Predicting Missing Characters in Sanskrit Words

Tatha Pandey and Samar Sabie
pandeytatha@gmail.com, samar.sabie@utoronto.ca

Abstract - Understanding the character-level structure of the Sanskrit language would help historians and linguists study Sanskrit's linguistic history, and therefore the history of the peoples who lived that history. Sanskrit's story was an influential part of humanity's history in South Asia. Thus by better understanding Sanskrit, researchers and linguists would better understand humanity's story as well. In this paper, different machine learning models such as Decision Trees, Random Forest, and K-Nearest Neighbors were used to predict a missing character in Sanskrit words through the use of character n-grams as features for the above models to better inform the internal structure of Sanskrit.

Keywords - sanskrit n-grams, missing characters, decision trees, random forest, k-nearest neighbors, devanagari script

I. Introduction

Language, and its evolution, is an extremely important and central characteristic of humanity's history. Languages have often grown, moved, and adapted based on the lifestyles and movements of their speakers throughout the world. Language and humanity are thus intricately and substantially linked. Therefore the study of ancient languages and their structural aspects, in particular, are of immense importance to researchers, scientists, and the general public in enhancing our collective understanding of our past [14]. Clearly, more emphasis must be placed to further progress in this area.

The specific aim of this paper is to provide and analyze different techniques and strategies involving the use of machine learning in the context of predicting missing characters in Sanskrit words/tokens.

There are a few motivations for focusing this research on Sanskrit. Sanskrit was chosen due to its lasting influence and heritage in South Asia's linguistic and cultural history, being related to many different languages and dialects. Being an ancient language, there is a high probability that some of the ancient texts that are retrieved can be damaged. Thus a light-weight and accessible tool that can predict missing characters would be invaluable to researchers working in and analyzing the large body of Sanskrit texts and documents found in South Asia and elsewhere. In addition, linguists would be empowered to make further observations about Sanskrit and be able to clarify the structural similarities of Sanskrit with other related languages. By being able to more easily analyze the structure and evolution of languages, linguists and historians could then also analyze historical continuities and shifts in South-Asian communities and societies and find possible linguistic connections. Another major factor for choosing Sanskrit in this paper is that it has a relatively accessible corpora available in different scripts (including the Roman script) and transliteration formats. This supply of data, relatively large in comparison to other ancient languages, helps decrease the difficulty of finding relevant corpora while also helping to validate or reject the concept of applying similar machine learning approaches for further research in other ancient languages.

II. Background/Related Work

Spell Check

The problem of fixing spelling errors in texts is done via classic Natural Language Processing (NLP) [1, 2]. This problem is extremely relevant to that of predicting missing characters in Sanskrit texts and an analysis can thus provide valuable insights.

Spelling errors (defined as the erroneous addition, removal, or replacement of characters) are often present in machine learning pipelines and other valuable and critical areas, threatening serious consequences. These errors are often introduced due to software bugs, machine error, or even just human error in manual transcription. As an example, archaeologists and historians conducting research on ancient records could accidentally make mistakes during the copying process for ancient texts and documents at archaeological sites or in laboratories, thus reducing the quality of these modified records. Obviously, some error (human or otherwise) should be expected. However, these mistakes undeniably have negative impacts on the process of generating corpora; they reduce the quality of new corpora and mislead researchers and analysts with imprecise and inexact observations/conclusions.

When it comes to analyzing and understanding humanity's past through the lens of language and its evolution, some extra processing of new corpora could help catch many mistakes and improve the accuracy of future models trained on the data. Thus the problem of both finding spelling errors and doing so efficiently is of absolute importance: computer programs that are able to detect these errors are invaluable to the study of language. To be more precise, the process of building a routine to fix spelling errors in languages can help researchers uncover interesting and useful insights about the structure of these languages. This relates well with the aim of this paper to better understand and analyze patterns and structures in ancient languages like Sanskrit.

As an aside, spelling checkers are integrated into many text editors and integrated development environments (IDE) used by programmers and developers [2], making the spell check problem even more important.

