Implement the Bernoulli Naive Bayes algorithm

Alright — let's implement **Bernoulli Naive Bayes** from scratch in Python, step by step.

The **Bernoulli Naive Bayes** algorithm is commonly used for **binary features** (e.g. presence/absence of a word in a document).
It's well suited for **text classification** using a **binary bag-of-words** model.

🧠 Step 1: Mathematical Overview

For a document $x = (x_1, x_2, \ldots, x_n)$
and a class c, the **posterior** is

$$P(c \mid x) \propto P(c) \prod_{i=1}^{n} P(x_i \mid c)$$

For Bernoulli NB:

- Each feature $x_i \in \{0, 1\}$ indicates **whether the word appears**.
- The likelihood:

$$P(x_i \mid c) = \begin{cases} \theta_{i|c} & \text{if } x_i = 1 \\ 1 - \theta_{i|c} & \text{if } x_i = 0 \end{cases}$$

- Where
  \theta_{i|c} = \frac{\text{# of documents in class c containing word i} + \alpha}{\text{# of documents in class c} + 2\alpha}

  (α is Laplace smoothing, usually α = 1)

The predicted class is the one maximizing log posterior:

$$\hat{c} = \arg\max_c \left[\log P(c) + \sum_{i=1}^{n} \log P(x_i \mid c) \right]$$

🧩 Step 2: Python Implementation

We'll implement:

- `fit(X, y)` : estimate priors and likelihoods
- `predict(X)` : classify new samples
- `predict_proba(X)` : optional, for class probabilities

python Copy code

```python
import numpy as np

class BernoulliNaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha
        self.classes = None
        self.class_log_prior_ = None
        self.feature_log_prob_ = None

    def fit(self, X, y):
        """
        X: binary matrix of shape (n_samples, n_features)
        y: array of shape (n_samples,)
        """
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        # Initialize arrays
        self.class_log_prior_ = np.zeros(n_classes)
        self.feature_log_prob_ = np.zeros((n_classes, n_features))

        for idx, c in enumerate(self.classes):
            X_c = X[y == c]
            # prior P(c)
            self.class_log_prior_[idx] = np.log(X_c.shape[0] / n_samples)

            # likelihoods with Laplace smoothing
            # count of docs in class c containing word i
            feature_count = np.sum(X_c, axis=0)
            smoothed = (feature_count + self.alpha) / (X_c.shape[0] + 2 * self.al
            self.feature_log_prob_[idx] = np.log(smoothed)

        return self

    def predict_log_proba(self, X):
        """
        Compute log posterior for each class
        """
        n_samples, n_features = X.shape
        log_probs = np.zeros((n_samples, len(self.classes)))

        for idx, c in enumerate(self.classes):
            # log P(c)
            log_prior = self.class_log_prior_[idx]
            log_likelihood = self.feature_log_prob_[idx]
            log_likelihood_neg = np.lo... - np.exp(log_likelihood))
```

Let's test it on a toy example:

```python
# Example dataset
# Features: [word1, word2, word3]
# Classes: 0 = spam, 1 = ham
X_train = np.array([
    [1, 0, 1],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
])

y_train = np.array([0, 0, 1, 1])

model = BernoulliNaiveBayes(alpha=1)
model.fit(X_train, y_train)

# New document: contains word1 and word2, but not word3
X_test = np.array([
    [1, 1, 0],
    [0, 0, 1]
])

print("Predictions:", model.predict(X_test))
```

✅ This should print class predictions for the test samples.

⚡ Step 4: Key Points

- **Bernoulli NB** considers **presence/absence**, not frequency.
- **Laplace smoothing** prevents zero probabilities.
- Works well with binary bag-of-words or TF-IDF binarized data.
- For multi-class, just repeat per class.

Would you like me to implement this again but using **scikit-learn's built-in** `BernoulliNB` for comparison? (It's much shorter.)

what is the formula for a single probability of xi

Good question 👌 — this is the **core** of Bernoulli Naive Bayes.

