Homophonic Pun Generation in Code Mixed Hindi English

Yash Sarrof
Saarland University
ysarrof@lst.uni-saarland.de

Abstract

In this study, we investigate Hinglish—a blend of Hindi and English commonly found in informal online communication—with a particular focus on automated pun generation. Our work [1] examines the applicability and adaptability of existing English pun generation pipelines to Hinglish. We assess the pun generation capabilities of Large Language Models (LLMs), particularly GPT-3.5. By employing Chain of Thought prompting and Self-Refine techniques, we identify cross-linguistic homophone detection as a central difficulty. To address this, we propose a novel algorithm for cross-lingual homophone identification and develop a Latin-to-Devanagari transliteration module to leverage the widespread use of Latin-script Hindi in online settings. Building on existing frameworks for pun generation, we incorporate our homophone and transliteration modules to improve output quality. Crowd-sourced human evaluations validate the effectiveness of our approach.

1 Introduction

Code-mixing has become widespread in written digital communication, evolving from a phenomenon primarily seen in spoken language (Winata et al., 2022). Similarly, the blend of Hindi and English, commonly known as Hinglish, is now prevalent across both spoken and written mediums.

While computational humor generation receives extensive attention in monolingual settings, especially in English, existing methods often struggle in code-mixed contexts. Puns based on homographs, for instance, tend to lose their impact when applied across languages. Homophonic puns, however, hold unique potential in code-mixed settings, as identical sounds can convey entirely different meanings in different languages.

Large Language Models (LLMs), such as those in the GPT series, demonstrate proficiency in generating English puns, largely due to their training on extensive datasets like Common Crawl. However, as noted by Jentzsch and Kersting (2023), while GPT-4 can produce puns, it frequently lacks diversity in its outputs—a challenge further amplified in code-mixed environments.

This work examines the performance of **GPT-3.5** in generating homophonic puns within the Hinglish context. Using Chain of Thought prompting (Wei et al., 2023) and the Self-Refine framework (Madaan et al., 2024)—techniques designed to enhance creative text generation—we find that a key difficulty in pun generation lies in reliably identifying cross-linguistic homophones. To address this, we propose an algorithm that detects homophones across languages.

Native Hindi speakers, who often use English as a second language, prefer the Latin script for online communication due to the complexities of the Devanagari script, which includes conjunct consonants, vowel diacritics, and a large character set. This, combined with the familiarity of the QWERTY keyboard, has led to the widespread adoption of Latin script for Hindi (Wolf-Sonkin et al., 2019). To quantify this shift, we compare Hindi word counts in Devanagari and Latin scripts across datasets like The Pile and C4, finding a substantial preference for Latin script. This discrepancy underscores the need for transliteration to leverage phonetic similarities across scripts, essential for generating homophonic puns in Hinglish. Therefore we introduce a lightweight Latin-to-Devanagari transliteration module to take advantage of this fact.

Building on He et al. (2019)'s hybrid pipeline for pun generation, we adapt it to integrate our homophone generation and transliteration modules. Results show that providing these homophones as input markedly enhances the quality of generated puns. A crowd-sourced evaluation validates these findings, providing robust evidence of the improvements.

[1] Code: https://github.com/yashYRS/HinglishPun

Proceedings of the 1st Workshop on Computational Humor (CHum), pages 23–31
January 19, 2025.

Vanilla Prompting Example

Input: Generate a Hindi English mixed pun.
Output: Why did the Hindi teacher bring a ladder to class?
Because he wanted to teach बंदर [bəndər] (bandar), how to climb the बांस [bãːs] (bamboo).

Figure 1: Vanilla Prompting leads to grammatical but incoherent statements. The IPA provided in all prompts from hereon is only for the paper readability and they are not a part of the prompt.

Criteria for SelfRefine

Pun Present: Does the text have a pun?
Algorithm followed: Was the algorithm described, if any followed?
Coherence: Is the text coherent?
Funny: Is the text funny?

Figure 2: Criteria used to apply the SelfRefine framework to each of our designed prompts.

