Final Project

Name: Jiaqing Dai, Yingzhi Ma, Zi Tao

Team Name: Never Converge

Link to the github repo: https://github.com/Remi2000/Gaussian-Naive-Bayes.git

1. Overview of Gaussian Naive Bayes

Gaussian Naive Bayes (GNB) is a probabilistic supervised learning algorithm used for classification tasks involving continuous-valued features. It is a variant of the Naive Bayes family, which is based on applying Bayes' Rule with the naive assumption of conditional independence between features given the class label. Unlike the Bernoulli Naive Bayes models that assume discrete features, Gaussian Naive Bayes models each feature using a Gaussian (normal) distribution for every class.

GNB is widely used due to its mathematical simplicity, fast training time, and competitive performance on datasets where the independence assumption is approximately satisfied. Despite its conceptual simplicity, GNB often performs well even when its assumptions are not fully met, making it a strong tool for many machine learning tasks.

Reference Shalev-Shwartz, S. and Ben-David, S. (2014) Understanding Machine Learning: From Theory to Algorithms. New York, NY: Cambridge University Press. Available at: https://doi.org/10.1017/CBO9781107298019

Advantage and Disadvantage

Advantages

1. Very fast training and inference – Gaussian Naive Bayes only requires calculating the mean and variance of each feature for each class. There is no need for iterative optimization, unlike algorithms such as logistic regression or neural networks. This makes both training and prediction extremely efficient, suitable for large datasets and real-time applications.

2. Works well on small datasets – The model relies on simple closed-form maximum likelihood estimates, which reduces variance and overfitting. Even with limited training data, GNB can produce stable probability estimates for class membership, making it effective when data collection is difficult or expensive.

3. Interpretable probabilistic model – GNB outputs estimated likelihoods for each class rather than only a predicted label. This probabilistic interpretation allows users to evaluate confidence in predictions, which is particularly valuable in sensitive areas like medical diagnosis or credit scoring.

4. Robust to irrelevant features – The independence assumption reduces the influence of features that carry little information about the target class. This reduces overfitting in high-dimensional settings and improves model robustness when noisy or redundant features exist.

5. Low memory footprint – The model only stores the mean and variance of each feature per class along with class priors. There is no need for large weight matrices or iterative parameter storage, which makes it suitable for memory-constrained devices.

Disadvantages

1. Strong independence assumption – In real datasets, features are often correlated, violating the independence assumption. This can lead to inaccurate probability estimates and reduced predictive performance.
2. Assumes Gaussian distribution – GNB models each feature as normally distributed. When features are skewed, multimodal, or categorical, the Gaussian assumption is violated, which may degrade classification accuracy.
3. Linear decision boundaries in log-probability space – The model's decision boundaries are effectively linear, which limits flexibility compared to non-linear models like neural networks. Complex datasets with interacting features may be poorly classified.
4. Sensitive to variance estimation – Small variances can lead to numerical instability when computing likelihoods. Smoothing or adding a small epsilon is often required to ensure stable predictions.

2. Representation: From Features to Predictions

Given a feature vector $\mathbf{x} = (x_1, x_2, \ldots, x_d)$,

GNB models the conditional likelihood for each class $y \in \{1, \ldots, K\}$ as:

$$P(x_i \mid y) = \mathcal{N}(x_i; \mu_{y,i}, \sigma^2_{y,i}) = \frac{1}{\sqrt{2\pi\sigma^2_{y,i}}} \exp\left(-\frac{(x_i - \mu_{y,i})^2}{2\sigma^2_{y,i}}\right)$$

.

Under the independence assumption, the joint likelihood becomes:

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

.

Using Bayes' rule, the posterior is:

$$P(y \mid \mathbf{x}) \propto P(y) \prod_{i=1}^{d} P(x_i \mid y)$$

.

To avoid floating-point underflow, we compute the log-posterior:

$$\log P(y \mid \mathbf{x}) = \log P(y) + \sum_{i=1}^{d} \log \mathcal{N}(x_i; \mu_{y,i}, \sigma_{y,i}^2)$$

.

The predicted class is:

$$\hat{y} = \arg\max_y \log P(y \mid \mathbf{x})$$

Thus, the feature representation converts continuous inputs into probabilities governed by class-specific Gaussian distributions.

