# SansGPT: Advancing Generative Pre-Training in Sanskrit

**Rhugved Chaudhari[1], Bhakti Jadhav[2], Pushpak Bhattacharyya[3], Malhar Kulkarni[4]**

[1]College of Engineering Pune,
[2,3,4] Indian Institute of Technology Bombay
chaudharirp22.extc@coeptech.ac.in, bhakti.sj@iitb.ac.in, pb@cse.iitb.ac.in,
malhar@hss.iitb.ac.in

## Abstract

In the past decade, significant progress has been made in digitizing Sanskrit texts and advancing computational analysis of the language. However, efforts to advance NLP for complex semantic downstream tasks like Semantic Analogy Prediction, Named Entity Recognition, and others remain limited. This gap is mainly due to the absence of a robust, pre-trained Sanskrit model built on large-scale Sanskrit text data since this demands considerable computational resources and data preparation. In this paper, we introduce **SansGPT**, a generative pre-trained model that has been trained on a large corpus of Sanskrit texts and is designed to facilitate fine-tuning and development for downstream NLP tasks. We aim for this model to serve as a catalyst for advancing NLP research in Sanskrit. Additionally, we developed a custom tokenizer specifically optimized for Sanskrit text, enabling effective tokenization of compound words and making it better suited for generative tasks. Our data collection and cleaning process encompassed a wide array of available Sanskrit literature, ensuring comprehensive representation for training. We further demonstrate the model's efficacy by fine-tuning it on Semantic Analogy Prediction and Simile Element Extraction, achieving an impressive accuracy of approximately 95.8% and 92.8%, respectively.

## 1 Introduction

Sanskrit, one of the most ancient and highly structured natural languages, consists of a vast and diverse corpus of literature, with domains spanning from foundational texts on *Vyākaraṇa* (grammar), *Śikṣā* (phonetics), and *Nirukta* (etymology), to critical works on *Mīmāṃsā* (exegesis), *Nyāya* (logic), *Kāvyaśāstra* (poetics), *Sāhitya* (literature), *Nāṭyaśāstra* (dramaturgy), *Dharmaśāstra* (jurisprudence), etc. With origins tracing back to approximately 1000 BCE[1], Sanskrit remains in use today, not only as a language of traditional knowledge but also as a subject of modern linguistic and computational studies.

In recent years, significant efforts have been directed towards the digitization of Sanskrit texts and the advancement of natural language processing (NLP) techniques tailored to process this classical language. Notable progress has been made in tasks such as word segmentation (Hellwig & Nehrdich, 2018) (Krishna et al., 2018), dependency parsing (Sandhan, Krishna, et al., 2021), and word-order linearization (Krishna et al., 2019) (Krishna et al., 2021). These advancements have been pivotal in addressing some of the unique challenges posed by Sanskrit's intricate grammatical and syntactic structures.

Despite these strides, there remains a notable gap in developing NLP models capable of handling more complex semantic tasks such as Semantic Analogy Prediction, Named Entity Recognition, and other sophisticated downstream applications. The lack of a robust Sanskrit model trained on diverse, large-scale data is a significant barrier to advancing complex NLP tasks in the language. Models pre-trained on extensive datasets are critical for evaluating downstream tasks. To address this, we introduce **SansGPT**, a pre-trained Sanskrit language model aimed at advancing NLP capabilities, particularly for semantic tasks,

[1] By conservative estimate the time period of the oldest text in Sanskrit i.e., *Ṛgveda* is considered as 1000 B.C.

thereby simplifying the research and development of downstream tasks.

Our contributions are:

1) **Release of a Cleaned Pre-Training Sanskrit Corpus**[2]: We provide a large, meticulously cleaned, and pre-processed Sanskrit corpus consisting of approximately 51 million tokens. This corpus has been cleaned and prepared specifically for use in GPT-based pre-training.
2) **Development of a Custom Sanskrit Tokenizer**: We developed a lightweight and efficient tokenizer using Byte Pair Encoding tailored for Sanskrit. It breaks long compound words while preserving the syntactic structure (e.g., *Sandhi*) for coherent text generation, ensuring that the tokenization maintains word integrity and natural flow.
3) **Pre-training and Release**[2] **of SansGPT:** We pre-train and release SansGPT, a generative model designed for the Sanskrit language. This model sets the foundation for advancements in generative pre-training and fine-tuning within the Sanskrit NLP domain which would help to advance the development of complex semantic down-stream tasks.
4) **High Performance in Semantic Analogy Prediction, Simile Element Extraction, and release**[2] **of Simile (Upamā) Element Extraction Dataset:** Demonstrating the efficacy of SansGPT, we achieved a remarkable validation accuracy of approximately 95.8% on the Semantic Analogy Prediction task. We also evaluate and release the Simile (Upamā) Element Extraction dataset, achieving a validation accuracy of 92.8% on the task.

## 2 Methodology

### 2.1 Data Collection and Preprocessing

#### 2.1.1 *Data Sources*

Our corpus draws data from two major sources 1) Digital Corpus of Sanskrit[3] and 2) GRETIL[4]. The details of the corpus and the texts that it covers are mentioned in Appendix A.

