Latin script keyboards for South Asian languages with finite-state normalization

Lawrence Wolf-Sonkin, Vlad Schogol, Brian Roark and Michael Riley
{wolfsonkin,vlads,roark,riley}@google.com
Google Research

Abstract

The use of the Latin script for text entry of South Asian languages is common, even though there is no standard orthography for these languages in the script. We explore several compact finite-state architectures that permit variable spellings of words during mobile text entry. We find that approaches making use of transliteration transducers provide large accuracy improvements over baselines, but that simpler approaches involving a compact representation of many attested alternatives yields much of the accuracy gain. This is particularly important when operating under constraints on model size (e.g., on inexpensive mobile devices with limited storage and memory for keyboard models), and on speed of inference, since people typing on mobile keyboards expect no perceptual delay in keyboard responsiveness.

1 Introduction

Many of the world's writing systems present challenges for machine readable text entry compared with alphabetic writing systems (such as the Latin script used for the English in this paper). For example, a very large character set, such as that used for Chinese, can be impractical to represent on a keyboard requiring direct selection of characters; hence specialized encoding methods are generally used based on smaller symbol sets. For example, the well-known pinyin system for text entry of Chinese relies on Latin alphabetic codes to input Chinese characters. South Asian languages, such as Tamil and Hindi, also use writing systems that, while lacking the thousands of characters as in Chinese, are nonetheless challenging for direct typing (particularly on mobile devices), and hence are frequently entered using the Latin alphabet. In those languages, however, unlike Chinese, there is no single system that is used for *romanization*, rather individuals typically provide a rough phonetic transcription of the words in the Latin script.

The use of pinyin for Chinese is generally part of a system for converting the text into the native script, and this can also be achieved for keyboards in South Asian languages (Hellsten et al., 2017). However, for these languages, many individuals prefer to simply leave the text in the Latin script rather than converting to the native script. To provide full mobile keyboard functionality in such a scenario – including, e.g., word prediction and completion, and automatic correction of so-called fat finger errors in typing – language model support must be provided. Yet in the absence of a standard orthography, encoding word-to-word dependencies becomes more complicated, since there may be many possible versions of any given word.

In this paper, we examine a few practical alternatives to address the lack of a conventionalized Latin script orthography for use in a finite-state keyboard decoder. We use several different transducers that normalize input romanizations to either a native script word form or a "canonical" Latin script form[1] in order to combine with a word-based language model. To produce Latin script after this normalization, we must produce text from the input tape of these transducers. We also present an alternative method involving a compact representation of a large supplementary lexicon that covers highly likely romanizations of in-vocabulary words. All of these methods provide accuracy improvements over the baseline

[1] We use *canonical* in quotes here and elsewhere because there is no standard orthography hence no true canonical form; rather, for each native script word in our lexicon, we choose one romanization as "canonical".

Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing, pages 108–117
Dresden, Germany, September 23-25, 2019. ©2019 Association for Computational Linguistics

(fixed vocabulary) method.

In the next section, we give some background on the problem before outlining our new methods. We then present experimental results of keyboard entry simulation for Hindi, in which we demonstrate over 50% relative reduction in error rate[2] over existing baselines.

2 Background and preliminaries

2.1 South Asian romanization

Romanized text entry is widely used in languages, such as Hindi and Arabic, for which there is no agreed-upon adherence to any particular conventionalized representation in the Latin script such as is found in Chinese. South Asian languages written natively in a Brahmic script, including Hindi, Bengali, Tamil and many others, are heavily romanized mainly due to the complexity of typing the scripts. These scripts are abugida (or alphasyllabary) writing systems that are based on consonant/vowel "syllables" (*akṣara*) that pair consonants with a default vowel. Alternative vowels (as well as the lack of a vowel) are designated through the use of various diacritic marks that can appear above, below, or on either side of the consonant (or consonant cluster). This, along with complex multi-consonant ligatures (known as conjuncts), makes direct use of native script keyboards relatively uncommon. An exemplar word in the Devanagari script containing such complex graphemes is given in Figure 1. Romanization is also used for Perso-Arabic scripts in South Asia, such as that used for Urdu, but not presumably due to complexities in representing such scripts on native keyboards, but rather due to historical reasons[3] and perhaps the influence of other regional languages.

