CSE 256

Final Project: Exploration of Decoder Language Model for Hinglish Token generation

Jishnu Raychaudhuri jraychaudhuri@ucsd.edu

Fall 2024

1 Introduction

Growing up in India, I was exposed to the phenomenon of "Hinglish", which is the mixing of the Hindi and English words in a sentence. A common feature of written Hinglish is transliteration where most, if not all, Hindi words are transliterated into the latin alphabet. For example, the sentence "Aapka naam kya hai?" is a transliteration of "आपका नाम क्या है?" which translates to "What is your name?"

My project aims to build a decoder transformer model to generate Hinglish text from a given context. The things I set out to do are:

- Create a dataset of Hinglish text for training and testing: DONE.
- Build and train a decoder language model with an untrained simple tokeniser: DONE.
- Compare with the decoder langauge model with other pretrained tokenisers: DONE.

2 Related work

A lot of the NLP work on Hinglish seems to be focused on translation from English to Hinglish text and synthetic generation of Hinglish text, which also employs machine translation from English to Hinglish. Hinglish is also a low resource language, which means that there aren't large Hinglish corpora readily available.

Researchers Vivek Srivastav and Mayank Singh have done a lot of work in the synthetic generation of quality Hinglish text. Their paper "HinglishEval Generation Challenge on Quality Estimation of Synthetic Code-Mixed Text: Overview and Results"[7] focuses on methods to evaluate the quality of synthetically generated Hinglish text. Their efforts culminated in the HinGE dataset[6] which consists of human and synthetically generated Hinglish text for Hinglish generation and evaluation.

The paper "MuRIL: Multilingual Representations for Indian Languages"[4] presents a mulitlingual language model trained on code switched Indian datasets, i.e. transliterated text from various Indian languages as well as code-mixed with English. The model is mainly a sequence classifier, used for classifying texts into various monolingual Indian or code-mixed languages.

The paper "Adapting Multilingual LLMs to Low-Resource Languages using Continued Pre-training and Synthetic Corpus"[3] focuses on adapting synthetically generated data for training LLMs in low resource languages. The paper focuses on Indian languages like Hindi and other code-mixed variations of it.

"Towards Code-Mixed Hinglish Dialogue Generation" proposes a model trained on synthertically generated Hinglish data to produce coherent Hinglish text [2]. This papers aligns the most with my project.

3 Data collection

I collected my Hinglish data from two datasets, Google Research's "Hinglish-TOP-Dataset"[5] and a dataset from Hugging Face called "english-to-hinglish"[1]. Both datasets contain an English sentence and it's Hinglish

translation. The "Hinglish-TOP-Dataset" is split into human annotated and and synthetically generated Hinglish. I used the human annotated dataset for my project.

The human annotated dataset contains an English query and it's translated Hinglish equivalent and is split into three files:

- `train.tsv` with 2993 entries
- `test.csv` with 6513 entries
- `validation.tsv` with 1390 entries

where each entry has columns `en_query`, `cs_query`, `en_parse`, `cs_parse`, and `domain`. The first five entries in `train.tsv` look like

	en_query	cs_query	en_parse	cs_parse	domain
0	Add a new weekly reminder for Sunday Brunch at...	9 : 30 am ko Sunday Brunch ke liye ek naya wee...	[IN:CREATE_ALARM Add a new [SL:PERIOD weekly]...	[IN:CREATE_ALARM [SL:DATE_TIME 9 : 30 am ko] ...	alarm
1	message danny and see if he wants to go to com...	danny ko message karo aur dekho ke he wants to...	[IN:SEND_MESSAGE message [SL:RECIPIENT danny]...	[IN:SEND_MESSAGE [SL:RECIPIENT danny] ko mess...	messaging
2	set alarm for 2 hours	do ghante ke liye alarm set kardo	[IN:CREATE_ALARM set alarm [SL:DATE_TIME for 2...	[IN:CREATE_ALARM [SL:DATE_TIME do ghante ke li...	alarm
3	kill the reminder for baking a cake for neil	neil ke liye cake bake karne ke reminder ko mi...	[IN:DELETE_REMINDER kill the reminder for [SL:...	[IN:DELETE_REMINDER [SL:TODO neil ke liye cake...	reminder
4	retrieve my chat requests please	Please mere chat requests ko retrieve kare	[IN:GET_MESSAGE retrieve my chat requests plea...	[IN:GET_MESSAGE Please mere chat requests ko r...	messaging

Figure 1: First five entries in `train.tsv`

where `en_query` is the English phrase, `cs_query` is the Hinglish translation, `en_parse` is the parsed version of the english phrase, `cs_parse` is the aprsed version of the Hinglish phrase, and `domain` is the area the content of the phrase is related to.

