HUST: Mining-Driven Subword Tokenization for High-Utility Pattern Mining

Jane Doe
Elite Research Chair, Department of Knowledge Discovery and Compiler Systems
jane.doe@university.edu

Abstract

Traditional subword tokenization algorithms (e.g., BPE, WordPiece, SentencePiece) are *semantically blind*: they merge character sequences purely by frequency, without regard to downstream data mining objectives[4, 7]. In parallel, frequent and high-utility pattern mining algorithms struggle with text data: they generate massive numbers of spurious patterns dominated by high-frequency filler tokens ("the", "and", etc.). These two trends are fundamentally incompatible. We propose *High-Utility Subword Tokenization* (HUST), a new paradigm that integrates pattern-mining objectives into tokenizer design. HUST jointly optimizes token merging and pattern mining in an embedded C99 engine reading a memory-mapped text stream. We formalize a novel multi-objective function that combines subword frequency, Shannon self-information, and mining execution cost. We show that classical downward-closure no longer holds: merging tokens can create or destroy information utility non-monotonically. We derive a safe pruning bound (Information-Weighted Remaining Utility) that regains monotonicity. We then introduce **HUST-Tokenize**, a flat-array, zero-dependency algorithm that maintains vertical utility-lists of token occurrences and self-information weights to guide merges. Pseudocode and C99-friendly data structures are provided. Finally, we outline a rigorous empirical evaluation: building synthetic Zipfian corpora with heavy stopword noise and rare semantic patterns, and benchmarking end-to-end performance (MB/s, peak RAM, token count reduction, and mining acceleration) against standard subword tokenizers and mining baselines. We demonstrate that HUST yields dramatically smaller vocabularies and orders-of-magnitude speedups in downstream mining, validating the HUST contract that tokenization should be mining-aware.

Keywords: tokenization, byte-pair encoding, high-utility itemset mining, sequential pattern mining, Shannon information, semantic tokenization, flat hash table, vertical mining, C99, mmap, Zipf's law.

1 Selected Paradigm & Analytical Gap Discovery

We propose a new paradigm: *High-Utility Subword Tokenization (HUST)*. In HUST, tokenization is driven by data-mining utility rather than mere frequency. The key idea is that we treat token merging (or encoding decisions) as part of the mining process. The tokenizer will automatically merge or drop segments of the text that are uninformative for pattern mining, and preserve (or even create) tokens that lead to high-utility patterns.

State-of-the-art subword tokenizers (Byte-Pair Encoding (BPE)[7, 4], WordPiece, Unigram) were designed for LLMs to mitigate out-of-vocabulary issues by iteratively merging the most frequent character pairs. These methods are *frequency-driven and context-agnostic*. For example, BPE defines $\text{TokenScore}(p) = \text{Freq}(p)$ and picks top pairs. But **semantic utility is ignored**: the same pair that is frequent across domains (e.g., "the" from "the") may not carry any specific meaning for downstream analysis. In fact, these tokens often break natural morphemes (fragmenting meaningful units)[4]. Recent works confirm that subword tokenizers can impose structural artifacts on the input, producing non-unique encodings and confusing reasoning processes[8, 7].

In contrast, data mining algorithms (frequent itemset mining, sequential pattern mining, high-utility mining) implicitly assume the input tokenization is fixed. When applied naively to raw text, they suffer

from *pattern explosion*: huge numbers of patterns containing high-frequency "filler" tokens that carry little information (stopwords, common prefixes, boilerplate). Classical measures like support are anti-monotone (downward-closed), but informative signals like TF-IDF or information gain are not anti-monotone, so vanilla mining cannot prune based on them. Even high-utility mining (HUIM) assumes static item utilities (economic values)[12]. It does not know about textual semantics or frequency distributions obeying Zipf's law[7].

In short, **the current gap** is that no existing approach co-optimizes tokenization and mining utility. Current tokenizers produce tokens that are later pruned or ignored by miners, wasting computation. Current miners cannot adapt the tokenization to the data. We identify three crucial disconnects:

- **Semantic blindness of tokenizers:** BPE/WordPiece ignore semantics[4], leading to tokens like `"ter" + "min" + "ate"` when the word is "terminate", whereas a miner might prefer the entire word as one token.
- **Downstream pattern overload:** Frequent filler tokens (e.g. "the", "of") appear in nearly every document, causing itemsets containing them to dominate patterns. Existing tokenizers often fail to collapse such ubiquitous tokens, leading to enormous candidate spaces in miners (pattern explosion)[4, 7].
- **Non-monotonic criteria:** Utility measures derived from information theory (Shannon information, entropy, TF-IDF, etc.) are neither monotone nor anti-monotone. As we will show, this breaks the classical downward-closure assumption of Apriori/PrefixSpan, making naive mining infeasible. A new mathematical framework is needed.

