The University of Southern Mississippi
The Aquila Digital Community

Honors Theses Honors College

5-2024

Reviving the Past: Enhancing Language Models with Historical Text Optimization

Heather D. Broome

Follow this and additional works at: https://aquila.usm.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation
Broome, Heather D., "Reviving the Past: Enhancing Language Models with Historical Text Optimization" (2024). *Honors Theses*. 955.
https://aquila.usm.edu/honors_theses/955

This Honors College Thesis is brought to you for free and open access by the Honors College at The Aquila Digital Community. It has been accepted for inclusion in Honors Theses by an authorized administrator of The Aquila Digital Community. For more information, please contact Joshua.Cromwell@usm.edu, Jennie.Vance@usm.edu.

Reviving the Past: Enhancing Language Models with Historical Text Optimization

by

Heather Dixieanna Broome

A Thesis
Submitted to the Honors College of
The University of Southern Mississippi
in Partial Fulfillment
of Honors Requirements

May 2024

Approved by:

Nick Rahimi, Ph.D., Thesis Advisor,
School of Computing Sciences and Computer Engineering

Sarah Lee, Ph.D., Director,
School of Computing Sciences and Computer Engineering

Joyce Inman, Ph.D., Dean
Honors College

ABSTRACT

Recent advancements in Natural Language Processing (NLP) have brought attention to the significant potential that exists for widespread applications of Large Language Models (LLMs). As demands and expectations for LLMs rise, ensuring efficiency and accuracy becomes paramount. Addressing these challenges requires more than just optimizing current techniques; it urges novel approaches to NLP as a whole. This study investigates novel data preprocessing methods designed to enhance LLM performance by mitigating inefficiencies rooted in natural language, particularly by simplifying the complexities presented by historical texts. Utilizing the classical text *The Odyssey* by Homer, two preprocessing techniques are introduced: tokenization of names and places, and substitution of outdated terms. After optimizing a Long Short-Term Memory (LSTM) network to perform well with the original text, the study examined how each methodology influenced the model's efficiency and precision through the analysis of training time and loss metrics. Tokenization significantly reduced the training time of the model by simplifying complex names and places, albeit with a slight degradation of output quality. Substitution of outdated terms not only decreased the training time of the model but also improved the model's comprehension. This study successfully demonstrated novel preprocessing methods for improving the efficiency of LLMs, providing insight for future research and contributing to the ongoing mitigation of NLP challenges.

Keywords: natural language processing, large language models, data preprocessing techniques, historical text simplification, LSTM network, LSTM optimization

DEDICATION

This thesis is dedicated to Rachandeep Singh Chahal. You have believed in me since the very beginning. I am eternally grateful for you.

ACKNOWLEDGMENTS

First and foremost, I extend my deepest gratitude to the Honors College for their continuous support of my personal and academic growth. Furthermore, I would like to thank the School of Computing Sciences and Computer Engineering for the knowledge I carry with me as I continue my education.

I would like to thank all of the amazing advisors and professors that have shaped my journey. Dr. Nick Rahimi, thank you for providing me with mentorship and opportunities—none of this would have been possible without you. Dr. Sarah Lee, thank you for your continuous support and leadership. These past four years have been a truly remarkable journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

LIST OF ABBREVIATIONS

GPT	Generative Pre-trained Transformer
LM	Language Model
LLM	Large Language Model
LSTM	Long Short-Term Memory
ML	Machine Learning
NLP	Natural Language Processing
RNN	Recurrent Neural Network

CHAPTER I: INTRODUCTION

Human language has evolved over the past six million years, a testament to our unique biology, anthropology, and cognitive sciences. This evolution is responsible for the limitless complexity of natural language and the immense juxtaposition it creates against the rigid, arithmetical nature of computing. This contrast has long rendered Natural Language Processing (NLP) a significant challenge within the field of computational science. However, in the age of Big Data, the capabilities of computational models to decipher and emulate natural language are unparalleled. As the field seeks novel ways to optimize and build upon these advancements, there is a need to adapt natural language itself to the properties of computation.

In recent years, Language Models (LMs) have empowered NLP in ways previously considered impossible. The introduction of the Markov chain in 1906, which predicted sequences of letters in a text, marked the beginning of what we now consider language processing. The concept of sequential predictions has evolved from simple, hand-computed frameworks to incredibly complex Large Language Models (LLMs). Among the most advanced of these today is GPT-4, which predicts sequences of text using 1.7 trillion parameters [1], [2].

The sophistication of LLMs has come with a growing need for data and computational resources. A major determinant of the computational demand and runtime for LLM training is the quality and breadth of training data. Although models have continuously evolved to learn from broad, suboptimal text, the adaptation of training data remains insufficient. Moreover, as the scale and architectural complexity of these models grow, there are escalating concerns regarding the sheer energy consumption and computational infrastructure required to support them [3]. Addressing these issues is crucial not only for optimizing LLMs, but also to ensure that hardware and energy sources can keep pace with the momentum in NLP progression.

This research aims to explore novel strategies for preprocessing historical text, proposing and analyzing potential techniques for increased efficiency of LM training. Large datasets, especially those containing historical text, carry obsolete writing patterns and vocabulary that contribute little to the objectives of most LMs. By identifying and mitigating these variables, this research aims to improve the speed and computational demand of LMs while preserving the integrity of classical text.

The importance of this research is found in its real-world applications. As language evolves and datasets continue to grow exponentially, the need to mitigate computational demands will only increase. If methods of data optimization can be applied in seconds while reducing a model's training time by minutes or hours, then the value of these techniques will be made clear. Furthermore, optimized datasets could be reusable indefinitely, allowing future models to continually reap their benefits. Effective pre-processing techniques will contribute to the field of NLP indefinitely.

In preparation for the core methodology to be applied, an LM is first designed, implemented, and optimized to best fit the selected dataset. The core methodology is then implemented, which contains two strategies: tokenization of proper nouns and substitution of outdated terms. Through tokenization, the dataset's complexity is lowered by reducing names and places into simple placeholders that are reinstated after the model produces output. Through substitution, text normalization is applied to archaic language, transforming it into a modern counterpart, which reduces the model's archaic vocabulary and increases the context surrounding modern vocabulary. Together, these methods reduce the complexity and size of tokens that a model must comprehend during training. This reduction is measured by a comparison of training losses, validation losses, and training times before and after each technique is applied.

