Sanskrit Sandhi Splitting using *seq*2(*seq*)^2

Rahul Aralikatte IBM Research
Neelamadhav Gantayat IBM Research
Naveen Panwar IBM Research
{rahul.a.r, neelamadhav, naveen.panwar}@in.ibm.com

Anush Sankaran IBM Research
anussank@in.ibm.com

Senthil Mani IBM Research
sentmani@in.ibm.com

Abstract

In Sanskrit, small words (morphemes) are combined to form compound words through a process known as *Sandhi*. Sandhi splitting is the process of splitting a given compound word into its constituent morphemes. Although rules governing word splitting exists in the language, it is highly challenging to identify the location of the splits in a compound word. Though existing Sandhi splitting systems incorporate these pre-defined splitting rules, they have a low accuracy as the same compound word might be broken down in multiple ways to provide syntactically correct splits.

In this research, we propose a novel deep learning architecture called Double Decoder RNN (DD-RNN), which (i) predicts the location of the split(s) with 95% accuracy, and (ii) predicts the constituent words (learning the Sandhi splitting rules) with 79.5% accuracy, outperforming the state-of-art by 20%. Additionally, we show the generalization capability of our deep learning model, by showing competitive results in the problem of Chinese word segmentation, as well.

1 Introduction

Compound word formation in Sanskrit is governed by a set of deterministic rules following a well-defined structure described in *Pāṇini's Aṣṭādhyāyī*, a seminal work on Sanskrit grammar. The process of merging two or more morphemes to form a word in Sanskrit is called *Sandhi* and the process of breaking a compound word into its constituent morphemes is called *Sandhi splitting*. In Japanese, *Rendaku* ('sequential voicing') is similar to Sandhi. For example, *'origami'* consists of *'ori'* (paper) + *'kami'* (folding), where *'kami'* changes to *'gami'* due to *Rendaku*.

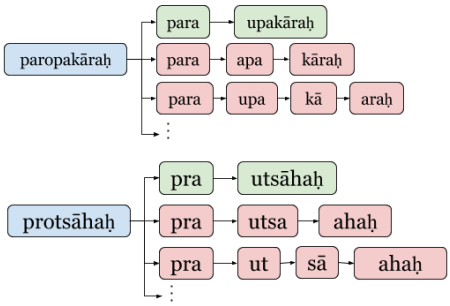

Figure 1: Different possible splits for the word *paropakāraḥ* and *protsāhaḥ*, provided by a standard Sandhi splitter.

Learning the process of sandhi splitting for Sanskrit could provide linguistic insights into the formation of words in a wide-variety of Dravidian languages. From an NLP perspective, automated learning of word formations in Sanskrit could provide a framework for learning word organization in other Indian languages, as well (Bharati et al., 2006). In literature, past works have explored sandhi splitting (Gillon, 2009) (Kulkarni and Shukl, 2009), as a rule based problem by applying the rules from *Aṣṭādhyāyī* in a brute force manner. Consider the example in Figure 1 illustrating the different possible splits of a compound word *paropakāraḥ*. While the correct split is *para* + *upakāraḥ*, other forms of splits such as, *para* + *apa* + *kāraḥ* are syntactically possible while semantically incorrect[1]. Thus, knowing all the rules of splitting is insufficient and it is essential to identify the location(s) of split(s) in a given compound word.

In this research, we propose an approach for au-

[1]Different syntactic splits given by one of the popular Sandhi splitters: https://goo.gl/0M5CPS and https://goo.gl/JHnpJw

tomated generation of split words by first learning the potential split locations in a compound word. We use a deep bi-directional character RNN encoder and two decoders with attention, $\boldsymbol{seq2(seq)^2}$. The accuracy of our approach on the benchmark dataset for split location prediction is 95% and for split words prediction is 79.5% respectively. To the best of our knowledge, this is the first research work to explore deep learning techniques for the problem of Sanskrit Sandhi splitting, along with producing state-of-art results. Additionally, we show the performance of our proposed model for Chinese word segmentation to demonstrate the model's generalization capability.

2 $\boldsymbol{seq2(seq)^2}$: Model Description

In this section, we present our double decoder model to address the Sandhi splitting problem. We first outline the issues with basic deep learning architectures and conceptually highlight the advantages of the double decoder model.