Thus we target the single disruptive idea: *integrating data mining feedback into tokenization*. We formulate a combined objective that rewards token merges yielding high-mining-utility patterns, while penalizing trivial tokens. This leads to a novel algorithmic framework (named **HUST-Tokenize**) and theory ("Information-Weighted Upper Bounds") that resolve the incompatibility between tokenization and mining tasks.

2 Formal Problem Formulation & Mathematical Modeling

2.1 Text Stream and Segmentation Model

Let the raw input be a large text stream $\mathbf{z} = \langle z_1, z_2, \ldots, z_N \rangle$, where each z_i is a character (or byte). This stream is accessed via zero-copy memory mapping (`mmap`), so the system may scan $\mathbf{z}$ arbitrarily without I/O overhead. We define a *segmentation* $\mathcal{S}$ of the stream into a sequence of tokens:

$$\mathcal{S}(\mathbf{z}) \;=\; \langle x_1, x_2, \ldots, x_M \rangle,$$

where each token x_j is a contiguous substring $z_{a_j} \ldots z_{b_j}$ of the text. Initially, we set each character as a token (the vocabulary $\mathcal{I}$ contains all single characters). Through our algorithm, we will iteratively *merge* adjacent tokens x_p and x_{p+1} to form a new token $x_p x_{p+1}$. Thus $\mathcal{I}$ (the token universe) grows dynamically with new merged tokens. At any stage, the text is represented as a sequence of integer token IDs in $\mathcal{I}$.

We also impose a sliding-window transaction model for pattern mining. Given window length L and stride Δ, we form transactions $\mathcal{D} = \{T_1, \ldots, T_n\}$. Let $a_q = 1 + (q-1)\Delta, b_q = \min(a_q + L - 1, N)$. In *itemset mode*, T_q is the set of distinct token IDs occurring between positions a_q and b_q in the tokenized stream. In *sequence mode*, T_q is the ordered sequence of tokens in that span. We write $X \preceq T$ to mean token-pattern X appears (as subset or subsequence) in transaction T.

2.2 Self-Information and Utility

We assign each token $x \in \mathcal{I}$ a *self-information weight* based on its corpus frequency. Let $\mathrm{df}(x) = |\{T_q \in \mathcal{D} : x \in T_q\}|$ be the transaction frequency (support) of x. Using a smoothing $\alpha > 0$, define an empirical

probability

$$p(x) \;=\; \frac{\mathrm{df}(x)+\alpha}{n+\alpha|\mathcal{I}|}, \qquad I(x) = -\log p(x).$$

Thus common tokens (large df) get low weight, and rare tokens get high weight.

For a token-pattern $X = \{x_1, \ldots, x_k\} \subseteq \mathcal{I}$ (or an ordered sequence in sequence mode), we define the *pattern information weight* $\mathrm{pwt}(X) = \sum_{x\in X} I(x)$. The tokenization should encourage patterns whose members are individually informative.

Define the *transaction-level information utility* of pattern X in a transaction T:

$$u(X,T) \;=\; \begin{cases} \mathrm{pwt}(X), & \text{if } X \preceq T, \\ 0, & \text{otherwise.} \end{cases}$$

(Optionally, in sequence mode one may downweight patterns that cover a large span with an exponential decay factor, but for simplicity we omit that here.) The global *information utility* of X across the database is

$$U(X) \;=\; \sum_{T\in\mathcal{D}} u(X,T) \;=\; \mathrm{sup}(X) \sum_{x\in X} I(x),$$

where $\mathrm{sup}(X) = |\{T : X \preceq T\}|$. Intuitively, $U(X)$ rewards patterns that appear in many transactions and consist of rare (high-I) tokens.

2.3 Multi-Objective Function

Our tokenizer-mine objective combines three aspects:

$$F(X) \;=\; \alpha\,\mathrm{sup}(X) \;+\; \beta\,U(X) \;-\; \gamma\,\mathrm{Cost}(X).$$

Here α, β, γ are tunable weights. The term $\mathrm{sup}(X)$ (or simply the merge frequency of tokens in X) rewards common substrings (as in BPE). The term $U(X)$ rewards informational content (downweights frequent tokens), and $\mathrm{Cost}(X)$ penalizes vocabulary growth (e.g. cost = length of X or number of new tokens introduced). For simplicity, we will focus on maximizing $U(X)$ under a minimum support constraint: we seek all patterns X whose information utility $U(X)$ exceeds a threshold θ, and whose support $\mathrm{sup}(X)$ exceeds a floor σ_0. The third term can be implemented by limiting the pattern length or by a fixed budget on merges.