As a result of having no conventionalized romanization system, text in, say, romanized Hindi has no standardized orthography, but rather words are usually represented via rough phonetic transcriptions in Latin script. For example, the Hindi words संस्कृत and संपूर्ण are commonly romanized as *sanskrit* and *sampuran*, respectively. Both words begin in Devanagari with the grapheme सं which is /sa/ with a diacritic indicating a nasal consonant (such as /n/) in the coda. Note that the nasal becomes either /n/ or /m/ depending on the following consonant,[4] demonstrating how these romanizations are driven by pronunciation rather than from the native orthography. Urdu has the same words, written سنسکرت and سمپورن respectively in the Perso-Arabic script, and they are romanized similarly to the Hindi words, also demonstrating the role of pronunciation rather than writing system in romanization for these languages.[5] In general, due to this lack of a standardized spelling in the Latin script, romanizations may vary due to dialectal variation, regional accent, or simply individual idiosyncrasies.

ब ⇒ ब /ba/
ब+ ◌् ⇒ ब् /b/
ब्+ र ⇒ ब्र /bra/
ब्र+◌ा ⇒ ब्रा /brā/
ह + ◌् + म+◌ी ⇒ ह्मी /hmī/
ब्रा+ ह्मी ⇒ब्राह्मी /brāhmī/

Figure 1: Demonstration of how the Hindi word ब्राह्मी /brāhmī/, meaning Brahmic, is decomposed into its unicode codepoints as written in Devanagari. Pronunciations are shown between slashes.

As a concrete example, a blog entry on the general topic of political corruption on a site run by the India Today Group from 2011 has comments in (1) English; (2) Hindi written in Devanagari (its native script and that is used in the blog post itself); and also extensively in (3) romanized Hindi.[6] One comment begins: "Bhrashtachar aam aadmi se chalu hota hai…", which presumably corresponds to the Devanagari: भ्रष्टाचार आम आदमी से चालू होता है and roughly translates to: "Corruption starts with the common man…" Given that corruption is the overall topic of the blog post, it is unsur-

[2] Word-error rate in this setting means recovery of the intended form typed by the user. We take the romanized strings in the validation set as the intended forms, despite spelling variation throughout.

[3] Languages using the Cyrillic script are also frequently romanized. There, and perhaps also for Perso-Arabic scripts, the issue is with historical lack of font and encoding support in certain scenarios.

[4] This is a process known as assimilation.

[5] Note that unlike in Devanagari, Perso-Arabic script Urdu does in fact graphically differentiate these phonetically differentiated onsets, spelling them respectively as سن ⟨sn⟩ and سم ⟨sm⟩.

[6] http://blogs.intoday.in/index.php?option=com_myblog&contentid=62323&show=Removal-of-corruption-from-the-beginning-itself&blogs=2

prising that the Hindi word for this shows up in many comments. It is, however, variously romanized. By our count: 16 times it is romanized as “bhrastachar”; 7 times as “bhrashtachar”; and once each as “barashtachaar”, “bharastachar”, “bharstachar”, “bhastachar” and “bhrstachar”. Google Translate provides both a translation and a romanization of the word (“bhrashtaachaar”[7]), a form which interestingly is not found in our (admittedly small) example blog comment sample.

2.2 Transliteration and romanized text

The need to transliterate between writing systems comes up in many application scenarios, but early work on the topic was largely focused on the needs of machine translation and information retrieval due to loanwords and proper names (Knight and Graehl, 1998; Chen et al., 1998; Virga and Khudanpur, 2003; Li et al., 2004). These approaches either explicitly modeled pronunciation in the languages (Knight and Graehl, 1998) or more directly modeled correspondences in the writing systems (Li et al., 2004). Models for machine transliteration have continued to improve, through the use of improved modeling methods including many-to-many substring alignment-based modeling, discriminative decoding, and multilingual multitask learning (Sherif and Kondrak, 2007; Cherry and Suzuki, 2009; Kunchukuttan et al., 2018), or by mining likely transliterations in large corpora (Sajjad et al., 2017). Transliteration models are also being deployed in increasingly challenging use scenarios, such as mixed-script information retrieval (Gupta et al., 2014) or for mobile text entry (Hellsten et al., 2017).

The volume of romanized text in languages that use other writing systems is an acknowledged issue, one which has grown in importance with the advent of SMS messaging and social media, due to the prevalence of romanized input method editors (IMEs) for these languages (Ahmed et al., 2011). The lack of standard orthography and resulting spelling variation found in romanization is also found in other natural language scenarios, such as OCR of historical documents (Garrette and Alpert-Abrams, 2016) and writing of dialectal Arabic (Habash et al., 2012).

For this study, we make use of Wikipedia data originally written in the native script that has been romanized, and our task is to permit accurate text entry on mobile keyboards, rather than transliteration to the native script or normalization for use in other downstream tasks. In this case “accurate text entry” means fidelity to the intended text, even if that intended text is written without consistent spelling. If the user noisily types “bgrashtachsr” while intending “bhrashtachar”, the keyboard should produce “bhrashtachar” not another romanization such as “bhrastachar”. Given annotator-romanized Wikipedia text, we evaluate our ability to correctly recognize the actual romanizations used.

