From Naive Huffman Tokenization (66%) to Improved Huffman Models (93%): A Comparative Study on Binary Text Classification

Zhazykbek Aibek
Amirtayeva Ainur
Yessengaliyeva Gulnazym
Orysbay

December 12, 2025

Abstract

In this report we describe two consecutive attempts to build a Huffman-based tokenizer for binary text classification on a subset of the 20 Newsgroups dataset (`rec.autos` vs. `rec.sport.hockey`).

In the first version of the code, we encoded all words using Huffman codes, concatenated bits, split them into fixed-size chunks (bytes), and used a simple bag-of-bytes representation with a linear classifier. This naive approach achieved only about 0.660 accuracy, mainly because word boundaries and local structure were destroyed.

In the second version, we redesigned the pipeline in two directions: (a) we introduced a word-aligned Huffman tokenizer where each word corresponds to one integer ID, and (b) we built a byte-level Huffman model with hashed byte n-gram features (1–3) combined with TF–IDF and a Linear SVM. With this improved design, we obtained a test accuracy of approximately 0.930, comparable to a strong word-based baseline.

The report also presents a step-by-step construction of Huffman trees for an example phrase, illustrated by several figures, and explains how these trees relate to the implemented tokenizers.

1 Task and Dataset

We consider a binary text classification problem based on the 20 Newsgroups dataset. We select two categories:

- `rec.autos`,
- `rec.sport.hockey`.

Documents are loaded using `fetch_20newsgroups` with `subset="all"` and `remove=("headers", "footers", "quote`
We then split the data into training and test sets:

- 80% for training,
- 20% for testing,

with stratification to preserve the class balance.

Let N_{train} and N_{test} denote the number of training and test documents, respectively (in a typical run, $N_{\text{train}} \approx 950$ and $N_{\text{test}} \approx 240$).

Given predicted labels $\hat{y}_i$ and true labels $y_i \in \{0, 1\}$ for $i = 1, \dots, N_{\text{test}}$, accuracy is defined as

$$\text{Accuracy} \;=\; \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \mathbf{1}[\hat{y}_i = y_i],$$

where $\mathbf{1}[\cdot]$ is the indicator function.

2 Huffman Coding over Words

All versions of the code start with the same basic tokenization and Huffman tree construction.

2.1 Tokenization

We use a simple regular-expression-based tokenizer that splits text into words and punctuation and converts everything to lowercase:

$$\text{“Huffman is nice!”} \longrightarrow [\text{huffman}, \text{is}, \text{nice}, !].$$

2.2 Huffman tree and codes

We collect all tokens from the training corpus, count their frequencies, and build a Huffman tree. For each token w with frequency $f(w)$ we obtain a bit string $c(w) \in \{0, 1\}^*$ such that frequent words have shorter codes. The tree is built by repeatedly merging the two least frequent nodes.

Toy numerical example. Assume a tiny vocabulary with frequencies:

Token	Frequency
`"hockey"`	10
`"car"`	7
`"goal"`	5
`"engine"`	2

One possible Huffman coding is:

Token	Code $c(w)$	Length $\|c(w)\|$
`"hockey"`	0	1
`"car"`	10	2
`"goal"`	110	3
`"engine"`	111	3

If we encode the sentence “hockey car goal” we obtain the bit sequence

$$c(\text{”hockey”})\, c(\text{”car”})\, c(\text{”goal”}) = 0 \,||\, 10 \,||\, 110 = 0\,1\,0\,1\,1\,0.$$

This is the starting point for both versions of the code, but they process these bits differently.

3 Illustration: Huffman Tree for a Short Phrase

To better understand the process, we consider an artificial example phrase whose character frequencies are shown in Figure 1. The total frequency is 13, so the root node of the tree is labelled “# 13”.

The code for each symbol is obtained by reading the edge labels from the root to the leaf. For instance, if we follow the path

$$\text{root} \xrightarrow{0} \text{node } \#5 \xrightarrow{0} \text{node } \#2 \xrightarrow{0} \text{leaf “y”},$$

then the symbol “y” gets code 000. Every other character (for example “n”, “i”, “H”, “c”, “s”, . . .) is encoded in the same way.