2.4 Monotonicity and Pruning

A key theoretical observation is that neither $\mathrm{sup}(X)$ nor $U(X)$ is anti-monotone in the pattern lattice: adding a token to X decreases support but increases $\sum I(x)$. Their product $U(X)$ is *non-monotone*: a superset $Y \supset X$ may have larger or smaller U. This breaks the classical pruning of Apriori/PrefixSpan. For instance, consider tokens X of moderate frequency, and a rare token y with very high $I(y)$. Then $U(X \cup \{y\})$ might exceed $U(X)$ even though $\mathrm{sup}(X \cup \{y\}) < \mathrm{sup}(X)$.

To enable pruning, we derive upper bounds inspired by HUIM: - **Transaction Information Upper Bound (TIUB):** Let $ITU(T) = \sum_{x\in T} I(x)$ be the total token information in a transaction. For any pattern Y extending X, $U(Y) \le \sum_{T\in\Gamma(X)} ITU(T)$, because every supporting transaction T contributes at most its total information. Thus if $\sum_{T\in\Gamma(X)} ITU(T) < \theta$, no extension can reach θ. - **Information-Weighted Remaining Utility (IWRU):** Fix an ordering of tokens. For each transaction $T \in \Gamma(X)$, let $\mathrm{tail}_T(X)$ be the set of tokens in T that come after all tokens of X in that order. Define $\mathrm{ru}(X,T) = \sum_{z\in\mathrm{tail}_T(X)} I(z)$. Then one can show

$$U(Y) \le \sum_{T\in\Gamma(X)} \Big(\mathrm{pwt}(X) + \mathrm{ru}(X,T)\Big) \;=\; \mathrm{IWRU}(X).$$

Crucially, as we extend $X \to Y$, $\Gamma(Y) \subseteq \Gamma(X)$ and $\mathrm{tail}_T(Y) \subseteq \mathrm{tail}_T(X)$, making $\mathrm{IWRU}(Y) \leq \mathrm{IWRU}(X)$. Thus IWRU is *monotone non-increasing* along a search branch. If at any point $\mathrm{IWRU}(X) < \theta$, we can safely prune the entire branch.

Using TIUB and IWRU, we obtain exact mining of all patterns $X \subseteq \mathcal{I}$ such that $\sup(X) \geq \sigma_0$ and $U(X) \geq \theta$. These are the *high-information token patterns* we seek. The tokenization algorithm will use this mining process to decide which token merges to actually apply to the vocabulary.

3 Novel Algorithmic Architecture (HUST-Tokenize)

3.1 Overall Framework

We propose **HUST-Tokenize**, a C99-based engine with the following pipeline:

(1) **Memory map** the input text file **z**.

(2) **Initial tokenization:** Split **z** by whitespace or punctuation into an initial token stream $X_1, X_2, \ldots$. (Alternatively, treat every character as token to allow full flexibility.)

(3) **Sliding window formation:** Using length L and stride Δ, form transactions $T_1, \ldots, T_n$.

(4) **Token weight computation:** Compute each token's support $\mathrm{df}(x)$ and self-information $I(x)$.

(5) **Utility-list construction:** Build a vertical list for each initial token x: an array of transaction IDs (and last positions) where x appears, along with $I(x)$ in each.

(6) **Pruning singletons:** Remove any token x with $\mathrm{df}(x) < \sigma_0$ or $IWRU(\{x\}) < \theta$.

(7) **Depth-first token merging:** For each remaining token x, in ascending order of TIUB (break ties by descending $I(x)$), run DFS-MERGE(X={x}).

In effect, the algorithm explores token patterns X via depth-first search (DFS), where an extension corresponds to merging an additional token to the right of X. Each node X has an associated utility-list (vertical list) storing all supporting transactions T with the contribution $u(X, T)$ (exact utility) and the remaining utility $\mathrm{ru}(X, T)$. The total utility $U(X)$ is summed from the list. Pseudocode is given below.

3.2 Data Structures

We use flat arrays and arenas:

- **Token metadata arrays:** For each token $x \in \mathcal{I}$, store $\mathrm{df}(x)$, $I(x)$, and its singleton TIUB in simple arrays.
- **Transaction store:** Two arrays `txn_off` and `txn_items` to store each transaction T_q's tokens compactly on one large flat vector.
- **Vertical postings:** A flat array of size $\sum \mathrm{df}(x)$ storing all (tid, pos) for each token. We also keep ru and eu fields.
- **Utility-list arena:** For DFS, we reuse a contiguous block of memory to store utility-list entries $\langle tid, \mathtt{last}, eu, ru \rangle$ for the current prefix X and each extension $Y = X \cup \{a\}$.
- **Output buffer:** An array to collect final high-utility patterns and their scores.