#### 2.1.2 *Data Cleaning Methods*

We have applied several data cleaning techniques to prepare the Sanskrit corpus suitable for pre-training. The goal was to remove noise, standardize formatting, and ensure the integrity of the text for pre-training. Below is a summary of the data cleaning methods used:

1) *Removal of Special Characters and Unwanted Symbols*: The text was cleaned by removing special characters like period, asterisk, and others that do not contribute meaningfully to the content. Additionally, we replaced slashes '/' and pipes '|' with *daṇḍa* (।), the punctuation symbol in Sanskrit, to maintain consistency in the corpus.
2) *Handling Numerical References Between Daṇḍas*: Custom functions were employed to handle text between double *daṇḍas* and single *daṇḍa*. These functions identify the text enclosed by these symbols. If the enclosed text is shorter than a threshold number (likely numerical references or irrelevant content), it is replaced with a single *daṇḍa* or double *daṇḍa* accordingly.
3) *Removal of HTML Tags and Annotations*: HTML tags and inline annotations (e.g., %%, <BR>, or numbers in square brackets [0-9]) were removed using regular expressions to ensure that only the textual content remained.
4) *Whitespace and Line Break Normalization*: Excessive whitespaces, such as multiple spaces between words and unnecessary numerous line breaks, were removed and normalized. This ensured that the text adhered to a clean and consistent structure.
5) *Parentheses and Unwanted Punctuation*: Parentheses containing citations or supplementary information were removed to focus on the main text, while unnecessary punctuation such as colons, underscores, and equal signs were stripped for uniformity. Additionally, numeric and hyphenated patterns, like digits followed by letters (e.g., [0-9][a-z]), were cleaned to eliminate irrelevant numeric references and artifacts, ensuring a smoother text flow.
6) *Manual Inspection*: To ensure the highest quality of the corpus, each file was manually

[2] https://github.com/rhugvedd/SansGPT-Advancing-Generative-Pre-Training-in-Sanskrit

[3] http://www.sanskrit-linguistics.org/dcs/index.php

[4] https://gretil.sub.uni-goettingen.de/gretil.html

inspected. This step allowed us to identify and make exceptions where necessary, addressing any specific issues that the automated processes might have missed.

The final pre-training corpus consists of **51 million tokens** (with vocab size = 12000) for training and **1 million tokens** for validation. This comprehensive approach to data cleaning and validation ensured that the corpus was well-prepared for effective model training, enhancing the accuracy and performance of SansGPT.

## 2.2 Tokenization

Sanskrit, with its complex inflectional morphology and extensive compound word formation due to processes like *Sandhi* (euphonic combination), poses unique challenges for tokenization in NLP. Traditional methods struggle with Sanskrit's long, multi-unit words. For effective text generation and semantic analysis, it's essential to break these compound words into smaller, meaningful sub-words or tokens. Assigning a single vector to long words, such as '*śaśikeśarikaranakharavidāryamāṇatamaḥkarikumbhasambhavena*', can result in over-compression of semantic information. Also, for effective computational processing and semantic representation in transformer models, it's essential to break down compound words into meaningful sub-words or tokens. This prevents over-compression of information and ensures each token contributes to a nuanced understanding. To achieve this, we developed a custom tokenizer using Byte Pair Encoding (BPE) (Sennrich et al., 2016), which breaks complex Sanskrit words into smaller tokens for more efficient neural network processing and improved model performance.

### 2.2.1 *Tokenizer Training Details*

BPE is a statistical method for tokenizing sequences of characters or bytes. The process involves iteratively merging the most frequent pairs of tokens in a corpus. This iterative merging continues until a desired vocabulary size is achieved or no further pairs are found. The core idea is to replace the most frequent pairs of characters (or sub-words) with a single new token, thus reducing the overall number of tokens while capturing common patterns and structures. The details of tokenizer training is mentioned in Appendix B.

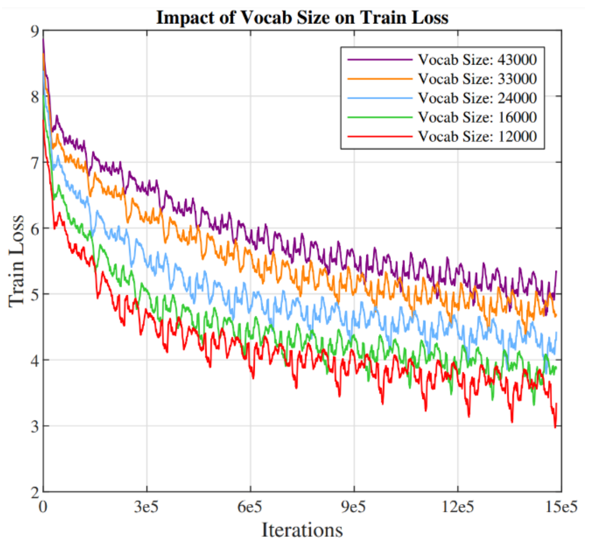

Figure 1: Impact of Vocab Size on Train Loss

### 2.2.2 *Deciding Vocabulary Size*

Before training the tokenizer, we split the entire text into meaningful segments. This process involves using 'space' and 'newline' characters as delimiters to split the text into segments and then feeding the segmented text for tokenizer training. Choosing an appropriate vocabulary size for tokenization is a balancing act. A large vocabulary size can lead to fewer tokens per word, potentially resulting in over-compression of semantic information into a single vector, which can be detrimental to the model's ability to learn nuanced meanings. Conversely, a small vocabulary size can split words into too many tokens, increasing the token count and potentially challenging the self-attention mechanism of transformers, which has a limited context length (Vaswani et al., 2023).

In our approach, we set the vocabulary size to 12,000 tokens. This decision was based on the monitoring and analysis of the frequency of the appearance of every new token being created in the entire corpus. As the vocabulary size increases the frequency of appearance of the new tokens in the corpus decreases, as rarer tokens are added to the corpus. Hence, it becomes harder for the model to learn to predict tokens occurring rarely in the corpus, as is evident in Figure 1. We also manually inspected the tokenized text of the entire corpus for different vocabulary sizes. We identified a balance that maximizes semantic preservation while keeping tokenization manageable. This carefully chosen vocabulary size helps ensure that the GPT model can efficiently process the Sanskrit text, maintaining the richness of the language while fitting within computational constraints.

