Phonetic text input for Indic scripts

Magnus Höjer, D04
d04mhj@student.lth.se

Supervisor:
Pierre Nugues

Lunds Tekniska Högskola
December 12, 2008

Abstract

The complicated structure of Indic scripts means they are not very well suited for text input on a computer or, especially, a mobile phone. An alternative approach is to let users type text in romanized versions of their languages, and automatically convert, *transliterate*, this into the native script. In this thesis, we investigate models for transliteration utilising decision trees and LIBLINEAR, suitable for implementation on a mobile phone.

We found that the model was not quite flexible enough to handle all the spelling variations in the test set. Although it should be good enough for simple use in e.g. an SMS application. We also found that ultimately the LIBLINEAR implementation was superior to the decision tree, but that the difference was small enough that choice of model could be based mainly on computational grounds.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The complicated structure of Indic scripts means they are not very well suited for text input on a computer or mobile phone. Their alphabets are considerably larger than for western languages. Historically there has also been no good way of rendering Indic text in a portable way. Because of this lack of support for native scripts, users have taken to using romanized versions of their languages when communicating on the internet, etc.

Lately however, support for rendering Indic text has become much better. Modern operating systems are very capable at rendering the complex ligatures and other features required in Indic scripts. Mobile phone makers as well have begun adding support for Indic languages.

Input is still a problem though. The standard keyboard layout that is supported in modern operating systems is called Inscript [1]. There has been research in specialised keyboards and input methods for Indic languages (see e.g. Shanbhag et al., 2002; Rathod and Joshi, 2002; Krishna et al., 2005) but this is a clumsy solution. It requires separate keyboards for each language which means the same computer cannot be used by people speaking different languages. Another issue is that people will probably want a regular qwerty keyboard for tasks like communicating internationally, writing code etc.

Using on-screen soft keyboards is not optimal for most devices either. On touch-screen devices, it can probably work well, but on a regular PC, it would require the user to simultaneously focus on an on-screen keyboard as well as the physical keyboard. On a regular mobile phone with a small screen and physical keypad (the type users in these countries is most likely to have access to), it will not work at all. A better solution would be to take advantage of users familiarity of romanized text and let them input text as they are used to but automatically convert it to native script which is then displayed.

This act of automatically converting text is called *transliteration*: to tran-

[1] http://tdil.mit.gov.in/keyoverlay.htm

scribe text from one writing system to another. There are existing standards for transliteration of Indic scripts, like *ISO 15919*[2] and *IAST*[3] but these are mostly intended for transliterating from native script to the Roman alphabet. They use a rich set of diacritics to preserve the greater expressiveness of the native alphabets, which means they are poorly suited for input on a keyboard or especially a mobile phone keypad. Additionally, these standards are not something ordinary people know or use, they will simply write words as they pronounce them without any specific system. In that sense, the transliteration we attempt to do is somewhat similar to transcription actually.

As an example, Surana and Singh (2008) looked at the number of results returned when searching Google for different transliterations of the Hindi word नौकरी, *"job"*. The most common transliterations are shown in Table 1.1. This is the type of variation a good transliteration system should handle.

naukri	722,000
nokri	19,800
naukari	10,500
naukary	5,490
nokari	665
naukarii	133
naukaree	102

Table 1.1: Different transliterations of नौकरी.

1.1 Design Goals

We had two major design goals for this thesis. The first was that the model developed should work for not just one, but many Indic scripts. However, of the Indic scripts, the largest one is Devanagari which is used for Hindi and Marathi as well as numerous other smaller languages. For this reason all the work in the thesis as well as all examples in this report are in Hindi. Where applicable, there will be comments on how models and algorithms generalise to the other Indic languages and scripts.

The second goal was that the resulting software should work on a mobile phone, to be used for tasks like SMS or email editing. This places demands on the size of databases used as well as the computational complexity of algorithms chosen.

[2]http://en.wikipedia.org/wiki/ISO_15919
[3]http://en.wikipedia.org/wiki/IAST

Chapter 2

Background

2.1 Structure of Devanagari

Devanagari is an Abugida script. Abugida scripts are based on syllables, or more specifically *aksharas* (orthographic syllables, Singh, 2006). Their alphabets all follow the same basic structure and can be divided in three major groups: consonants, dependent vowel signs, and independent vowels.

Consonants all have an inherent *a* sound. So क, ध and श are pronounced *ka*, *dha* and *śa* respectively. The sound of consonants can be changed by adding a *nukta* (.), so for example ग turns into ग़ which changes the pronunciation from *ga* to *ġa*. To kill the inherent *a* sound and to enable consonants to form conjuncts a *halant* (्) is added. Depending on the consonants involved (and in some cases on whether you are writing modern Hindi or traditional Sanskrit), these conjuncts will either be rendered with half forms or as special conjuncts. Some examples can be seen in Table 2.1.

क + ् = क्	k
क + ् + ष = क्ष	kṣa
च + ् + छ = च्छ	ccha
ष + ् + ट + ् + र = ष्ट्र	ṣṭra

Table 2.1: Examples of Devanagari conjuncts.

Vowels come in two forms, either as independent vowels which form a syllable of their own (usually at the beginning of words, but sometimes in the middle as well) or as dependent vowel signs, *mātrās*, which attach to consonants. Examples can be seen in Table 2.2.

As we can see in these examples, written Devanagari is not a simple linear composition of glyphs but considerably more complex. However in the logical Unicode representation (which is what we will be working with), it is indeed a linear composition. The transformation to correct visual shapes

आ	ā	का	kā
इ	i	कि	ki
ऊ	ū	कू	kū

Table 2.2: Independent vowels on the left, mātrās attached to the consonant क on the right.

is only done when rendering text on-screen. E.g. in the visual representation of कि, the ि stands to the left of क, but in the logical representation the order is the phonetic one, क, ि.

Of the diacritical marks, nukta has already been mentioned. Other diacritics are *anusvara* (ं) and *candrabindu* (ँ), which indicate nasalization. In traditional Sanskrit, they were distinct but in modern Hindi they are pronounced similarly. Which one is used depends on if the syllable they are to be placed over has room (e.g. हैं – haiṅ and माताएँ – mātāyeṅ). Some example sentences combining all these features:

इस विद्यालयमें पाँच सौ लड़के पड़ते हैं. *is vidyālayamēṅ pañch sau laṛke paṛhte haiṅ.*	Five hundred boys read in this school.
मैं आप या स्वयं ही चला जाऊँगा. *maiṅ āp yā swayaṁ hī chalā jāūṅgā.*	I shall go away by myself.

As was said, this structure is shared by all the Indic scripts. Where they differ is in how many characters there are (Devanagari has 37 consonants while Tamil only has 23 for example) and how the individual characters look. For example, in Bengali and a few other scripts, there are dependent vowels made up of two parts, one to the left and one to the right of the consonant they attach to.

For a complete listing of the alphabet, please see the Unicode code chart for Devanagari[1].

2.2 Issues to consider

The basic grammatical structure of Devanagari as shown in the previous section is simple enough, but there are issues we have to take into consideration when attempting to transliterate from Roman Hindi to Devanagari. One is that the canonical structure, where consonants either have an explicit vowel mātrā, an implicit *a* vowel or a halant attached, is not always followed. The most common example is that the last consonant in a word, if it has no explicit vowel attached, is often pronounced without the *a*. E.g.

[1] http://www.unicode.org/charts/

कलम is pronounced as *kalam* and not *kalama*. Consonants can also be pronounced without *a* in the middle of words (e.g. इतना – *itnā*).