2.1 Issues with standard architectures

Consider an example of splitting a sequence *abcdefg* as *abcdx + efg*. The primary task is to identify *d* as the split location. Further, for a given location *d* in the character sequence, the algorithm should take into account (i) the context of character sequence *abc*, (ii) the immediate previous character *c*, (iii) the immediate succeeding character *e*, to make an effective split. For such sequence learning problems, RNNs have become the most popular deep learning model (Pascanu et al., 2013) (Sak et al., 2014).

A basic RNN encoder-decoder model (Cho et al., 2014) with LSTM units (Hochreiter and Schmidhuber, 1997), similar to a machine translation model, was trained initially. The compound word's characters is fed as input to the encoder and is translated to a sequence of characters representing the split words ('+' symbol acts as a separator between the generated split words). However, the model did not yield adequate performance as it encoded only the context of the characters that appeared before the potential split location(s). Though we tried making the encoder bi-directional (referred to as B-RNN), the model's performance only improved marginally. Adding global attention (referred to as B-RNN-A) to the decoder enabled the model to attend to the characters surrounding the potential split location(s) and improved the split prediction performance, making it comparable with some of the best performing tools in the literature.

2.2 Double Decoder RNN (DD-RNN) model

The critical part of learning to split compound words is to correctly identify the location(s) of the split(s). Therefore, we added a two decoders to our bi-directional encoder-decoder model: (i) *location decoder* which learns to predict the split locations and (ii) *character decoder* which generates the split words. A compound word is fed into the encoder character by character. Each character's embedding x_i is passed to the encoders LSTM units. There are two LSTM layers which encode the word, one in forward direction and the other backward. The encoded context vector e_i is then passed to a global attention layer.

In the first phase of training, only the *location decoder* is trained and the *character decoder* is frozen. The character embeddings are learned from scratch in this phase along with the attention weights and other parameters. Here, the model learns to identify the split locations. For example, if the inputs are the embeddings for the compound word *protsāhaḥ*, the location decoder will generate a binary vector $[0, 0, 1, 0, 0, 0, 0, 0, 0]$ which indicates that the split occurs between the third and fourth characters. In the second phase, the *location decoder* is frozen and the *character decoder* is trained. The encoder and attention weights are allowed to be fine-tuned. This decoder learns the underlying rules of Sandhi splitting. Since the attention layer is already pre-trained to identify potential split locations in the previous phase, the *character decoder* can use this context and learn to split the words more accurately. For example, for the same input word *protsāhaḥ*, the *character decoder* will generate [p, r, a, +, u, t, s, ā, h, a, ḥ] as the output. Here the character *o* is split into two characters *a* and *u*.

In both the training phases, we use negative log likelihood as the loss function. Let X be the sequence of the input compound word's characters and Y be the binary vector which indicates the location of the split(s) in the first phase and the true target sequence of characters which form the split words in the second phase. If $Y = y_1, y_2, ..., y_n$, then the loss function is defined as:

$$loss = -\sum_{i=1}^{|Y|} \log P(y_i | y_{i-1}, \cdots, y_1, X)$$

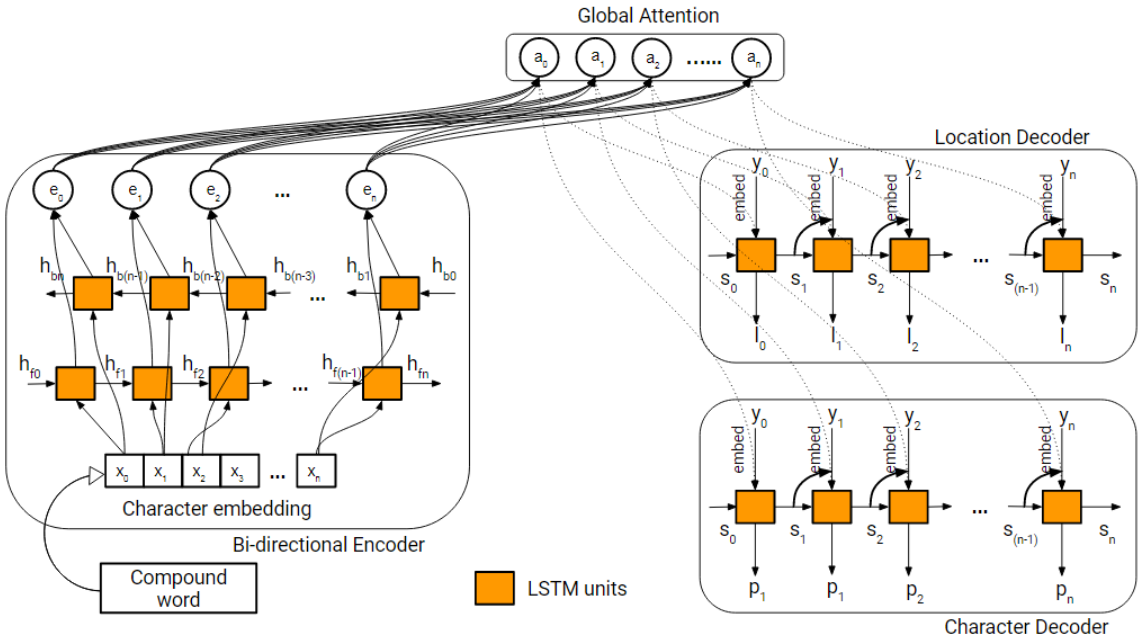

Figure 2: The bi-directional encoder and decoders with attention

We evaluate the DD-RNN and compare it with other tools and architectures in Section 4.

2.3 Implementation details

The architecture of the DD-RNN is shown in Figure 2. We used a character embedding size of 128. The bi-directional encoder and the two decoders are 2 layers deep with 512 LSTM units in each layer. A dropout layer with $p = 0.3$ is applied after each LSTM layer. The entire network is implemented in Torch [2].

Of the $71,747$ words in our benchmark dataset, we randomly sampled 80% of the data for training our models. The remaining 20% was used for testing. We used stochastic gradient descent for optimizing the model parameters with an initial learning rate of 1.0. The learning rate was decayed by a factor of 0.5 if the validation perplexity did not improve after an epoch. We used a batch size of 64 and trained the network for 10 epochs on four Tesla K80 GPUs. This setup remains the same for all the experiments we conduct.

3 Existing Datasets and Tools

In this section, we briefly introduce various Sanskirt Sandhi datasets and splitting tools available in literature. We also discuss the tools' drawbacks and the major challenges faced while creating such tools.

Datasets: The *UoH corpus*, created at the University of Hyderabad[3] contains $113,913$ words and their splits. This dataset is noisy with typing errors and incorrect splits. The recent *SandhiKosh corpus* (Shubham Bhardwaj, 2018) is a set of $13,930$ annotated splits. We combine these datasets and heuristically prune them to finally get $71,747$ words and their splits. The pruning is done by considering a data point to be valid only if the compound word and it's splits are present in a standard Sanskrit dictionary (Monier-Williams, 1970). We use this as our benchmark dataset and run all our experiments on it.

Tools: There exist multiple Sandhi splitters in the open domain such as (i) *JNU splitter* (Sachin, 2007), (ii) *UoH splitter* (Kumar et al., 2010) and (iii) *INRIA sanskrit reader companion* (Huet, 2003) (Goyal and Huet, 2013). Though each tool addresses the splitting problem in a specialized way, the general principle remains constant. For a given compound word, the set of all rules are applied to every character in the word and a large potential candidate list of word splits is obtained. Then, a morpheme dictionary of Sanskrit words is used with other heuristics to remove infeasible

[2] http://torch.ch/

[3] Available at: http://sanskrit.uohyd.ac.in/Corpus/

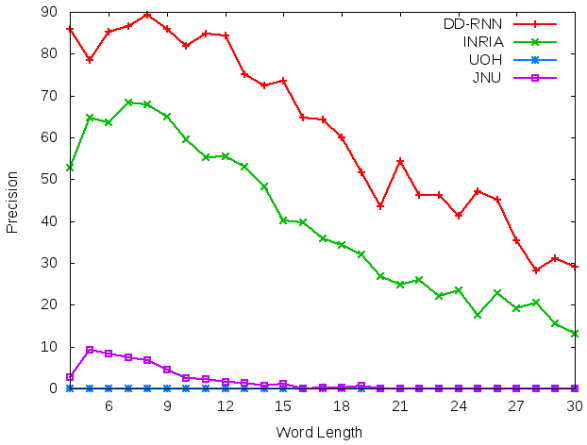

Figure 3: Top-1 split prediction accuracy comparison of different publicly available tools with DD-RNN