## 2.3 Pre-Training

We developed and pre-trained our own GPT-75M model from scratch on the Sanskrit corpus which we have collected and cleaned. The pre-training process for our model utilized the generative autoregressive language modeling approach, which has been proven effective for improving language understanding (Radford et al., 2018). For details on autoregressive language modelling, refer to Appendix C.

### 2.3.1 *Pre-Training Setup*

We employed **sequential batching** for training to preserve the continuity and sequential nature of the Sanskrit text. This approach ensures that the order of tokens in the text is respected, which is essential for a model like GPT that relies heavily on context. Sequential batching maintains the coherence of the sequences during training, which aids in preserving the structural dependencies in the text and results in better predictions during autoregressive generation.

We opted for a batch size of 8 and a context size of 512 tokens to balance between computational efficiency and model generalization. In the training process, the choice of batch size directly influences the behaviour of gradient noise and model generalization. Smaller batch sizes, like the one used (batch size = 8), introduce higher noise in the gradient estimates during training (Jastrzębski et al., 2018). This increased noise can act as a form of regularization, preventing the model from overfitting to the training data and helping it escape local minima in the loss landscape. The frequent parameter updates associated with smaller batch sizes lead to more iterations per epoch, which can aid in achieving better generalization but at the cost of noisier and potentially less stable training (Keskar et al., 2017).

Conversely, larger batch sizes tend to produce smoother gradients, leading to more stable updates but also increasing the risk of poor generalization. By using a batch size of 8, we strike a balance between frequent updates and manageable noise levels, allowing the model to generalize well without requiring excessive computational resources. We conduct experiments with batch sizes of 8, 16, 32, 64, and 128 and plot the validation loss for 15 epochs over the entire dataset. We use a scaled higher learning rate for larger batch sizes, starting from 2e-4 (batch size =8) to 9e-4 (batch size =128).

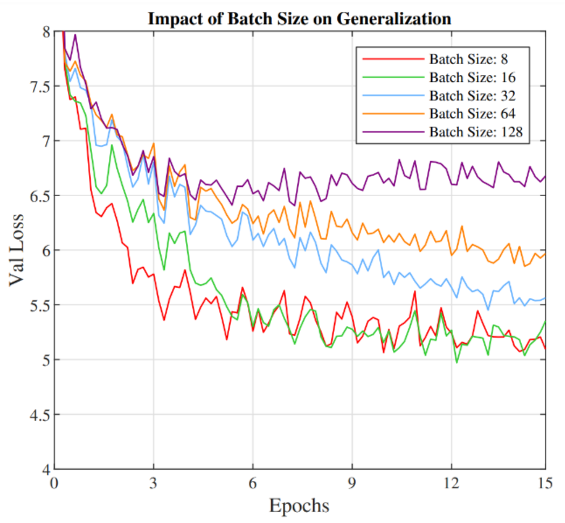

Figure 2: Impact of Batch Size on Generalization

Results obtained shown in Figure 2 demonstrate that larger batch sizes lead to poor generalization, with batch size = 128, even leading to overfitting.

### 2.3.2 *Optimization Setup*

We set a starting learning rate of 2e-4 and applied a cosine learning rate decay schedule to gradually decay the learning rate to 2e-5 by the end of the pre-training phase. A linear warm-up phase was employed for the initial 5000 iterations, during which the learning rate ramped up from 0 to 2e-4. This warm-up strategy helps prevent instabilities that can arise from using large learning rates early in training when the model is still adjusting its parameters.

The **relationship between batch size and learning rate** is crucial: smaller batch sizes necessitate smaller learning rates to maintain stable updates (Jastrzębski et al., 2018). With smaller batches, the gradient estimates are more variable, and smaller learning rates help prevent overshooting during optimization. We monitored the training loss in the initial stages of training and fine-tuned the learning rate accordingly to ensure a smooth training process. We train the model for a longer duration, specifically for a total of 300,000 iterations with a small batch size (=8) (~24 epochs over the entire dataset) (where one iteration corresponds to one batch), since longer training with smaller batch sizes helps generalization (Hoffer et al., 2018). We save model checkpoints every 50,000 iterations. The validation loss was evaluated every 1000 iterations for over 100 iterations on the validation set (again batch size = 8). Using a batch size of 8, we achieve a train loss of ~3 and a validation loss of ~5. The complete architecture details can be found in Appendix D.

## 2.4 Fine-Tuning

In the fine-tuning phase, we maintain the same architectural configuration that was used during the pre-training of our GPT model. This consistency ensures that the model's foundational capabilities, developed through extensive pre-training, are preserved while adapting the model to specific tasks or datasets. During fine-tuning, we utilize five special tokens to facilitate various aspects of sequence processing and task evaluation. These five tokens are the beginning of the sequence (<bos>), end of the sequence (<eos>), separator tokens (<sep1> and <sep2>), and the padding token (<pad>). We use decoder layer freezing (Howard & Ruder, 2018) of 6 layers and we pad the sequences to the context size of the GPT (=512). We also create an attention mask to ensure that the GPT only attends to meaningful tokens during fine-tuning and we exclude the pad tokens from the loss calculation. We discuss these aspects of fine-tuning in detail in Appendix E.

## 3 Evaluation

In this section, we evaluate SansGPT's performance on two tasks. We design the tasks, fine-tune the pre-trained model checkpoint, and report the results obtained.