Finally, we discuss the limitations of our approach and suggest directions for future research to advance humor generation in code-mixed languages further.

2 Related Work

The field of computational humor has long explored the creation and understanding of puns. Early work by Lessard and Levison (1992) focused on generating humor through an antonym-based system designed to create Tom Swifties. The Joke Analysis and Production Engine (JAPE), a pun generator, produced novel outputs and was later formalized (Ritchie, 2003). Researchers have called for more formal computational models of humor, emphasizing the need for experimental validation and advocating for the formalization of humor theories (Brône et al., 2006).

Over time, approaches to computational humor have evolved. Modern methods now include neural models (Yu et al., 2018), surprisal-based techniques (He et al., 2019), and generative adversarial networks (Luo et al., 2019). The exploration of code-mixed puns, particularly those combining Hindi and English, is a more recent development in the field. Efforts to create datasets for code-mixed Hindi-English include notable contributions from Singh et al. (2018), who curated a POS-tagged Hinglish corpus by scraping Twitter. Aggarwal et al. (2018) made pioneering efforts in identifying Hindi-English code-mixed puns. Utilizing advertisement datasets, they determined the language of each word, then identified potential pun locations using a combination of n-gram models, smoothing techniques, and phonetic similarity. Their goal was to minimize contextual differences between the word altered to create the pun and the original word, thereby maximizing contextual appropriateness. Agarwal and Narula (2021) focused on generating jokes in code-mixed Hindi-English using Hinglish word embeddings. They used deep learning approaches to generate jokes in English, which were then transformed into code-mixed Hindi-English by converting selective parts of the text or using a pre-trained encoder and transfer learning method. Their work primarily focused on general joke generation. However, to the best of our knowledge, our work on generating homophonic puns in code-mixed English-Hindi is novel.

3 CoT prompting to generate puns reveals problems with finding Homophones

Chain of Thought prompting (Wei et al., 2023) has proven to be a simple yet effective technique for enhancing outputs and identifying where LLMs struggle most in performing a given task.

Post-hoc refinement is a common strategy to fix mistakes, and some popular strategies suggested recently focus on automated feedback that tries to self-critique (Xu et al., 2024; Madaan et al., 2024; Akyürek et al., 2023) and improve its output iteratively. The SelfRefine (Madaan et al., 2024) framework has been shown to be effective in various settings and has been adapted to various contexts to improve performance (Wadhwa et al., 2024; Tian et al., 2024; Shridhar et al., 2023). The idea is to assess the generated output by the same LLM with certain criterion (see Figure : 2), and then feed the assessment back to the LLM to refine the prompt, and keep doing this iteratively. For all our generated puns, we report our final answers after going through each stage iteratively at least 3 times. We stop early in case, the pun generated does not differ in terms of Levenshtein distance from the previously generated pun or if a positive automated assessment is made by the LLM.

LLMs struggling to find homophones

Step 1: EN: **pen**, HI: पेन (pen) [pɛn], HIToEN: **pen**
Step 2: Since **pen** == **pen**, go back to step 1.

Step 1 (redo): EN: **apple**, HI: एप्पल (apple) ['æpəl], HIToEN: **apple**
Step 2: Since **apple** == **apple**, go back to step 1.

Step 1 (redo): EN: **ball**, HI: बॉल (ball) [bɔl], HIToEN: **ball**
Step 2: Since **ball** == **ball**, go back to step 1.
...

Figure 3: Model output stuck in a recursive loop, repeating steps due to failure in finding distinct homophones. While the model detects incorrect homophones, it fails to correct them, leading to repeated attempts. This excerpt illustrates the pattern; the full trace included 70 retries. Most cases where HIToEN equals EN occur due to loan words in Hindi.

Step 1: EN: **doctor**, HI: डाक्टर (doctor) ['dɒktə], HIToEN: **doctor**
Step 2: Since **doctor** == **doctor**, go back to step 1.

Step 1 (redo): EN: **dude**, HI: दूध (milk) [d̪ud̪ʱɐ], HIToEN: **milk**
Step 2: Since **milk** != **dude**, proceed to step 3.