There are a few phonotactic rules applicable. E.g. in a 3 syllable word, if the last syllable has an explicit vowel and the middle one does not, then the middle one is pronounced with the implicit vowel silent. These cannot be relied upon 100% however. People will pronounce words slightly different and thus write them differently (Singh, 2006).

Spelling variation applies not only to Roman Hindi but to Devanagari as well. Therefore a successful transliteration system cannot be entirely dependent on a wordlist (though it can certainly utilise one as complement) but should be based on a more general model of the language to be able to handle Out Of Vocabulary (OOV) words.

Another issue regarding vocabulary is the widespread usage of foreign (mainly English) words and spelling. For example, the city हैदराबाद is commonly spelt in Roman Hindi as *Hyderabad* and not the more phonetically correct *Haidarabad*. Other examples are ऑफीसर, फ्लाइट and डीजल which are preferably spelt as *officer*, *flight* and *diesel* respectively, rather than *aafisar*, *flaait* and *diizal*. A good transliteration system should therefore analyse word origin and treat foreign words differently.

2.3 Previous work

2.3.1 Commercial implementations

There are some transliteration solutions already available. The oldest and most well-known is ITRANS[2]. It employs a rigid set of rules for transliteration, where users have to use both lower- and uppercase letters as well as punctuation to write text. This is then fed through a converter which emits Indic text. For example, the Hindi sentence

मैं अपने दोस्तके साथ आया हूँ

would in ITRANS have to be written as *"mai.n apane dostake saath aayaa huu.N"*, while a more natural way to write it in Roman Hindi would be *"main apne dostke sath aya hun"*. ITRANS is thus not very relevant today.

More relevant solutions are Google Indic Transliteration[3] and Quillpad[4]. These are modern systems that can cope with spelling variation and handle foreign words. The user can type freely and the software transliterates the text in real time. If the first word suggested is not correct another can be chosen similar to T9 text input on mobile phones.

[2]http://www.aczoom.com/itrans/
[3]http://www.google.com/transliterate/indic
[4]http://quillpad.in/

2.3.2 Research

Most transliteration research focuses on transliterating names, technical terms and other words not likely to be present in a dictionary for use in cross lingual information retrieval (CLIR) or machine translation (MT) systems. Yoon et al. (2007) developed a transliteration system for named entities from English to Arabic, Chinese, Hindi and Korean using a phonetic scoring method. Jung et al. (2000) examined English to Korean transliteration using an extended Markov window utilising context on both the English and the Korean side.

The text input angle has begun to get more attention in the last few years though. Surana and Singh (2008) specifically mentions input as a use for their model which utilises fuzzy string matching and their own phonetic model of scripts to match transliteration candidates against a corpus. They also use n-gram models to analyse word origin and are thus able to give foreign words special treatment (as mentioned in Section 2.2). UzZaman et al. (2006), attempt transliteration from Roman (English) to Bengali using a simple phonetic mapping scheme and a phonetic lexicon.

Chapter 3

Design

3.1 Granularity

Since a pure dictionary-based approach was ruled out, our transliteration model must segment words into smaller pieces. One option is to work on syllables. In Devanagari, segmentation into syllables is easy. For example हिन्दी consists of the two syllables हि and न्दी. A possible drawback is the large number of syllables possible (and present) in Hindi writing which leads to high dimensionality for the resulting model.

We have even more problems on the Roman side. In Roman Hindi, हिन्दी is typically written *hindi* which in this case should be segmented as *hi-ndi* (since the *n* means the consonant न here). However, as we have seen before *n* could also mean anusvara or candrabindu as in जाऊँगा – *jaunga* which should be segmented as *ja-un-ga*. So segmentation on the Roman side is considerably more complicated (Ekbal et al., 2006 has more examples on the type of difficulties encountered when doing transliteration to Bengali).

We therefore looked at individual characters instead. Our example word हिन्दी would now be segmented as ह - ि - न् - द - ी and the typical Roman transliteration as *h-i-n-d-i* which leads to a neat 1-to-1 mapping (if we disregard the halant, which will be dealt with separately later). However, another equally valid spelling is *hindee*, which means our 1-to-1 mapping did not hold up for long.

There are two options for dealing with this. The first is to introduce a "null character" (denoted as ϵ) which would let Roman characters, depending on context, map to nothing. For the above example we would get the alignment in Figure 3.1.

However, as we will later see in Section 3.3, context on the Roman side can be hard to come by which leads us to the second option: let variable length segments on the Roman side map to single characters on the Devanagari side. Unlike splitting into syllables, splitting into these smaller

ह	ि	न्	द	ी	ϵ
|	|	|	|	|	|
h	i	n	d	e	e

Figure 3.1: Example of alignment using ϵ to denote an empty mapping.

ह	ि	न्	द	ी
|	|	|	|	|
h	i	n	d	ee

Figure 3.2: Example of alignment using variable length Roman segments.

segments is considerably easier. Roman Hindi words will mostly, with just a few exceptions, have a unique segmentation. To continue with the example, the alignment would now look like Figure 3.2.

The exceptions are a few vowels and consonants where the Roman representation could be split into two. For example, *au* could mean औ as in सौ – *sau* or it could mean अ उ as in जाऊँगा – *jaunga*.

Another issue that needs to be noted is the letter *x*. People will typically use *x* when they mean a conjunct like क्ष – *kṣa*. Therefore, to keep the many-to-one mapping each instance of *x* in the Roman Hindi word must be replaced with *ks*.

3.2 Intermediate phonetic representation

In the previous section, we showed examples of Roman Hindi mapping directly to Devanagari. Another option is to first convert Roman words into a phonetic form, as is done in several articles (UzZaman et al., 2006; Chaudhuri, 2006; Yoon et al., 2007). However, since Hindi words are already spelt "phonetically" when written romanized, we saw little potential benefit in this. The extra conversion step required is rather a potential source of noise.

There is one potential use for a phonetic representation though, transliterating foreign words, which are not spelt phonetically. Unfortunately we did not have time to investigate this class of words.

3.3 Data resources

The next step is to acquire the raw data from which to build our model. The optimal source would be a pre-aligned corpus of Roman Hindi-Devanagari word pairs from which to learn spelling variations. Of course we are not so lucky, which leads to our second option: acquire separate Roman Hindi and Devanagari corpora and attempt to align them ourselves.

We used the Devanagari corpus prepared by the *Resource Center for Indian Language Technology Solutions*[1]. It contains $\sim$ 37 Mb of text with $\sim 119,000$ unique words. Another option is the EMILLE corpus (Baker et al., 2004). There is also a decent amount of blogs, news and other sources of Indic text on the Internet so constructing a crawler to collect your own corpus from the web should also be feasible.

On the Roman side, it is considerably harder however. We could not find any pre-assembled corpus so instead we tried constructing our own from online collections of Bollywood lyrics. The quality was not great though and initial experiments with automatic alignment against the Devanagari corpus yielded either too much noise or, if we lowered tolerances, no better data than manually defining mappings. Another concern was the fact that if obtaining data for Hindi was this hard, how difficult would it be for other, smaller languages?

Therefore we decided to only utilise context on the Devanagari side and manually define the mappings from Roman Hindi to Devanagari. Looking at transliteration schemes like ISO 15919 and ITRANS, as well as the Roman Hindi text we had collected from the web we defined for each Devanagari character the possible ways it can be written in Roman Hindi. Some examples can be seen in Table 3.1, for a complete listing see Appendix A. This list was checked by a native speaker to ensure most spelling variations were covered.