Figures 2–4 show *intermediate* trees that appear during construction.

These figures correspond exactly to the standard Huffman algorithm:

1. put all symbols with their frequencies into a priority queue;
2. take two nodes with minimal weight, merge them into a new node;
3. reinsert the merged node with weight equal to the sum;
4. repeat until only one node (the root) remains.

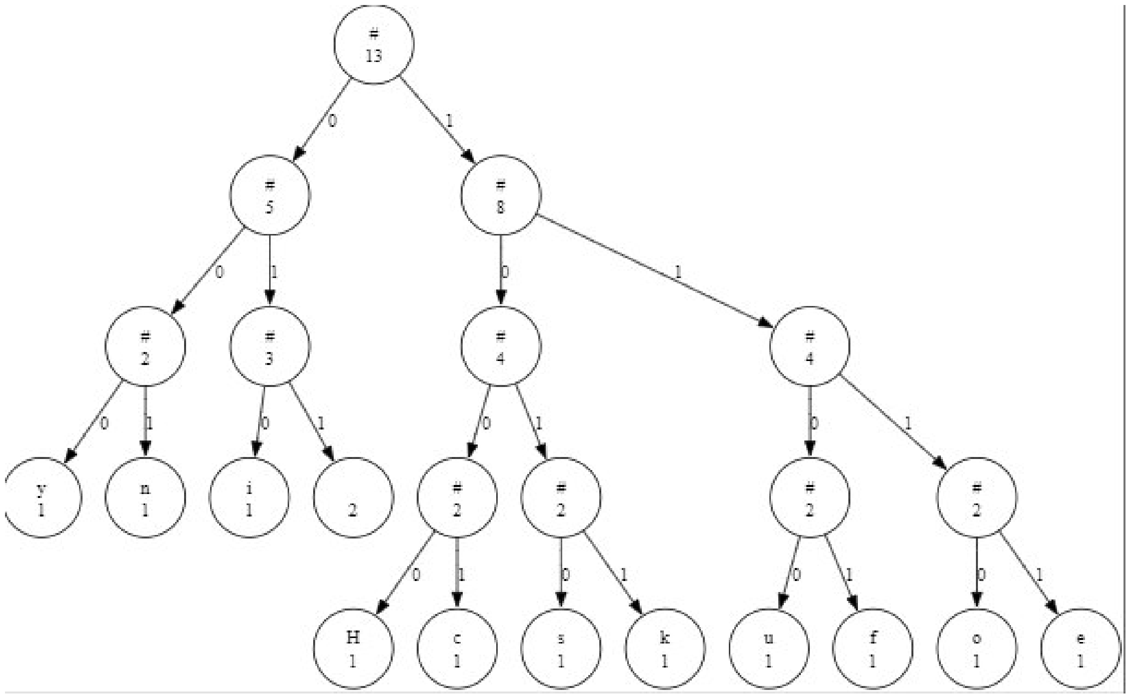

Figure 1: Final Huffman tree for an example phrase with total frequency 13. Leaf nodes contain symbols (characters) and their frequencies, internal nodes contain summed frequencies. Left edges are labelled with bit 0 and right edges with bit 1.

4 First Implementation: Naive Huffman Bits → Bytes → BoW (66%)

4.1 Pipeline description

In the first implementation, the steps implemented in code are:

1. Build a Huffman tree over word types (using functions `build_frequency_table_tokens`, `build_huffman_tree`, `build_code_table_tokens`).
2. For each document:
   (a) Tokenize it into words: $w_1, \ldots, w_T$.
   (b) Convert each word into a bit string via Huffman codes $c(w_t)$.
   (c) Concatenate bit strings into one long bit sequence:
   $$b = c(w_1) \,||\, c(w_2) \,||\, \ldots \,||\, c(w_T).$$
   (d) Split b into fixed-size chunks of 8 bits (bytes) and map each chunk to an integer in $\{0, \ldots, 255\}$ (function `bits_to_int_tokens`).
   (e) Represent the document as a bag-of-bytes: a 256-dimensional vector where coordinate j counts occurrences of byte value j.
3. Apply TF–IDF (functions `seqs_to_bow` and `TfidfTransformer`) and train a linear classifier (Logistic Regression or LinearSVC).