Figure 4: Illustrative trace showing rejected homophones and the selection process for acceptable ones.

When using vanilla prompting to generate code-mixed puns, the outputs typically result in grammatically correct but incoherent sentences that do not attempt to make puns (see Figure 1). To further understand this, we provided instructions based on the algorithm suggested by He et al. (2019) for creating puns, and tested the model in zero-shot, one-shot, and few-shot settings (with 2, 4, and 6 examples). Across all settings, the prompt included instructions to encourage CoT reasoning to help identify the bottleneck. The generation temperature was varied, and between 10 to 20 samples were generated and manually analysed for each configuration.

The general framework of the prompts designed is illustrated in Figure 5. This framework forced the model to regenerate its output until it successfully identified homophones. In zero-shot and one-shot learning settings, the model often incorrectly identified homophones, resulting in statements rather than actual attempts at puns.

Prompt for confirming the homophone before proceeding

Instruction: Construct a code-mixed Hindi-English pun based on the steps below.

Steps:
Step 1. Create a triplet of:
a. English word **EN**
b. Hindi word **HI**, homophone to **EN**
c. **HI** translated to English, labeled **HIToEN**
Step 2. If **HIToEN** == **EN**, redo step 1. Otherwise, proceed to step 3
Step 3. Construct short sentences (less than 10 words) with **EN** as the object of the sentence.
Step 4. Replace **EN** with **HI**.
Step 5. Replace the noun phrase at the start of the sentence with a contextualized phrase that is closely related to the **HIToEN** word.

Example:
Step 1: EN: **dude**, HI: दूध [d̪ud̪ʱɐ] (milk), HIToEN: **milk**
Step 2: Since **milk** != **dude**, proceed to step 3.
Step 3: Construct sentence with **EN** as the object: “Jack asked - What’s up dude?”
Step 4: Replace **EN** with **HI**: “Jack asked - What’s up दूध [d̪ud̪ʱɐ] (dude)?”
Step 5: Replace the noun phrase with a contextualized phrase: “The cow asked - What’s up दूध [d̪ud̪ʱɐ] (dude)?”

Figure 5: Prompt structure to guide the model to confirm the homophone before proceeding. This input prompt contains instructions that are modeled on the basis of the algorithm suggested by He et al. (2019) to generate puns in English with necessary modifications to fit the current context

However, the few-shot settings included examples with negative samples of homophones, followed by the rejection of these negative homophones to focus on actual homophones. A sample of this approach is shown in Figure 4. Including such examples in few-shot settings led to interesting cases where the model got stuck in a recursive loop, unable to progress beyond the initial step of identifying homophones. This often resulted in the maximum token limit being reached, with the API output oscillating between the first two steps for up to 70 turns. A portion of these recursive outputs is shown in Figure 3. We hypothesize that one of the reasons these models struggle to create convincing puns is their inability to move beyond the initial step of even identifying homophones.

4 Homophone Identification

The Epitran package [2] available in Python can be used to convert orthographic text into their International Phonetic Alphabet (IPA) forms, and its efficacy has been well studied and reported in this regard (Mortensen et al., 2018). The most common words found in English as well as Hindi are collected from various sources, and these words are converted into their IPA forms using Epitran. The resulting values are compared with each other and by using Minimum edit distance as a measure, similar IPAs (lower minimum edit measures) across the 2 languages are grouped with each other. This simple yet effective technique can be used to get homophones between any of the 61 languages that are supported at the moment by Epitran. However, it must be noted that some post processing is required to remove unnecessary and extra matches that might occur. Borrowed words from the same language need to be weeded out so that we are not left with simple inflections of the same word in the 2 languages. Therefore, the Hindi words are translated as well as transliterated into English. If the minimum edit distance of the translation and transliteration is low as well, then that implies that the word in Hindi was in all likelihood a borrowed word from English, and hence that word is discarded. Words with less than 3 characters are discarded as well, since they are most likely to be filler words and would not be interesting candidates for pun generation.