CHAPTER II: LITERATURE REVIEW

Over recent years, unprecedented strides have been made in machine interpretation and generation of natural language. Developments have greatly revolved around artificial neural networks for sequence modeling tasks, which involve training neural networks on sequentially ordered data such as text, audio, or video. This literature review will focus on three pivotal types of neural networks: Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Transformers. Each of these have contributed significantly to the advancement of NLP and sequence modeling.

The first of these network architectures to be developed was the RNN. RNNs use feedback loops between each sequential step in model training, allowing the network to retain memory over time. While feedback loops played a crucial role in the advancement of sequence modeling, they introduced the problem of vanishing and exploding gradients [4]. In traditional RNNs, gradients may vanish when the weight passed by a feedback loop is less than 1 and may explode when that weight is more than 1. This is caused by a compounding multiplication of gradients during backpropagation. The architecture of an RNN is illustrated in [5, Fig. 1]. Each network cluster represents the introduction of a sequential token to the training set, such as a letter, word, or phrase. Between each time step, denoted by 'T', weights are shared to retain memory and context.

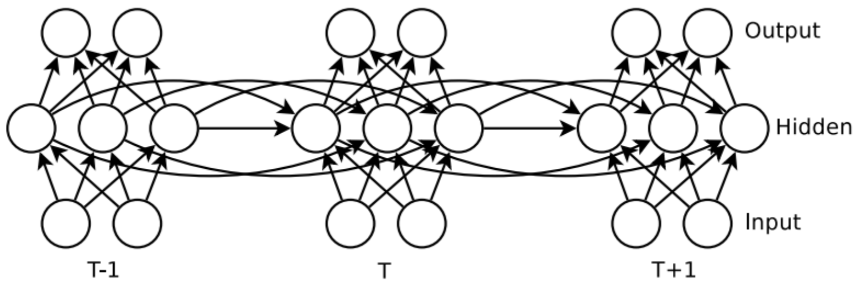

Figure 1. RNN Architecture

Unlike traditional RNNs, LSTM networks effectively combat vanishing and exploding gradients [6]. As illustrated in [7, Fig. 2], each cell in the LSTM architecture consists of input, forget, and output gates. These cells are each connected by a distinct cell state and hidden state. Long-term memory is represented by the cell state that is modified via gates, utilizing a sigmoid activation function to ensure that information is carefully added and forgotten. Short-term memory is represented by a hidden state that is modified by activations (represented by 'a') and biases (represented by 'b'). These states methodically control the sharing of memory across each sequential input, making the network less susceptible to the gradient issues of traditional RNNs. This structure is further described in the Methodology section. In practice, the LSTM structure allows a network to capture long-term dependencies across large datasets, which is crucial for capturing sequential information and maintaining continuity in narrative and style.

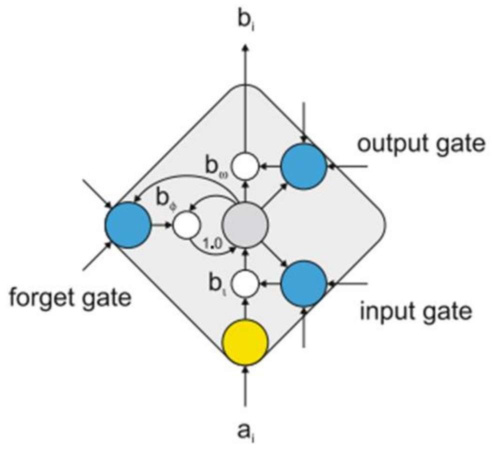

Figure 2. LSTM Memory Cell

Despite the significant improvements in long-term contextualization made possible by LSTMs, challenges still exist for the models. Specifically, the limited window of context available for each hidden state may prevent broader contextual dependencies from being recognized. This challenge led to the introduction of attention mechanisms, which enabled neural networks to dynamically focus on sections of the input sequence and assign sections with varying levels of importance [8]. These mechanisms paved the way for the invention of Transformer models, avoiding recurrence entirely for a route of self-attention [9]. This is especially notable, as it facilitated the parallelization of LLM training and the creation of large pre-trained models.

In addition to utilizing novel attention mechanisms, the Transformer model follows a novel, two-part structure: an encoder and a decoder. The encoder processes input while the decoder generates output. The complexity of these models is significantly higher than that of an LSTM, and both the encoder and decoder contain self-attention structures with multiple heads and feed-forward neural networks. This multi-head attention structure enables the dynamic focus capabilities of the model, allowing the Transformer to capture long-term context that otherwise may have been missed.

In this research, an LSTM framework will be employed. Two critical considerations are responsible for this decision: computational efficiency and the nature of the dataset. Transformer models, while able to handle highly complex tasks, have a parallel structure that requires significant computational resources. Furthermore, the dataset employed in this research is relatively small, making an LSTM model particularly fitting. LSTM models have lower computational demands while still being able to effectively capture long-term dependencies and context in smaller datasets. Nonetheless, acknowledgment of the Transformer model is necessary due to its unprecedented significance in the field of NLP.

The Transformer model is responsible for substantial advancements in NLP and revolutionary inventions such as GPT (Generative Pre-trained Transformer) models.

These models utilize vast datasets to achieve state-of-the-art fluency and understanding across a broad array of language-understanding tasks [1]. The transformer-based architecture utilized by GPT allows the model to process tokens in parallel, making it significantly more efficient and scalable than many past models. GPT-4 [10], one of the most recent GPT models, performs extremely well on complex natural language questions and knowledge-intensive trivia quizzes. This performance underscores the continuing strides being made by the GPT series. More broadly, it represents the continually growing capabilities of NLP models to master generalized knowledge without a need for task-specific tuning.

While breakthroughs in network architecture have significantly advanced NLP, techniques for the structural analysis of language within these models have also played a crucial role. Parsing and masking have enhanced accuracy and efficiency within LLMs. Parsing involves the analysis of elements within a sentence and their relationships to each other [11]. Traditional RNNs and LSTMs linearly parse text and generate sentences through probabilities. Not only does this lead some models to struggle with grammatically correct sentence structure, but it also risks the overlooking of dependencies between detached words. Parsing has become standard in advanced models, including parallelized models such as Transformers.

