Understanding and Mitigating Tokenization Bias in Language Models

Buu Phan [*1] **Marton Havasi** [2] **Matthew Muckley** [2] **Karen Ullrich** [2]

Abstract

State-of-the-art language models are autoregressive and operate on subword units known as tokens. Specifically, one must encode the conditioning string into a list of tokens before passing to the language models for next-token prediction. We show that popular encoding schemes, such as maximum prefix encoding (MPE) and byte-pair-encoding (BPE), induce a sampling bias that cannot be mitigated with more training or data. To counter this universal problem, for each encoding scheme above, we propose a novel algorithm to obtain unbiased estimates from any language model trained on tokenized data. Our methods do not require finetuning the model, and the complexity, defined as the number of model runs, scales linearly with the sequence length in the case of MPE. As a result, we show that one can simulate token-free behavior from a tokenized language model. We empirically verify the correctness of our method through a Markov-chain setup, where it accurately recovers the transition probabilities, as opposed to the conventional method of directly prompting tokens into the language model.

1. Introduction

Tokenization is a preprocessing procedure used in many state-of-the-art (SOTA) language models (LMs) such as GPTs (Brown et al., 2020), Llama (Touvron et al., 2023) and Gemini (Gemini, 2023). It divides the input text into smaller subword units while retaining linguistic importance, helping to address vocabulary limitations such as unknown words. Tokenization also shortens (compresses) the input context length (Sennrich et al., 2015; Kudo & Richardson, 2018). Since effective compression allows transformer-based LMs to handle longer context strings, many works (Zouhar et al., 2023; Gallé, 2019; Goldman et al., 2024) have focused on enhancing vocabulary design and encoding algorithms for better performance in downstream tasks. However, the relationship between compression and model performance remains unclear. Some research suggests the impact of compression is not always positive (Schmidt et al., 2024; Dagan et al., 2024; Goyal et al., 2023). Consequently, understanding tokenization's effect on model performance continues to be an open question.

Tokenization has been criticized for introducing many shortcomings in LMs. These include sensitivity to spelling and morphological structure (Xue et al., 2022), language-based biases (Petrov et al., 2024), subpar performance in specific tasks such as arithmetic (Singh & Strouse, 2024), or new domains (Liu et al., 2023a). One approach to address these issues is through fine-tuning the model with new vocabularies; however, this often complicates the training process and requires domain-specific expertise (Chen et al., 2023; Liu et al., 2023b). Furthermore, the performance gains do not provide a theoretical understanding of whether these limitations truly arise from the tokenization process or result from suboptimal model training. Another direction is to develop token-free LMs (Yu et al., 2024; Nawrot et al., 2022; Tay et al., 2021). While this approach has potential as it eliminates tokenization-related issues, it significantly increases the context length, resulting in performance that still lags behind the SOTA tokenized LMs [1](Yu et al., 2024).

In this work we offer new theoretical insights on the behavior of tokenized LMs. We show that they are statistically equivalent to their token-free counterparts. Specifically, we examine the maximum prefix encoding (MPE) scheme employed in the WordPiece tokenization method (Devlin et al., 2018; Song et al., 2020) and find that this process not only results in biased estimates of next token probabilities, but also leads to overall skewed estimates of subsequent character probabilities. In general, this bias persists despite an increase in training data, even within the simple setting of a 1st-order Markov chain. Such bias occurs due to the implicit disparity between the domain of the conditioning context, namely, characters versus tokens. Nevertheless, we will show that it is possible to correct this bias without resorting to finetuning. Once adjusted, it becomes possi-

[*] Work done during internship at Meta FAIR [1] University of Toronto [2] Meta AI. Correspondence to: Buu Phan <truong.phan@mail.utoronto.ca>, Karen Ullrich <karenu@meta.com>.

Work presented at TF2M workshop at ICML 2024, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

[1] We refer language models that process tokenized texts as tokenized language models (tokenized LMs).

ble to simulate the token-free behavior learned implicitly by the tokenized LM and even (theoretically) mimic the behavior of another tokenized model employing a distinct vocabulary set, all without requiring finetuning. Our specific contributions are as follows:

- We show the presence of a bias in the next-token distribution that arises as a result of the tokenization process.
- We present two novel algorithms to correct this bias for MPE and Byte-Pair-Encoding (BPE) respectively. Due to space limit, the analysis and algorithm for BPE are presented in Appendix H.
- We verify the correctness of our algorithms on learning the transition matrix of a k-th order Markov chain.

2. Problem Setup

We begin by establishing the tokenization and language models setup in our paper. We then describe the next-character sampling bias problem due to tokenization.

2.1. Notations and Setup.

String Notations. For any string s, we denote its substring from i to j as $x_i^j{:=}x_i x_{i+1}..x_j$, where each x is a character of the alphabet $\mathcal{A}$. For a given string x_1^N, we define the prefix function that generates a set containing all possible prefix strings of x_1^N, represented as $\text{prefix}(x_1^N){=}\{x_1^1, x_1^2, x_1^3, ..., x_1^N\}$. Also, we define a concatenation function $\text{concat}(.)$ that concatenates the given list of strings, e.g given $s_1{=}x_1^{N_1}$ and $s_2{=}y_1^{N_2}$, we obtain $\text{concat}(s_1, s_2){=}\text{concat}(x_1^{N_1}, y_1^{N_2}){=}x_1...x_{N_1}y_1...y_{N_2}$. Finally, we denote the set of all strings that start with a prefix x_1^n as $\mathcal{S}(x_1^n){=}\{s|x_1^n{\in}\text{prefix}(s)\}$.

Tokenization Setting. We assume having a predefined vocabulary $\mathcal{V}$ constructed using any tokenization algorithm such as BPE, with the condition that $\mathcal{A}{\subseteq}\mathcal{V}$. We use t to denote a token in $\mathcal{V}$, i.e. $t{\in}\mathcal{V}$. Importantly, we use the longest prefix matching strategy for tokenization (encoding), denoted as $\text{encode}(.)$, similar to the approach used in the Wordpiece algorithm (Devlin et al., 2018; Song et al., 2020). Given a sequence of tokens t_1^k, the function $\text{decode}(.)$ returns the concatenated string resulting from processing each token in the sequence. Finally, the set of all strings that starts with the tokens t_1^k is defined as $\mathcal{S}(t_1^k){=}\{s|t_1^k{=}\text{encode}(s)_1^k\}$.

Tokenized LMs. We assume having access to a tokenized autoregressive LM with parameters θ that is trained with tokens from $\mathcal{V}$ and maximum prefix matching. The target distributions on the character domain is denoted as $P_{\text{gt}}(x_{n+1}^N|x_1^n)$ and on the token domain is $P_{\text{gt}}(t_{i+1}|t_1^i)$. For simplicity, unless otherwise stated, we implicitly assume each probability term involves θ. Using the model, we assume that one can compute $P(t_{i+1}|t_1^i)$ for any integer $i > 0$. In this work, we consider LMs trained under the standard setup, where each string s in the dataset is first tokenized with the encoding function $\text{encode}(.)$ and vocabulary $\mathcal{V}$, and the parameters θ are optimized to maximize the predictive likelihood of the next token in the tokenized dataset.

2.2. Next-Character Sampling Bias

We first define the (next-character) sampling bias problem that describes the discrepancy between the character level and token level predictions for tokenized LMs.

Definition 2.1. (Next-Character Sampling Bias) Let the input prompt string x_1^n has $t_1^i{=}\text{encode}(x_1^n)$ as the corresponding encoding. The next-character sampling bias occurs for this prompt when $P_{\text{gt}}(x_{n+1}|x_1^n){\neq}P_{\text{gt}}(x_{n+1}|t_1^i)$ where $P_{\text{gt}}(x_{n+1}|t_1^i){=}\sum_{t\in\mathcal{E}} P_{\text{gt}}(t_{i+1}{=}t|t_1^i)$ where $\mathcal{E} = \{t \in \mathcal{V}|\text{decode}(t) \in \mathcal{S}(x_{n+1})\}$.

In other words, the probability of the next character being "c" may be different from the sum of the probabilities of all tokens that start with "c". Note that this character-level probability offers a broader perspective compared to the probability of the subsequent token being exactly "c".

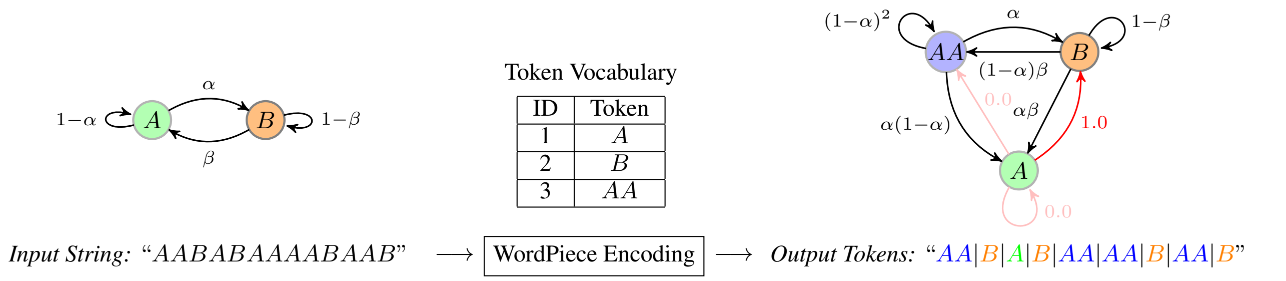
Example. Consider a first order Markov chain with two states $\{$"A", "B"$\}$ as shown in Figure 1 (left). Each string is tokenized with $\mathcal{V}{=}\{$"AA", "A", "B"$\}$, which leads to a new Markov chain whose states and transition matrix is shown in Figure 1 (right). Details on computing the transition matrix of the new Markov chain is in Appendix F. We first observe that for the prompt $s_1{=}$"AA" and $s_2{=}$"B", there is no bias problem after marginalization[2]. However, for the prompt $s_3{=}$"A", the sampling bias occurs as $P_{\text{gt}}(x_2{=}$"B"$|t_1{=}$"A"$){=}1.0$, which is not equal to $P_{\text{gt}}(x_2{=}$"B"$|x_1{=}$"A"$){=}\alpha$, i.e. the optimally trained LM will always output "B". In fact, for any context string that ends with token "A", e.g "$AA|A$" and "$B|A$" (tokens are separated by "|"), such LM will always output "B".

Since this applies to any optimally trained LM, increasing the training set size does not mitigate this problem. The reason for this sampling bias is that, during the tokenization process with longest prefix matching, the token "A" must be followed by the token "B". Else, MPE encoding will merge to create a longer token "AA". We generalize this phenomenon with the definition of invalid encodings.

Definition 2.2. (Invalid Encodings) The list of tokens (an encoding) t_1^i is invalid if $\text{encode}(\text{decode}(t_1^i)){\neq}t_1^i$. Otherwise, it is a valid encoding.