The Hugging face dataset also contains an English phrase with it's Hinglish equivalent stored in JSON format with three keys, `en`, `hi_ng`, and `source`. This dataset contains 189102 entries. The first five entries in this dataset look like

```
{ "en": "What's the name of the movie", "hi_ng": "film ka kya naam hai", "source": 1 }
{ "en": "Hi, the rotten tomatoes score is great but the meta critic score seems a little low a movie of this quality. ", "hi_ng": "namaste, sada hua tomatoes score mahaan hai, lekin meta critic score is gunavatta kee philm se thoda kam lagata hai.", "source": 1 }
{ "en": "Do you think you will like the movie", "hi_ng": "kya aapako lagata hai ki aapako film pasand aaegee", "source": 1 }
{ "en": "What kind of movie is it", "hi_ng": "yah kis tarah kee philm hai", "source": 1 }
{ "en": "when was the movie made?", "hi_ng": "film kab banee thee?", "source": 1 }
```

Figure 2: First five entries in the Hugging face dataset

where `en` is the English phrase, `hi_ng` is the Hinglish translation, and `source` is either 1 for human annotated or 0 for synthetically generated.

3.1 Data pre-processing

Both datasets have a Hinglish phrase as well as other information about the phrase. However, I am only interested in the Hinglish sentences to build my database to train and evaluate my model. I first combined the `train.tsv`, `test.tsv`, and `validation.tsv` datasets from the "Hinglish-TOP-Dataset" into one dataset with 9933 entries. I then extracted all the values under the `cs_query` columns to get the Hinglish phrases.

I extracted all the values associated with the key `hi_ng` from each row in the Hugging face dataset to extract all the Hinglish phrases. I then combined the Hinglish phrases from the Hugging face dataset with the ones extracted from the "Hinglish-TOP-Dataset". I removed all duplicate entries and sanitised them by removing newlines and tab characters.

The final result was a dataset with 162031 entries with only Hinglish phrases. The first five entries of my dataset looks like

```
0    9 : 30 am ko Sunday Brunch ke liye ek naya wee...
1    danny ko message karo aur dekho ke he wants to...
2                    do ghante ke liye alarm set kardo
3    neil ke liye cake bake karne ke reminder ko mi...
4           Please mere chat requests ko retrieve kare
```

Figure 3: First five entries in my dataset

where every line contains a separate Hinglish phrase.

This dataset was randomly split into a training set and testing set with a 70-30 split. All models were trained and tested using these training and testing sets.

4 My approach

4.1 The decoder language model

I implemented a transformer decoder language model in PyTorch from scratch with a masked self-attention head mechanism. The model is trained to minimise the cross-entropy loss between predictions and the true next word in the sequence. I used the AdamW optimiser with a learning rate of 0.001.

The model consists of an embedding layer with positional embeddings, a layer of masked attention heads, a layer norm, and a linear output layer. Each masked attention head layer consists a masked attention head, a linear layer with a ReLU activation function, and two layer norm layers. The model has 6,767,041 trainable parameters. The hyperparameters used are as follows:

1. `batch_size = 128`. Number of entries being trained / evaluated at the same time.
2. `block_size = 32`. Read the data in chunks of 32 tokens.
3. `n_embd = 64`. The embedding dimension.
4. `n_head = 2`. The number of attention heads.
5. `n_layer = 4`. The number of transformer layers.
6. `n_input = 64`. The maximum number of tokens the model can take as input.

A batch size of 128 was used to speed up training and evaluation and the other values for the hyperparameters were used to keep the model simple and reduce it's complexity.

4.2 Tokenisers

I implemented a simple whole word tokeniser and trained it on the entire dataset. My implementaion is heavily inspired from the simple whole word tokeniser provided as starter code in PA2. I then compared the performance of my model using the simple tokeniser and two pre-trained tokenisers, the tokeniser from the MuRIL[4] model and a pre-trained tokeniser from Hugging face[8] trained on Hinglish datasets.