Masking has also been beneficial for the advancement of LLMs. Masking involves randomly obscuring portions of input text and prompting the model to predict those portions [12]. This forces the model to rely on its understanding of context rather than sequentially related probabilities alone. This more complex standard for predictions during training allows for a better understanding of context and nuances to be established.

Both parsing and masking have allowed LLM development to emphasize bidirectional context, broader generalization, and syntactical relationships.

Evolution from the linear processing of RNNs to the parallel and bidirectional processing of Transformer models creates a clear trajectory for the advancement of NLP. Improved model architecture and data analysis methods have undoubtedly revolutionized the capabilities of LLMs. Still, there are many improvements to be made regarding computational cost, overall efficiency, and output accuracy for each model type. This makes it especially critical to ensure a nuanced approach to model selection and research as a whole, with thorough consideration given to the intricacy of human communication.

CHAPTER III: METHODOLOGY

Creating the Dataset

The foundation of building an effective LM is the deliberate selection and preprocessing of training data. In this research, *The Odyssey* by Homer was selected as training data for the model. The specific translation, completed by Samuel Butler in 1900, contains approximately 610,000 characters, making it a sufficiently large dataset for a robust NLP model. This translation was chosen for various reasons. To begin, the work is within the public domain, facilitating research use without copyright restrictions. Additionally, Butler's translation retains the integrity of the original writing. The use of this slightly modern translation enhances the readability of the original text and the output of models trained on it. This will facilitate a smoother process for future modifications to the data as well as interpretations made from the resulting model output.

The Odyssey is an excellent source for training a LM. The epic contains elaborate storytelling, rich vocabulary, and a long-form structure. These elements, along with the mixture of narrative storytelling and direct dialogue, allow the model to capture many of the complex factors of natural language. Perhaps the most critical attribute of *The Odyssey* is its consistent style. Consistency enables the LM to quickly develop a stable comprehension of the language, which is an especially important aspect given the limited hardware available for this research. Although the hardware being used is powerful, it does not compare to that which is used to train LLMs on broad and diverse datasets.

In order to ensure that the dataset was ready for use, some basic preprocessing tasks were performed. Data preprocessing is necessary for any NLP research to ensure that unintended characters and patterns are removed from the dataset before training takes place. The preprocessing of *The Odyssey* involved the removal of excess whitespace. Originally, newline characters were used to control the width of each row of text for readability purposes. While formatting is useful for human readers, the newlines were unrelated to the content of the story and would confuse a model during training. A Python script was written to identify and eliminate each superfluous newline character. This was carefully performed as newline characters indicating the separation of stanzas were retained. In the original text file, stanzas were indicated by two consecutive newline characters, allowing them to be easily differentiated from a simple line break. Given the literary significance of poetic structure, retaining stanza breaks ensured the preservation of the epic's narrative and style.

Building the Language Model

This research utilized the LSTM network, an RNN most notable for its ability to capture long-term dependencies in sequence modeling tasks. The LSTM model was constructed using the Python programming language [13] and the PyTorch deep learning framework [14], selected for the relative ease of model implementation and experimentation. The implementation of a model as complex as the LSTM network is challenging, and optimizing the model requires rigorous testing.

The LSTM model has a multi-gate structure with the following components: the Forget Gate, Input Gate, and Output Gate [7]. The Forget Gate determines which memory will be discarded from the cell state. The Input Gate determines which memory will be added to the cell state. The Output Gate determines which memory will contribute to the output of the given cycle. This architecture allows the LSTM to carefully retain and discard short-term and long-term memory, which is crucial for maintaining long-term dependencies while discarding unimportant information.

After implementing the LSTM model, the next essential step was hyperparameter tuning. This involved fine-tuning various configurations to ensure the model was learning effectively from the training data. In the initial phase of this process, the parameter optimization was conducted using only Book 1 of the 24 books in *The Odyssey*. The approach aimed to refine the model's parameters in a confined environment, allowing the tuning process to be guided early-on without exerting the time and resources necessary to train on the entire text. However, after switching to the entire text, it quickly became clear that the optimized parameters did not scale effectively. This emphasized the challenges that come with long, complex datasets. Subsequently, the parameter fine-tuning process followed many iterative adjustments which are detailed below.

The previously optimized parameters marked the initial values in this refinement process. The model commenced with 100 epochs, a batch size of 128, a learning rate of 0.001, 128 hidden dimensions, 2 layers, and sequence lengths of 100. At this point, the output was unintelligible.

In pursuit of high-quality output, the epochs were first adjusted to 200. The epochs, which represent the iterations a model executes when learning from training data, proved to become more effective with this increase. The model's writing was more coherent, and the training and validation losses were both lowered while remaining proportionate, indicating that overfitting had not been introduced by the change. Although there were some misspellings and still no clear storyline in the output, this improvement justified the increased runtime, so the adjustment was retained. The next refinement was in adjusting the sequence lengths to 200, a decision which seemed appropriate given the large increase in training data. The sequence length represents the amount of tokens used to train a model during each epoch, and it is helpful to increase its size with complex training sets where broader context is essential for the model's comprehension. This increase improved the model and eliminated misspellings. However, the output still struggled to establish a clear storyline.

Next, the learning rate was reduced to 0.0001 and epochs were increased to 300. The learning rate, which determines the step size made by the neural network as it adjusts weights, improved the performance when decreased. The increased epochs helped reduce the potential shortcomings that come with smaller steps, as simply decreasing the learning rate may hinder the model's ability to converge to the optimal loss given the restricted number of iterations. An adjustment to the batch size, specifically a decrease to 64, tested the impact of a smaller sample size taken within each epoch. This modification, despite the increase in training time, was kept due to its benefit in establishing clearer output. Lastly, the number of hidden dimensions was increased to 256. Although more dimensions within a neural network increase training time, the adjustment was deemed necessary as it helped the model better understand nuances and complex relationships between entities in the text. At this point, the model was considered optimized. Other parameter values were tested yet did not benefit the model. These included increasing the number of layers in the model to 3. Theoretically, this could have helped the model better understand complex nuances, but it ultimately made the output less coherent.

