. Therefore, in our system a document is represented as a weight vector:

$$v = [tf - idf_1, tf - idf_1, ..., tf - idf_i] \tag{2.4}$$

where i is the sum of the number of distinct words in two documents.

One another change is that, instead of find the angel between two vectors, our system takes a more common approach of calculating the cosine of an angle, which is equal to the dot product of the two vectors divided by the length of each of the vectors. The mathematical representation of this property for implementation is:

$$\cos(\theta) = \frac{v1 \cdot v2}{|v1| \cdot |v2|} \tag{2.5}$$

When $v1$ and $v2$ are perpendicular, $\cos(\theta)$ is zero. It means the documents represented by the two vectors have no match. In other words, the queried word does not exist in the considered document. In our system, $\cos(\theta)$ is always positive because it is only applied after the Boolean Model eliminates non-relevant documents in the previous text matching stage. Therefore, we can sort the results by their cosine similarities: the large the $\cos(\theta)$ is, the more relevant the document is to the query.

Incorporating the above changes, the sorting process works the following way:

1. Construct two initial document vectors of the same dimensions from the query and a document
2. Take the tf-idf weight values of the query and the document from the dictionary and fill them into the corresponding vector dimensions
3. Calculate the cosine value of the angel between the two vectors
4. Repeat step 1 to 3 for each document in the result set returned by Boolean text matching
5. Sort the result set by their cosine values. A larger number indicates higher relevance of the corresponding document

2.2.3 Phonetic Matching

So far, the search system has been running strictly on exact text matching. However, if the result set is empty or so small that it does not contain information a user looks for, the system starts a different kind of search that is based on pronunciations to hopefully increase the number of hits. We observe that when a query results a low number of hits, there are two possible types of reasons:

1. Query is spelled correctly but the data pool just does have a lot of documents containing the keyword.
2. Query is spelled incorrectly. Otherwise the search pool may return more hits.

The system first performs a search operation assuming the spelling is correct. If not enough results are returned, it then performs another search operation with spelling correction. We will discuss each of them in the following sections.

Low Hits Resulted from a Correctly Spelled Query

When a query is correctly spelled but results in few hits, the system will try to broaden the result by looking up words that sound similar to the query in the document set. Here is where another metadata, the Double Metaphone code built during the dictionary creation process, comes into play. Because words of same or similar pronunciations are encoded into the same or similar Double Metaphone code, a simple database query that compares the index Double Metaphone codes of two words will return a list of words that sound like the queried one - straightforward. The less trivial part is to sort this list of words so that those whose pronunciations are closer to the queried one are put before the others. The system makes such decisions based on the Levenshtein Distance of the candidate words from the original word. The Levenshtein Distance is a measure of difference between two sequences. When it is applied to two strings, it represents the minimum number of operations one needs to transform one string to the other[12]. The basic operations are insertion, deletion, and substitution. All operations are performed on the character level. In other words, each operation can only manipulate one single character in a string. For example, the Levenshtein Distance of the word "night" and the word "right" is one because one can transform one to the other by substituting the letter "n"

with "r", and vice versa. The word "knight" also has a Levenshtein Distance of one from "night" because inserting "k" at the beginning of "night" results the other. In a generalized version of the Levenshtein Distance definition, character transposition (permutation of two characters while keep the rest fixed) is also allowed. However our system only considers the basic three operations for simplicity concerns because it is always possible to replace one transposition operation with a deletion following by an insertion, and it does not affect the relative costs among string transformations.

Because Double Metaphone codes are strings, we can apply the Levenshtein Distance to measure their differences and thus calculate the similarities of their sounds. Words that are phonetically identical always have the same Double Metaphone code, so their Levenshtein Distance is zero. As the pronunciations of two words become less and less alike, their Double Metaphone codes will have more different characters from each other and thus result in a further Levenshtein Distance. Therefore, the system ranks the Levenshtein Distances between the query and the candidate words, and sort them based on the different Levenshtein Distances.

Low Hits Resulted from an Incorrectly Spelled Query

It is not uncommon for users to enter misspelled queries, and the number of hits resulted from it is almost guaranteed to be small, if not empty. From the system's perspective, misspelling is inevitable and its forms are unpredictable. It could be a missing-letter error, a double-letter error, or a swapped-letter error. Each of them could involve one or more letters and all of them could occur in one query. The challenges here are to identify the

errors, come up with a list of correction candidates, and sort them in terms of their proximities to the original misspelled word. There are quite a few spelling correction algorithms out there. Most of them are derived from the combination of Levenshtein-Distance-based and q-gram-based algorithms. We have seen how Levenshtein Distance is calculated. Next we are going to briefly introduce q-gram and q-gram-based algorithms.

A q-gram of a string is a substring of q characters. For example, if q = 2, the word "search" has the following q-grams:

SE EA AR RC CH

By comparison, for q = 2, the word "march" has the following q-grams:

MA AR RC CH

Therefore, these two words share the AR, RC and CH q-grams. A typical q-gram-based algorithm counts the number of q-grams two words have in common. The higher the count is, the stronger match it yields.

