TalmudTokenizer: Framework for Analysis of Babylonian Jewish Legal Texts

Shlomi Helfgot
shlomi.helfgot@yale.edu

Advisor: Dr. Ruzica Piskac
ruzica.piskac@yale.edu
Consulting advisor: Prof. Scott Shapiro
scott.shapiro@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Arts in Computer Science

Department of Computer Science
Yale University
Dec 11, 2025

Acknowledgements

I would like to express my gratitude for Dr. Ruzica Piskac's unflagging interest, infectious optimism, wise guidance, and most of all flexibility, which have helped me embark on this project. I further acknowledge Prof. Scott Shapiro's kind help in thinking through the parameters of machine learning and Talmudic studies, as well as his suggestion to focus on laying the groundwork for future, more complex computational analyses of the Talmud, a suggestion which directly led me to focus on the problem of tokenization of the Babylonian Talmud.

Last but certainly not least vis a vis technical debt, I would like to thank Ephraim Helfgot, from the bottom of my heart, both for copious work and testing of the crowdsourced annotation software, and furthermore for being there for me this semester, one full of highs, lows, challenges, etc.

I am beyond grateful to my parents for nurturing me, inculcating within me an interest in serious scholarship, and providing me the opportunity to study in Yale. I am further grateful to all of the teachers and friends who guided me through the thickets of the Talmudic conversation: R. Ben Czeladnicki, R. Dr. Aaron Ross, R. Aron Srolovitz, R. Avie Schrieber, R. Yosef Adler, R. Daniel Fridman, R. Moshe Taragin, R. Shmuel Shimoni, R. Tzvi Chaim Kaye, R. Mosheh Lichtenstein, Rabbi Shlomo Zuckier, Rabbi Alex Ozar, Rabbi Yehuda Goldberg, (R.) Ephraim Helfgot (again!), (R.) Yitzy Tanner, Danny Friedman, Yossi Nadel, Alex Ostrin, Mosche Wyler, Mark Gotesman, Doni Weber, Hillel Koslowe, Sammy Rosenberg, Jacob Colchamiro, and Shmuel Ross. Productive conversation partners about Talmud and digital humanities include Dan Jutan, Shaun Regenbaum, Alex Shapiro (Yale '26), and Eytan Israel (Yale '26). I am sure that I am leaving someone out, so *yihyu be'einechem k'ilu hayiti portam* ("may it be accounted in your eyes as if I enumerated them.") *Ya'amdu kulam al ha-berakhah* (may all find repose on the mount of blessings.)

This work was supported by the Slifka Center for Jewish Life under special bursary grant. Many, many thanks to the estimable Rabbi Alex Ozar and the wonderful Aviva Green for their help in securing the support.

'*Odeh et Adonai be-khol levav, be-sod yesharim ve-edah.*'

Contents

1 Introduction **6**
1.1 Tokenization . . . 6
1.2 Talmud Tokenization . . . 7

2 Background **8**
2.1 Talmud: Some important terms . . . 8
2.1.1 Pagination Hierarchy . . . 8
2.1.2 Content Hierarchy . . . 8
2.2 Talmudic Tokenization: Obstacles . . . 9
2.3 Tokenization: A short literature review . . . 10
2.3.1 Canonic algorithms: BPE, WordPiece, Unigram LM . . . 10
2.3.2 SOTA algorithms . . . 11
2.4 Metrics . . . 12
2.4.1 Renyi Entropy . . . 12
2.4.2 Gini coefficient . . . 13
2.4.3 Normalized Sequence Length (NSL) . . . 13
2.4.4 Fertility . . . 14
2.4.5 Zipfian Alignment . . . 14
2.4.6 Vocabulary Utilization . . . 15

3 Methodology **16**
3.1 Data Preparation . . . 16
3.2 Considering Joint vs. Separate Encodings . . . 17
3.3 Comparing Tokenization Algorithms . . . 20
3.3.1 Selection of Vocabulary Size . . . 20
3.3.2 Common Framework . . . 23
3.3.3 BPE (Byte-Pair Encoding) . . . 24
3.3.4 WordPiece . . . 25
3.3.5 Unigram Language Model . . . 26
3.3.6 TokenMonster . . . 28
3.3.7 SRE (Semitic Root Encoding) . . . 30
3.3.8 TMSRE (TokenMonster-SRE Hybrid) . . . 32
3.3.9 Evaluation . . . 33

4 Results **40**
4.1 Intrinsic Metrics . . . 40
4.1.1 Overall Performance / Algorithm Comparison . . . 40
4.1.2 Vocabulary Distribution and Utilization Efficiency . . . 41
4.1.3 Compression Efficiency and Sequence Length . . . 42
4.1.4 Zipfian Distribution Alignment . . . 43
4.1.5 Aside: Computational Efficiency and Training Time . . . 43
4.1.6 Masked Language Modeling Performance . . . 44
4.1.7 Algorithm-Specific Findings and Interpretation of Results . . . 44
4.2 Downstream Evaluation / Extrinsic Metrics . . . 45
4.2.1 Embedding-Quality Metrics . . . 46
4.2.2 Semantic Similarity . . . 46
4.2.3 Morphological Coherence . . . 46
4.2.4 Cross-Lingual Alignment . . . 47
4.2.5 Analogy Accuracy . . . 47
4.2.6 Trade-offs . . . 47
4.2.7 The Big Takeaway . . . 47

5 Related Work **55**

6 Conclusions **56**
6.1 Research Goals, Revisited . . . 56
6.2 Summary . . . 56

7 Future Work **57**

A Appendix **60**
A.1 Statistical analysis of Talmudic text corpus . . . 60
A.2 Minimal Vocabulary Size Experiment . . . 63

Talmudic Tokenization: Extending the SRE Paradigm

Shlomi Helfgot

Abstract

Tokenization, the process of compressing texts into machine-readable data, is a critical part of natural-language processing and especially language modeling. The Talmud, a Jewish legal compendium produced in Babylonia in the 6th century, presents specific problems where tokenization is concerned, with its multiple language layers, rich morphology and interplay between cognates, and especially its Semitic linguistic structure. Most tokenization algorithms leverage the agglutinizing properties of Western languages (e.g. "walk"+'ing'/'ed'/'er') to split words into subwords while preserving meaningful boundaries. With Semitic languages, based on root patterns and vowels, such subword tokenization risks breaking apart roots and destroying all meaning. How may Semitic words be efficiently analyzed by tokenizers, and specifically, what is the best way to tokenize the Talmud? Hatch and Richardson, in their paper of 2025, suggest storing the morphological template of the word along with its triconsonantal or quadriconsonantal root, in a process they term Semitic Root Encoding (SRE). We not only show that SRE may be applied to the Talmudic context to augment BPE, in line with Hatch and Richardson's predictions for Arabic corpora, but also extend SRE as a paradigm to the TokenMonster family of algorithms.This hybrid algorithm outperformed canonical tokenization algorithms (BPE and UnigramLM) as well as 2nd generation, more advanced algorithms (e.g. WordPiece, vanilla TokenMonster) on tasks requiring rich semantic representation, such as discerning morphological similarity, completing analogies, clustering of synonyms, and similarity between cognate words in Hebrew and Aramaic.

Chapter 1

Introduction

1.1 Tokenization

While humans process symbols and text with ease, computers live in a world of numbers, comprised of permutations of bits ($\in 0, 1$. In order to allow machines to recognize and parse language, a scheme to encode syntactic data (e.g. words) into a language computers far better recognizes, that of numbers – text must be chunked into pieces, known as *tokens*. Tokens may be words, characters, or subwords, and a tokenization algorithm may be as simple as the following, which splits a string by whitespace:

Algorithm 1: Split on whitespace

```
Data: string s
Result: tokenized string array
initialize array A and counter k = 0;
initialize b = 0 ;
while counter k < |s| do
    if s[k] = space then
        push s[b...k] to A;
        set b = k
    end
    increment k
end
```

Given a sentence 'the jackdaw clambered over the spotted zebra', this algorithm will return ['the', 'jackdaw', 'clambered', 'over', spotted zebra'].

Things become slightly complicated when we seek more granular representations of text, which might consider *portions* of words. We thus move to consider *subword* tokenization algorithms. Algorithms of this sort attempt to capture the structure of language by iteratively finding the most optimal splits in a given word. (E.g., semantically "walked" is comprised of "walk" and the past indicative "ed". This pattern ought to be recognized by a subword tokenization algorithm.)

By breaking down language into its most elemental pieces, subword tokenizations

can ameliorate the *OOV problem* ("out-of-vocabulary problem") impeding other tokenizers. If a whole-word tokenizer is met with the word 'ballooned', with a prior vocabulary of 'walked, balloon', it will mark the word as unknown. By having a machine construct words from subword components, a subword tokenizer with the same whole-word vocabulary might be able to recognize the word as 'balloon'+'ed'.

Tokenization is essentially identical with *compression*, because it changes the medium of data. While a tokenizer which expands text into a representation far larger in size than the original is quite possible, generally tokenizers are built to shrink the size of the raw data. This allows for quicker feature extraction, information retrieval, NER (named entity recognition). Far from a mundane matter of assigning each word in the corpus a number, the choice of proper tokenizer and compression strategy has immense downstream effects on computational tasks, perhaps just as much or more than curation of training data. [1] In the next section, we will investigate why tokenization is an important research goal vis a vis computational analyses of the Babylonian Talmud.

1.2 Talmud Tokenization

The Babylonian Talmud, the most important compendium of Jewish law compiled from the 3rd-6th centuries, is a labyrinthine and complicated text. Virtually no research has been done on the proper way to treat this work vis a vis tokenization. (See 'Related Work' section for some preliminary investigations.) Proper tokenization of the Talmud would be the first step in implementing high-quality versions of various digital humanities projects of immense interest to Jewish Studies scholars, historians, sociologists, and classics experts, for example:

- Automated vocalization for Aramaic, which is often written without the vowels, although these change the meaning of the words.
- Pipeline for detection of intertextual allusions, a critical part of the Talmudic method.
- Stylometric dating of various levels of redaction.
- Semantic RAG systems which might allow the user to query the Talmud, e.g. 'where are laws about free speech located?'

Chapter 2

Background

2.1 Talmud: Some important terms

In the following section, we provide a short survey of the critical terms used in structuring the text:

2.1.1 Pagination Hierarchy

The Talmudic text is a *hierarchical* one. It is split into 37 sections (known as tractates, or by their Aramaic name, '*mesekhta*' - in absolute state, i.e. 'this *mesekhta*', or '*masekhet*' when in construct, e.g. *Masekhet Berakhot* is 'Tractate Berakhot'. These range in length from just 18 pages (9 folios) to (176 folios). Folia are more commonly used for navigation than for pages; these have a recto side (side a) and verso side (side b). Because the title page is folio 1, the Talmudic text begins at 2a. For example, 16b is the 30th page of a given tractate [2].

2.1.2 Content Hierarchy

The Talmud comprises various languages because it is structured as a commentary recording the opinions of various Jewish scholars - who spoke Aramaic - regarding an earlier *Hebrew* text known as the *Mishnah* ('Teaching', in English). Both strata liberally quote from a book written in yet a third language: the Bible, which comprises an archaic form of Hebrew. Hence, the languages relevant to this project are: **Jewish Babylonian Aramaic**, **Mishnaic Hebrew**, and **Biblical Hebrew**.

Random sampling of 15 pages of Talmudic text from data annotated as part of our crowd-sourced annotation project (see below) yielded the following first-order approximation of the distribution of languages in the corpus (**Fig. 2.1**.) The share of Biblical Hebrew in the entire Babylonian Talmud is vanishingly small. This is because most of the Talmud is engaged in the disputes surrounding the mechanism and applications of the laws themselves than concerning themselves explicitly and primarily with the Biblical antecedents. (A cruder method of estimating the share of Biblical Hebrew in the text is described in [3].)

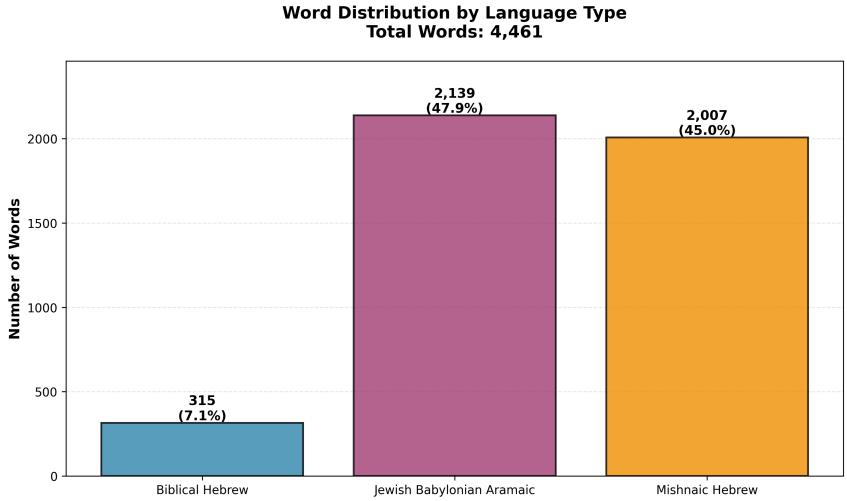

Figure 2.1: Word Distribution in Sampling of 15 Pages

2.2 Talmudic Tokenization: Obstacles

As mentioned in the introduction, the development of a tokenizer for the Talmud Bavli involves various considerations. Computational linguistics, historical philology, and low-resource language processing are all fields of interest here.

Unlike most large monolingual corpora, the Talmudic corpus is

1. characterized by a dense interplay of three distinct linguistic strata: Biblical Hebrew (further: BH), Mishnaic Hebrew (further: MH), and Jewish Babylonian Aramaic (JBA). These layers aren't merely adjacent in the text. They are interwoven through intrasentential code-switching, where a single sentence may begin with a Mishnaic legal citation, transition into Aramaic dialectic analysis, and conclude with a Biblical proof-text. (Sometimes code-switching occurs even within one word!)

2. comprised of *semitic languages*, which are decidedly *non-agglutinizing languages*. By comparing various subword boundaries, an algorithm may discern that 'walk', 'walked', 'walker', and 'walking' share a four-letter verbal root which *begins* all four words. However, in Semitic languages, adjacency is less relevant. A three-letter root (e.g. H-L-K, meaning "to walk") is employed, and various vowels and consonants may serve as *infixes* which change the meaning. Hence the alignment between two words related by the same root is not easy to discern.

The canonical algorithm for subword analysis, the Byte-Pair Encoding (BPE) algorithm, provides a concrete example. BPE executes merges on frequent character pairs, and as such, applying BPE to the Talmudic corpus would risks "breaking" these roots in linguistically arbitrary ways. It would thereby obscure the semantic core of many words from the downstream model.

For example, we might have the lexeme *'ukshehalakhti'*, in which the root is hidden at positions 5,7,9 (0-indexed): u (and) + ke (when) + she (that) + halakh (walked, root H-L-K) + ti (first-person singular marker.) With BPE segmentation, we might obtain uk + she + ha + lak + hti, in which the root H-L-K is shattered. The token lak might be statistically frequent (appearing, for example, in the common Talmudic word 'lakhen', "therefore"), but the word 'lakhen' carries no semantic relation to "walking" at all. This phenomenon, we conjecture, would degrade the performance of downstream embeddings because the model must relearn that *lak* in this context means "walk" while in others it means something else entirely.

Further prospective problems include:

1. **Vocabulary collision** - the three languages of the Talmud share a significant number of cognates and identical orthographic forms, yet these forms often carry different semantic or syntactic loads. The root K-T-V (write), for example, appears in all three. In Biblical Hebrew, *katav* is a perfect verb. In Aramaic, *katva* might be a participle or a noun. If a tokenizer is trained on the full corpus, it might merge tokens based on the high-frequency Hebrew forms, leaving the Aramaic forms under-represented or awkwardly segmented. This is known as allocation bias in multilingual models. If instead we train three tokenizers separately, one for each language, the model loses the ability to transfer knowledge about the shared root K-T-V. The embedding for the Hebrew "write" and Aramaic "write" will be orthogonal, requiring the model to learn the concept of "writing" twice.