For a **single feature** x_i (e.g. "does word i appear in the document?"),
the **class-conditional probability** is:

$$P(x_i \mid c) = \begin{cases} \theta_{i|c} & \text{if } x_i = 1 \\ 1 - \theta_{i|c} & \text{if } x_i = 0 \end{cases}$$

where

$$\theta_{i|c} = P(x_i = 1 \mid c)$$

is the probability that feature i is **present** in documents belonging to class c.

📌 Estimation Formula

During training, $\theta_{i|c}$ is estimated using **maximum likelihood with Laplace smoothing**:

$$\theta_{i|c} = \frac{N_{i,c} + \alpha}{N_c + 2\alpha}$$

where:

- $N_{i,c}$ = number of training documents in class c where feature i = 1
- N_c = total number of documents in class c
- α = smoothing parameter (usually 1)

✅ Example

Suppose:

- Class c = "spam"
- Word i = "free"
- There are $N_c = 10$ spam emails
- $N_{i,c} = 7$ of them contain the word "free"
- α = 1

Then

$$\theta_{i|c} = \frac{7 + 1}{10 + 2} = \frac{8}{12} = 0.6667$$

For a new email:

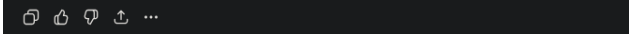
- If it **contains "free"** → $P(x_i = 1|c) = 0.6667$
- If it **doesn't** → $P(x_i = 0|c) = 1 - 0.6667 = 0.3333$

✅ **Summary Formula** for Bernoulli NB, single feature:

$$P(x_i \mid c) = \theta_{i|c}^{x_i} (1 - \theta_{i|c})^{1 - x_i}$$

This compact form works for both $x_i = 0$ and $x_i = 1$.
(You'll notice this is like a Bernoulli distribution formula.)

Would you like me to derive this compact formula step by step? (It's a nice little derivation.)



why is it plus two in the denominator

Excellent catch 👌 — that "**+2**" in the denominator of

$$\theta_{i|c} = \frac{N_{i,c} + \alpha}{N_c + 2\alpha}$$

is one of those small details that **makes Bernoulli Naive Bayes different** from multinomial NB.
Let's unpack it carefully:

🧠 1. We're estimating a Bernoulli probability

For each feature x_i, and class c,

$$\theta_{i|c} = P(x_i = 1 \mid c)$$

This is a probability of **presence** of feature i in class c.
But Bernoulli variables can be **0 or 1**, so there are **two possible outcomes**:

- $x_i = 1$ (feature present)
- $x_i = 0$ (feature absent)

This is different from multinomial NB, where we count occurrences of many possible words.

📊 2. Laplace smoothing adjusts both outcomes

Suppose:

- $N_{i,c}$ = number of documents in class c where feature i is 1
- N_c = total documents in class c

Without smoothing, the maximum likelihood estimate is simply:

$$\hat{\theta}_{i|c} = \frac{N_{i,c}}{N_c}$$

But if $N_{i,c} = 0$, this gives $\hat{\theta}_{i|c} = 0$,
and then $\log P(x_i|c)$ blows up during classification. ❌

So we apply **Laplace smoothing**:

- Add α "pseudo-counts" for $x_i = 1$
- Add α "pseudo-counts" for $x_i = 0$

So total pseudo-counts added = **2α** (because Bernoulli has 2 outcomes).

$$\theta_{i|c} = \frac{N_{i,c} + \alpha}{N_c + 2\alpha}$$

Denominator increases by **2α** because we added α to both numerator (for "present") and the implicit count for "absent".

📝 3. Intuition

Imagine α = 1 (Laplace smoothing):

- It's like **pretending you saw one extra "present" and one extra "absent" example** for each feature in each class.
- This prevents probabilities of exactly 0 or 1, making the model more robust.