[2]https://github.com/dmort27/epitran

5 Transliteration from Devanagari Script to Latin Script

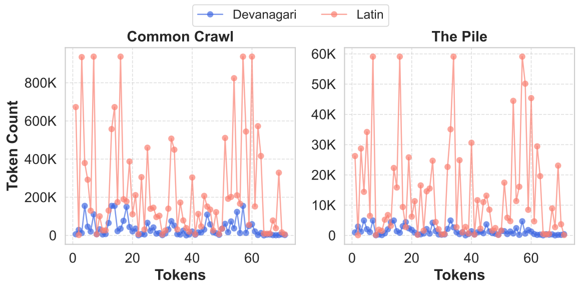

Figure 6: Comparing the count of tokens in the Devanagari script vs the same token in the Latin script for the 100 most common words in Hindi across both the Common Crawl and the Pile

To understand the prevalence and distribution of Hindi words in Devanagari and Latin scripts, we applied the What's In My Big Data? (WIMBD) framework (Elazar et al., 2024) to systematically analyze two large datasets: C4 (used to train T5 (Raffel et al., 2020)) and The Pile (Gao et al., 2020), both commonly used to train large language models (LLMs). WIMBD provides an accessible suite of tools designed to facilitate exploration and analysis of large language datasets. We specifically used the count functionality to compare the exact occurrences of Hindi words in both the scripts and found a substantially higher representation of the Latin script (see Figure: 6). This discrepancy likely results from a combination of factors, such as the dominance of Latin script in informal communication, presence of the same homographic English word, all of which align with our goal of finding homophones for pun generation. The Latin script's higher token counts increase the likelihood of identifying phonetically similar words, thus facilitating effective pun creation. Therefore to support our homophone generation step, we propose a rule-based transliteration algorithm.

The algorithm involves individual character mappings for the 42 extended consonants, 12 vowels, and 12 vowel markers of the Devanagari script to their Latin script counterparts. These mappings are stored locally. For any given word in Devanagari script, the algorithm iterates over each character. If a character is a raw vowel (not a vowel marker), the corresponding Latin equivalent is added directly from the stored mappings. If the next character is a vowel marker, it is combined with the current character. The subsequent character is then checked,

Data: devng: Word in Devanagari script
Result: latin: Word in Latin script
Initialize `latin` as an empty string; **for** *each character* `c` *in* `devng` **do**
    if *c has a following vowel marker* **then**
        c = c + vowel marker
    end
    if `c` *is a vowel* **then**
        Append the Latin equivalent to `latin`;
    end
    if `c` *is a consonant* **then**
        if *there is a consonant following c and it also has a vowel marker* **then**
            Append the Latin equivalent of c to `latin`;
        end
        else
            Append the Latin equivalent along with 'a' to handle the schwa sound to `latin`;
        end
    end
end

Algorithm 1: Transliteration Algorithm for Devanagari to Latin

effectively creating a lookahead of 2.

If the following character is a consonant with a vowel marker, the current character's mapping is appended with an "a" to denote the schwa sound typically associated with Hindi consonants. This process accounts for the rule in Hindi where a consonant is assumed to end with a schwa unless followed by another vowel. However, since two schwas do not appear consecutively in natural language, this lookahead prevents unnecessary additions of the schwa sound. If the subsequent character lacks a vowel marker, the "a" is omitted.

Word	IPA	Variation 1	Variation 2
भूख	[bʱuːkʰ]	bhook	bhuk
बिजली	[bɪd͡ʒlɪ]	bijli	bijlee

Table 1: Examples of Variations that are considered acceptable by native speakers owing to a lack of transcription standard

The proposed transliteration system achieves 64% accuracy on the Dakshina dataset (Roark et al., 2020), which, while lower than the 72.4% accuracy of the current state-of-the-art (Madhani et al., 2023) which are based on neural architectures, offers a computationally efficient alternative. Through error analysis, we identified issues within the dataset itself, including errors in ground annotations (see Table 2).