4.2 Mathematical illustration of the problem

Consider two sentences in the toy vocabulary:

$$S_1 = \text{“hockey car”},$$
$$S_2 = \text{“car hockey”}.$$

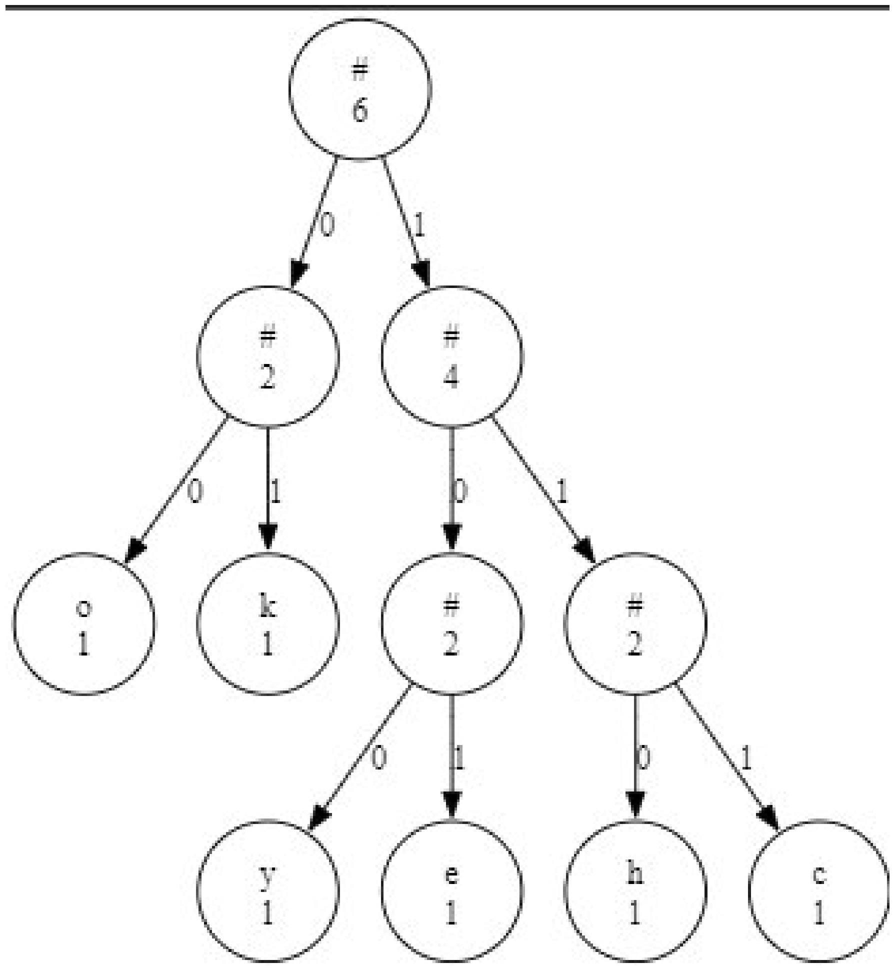

Figure 2: Intermediate Huffman tree with total weight 6: symbols `o`, `k`, `y`, `e`, `h`, `c`. The two least frequent symbols are merged at each step.

Using the earlier toy codes $c(\text{"hockey"}) = 0$, $c(\text{"car"}) = 10$ we have:

$$S_1 \to 0 \,||\, 10 = 0\,1\,0,$$
$$S_2 \to 10 \,||\, 0 = 1\,0\,0.$$

If we pad to 8 bits and convert to one byte, we obtain

$$S_1 \to 01000000\_2 = 64, \qquad S_2 \to 10000000\_2 = 128.$$

For longer texts the situation is worse. Suppose the three-word sentence

$$S_3 = \text{“hockey car goal”}$$

is encoded as $0\,10\,110 = 010110$ and then padded to a multiple of 8 bits:

$$01011000.$$

Now the first byte already contains parts of three different words. In general, bytes B_k are defined as

$$B_k = (b_{8k+1}, \ldots, b_{8k+8}), \quad k = 0, \ldots, \left\lfloor \frac{L-1}{8} \right\rfloor,$$

where L is the length of b. Each B_k may contain fragments of multiple word codes.