2.3 Mobile keyboard decoding

Virtual keyboards of the sort typically used on mobile devices convert a temporal sequence of interactions with the touchscreen (taps or continuous gestures) into text. Like speech recognition or optical character recognition, the mapping of noisy, continuous input signals to discrete text strings involves stochastic inference; further, given the low required latency during typing, models must be compact enough to run on the local device and inference with them must be fast. For this reason, the kinds of finite-state methods that have been used for speech recognition and OCR have also been used for this task (Ouyang et al., 2017). The work we present here will be in the context of such an FST-based keyboard decoder.

For touch typing, where the input consists of a sequence of taps, we designate with the term *literal* the string corresponding to the actual keys touched. The intended string may differ, due to such phenomena as so-called “fat finger” errors, i.e., hitting a neighboring key, omitting a key or including an extra tap.

Analogous to the acoustic model in speech recognition, which assigns probabilities to the continuous waveform given a sequence of phones, such a decoder makes use of a *spatial* model, assigning probabilities to the sequence of taps (or gestures) given a sequence of letters. Taps, for example, are modeled in Ouyang et al. (2017) with Gaussians centered on the middle of each key. Costs are thus assigned to alternative possible intended character strings

[7] translate.google.com/#en/hi/Corruption

which may have substitutions, deletions and insertions relative to the literal string.

In speech recognition, phones are typically split in the acoustic model based on the surrounding context, in order to capture co-articulation effects and other influences on the acoustics associated with a particular intended phone. Similarly, in the spatial model, keys are typically split based on the previous touched key, which we will term *bikey* representation. So, at the start of a word, the letter 'b' will be represented as '_b', whereas after the letter 'a', it would be represented as 'ab'. Later we will have methods that must be aware of the input representation.

The spatial model cost and the language model cost are combined by the decoder to score competing output strings. Typically these scored string alternatives will be compared to the literal string and only selected if the difference in score is above some threshold, to avoid spurious changes to what the user typed (Ouyang et al., 2017). To accept any string (including any possible literal string), a loop transition for every character with some fixed cost can be included at the unigram state (the base of the smoothing recursion, see Roark et al., 2012), so that every string in Σ^* has non-zero probability.

In addition to decoding for auto-correction, the language model may also be used for word prediction and completion, i.e., showing suggestions in a small dynamic portion of the keyboard. In this paper, we do not have much to say about this part of the process, other than to point out when its demands make certain approaches more complicated than others.

Such an architecture has also been used for transliteration from Latin script input to native script output (Hellsten et al., 2017), by interposing a finite-state transducer (FST) between the spatial model (defined over the Latin script) and the language model (defined over native script words). Some of our methods are related to these, although the output of the keyboard does not change script.

3 Methods

3.1 Word transliteration models

For both off-line model training and on-line transliteration-based decoding methods, we make use of pair n-gram (also known as "joint multi-gram") modeling methods (Bisani and Ney, 2008), which Hellsten et al. (2017) also use to train their transliteration models. Given a lexicon with words in the native script and possible romanizations of those words (see §4.1 for specifics on our data), expectation maximization is used to derive pairwise symbol alignments. For example, भ्रष्टाचार and "bhrashtachar" may yield a pairwise symbol alignment of:

भ:b ्:h र:r ϵ:a ष:s ्:h ट:t ा:a च:c ϵ:h ा:a र:r

where each symbol is composed of an input (native script) unicode codepoint (or ϵ, denoting the empty string) and an output (Latin script) unicode codepoint (or ϵ). These symbol pairs then become tokens in an n-gram language model encoded as an automaton. Finally, the automaton is converted to a transducer with native script on one side and Latin script on the other.

This model provides a joint probability distribution over input:output sequence pairs, e.g., for a word भ्रष्टाचार and a romanization "bhrashtachar", i.e., P(भ्रष्टाचार, bhrashtachar). As Hellsten et al. (2017) note, within most decoding settings that combine with a language model on the native script side, a conditional probability is actually what is needed: P(bhrashtachar | भ्रष्टाचार). We refer readers to that paper for details on how to incorporate the appropriate normalization into an FST-based decoder. We use similar methods, permitting the model to be used in both on-line and off-line scenerios.

Note that it is trivial to swap the input and output symbols for such a model, either by changing the ordering of the pair symbols in the training data or simply inverting the resulting WFST. The same model can thus be used for transliteration from input Latin script to native script forms; or from input native script to romanizations.