ई	i	ii	ee	yi
ी	i	yϕ	ii	ee
फ	ph	f		
ख	k	kh	khh	

Table 3.1: Examples of Devanagari-Roman Hindi mappings defined. ϕ denotes a word boundary.

Since all Abugida scripts share the same structure this type of mapping should be easy to develop for other Indic languages.

3.4 Extracting n-grams

We mainly used trigram models, although the LIBLINEAR implementation was also tested with bigrams. Therefore this section is written from the point of view of trigrams. Constructing bigram models was done in the same way, but with all n-grams one size smaller.

From the Devanagari corpus, we first extracted the words present and their frequencies. From this wordlist, we then extracted character uni-, bi-

[1] http://www.cfilt.iitb.ac.in/

हिन्दी → $\phi\phi$हिन्दी →

→ $<\phi,\phi,$ह$>$ $<\phi,$ह,ि$>$ <ह,ि,न> <ि,न,द> <न,द,ी>

Figure 3.3: Extracting character trigrams.

and trigrams. Halants were stripped out since they only decreased the precision. Having a halant in front does not help in deciding if *d* should map to द, ध, ड or ढ for example.

They could be useful in numeric mode (see Section 3.5.3), where the same key can map to both consonants and vowels, since halant then would imply a consonant. But for all the instances where we are not choosing between consonants and vowels the halants do not contribute any meaningful information and in general they lower precision here also.

For n-gram extraction $n-1$ word boundary markers, ϕ, are added to the beginning of each word (our model only looks at context to the left of the current character). See Figure 3.3 for an example.

One issue that needs special attention are the implicit *a* sounds present in Devanagari consonants. On the Roman Hindi side, these will be written explicitly as *a* but on the Devanagari side they do not have any Unicode representation, therefore we have to approximate their frequency of occurrence somehow. Our method for doing this is as follows:

We first extract character 4-grams $< c_1, c_2, c_3, c_4 >$ from the corpus *with halants*. From these we select all 4-grams where c_3 and c_4 are consonants. These can be viewed as 5-grams of the form $< c_1, c_2, c_3, \epsilon, c_4 >$ (where ϵ as before denotes a "null character").

Depending on whether c_2 is a halant or not we can then create trigrams $< c_1, c_3, \epsilon >$ or $< c_2, c_3, \epsilon >$ which can be used to estimate the occurrence of the implicit *a* sound. This method does overestimate the frequency some since as mentioned earlier not all *a*:s are pronounced (e.g. तुमने is pronounced *tumne*, not *tumane*) but it seems to work well enough in practice.

3.5 Machine learning algorithms

The mappings and n-grams can be combined and used to construct machine learning algorithms. In Section 3.3, we defined the possible ways each Devanagari character could be spelled. These can be reversed so we get a set of Roman tokens and for each token the possible Devanagari characters that token could be transliterated to, see Table 3.2 for an example.

For each Roman token and its list of associated Devanagari characters, we can select the subset of trigrams whose last character is in that list. This data can be used to train a machine learning algorithm implementing a function that given preceding Devanagari characters d_{i-2}, d_{i-1} as input re-

a	अ	आ	ा	
n	ं	़	ण	न
k	क	ख	क़	ख़

Table 3.2: Examples of Roman Hindi-Devanagari mappings.

turns a list of possible d_i with associated probabilities. For testing we implemented these functions with both decision trees (Quinlan, 1986) and logistic regression using LIBLINEAR (Fan et al., 2008).

3.5.1 Decision trees

We implemented a decision tree based on trigrams with fixed evaluation order, branching first on d_{i-1} and then d_{i-2}. Branching was done on every single Devanagari character present, so to handle the case where no matching branch can be found when classifying an instance interior nodes contain bigram based probabilities (and the root unigram based).

There are cases where it is unnecessary to branch. For example, *p* only maps to प no matter the context, *aa* can map to आ or ा, but at the start of a word only the independent form is valid. For this reason, we calculated the entropy of the data before branching. If it was 0 a leaf node could be put in place of the branch.

This could be generalised to abort branching not just for an entropy of 0 but for entropy below a set threshold for efficiency reasons. We did not attempt this, instead we used another approach. After branching is done each node has a set of Devanagari characters with attached probabilities. This set can be pruned by removing characters with low probabilities, e.g. if a node contained two choices, $d_1, p = 0.99$ and $d_2, p = 0.01$ this could be simplified to $d_1, p = 1.0$. Results for different levels of pruning can be seen in Section 5.1.

An example of part of a decision tree can be seen in Figure 3.4.

3.5.2 LIBLINEAR

With LIBLINEAR we used the L2-regularised logistic regression solver. Training data was encoded with one feature for each possible Devanagari character and position. The size of the part of the Devanagari code page we used is 101 (although there are a few holes). Therefore each training instance is a 202-dimensional vector where two elements have the value 1 (one in dimensions 1 – 101 and one in dimensions 2 – 202) and the rest are 0.

To determine the optimal C parameter 5-fold cross validation was employed in two phases. In the first phase $C = 2^k, k = -5, -3, ..., 5, 7$ was tried. This resulted in some optimal $C = 2^{k_{opt}}$. In the second phase $C =$

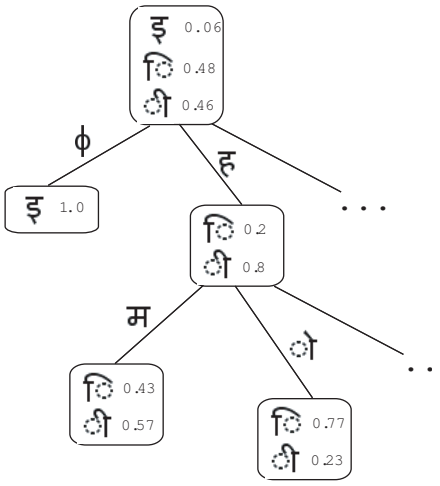

Figure 3.4: Part of the decision tree for *i*, branching first on d_{i-1} and then d_{i-2}.

$2^k, k = k_{opt} - 1, k_{opt} - 0.8, ..., k_{opt} + 0.8, k_{opt} + 1$ was then tried and the resulting optimal choice was used in subsequent training.

When classifying instances LIBLINEAR was run with the *-b* option to output probability estimates. Similarly to the decision tree above the returned result was pruned, removing low probability choices. Results are available along with the decision tree ones in Section 5.1.

The LIBLINEAR model was also tested with bigrams, employing the same model as above except instances were now 101-dimensional vectors instead. Results can be seen in Section 5.4.

3.5.3 Variation for mobile phones

The given model works well for regular keyboards, but on mobile phones it is somewhat cumbersome since it requires multi-tap. As an example, to type the word *hindi* on a mobile phone requires the key-presses 44, 444, 666, 3, 444. It would be better if users could type characters with just one key-press each the same way they do with T9 etc. So *hindi* would be typed as 44634 instead.

We can easily do this just by converting the mappings. So if ई previously had the mappings *i*, *ii*, *ee* and *yi* it will now have the mappings 4, 44, 33 and 94. We then construct our machine learning algorithms just like before.

This does of course increase the ambiguity since we have now reduced the number of Roman tokens defined from 61 to 33. However, as will be seen in the results in Section 5.3, we can get pretty close to the accuracy of the regular model. Albeit at the cost of evaluating significantly more words.

3.6 Main algorithm

Machine learning algorithms define what Devanagari character each Roman token transliterates into. For a complete algorithm we need to split up the Roman Hindi word into segments, iterate through these and collect the possible Devanagari words. During the algorithm these words are contained in $dev\text{-}word$ objects which contain the Devanagari word being built, its accumulated probability p as well as other data needed for the algorithm.

