Automatic Bilingual Legacy-Fonts Identification and Conversion System

Gurpreet Singh Lehal[1], Tejinder Singh[2], and Saini Pretpal Kaur Buttar[1]

[1] DCS, Punjabi University, Patiala, India

[2] ACTDPL, Punjabi University, Patiala, India

{gslehal, preetpalkaur15}@gmail.com, tej@pbi.ac.in

Abstract. The digital text written in an Indian script is difficult to use as such. This is because, there are a number of font formats available for typing, and these font-formats are not mutually compatible. Gurmukhi alone has more than 225 popular ASCII-based fonts whereas this figure is 180 in case of Devanagari. To read the text written in a particular font, that font is required to be installed on that system. This paper describes a language and font-detection system for Gurmukhi and Devanagari. It also explains a font conversion system for converting the ASCII based text into Unicode. Therefore, the proposed system works in two stages: the first stage suggests a statistical model for automatic language-detection (i.e., Gurmukhi or Devanagari) and font-detection; the second stage converts the detected text into Unicode as per font detection. Though we could not train our systems for some fonts due to non-availability of font converters but system and its architecture is open to accept any number of languages/fonts in the future. The existing system supports around 150 popular Gurmukhi font encodings and more than 100 popular Devanagari fonts. We have demonstrated the effectiveness of font detection is 99.6% and Unicode conversion is 100% in all the cases.

Keywords: n-gram language model, Gurmukhi, Devanagari, Punjabi, Hindi, fonts, font detection, font conversion, Unicode.

1 Introduction

The text on the Internet is available in numerous languages and encodings. These encodings are often not based on any standards. In any NLP application for Indian or any other language, the input text can be processed only after knowing its language and the underlying font-encoding. In many cases this language-encoding is not known in advance and has to be determined. This problem can be viewed as font as well as language identification problem. For languages like Punjabi, Hindi and others, there is no standard font encoding followed by everyone. A large amount of digital text written in Indian languages is in ASCII-based font-formats. It has been found there are many fonts which belong to the same keyboard-mapping, which can be grouped together. Gurmukhi alone has more than 225 popular ASCII-based fonts with 41 keyboard-mappings. In Devanagari, there are more than 180 popular Devanagari ASCII-based font-formats and with 52 different keyboard-mappings. According to Raj and Prahallad[3] the problem of font-identification could be defined as: given a set of words or sentences, identify the font-encoding by finding the minimum distance between the input glyph codes and the models representing font-encodings.

To solve this problem of mutual incompatibility among various ASCII-based font-formats, Unicode was developed as a standard that would assign a unique number known as a code point to every letter in every language. But still the popularity of Unicode is not fully accepted. There are some reasons for this:

- Lack of awareness
- Typing issues in Unicode standards
- Non availability of Unicode typing tools for Indian language
- Media Printing/Publishing houses do not have full support for Unicode
- Lack of variety in Unicode fonts for Indian Languages

In order to find a solution of this problem, we have developed an automatic system to bridge the gap between legacy-fonts and Unicode. Currently, it is working on Gurmukhi and Devanagari languages, but the system and its architecture is open to accept any number of languages/fonts in the future. The proposed system works in two stages: the first stage suggests a statistical n-gram model for automatic font-detection as well as language-detection (i.e., Gurmukhi or Devanagari); the output of this stage is ranked weighted list of trained fonts, from where the topmost font, which has the maximum weight, is selected by the system as the detected font. The second stage converts the text into Unicode as per detected font.

2 Related Work

The problem of font and language identification is addressed by many researchers in the literature [1-5]. The earliest approaches used for automatic language identification were based on unique strings. The proposed mathematical language models relied on orthographic features like characteristic letter sequences and frequencies for each language.

Singh and Gorla [5] presented their work on identifying the languages and encodings of a multilingual document. It involved the steps of monolingual identification, enumeration of languages and then identification of the language of every portion. For enumeration, they have been able to get a precision of 96.20%. Raj and Prahallad [3] have discussed the Term Frequency - Inverse Document Frequency (TF-IDF) weights based approach for font identification. They have modeled a vector space model and TF-IDF weights for each term in the font-data according to its uniqueness. In experiments, they have demonstrated the effectiveness of font data conversion to be as high as 99% on 10 Indian languages and for 37 different font-types. As described earlier, this font size is very small as compared to our research problem.

Font identification in monolingual, bilingual or multilingual system can be seen as a classification task. Like other NLP tasks, we can think of using some sophisticated pattern classification technique such as maximum entropy, for solving this task. But maximum entropy would require training and testing data which is not easy to prepare in our case. As described by Lehal et al. [2], [5] we found that a simpler n-gram model based similarity method is more suitable for this purpose. The advantage of this method is that only a small amount of training data per font-encoding is enough and yields excellent results without using any specially selected features.

