

Predictive text input engine for Indic scripts

Mitch Selander and Erik Svensson

March 5, 2009

Abstract

Languages with many letters pose a problem for text entry on reduced keyboards. Using multitap is time consuming as there can be 6-9 characters per key on a mobile phone. For singletap methods more letters per key results in more words per key sequence, i.e. greater ambiguity when selecting which word to present to the user. Today's singletap methods for mobile phones mostly rely on a dictionary and word frequencies, this works remarkably well with the Latin alphabet. But this is not enough when the number of letters per key increases.

In this master thesis we investigated different methods to improve the word disambiguation. These methods include word bigrams, part of speech n-grams and keypad remappings. We have chosen the Devanagari script for our implementation as it is one of the scripts with this problem. We have worked with Hindi for the language specific data.

We found that a dictionary based solution with word bigrams combined with a remapped keypad layout gave the desired results. The use of these techniques gave an increase in disambiguation accuracy, from 77% to 94%. We also saw an improvement in KSPC, from 1.0856 to 1.0154.

Unfortunately, we could not find an annotated corpus good enough for a part of speech based solution to be implemented, as we think it would improve the system further.

Acknowledgements

We would like to thank our supervisor Pierre Nugues for all the help and useful comments on our work along the way. We would also like to thank Mobile Labs Sweden AB for giving us the opportunity to do this Masters thesis at their company. We are very grateful to Petrus Vavamis for making the mobile phone graphics used in this thesis.

Contents

1	Introduction	9
1.1	Introduction	9
1.2	Design goals	10
2	Background	11
2.1	Devanagari	11
2.2	Hindi	12
2.3	Previous work	13
3	Design	15
3.1	Dictionary	16
3.2	Corpus	16
3.3	Trie	17
3.4	Language model	18
3.4.1	n-gram	18
3.4.2	Parts of speech	19
3.5	Compound letters	20
3.6	Keypad layout	21
3.7	Spelling variations	23
3.8	The algorithm	23
4	Test setup	25
4.1	Metrics	25
4.1.1	Keystrokes per character	25
4.1.2	Accuracy	26
4.1.3	Node Sizes	26
4.2	Test framework	27
4.2.1	4-button keypad for English	27
4.2.2	Devanagari keypad for Hindi	27
5	Results	31
5.1	Simulations for the 4-button English keypad	31
5.2	Simulations for the Devanagari keypads	32
5.2.1	Dictionary based disambiguation with bigrams	32
5.2.2	Compound letters	32
5.2.3	Keypad layout	33

6	Conclusions	37
6.1	Results	37
6.2	Other languages and scripts	37
6.3	Implementing for a mobile phone	38
6.4	Future work	38
A	The Devanagari alphabet	43
A.1	Vowels	43
A.2	Consonants	43
B	Illformatted files from the EMILLE corpus	45
C	Example keypad layout	47

List of Figures

1.1	Standard keypad for a mobile phone	9
3.1	The predictive engine and the system	15
3.2	Example of a trie	17
3.3	A trie including compound letters	20
4.1	The 4-button GUI for English	27
4.2	The Hindi GUI	28
4.3	The Hindi GUI after shift has been pressed.	29
5.1	The node sizes of KP1	34
5.2	The node sizes of KP2	35
5.3	The node sizes of KP3	35
C.1	An example keypad layout	47

List of Tables

3.1	The multimap keypad, developed by Mobile Labs Sweden AB . . .	21
3.2	The second keypad, with vowels on separate keys	21
3.3	Third keypad, with only 8 buttons	22

Chapter 1

Introduction

1.1 Introduction

The use of SMS text messages has increased a lot the last years. Traditionally, text has been entered with multitap methods and more recently singletap methods like T9 and eZiText.

In multitap, pressing a key will cycle through the characters assigned to it. So, pressing the 2-key once will give you A, twice B and three times C, and then the cycle will start over. This is easy to implement and easy to use. However, typing is slow with this method, especially for languages with large alphabets.

With singletap, you only press each button once, so if you want to write *and* you press 2-6-3 with the standard Latin keypad layout. For this to work, we need a dictionary, where we match key sequences to words. However, a key sequence can match several different words. This means that there will be a level of ambiguity that has to be resolved. The key sequence 2-6-3 would for example also match the word *cod*, in addition to *and*. A simple method is to show the most common word first, and then sort the words in descending order.

These methods have worked well for languages using the relatively small Latin alphabet. However, Indic languages have larger alphabets, this means there will be more characters per button. This makes multitap even more tedious and it increases the ambiguity in singletap. Singletap methods normally only look at the currently entered key sequence and proposes the most probable word for that sequence. Another approach would be to also look at the previous word and take that into account when choosing the most probable word.

In the year of 2007, there were 165 million mobile phone users in India (Telecom Regulatory of India, 2007). Given the economic growth of India in the last years, there is reason to believe that this figure is even greater today. A lot



1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz
*	0	#

Figure 1.1: Standard keypad for a mobile phone

of the text messages sent in India now are either written in English or Hindi written phonetically with the Latin alphabet. Given the continuing growth of the mobile phone market in India, there is a great need for better text entering methods for the native scripts.

1.2 Design goals

The first goal of this thesis is to design an text entry system. The core of the system should be generic in the sense that it should be able to handle any written language when given access to language specific data.

The second goal is that the effort, number of keystrokes, required by the user should be noticeably reduced without drastically increasing the time to learn how to use the system.

And finally, the system should be able to run on a mobile phone. This means that there are restrictions on the size of the database but also on the computational complexity of the algorithms used.

Chapter 2

Background

There are two scripts of interest in India that have large alphabets. They are the Devanagari and the Bengali alphabets. They are both used to write several languages each in the region. There is one more script in the region that could have been of interest, Tamil, but it has fewer letters and it is only used to write the Tamil language with some 70 million speakers. It is by no means a small language but compared to Hindi or Bengali, which uses the Devanagari and the Bengali scripts respectively, it is considerably smaller. We have chosen Hindi since it uses the largest alphabet.

2.1 Devanagari

The Devanagari script was originally developed in the 11th century AD for writing Sanskrit. But as the Latin alphabet, it has been adapted for writing several modern languages. These include Hindi, with more than 700 million fluent speakers, Marathi, with about 90 million speakers, and Nepali with 30 million speakers. In addition to them, there are many languages with 5-25 million speakers that uses the Devanagari alphabet.

Devanagari is an abugida script. This means that the letters represents syllables. This is accomplished by letting every consonant inherit an अ (a) vowel. For example, the letter प is pronounced pa and the letter क is pronounced ka. If you want to write the syllable ki, you use the letter क (ka) and simply add the dependent form of the vowel you want to write, क (ka) + ि (i) is कि (ki), “that”.

Then the question arises, what if you want only the consonant sound without the inherited vowel? This means that there is need to suppress the inherited vowel. This is done by adding a ् (virama, or halant in Hindi) to the syllable. To write a single ‘k’, the character क (ka) is used and a halant is added.