⚡ Quick Example

Suppose:

- Class c has N_c = 5 documents
- Feature i appears in N_{i,c} = 0 documents
- α = 1

Without smoothing:

$$\theta_{i|c} = \frac{0}{5} = 0$$

With Laplace smoothing:

$$\theta_{i|c} = \frac{0 + 1}{5 + 2} = \frac{1}{7} \approx 0.1429$$

P(Y = 1 |x) = σ(w
Tx) = 1
1 + exp (−wTx)
=
exp (w
Tx)
1 + exp (wTx

You're writing out the **logistic regression (or sigmoid)** probability function.
Let's rewrite it cleanly and step by step:

We define:

$$P(Y = 1 \mid x) = \sigma(w^\top x)$$

where

- x is the feature vector,
- w is the weight vector,
- $w^\top x$ is their dot product,
- and $\sigma(\cdot)$ is the sigmoid function.

The sigmoid function is:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

Substituting $z = w^\top x$:

$$P(Y = 1 \mid x) = \frac{1}{1 + \exp(-w^\top x)}.$$

We can multiply numerator and denominator by $\exp(w^\top x)$ to get an equivalent form:

$$P(Y = 1 \mid x) = \frac{\exp(w^\top x)}{\exp(w^\top x) + 1}.$$

✅ **Summary of the equivalent forms:**

$$P(Y = 1 \mid x) = \sigma(w^\top x) = \frac{1}{1 + \exp(-w^\top x)} = \frac{\exp(w^\top x)}{1 + \exp(w^\top x)}.$$

All three are mathematically the same — they're just different ways to express the logistic (sigmoid) function.

how would you implement a naive bays text classification on a bag of words in python

Good question 👍. Let's go step by step. I'll show you how to implement a **Naive Bayes text classifier** on a **Bag-of-Words (BoW)** representation from scratch (without scikit-learn), and then also show how you'd do it quickly using `sklearn`.

1. The Math Behind It (Multinomial Naive Bayes)

We want to classify a text d into class $c \in C$.

Naive Bayes computes:

$$P(c \mid d) \propto P(c) \prod_{w \in d} P(w \mid c)^{count(w,d)}$$

Where:

- $P(c)$ = prior probability of class (frequency in training set)
- $P(w \mid c)$ = probability of word w given class (estimated from training data with Laplace smoothing).

2. From Scratch Implementation

python Copy code

```python
import numpy as np
from collections import defaultdict, Counter

class NaiveBayesTextClassifier:
    def __init__(self):
        self.class_priors = {}
        self.word_likelihoods = {}
        self.vocab = set()
        self.class_word_counts = {}
        self.class_total_words = {}

    def fit(self, X, y):
        """
        X: list of documents (each doc is a list of words or tokens)
        y: list of class labels
        """
        class_counts = Counter(y)
        total_docs = len(y)

        # Priors
        self.class_priors = {c: count/total_docs for c, count in class_counts.ite

        # Collect word counts per class
        self.class_word_counts = {c: Counter() for c in class_counts}
        self.class_total_words = {c: 0 for c in class_counts}

        for doc, label in zip(X, y):
            self.class_word_counts[label].update(doc)
            self.class_total_words[label] += len(doc)
            self.vocab.update(doc)

        vocab_size = len(self.vocab)

        # Likelihoods with Laplace smoothing
        self.word_likelihoods = {c: {} for c in class_counts}
        for c in class_counts:
            for word in self.vocab:
                word_count = self.class_word_counts[c][word]
                self.word_likelihoods[c][word] = (word_count + 1) / (self.class_t

    def predict(self, doc):
        """Return the most likely class for a document (list of words)."""
        scores = {}
        for c in self.class_priors:
            # Start with log prior
            log_prob = np.log(self.class_priors[c])
            for word in doc:
                if word in self.vocab:  # ignore OOV
                    log_prob += np.log(self.word_likelihoods[c][word])
            scores[c] = log_prob
        return max(scores, key=scores.get)
```