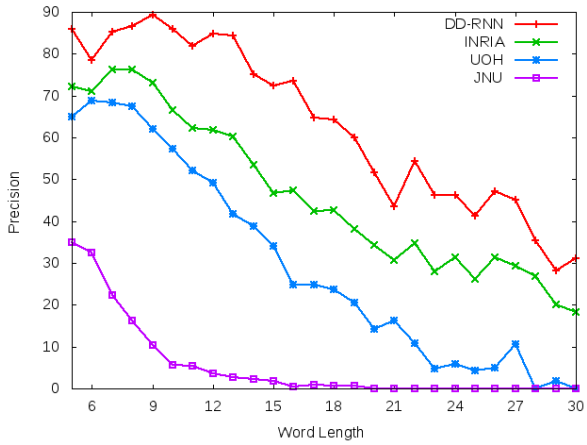

Figure 4: Split prediction accuracy comparison of different publicly available tools (Top-10) with DD-RNN (Top-1)

word split combinations. However, none of the approaches address the fundamental problem of identifying the location of the split before applying the rules, which will significantly reduce the number of rules that can be applied, hence resulting in more accurate splits.

4 Evaluation and Results

We evaluate the performance of our DD-RNN model by: (i) comparing the **split prediction** accuracy with other publicly available sandhi splitting tools, (ii) comparing the **split prediction** accuracy with other standard RNN architectures such as RNN, B-RNN, and B-RNN-A, and (iii) comparing the **location prediction** accuracy with the RNNs used for Chinese word segmentation (as they only predict the split locations and do not learn the rules of splitting)

4.1 Comparison with publicly available tools

The tools discussed in Section 3 take a compound word as input and provide a list of all possible splits as output (UoH and INRIA splitters provide weighted lists). Initially, we compared only the top prediction in each list with the true output. This gave a very low precision for the tools as shown in Figure 3. Therefore, we relaxed this constraint and considered an output to be correct if the true split is present in the top ten predictions of the list. This increased the precision of the tools as shown in Figure 4 and Table 1.

Even though DD-RNN generates only one output for every input, it clearly out-performs the other publicly available tools by a fair margin.

Model	Accuracy (%)	
	Location Prediction	**Split Prediction**
JNU (Top 10)	-	8.1
UoH (Top 10)	-	47.2
INRIA (Top 10)	-	59.9
RNN	79.10	56.6
B-RNN	84.62	58.6
B-RNN-A	88.53	69.3
DD-RNN	**95.0**	**79.5**
LSTM-4	70.2	-
GRNN-5	67.7	-

Table 1: Location and split prediction accuracy of all the tools and models under comparison

4.2 Comparison with standard RNN architectures

To compare the performance of DD-RNN with other standard RNN architectures, we trained the following three models to generate the split predictions on our benchmark dataset: (i) unidirectional encoder and decoder without attention (RNN), (ii) bi-directional encoder and decoder without attention (B-RNN), and (iii) bi-directional encoder and decoder with attention (B-RNN-A)

As seen from the middle part of Table 1, the DD-RNN performs much better than the other architectures with an accuracy of **79.5%**. It is to be noted that B-RNN-A is the same as DD-RNN without the *location decoder*. However, the accuracy of DD-RNN is 14.7% more than that the B-RNN-A and consistently outperforms B-RNN-

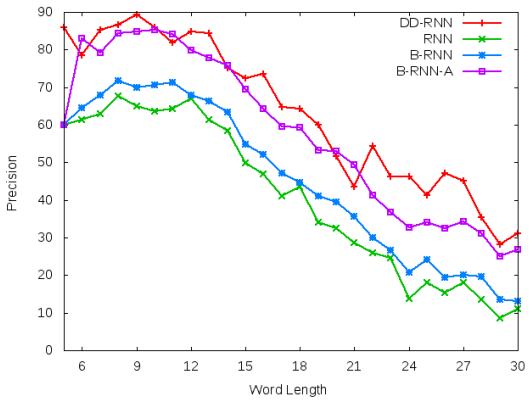

Figure 5: Split prediction accuracy comparison of different variations of RNN on words of different lengths

A on almost all word lengths (Figure 5). This indicates that the attention mechanism of DD-RNN has learned to better identify the split location(s) due to its pre-training with the *location decoder*.