3. Loss Function

GNB does not explicitly minimize a loss function during training. Instead, training consists of computing Maximum Likelihood Estimates (MLE):

Class priors:

$$P(y) = \frac{N_y}{N}$$

Feature mean:

$$\mu_{y,i} = \frac{1}{N_y} \sum_{x \in y} x_i$$

Feature variance:

$$\sigma_{y,i}^2 = \frac{1}{N_y} \sum_{x \in y} (x_i - \mu_{y,i})^2$$

At evaluation time, models are typically assessed using classification accuracy or cross-entropy loss:

$$\text{Loss} = -\sum_{n=1}^{N} \log P(y_n \mid x_n)$$

4. Optimizer: Numerical Algorithm to Estimate Parameters

GNB does not require iterative optimization. All parameters have closed-form MLE solutions, making the algorithm extremely efficient.

Parameters learned:

Class prior probabilities $P(y)$

Feature means $\mu_{y,i}$

Feature variances $\sigma^2_{y,i}$

Pseudo-code for Training (MLE)

Train the Gaussian Naive Bayes model using closed-form Maximum Likelihood Estimation (MLE) as follows:

```
initialize priors $P(y)$, means $\mu[y][i]$, variances $\sigma^2[y][i]$
  for each class $c$ :
    $X_c \leftarrow$ all rows of X where label $=$ c
    $N_c \leftarrow$ number of samples in class $c$
    $P(c) \leftarrow N_c/N$
    for each feature $i \in \{1, \ldots, d\}$ :
      $\mu[c][i] \leftarrow$ mean of $X_c[:, i]$
      $\sigma^2[c][i] \leftarrow$ variance of $X_c[:, i] + \epsilon \quad$ (smoothing)
  return priors, means, variances
```

Pseudo-code for Prediction
To predict a class for a given input $\mathbf{x}$:

```
for each class $c$ :
  $\log\_post[c] = \log P(c)$
  for each feature $i \in \{1, \ldots, d\}$ :
    $\log\_post[c] += \log \mathcal{N}(x[i] \mid \mu[c][i], \sigma^2[c][i])$
return $\arg\max_c \log\_post[c]$
```

Coding Part

Run the environment test.

In [45]:
```python
from __future__ import print_function
from packaging.version import parse as Version
from platform import python_version

OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "\x1b[41m[FAIL]\x1b[0m"

try:
    import importlib
except ImportError:
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % sys.version)

def import_version(pkg, min_ver, fail_msg=""):
    mod = None
    try:
        mod = importlib.import_module(pkg)
        if pkg in {'PIL'}:
            ver = mod.VERSION
        else:
            ver = mod.__version__
        if Version(ver) == Version(min_ver):
            print(OK, "%s version %s is installed."
                  % (lib, min_ver))
        else:
            print(FAIL, "%s version %s is required, but %s installed."
                  % (lib, min_ver, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return mod


# first check the python version
pyversion = Version(python_version())

if pyversion >= Version("3.12.11"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.11"):
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
requirements = {'matplotlib': "3.10.5", 'numpy': "2.3.2",'sklearn': "1.7.1",
                'pandas': "2.3.2", 'pytest': "8.4.1", 'torch':"2.7.1"}

# now the dependencies
for lib, required_version in list(requirements.items()):
    import_version(lib, required_version)
```

```
[ OK ] Python version is 3.12.11

[ OK ] matplotlib version 3.10.5 is installed.
[ OK ] numpy version 2.3.2 is installed.
[ OK ] sklearn version 1.7.1 is installed.
[ OK ] pandas version 2.3.2 is installed.
[ OK ] pytest version 8.4.1 is installed.
[ OK ] torch version 2.7.1 is installed.
```

In our Gaussian Naive Bayes model, we implement the following core methods:

- `train()` estimates the model parameters from the training data. For each class, it computes the class prior P(y=c), as well as the mean and variance of every feature under that class, assuming a Gaussian distribution.
- `predict()` outputs a predicted label for each input sample. It calls the internal helper _predict_single() on each row of the input matrix.
- `_predict_single()` computes the log-posterior score for each class for a single input x. For every class, it adds the log prior and the sum of log Gaussian likelihoods over all features, and then returns the class with the highest score.
- `loss()` computes an average negative log-likelihood loss over a dataset. It measures how much probability mass the model assigns to the true labels. Lower loss means the model is more confident and better calibrated.
- `accuracy()` computes the fraction of correctly classified examples on a given dataset.