A typical q-gram algorithm that has been used widely works the following way [7]:

1. A query word is first broken into q-grams, where q is, say, 4.
2. Each unique word in the dictionary is also broken into q-grams, where q is the same as that defined in step
3. All the q-grams are put in the Boolean model to perform a logic AND operation to retrieve the top correctly spelled words that share the most number of q-grams with the query word.

4. If there are not enough number of hits, decrement q by one and do step 1 to step 3 again. Repeat step 1 to step 4 till q is equal to 2 or enough hits are returned from step 3.

5. Finally, the retrieved words are ranked in terms of their Levenshtein Distances to the misspelled word, with the closest ones at the top.

While this method is good for its simplicity, it falls short on several aspects. First of all, it is inefficient. Breaking a word into q-grams has linear space complexity and such operations need to be performed on each single word in dictionary. Therefore, the amount of space step 1 and step 2 take is $O(nq)$ in total, where n is the dictionary size. To make spelling correction practical, a reasonably comprehensive dictionary with a few hundreds of thousands of words and phrases is necessary. Breaking each word in such a dictionary could cause heavy disk overhead, which in term affects performance. Furthermore, step 3 is time-consuming because it involves set intersections and sorting, which is $O(q^n)$ [3]. To make it worse, if what step 3 got does not meet with the requirement of step 4, it routes the search process to step 1 again and again until the result is satisfying.

Secondly, it is sometimes inaccurate. For example, let's set q to 2 and there is this misspelled word "cta" entered by a user, who actually meant the word "cat". The 2-grams of the incorrect word is "CT" and "TA", while the 2-grams of the correct word is "CA" and "AT". It is obvious that the above algorithm is not going to find the most desired word because the intersection of the two q-gram sets is empty. This is the limitation of the algorithm so that, even if q has been set to its finest possible granularity,

the correct word "cat" will never be returned. Such problems tend to occur often in short words because of the fewer q-grams that can be generated from them. Our system takes a different approach. Similar to the previous search process, it involves two steps:

1. Find correctly spelled candidate words that are close enough to the query word
2. Rank the candidates and return the most matched one(s)

The next two sections discuss each of the above steps in details.

Find Candidate Corrections We observed that in most cases a misspelled word had a Levenshtein Distance of no more than 3 from the correct word. We also noticed that errors causing the distances tend to occur towards the end of the misspelled words. Because we are only interested in the words that are the closest to the query word, the above two observations suggested that, instead of considering all possible words that have all possible Levenshtein Distances from the query word, we could just focus on the one-, two-, and three-Levenshtein Distances of the beginning portion of each word. The following is how it works:

1. Given a query word of length n, set $k = \lceil 0.6n \rceil$, where k is the number of beginning characters that are going to be taken from the query word.
2. If $k \leq 3$, $k = min(3, n)$; else if $k > 7$, $k = 7$. The lower bound of k is the less of 3 or the length of the word. It is to make sure there are

enough possible permutations to form Levenshtein Distance of three to maintain the level of accuracy. The upper bound of k is 8. This reflects our observation that the beginning portion of a query word is more likely to be correctly spelled, so the following correction process will use this portion as the base to find matches. We set the lower bound of k to 3 and upper bound to 7 because with the data set we experimented on, they seemed to be the golden numbers that balanced accuracy and efficiency.

3. Take the first k characters of the query word and generate a key set where each item is a key whose Levenshtein Distance are one, two, or three from the k-length string.

4. Check each key in the key set for a given query against the word list metadata of each document in the document set. Return the words that also start with those same keys.

For example, if the misspelled query word is "acommodation" (should be "accommodation"), the system first finds its length $n = 12$. Then it sets $k = \lceil 0.6 \cdot 12 \rceil = 8$, which is greater than 7, so k is set to 7. Now the first 7 characters "acommod" are taken from the original query and added to the key set as the first key. Next, new keys that have one Levenshtein Distances from the first key are generated, followed by generating new keys that have two Levenshtein Distances and three Levenshtein Distances from the first key. The following list is the key set snippet with Levenshtein Distances of one that are generated by inserting the character "c".

.....

acommodb (insert "b" after character 7th)

cacommod (insert "c" before 1st character)

accommod (insert "c" before character 2nd)

accommod (insert "c" before character 3rd)

cacocmmod (insert "c" before character 4th)

cacomcmod (insert "c" before character 5th)

cacommcod (insert "c" before character 6th)

cacommocd (insert "c" before character 7th)

cacommodc (insert "c" after character 7th)

dacommod (insert "d" before character 1st)

.....

Note that after "c" is inserted into "acommod" before 2nd character, the resultant string "accommod" is part of the correct word the system is expected to return. Finally, the keys in the above list and the rest of the keys in the key set are all checked against the word list metadata. If the word "accommodation" exists in at least one of the word lists, it will be returned, together with other candidates whose beginning portion has a Levenshtein distance of at most three from "acommod". For instance, "command" is another candidate because "comman" has a Levenshtein Distance of three from "acommod" (delete "a", replace "o" with "a", and "d" with "n").