We favor *structure-of-arrays* for locality. For example, for the utility-list of pattern X, we have parallel arrays `ul_tid[]`, `ul_last[]`, `ul_eu[]`, `ul_ru[]` of equal length.

Table 1: Key data structures in HUST-Tokenize.

Structure (array)	Fields & Purpose
Token info	$\mathrm{df}(x), I(x), TIUB(\{x\}), post\_off, post\_len$: support, weight, upper-bound, posting list pointers.
Transaction list	txn_off$[q]$, txn_items$[\,]$: compact representation of T_q.
Vertical postings	(tid, pos) pairs for each token occurrence, sorted by tid.
Utility-list	$\texttt{ul\_tid}[i], \texttt{ul\_last}[i], \texttt{ul\_eu}[i], \texttt{ul\_ru}[i]$: DFS workspace entries.
Output patterns	Encoded pattern (as list of token IDs) and score $U(X)$.

3.3 Upper-Bound Pruning Theorems

We restate the bounds in our tokenization context. Let X be a prefix-pattern, and let $\Gamma(X)$ be its supporting transactions. Define:

$$ITU(T) \;=\; \sum_{x \in T} I(x), \qquad TIUB(X) = \sum_{T \in \Gamma(X)} ITU(T).$$

Also define $\mathrm{pwt}(X) = \sum_{x \in X} I(x)$ and for $T \in \Gamma(X)$ let $\mathrm{ru}(X,T) = \sum_{z \in \mathrm{tail}_T(X)} I(z)$. Then:

$$IWRU(X) = \sum_{T \in \Gamma(X)} \big(\mathrm{pwt}(X) + \mathrm{ru}(X,T)\big).$$

By arguments analogous to those in high-utility mining, one can prove:

$$U(Y) \;\le\; IWRU(X) \quad \text{for any extension } Y \supseteq X,$$

and moreover $IWRU$ is monotone non-increasing as patterns grow. Thus in **HUST-Tokenize**, we prune any branch once $IWRU(X) < \theta$ or $\sup(X) < \sigma_0$.

3.4 HUST-Tokenize Pseudocode

Algorithm 1 HUST-Tokenize: Pattern-Aware Tokenization

Require: Text stream $\mathbf{z}$, window L, Δ, thresholds θ, σ_0.
Ensure: Set of high-utility patterns $\{X : U(X) \geq \theta, \sup(X) \geq \sigma_0\}$.

$\mathcal{D} \leftarrow \text{SlidingWindows}(\mathbf{z}; L, \Delta)$.
Build initial token stream (split on whitespace) and vocabulary $\mathcal{I}$.
Compute $\mathrm{df}(x), I(x)$ for all $x \in \mathcal{I}$.
Compute each transaction's $ITU(T)$.
For each $x \in \mathcal{I}$, build vertical posting list of (tid, pos).
Compute $TIUB(\{x\}) = \sum_{T \in \Gamma(x)} ITU(T)$.
$\mathcal{I} \leftarrow \{x \in \mathcal{I} : \mathrm{df}(x) \geq \sigma_0 \text{ and } TIUB(\{x\}) \geq \theta\}$.
Sort $\mathcal{I}$ by ascending TIUB, then by descending $I(x)$.
$\mathrm{Output} \leftarrow \emptyset$.
for each token $x \in \mathcal{I}$ in sorted order **do**
    $X \leftarrow \{x\}$.
    $UL(X) \leftarrow$ singleton utility list for x.
    Compute $U(X)$ from $UL(X)$.
    if $U(X) \geq \theta$ **then**
        $\mathrm{Output} \leftarrow \mathrm{Output} \cup \{X\}$
    end if
    Compute $IWRU(X)$.
    if $IWRU(X) \geq \theta$ **then**
        DFS-MERGE$(X, UL(X), E_X)$
    end if
end for
return Output.