Example Usage

python Copy code

```python
docs = [
    ["chinese", "beijing", "chinese"],
    ["chinese", "chinese", "shanghai"],
    ["chinese", "macao"],
    ["tokyo", "japan", "chinese"]
]
```

3. Using `scikit-learn`

If you want something quicker:

```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

# Example documents
docs = [
    "Chinese Beijing Chinese",
    "Chinese Chinese Shanghai",
    "Chinese Macao",
    "Tokyo Japan Chinese"
]
labels = ["china", "china", "china", "japan"]

# Bag of Words
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)

# Train NB
clf = MultinomialNB()
clf.fit(X, labels)

# Predict
test = ["Chinese Tokyo Japan"]
X_test = vectorizer.transform(test)
print(clf.predict(X_test))  # ['japan']
```

✅ The first version shows the *mechanics* of Naive Bayes on BoW,
✅ The second version is the *practical sklearn implementation*.

Do you want me to also show you how to extend the from-scratch version so it can handle **probabilities for each class** (instead of just the max)?

can you explain the math behind it in more detail

Absolutely — let's carefully unpack the math behind **Naive Bayes text classification with a bag-of-words model**.

1. Bayes' Rule Refresher

We want to classify a document d into one of the possible classes $c \in C$.

Bayes' theorem says:

$$P(c \mid d) = \frac{P(c)\,P(d \mid c)}{P(d)}$$

- $P(c \mid d)$ → Posterior: probability that class is c given document d.
- $P(c)$ → Prior: how likely class c is overall (class frequency).
- $P(d \mid c)$ → Likelihood: how likely we would see document d if class really were c.
- $P(d)$ → Normalizing constant (same for all classes).

For classification, we just need:

$$\hat{c} = \arg\max_{c}\ P(c)\,P(d \mid c)$$

because $P(d)$ is the same for all classes.

2. Bag-of-Words Representation

A document d is represented as a vector of word counts:

$$d = (n_1, n_2, \ldots, n_{|V|})$$

- V = vocabulary (all unique words).
- n_i = number of times word i occurs in the document.

So the text is reduced to word counts (order ignored).

3. The Naive Assumption

We **assume words are conditionally independent given the class**.
That means:

$$P(d \mid c) = \prod_{i=1}^{|V|} P(w_i \mid c)^{n_i}$$

4. Estimating Parameters from Training Data

We need two things:

1. **Class prior**:

P(c) = \frac{\text{# documents in class } c}{\text{total # documents}}

2. **Word likelihood** (with Laplace smoothing to avoid zero probabilities):

$$P(w_i \mid c) = \frac{\text{count}(w_i, c) + 1}{\sum_{j=1}^{|V|} \text{count}(w_j, c) + |V|}$$

- $\text{count}(w_i, c)$ = number of times word w_i occurs in all docs of class c.
- Denominator = total word count in class c + vocabulary size.

5. Scoring a New Document

Given a new document d, compute for each class c:

$$\log P(c \mid d) \propto \log P(c) + \sum_{i=1}^{|V|} n_i \cdot \log P(w_i \mid c)$$

Why log?

- Multiplying many probabilities → underflow.
- Logs turn products into sums.

Finally, pick the class with the highest score:

$$\hat{c} = \arg\max_c \left[\log P(c) + \sum_i n_i \log P(w_i \mid c) \right]$$

6. Example Walkthrough

Suppose we have two classes:

- $c_1 =$ "China"
- $c_2 =$ "Japan"

Training documents:

- "Chinese Beijing Chinese" → China
- "Chinese Chinese Shanghai" → China
- "Chinese Macao" → China
- "Tokyo Japan Chinese" → Japan

Vocabulary = {Chinese, Beijing, Shanghai, Macao, Tokyo, Japan}

- Priors:
  - $P(\text{China}) = 3/4$
  - $P(\text{Japan}) = 1/4$
- Likelihoods: count words in each class, apply smoothing.
  For example:

$$P(\text{Chinese} \mid \text{China}) = \frac{5+1}{8+6} = \frac{6}{14}$$

(5 times "Chinese" in China docs, 8 total words, 6 vocab size).