Lastly, an appropriate optimizer and loss function were selected. In neural networks, an optimizer manages learning rate throughout training. The optimizer used in this research, Adam [15], is well-suited for dynamically adjusting the learning rate effectively for complex datasets. A loss function measures the difference between a model's predicted sequence and actual sequence during training. The loss function used in this research, CrossEntropyLoss [16], penalizes incorrect sequence prediction during training. These functions are useful for enhancing and guiding text generation.

The exploration of hyperparameter testing was concluded at this stage. The model had reached an excellent level of performance, which was deemed suitable for the research objectives. The final hyperparameters, displayed in Table 1, represent a configuration that effectively balanced the complexity of the dataset with the present computational constraints.

Epochs	300
Batch Size	64
Learning Rate	0.0001
Hidden Dimensions	256
Number of Layers	2
Optimizer	Adam
Loss Function	CrossEntropyLoss

Table 1. Parameters of LSTM Model

A crucial measurement for the performance of an LM is loss, which quantifies the difference between the model's predictions and actual data. The lower the loss, the better the performance. Two measurements for loss are used: training and validation loss. These values represent the model's ability to generalize training data and make accurate predictions using unseen validation data. The losses achieved by the final LSTM are displayed in Fig. 3. The optimal loss progression should display a steady decline that slowly tapers off near its end, and this pattern can be observed in the fine-tuned LSTM. The initial rapid decrease in loss signifies a period of quick learning, followed by an eventual convergence indicating stability once the maximum amount of learning has taken place. An important detail of this convergence is the close proximity between the training and validation losses, suggesting that the model is not overfitting. This means that the model is predicting sequences well even on unseen data, rather than simply memorizing the training data.

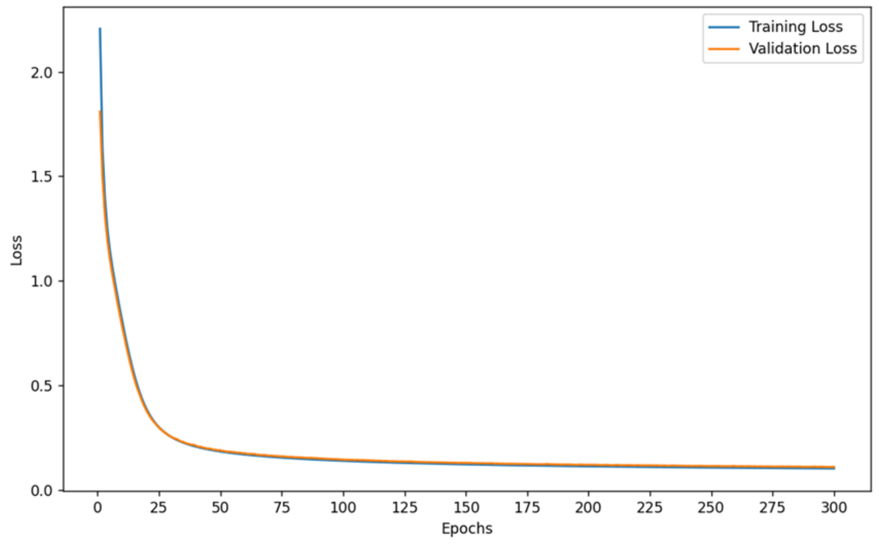

Figure 3. Original Training and Validation Loss Over Time

The model's refinement is demonstrated clearly by the comparison of text outputs before and after fine-tuning. This comparison is displayed in Table 2, which contains the first 500 characters of the LSTM model's output before and after hyperparameter tuning. Before fine-tuning, the model's output was largely made up of obscure series of characters. However, post-tuning, the output was intelligible, coherent, and accurately reflecting the writing style of *The Odyssey*. This progression signified the model's suitability for its intended use in this research. The final model took approximately 1387 minutes to train with the final epoch reporting a training loss of 0.1018 and a validation loss of 0.1092.

Before	After
They saw," said he, 'Ulysses, noble, did not father, finging restre Diume or indeed come away over her head and son. "When we will anson to Dulicher side whether of priead when you can think that island such a noble monsmend among themselves. Then Ulysses instagry, not even the gods down the ship bade for all that was chopping themselves of washed them to Aegout. Telemachus, marriel in husband of many grorned him when the ship could be away the body of my men with a good burden help you to the h	The ghosts of our ships, we found our comrades lamenting us, and anxiously awaiting our return. We ran our vessel upon the sands and got out of her on to the sea shore; we also landed the Cyclops' sheep, and divided them equitably amongst us so that none might have reason to complain. As for the ram, my companions agreed that I should have it as an extra share; so I sacrificed it on the sea shore, and burned its thigh bones to Jove, who is the lord of all. But he heeded not my sacrifice, and onl

Table 2. LSTM Output Before and After Parameter Tuning

Throughout this research, the LSTM training was conducted on an NVIDIA GeForce RTX 3070 Ti Laptop GPU. This high-performance GPU provided the power necessary to train on an extensive dataset within a reasonable timeframe.

Optimizing the Dataset

The primary objective of this research was to utilize and measure the effects of various preprocessing strategies on the efficiency of an LSTM model. These strategies focused on normalizing the text, modernizing outdated language, and simplifying complex vocabulary. The section outlines each preprocessing technique and the considerations behind their selection.

An essential consideration behind each technique is the preprocessing through Python scripts, a decision that underscores the pursuit of efficiency. Dynamic preprocessing techniques such as masking are extremely common in LLMs, and rightfully so, yet they require reapplication with each iteration of training. Python scripts, on the other hand, modify the dataset once into a new version that can be used indefinitely. In this research, scripts allow for a one-time application of tokenization for names and places and substitution for text modernization. Once preprocessed, the dataset can be used repeatedly without redundant measures increasing the computation load. Furthermore, an explicit approach preserves reversibility for each method, which is useful for restoring natural language in output. This method optimizes computational resources, improves the reproducibility of the study, and allows for more direct conclusions to be drawn on the effects of individual modifications.