For efficiency reasons, we do not actually split the input word beforehand. Instead each $dev\text{-}word$ keeps track of how much of the input word has been transliterated. MATCH-TOKENS$(roman\text{-}word, i)$, which returns the matching tokens for position i in the input word, is then called each iteration. As was mentioned in Section 3.1, Roman Hindi words mostly, but not always, have a unique segmentation. Therefore the set of Roman tokens is partitioned into subsets based on priority so that *kaa* will only generate the segmentation *k-aa* but *hai* will generate both *h-ai* and *h-a-i* for example.

Pseudo code for the main algorithm can be seen in Figure 3.5. Before the algorithm begins, the input words is prepared by adding a word boundary marker at the end (since some tokens match against the end of a word) and all instances of *x* are replaced by *ks* (see Section 3.1). We then use a beam search approach: On each iteration the algorithm tries to match every word in the working set against the Roman Hindi input word and generate new Devanagari words with one more character. At the end of the iteration these new words are sorted by probability and only the n best words are kept, to keep the number of words from blowing up exponentially. When all Devanagari words in the working set have worked through the entire input word the algorithm is done and the working set returned as the result.

The algorithm for performing one transliteration step is shown in Figure 3.6. The basic procedure is to look up all possible Devanagari characters in the correct decision tree or LIBLINEAR model and then generate new Devanagari words with these attached and probabilities updated. However, there are a few special cases that must be handled.

The first one applies only when the model is trained for the numeric keypad of a mobile phone directly (see Section 3.5.3). When trained for the regular alphabet, for example the word *kaa* will always be split as *k-aa* but on the numeric keypad the corresponding key sequence is *522* which as well as *kaa* also could mean *kab* and others. Therefore the algorithm has no choice but to try both *5-22* and *5-2-2*. In the second case, the algorithm might transliterate the first *2* as implicit 'a' which means the Devanagari word does not change. Therefore it might also try to transliterate the second *2* as implicit 'a' which the ELIMINATE-NULLS function prohibits.

The second function FORCE-INDEPENDENT-VOWEL does just what its

```
TRANSLITERATE(rw, n)
 1  REPLACE-X(rw)                    ▷ Replace every occurrence of 'x' with 'ks'
 2  m ← length[rw]
 3  rw ← rw + 'φ'
 4  working-set ← dev-word()         ▷ Initialise working-set with empty dev-word
 5  repeat
 6          results ← ∅
 7          for each dw ∈ working-set
 8              do if roman-pos[dw] < m
 9                    then tokens ← MATCH-TOKENS(rw, roman-pos[dw])
10                         results ← results ∪ TRANSLITERATE-STEP(dw, tokens)
11                    else results ← results ∪ {dw}
12          SORT(results)             ▷ Sort words by descending probability
13          working-set ← results[1 . . n]
14    until roman-pos[dw] ≥ m, ∀ dw ∈ working-set
15  return working-set
```

Figure 3.5: Main algorithm.

name says. Sometimes the model will erroneously suggest the dependent form of a vowel when only the independent is valid. This can happen in two cases. The first case is when the previous character was an implicit 'a'. In this case, as noted in the previous paragraph, the Devanagari word will remain unchanged and we need to force the next vowel into its independent form. The second case is if the previous character was a dependent vowel and the model still suggests a dependent vowel for the current character. This could happen with a decision tree if we encounter a previously unseen trigram which forces the tree to back off to unigrams.

When constructing the machine learning algorithms we disregarded halants, instead they are added here. The function HALANT-POSSIBLE (see Figure 3.7) determines whether it would be legal to insert a halant before the next character. If so, we add two versions of the word, one with the halant and one without. As has been shown in several examples consonants are often pronounced without their implicit 'a' in Hindi. Therefore a consonant conjunct in Roman Hindi is not guaranteed to be a conjunct in Devanagari and we have no choice but to add both version (e.g. *hindi* is written with halant, हिन्दी, but *tumne* without, तुमने).

A major drawback to this method is that every time it is possible to insert a halant we get two words with the exact same probability, which the algorithm cannot tell apart (since it ignores halants). When we later add a wordlist (see Section 3.7) that will often help by eliminating one of

```
TRANSLITERATE-STEP(dw, tokens)
 1  results ← ∅
 2  n ← length[dw]
 3  for each token ∈ tokens
 4       do m ← length[token]
 5          dev-chars ← LOOKUP(token, dw) ▷ Look up possible Devanagari
                                              characters in relevant decision tree or SVM
 6          if last-was-implicit-a[dw]
 7            then ELIMINATE-NULLS(dev-chars)
 8          for each (d, p) ∈ dev-chars
 9               do dw2 ← copy(dw)
10                  p[dw2] ← p[dw] · p
11                  roman-pos[dw2] ← roman-pos[dw] + m
12                  if last-was-implicit-a[dw] or dw[n] ∈ dependent vowels
13                    then FORCE-INDEPENDENT-VOWEL(d) ▷ If needed, change
                                                        a dependent vowel into an independent one
14                  if HALANT-POSSIBLE(dw, d)
15                    then dw3 ← dw2 +′ ्′ + d
16                         results ← results ∪{dw3}
17                  dw2 ← dw2 +d
18                  results ← results ∪{dw2}
19  return results
```

Figure 3.6: The algorithm for performing one transliteration step.

the versions.

3.6.1 Efficiency

The major issue when implementing the above algorithms for a mobile phone is the amount of words that need to be created each iteration. However this can be quite effectively solved by keeping a pool of discarded objects instead of freeing their memory. This way, after a few iterations to let the pools grow big enough, further transliteration requires very few, if any at all, expensive memory allocations and de-allocations.

3.7 Wordlist

In Section 2.2, we said that a good transliteration system cannot be dependent on a wordlist but it can still be used as a complement. We mentioned in the previous section that the way we added halants meant both versions

HALANT-POSSIBLE(dw, d)

1 $n \leftarrow length[dw]$
2 **if** IS-CONSONANT$(dw[n])$ and IS-CONSONANT(d) and not $last\text{-}was\text{-}implicit\text{-}a[dw]$
3 **then return** true
4 **else return** false

Figure 3.7: The algorithm to determine if it is possible to insert a halant at the current position.

got the same probability and so we could not separate them, a wordlist helps with this. We take the wordlist with frequencies we obtained from the corpus and store it in a trie (Fredkin, 1960), see Figure 3.8 for an example.

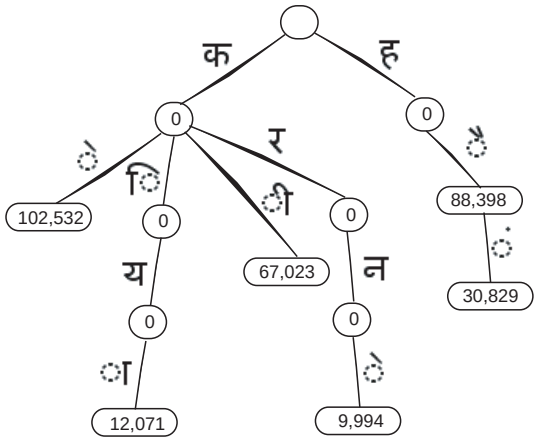

Figure 3.8: Part of a trie with Hindi words.