For example, let $\mathcal{V}{=}\{$"c", "a", "t", "at", "cat"$\}$ then ["c","at","t"] and ["c","a","t","t"] are invalid encodings of "$catt$". We now show in Proposition 2.3 that the existence of invalid encodings introduces sampling bias, generalizing the observed phenomenon in the Markov chain example to any autoregressive distribution.

[2] For example, we have $P_{\text{gt}}(t_{i+1}{=}$"AA"$|t_i{=}$"AA"$) + P_{\text{gt}}(t_{i+1}{=}$"$A$"$|t_i{=}$"$AA$"$) = \alpha = P_{\text{gt}}(x_{n+1}{=}$"$A$"$|x_n{=}$"$A$"$)$



Token Vocabulary

ID	Token
1	A
2	B
3	AA

Input String: "$AABABAAAABAAB$" ⟶ WordPiece Encoding ⟶ *Output Tokens:* "$AA|B|A|B|AA|AA|B|AA|B$"

Figure 1: Next-Character sampling bias introduced by the WordPiece encoding algorithm. In this example, given the context token "A", the model will always predict the next token as "B" with probability 1.0. We present a technique that, given a language model trained on tokenized domain, eliminate this bias and recover the accurate unbiased sampling distribution.

Proposition 2.3. *(Token-Induced Zero Probability) Let t_1^i be a sequence of input tokens. For any invalid encoding t_1^i, we have $P_{\text{gt}}(t_1^i){=}0.0$ and the conditional probability $P_{\text{gt}}(t_{i+1}|t_1^i)$ is undefined. In the case t_1^i is valid, then $P_{\text{gt}}(t_{i+1}|t_1^i){=}0.0$ if t_1^{i+1} is invalid. Furthermore, let $x_1^n{=}\text{decode}(t_1^i)$, then for any string x_{n+1}^N such that $\text{encode}(\text{concat}(\text{decode}(t_1^i), x_{n+1}^N)){\neq}t_1^i$, we have $P_{\text{gt}}(x_{n+1}^N|t_1^i){=}0.0$.*

Proof. See Appendix C. □

Remark 1. Proposition 2.3 implies that LMs may not function as expected when presented with invalid encodings, because these models will never be exposed to such inputs within the dataset. This directly implies that the practice of evaluating LMs under different encodings (Cao & Rimell, 2021; Chirkova et al., 2023) is suboptimal.

3. Alleviating Sampling Bias

We propose a method to remove the described bias and recover the original token-free autoregressive model, i.e. expressing the implicitly learned $P(x_{n+1}^N|x_1^n)$ using the tokenized LM that outputs the conditional probability $P(t_{i+1}|t_1^i)$. For $N{=}n{+}1$, this captures the behavior of a token-free model, i.e. sampling the next character instead of a whole token. We assume our LM follows Proposition 2.3 on zero probability events and undefined conditional probability for invalid encodings. Appendix G justifies this assumption and provides its practical implementation.

Our method consists of two stages. In the first stage, the idea is to identify the condition when $P(x_{n+1}^N|t_1^i) = P(x_{n+1}^N|x_1^n)$ where $t_1^i = \text{encode}(x_1^n)$. Once identified, we can refactor the conditional probability to match the conditioning events. In the second stage, we compute $P(x_{n+1}^N|t_1^i)$ using the LM output probability, i.e. $P(t_{i+1}|t_1^i)$, through the novel Maximum Prefix Correction (MPC) Algorithm.

3.1. Refactoring

Our method removes the bias by connecting character and token domains through a special subset of tokens $\mathcal{V}^*{\subset}\mathcal{V}$, whose elements $t^*{\in}\mathcal{V}^*$ are not a substring of any other tokens in $\mathcal{V}$ but itself. For example, given $\mathcal{V}{=}\{$"AAA", "AA", "CB", "A", "B", "C"$\}$, then $\mathcal{V}^* = \{$"AAA", "CB"$\}$. In the Markov example in Section 2, this corresponds to the tokens "AA" and "B". Also, we assume that any string x_1^N has the first token $t_1 \in \mathcal{V}^*$[3]. Consider the input string x_1^n and its corresponding encoding $t_1^i{=}\text{encode}(x_1^n)$, Proposition 3.1 shows the sufficient condition for $\mathcal{S}(t_1^i){=}\mathcal{S}(x_1^n)$.

Proposition 3.1. *Let $s^* = x_1^n$, where $t_1^i = \text{encode}(s^*) = \text{encode}(x_1^n)$. Then we have $\mathcal{S}(t_1^i) \subset \mathcal{S}(x_1^n)$, i.e. for any string s where $t_1^i = \text{encode}(s)_1^i$, we have $P(x_1^n|t_1^i) = 1.0$. In the case $t_i \in \mathcal{V}^*$, then we also have that $\mathcal{S}(t_1^i) = \mathcal{S}(x_1^n)$, i.e. any string s where $x_1^n \in \text{prefix}(s)$ must have the first i tokens as t_1^i and $P(t_1^i|x_1^n) = 1.0$.*

Proof. See Appendix D. □

The intuition for Proposition 3.1 is that the subsequent string after $t_i{\in}\mathcal{V}^*$ cannot change the tokenization for x_1^n. We now establish one of the main results in Corollary 3.2.

Corollary 3.2. *Following Proposition 3.1, suppose $t_i \in \mathcal{V}^*$ then we have $P(x_{n+1}^N|x_1^n){=}P(x_{n+1}^N|t_1^i)$. Similarly, we also have $P(t_{i+1}^j|x_1^n){=}P(t_{i+1}^j|t_1^i)$.*

Proof. See Appendix D. □

We note that Proposition 3.1 and Corollary 3.2 always hold, regardless of the value of θ. In general, consider when the last token of $\text{encode}(x_1^n)$ is not in $\mathcal{V}^*$, we can refactor $P(x_{n+1}^N|x_1^n)$ as follow:

$$P(x_{n+1}^N|x_1^n){=}\frac{P(x_{n_k+1}^N|t_1^k)}{P(x_{n_k+1}^n|t_1^k)}, \tag{1}$$

where k is the last token in $\text{encode}(x_1^n)$ such that $t_k{\in}\mathcal{V}^*$ and $x_1^{n_k}{=}\text{decode}(t_1^k)$, where $n_k \leq n$. Proof details of this step can be found in the Appendix E. We then use the MPC algorithm to compute each term in the RHS individually.

3.2. Maximum Prefix Correction Algorithm

We present the MPC algorithm in Algorithm 1, that allows us to compute the probabilities $P(x_{n_k+1}^N|t_1^k)$ and $P(x_{n_k+1}^n|t_1^k)$ in Equation (1). Note that this algorithm does not require $t_k{\in}\mathcal{V}^*$. Details on the algorithmic correctness are shown in Appendix E.

[3] Many current language models begins with a start token <start> in $\mathcal{V}^*$, e.g. in SentencePiece (Kudo & Richardson, 2018).

Algorithm 1 Maximum Prefix Correction Algorithm. This algorithm recursively computes $P(x^N_{n_k+1}|t^k_1)$.

```
1:  procedure COMPUTE($x^N_{n_k+1}, t^k_1$)
2:      // Branching Step:
3:      $\mathcal{B} = \{t \in \mathcal{V} | x^N_{n_k+1} \in \text{prefix}(\text{decode}(t))\}$
4:      $\text{b}_{\text{val}} = \sum_{t \in \mathcal{B}} P(t_{k+1} = t | t^k_1)$
5:      // Base Case:
6:      if encode($x^N_i$) $\in \mathcal{V}$ then
7:          return $\text{b}_{\text{val}}$
8:      end if
9:      //Extract the Next Token:
10:     $t_{k+1} = \text{encode}(x^N_{n_k+1})_1$
11:     // Passing Step:
12:     $\text{p}_{\text{val}} = P(t_{k+1}|t^k_1)$
13:     $\text{p}_{\text{val}} = \text{p}_{\text{val}} \times \text{COMPUTE}(x^N_{n_{k+1}+1}, t^{k+1}_1)$
14:     return $\text{b}_{\text{val}} + \text{p}_{\text{val}}$
15: end procedure
```

The idea is to marginalize out $P(x^N_{n_k+1}|t^k_1)$ by considering two complementary events: when the next token t_{k+1} has a prefix $x^N_{n_k+1}$ (b_{val} in the Branch Step) versus when the next token t_{k+1} is contained within $x^N_{n_k+1}$ (p_{val} in the Pass Step). Formally, MPC computes the following probabilities:

$$\text{b}_{\text{val}} = P(x^N_{n_k+1}, t_{k+1} \in \mathcal{B}(x^N_{n_k+1}))|t^k_1), \quad (2)$$

$$\text{p}_{\text{val}} = P(x^N_{n_k+1}, t_{k+1} \notin \mathcal{B}(x^N_{n_k+1}))|t^k_1), \quad (3)$$

where $\mathcal{B}(x^N_{n_k+1})$=$\{t\in\mathcal{V}|x^N_{n_k+1}\in\text{prefix}(\text{decode}(t))\}$ and we immediately see that $P(x^N_{n_k+1}|t^k_1)$=b_{val}+p_{val}.

We provide an intuitive explanation for the algorithm following the example in Figure 2. Here, we would like to compute the probability $P(x^{n_k+3}_{n_k+1}$=“bee”$|t^k_1)$. The first possibility is that “bee” is a prefix of the next token, so we search for all such tokens (line 3 in the algorithm) and sum up their probability (line 4), i.e. b_{val}=$P(t_{k+1}$=“$beer$”$|t^k_1)$. Figure 2 visualizes this step as branching out the tree by finding all tokens completing the string. Since “$beer$” is not the only string that contains “bee”, e.g. “$beep$”, “$been$”, etc. we need to compute the probability for these other scenarios, each of which has t_{k+1}=“b” (the first token in “bee”, line 10 and 12) due to maximum prefix encoding. Then, we want to compute the probability that the subsequent string is “ee” (line 13), given the previous t^k_1 and t_{k+1}=“b”, which is the output of the MPC algorithm but for $x^{n_k+3}_{n_k+2}$=“ee” and t^{k+1}_1. Formally, in the Passing step: p_{val}=$P(t_{k+1}$=“b”$|t^k_1)P(x^{n_k+3}_{n_k+2}$=“$ee$”$|t^k_1, t_{k+1}$=“$b$”). We continue the procedure until meeting the base case, where the string must be a prefix of the next token (usually, when there is only a single character left). Finally, by computing the sum of the branch and pass steps, we obtain the desired conditional probability b_{val}+p_{val}=$P(x^{n_k+3}_{n_k+1}$=“bee”$|t^k_1)$.