We avoid using any off-the-shelf Naive Bayes training routines. Instead, we implement the estimates of class priors, means, and variances directly from the data, and use the Gaussian likelihood formula to make predictions.

Model

In [25]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('Naive-Bayes-Classification-Data.csv')
print("DataFrame loaded successfully. Displaying the first 5 rows:")
df.head()

X = df[['glucose', 'bloodpressure']].values
y = df['diabetes'].values

print("X shape:", X.shape)
print("y shape:", y.shape)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```

```
print("Train size:", X_train.shape, y_train.shape)
print("Test size:", X_test.shape, y_test.shape)
```

```
DataFrame loaded successfully. Displaying the first 5 rows:
X shape: (995, 2)
y shape: (995,)
Train size: (696, 2) (696,)
Test size: (299, 2) (299,)
```

In [26]:
```
import numpy as np

class GaussianNaiveBayes:
    """
    A simple implementation of Gaussian Naive Bayes for classification.
    Each feature is modeled as an independent Gaussian distribution
    conditioned on the class label.
    """
    def __init__(self, epsilon=1e-9):
        """
        Initializes the Gaussian Naive Bayes model.

        @params:
            epsilon: a small smoothing value added to variances
                     to prevent division by zero
        """
        self.epsilon = epsilon
        self.classes_ = None
        self.priors_ = None
        self.means_ = None
        self.variances_ = None

    def train(self, X, y):
        """
        Trains the Gaussian Naive Bayes classifier using the training data.
        Computes class priors, per-class feature means, and per-class feature variances.
        """
        self.classes_ = np.unique(y)
        n_classes = len(self.classes_)
        n_features = X.shape[1]

        self.priors_ = np.zeros(n_classes)
        self.means_ = np.zeros((n_classes, n_features))
        self.variances_ = np.zeros((n_classes, n_features))

        for idx, c in enumerate(self.classes_):
            X_c = X[y == c]

            # Prior
            self.priors_[idx] = (X_c.shape[0] + 1e-6) / (X.shape[0] + n_classes * 1e-6)
            # Mean
            self.means_[idx, :] = X_c.mean(axis=0)

            # Variance smoothing: add (max_variance * 1e-9)
            var = X_c.var(axis=0)
            max_var = np.max(var)
            smoothing = max_var * 1e-9  # <-- your requirement

            self.variances_[idx, :] = var + smoothing

        return self




    def predict(self, X):
        """
        Predicts class labels for each input row in X.

        This method calls _predict_single() on each sample.

        @params:
            X: 2D NumPy array of shape (num_samples, num_features)
        @return:
            preds: a 1D NumPy array of predicted class labels
        """
        return np.array([self._predict_single(x) for x in X])


    def _predict_single(self, x):
        """
        Computes the log-posterior probability for every class
        and returns the class with maximum log-posterior.
        """

        posteriors = []

        for idx, c in enumerate(self.classes_):

        # SAFE variance: avoid division by zero / log(0)
            var = np.maximum(self.variances_[idx], 1e-12)

            prior_log = np.log(self.priors_[idx])

            cond_log = -0.5 * np.sum(
                np.log(2 * np.pi * var) +
                ((x - self.means_[idx]) ** 2) / var
            )
```

```python
            posteriors.append(prior_log + cond_log)

        return self.classes_[np.argmax(posteriors)]


    def loss(self, X, y_true):
        """
        Computes an average negative log-likelihood loss.

        This is NOT accuracy. It measures how confident the model is.
        A smaller loss means the model places high probability on the true labels.

        @params:
            X: 2D NumPy array of inputs
            y_true: 1D array of true class labels
        @return:
            average loss value (float)
        """
        n_samples = X.shape[0]
        loss_val = 0.0

        for i in range(n_samples):
            x = X[i]
            true_class = y_true[i]
            idx = np.where(self.classes_ == true_class)[0][0]

            prior = max(self.priors_[idx], 1e-12)
            var = np.maximum(self.variances_[idx], 1e-12)

            log_prob = np.log(prior) - 0.5 * np.sum(
                np.log(2 * np.pi * var) + ((x - self.means_[idx]) ** 2) / var
            )
            loss_val += max(0, -log_prob)

        return loss_val / n_samples

    def accuracy(self, X_test, y_test):
        """
        Computes prediction accuracy on a test set.

        @params:
            X_test: 2D NumPy array of test samples
            y_test: 1D NumPy array of true labels
        @return:
            accuracy as a float in [0, 1]
        """
        y_pred = self.predict(X_test)
        accuracy = np.mean(y_pred == y_test)
        return accuracy
```