From our experiment, the size of candidate corrections only ranges from a couple of words to at most several hundred in a document set consisting of a few hundreds of thousands of unique words. This is due to the large

number of false keys in the key set that are phonetically incorrectly. Because of the relatively small data pool, we are able to implement a reasonably comprehensive scoring system to rank the candidates in order to find the best match.

Rank Candidate Corrections Now that a list of candidate words close enough to the original query word have been found. The next task is to pick out the best match(es). The system has its scoring system that ranks every candidate by taking into account several factors, each of which contributes to the final score that a candidate gets and the highest scored candidate(s) are considered the best match(es). The factors are:

Weighted Levenshtein Distance from a candidate to the original misspelled query word. The reason to compare it with the complete word rather than its first k characters is to ensure the evaluation reflects the relevance of a candidate to the query word as a whole. Levenshtein Distance has been discussed and used in quite a few places in previous parts. In this section, we are going to applied one of its variations: Weighted Levenshtein Distance. The difference between this one and the original version is that, instead of treating insertion, deletion and substitution with equal weights, each of the three operations is now assigned a cost that may vary from each other. The concept has been commonly used in bioinformatics, known as the Needleman-Wunsch algorithm, for nucleotide sequence alignment[8]. It makes sense in our application domain because among all spelling mistakes, some are more likely to occur than the others. Table 2.1 is a list of allowed operations in calculating the Weighted Levenshtein Distance and the costs

associated with these operations.

Operation	Cost
Insertion	0.9
Deletion	1
Substitution	0.6
Transposition	0.6
Double Letter Error	0.4

Table 2.1: Operations for Weighted Levenshtein Distance calculation and their costs.

From the table, we see that Deletion costs more than any other operation, indicating that with everything else identical, if one candidate has an additional character from the original word (i.e. needs a deletion to transform it to the original word) while another candidate has the same characters but two of them are in a slightly different order (i.e. needs a transposition to transform it to the original word), the latter gets a higher rank because of its lower cost. We gave Insertion a slightly lower cost than Deletion because we observed that it was more likely for a user to miss a character than to have an extra unwanted one. Similarly, entering a wrong character or swapping two consecutive characters are more likely to occur than the previous two operations, and thus are assigned an even lower cost. The last row of the table is a special operation we defined particularly for word matching. Without it, a misspelled word with an extra character same as one of the two characters next to it would have to take the most expensive Deletion operation to match to the correct version. However, we observed that when this kind of mistakes happened, the most accurate results were usually generated by removing one of the double letters. Therefore, we de-

cided to include this supplemented low-cost operation to better model the real user behaviour. The Weighted Levenshtein Distance is the total cost of performing one or more operations in Table 2.1. Furthermore, because shorter words are more likely to get a smaller total cost due to the fewer operations that can possibly be performed on them, the score has to be normalized to avoid such bias. Therefore, if let c be the total operation cost to transform a candidate to the query word, and n be the query word length, the score from the Weight Levenshtein Distance can be calculated as:

$$1 - \frac{c}{n} \tag{2.6}$$

where c is always less than or equal to n because the maximum cost is no greater than 1. The smaller c is, and the larger n is, the larger the score is, which means the closer the candidate is to the misspelled query word. The extreme case, thought not going to happen in the system, is when the score is one, then c must be zero, meaning the candidate word and the query word are identical.

Second, **Starting and Ending Characters** of a candidate word are checked against those of the query word. The more beginning or ending characters the two words share in common, the more likely the candidate is the correction of the misspelled query. It was also from our tests that users tended to be more sure about the beginning and ending letters of words they enter. As the positions of the letter were closer to the middle of the word from both sides, spelling mistakes were more likely to happen. Therefore, we took into account this factor in the ranking system with a linear scoring

function which works the following way:

1. Set $s = 0$. Starting from the first letter of the candidate and the query word, check if they are identical. If they are, increment s by 1 and move on to the next letter (in this case, second one) of both. Repeat this process till:
   (a) the two letters at the same position from the two words are not the same,
   (b) or the letter position is equal to half of the length of the shorter word.
2. Set $e = 0$. Starting from the last letter of the candidate and the query word, do the same as Step 1 except that it checks for the second half of the words.
3. The final score for this factor is calculated as:

$$\frac{s+e}{min(n_q, n_c)} \tag{2.7}$$

   where n_q is the length of the query word, and n_c is the length of the candidate word. The division is necessary to normalize the score to prevent bias toward longer words. Also, because both Step 1 and Step 2 must stop at half of the shorter word, $s + e$ must be less than or equal to $min(n_q, n_c)$. Therefore, in the extreme case, if the candidate and the query word are identical, the score is one, which agrees with the mathematical meaning of the previous score of the Weighted Levenshtein Distance.