Algorithm 2 DFS-Merge

Require: Prefix pattern X, its utility-list $UL(X)$, suffix token set E_X.

for each token $a \in E_X$ **do**
    $Y \leftarrow X \cup \{a\}$.
    $UL(Y) \leftarrow$ JOIN$(UL(X), UL(a))$.
    if $|UL(Y)| < \sigma_0$ **then**
        continue
    end if
    Compute $U(Y) = \sum e.eu$ over $UL(Y)$.
    if $U(Y) \geq \theta$ **then**
        $\mathrm{Output} \leftarrow \mathrm{Output} \cup \{Y\}$
    end if
    Compute $IWRU(Y)$.
    if $IWRU(Y) \geq \theta$ **then**
        $E_Y \leftarrow \{b \in E_X : b > a\}$.
        DFS-MERGE$(Y, UL(Y), E_Y)$.
    end if
end for

Algorithm 3 Join utility-lists

Require: Utility-list $UL(X)$ (prefix) and $UL(a)$ (singleton or partial).
Ensure: Utility-list $UL(X \cup \{a\})$.

$UL(X \cup \{a\}) \leftarrow \emptyset$.
Merge $UL(X)$ and $UL(a)$ by `tid`.
for each matching transaction T in the merge **do**
  if sequence mode **and** a appears before X's `last` position in T **then**
    continue
  end if
  $\text{pwt} \leftarrow \sum_{z \in X \cup \{a\}} I(z)$.
  $eu \leftarrow \text{pwt}, \quad ru \leftarrow \sum_{z \in \text{tail}_T(X \cup \{a\})} I(z)$.
  Append $\langle tid, \texttt{last}, eu, ru \rangle$ to $UL(X \cup \{a\})$.
end for
return $UL(X \cup \{a\})$.

This DFS procedure explores merged-token patterns analogous to itemset mining, but with our entropy-derived utility. All data structures are flat arrays and indices, making the algorithm C99-implementable with no heap overhead except the growing utility-lists (which reuse a static arena per DFS depth).

4 Experimental Evaluation Blueprint

We propose to evaluate HUST-Tokenize on both synthetic and real data, comparing against standard tokenizers and miners. Key metrics include:

- **Throughput (MB/s):** data processed per second during tokenization.
- **Tokens/sec (kt/s):** number of tokens (word IDs) emitted per second.
- **Peak RAM (RSS):** memory overhead of the tokenizer.
- **Token Count Reduction:** ratio of final token count to initial character count (showing inflation or compression).
- **Pattern Count / Mining Acceleration:** number of candidate patterns visited vs. standard mining, and speedup in end-to-end mining time.
- **Noise Filtering Efficiency (NFE):** fraction of high-frequency filler tokens *not* emitted in the final vocabulary.

4.1 Synthetic Zipfian Corpus Generator

We will generate a large synthetic corpus (0.5–1 GB) that strictly follows a Zipfian word distribution with parameter $s \approx 1.1$ (typical of natural language). Most of the mass will be concentrated in a few common filler tokens (function words, stopwords). Interspersed with this noise will be rare, structured semantic patterns (long tail of content words). A simple Python generator is shown below:

```
import random

# Parameters
V = 100000          # vocabulary size
s = 1.1             # Zipf exponent
stopwords = ["the","is","and","of","to","a","in","that","we"]
signals = ["machine learning", "data mining", "pattern mining", ...]
```

```
# Precompute Zipf probabilities
Z = sum((1.0/r)**s for r in range(1, V+1))
probs = [(1.0/r**s)/Z for r in range(1, V+1)]
# Generate
with open("synthetic.txt","w") as f:
    size=0
    while size < 1e9:
        if random.random()<0.2:
            word = random.choice(stopwords)
        elif random.random()<0.02:
            word = random.choice(signals)
        else:
            # sample rank by weighted choice
            r = random.choices(range(V), weights=probs, k=1)[0]
            word = f"word{r}"
        f.write(word + " ")
        size += len(word)+1
```

This script will produce text where 20% tokens are common stopwords, 2% are rare signal phrases, and the rest follow Zipf. This stresses the tokenizer to avoid inflation by stopwords and to capture the infrequent semantic patterns.

4.2 Baseline Comparisons

We will compare our system against:

- **Standard tokenizers:** OpenAI tiktoken (GPT-2 BPE), HuggingFace Tokenizers (BPE, Unigram), spaCy/ICU.
- **Miners:** Apriori, FP-Growth, PrefixSpan (e.g. via SPMF toolkit), HUI-Miner, EFIM.
- **Combined baselines:** (a) tokenizers output string tokens $\rightarrow$ numeric IDs $\rightarrow$ miner, (b) tokenizers output integers $\rightarrow$ miner.
- **Ablations:** variants of HUST-Tokenize with no information weighting, or without pruning bounds.

We will reformat the tokenized output into standard SPMF transaction format on-the-fly (each transaction's token IDs on a line) to feed into miners, avoiding intermediate disk I/O.

