# Byte-Level Token Classification for Sanskrit Word Segmentation Using ByT5

Orr Zwebner (ID. 203253422), Rotem Weissman (ID. 203958103)

Submitted as final project report for the NLP course, RUNI, 2024

## 1 Introduction

**Background**: The task of Sanskrit Word Segmentation (SWS) is a challenging problem in natural language processing (NLP), especially because of the unique structure of Sanskrit texts. Traditional texts in Sanskrit often lack spaces between words which makes word segmentation a crucial task for processing these texts effectively. Moreover, Sanskrit also exhibits a linguistic phenomenon called *sandhi*, where compound words are commonly formed by combining multiple subwords, with specific rules governing changes at word boundaries. This process, known as *sandhi*, modifies the ending of one word and the beginning of another to form a compound. These sandhi rules vary depending on the characters at the word boundaries.

**Examples:**

Figure 1: Ancient Sanskrit Manuscript (16th century)


Figure 2: Sandhi-compounded phrases (www.learnsanskrit.org)

**Motivation**: Our work focuses on developing a model for Sanskrit word segmentation using the ByT5 model, which leverages character-level tokenization to process texts. The primary motivation behind this work is to address the segmentation of unspaced Sanskrit texts, which is a common format in ancient manuscripts. Although our focus is on unspaced texts, we believe the model could be extended to handle sandhi with further adjustments and additional training.

### 1.1 Related Works

*One Model is All You Need*: ByT5-Sanskrit, a Unified Model for Sanskrit NLP Tasks, Link.
*CharSS: Character-Level Transformer Model for Sanskrit Word Segmentation* (Krishnakant Bhatt, Karthika N J, Ganesh Ramakrishnan, Preethi Jyothi), Link [1].
*Sanskrit Segmentation Revisited* (Sriram Krishnan and Amba Kulkarni) Link

## 2 Solution

### 2.1 General approach

The primary challenge of Sanskrit word segmentation (SWS) arises from the absence of spaces in traditional Sanskrit texts and the phenomenon of sandhi, where words are combined in complex ways. Our approach leverages the ByT5 model, a transformer-based architecture designed to handle tokenization-free tasks by working directly with raw bytes. This model is particularly well-suited for languages with complex character sets like Sanskrit.

**SLP1 Transliteration:** In this project, we utilize SLP1 transliteration for representing Sanskrit characters in a format that can be easily processed by modern NLP models. SLP1 is a widely accepted scheme for encoding Devanagari script, which is used to write Sanskrit, into a Latin script. This transliteration scheme preserves the phonetic characteristics of the original text while allowing for easier manipulation in computational tasks.
An example of a transliterated Sanskrit word :"devo'smi". Each of these characters (of the word "devo'smi") in SLP1 is treated as an independent token by the model. However, certain characters in SLP1, such as 'ā', 'ī', 'ū', 'ṛ', 'ḷ', 'ṅ', 'ñ', 'ṭ', 'ḍ', 'ṇ', 'ś', 'ṣ', and 'ḥ', are tokenized into 2 tokens each because they are represented by multiple bytes. This is crucial for maintaining the integrity of the phonetic information when processing Sanskrit texts.
This aspect is essential to consider when training the model, as it influences the sequence length and the granularity at which the model learns to segment the text.

**Binary Classification Task**: The task of SWS is framed as a binary classification problem at the Byte level. Each Byte in a given input sequence is evaluated to determine whether a space should follow it, which effectively segments the text. This approach is advantageous as it preserves the granularity of the input sequence and enables the model to capture intricate dependencies between characters.

**Why ByT5**? ByT5 is chosen due to its ability to handle input as raw bytes, bypassing the need for tokenization into subwords or words. This property is crucial for Sanskrit, where characters and diacritics play a significant role in word formation. The model's design makes it robust to variations in input length and capable of learning rich representations directly from characters.

### 2.2 Design

**Model and Tokenization:**
Labels are assigned to each token in the sequence. A label of 1 indicates that a space should follow the token, while 0 means no space should be added. SLP1 Example: Consider the Sanskrit phrase "*ahar vā andhaḥ*" which transliterates to "ahar vā andhaḥ" in SLP1.