In a paper regarding correcting spelling errors in text editors, Peterson defines digrams and trigrams as pairs and triples of consecutive characters, respectively [2]. The fact that some digrams and trigrams in English are rare can then be exploited to help identify spelling errors: diagrams or trigrams with a relatively low frequency may indicate that the word that they're inside of could be misspelled.

Vector Word Embedding

Another important problem in NLP is word vectorization. Word vectorization is the process of extracting numerical features from input word/textual data. The resulting vectors (a list of numerical values), also called feature vectors, are the embeddings (numerical representation/mapping) of the input word/textual data. Word vectorization is necessary since most popular and useful machine learning models and algorithms can't process string/text data and thus require numerical features in order to successfully be trained and produce meaningful outputs.

Determining an effective process for conducting word vectorization is an important decision, as different vectorization techniques will produce different embeddings. Depending on the problem at hand, these different feature embeddings could thus produce wildly different results and be subject to a diverse range of biases even with otherwise mostly similar models and overall machine learning architectures.

One common word vectorization process is called one-hot encoding. Given a list L of unique words with a length of k, one-hot encoding transforms each word in L into a k-dimensional vector filled almost entirely with zeros except with a one at the position corresponding to the current word. The advantage of this method is that it avoids any numerical bias that could have occurred by simply ranking the words in L arbitrarily. However, it drastically increases the number of features that a model needs to consider due to its sparse representation. In addition, its ineffectiveness at handling relationships between words leads to this embedding technique having trouble when considering out-of-vocabulary words (OOV).

A more popular word embedding method/library is fastText [4]. A major issue faced by

most word embedding methods such as one-hot encoding or even just hashing is that resulting embeddings struggle to take into account the internal structure and morphology of transformed words. This prevents machine learning models from taking advantage of the intricate patterns and relationships in word structure found in many morphologically rich languages that could have improved the accuracy and flexibility of models built on these embeddings in areas such as handling OOV, predicting word relationships, analyzing word context, and many other NLP applications. The fastText library is able to achieve much greater performance gains compared to previous word embedding models due to its application of character n-grams to better consider the internal structure of words when computing word embeddings [4].

N-grams

N-grams, at both a word and character level, are a popular tool utilized in many NLP applications, including the fastText library introduced by Bojanowski et al. Extending from Peterson's definition of digrams and trigrams, n-grams can be defined as a consecutive list of n tokens, usually words or characters. The advantage of character n-grams in particular, is that they help shed light on the inner morphologies of languages. This produces valuable results, such as in the case of the fastText word embedding system which incorporates the set of inner character n-grams as additional considerations for the embedding of a certain word.

Although n-grams can be effective tools in many languages and scripts, they are particularly effective in analyzing morphologically heavy languages as they have the flexibility to consider the basic morphemes that make up words and characters in such languages, especially when undertaking tasks such as predicting or identifying missing characters.

The effectiveness of n-grams also opens up other interesting questions. What are the best values for n? Does this differ for different languages and different scripts? If so, to what extent can they differ? These questions are partially answered in the previously described paper; however, they focus primarily on English and German, leaving some room for further analysis and work in the space [4].

N-grams are an important part of NLP and are used widely in research, either in direct implementations or through the use of abstractions such as word embedding systems like fastText. For example, the fastText word embedding system has been used as an abstraction to help provide vector embeddings for Sanskrit words to help train a classifier model to predict whether Sanskrit words are compound or not [3]. The success of that paper, and thus the success of the vectorization process used, not only shows the power of n-grams and their ability to handle OOV cases, but also the possible potential in many other Sanskrit applications.

Sanskrit Scripts

Sanskrit is an ancient language that has existed for approximately 3500 years according to the Britannica Encyclopedia and has been extremely influential in South Asian culture [5]. For example, many religious and spiritual documents in major religions such as Buddhism and Hinduism are commonly found written in Sanskrit or any of its variants.

