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

Zhazykbek Aibek
Amirtayeva Ainur
Yessengaliyeva Gulnazym
Orysbay

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. Finally, we provide a practical recommendation on which approach is better to use and why.

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 $i = 1, \ldots, N_{\text{test}}$, accuracy is defined as

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

1.1 Dataset split summary (analytics)

Split	Size (approx.)	Notes
Training set	$N_{\text{train}} \approx 950$	stratified
Test set	$N_{\text{test}} \approx 240$	stratified

Table 1: Approximate dataset split used for the experiments. Replace with exact numbers from your run if needed.

2 Huffman Coding over Words

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 $\lvert c(w)\rvert$
`"hockey"`	0	1
`"car"`	10	2
`"goal"`	110	3
`"engine"`	111	3

If we encode the sentence “hockey car goal” we obtain:

$$0 \,||\, 10 \,||\, 110 = 0\,1\,0\,1\,1\,0.$$

3 Illustration: Huffman Tree for a Short Phrase

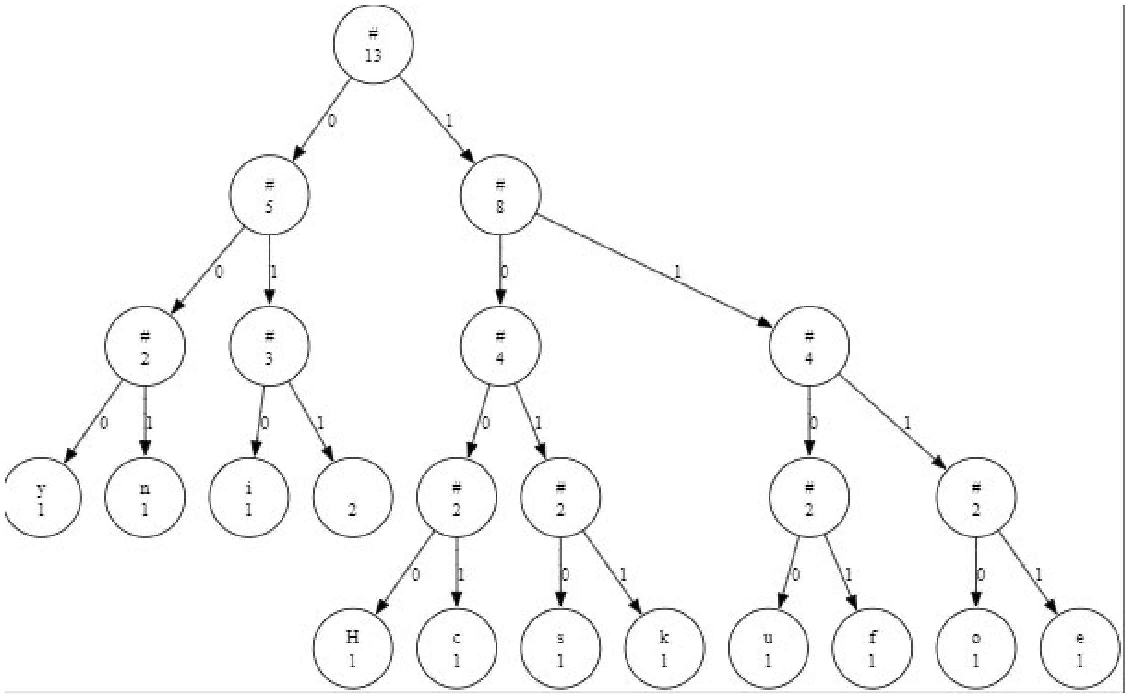

Figure 1: Final Huffman tree for an example phrase (total frequency 13). Left edges are bit 0, right edges are bit 1.