3 Problem Complexity

The lack of a standard poses difficulties in processing text written in some non-standard font-formats. The major issues we found are:

- There is no uniform mapping of a character to a code value in Indian languages. Most Indian language fonts assign different codes to same character. For example, consider the word ਪੰਜਾਬੀ, it is internally stored at different keys in different fonts as shown in Table 1.

Table 1. Internal representation of word ਪੰਜਾਬੀ /punjabi/ in different Gurmukhi fonts.

Sr.	Font	ASCII Character Code
1	Akhar	112, 077, 106, 119, 098, 073
2	Gold	102, 046, 117, 106, 087, 103
3	AnandpurSahib	112, 181, 106, 059, 098, 073
4	Asees	103, 122, 105, 107, 112, 104
5	Sukhmani	080, 094, 074, 065, 066, 073
6	Satluj	234, 179, 220, 197, 236, 198

- There is no standard which defines the number of glyphs per character or word in that language, and hence it differs between fonts of a specific language itself. The glyphs are shapes, and when 2 or more glyphs are combined together, they form a character in the scripts of Indian languages. The underlying codes for the individual characters are according to the glyphs they are broken into. This problem is more prevalent in Devanagari as compared to Gurmukhi fonts. The decomposition of glyphs and the codes assigned to them are both different in Devanagari. Table 2 shows how the same word is internally coded in two separate Devanagari fonts, viz., Chanakya and DV-TT-Surekh. In Chanakya, it was five character codes whereas it used seven character codes for same words in DV-TT-Surekh.

Table 2. Code mapping for Devanagari word स्थिति into two different fonts.

Font	**Chanakya**	**DV-TT-Surekh**
Char Code	231 83 205 231 204	202 186 108 201 202 105 201

- The third problem is that there is no standard procedure to align the characters while rendering. For example, consider the Gurmukhi word ਠੂੰ. The order of rendering the glyphs can be: first pivotal character, then bottom character, and then top character; or the top character can be rendered before the bottom character. However, Unicode allows only specific sequence of rendering and hence, some rendering may not be supported in Unicode text and needs to be handled in the final output as shown in figure 1.

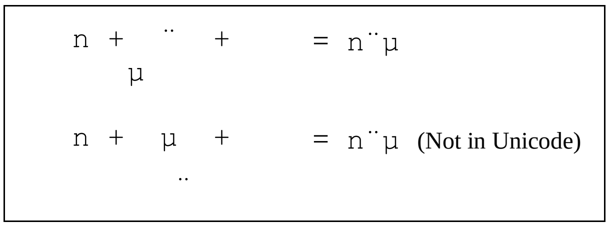


Fig. 1. Different orders of rendering in Gurmukhi fonts.

4 The Proposed System

The proposed system supports around 150 popular Gurmukhi font-encodings and around 100 Devanagari font-encodings. Some of the popular fonts supported are *Akhar, Anmol Lipi, Chatrik, Joy, Punjabi, Satluj, Chanakya, Agra, DV-TT-Yogesh, KrutiDev, Shusha*, etc. A detailed analysis of font encodings showed that many fonts belonging to same keyboard-map have same internal mappings for all the characters. For example, *Akhar* and *Akhar2010* font belong to same keyboard-map family. In fact, we analyzed that all these fonts correspond to 81 unique keyboard-mappings.

The Gurmukhi fonts correspond to 41 keyboard-mappings and Devanagari fonts correspond to 40 keyboard-mappings. It means, if k_0, k_1 ..... k_{80} be the 81 keyboard mappings and f_0, f_1 ...... f_{250} be the fonts, then each of fonts f_i will belong to one of the keyboard mapping k_i. The problem is thus reduced from 250 distinct fonts to just 81 group classes corresponding to each keyboard map. Therefore, our font detection problem is to classify the input text to one of these 81 keyboard mappings. It could be thought of as an 81 class pattern recognition problem. The system for font-detection is based on character-level trigram language model (figure 2).

A raw corpus of around 66,000 words for Gurmukhi and 50,000 words for Devanagari has been used for training the system. Corresponding to each keyboard-map, trigrams have been trained. Font identification is done by extracting the character-level trigrams from the input text and then a score is calculated for each keyboard-mapping which indicates the probability of the keyboard-mapping to be the font of the input text. The keyboard-map having maximum score is identified as the font of the input text and conversion to Unicode text is then performed.

