

# **DATA2060 Final Project**

## **Gaussian Naive Bayes for classification**

**Group Mambo: Heyang Liu, Luo Jin, Tong Su**

**01** Math(Tong Su)

**02** Model(Luo Jin)

**03** Reproduced(Heyang Liu)

**Goal:** classify input  $x \in \mathbb{R}^d$  into the class  $y \in \{0, \dots, K - 1\}$

**Naive Bayes assumption:** features are conditionally independent on the given class

$$p(\mathbf{x}|y) = \prod_{i=1}^d p(x_i|y)$$

**Gaussian assumption:** each feature is Gaussian within respective class

$$x_i|y = k \sim \mathcal{N}(\mu_{k,i}, \sigma_{k,i}^2)$$

Joint Model:

$$p(\mathbf{x}, y = k) = p(y = k)p(\mathbf{x}|y = k)$$

**Input:**  $\mathbf{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$ ,  $y$  of  $k$  class

For each class  $k$ , the class conditional likelihood:

$$p(\mathbf{x}|y = k) = \prod_{i=1}^d \mathcal{N}(x_i | \mu_{k,i}, \sigma_{k,i}^2)$$

where

$$\mathcal{N}(x_i | \mu_{k,i}, \sigma_{k,i}^2) = \frac{1}{\sqrt{2\pi\sigma_{k,i}^2}} \exp\left(-\frac{(x_i - \mu_{k,i})^2}{2\sigma_{k,i}^2}\right)$$

**Bayes rule** gives the unnormalized posterior:

$$p(y = k|\mathbf{x}) \propto p(y = k)p(\mathbf{x}|y = k)$$

**Bayes rule** gives the unnormalized posterior:

$$p(y = k|\mathbf{x}) \propto p(y = k)p(\mathbf{x}|y = k)$$

in code, transfer into **log space**:

$$\log p(y = k|\mathbf{x}) = \log p(y = k) + \sum_{i=1}^d \log p(x_i|y = k) + \text{const}$$

Prediction (**Maximum A Posteriori MAP**):

$$h_{Bayes}(\mathbf{x}) = \arg \max_k [\log p(y = k) + \sum_{i=1}^d \log p(x_i|y = k)]$$

If priors are equal, this reduces to **maximizing the likelihood (MLE)**

**Training data:**  $D = \{x, y\}^N$

**Parameter (class):**  $\theta = \{ p(y = k), \mu_{k,i}, \sigma_{k,i}^2 \}$

Find the parameters by MLE:

$$\hat{\theta} = \arg \max_{\theta} L(\theta|D) = \arg \max_{\theta} p(D|\theta)$$

**In log space:**

$$\ell(\theta|D) = \log p(D|\theta) = \sum_{j=1}^N \left[ \log p(y_j) + \sum_{i=1}^d \log(p(x_i|y_j)) \right]$$

set into minimum by:

$$\mathcal{L}(\theta) = -\ell(\theta|D)$$

Unlike models like logistic regression or neural network,  
GNB has analytic MLE solutions

each class  $k$ ,  $D_k = \{x, y = k\}$ ,  $N_k = |D_k|$

class **prior** (MLE):  $\hat{p}(y = k) = \frac{N_k}{N}$

initial **mean**:  $\hat{\mu}_{k,i} = \frac{1}{N_k} \sum_{x \in D_k} x_i$

initial **variance**:  $\sigma_{k,i}^2 = \frac{1}{N_k} \sum_{x \in D_k} (x_i - \hat{\mu}_{k,i})^2$

**No iterative optimizer!**

Unlike models like logistic regression or neural network,  
GNB has analytic MLE solutions

each class  $k$ ,  $D_k = \{x, y = k\}$ ,  $N_k = |D_k|$

class **prior** (MLE):  $\hat{p}(y = k) = \frac{N_k}{N}$

initial **mean**:  $\hat{\mu}_{k,i} = \frac{1}{N_k} \sum_{x \in D_k} x_i$

initial **variance**:  $\sigma_{k,i}^2 = \frac{1}{N_k} \sum_{x \in D_k} (x_i - \hat{\mu}_{k,i})^2$

**No iterative optimizer!**

## GaussianNaiveBayes Class

### Key Methods:

— <code>__init__()</code>	→ Initialize parameters
— <code>train(X, y)</code>	→ Fit model using MLE
— <code>_joint_log_likelihood(X)</code>	→ Compute log probabilities
— <code>predict(X)</code>	→ Return predicted labels
— <code>accuracy(X, y)</code>	→ Evaluate performance

**Core idea:** Store class-wise statistics ( $\mu$ ,  $\sigma^2$ , priors), then compute Bayes' rule in log-space

## INPUT:

$X_{\text{train}}$ : training features ( $n_{\text{examples}} \times n_{\text{features}}$ )

