

ABSTRACT

Digitization plays a crucial role in preserving historical documents by converting physical manuscripts into computer-readable formats. However, the successful digitization of ancient manuscripts faces numerous challenges such as brittleness, environmental damage, ink quality, and textual overlapping with background noise. This study focuses on enhancing digitized documents through the restoration of deteriorated and obscured textual contents using Language Model-based methods. By addressing degradation challenges like cracks, environmental influences, and de-acidification damages, the study aims to improve text readability, promote accessibility, and expedite information retrieval processes. The outcomes of this research not only contribute to the preservation of cultural heritage but also facilitate advancements in natural language processing operations, thus enriching scholarly endeavors and historical understanding.

TABLE OF CONTENTS

CHAPTER No.	TITLE OF THE CHAPTER	PAGE No.
	ABSTRACT	iv
	LIST OF TABLES	vii
	LIST OF FIGURES	viii
	LIST OF ABBREVIATIONS & SYMBOLS	ix
1	INTRODUCTION	1
	1.1 SANSKRIT MANUSCRIPT	1
	1.2 SANSKRIT MLM	2
2	LITERATURE SURVEY	3
3	SYSTEM ARCHITECTURE	11
	3.1 PROPOSED SYSTEM	11
	3.2 WORKFLOW	12
4	SYSTEM SPECIFICATION	13
	4.1 HARDWARE REQUIREMENTS	13
	4.2 SOFTWARE REQUIREMENTS	13
5	DESIGN AND IMPLEMENTATION	14
	5.1 INTRODUCTION	14
	5.2 PREPROCESSING FOR SANSKRIT MANUSCRIPT/ TEXT IMAGES	14
	5.3 OCR AND PREPROCESSING FOR EXTRACTED SANSKRIT TEXT	17
	5.4 SANSKRIT MLM FOR MISSING WORD/CHARACTER PREDICTION	18
	5.4.1 INTRODUCTION TO ROBERTA	18
	5.4.2 ROBERTA FINE-TUNING	19

	5.4.3 TOKENIZER FOR SANSKRIT MLM	20
	5.4.4 ROBERTA SET-UP FOR SANSKRIT MLM	23
	5.4.5 SANSKRIT MLM TRAINING	24
	5.4.6 PREDICTING MISSING SANSKRIT SEQUENCES USING SANSKRIT MLM	25
6	RESULTS AND DISCUSSION	27
7	FUTURE WORKS	30
8	CONCLUSION	31
9	REFERENCES	32
10	APPENDIX	34

LIST OF TABLES

TABLE No.	TITLE	PAGE No.
2.1	Literature Survey	9
2.2	Comparison with Existing Projects	10
4.1	Hardware Requirements	13
5.1	BPE Algorithm Working Process	21

LIST OF FIGURES

FIGURE No.	TITLE	PAGE No.
3.1	Architecture Diagram	11
3.2	Workflow Process	12
5.1	Adaptive Thresholding for Image Preprocessing	17
5.2	OCR on Preprocessed Image	17
5.3	Tokenizer Encoding and Attention Mask	22
5.4	Sanskrit MLM Predictions	26
6.1	Original Image	27
6.2	Destroyed Image	27
6.3	Training and Validation Loss	28
6.4	Perplexity	28
6.5	Extracted Text	28
6.6	Cleaned and Tokenized text	28
6.7	Final Output	29

LIST OF ABBREVIATIONS & SYMBOLS

ABBREVIATION	EXPANSION
MLM	Masked Language Model
LLM	Large Language Model
OCR	Optical Character Recognition
NLP	Natural Language Processing
WER	Word Error Rate
BERT	Bidirectional Encoder Representations from Transformers
RoBERTa	Robustly Optimized BERT-Pretraining Approach
NER	Named Entity Recognition
PPL	Perplexity
BPE	Byte Pair Encoding
RNN	Recurrent Neural Network
CNN	Convolutional Neural Network
GAN	Generative Adversarial Network
LSTM	Long Short-Term Memory
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
API	Application Programming Interface
GLUE	General Language Understanding Evaluation

CHAPTER 1

INTRODUCTION

1.1 SANSKRIT MANUSCRIPT

Sanskrit manuscripts represent a treasure trove of knowledge, spanning centuries of intellectual, spiritual, and cultural heritage. Comprising texts in various genres such as literature, philosophy, science, medicine, and spirituality, these manuscripts offer profound insights into the civilizations that nurtured them. From the Vedas, Upanishads, and epics like the Mahabharata and Ramayana to treatises on grammar, mathematics, and astronomy, Sanskrit manuscripts encapsulate the collective wisdom of ancient India. The beauty of Sanskrit manuscripts lies not only in their content but also in their material form. Written on materials ranging from palm leaves and birch bark to parchment and paper, these manuscripts are adorned with exquisite calligraphy and intricate illustrations, reflecting the aesthetic sensibilities of their creators. Each manuscript is a testament to the meticulous craftsmanship and intellectual rigor of the scribes who painstakingly transcribed and preserved these texts over generations. In the face of increasing environmental degradation, neglect, and the challenges of modernity, Sanskrit manuscripts are at risk of being lost forever. Recognizing the urgency of preserving this invaluable heritage, the Sanskrit Manuscript Revival initiative has emerged as a beacon of hope. Rooted in a deep reverence for Sanskrit literature and culture, this initiative seeks to revitalize the study and appreciation of Sanskrit manuscripts through comprehensive preservation, digitization, and dissemination efforts. At the core of the Sanskrit Manuscript Revival initiative is a commitment to accessibility and inclusivity. By harnessing digital technologies and collaborative networks, the initiative aims to make Sanskrit manuscripts accessible to scholars, researchers, and enthusiasts worldwide. Through digitization projects, online repositories, and educational programs, the initiative seeks to democratize access to Sanskrit heritage, fostering a deeper understanding and appreciation of this rich cultural legacy. Moreover, the Sanskrit Manuscript Revival initiative is not just about preservation; it is also about revitalization. By engaging with contemporary scholarship and interdisciplinary research, the initiative seeks to recontextualize Sanskrit manuscripts within the broader framework of global intellectual discourse. From comparative studies to interdisciplinary collaborations, the initiative endeavors to demonstrate the relevance and significance of Sanskrit manuscripts in the modern world.

1.2 SANSKRIT MLM

The Sanskrit MLM is used to predict missing Sanskrit words and characters in the cleaned and extracted Sanskrit text. The model is made by fine tuning the pretrained RoBERTa for MLM from Hugging Face on a classic Sanskrit dataset taken from Hugging Face. RoBERTa is a transformer model from Hugging Face that can be used for various NLP tasks like casual language model, MLM, question answering, sequence classification, and token classification. It builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates. The classic Sanskrit dataset that was used contains 3,42,033 short lines of Sanskrit sentences collected from various sources. This is used to fine tune the model on Sanskrit vocabulary so that model can learn classic Sanskrit language. Learning Sanskrit vocabulary is done by applying tokenizer on preprocessed Sanskrit dataset and after creating Sanskrit vocabulary for the model, we configure the RoBERTa for MLM according to our requirements and train it on our preprocessed dataset after splitting into training and validation sets. During training we mask tokens since it is going to be used as a MLM, and masking helps it to predict what the missing Sanskrit word or character is more accurately according to the context in the sentence. We can limit the vocabulary and length of predicted Sanskrit words in our configuration. After we fine tune our RoBERTa for MLM on the training dataset and ensure that both training loss and validation loss are low, we evaluate our model using perplexity as evaluation metric for the Sanskrit MLM created. A low perplexity indicates good performance for the Sanskrit MLM. After creation, we save the custom model and use a pipeline to use it on Sanskrit sentences where we replace missing Sanskrit word or character with <mask> keyword to indicate to the Sanskrit MLM that that's the location where we need to predict missing Sanskrit word/character, and it'll show top 4 predictions along with their score of how accurate that prediction is. Thus, the Sanskrit MLM is the core component in the process for restoration of erased and destroyed Sanskrit words in Sanskrit manuscripts and Sanskrit text images.

