perplexity

Multinomial naive bayes from scratch

To implement Multinomial Naive Bayes (MNB) from scratch for text classification, follow these steps. The algorithm is ideal for document classification (e.g., spam detection) and uses word frequencies to predict categories.

Algorithm Overview

1. **Prior Probability**: Probability of each class in the training data.
2. **Likelihood**: Probability of a word given a class (with smoothing).
3. **Posterior Probability**: Class probability given observed words (using Bayes' theorem).

Step-by-Step Implementation

```
import numpy as np

class MultinomialNaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Smoothing parameter
        self.log_priors = None
        self.log_likelihoods = None
        self.classes = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        # Calculate class priors (log scale)
        self.log_priors = np.zeros(n_classes)
        for idx, c in enumerate(self.classes):
            n_class = np.sum(y == c)
            self.log_priors[idx] = np.log(n_class / n_samples)

        # Compute word counts per class and total words per class
        total_words_per_class = np.zeros(n_classes)
        class_word_counts = np.zeros((n_classes, n_features))

        for idx, c in enumerate(self.classes):
            X_c = X[y == c]
            class_word_counts[idx] = np.sum(X_c, axis=0)
            total_words_per_class[idx] = np.sum(X_c)

        # Calculate log likelihoods with Laplace smoothing
        self.log_likelihoods = np.zeros((n_classes, n_features))
        for idx in range(n_classes):
            denominator = total_words_per_class[idx] + self.alpha * n_features
```

```
            self.log_likelihoods[idx] = np.log(
                (class_word_counts[idx] + self.alpha) / denominator
            )

    def predict(self, X):
        # Compute log posteriors: log_prior + (X dot log_likelihood)
        scores = np.dot(X, self.log_likelihoods.T) + self.log_priors
        return self.classes[np.argmax(scores, axis=1)]
```

Key Components Explained

1. **Smoothing (`alpha`)**
   Prevents zero probabilities for unseen words. Laplace smoothing adds `alpha` to each word count[1] [2].
2. **Log Probabilities**
   Avoids floating-point underflow by using logarithms instead of multiplying small probabilities[3] [2].
3. **Efficiency**
   Vectorized operations (e.g., `np.dot`) accelerate prediction[4] [5].

Example Usage

```
# Sample data: word counts for 3 documents
X_train = np.array([
    [2, 1, 0],  # "apple apple orange"
    [1, 0, 1],  # "apple banana"
    [0, 0, 2]   # "banana banana"
])
y_train = np.array(["fruit", "fruit", "not_fruit"])

# Test document: "apple banana"
X_test = np.array([[1, 0, 1]])

# Train and predict
nb = MultinomialNaiveBayes(alpha=1.0)
nb.fit(X_train, y_train)
print(nb.predict(X_test))  # Output: ["fruit"]
```

Mathematical Foundation

- **Prior**: $ P(c) = \frac{count(c)}{total documents} $
- **Likelihood**: $ P(w_i \mid c) = \frac{count(w_i, c) + \alpha}{total words in c + \alpha \times |vocabulary|} $
- **Log Posterior**: $ \log P(c \mid \mathbf{x}) = \log P(c) + \sum_{i} x_i \cdot \log P(w_i \mid c) $

Practical Considerations

- **Text Preprocessing**:
  Convert raw text to word-count vectors using bag-of-words or TF-IDF [3] [2].
- **Handling Unseen Words**:
  Ignore words absent in training data during prediction.
- **Real-World Use**:
  For large datasets, optimize with sparse matrices and parallel processing [3] [4].

This implementation mirrors scikit-learn's `MultinomialNB` [1] [5] while providing transparency for educational purposes. For production, use optimized libraries like scikit-learn with hyperparameter tuning [6] [1].