Third, **Double Metaphone code** of both the candidate word and the query word are compared to calculate the third score based on their pronunciations. As we saw early in the introduction section of the Double Metaphone algorithm, a Double Metaphone code consists of two keys, one primary and one alternate. The evaluation of how similarly two words sound like each other works the following way:

1. if the primary key of the candidate is the same as the primary key of the query word, the candidate gets 0.3
2. else if the primary key of the candidate is the same as the alternate key of the query word, or if the alternate key of the candidate is the same as the primary key of the query word, the candidate gets 0.2
3. else if the alternate key of the candidate is the same as the alternate key of the query word, the candidate gets 0.1
4. else if none of the above three conditions is met, the candidate gets 0

The score is calculated in the same order as the above three methods are presented. In other words, the maximum score a candidate can possibly get from this factor is 0.3. One might wonder why this factor does not contribute to the total score as much as the previous two. We made our decision based on two reasons. The first is that, as accurate as the Double Metaphone algorithm was designed based on sophisticated phonetic rules, the author also admitted that, in some cases, it may fail to generate unique codes to distinguish certain words that are pronounced only slightly differently, due to the complexity and the "maddeningly irrational spelling practices" [9] of

English. The second and more important reason is that, even if there were a perfect phonetic mapping algorithm that could distinguish every single different pronunciation, it is still not able to consider words that sound the same but differ in meanings. These words are known as homophone. For example, "right", "write", "Wright", and "rite" are homophones. Their spellings are considerably more different than their pronunciations are. Because it is unlikely that the users would misspell a word as one of its homophones, we had to be careful not to overly rely on phonetic similarity. This is why the Double Metaphone score is weighted only about 1/3 of the previous spelling-oriented factors. It serves more like a fall-back measure should there be a tie in the first two factors.

Lastly, if necessary, **Occurrences** of candidate words in the word list metadata of each document are compared. By necessary, it means that this step is performed only if all the above comparisons result a tie. What this factor considers is how important a candidate is to the document set. The more often a word appears in the document set, the higher score it gets. Because occurrences of words are highly unlikely to be equal in the whole document set, this is our last resort for picking out the golden word. If it ever happens that the occurrences of candidates are still the same, one of them will be randomly returned as the best fit.

Search on Phonetically Similar Word

From either of the two cases, low hits because of a rare query word or because of a spelling mistake, now that the best phonetically matched word has been found. The next task is straightforward. The system performs a

single word search using the new word as the query word. All it does is to repeat the Word Matching and Result Sorting steps discussed previously and find the result documents. This time, Phonetic Matching is not going to be performed again because the new query is from the document set, which means a none-empty return set is guaranteed.

2.2.4 Summary

The single word search consists of two stages that based on different approaches. The first stage is to match text between the query word and the words in documents. It works fine under the circumstances that a query word is correctly spelled and it exists in the document set. If either of the cases fails, the system moves on to the second stage. In the second stage, a more sophisticated scoring system is applied to find a new query word from the document set that is phonetically close to the original query word. The new query word is then used to perform another stage one search to return more relevant hits.

2.3 Multiple Word Search

So far we have seen how a single word can be searched phonetically in a document set. Now we are going to deal with phrases, queries consisting of multiple words. Similar to single word search, there are two stages involved in searching for a phrase:

1. The system performs text matching search. If the queried phrase is found in more than the Result Size number of documents, the system

sorts and returns all of them.

2. If the queries word is not found or only exists in the number of documents smaller than the Result Size threshold, the system performs phonetic matching search, sorts and returns the results.

The two stages can also be further categorized into three steps: Phrase Matching, Resulting Sorting, and Phonetic Matching. However, the implementations of them are somewhat different from those of their single-word counterparts. The following sections are going to discuss them in detail.

2.3.1 Phrase Matching

Again, this is the most straightforward part. Similar to single word matching, it also applies the Boolean Model to treat documents and the query string as sets of phrases. However, the variation here is that, instead of looking at the word list metadata of each document, the phrase is checked against the local phrase frequency tables. Since each table consists of entries of two or more words separated by a whitespace, the query also needs to be re-formatted by removing punctuation, non-single whitespaces, etc., and inserting one and only one whitespace in between words to make it more search-friendly. Furthermore, phrase entries whose word lengths are less than that of the query string are neglected because it is impossible for them to hold the query string. At the end of this step, a set of documents containing with the query string are returned.

2.3.2 Result Sorting

Sorting is based on the importance of the query string to the document and the whole document set. Therefore, this is where both the local and the global phrase frequency tables are needed. As a matter of fact, they are the only two factors for the system to decide the order of the result documents. One might wonder why sorting for multi-word queries are so much easier than doing the same for single-word queries. The reason is that, the Vector Space Model with tf-idf weights used for the single-word ones does not apply here because the counted frequencies of phrases are not all meaningful. For example, given the sentence "I live in San Jose", "in San" is a valid phrase in our system but it does not function as a meaningful unit in the sentence. In statistics, such cases are called noise. If vectors were created with too much noise in the dimensions, the result could turn out unpredictable. For example, a document with more meaningless phrases but shorter in word length could be placed before others that are actually more relevant. On the other hand, from our tests, the simple phrase frequency comparison worked well. Each document gets a score which is the product of the local and global phrase frequencies of the query string. The higher the score is, the more relevant that document is to the query string. This method produces reasonably good results because it takes into account the importance of a phrase both locally to the document and globally to the whole document set. Therefore, although not as accurate as the Vector Space Model to single-word queries due to not considering other phrases, such ignorance prevents unpredictable behaviours from data noise and produces good results.

