Simplifying Urdu for Gen Z: An Urdu to Roman-Urdu Transliterator

Sumair Ijaz Hashmi*
24100004@lums.edu.pk
Dept of Computer Science
Lahore University of Management Sciences (LUMS)
Pakistan

Naafey Aamer*
24100117@lums.edu.pk
Dept of Computer Science
Lahore University of Management Sciences (LUMS)
Pakistan

ABSTRACT

Neural machine translation (NMT) is a widely studied and common application in Natural Language Processing. The recent rise in data-driven approaches, primarily of attention-based encoder-decoder models, has revolutionized the resulting solutions that perform well on various performance metrics. However, despite the mind-boggling accuracy of such approaches, their application in the local languages of Pakistan is understudied. Hence, for the final project of our course on Natural Language Processing, we present the design, implementation, and analysis of an Urdu-to-Roman-Urdu transliterator based on recurrent neural networks. Our final attention-based encoder-decoder model performs well on medium-sized Urdu sentences (sentence length=20-30), with a BLEU-1 score of 74.3 and a ROUGE-1 score of 0.9. This tool contributes significantly to the development of Urdu language technologies as it will ease the learning of Urdu for the younger generations.

ACM Reference Format:
Sumair Ijaz Hashmi and Naafey Aamer. 2023. Simplifying Urdu for Gen Z: An Urdu to Roman-Urdu Transliterator. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION & RELATED WORK

The first Pakistani generation to grow up in the onset of technology, generation Z, although fluent in their mother tongue, has difficulty writing and reading Urdu in the traditional Arabic (or Nastaliq) script. In contrast, they are more comfortable with the transliterated version of Urdu written in the Roman script due to their constant exposure to social media. This pattern is becoming increasingly evident with each incoming generation of youngsters. Keeping this in mind, we built a model that transliterates Nastaliq script Urdu into Roman script Urdu. Recognizing the evolving preferences and challenges the younger generation faces, our decision to develop a transliteration model for Nastaliq script Urdu (from here on, referred to as Urdu) into Roman Urdu aims to bridge the gap between the traditional script and the current needs of the digital age.

Fatima et al. [6] highlight the main challenges that are novel with Urdu Language Processing by emphasizing the morphological richness of the language (several variants existing for the same word), a lack of syntactical and spelling standardization (leading to issues in word tokenization and stemming), word sense disambiguation, and orthographical standardization (for problems in transliteration, as the same word can have several different spellings). Additionally, the lack of large datasets and proper research for studying the low-resource language may also be attributed to the limited nature of non-WEIRD and specifically Pakistani researchers in the NLP community, as is true in any other domain of scientific research [5]. These challenges and issues motivate us to contribute to this field.

Although this is a fairly unexplored topic, the closest work we could find was that of Mehreen Alam and Sibt ul Hussain's "Deep Learning based Roman-Urdu to Urdu Transliteration" [2] published in the International Journal of Pattern Recognition and Artificial Intelligence in 2021. Alam and Hussain compiled the first-ever Roman-Urdu-to-Urdu parallel corpus of 1.1 million sentences. This corpus contains variable-length sequences of up to 252 words. Using this corpus, they developed three state-of-the-art encoder-decoder-based sequence-to-sequence models, where encoders and decoders are built using two separate Recurrent Neural Networks (RNNs). Their findings show that a layered LSTM bidirectional encoder-decoder model with attention performed the best during evaluation based on various performance metrics, such as BLEU and ROUGE.

Outside of the domain of Urdu, numerous implementations of machine translation exist, such as those by Google [7] and other various open-source models provided by independent entities [1], with varying degrees of implementation details based on the basic encoder-decoder architecture using recurrent neural networks or transformers.

We took inspiration from the work by Alam and Hussain [2], and in this paper, we report on our attempt to develop a similar encoder-decoder model using an attention mechanism to perform the task of Urdu to Roman Urdu sentence transliteration. We discuss the design, implementation, training, and empirical analysis of our model on the parallel corpus of Urdu and Roman Urdu sentences provided by Alam and Hussain [2]. We conclude by providing directions for improvement and future work.

*Both authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DATA PREPARATION & PROCESSING

The first step of any Deep Learning based task is to gather and preprocess large amounts of data to feed into your model(s). Due to the low-resource nature of Urdu, as explained earlier, we were initially unable to find large publicly-available datasets of Roman Urdu and Urdu text in abundance. We finally stumbled upon the dataset collected by Alam and Hussain and decided to use this to train and evaluate our neural network approach for the transliteration task.

2.1 Dataset

The dataset collected by Alam and Hussain is a parallel corpus of **1.1 million** Roman-Urdu and Urdu sentences. The authors have made this dataset publicly available in the form of two plain-text files [1]. Due to computational restraints and the complexity of our model, we only utilized **60%** of this text corpus. In total, the corpus used for our model contained **664,293 sentences**. The authors collected corpus utilizing a variety of methods, such as web scraping, local crowd-sourcing, and computer-aided machine translation using iJunoon [2]. Please look at the original paper by Alam and Hussain for more details on the data collection and annotation [2]. Figure 1 shows some example Urdu sentences and their transliterated Roman Urdu counterparts to visualize the data. Although a quick visual scan of the dataset seemed to put our trust in its reliability, we should point out that while playing with the data to explore it, we came across certain sentences with inconsistent transliterations, such as those in Table 2. For the purposes of this paper, we have not made any attempt to fix these anomalies.