### 3.1 Semantic Analogy Prediction

#### 3.1.1 *Testing Relational Understanding*

The Semantic Analogy Prediction task evaluates a model's ability to predict the “word d” in an analogy of the form “word a : word b :: word c : word d”, given the words only till word c. We chose the semantic analogy task because it directly assesses the model’s understanding of relational semantics. In languages like Sanskrit, where inflection and morphology play crucial roles, the ability to predict the correct relational word requires the model to have a strong grasp of syntactic and semantic dependencies. The analogy-based evaluation forces the model to extrapolate the relationship between word pairs from one analogy and apply it to another.

#### 3.1.2 *Handling Low-Resource Settings:*

The dataset used for this task consists of six categories: husband-wife, son-father, daughter-father, charioteer-warrior, defeated-victorious, and son-mother. The dataset contains a total of 6,415 examples, with 5,773 examples used for training and 642 examples reserved for validation. This dataset is derived from (Sandhan, Adideva, et al., 2021). Sanskrit is a low-resource language, and the dataset described (6,415 examples) reflects this limitation. Our task design tests the model to work within such constraints, where robust pre-training and fine-tuning are required to generalize well on tasks with limited data. Our approach towards the task formulation, execution and the results show the essential steps that researchers can take for handling fine-tuning for low-resource settings and languages.

#### 3.1.3 *Token Separation and Sequence Structuring for Enhanced Analogy Learning*

We formulate this analogy prediction task as follows: for each training example, we construct input sequences by tokenizing the analogy into the following structured format:

<bos> <tokens of word a> <sep1> <tokens of word b> <sep1> <sep1> <sep1> <tokens of word c> <sep1> <tokens of word d> <eos> <pad till end>.

The <bos> token indicates the start of the sequence, and the <eos> token marks the end. Each word is split using <sep1> token(s), with two additional <sep1> separators inserted between the two analogy pairs to ensure clear differentiation between them. The deliberate use of multiple <sep1> tokens between these pairs plays a crucial role in preserving the structural integrity of the analogy. By inserting these extra separators, we make sure that the model doesn’t conflate or blur the boundaries between the two analogy relationships—ensuring that “word A is to word B” remains distinct from “word C is to word D.” This separation is important for preventing *semantic bleed* between the analogy pairs, which can lead to confusion in the model’s understanding of how the words relate to each other. In transformer-based models, such delineation is vital for accurate learning, as the model relies on positional and contextual clues to process the relationships between tokens. The use of multiple <sep1> tokens helps the model maintain this distinction, improving its ability to correctly predict and generate analogies. Furthermore, padding tokens (<pad>) are applied to standardize sequence lengths to 512 tokens, ensuring uniformity in batch processing and making sure that shorter sequences do not affect the model's attention span or computational efficiency. The combination of multiple <sep1> tokens and proper padding optimizes both the semantic clarity and training efficiency of the model.

### 3.1.4 *Masking and Self-Attention Modification*

For each example, a mask is created, where all tokens until the <eos> token are assigned a value of 1, while all tokens after <eos> are assigned a value of 0. Padding tokens are introduced to standardize sequence lengths, but they do not carry any semantic value and are only placeholders. In a self-attention mechanism, if padding tokens were not masked, the model could waste computational resources by attending to these irrelevant tokens, leading to suboptimal learning. By masking, we prevent the model from incorporating padding into its predictions and gradient updates, ensuring that only valid tokens influence the attention distribution and the overall training process. This improves both the efficiency and accuracy of the model. We feed this input sequence into the GPT model, and the target sequence is the same as the input just left-shifted by one token. The model is trained with self-attention with the mask, as mentioned, and a loss function that ignores padding tokens, ensuring that the model learns to predict tokens while ignoring irrelevant padding.

### 3.1.5 *Top-k Sampling*

During the generation and evaluation of the validation set, we input tokens up to the <sep1> token following the word C (the second analogy). To predict word D, we employ **top-k sampling** (Fan et al., 2018) with a value of **k = 50**. Top-k sampling restricts the model's output to the top 50 most likely tokens based on the model's predicted probability distribution. From these top 50 tokens, we perform multinomial sampling, selecting a token based on its probability within this reduced set. This approach allows for more diversity in generation while ensuring that the output remains plausible, as it limits the choice to high-probability tokens while still allowing some flexibility in the prediction process.

### 3.1.6 *Results*

Our model achieves an impressive validation accuracy of 95%, significantly outperforming the accuracy of 32.7% reported by (Sandhan, Adideva, et al., 2021). We also calculate and report the Precision, Recall, and F1 scores, all of which come out to be ~95% (0.95). The graph of the evaluation metrics versus the training Epochs is shown in Figure 3. Following are some of the analogies generated by the fine-tuned model:

1) **Input:** Bhīṣma : Śantanu :: Jatāyu :
   **Output:** Aruṇa
   Bhīṣma and Śantanu have a son-father relationship. Similarly, Jatāyu and Aruṇa have a son-father relationship.
   **Category:** Son – Father
2) **Input:** Nakula : Draupadī :: Pururavā :
   **Output:** Urvaśī
   Nakula and Draupadī are husband-wife, just as Pururavā and Urvaśī.
   **Category:** Husband - Wife
3) **Input:** Kumbhakarṇa : Rāma :: Vṛṣaketu :
   **Output:** Vabhruvāhana
   In the battle, Rāma defeats Kumbhkarṇa, which mirrors Vṛṣaketu's relationship with Vabhruvāhana, who is victorious over him.
   **Category:** Defeated – Victorious
4) **Input:** Janamejaya: Īrāvati :: Bhīṣma :
   **Output:** Gaṅgā
   Janamejaya is the son of Īrāvati, which mirrors the relationship between Bhīṣma and Gaṅgā.
   **Category:** Son – Mother

## 3.2 Simile Element Extraction

### 3.2.1 *Data Description*

For this task, we have developed a new dataset specifically curated for Simile Element Extraction in Sanskrit texts. Our data was based on instances from *Vālmīkīya Rāmāyaṇa*. This annotated dataset contains sentences where a simile (*Upamā*) relation exists between two words. The goal is to extract the two words that form the simile along with the word that signals the similarity between them. The initial dataset consists of 400 manually annotated examples. To enhance the dataset, we employed data augmentation techniques, increasing the size to approximately 17,000 examples. The data was augmented with the help of the traditional Sanskrit lexicon 'Amarakośā.' The augmentation involved replacing the words involved in the simile with their synonyms or other suitable words, preserving the simile relation while generating new instances for training.