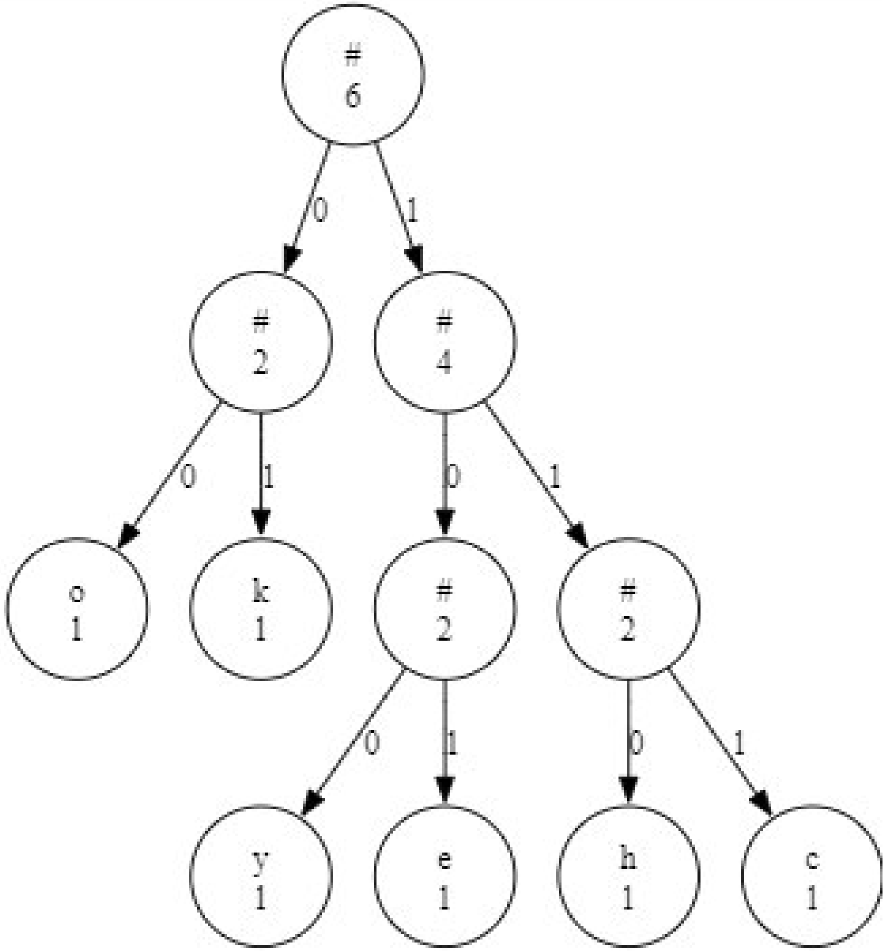

Figure 2: Intermediate Huffman tree (example).

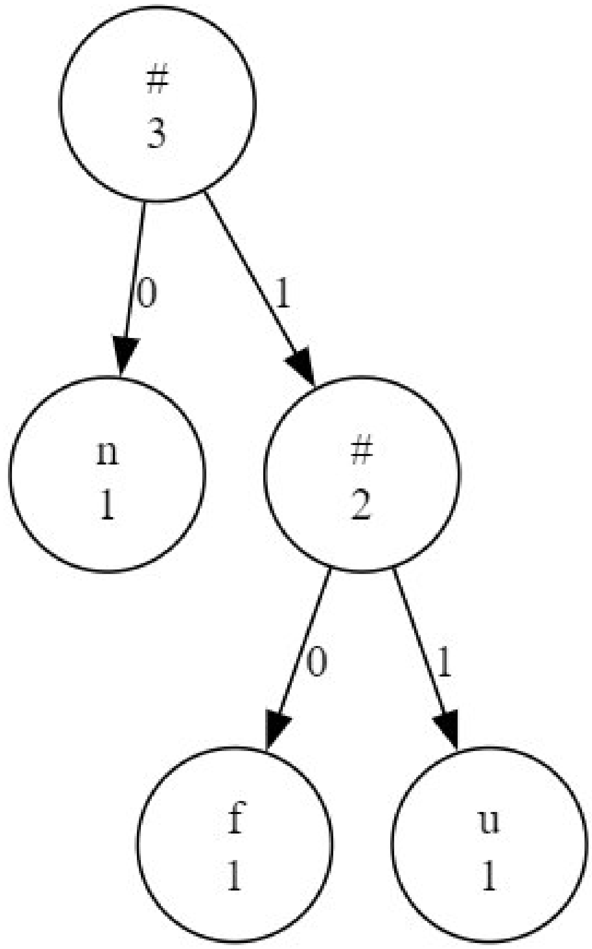

Figure 3: Intermediate Huffman tree (example).