Word	IPA	Present	Actual
इनके	[ɪnkɛ]	unke	inke
में	[meːn]	be	mein
है	[hɛ]	ahai	hai

Table 2: Examples of Mistakes in the annotation of Google's Dakshina dataset, along with the correct transliterations

Additionally, the lack of standard spellings in the Latin script for transliterated Hindi poses a challenge. Since the script isn't native, multiple correct spellings often exist, yet most datasets, including those available today, only account for one variation. This results in perfectly acceptable alternative spellings being incorrectly marked as errors (see Table 1).

In conclusion, while transliteration significantly aids in processing Hinglish datasets and is crucial for the post-processing needed in homophone generation for Hinglish pun creation, its development is not without fundamental challenges. Addressing these challenges, such as handling spelling variations and inaccuracies in datasets, would make it easier to integrate transliteration systems into models like ours more effectively.

6 Modifying Pun Generation with Surprise

A stripped down version of the methodology suggested in He et al. (2019) is recreated for the current context. Given a pair of homophones (w_1, w_2), candidate sentences are found from a corpus (Brown corpus found in the NLTK chosen in our case). Sentences where the w_1 appears at the end is retained, since puns are generally considered funnier if there is an element of surprise, and the pun word appearing at the end increases the likelihood of the same. In the candidate sentences, we replace the w_1 with w_2 and subsequently try to replace the noun phrase at the start of the sentence with a topic that is related to the w_2 word instead of the w_1 word. The modified phrase results in a pun after post processing that checks for grammaticality. Since, in our case the w_2 is from a different language, we translate w_2 into English, before trying to find an appropriate topic for the same (through GLoVe Embeddings or other such distributional semantic similarity mechanisms).

A sample execution

- Homophone pairs: *city* , *seeti* (whistle)
- Find short candidate sentences from the Brown corpus where *city* appears at the end.
  The man lives in the heart of the city
- Replace *city* with the homophonic pair
  The man lives in the heart of the seeti
- Find apt topic to replace the subject.
  The referee lives in the heart of the seeti

Although the resulting phrase can lead to a pun, a filtration mechanism is required that can remove phrases / puns that don't make sense due to the change in language and can handle the grammaticality requirements of both the languages in question. This is a major limitation of this method while applying it in Code-Mixed Settings which will need to be improved upon in future extensions of this work.

7 Prompting LLMs by appending pregenerated list of Homophones

Since the cause of failure in GPT3.5 during pun generations was shown to be identification of homophones, the modules introduced in Sections 3 & 4 are leveraged and locally generated transliterated homophones are appended to the end of standardised prompts. A standard prompt structure is created for Zero shot, one shot, 4 shot (see Figure 7 as an example), 8 shot and finally 16 shot settings. In all cases, the input homophones are always appended at the end, and each input is run in a separate session without any history except for the examples in the few shot settings. This modified hybrid prompt setting improves the output drastically with many of the outputs generated fulfilling all criterion of being qualified as puns. Even, when the outputs generated are not funny, they are at least attempts at making a pun out of the given inputs.

Survey to Evaluate Hybrid Approach

A small survey was conducted to assess the quality of puns generated by our system across various settings. Fifty-six participants volunteered to rate the puns on a scale of 1 (not funny) to 5 (extremely funny). Given the voluntary nature of the survey, we evaluated 4-5 random samples per category. To ensure data integrity and prevent biased participation, we included human-generated puns from joke websites.

Each participant received the puns in a random order, presented via Google Forms. To minimize bias, participants were unaware of the pun's origin. All participants were bilingual in Hindi and English, aged 20-30.

After the survey concluded, we calculated the mean ratings for each category: Human, Zero Shot, One Shot, 4 Shot, 8 Shot, and 16 Shot. Submissions with a mean rating below 1 for human-generated puns were discarded, as they likely indicated a lack of engagement. Additionally, submissions with all puns rated 5 were removed to eliminate potentially casual or insincere participation. Following this filtration, 39 submissions remained. The results presented in all graphs are based on these 39 submissions.