⁂

Use this dataset

data = [
['Sunny','Hot','High','Weak','No'],
['Sunny','Hot','High','Strong','No'],
['Overcast','Hot','High','Weak','Yes'],
['Rain','Mild','High','Weak','Yes'],
['Rain','Cool','Normal','Weak','Yes'],
['Rain','Cool','Normal','Strong','No'],
['Overcast','Cool','Normal','Strong','Yes'],
['Sunny','Mild','High','Weak','No'],
['Sunny','Cool','Normal','Weak','Yes'],
['Rain','Mild','Normal','Weak','Yes'],
['Sunny','Mild','Normal','Strong','Yes'],
['Overcast','Mild','High','Strong','Yes'],
['Overcast','Hot','Normal','Weak','Yes'],
['Rain','Mild','High','Strong','No']
]
columns = ['Outlook','Temperature','Humidity','Windy','Play']
df = pd.DataFrame(data, columns=columns)

Here is how the Multinomial Naive Bayes classifier was applied from scratch to your weather dataset:

Data Preparation

- The categorical features (`Outlook`, `Temperature`, `Humidity`, `Windy`) and the target (`Play`) were label-encoded into integers, making them suitable for the numerical computations required by the classifier.
- Example of encoded features:

```
   Outlook  Temperature  Humidity  Windy
0        2            1         0      1
1        2            1         0      0
2        0            1         0      1
...
```

- The target (`Play`) was encoded as: `No` → 0, `Yes` → 1.

Model Training

- A Multinomial Naive Bayes classifier was implemented from scratch, including Laplace smoothing (`alpha=1.0`).
- The model was trained using the encoded features and target values.

Prediction

- The model was used to predict the target on the training data.
- The predicted labels were decoded back to their original string values (`Yes` or `No`).

Results

The predicted labels for the training data are:

```
['No', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
```

Summary Table

Actual Play	Predicted Play
No	No
No	No
Yes	Yes
Yes	Yes
Yes	Yes
No	Yes
Yes	Yes
No	No
Yes	Yes
Yes	Yes
Yes	Yes
Yes	Yes
Yes	Yes

Actual Play	Predicted Play
No	Yes

Notes

- The model predicts most labels correctly, but does misclassify some "No" cases as "Yes" (e.g., row 5 and row 14 in the original dataset).
- This implementation demonstrates how Multinomial Naive Bayes can be used for categorical data by encoding categories as integers and applying the algorithm directly.

If you want to test the model on new samples or see the full code, let me know!

⁂

without label encoding

To implement Multinomial Naive Bayes **without label encoding** for the provided weather dataset, we'll use direct categorical feature handling. The approach calculates probabilities using raw category strings, avoiding numerical conversion. Below is a complete implementation and analysis:

Solution Overview

- **Algorithm**: Categorical Naive Bayes (variant of Multinomial NB for categorical features)
- **Smoothing**: Laplace smoothing (`alpha=1.0`) to handle unseen categories
- **Features**: `Outlook`, `Temperature`, `Humidity`, `Windy` (all categorical)
- **Target**: `Play` (binary: `Yes`/`No`)