CHAPTER 2

LITERATURE SURVEY

[1] **Title:** Ancient Textual Restoration Using Deep Neural Networks

Authors :

Ali Abbas Ali Alkhazraji , Baheeja Khudair and Asia Mahdi Naser Alzubaidi

Methodology:

This study utilized the Codex Sinaiticus dataset, which underwent preprocessing involving encoding, removal of numbers, special characters, and new line symbols. Tokenization was then applied to segment each word into individual instances. Class targets were generated by replacing characters with special symbols. The study employed Generative Adversarial Networks (GANs), comprising a generator responsible for generating missing text and a discriminator evaluating the generated text. Through iterative collaboration, these networks facilitated sensitive reconstruction operations, preserving the format of ancient manuscripts, inscriptions, and documents. Three prediction models—LSTM, RNN, and GAN—were employed as proposed techniques for retrieving missing ancient texts. Validation accuracy results were 86%, 92%, and 98%, respectively.

Limitations:

This model only focuses on Ancient Greek Language's Manuscript for Text Restoration process with the help of LSTM, RNN and GAN.

[2] Title: Optical Character Recognition Of Sanskrit Manuscripts Using Convolution Neural Networks

Authors :

Bhavesh Kataria and Dr. Harikrishna B. Jethva

Methodology:

The work is being done on using Convolutional Neural Networks to recognise Sanskrit (Devanagari) characters (LSTM and BLSTM). An experiment was conducted on a large dataset to test the performance of new LSTM-BLSTM based Convolutional Neural Network techniques for sanskrit character recognition. The study lays the path for the creation of high-performance OCRs that can be used to the huge traditional Indian document collections that are currently available.

Limitations:

They want to improve the LSTM-BLSTM model in the future, with a goal of lowering WER even further, incorporating OCR correction systems , and reducing the dependency on real data for training.

[3] Title: EA-GAN: Ancient books text restoration model based on example attention

Authors :

Zheng Wenjun, Su Benpeng, Feng Ruiqi, Peng Xihua and Chen Shanxiong

Methodology:

This paper tells about a new approach, EA-GAN, combines generative adversarial networks and reference examples to accurately restore damaged characters, even in large areas. EA-GAN extracts features from both damaged and example characters, utilizing neighborhood information and example features during upsampling. An Example Attention block aligns example and character features, addressing alignment issues and small convolution receptive fields. Experiments on MSACCSO dataset and real scene pictures show significant improvements over current methods. the accuracy improvements of EA-GAN, including a 9.82% increase in PSNR, a 1.82% increase in SSIM, and a significant decrease in LPIPS values compared to current methods. In conclusion, the proposed EA-GAN model represents a significant advancement in the field of ancient book preservation and restoration, with potential for further enhancements in future research endeavors.

Limitations:

They only focused on the structure of the text. So they lack in the texture of the restoration.

[4] Title : Restoring and attributing ancient texts using deep neural networks

Authors :

Yannis Assael , Thea Sommerschild , Brendan Shillingford , Mahyar Bordbar , John Pavlopoulos , Marita Chatzipanagiotou , Ion Androutsopoulos , Jonathan Prag and Nando de Freitas

Methodology:

Ithaca is a new tool that helps experts study ancient writings on stone or metal. It's the first of its kind and makes the work of deciphering these inscriptions faster and more accurate. This tool can be really useful for historians studying newly found or unclear writings, making them more valuable as historical evidence. By using Ithaca, historians can understand ancient writing habits better, and it's available for anyone to use online. It's not just for historians - it can also be used for studying other ancient texts, like old documents or coins, in any language. Plus, Ithaca can be improved even more with input from users, making it a great tool for future research in machine learning. Overall, Ithaca is a game-changer for ancient history and humanities, providing advanced tools to explore the past.

Limitations:

While the model claims applicability to any language, including ancient and modern, its effectiveness may vary depending on the complexity and characteristics of different scripts and languages. Some languages or scripts may pose challenges that the model is not adequately equipped to handle.

[5] Title : Deep Learning Model to Revive Indian Manuscripts

Authors :

Puran Bhat , Kannagi Rajkhowa

Methodology:

They have conducted a comprehensive survey of character recognition efforts focused on Devanagari script, particularly handwritten characters. In this project , they have observed a range of techniques employed to enhance recognition accuracy. Notably, these techniques have demonstrated promising results in boosting accuracy levels. Additionally, there's potential for further advancements through the introduction of novel features.

Limitations:

There is a need to develop the standard database for Devanagari, Sharda Script, etc.

[6] Title: Virtual restoration and content analysis of ancient degraded manuscripts

Authors:

Anna Tonazzini, Pasquale Savino, Emanuele Salerno, Muhammad Hanif, Franca Debole

Methodology:

This paper have outlined the methods for digitally restoring ancient manuscripts afflicted by bleed-through distortion, a common issue in degraded documents. Their approach involves treating the manuscript image as a composite of distinct layers, which can be separated using spectral diversity observed in various acquisition modes such as multispectral (e.g., RGB) and recto-verso scans. By leveraging this diversity, They are effectively eliminate the interfering bleed-through patterns while preserving the valuable content of the manuscripts. The algorithms developed within this framework ensure that the restored manuscripts retain their original appearance, meeting two critical objectives: enhancing readability and interpretation for scholars and facilitating automated analysis tasks. Additionally, They have introduced a straightforward yet efficient algorithm for the initial alignment of multimodal acquisitions. These algorithms are characterized by their speed and suitability for routine use in libraries and archives.

Limitations:

They have to focus on exploring the potential of utilizing non-visible acquisition bands, such as Infrared and Ultraviolet, which are frequently accessible. These bands can serve as supplementary sources of information, potentially replacing the need for the verso side in our restoration processes.

S.No	Title	Year	Author	Methodology	Limitations	Journal
1	Ancient Textual Restoration Using Deep Neural Networks	2024	Ali Abbas Ali Alkhazraji , Baheeja Khudair and Asia Mahdi Naser Alzubaidi	LSTM, RNN, and GAN	Only focuses on Ancient Greek Language's Manuscript	ISCKU
2	Optical Character Recognition Of Sanskrit Manuscripts Using Convolution Neural Networks	2021	Bhaves Kataria and Dr. Harikrishna B. Jethva	LSTM-BSLTM based Convolutional Neural Network techniques	Lowering WER, incorporating OCR correction systems	Research Gate
3	EA-GAN: Ancient books text restoration model based on example attention	2022	Zheng Wenjun, Su Benpeng, Feng Ruiqi, Peng Xihua and Chen Shanxiong	EA-GAN	Texture of the restoration	Springer
4	Restoring and attributing ancient texts using deep neural networks	2022	Yannis Assael, Thea Sommerschild, Brendan Shillingford, Mahyar Bordbar, John Pavlopoulos and Marita Chatzipanagiotou	Ithaca	Model is not adequately equipped to handle all the languages	Springer
5	Deep Learning Model to Revive Indian Manuscripts	2023	Puran Bhat , Kannagi Rajkhowa	Zoning, Projection histogram, N-topple	Develop standard database for Devanagari , Sharda script	IJSR

Table 2.1 : Literature Survey

S.NO	TITLE	YEAR	AUTHOR	MODEL	ACCURACY	OUR MODEL ACCURACY
1	Ancient Textual Restoration Using Deep Neural Networks	2024	Ali Abbas Ali Alkhazraji , Baheeja Khudair and Asia Mahdi Naser Alzubaidi	LSTM RNN GAN	86% 92% 98%	Perplexity: 2.27
2	Optical Character Recognition Of Sanskrit Manuscripts Using Convolution Neural Networks	2021	Bhavesh Kataria and Dr. Harikrishna B. Jethva	LSTM-BSLTM based Convolutional Neural Network techniques	94.56%	
3	EA-GAN: Ancient books text restoration model based on example attention	2022	Zheng Wenjun, Su Benpeng, Feng Ruiqi, Peng Xihua and Chen Shanxiong	EA-GAN	90.13%	
4	Restoring and attributing ancient texts using deep neural networks	2022	Yannis Assael, Thea Sommerschildt, Brendan Shillingford, Mahyar Bordbar, John Pavlopoulos and Marita Chatzipanagiotou	Ithaca	71%	
5	Deep Learning Model to Revive Indian Manuscripts	2023	Puran Bhat , Kannagi Rajkhowa	Zoning, Projection histogram, N-topple	87%	