Mean Ratings per Category

Since humour is subjective, even the human generated puns do not always get a full score, and the mean ratings across all the samples and participants turned out to 4.32 (see Figure 8). Compared to the same, the ratings of all the other categories were pretty decent, with even Zero shot getting an average of 3.31 with slight increase in averages resulting in a 3.61 for 16 shot settings. Although these means

Example Prompt for Hybrid Approach

Task: Generate a code-mixed Hindi-English pun based on the homophones provided as input. Some example input-output pairs are provided as reference.

Input: 'Submit', 'Sab Mit' (everything gets erased)
Output: "Exam ki answer sheet return karte hi SUBMIT jata hai"

Input: <EnWord>, <HiTransliteratedHomophone><(EnglishTranslation)>

Figure 7: Example prompt for the hybrid approach. This sample has been truncated to show the most relevant parts of the prompt. The final line of input is dynamically modified at runtime.

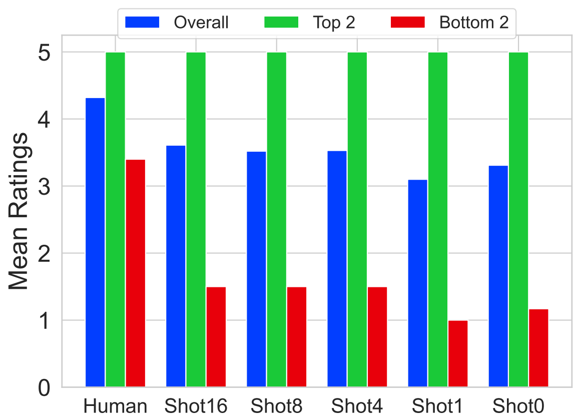

Figure 8: Mean ratings given overall, achieved by top 2 and achieved by bottom 2 puns per category of prompts

are high, the reason for the differences in the mean with the human ratings was analysed. The top 2 puns that got the highest mean scores in each category were taken, and for each category this mean turned out to be a perfect score (see Figure 8). Conversely, while trying to plot the same thing for the puns with the lowest scores, the main difference between the human generated ones get highlighted. The mean rating for Humans drops to a 3.4 which can still be considered a pretty decent score. However, for the generated puns, the score drops down significantly (see Figure 8). Thus, although our model can generate good puns, there is a lack of consistency in the generation.

7.1 Comparing Against Human Benchmarks

A box plot (see Figure 9) is created to show the variability between different samples and participants in each category. When raw ratings are considered, the scores Shot 16, Shot 4 show lower variance in their scores compared to the variances shown by the one shot and zero shot settings. As shown from the survey, it is unlikely for the mean to be a perfect 5 even for a set of human generated puns due to the subjectivity of humour. Therefore we compare the scores generated automatically against the benchmarks set by human performance. We calculate the mean rating given by each participant to the human generated jokes and per participant, we normalise the scores for all other categories based on this value. The resulting normalised scores are plotted as a box plot in Figure 10. It can be seen that Shot4, Shot8 and Shot16 perform the best. Zero Shot setting performs better than One shot settings, although they are both clearly inferior to the Few shot settings. All in all, the performance of such models in such hybrid settings is encouraging and further proves that LLMs can perform exceedingly well on highly specialised tasks such as generating puns in code mixed settings, provided they are leveraged aptly and their weaknesses are well understood.