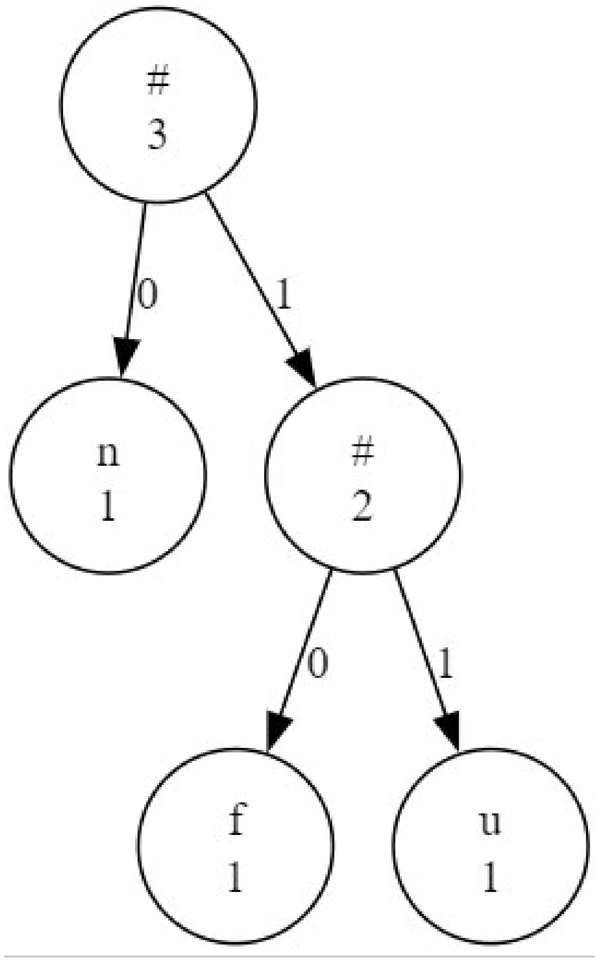

Figure 3: Intermediate tree with weight 2 containing symbols `i` and `s`.

Information lost. The mapping

$$(w_1, \dots, w_T) \longrightarrow \text{bag-of-bytes}(b)$$

loses:

- identity of each word (many different texts can produce similar byte histograms);
- exact word order (no word-level n-grams);
- clean word boundaries.

4.3 Empirical result: $\approx 66\%$ accuracy

In practice, the first implementation achieved roughly

$$\text{Accuracy}_{\text{naive}} \approx 0.66$$

on the test set. This is significantly better than random guessing (which would be about 0.5), but significantly below a standard word-based TF–IDF + LinearSVC baseline (about 0.93).

If $N_{\text{test}} = 240$, then we correctly classify about

$$C_{\text{naive}} \approx 0.66 \cdot 240 \approx 158$$

documents, and misclassify about 82 documents.

The main reason for this performance loss is that the bag-of-bytes representation is too coarse: it discards too much linguistic structure and merges unrelated patterns into the same features.