$y_{\text{train}}$ : training labels ( $n_{\text{examples}}$ )

### 1. Identify basic information

classes: `unique( $y_{\text{train}}$ )`

$n_{\text{classes}}$ : `length(classes)`

$n_{\text{features}}$ : number of features

### 2. Initialize model parameters: $\mu$ , $\sigma^2$ , priors

### 3. FOR $\text{idx} = 0$ to $(\text{n\_classes} - 1)$ :

$c \leftarrow \text{class}[\text{idx}]$

a) Extract samples of class  $c$

$X\_c \leftarrow X\_train[\text{where } y\_train == c]$

b) Compute MLE estimates

$\mu[\text{idx}, :] \leftarrow \text{mean}(X\_c) \text{ over samples}$

$\sigma^2[\text{idx}, :] \leftarrow \text{variance}(X\_c) \text{ over samples}$

c) Compute class prior probability

$\text{priors}[\text{idx}] \leftarrow \text{count}(\text{class } c) / \text{total samples}$

### 4. Apply variance smoothing

$\varepsilon \leftarrow \text{self.var\_smoothing} \times \max(\text{variance}(X\_train))$

$\sigma^2 \leftarrow \sigma^2 + \varepsilon$

# Joint Log-Likelihood Computation

## INPUT:

X\_input: input features ( $n\_examples \times n\_features$ )

**1. Initialize joint\_log\_likelihood as zero matrix**

**2. FOR each class idx = 0 to n\_classes -1:**

a) Compute log\_prior from class probability

b) Compute Gaussian log-likelihood

log\_likelihood =  $-0.5 \times \text{sum}(\log(2\pi \times \sigma^2[c, :]))$

FOR each sample i:

log\_likelihood[i] -=  $\text{sum}((X[i] - \mu[c])^2 / (2\sigma^2[c]))$

$$\log p(x \mid \mu, \sigma^2) = -\frac{1}{2} \log(2\pi\sigma^2) - \frac{(x - \mu)^2}{2\sigma^2}$$

c) Combine: joint\_log\_likelihood = log\_prior + log\_likelihood

**Return joint\_log\_likelihood**

## INPUT:

$X_{\text{input}}$ :  $n_{\text{samples}} \times n_{\text{features}}$

1. **Compute joint log-likelihoods, then find class with highest log-likelihood for each sample**
2. **Map indices back to actual class labels**

**Return**  $y_{\text{pred}}$ :  $n_{\text{samples}}$  array of predicted class labels

**INPUT:**

$X_{\text{test}}, y_{\text{test}}$

1. **Get predictions for  $X_{\text{test}}$**
2. **Compute proportion of correct predictions**

**Return accuracy**

## Dataset description:

The dataset used is the *Wine Quality* dataset from the UCI Machine Learning Repository. The target variable is *quality*, representing the quality score of each wine sample. The dataset contains 11 continuous features, as well as one categorical attribute, *color*, which indicates whether a sample is red or white wine.

For consistency, we split the dataset into two separate subsets—one for red wine and one for white wine—based on the color attribute. This separation prevents potential distributional shifts or confounding effects, while also ensures that all features within each subset are continuous.

There are 1599 samples in the red wine dataset, and 4898 samples in the white wine dataset, which provides a sufficient number of samples to maintain distributions inside training and testing sets.

1. Load the dataset.
2. Split the dataset into training and test sets using stratified sampling to preserve the class distribution.
3. Train both models — sklearn's GaussianNB and our implemented GaussianNaiveBayes — using the same `var_smoothing` parameter.
4. To verify the correctness of the smoothing step in our implementation, this procedure is repeated for multiple `var_smoothing` values.
5. Compare model components, ensuring our implementation matches sklearn in estimated class means, estimated variances (after smoothing), class priors, joint log-likelihood probabilities, final predictions, and test accuracies.
6. After all checks pass, print the test accuracy for both models using the default `var_smoothing = 1e-9`.

We use the `np.allclose` method to compare numerical outputs. The absolute tolerance (`atol`) is set to  $1\text{e-}8$  and the relative tolerance (`rtol`) is set to  $1\text{e-}9$ , which together ensure a strict and consistent numerical check. These tolerance levels are sufficiently small to detect meaningful differences while accounting for possible inevitable floating-point rounding errors.