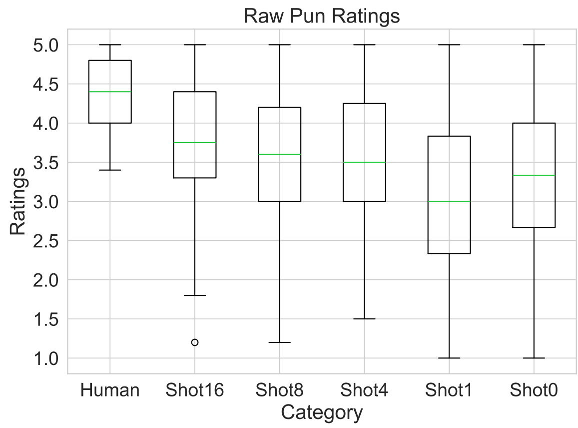

Figure 9: Box Plot with Mean Scores Per category of Prompts

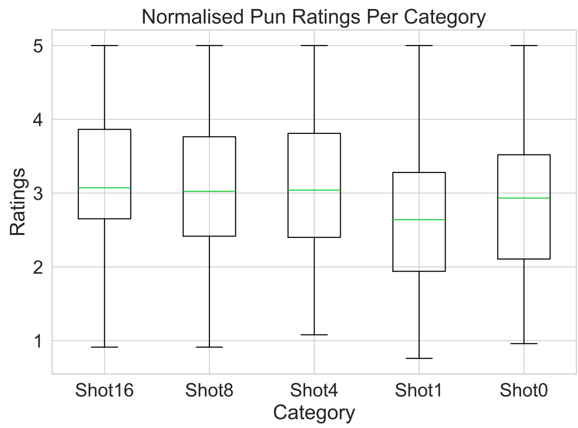

Figure 10: Box Plot with Normalised ratings according to Human Benchmarks

8 Limitations and Future Work

- Our homophone generation is constrained to single whole words. $I\ will\ see$ vs $Aalsi$ (lazy) are homophonic but span multiple words. Leveraging such multi word homophones could enhance the creativity of the generated content.

- The prompting experiments were conducted exclusively using $gpt-3.5-turbo$. Our current strategies do not cover the full range of possible prompts, and future work could expand on this.

- Our pun generation using Surprisal focuses on replacing individual nouns, rather than entire noun phrases. This may result in contextually weaker replacements. Fine-tuning embeddings on Hinglish data to better handle the semantics of noun phrases could lead to more contextually appropriate and higher-quality puns.

- The survey included only a small number of samples per method, as participants volunteered without compensation. A larger sample size would be needed to draw more robust conclusions.

- The pun generation methodology is not language-specific. Future work could apply the framework to other code-mixed languages, such as Spanglish (Spanish-English) or Arabizi (Arabic-English), to assess its broader applicability.

Additionally, it must be mentioned that publicly available datasets of Hinglish often suffer from various shortcomings, including inadequate maintenance, annotation issues, and insufficient scale. These limitations hinder their suitability for training and evaluating AI models effectively. Addressing these challenges and improving their quality, availability and out of the box usability is essential.

9 Conclusion

This study presents a novel exploration of homophonic pun generation within the code-mixed Hinglish context. By analyzing and adapting English-based humor generation techniques, particularly through hybrid prompting approaches with Large Language Models (LLMs), we demonstrate the feasibility of creating Hinglish puns that align with native humoristic nuances. Feeding cross-lingual homophones as well as incorporating transliteration techniques improved the pun generation capability of LLMs in the given setting, making the outputs more contextually relevant. Our findings, supported by crowd-sourced evaluations, suggest that while hybrid prompting strategies can generate engaging Hinglish puns, challenges remain, especially in achieving consistently high humor quality. Future work will benefit from expanding homophone recognition to multi-word phrases, refining the selection of humor-optimized LLMs, and developing more robust code mixed datasets to enhance the accuracy and cultural relevance of computational humor in multilingual contexts.

Ethical Statement

The evaluation survey was conducted anonymously and no private data of any individual was collected. External datasets, packages used have been attributed to the original authors and were used only after confirming that they were distributed under licenses that permit research work.

Acknowledgments

I thank Michael Hahn, Alexander Koller and the anonymous reviewers for helpful comments.

References

Kaustubh Agarwal and Rhythm Narula. 2021. Humor generation and detection in code-mixed hindi-english. In *Proceedings of the Student Research Workshop Associated with RANLP 2021*, pages 1–6.

Srishti Aggarwal, Kritik Mathur, and Radhika Mamidi. 2018. Automatic target recovery for hindi-english code mixed puns. *arXiv preprint arXiv:1806.04535*.

Afra Feyza Akyürek, Ekin Akyürek, Aman Madaan, A. Kalyan, Peter Clark, D. Wijaya, and Niket Tandon. 2023. Rl4f: Generating natural language feedback with reinforcement learning for repairing model outputs. In *Annual Meeting of the Association for Computational Linguistics*.