Our corpus has a total Urdu vocabulary of **34,203 words** and a total The Roman-Urdu vocabulary of **32,525 words**. It is noteworthy, and has also been pointed out by Alam and Hussain, that the Urdu vocabulary has more words than Roman-Urdu words. The reason for this disparity is because, in Urdu, many words are written with a space between them but are semantically and contextually interpreted as single words only, whereas the same compound word structure is usually represented as a single word in Roman Urdu. Additionally, the corpus contains sentences of variable lengths, from less than **10** up to a maximum length of **252**. Before addressing this immense variability in sentence length, we applied extensive data-cleaning techniques to ensure the corpus's reliability and prepare it for further analysis and modeling.

2.2 Preprocessing

The data was divided into train and test datasets with a train test split of **90:10** to provide to our upcoming model. The same preprocessing steps were applied to all the Roman Urdu and Urdu sentences in both the training and test datasets. For the preprocessing, we carried out standard data preprocessing strategies for machine translation as provided in various implementations of NMT [3] [4]. In particular, the following steps were followed:

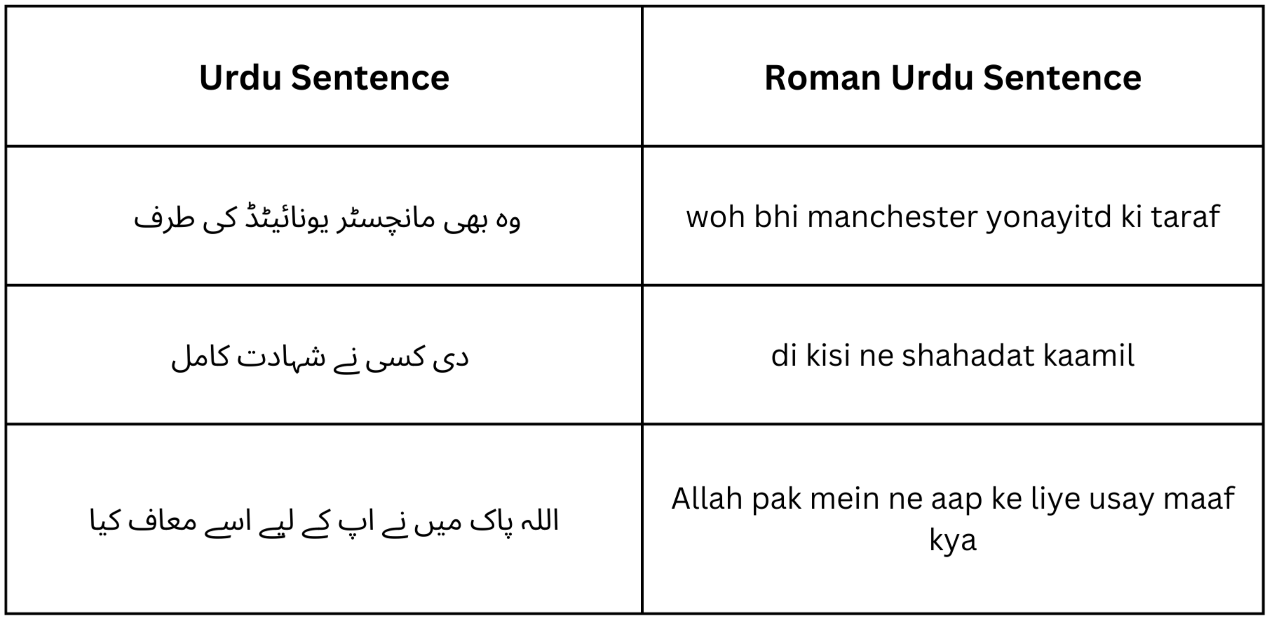
- Conversion and Normalization of Unicode character representation to ASCII - this was done to ensure any non-standard characters are removed as they are likely to be anamolous.
- Add whitespace between all punctuation marks - this was done to ensure that there are no glitches in the subsequent tokenization
- Removal of any non-standard punctuation marks that are not commonly found in the Nastaliq and Latin / Roman script - this was done to ensure that any weird mapping is not learnt by the model
- Removal of any trailing and leading whitespace - this was done to clean the data
- Addition of start and end tokens for each sentence - this was done to mark the sentences with clear start and end signs for the model to learn

Once the preprocessing was done, we tokenized the data to get the individual words for Urdu and Roman Urdu. The tokenized sentences were then converted to integer sequences (each word was mapped onto an integer-key representation) so that we could use them for our transliterator model. Figure 3 shows an example Urdu sentence, its corresponding integer sequence, its corresponding transliterated integer sequence that our model will output, and the associated transliteration of the sentence in Roman Urdu. As done in the original paper by Alam and Hussain, we did not use any pre-trained word embedding or features for our model; rather we relied on the sequence-to-sequence model itself to learn the mappings of each integer (or word). This concept of representation learning is commonly used in Deep Learning, as the neural network model can itself learn very complicated, albeit unexplainable, representations of the dataset that result in very accurate predictions for sequence-to-sequence modeling tasks [4].