Table 2.2 : Comparison with Existing Projects

CHAPTER 3

SYSTEM ARCHITECTURE

3.1 PROPOSED SYSTEM

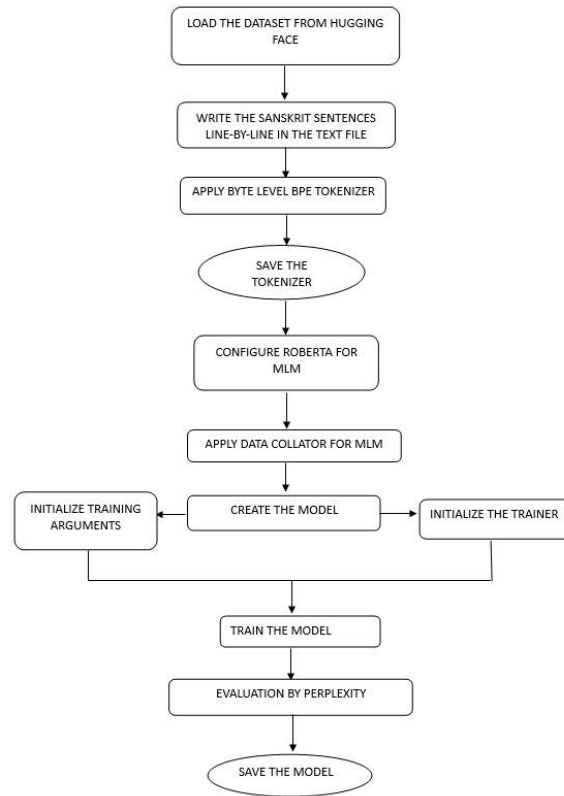


Fig 3.1 : Architecture Diagram

In this project, we have trained MLM model (Masked Language Model) .The system architecture of the model is shown in Fig 3.1. Here we have taken a dataset of Hugging Face which is a library that develops NLP models and tools, particularly known for its work in the field of transfer learning for NLP tasks. Hugging Face dataset is used to train our model. The text from the Sanskrit Manuscript has been extracted to the text file. Then Byte Level BPE Tokenizer has applied in the model. For Language modelling task , Transformers are used to collate batches. Then the MLM randomly masks some tokens in the input sequence, and the model is trained to predict the original tokens. We saved the model in hugging face hub finally.

3.2 WORKFLOW

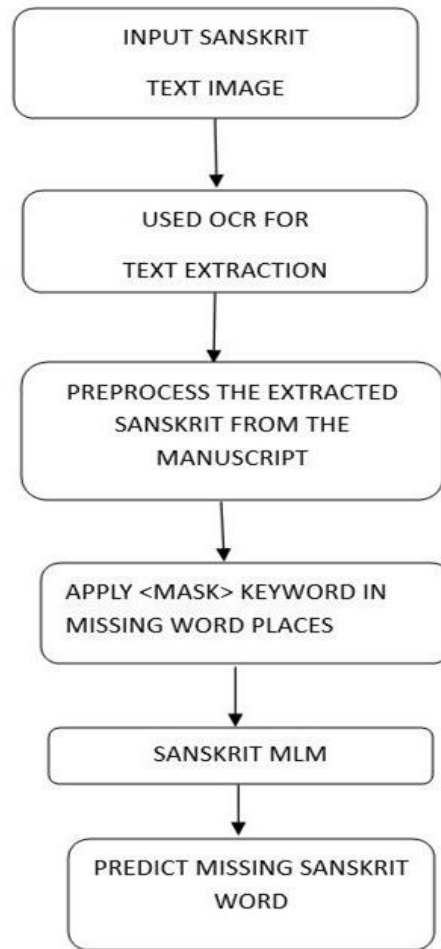


Fig 3.2 : Workflow

1. **Input Image:** Begin with an image of a Sanskrit manuscript.
2. **OCR Extraction:** Use OCR (Optical Character Recognition) to convert the image text into machine-readable format.
3. **Preprocessing:** Clean and organize the extracted Sanskrit text.
4. **Mask Keywords:** Insert placeholders for missing words or damaged portions.
5. **Sanskrit MLM:** Apply a specialized language model for Sanskrit to predict missing words.
6. **Prediction:** The model fills in the gaps, reconstructing the complete text.
7. **Revival:** This process revives ancient Sanskrit manuscripts.

CHAPTER 4

SYSTEM SPECIFICATIONS

This chapter includes the System Specification of our project.

4.1 HARDWARE REQUIREMENTS

S.No.	Parameter	Minimum Requirement
1	RAM	16GB
2	Processor	INTEL CORE I5
3	GPU/TPU	

Table 4.1 : Hardware Requirements

4.2 SOFTWARE REQUIREMENTS

Platform :

1. Operating System: Windows 8+
2. Programming Language : Python 3.0
3. Google colab

CHAPTER 5

DESIGN AND IMPLEMENTATION

5.1 INTRODUCTION

The implementation can be divided into 3 parts- preprocessing and OCR for input Sanskrit manuscript or text images, preprocessing and cleaning of text extracted through OCR, and finally feeding the clean extracted text to the Sanskrit MLM after proper formatting with <mask> keyword.

5.2 PREPROCESSING FOR SANSKRIT MANUSCRIPT/ TEXT IMAGES

The input images can be Sanskrit manuscript images or images having Sanskrit text from any source with varying levels of degradations. Depending on level and type of degradation in image, we use different methods from cv2 library of Python. Primarily, the Sanskrit manuscript images used were taken from various sources in Google search images and digital images of ancient Sanskrit book pages were taken from GitHub.

Here, cv2 refers to the OpenCV library, which stands for Open Source Computer Vision Library. OpenCV is a popular open-source computer vision and machine learning software library used for various image and video processing tasks, such as object detection, face recognition, image filtering, and more. The cv2 module provides a Python interface for OpenCV, allowing developers to leverage its powerful functionalities within Python scripts. With cv2, you can perform a wide range of computer vision tasks using Python.

The first step of image preprocessing is to convert the input image to grayscale using the cv2.cvtColor() function. Grayscale images have only one channel representing the intensity of each pixel, which simplifies further processing. If we directly use colour image without converting to grayscale image first, the upcoming preprocessing step that is adaptive thresholding may not work as expected or may produce suboptimal results. This is because colour images typically have three channels (Red, Green, Blue) representing the intensity of each colour channel at each pixel, whereas grayscale images have only one channel representing the overall intensity.

When adaptive thresholding is applied directly to a colour image, the algorithm will treat each colour channel independently, which may lead to inconsistent thresholding across the

channels and potentially undesirable results. Additionally, adaptive thresholding algorithms are typically designed to work with single-channel grayscale images, so applying them directly to a colour image might not be optimized. Converting the colour image to grayscale before applying adaptive thresholding ensures that the algorithm operates on a single channel representing the overall intensity of each pixel, leading to more consistent and predictable results. Grayscale images also require less computational resources compared to colour images, making processing more efficient.

Gaussian adaptive thresholding is the second step for preprocessing of input images, where this method is an image processing technique used for binarization, particularly in scenarios where the lighting conditions or background are uneven across the image. Binarization in adaptive thresholding is a technique used in image processing to separate regions or objects of interest, Sanskrit text in this case, from the background by converting an input grayscale image into a binary image. In a binary image, each pixel is represented as either black (0) or white (1), based on whether its intensity value exceeds a certain threshold. It calculates adaptive thresholds for each pixel based on the pixel values in its local neighbourhood, giving more weight to pixels closer to the centre of the neighbourhood.

The threshold value (x,y) at each pixel (x,y) is calculated as the weighted sum of the intensities of the pixels in its local neighborhood, with weights determined by a Gaussian window.

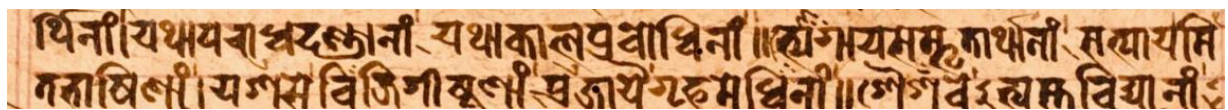
$$T(x, y) = \frac{\sum_{i=0}^{N-1} w(x_i, y_i) \cdot I(x_i, y_i)}{\sum_{i=0}^{N-1} w(x_i, y_i)}$$

The image is divided into small tiles or regions. The size of these tiles is determined by a parameter called the block size which is the parameter that determines the size of the neighborhood area around each pixel used to calculate the adaptive threshold. We set block size to 31, which means that adaptive thresholding algorithm considers a square neighborhood of size 31x31 pixels around each pixel when calculating the threshold value. Larger values of block size result in smoother and more globally consistent thresholding, but they may blur the edges of objects in the image. Whereas smaller values allow for finer detail but may be more sensitive to noise. For each tile, a threshold value is calculated based on the pixel intensities within that tile.