In SLP1 transliteration, this phrase would be represented as:

"a", "h", "a", "r", " ", "v", "ā", " ", "a", "n", "d", "h", "a", "ḥ" In terms of bytes the characters: "a", "h", "r", "v", "n", "d", "h" and " " (space) are each represented by 1 byte. The characters "ā" and "ḥ" are each represented by 2 bytes due to their diacritical marks. Thus, the full sequence of bytes is:
Now, let's assign the labels for space segmentation:
"a" (1 byte) - Label: 0
"h" (1 byte) - Label: 0
"a" (1 byte) - Label: 0
"r" (1 byte) - Label: 1 (Space after "ahar")
" " (1 byte) - No label (as it's already a space)
"v" (1 byte) - Label: 0
"ā" (2 bytes) - Labels: [0,1] (Space after "vā")
" " (1 byte) - No label (as it's already a space)
"a" (1 byte) - Label: 0
"n" (1 byte) - Label: 0
"d" (1 byte) - Label: 0
"h" (1 byte) - Label: 0
"a" (1 byte) - Label: 0
"ḥ" (2 bytes) - Labels: [0,0] (End of the phrase)

The model's task is to take the unspaced input text: *'aharvāandhaḥ'*, predict the labels where spaces should be inserted, and output the correctly spaced sentence *'ahar vā andhaḥ'*.

**Model Architecture:**
**ByT5 Model:** The ByT5 model used in our project has approximately 582 million parameters (small version), making it a powerful model capable of capturing complex relationships between byte-level tokens.

**General Scheme of the Model:**
- **Input Byte Tokenization**: The input text is tokenized at the byte level. Each character in the input text is converted into one or more byte tokens, depending on whether the character is represented by a single byte or multiple bytes (as in the case described above).

- **Embedding Layer**: The byte tokens are passed through an embedding layer that converts each byte token into a dense vector representation. This layer maps the byte-level inputs into a high-dimensional space where similar tokens have similar representations.

- **Transformer Layers**: The embedded tokens are then passed through several transformer layers. Each layer consists of multi-head self-attention mechanisms and feedforward neural networks that process the input tokens in parallel, capturing both local and global dependencies within the sequence.

- **Token Classification Head**: The final hidden states from the transformer layers are passed through a classification head, specifically designed for token classification. This head outputs logits for each byte token, indicating the likelihood of a space following the token. **This is the stage where we fine-tune the model using our prepared dataset. The fine-tuning process involves training this classification head on our specific task of predicting space insertions in unspaced Sanskrit text, adjusting the model weights based on the loss calculated from our labeled data.**

- **Softmax and Prediction**: The logits are passed through a softmax layer to convert them into probabilities. The model then assigns a label (0 for no space, 1 for space) to each byte token based on these probabilities.

- **Output**: The predicted labels are used to insert spaces into the unsegmented input text, producing the final segmented output.

**Data Preparation:** During data preparation for training and evaluation, all tokens, except for spaces, are retained along with their corresponding labels. This ensures that the model is trained and evaluated on

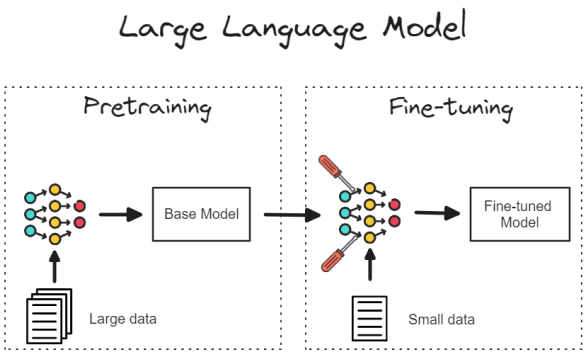


Figure 3: Fine-tuning process (source: Medium article)

sequences where the spaces have been removed, and it must predict the correct placement of these spaces based solely on the sequence of byte tokens.
The input sequences are padded to match the length of the longest sequence in each batch using Data Collator, ensuring uniformity without truncating any part of the data. This dynamic padding approach allows each batch to have a consistent input size while preserving all the original data. The labels are padded in the same way to ensure they match the length of the input sequences. A special padding token, -100, is used to mark the padding positions in the label sequence. This token ensures that these padding positions are ignored during the loss calculation.