Check Model

Reference   Himanshu Nakrani (no date) ‘Naive Bayes Classification Data’ [online]. Kaggle. Available at: https://www.kaggle.com/datasets/himanshunakrani/naive-bayes-classification-data/data

In [27]:
```
# DO NOT EDIT!
import pytest
np.random.seed(0)

# Create test models
test_gnb1 = GaussianNaiveBayes()
test_gnb2 = GaussianNaiveBayes()
test_gnb3 = GaussianNaiveBayes()

# ============================
# Test Data
# ============================
x1 = np.array([[0,0,1], [0,1,0], [1,0,1], [1,1,1], [0,0,1]])
y1 = np.array([0,0,1,1,0])
x_test1 = np.array([[1,0,0],[0,0,0],[1,1,1],[0,1,0],[1,1,0]])
y_test1 = np.array([0,0,1,0,1])

x2 = np.array([[0,0,1], [0,1,1], [1,1,1], [1,1,1], [0,0,0], [1,1,0]])
y2 = np.array([0,1,1,1,0,1])
x_test2 = np.array([[0,0,1],[0,1,1],[1,1,1],[1,0,0]])
y_test2 = np.array([0,1,1,0])

x3 = np.array([
    [0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],
    [0,1,0,0],[0,1,0,1],[0,1,1,0],[0,1,1,1],
    [1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1]
])
y3 = np.array([0,0,1,1,1,0,2,0,0,2,1,1])
x_test3 = np.array([[1,1,0,0],[1,1,0,1],[1,1,1,0],[1,1,1,1]])
y_test3 = np.array([1,1,0,0])

# ==================================================
# Helper functions
# ==================================================
def check_train_dtype(model, means, variances, priors, x_train):
    assert isinstance(means, np.ndarray)
    assert means.ndim == 2 and means.shape == (len(model.classes_), x_train.shape[1])
    assert isinstance(variances, np.ndarray)
    assert variances.ndim == 2 and variances.shape == (len(model.classes_), x_train.shape[1])
```

```python
    assert isinstance(priors, np.ndarray)
    assert priors.ndim == 1 and priors.shape == (len(model.classes_),)


def check_test_dtype(pred, x_test):
    assert isinstance(pred, np.ndarray)
    assert pred.ndim == 1 and pred.shape == (x_test.shape[0], )


# ==================================================
# Train Tests
# ==================================================
test_gnb1.train(x1, y1)
check_train_dtype(test_gnb1, test_gnb1.means_, test_gnb1.variances_, test_gnb1.priors_, x1)

expected_means1 = np.array([
    x1[y1 == 0].mean(axis=0),
    x1[y1 == 1].mean(axis=0)
])
assert test_gnb1.means_ == pytest.approx(expected_means1, 0.001)

expected_priors1 = np.array([
    3 / 5,
    2 / 5
])
assert test_gnb1.priors_ == pytest.approx(expected_priors1, 0.01)


# ==================================================
# Predict Tests
# ==================================================
pred1 = test_gnb1.predict(x_test1)
check_test_dtype(pred1, x_test1)

pred2 = test_gnb2.train(x2, y2).predict(x_test2)
check_test_dtype(pred2, x_test2)

pred3 = test_gnb3.train(x3, y3).predict(x_test3)
check_test_dtype(pred3, x_test3)


# ==================================================
# Accuracy Tests
# ==================================================
acc1 = test_gnb1.accuracy(x_test1, y_test1)
acc2 = test_gnb2.accuracy(x_test2, y_test2)
acc3 = test_gnb3.accuracy(x_test3, y_test3)

assert 0.0 <= acc1 <= 1.0
assert 0.0 <= acc2 <= 1.0
assert 0.0 <= acc3 <= 1.0

# ===== Edge Case 1: Zero Variance Feature =====
X_zero_var = np.array([
    [1.0, 5.0],
    [1.0, 5.0],
    [1.0, 5.0],
    [1.0, 5.0]
])
y_zero_var = np.array([0, 0, 1, 1])

model = GaussianNaiveBayes()
model.train(X_zero_var, y_zero_var)

# Variance may be zero now (because smoothing depends on max variance),
# but predict() must still work without errors.
try:
    pred = model.predict(X_zero_var)
    assert pred.shape == (4,), "Prediction shape incorrect"
except Exception as e:
    assert False, f"Model failed on zero-variance data: {e}"

print("Zero variance feature test passed.")

# ===== Edge Case 2: Contradictory Labels =====
X_contradict = np.array([
    [1.0, 2.0],
    [1.0, 2.0]
])
y_contradict = np.array([0, 1])

model = GaussianNaiveBayes()
model.train(X_contradict, y_contradict)

# means should be the same because our data is the same
assert np.allclose(model.means_[0], model.means_[1]), \
       "Means must match because X values are identical."

pred = model.predict(np.array([[1.0, 2.0]]))

assert pred[0] in [0, 1], "Prediction must be one of the two labels."

print("Contradictory label test passed.")
```

```
Zero variance feature test passed.
Contradictory label test passed.
```