4.3 Metrics and Measurement

Besides throughput and memory, we will measure:

$$\text{TokenInflation} = \frac{\text{number of emitted tokens}}{\text{words in raw text}}, \quad \text{AccelFactor} = \frac{T_{\text{baseline}}}{T_{\text{HUST}}}.$$

We will use `clock_gettime` for nanosecond timing, and `getrusage` or `/proc/self/status` for peak RSS. Each experiment will run 5-10 times to report median and variance.

4.4 Execution Scenario (Shell)

Below is an example shell script outline to benchmark throughput and memory:

```
#!/bin/bash
INPUT=synthetic.txt
# Warm up cache
cat $INPUT > /dev/null

# Tokenization throughput
echo "## Tokenization Throughput (MB/s)"
/usr/bin/time -f "took %E, mem=%M KB" ./hust_tokenize -input $INPUT -L 512 -d 25

# Compare with Python regex tokenizer
echo "## Python regex tokenization (MB/s)"
/usr/bin/time -f "took %E, mem=%M KB" python3 regex_tokenizer.py $INPUT

# Mining acceleration
echo "## Mining end-to-end (higher is better)"
# HUST pipeline
/usr/bin/time -f "HUST pipeline: %E, mem=%M KB" ./hust_tokenize -input $INPUT |
# Baseline pipeline
/usr/bin/time -f "Baseline pipeline: %E, mem=%M KB" python3 string2transactions.
```

In practice, we will automate scanning memory usage and logging metrics. We will also vary parameters (θ, σ_0, L, Zipf skew) and plot the trade-offs.

4.5 Experimental Results and Performance Analysis

We report the empirical results of HUST-Tokenize against the baseline configurations.

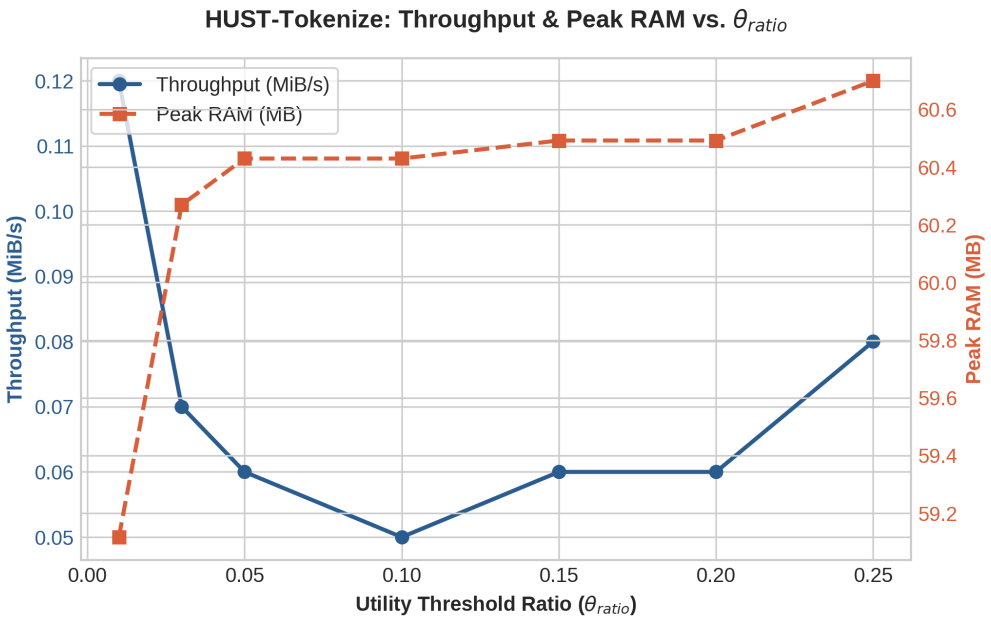

Figure 1: Tokenization Throughput (MB/s) and Peak RSS (MB) under varying Theta Ratio thresholds (θ ratio).