Implementation Code

```
import numpy as np
import pandas as pd

class CategoricalNaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Smoothing parameter
        self.classes = None
        self.priors = {}
        self.likelihoods = {}  # Format: {feature: {class: {category: prob}}}

    def fit(self, X, y):
        self.classes = np.unique(y)
        n_samples = X.shape[0]

        # Calculate class priors
        for c in self.classes:
            self.priors[c] = (y == c).sum() / n_samples
```

```
        # Calculate feature likelihoods per class
        for feature in X.columns:
            self.likelihoods[feature] = {}
            all_categories = np.unique(X[feature])

            for c in self.classes:
                subset = X[y == c]
                total_in_class = len(subset)
                self.likelihoods[feature][c] = {}

                for cat in all_categories:
                    count = (subset[feature] == cat).sum()
                    prob = (count + self.alpha) / (total_in_class + self.alpha * len(all_
                    self.likelihoods[feature][c][cat] = prob

    def predict(self, X):
        preds = []
        for _, row in X.iterrows():
            max_log_prob = -np.inf
            best_class = None

            for c in self.classes:
                log_prob = np.log(self.priors[c])
                valid = True

                for feature in X.columns:
                    cat = row[feature]
                    if cat in self.likelihoods[feature][c]:
                        log_prob += np.log(self.likelihoods[feature][c][cat])
                    else:
                        # Skip if category was unseen during training
                        valid = False
                        break

                if valid and log_prob > max_log_prob:
                    max_log_prob = log_prob
                    best_class = c

            preds.append(best_class if best_class else self.classes[0])
        return preds

# Create dataset
data = [
    ['Sunny','Hot','High','Weak','No'],
    ['Sunny','Hot','High','Strong','No'],
    ['Overcast','Hot','High','Weak','Yes'],
    ['Rain','Mild','High','Weak','Yes'],
    ['Rain','Cool','Normal','Weak','Yes'],
    ['Rain','Cool','Normal','Strong','No'],
    ['Overcast','Cool','Normal','Strong','Yes'],
    ['Sunny','Mild','High','Weak','No'],
    ['Sunny','Cool','Normal','Weak','Yes'],
    ['Rain','Mild','Normal','Weak','Yes'],
    ['Sunny','Mild','Normal','Strong','Yes'],
    ['Overcast','Mild','High','Strong','Yes'],
    ['Overcast','Hot','Normal','Weak','Yes'],
```

```
    ['Rain','Mild','High','Strong','No']
]
columns = ['Outlook','Temperature','Humidity','Windy','Play']
df = pd.DataFrame(data, columns=columns)

# Train and predict
X = df.drop('Play', axis=1)
y = df['Play']
model = CategoricalNaiveBayes(alpha=1.0)
model.fit(X, y)
predictions = model.predict(X)
```

Key Steps Explained

1. **Prior Calculation**:
   Computes class probabilities:
   - `P(Play="Yes") = 9/14 ≈ 0.643`
   - `P(Play="No") = 5/14 ≈ 0.357`
2. **Likelihood Calculation (with Smoothing)**:
   For each feature and class, computes conditional probabilities.
   Example:
   - `P(Outlook="Sunny" | Play="No")`:
     - Count(`Sunny` in `No`) = 3
     - Total `No` samples = 5
     - Categories in `Outlook` = 3 (`Sunny`, `Overcast`, `Rain`)
     - **Probability** = `(3 + 1) / (5 + 1*3) = 4/8 = 0.5`
3. **Prediction**:
   For a new instance, computes log-probability for each class:
   ```
   log(P(c)) + Σ log(P(x_i | c))
   ```
   and selects the class with the highest value.

Training Predictions

Actual	Predicted	
No	No	
No	No	
Yes	Yes	
Yes	Yes	
Yes	Yes	
No	**Yes**	# Misclassification

Actual	Predicted	
Yes	Yes	
No	No	
Yes	Yes	
Yes	Yes	
Yes	Yes	
Yes	Yes	
Yes	Yes	
No	No	

Accuracy: 13/14 ≈ 92.86% (1 error: Row 5 misclassified as `Yes`)

Why This Works

- **Direct Categorical Handling**: Uses raw category strings without numerical conversion.
- **Smoothing**: Prevents zero probabilities for categories unseen in a class.
- **Efficiency**: Computes probabilities during training for fast prediction.

When to Use

- Ideal for **categorical datasets** with string features.
- Avoids information loss from label encoding.
- Use Multinomial NB with count vectors for text data (e.g., word frequencies).

For continuous features, consider Gaussian Naive Bayes instead.

separate them into different functions, not encapsulating in only one class

Here is a modular implementation of Multinomial (Categorical) Naive Bayes for your weather dataset, **without label encoding** and with each major step in a separate function—not encapsulated in a single class.