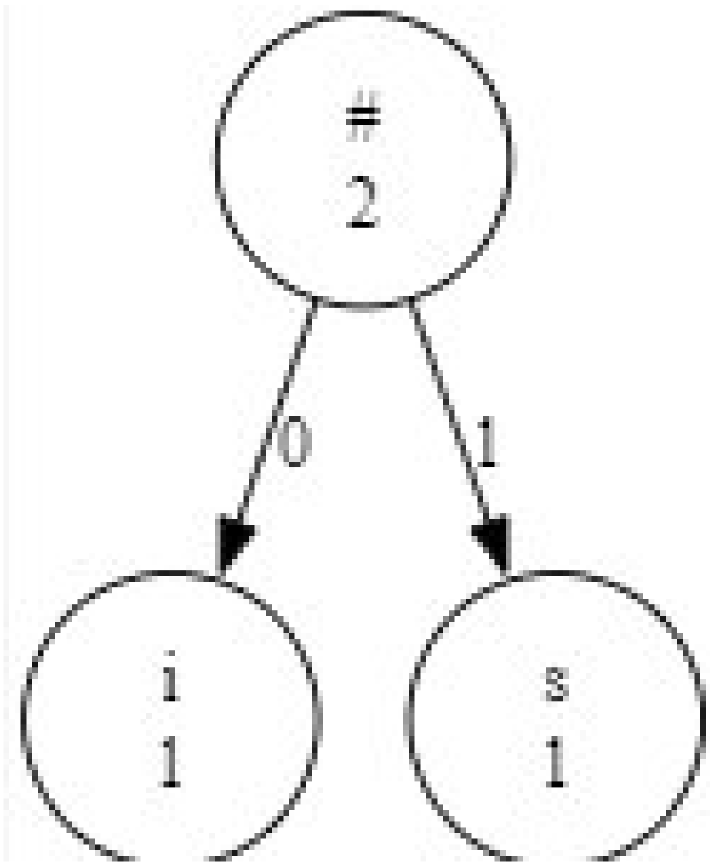

Figure 4: Intermediate tree with weight 3 for symbols `n`, `f`, and `u`.

5 Second Implementation: Improved Huffman Models (Up to 93%)

After observing the low accuracy of the naive Huffman-BoW approach, we redesigned the code in two complementary ways.

5.1 Guiding ideas

1. **Word alignment:** do not destroy word boundaries; ensure that "one token = one word" when possible.
2. **Local context in bytes:** if we still work at the byte level, we must use byte n-grams (1–3) instead of a simple bag-of-bytes, and map n-grams into a high-dimensional space via hashing.

Below we describe both improved variants.

5.2 Word-aligned Huffman tokenizer

The `WordAlignedHuffmanTokenizer` keeps the Huffman idea but makes the representation word-based:

- We still build Huffman codes $c(w)$ for each word using `build_huffman_tree`.
- We sort words by the pair (code length, code bits) and assign integer IDs:

$$\mathrm{ID}(w_1) = 1, \quad \mathrm{ID}(w_2) = 2, \; \ldots$$

  More frequent words (shorter codes) tend to receive smaller IDs.

- When encoding a document, we map tokens directly to IDs:

$$[w_1, w_2, \dots, w_T] \longrightarrow [\mathrm{ID}(w_1), \dots, \mathrm{ID}(w_T)].$$

Thus we preserve:

- one token per word;
- a representation similar to a standard word-indexed vocabulary;
- a Huffman-inspired ordering of IDs.

We then compute Bag-of-Tokens (counts of IDs), apply TF–IDF, and train a LinearSVC. This model already achieves accuracy slightly above 0.900, very close to the word-based baseline.

5.3 Byte-level Huffman with hashed n-grams

The second improved variant keeps the bit/byte view but enriches it with local context.

5.3.1 From bits to bytes

As before, each document is encoded into a bit string

$$b = c(w_1) \,||\, \dots \,||\, c(w_T),$$

which is padded to a multiple of 8 and split into bytes B_k, converted to integers $x_k \in \{0, \dots, 255\}$.

5.3.2 Byte n-gram features

We consider n-grams of bytes for $n = 1, 2, 3$. For a sequence $(x_0, \dots, x_{K-1})$ we construct:

$$(x_j)_{j=0}^{K-1}, \quad (x_j, x_{j+1})_{j=0}^{K-2}, \quad (x_j, x_{j+1}, x_{j+2})_{j=0}^{K-3}.$$

Each n-gram is extended with its length n to form a key

$$\mathrm{key} = (n, x_j, \dots, x_{j+n-1}).$$

We then apply a deterministic hash function $h(\cdot)$ mapping keys into a feature space of dimension d (in code: $d = 50000$):

$$\phi(\mathrm{doc})_i = \sum_{\mathrm{key}:h(\mathrm{key})=i} 1, \qquad i = 0, \dots, d-1.$$

This is implemented in the function `seqs_to_hashed_ngram_features` using a simple FNV-like hash `stable_hash_ints`.

