Naive Bayes: Algorithm and Implementation

Repository `naive_bayes`
Author: Mateusz Oleksy

December 5, 2025

Overview

Introduction

Naive Bayes is a simple probabilistic classifier that applies Bayes' rule under the assumption that features are independent given the class.
Despite this assumption, it often performs well in tasks such as text classification.

Bayes Classifier

Let $X = (X_1, \ldots, X_d)$ be a feature vector and $Y \in \{0, 1\}$.

$$\hat{y} = \arg\max_y P(Y = y) \prod_{j=1}^{d} P(X_j \mid Y = y)$$

We typically use log-probabilities:

$$\log P(Y = y) + \sum_{j=1}^{d} \log P(X_j \mid Y = y)$$

Binary Features and Smoothing

For binary features:

$$P(X_j = 1 \mid Y = y)$$

Laplace smoothing:

$$\hat{P}(X_j = 1 \mid Y = 1) = \frac{N_{j,1} + \alpha}{N_1 + 2\alpha}$$

with $\alpha = 1$ typically.

Feature Normalization

The implementation converts features to binary values.

`normalize_feature_by_max_half`:

- Computes global maximum over all features
- Maps value to 1 if it is $\geq \frac{1}{2}$ of the maximum
- Otherwise maps to 0

Implementation Overview

Key functions in `naive_bayes.py`:

- **is_number**
- **normalize_feature_by_max_half**
- **read_data**
- **train_algorithm**
- **predict**

Training Algorithm

```python
def train_algorithm(X, Y, smoothing=1.0):
    n_samples = len(Y)
    n_features = len(X[0])
    num_pos = sum(Y)
    num_neg = n_samples - num_pos
    prior_pos = num_pos / n_samples
    prior_neg = num_neg / n_samples
```

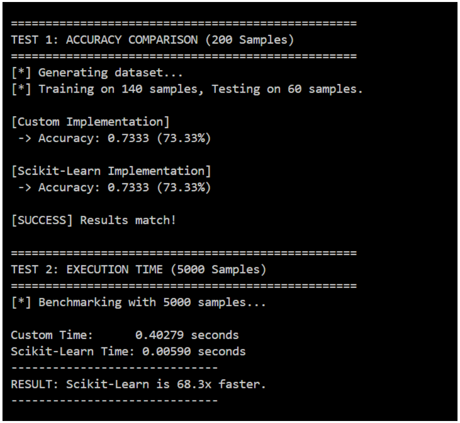
Training Algorithm (cont.)

```
    model = {}
    for j in range(n_features):
        count_pos = 0
        count_neg = 0
        for i in range(n_samples):
            val = X[i][j]
            if val == 1:
                if Y[i] == 1:
                    count_pos += 1
                else:
                    count_neg += 1
        pos_prob = (count_pos + smoothing) / (num_pos
            + 2*smoothing)
        neg_prob = (count_neg + smoothing) / (num_neg
            + 2*smoothing)
        model[j] = {"positive": pos_prob, "negative":
            neg_prob}
    return model, prior_pos, prior_neg
```

Prediction

```
def predict(model, prior_pos, prior_neg, x_test):
    pos_log = math.log(prior_pos)
    neg_log = math.log(prior_neg)
    for j, val in enumerate(x_test):
        feat = model.get(j)
        p_pos = max(min(feat["positive"], 1-1e-12), 1e
            -12)
        p_neg = max(min(feat["negative"], 1-1e-12), 1e
            -12)
        if val == 1:
            pos_log += math.log(p_pos)
            neg_log += math.log(p_neg)
        else:
            pos_log += math.log(1 - p_pos)
            neg_log += math.log(1 - p_neg)
    return 1 if pos_log > neg_log else 0
```

Comparison with Scikit-learn



```
==================================================
TEST 1: ACCURACY COMPARISON (200 Samples)
==================================================
[*] Generating dataset...
[*] Training on 140 samples, Testing on 60 samples.

[Custom Implementation]
 -> Accuracy: 0.7333 (73.33%)

[Scikit-Learn Implementation]
 -> Accuracy: 0.7333 (73.33%)

[SUCCESS] Results match!

==================================================
TEST 2: EXECUTION TIME (5000 Samples)
==================================================
[*] Benchmarking with 5000 samples...

Custom Time:      0.40279 seconds
Scikit-Learn Time: 0.00590 seconds
------------------------------
RESULT: Scikit-Learn is 68.3x faster.
------------------------------
```

Figure: Comparison with Scikit-learn

Usage Example

1. Load data: `X, Y, features = read_data()`
2. Train model: `model, prior_pos, prior_neg = train_algorithm(X,Y)`
3. Predict: `pred = predict(model, prior_pos, prior_neg, X[0])`

Thank you!