With this trie, we can easily check if a given word is a prefix of some word in the wordlist. This means that we can now improve the filtering and sorting at each iteration algorithm. Lines 12 to the end of TRANSLITERATE that previously looked like

12 SORT$(results)$
13 $working\text{-}set \leftarrow results[1\,..\,n]$
14 **until** $roman\text{-}pos[dw] \geq m, \forall\, dw \in working\text{-}set$
15 **return** $working\text{-}set$

will now change to

12 $prefixes \leftarrow \{dw \mid dw \in results$ and dw is prefix of some $w \in wordlist\}$
13 $results \leftarrow results \setminus prefixes$
14 $k \leftarrow length[prefixes]$
15 **if** $k < n$
16 **then** SORT$(results) \triangleright$ Sort as before
17 $working\text{-}set \leftarrow prefixes \cup results[1 \,..\, n-k]$
18 **else** $working\text{-}set \leftarrow prefixes$
19 **until** $roman\text{-}pos[dw] \geq m, \forall\, dw \in working\text{-}set$
20 SORT$(working\text{-}set, wordlist) \triangleright$ Sort words first by wordlist frequency then by probability
21 **return** $working\text{-}set$

Instead of just sorting the results and selecting the n best, we now first select all the words that are prefixes and only if necessary we sort the rest and select the $n - k$ best of these. At the end of the algorithm we sort the words by comparing first their frequency and then their probability.

With the wordlist, when TRANSLITERATE-STEP generates words both with and without halants we can often discard one of them directly. It also helps with filtering in general and ensures that for common words, the correct spelling will always be at the top of the list returned to the user. This is especially significant when training the model for a numeric keypad directly.

The full wordlist might be too large to be practically usable, especially when we are targeting a mobile phone. We can then reduce it in size by removing low frequency words. The naive way to do this would be to remove all words below a set frequency before building the trie. However, in a trie structure such as the one we are using, words entirely contained in interior nodes cost no extra space. We therefore build the trie from the full wordlist and afterwards we iteratively remove leaf nodes with frequency below the set threshold until no more can be removed. This way we get a trie of the same size as with the naive approach, but usually with a few thousand more words in it.

Another benefit of the wordlist is that it lets us generalise our system to predict words before the user has finished typing them. Instead of just checking if a given word is a prefix in the wordlist we can return all words it is a prefix of. Some care needs to be taken though, like restricting the depth to which we search in the tree, to keep the number of words returned manageable. For example, if the user typed the single character क there would probably be a few thousand words in the wordlist beginning with that character, so limiting the search depth to two or three times the length of the word actually typed is probably a good idea.

Chapter 4

Experimental set-up

All n-gram extraction and model building as well as the main algorithm was implemented in Ruby. The only exception is the LIBLINEAR classifier. The Ruby code communicated with it by writing the instance to be classified to a temporary file, executing the `predict` binary and then reading back the results from file.

The training set consisted of 65,711 weighted instances. Since LIBLINEAR lacks support for weighted instances, we instead listed each instance multiple times for a total of around 18 million instances.

4.1 Test framework

Using this code we built a test framework to measure the accuracy of our model and compare the decision tree and LIBLINEAR implementations. This framework took a set of Roman Hindi-Devanagari word pairs and tried to transliterate them, measuring the number of correct words, the average rank of the correct word (when found) in the result set and the maximum number of words that had to be created (i.e. the maximum size of $results$ in TRANSLITERATE) during transliteration. The main parameter that can be varied in TRANSLITERATE is n, the width of the beam. Therefore the test framework executed each configuration for values of n from 1 to 15.

For testing we selected 107 words from our corpus. We tried to select words that covered as much of the Devanagari alphabet as possible and that were uniformly distributed from common to uncommon. Although we did not include some of the most common words as we felt that would skew the distribution of our corpus too much when removed (the top ten words accounted for 20% of the corpus in our case). These words were transliterated by a native speaker, sometimes in several variants, giving us a total of 126 word pairs to test our model on.

When doing testing, the set of training data should be separate from the test set. However, we thought it would be interesting to test our system

on ”known” words as well, to test the design of our model. Therefore we tested three different levels of configuration.

In the first, denoted *(corpus + wordlist)* in the graphs, the test words were not removed from the corpus used to build the decision trees and LIBLINEAR models, nor the wordlist used during running. This to test the validity of our model, i.e. did our design with manually defined mappings from Roman Hindi to Devanagari actually cover real typing patterns?

In the second level, *(corpus)*, words were kept in the corpus but removed from the wordlist. This way the test words were still ”known” but the algorithm did not receive help from the wordlist to automatically rank the words near the top each iteration.

Finally we also tested with the test words removed from both corpus and wordlist. This way the result was entirely dependent on the machine learning algorithms to classify new instances on the models built from the training data.

4.2 Java GUI prototype

In addition to the Ruby code, we also developed a GUI prototype in Java to simulate actual use. A screen-shot can be seen in Figure 4.1. The application lets users type text, transliterates it in realtime and displays the five best suggestions, similarly to e.g. T9 on mobile phone. It also lets users go back and edit words, synchronising caret movement between the Roman Hindi and Devanagari word.

For transliteration, the application used the decision tree model, since that was the easiest to implement.

In the Java application we also tried the predictive functionality as outlined in Section 3.7, see Figure 4.2 for an example.

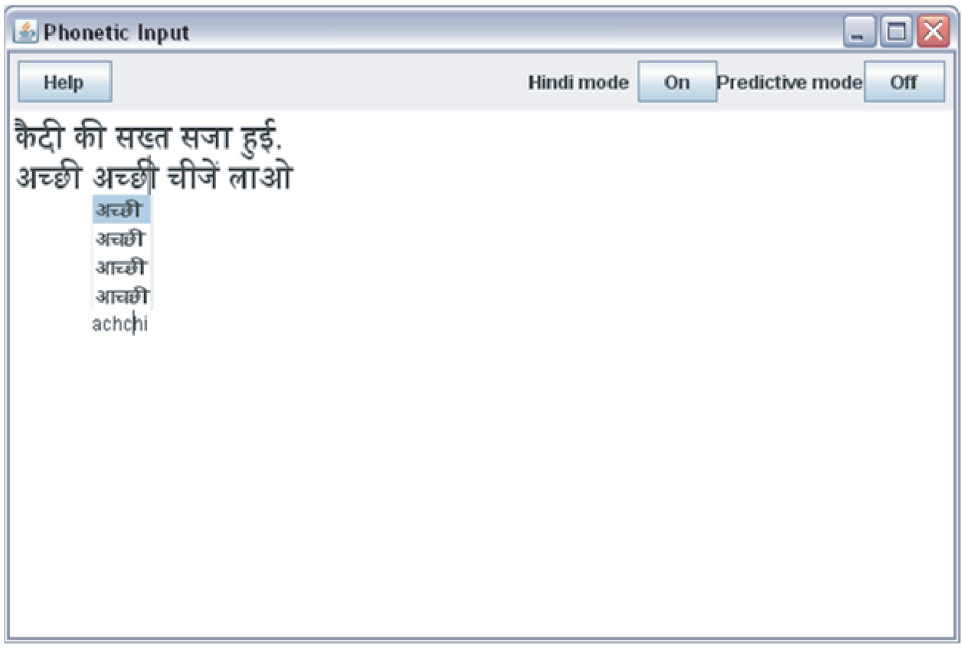

Figure 4.1: A screen-shot of the GUI prototype we developed in Java.

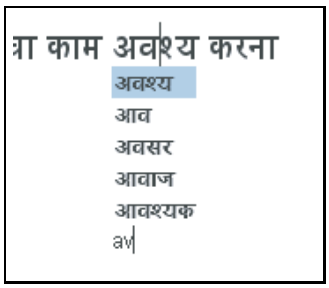

Figure 4.2: An example of predicting words before the user has finished typing them.