2.3 Bucketing

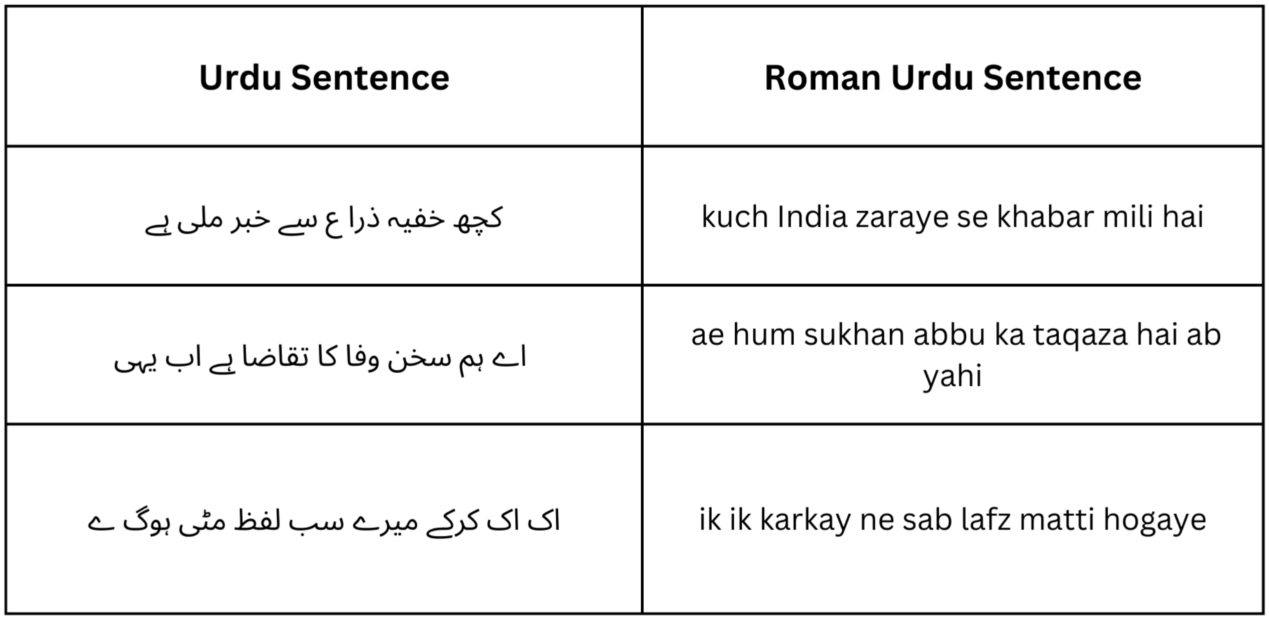
Due to the extreme variability in the sentence lengths (ranging from length < 10 to a max length of 252), padding the sentences would have saturated the shorter sentences (a sentence of length 5 padded to match a sentence of length 252 means this integer sequence is 98% padding!). Another possibility would be to truncate the longer sentences to a fixed length, but that would result in loss of data and meaning (or context) for the longer sentences. After considering these possibilities, we took inspiration from the work by Alam and Hussain, and settled on creating separate buckets of fixed lengths. We created five buckets of the following sentence lengths: 0-10, 10-20, 20-30, 30-40, and > 40. For example, if a sentence is of length 14, it is padded to a length of 20; in the same way, a sentence of length 3 is padded to a length of 10. Our use of sentence length buckets successfully dealt with the issue of varying sentence lengths by avoiding padding saturation for shorter sentences and preserving the integrity of longer ones without data loss or truncation. We then trained and tested on each of the different models separately. The number of examples in the train and test sets for each of these buckets is given in Table 1. Unfortunately, due to computational constraints, we were forced to discard the buckets with sentence length > 30; hence, the remaining report presents our training and analysis on buckets of sentence lengths 0-10, 10-20, and 20-30 only.

[1] https://bit.ly/2MB5QXR
[2] https://www.ijunoon.com/
[3] https://towardsdatascience.com/data-preprocessing-for-machine-translation-fcbedef0e26a
[4] https://www.tensorflow.org/addons/tutorials/networks_seq2seq_nmt



Urdu Sentence	Roman Urdu Sentence
وہ بھی مانچسٹر یونائیٹڈ کی طرف	woh bhi manchester yonayitd ki taraf
دی کسی نے شہادت کامل	di kisi ne shahadat kaamil
اللہ پاک میں نے اپ کے لیے اسے معاف کیا	Allah pak mein ne aap ke liye usay maaf kya

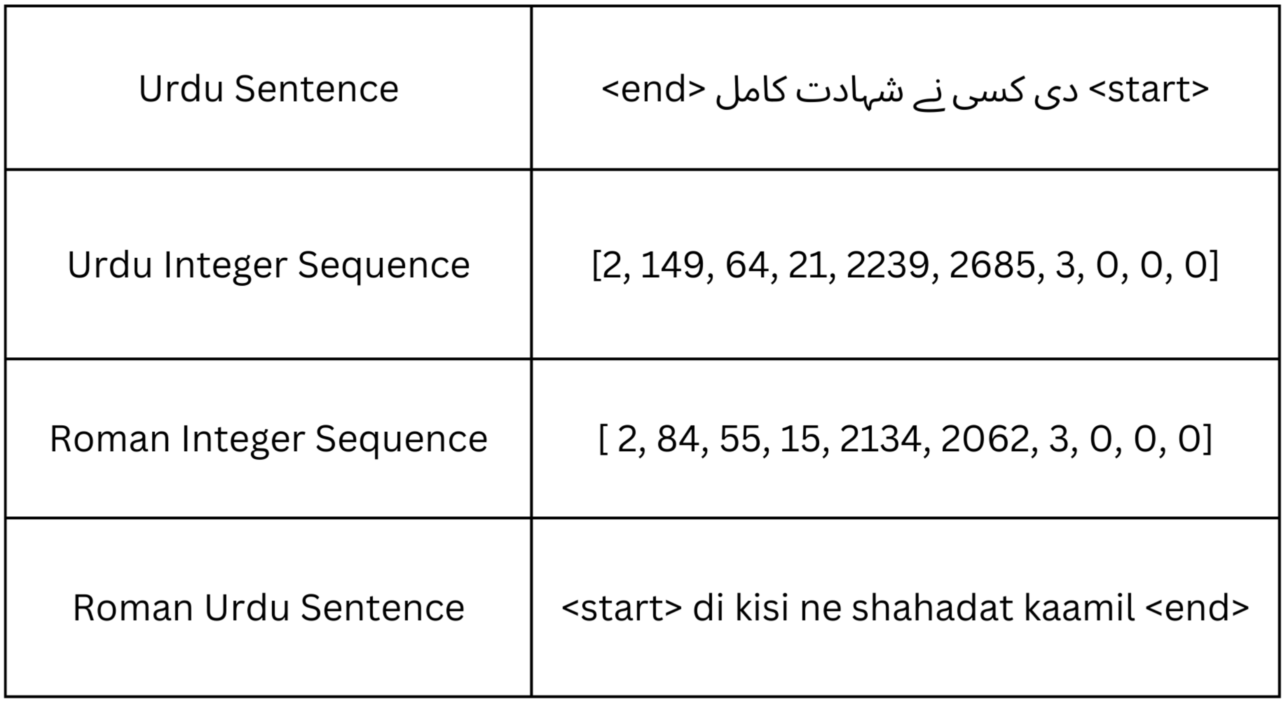
Figure 1: Three example Urdu sentences and their Roman Urdu transliterations.