Although Sanskrit has historically been written in many scripts, this paper will focus on the Roman script as it will require less processing and cleaning to prepare the data.

To use Roman script, we use transliteration, which is simply a process/approach for converting a message from one script (system of writing with its own unique alphabet and grammatical structure) to a different script in a manner that preserves the meaning and structure of the original message as accurately and correctly as possible [6].

The existence of standard transliteration schema for converting Sanskrit from the Devanagari script to the Roman script, such as IAST (**I**nternational **A**lphabet of **S**anskrit **T**ransliteration) and ITRANS (**I**ndian languages **Trans**literation), enables researchers to more easily process and thus understand and analyze hidden Sanskrit structures [7]. The IAST transliteration scheme uses diacritics, colloquially known as accents, to provide romanizations that can unequivocally be translated and that can mostly be supported by unicode standards [6, 7]. ITRANS, on the other hand, provides a transliteration scheme that uses ASCII characters [6].

Although ITRANS does provide a more limited range of transliteration options due to ASCII having fewer available characters than unicode for

encoding, it does make Sanskrit texts easier to process programmatically than if they had been in Devanagari or the IAST transliteration due to the relative ease of handling ASCII characters [6, 7]. Thus, ITRANS was chosen as the main transliteration scheme used for analyzing Sanskrit texts in this paper.

III. Methodology

Datasets: The dataset used is an available corpus of Sanskrit words in the ITRANS romanization scheme [13]. The dataset is from a Sanskrit document compilation website that has Sanskrit texts and corpora in different scripts and transliteration schema available for use. The corpus in question is available for educational uses [12]. The corpus (containing approximately 10,000 words along with corresponding definitions) was copy-pasted onto a text file named "sanskritDict.txt".

Setup: Visual Studio Code (VS Code) along with Jupyter Notebook (which came as a part of Anaconda) were used to conduct the experiment. VS Code was used for data preprocessing in order to "clean" the data and ready it for use in machine learning models. Jupyter Notebook was used to test different models, strategies and to compute the results/statistics using several metrics.

Process: Visual Studio Code (VS Code) was used to write a python script to "clean up" the data. For each Sanskrit word's ITRANS transliteration in "sanskritDict.txt", the script ("cleanUp.py") would store the character n-gram ending at a fixed index of the word, with the value of n being chosen beforehand. This data was organized into a csv format with columns representing the complete word, and the n characters of the n-gram of the current word. The characters were represented by their ASCII value. This data was then written onto a new file titled "updated_sanskrit_words.txt". The process of pruning and modifying the data ensured that the data was numerical and could safely be used as features to train and test models. This process was replicated in another Python script titled "cleanUp2.py", except the position of the missing character was randomized instead of being predetermined. The resulting data was written onto a separate file titled "updated_sanskrit_words2.txt". Next, Jupyter Notebook was used to train a variety of models by taking advantage of a popular library used in machine learning in Python called scikit-learn [8]. This library provides various machine learning models that have been optimized and efficiently implemented such as Random Forests, Decision Trees, and K-Nearest Neighbors, all of which are used as models in this paper for comparison.

Metrics: The models were compared based on their accuracy (the fraction of correct predictions) through the use of k-fold cross validation, a popular technique that involves splitting a dataset into k groups, or "folds". Then k separate experiments are conducted, where one group is chosen as the testing set and the other k-1 groups are consolidated into a training set for the model. The average of all k testing results is then used as a metric to help evaluate the success of the model. K-fold cross validation is widely used due to its simplicity and due to the fact that it allows machine learning models to avoid the dangers of overfitting, thus providing more accurate results.

IV. Results

As stated before, the three models used were the Random Forest, Decision Trees, and K-Nearest Neighbors. All implementations for these models were provided by Python's scikit-learn [8]. The accuracy of these models were compared using k-fold cross validation. The results are shown in Fig 1. The Y-axis in the Figure shows the accuracy of prediction, where an accuracy of 1.0 means that the model prediction is always correct and an accuracy of 0.0 means that the model prediction is always incorrect.