For example, suppose T is a transliteration transducer (Latin script on the input side and native script on the output side) and S is an automaton that accepts a single native script word w for which we wish to find likely romanizations. If we compose $T \circ S$, this yields a transducer encoding alternative Latin/native script string relations with w as the output

string. We can convert this transducer into an automaton accepting alternative romanizations of w by *projecting* all transitions onto their input labels, i.e., preserving only the input label on every transition. Some transitions in this acceptor of alternative romanizations might be epsilons, so we can remove epsilon transitions (Mohri, 2002), then select the n most likely paths (Mohri and Riley, 2002). All of these operations are general operations supported by the OpenFst library (Allauzen et al., 2007). The n unique shortest paths in $\textsc{RmEpsilon}(\textsc{ProjectInput}(T \circ S))$ are the n most likely romanizations of w.

3.2 Baseline fixed-vocabulary system

Our baseline system relies on automatic romanization of native script language model training data. Using a transliteration transducer, trained as described in §3.1, each word in our fixed vocabulary is assigned a "canonical" (i.e., best scoring) romanization as its Latin script representation. We then replace the native script words in our language model training corpus with their canonical romanizations and retrain the model, yielding a language model over strings in the Latin script.

If the distribution over romanized alternatives for भ्रष्टाचार in the blog comments that were mentioned in §2.1 represented the distribution provided by our model, then "bhrastachar" would become its canonical romanization. Alternative spellings (e.g., bhrash̲tachar) would only match that word via the character loop method (mentioned in §2.3) permitting the omitted letter, generally with a cost.

3.3 Compact supplemental unigram

Not every substituted, inserted or omitted tap is created equal when it comes to likely romanization variants. For example, as we have seen in our running example, the use of 'h' to indicate aspiration for consonants such as भ may or may not be used in romanizations. Similarly long vowels and geminates are sometimes represented by doubling of Latin symbols, but often not. These variants are not random in the way that a character loop model would score them. One method for adding likely alternative romanizations is to simply add them as alternative word forms to the language model. These romanizations can be computed using the method outlined in §3.1.

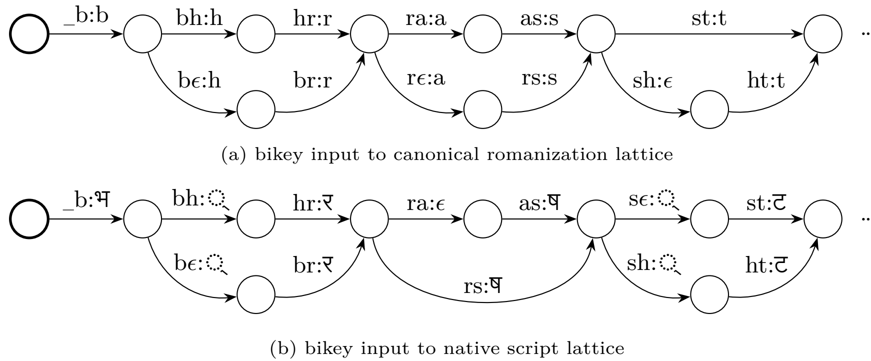
However, adding many alternative romanizations of the same word can become space prohibitive, particulary for on-device methods, where storage and active memory usage are both at a premium. It is possible, however, to provide a very compact encoding specifically of the words stored exclusively in the unigram, i.e., words that are neither prefixes nor suffixes of any higher order n-grams in the language model. We achieve this in two steps. First, we build two automata that accept all and only this set of words: a weighted automaton W, which weights the path for each word with the appropriate cost for that word within the language model; and an unweighted automaton A which encodes the same set of words as W and has been determinized and minimized. Next we create a weighted automaton W_m that has the same topology as A, but which is weighted to minimize the KL-divergence (Kullback and Leibler, 1951) between W and W_m, using methods from Suresh et al. (2019). This is an approximation of the distribution represented in W over a much more compact topology. The methods to perform this approximation are part of the open-source `OpenGrm` stochastic automata (`SFst`) library (available at `http://www.opengrm.org`). We then integrate W_m into the larger language model automaton, with the unigram state of the language model serving as both the start and final state for the paths corresponding to those in W_m. This can be straightforwardly accomplished by using the *Replace* operation in the OpenFst library (`http://www.openfst.org`).