Urdu Sentence	Roman Urdu Sentence
کچھ خفیہ ذرا ع سے خبر ملی ہے	kuch India zaraye se khabar mili hai
اے ہم سخن وفا کا تقاضا ہے اب یہی	ae hum sukhan abbu ka taqaza hai ab yahi
اک اک کرکے میرے سب لفظ مٹی ہوگ ے	ik ik karkay ne sab lafz matti hogaye

Figure 2: Three example Urdu sentences and their anomalous Roman Urdu transliterations.

/ 2/



Urdu Sentence	<end> کامل شہادت نے کسی دی <start>
Urdu Integer Sequence	[2, 149, 64, 21, 2239, 2685, 3, 0, 0, 0]
Roman Integer Sequence	[2, 84, 55, 15, 2134, 2062, 3, 0, 0, 0]
Roman Urdu Sentence	<start> di kisi ne shahadat kaamil <end>

2

Figure 3: An example mapping of an Urdu sentence to its integer sequence, to its transliterated integer sequence to its Roman Urdu sentence.

Table 1: Number of examples in each bucket.

Bucket (Sentence Length)	Number of examples in Train Set	Number of examples in Test Set
0-10	320,766	35,629
10-20	216,725	24,106
20-30	39,607	4408
30-40	12,787	1389
> 40	7978	898

3 MODEL ARCHITECTURE

3.1 Overview

The general architecture of our encoder-decoder model consists of two separate neural networks, based on recurrent neural networks, that are trained together to encode the input sequence to a context vector, and then decode this context vector to a target sequence. This design outputs the conditional probability of a Roman Urdu word given the previous Roman Urdu word at each time step. Hence, the conditional probability of a target sentence r, given an input sentence u is:

$$p(r|u) = \prod_{j=1}^{m} p(r_j|r_{<j}, c) \tag{1}$$

where c is the context vector, that is generated by the encoder.

The model we decided to work with was inspired by the best performing model used by Alam and Hussain [2]. They implemented a three layered LSTM bidirectional encoder-decoder model with the attention mechanism. As explained previously, due to the fact that we implemented bucketing, our data was categorized into 5 different sequence lengths. We could not train one model on all 5 buckets due to the fact that an RNN model takes a fixed shape input. Hence, ideally, we wanted to train and test 5 different models, one for each bucket, but due to computational restraints we had to settle on using the first three buckets (sentence lengths: 0-10, 10-20, and 20-30) which accounted for over 70% of the total data in the buckets. Hence, a separate model was created for each of the chosen buckets. The basic architecture of each model was the same. It consisted of two main components: an encoder and a decoder, with the outputs from the encoder being passed via a layer of attention. Figure 4 illustrates this model to give a high-level overview.

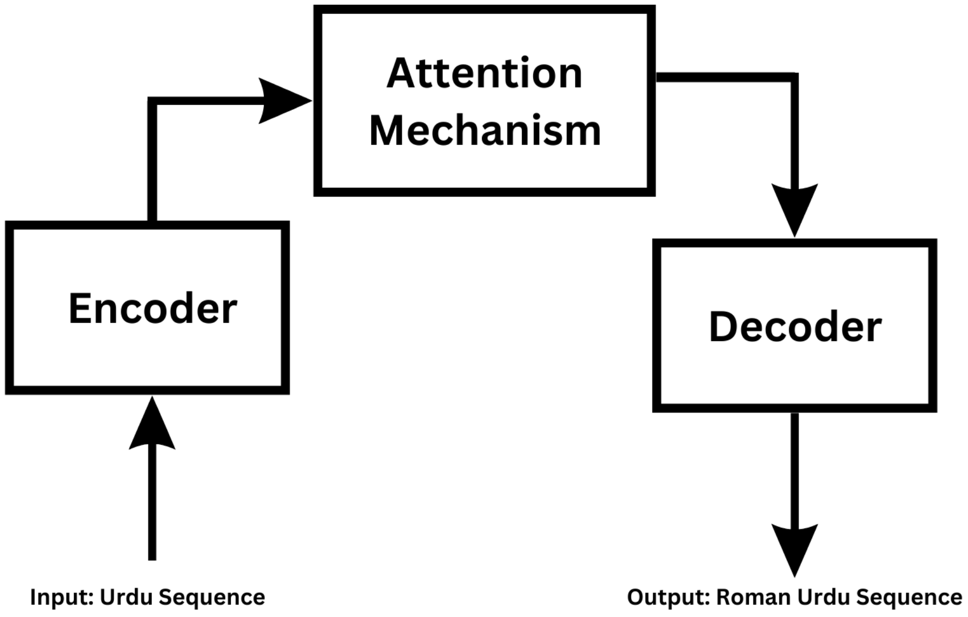


Figure 4: The overall generic architecture for our model.