**Training Process:** The training loop is managed using the Hugging Face Trainer class, which handles loading the data in batches, forward passes through the model, backpropagation, and optimization steps (minimizing cross-entropy loss). The Adam optimizer is used with a learning rate of 5e-5. This learning rate was chosen after conducting a search for the best parameters for fine-tuning, ensuring a balance between convergence speed and model stability. The model was trained for 5 epochs, which was determined to be sufficient for the Adam optimizer to guide the model towards convergence, resulting in satisfactory performance on the task.

### 2.3 Dataset: Digital Corpus of Sanskrit (DCS)

The dataset used in our experiments is the *Digital Corpus of Sanskrit (DCS)* [3], created by Oliver Hellwig (2010-2021). The DCS is a comprehensive, *Sandhi-split* corpus of Sanskrit texts. Each string in the corpus has been analyzed and verified by a single annotator.
The dataset's overall size includes 670,479 lines and 5,989,632 words, and exhibits a mean string length of 50.69 characters and a median length of 46 characters, indicating a balanced mix of short and long strings in the corpus.
The DCS dataset - Link

## 3 Experimental Results

**Examples of Model Predictions over the Test set:**

**Original:** *yasyaabhyāseprasādenaniṣpadyantedhanurdharāḥ*
**Ground Truth:** *yasya abhyāse prasādena niṣpadyante dhanurdharāḥ*
**Predicted:** *yasya abhyāse prasādena niṣpadyante dhanurdharāḥ*

**Original:** *eṣāvāasyaanavaruddhātanūḥ*
**Ground Truth:** *eṣā vā asya an avaruddhā tanūḥ*

**Predicted:** *eṣā vā asya anavaruddhā tanūḥ*

**Original:** *labhyaḥsatyamasaktenasukhināsaṃsṛtikṣayaḥ*
**Ground Truth:** *labhyaḥ satyam asaktena sukhinā saṃsṛti kṣayaḥ*
**Predicted:** *labhyaḥ satyam asaktena sukhinā saṃsṛti kṣayaḥ*

**Original:** *eṣātepṛthivīrājanbhuṅkṣvaenāmvigatajvaraḥ*
**Ground Truth:** *eṣā te pṛthivī rājan bhuṅkṣva enām vigata jvaraḥ*
**Predicted:** *eṣā te pṛthivī rājan bhuṅkṣva enām vigata jvaraḥ*

**Original:** *imaṃyajñaṃvitataṃviśvakarmaṇādevāyantusumanasyamānāḥ*
**Ground Truth:** *imaṃ yajñaṃ vitataṃ viśva karmaṇā devā yantu sumanasyamānāḥ*
**Predicted:** *imaṃ yajñaṃ vitataṃ viśvakarmaṇā devā yantu sumanasyamānāḥ*

**Original:** *tānsarvasmātevaantarāyan*
**Ground Truth:** *tān sarvasmāt eva antarāyan*
**Predicted:** *tān sarvasmāt eva antarāyan*

**Original:** *sarvetapasbalautkṛṣṭāḥrudrabhaktāḥsubhīṣaṇāḥ*
**Ground Truth:** *sarve tapas bala utkṛṣṭāḥ rudra bhaktāḥ su bhīṣaṇāḥ*
**Predicted:** *sarve tapas bala utkṛṣṭāḥ rudra bhaktāḥ su bhīṣaṇāḥ*

**Evaluation Metrics for Sanskrit Word Segmentation (SWS)**
The evaluation of Sanskrit Word Segmentation (SWS) typically involves three key metrics: Precision, Recall, and F1 Score. These metrics are calculated based on the accuracy of word segmentation within the text.
A **true positive** is when the model correctly predicts a segmentation point that exists in the ground truth. A **false positive** occurs when the model predicts a segmentation where there shouldn't be one. A **false negative** is when the model misses a true segmentation point, and a **true negative** is when the model correctly predicts no segmentation. For example, If the ground truth for "*devanamindraḥ*" is "*devanam indraḥ*" and the model predicts "*devanam indraḥ*", the correct segmentation after "*devanam*" is a true positive. If the model predicts "*deva namindraḥ*", the incorrect segmentation between "*deva*" and "*nam*" is a false positive, and missing the correct segmentation after "*nam*" is a false negative.