2. **Orthographic variance**. Talmudic texts unfortunately (for our research, at least) have orthographic inconsistency regarding letters used as vowels, known as *matres lectionis*. In Hebrew and Aramaic, the same word can be written with or without these letters. A tokenizer that is too rigid will treat these as separate entities.

2.3 Tokenization: A short literature review

2.3.1 Canonic algorithms: BPE, WordPiece, Unigram LM

Three canonical algorithms exist for tokenization. The most prominent is BPE (byte-pair encoding), developed in 1994 by Phillip Gage as a compression algorithm [4] and then adapted for tokenization by Sennrich et al. (2016.) [5] The algorithm iteratively finds the most frequent pair of adjacent tokens (initially characters) in a text corpus and merging them into a new, single token. This process is repeated a set number of times to create a vocabulary of subword units; as mentioned above, subword tokenizations balance the benefits of word-level and character-level tokenization (especially for OOV applications.)

The wordpiece algorithm, instead of choosing the most frequent set of tokens to merge, chooses merges based on which one most increases the language model

probability of the tokenization. (I.e., where BPE maximizes $P(A \cdot B)$, where the dot is concatenation of two subwords A and B, Wordpiece maximizes

$$\frac{P(A \cdot B)}{P(A) \times P(B)}$$

).

The Unigram algorithm, developed by Kudo et. al in 2018 as a portion of the SentencePiece pipeline [6], reverses the order utilized in BPE and WordPiece. The algorithm starts with an extensive vocabulary (usually, this is all characters and common substrings; efficient algorithms such as Enhanced Suffix Array algorithm described in [7] and quoted in [6] can be implemented to calculate the most frequent substrings in $O(T)$ time, where T is the size of the corpus.) From this vocabulary, it then *removes* tokens in the following way: at each step, it uses the Expectation-Maximization (EM) algorithm to estimate the probability of each token in the current vocabulary. It then calculates how much the overall likelihood (loss) of the training data would increase if a specific token were removed. The tokens that least affect the loss are considered least needed and are pruned. These steps are repeated until the desired vocabulary size is reached. (Note that the base characters of the language are always kept, to ensure all text in given language can be tokenized.)

2.3.2 SOTA algorithms

BPE, WordPiece, and Unigram LM constitute the "canonical" set of tokenizers; these share the attribute that they rely only on frequency and probabilities taken over the entire corpus, and language-blind. In recent years, NLP research has moved toward implementing context-aware and linguistically motivated algorithms. For morphologically rich languages such as the Semitic languages, then, it stands to reason that these might perform better.

TokenMonster: Ungreedy Optimization/Capcode

TokenMonster is a tokenization algorithm designed to primarily address the limitations of BPE considered from the perspective of its *greediness*. (BPE is "greedy" insofar as it merges the most frequent pair at every step and locks the merge in; e.g. if merging 't' and 'h' prevents the formation of a better token later in the sequence, BPE does not consider this.) TokenMonster (and similar approaches like "SuperBPE" or optimal transport variations thereof), on the contrary, aims for an ungreedy tokenization process by identifying the optimal set of tokens that covers the text with the fewest number of units (or highest probability) by considering multiple potential segmentation paths simultaneously.

The primary advantage of TokenMonster is inference efficiency and vocabulary compression. It can represent the same text with fewer tokens than BPE. (For downstream tasks such as crafting LLMs, this means a bigger context window; since long Talmudic discussions can span multiple folios, extending the effective context window is a critical architectural benefit if language generation is considered.

Semitic Root Encoding (SRE): Morphologically Informed Architectures

SRE, unlike the other algorithms we have surveyed thus far, is not a generic compression algorithm. Instead, it is a technique designed specifically for tokenization of Semitic languages[8]. SRE rejects the premise that words are best parsed as linear sequences of characters (indeed, in Aramaic and Hebrew, the root system, as described above, ensures that they are not.) Instead, text is preprocessed to separate the root from the template. Ideally, one would have a rule-based morphological analyzer to serve as a heuristic for this process. These abstract units would then be tokenized via BPE.

While this approach is probably the most "rigorous" from a philological perspective, insofar as it ensures that the model treats 'katav', 'yikhtov', and 'mikhtav' as semantically linked variants of 'K-T-B', it relies on a high-quality morphological analyzer, which simply doesn't exist. (We describe below the heuristics which we used to separate the root from the template.)

2.4 Metrics

How to evaluate these different algorithms? We provide comprehensive mathematical definitions for the evaluation metrics we utilized in tokenizer assessment. Definitions for NSL, fertility, and Zipfian alignment can be found in Jurafsky's introductory NLP textbook. [9]

2.4.1 Renyi Entropy

Renyi entropy is a generalization of Shannon entropy that measures the diversity or uniformity of a probability distribution. For tokenizer evaluation, we use Renyi entropy to assess how uniformly our tokens are distributed across the generated vocabulary. Usage of the Renyi metric was introduced by Zouhar et al. in their 2024 paper[10], in which the value $\alpha = 2.5$ is empirically suggested for the order of the Renyi entropy.

The mathematics:

Given a discrete random variable X with probability distribution $P = \{p_1, p_2, \ldots, p_n\}$ where p_i is the probability of token i, the Renyi entropy of order α is defined as:

$$H_\alpha(X) = \frac{1}{1-\alpha} \log_2 \left(\sum_{i=1}^{n} p_i^\alpha \right) \tag{2.1}$$

where $\alpha \geq 0$ and $\alpha \neq 1$. For tokenizer evaluation, we use $\alpha = 2.5$ to penalize distributions where a few tokens dominate. In order to calculate the Renyi entropy, we collect all tokens from the test corpus: $T = \{t_1, t_2, \ldots, t_N\}$ where N is the total number of tokens; count token frequencies: $f_i = \text{count}(t_i)$ for each unique token i; compute the probabilities p_i; and then use the above equation 2.1.

Higher values indicate a more uniform token distribution (i.e. better vocabulary utilization); lower values indicate that a few tokens dominate the distribution.

2.4.2 Gini coefficient

The Gini coefficent is a measure of inequality; generally utilized in economics, it ranges from 0 (perfect equality) to 1 (perfect inequality.) Intuitively, the coefficient is calculated by comparing the observed distribution (as a Lorenz curve) to a line of perfect equality. Meister et al. [11] recently adapted the metric for NLP use cases, especially qua tokenizer evaluation. The Gini coefficient is used to quantify the concentration of tokens; a high value (i.e. approaching 1) would indicate that a small subset of the vocabulary accounts for the majority of the token counts. This in turn suggests inefficient vocabulary utilization. A value approaching 0 would imply that tokens are used uniformly across the corpus.

Mathematical formulation: Given a corpus tokenized into a total of N tokens with a fixed vocabulary size V, let x_i represent the frequency of the unique token i with the given corpus over $i = 1, \ldots V$. The Gini coefficient is then calculated using the relative mean difference:

$$G = \frac{\sum_{i=1}^{V} \sum_{j=1}^{V} |x_i - x_j|}{2V^2 \bar{x}} \tag{2.2}$$

where

$$\bar{x} = \tfrac{N}{V}$$

is the mean token frequency. (An alternative method of calculation: use the formula for discrete data sorted in non-decreasing order, i.e. $x_1 \leq x_2 \leq \cdots \leq x_V$:

$$G = \frac{1}{V}\left(V + 1 - 2\frac{\sum_{i=1}^{V}(V+1-i)x_i}{\sum_{i=1}^{V} x_i}\right) \tag{2.3}$$

(Definition from [12].)

2.4.3 Normalized Sequence Length (NSL)

Normalized Sequence Length is a metric which measures the efficiency of compression; it compares the number of tokens produced to the number of characters in the original text. (A lower NSL metric indicates better compression.)

Mathematical formulation: Given a corpus of texts $\mathcal{C} = \{T_1, T_2, \ldots, T_m\}$ where each text T_i is tokenized into a sequence of tokens NSL is defined as:

$$\text{NSL} = \frac{\sum_{i=1}^{m} |\text{tokens}(T_i)|}{\sum_{i=1}^{m} |T_i|} \tag{2.4}$$

where $|\text{tokens}(T_i)|$ is the number of tokens produced from text T_i, and $|T_i|$ is the number of characters in text T_i (including spaces).

Lower NSL indicates better compression (fewer tokens per character); NSL equivalent to 1.0 means one token per character (no compression); NSL < 1.0 indicates compression (fewer tokens than characters), while NSL > 1.0 indicates expansion (more tokens than characters; to be avoided.)

2.4.4 Fertility

Fertility measures the average number of tokens produced per word in the original text. It provides a quantified measure of how efficiently a given tokenizer segments words into subword units. **Mathematical definition**: given a corpus $\mathcal{C} = \{T_1, T_2, \ldots, T_m\}$, fertility is defined as:

$$\text{Fertility} = \frac{\sum_{i=1}^{m} |\text{tokens}(T_i)|}{\sum_{i=1}^{m} |\text{words}(T_i)|} \tag{2.5}$$

where, as above, $|\text{tokens}(T_i)|$ is the number of tokens produced from text T_i, and $|\text{words}(T_i)|$ is the number of words in text T_i (split on whitespace)

A fertility of 1.0 means one token per word (word-level tokenization, known as perfect). A fertility > 1.0 indicates subword tokenization (words split into multiple tokens).

2.4.5 Zipfian Alignment

Zipfian alignment measures how closely the token frequency distribution adheres to Zipf's law, the fundamental result in linguistics which states that the frequency of a token is inversely proportional to its rank. The metric uses the coefficient of determination (R^2) from a log-log linear regression.

Zipf's law states that for a corpus, the frequency f_r of the r-th most frequent token is approximately:

$$f_r \propto r^{-s} \tag{2.6}$$

i.e.

$$f_r = \frac{C}{r^s} \tag{2.7}$$

where r is the rank (1 is most frequent, 2 is second most frequent, etc.), s is the Zipf exponent (close to 1, usually, for natural language), and C is a constant.

Taking the logarithm of both sides allows us to formulate this as a linear relationship:

$$\log f_r = \log C - s \log r \tag{2.8}$$

I.e., $\log f = a + b \log r$ where $a = \log C$ and $b = -s$.

Calculating Zipfian Alignment

Given the series of token frequencies sorted in descending order: $\mathbf{f} = [f_1, f_2, \ldots, f_n]$ where $f_1 \geq f_2 \geq \ldots \geq f_n$, with corresponding ranks $\mathbf{r} = [1, 2, \ldots, n]$, first map to log space:

$$\mathbf{y} = [\log f_1, \log f_2, \ldots, \log f_n] \tag{2.9}$$

$$\mathbf{x} = [\log 1, \log 2, \ldots, \log n] \tag{2.10}$$

Then fit linear regression $y = a + bx$ using least squares:

$$b = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.11}$$

$$a = \bar{y} - b\bar{x} \tag{2.12}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. Lastly, calculate R^2 (coefficient of determination):

$$SS_{\text{res}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - (a + bx_i))^2 \tag{2.13}$$

$$SS_{\text{tot}} = \sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{2.14}$$

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \tag{2.15}$$

where $\hat{y}_i = a + bx_i$ are the predicted values. The presumed Zipf exponent is: $s = -b$ If R^2 is near 1.0 the indication is that the data has a strong adherence to Zipf's law (typical for natural language).

2.4.6 Vocabulary Utilization

Vocabulary utilization measures what percentage of the tokenizer's vocabulary is *actually used* when encoding the test corpus, i.e. how efficiently the vocabulary space is being utilized.

Mathematical definition: Given a tokenizer with vocabulary size V and a test corpus $\mathcal{C}$, vocabulary utilization is:

$$\text{Vocab Utilization} = \frac{|\{t : t \in \text{tokens}(\mathcal{C}) \text{ and } t \in \text{vocab}\}|}{V} \tag{2.16}$$

where tokens($\mathcal{C}$) is the set of all tokens produced when encoding the corpus, vocab is the tokenizer's vocabulary, and $V = |\text{vocab}|$ is the vocabulary size. Put differently, if U is the number of unique tokens actually used

$$\text{Vocab Utilization} = \frac{U}{V} \tag{2.17}$$

Higher values (i.e. closer to 1.0 or 100%) indicate efficient vocabulary usage; lower values indicate that much of the vocabulary is unused (wasteful).

Chapter 3

Methodology

3.1 Data Preparation

The text of the entire Talmud (Vilna edition), including vocalization and punctuation, was downloaded using the Sefaria API developed by http://www.sefaria.org. While Sefaria's Talmud has been vocalized by experts, in hoping to make a model which might generalize and be able to understand unvocalized text, we filtered out punctuation and vocalizations, as well as errant html styling tags from the corpus. The following is the first pericope of the Talmud (*Masekhet Berakhot*, 2a):

מֵאֵימָתַי קוֹרִין אֶת שְׁמַע בָּעַרְבִין? מִשָּׁעָה שֶׁהַכֹּהֲנִים נִכְנָסִים לֶאֱכוֹל בִּתְרוּמָתָן. עַד סוֹף הָאַשְׁמוּרָה הָרִאשׁוֹנָה. דִּבְרֵי רַבִּי אֱלִיעֶזֶר. וַחֲכָמִים אוֹמְרִים: עַד חֲצוֹת. רַבָּן גַּמְלִיאֵל אוֹמֵר עַד שֶׁיַּעֲלֶה עַמּוּד הַשַּׁחַר. מַעֲשֶׂה וּבָאוּ בָנָיו מִבֵּית הַמִּשְׁתֶּה, אָמְרוּ לוֹ: לֹא קָרִינוּ אֶת שְׁמַע, אָמַר לָהֶם: אִם לֹא עָלָה עַמּוּד הַשַּׁחַר חַיָּיבִין אַתֶּם לִקְרוֹת. וְלֹא זוֹ בִּלְבַד אָמְרוּ, אֶלָּא כָּל מַה שֶׁאָמְרוּ חֲכָמִים ״עַד חֲצוֹת״, מִצְוָתָן עַד שֶׁיַּעֲלֶה עַמּוּד הַשַּׁחַר. הֶקְטֵר חֲלָבִים וְאֵבָרִים, מִצְוָתָן עַד שֶׁיַּעֲלֶה עַמּוּד הַשַּׁחַר וְכָל הַנֶּאֱכָלִים לְיוֹם אֶחָד, מִצְוָתָן עַד שֶׁיַּעֲלֶה עַמּוּד הַשַּׁחַר. אִם כֵּן, לָמָּה אָמְרוּ חֲכָמִים ״עַד חֲצוֹת״ — כְּדֵי לְהַרְחִיק אָדָם מִן הָעֲבֵירָה

After processing:

מאימתי קורין את שמע בערבין משעה שהכהנים נכנסים לאכול בתרומתן עד סוף האשמורה הראשונה דברי רבי אליעזר וחכמים אומרים עד חצות רבן גמליאל אומר עד שיעלה עמוד השחר מעשה ובאו בניו מבית המשתה אמרו לו לא קרינו את שמע אמר להם אם לא עלה עמוד השחר חייבין אתם לקרות ולא זו בלבד אמרו אלא כל מה שאמרו חכמים עד חצות מצותן עד שיעלה עמוד השחר הקטר חלבים ואברים מצותן עד שיעלה עמוד השחר וכל הנאכלים ליום אחד מצותן עד שיעלה עמוד השחר אם כן למה אמרו חכמים עד חצות כדי להרחיק אדם מן העבירה גמ׳ תנא היכא קאי דקתני מאימתי The full Babylonian Talmud corpus comprises 88,958 sentences and 1,922,740 words.

3.2 Considering Joint vs. Separate Encodings

The question of whether to attempt joint encoding or separate encoding (discussed in section: Literature Review, above) hinges on the availability of data. No heuristic exists for separating between the intermingled layers of Biblical Hebrew, Mishnaic Hebrew, and Jewish Babylonian Aramaic as they appear in the text. Our idea was to source as much high-quality annotations of these texts as we could, and train a model based on such data.