3.2 Encoder

The encoder takes in the input sequence and transforms it into dense vectors using an embedding layer. The architecture of the encoder is built of N stacked bidirectional LSTM layers, where N = 1 for Bucket_{0-10} and Bucket_{20-30} and N = 2 for Bucket_{10-20}. Each LSTM layer processes the input sequence in both the forward and backward directions, capturing information from both the past and future contexts of each word. This bidirectional nature enhances the encoder's understanding of the input sequence.

During the encoding process, the input sequence is passed through each LSTM layer. The initial hidden states for each layer are provided, which help retain the previous state information. The bidirectional LSTM layer takes the input sequence and initial states, and produces forward and backward hidden states, representing the encoded information for that layer. To form a comprehensive representation of the input sequence, the forward and backward hidden states are concatenated, resulting in new hidden states for each layer. This concatenation process ensures that the encoder captures information from both directions.

The output of the encoding process consists of the encoded sequence, which contains contextual information about each word, and the final hidden states. The encoded sequence represents a condensed representation of the input sequence, capturing its important features. The hidden states encapsulate the learned representations from each LSTM layer and serve as a summary of the input sequence's context.

These features enable the encoder to understand the input sequences thoroughly, using information from both the past and future to understand the context, which is essential for accurate Neural Machine Translation. Figure 5 illustrates this encoder model for clarity.

3.3 Decoder

The decoder consists of several key components, including LSTM layers, an attention mechanism, and an output layer.

During the decoding process, the decoder takes as input an embedded token and the previous hidden states. It utilizes stacked LSTM layers to capture sequential dependencies and generate a sequence of hidden states. These LSTM layers operate in a unidirectional manner, unlike the bidirectional LSTM layers in the encoder.

To incorporate information from the encoder, an attention mechanism is employed. The attention mechanism calculates a context vector by considering a query vector derived from the current hidden state and the encoder's output. This context vector represents the relevant information in the encoder output for the current decoding step. The embedded token and the context vector are concatenated and used as input for the decoder's LSTM layers.

Attention was first introduced by Bahdanau et al. [3] and has been widely used due to its role in drastically improving the output of sequence-to-sequence modeling: for example, it has shown great results in translation tasks where long-range dependencies (for

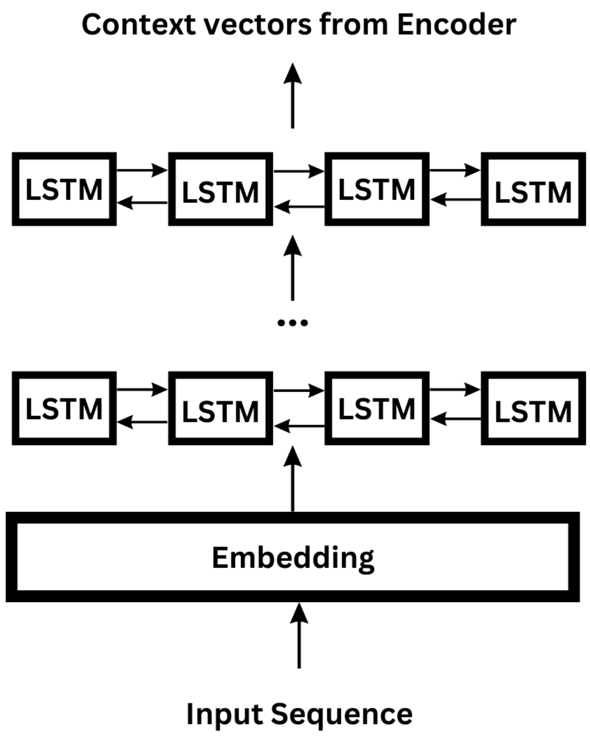


Figure 5: The architecture for the encoder.

grammar, sentence structure, and coherence, etc.) between sentences needs to be preserved. Intuitively, attention at time step t basically means how "attention" should the decoder give to the activation output of the encoder for each t' activations of the encoder. More formally, the attention mechanism is described as follows. The context vector output of the encoder is defined as the weighted sum (such as, softmax) of the attention weights times the output activations of the encoder. The attention weights are defined in a manner similar to probabilities, such that the sum of all the attention weights equals to 1.

$$\sum_{t'=1}^{T_x} \alpha^{<t,t'>}, \tag{2}$$

where T_x is the length of the input sequence, and $\alpha^{<t,t'>}$ represents the attention weight between the t'-th decoder time step and the t-th encoder time step. Each context vector c is then defined as:

$$c^t = \sum_{t'=1}^{T_x} a^{<t'>} \alpha^{<t,t'>}, \tag{3}$$

The actual attention weights, $a^{<}t, t' >$ is computed as:

$$a^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} e^{<t,t'>}} \tag{4}$$

where $e^{<t,t'>}$ is learned by training a small neural network. This attention mechanism is illustrated in Figure 6 for clarity.

The LSTM layers update the hidden states based on the input and previous hidden states, capturing the sequential information necessary for generating accurate output sequences.

After the final LSTM layer, the output sequence is reshaped and passed through a fully connected (dense) layer. This layer maps the hidden states to the vocabulary size, producing a probability distribution over the output vocabulary. The output token is selected based on this distribution, determining the next token in the generated output sequence.

By utilizing sequential dependencies, context from the encoder, and a probability distribution over the output vocabulary, the decoder generates meaningful and accurate output sequences that correspond to the given input. The architecture for the decoder is illustrated in Figure 7.

4 TRAINING

4.1 Overview

We trained 3 different models, one for each chosen bucket, but the general training process was the same for each model. We first passed the input sequence through the encoder, which produced an encoded representation of the input and an internal hidden state. This encoded representation captured the essential features of the input sequence, while the hidden state preserved the encoder's internal memory.