**Evaluation on the Test Set**
We first evaluated our model's performance on our own test dataset that consists of *unsandhied* texts with no spaces, which the model was not trained on:

| Model | Precision (%) | Recall (%) | F1 Score (%) |
|---|---|---|---|
| **Our Model** | **97.08** | **96.86** | **96.97** |
| "Vailla" byt5* | 15.13 | 18.07 | 16.47 |
| CANINE** | 14.83 | 89.67 | 25.46 |

Table 1: SWS Evaluation Results of Our Model on the Test Set

*The weights of the token classifying layer (last layer) of the model are generated without pre training with the same Tokenizer as we have used.
**The high recall and low precision clearly indicate that the CANINE model [2] tends to insert too many spaces, resulting in many false positives (incorrectly adding spaces) and a small number of false negatives (failing to indicate necessary spaces). For example, the model might incorrectly segment "*tā n s arv asm āt ev a anta rāya n*" (false positive) instead of the correct "*tān sarvasmāt eva antarāyan*".
These results demonstrate the effectiveness of our model when applied to the specific task it was trained for — segmenting *unsandhied* texts. The high Recall, Precision, and F1 Score indicate that the model performs exceptionally well on this dataset, accurately predicting segmentation points.

Due to the fact that all publicly available token classification models (e.g., those on Hugging Face) generate a classification layer that is not pre-trained, our prediction model based on token classification is ineffective without pre-training. Therefore, we did not include additional models for comparison. However, we will compare our model to those solving the SWS problem with sandhied texts.

**Experiments: Comparison with State-of-the-Art Models**
According to the paper [1], which conducted experiments on the SIGHUM dataset (Krishna et al., 2017), we present a comparison of their results with ours. It is important to note that the paper focused on solving the *sandhi* problem, while our model was trained exclusively on *unsandhied* texts.

The models presented in the paper, such as rcNN-SS, TransLIST, and charSS, are state-of-the-art for segmenting *sandhied* Sanskrit texts.
Below are the results from their paper and our model:

| **Model** | **Precision (%)** | **Recall (%)** | **F1 Score (%)** |
|---|---|---|---|
| rcNN-SS | 96.86 | 96.83 | 96.84 |
| TransLIST | 98.80 | 98.93 | 98.86 |
| charSS | 98.68 | 98.42 | 98.53 |
| **Our Model (Unsandhied)** | **66.53** | **59.77** | **62.97** |
| "Vailla" byt5 | 10.08 | 17.38 | 12.76 |
| CANINE | 10.03 | 85.02 | 17.94 |

Table 2: Comparison of Word Segmentation Results on Sandhied Texts (CharSS) and Unsandhied Texts (Our Model)

It is crucial to emphasize that our model was not trained for the *sandhi* task, in contrast to the dataset used in these comparisons, which focused on *sandhied* texts. Our model was trained exclusively on *unsandhied* compounds. The purpose of showing these results is to demonstrate the potential of our model for solving the *sandhi* problem, despite being trained on a different task.

# 4 Discussion

**Memory Management and Challenges** Throughout the training process, we encountered severe memory limitations on the GPUs which caused many of our initial experiments to fail. The following strategies were employed to address these challenges:

- Floating-Point Precision: We initially attempted to use mixed precision (fp16) to reduce memory usage, but this led to gradient instability, manifesting as NaN values during training. As a result, we reverted to training in full precision (fp32), which, while more memory-intensive, provided stability.

- Batch Size Reduction: To cope with the memory constraints, we experimented with various batch sizes, starting from 16 down to 1. Every time the batch size was increased, the model crashed due to GPU memory overflow. Ultimately, we settled on a batch size of 1, which allowed the model to complete training, albeit with a significant increase in training time (thus no padding was performed).

- Gradient Accumulation: Gradient accumulation was also attempted as a method to simulate larger batch sizes while staying within memory limits. Unfortunately, this approach did not sufficiently alleviate the memory issues given the model's architecture and the size of the data.