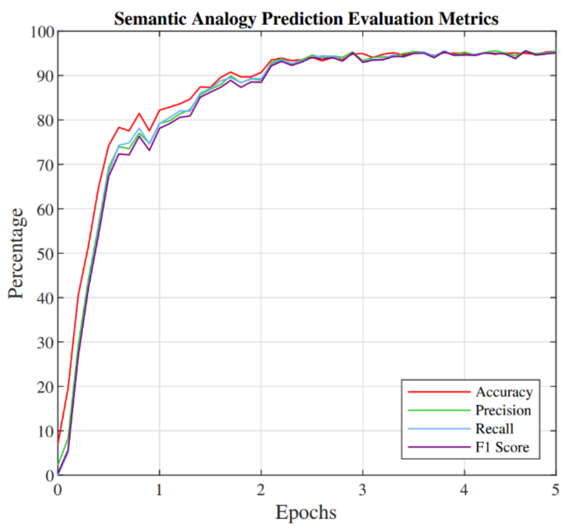

Figure 3: Semantic Analogy Prediction Eval Metrics

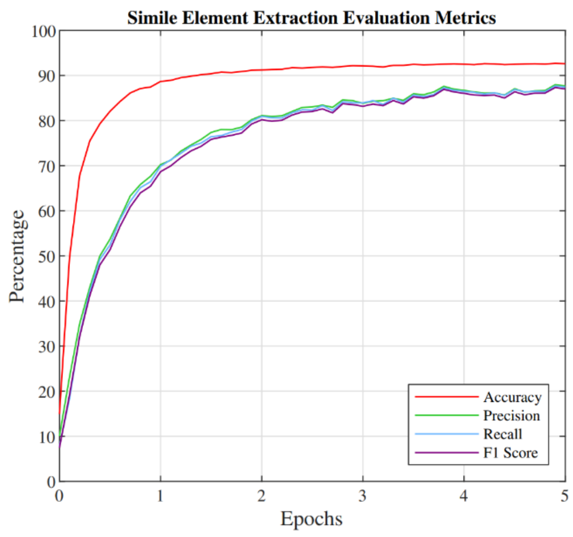

Figure 4: Simile Element Extraction Eval Metrics

### 3.2.2 *Task Design*

We design this task by structuring each example into a formal tokenized sequence, allowing the model to predict the simile elements efficiently. For each training example, we tokenize and format the input in the following manner:

1. **in_sen**: Tokenized input sentence.
2. **sp_word_sim**: Tokenized "similarity-indicating word prefixed by a space".
3. **word_sim**: Tokenized similarity-indicating word.
4. **sp_OoC**: Tokenized "Object of Comparison (Upameya) prefixed by a space"
5. **OoC**: Tokenized Object of Comparison (Upameya).
6. **sp_SoC**: Tokenized "Standard of Comparison (Upamāna) prefixed by a space".
7. **SoC**: Tokenized Standard of Comparison (Upamāna).

The structured input sequence for fine-tuning takes the following form:

<bos> <in_sen> <sep_1> <sep_1> <sep_1> <sp_word_sim> <sep_2> <word_sim> <sep_2> <sp_OoC> <sep_2> <OoC> <sep_2> <sp_SoC> <sep_2> <SoC> <eos>

### 3.2.3 *Dual Separator Tokenization*

The use of two distinct separator tokens, <sep_1> and <sep_2>, is critical in this task design to guide the model in effectively handling different parts of the sequence. The <sep_1> tokens delineate the input sentence from the output simile elements, ensuring that the model treats these as two distinct stages: first processing the input context and then focusing on extracting the simile components. This separation is crucial because it prevents the model from confusing the general sentence structure with the specific task of simile extraction, improving the clarity of the task during training and inference.

On the other hand, the <sep_2> tokens serve to distinguish between the different simile components: the word indicating similarity, the Object of Comparison (Upameya), and the Standard of Comparison (Upamāna). This differentiation is vital for the model to learn the hierarchical structure within the simile, ensuring that each element is correctly identified and mapped. The dual usage of <sep_1> and <sep_2> tokens ensures that both the contextual boundaries (input vs. output) and the internal structure (different simile components) are clearly differentiated, leading to more accurate predictions and better generalization across varied sentence structures. Padding and masking are applied as per our standard fine-tuning process described in task 3.1.

### 3.2.4 *Dual-Form Token Handling for Space-Sensitive Tokenization*

While formulating the Simile Element Extraction task, we include both forms of the output tokens—one with a leading space (e.g., <sp_word_sim>, <sp_OoC>, <sp_SoC>) and one without (e.g., <word_sim>, <OoC>, <SoC>). This design choice is essential because, in our tokenizer, tokens with and without leading spaces are treated as distinct entities. By training the model to predict both forms, we ensure that the model learns to handle variations in tokenization that arise due to spacing differences. This approach helps the model generalize better, as it becomes capable of predicting the correct output regardless of whether a token with a leading space or without one, appears in the input. In real-world text, spacing can vary based on context, and this formulation ensures that the model can handle such inconsistencies robustly. By incorporating both forms during training, the model becomes more adaptable and better equipped to extract simile components under different formatting or tokenization conditions, ensuring higher accuracy and resilience.