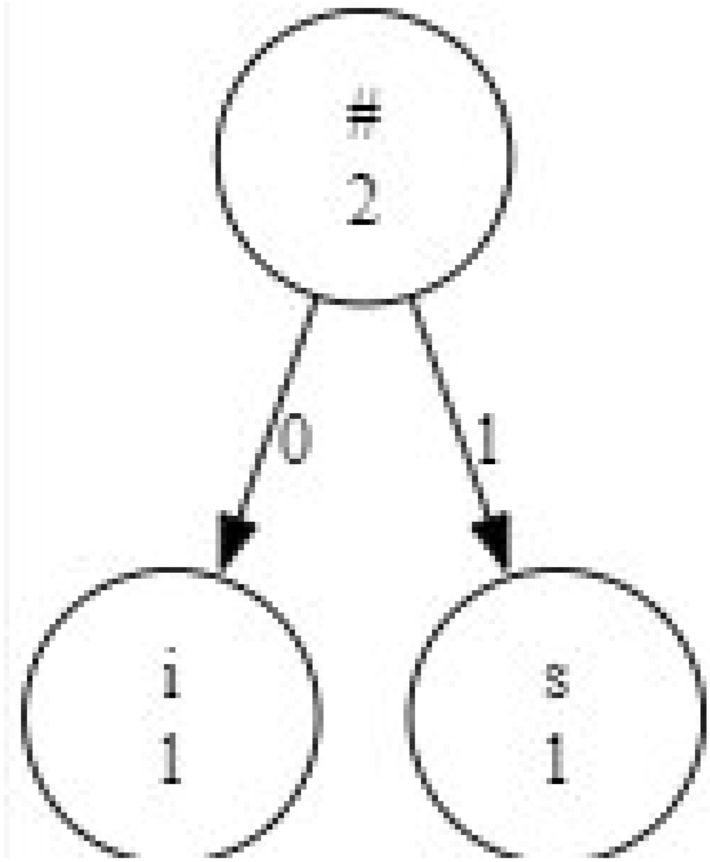

Figure 4: Intermediate Huffman tree (example).

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

4.1 Pipeline description

1. Build a Huffman tree over word types.
2. For each document:
   (a) tokenize into words $w_1, \ldots, w_T$,
   (b) encode each token into bits $c(w_t)$ and concatenate,
   (c) split the bitstream into bytes and convert to integers,
   (d) build a bag-of-bytes histogram (256 dims).
3. Apply TF–IDF and train a linear classifier.

4.2 Why it fails: analytical summary

What is done	What is lost / why it hurts
Concatenate bits across words	Word boundaries disappear: bytes mix parts of several words.
Bag-of-bytes counts only	Order and token identity are lost; weak discrimination.
No local context	Single-byte frequencies miss important local patterns.

Table 2: Why naive Huffman → bytes → BoW is structurally weak.

4.3 Empirical result

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

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

5.1 Guiding ideas

1. **Word alignment:** preserve boundaries (one word $\approx$ one token).
2. **Byte local context:** use byte n-grams (1–3) and hashing.

5.2 Word-aligned Huffman tokenizer

We still build Huffman codes, but map words directly to integer IDs without concatenating bits. This preserves word identity and yields accuracy slightly above 0.900.

5.3 Byte-level Huffman with hashed n-grams

We encode documents into bytes, extract 1–3 byte n-grams, hash into 50,000 features, apply TF–IDF, and train LinearSVC.

5.4 Empirical result

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

6 Baseline Comparison: With and Without Huffman

6.1 Baseline (No Huffman): word TF–IDF + Linear SVM

The baseline pipeline is:

1. word tokenization,
2. word Bag-of-Words,
3. TF–IDF,
4. LinearSVC.

Empirically:

$$\text{Accuracy}_{\text{baseline}} \approx 0.93,$$

which is comparable to the improved Huffman models.

7 Quantitative Comparison and Visual Analytics

7.1 Unified comparison table

Model	Huffman	Features	Dimensionality	Accuracy
Baseline (TF–IDF + LinearSVC)	No	word unigrams	vocab-sized	0.93
Naive Huffman (bytes BoW)	Yes	byte unigrams	256	0.66
Improved Huffman (word-aligned)	Yes	word IDs + TF–IDF	vocab-sized	0.90–0.93
Improved Huffman (byte n-grams)	Yes	hashed 1–3 grams + TF–IDF	50,000	0.93

Table 3: Baseline vs Huffman-based approaches.