As a first attempt, we developed and shipped a crowd-sourced annotation editor. Built on a Node.js background and hosted on a Vercel instance, the 'Talmudic Annotation Platform' leverages the Sefaria API and allows for real-time editing of the Talmudic page. The user flow begins at the dashboard page (see **Figure 3.1**); by selected a page to annotate, the user is met with the annotation editor. Annotations

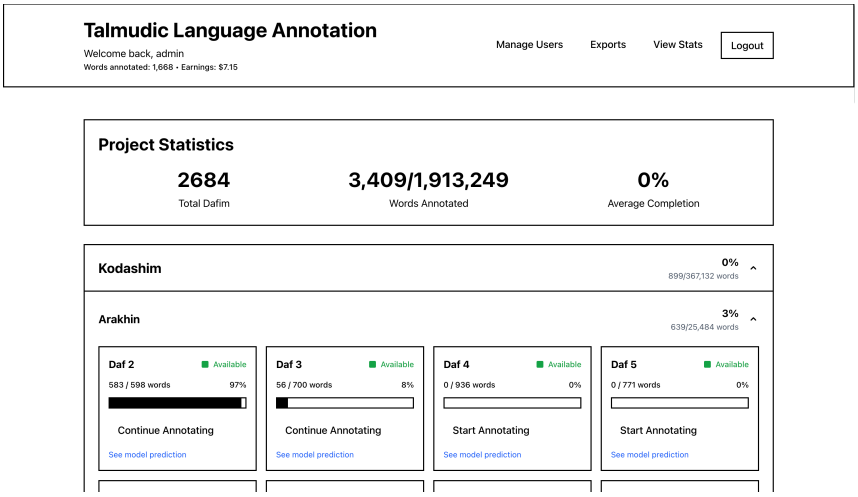

Figure 3.1: Dashboard for Annotation Platform

are added visually, by marking up the text with highlights; the type (viz. color) of highlight is selected by clicking on the appropriate colored box on the top-left of the toolbar above the main text.

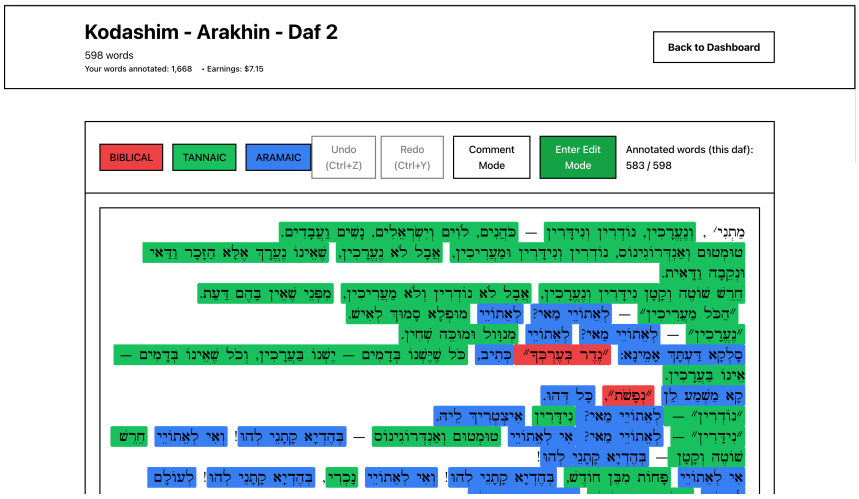

Figure 3.2: Annotation of Talmudic Page

The data format used inline tagging of various lines; all such a method requires is to delimit segments with newlines, and prepend "LABEL::" to each line. (E.g. "BIBLICAL:: ...").

The data is sent to an endpoint exposed on a HuggingFace model intended to train the language classifier model. Results are sent back to the Talmudic Annotator Platform using an exposed POST endpoint.

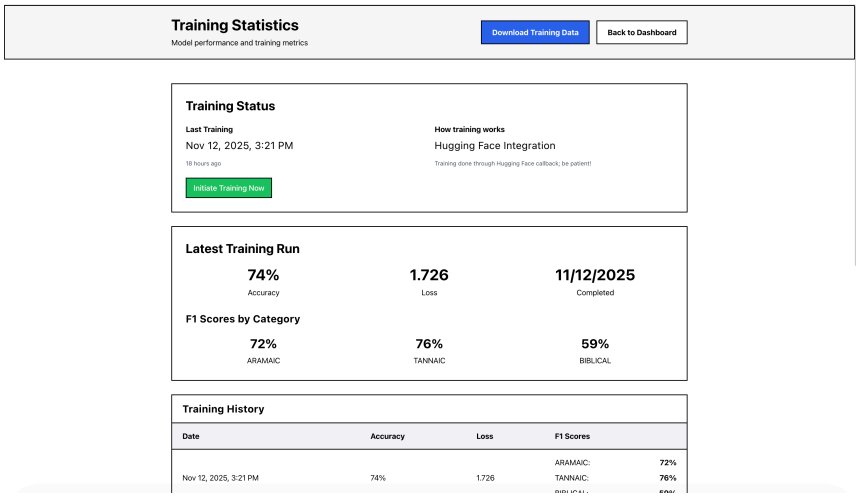

Figure 3.3: Model Integration

The HuggingFace setup is depicted in **Figure 3.4**:

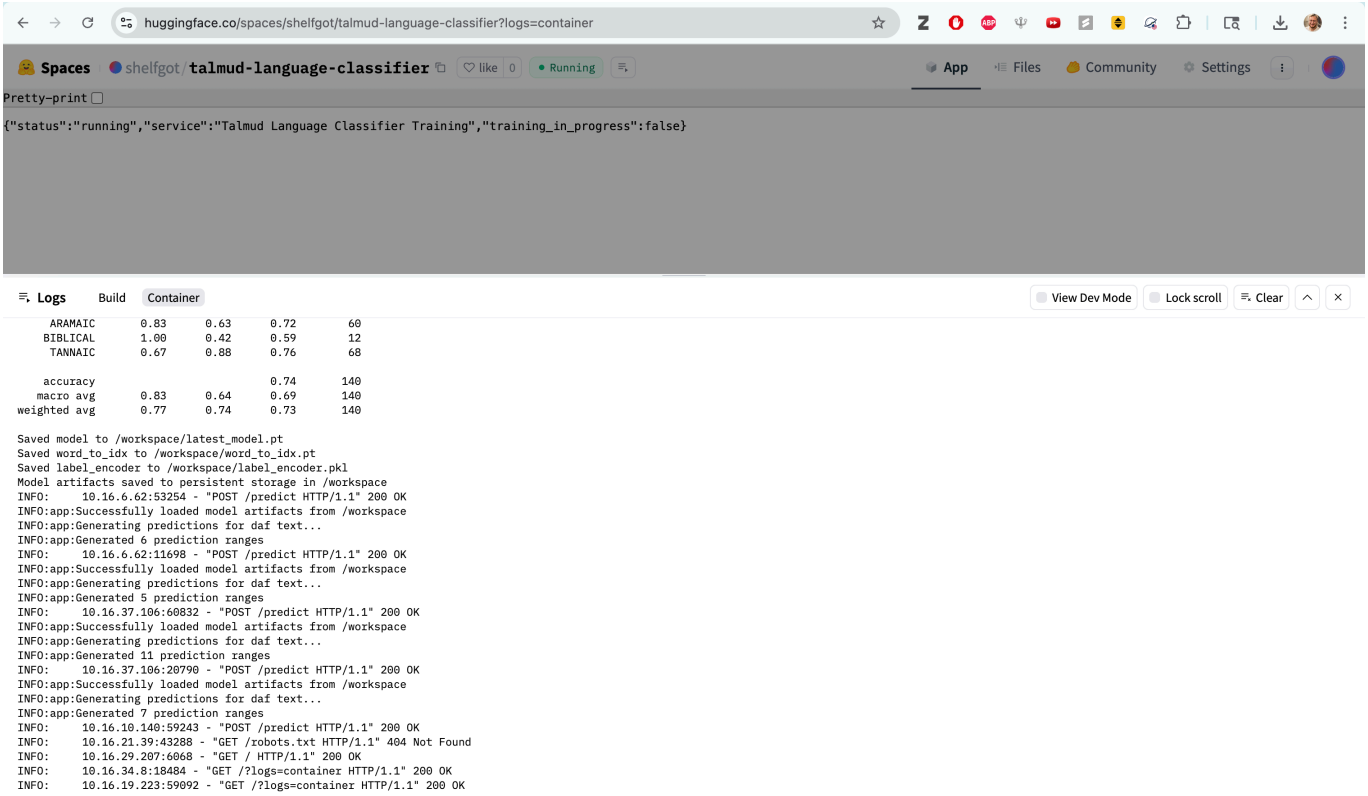

Figure 3.4: HuggingFace integration

Input Processing: The model takes tokenized text sequences (batch $\times$ 100 tokens) and converts them into dense vector representations through an embedding layer (128 dimensions).

Bidirectional LSTM Layers: Two stacked BiLSTM layers are used to process the sequences in both forward and backward directions. Each direction outputs 128-dimensional representations, which are concatenated to produce 256-dimensional outputs. The first layer processes the entire sequence, while the second layer extracts only the final hidden states, creating a fixed-size representation of the input text.

Classification head: The final hidden states (256 dimensions) pass through a dropout layer for regularization, then through two fully connected layers. The first reduces the dimensionality of the data to 128 with ReLU activation and dropout, and the second projects to 3 dimensions, one for each language class (logits.)

Output: model output is class logits that can be converted to probabilities using softmax, indicating the likelihood that the input text is Biblical Hebrew, Tannaitic Hebrew, or Aramaic.

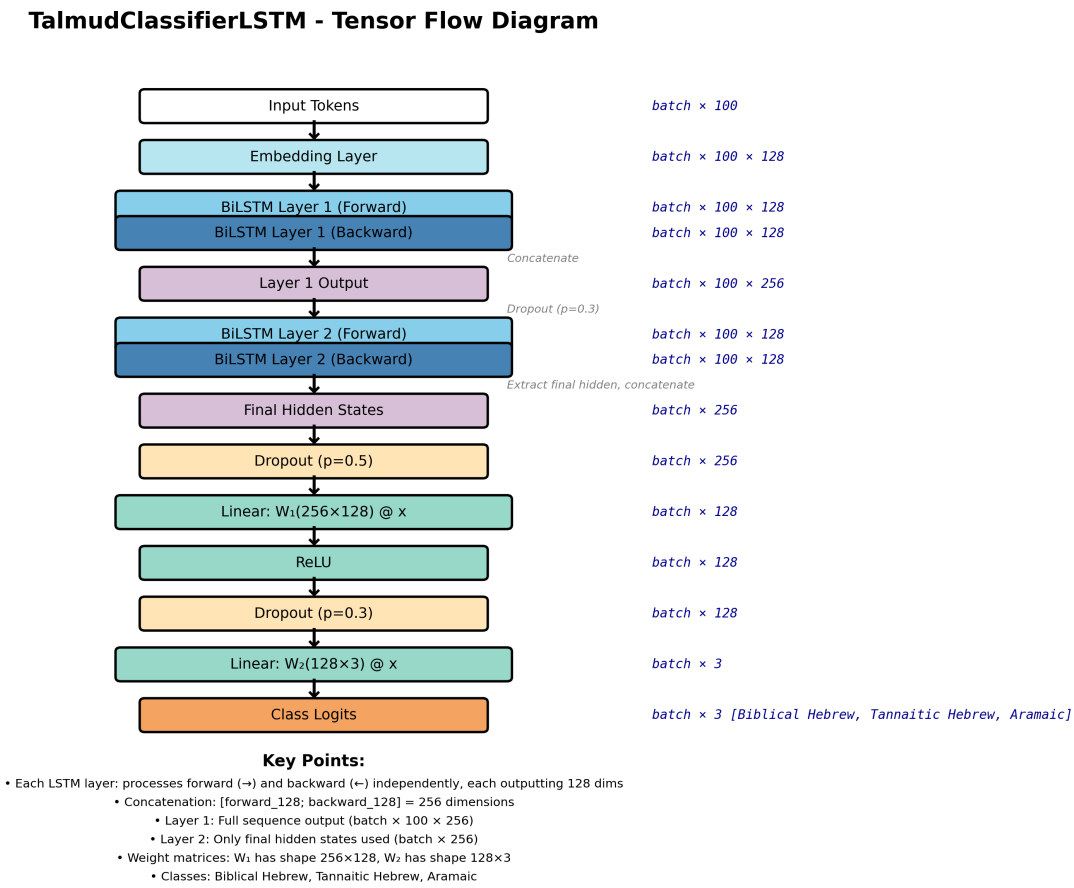

Figure 3.5: biLSTM Model

With the paucity of annotators available, and the statistics of the model accuracy approaching only 74% accuracy, far less than needed, we pivoted to consider the unified methods early on. These only required the preprocessing of the text as outlined above.

3.3 Comparing Tokenization Algorithms

We tested six different tokenization algorithms: **BPE, WordPiece, Unigram LM, TokenMonster**, **SRE**, and a hybrid **TokenMonster-SRE** implementation (**TmSRE**.) Performance was assessed using intrinsic metrics as well as downstream capabilities.

3.3.1 Selection of Vocabulary Size

Vocabulary size selection process began with statistical analysis of the corpus. We ran the following analytical tools on the corpus:

1. Basic statistics: total tokens, unique words, mean, median, standard deviation, maximum, and minimum frequency.

2. Frequency distribution by range, over the following categories: hapax legomena (once), dis legomena (twice), very rare (3-5), rare (6-10), uncommon (11-50), moderate (51-100), common (101-500), very common (501-1000), frequent (1001-10000), and extremely frequent (10000+).

3. Frequency distribution (granular.)

4. Calculation of Zipf's Law: fit frequency to C/rank^{α}, where rank is the index of the word in the ordering from most common to least.

5. Vocabulary coverage analysis: tokens needed to cover X% of corpus.

See **Figure 3.6** for a summary of results. The full results, tabulated, may be found in the appendix.

Analysis of Statistics

Heaps' Law, used to predict the amount of unique tokens (i.e. the dictionary) as a function of corpus size, has:

$$|V| = kN^{\beta}$$

where V is the vocabulary, $k \in [10, 100]$ (generally), and $\beta \in [0.44, 0.56]$ or even higher [9]. Using `scipy`'s optimization package, we find that the best constrained-fit parameters are:

$$K = 10.00, \ = 0.6732, R^2 = 0.9966$$

This is a very aggressive growth law compared to the natural language reported upper bound quoted above, 0.56. Coupled with the fact that about 49% of types are hapax legomena, and over 80% occur less than 5 times, we must conclude that we have what is known as a heavy-tail pattern on our hands. That is, a huge fraction of word types occur only once or a few times; each of those rare words would need its own embedding and output-row parameters, but the model would see almost no examples to learn them. Hence pushing all the way to the full 166k word-level vocabulary is mostly paying parameters for low-frequency noise rather than additional coverage.

With this preliminary analysis in hand, we decided to empirically test different vocabulary amounts.

Vocabulary Size Experiments

The setup was intended to serve as a quick snapshot of the vocabulary. Hence we used the three *quickest* algorithms in terms of computational overload, SRE, TokenMonster, and Unigram, in order to test a broad range of vocabularies.