I chose the MuRIL tokeniser as it has been trained not only on Hinglish but also on other transliterated versions of Indian languages, so it captures the structures and patterns of all Indian languages. I chose the Hugging face tokeniser to see if a tokeniser trained on a reduced set of data, particularly Hinglish, would provide an improvement in performance, especially for my task of generating Hinglish tokens.

Nothing was changed about the decoder language model when running with different tokenisers.

4.3 Compute

I trained and tested my model in Google Colab with a A100 GPU. I trained my model for about 2000 iterations as performance for each of the tokenisers seemed to bottom out at roughly 2000 iterations. Each training and testing loop took around 5 minutes to complete.

4.4 Results

4.4.1 Simple tokeniser

The decoder language model with the simple tokeniser performed decently well with a starting testing perpelexity of 62916.12109375 and bottoming out at around 84.78379821777344 testing perplexity. The results are given in a table and a plot below:

Iteration	Train	Test
0	62931.961	62916.121
100	743.178	747.214
200	456.923	474.400
300	263.534	275.900
400	191.408	209.027
500	156.853	171.615
600	132.455	150.933
700	114.457	137.198
800	103.470	123.402
900	94.243	115.911
1000	85.903	110.296
1100	80.801	103.117
1200	76.059	101.167
1300	70.565	97.938
1400	66.996	93.792
1500	65.081	91.520
1600	61.754	91.008
1700	59.715	88.333
1800	56.806	87.400
1900	54.154	87.484
1999	52.999	84.784

Table 1: Train and test perplexities of the simple tokeniser over 2000 iterations

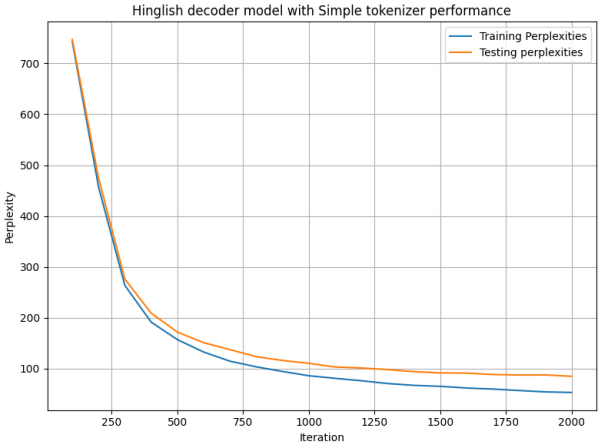

Figure 4: Decoder model with simple tokeniser performance over 2000 iterations

The perplexity for the first iteration isn't included the plot above to better showcase the perplexities for higher iterations.

4.4.2 MuRIL tokeniser

The decoder language model with the pretrained MuRIL tokeniser performed much better than the simple tokeniser. Although it's initial testing perplexity was 235794.3125, which is much higher than the initial perplexity of the model with the simple tokeniser, it quickly bottomed out at 58.56596374511719. The results are given in a table and a plot below:

Iteration	Train	Test
0	235833.891	235794.312
100	815.767	818.770
200	471.811	482.176
300	231.719	236.222
400	159.064	164.684
500	125.382	132.748
600	106.535	114.082
700	93.904	101.919
800	82.986	90.909
900	76.259	84.717
1000	70.679	81.757
1100	65.333	76.545
1200	61.743	70.648
1300	58.149	69.369
1400	56.560	65.773
1500	53.308	64.851
1600	51.563	62.690
1700	50.081	61.094
1800	48.446	59.612
1900	47.144	59.060
1999	44.733	58.566

Table 2: Train and test perplexities of the MuRIL tokeniser over 2000 iterations

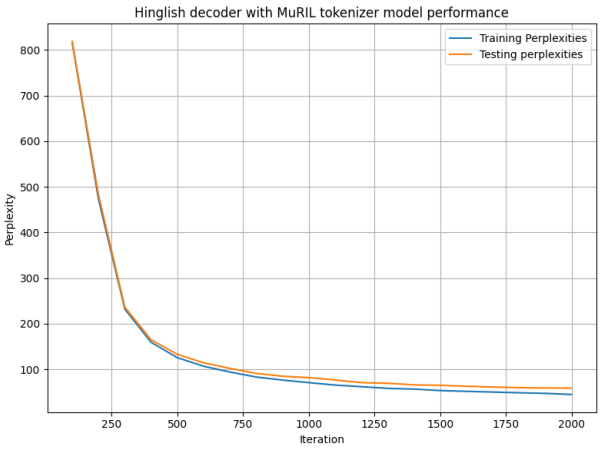

Figure 5: Decoder model with MuRIL tokeniser performance over 2000 iterations

The perplexity for the first iteration isn't included the plot above to better showcase the perplexities for higher iterations. We can see that the MuRIL tokeniser model converges at a slightly faster rate than the simple tokeniser model.