Main

Reproducing sklearn GaussianNB on a public Kaggle dataset

To validate that our implementation of Gaussian Naive Bayes is correct, we compare it against `sklearn.naive_bayes.GaussianNB` on a public dataset from Kaggle:

> **Dataset:** *Naive-Bayes-Classification-Data.csv*
> **Source:** Kaggle – "glucose and blood pressure data to classify whether the patient has diabetes or not."
> The dataset contains 995 entries and 3 columns:
>
> - `glucose` : numeric feature (blood glucose level)
> - `bloodpressure` : numeric feature (blood pressure)
> - `diabetes` : binary label (0 = no diabetes, 1 = diabetes)

We process the data as follows:

- Use `glucose` and `bloodpressure` as continuous input features, which fits the Gaussian Naive Bayes assumption.
- Use `diabetes` as the target label.
- Split the dataset into 70% training and 30% test data using `train_test_split(test_size=0.3, random_state=42, stratify=y)` .

For the comparison:

1. Train our own `GaussianNaiveBayes` implementation on the training split and evaluate its accuracy on the test split.
2. Train sklearn's `GaussianNB` on the **same** `(X_train, y_train)` and evaluate it on the **same** `X_test` .
3. Compare both the test accuracies and the predicted labels on the test set.

If our implementation is correct, it should achieve the same accuracy as sklearn on this dataset and produce an identical sequence of predictions.

Reference [1] NaKrani, H. (2023). *Naive Bayes Classification Data*. Kaggle Dataset. https://www.kaggle.com/datasets/himanshunakrani/naive-bayes-classification-data

In [30]:
```python
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

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

acc_my = model_my.accuracy(X_test, y_test)
print("My Gaussian NB accuracy:", acc_my)

model_sk = GaussianNB()
model_sk.fit(X_train, y_train)
```

```python
y_pred_sk = model_sk.predict(X_test)
acc_sk = accuracy_score(y_test, y_pred_sk)
print("Sklearn GaussianNB accuracy:", acc_sk)


print("Predictions identical:", np.array_equal(y_pred_my, y_pred_sk))
proba_sk = model_sk.predict_proba(X_test)
pred_sk_by_proba = np.argmax(proba_sk, axis=1)

print("Posterior ranking identical:",
      np.array_equal(pred_sk_by_proba, y_pred_my))

print("\nConclusion: Since priors, means, variances, and posterior ranking match,")
print("our implementation reproduces sklearn at both the parameter level and the prediction level.")
```

```
My Gaussian NB accuracy: 0.9264214046822743
Sklearn GaussianNB accuracy: 0.9264214046822743
Predictions identical: True
Posterior ranking identical: True

Conclusion: Since priors, means, variances, and posterior ranking match,
our implementation reproduces sklearn at both the parameter level and the prediction level.
```

Reference:

Shalev-Shwartz, S. and Ben-David, S. (2014) Understanding Machine Learning: From Theory to Algorithms. New York, NY: Cambridge University Press. Available at: https://doi.org/10.1017/CBO9781107298019 .

Himanshu Nakrani (no date) Naive Bayes Classification Data. Kaggle. Available at: https://www.kaggle.com/datasets/himanshunakrani/naive-bayes-classification-data/data (Accessed: 8 December 2025).

Pedregosa, F. et al. (2011) ‘Scikit-learn: Machine Learning in Python’. Journal of Machine Learning Research, 12, pp. 2825–2830. Documentation available at: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html