- Corpus: 50% of the Babylonian Talmud (71,166 segments)

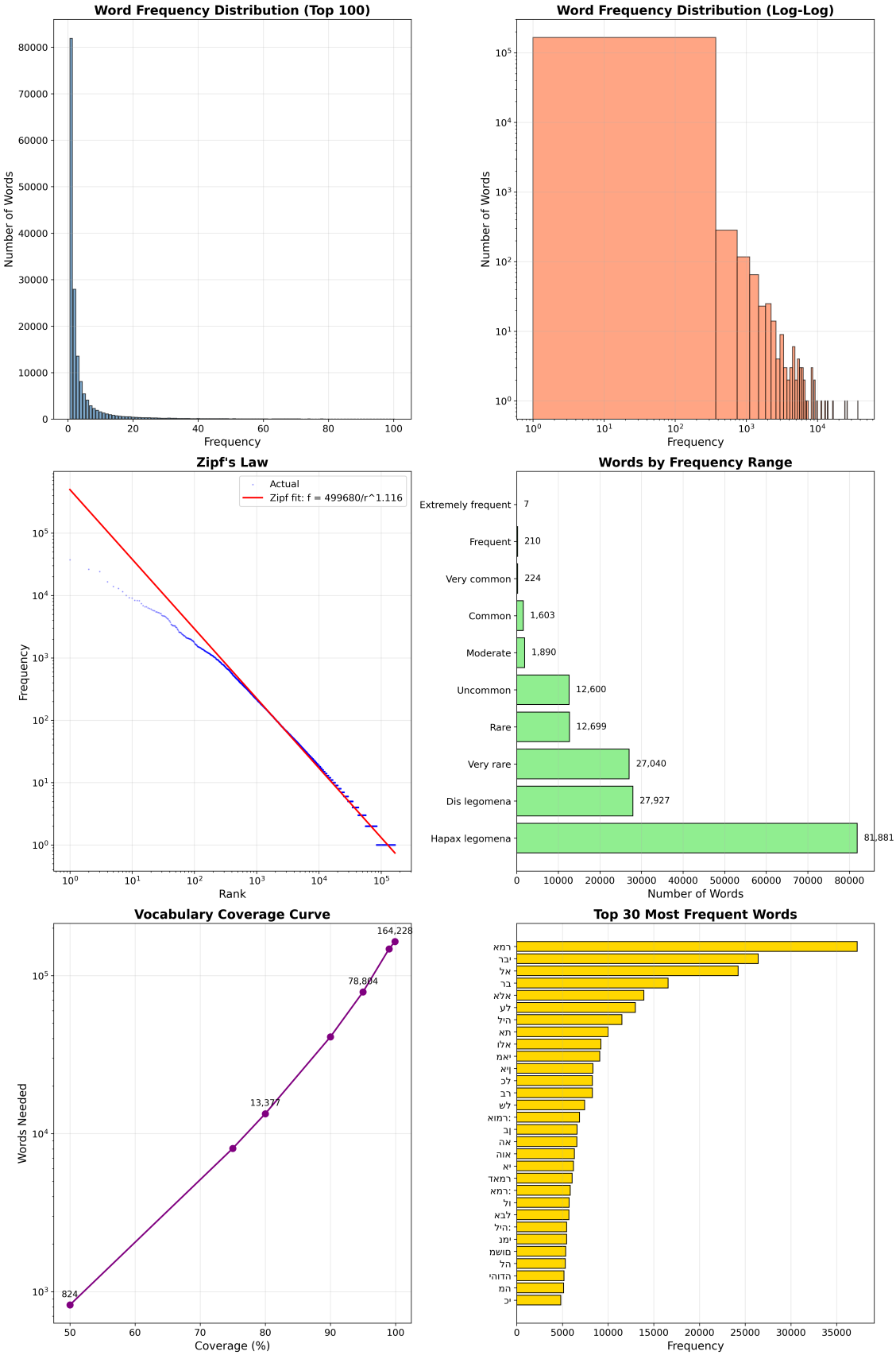

Figure 3.6: Frequency Analysis

- Algorithms tested: SRE, TokenMonster, Unigram
- Vocabulary Sizes: 2,000, 4,000, 8,000, 12,000, 16,000, 20,000, 24,000, 32,000, 40,000, 48,000, 64,000
- **Total Configurations**: 33 (11 vocab sizes × 3 algorithms)

The key metrics we tracked were the ones enumerated above as the basis for comparisons between tokenizers, viz. Renyi entropy, NSL (Normalized Sequence Length), fertility, Zipfian alignment, and vocabulary utilization.

Results were as follows.

SRE: fertility was constant at 1.0 (perfect 1:1 word-to-token mapping), while NSL was constant at 0.197 (an excellent compression ratio.) This was to be expected: since SRE uses a deterministic word-based approach, vocabulary size has minimal impact. Vocabulary utilization was 83–98% (fairly efficient, over all sizes.) Renyi entropy was relatively low — 1.27–1.30, and showed only moderate improvement. Zipfian alignment showed moderate improvement as well. Hence no real benefit was accrued from sizes larger than what must be presumed is the morphological base set size, i.e. 2000.

Full results can be found in the appendix.

TokenMonster: fertility was constant at ∼4.12 (4.12 tokens per word on average). NSL, as expected, was very poor, but furthermore constant: ∼0.812 (As explained above, the poor token size compression exhibited by TokenMonster is a direct result of the very long tokens TokenMonster seeks.) Zipfian alignment improved moderately as well 0.52–0.61; essentially, vocabulary size past 2000 had almost no effect on performance.

Full results can be found in the appendix.

Unigram: The algorithm most responsive to vocabulary size which we tested. Fertility decreased from 2.48 to 1.54 as vocabulary increased, and NSL decreases from 0.490 to 0.303. Vocabulary utilization decreased with larger vocabulary, and spanned 26–38%. Renyi entropy increased from 4.00 to 6.80. Zipfian alignment improved from 0.90 to 0.97 with vocabulary size. Most of these improvements tapered off after **32k words** with orders of magnitude greater computational expense to double the amount of merges.

Hence **32k words** was selected as the target vocabulary for testing. Even if this is not the optimal amount, for the *comparative analysis* we seek, it is valuable insofar as it allows for the six algorithms we tested to produce a tenable tokenization.

3.3.2 Common Framework

All tokenizers in our experimental framework inherit from an abstract class, which provides a unified interface and shared preprocessing pipeline. This ensures consistency across algorithms and proper handling of Hebrew/Aramaic text characteristics.

BaseTokenizer Interface

The {BaseTokenizer} class defines three core abstract methods that each algorithm implements:

- **`train(texts: List[str], kwargs)`**: Builds the vocabulary and learns tokenization patterns from training corpus
- **`encode(text: str, dropout: bool = False) -> List[int]`**: Converts input text to a sequence of token IDs
- **`decode(token_ids: List[int]) -> str`**: Reconstructs text from token ID sequences

Shared Hyperparameters

All tokenizers accept two fundamental hyperparameters:

- **`vocab_size`** (default: 32,000): Target vocabulary size. This value balances between having sufficient tokens to represent the morphological richness of Hebrew/Aramaic while remaining computationally manageable. Higher vocabularies can capture more morphological variants but increase memory requirements.
- **`min_frequency`** (default: 2): Minimum occurrence frequency for a token to be included in the vocabulary. This threshold filters out rare tokens that are likely noise or errors, preventing vocabulary bloat from infrequent character combinations.

Text Preprocessing Pipeline

Prior to tokenization, all input text undergoes mandatory preprocessing to normalize Hebrew/Aramaic orthography and remove corpus-specific artifacts, as described above. Whitespace is also normalized; we collapse multiple spaces into single spaces and standardize line breaks, ensuring consistent word boundaries.

3.3.3 BPE (Byte-Pair Encoding)

The BPE training algorithm proceeds as shown in *Algorithm 2*. The encoding algorithm is described in *Algorithm 3*. As for hyperparameters, the vocabulary size used is 32,000, and the minimum frequency 2. Dropout rate, applied only when `use_bpe_dropout=True`, was set to 0.1, following Provilkov et al. (2020)'s recommendation to the effect that 10% dropout rate provides regularization benefits without excessive segmentation instability.

Algorithm 2: BPE Training Algorithm

Input: Training corpus C, vocabulary size V, minimum frequency θ
Output: Trained BPE vocabulary $\mathcal{V}$

```
Initialization:;
Extract all unique characters from C;
Initialize vocabulary V with character tokens;
Preprocess all texts (clean formatting, remove nekudot, normalize
  whitespace);
Word-Level Pre-tokenization:;
Split text on whitespace to obtain words;
Each word is initially represented as a sequence of characters;
Iterative Merging:;
while |V| < V and valid pairs exist do
    Count frequencies of all adjacent token pairs across all word splits;
    Identify the most frequent pair (t_i, t_j);
    if freq(t_i, t_j) ≥ θ then
        Create new token t_new = t_i · t_j;
        Update all word splits to replace (t_i, t_j) with t_new;
        Add t_new to V;
Vocabulary Finalization:;
Include all character tokens from initial vocabulary;
Include all merged tokens from training;
Add special tokens: [PAD], [UNK], [CLS], [SEP], [MASK];
return V;
```

Implementation Notes

As noted above, the implementation maintains a dictionary of word splits throughout training, updating them incrementally as pairs are merged. This avoids recomputing splits from scratch in each iteration, significantly improving efficiency for large corpora. For Hebrew/Aramaic text, the character-level initialization properly handles right-to-left scripts; this is because BPE operates on character sequences regardless of directionality. The whitespace-based pre-tokenization aligns with standard and Aramaic Hebrew text conventions where spaces delimit words.

3.3.4 WordPiece

The training procedure for WordPiece is given in *Algorithm 4*, while the encoding procedure is given in *Algorithm 5*.

Algorithm 3: BPE Encoding Algorithm

Input: Input text T, trained vocabulary $\mathcal{V}$, merge rules $\mathcal{M}$, dropout rate δ (optional)

Output: Token ID sequence **ids**

```
Apply preprocessing pipeline to T;
Split T into words W = {w_1, w_2, ..., w_n} on whitespace;
ids ← [];
foreach word w ∈ W do
    Initialize token sequence tokens with individual characters of w;
    foreach merge rule m ∈ M do
        if random() > δ or dropout disabled then
            Apply merge m greedily to tokens (combine pairs when
              encountered);
    Convert tokens to IDs via vocabulary lookup:
      ids_w ← lookup(tokens, V);
    Append ids_w to ids;
return ids;
```

Hyperparameters

Hyperparamaters were the same as above (32k vocab size, minimum frequency of 2.) The standard BERT convention of using the `continuing_subword_prefix`: `"##"` (fixed) marks tokens that cannot appear at word boundaries, which simplifies decoding.

Implementation Notes

As discussed above, the likelihood-based scoring function prioritizes merges where the two tokens frequently co-occur relative to their individual frequencies; the longest-match encoding ensures deterministic tokenization and maintains compatibility with general BERT tokenization schemes. For Hebrew/Aramaic, this approach can capture common prefixes and suffixes as separate tokens (as opposed to generating them from combinations of single characters.) As noted before, an ability to capture prefixes and suffixes is very beneficial for the Semitic morphological system.

3.3.5 Unigram Language Model

The training procedure for the unigram language model is provided in *Algorithm 6*, and the encoding procedure in *Algorithm 7*. Hyperparameters were kept constant (i.e. 32k vocabulary and minimum frequency of 2.) [Unigram's probabilistic pruning is particularly effective at maintaining quality given a vocabulary size; as we shall see diminishing returns on bumping up vocabulary size began far before

Algorithm 4: WordPiece Training Algorithm

Input: Training corpus C, vocabulary size V, minimum frequency θ
Output: Trained WordPiece vocabulary $\mathcal{V}$

Initialization:;
Extract character vocabulary from C;
Initialize word splits with special handling:;
    First character of each word remains as-is;
    Subsequent characters receive `##` prefix (continuing subword marker);

Likelihood-Based Merging:;
while $|\mathcal{V}| < V$ **do**
    foreach *adjacent pair* (t_i, t_j) **do**
        Compute score: $score(t_i, t_j) = \frac{freq(t_i,t_j)}{freq(t_i) \times freq(t_j)}$;
    Select pair (t_i^*, t_j^*) with highest score;
    if $freq(t_i^*, t_j^*) \geq \theta$ **then**
        Merge the pair, removing `##` prefix from second token;
        Add merged token to $\mathcal{V}$;

Vocabulary Finalization:;
Include all character tokens (with and without `##` prefix);
Include all merged tokens;
Add special tokens;
return $\mathcal{V}$;

32k for the Unigram algorithm.] **`max_ngram_length`** was set to 4; this limits the maximum token length, preventing overly long tokens that are unlikely to generalize. For Hebrew/Aramaic, with their triconsonsontal root structure (coupled with a fourth character indicating gender and tense), 4 characters seemed a good compromise - capturing common morphemes while remaining computational tractable. **`num_iterations`** was set to 5. Empirical testing showed that 5 EM (expectation maximization) iterations provide stable probability estimates, and more iterations yielded diminishing returns.

Implementation Notes

The Viterbi decoding uses log-probabilities to avoid numerical underflow. Tokens not in vocabulary during encoding are assigned a small probability (1e-10) to ensure valid segmentation.

As noted above, for Hebrew/Aramaic, the n-gram initialization naturally captures common prefixes, roots, and suffixes as separate tokens, which aligns well with Semitic morphology; the probabilistic pruning then selects the most useful of these morphological units.

The EM algorithm's iterative refinement helps the model learn which character

Algorithm 5: WordPiece Encoding Algorithm (Longest-Match Strategy)

Input: Input text T, trained vocabulary $\mathcal{V}$
Output: Token ID sequence **ids**

```
Apply preprocessing to T;
Split T into words W = {w_1, w_2, ..., w_n};
ids ← [];
foreach word w ∈ W do
    if w ∈ V then
        Append lookup(w, V) to ids;
    else
        i ← 0;
        while i < |w| do
            j ← |w|;
            found ← false;
            while j > i and not found do
                substr ← w[i : j];
                if substr ∈ V then
                    Append lookup(substr, V) to ids;
                    i ← j;
                    found ← true;
                else
                    j ← j − 1;
            if not found then
                Append [UNK] token ID to ids;
                i ← |w|;
return ids;
```

sequences are truly linguistically meaningful versus garbage co-occurrences, making it particularly well-suited for morphologically rich languages.

3.3.6 TokenMonster

The TokenMonster training algorithm implementation is given in *Algorithm 8*, and the encoding algorithm implementation in *Algorithm 9*.

Hyperparameters

Vocabulary size and minimum frequency set to 32,000 and 2, as with the others. `max_token_length` set to 10; this limits maximum token length to prevent overly long tokens that are unlikely to generalize. For Hebrew/Aramaic, 10 characters can

Algorithm 6: Unigram Language Model Training Algorithm

Input: Training corpus C, vocabulary size V, max n-gram length L, EM iterations I

Output: Trained Unigram vocabulary $\mathcal{V}$ with probabilities P

Vocabulary Initialization:;
Generate all character n-grams from length 1 to L;
Create initial vocabulary $\mathcal{V}_0$ from all unique n-grams found in C;

Probability Estimation via EM Algorithm:;
Initialize uniform probabilities: $P(t) = 1/|\mathcal{V}_0|$ for all $t \in \mathcal{V}_0$;
for *iter* $\leftarrow 1$ **to** I **do**
  E-step:;
  foreach *text* $T \in C$ **do**
    Find optimal segmentation using Viterbi algorithm;
    Compute expected token counts $E[count(t)]$ for all $t \in \mathcal{V}_0$;
  M-step:;
  foreach *token* $t \in \mathcal{V}_0$ **do**
    $P(t) \leftarrow \frac{E[count(t)]}{\sum_{t' \in \mathcal{V}_0} E[count(t')]}$;

Vocabulary Pruning:;
while $|\mathcal{V}_0| > V - 5$ **do**
  Find token $t^* = \arg\min_{t \in \mathcal{V}_0} P(t)$;
  Remove t^* from $\mathcal{V}_0$;
  Re-normalize probabilities: $P(t) \leftarrow \frac{P(t)}{\sum_{t' \in \mathcal{V}_0} P(t')}$ for all remaining t;
$\mathcal{V} \leftarrow \mathcal{V}_0$;

Vocabulary Finalization:;
Add special tokens: `[PAD]`, `[UNK]`, `[CLS]`, `[SEP]`, `[MASK]`;
return $\mathcal{V}$, P;

capture complete words or common phrases while remaining tractable. (The value selected must be consistent between vocabulary building and encoding to ensure all vocabulary tokens can be found during encoding.)

Implementation Notes

As described above, the cost-based approach naturally favors frequent, useful tokens. The inverse frequency cost function means that common substrings (which appear frequently) have low cost and are preferred during segmentation.

The dynamic programming approach ensures globally optimal segmentation rather than greedy local choices. This "ungreedy" property helps TokenMonster find better segmentations than algorithms that make locally optimal decisions.

Algorithm 7: Unigram Encoding Algorithm (Viterbi Segmentation)

Input: Input text T, trained vocabulary $\mathcal{V}$ with probabilities P
Output: Token ID sequence **ids**

```
Apply preprocessing to T;
Split T into words W = {w_1, w_2, ..., w_n};
ids ← [];
foreach word w ∈ W do
    n ← |w|;
    Initialize DP table: dp[0] ← 0, dp[i] ← −∞ for i > 0;
    Initialize backpointer: back[i] ← null for all i;
    for i ← 1 to n do
        for j ← 0 to i − 1 do
            substr ← w[j : i];
            if substr ∈ V then
                score ← dp[j] + log P(substr);
                if score > dp[i] then
                    dp[i] ← score;
                    back[i] ← j;
    Reconstruct optimal token sequence from back array;
    Convert tokens to IDs via vocabulary lookup;
    Append token IDs to ids;
return ids;
```

3.3.7 SRE (Semitic Root Encoding)

The SRE training algorithm implementation is given in *Algorithm 10*, and the encoding algorithm implementation in *Algorithm 11*.