2.3.3 Phonetic Matching

Similar to single-word search, if the strict text-based matching does not return satisfying results for the phrase, the system starts the sound-based search. As we have discussed earlier, the phonetic matching algorithm for single-word queries returns a list of sound-like candidates and ranks them in terms of the phonetic closeness to the query word. For phrase queries, we reuse the same algorithm for phonetic matching in the following way:

1. Break a query phrase into a list of single words.
2. For each word, perform the single-word phonetic matching operation to retrieve a list of top candidates.
3. Consider all possible permutations of the candidate lists by taking one word from each of them. For each permutation, refer to the global phrase frequency table to get its global frequency in the whole document set. This is called correlation check.
4. After all permutations are generated and their global phrase frequencies are check, return the one with the highest frequency.

For example, given a query phrase consisting of three words, the first two steps of this method are simply to repeat the single-word phonetic matching operation on each of the three words. In step three, if every word has 100 candidate words, there are at most 1,000,000 ($100 \cdot 100 \cdot 100$) combinations of candidate phrases to be checked against the global frequency table. We say "at most" because optimizations are implemented to significantly shrink

the exam pool. We will come back to it very soon. The last step is just to return the one with the highest global frequency.

One may wonder if it would still generate good output by just considering each phrase word independently and combining the results together as the phrase search hits. From our tests, the answer is no because it would get a lot of cases severely wrong. For example, a user wants to query for "San Jose" but accidentally spelled the query "Ssn Jose". For "Ssn", the single-word correction algorithm will probably return candidates such as "sun", "son", and "San". If we do not consider the correlation between the query words, because the word "sun" or "son" seems to be more commonly used than "San", it is very likely that one of them will be chosen as the best fit to form the "corrected" phrase "Sun Jose" or "Son Jose", whereas the user really meant to search for "San Jose". This is why collocation check is performed to prevent such cases from being returned. Consider the above example. Even if the individual candidates may have higher frequencies, they have to be evaluated by how often they are seen together. As a result, "San Jose" will get a higher frequency from the frequency tables than any other incorrect phrases combined by correct words.

As the number of words in a query increases, the number of permutations from all candidate word lists grows exponentially. We have seen in one of the previous examples that, for a three-word query, there are 1,000,000 permutations because in order to retain good proximities each candidate list contains the top 100 closest words. If a query has five words, the number of permutations quickly boosts to 10 billion! Therefore, a brute-force approach going through all possible combinations is infeasible due to its huge time and

space complexities. Fortunately, we realized that a permutation could be generated by selecting one word from each candidate list and then concatenating the selections together. It means before a permutation is formed, all entries in the global phrase frequency table are possible matches. Then, the first word from the first candidate list is chosen as the first element of the permutation. At this point, those phrase frequency entries not containing the same first word do not need to be further considered because it is guaranteed they will not be a match to the whole permutation. Next, the second word from the second candidate list is chosen as the second element of the permutation. Now, among the phrase frequency entries left from the previous selection, those that do not contain the same second word do not need to be further considered either because they will not be a match to the whole permutation for sure. The process goes on till either the permutation is completed or there are no phrase frequency entries left to be selected. If the permutation is completed, it means there is a match in the phrase frequency table. Otherwise, it means there are no such phrases that can match the incomplete permutation from its first element up to its last element that is generated right before the process stops. Therefore, all permutations with the same beginning elements as the incomplete one can also be "purged" because matching to them is also guaranteed to be empty. Moreover, before any permutation is formed, we can further optimize the process by reducing the phrase frequency pool to only those entries with the same number of words as that of a complete permutation. This has the effect of limiting the initial size of the data set to make it converge more quickly. Figure 1 is an example of the optimized permutation generation process on a three-word

query.