Tokenizing Names and Places

The first technique employed was the tokenization of names and places. Throughout *The Odyssey* and many archaic works in general, proper nouns are particularly complex. While these terms are essential placeholders for understanding the narrative as a whole, the exact composition of the term itself is typically unnecessary. Therefore, representing names and places with generic tokens can help reduce the model's vocabulary size and, for works that use particularly long terms, significantly reduce the dataset as a whole. Furthermore, generic tokens such as 'N1', 'N2', etc. for names and 'P1', 'P2', etc. for places can help the model begin to quickly infer when unfamiliar tokens represent a name or place. The use of a numeric token, which would not occur naturally in *The Odyssey*, also eliminated false positives when reinstating terms via a Python script in the output of the LSTM, as the placeholders would not have any direct matches in the original dataset like a nickname or alias might.

In order to perform this tokenization, a Python script was designed to systematically identify and tokenize the names and places within *The Odyssey*. The script first extracted all words that began with a capital letter, as these were likely to be proper nouns. To further refine the resulting list, the script was tailored to exclude words that occurred less than ten times. This not only simplified the output into a list of more relevant terms, but it also significantly reduced the inclusion of words that were capitalized simply because they were at the beginning of sentences. Although that exclusion could have been performed by ignoring words that came directly after a newline or period, it would have conflicted with names or places being used to open a sentence. This refinement allowed the remaining terms to be manually filtered.

When manually filtering the remaining terms, many criteria were considered. Major characters who were central to the storyline and appeared frequently in the text, such as Ulysses, Telemachus, and Penelope, were tokenized. Similarly, significant locations like Ithaca and Troy were also tokenized. However, gods, mythical entities, and terms referring to groups of people were intentionally not tokenized in this process. Gods and mythical entities often represent broad thematic elements that may be diluted by tokenization. This dilution would be exacerbated if the optimized text were combined with a larger training dataset which may contain specific works relating to mythology. Collective names like Achaeans and Phaeacians may create a similar issue, potentially confusing the model with their varying contexts and connotations. Therefore, while the tokenization of all names and places may streamline computational processing, the preservation of gods, mythical entities, and collective terms in their original form was necessary for the integrity of both the text and model.

The tokenized names, along with their total occurrences in the text, appear in Table 3. Similarly, the tokenized places appear in Table 4.

Name	Token	Freq.	Name	Token	Freq.	Name	Token	Freq.
Ulysses	N1	580	Euryclea	N12	33	Eurylochus	N23	13
Telemachus	N2	259	Eurymachus	N13	28	Irus	N24	13
Penelope	N3	104	Aegisthus	N14	25	Dolius	N25	12
Eumaeus	N4	71	Pisistratus	N15	23	Polybus	N26	11
Menelaus	N5	63	Helen	N16	22	Medon	N27	11
Alcinous	N6	62	Melanthius	N17	22	Theoclymenus	N28	11
Antinous	N7	58	Achilles	N18	17	Philoetius	N29	11
Laertes	N8	45	Atreus	N19	16	Icarius	N30	10
Nestor	N9	38	Autolycus	N20	14	Peleus	N31	10
Agamemnon	N10	37	Nausicaa	N21	13	Amphinomus	N32	10
Calypso	N11	34	Arete	N22	13	Eurynome	N33	10

Table 3. Tokenized Names

Place	Token	Freq.
Ithaca	P1	91
Troy	P2	68
Pylos	P3	39
Olympus	P4	12
Dulichium	P5	12
Crete	P6	12
Egypt	P7	11
Lacedaemon	P8	11

Table 4. Tokenized Places

Substituting Outdated Terms

The second technique employed was text normalization. This method involved converting outdated terms into their modern equivalents. There were various reasons for the modernization of certain terms, and the choice to do so was fully dependent on the context of the dataset and its uses. For this research specifically, outdated language was reduced due to its complex structure and spelling. Additionally, this study aimed to improve efficiency by enabling the dataset to be reused, creating the potential for future research to train the dataset alongside modern texts. Reducing the contrast between historical text and modern language helped to limit the model's need to learn outdated vocabulary. Modernization across languages would allow future models to reduce their vocabulary size while simultaneously gaining context on the meaning and history of terms. It should be noted, however, that in cases where historical language is necessary, such an adjustment would not have been appropriate.

This approach involved a multi-phased process to identify outdated terms and replace them with their modern equivalents. First, a Python script was utilized to generate a list of the top 100 most frequently used words within *The Odyssey*. While helpful, this list predominantly consisted of prepositions, pronouns, and names, which limited its potential insights into outdated terminology. To refine this search, the spaCy NLP library [17] was employed to specifically extract nouns and adjectives, addressing the problems present in the initial approach. The spaCy library simplified the extraction of outdated terms, as these word classes were likely to consist of outdated terminology. The final phase of this process involved manual review, where sections of the text were skimmed for any frequent, clearly outdated words that may have been missed by previous methods.

Finally, a list of outdated terms was compiled. Each term was evaluated carefully with consideration given to its exact meaning and whether an exact modern synonym existed. For instance, the term "voyage" was retained despite its mildly archaic tone, as it conveyed the sense of sea travel that is central to the epic's narrative. Similarly, the word "swineherd," while outdated, was preserved due to the absence of a modern synonym that explicitly implied the herding of pigs. Each substitution meticulously balanced the importance of contextual richness with computational efficiency. By converting specific terms to their modern equivalents, the training data became more compatible with modern texts and, in many cases, reduced the length and complexity of certain tokens.

Incidentally, the vocabulary present in the text was reduced, as 20 of the 26 terms that served as replacements were already present in the original text. This overlap in vocabulary allowed the model to learn more efficiently, as it minimized the introduction of new terms and reinforced existing linguistic patterns. The final substitutions, along with their total occurrences, are illustrated in Table 5. It should be noted that past, active, and plural versions of some terms were also present in the text and were substituted accordingly for consistency. The frequency counts in Table 5 include the occurrences of these varied forms. For a detailed enumeration of the exact substitutions, their frequencies, and indications of whether a term was pre-existing in the original text, refer to Appendix B.