In Gaussian adaptive thresholding, these thresholds are calculated as a weighted sum of the intensities of the pixels in the tile. The weights are determined by a Gaussian window, which gives more weight to pixels closer to the center of the tile. Each pixel in the image is compared against its local threshold. If its intensity is greater than the threshold, it is set to a maximum value (1 or 255 for white color, typically 255 for binary images), indicating foreground, that is, Sanskrit text or characters. Otherwise, it is set to zero (black color), indicating background. The result is a binary image where pixels are either foreground (objects of interest) or background, depending on whether they passed the local threshold.

We also use another parameter 'C' set to value 11, which is a constant value subtracted from the calculated mean or weighted mean of the neighborhood pixels to obtain the final threshold value for each pixel. It's commonly used to adjust the thresholding behavior and fine tune threshold value $T(x,y)$, compensating for variations in local illumination and contrast. A higher value of C makes the threshold more permissive, resulting in more pixels being classified as foreground (white) pixels. Conversely, a lower value of C makes the threshold more strict, resulting in fewer pixels being classified as foreground pixels.

Gaussian adaptive thresholding is preferred here since it's more robust to variations in illumination and background compared to simple thresholding methods, thus making it suitable for Sanskrit manuscript images with non-uniform illumination, varying contrast, or complex backgrounds, where traditional global thresholding methods may not produce satisfactory results. Finally, we invert color of resulting thresholded image having black background and white Sanskrit text (foreground) using bitwise NOT to get a binary image of white background and black Sanskrit text (foreground). It improves segmentation accuracy and feature extraction in diverse types of images, making it easier for OCR to recognize Sanskrit characters in text during extraction and thus enabling us to extract Sanskrit text more accurately with less loss of words and characters during extraction.



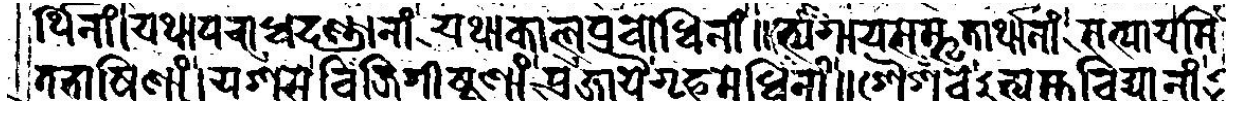


Fig 5.1 : Sanskrit manuscript image before (top) and after (bottom) applying Gaussian Adaptive Thresholding

5.3 OCR AND PREPROCESSING FOR EXTRACTED SANSKRIT TEXT

After preprocessing the Sanskrit images, we use pytesseract for performing OCR. Python-tesseract is an optical character recognition (OCR) tool for python which can do OCR for thousands of languages including Sanskrit. It will recognize and “read” the text embedded in images. Python-tesseract is a wrapper for Google’s Tesseract-OCR Engine. It is also useful as a stand-alone invocation script to tesseract, as it can read all image types supported by the Pillow and Leptonica imaging libraries, including jpeg, png, gif, bmp, tiff, and others. Finally, we clean the extracted Sanskrit text by removing unwanted characters like newline, extra whitespaces, and unwanted punctuations using re Python library. This module provides regular expression matching operations and allows us to work with patterns and text matching. With regular expressions, we can search, replace, split, and manipulate strings based on patterns.

उक्तं योगवासिष्ठे ॥
अक्षरावगमलब्धये स्थूलवर्तुलट्षत्परिमहः ।
शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिलामयार्चनम्-इति ॥

उक्तं योगवासिष्ठे ।
९९ ष 4
अक्षरावगमलब्धये स्वरूखन्तुखटट्षत्परिमहः ।
शुद्धबुद्धपरिन्धये तथा दारुमन्मयशिखामय चैनम्-इति ॥

उक्तं योगवासिष्ठे ९९ ष 4 अक्षरावगमलब्धये स्वरूखन्तुखटट्षत्परिमहः । शुद्धबुद्धपरिन्धये तथा दारुमन्मयशिखामय चैनम्-इति ॥

Fig 5.2 : Preprocessed Sanskrit digital image (top), Sanskrit text extracted from preprocessed image by OCR (middle), and cleaned extracted text (bottom)

5.4 SANSKRIT MLM FOR MISSING WORD/CHARACTER PREDICTION

5.4.1 INTRODUCTION TO ROBERTA

This is a MLM used for predicting missing Sanskrit words and characters in cleaned and preprocessed Sanskrit sentences extracted from Sanskrit manuscript images or Sanskrit text images. The Sanskrit MLM is created using Hugging Face's RoBERTa which is a Transformer-based model. A Transformer is a deep learning model architecture known for its effectiveness in handling sequential data like natural language. The key innovation of the Transformer architecture is the self-attention mechanism, which allows the model to weigh the importance of different input tokens when making predictions. This mechanism enables the Transformer to capture long-range dependencies in sequences more effectively compared to traditional RNNs or CNNs. The Transformer architecture consists of an encoder-decoder structure, where the encoder processes the input sequence, and the decoder generates the output sequence. Hugging Face is a company and an open-source community that focuses on NLP technologies. They provide a wide range of tools and resources for developers and researchers working with NLP, including pre-trained models, libraries, datasets, and training pipelines.

The RoBERTa model is known for its strong performance across various NLP tasks, including text classification, question answering, NER, and language generation. RoBERTa is pre-trained on a large corpus of text data using a self-supervised learning approach, where a corpus refers to a large collection of text or speech data that is systematically compiled and used for linguistic analysis, research, and development of language models and algorithms. Self-supervised learning is a machine learning process where the model trains itself to learn one part of the input from another part of the input without need for labels, thus learning meaningful representations of the input data. It is also known as predictive or pretext learning. We've chosen RoBERTa to build the Sanskrit MLM because it is an encoder model, using only the encoder of a Transformer model. At each stage, the attention layers can access all the words in the initial sentence. These models are often characterized as having "bi-directional" attention, and are often called auto-encoding models. The pretraining of these models usually revolves around somehow corrupting a given sentence (like by masking random words in it) and tasking the model with finding or reconstructing the initial sentence, thus being effective for our masked word prediction task.

It is a variant of the BERT model, which is a bidirectional transformer pretrained using a combination of masked language modeling objective and next sentence prediction on a large corpus comprising the Toronto Book Corpus and Wikipedia. BERT on which RoBERTa was built on corrupts the inputs by using random masking during pretraining. The difference is that RoBERTa has the same architecture as BERT, but uses a byte-level BPE as a tokenizer (same as GPT-2) and uses a different pretraining scheme, tokens are masked differently at each epoch together to reach 512 tokens (so the sentences are in an order than may span several documents) whereas BERT does it once and for all, trains with larger batches (for more efficiency), and uses BPE with bytes as a subunit and not characters (because of unicode characters).

5.4.2 ROBERTA FINE-TUNING

To make a MLM that fits our task of predicting missing Sanskrit words, we have to fine tune the already pretrained RoBERTa model. Fine-tuning is the process of taking a pre-trained language model, which has been trained on a large corpus of text data, and further training it on a smaller, domain-specific dataset for a specific task or application to adapt the pre-trained model to perform well on the target task by updating its parameters based on the task-specific data. We fine-tune instead of training model from scratch because training a model, especially a large one, requires a large amount of data. This becomes very costly in terms of time and computer resources and has heavy environmental impacts due to extremely high carbon footprint. Thus, sharing the trained weights and building on top of already trained weights (in pretrained model) reduces the overall compute cost and carbon footprint of the community. Since the pretrained model was already trained on lots of data, the fine-tuning requires way less data, time, and resources to get decent results since the knowledge which the pretrained model has acquired is “transferred” to it (transfer learning).