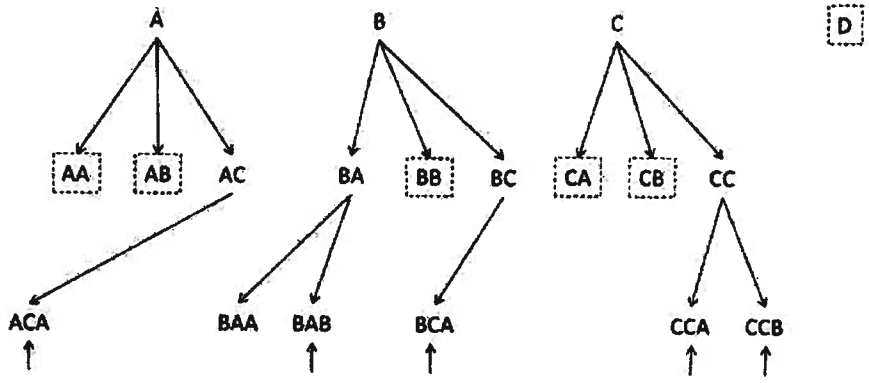

Figure 2.1: Optimized permutation generation process on a three-word query

Suppose there are five three-word phrases in the phrase frequency table. They are "A C A", "B A B", "B C A", "C C A", and "C C B". No matter how many phrases are there in the original phrase frequency table, these five are always the ones to beginning with because any other phrases with more or less words are not considered. For simplicity, each candidate list has four words, "A", "B", "C" and "D", to be chosen from, and there are three such candidate lists. Therefore, without optimization, a total of 64 ($4 \cdot 4 \cdot 4$) permutations need to be generated and checked against the five existing phrases. Let's see how optimization can speed up this process. We begin by picking "A", the first word in the first list, as the first element of the permutation. Because there is only one phrase of the five starting with "A", the other four do not need to be checked for the rest of this permutation. Then another "A", the first word in the second list,

is picked as the second element of the permutation. Now this permutation starts with the words "A A". Because the only phrase left from the last selection does not contain "A A" as its beginning portion, we can stop this permutation at this point. More importantly, any other permutations starting with "A A" do not need to be further considered. In Figure 1, we represent the termination of such a "branch" with a dotted board around the last generated partial permutation. Thus, "A A" is surrounded by a dotted board. Next, we pick the second word "B" from the second list. Similarly, there are no phrases with the "A B" beginning portion, so any permutations of "A B X", where "X" can be either "A", "B", "C" or "D", are ignored. Thus, "A B" is also surrounded by a dotted board. Next, we pick "C" from the second list to form "A C". Because "A C A" matches to "A C" for now, we can move on to the third list and select "A" from it to for the first complete permutation "A C A". At this point, we find a match and no further permutations of "A C X" will be performed because we know there is only one phrase in the form of "A C X", so we stop exploring any other permutations starting with "A". Instead of 16 possible permutations and comparisons, only 1 complete permutation is generated and 5 comparisons are made between the incomplete permutation and the phrase entries for all permutations starting with "A". The same steps are repeated for the rests till all five phrases are found. One extreme case is when a permutation starts with "D". All the 16 "D X X" combinations are ignored. A save of 16 generations and 15 comparisons! In figure one, only a total of six complete permutations are generated and five of them are the matches. From out tests, such optimization could save over 90% of

computation time on average.

2.3.4 Summary

Similar to single word search, phrase search also consists of two states. The first stage matches text between the query phrase and the phrase entries extracted from their documents. If results are found, they are sorted in terms of their local and global phrase frequencies to the documents. The higher frequency a phrase is in a document and to the whole document set, the more relevant that document is to the query phrase. If there are no hits returned from the first stage, phonetic matching kicks in. A query phrase is broken into words to find sound-like candidates. These candidates are then combined to form different permutations that are checked against the global phrase frequency table. The one with the highest frequency is considered the golden match and text search is performed again using this new phrase to return more hits.

Chapter 3

Evaluation

In this chapter, we evaluate the performance of our system. To do this, we have created our simulator and test data to search for restaurant names throughout the Greater Vancouver Region. We examine the effect of single- and multiple-word searches with phonetic matching. By comparing the results to the actual data in the test document set, we evaluate the search accuracy and running time of the system with different types of inputs.

3.1 Simulation Setup

3.1.1 Test Data Pool

The test data is a set of restaurant names in the Great Vancouver Region. We have built a crawler with the free software Web Scraper Lite [6] to grab restaurant listings from Yellowpages.ca[18], extract their names and store them in a database table in MySQL. The data pool consists of more than 3800 documents each of which is the name of a restaurant. We chose restaurant names as our test data because of two reasons. First, our system was designed specifically for short text documents. We observed the length of a restaurant name usually varied from one to eight words, and thus they

would make good test data for our evaluation. Secondly, because of Vancouver's diversity of cuisines from all over the world, some names of these restaurants originated from languages other than English could be difficult to spell right if one only knows the pronunciations approximately. For example, "Strozzapreti" is an Italian word for a kind of pasta, pronounced as "stroh-tzuh-pray-tee". Without knowing Italian, it is not unlikely for one to misspell it as "Strozapreti", "Strozzapreiti", or "Strozzaprety" because all of them share the same or very close pronunciations as that of the correct version. Searching for such misspelled queries in the test data can help us understand the level of accuracy and usability the system offers.

3.1.2 Test Input

We created the test input in two stages. First, a set of correctly spelled words and phrases were generated. These words and phrases must not appear in the test data pool. Second, we created a set of misspelled words and phrases with a Levenshtein Distance greater than zero but less than or equal to five from the existing words and phrases in the test data pool. There are 1000 inputs in total for the test. Table 3.1 is a summary of the types and the sizes of the input we tested on.