क (ka) + ् (halant) → क् (k).

One reason for the existence of many letters is that vowels exist in both independent and dependent forms. The independent forms are generally used in the beginning of words, and the dependent forms are used when a consonant has had its inherited vowel replaced and the new vowel is attached to it.

For example:

हिन्दी (hindi), ``Hindi".

Here we can see both of the dependent forms, ि and ी, of इ (i).

The letters in the Devanagari alphabet are ordered with the letters formed in the back of the mouth first followed by the ones formed in the middle of the mouth. After that there are the ones formed with the tongue against the teeth and lastly the ones formed by the lips. Without this knowledge about the logic behind in the structure of the Devanagari script, it is very hard to see that it is in fact very well ordered.

In addition to the standard letters of the Devanagari and Bengali alphabets, there are conjuncts of letters, compound letters or ligatures. These are special characters for commonly occurring consonant sequences. The use of halants allows for sequences of consonants without their inherited vowel to be formed:

क (ka) + ः (halant) + स (sa) → क्स (ksa).

As can be seen, the halants are there logically but are excluded in the graphic representation. In fact, the graphical appearance of the conjunct is quite different from the characters it consists of.

There is a great number of these conjuncts, but only the most common ones are widely used. However, the most commonly used conjuncts in any given language have to be taken into account when designing the system. Any user friendly text entering system, has to support most important compound letters.

There is also a set of characters that can be added to letters to alter their pronunciation: ्र, ्रः, ्रः, ्रः, ्रः. For instance, the diacritic ं (anusvara) is used to indicate nasalization of a syllable. To be able to accurately enter text in any language using the Devanagari alphabet, these diacritics have to be supported.

2.2 Hindi

We have to choose a language for our implementation since the system will need language specific data in the form of at least a dictionary. We have chosen to do our implementation for the largest language using the Devanagari alphabet. This is without a doubt Hindi. It is the native language of 600 – 700 million people and the largest languages in India. It is also one of the two official languages of communication in India, the other being English. For an introduction to Hindi see Dasgupta (2001).

As India is a rapidly growing mobile phone market, the need for an efficient text entering method using the Devanagari alphabet for Hindi is great. With seven or eight letters on every key, the multitap method is demanding. A word with five letters can easily take 15 – 25 keystrokes to enter.

There is one language specific addition to spelling in Hindi. There are, as in most languages, plenty of loanwords in Hindi and there is a need to write them with the Devanagari script. But if the words contain syllables that do not exist in traditional Hindi, there will be a problem in reading and writing the word. The way this is solved is by adding a ः (nukta) to alter a syllables pronunciation. For example:

क (ka) + ः (nukta) → कः (qa)

The full list is:

क (ka) + ◉	क़ (qa)
ख (kha) + ◉	ख़ (khha)
ग (ga) + ◉	ग़ (ghha)
ज (ja) + ◉	ज़ (za)
ड (dda) + ◉	ड़ (dddha)
ढ (ddha) + ◉	ढ़ (rha)
फ (pha) + ◉	फ़ (fa)
य (ya) + ◉	य़ (yya)

These altered syllables will be added as standalone syllables to the keypad.

2.3 Previous work

One of the first major improvements to dictionary based text entry methods, was the use of word bigram statistics (Hasselgren et al., 2003). This makes use of the fact that different word sequences occur at different rates. These statistics are then used as support to the frequencies in the dictionary.

The text entry problem is not limited to devices with fewer buttons than letters. Even though there is no ambiguity in the key pressed when there is only one letter per key there can still be ambiguity in what the user is trying to write. One example is word completion, where the system tries to autocomplete the word the user is writing. Another is grammar checking and spell checking. The techniques used to solve this ambiguity is for instance part of speech tagging, see for example Fazly and Hirst (2003), and semantic relatedness, see for example Gong and Tarasewich (2005) and Li and Hirst (2005). As mobile phones get faster CPUs and more memory, these methods of disambiguation become more relevant in that context.

Most of the research done on reduced keyboards has been designed for the Latin alphabet. There are commercial implementations for both the Devanagari and the Bengali scripts, provided by Tegic and Zi Corporation for example. However, today the singletap solutions for Hindi are not widely used. The problem of entering text written with the Devanagari script is instead worked around by either using a multitap solution or by entering the text phonetically with the Latin alphabet (Gupta, 2006).

Chapter 3

Design

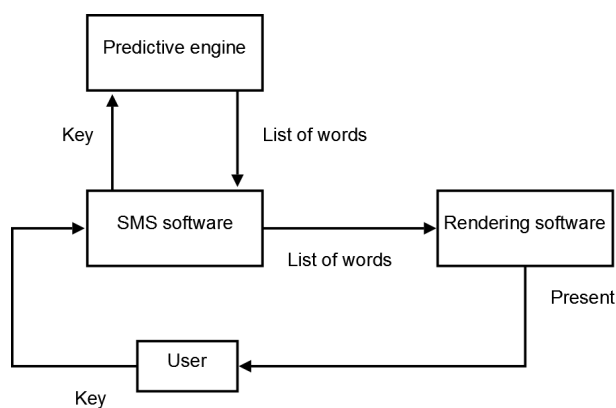


Figure 3.1: The predictive engine and the system

We will develop a text input engine that is aimed at being a module, that is part of a text entering application (not necessarily SMS). When the user presses a key, this key is sent to the predictive module. The key is concatenated to the previously entered keys (if there are any), the predictive engine then tries to disambiguate the whole key sequence and produces a list of matching words. The application then sends this list to the rendering software.

As mentioned earlier, vowels exist in both independent and dependent forms. For ‘i’ these are: इ, ई, ि, ि (i). As can be seen, the last of the dependent forms is attached to the left of the consonant which it depends upon. For instance in the word सिक्का (sikka), “coin”, we have both a dependent ‘i’ attached to the left of its consonant and a dependent ‘a’ attached at the end. When a word like सिक्का is handled within the system, it is viewed as a logical string of Unicode (Unicode Consortium, 2007) characters. This means that for all intended purposes, we treat the word सिक्का like this:

स + ि + क + ं + क + ा (sa + i + ka + halant + ka + a).

We can see that there is also a halant in the logical string that is not shown after the rendering phase as it is part of a compound letter. The words are in this form when sent to the rendering software.

We also mentioned earlier that a ङ (nukta) is added to consonants to be able to write some loanwords. In Unicode, this can be represented in two ways, either by using the consonant and then nukta, for example U+0915 + U+093C, but it can also be represented with the code U+0958. A commercial application should be able to handle both, and theoretically we do. However, if the same word is encoded in two different ways our application will see them as two different words.

3.1 Dictionary

Since most of the letter permutations that result from a key sequence on a reduced keyboard are not actual words, the first step has to identify the letter permutations that are words. By keeping a list of words, the task becomes easy. This list is called a dictionary, or lexicon, and is a set of words from either a complete language or a subset of one. The dictionary is the foundation of most word disambiguation techniques as it is the way to keep track of which words match a given key sequence.