Geert Brône, Kurt Feyaerts, and Tony Veale. 2006. Introduction: Cognitive linguistic approaches to humor.

Yanai Elazar, Akshita Bhagia, Ian Helgi Magnusson, Abhilasha Ravichander, Dustin Schwenk, Alane Suhr, Evan Pete Walsh, Dirk Groeneveld, Luca Soldaini, Sameer Singh, et al. 2024. What's in my big data? In *The Twelfth International Conference on Learning Representations*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

He He, Nanyun Peng, and Percy Liang. 2019. Pun generation with surprise. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1734–1744, Minneapolis, Minnesota. Association for Computational Linguistics.

Sophie Jentzsch and Kristian Kersting. 2023. Chatgpt is fun, but it is not funny! humor is still challenging large language models. *arXiv preprint arXiv:2306.04563*.

G Lessard and M Levison. 1992. Computational modelling of linguistic humour: Tom swifty. In *Paper Delivered at the ALLC/ACH Joint Annual Conference. Christ Church, Oxford*.

Fuli Luo, Shunyao Li, Pengcheng Yang, Baobao Chang, Zhifang Sui, Xu Sun, et al. 2019. Pun-gan: Generative adversarial network for pun generation. *arXiv preprint arXiv:1910.10950*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Yash Madhani, Sushane Parthan, Priyanka Bedekar, Gokul Nc, Ruchi Khapra, Anoop Kunchukuttan, Pratyush Kumar, and Mitesh Khapra. 2023. Aksharantar: Open indic-language transliteration datasets and models for the next billion users. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 40–57, Singapore. Association for Computational Linguistics.

David R. Mortensen, Siddharth Dalmia, and Patrick Littell. 2018. Epitran: Precision g2p for many languages. In *Proceedings of the International Conference on Language Resources and Evaluation*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Graeme Ritchie. 2003. The jape riddle generator: technical specification. *Institute for Communicating and Collaborative Systems*.

Brian Roark, Lawrence Wolf-Sonkin, Christo Kirov, Sabrina J. Mielke, Cibu Johny, Işın Demirşahin, and Keith Hall. 2020. Processing south asian languages written in the latin script: the dakshina dataset. In *Proceedings of The 12th Language Resources and Evaluation Conference (LREC)*, pages 2413–2423.

Kumar Shridhar, Koustuv Sinha, Andrew Cohen, Tianlu Wang, Ping Yu, Ramakanth Pasunuru, Mrinmaya Sachan, Jason Weston, and Asli Celikyilmaz. 2023. The art of llm refinement: Ask, refine, and trust. In *North American Chapter of the Association for Computational Linguistics*.

Kushagra Singh, Indira Sen, and Ponnurangam Kumaraguru. 2018. A twitter corpus for hindi-english code mixed pos tagging. In *Proceedings of the sixth international workshop on natural language processing for social media*, pages 12–17.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. 2024. Toward self-improvement of llms via imagination, searching, and criticizing. *ArXiv*, abs/2404.12253.

Manya Wadhwa, Xinyu Zhao, Junyi Jessy Li, and Greg Durrett. 2024. Learning to refine with fine-grained natural language feedback. *ArXiv*, abs/2407.02397.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Genta Indra Winata, Alham Fikri Aji, Zheng-Xin Yong, and Thamar Solorio. 2022. The decades progress on code-switching research in nlp: A systematic survey on trends and challenges. *arXiv preprint arXiv:2212.09660*.

Lawrence Wolf-Sonkin, Vlad Schogol, Brian Roark, and Michael Riley. 2019. Latin script keyboards for south asian languages with finite-state normalization. In *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*, pages 108–117.

Wenda Xu, Daniel Deutsch, Mara Finkelstein, Juraj Juraska, Biao Zhang, Zhongtao Liu, William Yang Wang, Lei Li, and Markus Freitag. 2024. Llmrefine: Pinpointing and refining large language models via fine-grained actionable feedback. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 1429–1445.

Zhiwei Yu, Jiwei Tan, and Xiaojun Wan. 2018. A neural approach to pun generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1650–1660.