Implementation

```
import pandas as pd
import numpy as np

def calculate_priors(y):
    """Calculate prior probabilities for each class."""
    priors = {}
    n_samples = len(y)
    for c in y.unique():
```

```
        priors[c] = (y == c).sum() / n_samples
    return priors

def calculate_likelihoods(X, y, alpha=1.0):
    """Calculate likelihoods P(feature_value | class) with Laplace smoothing."""
    likelihoods = {}
    for feature in X.columns:
        likelihoods[feature] = {}
        categories = X[feature].unique()
        for c in y.unique():
            subset = X[y == c]
            total = len(subset)
            likelihoods[feature][c] = {}
            for cat in categories:
                count = (subset[feature] == cat).sum()
                prob = (count + alpha) / (total + alpha * len(categories))
                likelihoods[feature][c][cat] = prob
    return likelihoods

def predict_single(x, priors, likelihoods, X_columns, classes):
    """Predict the class for a single sample x (as a Series)."""
    log_probs = {}
    for c in classes:
        log_prob = np.log(priors[c])
        for feature in X_columns:
            cat = x[feature]
            # If unseen category, apply minimal smoothing
            prob = likelihoods[feature][c].get(cat, 1e-9)
            log_prob += np.log(prob)
        log_probs[c] = log_prob
    return max(log_probs, key=log_probs.get)

def predict(X, priors, likelihoods, classes):
    """Predict classes for all samples in X."""
    return [predict_single(row, priors, likelihoods, X.columns, classes) for _, row in X.

# --- Example usage with your dataset ---

data = [
    ['Sunny','Hot','High','Weak','No'],
    ['Sunny','Hot','High','Strong','No'],
    ['Overcast','Hot','High','Weak','Yes'],
    ['Rain','Mild','High','Weak','Yes'],
    ['Rain','Cool','Normal','Weak','Yes'],
    ['Rain','Cool','Normal','Strong','No'],
    ['Overcast','Cool','Normal','Strong','Yes'],
    ['Sunny','Mild','High','Weak','No'],
    ['Sunny','Cool','Normal','Weak','Yes'],
    ['Rain','Mild','Normal','Weak','Yes'],
    ['Sunny','Mild','Normal','Strong','Yes'],
    ['Overcast','Mild','High','Strong','Yes'],
    ['Overcast','Hot','Normal','Weak','Yes'],
    ['Rain','Mild','High','Strong','No']
]
columns = ['Outlook','Temperature','Humidity','Windy','Play']
df = pd.DataFrame(data, columns=columns)
```

```
X = df.drop('Play', axis=1)
y = df['Play']

priors = calculate_priors(y)
likelihoods = calculate_likelihoods(X, y, alpha=1.0)
classes = y.unique()
predictions = predict(X, priors, likelihoods, classes)

print("Actual:    ", list(y))
print("Predicted: ", predictions)
```

How the Functions Work

- **calculate_priors(y)**
  Returns a dictionary of prior probabilities for each class in `y`.
- **calculate_likelihoods(X, y, alpha)**
  Returns nested dictionaries of conditional probabilities for each feature value given each class, with Laplace smoothing.
- **predict_single(x, priors, likelihoods, X_columns, classes)**
  Computes the log-probability for each class for a single sample and returns the most probable class.
- **predict(X, priors, likelihoods, classes)**
  Applies `predict_single` to each row in `X` to generate predictions for the whole dataset.

Output Example

```
Actual:     ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Ye
Predicted:  ['No', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Y
```

This modular structure makes each step explicit and easy to test or modify individually, and it works directly with categorical data[7].

⁂

create calculate_posterior() function

Here is a function, `calculate_posterior()`, that computes the (log) posterior probability for each class given a single sample, using the priors and likelihoods computed earlier. This function is modular and works directly with categorical features, as in your previous code.

```
import numpy as np

def calculate_posterior(x, priors, likelihoods, X_columns, classes):
    """
    Calculate the log-posterior probability for each class given a sample x.

    Parameters:
```

```
        x          : pandas Series representing a single sample (row of X)
        priors     : dict of class priors {class: P(class)}
        likelihoods: nested dict of conditional probabilities
                     {feature: {class: {category: P(feature=category|class)}}}
        X_columns : list of feature names
        classes    : list or array of possible class labels

    Returns:
        log_posteriors: dict {class: log-posterior}
    """
    log_posteriors = {}
    for c in classes:
        log_prob = np.log(priors[c])
        for feature in X_columns:
            cat = x[feature]
            # Use a small value if the category was unseen during training
            prob = likelihoods[feature][c].get(cat, 1e-9)
            log_prob += np.log(prob)
        log_posteriors[c] = log_prob
    return log_posteriors
```