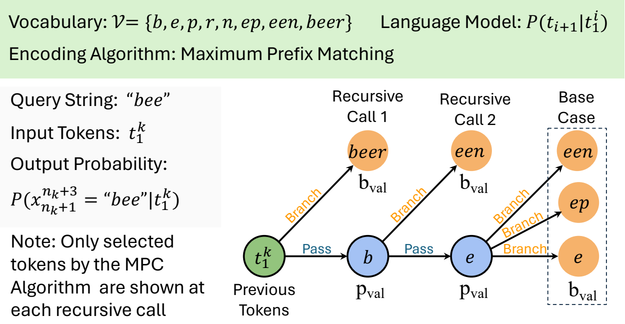


Figure 2: MPC Visualization. At each recursive call, the Branch step finds tokens that starts with the query string while the Pass step extracts and employs the next token and leftover string for the next recursive call until meeting the base case.

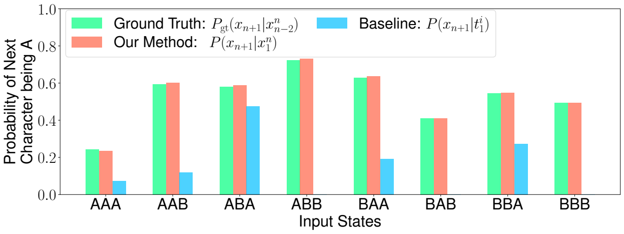


Figure 3: Our method accurately estimates the transition probability of a 3rd order Markov chain while the baseline method fails to.

4. Experiments

We validate our method on a 3rd order Markov chain experiment with $\mathcal{A}$={“A”,“B”}, where we randomly construct the transition matrix and the vocabulary $\mathcal{V}$={“A”, “B”, “AA”, “$BAAB$”, “$BBAA$”, “$BBBA$”, “BA”, “BBA”}. We train a LM model using GPT-2 architecture with 6 hidden layers. Since the model is agnostic to the Markov chain order, we average the probability from 100 runs on different context length while fixing the last 3 characters. We compare our method with the baseline estimator $P(x_{n+1}|t^i_1)$, equivalent to one Branch step in the MPC algorithm. Figure 3 shows the results where the baseline method exhibits significant sampling bias due to tokenization. Following Proposition 2.3, one can clarify the zero probability events output from the baseline estimator. Our method, in contrast, accurately estimates the ground truth probability used to generate the data, showing that it is possible to recover the implicitly learned character information from the tokenized LMs.

5. Conclusion

This work identifies the next-character sampling gap between a tokenized model and a token-free one, which persists even for optimally trained models. We present a probabilistic approach to effectively eliminate this bias without requiring additional training. This closes the sampling gap between tokenized and token-free models, suggesting that language models implicitly absorb character-level information despite being trained solely on tokenized text. This result implies that it is theoretically possible to simulate the behavior of another language model trained using different vocabulary without any fine-tuning, since it is possible to transfer from token-free models to tokenized counterparts.

References

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Cao, K. and Rimell, L. You should evaluate your language model on marginal likelihood over tokenisations. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 2104–2114, 2021.

Chen, Y., Marchisio, K., Raileanu, R., Adelani, D., Saito Stenetorp, P. L. E., Riedel, S., and Artetxe, M. Improving language plasticity via pretraining with active forgetting. *Advances in Neural Information Processing Systems*, 36:31543–31557, 2023.

Chirkova, N., Kruszewski, G., Rozen, J., and Dymetman, M. Should you marginalize over possible tokenizations? In *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

Cleary, J. and Witten, I. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.

Cognetta, M., Zouhar, V., Moon, S., and Okazaki, N. Two counterexamples to\textit {Tokenization and the Noiseless Channel}. *arXiv preprint arXiv:2402.14614*, 2024.

Dagan, G., Synnaeve, G., and Rozière, B. Getting the most out of your tokenizer for pre-training and domain adaptation. *arXiv preprint arXiv:2402.01035*, 2024.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Gallé, M. Investigating the effectiveness of bpe: The power of shorter sequences. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 1375–1381, 2019.

Gemini, T. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Goldman, O., Caciularu, A., Eyal, M., Cao, K., Szpektor, I., and Tsarfaty, R. Unpacking tokenization: Evaluating text compression and its correlation with model performance. *arXiv preprint arXiv:2403.06265*, 2024.

Goyal, S., Ji, Z., Rawat, A. S., Menon, A. K., Kumar, S., and Nagarajan, V. Think before you speak: Training language models with pause tokens. In *The Twelfth International Conference on Learning Representations*, 2023.

guidance ai. Guidance ai, 2023. URL https://github.com/guidance-ai/guidance. GitHub repository.

Gutierrez-Vasques, X., Bentz, C., and Samardžić, T. Languages through the looking glass of bpe compression. *Computational Linguistics*, 49(4):943–1001, 2023.

Kudo, T. and Richardson, J. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

Liu, S., Deng, N., Sabour, S., Jia, Y., Huang, M., and Mihalcea, R. Task-adaptive tokenization: Enhancing long-form text generation efficacy in mental health and beyond. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023a.

Liu, Y., Lin, P., Wang, M., and Schütze, H. Ofa: A framework of initializing unseen subword embeddings for efficient large-scale multilingual continued pretraining. *arXiv preprint arXiv:2311.08849*, 2023b.

Makkuva, A. V., Bondaschi, M., Girish, A., Nagle, A., Jaggi, M., Kim, H., and Gastpar, M. Attention with markov: A framework for principled analysis of transformers via markov chains. *arXiv preprint arXiv:2402.04161*, 2024.

Minixhofer, B., Ponti, E. M., and Vulić, I. Zero-shot tokenizer transfer. *arXiv preprint arXiv:2405.07883*, 2024.

Nawrot, P., Chorowski, J., Łańcucki, A., and Ponti, E. M. Efficient transformers with dynamic token pooling. *arXiv preprint arXiv:2211.09761*, 2022.

Petrov, A., La Malfa, E., Torr, P., and Bibi, A. Language model tokenizers introduce unfairness between languages. *Advances in Neural Information Processing Systems*, 36, 2024.

Provilkov, I., Emelianenko, D., and Voita, E. Bpe-dropout: Simple and effective subword regularization. *arXiv preprint arXiv:1910.13267*, 2019.

Rajaraman, N., Jiao, J., and Ramchandran, K. Toward a theory of tokenization in llms. *arXiv preprint arXiv:2404.08335*, 2024.

Schmidt, C. W., Reddy, V., Zhang, H., Alameddine, A., Uzan, O., Pinter, Y., and Tanner, C. Tokenization is more than compression. *arXiv preprint arXiv:2402.18376*, 2024.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Singh, A. K. and Strouse, D. Tokenization counts: the impact of tokenization on arithmetic in frontier llms. *arXiv preprint arXiv:2402.14903*, 2024.

Song, X., Salcianu, A., Song, Y., Dopson, D., and Zhou, D. Fast wordpiece tokenization. *arXiv preprint arXiv:2012.15524*, 2020.

Tay, Y., Tran, V. Q., Ruder, S., Gupta, J., Chung, H. W., Bahri, D., Qin, Z., Baumgartner, S., Yu, C., and Metzler, D. Charformer: Fast character transformers via gradient-based subword tokenization. In *International Conference on Learning Representations*, 2021.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Willems, F. M., Shtarkov, Y. M., and Tjalkens, T. J. The context-tree weighting method: Basic properties. *IEEE transactions on information theory*, 41(3):653–664, 1995.

Xue, L., Barua, A., Constant, N., Al-Rfou, R., Narang, S., Kale, M., Roberts, A., and Raffel, C. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022.

Yu, L., Simig, D., Flaherty, C., Aghajanyan, A., Zettlemoyer, L., and Lewis, M. Megabyte: Predicting million-byte sequences with multiscale transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

Zouhar, V., Meister, C., Gastaldi, J., Du, L., Sachan, M., and Cotterell, R. Tokenization and the noiseless channel. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5184–5207, 2023.

A. Related Work

Theory of Tokenization. Existing works on tokenization generally support the idea that compressing tokens enhances model performance (Gallé, 2019; Gutierrez-Vasques et al., 2023; Zouhar et al., 2023). However, these emperically findings are in conflicted with other later studies Cognetta et al. (2024); Schmidt et al. (2024). On the theoretical side, Rajaraman et al. (2024) examined tokenization through the lens of unigram models, motivated by the observation made by Makkuva et al. (2024) that transformers struggles to learn 2nd-order Markov chains. We, however, do not observe this phenomenon in our experiment. As such, our work on bias due to tokenization is not affected by their observation.

Tokenization and Perplexity. Our work relates to the statistical evaluation of LMs, where we provide an algorithm to directly evaluate the character-level perplexity $p(x_{n+1}^N|x_1^n)$, using a tokenized LM. In terms of token-level perplexity evaluation, some recent studies (Cao & Rimell, 2021; Chirkova et al., 2023) have suggested using stochastic tokenization (Provilkov et al., 2019) at test time to evaluate perplexity scores of LMs ($p(t_1^i)$). However, these evaluations were done on LMs trained with deterministic tokenization which could be suboptimal as demonstrated by our examination of undefined states in Section 2. As such, by utilizing our approach, one can obtain a much more accurate insights on LMs evaluation.

Related Algorithms. Our algorithm is inspired from the literature of universal compression such as prediction by partial matching (Cleary & Witten, 1984) and context-tree weighting (Willems et al., 1995), which have been applied for text prediction but for much simpler settings without any tokenization involved. Recently, Minixhofer et al. (2024); Liu et al. (2023a) propose tokenization adaptation methods, which still requires a heuristic optimization that complicates the training pipeline. Some recent studies have proposed method to target the problem of language models encountering difficulties generating text near prompt boundaries (Dagan et al., 2024; guidance ai, 2023), which bears some resemblance to our proposed algorithm. These methods, however, are heuristic and only applicable to certain scenarios. On the other hand, our bias removal algorithm is theoretically correct, versatile for various situations, and enables conversion between token-free and tokenized LMs due to its accurate representation of conditional sampling distributions.

B. Supporting Theorems on Maximum Prefix Encoding

This section provides supporting theorems for the proof of the main results. We first remind the readers that the set $\mathcal{S}(x_1^n)$ corresponds to the set of all strings that contain x_1^n as a prefix. Similarly, the event set $\mathcal{S}(t_1^i)$ corresponds to the set of all strings whose first i tokens are t_1^i. Consider when $t_1^i = \text{encode}(x_1^n)$, it should be noted that the two sets $S(t_1^i)$ and $S(x_1^n)$ are not guaranteed to be equivalent. That is because the subsequent characters after x_1^n can affect the tokenization within the first n character. We illustrate this in more detail in the following example.

Example. Consider the Markov chain example in Section 2, where $\mathcal{V} = \{\text{“}AA\text{”}, \text{“}A\text{”}, \text{“}B\text{”}\}$. Then, the string $s_1 =$ “$AABAABAB$”, then $s_1 \in \mathcal{S}(x_1 = \text{“}A\text{”})$ and $s_1 \in \mathcal{S}(t_1 = \text{“}AA\text{”})$ since the first character of s_1 is “A” and the first token of s_1 is “AA”. On the other hand, $s_1 \notin \mathcal{S}(t_1 = \text{encode}(x_1) = \text{“}A\text{”})$ since its first token is “AA”, not “A”.