The `var_smoothing` values checked are: 0,  $1\text{e-}8$ ,  $1\text{e-}9$ ,  $1\text{e-}10$ , and  $1\text{e-}12$ . Among these,  $1\text{e-}8$ ,  $1\text{e-}9$ , and  $1\text{e-}10$  are commonly used in actual implementations, while 0 and  $1\text{e-}12$  represent extreme boundary cases for testing numerical stability.

```
=====
Testing GaussianNB implementation on the Red Wine dataset
=====
```

```
Initial checks passed for var_smoothing = 1e-9
Passed checks for var_smoothing = 0
Passed checks for var_smoothing = 1e-08
Passed checks for var_smoothing = 1e-09
Passed checks for var_smoothing = 1e-10
Passed checks for var_smoothing = 1e-12
```

```
Accuracy comparison for var_smoothing = 1e-9:
  sklearn GaussianNB accuracy      : 0.5687
  implemented GaussianNB accuracy  : 0.5687
```

```
All checks passed for the dataset: Red Wine
=====
```

```
=====
Testing GaussianNB implementation on the White Wine dataset
=====
```

```
Initial checks passed for var_smoothing = 1e-9
Passed checks for var_smoothing = 0
Passed checks for var_smoothing = 1e-08
Passed checks for var_smoothing = 1e-09
Passed checks for var_smoothing = 1e-10
Passed checks for var_smoothing = 1e-12
```

```
Accuracy comparison for var_smoothing = 1e-9:
  sklearn GaussianNB accuracy      : 0.4585
  implemented GaussianNB accuracy  : 0.4585
```

```
All checks passed for the dataset: White Wine
=====
```

# In fact, the differences are 0

```
X_train, X_test, y_train, y_test = \
    train_test_split(X_red, y_red, \
                    test_size=0.3, random_state=2025, stratify=y_red)

g = GaussianNB()
g.fit(X_train, y_train)

m = GaussianNaiveBayes()
m.train(X_train, y_train)

print(g.theta_ - m.mu)

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

print(g.var_ - m.var)

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

diff = g._joint_log_likelihood(X_test) - m._joint_log_likelihood(X_test)

print(np.allclose(diff, 0, atol=1e-12))

True
```

This is achievable because in our `train()` method, we convert `X_train` and `y_train` into NumPy arrays using:

```
if isinstance(X_train, pd.DataFrame):  
    X_train = X_train.values  
if isinstance(y_train, pd.Series):  
    y_train = y_train.values
```

which is exactly what sklearn does when training on the dataset. This is “cheating” to some extent because it only works for pandas datasets (it could be expanded to other frameworks like Polars as well).

Without such conversion, due to slight differences in how arithmetic operations are handled between pandas objects and NumPy arrays, we would get results that differ slightly from sklearn’s results, but the magnitude is extremely small, on the order of  $1e-12$  to  $1e-17$ , which does not affect the final outcome.

We also verified that after removing this conversion, all checks still pass, and the accuracy score remains unchanged.

# Without Conversion

Mean difference without conversion

```
[[ -4.44089210e-16  0.00000000e+00  0.00000000e+00  0.00000000e+00
   2.71050543e-20  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00 -1.73472348e-18  0.00000000e+00]
 [ 4.44089210e-16 -1.38777878e-17 -6.93889390e-18  7.10542736e-15
  -1.08420217e-19 -5.68434189e-14  0.00000000e+00  1.69406589e-21
  -3.46944695e-18 -5.20417043e-18  4.44089210e-16]
 [ 1.44328993e-15  1.73472348e-17 -5.55111512e-17 -3.55271368e-14
  -1.08420217e-19  6.25277607e-13  2.95585778e-12  8.47032947e-21
   5.55111512e-17 -1.38777878e-17  3.10862447e-15]
 [ 0.00000000e+00 -5.72458747e-17  3.46944695e-17 -4.97379915e-14
  -5.42101086e-19  1.13686838e-13  1.81898940e-12 -2.20228566e-20
  -3.46944695e-17  2.25514052e-17  0.00000000e+00]
 [-1.22124533e-15  8.67361738e-18  8.67361738e-19 -1.77635684e-14
  1.89735380e-19  1.13686838e-13 -1.36424205e-12 -1.69406589e-21
  2.08166817e-17 -3.12250226e-17  1.11022302e-15]
 [ 0.00000000e+00  8.67361738e-18 -1.73472348e-18 -3.55271368e-15
  1.89735380e-19  0.00000000e+00  1.13686838e-13 -1.69406589e-21
  -2.77555756e-17 -1.38777878e-17 -8.88178420e-16]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

## What is interesting?

1. it can handle continuous features by gaussian
2. only need to estimate the mean and variance
3. closed-form, easy to implement
4. flexible framework: Gaussian can be optional
  - Bernoulli, Multinomial
  - as long as domain-match

## What was challenging?

### 1. Variance smoothing and numerical stability

```
epsilon = self.var_smoothing * np.var(X_train, axis=0).max()  
self.var += epsilon
```

### 2. Data conversion and shape alignment

we have to converge to numpy arrays to be 100% same as sklearn