Input Type	LD1	LD2	LD3	LD4	LD5	Total
Correct Word	N/A	N/A	N/A	N/A	N/A	250
Correct Phrase	N/A	N/A	N/A	N/A	N/A	250
Misspelled Word	61	59	49	44	38	250
Misspelled Phrase	59	55	53	46	37	250

Table 3.1: Test input types and sizes

For misspelled words and phrases with different Levenshtein Distances, we tried to make the size of each group as close to each other's as possible. However, due to the various lengths of restaurant names, it was inevitable that misspellings with smaller distance tend to occur more often. This is because the group size grows reversely to the Levensthtein Distance. Furthermore, we categorized the misspelled words and phrases by the operations making them incorrect. Table 3.2 is a breakdown of the operation types and their sizes. The sums of the number of operations in both types are over 250 because some test cases require multiple operations to be corrected to the query word or phrase.

Operation	Misspelled Word	Misspelled Phrase
Insertion	83	79
Deletion	60	70
Substitution	134	115
Transposition	146	121
Double Letter Error	170	164

Table 3.2: Operation types and sizes

3.1.3 Simulator

We implemented a simulator in PHP to query the test data pool with the test inputs and to collect the test results. What it does essentially is the two searching stages for single- and multiple-word queries described in the previous chapter. We chose PHP as the implementation language for two reasons. First, a lot of libraries have been made available for PHP to make coding easier. For our program, we have used the Alix Math library [5] to

handle all the mathematical details such as vector calculation, normalization, Weighted Levenshtein Distance, etc. Secondly, PHP is one of the most popular languages for web development. Consider the system was designed primarily for web-based searching, implementing it in PHP gives us a feel of how it would perform in real web applications.

3.1.4 Environment

The test was conducted on an Ubuntu server with an Intel Core 2 Duo E7200 CPU (running at 2.5GHz, 3MB cache, 1066FSB) and 2GB memory. We also installed Apache 2.2.8, MySQL 5.0.51b, and PHP 5.2.6 for the simulator to query the database. The combination of these technologies is commonly referred to as LAMP (Linux, Apache, MySQL and PHP). It defines a popular web service infrastructure for its high flexibility, good scalability, and low acquisition cost. This LAMP server, both its hardware and software, represents the majority of the environments where our system was meant to be run, and therefore was chosen to be the test configuration.

3.2 Simulation Results

The primary goal of the simulation is to evaluate the accuracy of phonetic search when dealing with different types of input: correct word, correct phrase, misspelled word, and misspelled phrase. We will discuss each of them in this section.

The first two rows of Table 3.3 are the search results when a query was a correctly spelled word or phrase. The system yielded a 95.6% success-

Input Type	Number of Queries	Number of Matches	%
Correct Word	250	239	95.6
Correct Phrase	250	216	86.4
Misspelled Word	250	223	89.2
Misspelled Phrase	250	212	84.8

Table 3.3: Number of phonetic matches from the correct word, phrase queries, and from misspelled word, phrase queries

ful rate when dealing with single-word queries. It was because the search process took a regressing pattern to gradually increase the Levenshtein Distance between the Double Metaphone code of the query word and that of a document until it found the first match. For those the system did not find a match, it was because they were so randomly generated that their Double Metaphone Levenshtein Distance from any document was equal to the length of the Double Metaphone code itself. In other words, these words did not sound like any words in the test data. However, the 4.4% unsuccessful rate ($1 - 95.6\%$) indicates that, such cases are rare. On the other hand, searching for correctly spelled phrases yielded 86.4%, a considerably lower successful rate. This is because the system needs to find candidate words that are phonetically close to every word in a query phrase. If any word returns an empty candidate list, the matching stops. Furthermore, the more words a query phrase has, the less likely there is a match in the document set. This observation was proven by the fact that, among the 13.6% unsuccessful query phrases, most of them consisted of five or more words.

The last two rows of Table 6 are the search results when a query word or

phrase was misspelled. Similar to the previous results, single word queries yielded a high 89.2% successful rate. When we were generating the test input, we intentionally made all queries have a Levenshtein Distance no more than 5 to model the common error patterns. This is why the phonetic matching still worked well with spelling mistakes. It came a little surprise that the unsuccessful words were the smaller ones. We think it was because after normalization, even close Levenshtein Distance could be proportionally large to a small word. For example, if a word "aaa" has Levenshtein Distance of 3, it means every character of the word has to be changed to transform it to another word. For misspelled phrases, the successful rate is close to its correctly spelled counterpart. It was expected because even though they have different scoring mechanisms, the decisions made by these mechanisms are both based on the same factors - the Local and Global Frequencies. On the other hand, the relatively big accuracy gap between the correct and misspelled single-word queries suggested that even though their scoring mechanisms were similar, the different factors that each mechanism took into account resulted in different levels of accuracy .

In addition to accuracy, we have also evaluated the running time of performing various types of searches.