3.4 Transducer to canonical form

As we noted in §3.1, we build pair n-gram transliteration models between native script and romanized forms. In a similar way, we can build a transducer between *canonical* romanized forms and alternative romanizations. To re-use our example, if "bhrastachar" is the canonical romanization, and "bhrash̲tachar" is another attested form, we can use expectation maximization to derive an alignment:

b:b h:h r:r a:a s:s ϵ:h t:t a:a c:c h:h a:a r:r

A pair n-gram model built from this would allow weighted transduction from input romanizations to the canonical form, which corresponds to tokens in the language model. We



(a) bikey input to canonical romanization lattice

(b) bikey input to native script lattice

Figure 2: Lattices illustrating how reading from the input tape rather than the output tape can provide support for varied romanized input, for either native script or "canonical" romanized output.

can thus use a transducer much in the way described in Hellsten et al. (2017) to transliterate[8] between variant romanizations and the chosen canonical romanizations.

There is, however, a complication with using this method within the keyboard, in contrast to the earlier described methods. If the keyboard actually performed the transduction from input romanization to the canonical Latin script form, then it would enforce a normalization on the user's spelling of the word. If the user types "bhras<u>h</u>tachar", this system, as it has been described, will output "bhrastachar" (without the 'h'), since that is our chosen canonical form. However, there is no standard orthography in the Latin script for Hindi, i.e., there is no correct spelling. Our canonical form is chosen for convenience to be the highest scoring romanization from the model. Even if we were to choose some kind of generally common version as our canonical form, however, for any given individual we may end up coercing the output of a form that they disprefer. Instead, we would like to allow them to maintain their preferred form, i.e., they should be able to type their intended string.

Because the decoder is based on WFSTs, we have a particularly straightforward solution to this: output the string from the input tape rather than the output tape. That is, we use the transducer within the decoder just as is done in Hellsten et al. (2017), however we output the string on the input side corresponding to the best scoring solution. In this way, we derive the modeling benefit from the language model without imposing a canonical romanization on the user. Note that it would be possible to perform this large composition and project onto input labels off-line rather than on-the-fly, but the size of the off-line composition is prohibitively large for on-device operation, for reasons similar to those discussed in Hellsten et al. (2017). The output labels are thus preserved in one of the transducers used during on-the-fly composition.

Figure 2a shows a WFST lattice representing alternative paths through the decoder, with bikey inputs and canonical romanization outputs.[9] For convenience, bikey representations of outputs with no corresponding input display the previous key followed by an epsilon, e.g., 'bϵ' signifies an omitted key following a 'b'. Note that every path through this lattice has an output string corresponding to "bhrast", the prefix of the canonical romanization of our running example. Different paths represent different input string variations corresponding to this word.

To read a string off of the input side of such a lattice, we take the last symbol of the bikey at each transition, with ϵ representing the empty string. In such a way, the presented lattice encodes the alternatives *bhrast*, *brast*, *bhrsht*, *bhrasht*, etc. Whichever path has the lowest cost during decoding would be the version that is produced by the keyboard.

3.5 Transducer to native script

If, as discussed in the previous section, we output from the input tape of our WFSTs, then

[8]Note this isn't quite transliteration in the usual sense since the strings stay in the same writing system, but we co-opt the term since Hellsten et al. (2017) used an identical architecture for transliteration.

[9]Note that, during decoding, partial results may be displayed to the user to improve responsiveness, so these figures should be taken as an illustration not a depiction of the decoding process.

there is no strong reason[10] to have Latin script on the output tape. Instead, we use a native script language model and the same sort of pair n-gram transducer as used in a transliterating keyboard, then simply read the result from the input tape.

Figure 2b has a set of paths, all of which produce the native script word prefix (भ्रष) on the output side. As with the other lattice, the paths represent different input romanizations corresponding to this string, which can be recovered from the transition labels.

3.6 Native script OOV modeling

One question we have only briefly touched upon is how to deal with out-of-vocabulary (OOV) items when using a transliteration transducer, i.e., words not in the language model being used for decoding. The simple default method is to have a *character loop* at the unigram state of the language model that accepts each character. That loop then gets an approximated language model cost from the transliteration transducer, to the extent that that model provides a joint probability of input and output strings. Alternatively, we can build a character language model, or even more complicated data structures, to assign probabilities to OOV words.

