Naive Bayes Sentiment Analysis on IMDB Reviews

Paola Calle

November 29, 2025

Abstract

This short report summarizes a set of experiments using Naive Bayes classifiers for sentiment analysis on the IMDB movie review dataset. I compare three implementations: (1) scikit-learn's Multinomial Naive Bayes with CountVectorizer features, (2) a from-scratch Naive Bayes model operating directly on cleaned text, and (3) a from-scratch Naive Bayes model built on top of CountVectorizer, designed to match the Multinomial Naive Bayes formulation. I study the effect of the smoothing parameter α and inspect the most sentiment-indicative words according to the model.

1 Introduction

Naive Bayes is a simple probabilistic classifier often used as a baseline for text classification tasks. Despite its strong (and usually false) independence assumptions, it can perform surprisingly well on high-dimensional sparse features such as bag-of-words representations.

In this project, I focus on binary sentiment classification of IMDB movie reviews (negative vs. positive). The goals are:

- to implement Naive Bayes from scratch and compare it to scikit-learn's implementation,
- to understand the impact of the smoothing parameter α on accuracy,
- and to interpret which words are most strongly associated with each sentiment.

2 Data and Preprocessing

I use the IMDB dataset as provided by the HuggingFace `datasets` library. The dataset contains 25,000 training reviews and 25,000 test reviews with binary labels:

$$\text{label} = \begin{cases} 0 & \text{negative review} \\ 1 & \text{positive review.} \end{cases}$$

For the *CountVectorizer-based* experiments, I transform the raw text into bag-of-words features using:

- unigrams and bigrams: `ngram_range = (1, 2)`,
- minimum document frequency: `min_df = 5`,
- maximum document frequency: `max_df = 0.8`,

- tokens restricted to at least 2 word characters.

For the *raw-text* Naive Bayes implementation, I use a simpler manual tokenization based on lowercasing and stripping punctuation, without n-grams or document-frequency pruning. This deliberately weaker preprocessing lets me see how much performance is driven by the model vs. the features.

3 Models

I evaluate three models:

1. **MultinomialNB (sklearn):** Scikit-learn's `MultinomialNB` trained on CountVectorizer features. This serves as the reference implementation.
2. **Custom Naive Bayes (raw text):** A from-scratch implementation that:
   - splits cleaned text into tokens,
   - counts token occurrences per class,
   - applies Laplace/Lidstone smoothing with parameter α,
   - uses log-probabilities and the Naive Bayes assumption

$$\log P(y \mid x) \propto \log P(y) + \sum_j \log P(x_j \mid y).$$

3. **Custom Naive Bayes + CountVectorizer:** A from-scratch implementation that operates on the CountVectorizer document–term matrix but implements the Multinomial Naive Bayes math manually. This model is designed to mirror `MultinomialNB` and serves primarily as a correctness check.

4 Effect of the Smoothing Parameter α

To study the effect of smoothing, I sweep α over a range (for example, $\alpha \in \{0.001, 0.002, \ldots, 1.0\}$ on a log scale) and record test accuracy for each model.

Table **??** shows a summary of the best performance achieved by each model across all tested α values.

Model	**Best α**	**Best Accuracy**
MultinomialNB (sklearn)	0.10	0.84
Custom NB + CountVectorizer	0.10	0.84
Custom NB (raw text)	0.10	0.75

Table 1: Best test accuracy over the α sweep for each model. (Fill in with your actual values.)

In my experiments, the CountVectorizer-based custom Naive Bayes matches the performance of scikit-learn's `MultinomialNB` almost exactly for every α tested. This indicates that the from-scratch implementation correctly reproduces the Multinomial Naive Bayes formulation when given the same features.

The raw-text model consistently underperforms, with accuracy several points lower than the CountVectorizer-based models. This is expected: its tokenization and vocabulary handling are much simpler, it does not use n-grams, and it does not prune rare or overly frequent terms.

If desired, the accuracy as a function of α can be visualized in a plot (e.g., with α on a log scale and separate curves for each model).

5 Vocabulary Importance

One advantage of Naive Bayes is that it is easy to inspect which words are most indicative of each class. Using the learned conditional probabilities, I compute the log-odds

$$\text{log-odds}(w) = \log P(w \mid \text{positive}) - \log P(w \mid \text{negative}).$$

Words with large positive log-odds are strongly associated with positive reviews, while words with large negative log-odds are strongly associated with negative reviews.

In my runs, typical strongly positive tokens included words and phrases like:

- "excellent",
- "highly recommend",
- "wonderful",

while strongly negative tokens included:

- "worst",
- "boring",
- "waste of time",

and similar expressions.

These patterns are intuitive and confirm that even a simple Naive Bayes model is able to extract meaningful sentiment cues from text.

6 Conclusion

This small project shows that:

- Naive Bayes is a competitive baseline for sentiment analysis when paired with strong preprocessing (CountVectorizer, n-grams, and vocabulary pruning).
- A from-scratch implementation of Multinomial Naive Bayes can closely match scikit-learn's `MultinomialNB` when given the same features and smoothing parameter.
- The performance gap between the CountVectorizer-based models and the raw-text model highlights how important feature engineering and tokenization decisions are for text classification.

Overall, the experiments were useful both for validating the probabilistic implementation and for building intuition about how preprocessing and smoothing interact in Naive Bayes classifiers.