Chapter 5

Results

In this chapter, we describe the results from testing. All graphs are collected at the end of the chapter.

5.1 Pruning thresholds

When building the decision tree in Section 3.5.1 (and similarly when returning results from LIBLINEAR) we removed low probability choices. The first thing to be tested was different thresholds for this pruning to determine optimal values to use.

Results for the decision tree can be seen in Figure 5.1. As can be seen, higher thresholds work better for low values of n. This helps filter out the noise so to speak, while lower thresholds work better for high values of n, although at the expense of having to evaluate more words. We therefore felt a threshold of 0.1 was a good compromise.

For LIBLINEAR (Figure 5.2) the results were similar, although less pronounced. We again found 0.1 to be a good setting.

When training the model directly for a numeric keypad we thought a lower threshold might be needed. Since we now had greater ambiguity there would in general be more choices (and thus lower probabilities). However, as can be seen in Figure 5.3 the best result for all but very large values of n was 0.1.

5.2 Comparing decision trees and LIBLINEAR

Decision tree and LIBLINEAR models were tested in all three configurations defined in Section 4.1. The first thing that can be seen from the results in Figure 5.4 is that even in optimal circumstances the best our model can do is 87 out of 126 correct words. This will be discussed further in Section 6.1.

We can also see that for the case where test words are kept in the training corpus, the decision tree is actually slightly better than LIBLINEAR, while LIBLINEAR wins out if the words are removed. Another perspective can be seen in Figure 5.5 where we show the average rank in the result set for the correct word. We can see that even though the decision tree might not handle as many words as LIBLINEAR, when it does find the correct word it will usually put it closer to the top.

Comparing the results further we see in Figure 5.4 that, with the test words removed from the training corpus and wordlist, the LIBLINEAR model correctly transliterates 78 words for $n = 15$ while the decision tree model only manages 73. Those five words that only LIBLINEAR handles are shown in Table 5.1.

With the exception of हूँ the decision tree model fails on all these words for basically the same reason: they all contain several vowels that are available in both short and long versions.

For हूँ the decision tree incorrectly selects हूं instead (since we have removed the word हूँ from the corpus).

रहा	raha
परन्तु	parantu
हूँ	hun
अतिरिक्त	atirikt
महत्वपूर्ण	mahatwapurna
आतंकवादियों	atankvadiyon

Table 5.1: The words that only the LIBLINEAR model could handle.

5.3 Comparing models trained on the regular alphabet vs. a numeric keypad

In Figure 5.6, we compare decision trees trained on the regular alphabet and trees trained for a mobile keypad directly. We can see that, for the cases without wordlist support, the numeric model is significantly worse than the regular one. However, with wordlist support, the numeric model equals the regular one (albeit at the cost evaluating significantly more words, see Figure 5.7). So the numeric model could be feasible for e.g. SMS applications on a mobile phone, provided the number of words evaluated could be kept down.

5.4 Comparing bigram- and trigram-based LIBLINEAR models

We also tried training the LIBLINEAR model on bigrams. Results comparing it against the trigram model can be seen in Figure 5.8. The bigram model is significantly worse, except for the final configuration where it suddenly matches the trigram model.

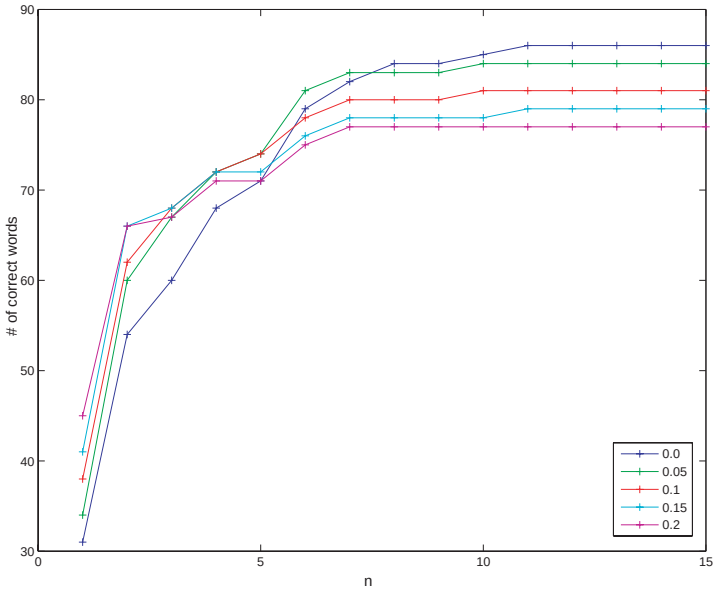

Figure 5.1: Using the decision tree with various thresholds for pruning.

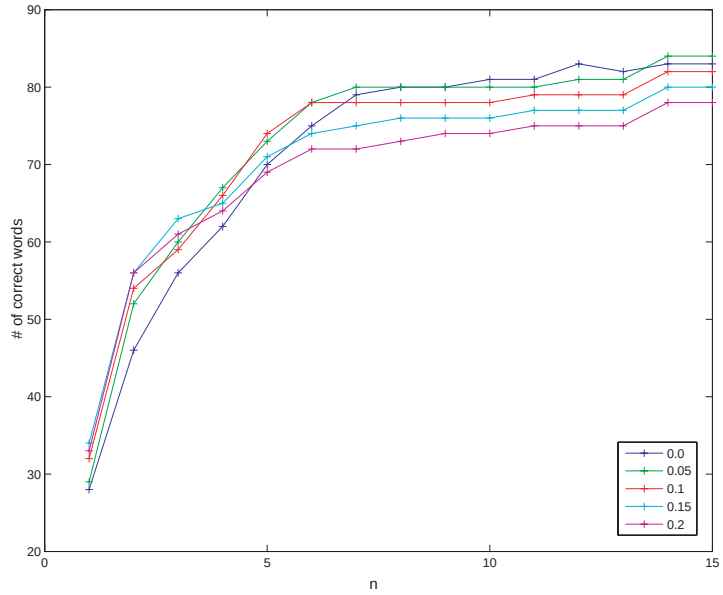

Figure 5.2: Using LIBLINEAR with various thresholds for pruning.

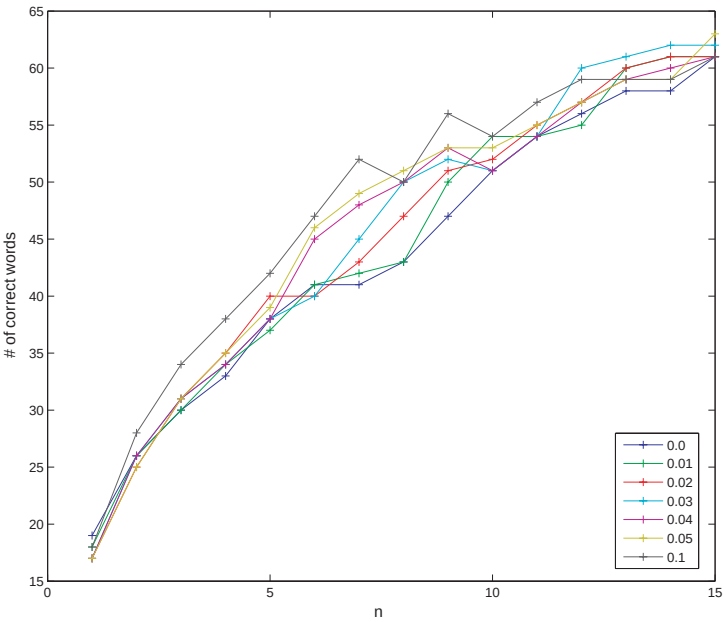

Figure 5.3: Using the decision tree trained directly for a numeric keypad with various thresholds for pruning.