We opt to follow an approach that provides flexibility to move between two extremes, the most permissive but least accurate being the character loop, and the most restrictive but most accurate being a weighted character trie. The trade-off is controlled by a single parameter N, which is the number of states we wish to use to represent the OOV model. We start with a large collection of native script words and their unigram probabilities. We first build a weighted character trie representing these words. The trie is weight-pushed so that the probability mass of a state is seen as early as possible by the decoder. Next we rank each state of the trie by the total probability mass of all words reachable from that state. Finally we remove any state beyond the first N states in the ranking. All transitions from retained states to removed states are redirected to a state with the original character loop. In this way, we provide a mechanism to smoothly scale between a full trie representation (no states removed) down to a single-state character loop (all states removed), and everything in between. This approach, like the character loop baseline, permits arbitrary word-forms to be typed, but it does so in a way that better captures the distribution of word forms in the language. The pruned trie provides an approximation to the distribution in the full trie, which permits a graceful tradeoff between the size of the encoding and the quality of the approximation.

4 Experiments

4.1 Data

Transliteration models were trained from a proprietary lexicon of Hindi words and attested romanizations, consisting of approximately 110,000 native script words and on average 3.1 romanizations per word. These aligned Devanagari–Latin word pairs were used to build a WFST transliteration model using methods detailed in §3.1. We built pair 3-gram models, pruned to contain just 110,000 n-grams prior to conversion to a transducer.

Language models, both in Devanagari and canonical Latin forms, were trained on a large and diverse set of Devanagari Hindi text collected from the web, and were not trained for any specific domain. The 150,000 most frequent words in the training set were retained in the language model, and trigram word-based models were trained and then pruned to retain just 750,000 n-grams, so as to fit within on-device space limits.[11] For a single experiment, we additionally considered a language model containing 1,500,000 n-grams, double the n-gram count of the others.

For methods using a transducer to native script within decoding, the language model is in the native script; whereas in other conditions, the language model is in the Latin script. To train the Latin script language

[10] This is not strictly speaking true, since, as is pointed out in §2.3, the models may also be used for word prediction and completion. In order to seamlessly integrate with such processes, predicted and completed words would have to be presented in the Latin script, hence some additional information would need to be provided for each word in the vocabulary.

[11] Our work as targeted South Asian languages, where inexpensive smartphones are the norm, hence, as mentioned in Hellsten et al. (2017) we have generally targeted total model sizes around 10MB.

Method	Word error rate (%)	Tap avg ms	Avg active states	arcs	Model size (MB)
Literal	45.0	-	-	-	-
Fixed vocabulary (canonical only)	22.6	0.95	164.8	417.9	4.9
Fixed vocab + supplemental unigram	12.4	1.00	122.3	404.4	10.6
Fixed vocab + transducer to canonical	12.9	1.01	75.6	225.6	9.3
Transducer to native script	13.1	0.95	66.7	184.7	11.0
Transducer to native + OOV model	**10.5**	1.08	67.5	181.8	11.2

(a) An operating point of approximately 1.0ms per tap

Method	Word error rate (%)	Tap avg ms	Avg active states	arcs	Model size (MB)
Literal	45.0	-	-	-	-
Fixed vocabulary (canonical only)	22.5	0.59	71.8	202.5	4.9
Fixed vocab + supplemental unigram	**12.1**	0.53	38.1	132.0	10.6
Fixed vocab + transducer to canonical	14.1	0.54	24.4	80.9	9.3
Transducer to native script	14.0	0.60	31.8	91.6	11.0
Transducer to native + OOV model	12.3	0.54	19.7	54.9	11.2

(b) An operating point of approximately 0.55ms per tap

Table 1: The word error rate for the decoding of noisy touchpoints into Latin script strings at two operating points along the speed–accuracy tradeoff. The average number of milliseconds required per character as well as the average number of states and transitions active during decoding and the model size in megabytes are listed. The best performing (lowest) word error rate method for each operating point is bolded.

model for Hindi, each word in the vocabulary (each of which is in Devanagari), is replaced with its "canonical" romanization, i.e., the highest probability romanization according to the trained transliteration model.

Devanagari script sentences from Hindi Wikipedia were manually romanized by native speakers, and 4,000 of these (for a total of 36,027 word tokens) were used as our development set for validation of the methods presented above.

4.2 Evaluation

To evaluate our methods, we simulate touch points of a tapping keyboard as follows. For each symbol in the (Latin script) input strings, we sample a touch point from Gaussian distributions in two dimensions, with mean value at the center of the key. To establish how much noise is introduced by this method, we evaluate the error rate of simply emitting the literal sequence, i.e., the symbols associated with the keys that our noisy touch points actually fall within. Improvements over the literal baseline are due to decoder auto-correction.

The resulting touchpoints are then fed into the decoder under each of our conditions, and the strings output from the decoder are then compared with the original text strings, which are taken to be the intended strings. As mentioned elsewhere in the paper, the goal is to allow users to type their intended strings, without normalizing away their versions of the romanized words. Thus we measure word-error rate versus the reference version in the romanized string. Note that the keyboard decoder has various meta-parameters that can impact, e.g., the speed–accuracy trade-off. In addition to sweeping over such parameters for a given method, as shown in Figure 3, we compare performance across the methods at comparable operating points (in terms of average milliseconds per character) in Tables 1a and 1b.