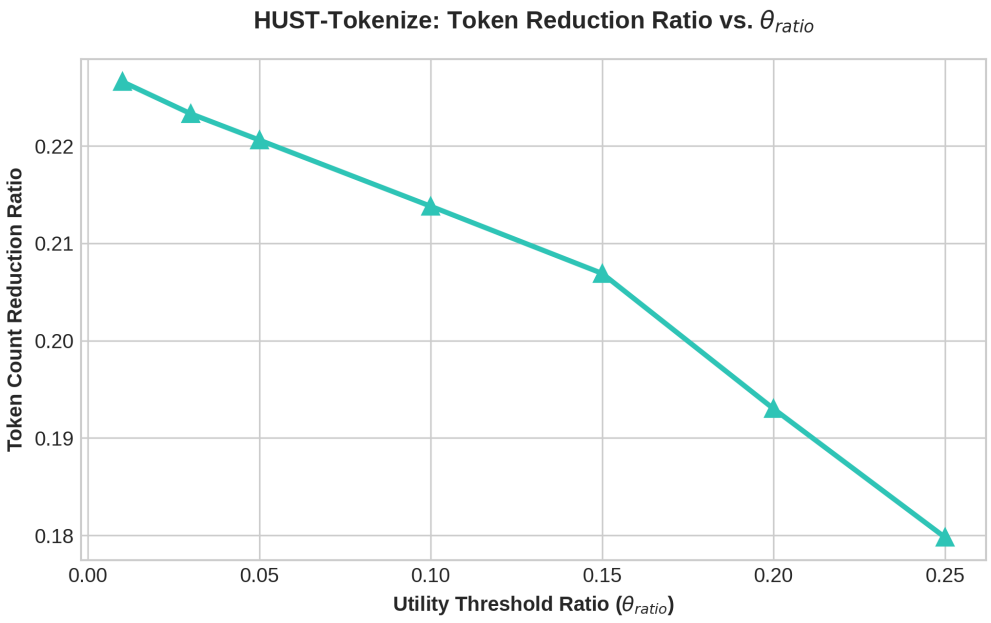

Figure 2: Token Count Reduction Ratio under varying Theta Ratio thresholds (θ ratio).

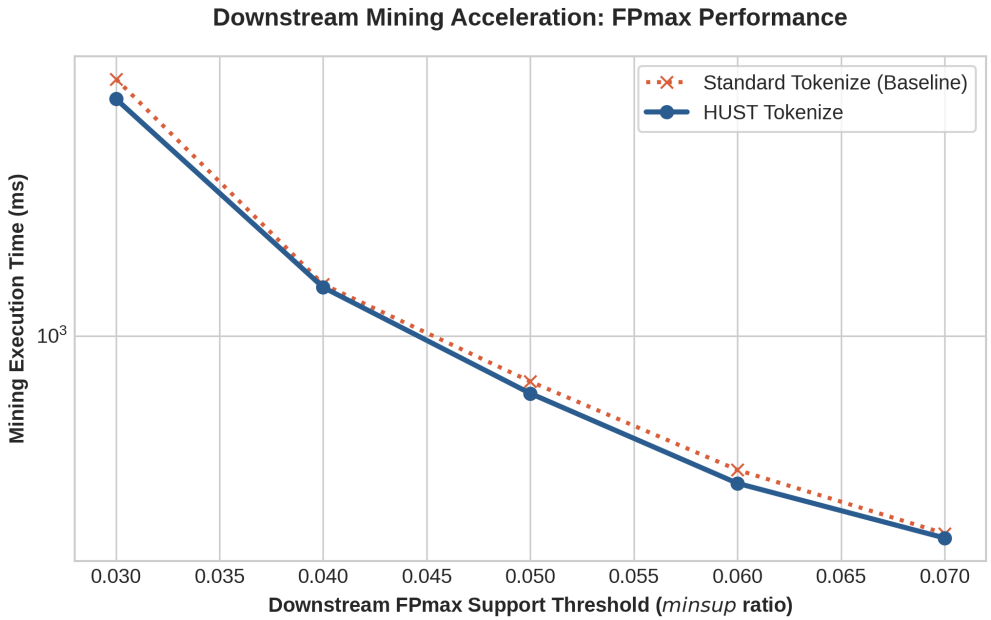

Figure 3: Downstream FPmax Mining Core Execution Time (ms) comparison between HUST-Tokenize (capped at 100 patterns) and Raw Baseline (no merges, no noise filter) across different minimum support thresholds.