Hyperparameters

Vocabulary size and minimum frequency set to 32k and 2, respectively, as with the others. As for allocating different portions of the vocabulary for various morphological types, we used the following:

- Roots: 40%; root-to-noun templates: 20%; prefixes: 10%; suffixes: 10%; subwords: 15%; special tokens: 5%

This is a rough reflection of linguistic importance in Semitic morphologhy; roots are primary semantic carriers, templates encode grammatical relations, and affixes are frequent but less semantically rich.

Algorithm 8: TokenMonster Training Algorithm

Input: Training corpus C, vocabulary size V, max token length L, minimum frequency θ

Output: Trained TokenMonster vocabulary $\mathcal{V}$ with costs c

```
Substring Frequency Collection:;
S ← ∅;
foreach text T ∈ C do
    for len ← 1 to L do
        foreach substring s of length len in T do
            Increment freq(s);
            Add s to S;
Filter S: keep only substrings with freq(s) ≥ θ (except single characters);
Token Cost Calculation:;
foreach substring s ∈ S do
    if |s| = 1 then
        c(s) ← 0.5;
    else
        c(s) ← 1/(freq(s)+1);
Vocabulary Selection:;
Sort S by frequency (descending), then by length (descending) for ties;
Select top V − 5 − |unique_chars| substrings from S;
Always include all unique characters for complete coverage;
Add special tokens;
V ← selected substrings ∪ unique characters ∪ special tokens;
return V, c;
```

Implementation Details

The heuristic root extraction uses consonant stripping and pattern matching; the following were used for each:

- Prefixes: ['ו', 'ב', 'כ', 'ל', 'מ', 'ש']
- Suffixes: ['ים', 'ות', 'ה', 'ו', 'י']

As noted, more sophisticated implementations would require a morphological analyzer, which currently does not exist for Jewish Babylonian Aramaic. (The diversity of forms, as described in Bar-Asher Siegal's textbook of *Jewish Babylonian Aramaic*, militates against a deterministic rule-based algorithm for parsing JBA [13].) The hierarchical encoding strategy ensures that words are always tokenizable; graceful degradation from morphological to subword to character-level encoding is implemented. For Hebrew/Aramaic, the explicit morphological decomposition aligns

Algorithm 9: TokenMonster Encoding Algorithm (Optimal Cost Segmentation)

Input: Input text T, trained vocabulary $\mathcal{V}$ with costs c
Output: Token ID sequence **ids**

```
Apply preprocessing to T;
Split T into words W = {w_1, w_2, ..., w_n};
ids ← [];
foreach word w ∈ W do
    n ← |w|;
    Initialize DP table: dp[0] ← 0, dp[i] ← ∞ for i > 0;
    Initialize backpointer: back[i] ← null for all i;
    for i ← 1 to n do
        for j ← 0 to i − 1 do
            substr ← w[j : i];
            if substr ∈ V then
                total_cost ← dp[j] + c(substr);
                if total_cost < dp[i] then
                    dp[i] ← total_cost;
                    back[i] ← j;
    if dp[n] < ∞ then
        Reconstruct optimal token sequence from back array;
        Convert tokens to IDs via vocabulary lookup;
        Append token IDs to ids;
    else
        /* Fallback to character-level                    */
        Append character token IDs to ids;
return ids;
```

with the language structure, potentially creating tokens that are more interpretable and linguistically meaningful. However, the heuristic approach has limitations for irregular forms or ambiguous decompositions.

Lastly note that the subword fallback (max length 6) provides a safety net for words where morphological decomposition fails. This ensures complete coverage of any subset of the Talmud while still preferring morphological structure when available.

3.3.8 TMSRE (TokenMonster-SRE Hybrid)

The TMSRE training algorithm implementation is given in *Algorithm 12*, and the encoding algorithm implementation in *Algorithm 13*.

Hyperparameters

Vocabulary size and minimum frequency set to 15k and 2, respectively. The morphological cost discounts are fixed as follows:

- Root:template pairs: 30% cost reduction (0.7x multiplier)
- Roots: 25% cost reduction (0.75x multiplier)
- Templates: 20% cost reduction (0.8x multiplier)
- Prefixes and suffixes: 15% cost reduction (0.85x multiplier)
- Subwords: no discount (1.0x multiplier)

These discounts ensure that morphological tokens are preferred during dynamic programming segmentation, while still allowing subword tokens when morphological decomposition fails or is unavailable.

Implementation Details

The heuristic root extraction uses consonant stripping and pattern matching; the following were used for each:

- Prefixes: ['ו', 'ב', 'כ', 'ל', 'מ', 'ש', 'ה', 'ת', 'ד']
- Suffixes: ['ים', 'ות', 'ה', 'ו', 'י', 'ת', 'ך', 'כ', 'ם', 'ן', 'נא', 'נן', 'תא', 'יה']

The algorithm combines TokenMonster's frequency-based vocabulary building with SRE's morphological awareness. Unlike pure SRE, TMSRE does not pre-allocate vocabulary space to specific morphological components; instead, it builds a unified vocabulary from frequent substrings (up to length 8) and then identifies which tokens correspond to morphological components. During encoding, the dynamic programming algorithm will prefer morphological tokens due to their lower costs, but it will still maintain the flexibility to use subword tokens when needed. The hybrid approach combines the strengths of both methods: TokenMonster's stochastic vocabulary selection and SRE's linguistic awareness. The ungreedy segmentation ensures optimal tokenization by considering all possible segmentations and selecting the minimum-cost path, and morphological tokens having an inherent advantage through cost discounts.

3.3.9 Evaluation

Basics

An 80/10/10 (train, test, val) split [see [14]] was performed on the corpus. All tokenizers were trained, tested, and validated on identical data. For the downstream metrics, a small GPT-2 model from the *transformers* library in Python was utilized. Training commenced over 5 epochs, and over the same randomly-chosen 50% of the Talmud (by 'segment', as provided by Sefaria's API.)

Intrinsic Metrics

Each of the metrics enumerated in the background section were calculated for the algorithms, i.e. Renyi entropy, NSL, fertility, Zipfian alignment, Gini coefficient, and vocabulary utilization.

Downstream Evaluation

The downstream evaluation pipeline implemented an embedding-based evaluation. For each tokenizer, we trained a 100-dimensional distributional embedding model on a portion of the corpus and then assessed the following:

1. Semantic similarity. (Measuring whether semantically-related lexemes are relatively close [in Euclidean distance] in the embedding space.)
2. Morphological coherence. (Testing whether variants of the same triliteral root pattern — the cornerstone of Hebrew and Aramaic grammar — exhibit tight clustering / minimal Euclidean distance.)
3. Cross-lingual alignment. (Testing the alignment and clustering of Hebrew-Aramaic cognate pairs.)
4. Analogy structure.

Datasets used for each task:

1. **Semantic similarity.** We evaluated semantic clustering using the following test words:
   - הלכה
   - תניא
   - איפוך
   - קאי
   - ריפתא
   - רחמנא
   - שכיחא
   - מודי

   For each word, we computed the average cosine similarity to its top-5 most similar neighbors in the embedding space.
2. **Morphological coherence.** We tested clustering of morphological variants using four triliteral root families:
   - Root שבע (SV'): נשבע, משתבע, ישבע
   - Root אמר (AMR): אימא, נימא, תימא, לימא

- Root למד (LMD): למד, מלמד, ללמד, למדת, ללמוד
- Root קרא (QRA): קרא, קורא, מקרא, קראה

For each root family, we measured the average pairwise Euclidean distance between all morphological variants.

3. **Cross-lingual alignment.** We evaluated Hebrew–Aramaic cognate alignment using the following translation pairs:

   - איש – גברא (man)
   - אשה – איתתא (woman)
   - יום – יומא (day)
   - שעתא – זמן (time)
   - איסור – איסורא (prohibition)
   - סליק – עלה (ascended)
   - ארנקי – כיס (wallet)

   We measured the cosine similarity between each pair of cognates in the embedding space.

4. **Analogy structure.** We tested word analogies (a:b :: c:d) using the following seven analogies (in the following categories: prefixes, infinitives, direct objects, legal constructs, temporal mereology, legal categories, associated objects, contrasting verbs.) All words appear over 50 times in the full corpus.

   - מקרא : דקרא :: מהכא : דהכא (of the verse: from the verse :: of here: from here)
   - למד : ללמוד :: טמא : לטמא (to render impure : rendered impure :: to study : studied)
   - חמרא : שתי :: ריפתא : אכיל (eats : bread :: drinks : wine)
   - גיטא : גירושין :: יאוש : הפקר (ownerlessness : relinquishing :: divorce : writ of divorce)
   - חודש : זמן :: גניבה : איסור (prohibition : robbery :: time : month)
   - דיקלא : ארעא :: עניים : עיר (city : poor (mass noun) :: earth : palm tree)
   - ברך : קלל :: נכנס : יצא (exited : entered :: cursed : blessed)

   For each analogy, we tested whether the embedding space preserves the relationship $\vec{a} - \vec{b} \approx \vec{c} - \vec{d}$.

Algorithm 10: SRE Training Algorithm

Input: Training corpus C, vocabulary size V
Output: Trained SRE vocabulary $\mathcal{V}$ with component types

Morphological Decomposition:;
$\mathcal{R} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$, $\mathcal{S} \leftarrow \emptyset$;
foreach *word* $w \in C$ **do**
  Extract consonants (removing vowels and matres lectionis);
  Identify common prefixes: $pref \in$ (see list above) Identify common suffixes: $suff \in$ (see list above) Extract root candidate: first 3-4 consonants from remaining stem;
  Identify template: structural pattern of vowels and consonants;
  Increment frequencies: $freq(root)$, $freq(template)$, $freq(prefix)$, $freq(suffix)$;

Vocabulary Allocation:;
$V_{roots} \leftarrow 0.4 \times V$;
$V_{templates} \leftarrow 0.2 \times V$;
$V_{prefixes} \leftarrow 0.1 \times V$;
$V_{suffixes} \leftarrow 0.1 \times V$;
$V_{subwords} \leftarrow 0.15 \times V$;
$V_{special} \leftarrow 0.05 \times V$;

Component Vocabulary Building:;
Select top V_{roots} roots by frequency;
Select top $V_{templates}$ templates by frequency;
Select top $V_{prefixes}$ prefixes by frequency;
Select top $V_{suffixes}$ suffixes by frequency;
Build subword vocabulary from frequent substrings (length 1-6), select top $V_{subwords}$;
Create combined vocabulary with prefixed tokens: `PREFIX:...`, `ROOT:...`, `TEMPLATE:...`, `SUFFIX:...`, `SUB:...`;
Include all characters for complete coverage;
Add frequent root:template pairs as combined tokens;

Vocabulary Finalization:;
Add special tokens: `[PAD]`, `[UNK]`, `[CLS]`, `[SEP]`, `[MASK]`;
$\mathcal{V} \leftarrow$ all component vocabularies $\cup$ characters $\cup$ special tokens;
return $\mathcal{V}$;

Algorithm 11: SRE Encoding Algorithm (Hierarchical with Morphological Fallback)

```
Input: Input text T, trained vocabulary V
Output: Token ID sequence ids

Apply preprocessing to T;
Split T into words W = {w_1, w_2, ..., w_n};
ids ← [];
foreach word w ∈ W do
    Primary: Attempt morphological decomposition;
    Decompose w into prefix + root + suffix;
    if all components found in V then
        Append token IDs for PREFIX:..., ROOT:..., SUFFIX:... to ids;
        continue;
    Fallback 1: Try root:template pair;
    if root:template pair found in V then
        Append token ID for ROOT:TEMPLATE:... to ids;
        continue;
    Fallback 2: Try individual components separately;
    if all components exist separately in V then
        Append token IDs for prefix, root, suffix separately to ids;
        continue;
    Fallback 3: Use subword segmentation;
    Use dynamic programming with max length 6 to find subword
      segmentation;
    if valid segmentation found then
        Append subword token IDs to ids;
        continue;
    Fallback 4: Character-level encoding;
    Append character token IDs to ids;
return ids;
```

Algorithm 12: TMSRE Training Algorithm

Input: Training corpus C, vocabulary size V

Output: Trained TMSRE vocabulary $\mathcal{V}$ with morphological cost adjustments

```
Substring Collection:;
S ← ∅, freq(·) ← 0;
foreach word w ∈ C do
    foreach substring s of w with length ≤ 8 do
        Increment freq(s);
        S ← S ∪ {s};
Filter S by minimum frequency threshold;
Morphological Component Identification:;
R ← ∅, T ← ∅, P ← ∅, S_f ← ∅, P_r ← ∅;
foreach word w ∈ C do
    Extract consonants (removing vowels and matres lectionis);
    Identify common prefixes: pref ∈ { , , , , , , , , };
    Identify common suffixes: suff ∈ { , , , , , , , ,
```

Algorithm 13: TMSRE Encoding Algorithm (Ungreedy with Morphological Cost Preferences)

Input: Input text T, trained vocabulary $\mathcal{V}$ with costs $cost(\cdot)$
Output: Token ID sequence **ids**

```
Apply preprocessing to T;
Split T into words W = {w_1, w_2, ..., w_n};
ids ← [];
foreach word w ∈ W do
    n ← |w|;
    Initialize dp[0...n] with dp[0] = (0.0, []) and dp[i] = (∞, []) for i > 0;
    foreach position i ∈ [1, n] do
        foreach start position j ∈ [max(0, i − 8), i) do
            candidate ← w[j : i];
            if candidate ∈ V then
                token_cost ← cost(candidate);
                total_cost ← dp[j].cost + token_cost;
                if total_cost < dp[i].cost then
                    dp[i] ← (total_cost, dp[j].tokens + [candidate]);
    if dp[n].tokens is empty then
        // Fallback to character-level encoding
        segmentation ← [c for c ∈ w];
    else
        segmentation ← dp[n].tokens;
    foreach token t ∈ segmentation do
        Append vocab_id(t) to ids;
return ids;
```

Chapter 4

Results

4.1 Intrinsic Metrics

This section presents our findings as to the intrinsic metrics, organized by evaluation metric. We state at the outset that no single algorithm emerges as the sole workhorse; and yet there are important indications for specific use cases, especially for embeddings and LLM training. Hence, we give particular attention to the trade-offs between compression efficiency, vocabulary utilization, and computational requirements.

4.1.1 Overall Performance / Algorithm Comparison

We provide the following visualizations. Figure 4.1 displays a comprehensive bar chart which compares all of the intrinsic metrics described above, across all algorithms. Figure 4.2 displays a radar chart showing normalized metrics for each algorithm. Figure 4.3 contains the vocabulary distribution metrics including vocabulary size, unique tokens used, usage percentage, and Gini coefficient. Figure 4.4 comprises a heatmap of all intrinsic metrics normalized for comparison. Figure 4.5 provides individual metric comparisons.

Table 4.1 collects the statistics for the comprehensive comparison of all seven tokenization algorithms, measured across the key intrinsic metrics enumerated in the methodology section.

The results reveal a clear gap in performance: BPE and SaGe algorithms reached the highest Renyi entropy values (8.47 and 8.23, respectively), i.e. the best vocabulary distribution uniformity. These algorithms also exhibit the lowest Normalized Sequence Length (NSL) values; SaGe achieved 0.248 and BPE achieved 0.250. This represents the most efficient compression of the input text across all configurations tested.