The dictionary is created by going through a collection of texts, a corpus, and extracting the different words from it and counting their occurrences. We then save all the words with occurrences above a set threshold. We try to set this threshold so that we capture uncommon words but avoid common misspellings.

One can not stress enough the importance of a well-designed dictionary. There should not be any misspellings for instance, and there has to be a balance between capturing as many words as possible and keeping the size of the database down.

Because of the pruning of uncommon words, there will be words in the language that are not included in the dictionary. In a commercial implementation, there has to be functionality that enables the user to insert out-of-vocabulary words (OOV words) into the dictionary. Normally, this will be done with multitap.

3.2 Corpus

A corpus is a collection of texts. The texts may have been collected from various sources such as newspapers, the Internet, letters, etc. The corpus is an integral part of this kind of system as we gather the words and bigrams and their frequencies from there. The source of the text files in the corpus determines to a large degree how the system behaves. Since our system is intended to be used for entering text for SMSes, a corpus consisting of SMSes would have been preferred, however we could not find one publicly available for Hindi. There is however one in English from the Department of Computer Science at the National University of Singapore, called NUS (How and Kan, 2005). The corpus is quite small, around 10 000 messages. There is also a corpus available for SMS written in French (Fairon and Paumier, 2006). In the latter corpus, they have decided to normalize the SMS language to ordinary French, since the spelling used in SMSes greatly vary between users.

We used three corpora during our work on this thesis. For the initial de-

velopment, we used an English corpus called WaCky¹, which contained texts gathered from .uk websites. During the main development, we used a corpus from Indian Institute of Technology Bombay². Later during the training and testing, we used the EMILLE corpus (Baker et al., 2002) from Lancaster University, UK, and the Central Institute of Indian Languages (CIIL), Mysore, India. This corpus is multilingual and contains texts from 14 South Asian languages: Assamese, Bengali, Gujarati, Hindi, Kannada, Kashmiri, Malayalam, Marathi, Oriya, Punjabi, Sinhala, Tamil, Telegu and Urdu. Some languages have both texts from written and spoken sources. The Urdu part is also annotated with part-of-speech tags.

During the work on our thesis, we found that some files in the EMILLE corpus were badly encoded. These files were removed from the corpus. We have listed them in Appendix B.

3.3 Trie

When searching for the matching words of a given key sequence, we use a trie structure (Fredkin, 1960). But instead of branching on characters, we branch on the keys, so that each node in the trie represents an unique key sequence, see Figure 3.2. The list of matching words is then stored within this node. This list is sorted with the most probable word first.

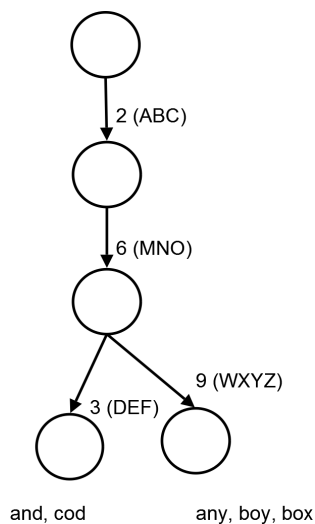


Figure 3.2: Example of a trie

¹Downloadable from: <http://wacky.sslmit.unibo.it/doku.php?id=start>

²Downloadable from: <http://www.cfilt.iitb.ac.in/>

3.4 Language model

3.4.1 n-gram

Simply put, an n -gram is a sequence of n items, in our case words. In this thesis, we only use unigrams and bigrams (1- and 2-grams). These unigrams and bigrams are extracted from our corpus. We also count their occurrences.

The unigrams make up our dictionary along with their probabilities. We use normalized probabilities, i.e. the sum of the probabilities of all the matching words for a given key sequence is equal to 1. If a word w matches a key sequence ks_i , we say $w \in match(ks_i)$. $match(ks_i)$ is the set of all words that match the keystroke ks_i .

$$\sum_{w \in match(ks_i)} p(w) = 1 \quad (3.1)$$

This means, that the probability of a unigram determines how likely it is to be the users intended word.

In the basic singletap method when the user presses a key, we traverse the trie to the matching node. From the node, we get the list of the matching words which we present to the user. The user can then select the intended word.

We can extend this by also looking at the previous word. This is where we introduce the bigrams. First we will show how we extract the bigrams. Consider the following sentence:

A simple example.

The extracted bigrams are then: (a, simple) and (simple, example). In our application bigrams are only extracted within sentences, i.e. the last word of a sentence and the first word of the following sentence will not be counted as a bigram. If \emptyset denotes the start of a sentence, we could also save (\emptyset, a) to indicate that a can be at the start of a sentence. This can be used to further improve a systems accuracy. However, we do not use this in our application, mainly to limit the number of bigrams.

The bigram probabilities are used along with the unigram probabilities in the disambiguation process. Once the user has pressed a key, a list of matching words is generated. Since there may and may not be any defined bigrams for these words we need to have some sort of backoff method. In our algorithm we use a very simple method, similar to Katz backoff model (Katz, 1987), where we use the unigram probabilities if no bigram were found. The weight of a word w_i is then defined as

$$w(w_i) = p(w_{i-1}, w_i), \text{ if there is a matching bigram} \quad (3.2)$$

$$w(w_i) = a \cdot p(w_i), \text{ otherwise.} \quad (3.3)$$

Where $a < 1$ and w_{i-1} is the previously entered word. Note that the weights we calculated with this method are not real probabilities since their sum may be larger or smaller than 1.

Since the list of bigrams could get huge, we need ways to limit it. First we only allow bigrams, where both words are available in the systems dictionary.

Then we set a limit for the minimum number of occurrences for the bigrams. To further reduce the number of bigrams, we removed the bigrams whose second word are unique for its key sequence. As it is the second word we're trying to disambiguate, if it is the only word for its key sequence, there is no need for the bigram. This could also be extended to words that share their key sequence with one or two, or even more words, depending on the intended systems limitations. Another idea is to remove bigrams that do not change the order of the proposed words.

3.4.2 Parts of speech

A part of speech (POS), or a lexical category, is a set of words that behave in the same way syntactically. Traditionally, English is said to have eight parts of speech. They are verb, noun, adjective, adverb, pronoun, preposition, conjunction and interjection. However, it is not uncommon to say that English have nine or ten parts of speech. For instance, determiners are often considered as their own part of speech instead of being grouped together with the adjectives. Verbs can be seen divided into the lexical verbs and the auxiliary verbs. In fact, when a corpus is annotated with information about a words parts of speech the word classes are much further divided. If the corpus lack the parts of speech annotation there are programs that can do this.

The idea is that instead of looking at the order of words we look at the order of the words parts of speech in sentences. We will then see that some sequences of parts of speech are more common than others. That means that in the same way as word n-grams can be used to disambiguate a word, the same can be done with n-grams for parts of speech. The idea is to use the POS tags to increase the accuracy of the disambiguation process by adding a method for checking the POS n-grams for the word that is being resolved. For an introduction on computational linguistics, including part of speech, see Nugues (2006).

