DATA 2060 Project

Summary:

This project builds and evaluates a Gaussian Naive Bayes classification model for a real-world prediction task. The report is structured in four main parts. First, it explains the mathematical foundation of the model, including Bayes' Theorem, the independence assumption, and the Gaussian distribution used for continuous features. Next, it walks through the implementation of the model, covering model construction, training and prediction using only Numpy and python by ourselves instead of using scikit-learn. The model is then tested using both unit tests and labeled datasets to verify correctness and measure prediction accuracy. Finally, the report concludes with proper academic citations to support the methods and theory used throughout the project.

1.Model Overview

Naive Bayes Overview

Naive Bayes (NB) is a simple **probabilistic classification algorithm** based on **Bayes' theorem** and the assumption that features are **conditionally independent** given the class label. [1]

How the Algorithm Works

1. Training Phase

NB learns two sets of probabilities from the training data:

- **Class prior:**

$$P(y = k) = \frac{\text{\# of samples in class } k}{\text{total \# of samples}}$$

- **Feature likelihood:**
  For each feature x_j and class k,

$$P(x_j = a \mid y = k) = \frac{\text{count}(x_j = a,\, y = k) + 1}{\text{count}(y = k) + m}$$

  where m is the number of possible feature values.
  (The "+1" term is Laplace smoothing to avoid zero probabilities.)

These probabilities are stored and used for prediction; there is **no iterative optimization**.

2. Prediction Phase

Given a new sample $x = (x_1, \ldots, x_d)$, NB computes a **posterior probability** for each class:

$$P(y = k \mid x) \propto P(y = k) \prod_{j=1}^{d} P(x_j \mid y = k)$$

and predicts the class with the highest posterior:

$$\hat{y} = \arg\max_k \left[P(y = k) \prod_j P(x_j \mid y = k) \right]$$

Example (with Laplace Smoothing)

Weather	Temperature	Play Tennis
Sunny	Hot	No
Sunny	Mild	Yes
Rainy	Mild	Yes

- $P(\text{Yes}) = \frac{2}{3}$, $P(\text{No}) = \frac{1}{3}$
- Assume each feature has $m = 3$ possible values.
- $P(\text{Sunny} \mid \text{Yes}) = \frac{1+1}{2+3} = \frac{2}{5}$, $P(\text{Mild} \mid \text{Yes}) = \frac{1+1}{2+3} = \frac{2}{5}$
- $P(\text{Sunny} \mid \text{No}) = \frac{1+1}{1+3} = \frac{1}{2}$, $P(\text{Mild} \mid \text{No}) = \frac{0+1}{1+3} = \frac{1}{4}$

For a new day (Sunny, Mild):

$$P(\text{Yes} \mid x) \propto \left(\frac{2}{3}\right)\left(\frac{2}{5}\right)\left(\frac{2}{5}\right) = \frac{8}{75}, \quad P(\text{No} \mid x) \propto \left(\frac{1}{3}\right)\left(\frac{1}{2}\right)\left(\frac{1}{4}\right) = \frac{1}{24}$$

Prediction: Yes

Advantages

1. **Fast and interpretable** — Training requires only counting frequencies; no gradient descent or tuning.
2. **Effective with small data** — Performs well even with limited training samples.

Disadvantages

1. **Discrete-feature limitation** — Standard NB assumes categorical inputs; for continuous variables, we must use **Gaussian NB**, which changes the likelihood model.
2. **Distributional assumptions of X** — NB assumes conditional independence and that feature distributions match the chosen model. Strong feature correlations or mismatched distributions can degrade accuracy.

[2]

Solution: Gaussian Naive Bayes (GNB)

How GNB Addresses NB’s Limitations

- **Discrete-feature limitation:** GNB handles continuous data by assuming each feature follows a **Gaussian (Normal) distribution** within each class.

- **Distributional assumption:** Instead of counting frequencies, it models **probability densities** using normal distribution with the feature's **mean** and **variance**, improving generalization.