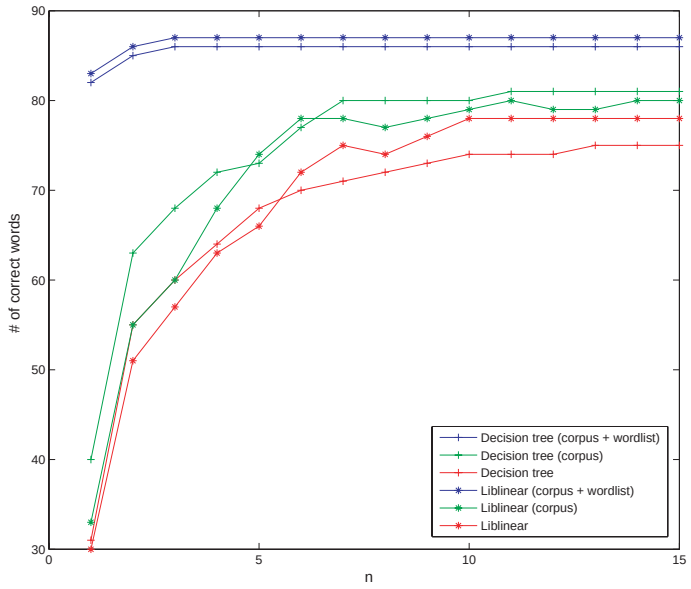

Figure 5.4: Comparing decision trees to LIBLINEAR.

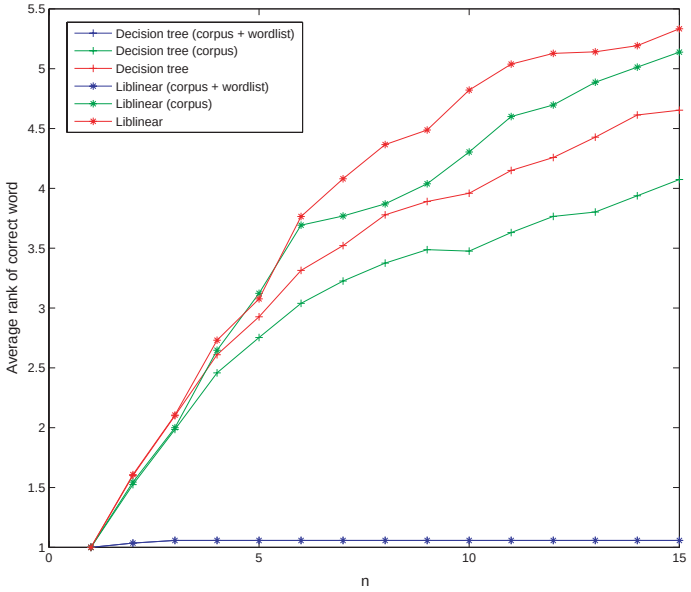

Figure 5.5: Comparing average rank of the correct word for decision trees and LIBLINEAR.

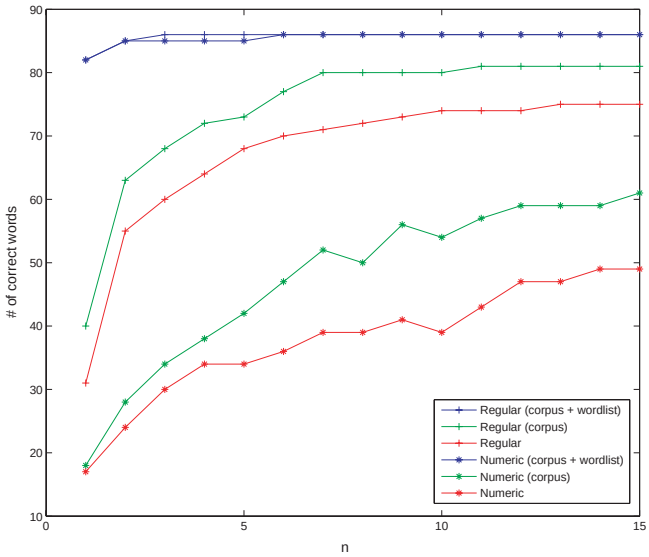

Figure 5.6: Comparing decision trees trained on the regular alphabet vs. trees trained directly for a mobile keypad.

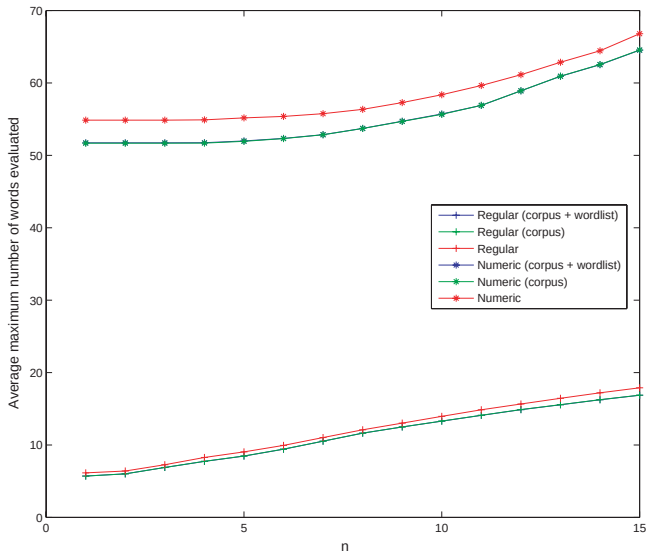

Figure 5.7: Comparing the maximum number of words evaluated for decision trees trained on the regular alphabet vs. trees trained directly for a mobile keypad.

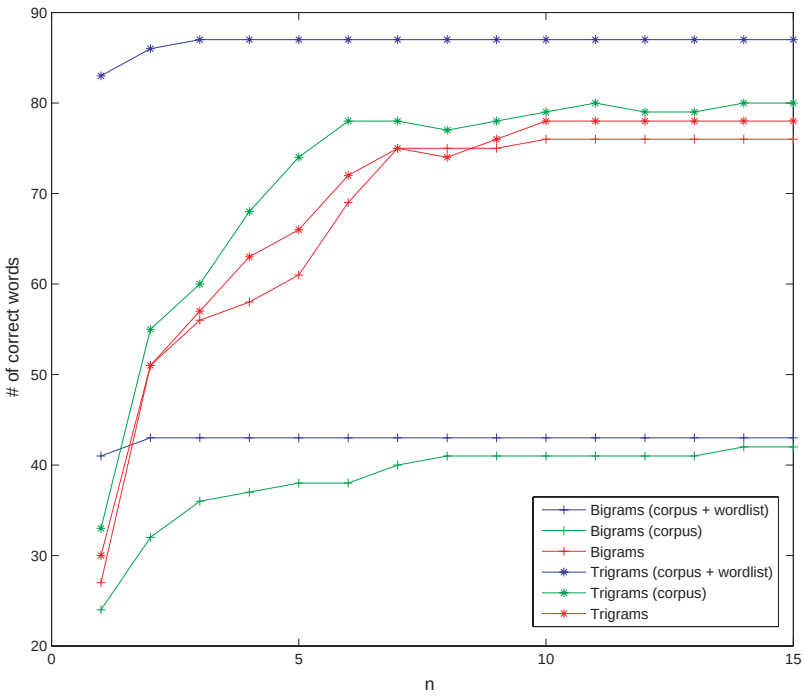

Figure 5.8: Comparing LIBLINEAR trained on bigram models vs. trigram models.