This is the reason we opted for fine-tuning a pretrained model instead of training a model like CNN, RNN, and CycleGAN from scratch which would require an enormous amount of data and would train for days even on a GPU. And setting up just a GPU or TPU for such tasks on our system is an expensive, lengthy, and complex process, thus we opt for Google Colab (for

GPU/TPU usage) and Hugging Face’s pretrained models. Fine-tuning RoBERTa is done by taking the Sanskrit classic dataset from Hugging Face as the domain-specific dataset, containing 3,42,033 short lines of Sanskrit sentences collected from various sources and our specific task being predicting missing Sanskrit words and characters. We copy the Sanskrit sentences in the dataset as a continuous paragraph into a text file without writing line-by-line for easy processing.

5.4.3 TOKENIZER FOR SANSKRIT MLM

Our next task is to train the tokenizer for our Sanskrit MLM. Tokenizers are one of the core components of the NLP pipeline. Their purpose is to translate text into data that can be processed by the model. Models can only process numbers, so tokenizers need to convert our Sanskrit text inputs to numerical data. The goal is to find the most meaningful representation of the text that makes the most sense to the model and if possible, the smallest representation. Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or subwords.

The Byte Level BPE tokenizer is used because RoBERTa was trained using that tokenizer, and we have to use the same while fine-tuning RoBERTa. Byte Level BPE tokenizer training starts by computing the unique set of words used in the Sanskrit corpus in our Sanskrit text file after the normalization and pre-tokenization steps are completed, then building the vocabulary by taking all the symbols used to write those words. After getting this base vocabulary, we add new tokens until our desired vocabulary size of 52,000 tokens is reached by learning merges, which are rules to merge two elements of the existing vocabulary together into a new one. At the beginning, these merges will create tokens with two characters, and then, as training progresses, longer subwords. At any step during the tokenizer training, the BPE algorithm will search for the most frequent pair of existing tokens (by “pair” means two consecutive tokens in a word). That most frequent pair is the one that will be merged, and we repeat this for the next step. We’ve set minimum frequency as 2, which means tokens must occur at least twice in the training data to be included in the vocabulary.

We also mention special tokens used by RoBERTa in tokenizer training, and these special tokens are <s>, <pad>, </s>, <unk>, and <mask>. The <s> token marks the beginning of a sequence (sentences, paragraphs, or any other units of text) in input data. To efficiently train neural networks, input sequences are typically grouped together into batches.

Model	BPE
Training	Starts from a small vocabulary and learns rules to merge tokens
Training step	Merges the tokens corresponding to the most common pair
Learns	Merge rules and a vocabulary
Encoding	Splits a word into characters and applies the merges learned during training

Table 5.1 : BPE algorithm working for each process in the model

Hugging Face Transformers models expect multiple sentences by default, and batching is the act of sending multiple sentences through the model all at once. If we only have one sentence, we can just build a batch with a single sequence by creating a batch of two identical sequences. However, sequences within a batch may have different lengths. This is a problem because when we convert it into numeric format of tensors, they need to be of rectangular shape, so we won't be able to convert the list of input sentences into a tensor directly. Thus, we use padding to make our tensors have a rectangular shape. To ensure that all sequences in a batch have the same length, padding token <pad> is used such that shorter sequences are padded with a special padding token. This token <pad> is used to fill the empty spaces in shorter sequences. Padding makes sure all our sentences have the same length by adding a special word called the padding token to the sentences with fewer values. Attention masks are tensors with the exact same shape as the input tensor, filled with 0s and 1s: 1s indicate the corresponding tokens should be attended to, and 0s indicate the corresponding <pad> tokens should not be attended to and should be ignored by the attention layers of the model. The conversion of tokenized sequences into tensors occurs automatically when we pass the tokenized inputs to the RoBERTa model for inference or training. The </s> token marks the end of a sequence in input data.

<unk> token represents unknown words or tokens that are not present in the vocabulary of the model. During tokenization, if a word is encountered that is not in the vocabulary, it is replaced with the <unk> token. This allows the model to handle out-of-vocabulary words during training and inference. If an example we are tokenizing uses a character that is not in the training corpus taken from our Sanskrit text file, that character will be converted to the unknown token <unk>. That's one reason why lots of NLP models are very bad at analyzing content with emojis, for instance. RoBERTa's byte-level BPE tokenizer has a clever way to deal with this: they don't look at words as being written with Unicode characters, but with bytes. This way the base vocabulary has a small size (256), but every character we can think of will still be included and not end up being converted to the unknown token. Byte-level BPE tokenization involves treating text not as sequences of Unicode characters but as sequences of bytes. In byte-level BPE, each character is represented by a single byte (8 bits), and the tokenizer builds its vocabulary based on these bytes. Since there are 256 possible values for a byte (ranging from 0 to 255), the base vocabulary (vocabulary before the merging process in BPE algorithm to create new tokens) size is 256.

```
tokenizer.encode("रामो वनं गच्छति।").tokens
```

```
['<s>',
 'à¤',
 'à¤%',
 'à¤®',
 'à¤',
 'Gà¤¤¤¤¤¤¤¤¤¤',
 'à¤¤¤',
 'Gà¤¤¤¤¤¤¤¤¤¤',
 'à¤¤¤¤',
 'à¤¤¤¤¤¤¤¤¤¤',
 'à¤¤¤¤¤¤¤¤¤¤',
 '</s>']
```

```
tokenizer.encode("रामो वनं गच्छति।").attention_mask
```

$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

Fig 5.3 : Tokens of encoded Sanskrit text and it's attention mask

In MLMs, the <mask> token is used to represent masked tokens in the input sequence. During pre-training, a certain percentage of tokens in the input sequence are randomly masked, and the model is trained to predict these masked tokens based on the surrounding

context. The <mask> token indicates the position of masked tokens in the input sequence. We then save our tokenizer to a separate directory and we'll use this to fine tune RoBERTa MLM in Sanskrit vocabulary.

5.4.4 ROBERTA SET-UP FOR SANSKRIT MLM

We use the RobertaConfig class to instantiate a RoBERTa model according to the specified arguments, defining the model architecture. The custom specified arguments passed are vocabulary size of 52,000 tokens, maximum length of input sequences that the model can accept/process as 514 tokens (Sequences longer than this length will be truncated, and sequences shorter than this length will be padded), 12 attention heads per layer in model (Attention heads are a crucial component of the Transformer architecture. They allow the model to focus on different parts of the input sequence simultaneously, enabling it to capture complex patterns and relationships within the data), 6 hidden layers (increasing this will result in increased computational cost and slower training, and possibly overfitting if training data is small), and vocabulary size of the token_type_ids passed as 1. The remaining parameters retain the default values of RoBERTa similar to Facebook AI's RoBERTa-base architecture.

We then use RobertaTokenizerFast class which is a "fast" RoBERTa tokenizer (backed by HuggingFace's tokenizers library), faster than the regular RobertaTokenizer class. This class provides methods for tokenizing and encoding input text data for RoBERTa models, and we use the tokenizer we trained and saved earlier for Sanskrit vocabulary in the previous step. We finally create our Sanskrit MLM by passing our custom configuration mentioned before as argument into RobertaForMaskedLM class, having a total of 83,504,416 trainable parameters from embedding, Transformer, and output layers. Before passing our Sanskrit data stored in text file as training data, we need to preprocess it. We import the LineByLineTextDataset class from the Transformers library. This class is used to create datasets for training language models based on line-by-line text files. The maximum block size (sequence length) for each training example is set to 128 tokens while we use our custom tokenizer created earlier to tokenize the data in the Sanskrit text file. Sequences longer than this block size will be split into multiple blocks since the model needs input in batches for

processing and handle long sequences that exceed the model's maximum input length. This is memory efficient as well as can help ensure that each block contains sufficient context for the model to learn from. This can potentially improve the model's performance, especially for tasks where long-range dependencies are important. Finally, we split the preprocessed data into training dataset and validation dataset through 80:20 split.

The `DataCollatorForLanguageModeling` class is used to instantiate a data collator object used for language modeling. Inputs are dynamically padded to the maximum length of a batch if they are not all of the same length, and it uses the custom tokenizer we created. Data collators are objects that will form a batch by using a list of dataset elements as input. We specify the data collator to use masked language modelling (the labels are -100 for non-masked tokens, and the value to predict in case of masked token), and set the probability with which to randomly mask tokens in the input as 0.15. This means that each token in the input text has a 15% chance of being randomly masked during training iterations. Masking helps the model to predict what the missing Sanskrit word or character is more accurately according to the context in the sentence.