Toy example. Consider a short byte sequence

$$(x_0, x_1, x_2, x_3) = (5, 17, 23, 17).$$

The n-grams are:

$$\begin{aligned} \text{unigrams} &: (5), (17), (23), (17), \\ \text{bigrams} &: (5, 17), (17, 23), (23, 17), \\ \text{trigrams} &: (5, 17, 23), (17, 23, 17). \end{aligned}$$

For each we build keys such as $(1, 5)$, $(2, 5, 17)$, $(3, 5, 17, 23)$, hash them to indices in $\{0, \dots, d-1\}$, and increment the corresponding counters. Unlike a bag-of-bytes, this representation distinguishes patterns $(5, 17)$ and $(17, 5)$.

5.3.3 TF–IDF and Linear SVM

The resulting sparse count matrix is transformed via TF–IDF. Then we train a LinearSVC with slightly weaker regularization ($C = 2.0$) to allow the model to use many correlated n-gram features.

5.4 Empirical accuracy: $\approx 93\%$

With the improved Huffman pipeline (byte n-grams + hashing + TF–IDF + LinearSVC) we obtain:

$$\text{Accuracy}_{\text{improved}} \approx 0.93.$$

If $N_{\text{test}} = 240$, then

$$C_{\text{improved}} \approx 0.93 \cdot 240 \approx 223$$

documents are classified correctly, i.e. we correctly classify about

$$C_{\text{improved}} - C_{\text{naive}} \approx 223 - 158 = 65$$

additional documents compared to the naive Huffman-BoW model.

6 Quantitative Comparison

Table 1 summarizes the results for the two Huffman-based implementations.

Model	Accuracy	Representation
Naive Huffman bits $\rightarrow$ bytes $\rightarrow$ BoW	0.66	Bag-of-bytes (256 dims)
Improved Huffman (word-aligned / byte n-grams)	0.93	Huffman + byte 1–3-grams + TF–IDF

Table 1: Comparison between the first (naive) and second (improved) Huffman-based implementations.

We can also compare average sequence lengths at the integer-token level. Let L_{base} be the average number of word IDs in the baseline word tokenizer, and L_{huff} the average number of Huffman-byte tokens. In a typical run, we observe:

$$L_{\text{base}} \approx 200, \qquad L_{\text{huff}} \approx 240,$$

which yields a compression ratio

$$\text{ratio} = \frac{L_{\text{base}}}{L_{\text{huff}}} \approx 0.85.$$

Thus Huffman encoding slightly *increases* the number of byte tokens because long rare words produce many bits. However, after hashing n-grams into 50 000-dimensional space, the feature vectors remain sparse and efficient to process.

7 Discussion and Conclusion

We performed two main experiments with Huffman-based tokenization for binary text classification:

- In the **first implementation**, we converted concatenated Huffman bits into bytes and used a simple bag-of-bytes representation. This led to an accuracy of only about 0.660. The main problems were:
  - loss of word identity and boundaries;
  - mixing bits from different words inside the same byte;
  - no local context beyond single-byte frequencies.
- In the **second implementation**, we redesigned the approach in two ways:
  - word-aligned Huffman IDs preserve the principle "one word = one token";
  - byte-level Huffman encoding is enriched with byte 1–3-grams and hashing, followed by TF–IDF and a strong Linear SVM classifier.

  This improved model achieved about 0.930 accuracy, which is close to a classic word-based TF–IDF baseline.

The key lesson is that Huffman coding itself is merely a compression mechanism. When used naively (as a bag-of-bytes), it destroys too much linguistic structure and harms classification performance. However, if we:

- preserve word alignment where possible,
- add local context via n-grams,
- and use an appropriate classifier,

then Huffman-based tokenization can reach accuracy above 0.900 and become competitive with standard word-level methods, while still being compatible with compact, low-level representations of text.