Chapter 6

Conclusions

6.1 Validity of our model

In the previous chapter, we showed that, even in the best case with full corpus and wordlist support, the model could not correctly transliterate more than 87 out of 126 words. The main reason for this is the way we designed our model in Section 3.1 and 3.3, with manually defined mappings, operating on an individual character level on the Devanagari side.

One example of things it cannot handle is the syllable व्य. The rigid transliteration of this would be *vya*, but in the test set there are several examples of other variations such as in the word व्यक्ति, transliterated as *waikti*, *vekti* or *wakti*.

Other examples are vowels transliterated more loosely than we have defined them, such as प्रयोग (literally *prayog*) transliterated as *proyog*, ऐसा (*aisaa*) transliterated as *aise* or प्रत्येक (*pratyek*) as *pratyak*.

All the examples mentioned could in theory be handled by defining more mappings. Doing this would put a greater burden on the machine learning algorithms to make correct choices though. Another option would be to investigate syllable based models instead, with the difficulties in segmentation that brings.

6.2 Applying the model to other Indic scripts

We have mentioned in this report the shared structure of Indic scripts. We also mentioned how the mappings defined for Devanagari here should be easily adaptable to other scripts. It should therefore be easy to implement transliteration for other languages. Unfortunately we did not have time during this thesis to collect a corpus for another language and actually test this ourselves.

6.3 Implementing the model on a mobile phone

We mentioned that the main algorithm could be implemented quite efficiently with the help of object pools. The two other areas that need to be considered are the wordlist and the machine learning algorithm.

During this thesis we did some prototype work on a trie implementation in C that, utilising path compression and some creative packing of the C structures, could fit a wordlist of 19,195 words in about 160 kB of memory. It could probably be reduced further with more work.

The other issue that needs to be decided is the choice of machine learning algorithm. As was shown in the testing, although LIBLINEAR was perhaps ultimately better, operating on known words and with wordlist support the two models were pretty similar. Therefore the choice should perhaps mainly be made on computational grounds.

The decision tree has the advantage that it is computationally trivial, simply traverse two levels of a tree. Its main disadvantage is its size. We did not have time to develop a full prototype of this as with the wordlist, but initial research showed that a reasonable (not fully optimised, but certainly not naive) implementation would probably require at least 300 kB of memory in total.

LIBLINEAR on the other hand has rather modest space requirements. Even in the plain-text format it stores its data in, the entire model could be fit in 286 kB. An optimised binary format would require much less (for example, floating point numbers are stored with 15-20 digits so even a naive binary format could easily cut the space requirements to a third). LIBLINEAR is however computationally rather more complex. In particular, it utilises a lot of floating point math which would probably need to be changed to fixed point to be viable on a mobile phone.

6.4 Handling foreign words

One thing that is missing in our model is the ability to handle foreign words. When we had a native speaker test the Java GUI prototype this was something that he discovered was missing pretty fast. For commercial feasibility it would therefore be vital to have some system to handle these words.

The superior option would be a solution like the one in Surana and Singh (2008). But lacking that, a small lexicon of English-Hindi word pairs could perhaps be an acceptable temporary solution.

Appendix A

Mappings

ँ	n	m		
ं	n	m		
ः	h			
अ	a			
आ	a	aa		
इ	i	yi		
ई	i	ii	ee	yi
उ	u			
ऊ	u	uu	oo	
ऋ	ri			
ऌ	l			
ऍ	e	ye		
ए	e	ye		
ऐ	ai			
ऑ	o			
ओ	o			
औ	au	ou		
क	k			
ख	k	kh		
ग	g			
घ	g	gh		
ङ	ng			
च	c	ch		
छ	ch	chh		
ज	j			
झ	j	jh		
ञ	ny			
ट	t	tt	th	
ठ	t	tt	th	tth

ड	d	dd		
ढ	d	dh	dd	ddh
ण	n	nn		
त	t			
थ	t	th		
द	d			
ध	d	dh		
न	n			
प	p			
फ	ph	f		
ब	b			
भ	b	bh		
म	m			
य	y	yϕ		
र	r			
ल	l			
ळ	l	ll		
व	v	w		
श	s	ss	sh	
ष	s	ss	sh	
स	s	ss	sh	
ह	h			
ा	a	aa		
ि	i	yϕ		
ी	i	yϕ	ii	ee
ु	u			
ू	u	uu	oo	
ृ	ri			
ॄ	ri			

ॅ	e				
ॅ	e				
ॉ	o				
ॉ	o				
ौ	au	ou			
क़	k	q			
ख़	k	kh	khh		
ग़	g	gh	ghh		
ज़	z				
ड़	d	dh	dd	ddh	r
ढ़	r	rh			
फ़	f	ph			

Bibliography

Baker, P., Hardie, A., McEnery, T., Xiao, R., Bontcheva, K., Cunningham, H., Gaizauskas, R., Hamza, O., Maynard, D., Tablan, V., Ursu, C., Jayaram, B. D., and Leisher, M. (2004). Corpus Linguistics and South Asian Languages: Corpus Creation and Tool Development. *Lit Linguist Computing*, 19(4):509–524.

Chaudhuri, S. (2006). Transliteration from non-standard phonetic bengali to standard bengali. In *Satellite Workshop on Language, Artificial Intelligence and Computer Science for Natural Language Processing Applications*.

Ekbal, A., Naskar, S. K., and Bandyopadhyay, S. (2006). A modified joint source-channel model for transliteration. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 191–198, Morristown, NJ, USA. Association for Computational Linguistics.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874.

Fredkin, E. (1960). Trie memory. *Commun. ACM*, 3(9):490–499.

Jung, S. Y., Hong, S., and Paek, E. (2000). An english to korean transliteration model of extended markov window. In *Proceedings of the 18th conference on Computational linguistics*, pages 383–389, Morristown, NJ, USA. Association for Computational Linguistics.

Krishna, A., Ajmera, R., Halarnkar, S., and Pandit, P. (2005). Gesture keyboard – user centered design of a unique input device for indic scripts. Technical report, HP Laboratories India. HPL-2005-56.

Quinlan, J. R. (1986). Induction of decision trees. *Mach. Learn.*, 1(1):81–106.

Rathod, A. and Joshi, A. (2002). A dynamic text input scheme for phonetic scripts like devanagari. In *Proceedings of Development by Design (DYD)*.

Shanbhag, S., Rao, D., and Joshi, R. K. (2002). An intelligent multi-layered input scheme for phonetic scripts. In *SMARTGRAPH '02: Proceedings of*

the 2nd international symposium on Smart graphics, pages 35–38, New York, NY, USA. ACM.

Singh, A. K. (2006). A computational phonetic model for indian language scripts.

Surana, H. and Singh, A. K. (2008). A more discerning and adaptable multilingual transliteration mechanism for indian languages. In *Proceedings of the Third International Joint Conference on Natural Language Processing (IJCNLP)*, Hyderabad, India. Asian Federation of Natural Language Processing.

UzZaman, N., Zaheen, A., and Khan, M. (2006). A comprehensive roman (english) to bangla transliteration scheme. In *Proc. International Conference on Computer Processing on Bangla (ICCPB-2006)*.

Yoon, S.-Y., Kim, K.-Y., and Sproat, R. (2007). Multilingual transliteration using feature based phonetic method. In *ACL*. The Association for Computer Linguistics.