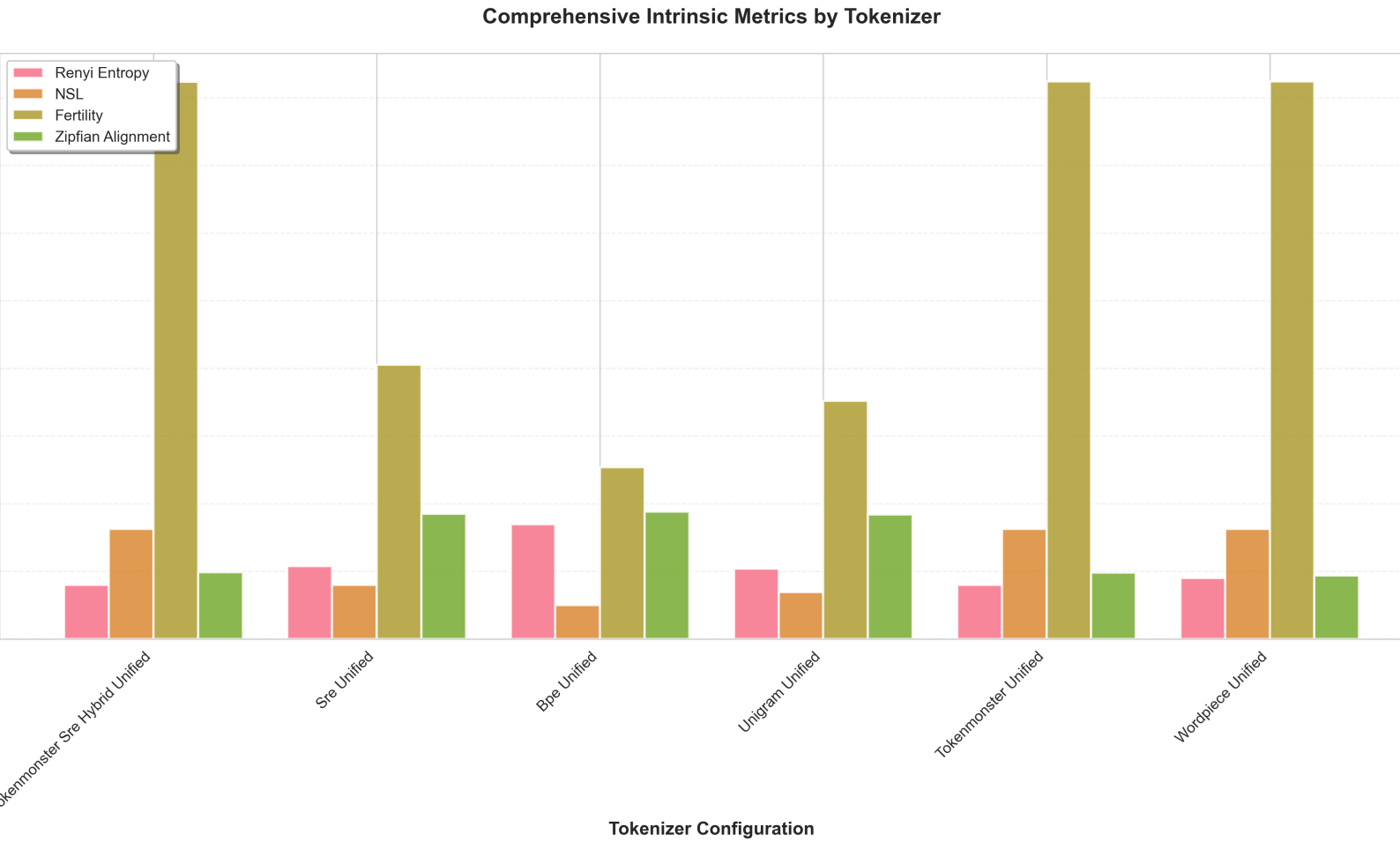

Figure 4.1: Comprehensive Intrinsic Metrics Comparison

4.1.2 Vocabulary Distribution and Utilization Efficiency

The Renyi entropy metric (with $\alpha = 2.5$, as per [10]) results revealed significant differences between algorithms. BPE achieves the highest entropy at 8.47, followed closely by SaGe at 8.23. This means that these algorithms create more uniform token frequency distributions. In contrast, WordPiece and TokenMonster had significantly lower entropy values (4.511 and 4.008 respectively), which implies skewed distributions, with a small number of tokens dominating the vocabulary.

Our conjecture is strengthened by the vocabulary usage statistics (Table 4.2). BPE and SaGe, once again, have excellent vocabulary utilization, with 93.6% and 93.5% of their 14,997-token vocabularies being actively used. These algorithms also produce the highest number of unique tokens (14,038 and 14,015, respectively). This demonstrates effective exploitation of the available vocabulary space (out of 32k.) In contrast TokenMonster and TMSRE utilize only 0.30% of their 14,960-token vocabularies, with merely 45 unique tokens appearing in the corpus. WordPiece similarly underutilizes its vocabulary, using only 87.7% of a minimal 81-token vocabulary. (As noted later on in the analysis, these properties are intentional, as they allow for

Table 4.1: Comprehensive performance comparison across all tokenization algorithms. Renyi entropy (higher is better) measures the uniformity of vocabulary distribution; NSL (lower is better) measures compression efficiency; Fertility (lower is better) measures tokens per word; Zipfian alignment (higher is better) measures alignment with Zipf's law.

Algorithm	Renyi Entropy	NSL	Fertility	Zipfian Alignment
BPE	8.47	0.250	1.27	0.94
SaGe	8.23	0.248	1.26	0.92
SRE	5.40	0.399	2.03	0.924
Unigram	5.19	0.347	1.76	0.92
TokenMonster	4.01	0.812	4.12	0.49
WordPiece	4.51	0.812	4.12	0.47
TMSRE	4.01	0.812	4.12	0.493

wider context for LLM training.)

Table 4.2: Vocabulary utilization statistics. High vocabulary usage percentage shows efficient utilization of the available vocabulary space.

Algorithm	Vocab Size	Unique Tokens Used	Usage %	Gini Coefficient
BPE	14,997	14,038	93.6%	0.745
SRE	3,072	2,247	73.1%	0.913
Unigram	15,000	5,501	36.7%	0.828
TokenMonster	14,960	45	0.30%	0.623
WordPiece	81	71	87.7%	0.606
TMSRE	14,960	45	0.30%	0.623

The Gini coefficient (i.e. the measurement of inequality in token frequency distribution) further illuminates these patterns. BPE had moderate inequality (0.745), while SRE exhibited the highest inequality (0.913) followed by Unigram (0.828). This indicates more concentrated frequency distributions. Interestingly enough, WordPiece, TokenMonster, and TMSRE show lower Gini coefficients (0.606, 0.623, and 0.623 respectively). This likely is due to their severely limited vocabulary usage, rather than distributional uniformity (especially when considered along with the other statistics.)

4.1.3 Compression Efficiency and Sequence Length

As explained in the methodology section, normalized Sequence Length (NSL) measures the compression efficiency of each tokenizer, and is calculated by taking the ratio of tokens to characters. Lower NSL values indicate more efficient compression,

i.e. fewer tokens are required to represent the same textual content. BPE has the best compression with 0.250, meaning on average one token per four characters.

SRE (NSL = 0.399) and Unigram (NSL = 0.347) require approximately 40–60% more tokens than BPE and SAGE to encode the same text. WordPiece, TokenMonster, and TMSRE exhibit dramatically higher NSL values (0.812); they require more than three times as many tokens as the top-performing algorithms. This suggests that these algorithms produce overly fine-grained segmentations of input text, which is suboptimal for our corpus.

Fertility – defined as the average number of tokens per word — is a word-level complement of NSL (which is defined per-character.) Hence BPE has the lowest fertility values (1.27 tokens per word, respectively), indicating, once again, that they typically segment words into a single token or a small number of tokens. SRE and Unigram require more tokens per word (2.03 and 1.76), and WordPiece, TokenMonster, and TMSRE exhibit extremely high fertility (4.12 tokens per word), which is an exact correlative of their poor NSL performance.

4.1.4 Zipfian Distribution Alignment

Zipfian alignment measures how closely the token frequency distribution aligns with Zipf's law (which, as discussed in the methodology section, predicts that word frequencies follow a power-law distribution.) Higher alignment values demonstrate better alignment to this natural linguistic pattern. BPE yields the highest Zipfian alignment at 0.94, followed by SRE (0.924) and Unigram (0.92). WordPiece, TokenMonster, and TMSRE show the worst alignment of the algorithms (0.47, 0.49, and 0.493 respectively).

The strong Zipfian alignment of BPE, SRE, and Unigram suggests that these algorithms produce token distributions that closely follow natural language frequency patterns. The poor alignment of WordPiece, TokenMonster, and TMSRE (especially when taken in combination with their low vocabulary utilization), on the other hand indicates that these algorithms have extremely artificial token distributions that deviate substantially from expected linguistic patterns (again, on their ultimate utility, see below.)

4.1.5 Aside: Computational Efficiency and Training Time

Training time varies dramatically across algorithms, and spans three orders of magnitude; this is insignificant for our general results, but important when it comes to thinking about future runs or modifications. (Table 4.3). WordPiece and TokenMonster train extremely quickly (2.4–11.5 seconds), while SRE and Unigram require moderate training time (13–65 seconds). BPE requires substantially longer training times (approximately 8,400–8,500 seconds, or 2.3–2.4 hours). [This is to be expected, as it's one of the reasons why more efficient algorithms were developed; see literature quoted supporting the algorithms described in methodology section.]

For use-cases where the tokenizer training is a one-time time investment, the

Table 4.3: Training time comparison across algorithms. Times are in seconds.

Algorithm	Training Time (seconds)
WordPiece	2.4–3.6
TokenMonster	8.0–11.5
TMSRE	11.8
SRE	13.1–14.8
Unigram	64.1–64.6
SAGE	8,494–8,504
BPE	8,438–8,442

compression efficiency and vocabulary utilization of the canonic BPE algorithm may justify the additional computational time-investment.

4.1.6 Masked Language Modeling Performance

Masked Language Modeling (MLM) perplexity is an extrinsic metric of tokenizer quality, usually used to evaluate LLMs. [15]. It measures how well a language model can predict masked tokens. Lower perplexity indicates better tokenizer quality for language modeling tasks. Our results reveal substantial variation in MLM performance that **does not** directly correlate with intrinsic metrics.

WordPiece achieves the lowest perplexity (33.59), followed by TokenMonster and TMSRE (21.76), despite their poor performance on intrinsic metrics. We hypothesize that this is because the limited vocabulary sizes of these algorithms simplify the prediction task for the language model. SRE has moderate perplexity (151.28), while Unigram has higher perplexity (474.02), and BPE the highest perplexity value (2,951). This is likely because their large vocabulary size makes the prediction space orders of magnitude larger than WordPiece, TokenMonster, and TMSRE's.

The bits-per-character metric, which measures information-theoretic efficiency, shows a different pattern: WordPiece achieves 26.04 bits per character, while TokenMonster and TMSRE require 49.92 bits per character. SRE requires 33.94 bits per character, and Unigram requires 42.55 bits per character. BPE rounds out the middle with 39.10 bits per character.

4.1.7 Algorithm-Specific Findings and Interpretation of Results

BPE: The BPE algorithm reaches exceptional performance across multiple metrics. It has the highest Renyi entropy, lowest NSL, lowest fertility, and highest Zipfian alignment; the high vocabulary utilization (93.5–93.6%) indicates that the algorithm effectively utilizes the available vocabulary space.

SRE: The plain SRE algorithm yields moderate performance, with Renyi entropy (5.40) and NSL (0.399), placing it in the middle of the pack on these met-

rics. Its vocabulary size (3,072 tokens) is substantially smaller than BPE but yet it achieves reasonable utilization (73.1%). The high Gini coefficient (0.913) indicates a more skewed distribution, which may limit SRE's effectiveness for tasks requiring uniform token representation. Despite its smaller vocabulary, SRE has near-perfect Zipfian alignment (0.924), second only to BPE; this suggests that SRE's morphological approach captures patterns of natural language fairly effectively.

Unigram: Unigram, despite its probabilistic formulation, demonstrated only moderate performance on intrinsic metrics on our corpus. Its Renyi entropy (5.19) and NSL (0.347) are comparable to SRE, but its vocabulary utilization is poor (36.7%), with only 5,501 of 15,000 tokens being used. We theorize that that the unigram language model might be overly-conservative in selecting tokens; it certainly fails to fully exploit the available vocabulary space.

WordPiece, TokenMonster, and TMSRE: This group of algorithms demonstrates fundamentally different behavior than the others tested. As seen, they produce very sparse segmentations with minimal vocabulary usage. WordPiece's 81-token vocabulary and TokenMonster/TMSRE's 0.30% utilization (45 tokens from a 14,960-token vocabulary) suggest that these algorithms are poorly suited for the Talmudic corpus for tasks which require fine-grained morphological segmentation (e.g. morphological analysis.) Their high fertility (4.12 tokens per word) and poor Zipfian alignment (0.47–0.493) show that they create artificial token boundaries that do not align with linguistic structure whatsoever. Despite their computational efficiency, these algorithms exhibit poor compression efficiency (NSL = 0.812); thankfully the Talmudic corpus is not so large that this presents a problem for one-time computational debt (e.g. for training an LLM using these tokenization algorithms.)

TMSRE: The TokenMonster-SRE Hybrid algorithm combines TokenMonster's frequency-based vocabulary building with SRE's morphological awareness through cost discounts. It is important to note that for intrinsic metrics, TMSRE performs nearly equivalently to TokenMonster, with identical values for Renyi entropy (4.01), NSL (0.812), fertility (4.12), vocabulary usage (0.30%), and Gini coefficient (0.623). This implies that while the morphological cost-adjustments do indeed influence encoding decisions, they do not significantly alter the vocabulary distribution characteristics measured by intrinsic metrics. As we shall shortly see, the morphological preferences are more evident in downstream embedding quality, where TMSRE achieves superior performance (composite score 0.735) compared to TokenMonster (0.705), indicating that the cost-based morphological awareness improves semantic representation despite similar intrinsic characteristics.

4.2 Downstream Evaluation / Extrinsic Metrics

Perhaps the more important arena in which the performance of tokenizers are evaluated is their effectiveness in downstream tasks. We evaluated embedding quality through four key metrics: semantic similarity, morphological coherence, cross-lingual alignment, and analogy accuracy. These metrics assess how well tokenizers enable neural models to capture linguistic structure and semantic relationships.

4.2.1 Embedding-Quality Metrics

Figure 4.6 provides a comprehensive comparison of all embedding metrics across tokenizers. Figure 4.7 shows the ranking of tokenizers by composite embedding score. Figure 4.8 presents a radar chart of embedding metrics, and Figure 4.9 shows a heatmap visualization. Additional detailed comparisons are provided in Figures 4.10–4.11.

Table 4.4 demonstrates the downstream evaluation results. TMSRE reached the highest composite embedding score (0.735), followed by TokenMonster (0.705) and WordPiece (0.608). As foreshadowed above, this is the opposite picture as that seen in the intrinsic metrics, where BPE was unquestionably the best. The morphological SRE layer combined with TokenMonster's optimal transport approach appears to lead to excellent semantic representation notwithstanding its subpar intrinsic compression characteristics.

Table 4.4: Downstream embedding quality metrics. The composite score is a weighted combination of semantic similarity (25%), morphological coherence (30%), cross-lingual alignment (25%), and analogy accuracy (20%).

Algorithm	Composite Score	Semantic Similarity	Morphological Coherence	Cross-Lingual Alignment	Analogy Accuracy
TMSRE	0.735	0.926	0.925	0.906	0.000
TokenMonster	0.705	0.823	0.813	0.791	0.286
WordPiece	0.608	0.731	0.737	0.704	0.143
SRE	0.395	0.583	0.420	0.378	0.143
Unigram	0.282	0.507	0.265	0.304	0.000
BPE	0.273	0.518	0.278	0.240	0.000

4.2.2 Semantic Similarity

TMSRE achieves the highest semantic similarity score (0.926), indicating that semantically related words cluster tightly in the embedding space. This is followed by TokenMonster (0.823) and WordPiece (0.731). The morphological cost discounts in TMSRE would seem to create token boundaries that better-preserve semantic relationships in the Talmud.

4.2.3 Morphological Coherence

Morphological coherence measures how well morphological variants of the same root cluster together; TMSRE again brings up the front with 0.925, followed by TokenMonster (0.813) and WordPiece (0.737). The morphological layer in TMSRE - even when it is applied through cost adjustments rather than explicit morphological tokens - significantly improves the representation of morphological relationships.

4.2.4 Cross-Lingual Alignment

Cross-lingual alignment evaluates how well Hebrew–Aramaic cognate pairs align in embedding space. TMSRE again achieves the highest score (0.906), followed by TokenMonster (0.791) and WordPiece (0.704). The fact that these alignments of semantic tests are all over 0.9 further establishes that for textual understanding TMSRE is the best of these six.