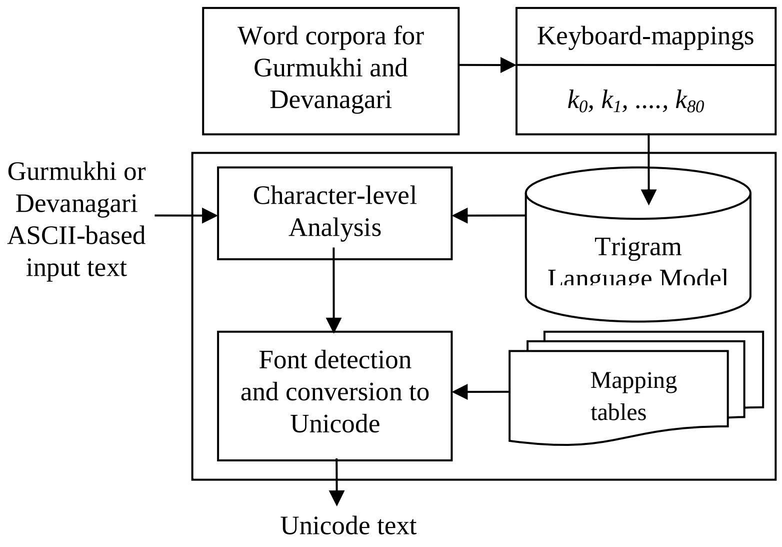

Word corpora for Gurmukhi and Devanagari
Keyboard-mappings
$k_0, k_1, \ldots., k_{80}$
Gurmukhi or Devanagari ASCII-based input text
Character-level Analysis
Trigram Language Model
Font detection and conversion to Unicode
Mapping tables
Unicode text

Fig. 2. Framework for font detection and conversion.

4.1 The Proposed Data Structure

Language and encoding are closely interconnected in such a way that if we could identify the font-encoding, we would most probably have also identified the language. The font detection problem is formulated as character level trigram language model. The single font has 256 character code points. Therefore, in a trigram model, $256 \times 256 \times 256 = 256^3$ code points need to be processed for a single keyboard map. But in order to deal with 81 distinct keyboard maps, the memory requirements will further increase to process and hold 81×256^3 code points. After detailed analysis, it has been found that the array representation of this task is sparse in nature i.e. the majority of code points in omni fonts have zero values. It has been observed that each keyboard map has around 26,000 non-zero code points which is 0.156% of the original code points. Hence, to avoid sparse array formation, binary search tree representation has been created. Each node of the tree contains single unique trigram that exists in one or more of the keyboard-mappings' training data. Every tree node has a linked list associated with it. Each node of this list contains the keyboard-mapping number (k_0, k_1, ..., k_{80}) of the training data in which the corresponding trigram exists and the probability of the trigram in the training data of that keyboard-mapping. The structures of the nodes are expressed in figure 3.

The total number of nodes in the tree are 4,41,648. The height of the tree is 80. The shortest list linked to a node in the tree has length 1, i.e., the shortest linked list contains 1 node. The longest node contains 76 nodes. The average number of nodes in the linked lists is found to be 4.

The major advantage of using this representation is performance enhancement. As we know, searching in a binary search tree is faster as compared to that in a linked list. The structure is dynamic, as the lists linked to the nodes of the trees have their sizes dependent on the number of fonts which have non-zero probability of the trigram stored at the node. In other words, length of the list of fonts is not fixed to be equal to the total number of fonts. This is because most fonts can have non-zero probability for that trigram at the node. Moreover, by combining the list of those fonts in which the trigram stored at the tree-node exists, we access the probabilities of that trigram in all the fonts right there. Additionally, this architecture is open to add more language/fonts without affecting the performance.

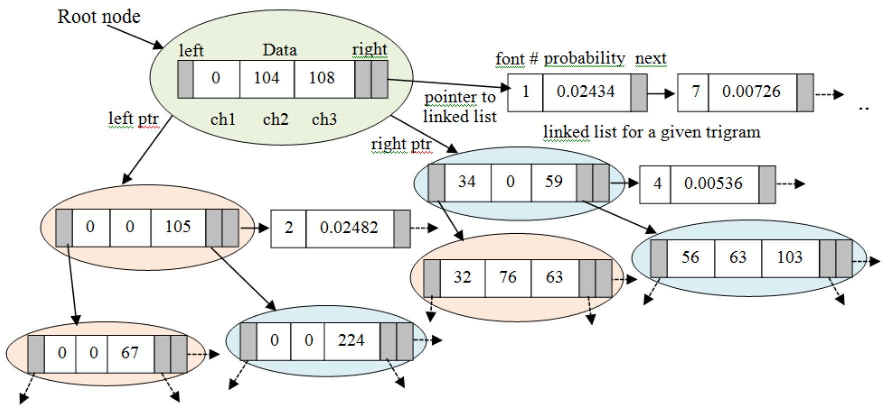


Fig. 3. Binary search tree node structure.