We introduce the Proposition B.1 that contains two facts regarding the MPE process, visually presented in Figure 4.

Proposition B.1. *Let s be a string with the prefix x_1^n ($x_1^n \in \text{prefix}(s)$). Define the minimal superstring r to be the prefix of s with the fewest tokens that contains x_1^n as a prefix: $r = \text{argmin}_r(k|t_1^k = \text{encode}(r) \wedge x_1^n \in \text{prefix}(r) \wedge r \in \text{prefix}(s))$. Then, we have the followings:*

1. *For $1 \leq i < k$, $\text{encode}(s)_i = \text{encode}(x_1^n)_i$. Furthermore, when $r = x_1^n$, we also have $\text{encode}(s)_k = \text{encode}(x_1^n)_k$.*
2. *Let ℓ be the number of tokens in $\text{encode}(x_1^n)$, then we have $\text{decode}(\text{encode}(x_1^n)_k^\ell) \in \text{prefix}(\text{decode}(\text{encode}(s)_k))$.*

Proof. (Result 1.) Proof by contradiction. Let s be the counter-example with the fewest number of tokens. Assume that for $1 \leq i < k$, $\text{encode}(s)_i \neq \text{encode}(x_1^n)_i$. Let j be the smallest of such i.

Consider $\text{encode}(s)_j$ and $\text{encode}(x_1^n)_j$.

- Case 1: $|\text{decode}(\text{encode}(s)_1^j)| < |x_1^n|$.
  - Case 1.a: $|\text{decode}(\text{encode}(s)_j)| < |\text{decode}(\text{encode}(x_1^n)_j)|$. This leads to a contradiction, since x_1^n is a prefix of s, therefore a longest prefix matching algorithm would always generate the longer token ($\text{encode}(x_1^n)_j$) over the shorter one ($\text{encode}(s)_j$) when it is available.
  - Case 1.b: $|\text{decode}(\text{encode}(s)_j)| > |\text{decode}(\text{encode}(x_1^n)_j)|$. This leads to a contradiction, since $\text{concat}(\text{encode}(s)_1^j)$

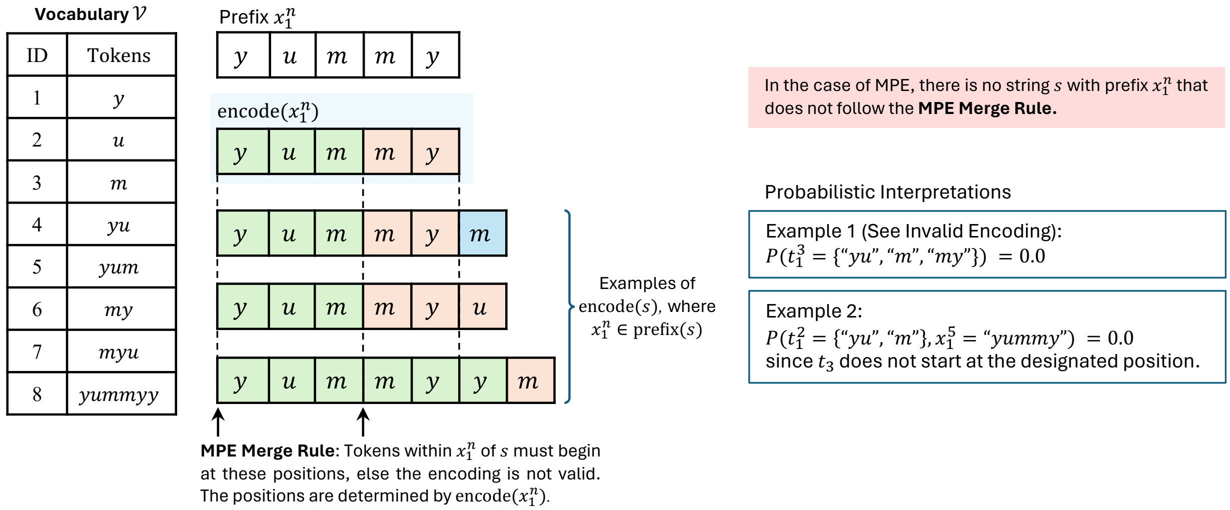

Vocabulary $\mathcal{V}$
ID
Tokens
1 y
2 u
3 m
4 yu
5 yum
6 my
7 myu
8 $yummyy$
Prefix x_1^n
y u m m y
encode(x_1^n)
y u m m y
y u m m y m
y u m m y u
y u m m y y m
Examples of encode(s), where $x_1^n \in$ prefix(s)
MPE Merge Rule: Tokens within x_1^n of s must begin at these positions, else the encoding is not valid. The positions are determined by encode(x_1^n).
In the case of MPE, there is no string s with prefix x_1^n that does not follow the **MPE Merge Rule.**
Probabilistic Interpretations
Example 1 (See Invalid Encoding):
$P(t_1^3 = \{\text{"}yu\text{"}, \text{"}m\text{"}, \text{"}my\text{"}\}) = 0.0$
Example 2:
$P(t_1^2 = \{\text{"}yu\text{"}, \text{"}m\text{"}\}, x_1^5 = \text{"}yummy\text{"}) = 0.0$
since t_3 does not start at the designated position.

Figure 4: Interpretations of Proposition B.1, which shows that for any string s with prefix x_1^n, its token within x_1^n must start at a certain designated positions. For each encoding, same color denotes belonging to the same token. This would later allows us to construct an efficient algorithm to correct the bias. See Corollary B.3 for details on invalid encoding.

is a prefix of x_1^n (Case 1 assumption), therefore a longest prefix matching algorithm would always generate the longer token ($\text{encode}(s)_j$) over the shorter one ($\text{encode}(x_1^n)_j$) when it is available.

- Case 1.c: $|\text{decode}(\text{encode}(s)_j)| = |\text{decode}(\text{encode}(x_1^n)_j)|$. This means that the two tokens are the same, contradicting our initial assumption.

- Case 2: $|\text{decode}(\text{encode}(s)_1^j)| \geq |x_1^n|$. In this case, $r = \text{decode}(\text{encode}(s)_1^j)$ is a superstring of x_1^n implying that k is at most j, which contradicts our initial assumption that $1 \leq j < k$.

Finally, in the case $r = x_1^n$, this means $\text{decode}(\text{encode}(s)_k)$ is a suffix of x_1^n. Since all the tokens before k within x_1^n has been matched, i.e. $\text{encode}(s)_i = \text{encode}(x_1^n)_i$ for $1 \leq i < k$, the last token must also match as the result (else, $|r| \neq |x_1^n|$, leads to contradiction), we have $\text{encode}(s)_k = \text{encode}(x_1^n)_k$.

(Result 2.) The proof idea is that since r contains x_1^n and any tokens within r and x_1^n has been matched up to $k-1$, then what is left in x_1^n must be in the last token in r (which is the kth token of r). Formally, following Result 1, we have $\text{decode}(\text{encode}(x_1^n)_1^{k-1}) = \text{decode}(\text{encode}(s)_1^{k-1})$. Since r has k tokens in total and $x_1^n \in \text{prefix}(r)$, this means that $\text{decode}(\text{encode}(s)_k)$ must cover the rest of x_1^n, i.e. $\text{decode}(\text{encode}(x_1^n)_k^\ell)$. As the result, we must have $\text{decode}(\text{encode}(x_1^n)_k^\ell) \in \text{prefix}(\text{decode}(\text{encode}(s)_k))$. □

We remind the reader the definition of invalid encoding below.

Definition B.2. (Invalid Encodings) The list of tokens (an encoding) t_1^k is invalid if $\text{encode}(\text{decode}(t_1^k)) \neq t_1^k$. Otherwise, it is a valid encoding.

Corollary B.3. *$\mathcal{S}(t_1^k) = \emptyset$ if and only if t_1^k is invalid.*

Proof. We prove each direction as follow.

- If $\mathcal{S}(t_1^k) = \emptyset$ then t_1^k is invalid: Since $\mathcal{S}(t_1^k) = \emptyset$, we know that there exist no string s such that $\text{encode}(s)_1^k = t_1^k$. As such, for $s = \text{decode}(t_1^k)$, we do not have $\text{encode}(\text{decode}(t_1^k)) = t_1^k$, which proves the result.

- If t_1^k is invalid then $\mathcal{S}(t_1^k) = \emptyset$: Let $x_1^n = \text{decode}(t_1^k)$ and $t_1^i = \text{encode}(x_1^n)$. Let $s_1 \in \mathcal{S}(t_1^i)$ and suppose there exist a string $s_2 \in \mathcal{S}(t_1^k)$. Re-running the MPE procedure on s_1 and s_2 in parallel, then every time a token is selected within x_1^n in s_1, it must also be selected at the same position in s_2 as well. Thus, we cannot have $t_1^i \neq t_1^k$, which proves the result.

□

C. Proof of Proposition 2.3 in the Main Paper

Proposition 2.3 *(Token-Induced Zero Probability) Let t_1^i be a sequence of input tokens. For any invalid encoding t_1^i, we have $P_{\text{gt}}(t_1^i)$=0.0 and the conditional probability $P_{\text{gt}}(t_{i+1}|t_1^i)$ is undefined. In the case t_1^i is valid, then $P_{\text{gt}}(t_{i+1}|t_1^i)$=0.0 if t_1^{i+1} is invalid. Furthermore, let x_1^n=decode(t_1^i), then for any string x_{n+1}^N such that* $\text{encode}(\text{concat}(\text{decode}(t_1^i), x_{n+1}^N)) \neq t_1^i$, *we have $P_{\text{gt}}(x_{n+1}^N|t_1^i)$=0.0.*

Proof. For the first two statements, we have:

- For an invalid t_1^i where $t_1^i \neq \text{encode}(\text{decode}(t_1^i))$, we have $\mathcal{S}(t_1^i) = \emptyset$, as implied by Corollary B.3. As such, we have $P_{\text{gt}}(t_1^i)$=0.0 which leads $P_{\text{gt}}(t_{i+1}|t_1^i)$ to be an undefined conditional probability .
- For a valid t_1^i but invalid t_1^{i+1}, we know that $P_{\text{gt}}(t_1^{i+1})$=0.0, which results in $P_{\text{gt}}(t_{i+1}|t_1^i) = 0.0$.

For the last statement, we first note the following:

1. Note that $P_{\text{gt}}(x_{n+1}^N, t_1^i) = P_{\text{gt}}(x_1^N, t_1^i)$ where $\text{concat}(x_1^n, x_{n+1}^N) = x_1^N$.
2. Consider $P_{\text{gt}}(x_1^N, t_1^i) = P_{\text{gt}}(x_1^N)P_{\text{gt}}(t_1^i|x_1^N)$, we will prove that $P_{\text{gt}}(t_1^i|x_1^N) = 0.0$ if $\text{encode}(x_1^N)_1^i \neq t_1^i$.