Next, we initialized the decoder with the hidden state from the encoder and started generating the target sequence. At each time

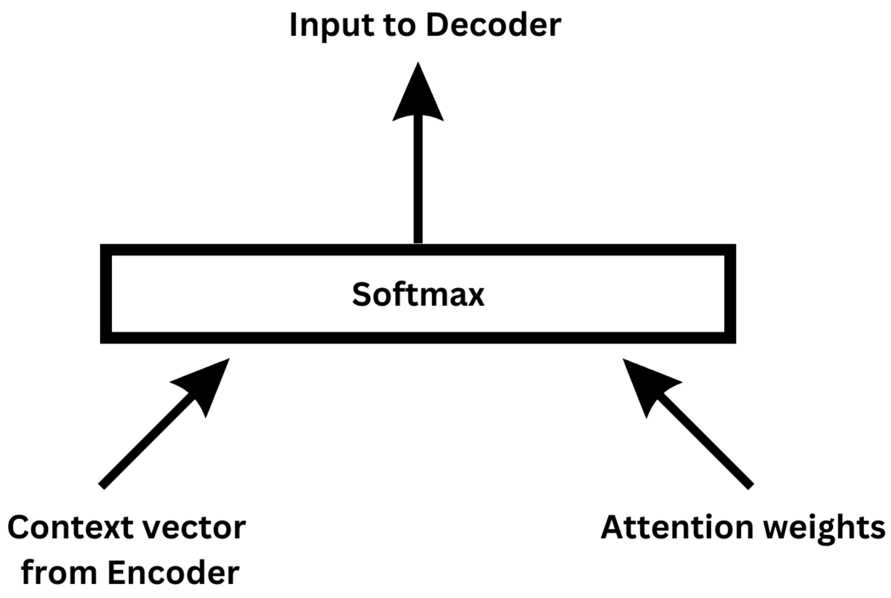


Figure 6: The attention mechanism.

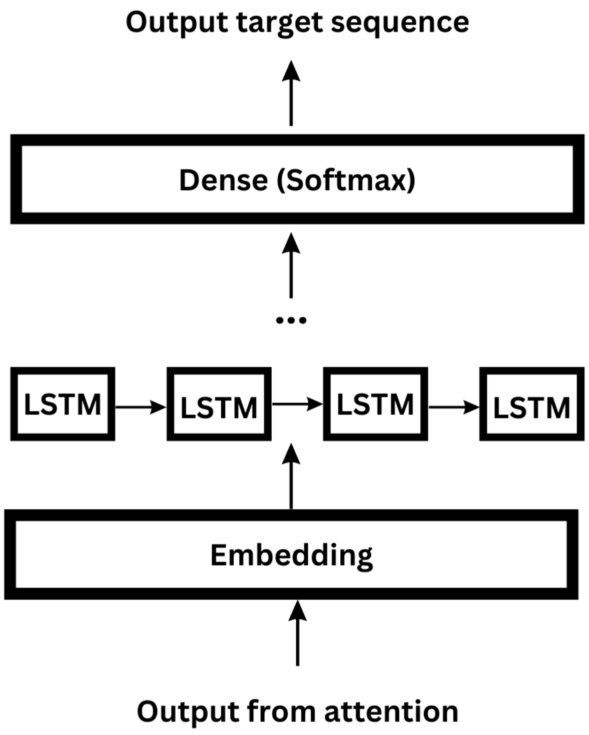


Figure 7: The architecture for the decoder.

step, we took the previous target token and the decoder's own hidden state as input. Using these inputs, we predicted the probability distribution over possible target tokens.

4.2 Loss Evaluation

To evaluate the quality of our predictions, we utilized a loss function, specifically the sparse categorical cross-entropy loss. This loss measured the discrepancy between the predicted token probabilities and the true token. We treated each token prediction independently and assigned a higher loss to inaccurate predictions. We also utilized a masking technique to disregard the padded tokens of each sentence. By masking the loss function, we ensured that padded tokens did not contribute to the overall loss calculation.

4.3 Optimizing Weights

We estimated the gradients of the loss with respect to the encoder-decoder model weights in order to determine how best to set the parameters for our model. These gradients helped us determine the size and direction of the changes that would be necessary to reduce the loss. We used the Adam optimizer to apply these gradients and adjust the model's parameters as necessary. Hence, we reach our ultimate objective which was to minimize the loss, which in turn guides the model towards generating target sequences that closely resembled the true target sequences.

4.4 Hyperparameters

We slightly varied the number of layers, the hyperparameters and the number of epochs for each model after considering the time and computational restraints as well as the shape of each bucket. For each model, we chose a batch size of 64, 1024 LSTM units, 17000 steps per epoch, and an embedding size of 256.

The first bucket contained the largest number of sentences (**320,766 sentences**). Hence keeping computational constraints in mind, we set the number of epochs to 10, while reducing the number of LSTM layers to 1. With the reduced complexity of the model, training the first bucket took approximately **1 hour and 45 minutes**.

The second bucket contained the **216,725 sentences**. Due to unsatisfactory loss minimization with the first bucket, we increased the complexity of the model, we set the number of epochs to 50, while increasing the number of LSTM layers to 2. This came with troubling computational overhead, training the second bucket took approximately **5 hours and 45 minutes**. This increased overhead later redeemed itself as you will see in the results (section 6).

The third bucket contained **39,607 sentences**. Keeping the overhead while training the second bucket in mind, we set the number of epochs to 10, while reducing the number of LSTM layers to 1. With the reduced complexity of the model, training the first bucket took approximately **1 hour**.

We trained the models using varying configurations of layers, hyperparameters, and epochs, while carefully considering computational constraints and the characteristics of each bucket. The differences will later be resonated when we evaluate each model.