Table 3.4 shows the running time of all six different types of searches. It was no surprise that the simplest text-based searches took the least time while the sound-based misspelled word and phrase searches took the longest. It is worth mentioning that the maximum average search time is merely over a second and all types of searches have a small standard deviation comparing to its average time. Combining Table 3.3, we conclude that the

Input Type	Queries	Min.	Max.	Avg.	Std. Dev.
Text-based Word Search	100/100	110ms	371ms	193ms	76ms
Text-based Phrase Search	100/100	216ms	478ms	305ms	93ms
Sound-based Correct Word Search	239/250	607ms	1131ms	866ms	137ms
Sound-based Correct Phrase Search	216/250	715ms	1206ms	893ms	99ms
Sound-based Mis spelled Word Search	223/250	975ms	1704ms	1252ms	281ms
Sound-based Mis spelled Phrase Search	212/250	874ms	1569ms	1177ms	325ms

Table 3.4: Running time of different types of searches

system has behaved reasonably fast and stable with an over 80% successful rate regardless of the various types of inputs.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this thesis, we introduce a search system based on both text and sound matching for short text documents. The system makes novel use of some of the widely adapted algorithms and mechanisms in several fields by incorporating them into its staged searching process. The different stages have been created to deal with different search scenarios, and each stage has its own scoring model built upon some common algorithms and the metadata specially prepared for it. The various metadata associated with documents are the keys to the dictionary-based approach our system takes for phonetic searching. We provide a high level design specifying the system implementation from dictionary creation and maintenance to text- and sound-based matching for various types of queries. We also evaluate the system performance under these different circumstances. The results suggest that our system meets its design goal with respect to accuracy and efficiency.

4.2 Future Work

There are several areas in the development of the system that deserve further exploration. First of all, there are certain words, such as "the", "to", "not" or "is", that appear much more often than others but carry very little information. They are called stopwords. Due to their high occurrences in almost all documents, building dictionary metadata such as tf-idf weights, global and local phrase frequencies around them is expensive and usually not so useful. Therefore, it could be helpful to further increase the accuracy and efficiency of the system by taking into account these stopwords perhaps during the dictionary creation and phonetic searching phases. However, it is important to not overly rely on it. For example, the famous phrase "to be or not to be" consists of all words that can be considered as stopwords. Yet they cannot be all treated the same way as others. Therefore, while checking for stopwords looks like a promising way to improve performance on general queries, it requires further investigation to make it really useful without sacrificing the accuracy in special cases.

Secondly, when searching for misspelled words, the current design does not take missing whitespaces into account. Consider the word "georgebush". The best match might be "Georgetown" while what the user really meant was the phrase "George Bush". If we consider inserting whitespaces in "georgebush", we would get "g eorgebush", "ge orgebush", "george bush", etc., and thus find that the last one is the best match. Similarly, we can consider combining words to check for better matches. Again, the challenge here is to find the right granularity to balance between accuracy and

efficiency.

Last but not the least, the dictionary-based search approach usually promises an acceptable level of accuracy for relatively small document sets. However, as the number of documents grows, one might want to consider alternatives such as query log. Query log is a statistic model of an extensive set of prior queries that are compared to a user query and to suggest new queries that will result in more hits. In our system, the new queries would be phonetically similar to the original one. Because of the staged search behaviours our system was designed to, it is possible to allow query log or other techniques to be plugged in to further improve its performance.

Bibliography

[1] Daniel M. Everett and Steven C. Cater. Topology of document retrieval systems. *Journal of the American Society for Information Science*, 43:658–673, 1999.

[2] William B. Frakes and Ricardo A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms.* Prentice-Hall, 1992.

[3] George Karakostas, Richard J. Lipton, and Anastasios Viglas. On the complexity of intersecting finite state automata. In *IEEE Conference on Computational Complexity*, pages 229–234, 2000.

[4] Yuhua Li, Zuhair A. Bandar, and David McLean. An approach for measuring semantic similarity between words using multiple information sources. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):871–882, 2003.

[5] PHP Math Library.
http://www.alixaxel.com/wordpress/2007/05/19/php-math-library.

[6] Web Scraper Lite.
http://www.velocityscape.com/products/webscraperlite.aspx.

[7] Gonzalo Navarro, Erkki Sutinen, Jani Tanninen, and Jorma Tarhio. Indexing text with approximate q-grams. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 350–363, London, UK, 2000. Springer-Verlag.

[8] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[9] Lawrence Philips. The double metaphone search algorithm. *C/C++ Users J.*, 18(6):38–43, 2000.

[10] Vijay V. Raghavan and S. K. M. Wong. A critical analysis of vector space model for information retrieval. *Journal of the American Society for Information Science*, 37(5):279–287, 1986.

[11] Soo Young Rieh and Hong iris Xie. Patterns and sequences of multiple query reformulations in web searching: A preliminary study. In *Proceedings of ASIS&T Annual Meeting, Washington DC, Nov 2001*, pages 246–255, 2001.

[12] Eric Sven Ristad and Peter N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[13] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[14] Gerard Salton and Christopher Buckley. Term-weighting approaches in

automatic text retrieval. In *Information Processing and Management*, pages 513–523, 1988.

[15] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management*, pages 513–523, 1988.

[16] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.

[17] Daniel Yacob. Application of the double metaphone algorithm to amharic orthography, 2004.

[18] Yellowpages.ca.
http://www.yellowpages.ca.