The proof idea for this is shown in Figure 4 (Example 2, Right). Formally:

- Let j be the first position such that $\text{encode}(x_1^N)_j \neq t_j$ then we know that $|\text{decode}(\text{encode}(x_1^N)_j))| > |\text{decode}(t_j)|$ (Proposition B.1 (Result 2)).
- Following Proposition B.1 (Result 2), let $s \in \mathcal{S}(x_1^N)$, then we know that $\text{decode}(\text{encode}(x_1^N)_j)$ must be a substring of within another longer token (it cannot be broken down) in s. Hence, no string s will have a j-th token as t_j, so $P_{\text{gt}}(t_1^i|x_1^N) = 0.0$. This completes the proof.

Finally, we note that $P_{\text{gt}}(t_1^i) = 0.0$ does not implies $\text{encode}(\text{decode}(t_1^i)) \in \mathcal{V}$, since it can be due to the original distribution on the character domain. A classic example for this is a Markov model with an absorption state. □

D. Proof of Proposition 3.1 and Corollary 3.2 in the Main Paper

Proposition 3.1 *Let $s^* = x_1^n$, where $t_1^i = \text{encode}(s^*) = \text{encode}(x_1^n)$. Then we have $\mathcal{S}(t_1^i) \subset \mathcal{S}(x_1^n)$, i.e. for any string s where $t_1^i = \text{encode}(s)_1^i$, we have $P(x_1^n|t_1^i) = 1.0$. In the case $t_i \in \mathcal{V}^*$, then we also have that $\mathcal{S}(t_1^i) = \mathcal{S}(x_1^n)$, i.e. any string s where $x_1^n \in \text{prefix}(s)$ must have the first i tokens as t_1^i and $P(t_1^i|x_1^n) = 1.0$.*

Proof. We prove each case as follow.

1) *General Case*: There exists a string $s \in \mathcal{S}(x_1^n)$ where $\text{encode}(s)_1^i \neq t_1^i$, following directly from our 1st order Markov chain example in the main paper, i.e. the string s = "AA" has "A" as prefix but have the t_1 = "AA" ≠ "A". Also, any string s that has the first i tokens as t_1^i must have the first n characters as x_1^n, hence $\mathcal{S}(t_1^i) \subset \mathcal{S}(x_1^n)$ and $P(x_1^n|t_1^i) = 1.0$.

2) $t_i \in \mathcal{V}^*$: The proof idea is that, since t_i cannot be a part of any token in $\mathcal{V}$, it is impossible to merge it by appending additional characters after t_i. Formally, similar to Proposition B.1:

- For any string $s \in \mathcal{S}(\text{decode}(t_1^i))$, let ℓ be the number of tokens in the minimal superstring r of s that contains x_1^n as a prefix.
- Following Proposition B.1 (Result 2), we know that t_i must be a substring of $\text{decode}(\text{encode}(s)_\ell)$.
- Due to $t_i \in \mathcal{V}^*$, then $t_i = \text{encode}(s)_\ell$. We also know from Proposition B.1 (Result 1) that $\text{encode}(s)_i = t_i$ for $1 \leq i < \ell$, this means that $\ell = i$. This gives us $t_1^i = \text{encode}(s)_1^i$ and $P(t_1^i|x_1^n) = 1.0$.

This completes the proof. □

Remarks. We briefly note that the condition $t_i \in \mathcal{V}^*$ is the sufficient condition. In general, any token sequence t_1^i that satisfies $\mathcal{S}(t_1^i) = \mathcal{S}(x_1^n)$ will have $P(t_1^i|x_1^n) = 1.0$. One potential strategy is to find the first index $i = 0, 1, ...k - 1$ such that t_{k-i}^k cannot be merged into another token in $\mathcal{V}$.

Corollary 3.2 *Following Proposition 3.1, suppose $t_i \in \mathcal{V}^*$ then we have $P(x_{n+1}^N|x_1^n){=}P(x_{n+1}^N|t_1^i)$. Similarly, we also have $P(t_{i+1}^j|x_1^n){=}P(t_{i+1}^j|t_1^i)$.*

Proof. For the first case, we prove through the following equations:

$$\begin{aligned} P(x_{n+1}^N|t_1^i) &= P(x_{n+1}^N|t_1^i, x_1^n) && (4)\\ &= \frac{P(x_{n+1}^N, t_1^i|x_1^n)}{P(t_1^i|x_1^n)} && (5)\\ &= \frac{P(x_{n+1}^N|x_1^n)P(t_1^i|x_1^n, x_{n+1}^N)}{P(t_1^i|x_1^n)} && (6)\\ &= P(x_{n+1}^N|x_1^n) && (7) \end{aligned}$$

where the first equality is due to $P(x_1^n|t_1^i) = 1.0$ and the last equality is due to $P(t_1^i|x_1^n) = 1.0$ for $t_i \in \mathcal{V}^*$.

Similarly, for the second case, we have:

$$\begin{aligned} P(t_{i+1}^j|t_1^i) &= P(t_{i+1}^j|x_1^n, t_1^i) && (8)\\ &= \frac{P(t_{i+1}^j|x_1^n)P(t_1^i|x_1^n, t_{i+1}^j)}{P(t_1^i|x_1^n)} && (9)\\ &= P(t_{i+1}^j|x_1^n), && (10) \end{aligned}$$

which completes the proof. □

E. Proof for The Bias Removal Method

E.1. Refactoring

Our goal is to express the quantity $P(x_{n+1}^N|x_1^n)$ using the tokenized LM that outputs the conditional probability $P(t_i|t_1^{i-1})$. Let $x_1^{n_k} \in \text{prefix}(x_1^n)$ where $t_1^k = \text{encode}(x_1^{n_k})$ and $t_k \in \mathcal{V}^*$. Following Proposition 3.1, any string s with prefix $x_1^{n_k}$ must have the first k tokens as t_1^k. We now perform the following factorization:

$$\begin{aligned} P(x_{n+1}^N|x_1^n) &= P(x_{n+1}^N|x_1^{n_k}, x_{n_k+1}^n) && (11)\\ &= \frac{P(x_{n+1}^N, x_{n_k+1}^n|x_1^{n_k})}{P(x_{n_k+1}^n|x_1^{n_k})} && (12)\\ &= \frac{P(x_{n_k+1}^N|x_1^{n_k})}{P(x_{n_k+1}^n|x_1^{n_k})} && (13)\\ &= \frac{P(x_{n_k+1}^N|t_1^k)}{P(x_{n_k+1}^n|t_1^k)}, && (14) \end{aligned}$$

where the last inequality is due to Corollary 3.2. Finally, we will use the Maximum Prefix Correction (MPE) Algorithm to compute each term in (14) individually. Note that the algorithm does not require $t_k \in \mathcal{V}^*$. Here, we explicitly highlight the importance of having $t_k \in \mathcal{V}^*$, as it bridges between the character and token domain through Equation (14).

E.2. Maximum Prefix Correction Algorithm

Overview. The MPC algorithm computes $P(x_{n_k+1}^N|t_1^k)$. Note that we do not require $t_k \in \mathcal{V}^*$ in the MPC algorithm. Using marginalization, we have the following:

$$\begin{aligned} P(x_{n_k+1}^N|t_1^k) &= \sum_{t \in \mathcal{V}} P(x_{n_k+1}^N, t_{k+1} = t|t_1^k) && (15)\\ &= \underbrace{\sum_{t \in \mathcal{T}_{\mathrm{b_{val}}}} P(x_{n_k+1}^N, t_{k+1} = t|t_1^k)}_{\mathrm{b_{val}}} + \underbrace{\sum_{t \in \mathcal{T}_{\mathrm{p_{val}}}} P(x_{n_k+1}^N, t_{k+1} = t|t_1^k)}_{\mathrm{p_{val}}} && (16) \end{aligned}$$

where:

- $\mathcal{T}_{\mathrm{b}_{\mathrm{val}}} = \{t \in \mathcal{V} | x^N_{n_k+1} \in \mathrm{prefix}(\mathrm{decode}(t))\}$ is the set of tokens that have a prefix $x^N_{n_k+1}$.

- $\mathcal{T}_{\mathrm{p}_{\mathrm{val}}} = \{t \in \mathcal{V} | x^N_{n_k+1} \notin \mathrm{prefix}(\mathrm{decode}(t))\}$ is the ones that do not.

and $\mathcal{T}_{\mathrm{b}_{\mathrm{val}}} \cap \mathcal{T}_{\mathrm{p}_{\mathrm{val}}} = \emptyset$.

Branch Step. Here, $\mathrm{b}_{\mathrm{val}}$ is the probability that, given the list of previous tokens t^k_1, the next token of the string s has $x^N_{n_k+1}$ as a prefix. To compute this term, we obtain $P(t_{k+1} = t | t^k_1)$ for all $t \in \mathcal{V}$ using one model run, then sum the probabilities corresponds to all tokens whose prefix is $x^N_{n_k+1}$.

$$\mathrm{b}_{\mathrm{val}} = \sum_{t \in \mathcal{T}_{\mathrm{b}_{\mathrm{val}}}} P(t_{k+1} = t | t^k_1), \tag{17}$$

Proof. To see this, for each summand of $\mathrm{b}_{\mathrm{val}}$ in Eq.(16), we have:

$$\begin{aligned} P(x^N_{n_k+1}, t_{k+1} = t | t^k_1) &= P(t_{k+1} = t | t^k_1) \times P(x^N_{n_k+1} | t^k_1, t_{k+1}{=}t) \quad (18) \\ &= P(t_{k+1} = t | t^k_1), \quad (19) \end{aligned}$$

where $P(x^N_{n_k+1} | t^k_1, t_{k+1}{=}t) = 1.0$ is due to $x^N_{n_k+1} \in \mathrm{prefix}(t)$. This concludes the proof. □

Pass Step. Here, $\mathrm{p}_{\mathrm{val}}$ is the probability that, given the list of previous tokens t^k_1, the subsequent string $x^N_{n_k+1}$ is **not** a prefix of the next token. Under the MPE, we compute the value $\mathrm{p}_{\mathrm{val}}$ as follow:

$$\mathrm{p}_{\mathrm{val}} = P(t_{k+1} = t | t^k_1) \times P(x^N_{n_{k+1}+1} | t^k_1, t_{k+1} = t), \tag{20}$$

where $t = \mathrm{encode}(x^N_{n_k+1})_1$ and $x^{n_{k+1}}_{n_k+1} = \mathrm{decode}(t)$. That is, during the passing step, there are two subroutines:

1. Extract the next token t within $x^N_{n_k+1}$ and compute $P(t_{k+1} = t | t^k_1)$. If $x^N_{n_k+1} = \mathrm{decode}(t)$, then returns 0.0 since this is not allowed according to the condition required in $\mathcal{T}_{\mathrm{p}_{\mathrm{val}}}$.