4.5 Training Loss

Sparse categorical cross entropy was used as the loss function due to the multi-class classification problem. The model produces a probability for each word in the target language (Roman Urdu's) vocabulary. The loss can be modeled as the following function:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{T_y} y_i \log(p_i), \tag{5}$$

Figure 8 shows the training loss of each bucket over the train steps during the training. As seen in the figure, the loss decreases over each time step, hence showing that the model has trained on the train dataset.

5 EVALUATION

To analyze our trained models on the testing data, we chose two search methods for predictions and two performance metrics in hopes of doing a detailed empirical analysis. For each model, we first used beam search to find predictions for each Arabic script Urdu sentence in the test data and then evaluated the predictions using ROGUE-1 and the BLEU-1 performance metrics.

5.1 Predictions

Our goal during prediction is to pick the best and most likely translated word for each token, so we choose the target word with the maximum probability from our vocabulary based on the source sentence.

So how do we optimally pick out each corresponding word? Do we just choose the most likely word at that particular token, or should we consider the entire sentence? To find the optimal method for predictions in our settings, we performed the two major decoding algorithms for each prediction.

5.1.1 Greedy Search. Greedy search is the simplest decoding algorithm, and has comparatively a lot less computational overhead compared to other algorithms. During the decoding process, the model generates the probabilities for the next token in the sequence. With greedy search, the token with the highest conditional probability from the vocabulary of the target language, which in our case is Roman Urdu, is chosen.

5.1.2 Beam Search. Beam search is a decoding technique that selects multiple alternatives for an input sequence at each timestep based on conditional probability. The number of multiple alternatives depends on a parameter called Beam Width. Keeping computational constraints in mind, we used a beam width of 5. In each decoding step, the model generates a set of candidate tokens for each beam and calculates their scores based on the model's predictions. The beams with the highest scores are retained, and the process is repeated for the next decoding step. This allows the model to consider different possibilities and potentially generate more diverse output sequences.

5.2 Performance Metrics

Now we have successfully predicted sentences in Roman Urdu, the next step was to test the quality of our predictions. We chose not to narrow down our tests for quality. We calculated both ROGUE-1 and BLEU-1 scores for each of our models and decoding algorithms. Both of these metrics focus on difference qualities of the predictions.

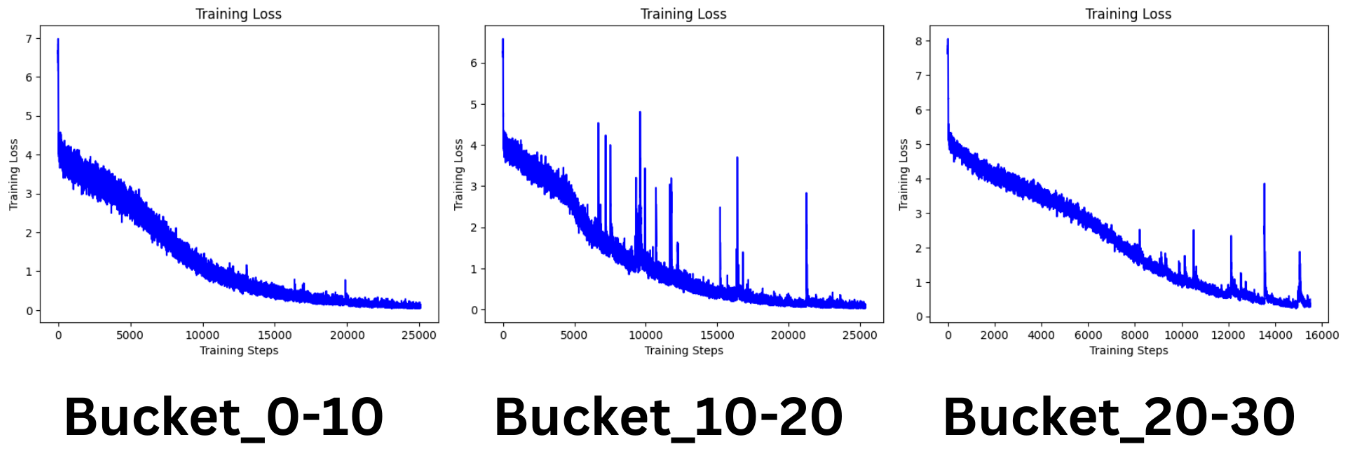


Figure 8: Graph of training loss for each bucket.

5.2.1 BLEU-1. The BLEU (Bilingual Evaluation Understudy) Score is a metric used to evaluate the similarity between a generated sentence and a reference sentence. It focuses on the precision of the prediction by measuring the matching of individual tokens, also known as unigrams. A perfect match between the translation and the reference results in a score of 100. Alternatively, a complete mismatch between the two yields a score of 0. The evaluation process involves comparing the individual tokens in the candidate translation with those in the reference text, focusing solely on the unigrams. Each token is assessed independently, disregarding the order of the words. By examining the agreement of unigrams, the BLEU Score quantifies the level of similarity based on the match of individual tokens, and hence focuses on the precision of a NMT task.

$$\text{BLEU-1 score} = \frac{\sum_{\text{predicted} \in C} \text{count}(\text{prediction})}{\sum_{\text{actual} \in C} \text{count}(\text{actual})}$$

5.2.2 ROGUE-1. ROUGE stands for "Recall-Oriented Understudy for Gisting Evaluation." ROUGE-1 focuses on evaluating the similarity of unigram sequences between the generated summary or translation and the reference summary or translation. ROUGE-1 score calculates the overlap of unigrams between the generated and reference texts. It quantifies the recall of unigrams, indicating how well the generated output captures important words from the reference text. The score is the ratio of overlapping unigrams to the total count of unigrams in the reference text, measuring the alignment of content on a word level.