5.4.5 SANSKRIT MLM TRAINING

After RoBERTa set-up and model creation, we're finally ready to fine-tune RoBERTa by training it on our train dataset and validate using validation dataset. This step requires a GPU/TPU since the training process can take 2 hours to more than 4 hours depending on Trainer parameters (CPU will take extremely long time to train like 68 hours). Using Trainer class, we make our Sanskrit MLM ready for training by using data collator and model instantiated in previous step. The Trainer class provides an API for feature-complete training in PyTorch, and it supports distributed training on multiple GPUs/TPUs, mixed precision for NVIDIA GPUs, AMD GPUs, and torch.amp for PyTorch. Trainer goes hand-in-hand with the TrainingArguments class, which offers a wide range of options to customize how a model is trained. Together, these two classes provide a complete training API.

In TrainingArguments we pass arguments to do evaluation and log it every 2500 steps (reducing number of steps increases training time), return only loss when performing evaluation and generating predictions, perform 1 training epoch (increasing number of epochs increases training time), update checkpoints every 10,000 steps with total save limit set as 2 (older checkpoints get deleted, so latest 2 checkpoints will remain at end), and specify the same output directory for saving model in which our custom tokenizer is saved. The remaining trainable parameters will retain their default values. We also create a function to use perplexity as metric to evaluate performance of our Sanskrit MLM:

$PPL = 2^{\text{loss}}$ where loss is prediction loss calculated during model evaluation. This formula is for MLMs, PPL is calculated differently for LLMs. PPL reflects how well a model predicts new data, with lower scores indicating less "surprise" (uncertainty) and better predictive accuracy. Thus, a low perplexity indicates good performance for the Sanskrit MLM. We finally start training the model using Trainer object. After training end, the saved model and tokenizer are uploaded to Hugging Face hub for immediate and anytime use instead of running the entire code again.

5.4.6 PREDICTING MISSING SANSKRIT SEQUENCES USING SANSKRIT MLM

After uploading our saved model and tokenizer to Hugging Face hub, we use the pipeline library to access them. The pipelines are a great and easy way to use models for inference. These pipelines are objects that abstract most of the complex code from the library, offering a simple API dedicated to several tasks, including Named Entity Recognition, Masked Language Modeling, Sentiment Analysis, Feature Extraction and Question Answering. Since our model is a MLM, we use the FillMaskPipeline and specify our uploaded Sanskrit MLM and it's tokenizer as model that will be used by the pipeline to make predictions and tokenizer that will be used by the pipeline to encode data for the model respectively. Finally, we pass our masked Sanskrit sentence which was cleaned after extraction into the fill-mask object. Masked sentence means we replace places where Sanskrit characters or words are missing with <mask> token in the sentence. Although multiple <mask> tokens can be added in the sentence, it predicts only one missing word at a time. For a single <mask> in the sentence, it'll predict top 5 Sanskrit words/characters along with raw model output corresponding to disjoint probabilities as prediction scores in descending order. Thus, the Sanskrit MLM is

the core component in the process for restoration of erased and destroyed Sanskrit words in Sanskrit manuscripts and Sanskrit text images.

```
'''
रामो सीतया सह वनम् अयोध्यायाः पित्रे गच्छति।
rāmo sītayā saha vanam ayodhyāyāḥ pitre gacchati.
Rama goes with Sita to the forest from Ayodhya for his father.
'''

fill_mask("रामो सीतया सह वनम् अयोध्यायाः <mask> गच्छति।")

[{'score': 0.0002475932124070823,
  'token': 12059,
  'token_str': ' कलकल',
  'sequence': 'रामो सीतया सह वनम् अयोध्यायाः कलकल गच्छति।'},
 {'score': 0.00017034010670613497,
  'token': 15205,
  'token_str': 'रमसमञ्ज',
  'sequence': 'रामो सीतया सह वनम् अयोध्यायाःरमसमञ्ज गच्छति।'},
 {'score': 0.0001526466803625226,
  'token': 43259,
  'token_str': '',
  'sequence': 'रामो सीतया सह वनम् अयोध्यायाः गच्छति।'},
```

Fig 5.4 : Original and masked Sanskrit sentences (top) and the predictions for masked token (bottom)

Perplexity was used rather than accuracy because if actual word is label and it predicts missing word as a synonym instead of the exact word, then it's wrong way to measure performance. There's also possibility that a completely different word can fit correctly in the sentence and match it's context. Thus, finding accuracy based on labels and evaluating based on the calculated accuracy is wrong way to measure performance. Common evaluation metrics like GLUE couldn't be used since corresponding benchmark datasets against which we have to test performance of our model are for English language models, not for Sanskrit models.

CHAPTER 6

RESULTS AND DISCUSSIONS

In this Project We had taken two images Original image and Destroyed image. Original Image presumably contains the original, undistorted version of some content. Destroyed/Input Image is labeled as "Destroyed", suggesting that intentionally distorted or altered from the original image. The degree of distortion or alteration would depend on how the image was processed or created. We are using Tesseract OCR to extract text from an image in the Sanskrit language. The function `clean_and_tokenize_text` is then applied to extracted text. This function removes punctuation and extra whitespace from the text and then print the extracted text. The `fill_mask` function is typically used with a pre-trained language model that supports masked language model (MLM) predictions. It replaces a masked token (`<mask>`) with the most likely words based on the context provided in the image.

The training process demonstrates a positive correlation between increasing validation loss and decreasing validation loss, indicating that as the model undergoes further training, its performance improves in accurately predicting masked tokens within the given context of the image.

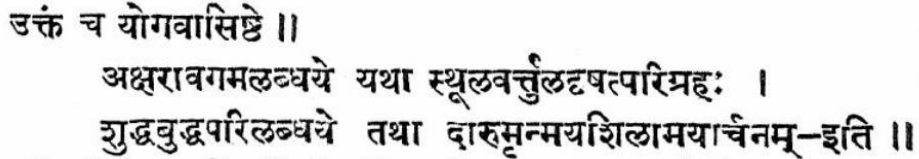
A photograph of a piece of paper with handwritten Sanskrit text. The text is written in black ink on a light-colored background. The first line reads 'उक्तं च योगवासिष्ठे ॥'. The second line reads 'अक्षरावगमलब्धये यथा स्थूलवर्तुलदृष्टपरिग्रहः ।'. The third line reads 'शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिलामयार्चनम्-इति ॥'.

Fig 6.1 : Original Image

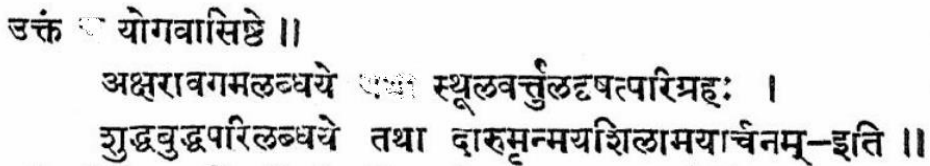
A photograph of the same piece of paper as in Fig 6.1, but the text is heavily distorted and blurred. The first line 'उक्तं च योगवासिष्ठे ॥' is still somewhat legible. The second line 'अक्षरावगमलब्धये यथा स्थूलवर्तुलदृष्टपरिग्रहः ।' is mostly illegible due to blurring. The third line 'शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिलामयार्चनम्-इति ॥' is also mostly illegible.