- Data Parallelism: We used DataParallel across three GPU devices to distribute the memory load. While this strategy provided some relief, it was not enough to completely prevent crashes during training, especially with larger datasets.

- Dataset Size Reduction: Due to persistent memory issues, we were forced to reduce the dataset size significantly. We randomly sampled 0.1 of the available data for training and testing. This reduction allowed us to train the model on a representative subset while managing memory constraints.

In conclusion, after extensive exploration and investigation into memory issues, including consulting external resources, we have determined that the primary cause of our memory limitations is not the size of the model itself but rather the size of the dataset. The large volume of data processed during training significantly strained the available GPU memory, leading to the challenges described above. It is also worth noting that the CANINE model by Google, which is another token-free approach, was tested but failed to produce satisfying results in the context of Sanskrit word segmentation, likely due to its inability to handle the specific challenges posed by the language's orthographic and syntactic complexities.

**Future Experiments and Research Considerations:**

**Distributed Data Parallelism (DDP):** Implementing Distributed Data Parallelism with additional devices to better handle large datasets and model training, potentially mitigating memory issues encountered during this project.

**Training Datasets:** Experimenting with more efficient sampling techniques to reduce memory load and improve training by increasing the batch size. In addition, training on larger dataset can improve the results as well.

**Additional Fine-tuning for Sandhi:** Conducting further fine-tuning or retraining the model with texts that include *sandhi* phrases. This could involve augmenting the dataset with *sandhi*-rich texts to enhance the model's ability to segment words in more complex linguistic contexts.

**Weighted Labels:** Implementing label weighting during training to address class imbalance which ensures that the model can learn effectively despite the disproportionately higher number of non-space (label 0) instances.

**Exploring Different Models:** Trying other models, such as m5 or another transformer model, which utilize subword tokens rather than bytes. These models could potentially offer a more balanced approach between the granularity of byte-level models and the efficiency of subword tokenization. Additionally, exploring the use of BERT, specifically models pre-trained for multilingual texts such as *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, could provide an alternative approach, leveraging subword tokenization to handle complex linguistic structures more effectively.

**Comparative Evaluation of SWS Models**: Running the state-of-the-art models like rcNN-SS, TransLIST, and charSS for prediction of Sanskrit Word Segmentation (SWS) on our *unsandhied* dataset is crucial and important for a direct comparison to our model's performance. However, this was beyond the scope of the current project, as it would require managing and importing their pre-trained models into our environment. This is an area for potential future work, as it would offer valuable insights into how well these models generalize to *unsandhied* texts.

**Expanding the Tokenizer:** Considering expanding the Tokenizer to include single tokens for SLP1 characters that are currently split into two tokens. This would involve modifying the tokenizer to recognize and treat characters like $\bar{a}$, $\bar{\imath}$, $\bar{u}$, $\underline{r}$, $\underline{l}$, $\acute{n}$, $\tilde{n}$, $\underline{t}$, $\underline{d}$, $\underline{n}$, $\acute{s}$, $\underline{s}$, and $\underline{h}$ as single tokens rather than two separate tokens. Such an adjustment could reduce the sequence length and potentially improve the model's performance by allowing it to focus on more meaningful units of information within the text.

# 5 Code

**Github**]
**Weights and Bias**
**Data**

# References


[1] Krishnakant Bhatt, Karthika N J, Ganesh Ramakrishnan, and Preethi Jyothi. Charss: Character-level transformer model for sanskrit word segmentation. In *arXiv*. Indian Institute of Technology Bombay, 2023.

[2] Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. Canine: Pre-training an efficient tokenization-free encoder for language representation. *arXiv*, 2021.

[3] Oliver Hellwig. *The Digital Corpus of Sanskrit (DCS)*, 2010–2022.

[4] Oliver Hellwig and Sebastian Nehrdich. Sanskrit word segmentation using character-level recurrent and convolutional neural networks. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2754–2763, Brussels, Belgium, 2018. Association for Computational Linguistics.

[5] Jivnesh Sandhan, Rathin Singha, Narein Rao, Suvendu Samanta, Laxmidhar Behera, and Pawan Goyal. Translist: A transformer-based linguistically informed sanskrit tokenizer. *arXiv*, 2022.

[6] Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *arXiv*, 2021.