#### 3.2.5 *Simile-Aware Tokenization*

For the Simile Element Extraction task, we employ a specialized Simile-Aware Tokenization method designed to handle the unique challenges of compound words and sandhi in Sanskrit. In this method, the system segments the input sentence by identifying key similarity-indicating words (e.g., iva, ābham, and others). For e.g.:

*bṛhaspatisamo –> bṛhaspati + samo*

This ensures that these words, which play a crucial role in similes, are tokenized as distinct, complete units rather than a possibility of being broken down unevenly by traditional tokenization. By preserving the integrity of these words and generating separate tokens for them, the model is able to more effectively recognize and predict the elements of a simile, as it receives a clear and coherent representation of the similarity relationship. This approach significantly enhances the model's ability to process similes within the complex morphological structure of Sanskrit text.

#### 3.2.6 *Results*

The model demonstrates robust performance, achieving an accuracy of 92.8% on the Simile Element Extraction task. We also evaluate the Precision, Recall, and F1 Score, all of which come out to be ~87% (0.87). The graph of the evaluation metrics versus the training Epochs is shown in Figure 4. Following are some of the similes extracted by the fine-tuned model:

1) **Input:** *ayodhyānātha kṣamayā pṛthvīsamaḥ*

   Translation- The king of Ayodhyā is like Earth in terms of forgivingness.
   **Output:** Word Indicating Similarity: *samaḥ*
   Upameya: *ayodhyānātha* (lord of Ayodhya)
   Upamāna: *pṛthvī* (earth)

2) **Input:**
   *tato mahājīmūtamahīdhara ābhaṃ prabhinnamatyaṅkuśamatyasahyam ।*
   *rāmaupavāhyaṃ ruciraṃ dadarśa śatruñjayaṃ vāraṇaudagrakāyam ॥*

   Translation- He beheld a beautiful elephant named *Śatruñjayaṃ* with a huge body on which Rama was to mount. It looked like a vast mountain or a huge cloud. With ichor flowing from his temples and without caring for the goad, the elephant was intolerant.[5]
   **Output:** Word Indicating Similarity: *ābhaṃ*
   Upameya: *vāraṇa* (Elephant)
   Upamāna: *mahājīmūta* (Huge cloud)

3) **Input:**
   *ucitaṃ ca mahābāhuḥ na jahau harṣaṃ ātmanaḥ ।*
   *śāradaḥ samudīrṇāṃśuḥ kṣapākara teja iva ātmajam ॥*

   Translation- The mighty-armed i.e., Rāma did not leave his habitual cheerfulness like the autumnal Moon his own brightness.[6]
   **Output:** Word Indicating Similarity: *iva*
   Upameya: *mahābāhuḥ* (mighty-armed *Rāma*)
   Upamāna: *kṣapākara* (Moon)

## 4 Conclusion and Future Scope

SansGPT represents a significant step in the field of Sanskrit natural language processing. By providing a robust, pre-trained model, it addresses the critical gap in the development of advanced tools for analysing and understanding Sanskrit texts. The model's ability to effectively handle complex semantic tasks, as demonstrated by its evaluation performance, highlights its potential to facilitate a wide range of applications. SansGPT offers a solid starting point for various downstream tasks, including Named Entity Recognition, Text Summarization, and Sentiment Analysis.

[5] This verse is an augmented version of the verse 2.15.47 from Valmīkīya Rāmāyaṇā.
https://www.valmiki.iitk.ac.in/content?field_kanda_tid=2&language=dv&field_sarga_value=15&field_sloka_value=47&

[6] This verse is an augmented version of the verse 2.19.37 from Valmīkīya Rāmāyaṇā.
https://www.valmiki.iitk.ac.in/content?language=dv&field_kanda_tid=2&field_sarga_value=19&field_sloka_value=37

## References

Amarakośa of Amarasinha: https://sanskrit.uohyd.ac.in/scl/amarakosha/frame.html

Göttingen Register of Electronic Texts in Indian Languages (GRETIL). (n.d.). Sanskrit Texts. Retrieved from https://gretil.sub.uni-goettingen.de/gretil.html#Sanskrit

Sanskrit Linguistic Database (DCS). (n.d.). Digital Corpus of Sanskrit. Retrieved from http://www.sanskrit-linguistics.org/dcs/index.php

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization (No. arXiv:1607.06450). arXiv. http://arxiv.org/abs/1607.06450

Fan, A., Lewis, M., & Dauphin, Y. (2018). Hierarchical Neural Story Generation (No. arXiv:1805.04833). arXiv. http://arxiv.org/abs/1805.04833

Hellwig, O., & Nehrdich, S. (2018). Sanskrit Word Segmentation Using Character-level Recurrent and Convolutional Neural Networks. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2754–2763. https://doi.org/10.18653/v1/D18-1295

Hoffer, E., Hubara, I., & Soudry, D. (2018). Train longer, generalize better: Closing the generalization gap in large batch training of neural networks (No. arXiv:1705.08741). arXiv. http://arxiv.org/abs/1705.08741

Howard, J., & Ruder, S. (2018). Universal Language Model Fine-tuning for Text Classification (No. arXiv:1801.06146). arXiv. http://arxiv.org/abs/1801.06146

Jastrzębski, S., Kenton, Z., Arpit, D., Ballas, N., Fischer, A., Bengio, Y., & Storkey, A. (2018). Three Factors Influencing Minima in SGD (No. arXiv:1711.04623). arXiv. http://arxiv.org/abs/1711.04623