The use of POS tags in word prediction has been proven to increase the accuracy by a small but significant amount (Fazly and Hirst, 2003). They also conclude that there is a great overlap between the word n-grams and the POS n-grams and that the word n-grams in this way covers up part of the POS n-grams contribution to the disambiguation process. We tried to utilize this by decreasing the size of the bigrams list, which uses a lot of memory, and let the POS n-grams, which do not strain the memory as much, make up for this lost information.

Unfortunately there is a drawback. As Fazly and Hirst (2003) showed, the use of POS tags also increased the load on the processor. The bigram method was, in their case, 6.5 times faster than the combined bigram and POS tags method. Of course, these numbers are highly dependent on the input data aswell as other external factors. But that there is a great increase in computational complexity cannot be ignored as this is supposed to be run on a mobile phone.

Since the Hindi parts of the EMILLE corpus are not POS tagged, we have to do this ourselves. To do this we need find another annotated corpus that is large enough with which to train a POS tagger with. This proved to be quite difficult for Hindi. This was to large of an obstacle for us, so we decided not to implement a POS based disambiguation method.

3.5 Compound letters

Compound letters, or ligatures, are special characters that are used in place of common letter sequences. These ligatures are often referred to as letters and are, when writing, seen as part of the alphabet even though they are conjuncts of two or more actual letters.

Every compound letter can be entered by entering the individual letters it consists of. Unfortunately this is not enough. The users need access to at least the most commonly occurring of these conjuncts. We have included the three most essential of these. They are: क्ष (kssa), ज्ञ (gyaa), त्र (tra).

Apart from taking up space in an already cramped keyboard, the compound letters pose another problem. If they share a key with ordinary letters, they will force the system to keep track of two different places in the trie, as illustrated in Figure 3.5.

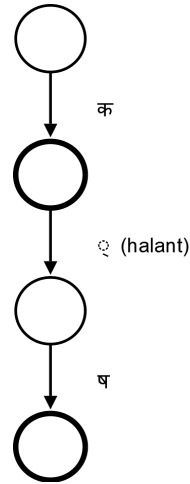


Figure 3.3: In this example, the user has pressed a key, which is shared by the compound letter क्ष (kssa) and क (ka). There is no way for the system to know if the system should move down one step, or three steps in the trie. So it has to keep track of two nodes, marked with bold outlining.

If, on the other hand, the compound letters are treated as an ordinary letter the user is forced to use the ligature and can not enter it by entering the letters it is comprised of. Our solution to this has been to use a shift key and let the three compound letters be on the shift-7, shift-8 and shift-9 keys. This allows the user to enter a compound letter by either the letters it is comprised of, or the whole compound.

This way of solving the problem with the ligatures gives us an opportunity to decrease the ambiguity when resolving the word entered by the user. Since the compound letter is unique for its key, there is no ambiguity in that part of the word. By checking all the words in a node, we can eliminate the ones that do not have the ligature in the right place and by that decrease the ambiguity.

3.6 Keypad layout

As mentioned earlier, the way the letters are distributed across the keypad effect the systems performance in multitap. But it can have an even greater effect on singletap solutions. Different layouts will distribute the words differently within the trie. Every layout will have its own unique trie structure. Some layouts will cause collisions between common words that other layouts will avoid. And some layouts may cause fewer collisions in all. But when looking at keypad layouts, there is another aspect that has to be considered: the usability. The system should be intuitive both for beginners and for users that are experts in other keypad layouts. This means that the speed that the user can enter the desired input has to be factored in.

The keypad layout we started with, was developed for multitap use. It had all of the vowels on the three first keys and then the consonants distributed over the rest of the keys, see Table 3.1. All of the diacritics were assigned to one key. This complies well with the order in the Devanagari script.

Button	Vowels	Consonants	Diacritics
2	अ, आ, ा, इ, ई, ि, िी		
3	उ, ऊ, उु, ू, ऋ, ृ		
4	ए, ऐ, ऐै, ओ, ौ, ो, औ, ौ		
5		क, ख, ग, घ, क़, ख़, ग़, घ़, ङ, च	
6		छ, ज, ज़, झ, ञ, ट, ठ, ड, ढ, ढ़	
7		ण, त, थ, द, ध, न, प, फ, फ़, ब	
8		भ, म, य, ञ, र, ल, व, श	
9		ष, स, ह	
0			्, ँ, ः, ॅ, ॊ

Table 3.1: The multitap keypad, developed by Mobile Labs Sweden AB

Button	Vowels	Consonants	Diacritics
2	अ, आ, ा	क, ख, ग, घ, ङ	
3	इ, ई, ि, िी	क़, ख़, ग़, च, छ	
4	उ, ऊ, उु, ू	ज, ज़, झ, ञ, ट	
5	ऋ, ृ	ठ, ड, ढ, ढ़	
6	ए, ऐ	ण, त, थ, द, ध	
7	ऐ, ऐै	न, प, फ, फ़, ब	
8	ओ, ौ, ो	भ, म, य, ञ, र, ल	
9	औ, ौ	व, श, ष, स, ह	
0			्, ँ, ः, ॅ, ॊ

Table 3.2: The second keypad, with vowels on separate keys

The first indication that this was not a well suited layout for a dictionary based solution, was that the letters अ, आ, ा (a) and इ, ई, ि, िी (i) that distinguish between the two genders in Hindi were placed on the same key. In Hindi there is no neuter gender, and it is common to generate animate nouns from a word

Button	Vowels	Consonants	Diacritics
2	अ, आ, ा	क, ख, ग, घ, ङ	
3	इ, ई, ि, िी	क, ख, ग, च, छ	
4	उ, ऊ, उ, ू	ज, झ, ञ, ज, ट	
5	ऋ, ृ	ठ, ड, ढ, ढ, ढ	
6	ए, े	ण, त, थ, द, ध	
7	ऐ, ै	न, प, फ, फ, ब	
8	ओ, ॉ, ो	भ, म, य, य, र, ल	
9	औ, ौ	व, श, ष, स, ह	्, ॅ, ॄ, ॅ, ॆ
0			

Table 3.3: Third keypad, with only 8 buttons

stem and then the ending for the right gender is added. For example, the Hindi words for boy and girl are made up of the word stem lark- and the inflections -a for masculine and -i for feminine. This gives us the words larka for boy and larki for girl.

Masculine	Feminine
लुडका (larka), "boy"	लुडकी (larki), "girl"
घोड़ा (ghora), "horse"	घोड़ी (ghori), "mare"

By having these letters on the same key the system could not resolve that ambiguity since it had to “guess” the inflection of the word. The inanimate nouns do not have this problem as they only have one fixed gender. This led us to the conclusion that we had to separate these two characters on the keypad as they caused a lot of collisions.

The simplest solution would be to just move either all the ‘a’s or all the ‘i’s to another key. However, when an alphabetically unconstrained approach to the layout of the keys is applied there are usability issues. Text entry with an unconstrained keypad has been shown to be significantly slower than a constrained keypad (Gong and Tarasewich, 2005). These results would probably be even worse with scripts with 60+ characters to search through. The only viable solution would be a keypad layout that is constrained by the order of the script.