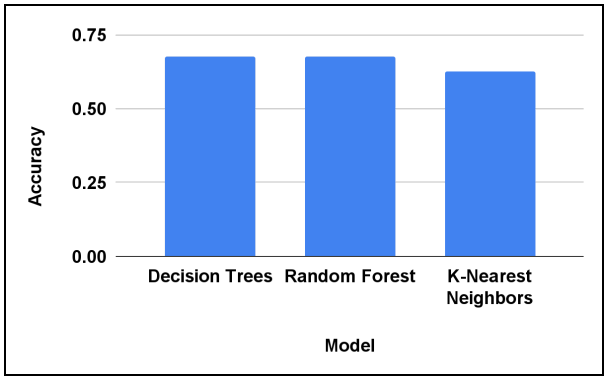

Fig. 1: Accuracy vs. Model Type trained using character 4-grams and fixing the last character as missing

Note that the models were trained based on fixing the last character as missing (using "updated_sanskrit_words.txt") in the training and testing sets and considering the previous 3 characters, thus using character 4-grams.

In addition, the models were also compared based on which index was marked as missing. Fig 2 show the results for each model when fixing the ultimate, penultimate, antepenultimate, and preantepenultimate characters, respectively. Note that the training data for the models involved the same file ("updated_sanskrit_words.txt") as before, with the only difference being that the initial Python script used to populate the text file was modified to change the fixed missing character value instead of simply defaulting to marking and building n-grams off of the last character.

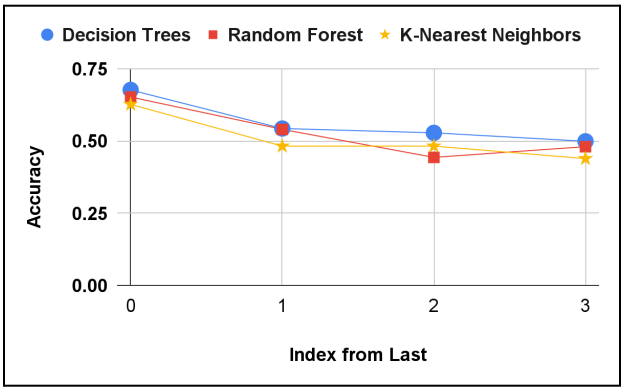

Fig. 2: Plot of Accuracy for each model based on the index of the character that would be chosen to be predicted. The 0 corresponds to the last index, the 1 corresponds to the second to last index, and so on.

The models were also tested based on randomizing the chosen index to test. In the previous setups, the index from last was initially chosen in the Python script titled "cleanUp.py" in order to repopulate the data of "updated_sanskrit_words.txt". The second Python script, titled "cleanUp2.py", was used, with the only difference being that the testing index for each word was randomized, thus providing more robust and powerful data to use that would help to further analyze the performance of all 3 models when given a less simpler scenario. The results are shown in Fig 3.

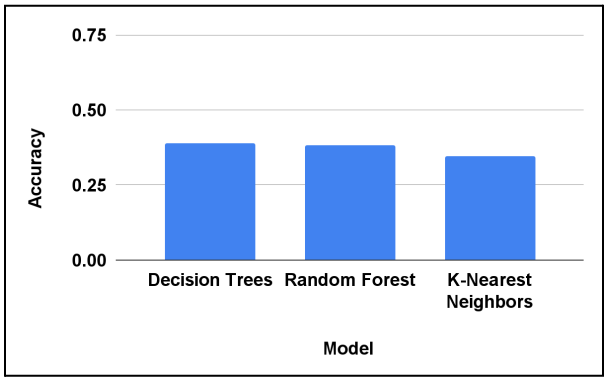

Fig 3: Plot of the accuracy score for each model. In this situation however, the chosen index to predict was randomized for each token of the training and testing data.