Then for test doc: "Chinese Tokyo Japan"

$$\log P(\text{China} \mid d) \propto \log(3/4) + \log P(\text{Chinese}|\text{China}) + \log P(\text{Tokyo}|\text{China}) + \log P(\text{Japan}|\text{China})$$

$$\log P(\text{Japan} \mid d) \propto \log(1/4) + \log P(\text{Chinese}|\text{Japan}) + \log P(\text{Tokyo}|\text{Japan}) + \log P(\text{Japan}|\text{Japan})$$

Whichever is higher → predicted class.

✅ That's the full math behind Naive Bayes text classification on a bag-of-words.

Would you like me to actually **compute through that toy example step by step with numbers**, so you can see why "Chinese Tokyo Japan" ends up being classified as *Japan*?

yes

Thought

Great — let's walk through the toy example **with numbers** step-by-step so you can see exactly why `"Chinese Tokyo Japan"` is classified as **Japan**.

1) Training data (repeated from before)

1. `"Chinese Beijing Chinese"` → **China**
2. `"Chinese Chinese Shanghai"` → **China**
3. `"Chinese Macao"` → **China**
4. `"Tokyo Japan Chinese"` → **Japan**

Vocabulary V (unique words):
`{Chinese, Beijing, Shanghai, Macao, Tokyo, Japan}` so $|V| = 6$.

↓

2) Word counts per class

Thought

Great — let's walk through the toy example **with numbers** step-by-step so you can see exactly why `"Chinese Tokyo Japan"` is classified as **Japan**.

1) Training data (repeated from before)

1. `"Chinese Beijing Chinese"` → **China**
2. `"Chinese Chinese Shanghai"` → **China**
3. `"Chinese Macao"` → **China**
4. `"Tokyo Japan Chinese"` → **Japan**

Vocabulary V (unique words):
`{Chinese, Beijing, Shanghai, Macao, Tokyo, Japan}` so $|V| = 6$.

2) Word counts per class

China class (3 docs): total words = 3 + 3 + 2 = 8

- Chinese: 5
- Beijing: 1
- Shanghai: 1
- Macao: 1
- Tokyo: 0
- Japan: 0

Japan class (1 doc): total words = 3

- Chinese: 1
- Tokyo: 1
- Japan: 1
- Beijing, Shanghai, Macao: 0

3) Priors

$$P(\text{China}) = \frac{3}{4} = 0.75, \qquad P(\text{Japan}) = \frac{1}{4} = 0.25.$$

4) Likelihoods with Laplace (add-one) smoothing

Formula:

$$P(w \mid c) = \frac{\text{count}(w, c) + 1}{\text{total\_words\_in\_class} + |V|}$$

For **China**: denominator $= 8 + 6 = 14$

- $P(\text{Chinese} \mid \text{China}) = (5+1)/14 = 6/14 = 3/7 \approx 0.428571$
- $P(\text{Beijing} \mid \text{China}) = (1+1)/14 = 2/14 = 1/7 \approx 0.142857$
- $P(\text{Shanghai} \mid \text{China}) = 1/7$
- $P(\text{Macao} \mid \text{China}) = 1/7$
- $P(\text{Tokyo} \mid \text{China}) = (0+1)/14 = 1/14 \approx 0.0714286$
- $P(\text{Japan} \mid \text{China}) = 1/14$

For **Japan**: denominator $= 3 + 6 = 9$

- $P(\text{Chinese} \mid \text{Japan}) = (1+1)/9 = 2/9 \approx 0.222222$
- $P(\text{Tokyo} \mid \text{Japan}) = 2/9$
- $P(\text{Japan} \mid \text{Japan}) = 2/9$
- $P(\text{Beijing} \mid \text{Japan}) = (0+1)/9 = 1/9 \approx 0.111111$
- $P(\text{Shanghai} \mid \text{Japan}) = 1/9$
- $P(\text{Macao} \mid \text{Japan}) = 1/9$