We discovered that the Devanagari script was always presented as two separate sets of letters. The vowels were ordered in one set and the consonants ordered in another one. We decided to try to distribute the sets separately. By first distributing the vowels over all the keys and then do the same with the consonants we ended up with a new key layout, see Table 3.2. We kept the diacritics on a separate key.

We also tried a version using only 8 buttons to see how much this would affect the ambiguity. This keypad layout can be seen in Table 3.3. This is by no means an optimized layout, we used it merely to demonstrate the differences in ambiguity. The only difference to the previous keypad layout is that we moved the diacritics to another button.

3.7 Spelling variations

As mentioned earlier, both the Devanagari and the Bengali scripts are abugidas. This means that they are based on syllables instead of letters. The apparent gain of this way of organizing the written language is that spelling and reading becomes very simple. You just use the appropriate syllables to form your word. There are no special rules for different pronunciation of a letter depending on which letter it follows and so on. But there is a backside that maybe historically has not been of great importance but now has become a nuisance. That is, the fact that you write a word precisely as it is pronounced causes trouble with dialects. Words can have different spellings depending on which dialect of a language the writer speaks. This problem is much greater in Bengali than in for instance Hindi. One reason for this is negative impact of the printing media. Newspapers intentionally create their own rules for spelling. Another reason is that there is a lack of syllables to write many of the foreign words that has found their way into the language (Dash, 2005).

This creates problems for anyone who is interested in statistics of the language. The frequency of a word in the written language should reflect how common the word is and not how common a way of spelling it is. This means that all the different ways of spelling the word has to be identified and the sum of their frequencies has to be added up before their probabilities within their respective node is calculated. This is to allow at least the most common ways of spelling the words and at the same time reflect their frequency in the language. Care has to be taken that unusual, but “correct”, spellings of a common word does not take precedence over an almost as common word with a common spelling.

As this is not a great problem in Hindi, we have not taken any special care of spelling variations but this would be an area where a commercial implementation could work to improve the system.

3.8 The algorithm

This is the final algorithm we used in our system, written in pseudo code. `button` holds the button the user pressed and `alpha` is the backoff multiplier a in Equation 3.3.

```
if (currentNode.hasSubNode(button)) then
  resultList.clear()
  newNode = currentNode.subNode(button)
  for each (word in newNode.words)
    if (bigrams.exists(previousWord, word)) then
      newWord = word
      newWord.weight =
        bigrams.probability(previousWord, word)
      resultList.add(newWord)
    else
      newWord = word
      newWord.probability = word.probability * alpha
  sort(resultList)
  return resultList
```

```
else  
    return emptyList
```

Chapter 4

Test setup

4.1 Metrics

To know how well an application behaves, we need ways to measure the systems performance after different optimizations have been added. To do this, we have decided on a few metrics that will allow us to compare different versions of our system with each other as well as one that is well suited for comparisons with similar systems.

4.1.1 Keystrokes per character

KSPC (MacKenzie, 2002) is a common metric for text entry methods and is normally calculated as following:

$$KSPC = \frac{\sum_{w \in D} K_w \cdot F_w}{\sum_{w \in D} C_w \cdot F_w}, \quad (4.1)$$

where K_w is the number of keystrokes required to enter the word, w . If the word is not in the first place of the list presented to the user 1 is added to K_w for every step the user has to iterate through the list to reach the right word. F_w is the frequency of the word in the corpus and C_w is the number of characters in the word. Both K_w and C_w are adjusted to include spaces. This is calculated for every word w in the dictionary D .

A bit simplified, one could say that a system with great ambiguity will have a KSPC much greater than 1. The less ambiguity there is the closer we will come to 1. The only way to get below 1 is if the system also predicts words that are longer than the currently entered key sequence.

Since our system uses bigrams, we can not simply calculate the KSPC for the words in our dictionary. We need to do this with actual sentences (or at least actual bigrams). This is where the test corpus comes in. Our simulation program will get sentences from the test corpus and then enter each word separately in their intended order, as the user would. We will then calculate KSPC like this:

$$KSPC = \frac{\sum_{w \in T} K_w}{\sum_{w \in T} C_w}. \quad (4.2)$$

Where T is the test corpus. As a single word may occur several times in the test corpus we would then sum its K_w and C_w several times. So there's no need to multiply with F_w as it is in the formula implicitly.

Once we have the KSPC, it is easy to calculate the overhead per keystroke. The following formula gives us the overhead in percent.

$$\text{Overhead} = (KSPC - 1) \cdot 100. \quad (4.3)$$

So, a KSPC of 1.017 gives us an overhead of $(1.017 - 1) \cdot 100 = 1.7\%$ per character. This means we have to press an additional key every $1/0.017 \approx 59$ keystroke.

4.1.2 Accuracy

Another way to test how our application behaves is to see where in the list of proposed words the sought after word is placed, i.e. the systems accuracy. We will mainly look at the percentage of words that are placed first, in the top 3 and the top 5.

Another interesting statistic is how often a word in the test corpus is not in our dictionary. It is very important that a system is able to handle so called out-of-vocabulary, or OOV, words. However, as our system is completely based on a dictionary, once a word is not in our dictionary the user will have to enter it manually with multitap. We have not implemented this part of the system. In a commercial implementation, the manually entered word would be added to the dictionary to simplify future uses of it.

4.1.3 Node Sizes

We have used two different metrics to try to capture the systems inherent ability to handle the ambiguity without any help from frequencies, probabilities or bigrams. As mentioned earlier, every keypad layout produces its own trie and every trie have different distributions of the words in it. These metrics try to help in deciding how good a specific keypad layout is in comparison to another.

Largest Node Size (LSN): The size of a node is equal to the number of words it contains. LSN is the size of the largest node in the trie. This metric is used as an indication of how badly the system can perform in worst case.

Natural Disambiguation Accuracy (NDA): This is the measurement that shows how much of the ambiguity the system handles by itself. For instance, if there is only one word in a node there is no need for either frequencies or bigrams as there is no ambiguity. We have looked at two different NDAs.

- NDA_1 : The percentage of the words that are in nodes of size one.
- NDA_3 : The percentage of the words that are in nodes with three words or less.

We have chosen to include NDA_3 as a metric, since it is common to present the user with the three top scoring words in a list and then allowing to scroll down in that list to see the remaining words in the node.

These metrics indicate how common and bad collisions between words are in the trie that a specific keypad layout produces. Combined they give a fairly good

picture of how well a system will perform when the different disambiguation methods are added.

4.2 Test framework

During development and testing, we used Java, as we are most comfortable with it. The large amount of standard classes for different purposes were also a factor when we chose the language. For text processing, i.e. collecting data from the corpus, we used Perl.