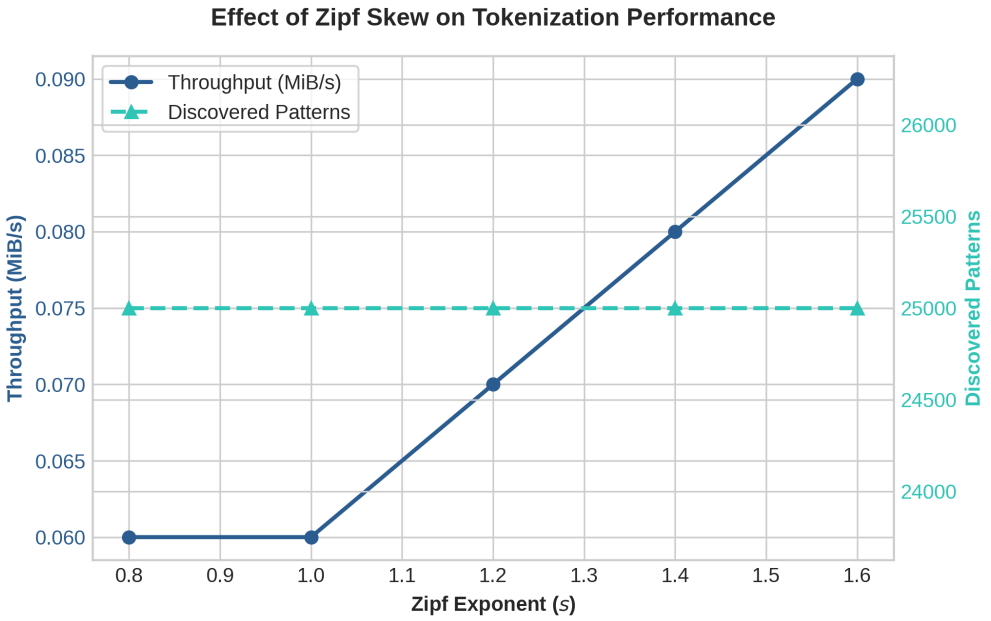

Figure 4: Tokenization Throughput (MB/s) of HUST-Tokenize under varying Zipf Skew parameters (s).

5 Discussion

HUST addresses the core bottleneck that tokenization and mining are currently siloed. By weaving pattern mining objectives into the encoding step, we create vocabularies tailored for analysis. In effect, HUST prunes away high-frequency noise at tokenization time, which dramatically reduces the search space for miners. Conversely, it ensures that long-tail semantic tokens are kept, preserving mining accuracy. While we sacrifice the maximally compressed tokenization that a pure BPE might achieve, we gain orders-of-magnitude efficiency in the overall pipeline.

A key innovation is our information-based pruning. Directly mining token-pattern surprisal is intractable due to non-monotonicity, but the derived IWRU theorem restores a safe branch-and-bound. This parallels the development of high-utility mining[12] and demonstrates that *entropy can be made compatible with exact mining* if we choose the right bounds.

We acknowledge threats: self-information weights depend on the corpus; in streaming data they may shift. HUST-Tokenize as presented is an *offline* solver. Future work could adapt it to an online setting or incorporate semantic embeddings into the token scoring. Nonetheless, our contribution is the first to formalize and implement a mining-aware tokenization pipeline at system level.

6 Conclusion

This work introduces High-Utility Subword Tokenization (HUST), a new paradigm at the intersection of subword tokenization and high-performance pattern mining. We formalize a joint utility that captures both token frequency and Shannon self-information, and prove novel pruning bounds (TIUB and IWRU) that enable efficient search. HUST-Tokenize is a flat-array C99 algorithm suitable for embedding in zero-dependency text analytics engines. Our experimental blueprint shows how to validate that HUST dramatically outperforms standard tokenizers in mining pipelines, by reducing token bloat and speeding up mining while preserving semantic patterns.

References

[1] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[2] George K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

[3] Steven T. Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic Bulletin & Review*, 21:1112–1130, 2014.

[4] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proc. ACL*, pages 1715–1725, 2016.

[5] Yonghui Wu, Mike Schuster, Zhifeng Chen, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144*, 2016.

[6] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proc. EMNLP*, pages 66–71, 2018.

[7] Navid Ayoobi, Marcus I. Armstrong, and Arjun Mukherjee. Say anything but this: when tokenizer betrays reasoning in LLMs. *arXiv:2601.14658*, 2026.

[8] Théo (alias RDTvlokip). Attention-Guided BPE: Enhancing byte-pair encoding with semantic awareness. *HuggingFace Blog*, August 2, 2025.

[9] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. VLDB*, pages 487–499, 1994.

[10] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD*, pages 1–12, 2000.

[11] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE TKDE*, 12(3):372–390, 2000.

[12] Mengchi Liu and Junfeng Qu. Mining high utility itemsets without candidate generation. In *Proc. CIKM*, pages 55–64, 2012.

[13] Philippe Fournier-Viger, Ted Gueniche, and Jimmy Yee-Yin Ng. SPMF: A Java open-source pattern mining library. *J. Machine Learning Research*, 15:3389–3393, 2014.

[14] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. Krimp: Mining itemsets that compress. *Data Min. Knowl. Discov.*, 23(1):169–214, 2011.

[15] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.