Adding Tree Node: To store trigrams and their probability in the data structure, the tree is searched for each trigram one by one. Let us suppose, the tree is being searched for the trigram (0, 104, and 108). If the trigram is found in the tree, we access the linked list of that trigram and add a new node containing the font number and the probability of that trigram in that font. If the trigram is not found, a new node of the tree is created containing that trigram and a linked list containing the font number and probability of the trigram.

Searching the Binary Tree: Suppose, we have a word ਕਲਮ, whose font is to be detected. First, the system will break down this word into trigrams. Then, the system will search for each trigram in the binary search tree one by one. Thus, first it searches for the trigram (0, 0, and 107). On finding the trigram, it goes to the list linked with that trigram's node. There it finds the probabilities of that trigram in all the fonts from 0 to 80. For example, the trigram (0, 0, and 107) has non-zero probabilities in the font-numbers 1, 2, 11, and 23 which are 0.2434, 0.435, 0.04364, and 0.06433 respectively. For all the other fonts, probability of this trigram is considered to be zero. Similarly, the system searches all other trigrams and collects their probabilities.

5 Font Detection

The task of font detection can be classified as an n-gram based text similarity method. The advantage of this method is that only a small amount of training data per font encoding is enough to get the desired results. As discussed earlier we have created a raw corpus of around 50000 words for training the system. The proposed methods of font detections are:

- Word level prediction
- Character level prediction
- Hybrid Approach using Unseen probability

The following two methods have been used to calculate the score of a keyboard-mapping for the prediction of font of the input text.

5.1 Word Level Prediction

The trigram probability of a word w^l of length l is calculated as a product of trigram probability for all possible combinations as shown in equation 1. Clearly, if any trigram has zero probability then the probability of the word becomes zero. The total probability of the input text of n words, for defined keyboard-mapping k is calculated as sum of all words probabilities as shown in equation 2 and the best prediction for keyboard-mapping index is detected corresponding to the maximum weight of probability as shown in equation 3.

$$w^l = \prod_{i=1}^{l} P(c_i \mid c_{i-1}, c_{i-2}) \qquad (1)$$

$$W_k = w_{1,n} = \sum_{1}^{n} w^l, \qquad (2)$$

$$K = argmax\ W_k \qquad (3)$$

For example, consider a sentence of four words ਪਾਣੀ ਵਿਅਰਥ ਨਾ ਵਹਾਓ. In the word level prediction, the probability of the word ਪਾਣੀ can be calculated as:

$$w^4 = \prod_{i=1}^{4} P(c_i \mid c_{i-1}, c_{i-2}) = 0.009001276 * 0.000873424 * 0.000085518 * 0.000198584 * 0.000591988 * 0.023062862 = 1.822875e\text{-}18$$

Similarly, the total probabilities of remaining three words are calculated and the overall probability of the whole sentence is:

$$W_1 = w_{1,4} = 1.822875e\text{-}18 + 8.063544e\text{-}28 + 2.290603e\text{-}10 + 4.744855e\text{-}28 = 2.290603e\text{-}10$$

In this way, the probability of all input words is calculated for all trained fonts and the detected keyboard-mapping index pops up corresponding to the maximum weight of probability.

5.2 Character Level Prediction

Unlike word level prediction, this method considers all trigram probabilities at character level not at word level. In other words, the total probability of the input text for keyboard-mapping k is calculated as sum of all valid trigrams of the input text as shown in equation 4 and the detected font index is corresponding to the maximum weight of probability as $K = argmax\ C_k$.

$$C_k = \sum_{i=1}^{n} P(c_i \mid c_{i-1}, c_{i-2}), \quad (4)$$

Using the same example as given above, there are 23 trigrams corresponding to sentence of four words and the probability of the text comes out to be:

$$C_1 = \sum_{i=1}^{23} P(c_i \mid c_{i-1}, c_{i-2}) = 0.102334869$$

5.3 Hybrid Approach using Unseen probability

Unseen Probability factor (*B_k*). The above two methods detect the probability of the input text in the training data. On the other hand, we can think of another way out to calculate the probability, i.e., how much of the input text does not belong to the training data. This can be seen as a probability of unseen trigrams. We can quantify this by using a variable 'B_{ck}' for each trained font, initialized to 0, which is incremented by one for an unseen trigram, as shown in equation 5. Then the unseen probability can be calculated with the equation 6.

$$B_{ck} = \sum_{i:P(c_i \mid c_{i-1}, c_{i-2})=0} 1 \quad (5)$$

$$B_k = 1 - [B_{ck} / (\text{Total number of trigrams})] \quad (6)$$

Now, the two proposed methods have been configured to incorporate unseen probability factor, so that each contributes towards the final selection. Now, the overall probability of the word- and character-level methods will be combined as shown in equation 7 and 8 respectively. The value of α is used to determine each factor's contribution towards the final outcome.

$$K = argmax\ (\alpha * W_k + (1 - \alpha) * B_k), \text{ where } 0 \le \alpha \le 1 \quad (7)$$

$$K = argmax\ (\alpha * C_k + (1 - \alpha) * B_k), \text{ where } 0 \le \alpha \le 1 \quad (8)$$

It has been concluded from the test results that the hybrid methods gave more accurate results when the value of α was kept equal to or slightly less than 0.3. That is, the unseen-trigram factor was contributing more towards the oveall probability.