The ROUGE-1 score is calculated as:

$$\text{ROUGE-1} = \frac{\text{Number of overlapping unigrams}}{\text{Total number of unigrams in reference text}}$$

6 RESULTS

We trained 3 separate models for buckets of 3 different sizes (=<10, 10<len=<20, 20<len=<30). Predictions were calculated using two different decoding algorithms for each model (Beam Search and Greedy Search). For each type of prediction for each model, 2 different performance metrics were calculated (BLEU-1 and ROGUE-1). Table 2 show the performance metrics of each model.

We got very distinguishable and unexpected results for each type of decoding algorithm. The first thing to consider before we delve deeper into our analysis is that all scores are relatively better for the second bucket compared to the other two. We trained the second bucket's model much more extensively, and it's effects are evident here. The second bucket scored 74 in BLEU and 0.90 in ROGUE with the greedy search algorithm. This is distinguishably much higher than the other buckets implemented with greedy search.

6.1 Beam Search vs Greedy Search

We were greeted with very unexpected scores with the beam search. If there was something wrong with the training of the model, we would not get such high scores with the greedy search. There could be several reasons behind an unusally low score for beam search:

Table 2: Performance of each model on BLEU-1 and ROUGE-1 using both Greedy and Beam Search.

Buckets	**BLEU-1** (out of 100.0)		**ROUGE-1** (out of 1.0)	
	Beam Search	**Greedy Search**	**Beam Search**	**Greedy Search**
Bucket$_{0-10}$	8.79e-77	54.66	0.24	0.63
Bucket$_{10-20}$	1.20e-76	74.0	0.31	0.90
Bucket$_{20-30}$	8.97e-77	49.7	0.22	0.71

(1) Beam search explores multiple potential translations by maintaining a beam of top sequences. Beam search's exploration allows for consideration of more possibilities, but it can also lead to suboptimal translations, especially when the beam width is small.
(2) A small beam width limits the exploration capability, potentially missing out on better translations. We could not use a higher beam width than 5 due to computational restraints, which may be one of the reasons for such a low score
(3) Beam search tends to generate repetitive sequences due to its tendency to favor high-probability tokens. When we looked at some of the predictions made by the beam search, there is constant repetition of common words like "ki", "acha ", and "haan" were constantly repeated in a single prediction. This repetition can negatively impact the scores since the reference translations usually do not contain such repetitions.
(4) Beam search is vulnerable to search errors, especially when the beam width is not large enough. It tends to get stuck in suboptimal paths which leads to poor translation quality.

7 CONCLUSION

In this project, we have developed a sequence-to-sequence encoder-decoder model for neural machine transliteration from Urdu to Roman Urdu text. We presented our data methodology for data preprocessing, the logic, intuition, and math behind our model's architecture, the steps involved in training our model fine-tuning its hyperparameters, and lastly, we have presented our quantitative analysis of the models on the test buckets.

As with all NLP projects, our project has some limitations. First and foremost, due to limited computational power at our end (despite paying for extra compute units by Google Colab), we were unable to copy the model architecture from the works of Alam and Hussain. This issue also restricted us to train and test the data with sentence lengths less than 30. Additionally, due to the large time taken for training the dataset, despite us only using 60% of the original data provided by Alam and Hussain, we were unable to effectively test a large set of hyperparameters. Future work would cater to this by not having to worry about computational and time restraints. We were also unable to produce a general model as the final output of this project that could transliterate Urdu sentences of arbitrary size (the current implementation would require a user with a novel test sentence to manually run the appropriate model that matches the corresponding bucket's sentence length). Further development is required to concatenate or combine the models to generate a final holistic model. Furthermore, it is likely that our code for Beam Search is incorrect, given the stark difference in the BLEU-1 and ROUGE-1 scores between Beam Search and Greedy Search. This issue can be improved by carefully analyzing the current implementation, or by using built-in library's implementation, such as that by TensorFlow. Despite these limitations, our model performed well on sentence lengths <= 20, while achieving maximum performance on sentences with lengths 10-20 (BLEU-1 score = 74.3, ROUGE-1 score of 0.9).

In addition to the above-mentioned improvements to our methodology, future work could also first improve the original dataset by performing automated (or manual) ways to correct the inconsistencies within the dataset, as explained in Section 2.1. Further research could also look at generating a larger dataset that a NLP model could learn more vividly. In terms of using more complex model architectures, one can also move towards using transformer-based networks to solve the task of machine transliteration. In today's age, LLMs (large language models), such as GPT-3 by OpenAI can also be fine-tuned and specialized for the task of machine transliteration. Lastly, Urdu is but one of the many low-resource languages in the Pakistan: research and development is needed in the domain of other low-resource languages such as Balochi, Pashto, Sindhi, and Seraiki.

REFERENCES

[1] 2023. GitHub search query. https://github.com/search?q=nmt&type=repositories. Accessed: 11 May 2023.
[2] Mehreen Alam and Sibt ul Hussain. 2021. Deep learning-based Roman-Urdu to Urdu transliteration. *International Journal of Pattern Recognition and Artificial Intelligence* 35, 04 (2021), 2152001.
[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[4] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.
[5] Joseph Henrich, Steven J Heine, and Ara Norenzayan. 2010. The weirdest people in the world? *Behavioral and brain sciences* 33, 2-3 (2010), 61–83.
[6] Raees Ul Islam Tayyaba Fatima and Muhammad Waqas Anwar. 2018. Morphological and Orthographic Challenges in Urdu Language Processing: A Review. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)* (Miyazaki, Japan, 7-12), Kiyoaki Shirai (Ed.). European Language Resources Association (ELRA), Paris, France.
[7] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).