2. Recursively compute $P(x^N_{n_{k+1}+1} | t^k_1, t_{k+1} = t)$.

Proof. Following Proposition 2.3 for invalid encodings, we only need to consider t such that t^{k+1}_1 is valid. Under Proposition B.1 for MPE on x^N_1, only first token of $\mathrm{encode}(x^N_{n_k+1})$ is allowed (also see Example 2 in Figure 4(Right)). Finally, applying the chain rule of probability, we obtain Equation 20. For the case of non-optimal LM, see Section G.2 for non-optimal LM. This completes the proof. □

Base Case. We note that the base case of our algorithm corresponds to the situation where $x^N_{n_k+1} = \mathrm{decode}(t)$. In this scenario, we only needs to compute $\mathrm{b}_{\mathrm{val}}$ (branching step) while $\mathrm{p}_{\mathrm{val}} = 0.0$.

Complexity Analysis. The complexity of our algorithm (number of inferences on the language model) scales with the length of the the query string, i.e. $N - n_k$. Note that the complexity of the summation at the Branching step is relatively cheap compared to the runtime of the language model.

F. Converting Token-Free Language Model to Tokenized Language Model for MPE.

We introduce an algorithm to compute $P(t_{k+1} | t^k_1)$ using a token-free language model $P(x^N_{n+1} | x^n_1)$, despite having no access to any tokenized LM. This approach enables theoretical conversion of a token-free model to a tokenized one. The method involves two stages. First, we refactor the conditional probability similar to the technique presented in Section E. Next, we aggregate the probabilities of all possible strings leading to the desired tokenization. It is important to note that a Markov chain is a special type of autoregressive model, meaning this method can be employed to effortlessly calculate Markov chain transition matrices within the tokenized domain.

F.1. Refactoring

Consider the probability $P(t_{i+1}|t_1^i)$ that we would like to expressed using $P(x_{n+1}^N|x_1^n)$. Let t_k be the last token within t_1^i such that $t_k \in \mathcal{V}^*$. We now perform the following factorization:

$$P(t_{i+1}|t_1^i) = \frac{P(t_{k+1}^{i+1}|t_1^k)}{P(t_{k+1}^{i}|t_1^k)} \tag{21}$$

$$= \frac{P(t_{k+1}^{i+1}|x_1^{n_k})}{P(t_{k+1}^{i}|x_1^{n_k})}, \tag{22}$$

where $x_1^{n_k} = \text{decode}(t_1^k)$. The second equality is due to Corollary 3.2. Each term can then be computed using the aggregation procedure shown next.

F.2. Aggregation.

In this step, we would like to compute $P(t_{k+1}^i|x_1^{n_k})$ where $\text{encode}(x_1^{n_k}) = t_1^k$ and $t_k \in \mathcal{V}^*$, using the token-free representation $P(x_{n+1}|x_1^n)$. Here, we denote $\text{decode}(t_{k+1}^i) = x_{n_k+1}^{n_i}$ and $M = \max_{t\in\mathcal{V}} |\text{decode}(t)|$ be the length of the longest token in V and $\Omega = \mathcal{A}^M$ is the enumeration of all string of length M.

Computing $P(t_{k+1}^i|x_1^{n_k})$ involves considering all possible strings s with prefix $x_1^{n_i}$ and $t_{k+1}^i = \text{encode}(s)_{k+1}^i$. Although iterating through every possible string is infeasible, we can restrict our search by only examining strings with length $|s| = n_i + M$, as any additional string beyond this point will not impact the tokenization of prefix $x_1^{n_i}$due to M being the maximum token length. Formally, we will show that one can express $P(t_{k+1}^i|x_1^{n_k})$ as follows:

$$P(t_{k+1}^i|x_1^{n_k}) = \sum_{s'\in\mathcal{A}^M} P(x_{n_k+1}^{n_i+M} = c_1(s')|x_1^{n_k}) \mathbb{1}(t_{k+1}^i = \text{encode}(c_2(s'))_{k+1}^i), \tag{23}$$

where $c_1(s') := \text{concat}(x_{n_k+1}^{n_i}, s')$ and $c_2(s') := \text{concat}(x_1^{n_i}, s')$. The first term can be computed using the given token-free LM, i.e. $P(x_{n_k+1}^{n_i+M}|x_1^{n_k})$. The second term is an indicator function that checks whether $t_{k+1}^i = \text{encode}(s)_{k+1}^i$ and can be computed deterministically.

Proof. We have:

$$P(t_{k+1}^i|x_1^{n_k}) = P(t_{k+1}^i, x_{n_k+1}^{n_i}|x_1^{n_k}) \tag{24}$$

$$= \sum_{s'\in\mathcal{A}^M} P(t_{k+1}^i, x_{n_k+1}^{n_i+M} = c_1(s')|x_1^{n_k}) \tag{25}$$

$$= \sum_{s'\in\mathcal{A}^M} P(x_{n_k+1}^{n_i+M} = c_1(s')|x_1^{n_k}) P(t_{k+1}^i|x_1^{n_i+M} = c_2(s')) \tag{26}$$

$$= \sum_{s'\in\mathcal{A}^M} P(x_{n_k+1}^{n_i+M} = c_1(s')|x_1^{n_k}) \mathbb{1}(t_{k+1}^i = \text{encode}(c_2(s'))_{k+1}^i) \tag{27}$$

The rest is to prove the following equality:

$$P(t_{k+1}^i|x_1^{n_i+M} = c_2(s')) = \mathbb{1}(t_{k+1}^i = \text{encode}(c_2(s'))_{k+1}^i) \tag{28}$$

We first note that the first k tokens must be $t_1^k = \text{encode}(x_1^{n_k})$ due to our condition that $t_k \in \mathcal{V}^*$. Since M is the length of the longest token in $\mathcal{V}$, appending extra characters cannot change the tokenization happened for $x_1^{n_i}$. In other words, any string s with prefix $c_2(s')$ must have the same minimal superstring r containing $x_1^{n_i}$ (see Proposition B.1). We then apply this principle to the two cases:

- $t_{k+1}^i = \text{encode}(c_2(s'))_{k+1}^i$: In this case, we know that the string must contains the first i tokens as t_1^i, hence $P(t_{k+1}^i|x_1^{n_i+M} = c_2(s')) = 1.0$

- $t_{k+1}^i \neq \text{encode}(c_2(s'))_{k+1}^i$: In contrast, this case is equivalent to $P(t_{k+1}^i|x_1^{n_i+M} = c_2(s')) = 0.0$ since we are sure that the string do not contains the tokens t_{k+1}^i.

This concludes the proof. □

F.3. The Markov Chain Example.

We provide a detail computation of the Markov chain example in the main paper. Recall that in the original chain (in the character domain), we have the following:

$$P(x_2 = “A” | x_1 = “A”) = \alpha \tag{29}$$
$$P(x_2 = “B” | x_1 = “A”) = 1 - \alpha \tag{30}$$
$$P(x_2 = “A” | x_1 = “B”) = \beta \tag{31}$$
$$P(x_2 = “B” | x_1 = “B”) = 1 - \beta \tag{32}$$

We also assume the initial probability $\pi = \{\gamma, 1 - \gamma\}$ for “A” and “B” respectively. In the token domain, let first compute $P(t_2 = “A” | t_1 = “AA”)$, where we do not have to do the refactoring step since we know that $t_1 \in \mathcal{V}^*$. Following the Aggregation step, we have:

$$\begin{aligned} P(t_2 = “A” | t_1 = “AA”) &= P(x_3^6 = “ABA” | x_1^2 = “AA”) + P(x_3^6 = “ABB” | x_1^2 = “AA”) && (33) \\ &= P(x_3^5 = “AB” | x_1^2 = “AA”) && (34) \\ &= \alpha(1 - \alpha), && (35) \end{aligned}$$

where in the first equality, we do not include the case $x_3^6 = ”AAA”$ and $x_3^6 = ”AAB”$ since $\mathrm{encode}(”AAA”)_1 = “AA”$ and $\mathrm{encode}(”AAB”)_1 = “AA”$, which are not the token “A” that we are interested in. For other tokens and when $t_1 = “B”$, the computation follows the same arguments.

We now consider the case $P(t_2 = “B” | t_1 = “A”)$, we can refactor it as:

$$P(t_2 = “B” | t_1 = “A”) = \frac{P(t_2 = “B”, t_1 = “A”)}{P(t_1 = “A”)} \tag{36}$$

We first compute $P(t_1 = “A”)$ using the aggregation step:

$$\begin{aligned} P(t_1 = “A”) &= P(x_1^3 = “ABB”) + P(x_1^3 = “ABA”) && (37) \\ &= P(x_1^2 = “AB”) && (38) \\ &= \gamma(1 - \alpha), && (39) \end{aligned}$$

where we do again include the case $x_3^6 = ”AAA”$ and $x_3^6 = ”AAB”$ for the same reason above. For $P(t_2 = “A”, t_1 = “A”)$ we have:

$$\begin{aligned} P(t_2{=}“B”, t_1{=}“A”) &{=} P(x_1^4{=}“ABAA”) + P(x_1^4{=}“ABAB”) + P(x_1^4{=}“ABBA”) + P(x_1^4{=}“ABBB”) && (40) \\ &{=} P(x_1^2 = “AB”) && (41) \\ &{=} \gamma(1 - \alpha) && (42) \end{aligned}$$

which gives us $P(t_2 = “B” | t_1 = “A”) = 1.0$. Finally, in this specific case, since order of the Markov chain in the character domain is 1, we do not need to consider the higher order of the Markov chain in the token domain.

G. On Predictive Distribution of Language Models

In practice, LMs often do not follow Proposition 2.3 due to softmax activations. As such, in our MPC algorithm, when $t \in \mathcal{T}_{\mathrm{pval}}$ and $t \neq \mathrm{encode}(x_{n_k+1}^N)_1$, then $P_\theta(x_{n_k+1}^N, t_{k+1} = t | t_1^k)$ may not be 0.0 (where θ is the model weights). Eventually, this can potentially increase the complexity of our MPC algorithm during the Passing step.

In this section, we show that given any tokenized LM, we can force its output probabilities to obey Proposition 2.3, without any loss in terms of perplexity score on the token domain. This means that a tokenized LM satisfying Proposition 2.3 will guarantee the correctness of the Passing step in our MPC algorithm.

Finally, before going to the method, we remind the readers that Proposition 3.1 and Corollary 3.2 are factually correct and hold for all θ. As such, the refactoring step holds regardless.

G.1. Truncate-Renormalization Process

We justify the assumption that our tokenized language model $P_\theta(t_{i+1}|t_1^i)$ follows Proposition 2.3. The idea is that we can turn a language model that does not follow Proposition 2.3 to the one that does while guaranteeing that the new model will always result in a lower token-level perplexity score.

We first introduce Proposition G.1. In this proposition, we are given a target discrete probability distribution p where we know some of the values will not happen, says Φ^*. Assume that we have another distribution q that approximates p, then we can produce another distribution q^* that is closer to p in terms of KL divergence by setting corresponding probabilities of q in Φ^* to 0.0 and renormalize it (similar to rejection sampling).