Finally, the accuracy scores of the models were compared based on the size of the character n-grams used to construct the training and testing datasets. Note that the chosen index to predict was fixed to be at the last character for each Sanskrit word; this was done in order to make the results cleaner and easier to understand. This was implemented using "cleanUp.py", by testing each value of n from two to five inclusive, and the results were computed for each of the models on Jupyter Notebook. These results are shown in Fig 4.

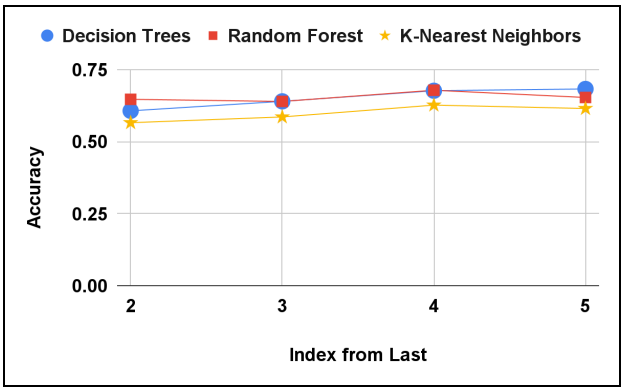

Fig. 4: Plot of the accuracy score for each model and for each character n-gram size for n = 2, 3, 4, 5.

V. Discussion

Firstly, all 3 models tested had higher accuracy scores when they were trained and tested on characters that appeared closer to the end of each Sanskrit token, with the notable exception of the preantepenultimate (fourth last) characters. This could simply be due to the nature of Sanskrit as a highly morphologically rich language. Sanskrit has many different affixes and compounding techniques between morphemes. This means that it would likely be easier to predict later characters in Sanskrit words since the character n-grams used as features to help predict those characters would align closer to important Sanskrit suffixes and would have less overlap with distinct morphemes. Viewing Sanskrit in such a way helps shed light on the accuracy score when considering the fourth last character. One possible explanation for this slight discrepancy is that the preantepenultimate character could on average be part of a different morphological unit than the remaining suffix in most Sanskrit words, thus allowing its character n-grams to align more closely with the morphemes that relate to it.

Another interesting trend is that the accuracy scores of every model increase as the size of the character n-grams used to train the models grows bigger. As mentioned before, one particular benefit of character n-grams is that they align relatively well with the affixes and morphemes inherent to most morphologically rich languages like Sanskrit [4]. Thus the use of larger values of n in character n-grams allows for the easier analysis of the internal structure of Sanskrit words and thus likely provides more accurate results. Results could suffer if extremely large values of n were chosen due to the chance of overfitting. In addition, larger values of n for character n-grams would be less likely to accurately represent the morphemes of Sanskrit and would be more likely to hold a combination of different morphemes and sandhi shifts [9]. This would likely lead to poorer results.

The drawback of the models used is that they performed less successfully when the index from the end was randomized. This is important since the nature of the problem at hand demands more flexibility to be able to predict missing characters at various positions. One possible solution to this problem could be to adjust the parameters for the machine learning models used. The Random Forest classification model could likely be optimized, for example, to improve its performance over the Decision Tree classification model to provide further performance gains. Also, the size of the character n-grams used to train the models could be further adjusted to hunt for possible improvement.

Another drawback that may have hurt the ability of the models to more accurately predict missing characters when the indices of those characters were randomized is that of the transliteration scheme used. While easier to process, the ITRANS transliteration scheme doesn't retain as much information about the structure of Sanskrit words as the IAST transliteration scheme. The IAST transliteration scheme ensures that no information is lost from Devanagari, thus providing a compatible transliteration that is more accurate than ITRANS. Although it would have been slightly harder to incorporate the IAST transliteration scheme into the machine learning pipeline and source IAST datasets, it could have provided more accurate data with fewer contradictions and inconsistencies in the character level structure of Sanskrit words that would have allowed all the models to perform more successfully.