4.4.3 Hinglish tokeniser

The decoder model with the pretrained Hinglish tokeniser performed the best out of the three. It started with an initial testing perplexity of 34309.1953125, the lowest out of the three initial testing perplexities, and bottomed out at 23.862625122070312 which is far lower than the best perplexity achieved by the previous tokenisers. The results are given in a table and a plot below:

Iteration	Train	Test
0	34345.992	34309.195
100	318.471	319.453
200	123.970	125.041
300	72.768	72.983
400	54.788	55.412
500	44.910	46.917
600	39.745	40.707
700	35.486	36.914
800	33.030	34.276
900	30.392	32.402
1000	28.982	30.915
1100	27.615	29.397
1200	26.157	28.206
1300	25.290	27.395
1400	24.326	26.778
1500	23.318	25.964
1600	23.026	25.044
1700	22.469	24.776
1800	21.843	24.641
1900	21.573	24.007
1999	20.900	23.863

Table 3: Train and test perplexities of the Hinglish tokeniser over 2000 iterations

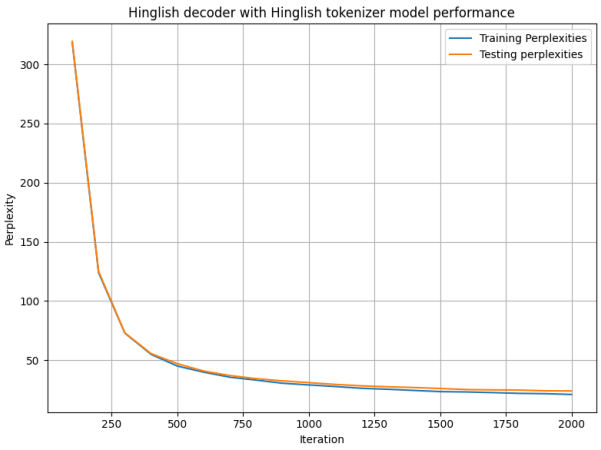

Figure 6: Decoder model with Hinglish tokeniser performance over 2000 iterations

The perplexity for the first iteration isn't included the plot above to better showcase the perplexities for higher iterations. The decoder model with the Hinglish tokeniser converges the fastest out of the three tokenisers.

4.4.4 Overall

Comparing the model using three different tokenisers, we can see that the choice of tokeniser makes a significant difference in model performance. It's perhaps not surprising that the tokeniser trained specifically on Hinglish datasets performs the best at generating Hinglish tokens. The MuRIL tokeniser performs better than the simple tokeniser but doesn't adequately capture the specific structures and patterns present in Hinglish. This is due to the fact that its trained on other transliterated Indian languages that, very often, have completely different structures and patterns when compared to Hindi.

The models are better compared in the plot below:

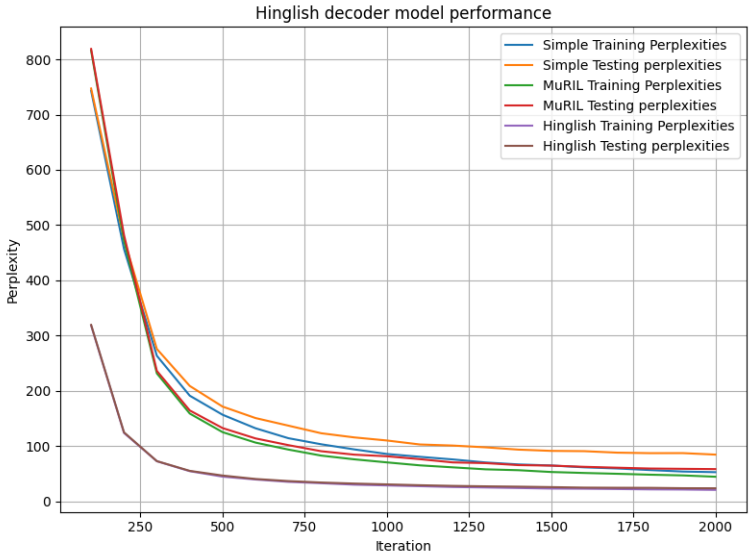

Figure 7: Decoder language model performance with different tokenisers over 2000 iterations

The simple tokeniser and the MuRIL tokeniser converge at a similar rate, but the MuRIL tokeniser wins out in terms of dinal perplixity values. The Hinglish tokeniser, however, converges much quicker and to a lower perplexity value.

