Naive Bayes – Algorithm and Implementation

Repository `naive_bayes` Author: Mateusz Oleksy

December 5, 2025

Abstract

This document explains the fundamentals of the Naive Bayes classifier and details the implementation found in `src/naive_bayes/naive_bayes.py`.

It includes the algorithm description, mathematical formulas, a note about feature normalization, smoothing, and selected code excerpts.

1 Introduction

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

2 Theory

Let $X = (X_1, X_2, \ldots, X_d)$ be a feature vector and $Y \in \{0, 1\}$ be the target variable (binary classification). The Bayes classifier selects the class that maximizes the posterior probability:

$$\hat{y} = \arg\max_{y \in \{0,1\}} P(Y = y \mid X) = \arg\max_{y} P(Y = y) \prod_{j=1}^{d} P(X_j \mid Y = y).$$

In practice we use logarithms to avoid numerical underflow:

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

For binary features $X_j \in \{0, 1\}$ the model estimates:

$$P(X_j = 1 \mid Y = 1), \quad P(X_j = 1 \mid Y = 0).$$

To prevent zero probabilities we apply Laplace smoothing:

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

where $N_{j,1}$ is the number of samples of class 1 with $X_j = 1$, N_1 is the number of samples in class 1, and α is the smoothing parameter (commonly $\alpha = 1$).

3 Feature normalization

The implementation converts features to binary before training.

The function `normalize_feature_by_max_half` computes the global maximum across all features and maps each value to 1 if it is greater than or equal to half of that maximum, otherwise to 0.

4 Implementation

The file `src/naive_bayes/naive_bayes.py` contains the following key functions:

- **is_number**: checks whether a token can be parsed as a number;
- **normalize_feature_by_max_half**: converts features to 0/1;
- **read_data**: loads feature matrix and labels;
- **train_algorithm**: estimates conditional probabilities and class priors;
- **predict**: classifies a feature vector using log-probabilities.

Selected excerpts:

```
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

    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) if num_pos > 0 else 0.5
        neg_prob = (count_neg + smoothing) / (num_neg + 2 *
           smoothing) if num_neg > 0 else 0.5
        model[j] = {"positive": pos_prob, "negative": neg_prob}
    return model, prior_pos, prior_neg
```

```
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
```

5 Usage example

Typical workflow in Python:

1. Load data: `X, Y, feature_names = 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])`