Usage Example:

```
# For a single row/sample:
row = X.iloc[^5_0]
log_posteriors = calculate_posterior(row, priors, likelihoods, X.columns, classes)
print(log_posteriors)
# To get the predicted class:
predicted_class = max(log_posteriors, key=log_posteriors.get)
```

Explanation:

- For each class, the function sums the log of the prior and the log likelihoods for each feature value observed in the sample.
- If a feature value was not seen in training for a class, it uses a very small probability (`1e-9`) to avoid log(0).
- Returns a dictionary of log-posterior probabilities for all classes.

This function is directly compatible with your modular Naive Bayes code and can be used within your prediction loop or for analysis of individual samples[8].

⁂

What is the equation for posteriors ?

$$P(C|X) \propto P(C) \prod_{i=1}^{n} P(x_i|C)$$

To prevent underflow, we use log probabilities:

$$\log P(C|X) = \log P(C) + \sum_{i=1}^{n} x_i \cdot \log P(x_i|C)$$

Is this correct?

The equation for the log-posterior in Naive Bayes depends on the feature representation:

1. For Categorical Features (like your weather dataset):

$$\log P(C|X) = \log P(C) + \sum_{i=1}^{n} \log P(x_i|C)$$

- **No x_i multiplier**: Each feature x_i is a categorical value (e.g., "Sunny"), not a count.
- **Implementation**:

```
log_posterior = np.log(prior) + np.sum([np.log(P(x_i | C)) for each feature])
```

2. For Multinomial Features (e.g., word counts in text):

$$\log P(C|X) = \log P(C) + \sum_{i=1}^{n} x_i \cdot \log P(x_i|C)$$

- **Includes x_i**: x_i is the count of feature i (e.g., word frequency).
- **Applies to**: Bag-of-words models like spam detection.

Key Differences

Feature Type	Likelihood Term	Example Use Case
Categorical	$\log P(x_i \|\| C)$	Weather dataset (your example)
Multinomial	$x_i \cdot \log P(x_i \|\| C)$	Document classification

Why Your Second Equation is Partially Correct

- ✅ Correct for **Multinomial Naive Bayes** (count-based features).
- ❌ Incorrect for **Categorical Naive Bayes** (non-count features).

Practical Example

For your weather dataset row `["Sunny", "Hot", "High", "Weak"]` (class "No"):

1. **Priors**: $P(\text{"No"}) = \frac{5}{14}$
2. **Likelihoods**:
   - $P(\text{"Sunny"} \mid \text{"No"}) = \frac{3+1}{5+3 \cdot 1} = \frac{4}{8}$
   - $P(\text{"Hot"} \mid \text{"No"}) = \frac{2+1}{5+3 \cdot 1} = \frac{3}{8}$

- ... (similar for other features)

3. **Log-Posterior**:

$$\log P(\text{"No"}|X) = \log \frac{5}{14} + \log \frac{4}{8} + \log \frac{3}{8} + \log \frac{1}{8} + \log \frac{4}{8}$$

No x_i multipliers – categorical features use direct probabilities.

Summary

- Use $\sum \log P(x_i|C)$ for categorical data (your weather dataset).
- Use $\sum x_i \cdot \log P(x_i|C)$ for count-based data (e.g., word frequencies).
- Both approaches use log probabilities to prevent underflow.

1. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html
2. https://towardsdatascience.com/multinomial-naive-bayes-for-documents-classification-and-natural-language-processing-nlp-e08cc848ce6/
3. https://github.com/akash18tripathi/Multinomial-Naive-Bayes-from-Scratch
4. https://jonndata.github.io/2020-07-30-Naive-Bayes-From-Scratch/
5. https://stackoverflow.com/questions/60969884/multinomial-naive-bayes-for-python-from-scratch
6. https://visualstudiomagazine.com/articles/2023/04/17/multinomial-naive-bayes.aspx
7. programming.machine_learning
8. programming.machine_learning