6 Language Detection

The total number of keyboard-mappings supported by the system is 81. The keyboard mappings ranging 0 to 40 correspond to Gurmukhi fonts and the rest 41 to 80 correspond to Devanagari. Thus, the language of the input text is Punjabi, if the detected keyboard mapping is lying anywhere in the range 0 to 40, and Hindi if otherwise.

7 Conversion to Unicode

Unicode conversion is performed by mapping all fonts to a single intermediate form with the help of mapping tables generated for each font. Then, the rule based approach is followed for conversion of text from intermediate form to Unicode. Many rules have been formulated for proper rendering of text in Unicode format. The steps for font-data conversion are explained as follows:

7.1 Mapping Tables and Intermediate Form

For performing the conversion of input text into Unicode text on the basis of detected font encoding, first, a mapping table of the detected font encoding is used. The mapping tables are static alignments between all font glyphs and an intermediate form. The intermediate form is a list of all the glyphs which exist in a particular language. These glyphs are assigned an internal code. A codepoint corresponding to a gylph in a font is mapped to an internal code corresponding to the same glyph in the intermediate form. In this manner, the mapping tables have been built for all the trained fonts. The advantage of using intermediate form is that it reduces the complication, as we have to create less number of rules because similar glyphs are mapped onto single internal code in the intermediate form.

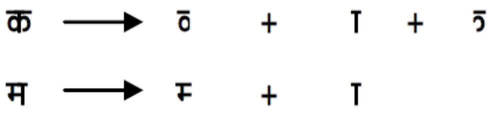
For example, consider the *Chanakya* font's glyph % and another glyph combination ^Ù which means the same, but there is minor difference in shape. But Unicode transformation of both the glyphs is same. Hence, these types of glyphs are treated as one in our intermediate and Unicode transformations. Also, it is easy to add a new font to the system as we need only a training file and a mapping table of that font. There is no need to create new transformations rules for the newly added font as transformation to Unicode is not done directly from font to Unicode, but from intermediate form to Unicode. We only need to build a trigram-probability training file and a mapping table for the new font.

The input text written in the detected font is converted into the intermediate form using the mapping table for that font.

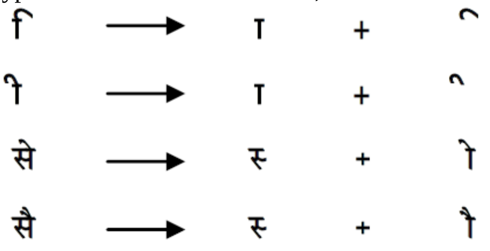
7.2 Intermediate Form to Unicode Conversion

The next step is the conversion of intermediate form into Unicode text. The conversion to Unicode is not straight forward due to complex Unicode transformation rules. We have crafted various conversion rules to perform this transformation.

Handling full characters in Devanagari fonts. In some Devanagari fonts, there are no separate glyphs for full-characters. They are formed by combining other glyphs together. For example, in *APS-C-DV-Prakash* font,



Similarly, other glyph combinations are formed, such as:



To handle such combinations, rules have been formulated, for example, we know that a half-character cannot be combined with a *matra*, it exists only in conjunction with a full-character. Thus, if there exists a half-character in combination with a *matra* i.e., ा, ो, or ौ, then it should be converted into the corresponding full-character applying corresponding *matra*, if any, such as none, े, ै respectively.

Transforming Long Vowels. Next, rules have been devised for transforming Gurmukhi long vowels ਉ[ʊ], ਊ[u], ਓ[o], ਆ[ɑ], ਇ[ɪ], ਈ[i], ਏ[e], ਐ[æ], ਔ[ɔ]. This is because, in Unicode these nine independent vowels with three bearer characters Ura ੳ[ʊ], Aira ਅ [ə] and Iri ੲ[ɪ] have single code points and need to be mapped accordingly as shown in Figure 5. Similarly, Devanagari independent long vowels आ[ɑ], ई[i], ऊ[u], ऍ[eɪ], ऐ[æ], ऑ[ɒ], ओ[o], औ[ɔ] with four bearer characters अ[ə], इ[ɪ], उ[ʊ] and ए[e], and short vowel ॉ[ɒ] were mapped accordingly.