Note that the absolute numerical values of the latencies are not meaningful, just the comparisons between the latencies. As discussed in Hellsten et al. (2017) and mentioned earlier, latencies must be low enough that no lag in keyboard responsiveness is perceived, and target values on device are often around 20ms per tap. However this must be the case also for inexpensive devices with low processing power, and the decision to deploy a model would depend on device trials. For the purposes of this paper, however, we just report values on a single device that can be used for comparison purposes. The operating points chosen for the Tables are two that are plausible candidates for use on such inexpensive devices.

4.3 Results

While analyzing the entire operating curve as shown in Figure 3 gives us an idea of the full potential of any particular model, in a resource-constrained scenario such as a mobile keyboard, we are ultimately restricted to working at a particular operating point on the speed–accuracy tradeoff. At a higher operat-

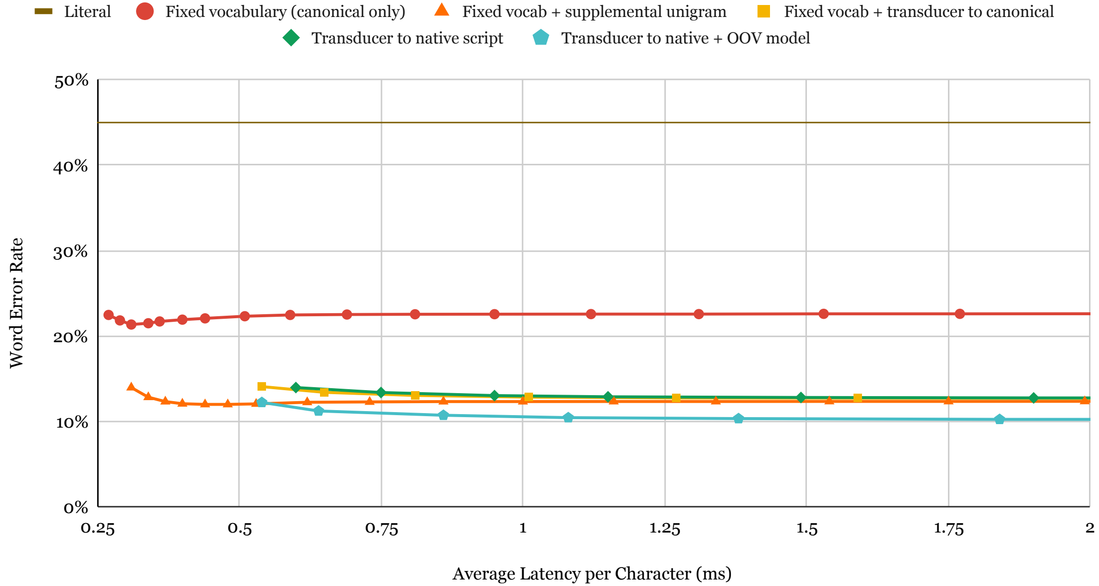


Figure 3: Word error rate versus latency for the models presented in §3 as evaluated on simulated touchpoints for Latin Hindi text. Various decoding-time parametrizations were explored to demonstrate the word error rate–latency tradeoff. The literal baseline shows how accurate the decoding procedure would be without a model at all on this data set.

ing point of 1.0ms per tap (as in Table 1a), the native script transducer with OOV model is the best option, bringing the WER down from 22.6% for the fixed vocabulary to 10.5%, a substantial 54% relative decrease. This does come at the cost of model size, with the model taking up 2.3x as much space. For the case where one chooses a lower operating point such as 0.55ms per tap (as in Table 1b), the supplemental unigram wins out providing a 46% relative decrease in word error rate compared to the fixed vocabulary baseline's WER of 22.5%; all in a relatively compact model taking up only 2.2x more space than the baseline. Additionally, we note that at all operating points, a doubly-sized fixed-vocabulary system (described in §4.1) in fact achieves a slightly worse WER compared with a commensurately sized, otherwise identical language model. We take this as evidence that this model's inability to capture the variant orthographic forms found in this domain is not corrected by simply increasing the model's n-gram count.

Looking at Figure 3, in the limit as latency increases, we find that while the supplemental unigram, fixed vocabulary with transducer to canonical, and transducer to native script converge to similar word error rates of about 12.4 $\sim$ 12.8%, the transducer to native with OOV model can reach even 10.3% WER.