4.2.1 4-button keypad for English

Since neither of us speak Hindi, we decided to start with an implementation for English. As we have already discussed, Devanagari is a considerably larger alphabet than the Latin alphabet. To approximate the increased ambiguity created by more characters per key, we developed a 4 button version of the mobile phone keypad for English. This gave us 6-7 characters per key, to compare with the Devanagari keypad with 7-8 characters per key. The keypad can be seen in Figure 4.1.

We used the WaCky corpus and set the limit of 2500 occurrences for a word to be added to the dictionary. Every word bigram, with both its words in the dictionary, with more than 50 occurrences was added to the bigrams list.

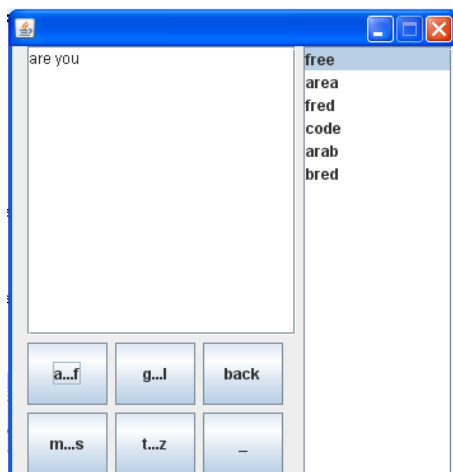


Figure 4.1: The 4-button GUI for English

4.2.2 Devanagari keypad for Hindi

Once the English version was working, we made it more generic to support Hindi as well. This was not that hard since both Perl and Java support Unicode. Apart from reworking the disambiguation engine, we also developed a prototype for the Hindi version, see Figure 4.2. During the simulations we tested different keypad layouts. However, we only developed a GUI for one of them.



Figure 4.2: The Hindi GUI



Figure 4.3: The Hindi GUI after shift has been pressed. The compound letters can be seen in the fourth row from the top.

Chapter 5

Results

5.1 Simulations for the 4-button English keypad

To simulate the problems of distributing a large alphabet over the keys of a mobile phone, we started to reduce the Latin keyboard even further. We made an alphabetically constrained layout, mapping the Latin alphabet to four keys. We did the simplest mapping possible and distributed the characters evenly over the keys, see Figure 4.1. We compared our results to those of Gong et al. (2008).

Our simulations were done with the WaCky corpus, where about 90% of the corpus was used for the training of the system and the remaining 10% for the testing. The test corpus contained a total of 67719092 words. Our dictionary contained 26172 words and we kept a list of 635559 bigrams. In retrospect, we should have done more pruning.

The backoff multiplier, a , is set to 0.3 in the following simulations.

Number of keys	3 keys	4 keys	5 keys
Button 1	ABCDEFG	ABCDEF	ABCD
Button 2	HJKLMNO	GHIJKL	EFGHIJ
Button 3	PQRSTUVWXYZ	MNOPQRS	KLMNO
Button 4	-	TUVWXYZ	PQRS
Button 5	-	-	TUVWXYZ
KSPC	1.2124	1.0746	1.0449
DA	67.58%	80.92%	87.46%

The results from the 3 and 5 button are from Gong et al. (2008). In the table, we only compared the frequency based disambiguation. They also tested a context based disambiguation method, however it is not entirely comparable to the method we used.

The following table contains results from our tests using bigrams. As we discussed in Section 4.1.2, we looked at three different ranges, whether the intended word is in the first position, among the first three or among the first five. We call them DA_1 , DA_3 and DA_5 respectively.

	Without bigrams	With bigrams
KSPC	1.0746	1.0499
DA_1	80.92%	87.92%
DA_3	95.92%	97.38%
DA_5	98.34%	98.80%

What we can see here is that the main advantage with the bigrams is the reordering of the first three words.

5.2 Simulations for the Devanagari keypads

For the simulations for the Hindi system, we used the EMILLE corpus where about 90% of the corpus was used for the training of the system and 10% for the tests. The tests were run on three different keypad layouts. The test corpus contained 673570 words. In addition to this the corpus contained 22745 sentences with words not in our dictionary (OOV words). These were discarded before testing. We had 28310 words in our dictionary and a list of 223696 bigrams.

5.2.1 Dictionary based disambiguation with bigrams

To establish a baseline, to which we could compare our future results, we started with a simple dictionary based approach. Initially we used the keypad seen in Table 3.1, which is mainly designed for multitap use.

The backoff multiplier, a , is set to 0.3 in the following simulations.

	Only dictionary	With bigrams
KSPC	1.0896	1.0491
DA_1	77.093%	86.815%
DA_3	95.784%	97.904%
DA_5	98.555%	99.145%

With only the dictionary and the words normalized probabilities, we get an overhead close to 9% per character. This comes down to about 5% with bigrams. One can easily see how it can become frustrating to write on a mobile phone when the word you are trying to write does not show up first in the list as often as 23% of the time. With bigrams, this figure comes down to 13% of the time which is a great improvement, but it is still much too high. The KSPC of the Devanagari keypad is comparable to the four button English keypad, but the gain from utilizing bigrams is greater with the Devanagari script.

5.2.2 Compound letters

We also ran simulations on the effects of adding support for compound letters. The compound letters we support are: क्ष (kssa), ग्या (gyaa), त्र (tra).

We tested the shift method. This will improve KSPC because the user has to press two keys (shift and then 7, 8 or 9 depending on which cluster the user wants), and the clusters all consists of 3 characters. But we're also interested in how this effects the disambiguation accuracy.

The simulation was done so that if a compound letter is found in a word, it is always used, instead of entering the characters separately which a real user might sometimes do.

Our simulations showed us that the test corpus contained 17751 words (2.64% of the words in the test corpus) with at least one of the three compound letters in them.

	Only dictionary	With bigrams
KSPC	1.0856	1.0451
DA_1	77.109%	86.817%
DA_3	95.785%	97.904%
DA_5	98.555%	99.145%

We can see that there is a small gain statistically from using compound letters, however adding support for compound letters is mostly a usability issue.

5.2.3 Keypad layout

Finally we tested the different keypad layouts we discussed in Section 3.6. We have named them KP1 (Table 3.1), KP2 (Table 3.2) and KP3 (Table 3.3) for simplicity.

As seen in the previous tests, the multitap keypad could only resolve the correct word 77% of the times with the use of a dictionary. We decided to look at the underlying problem. That is, how the words are distributed in the trie. To do this we looked at the Largest Node Size (LNS) and the two Natural Disambiguation Accuracy (NDA) metrics.

	LNS	55 words
KP1	NDA_1	45.726%
	NDA_3	69.823%
	LNS	25 words
KP2	NDA_1	59.389%
	NDA_3	82.932%
	LNS	25 words
KP3	NDA_1	56.401%
	NDA_3	80.932%

With these figures, it is easy to see that there is much less ambiguity with the second and third keypad than with the first. In the second keypad, KP2, almost 60% of the words are alone in their nodes, i.e. there is no ambiguity at all for their key sequences. This is a considerable improvement to the 46% that KP1 shows. The NDA_3 for KP2 shows that 83% of the words is placed in a list with at most three words.