Rendering half-character in Unicode. There are no explicit code points for half-characters in Unicode. They will be generated automatically by the Unicode rendering system when it finds special symbol called *halant* or *virama* [्]. Therefore, Gurmukhi/Devanagari transformation of subjoined consonants is performed by prefixing *halant* ् symbol along with the respective consonant, as shown in Figure 4.

Handling of short vowel ਿ. Mapping of short vowel sign ਿ[ɪ] has been done according to Unicode rendering system. The Unicode transformation becomes complex when short vowel ਿ[ɪ] and subjoined consonants come together at a single word position.

Long Vowels	ਅ + ਾ → ਆ		ਉ + ੁ → ਉ		ੲ + ਿ → ਇ	
	ਅ + ੈ → ਐ		ਉ + ੂ → ਊ		ੲ + ੀ → ਈ	
	ਅ + ੌ → ਔ		ਉ + ੋ → ਓ		ੲ + ੇ → ਏ	
	अ + ा → आ		इ + ʿ → ई		ए + े → ऐ	
	अ + ो → ओ		अ + ौ → औ		ए + ॅ → ऍ	
	ा + ॅ → ॉ		अ + ा + ॅ → ऑ		। + । → ॥	
Short Vowel	ਕ + ਿ → ਕਿ		क + ि → कि			
Nukta	ਸ + ਼ → ਸ਼	ਖ + ਼ → ਖ਼	ਗ + ਼ → ਗ਼	ਜ + ਼ → ਜ਼	ਫ + ਼ → ਫ਼	ਲ + ਼ → ਲ਼
Symbol	क + ़ → क़	ख + ़ → ख़	ग + ़ → ग़	ज + ़ → ज़	ड + ़ → ड़	ढ + ़ → ढ़
	फ + ़ → फ़	य + ़ → य़	न + ़ → ऩ	र + ़ → ऱ	ळ + ़ → ऴ	
Subjoined	ਨ + ੍ + ਹ → ਨ੍ਹ		ਪ + ੍ + ਰ → ਪ੍ਰ		ਸ + ੍ + ਵ → ਸ੍ਵ	
Consonants	ट + ् + र → ट्र		प + ् + र → प्र		स + ् + व → स्व	

Fig. 4. Unicode transforming rules.

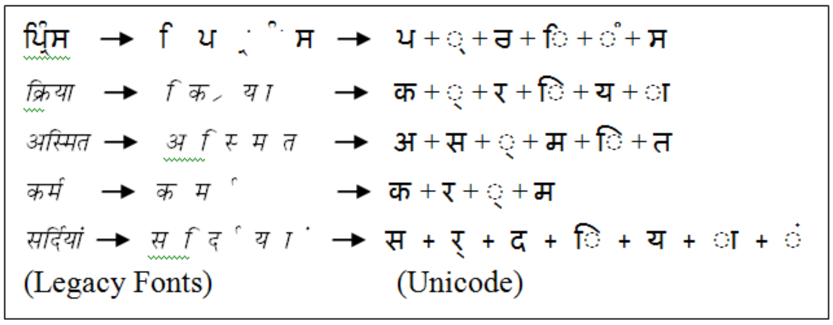


Fig. 5. Complex Unicode transformations.

For example, consider the Gurmukhi word ਪ੍ਰਿੰਸ .In omni fonts, [ਿ] appears as the first character. But according to Unicode rendering, it must be after the bearing consonant. Therefore, it must go after the subjoined consonant ੍ + ਹ, as shown in Figure 5.

Similarly, when half-character comes in between [ि] and a full-character, then according to Unicode rendering system, half-character must be written first, then the full-character and [ि] at the last position as shown in the word अस्मित in Figure 5.

Other Issues. Some Devanagari fonts consist of glyphs depicting some combinations of matras, such as िँ िँ िं . Consider the word शर्मिंदा this word uses matra िँ . In order to handle this word as per Unicode rendering, the following rule-based transformations are done:

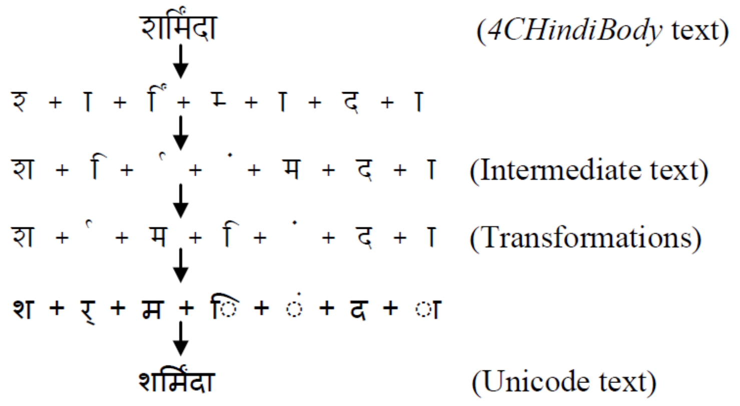


Fig. 6. शर्मिंदा Word transformations while converting to Unicode.