5 Summary

We have presented results for various approaches for handling romanized text entry for South Asian languages within an FST-based mobile keyboard decoder. Compared to baseline methods that naïvely rely upon a single canonical romanization for each word in the vocabulary, we can achieve 54% relative error rate reduction by making use of a transliteration transducer and reading the output from the input tape. Even at very constrained operating points, our best method cuts the error rate nearly in half.

Further, we have demonstrated that an alternative of compactly encoding a large supplemental lexicon in the language model, consisting of alternative romanizations of words, is competitive to the transducer-based normalization, at some space savings. This method has the further virtue of relatively straightforward support for other parts of the keyboard application – such as word prediction and completion, as well as personalization mechanisms – since the decoder outputs from its output tape as in typical operation. Deploying methods presented here that read from the input tape into a keyboard app requires additional integration with these other modules.

References

Umair Z Ahmed, Kalika Bali, Monojit Choudhury, and VB Sowmya. 2011. Challenges in designing input method editors for Indian languages: The role of word-origin and context. In *Proceedings of the Workshop on Advances in Text Input Methods (WTIM 2011)*, pages 1–9.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. Openfst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pages 11–23. Springer.

Maximilian Bisani and Hermann Ney. 2008. Joint-sequence models for grapheme-to-phoneme conversion. *Speech Communication*, 50(5):434–451.

Hsin-Hsi Chen, Sheng-Jie Hueng, Yung-Wei Ding, and Shih-Chung Tsai. 1998. Proper name translation in cross-language information retrieval. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, pages 232–236. Association for Computational Linguistics.

Colin Cherry and Hisami Suzuki. 2009. Discriminative substring decoding for transliteration. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3-Volume 3*, pages 1066–1075. Association for Computational Linguistics.

Dan Garrette and Hannah Alpert-Abrams. 2016. An unsupervised model of orthographic variation for historical document transcription. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 467–472.

Parth Gupta, Kalika Bali, Rafael E Banchs, Monojit Choudhury, and Paolo Rosso. 2014. Query expansion for mixed-script information retrieval. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 677–686. ACM.

Nizar Habash, Mona T Diab, and Owen Rambow. 2012. Conventional orthography for dialectal Arabic. In *LREC*, pages 711–718.

Lars Hellsten, Brian Roark, Prasoon Goyal, Cyril Allauzen, Françoise Beaufays, Tom Ouyang, Michael Riley, and David Rybach. 2017. Transliterated mobile keyboard input via weighted finite-state transducers. In *Proceedings of the 13th International Conference on Finite State Methods and Natural Language Processing (FSMNLP 2017)*, pages 10–19.

Kevin Knight and Jonathan Graehl. 1998. Machine transliteration. *Computational Linguistics*, 24(4):599–612.

Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

Anoop Kunchukuttan, Mitesh Khapra, Gurneet Singh, and Pushpak Bhattacharyya. 2018. Leveraging orthographic similarity for multilingual neural transliteration. *Transactions of the Association of Computational Linguistics*, 6:303–316.

Haizhou Li, Min Zhang, and Jian Su. 2004. A joint source-channel model for machine transliteration. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 159–166.

Mehryar Mohri. 2002. Generic ϵ-removal and input ϵ-normalization algorithms for weighted transducers. *International Journal of Foundations of Computer Science*, 13(01):129–143.

Mehryar Mohri and Michael Riley. 2002. An efficient algorithm for the n-best-strings problem. In *Seventh International Conference on Spoken Language Processing*.

Tom Ouyang, David Rybach, Françoise Beaufays, and Michael Riley. 2017. Mobile keyboard input decoding with finite-state transducers. *arXiv preprint arXiv:1704.03987*.

Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66.

Hassan Sajjad, Helmut Schmid, Alexander Fraser, and Hinrich Schütze. 2017. Statistical models for unsupervised, semi-supervised, and supervised transliteration mining. *Computational Linguistics*, 43(2):349–375.

Tarek Sherif and Grzegorz Kondrak. 2007. Substring-based transliteration. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 944–951.

Ananda Theertha Suresh, Brian Roark, Michael Riley, and Vlad Schogol. 2019. Distilling weighted finite automata from arbitrary probabilistic models. In *Proceedings of the 14th International Conference on Finite State Methods and Natural Language Processing (FSMNLP 2019)*.

Paola Virga and Sanjeev Khudanpur. 2003. Transliteration of proper names in cross-lingual information retrieval. In *Proceedings of the ACL 2003 workshop on Multilingual and mixed-language named entity recognition-Volume 15*, pages 57–64. Association for Computational Linguistics.