We can also see that we can create an 8-button keypad without a great loss in NDA or increase in LNS. KP2 and KP3 also show less than half the size in the LNS, which means that there are a lot fewer collisions in the worst case scenario.

We have included the total distribution of the words over the nodes for all tree keypads, see Figures 5.1, 5.2 and 5.3. Here one can clearly see that KP1 has many more large nodes than KP2 and a lot less nodes of size one. The

consequence of this is that there will be more strain put on the disambiguation methods and that the cost for failure will be greater.

When we ran the final simulations, we utilized compound letters. In this case, we can see that the improvement of the keypad layout is as great as the improvement from the bigrams. The KSPC of KP2 with bigrams is 1.0154. This gives us an overhead of only 1.54%.

		Only dictionary	With bigrams
KP1	KSPC	1.0856	1.0451
	DA_1	77.109%	86.817%
	DA_3	95.785%	97.904%
	DA_5	98.555%	99.145%
KP2	KSPC	1.0350	1.0154
	DA_1	87.842%	94.256%
	DA_3	98.339%	99.101%
	DA_5	99.479%	99.698%
KP3	KSPC	1.0376	1.0171
	DA_1	87.206%	93.837%
	DA_3	98.210%	99.011%
	DA_5	99.426%	99.660%

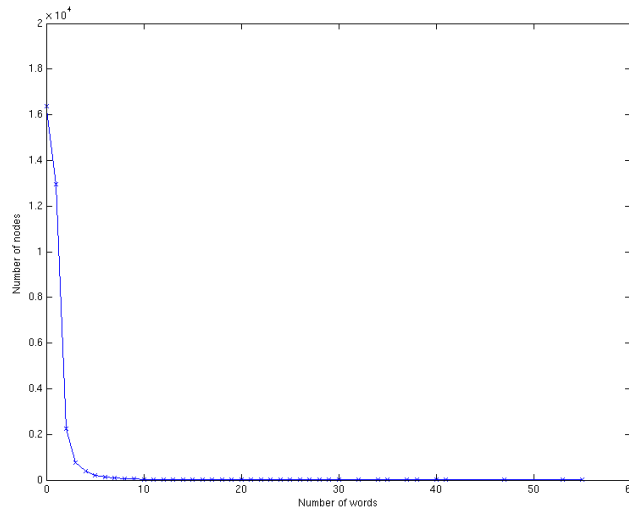


Figure 5.1: The node sizes of KP1

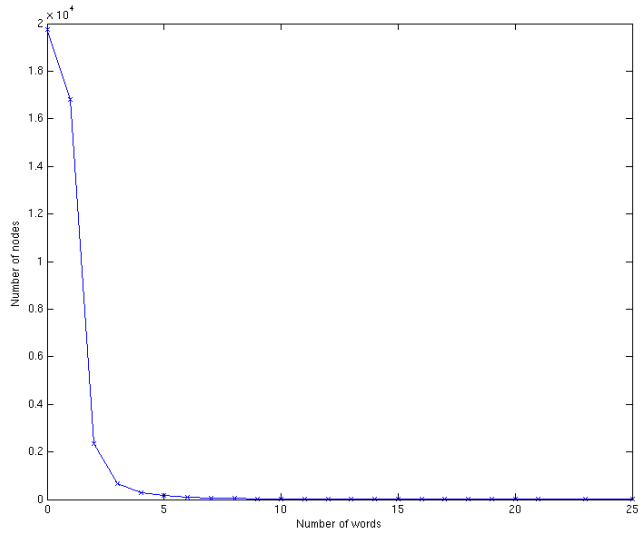


Figure 5.2: The node sizes of KP2

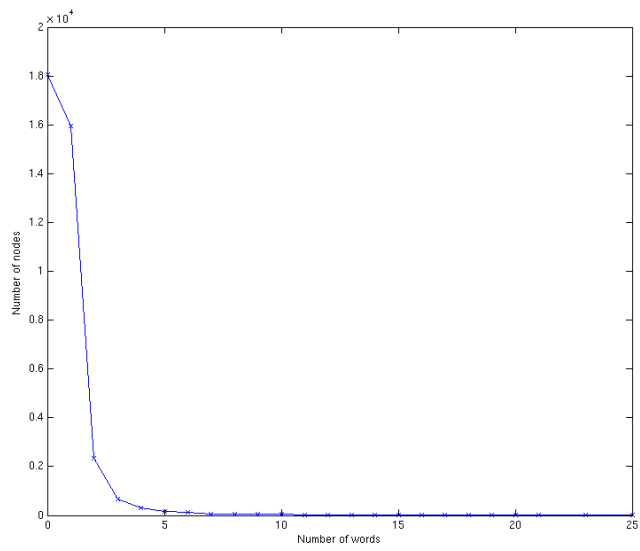


Figure 5.3: The node sizes of KP3

Chapter 6

Conclusions

6.1 Results

We have seen the impact bigrams as a language model has on the performance on the system. The disambiguation accuracy increased by 7-9 percentage points for the different keypads we tried. The layouts of the keypads gave just a big increase as bigrams did.

Our conclusion is that it is possible to implement an efficient predictive text input engine even with large alphabets. We have shown that even with simple techniques that can be used on low-to-medium cost mobile phones, one can come very far.

6.2 Other languages and scripts

Even though we have worked with Hindi and Hindi specific solutions have had a prominent place in this report. The main objective has been to find a good solution to entering text using the Devanagari script. As mentioned before, there are a plethora of languages that uses this script and our solution has to be generic enough to work with all of them given the language specific data. To add support for another language that uses the Devanagari script there is of course need for a dictionary and word bigrams in that language. But apart from that the system should not have to be adapted in any way. The EMILLE (Baker et al., 2002) corpus covers a lot of the South Asian languages and is a great starting point if one wants to implement our system for other languages using Devanagari.

The new keypad layout we propose is not optimized for Hindi as we have tried to conserve the idea and order in the Devanagari script rather than just make the “best” possible keypad layout for one particular language. As we have only looked at the effects of the remapped keypad on Hindi there is of course a risk that the proposed keypad layout does not work as well as intended with another language. However, our opinion is that it is always a good idea to distribute the vowels evenly over the keys. We first encountered this problem during the development of the four button English keypad. When we ran into the problems caused by the vowels in the traditional Hindi keypad layout we

got even more convinced that clustering vowels together poses a real problem and causes the system to have a lower NDA.

Our system should be quite easy to adapt to other scripts. We have already seen that it works very well with both the Latin alphabet and the Devanagari script with very little adaption. Apart from needing a dictionary and bigrams for every language that uses a script it is implemented for, an adequate keypad layout is needed. We have not had time to look into the preconditions for other interesting scripts. But problems like spelling variations in Bengali are only effected by the quality of the dictionary and the bigrams. Which in turn are only effected by the quality of the corpus. That is, it can only be solved by the data supplied to the system not by the system itself.