Original Token	Replacement Token	Frequency
shall	should	207
round	around	130
whereon	where	55
foremost	leading	12
cloak	jacket	57
cloister	walkway	60
ere	before	9
hecatomb	sacrifice	15
perish	die	26
supper	dinner	37
quarrel	fight	11
estate	property	26
to-morrow	tomorrow	14
till	until	155
yoke	harness	13
folly	foolishness	10
converse	talk	17

Table 5. Outdated Term Substitutions

CHAPTER IV: RESULTS

The results of this research represent a thorough examination of the impacts of tokenization and substitution on the LSTM model. The ultimate goal was to observe an increase in efficiency for the LSTM model's computational efficiency.

Impact of Tokenization

The process of tokenizing common character names and places into standardized representations played a critical role in increasing the computational efficiency of the model. This tokenization reduced the size of the training dataset, allowing the model to train more quickly. This resulted in a total training time of 1316 minutes, demonstrating that the tokenization decreased training time by approximately 71 minutes when compared to the original model.

Despite the improvement in training time, tokenization resulted in lower quality output from the model. The final training and validation losses were higher than those of the original model, with the final epoch achieving a training loss of 0.1021 and a validation loss of 0.1097. This signifies an increase of training loss by 0.0003 and validation loss by 0.0005. The complete training and validation loss curves after tokenization are depicted in Fig. 4. This deterioration of the model's understanding of the text reveals a disadvantage of tokenization. This can be observed in Table 6, which displays the first 500 characters of the model's output after tokenization.

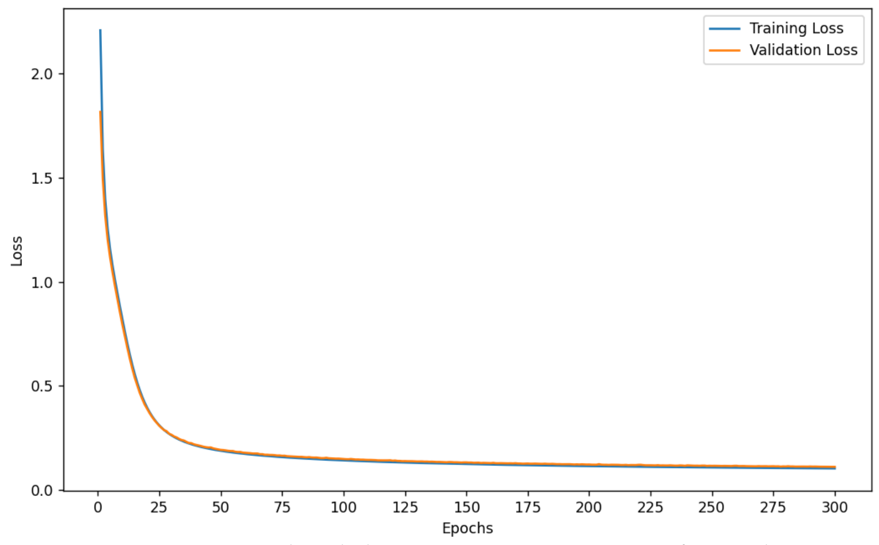

Figure 4. Training and Validation Loss Over Time After Tokenization

Output After Tokenization
They took me in and was kind to me, but I need say no more about this, for I told you and your noble wife all about it yesterday, and I hate saying the same thing over and over again." Thus did he speak, and they all held their peace throughout the covered cloister, enthralled by the charm of his story, they became mend youthough they wanted me by the hand. "fout is not your fill, either bronze- for there wereding meen than Ne6 may be pig to harn work. When they had done dinner they the jar roffi

Table 6. Model Output After Tokenization

Impact of Substitution

The process of substituting archaic words for their modern equivalents successfully increased the computational efficiency of the model. This substitution reduced the complexity of the training dataset by condensing the model's overall vocabulary to a more manageable size. This simplification resulted in a total training time of 1361 minutes, demonstrating that the substitution decreased training time by approximately 26 minutes when compared to the original model.

The final training and validation losses were also lower than those of the original model, with the final epoch achieving a training loss of 0.0999 and a validation loss of 0.1079. This signifies a decrease of training loss by 0.0019 and validation loss by 0.0013. The complete training and validation loss curves are depicted in Fig. 5, illustrating the model's learning progression over time. This graph demonstrates the model's improved understanding of the text, highlighting the effectiveness of archaic term substitution for optimizing language model efficiency and performance. Table 7 displays the first 500 characters of the model's output after substitution.

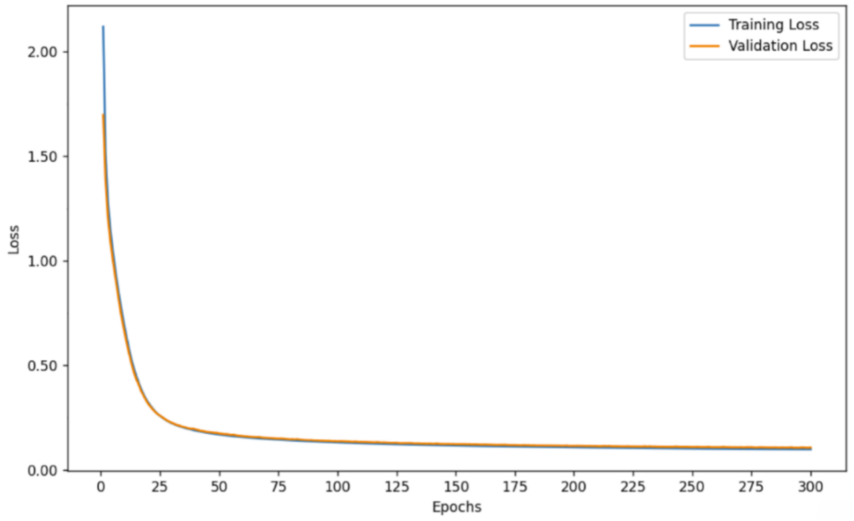

Figure 5. Training and Validation Loss Over Time After Substitution

Output After Substitution
The old man of the sea told me, so much will I tell you in full. He said he could see Ulysses on an island sorrowing bitterly in the house of the nymph Calypso, who was keeping him prisoner, and he could not reach his home, for he had no ships nor sailors to take him over the sea.'This was what Menelaus told me, and when I had heard his story I came away; the gods then gave me a fair wind and soon brought me safe home again." With these words he moved the heart of Penelope. Then Theoclymenus said

Table 7. Model Output After Substitution

CHAPTER V: DISCUSSION

This study explored preprocessing strategies to boost LM efficiency by simplifying complex and outdated terms within historical text. The methodology was primarily motivated by the escalating scale, complexity, and computational demands of LLMs. While traditional preprocessing approaches focus on formatting and quality improvements, this research introduced the idea of addressing redundant and dynamically evolving natural language constructs as essential preprocessing considerations. By focusing on these constructs and emphasizing their presence in outdated language, methodologies were successfully developed and applied to improve the efficiency of LSTM training on historical text.