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima (No. arXiv:1609.04836). arXiv. http://arxiv.org/abs/1609.04836

Krishna, A., Santra, B., Bandaru, S. P., Sahu, G., Sharma, V. D., Satuluri, P., & Goyal, P. (2018). Free as in Free Word Order: An Energy Based Model for Word Segmentation and Morphological Tagging in Sanskrit. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2550–2561. https://doi.org/10.18653/v1/D18-1276

Krishna, A., Santra, B., Gupta, A., Satuluri, P., & Goyal, P. (2021). A Graph-Based Framework for Structured Prediction Tasks in Sanskrit. Computational Linguistics, 46(4), 785–845. https://doi.org/10.1162/coli_a_00390

Krishna, A., Sharma, V., Santra, B., Chakraborty, A., Satuluri, P., & Goyal, P. (2019). Poetry to Prose Conversion in Sanskrit as a Linearisation Task: A Case for Low-Resource Languages. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 1160–1166. https://doi.org/10.18653/v1/P19-1111

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training.

Sandhan, J., Adideva, O., Komal, D., Behera, L., & Goyal, P. (2021). Evaluating Neural Word Embeddings for Sanskrit (No. arXiv:2104.00270). arXiv. http://arxiv.org/abs/2104.00270

Sandhan, J., Krishna, A., Gupta, A., Behera, L., & Goyal, P. (2021). A Little Pretraining Goes a Long Way: A Case Study on Dependency Parsing Task for Low-resource Morphologically Rich Languages (No. arXiv:2102.06551). arXiv. http://arxiv.org/abs/2102.06551

Sennrich, R., Haddow, B., & Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units (No. arXiv:1508.07909). arXiv. http://arxiv.org/abs/1508.07909

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention Is All You Need (No. arXiv:1706.03762). arXiv. http://arxiv.org/abs/1706.03762

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T.-Y. (2020). On Layer Normalization in the Transformer Architecture (No. arXiv:2002.04745). arXiv. http://arxiv.org/abs/2002.04745

Kṛishnamohana Shastri (Ed.&Tr.) 2020. Kādambarī. Chaukhamba Surbharati Prakashan. Varanasi.

Valmiki Ramayanam "https://www.valmiki.iitk.ac.in-/content?language=dv&field_kanda_tid=1&field_sarga_value=1&field_sloka_value=1"

Śrimadvālmīkīya Rāmāyaṇā, Gita Press, Gorakhpur.

MANI, V. (1975). Puranic encyclopedia: a comprehensive dictionary with special reference to the epic and Puranic literature. Delhi, Motilal Banarsidass. https://www.sanskrit-lexicon.uni-koeln.de/scans/PEScan/2020/web/webtc1/index.php

G.H. Bhatt (Ed.) 1958 The Valmiki Ramayana Vol.1 .Maharaja Sayajirao University of Baroda. Baroda

## Appendices

### A. Corpus Details

The gathered extensive corpus encompasses a diverse range of literature, including epics such as the *Rāmāyaṇa* and *Mahābhārata*, the *Bhagavadgīta* with its various commentaries. It also features *Purāṇa* literature, religious texts spanning *Śaiva*, *Vaiṣṇava*, *Āgama*, *Tantra*, Buddhist traditions, and others. In addition, our corpus includes works on *Alaṅkāraśastra* (poetics), *Nāṭyaśastra* (dramaturgy), *Chandaśśastra* (prosody), as well as drama, narrative literature, and *Subhāṣitas*. We have also included texts related to various philosophical traditions, including *Mīmāṃsā*, *Vedānta*, *Sāṅkhya*, *Yoga*, *Nyāya*, *Vaiśeṣika*, *Śaiva*, Buddhist philosophies, *Dharmaśastra,* and others.

We have excluded Vedic texts, such as the *Veda[s]*, *Brahmaṇa[s]*, *Āraṇyaka[s],* and *Upaniṣad[s],* from our corpus. The Vedic language differs significantly from classical Sanskrit in its morphological features. For instance, words declined in classical Sanskrit as '*devāḥ'* are rendered as '*devāsaḥ'* in Vedic texts. Additionally, in the Vedic texts, prefixes or '*upasarga'*, can be dissociated from verbs in a sentence, unlike in classical Sanskrit language where they are connected with the verb. To avoid incorporating these distinctive linguistic features that could affect the training results, we chose not to include Vedic data in our corpus.

### B. Tokenizer Training

Let $D$ represent the corpus of Sanskrit text, initially tokenized into byte-level tokens (0-255):

$$T = \{t_1, t_2, \dots, t_n\}$$

Given $T$ we define a function to compute the frequency of adjacent token pairs:

$$\delta(\mathrm{t_i}, \mathrm{t_{i+1}}, \mathrm{a}, \mathrm{b}) = \begin{cases} 1 & if\, t_i = a \; and \; t_{i+1} = b \\ 0 & otherwise \end{cases}$$

$$f(a, b) = \sum_{i=1}^{n-1} \delta(t_i, t_{i+1}, \mathrm{a}, \mathrm{b})$$

To find the most frequent pair (a*, b*), we maximize the frequency over all consecutive token pairs (a, b) appearing in the corpus:

$$(a*, b*) = \underset{(a,b)}{\arg\max} f(a, b)$$

Once the most frequent pair (a*, b*) is identified, we merge it into a new token a*b*, and the token sequence $T$ is updated by replacing all occurrences of (a*, b*) with the new token a*b*. We repeat the process of counting pairs, identifying the most frequent one, and merging it until the desired vocabulary size $V$ is reached, or no more pairs can be merged.