I took the decoder model with the Hinglish tokeniser and looked at some of the outputs produced. When asked to generate 100 tokens with no context, the module produced,

> i expectpanth se pahuchne mei kitni der lagegi [UNK] kya hurricane ka kam koi tarah block wali hai [UNK] to [UNK] aur aage kaam madation hai [UNK] politicaar playrykkrin virodhing sankalwa gayakos hairowkaring log gu gayeused [UNK]ing cover tak kam kon par nahi singth ka istemaal karke [UNK] movieotan ke liye [UNK] salman dwar [UNK] [UNK] ke liye [UNK] n firstit me jane par don.

The output produces actual Hinglish words at the start, but the quality of the tokens produced degrades as the sentence gets longer. The entire sentence is still gibberish.

When given an input context of “ghar jaane ke baad”, which translates to “after going home” the model produces,

> ghar jaane ke baad [SEP] karo [UNK] pm ko school ke liye mere paas kitne reminders hai [UNK] mujhe mere plan ke milne ke liye ekoo aaj friedning kiadaainberachus in st clocke par time hai towno karaye mujhe ke liyety recomlist me raat ke liyeellers kabfic inca kiya baslale ke baad don kabo the karenge miney traffic ke liye destination onie ke liye futuro len kar.

When given an input context, the model output is slightly better. The tokens produced at the start are actual Hinglish tokens and the sentence is somewhat coherent. However, as the generated sequence get’s longer, the quality of the tokens generated degrades and thr sentence becomes gibberish once again. This is possibly due to the low amount of Hinglish text that the model has been trained on. The quality should improve with more data.

5 Conclusion

In conclusion, implementing a decoder language model for generating Hinglish tokens didn’t prove to be too hard. Although most of the sequences generated were gibberish, this can be solved by training the model on much more data, which is hard as Hinglish is a low resource language. I am not surprised that the model with the Hinglish tokeniser performed the best.

In the future, I would like to finetune pretrained models like GPT-4 to generate Hinglish tokens and try to create chatbot that takes Hinglish text as input and outputs coherent Hinglish text. I would also like to explore other methods of evaluation such as human feedback.

References

[1] Nitai Agarwal. *english-to-hinglish*. 2023. URL: https://huggingface.co/datasets/findnitai/english-to-hinglish.

[2] Vibhav Agarwal, Pooja Rao, and Dinesh Jayagopi. "Towards Code-Mixed Hinglish Dialogue Generation". In: Jan. 2021, pp. 271–280. DOI: 10.18653/v1/2021.nlp4convai-1.26.

[3] Raviraj Joshi et al. 2024.

[4] Simran Khanuja et al. *MuRIL: Multilingual Representations for Indian Languages*. 2021. arXiv: 2103.10730 [cs.CL].

[5] Google Research. *Hinglish-TOP-Dataset*. 2022. URL: https://github.com/google-research-datasets/Hinglish-TOP-Dataset.

[6] Vivek Srivastava and Mayank Singh. *HinGE: A Dataset for Generation and Evaluation of Code-Mixed Hinglish Text*. 2021. arXiv: 2107.03760 [cs.CL]. URL: https://arxiv.org/abs/2107.03760.

[7] Vivek Srivastava and Mayank Singh. "HinglishEval Generation Challenge on Quality Estimation of Synthetic Code-Mixed Text: Overview and Results". In: *Proceedings of the 15th International Conference on Natural Language Generation: Generation Challenges*. Ed. by Samira Shaikh, Thiago Ferreira, and Amanda Stent. Waterville, Maine, USA and virtual meeting: Association for Computational Linguistics, July 2022, pp. 19–25. URL: https://aclanthology.org/2022.inlg-genchal.3.

[8] Obaid Tamboli. *BERT Tokenizer for Hinglish*. URL: https://huggingface.co/obaidtambo/hinglish_bert_tokenizer.