Proposition G.1. *Given a discrete distribution $p = \{p_1, p_2, ..., p_m\}$ and $q = \{q_1, q_2, ..., q_m\}$ with $q_i > 0.0$ for all i. Let $\Phi = \{i \in \mathbb{Z} | p_i = 0.0\}$ and $\Phi^* \subseteq \Phi$, we define $q^* = \{q_1^*, q_2^*, ..., q_m^*\}$ where $q_i^* = 0.0$ for $i \in \Phi^*$, and $q_j^* = q_j / (\sum_{l \notin \Phi^*} q_l)$. Then we have:*

$$D_{\mathrm{KL}}(p||q^*) \leq D_{\mathrm{KL}}(p||q), \tag{43}$$

which implies that q^* is closer to p than q. We refer to the process of producing q^* as truncate-renormalization (TR).

Proof. Let $Z = (\sum_{l \notin \Phi} q_l)$ is the normalizing factor in q^*. Note that $Z \leq 1$ and as such $\log(Z) \leq 0$. Then:

$$D_{\mathrm{KL}}(p||q^*) = \sum_i p_i \log\left(\frac{p_i}{q_i^*}\right) \tag{44}$$

$$= \sum_{i \notin \Phi^*} p_i \log\left(\frac{p_i}{q_i^*}\right) \quad \text{, use } 0 \log 0 = 0.0 \tag{45}$$

$$= \sum_{i \notin \Phi^*} p_i \log\left(\frac{p_i}{q_i / Z}\right) \tag{46}$$

$$= \left[\sum_{i \notin \Phi^*} p_i \log\left(\frac{p_i}{q_i}\right)\right] + \log(Z) \tag{47}$$

$$\leq \sum_{i \notin \Phi^*} p_i \log\left(\frac{p_i}{q_i}\right) = D_{\mathrm{KL}}(p||q), \tag{48}$$

which completes the proof. □

Applying to our scenario, for any autoregressive language models $\hat{P}_\theta(t_{i+1}|t_1^i)$ that does not follow Proposition 2.3 (due to the softmax activations), we can perform the TR process (since we know which encoding is invalid) to obtain a new LM $P_\theta(t_{i+1}|t_1^i)$, which is guaranteed to better approximate the ground-truth model $P_{\mathrm{gt}}(t_{i+1}|t_1^i)$. Thus, we are guaranteed that the token-level perplexity score of $P_\theta(t_{i+1}|t_1^i)$ is always lower than or equal to $\hat{P}_\theta(t_{i+1}|t_1^i)$.

G.2. On Passing Step in Maximum Prefix Correction Algorithm.

Once our tokenized LM follows Proposition 2.3, it does not alternate the correctness of the Passing step. In other words, under Proposition 2.3, the LM will always output zero probability for invalid encodings t_1^k. As a result, the Passing step in the MPC algorithm remains the same in this case.

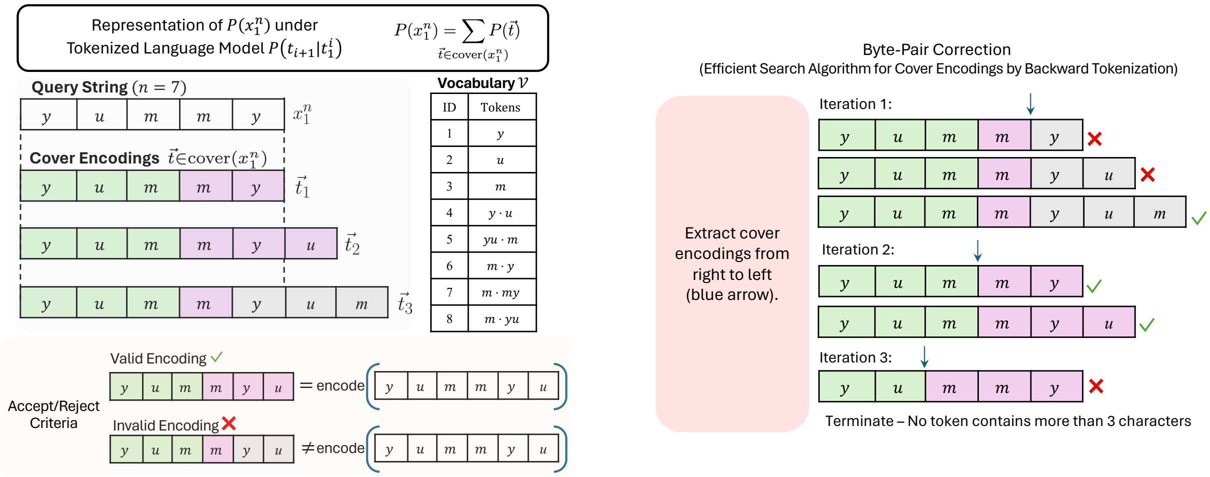
H. Algorithms for Byte Pair Encoding

H.1. Overview

We begin by introducing the Byte-Pair Correction (BPC) Algorithm for bias correction in Byte-Pair Encoding, which is more general than the MPC algorithm and also works for case of MPE. We then follow with a detail analysis to show the correctness of the algorithm.

Here, we introduce the definitions of invalid encodings (for BPE) and cover encodings.

Definition H.1. (Invalid Encodings) The list of tokens (an encoding) t_1^k is invalid if $\mathrm{encode}(\mathrm{decode}(t_1^k)) \neq t_1^k$. Otherwise, it is a valid encoding. We denote a valid t_1^k as $\mathrm{valid}(t_1^k)$.


Representation of $P(x_1^n)$ under
Tokenized Language Model $P(t_{i+1}|t_1^i)$
$P(x_1^n) = \sum_{\vec{t}\in\text{cover}(x_1^n)} P(\vec{t})$
Query String ($n = 7$)
y u m m y
x_1^n
Cover Encodings $\vec{t}\in\text{cover}(x_1^n)$
y u m m y
$\vec{t}_1$
y u m m y u
$\vec{t}_2$
y u m m y u m
$\vec{t}_3$
Vocabulary $\mathcal{V}$
Accept/Reject Criteria
Valid Encoding ✓
y u m m y u
= encode
Invalid Encoding ✗
y u m m y u
≠encode
y u m m y u
Byte-Pair Correction
(Efficient Search Algorithm for Cover Encodings by Backward Tokenization)
Extract cover encodings from right to left (blue arrow).
Iteration 1:
y u m m y ✗
y u m m y u ✗
y u m m y u m ✓
Iteration 2:
y u m m y ✓
y u m m y u ✓
Iteration 3:
y u m m y ✗
Terminate – No token contains more than 3 characters

ID	Tokens
1	y
2	u
3	m
4	$y \cdot u$
5	$yu \cdot m$
6	$m \cdot y$
7	$m \cdot my$
8	$m \cdot yu$

Figure 5: (Left) Representation of $P(x_1^n)$ using tokenized LM. We also show an example of cover encodings and valid/invalid encoding. (Right) Illustration of the Byte-Pair Correction Algorithm for BPE encoding. Green tick and red cross denotes valid and invalid encodings respectively, which can be checked using Definition H.1. The termination step does not appear in the original algorithm (for simplicity) but can be easily implemented.

Definition H.2. (Cover Encodings) Given a string x_1^n, an encoding t_1^k is said to be covering x_1^n when all the following conditions satisfied:

1. t_1^k is valid.

2. $x_1^n \in \text{prefix}(\text{decode}(t_1^k))$.

3. $x_i^n \in t_k$ for some $1 \leq i \leq n$, i.e. the last token t_i covers a part of the string x_1^n.

We denote $\text{cover}(x_1^n)$ to be the set of all cover encodings of x_1^n and $\vec{t} \in \text{cover}(x_1^n)$ is an encoding in $\text{cover}(x_1^n)$.

Having established these two definitions, we will later show that for BPE (and MPE), the probability $P(x_1^n)$ can be represented using a tokenized LM $P(t_{i+1}|t_1^i)$ as follows:

$$P(x_1^n) = \sum_{\vec{t}\in\text{cover}(x_1^n)} P(\vec{t}), \tag{49}$$

and the main goal of the BPC algorithm is to search through all cover encodings of x_1^n. We can then apply this algorithm and compute any conditional probability $P(x_{n+1}^N|x_1^n)$ through factorization. Figure 5 (Left) illustrates this with examples cover encodings and invalid/valid encodings.

H.2. Byte-Pair Correction Algorithm

For MPE, the MPC algorithm computes $P(x_1^n)$ by searching all possible valid encodings that cover x_1^n, where the probability of each encoding are computed using the LMs $P(t_{i+1}|t_1^i)$ through greedy search. However, this does not work for the case of BPE. For example, under the BPE encoding rule of Llama 2, the string "hello" is tokenized as an individual token while the string "hellow" is tokenized into 2 tokens "h and "ellow". Note that naive search for all tokens within $\mathcal{V}$ from left to right that cover x_1^n is computationally expensive.

The Byte-Pair Correction (BPC) algorithm, shown in Algorithm 2 and visualized in Figure 5 (right), which is an efficient algorithm that can search all valid encodings covering x_1^n. The idea is that, for each cover encoding $\vec{t}$, once the starting position of the last token is determined (say x_{i+1}), we are guaranteed the prior tokens is unique and must be $\text{encode}(x_1^i)$. Then one will accept the extracted $\vec{t}$ if it is valid, otherwise reject it. Corollary H.3 provides justification for this procedure. Here, we assume $P(\vec{t}) = 0.0$ for invalid $\vec{t}$, see Proposition H.4 and justifications as well as implementation in Appendix G.

Remark. The BPC algorithm can also be applied for the case of MPE. In fact, it is more general than the original MPC algorithm as it only relies on the property of invalid encodings.

Algorithm 2 Byte-Pair Correction Algorithm. This algorithm computes $P(x_1^n)$ by gradually reducing the search space.

```
procedure COMPUTE($x_1^N$)
    //Initialize $P(x_1^n)$:
    $P_{out} = 0.0$
    //Probability Aggregation
    for $i = n-1 ... 0$ do
        //The last token that partially covers $x_1^n$ begins with $x_{i+1}^n$
        $\mathcal{B} = \{t \in \mathcal{V} | x_{i+1}^n \in \text{prefix}(\text{decode}(t))\}$
        //Once the last token position is known, the remaining previous tokens can be determined
        $t_1^{k-1} = \text{encode}(x_1^i)$
        //Similar to the branching step in MPC
        $\text{b}_{\text{val}} = \sum_{t \in \mathcal{B}} P(t_k = t | t_1^{k-1})$
        $P_{out} = P_{out} + P(t_1^{k-1}) \times \text{b}_{\text{val}}$
    end for
return $P_{out}$
end procedure
```

H.3. Analysis

Notations. We extend the notation of the vocabulary $\mathcal{V}$ for the case of BPE. Here, $\mathcal{V}$ is an ordered list that determines the merging order in the BPE algorithm. Each $v \in \mathcal{V}$ is a triplet of $(t_{\text{left}}, t_{\text{right}}, t_{\text{new}})$ which corresponds to the merging tokens (left and right) and the new token. For simplicity, when we write $t \in \mathcal{V}$, it corresponds to the merged token, i.e. $t_{\text{new}} \in v$. Finally, the first $|\mathcal{A}|$ entries in $\mathcal{V}$ correspond to the alphabet $\mathcal{A}$, where no merging will happen.