The application of the tokenization process to frequently occurring names and places within *The Odyssey* significantly enhanced the training efficiency of an LSTM model. This improvement, demonstrated by a significant reduction in training time, highlighted the computational load imposed by complex historical names and places. Nonetheless, this research noted a slight decline in output quality, suggesting that artificial placeholders may interfere with the learning trajectory of LMs.

The substitution process, which involved replacing outdated terms with their modern equivalents, not only improved training efficiency but also strengthened the model's understanding of the text. This improvement, demonstrated by a reduction in both the training time and loss metrics, highlighted the computational demand and complexity that is introduced by obsolete language. By modernizing the language, the model gained a finer understanding of historical nuances, thereby boosting both its applicability and efficiency.

These results offer compelling evidence for the effectiveness of preprocessing techniques specifically tailored to historical text. These techniques aid in the improvement of various NLP applications, particularly those that require significant amounts of data spanning various eras. As natural language continues to evolve, these findings support the incorporation of similar techniques to mitigate challenges introduced by that evolution. Furthermore, these techniques offer insight for future NLP research, emphasizing the importance of striking a balance between the simplification of data and the retention of linguistic richness.

For future work, this research proposes many compelling investigations. The effects of tokenization and substitution on leading LLMs, especially Transformers, justify additional investigation. Furthermore, the development of an automated end-to-end framework for the tokenization and substitution of complex historical terms could streamline the incorporation of classical texts in certain NLP applications. An end-to-end framework would also facilitate research on the scalability of these techniques and their effects across various languages and dialects. This research not only advances our understanding of NLP techniques but also lays the foundation for future work in harnessing the full potential of historical texts within broader applications.

CHAPTER VI: CONCLUSION

This investigation has underscored the significant potential of historical considerations in data preprocessing techniques. By refining complex historical literature, it is not only possible to improve the efficiency and accuracy of LMs but also to expand their applications. By investigating the effects of tokenization and substitution for outdated terms in historical text, this research demonstrated notable improvements in LSTM model efficiency and accuracy. The demonstration and analyses of these techniques provide insight for future research and contribute to the ongoing mitigation of NLP challenges. Advanced historical text processing contributes not only to technology but to the preservation and accessibility of historical literature, signifying a pivotal step toward a deeper understanding and portrayal of evolution within NLP

APPENDIX A: LSTM IMPLEMENTATION CODE

```
import torch
import torch.nn as nn
import numpy as np
import random
from torch.utils.data import Dataset, DataLoader
import time
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

print(torch.cuda.is_available())
print(torch.cuda.get_device_name(0))

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

torch.manual_seed(42)
np.random.seed(42)
random.seed(42)

# Load in dataset
with open('C://Users//path_to_dataset.txt, 'r', encoding='utf-8') as f:
    text = f.read()

# Encode characters to integers
chars = tuple(set(text))
int2char = dict(enumerate(chars))
char2int = {ch: ii for ii, ch in int2char.items()}
encoded = np.array([char2int[ch] for ch in text])
seq_length = 200
batch_size = 64

# Create training examples
def create_sequences(encoded_text, seq_length):
    inputs = []
    targets = []

    for i in range(0, len(encoded_text) - seq_length, 1):
        sequence_in = encoded_text[i:i + seq_length]
        sequence_out = encoded_text[i+1:i+seq_length+1]
        inputs.append(sequence_in)
        targets.append(sequence_out)

    return inputs, targets
```

```
inputs, targets = create_sequences(encoded, seq_length)

# Define the dataset
class CharDataset(Dataset):
    def __init__(self, inputs, targets):
        self.inputs = inputs
        self.targets = targets

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self, idx):
        return torch.tensor(self.inputs[idx], dtype=torch.long),
torch.tensor(self.targets[idx], dtype=torch.long)


train_inputs, val_inputs, train_targets, val_targets = train_test_split(
    inputs, targets, test_size=0.1, random_state=42)  # 10% for validation

train_dataset = CharDataset(train_inputs, train_targets)
val_dataset = CharDataset(val_inputs, val_targets)

train_dataloader = DataLoader(train_dataset, batch_size, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size, shuffle=False)

# Define the model
class LSTMModel(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(LSTMModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.output_size = output_size

        self.lstm = nn.LSTM(input_size, hidden_dim, n_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x, hidden):
        out, hidden = self.lstm(x, hidden)
        out = out.contiguous().view(-1, self.hidden_dim)
        out = self.fc(out)
        return out, hidden

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new_zeros(self.n_layers, batch_size, self.hidden_dim,
device=device),
                  weight.new_zeros(self.n_layers, batch_size, self.hidden_dim,
device=device))
        return hidden


# Define the hyperparameters
input_size = len(chars)
output_size = len(chars)
hidden_dim = 256
n_layers = 2
epochs = 300

model = LSTMModel(input_size, output_size, hidden_dim, n_layers).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

# Start timer
print ("Training started.")
start_time = time.time()

train_losses = []
val_losses = []

model = model.to(device)

# Training phase
for epoch in range(epochs):
    model.train()
    train_loss = 0
    for inputs, targets in train_dataloader:
        inputs, targets = inputs.to(device), targets.to(device)

        current_batch_size = inputs.size(0)

        h = model.init_hidden(current_batch_size)

        h = tuple(h_state.to(device) for h_state in h)
        inputs = nn.functional.one_hot(inputs.to(torch.int64),
input_size).float().to(device)
        targets = targets.to(torch.int64).to(device)

        model.zero_grad()
        output, h = model(inputs, h)
        output = output.view(-1, model.output_size)
        loss = criterion(output, targets.view(-1))
        train_loss += loss.item()
        loss.backward()
        optimizer.step()


    average_train_loss = train_loss / len(train_dataloader)
    train_losses.append(average_train_loss)

    # Validation phase
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for inputs, targets in val_dataloader:
            inputs, targets = inputs.to(device), targets.to(device)

            current_batch_size = inputs.size(0)
            h = model.init_hidden(current_batch_size)

            h = tuple([each.data.to(device) for each in h])

            inputs = inputs.to(torch.int64)
            inputs = nn.functional.one_hot(inputs,
input_size).float().to(device)
            targets = targets.to(torch.int64).to(device)

            output, h = model(inputs, h)
            output = output.view(-1, model.output_size)
            loss = criterion(output, targets.view(-1))
            val_loss += loss.item()

        average_val_loss = val_loss / len(val_dataloader)
        val_losses.append(average_val_loss)

    print(f'Epoch: {epoch + 1}, Training Loss: {average_train_loss:.4f},
Validation Loss: {average_val_loss:.4f}')
```

```
# Stop the timer
print("Training finished!")
end_time = time.time()
elapsed_time = end_time - start_time
print(f"Training completed in: {elapsed_time // 60} min {elapsed_time % 60}
sec")

# Text generation
def predict(model, character, hidden=None, top_k=None):
    model.eval()
    model.to(device)


    x = torch.tensor([[char2int[character]]], dtype=torch.long).to(device)

    x = nn.functional.one_hot(x, num_classes=len(chars)).float().to(device)

    if hidden:
        hidden = tuple([each.data.to(device) for each in hidden])

    out, hidden = model(x, hidden)

    p = nn.functional.softmax(out, dim=1).data

    if top_k is None:
        top_ch = torch.arange(len(chars))
    else:
        p, top_ch = p.topk(top_k)

    p = p.cpu().numpy().squeeze()
    top_ch = top_ch.cpu().numpy().squeeze()

    char = np.random.choice(top_ch, p=p/p.sum())

    return int2char[char], hidden


def generate_text(model, size, prime='The', top_k=None):
    model.eval()

    chars = [ch for ch in prime]
    h = model.init_hidden(1)
    for ch in prime[:-1]:
        char, h = predict(model, ch, h, top_k=top_k)
    char = prime[-1]
    for _ in range(size):
        char, h = predict(model, char, h, top_k=top_k)
        chars.append(char)

    return ''.join(chars)


generated_text = generate_text(model, 2000, prime="The", top_k=5)
print(generated_text)
```