VI. Conclusion

The goal of this paper was to develop models to successfully predict a missing character in Sanskrit words. Three models that were used were

the Random Forest, Decision Tree, and K-Nearest Neighbors and the training and testing data involved the use of character n-grams to better allow the models to understand the complexities and intricacies of Sanskrit's morphological structure. The models were tested and compared based on the index from the end of the missing character chosen, and the size of the character n-grams used. The results showed that the models generally performed better when predicting missing characters towards the end of Sanskrit words, and when using larger character n-grams. Using larger character n-grams and predicting missing characters towards the end of the word likely increases the probability of the character n-grams matching with common suffix morphemes, thus increasing the likelihood that they would be predicted correctly by the classifier models.

When randomly choosing indices to predict, the accuracy scores of the models declined. While still respectable, the decline of the scores can partly be explained by the increased complexity of the models having to detect and understand multiple types of affixes and having to deal with sandhi; these problems were generally less frequent when having to deal with suffix-like morphemes. The models trained and demonstrated in this paper could likely be improved through further parameter tuning by taking advantage of the functionality provided by scikit-learn models in specifying the architecture of a chosen machine learning model. Another area where large improvements in accuracy could be seen could be in using the more robust IAST transliteration system over the ITRANS transliteration system. While harder to process, Sanskrit corpora in the IAST transliteration scheme could likely be incorporated into the pipeline relatively easily through the use of several popular transliteration websites. Incorporating IAST datasets into the training and testing of models would help to further improve the accuracy of the models due to IAST's unambiguous translations with the Devanagari script. Another potential avenue through which more accurate models could be developed could be through the use of neural networks. Complex versions of neural network architectures have already seen use in tackling similar character level problems such as through translating Hawaiian text [10] and even in predicting missing characters in Ancient Greek [11]. The field of Sanskrit character prediction could thus see even further improvement through any of these avenues.

References

[1] **Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, *7*(3), 171–176. https://doi.org/10.1145/363958.363994**

[2] **Peterson, J. L. (1980). Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, *23*(12), 676–687. https://doi.org/10.1145/359038.359041**

[3] **P. B, C. Chandran V, S. Bhat, et al. "A Machine Learning Approach for Identifying Compound Words from a Sanskrit Text"**

[4] **Piotr Bojanowski, Edouard Grave, Armand Joulin, et al. 2016. Enriching word vectors with subword information. arXiv preprint arXiv:1607.04606.**

[5] **Encyclopædia Britannica, inc. (2024, April 26). *Sanskrit language*. Encyclopædia Britannica. https://www.britannica.com/topic/Sanskrit-language**

[6] **S. Chandra, V. Kumar, Anju. 2019. DS-IASTConvert: An Automatic Script Converter between Devanagari and International Alphabet of Sanskrit Transliteration. IJRAR Feb 2019, Vol 6. Issue 1**

[7] **N. Chaturvedi, R. Garg. 2018. A Tool for Transliteration of Bilingual Texts Involving Sanskrit. In Computational Sanskrit & Digital Humanities**

[8] **F. Pedregosa, G. Varoquaux, A. Gramfort, et al. 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825– 2830.**

[9] **B. Kessler. 1994. Sandhi and Syllables in Classical Sanskrit. Published in** The proceedings of the Twelfth West Coast Conference on Formal Linguistics. Stanford, CA

[10] B. Shillingford, O. Parker Jones. Recovering Missing Characters in Old Hawaiian Writing, 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP).

[11] **Assael, Y., Sommerschield, T., Shillingford, B. *et al.* Restoring and attributing ancient texts using deep neural networks. *Nature* 603, 280–283 (2022).** https://doi.org/10.1038/s41586-022-04448-z

[12] Online Sanskrit Dictionary: https://sanskritdocuments.org/dict/dictall.txt

[13] Sanskrit Documents in Devanagari and IAST, https://sanskritdocuments.org/dict/dictall.txt

[14] Sommerschield, T., Assael, Y., Pavlopoulos, J., Stefanak, V., Senior, A., Dyer, C., et. al. (2023). Machine Learning for Ancient Languages: A Survey. Computational Linguistics, 49(3).