In Unicode, there are no explicit glyphs for rendering complex consonants, such as क्ष, श्र etc. These are generated by using combinations of consonants with halant (्) as shown in Figure 7.

क्ष = क + ् + ष   त्र = त + ् + र

ज्ञ = ज + ् + ञ   श्र = श + ् + र

Fig. 7. Rendering of complex consonants in Unicode.

8 Evaluation and Results

8.1 Test Data Preparation

For evaluation purpose, we collected random text written in Gurmukhi and Devanagari from 27 different sources, for example, AmarUjala, BBC Hindi News, Punjabi Tribune, etc. The data was categorized under five sections namely, articles, books, news, poems and stories. For each section, 4 sets of about 1000 words each were prepared for each trained font. The fonts for which real ASCII-based data was not available, font-converters were used to convert Unicode text into font-data.

8.2 Font Detection Results

All the methods are then tested over the input-text and the results of font detection among proposed methods are shown in Table 3. Clearly, character-level prediction method has shown better results in both Gurmukhi as well as Devanagari font detection as compared to word-level prediction.

Table 3. Font detection accuracy of different methods.

Data Set Domain (200 words)	**Word-level**		**Char-level**		**Hybrid Word-level (α=0.3)**		**Hybrid Char-level (α=0.3)**	
	Gur	**Dev**	**Gur**	**Dev**	**Gur**	**Dev**	**Gur**	**Dev**
Articles	51	75	95	90	100	98	100	100
Books	51	68	83	93	98	100	100	100
News	76	78	66	90	100	100	100	100
Poems	51	73	73	90	98	98	98	100
Stories	78	50	76	90	98	100	100	98

The hybrid approach has shown the overall improvement in font detection as compared to single method approach. Unseen probability factor has shown better results. The optimal value of system combination factor α is found to be 0.3. Again, it has been found that hybrid character-level method is the best among all the other methods we have discussed. The average detection results of this method are 99.6% for both Gurmukhi and Devanagari fonts. It is also come out from the results that the proposed n-gram approach with limited training data has successfully overcome the confusion amongst Gurmukhi and Devanagari fonts. It has been found that for the input text of any size, we need at most first 200 words for the task of font-detection. This factor contributes towards the improvement of performance of the system.

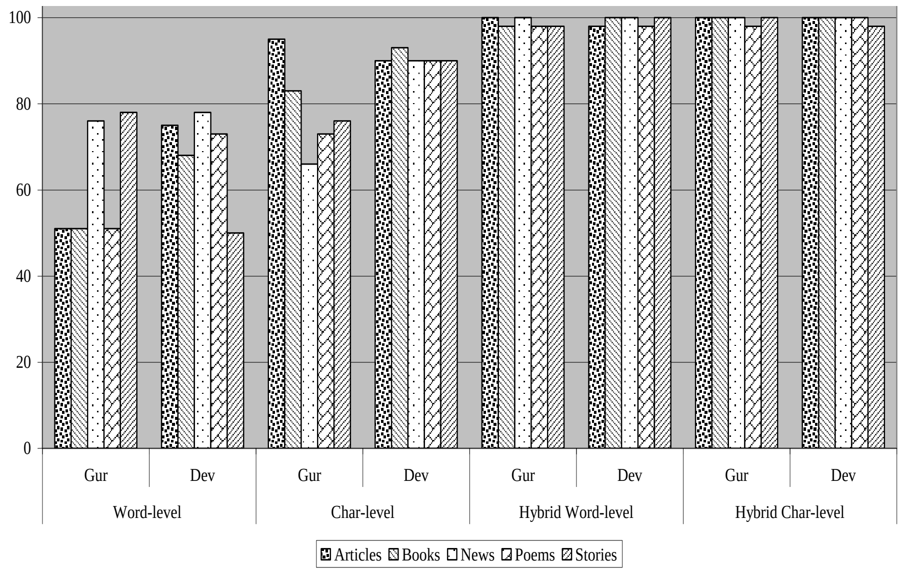


Fig. 8. Font detection accuracy of different methods.

Conversion Accuracy of the system is found to be 100% in all the test data.

Font Detection vs. Input word length. The figure 8 shows the trend of detection of Gurmukhi and Devanagari fonts when varying number of most frequent words in a language are taken as input to the system. The results shown in the following graph are based on the best detection method.