We did not test POS based disambiguation for Hindi since there we could not find an annotated corpus good enough for training a POS tagger. This problem is likely to be even greater for smaller languages.

6.3 Implementing for a mobile phone

We have focused our work entirely on methods that are well suited for the mobile phones of today. There are mobile phones with very large memories and fast CPUs, but we have tried to keep our focus on the performance of low to average cost mobile phones. By avoiding methods that require a lot of computational power or a lot of memory we arrived at the conclusion that a dictionary with normalized probabilities and word bigrams was the way to go. But even for these quite simple methods one encounters problems when moving from a desktop computer to a mobile phone. For instance, mobile phones are not well suited for floating point operations. This is of course a problem when working with probabilities. Switching over to integers of an appropriate size is most likely the best way to go. Another way would be to exclude the probabilities completely and having lists that are already sorted. This means that it all comes down to finding the right list of words from the bigrams and the list of words from the right node and combining them in some way before presenting them to the user.

The biggest problem with implementing for a mobile phone would be the bigrams list. The list for one language alone can be very large, and often a mobile phone needs to support several different languages. We have already discussed some ways of shrinking it, but they're not enough. Some form of compression needs to be applied.

6.4 Future work

The only tests we have done were simulated, it would be very interesting to do some user tests. The layout of our keypad seems logical to us, but we have no idea how a native Hindi speaker would respond to it.

It would also be interesting to do a more in-depth look at a better keypad. Both from the users perspective (with usability testing) and the systems perspective. There has been a lot of research on this for the Latin alphabet and English. However, we could not find any for Indic languages and their respective scripts.

Our method could also be adapted to complete words before the user has

pressed all the keys. However, the use of normalized probabilities for the words is not a good idea then, as we're looking in the whole trie instead of a single node. For a word completion system to work ideally it needs a lot of bigrams. This means it is likely that we cannot remove bigrams on the same criteria as we have done.

Another feature that could be added is spelling correction. This is quite easy thanks to the structure of a trie. For example, the user has pressed a key and the current node does not have a subnode. We could assume that the user has pressed the wrong key. We can then try other subnodes instead and present their words to the user. Normalized probabilities are probably not suited for a system implementing this kind of spelling correction.

Bibliography

- Baker, P., Hardie, A., McEnery, T., Cunningham, H., and Gaizauskas, R. (2002). EMILLE, a 67-million word corpus of indic languages: data collection, mark-up and harmonization. In *LREC 2002 Proceedings*, pages 819–827.
- Dasgupta, B. B. (2001). *Learn Hindi yourself*. A. Das Gupta.
- Dash, N. S. (2005). Methods in madness of Bengali spelling: A corpus-based investigation. *South Asian Language Review*, 15(2).
- Fairon, C. and Paumier, S. (2006). A translated corpus of 30,000 french SMS. In *Proceedings of LREC 2006, Genoa, Italy*.
- Fazly, A. and Hirst, G. (2003). Testing the efficacy of part-of-speech information in word completion. In *Proceedings of the Workshop on Language Modeling for Text Entry Methods, 11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 9–16.
- Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3(9):490–499.
- Gong, J. and Tarasewich, P. (2005). Testing predictive text entry methods with constrained keypad designs. In *Proceedings of Human Computer Interfaces International (HCII 05)*.
- Gong, J., Tarasewich, P., and MacKenzie, I. S. (2008). Improved word list ordering for text entry on ambiguous keyboards. In *Proceedings of the Fifth Nordic Conference on Human-Computer Interaction - NordiCHI 2008*.
- Gupta, R. (2006). Technology for Indic scripts: A user perspective. *Language in India*, 6(7).
- Hasselgren, J., Montnemery, E., Nugues, P., and Svensson, M. (2003). HMS: A predictive text entry method using bigrams. In *Proceedings of the Workshop on Language Modeling for Text Entry Methods*, pages 43–49, Budapest.
- How, Y. and Kan, M.-Y. (2005). Optimizing predictive text entry for short message service on mobile phones. In *Proceedings of Human Computer Interfaces International (HCII 05)*.
- Katz, S. M. (1987). Estimation of probabilities from sparse data for a language model component of a speech recognizer. *IEEE Transaction on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

- Li, J. and Hirst, G. (2005). Semantic knowledge in word completion. In *Proceedings of the 7th International ACM SIGACCESS Conference on Computers and Accessibility*, Baltimore.
- MacKenzie, I. S. (2002). KSPC (keystrokes per character) as a characteristic of text entry techniques. In *Proceedings of the Fourth International Symposium on Human-Computer Interaction with Mobile Devices*, pages 195–210.
- Nugues, P. (2006). *An Introduction to Language Processing with Perl and Prolog*. Springer.
- Telecom Regulatory of India (2007). Annual report 2006-2007.
- Unicode Consortium, T. (2007). *The Unicode Standard 5.0*. Addison Wesley.

Appendix A

The Devanagari alphabet

A.1 Vowels

Independent	dependent
अ (a)	-
आ (aa)	ा
इ (i)	ि
ई (ii)	ी
उ (u)	ु
ऊ (uu)	ू
ऋ (ri)	ृ
ॠ (rii)	ॄ
ए (e)	े
ऐ (ai)	ै
ओ (o)	ो
औ (au)	ौ

A.2 Consonants

क (ka)	ख (kha)	ग (ga)	घ (gha)	ङ (nga)	-	ह (ha)
च (ca)	छ (cha)	ज (ja)	झ (jha)	ञ (nya)	य (ya)	श (sha)
ट (ṭa)	ठ (ṭha)	ड (ḍa)	ढ (ḍha)	ण (ṇa)	र (ra)	ष (ṣa)
त (ta)	थ (tha)	द (da)	ध (dha)	न (na)	ल (la)	स (sa)
प (pa)	फ (pha)	ब (ba)	भ (bha)	म (ma)	व (va)	-

Appendix B

Illformatted files from the EMILLE corpus

hin-w-administration-lot13aa
hin-w-administration-lot13b
hin-w-literature-eductext-lot191
hin-w-literature-essay-lota11
hin-w-literature-essay-lotaa11
hin-w-literature-essay-lotb11
hin-w-literature-essay-lotbb11
hin-w-literature-essay-lotc11
hin-w-literature-essay-lotcc11
hin-w-literature-essay-lotd11
hin-w-literature-essay-lotdd11
hin-w-literature-essay-lote11
hin-w-literature-essay-lotee11
hin-w-literature-essay-lotf11
hin-w-literature-novel-lota12
hin-w-literature-novel-lotb12
hin-w-literature-novel-lotc12
hin-w-literature-novel-lotcc12
hin-w-literature-novel-lotd12
hin-w-literature-novel-lotdd12
hin-w-literature-novel-lote12
hin-w-literature-novel-lotee12
hin-w-literature-personal-lotaa12
hin-w-literature-personal-lotbb12
hin-w-media-lot13a
hin-w-media-lot19bb
hin-w-media-lot19bc
hin-w-socsci-ling-lot19bd

Appendix C

Example keypad layout



Figure C.1: An example keypad layout