APPENDIX B: COMPLETE LIST OF TERM SUBSTITUTIONS

Original Token	Replacement Token	Frequency	Pre-existing
shall	should	207	✓
round	around	130	✓
whereon	where	55	✓
foremost	leading	12	✓
cloak	jacket	42	×
cloaks	jackets	15	×
cloister	walkway	40	×
cloisters	walkways	20	×
ere	before	9	✓
hecatomb	sacrifice	4	✓
hecatombs	sacrifices	11	✓
perish	die	9	✓
perished	died	17	✓
supper	dinner	34	✓
suppers	dinners	3	✓
quarrel	fight	10	✓
quarreled	fought	1	✓
estate	property	25	✓
estates	properties	1	×
to-morrow	tomorrow	14	✓
till	until	155	✓
yoke	harness	8	✓
yoked	harnessed	5	✓
folly	foolishness	10	×
converse	talk	16	✓
conversing	talking	1	✓

Table 8. Complete List of Term Substitutions

REFERENCES

[1] H. Li, “Language Models: Past, Present, and Future,” *Commun. ACM*, vol. 65, no. 7, pp. 56-63, June 2022.

[2] X. Ding et al., “HPC-GPT: Integrating Large Language Model for High-Performance Computing,” In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023)*, pp. 951-960, Nov. 2023.

[3] C. Kachris, "A Survey on Hardware Accelerators for Large Language Models," arXiv preprint arXiv:2401.09890, Jan. 2024.

[4] Y. Bengio, P. Simard, and P. Frasconi, “Learning Long-Term Dependencies with Gradient Descent is Difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157-166, Mar. 1994.

[5] I. Sutskever, M. Martens, and G. Hinton, “Generating Text with Recurrent Neural Networks,” In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017-1024, Jan. 2011.

[6] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, Dec. 1997.

[7] M. Sundermeyer, R. Schlüter, and H. Ney, “LSTM Neural Networks for Language Modeling,” In *Proceedings of the 13th Annual Conference of the International Speech Communication Association*, pp. 194-197, Sep. 2012.

[8] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, “Attention-Based Models for Speech Recognition,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, vol. 1, pp. 577–585, Dec. 2015.

[9] A. Vaswani et al., “Attention is All you Need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 6000-6010, Dec. 2017.

[10] T. Brown et al., “Language Models are Few-Shot Learners,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, July 2020.

[11] G. Ferraro and H. Suominen, “Transformer Semantic Parsing,” in *Proceedings of the 18th Annual Workshop of the Australasian Language Technology Association*, pp. 121-126, Dec. 2020.

[12] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in *Proceedings of the 17th Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 4171–4186, June 2019.

[13] G. Rossum and F. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[14] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proceedings of the 33rd Conference on Neural Information Processing Systems*, pp. 8024-8035, Dec. 2019.

[15] D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in *Proceedings of the 3rd International Conference for Learning Representations*, Dec. 2013.

[16] A. Mao, M. Mohri, and Y. Zhong, "Cross-Entropy Loss Functions: Theoretical Analysis and Applications," in *Proceedings of the 40th International Conference on Machine Learning*, pp. 23803-23828, July 2023.

[17] S. Jugran, A. Kumar, B. Tyagi, and V. Anand, “Extractive Automatic Text Summarization using SpaCy in Python & NLP,” in *Proceedings of the 2021 International Conference on Advance Computing and Innovative Technologies in Engineering*, pp. 582–585, Mar. 2021.