4.3 Comparison with similar works

(Reddy et al., 2018) propose a *seq2seq* model with attention to tackle the Sandhi problem. Their model is similar to B-RNN-A and is outperformed by our proposed DD-RNN by $\tilde{6}.47\%$. We also compared our proposed DD-RNN with a unidirectional LSTM with a depth of 4 (Chen et al., 2015b) (**LSTM-4**) and a Gated Recursive Neural Network with a depth of 5 (Chen et al., 2015a) (**GRNN-5**). These models were used to get state of the art results for Chinese word segmentation and their source code is made available online.[4] Since these models can only predict the location(s) of the split(s) and cannot generate the split words themselves, we used the location prediction accuracy as the metric. We trained these models on our benchmark dataset and the results are shown in Table 1. DD-RNN's precision is 35.3% and 40.3% better than LSTM-4 and GRNN-5 respectively. Conversely, we trained the DD-RNN for the Chinese word segmentation task to test the generalizability of the model. Since there are no morphological changes during segmentation in Chinese, the character decoder is redundant and the model collapses to simple *seq2seq*. We used the PKU dataset which is also used in (Chen et al., 2015b) & (Chen et al., 2015a) and obtained an accuracy of 64.25% which is comparable to the results of other standard models.

[4] https://github.com/FudanNLP

To summarize, we have used our benchmark dataset to compare the DD-RNN model with existing publicly available Sandhi splitting tools, other RNN architectures and models used for Chinese word segmentation task. Among the existing tools, the INRIA splitter gives the highest split prediction accuracy of 59.9%. Among the standard RNN architectures, B-RNN-A performs the best with a split prediction accuracy of 69.3%. LSTM-4 performs the best among the Chinese word segmentation models with a location prediction accuracy of 70.2%. DD-RNN outperforms all the models both in location and split predictions with 95% and 79.5% accuracies, respectively.

5 Research Impact

This work can be foundational to other Sanskrit based NLP tasks. Let us consider translation as an example. In Sanskrit, arbitrary number of words can be joined together to form a compound word. Literary works, especially from the *Vedic* era often contain words which are a concatenation of three or more simpler words. Presence of such compound words will increase the vocabulary size exponentially and hinder the translation process. However, as a pre-processing step, if all the compound words are split before training a translation model, the number of unique words in the vocabulary reduces which will ease the learning process.

6 Conclusion

In this research, we propose a novel double decoder RNN architecture with attention for Sanskrit Sandhi splitting. A deep bi-directional encoder is used to encode the character sequence of a Sanskrit word. Using this encoded context vector, a *location decoder* is first used to learn the location(s) of the split(s). Then the *character decoder* is used to generate the split words. We evaluate the performance of the proposed approach on the benchmark dataset in comparison with other publicly available tools, standard RNN architectures and with prior work which tackle similar problems in other languages. As future work, we intend to tackle the harder *Samasa* problem which requires semantic information of a word in addition to the characters' context.

References

Akshar Bharati, Amba P Kulkarni, and V Sheeba. 2006. Building a wide coverage sanskrit morphological analyser: A practical approach. In *The First National Symposium on Modelling and Shallow Parsing of Indian Languages, IIT-Bombay*.

Xinchi Chen, Xipeng Qiu, Chenxi Zhu, and Xuanjing Huang. 2015a. Gated recursive neural network for chinese word segmentation. In *ACL (1)*, pages 1744–1753.

Xinchi Chen, Xipeng Qiu, Chenxi Zhu, Pengfei Liu, and Xuanjing Huang. 2015b. Long short-term memory neural networks for chinese word segmentation. In *EMNLP*, pages 1197–1206.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

Brendan S. Gillon. 2009. *Tagging Classical Sanskrit Compounds*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Pawan Goyal and Gérard Huet. 2013. Completeness analysis of a sanskrit reader. In *International Symposium on Sanskrit Computational Linguistics*, pages 130–171.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Gérard Huet. 2003. Towards computational processing of sanskrit. In *International Conference on Natural Language Processing (ICON)*. Citeseer.