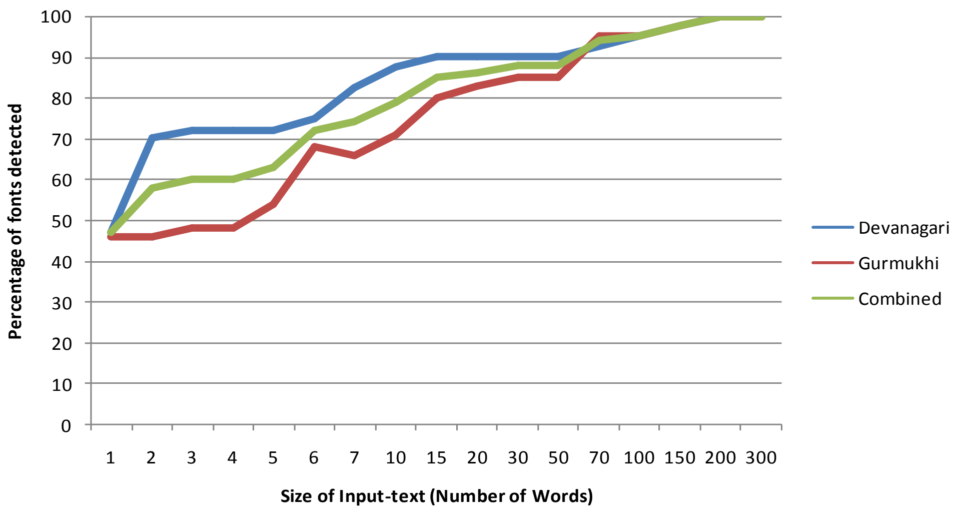


Fig. 9. Comparison of font-detection at varying sizes of input-text.

The different word sizes are from 1 to 300 words with a random interval. According to our observation the system is capable to identify 45% fonts correctly with just single word as input. Next, with two input words, there is sharp rise in Devanagari font detection from 45% to 70% while the Gurmukhi font detection is not affected. There is nearly 70% detection in Gurmukhi fonts is when the input size is at least six words. Unlike Devanagari fonts Gurmukhi fonts, detection decreases when input words are increased from six to eight. In overall trend we can say that during the input word size 1 to 70, the percentage of Devanagari font detection is always higher than the Gurmukhi font detection and reverse trend is seen between input word sizes greater than 70 to 125. Finally, it has been observed that input word size of around 200 words is reasonably sufficient for the system to give best detection results.

9 Conclusion

This paper describes a language and font detection system for Gurmukhi and Devanagari. It also delineates a font conversion system for converting the ASCII based text into Unicode. Hence, the proposed system works in two stages: the first stage suggests a statistical model for automatic language detection (i.e., Gurmukhi or Devanagari) and font-detection; the second stage converts the detected text into Unicode. The existing system supports around 150 popular Gurmukhi font-encodings and more than 100 popular Devanagari fonts. We have demonstrated the effectiveness of font detection is 99.6% and Unicode conversion is 100% in all the cases. Though we could not train our systems for some fonts due to non-availability of font converters but system and its architecture is open to accept any number of languages/fonts in the future without affecting its speed and performance.

References

1. Chaudhury, S., Sen S., Nandi, G.R.: A Finite State Transducer (FST) based Font Converter. International Journal of Computer Applications, Volume 58, No. 17, pp. 35–39 (2012)
2. Saini, T. S., Lehal, G. S., Chowdhary, S.K.: An Omni-font Gurmukhi to Shahmukhi Transliteration System, In: Proceedings of Conference on Computational Linguistics (COLING), pp. 313–319, Mumbai (2012)
3. Raj, A.A., Prahallad, K.: Identification and Conversion of Font-Data in Indian Languages, In: International Conference on Universal Digital Library (ICUDL2007), Pittsberg, USA (2007)
4. Singh, A.K.: Study of some distance measures for language and encoding identification, In: Proceedings of the Workshop on Linguistic Distances, Sydney, Australia, pp. 63–72, ACL (2006)
5. Singh, A.K., Gorla, J..: Identification of languages and encodings in a multilingual document. In: Proceedings of the 3rd ACL SIGWAC Workshop on Web As Corpus, Lovain-la-Neuve, Belgium (2007)
6. Cavnar, W. B., Trenkle J. M.: N-Gram-Based Text Categorization, In: Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval, pp. 161–175 (1994)
7. Davis, M., Whistler, K. (Eds.): Unicode Normalization Forms. Technical Reports, Unicode Standard Annex #15, Revision 33. (2010) Retrieved February 22, 2011, from http://www.unicode.org/reports/tr15/tr15-33.html
8. Bharati, A., Sangal, N., Chaitanya, V., Kulkarni, A.P., Sangal, R.: Generating converters between fonts semi-automatically, In: Proceedings of SAARC conference on Multi-lingual and Multi-media Information Technology, CDAC, Pune, India (1998)
9. Kikui, G.: Identifying the coding system and language of on-line documents on the Internet. In: Proceedings of Conference on Computational Linguistics (COLING), pp. 652–657 (1996)