5) Score the test document `d = ["Chinese","Tokyo","Japan"]`

We compute (log) posterior score for each class:

$$\log P(c \mid d) \propto \log P(c) + \sum_{w \in d} \log P(w \mid c).$$

Compute numeric logs:

China

- $\log P(\text{China}) = \log(0.75) \approx -0.28768207$
- $\log P(\text{Chinese} \mid \text{China}) = \log(3/7) \approx -0.84729786$
- $\log P(\text{Tokyo} \mid \text{China}) = \log(1/14) \approx -2.63905733$
- $\log P(\text{Japan} \mid \text{China}) = \log(1/14) \approx -2.63905733$

Sum:

$$\log \text{score}_{\text{China}} \approx -0.28768207 - 0.84729786 - 2.63905733 - 2.63905733 = -6.41309459$$

Japan

- $\log P(\text{Japan}) = \log(0.25) \approx -1.38629436$
- $\log P(\text{Chinese} \mid \text{Japan}) = \log(2/9) \approx -1.50407740$
- $\log P(\text{Tokyo} \mid \text{Japan}) = \log(2/9) \approx -1.50407740$
- $\log P(\text{Japan} \mid \text{Japan}) = \log(2/9) \approx -1.50407740$

S

Sum:

$$\log \text{score}_{\text{China}} \approx -0.28768207 - 0.84729786 - 2.63905733 - 2.63905733 = -6.41309459$$

Japan

- $\log P(\text{Japan}) = \log(0.25) \approx -1.38629436$
- $\log P(\text{Chinese} \mid \text{Japan}) = \log(2/9) \approx -1.50407740$
- $\log P(\text{Tokyo} \mid \text{Japan}) = \log(2/9) \approx -1.50407740$
- $\log P(\text{Japan} \mid \text{Japan}) = \log(2/9) \approx -1.50407740$

Sum:

$$\log \text{score}_{\text{Japan}} \approx -1.38629436 - 1.50407740 - 1.50407740 - 1.50407740 = -5.89852655$$

---

6) Compare & convert to probabilities

Exponentiate (for intuition):

- $\exp(-6.41309459) \approx 0.00163994$
- $\exp(-5.89852655) \approx 0.00274348$

Normalize to get posterior probabilities:

$$P(\text{China} \mid d) \approx \frac{0.00163994}{0.00163994 + 0.00274348} \approx 0.3741$$

$$P(\text{Japan} \mid d) \approx 0.6259$$

So the classifier predicts **Japan** (higher posterior probability ≈ 62.6%).

---

7) Intuition why Japan wins here

- The test doc contains `Tokyo` and `Japan` which are **much more likely** under the *Japan* class than under *China*.
- Although `Chinese` strongly favors the China class, the combined low likelihood of `Tokyo` and `Japan` under **China** (because they never appeared in China docs) drags the China score down more than the prior advantage (China has higher prior 0.75 vs 0.25).

If you want, I can:

- run this same calculation in code and show exact printed steps, or
- show how the result would change with different smoothing (e.g., no smoothing), or
- show how to produce class probabilities from the from-scratch classifier implementation.

run it in code

Here's the output of the step-by-step Naive Bayes classification in code:

- **Log-scores**
  - China: -6.41
  - Japan: -5.90
- **Posterior probabilities**
  - $P(\text{China} \mid d) \approx 0.374$
  - $P(\text{Japan} \mid d) \approx 0.626$

✅ The classifier predicts **Japan** for `"Chinese Tokyo Japan"`.

Do you want me to extend the code so you can train it on *any* dataset of text strings (instead of manually tokenized lists of words)? ‹/›