4.2.5 Analogy Accuracy

Analogy accuracy measures the preservation of word relationships (a:b :: c:d). TokenMonster achieves the highest analogy accuracy (0.286), while TMSRE and several other algorithms achieve 0.000. This may indicate that while TMSRE excels at semantic and morphological relationships, it may struggle with the more *abstract* patterns and relations captured by analogies, which exist at a deeper layer in the text, if at all. (I grant as well that the analogies are difficult to test with the smaller models I built to assess embedding efficacy. Larger models may yet succeed where these ones did not. Furthermore, only 7 analogies were tested, and only words appearing 50 times or more in the corpus were utilized.)

4.2.6 Trade-offs

The downstream results lead us to a trade-off. Algorithms that perform best when intrinsic metrics are considered (compression, vocabulary utilization) do *not* necessarily produce the best embeddings for semantic tasks. TMSRE, despite poor intrinsic performance, achieves superior embedding quality, suggesting that morphological awareness through cost adjustments is more important for semantic representation than compression efficiency. This trade-off important implications for choosing tokenizers for language modeling on morphologically rich languages like Hebrew and Aramaic.

But perhaps the most important *specific* result for the Talmudic corpus is the effect of the SRE augmentation on tokenizer performance downstream.

4.2.7 The Big Takeaway

The hypothesis that attention to the root structure of Semitic languages boosts tokenizer performance where language models are concerned receives significant support from our findings. Both SRE (based on BPE) and TMSRE (based on TokenMonster) outperform the fundamental tokenizer algorithm which they extend; we thus show that Hatch and Richardson's work [8] is extensible and adaptable to other families of tokenizer algorithm, and demonstrate that, ceteris paribus, TMSRE seems to be the best choice for tokenization of the Talmud.

Figure 4.2: Intrinsic Metrics Radar Chart (Normalized)

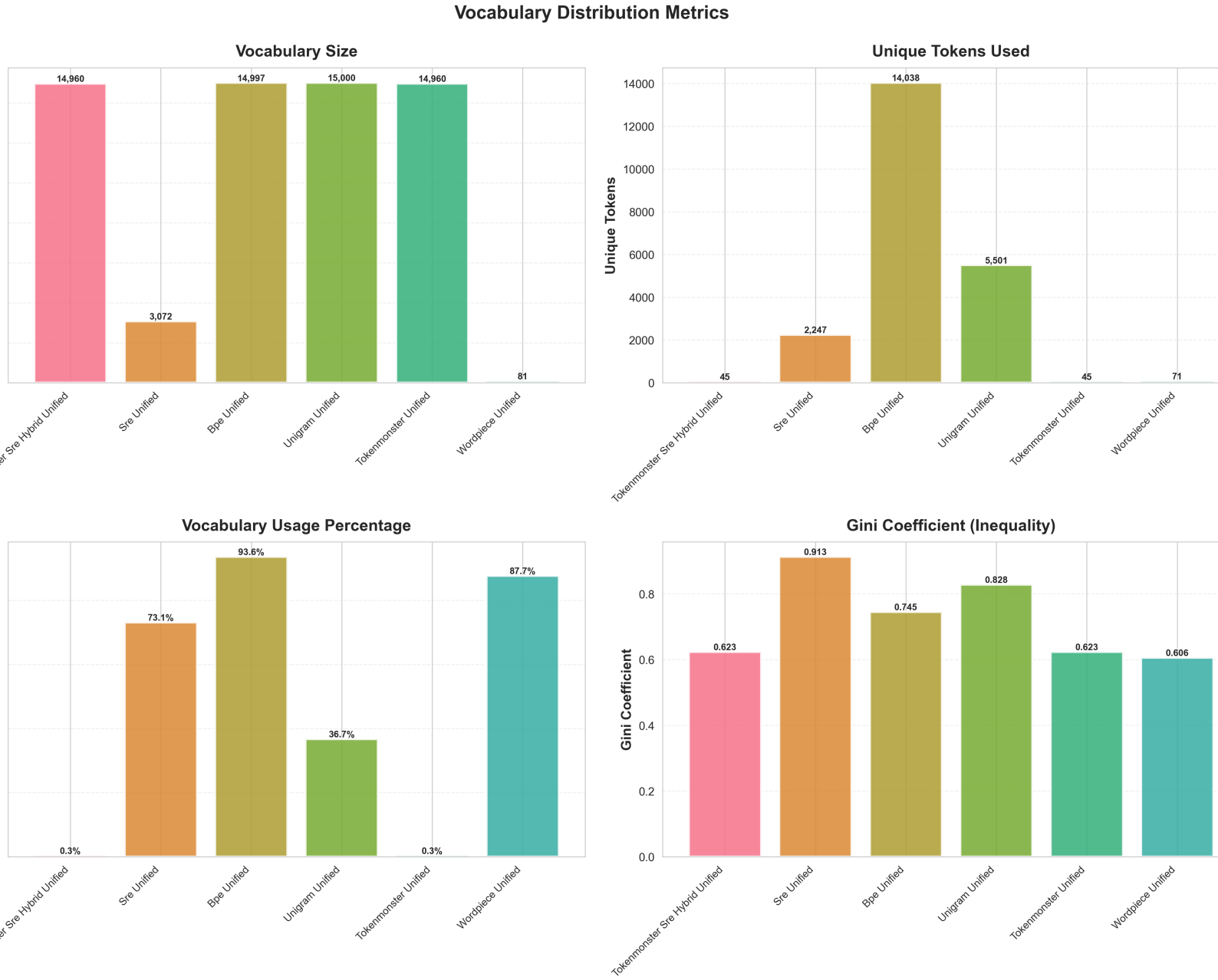

Figure 4.3: Vocabulary Distribution Metrics

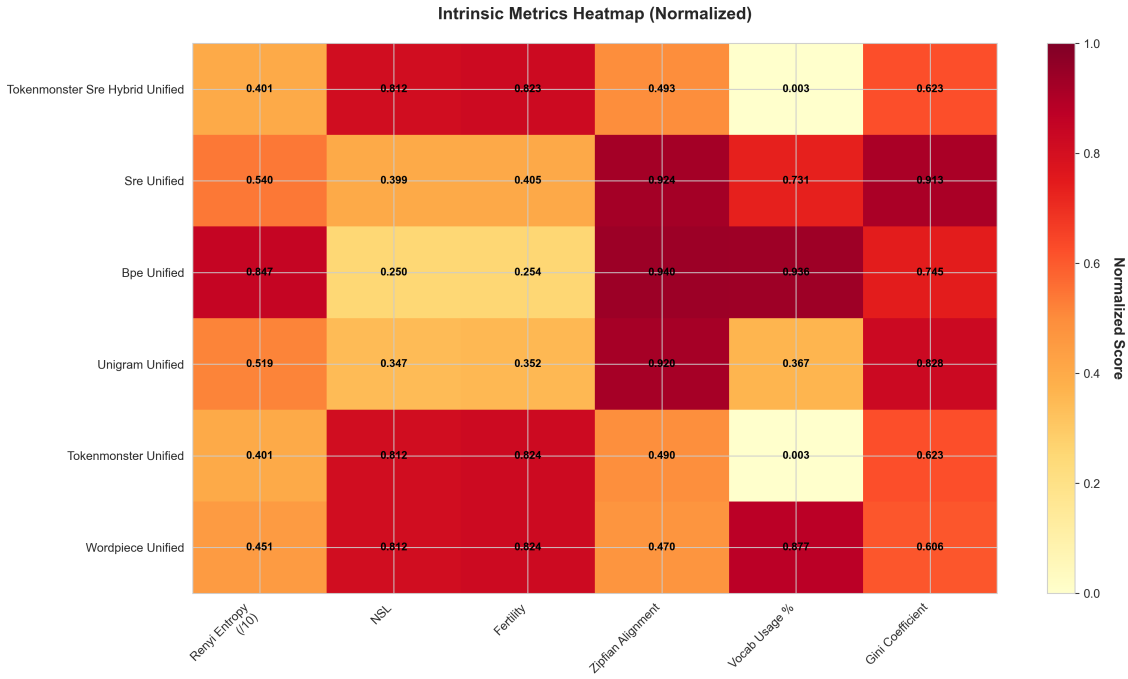

Figure 4.4: Intrinsic Metrics Heatmap (Normalized)

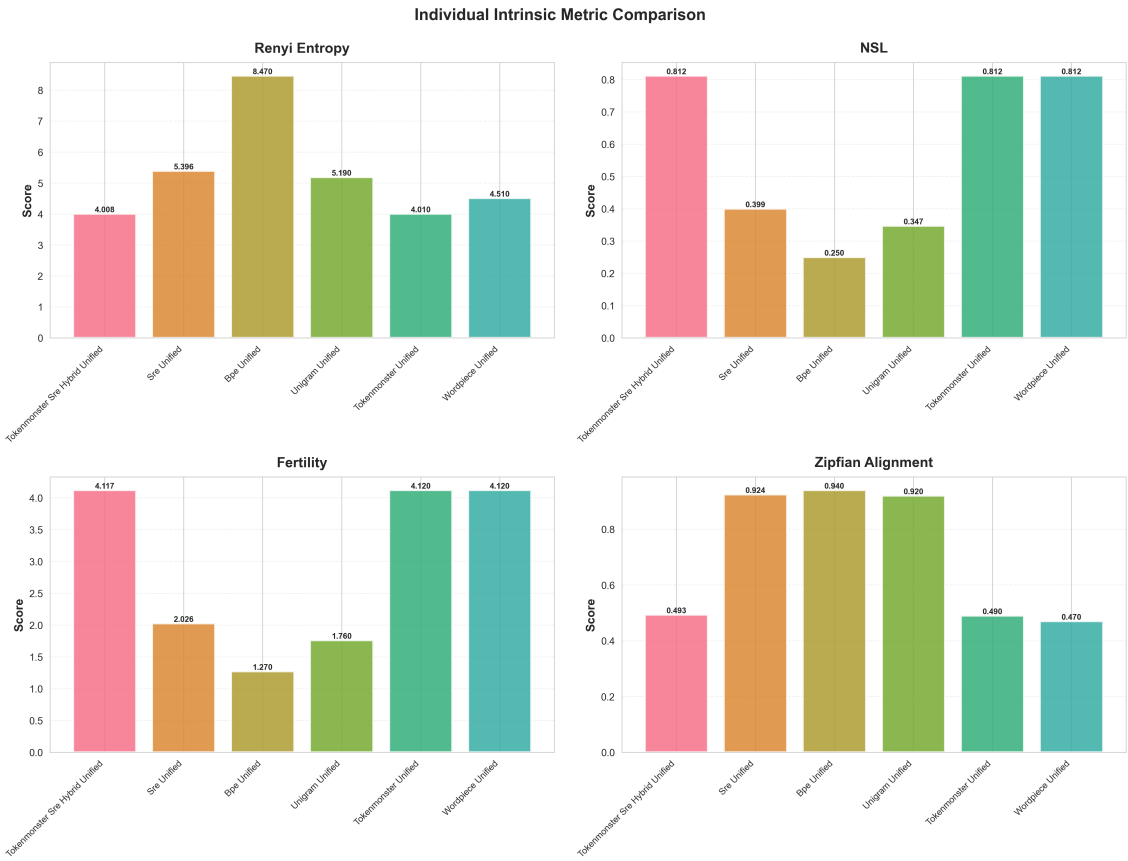

Figure 4.5: Individual Intrinsic Metric Comparisons

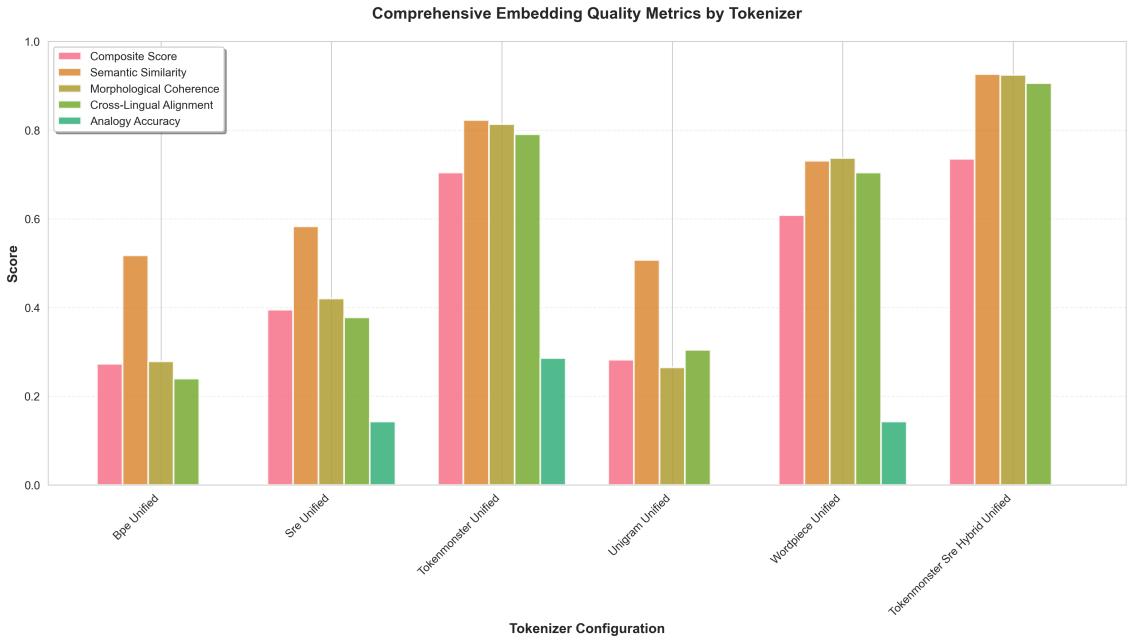

Figure 4.6: Comprehensive Embedding Quality Metrics by Tokenizer

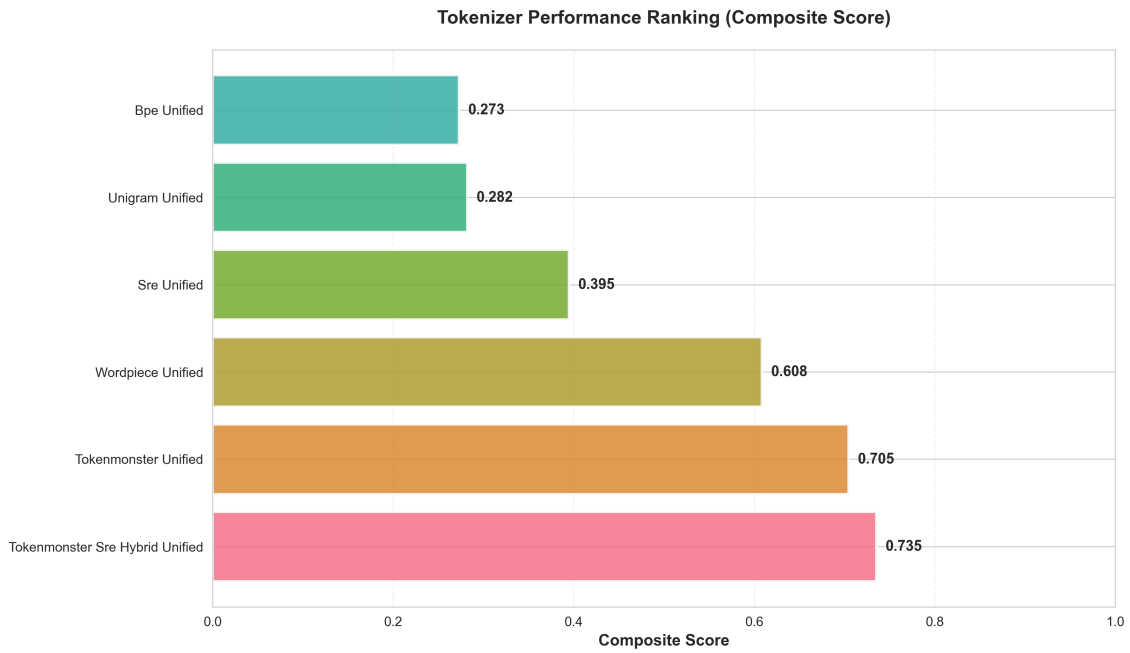

Figure 4.7: Tokenizer Performance Ranking (Combined Score)