Byte-Pair Encoding. We revise the encoding rule for BPE, shown in Algorithm 3. In practice, pre-tokenization is often used, where tokens are separated by whitespace or special characters. In this case, we can adjust our vocabulary $\mathcal{V}$ by removing tokens with special characters in the middle of the string.

Overview. We begin our analysis with theoretical results on invalid encodings (Corollary H.3 and Proposition H.4), which characterizes the zero probability events for an optimal tokenized LM. This will allow us to prove the representation of $P(x_1^n)$, i.e. Proposition H.5, previously shown in Equation (49). Finally, we conclude this section with the proof of correctness of the BPC algorithm, using Proposition H.5 and H.6.

We begin with theoretical results on invalid encodings in the case of BPE.

Corollary H.3. *$\mathcal{S}(t_1^k) = \emptyset$ if and only if t_1^k is invalid.*

Proof. We prove each direction as follows.

- If $\mathcal{S}(t_1^k) = \emptyset$ then t_1^k is invalid: Since $\mathcal{S}(t_1^k) = \emptyset$, we know that there exist no string s such that $\text{encode}(s)_1^k = t_1^k$. As such, for $s = \text{decode}(t_1^k)$, we do not have $\text{encode}(\text{decode}(t_1^k)) = t_1^k$, which proves the result.
- If t_1^k is invalid then $\mathcal{S}(t_1^k) = \emptyset$: Let $x_1^n = \text{decode}(t_1^k)$, we consider two string s_1 and s_2 that both have prefix x_1^n. Furthermore, we assume the first i tokens of s_1 covers exactly x_1^n, i.e. $x_1^n = \text{decode}(t_1^i)$ and similarly, the first j tokens of s_2 covers exactly x_1^n, i.e. $x_1^n = \text{decode}(t_1^j)$. Then:
  1. Proving invalid t_1^k leads to $\mathcal{S}(t_1^k) = \emptyset$ is equivalently to proving $t_1^i = t_1^j$ for any s_1, s_2.
  2. Re-running the BPE algorithm for s_1 and s_2 in parallel, we know that there will be no merge between any suffix of x_1^n and the rest of strings, i.e. $s_1 \backslash x_1^n$ and $s_2 \backslash x_1^n$ due to the condition above (See Algorithm 3, line 6).
  3. Furthermore, for s_1, any time a merge happens within x_1^n then the same merge must also happen within x_1^n for s_2 and vice versa.

  As the result, we have $t_1^i = t_1^j$ and they must be equal to $\text{encode}(x_1^n)$.

□

Proposition H.4. *(Token-Induced Zero Probability-BPE) Let t_1^i be a sequence of input tokens. For any invalid encoding t_1^i, we have $P_{\text{gt}}(t_1^i)$=0.0 and the conditional probability $P_{\text{gt}}(t_{i+1}|t_1^i)$ is undefined. In the case t_1^i is valid, then $P_{\text{gt}}(t_{i+1}|t_1^i)$=0.0 if t_1^{i+1} is invalid.*

Proof. The proof is the same as the MPE version (Proposition 2.3). □

Algorithm 3 Byte Pair Encoding Algorithm .

```
 1: procedure ENCODE_BPE($x_1^N$, $\mathcal{V}$)
 2:     //Set initial encodings:
 3:     $\text{c\_tokens} = x_1^N$
 4:     //Iterate over merging order in $\mathcal{V}$, the first $|\mathcal{A}|$ entries correspond the the alphabet (no merge happens):
 5:     for $i = |\mathcal{A}| + 1, ...|\mathcal{V}|$ do
 6:         $\text{c\_tokens} \leftarrow \text{find\_merge}(\text{c\_tokens}, \mathcal{V}[i])$
 7:     end for
 8:     return c_tokens
 9: end procedure
10:
11: procedure find_merge(c_tokens, $v$)
12:     // Left and right tokens for merging
13:     $t_{\text{left}}, t_{\text{right}}, t_{\text{new}} = v[1], v[2], v[3]$
14:     $\text{old\_tokens} = \text{c\_tokens}$
15:     $\text{new\_tokens} = []$
16:     // Find and merge tokens from left to right
17:     $j = 1$
18:     while $j < |\text{old\_tokens}|$ do
19:         if $\text{old\_tokens}[i, i+1] = t_{\text{left}}, t_{\text{right}}$ then
20:             $\text{new\_tokens.append}(t_{\text{new}})$
21:             $j = j + 2$
22:         else
23:             $\text{new\_tokens.append}(\text{old\_tokens}[i])$
24:             $j = j + 1$
25:         end if
26:     end while
27:     return new_tokens
28: end procedure
```

Correctness of BPC Algorithm. We show in Proposition H.5 that computing the string probability $P(x_1^n)$ is equivalent to marginalizing the probability of all covering tokens of x_1^n. As such, the main task of computing $P(x_1^n)$ is to iterate all the valid encodings that cover x_1^n.

Proposition H.5. *(Prefix Probability Representation) Given a prefix x_1^n, we have the followings:*

1. *For any distinct $\vec{t}_i, \vec{t}_j \in \text{cover}(x_1^n)$, then $\mathcal{S}(\vec{t}_i) \cap \mathcal{S}(\vec{t}_j) = \emptyset$.*

2. $\mathcal{S}(x_1^n) = \bigcup_{\vec{t} \in \text{cover}(x_1^n)} \mathcal{S}(\vec{t})$.

As a result, $P(x_1^n)$ can be expressed as the marginal probability of all covering tokens of x_1^n

$$P(x_1^n) = \sum_{\vec{t} \in \text{cover}(x_1^n)} P(\vec{t}) \tag{50}$$

Proof. We prove each point as follows:

1. Proof by contradiction, suppose that $\mathcal{S}(\vec{t}_i) \cap \mathcal{S}(\vec{t}_j) \neq \emptyset$, then there exists a string s that has two different cover encodings $\vec{t}_1$ and $\vec{t}_2$. This is impossible since each string s has only one unique encoding.

2. This follows the definition of cover encodings.

Since each $\mathcal{S}(\vec{t})$ is pair-wise disjoint, we arrive at the final equation. We illustrate this Proposition in Figure 6. □

Finally, we prove that BPC extracts all the cover encodings of x_1^n. Proposition H.6 shows the correctness of line 9 in the algorithm, where suppose that the last token starts from x_{i+1}, then $\text{encode}(x_1^i)$ must be the encoding before that last token. Since each cover encoding $\vec{t}$ must have a last token cover a suffix within x_1^n, iterating over all positions from 1 to n guarantees that we extract all possible encodings.

Proposition H.6. *Let $\vec{t} \in \text{cover}(x_1^N)$ and $k = |\vec{t}|$ is the number of tokens in $\vec{t}$. Suppose that x_{j+1}^N is a prefix of the t_k for some $0 \leq j \leq N$, i.e. $x_j^N \in \text{prefix}(\vec{t}_k)$, then $t_1^{k-1} = \text{encode}(x_1^j)$.*

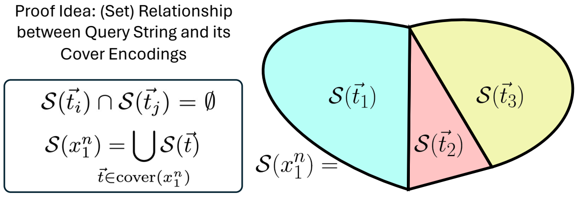

Proof Idea: (Set) Relationship between Query String and its Cover Encodings
$\mathcal{S}(\vec{t}_i) \cap \mathcal{S}(\vec{t}_j) = \emptyset$
$\mathcal{S}(x_1^n) = \bigcup_{\vec{t} \in \text{cover}(x_1^n)} \mathcal{S}(\vec{t})$
$\mathcal{S}(x_1^n) =$
$\mathcal{S}(\vec{t}_1)$
$\mathcal{S}(\vec{t}_2)$
$\mathcal{S}(\vec{t}_3)$

Figure 6: Illustration of Proposition H.5 when $|\text{cover}(x_1^n)|$=3.

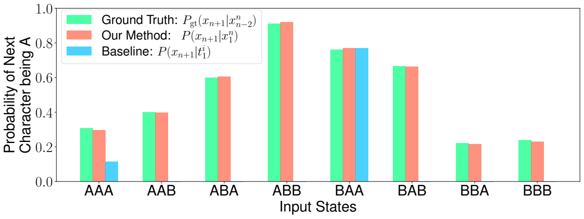


Figure 7: Our method accurately estimates the transition probability of a 3rd order Markov chain while the baseline method fails to.

Proof. Since $\vec{t} = t_1^k \in \text{cover}(x_1^N)$, then t_1^{k-1} must be a valid encoding. As a result, we must have $t_1^{k-1} = \text{encode}(x_1^j)$. □

Refactoring. Unlike the case for MPE, identifying the case when $P(x_{n+1}^N|t_1^i) = P(x_{n+1}^N|x_1^n)$ is nontrivial in general for BPE. Nevertheless, the refactoring step aims to reduce the computational complexity, and in general, we can still compute $P(x_{n+1}^N|x_1^n)$ by refactoring.

$$P(x_{n+1}^N|x_1^n) = \frac{P(x_1^N)}{P(x_1^n)}, \tag{51}$$

and we use the BPC algorithm to compute $P(x_1^N)$ and $P(x_1^n)$ respectively. Note that this is equivalent to assuming t_1 is a $<$start$>$ token within $\mathcal{V}^*$ (and consider $P(x_2^N|t_1), P(x_2^n|t_1)$ instead of $P(x_1^N), P(x_1^n)$). When pretokenization is used, e.g. split by white spaces, we can identify when $P(x_{n+1}^N|t_1^i) = P(x_{n+1}^N|x_1^n)$ by using the pretokenization pattern.

H.4. Experiments

The experiment setup for BPE is the same as the one in Section 4, except we use the vocabulary $\mathcal{V} = \{$"A", "B", "$B{\cdot}A$", "$BA{\cdot}A$", "$B{\cdot}BAA$", "$A{\cdot}A$", "$BA{\cdot}BA$", "$B{\cdot}B$"$\}$, where the order within $\mathcal{V}$ is the merging order for the BPE encoding process and the "$\cdot$" separates the merging tokens. The result is shown in Figure 7, where our method can accurately recover the ground truth probability $P(x_{n+1}|x_1^n)$ while the baseline fails to. Notice that for the state "BAA", the baseline approach can output the correct probability, which is because the merging for the token "BAA" happens before any mergs where "A" is the left token happens. This experiment also shows the existence of bias within the BPE process and our method can recover the exact ground truth probability.