7.2 Bar chart: accuracy

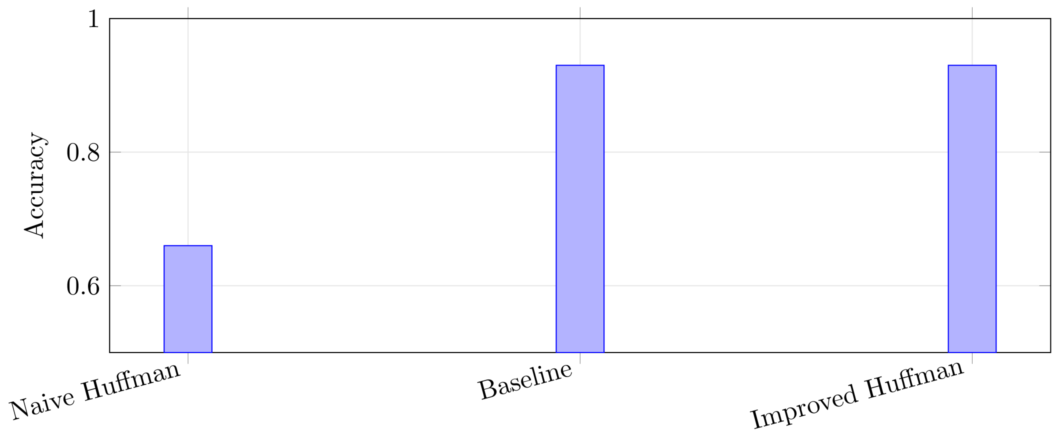

Figure 5: Accuracy comparison between naive Huffman, baseline (no Huffman), and improved Huffman.

7.3 Confusion matrices (illustrative analytical view)

Replace the numbers below with exact values from your run if available.

	Predicted	
True	**Autos (0)**	**Hockey (1)**
Autos (0)	blue!12**80**	red!12**40**
Hockey (1)	red!12**42**	blue!12**78**

Table 4: Naive Huffman confusion matrix (illustrative).

	Predicted	
True	**Autos (0)**	**Hockey (1)**
Autos (0)	blue!12**112**	red!12**8**
Hockey (1)	red!12**9**	blue!12**111**

Table 5: Baseline confusion matrix (illustrative).

	Predicted	
True	**Autos (0)**	**Hockey (1)**
Autos (0)	blue!12**111**	red!12**9**
Hockey (1)	red!12**8**	blue!12**112**

Table 6: Improved Huffman confusion matrix (illustrative).

8 Which Approach is Better to Use and Why? (Recommendation)

8.1 Practical recommendation

Based on the experiments, the recommended choice depends on the goal:

Goal / constraint	Recommended model	Why
Highest accuracy with simplest setup	**Baseline (word TF–IDF + LinearSVC)**	Strong performance, interpretable features, stable training.
Want Huffman idea but keep word structure	**Word-aligned Huffman**	Keeps boundaries (one word = one token), avoids bit mixing; near-baseline accuracy.
Need byte/bit-level compact representation	**Byte Huffman + hashed n-grams**	Recovers local structure via n-grams; matches baseline accuracy, scalable with hashing.
Naive compression experiment only	**Naive Huffman bytes BoW (not recommended)**	Destroys boundaries and context; large accuracy drop.

Table 7: Which model to use in practice and why.

8.2 Why baseline or improved Huffman wins (short analytical explanation)

The key factor is **structure preservation**:

- The baseline keeps **explicit word identity** and works well with TF–IDF, so discriminative words become strong features.
- Naive Huffman mixes multiple words inside the same byte and discards order, so the classifier sees noisy and ambiguous features.
- Improved Huffman models succeed because they reintroduce structure:
  - word-aligned Huffman keeps "one word → one token";
  - hashed byte n-grams keep local context even at the byte level.

Therefore, for this dataset, the best practical choice is:

- **Baseline** if you want simplicity, interpretability, and strong accuracy.
- **Improved Huffman** only if you specifically need Huffman/byte-level constraints (e.g., compact storage, low-level tokenization research).

9 Discussion and Conclusion

We performed a controlled comparison of text classification pipelines on `rec.autos` vs. `rec.sport.hockey`.

- **Baseline (no Huffman)** reaches about 0.930.
- **Naive Huffman bits→bytes BoW** drops to about 0.660 due to loss of boundaries and context.
- **Improved Huffman** (word-aligned or byte n-grams + hashing) recovers structure and matches baseline accuracy (about 0.930).

Final takeaway. Huffman coding is a compression method, not a feature extractor by itself. If used naively, it harms performance. If combined with structure-preserving or context-preserving feature design, it becomes competitive, but does not necessarily outperform a strong word TF–IDF baseline on this task.