### C. Auto-regressive Language Modelling

Auto-regressive language modelling is a fundamental approach used in the pre-training of generative models like GPT, where the task involves predicting the next token in a sequence given the previous tokens. The objective of autoregressive language modelling is to maximize the probability of a token sequence, where the model learns to predict each token by leveraging the preceding tokens as context. The model generates tokens sequentially, with each predicted token fed back into the model to form part of the context for the next prediction. This method models language in a sequential manner, allowing the model to generate coherent text by learning to predict each word based on prior context. Below, we delve into the mathematical formulation and process that defines this approach.

*Mathematical Formulation:*

The goal of the language modeling task is to maximize the likelihood of a sequence of tokens $w_1, w_2, \dots, w_N$, where the model learns to predict each token based on the context provided by the preceding tokens. This can be mathematically represented as:

$$P(w_1, w_2, \dots, w_N) = \prod_{i=1}^{N} P(w_i | w_1, w_2, \dots, w_{i-1})$$

Here, $P(w_i | w_1, w_2, \dots, w_{i-1})$ denotes the probability of token $w_i$ given the preceding sequence $w_1, w_2, \dots, w_{i-1}$. The training objective is to maximize this conditional probability, enabling the model to generate coherent sequences by predicting each token based on its preceding context. During pre-training, we create batches by selecting a sequence of tokens up to a predefined context size. The target sequence for the batch is generated by shifting these tokens to the left by one position. This setup allows the model to predict the next token based on all preceding tokens. Specifically, the model learns to predict each subsequent token in a sequence by leveraging the context of prior tokens.

At each time step $t$, the model computes a probability distribution over the entire vocabulary to determine the most likely next token $w_t$, given the previous tokens $w_1, w_2, \ldots, w_{t-1}$. This prediction process can be mathematically described as:

$$P(w_t | w_1, \ldots, w_{t-1}) = softmax\big(f(w_1, \ldots, w_{t-1})\big)$$

Where:

- $P(w_t | w_1, \ldots, w_{t-1})$ represents the conditional probability of predicting token $w_t$, given the previous tokens.
- $f(w_1, \ldots, w_{t-1})$ is the function learned by the model, which processes the input tokens through multiple transformer layers and computes logits for the softmax function.

The average loss (to be minimized) in language modeling is the negative log-likelihood of the predicted probability distribution over the target sequence. Formally, if the sequence of tokens is $w_1, w_2, \ldots, w_N$, the average loss $L$ is given by:

$$L = -\frac{1}{N} \sum_{i=1}^{N} log\ P(w_i | w_1, w_2, \ldots, w_{i-1})$$

## D. Complete Architecture Details

The final SansGPT model has 75M parameters and the architecture used the following configuration:

1. Vocabulary Size: 12,000
2. Model Dimension (d_model): 768
3. Number of Decoder Blocks: 12
4. Number of Attention Heads: 12
5. Feedforward Dimension: 2048
6. Model Dropout Probability: 0
7. Positional Encoding Dropout: 0
8. Weight Decay: 1e-2
9. Gradient Clipping Norm: 1.0
10. Optimizer Betas: (0.9, 0.95)
11. Mask Attention: True
12. Pre-Norm: True
13. Batch Overlap: None
14. Gradient Accumulation: None

In contrast to the original transformer model, which applies the Post-Layer Normalization (Post-LN) formulation, our model uses the Pre-Norm formulation (Xiong et al., 2020) for Layer Normalization (Ba et al., 2016) between sublayers in both the encoder and decoder. This deviation is crucial because Post-LN can result in larger gradients for parameters near the output layer, which, when combined with a large learning rate, leads to training instability and requires a longer warm-up phase. Pre-Norm, on the other hand, stabilizes gradients during initialization, enabling faster convergence by reducing the warm-up phase while speeding up training and minimizing the loss early on.

## E. Fine-Tuning Details

During fine-tuning, we utilize five special tokens to facilitate various aspects of sequence processing and task evaluation.

1. Beginning of Sequence Token (<bos>): Marks the start of a new sequence, enabling the model to understand where each sequence begins.
2. End of Sequence Token (<eos>): Signals the end of a sequence, helping the model to determine when to stop generating or processing tokens.
3. Separator Tokens (<sep1> and <sep2>): Used to separate distinct segments within the same input, such as different sentences or contextually relevant segments.
4. Padding Token (<pad>): Utilized to pad sequences to a uniform length, ensuring consistent input sizes across batches.

To enhance fine-tuning effectiveness, we implement decoder layer freezing. We freeze the weights of the initial 6 decoder layers, meaning that during training, these layers' gradients are not computed, and their weights remain unchanged. This method preserves the foundational knowledge and representations learned by these initial layers during pre-training. The later 6 decoder layers, which are not frozen, are updated during fine-tuning to tailor the model's features and parameters to the specific tasks at hand. Freezing the earlier layers has several benefits: it reduces the computational cost of training, minimizes the risk of overfitting, and ensures that the model's general knowledge is retained while adapting the model to new, task-specific patterns.

Sequences of varying lengths are padded to a standard length of 512 tokens using the <pad> token. This padding facilitates efficient batch processing by ensuring uniform input sizes. In the self-attention mechanism of SansGPT, padding tokens are masked using an attention mask. This mask ensures that the model does not attend to or generate output based on these padding tokens, which are non-informative. This helps the model focus solely on meaningful tokens within the sequence.

Additionally, during the loss calculation, contributions from padding tokens are excluded to prevent skewing the loss metric. This ensures that the model's training focuses on the meaningful portions of the data. The generation process is designed to stop as soon as an <eos> token is encountered, aligning the model's output with the expected sequence length and content and ensuring that the generated results are coherent and relevant to the given task.