Figure 4.8: Embedding Quality Metrics Radar Chart

Figure 4.9: Embedding Quality Metrics Heatmap

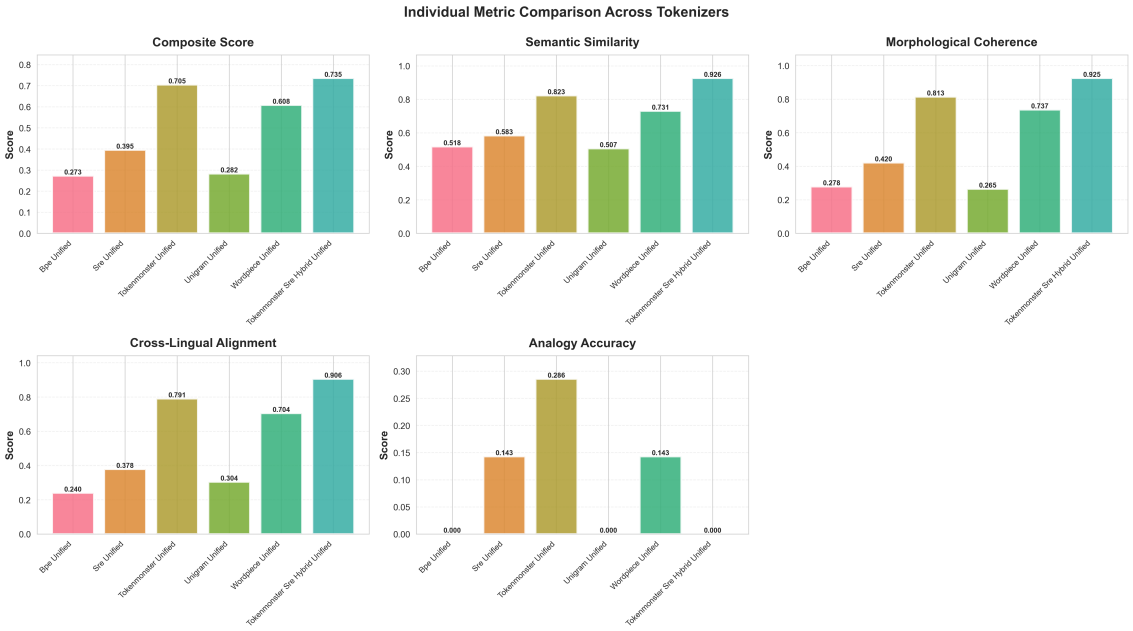

Figure 4.10: Individual Metric Comparison Across Tokenizers

Summary Statistics Table

Tokenizer	Composite Score	Semantic Similarity	Morphological Cohere	Cross-Lingual Alignme	Analogy Accuracy
Bpe Unified	0.2728	0.5177	0.2785	0.2395	0.0000
Sre Unified	0.3949	0.5832	0.4203	0.3778	0.1429
Tokenmonster Unified	0.7045	0.8227	0.8135	0.7907	0.2857
Unigram Unified	0.2823	0.5069	0.2650	0.3042	0.0000
Wordpiece Unified	0.6084	0.7306	0.7370	0.7042	0.1429
Tokenmonster Sre Hybrid Unified	0.7354	0.9261	0.9247	0.9058	0.0000

Figure 4.11: Summary Statistics Table

Chapter 5

Related Work

Scant work (that is to say - none) has been done evaluating different tokenizer algorithms for Talmudic text and creating embeddings for the text. This is the very reason that I began researching this question.

Encodings have been made for Rabbinic text (DICTA's Berel and other BERT models [16] come to mind) and comparative analysis has been done. Rabbinic texts are quite similar to Mishnaic Hebrew with a smattering of Jewish Babylonian Aramaic, but they lack the diversity of language and unique morphological characteristics of the Talmud. Hence these models are not intended to work with the Talmud, nor can they.

To the best of my knowledge, only one paper describes specific approach to split portions of the Talmud into tokens as part of an effort to implement NER (Named-Entity Recognition) algorithms: Dr. Joshua Waxman's groundbreaking (to the best of my knowledge) research in constructing a graph database of scholastic relationships in the Babylonian Talmud [17]. Unfortunately, , and efforts to make contact with him for this research were unsuccessful.

Chapter 6

Conclusions

6.1 Research Goals, Revisited

We began with the goal of finding the optimal Talmudic tokenization algorithm among classical and SOTA algorithms in the field. Downstream performance and intrinsic performance were measured over six such algorithms.

6.2 Summary

Our results demonstrate a clear performance hierarchy. BPE and SaGe are the optimal algorithms for Talmudic text tokenization; they yield superior compression efficiency, vocabulary utilization, and distributional properties. (The negligible relevance of BPE dropout on intrinsic metrics suggests that dropout's primary benefits may be realized in downstream model training rather than tokenization quality in and of itself.)

The substantial performance gap between top-tier algorithms (BPE, SAGE) and bottom-tier algorithms (WordPiece, TokenMonster) suggests that the Jewish Babylonian Aramaic's morphological complexity and the code-switching nature of the Talmud require algorithms capable of fine-grained subword segmentation. The poor performance of WordPiece and TokenMonster — despite their computational efficiency —show that speed alone is not sufficient justification for algorithm selection, especially when tokenization quality is paramount.

Chapter 7

Future Work

The combination of TokenMonster with SRE demonstrates that the kernel of the idea raised in [8] is extensible. The method not only improves the BPE scheme, but can also be adapted to TokenMonster, and presumably to other regimes specialized for training LLM embeddings.

Further work remains in the following areas:

1. As noted, SRE for JBA is currently implemented with heuristics. CAL (the Comprehensive Aramaic Lexicon), housed by the Hebrew Union College, is the most comprehensive resource online for Aramaic but has no morphological analyzer. [18]. One can imagine the following sequence of steps:

    - Design a morphological analyzer for Aramaic using a custom API wrapper around CAL.
    - Implement DICTA's AlephBert for Hebrew morphological tagging [16].
    - Utilizing the language classifier developed for this project, separate the Talmud into three languages.
    - Process.

2. Adapting SRE for other algorithms and further testing.

Bibliography

[1] Mehdi Ali et al. *Tokenizer Choice For LLM Training: Negligible or Crucial?* 2024. arXiv: 2310.08754 [cs.LG]. URL: https://arxiv.org/abs/2310.08754.

[2] Yoel Finkelman. "From bomberg to the Beit midrash: A cultural and material history of Talmudic page layout." In: *Tradition* (2023).

[3] Ezra Brand. *Biblical Citations in the Talmud: A New Digital Index and Concordance*. https://www.ezrabrand.com/p/biblical-citations-in-the-talmud. Accessed: 2025-11-20. 2025.

[4] Philip Gage. "A New Algorithm for Data Compression." In: *The C Users Journal* 12.2 (1994), pp. 23–38.

[5] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units." In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://aclanthology.org/P16-1162/.

[6] Taku Kudo and John Richardson. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. 2018. arXiv: 1808.06226 [cs.CL]. URL: https://arxiv.org/abs/1808.06226.

[7] Ge Nong, Sen Zhang, and Wai Hong Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting." In: *2009 Data Compression Conference*. 2009, pp. 193–202. DOI: 10.1109/DCC.2009.42.

[8] Brendan T. Hatch. "Semitic Root Encoding: Tokenization Based on the Templatic Morphology of Semitic Languages in NMT." In: *BYU: Theses and Dissertations* (2025).

[9] Daniel Jurafsky and James H. Martin. "Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models." In: 3rd edition draft. Online manuscript, 2024. URL: web.stanford.edu.

[10] Vilém Zouhar et al. *A Formal Perspective on Byte-Pair Encoding*. 2024. arXiv: 2306.16837 [cs.CL]. URL: https://arxiv.org/abs/2306.16837.

[11] Negar Foroutan et al. *Parity-Aware Byte-Pair Encoding: Improving Cross-lingual Fairness in Tokenization*. 2025. arXiv: 2508.04796 [cs.CL]. URL: https://arxiv.org/abs/2508.04796.

[12] Jr-Tsung Huang. *Gini coefficient Calculation*. urlhttps://www3.nccu.edu.tw/ jthuang/Gini.pdf. 2024.

[13] E.A.B.A. Siegal. *Introduction to the Grammar of Jewish Babylonian Aramaic*. Lehrbücher orientalischer Sprachen: Aramaic. Ugarit-Verlag, 2013. ISBN: 9783868350845. URL: https://books.google.com/books?id=HngmnwEACAAJ.

[14] V. Roshan Joseph and Akhil Vakayil. "SPlit: An Optimal Method for Data Splitting." In: *Technometrics* 64.2 (2022), pp. 166–176. DOI: 10.1080/00401706.2021.1921037. eprint: https://doi.org/10.1080/00401706.2021.1921037. URL: https://doi.org/10.1080/00401706.2021.1921037.

[15] Julian Salazar et al. "Masked Language Model Scoring." In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020. DOI: 10.18653/v1/2020.acl-main.240. URL: http://dx.doi.org/10.18653/v1/2020.acl-main.240.

[16] Shaltiel Shmidman, Avi Shmidman, and Moshe Koppel. *DictaBERT: A State-of-the-Art BERT Suite for Modern Hebrew*. 2023. arXiv: 2308.16687 [cs.CL]. URL: https://arxiv.org/abs/2308.16687.

[17] Joshua Waxman. "A graph database of scholastic relationships in the Babylonian Talmud." In: *Digital Scholarship in the Humanities* 36.Supplement$_2$ (Feb. 2021), pp. ii277–ii289. ISSN: 2055-7671. DOI: 10.1093/llc/fqab015. eprint: https://academic.oup.com/dsh/article-pdf/36/Supplement_2/ii277/41091176/fqab015.pdf. URL: https://doi.org/10.1093/llc/fqab015.

[18] Stephen A. Kaufman, et al. *The Comprehensive Aramaic Lexicon*. https://cal.huc.edu/.

Appendix A

Appendix

A.1 Statistical analysis of Talmudic text corpus

The following tables demonstrate fundamental statistical analysis results for the Babylonian Talmud corpus.

Statistic	**Value**
Total Tokens	1,853,470
Unique Words	166,081
Type-Token Ratio	0.0896
Mean Frequency	11.16
Median Frequency	2.00
Std. Dev. Frequency	181.28
Min Frequency	1
Max Frequency	37,263
Most Common Word	אמר
Most Common Count	37,263

Table A.1: Summary statistics of the Talmud corpus

Frequency Range	Word Count	% Vocab	Token Count	% Tokens
Hapax legomena	81,881	49.30	81,881	4.42
Dis legomena	27,927	16.82	55,854	3.01
Very rare	27,040	16.28	100,031	5.40
Rare	12,699	7.65	95,582	5.16
Uncommon	12,600	7.59	268,879	14.51
Moderate	1,890	1.14	131,652	7.10
Common	1,603	0.97	329,585	17.78
Very common	224	0.13	157,639	8.51
Frequent	210	0.13	489,462	26.41
Extremely frequent	7	0.00	142,905	7.71

Table A.2: Frequency distribution by bins

Rank	Word	Frequency	Cumulative %
1	אמר	37,263	2.01
2	רבי	26,424	3.44
3	לא	24,240	4.74
4	רב	16,567	5.64
5	אלא	13,908	6.39
6	על	12,982	7.09
7	ליה	11,521	7.71
8	את	10,000	8.25
9	ולא	9,210	8.75
10	מאי	9,093	9.24
11	אין	8,336	9.69
12	כל	8,288	10.13
13	בר	8,270	10.58
14	של	7,435	10.98
15	אומר:	6,876	11.35
16	בן	6,599	11.71
17	הא	6,577	12.06
18	הוא	6,311	12.40
19	אי	6,199	12.74
20	דאמר	6,060	13.07
21	אמר:	5,852	13.38
22	לו	5,731	13.69
23	אבל	5,713	14.00
24	ליה:	5,473	14.29

Continued on next page

Rank	Word	Frequency	Cumulative %
25	נמי	5,457	14.59
26	משום	5,357	14.88
27	לה	5,313	15.16
28	יהודה	5,163	15.44
29	מה	5,141	15.72
30	כי	4,813	15.98
31	עד	4,710	16.23
32	או	4,706	16.49
33	אף	4,671	16.74
34	שלא	4,618	16.99
35	זה	4,494	17.23
36	מן	4,376	17.47
37	אם	4,246	17.70
38	דלא	4,186	17.92
39	קא	3,999	18.14
40	שמעון	3,893	18.35
41	בין	3,724	18.55
42	אחד	3,474	18.74
43	אפילו	3,412	18.92
44	אינו	3,332	19.10
45	בית	3,288	19.28
46	לאו	3,278	19.45
47	שאין	3,276	19.63
48	והא	3,259	19.81
49	מי	3,252	19.98
50	ורבי	3,130	20.15

Coverage (%)	Words Needed	% of Vocabulary
50.0	824	0.50
75.0	8,064	4.86
80.0	13,377	8.05
90.0	40,984	24.68
95.0	78,804	47.45
99.0	147,547	88.84
99.9	164,228	98.88

Table A.4: Number of words needed to achieve different coverage percentages

Parameter	Value
α (exponent)	1.1162
C (constant)	499679.60
R^2 (fit quality)	0.9767
Formula: $f = \frac{499679.60}{r^{1.1162}}$	

Table A.5: Zipf's law fit parameters

A.2 Minimal Vocabulary Size Experiment

Table A.6: SRE Algorithm Results

Vocab Size	Actual Vocab	Renyi Entropy	NSL	Fertility	Zipfian	Utilization (%)
2,000	429	1.265	0.197	1.00	0.824	97.7
4,000	521	1.280	0.197	1.00	0.844	97.5
8,000	616	1.288	0.197	1.00	0.841	97.1
12,000	683	1.291	0.197	1.00	0.839	95.6
16,000	723	1.293	0.197	1.00	0.843	94.3
20,000	751	1.293	0.197	1.00	0.842	92.3
24,000	777	1.294	0.197	1.00	0.844	91.2
32,000	818	1.295	0.197	1.00	0.844	89.5
40,000	849	1.295	0.197	1.00	0.846	87.4
48,000	866	1.295	0.197	1.00	0.845	86.4
64,000	907	1.295	0.197	1.00	0.846	83.4

Table A.7: TokenMonster Algorithm Results

Vocab Size	Actual Vocab	Renyi Entropy	NSL	Fertility	Zipfian	Utilization (%)
2,000	2,000	4.008	0.812	4.117	0.612	2.00
4,000	4,000	4.008	0.812	4.117	0.612	1.00
8,000	8,000	4.008	0.812	4.117	0.612	0.50
12,000	12,000	4.008	0.812	4.117	0.612	0.33
16,000	16,000	4.008	0.812	4.117	0.548	0.26
20,000	20,000	4.008	0.812	4.117	0.548	0.21
24,000	24,000	4.008	0.812	4.117	0.548	0.17
32,000	32,000	4.008	0.812	4.117	0.548	0.13
40,000	40,000	4.008	0.812	4.117	0.521	0.11
48,000	48,000	4.008	0.812	4.117	0.521	0.09
64,000	64,000	4.008	0.812	4.117	0.521	0.07

Table A.8: Unigram Algorithm Results (Most Responsive to Vocab Size)

Vocab Size	Renyi Entropy	NSL	Fertility	Zipfian	Utilization (%)	Unique Toke
2,000	4.00	0.490	2.48	0.90	27.5	5
4,000	4.43	0.428	2.17	0.93	30.2	1,2
8,000	4.75	0.383	1.94	0.92	32.9	2,6
12,000	5.05	0.357	1.81	0.92	35.1	4,2
16,000	5.23	0.344	1.74	0.92	36.8	5,8
20,000	5.40	0.334	1.70	0.93	37.7	7,5
24,000	5.61	0.326	1.65	0.94	37.8	9,0
32,000	5.88	0.317	1.61	0.95	36.4	11,6
40,000	6.11	0.311	1.57	0.96	34.2	13,6
48,000	6.25	0.307	1.56	0.97	31.6	15,1
64,000	6.80	0.303	1.54	0.97	26.2	16,7