Fig 6.2 : Destroyed Image

[34203/34203 1:37:09, Epoch 1/1]		
Step	Training Loss	Validation Loss
2500	3.650100	3.608944
5000	3.248900	3.189839
7500	3.125500	3.031345
10000	2.930600	2.896438
12500	2.813800	2.771821
15000	2.762400	2.679377
17500	2.666500	2.595896
20000	2.582900	2.506189
22500	2.503900	2.438940
25000	2.456400	2.386344
27500	2.403200	2.340698
30000	2.386500	2.304336
32500	2.313800	2.286036

TrainOutput(global_step=34203, training_loss=2.8036817913079397, metrics={'train_runtime': 5832.1209, 'train_samples_per_second': 46.917, 'train_steps_per_second': 5.865, 'total_flos': 3177617708289024.0, 'train_loss': 2.8036817913079397, 'epoch': 1.0})

Fig 6.3 : Training and Validation loss

```
print("Perplexity:", evaluation_results["eval_loss"])
```

Perplexity: 2.2737033367156982

Fig 6.4 : Perplexity

उक्त ॐ योगवासिष्ठे ।
 ॐ ष ४
 अक्षरावगमच्छये स्वरूखन्तुखटटषत्परिग्रहः ।
 शुद्धबुद्धपरिग्रहे तथा दारमन्मयशिखामय चैनम्-इति ॥

Fig 6.5 : Extracted text

उक्त योगवासिष्ठे ॐ ष ४ अक्षरावगमच्छये स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिग्रहे तथा दारमन्मयशिखामय चैनम्-इति ॥

Fig 6.6 : Cleaned and Tokenized text

```

[[{'score': 0.00048118041013367474,
  'token': 11661,
  'token_str': 'पफ',
  'sequence': '<s>उक्तपफ योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वये<mask> स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},
{'score': 0.0004687323234975338,
  'token': 6138,
  'token_str': 'कलहस',
  'sequence': '<s>उक्त कलहस योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वये<mask> स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},
{'score': 0.00046200878568924963,
  'token': 917,
  'token_str': 'अल',
  'sequence': '<s>उक्त अल योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वये<mask> स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},
{'score': 0.00044733111280947924,
  'token': 2651,
  'token_str': 'वडव',
  'sequence': '<s>उक्त वडव योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वये<mask> स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},

{'score': 0.00040791768697090447,
  'token': 6477,
  'token_str': 'ोः',
  'sequence': '<s>उक्तोः योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वये<mask> स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},
[{'score': 0.00027602611226029694,
  'token': 12486,
  'token_str': 'रमसह',
  'sequence': '<s>उक्त<mask> योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वयेरमसह स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},
{'score': 0.00018151519179809839,
  ...
  'sequence': '<s>उक्त<mask> योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वयेरमसह स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'},
{'score': 0.00015254324534907937,
  'token': 8863,
  'token_str': 'दघन',
  'sequence': '<s>उक्त<mask> योगवासिष्ठे ॐ\u200c ष 4 अक्षरावगमच्छ्वयेदघन स्वरूखन्तुखटटषत्परिग्रहः । शुद्धबुद्धपरिस्थये तथा दारमन्मयशिखामय चैनम्\u200cइति \</s>'}}}

```

Fig 6.7 : Final Output

CHAPTER 7

FUTURE WORKS

The task of Optical Character Recognition (OCR) for ancient Sanskrit texts poses unique challenges due to the language's rich morphology and historical variations. Thus, an OCR designed specifically to efficiently identify Sanskrit characters of various scripts and handwriting styles can be created in future. Furthermore, we plan to extend our OCR methodology from extracting single line or double lines of Sanskrit text to extracting long sentences and paragraphs (including cluttered text), enabling comprehensive text analysis. While our current focus is on a single image containing classic Sanskrit script whose missing text predictions were additionally verified by Sanskrit dictionary/online Sanskrit translators and a Sanskrit school teacher, our future work aims to expand the scope to digitizing entire books, documents, and manuscripts written in Vedic Sanskrit which is the actual ancient Sanskrit and requires verification from Sanskrit historians and scholars for our MLM's predictions since Vedic Sanskrit was lost long ago and not in common use today. We also plan to design a model that automatically detects places where words or characters are missing in the Sanskrit text since our current Sanskrit MLM needs <mask> token to be manually inserted in places to predict missing words in Sanskrit text. The model can be further enhanced to predict multiple missing words or characters in input Sanskrit text simultaneously. By leveraging the power of MLMs, we anticipate significant advancements in Sanskrit OCR technology, facilitating the preservation and dissemination of invaluable cultural and historical knowledge encoded in Sanskrit manuscripts.

CHAPTER 8

CONCLUSION

In this project we had taken two types of images- Original image and Destroyed image for digitalized Sanskrit book images, and Destroyed Sanskrit manuscript images since original wasn't available. Original image contains the actual, undistorted version of Sanskrit content. Destroyed/Input image is distorted or altered from the original image, having stains, holes or rips in paper, and other forms of damage to text. The degree of distortion or alteration varies for different images, manuscript images being the most damaged unlike digitalized images which don't even need to undergo preprocessing before OCR most of the time. After extracting and cleaning the extracted text from input image through OCR, we manually insert <mask> token in the sentence at the position in the cleaned text where we want missing word/character to be predicted. Finally, we call our Sanskrit MLM through a pipeline for missing text prediction. It replaces the masked token (<mask>) with the most likely words based on the context provided in the input Sanskrit text. Thus, we've successfully completed our task of Sanskrit text restoration in old Sanskrit manuscripts and documents.

CHAPTER 9

REFERENCES

- [1] Ali Abbas Ali Alkhazraj, Baheeja Khudair, and Asia Mahdi Naser Alzubaidi. (pdf) ancient textual restoration using Deep Neural Networks: A literature review, July 4, 2023.
- [2] Bhavesh C Kataria, and Dr. Harikrishna B. Jethva. (PDF) Optical character recognition of sanskrit manuscripts using convolution neural networks, January 2021.
- [3] Wenjun, Zheng, Su Benpeng, Feng Ruiqi, Peng Xihua, and Chen Shanxiong. "EA-Gan: Restoration of Text in Ancient Chinese Books Based on an Example Attention Generative Adversarial Network - Heritage Science." SpringerOpen, March 1, 2023.
- [4] Yannis Assael, Thea Sommerschild, Brendan Shillingford, and Mahyar Bordbar. (pdf) restoring and attributing ancient texts using deep neural ..., March 2022.
- [5] Anna Tonazzini, Pasquale Savino, Emanuele Salerno, and Muhammad Hanif. Virtual restoration and content analysis of ancient degraded manuscripts, September 2019.
- [6] Puran Bhat, Kannagi Rajkhowa, "Deep Learning Model to Revive Indian Manuscripts" , International Journal of Science and Research (IJSR), Volume 12 Issue 4, April 2023, pp. 1365-1368.
- [7] Johnson, Kyle P., Patrick Burns, John Stewart, and Todd Cook. 2014-2020. CLTK: The Classical Language Toolkit.
- [8] Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "RoBERTa: [3]A Robustly Optimized BERT Pretraining Approach." arXiv, 2019.
- [9] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units." arXiv, August 31, 2015, v1. Last revised June 10, 2016, v5.

[10] Salazar, Julian, Davis Liang, Toan Q. Nguyen, and Katrin Kirchhoff. 2020. "Masked Language Model Scoring." In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2699-2712.

[11] Maheshwari, Ayush, Nikhil Singh, Amrith Krishna, and Ganesh Ramakrishnan. "A Benchmark and Dataset for Post-OCR Text Correction in Sanskrit." In Findings of the Association for Computational Linguistics: EMNLP 2022, 6258-6265. Chicago: Association for Computational Linguistics, 2022.

CHAPTER 10

APPENDIX-I

OUTPUT SNAPSHOTS

SANSKRIT MLM USING HUGGING FACE ROBERTA

```
!huggingface-cli login
```

```
To login, 'huggingface_hub' requires a token generated from https://huggingface.co/settings/tokens .
Token:
Add token as git credential? (Y/n) y
Token is valid (permission: write).
Cannot authenticate through git-credential as no helper is defined on your machine.
You might have to re-authenticate when pushing to the Hugging Face Hub.
Run the following command in your terminal in case you want to set the 'store' credential helper as default.



git config --global credential.helper store

Read https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage for more details.
Token has not been saved to git credential helper.
Your token has been saved to /root/.cache/huggingface/token
Login successful
```

```
from datasets import load_dataset
```

```
san_data = load_dataset("sanskrit_classic")
san_data
```

```

/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens)
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
Downloading data: 100%  16.9M/16.9M [00:00<00:00, 26.8MB/s]
Generating train split: 100%  342033/342033 [00:00<00:00, 842586.13 examples/s]
DatasetDict({
  train: Dataset({
    features: ['text'],
    num_rows: 342033
  })
})