Amba Kulkarni and Devanand Shukl. 2009. Sanskrit morphological analyser: Some issues. *Indian Linguistics*, 70(1-4):169–177.

Anil Kumar, Vipul Mittal, and Amba Kulkarni. 2010. *Sanskrit Compound Processor*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Monier Monier-Williams. 1970. *Sanskrit-English Dictionary*.

Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2013. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

Vikas Reddy, Amrith Krishna, Vishnu Dutt Sharma, Prateek Gupta, Vineeth M. R, and Pawan Goyal. 2018. Building a word segmenter for sanskrit overnight. *CoRR*, abs/1802.06185.

Kumar Sachin. 2007. Sandhi splitter and analyzer for sanskrit (with reference to ac sandhi). *M. Phil. degree at SCSS, JNU (submitted, 2007)*.

Hasim Sak, Andrew W Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342.

Rahul Garg Sumeet Agarwal Shubham Bhardwaj, Neelamadhav Gantayat. 2018. Sandhikosh: A benchmark corpus for evaluating sanskrit sandhi tools. In *11th International Conference on Language Resources and Evaluation*.

Supplementary material for Sanskrit Sandhi Splitting using *seq*2(*seq*)^2

Rahul Aralikatte IBM Research
Neelamadhav Gantayat IBM Research
Naveen Panwar IBM Research
{rahul.a.r, neelamadhav, naveen.panwar}@in.ibm.com

Anush Sankaran IBM Research
anussank@in.ibm.com

Senthil Mani IBM Research
sentmani@in.ibm.com

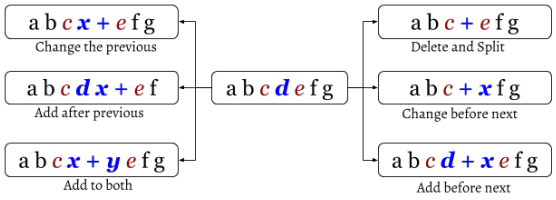

Figure 1: An example illustrating the different kinds of syntactical splits and the challenges for a sequence learning algorithm.

A Sandhi splitting challenges

Some of the major challenges faced by existing Sandhi splitting tools are briefly described below to motivate the hardness of the problem:

1. **Identifying multiple locations of split:** Identifying the location in a word where split has to be performed is the most challenging problem in performing splitting. As shown in Figure 1, transformation can happen in any location and in any form. Further, sandhi splitting involves identifying multiple potential locations, and validating them based on the previous locations.
2. **Cascading split effect:** There are some rules in which the effect of a split is not merely restricted to the immediate vicinity (neighboring characters). For example, in *uttarāyaṇa* → *uttara* + *ayana*, the *r* of *uttara* changes the *ṇ* of *ayana* to *n*.
3. **Samāsa:** The process of *Samāsa* is a process similar to Sandhi where words come together by discarding majority of their characters. A subset of the rules governing *Samāsa* overlaps with Sandhi. Thus, Sandhi splitters need to maintain two rule sets to correctly identify the constituent words. Existing systems require the user to explicitly pass the intermediate results back to it to perform the splitting correctly. For example, existing systems correctly split the word with a *Samāsa* *lakṣyasyārthatvavyavahārānurodhena* to form *lakṣyasya* + *arthatvavyavahāra* + *anurodhena*. However the second word, *arthatvavyavahāra*, contains a Sandhi and must be sent back into the system to get its constituent words.
4. **Incomplete rule set:** Though most of the splitting rules can easily be identified, there are many nuances which are often difficult to handle. There are also some rules which occur very rarely. For example, *sa yogī* → *saḥ* + *yogī*. Incomplete rule set during splitting will result in false negatives, such as, none of the existing splitters split (*a* + *chedyaḥ* → *acchedyaḥ*), correctly as *a* may not have the associated rule captured. Thus, heuristically defining all the splitting rules will be intractable, while learning them from examples is more generalizable.