```



```
dataset = san_data["train"]
dataset[:10]
```

```
{'text': ['रागादि-रोगान् सततानुषक्तान् अ-शेष-काय-प्रसृतान् अ-शेषान् ।\n',
  'औत्सुक्य-मोहा-रति-दाज् जघान यो ऽ-पूर्व-वैद्याय नमो ऽस्तु तस्मै ॥\n',
  'आयुः-कामयमानेन धर्मार्थ-सुख-साधनम् ।\n',
  'आयुर्-वेदोपदेशेषु विधेयः परम् आदरः ॥\n',
  'ब्रह्मा स्मृत्वायुषो वेदं प्रजापतिम् अजिग्रहत् ।\n',
  'सो ऽक्षिनौ तौ सहस्राक्षं सो ऽत्रि-पुत्रादिकान् मुनीन् ॥\n',
  'ते ऽग्निवेशादिकांस् ते तु पृथक् तन्त्राणि तेनरे ।\n',
  'तेभ्यो ऽति-विप्रकीर्णभ्यः प्रायः सार-तरोच्यः ॥\n',
  'क्रियते ऽष्टाङ्ग-हृदयं नाति-संक्षेप-विस्तरम् ।\n',
  'काय-बाल-ग्रहोर्ध्वाङ्ग-शल्य-दंष्ट्रा-जरा-वृषान् ॥\n']}]}
```

```
# Read and print the first 5 lines of sa.txt
with open('sa.txt', 'r') as sa_file:
    first_five_lines = [next(sa_file) for _ in range(5)]

# Print the first 5 lines
for line_number, line in enumerate(first_five_lines, start=1):
    print(f"Line {line_number}: {line}")
```

Line 1: रागादि-रोगान् सततानुषक्तान् अ-शेष-काय-प्रसृतान् अ-शेषान् ।

Line 2: औत्सुक्य-मोहा-रति-दाज् जघान यो ऽ-पूर्व-वैद्याय नमो ऽस्तु तस्मै ॥

Line 3: आयुः-कामयमानेन धर्मार्थ-सुख-साधनम् ।

Line 4: आयुर्-वेदोपदेशेषु विधेयः परम् आदरः ॥

Line 5: ब्रह्मा स्मृत्वायुषो वेदं प्रजापतिम् अजिग्रहत् ।

```
tokenizer.encode("तत्र सत्यस्य परमं निधानं यः न प्रियः ।")
```

```
Encoding(num_tokens=28, attributes=[ids, type_ids, tokens, offsets, attention_mask, special_tokens_mask,
overflowing])
```

```
tokenizer.encode("तत्र सत्यस्य परमं निधानं यः न प्रियः ।").tokens
```

```
['<s>',
  'à¤¤à¤¤',
  'à¥ì',
  'à¤°',
  'òà¤,à¤¤',
  'à¥ì',
  'à¤°à¤,',
  'à¥ì',
  'à¤°',
  'òà¤à¤à¤à¤',
  'à¤¤',
  'òà¤',
  'à¤',
  'à¤§',
  'à¤¼',
  'à¤°',
  'à¤¤',
  'à¤¤']
```

[34203/34203 1:37:09, Epoch 1/1]

Step	Training Loss	Validation Loss
2500	3.650100	3.608944
5000	3.248900	3.189839
7500	3.125500	3.031345
10000	2.930600	2.896438
12500	2.813800	2.771821
15000	2.762400	2.679377
17500	2.666500	2.595896
20000	2.582900	2.506189
22500	2.503900	2.438940
25000	2.456400	2.386344
27500	2.403200	2.340698
30000	2.386500	2.304336

32500 2.313800 2.286036

```
TrainOutput(global_step=34203, training_loss=2.8036817913079397, metrics={'train_runtime': 5832.1209, 'train_samples_per_second': 46.917, 'train_steps_per_second': 5.865, 'total_flos': 3177617708289024.0, 'train_loss': 2.8036817913079397, 'epoch': 1.0})
```

```
evaluation_results = trainer.evaluate()
```

[8551/8551 08:59]

```
print("Perplexity:", evaluation_results["eval_loss"])
```

Perplexity: 2.2737033367156982

```
tokenizer.push_to_hub("my-mini-project-model")
```

```
CommitInfo(commit_url='https://huggingface.co/Samuela39/my-mini-project-model/commit/f5a400f6c70db22e1a4a8965cb9da2a17b142d89', commit_message='Upload tokenizer', commit_description='', oid='f5a400f6c70db22e1a4a8965cb9da2a17b142d89', pr_url=None, pr_revision=None, pr_num=None)
```

```
from transformers import AutoModel

model = AutoModel.from_pretrained("./MySan2")
model.push_to_hub("my-mini-project-model")
```

Some weights of RobertaModel were not initialized from the model checkpoint at ./MySan2 and are newly initialized. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

README.md: 100% [green bar] 5.18k/5.18k [00:00<00:00, 329kB/s]

model.safetensors: 100% [green bar] 334M/334M [00:07<00:00, 47.2MB/s]

```
CommitInfo(commit_url='https://huggingface.co/Samuela39/my-mini-project-model/commit/bde5080cf2b9464ac8b67adcb9ce77995110eff0', commit_message='Upload model', commit_description='', oid='bde5080cf2b9464ac8b67adcb9ce77995110eff0', pr_url=None, pr_revision=None, pr_num=None)
```

OCR ON SANSKRIT TEXT IMAGES AND MANUSCRIPTS

Original image

उक्तं च योगवासिष्ठे ॥
अक्षरावगमलब्धये यथा स्थूलवर्तुलदृष्टपरिग्रहः ।
शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिलामयार्चनम्-इति ॥

Destroyed image

उक्तं च योगवासिष्ठे ॥
अक्षरावगमलब्धये यथा स्थूलवर्तुलदृष्टपरिग्रहः ।
शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिलामयार्चनम्-इति ॥

Extracted Text:

धिनायधौयर्श्वरस्मर्ना यथौक् वाचिन् 1 यमम षाथेती
नावि यम्य चिजिगीदलौ.चे िर्नी॥मेनर्व:दुजियानी:

```
from google.colab.patches import cv2_imshow
import cv2

image2 = cv2.imread(image_path)
cv2_imshow(image2)
```

धिनीयथायराध्रदल्लानी यथाकालपुत्राधिनी ॥ स्यांभयसमूहाथेती सत्यायमि
नहायिनी। यज्जसविजिगीयूनी सजायेग्रहमधिनी ॥ गेजवैरह्यमविद्यानी

धिनीयथायराध्रदल्लानी यथाकालपुत्राधिनी ॥ स्यांभयसमूहाथेती सत्यायमि
नहायिनी। यज्जसविजिगीयूनी सजायेग्रहमधिनी ॥ गेजवैरह्यमविद्यानी

EXTRACTED SANSKRIT TEXT CLEANING

```
#Extract and clean text from the binary image using OCR
extracted_text = pytesseract.image_to_string(image2, lang='san')
cleaned_and_tokenized_text = clean_and_tokenize_text(extracted_text)

print(extracted_text)
```

उक्तं च योगवासिष्ठे ।
र्ष 4
अक्षरावगमलब्धये <: स्थूलवर्तुलदृष्टपरिग्रहः ।
शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिखामय चैनम्-इति ॥

```
print(cleaned_and_tokenized_text)
```

उक्तं योगवासिष्ठे र्ष 4 अक्षरावगमलब्धये स्थूलवर्तुलदृष्टपरिग्रहः । शुद्धबुद्धपरिलब्धये तथा दारुमन्मयशिखामय चैनम्-इति ॥

```
print(cleaned_and_tokenized_text)
```

धिनायधौयर्श्वरस्मर्ना यथौक् वाचिन् 1 यमम षाथेती नावि यम्य चिजिगीदलौचे िर्नी॥मेनर्व:दुजियानी:

MISSING SANSKRIT WORD PREDICTION

#सः वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”

fill_mask("<mask> वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”")

```
[{'score': 0.2613905370235443,
  'token': 269,
  'token_str': 'न',
  'sequence': 'न वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”'},
 {'score': 0.22639994323253632,
  'token': 276,
  'token_str': 'स',
  'sequence': 'स वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”'},
 {'score': 0.058171600103378296,
  'token': 817,
  'token_str': 'ॐ',
  'sequence': 'ॐ वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”'},
 {'score': 0.020324496552348137,
  'token': 270,
  'token_str': 'य',
  'sequence': 'य वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”'},
 {'score': 0.019218409433960915,
  'token': 396,
  'token_str': 'तद',
  'sequence': 'तद वदति - “अद्य वर्षान्तस्य समारोहम् आगच्छतु”'}]
```