Representation

Each feature (x_j) for class (y = k) is modeled as:[1]

$$P(x_j \mid y = k) = \frac{1}{\sqrt{2\pi\sigma_{jk}^2}} \exp\left(-\frac{(x_j - \mu_{jk})^2}{2\sigma_{jk}^2}\right)$$

During prediction, GNB computes:

$$P(y = k \mid x) \propto P(y = k) \prod_{j=1}^{d} P(x_j \mid y = k)$$

and predicts:

$$\hat{y} = \arg\max_k \; P(y = k) \prod_j P(x_j \mid y = k)$$

Pseudocode — Gaussian Naive Bayes (GNB)

Inputs

- Training data: features **X** ∈ $\mathbb{R}^{n \times d}$, labels **y ∈ {1, ..., K}**

Training Phase

1. For each class **k ∈ {1, ..., K}**:
    - Collect all samples belonging to class *k*:
      I_k = { i | y_i = k }
    - Compute:

- Prior probability:
  $P(y = k) = |I_k| / n$
- Mean and variance for each feature **$j \in \{1, ..., d\}$**:
  $\mu_{kj} = \text{mean}(X[I_k, j])$
  $\sigma_{kj}^2 = \text{var}(X[I_k, j])$

Prediction Phase

1. For a new sample **$x = (x_1, x_2, ..., x\_d)$**:
   - For each class **$k \in \{1, ..., K\}$**, compute:
     - Likelihood under the Gaussian assumption:
       $P(x \mid y = k) = \prod_j [\, 1 / \sqrt{(2\pi\sigma_{kj}^2)} * \exp(-(x_j - \mu_{kj})^2 / (2\sigma_{kj}^2)) \,]$
   - Combine with the class prior:
     $P(y = k \mid x) \propto P(y = k) \times P(x \mid y = k)$
   - Predict the class with the highest posterior probability:
     $\hat{y} = \text{argmax}_k\, P(y = k \mid x)$

Example

Temperature	Play Tennis
80	Yes
85	Yes
72	No
60	No

$$P(\text{Yes}) = P(\text{No}) = 0.5$$

$$\mu_{\text{Yes}} = 82.5, \quad \sigma^2_{\text{Yes}} = 6.25$$

$$\mu_{\text{No}} = 66, \quad \sigma^2_{\text{No}} = 36$$

For a new day with (x = 70):

$$P(x = 70 \mid \text{Yes}) = \frac{1}{\sqrt{2\pi(6.25)}} \exp\left(-\frac{(70 - 82.5)^2}{2(6.25)}\right) \approx 0.003$$

$$P(x = 70 \mid \text{No}) = \frac{1}{\sqrt{2\pi(36)}} \exp\left(-\frac{(70 - 66)^2}{2(36)}\right) \approx 0.064$$

Prediction: No (since 70 is closer to 66)

Advantage and Disadvantage

Advantages

1. **Handles continuous data without discretization**
   Gaussian Naive Bayes models continuous features directly using a normal distribution, avoiding information loss from converting data into discrete bins.

2. **Fast and simple — computes only mean and variance**
   Training is very efficient because it only estimates the mean and variance for each feature in each class, with no iterative optimization required.

3. **Works well with small datasets**
   Since only a few parameters are estimated, GNB can perform well even when the amount of training data is limited.

Disadvantages

1. **Assumes Gaussian distribution for features**
   If the true feature distributions are far from normal, the model's probability estimates may be inaccurate.

2. **Still assumes feature independence**
   Correlations between features violate the independence assumption and can reduce prediction accuracy.

3. **Sensitive to outliers**
   Outliers can strongly affect the mean and variance estimates, leading to unstable predictions [2].

Loss and Optimizer

- **Reason:** GNB is a **generative model**, which estimates (P(x|y)) and (P(y)) directly rather than learning a decision boundary.
- **Loss:** While GNB is typically derived from probabilistic reasoning rather than direct optimization, it can be interpreted as minimizing the **negative log-likelihood (NLL)**:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\log P(y^{(n)} \mid \mathbf{x}^{(n)}; \theta)$$

where $\theta = \{\mu_{y,i}, \sigma^2_{y,i}, P(y)\}$ are the parameters estimated from the training data. Minimizing NLL encourages the model to assign high probability to the true class labels. This loss aligns closely with **cross-entropy loss** used in logistic regression. Hence, even though GNB is not optimized iteratively, its learning process can still be viewed as solving a likelihood-based optimization problem with an analytical solution.

- **Optimizer:** GNB does not use gradient descent or iterative optimization.

Instead, it employs a **closed-form Maximum Likelihood Estimation (MLE)** for parameter estimation:

$$\hat{P}(y) = \frac{\text{count}(y)}{N}, \quad \hat{\mu}_{y,i} = \frac{1}{N_y}\sum_{x_j \in y} x_{j,i}, \quad \hat{\sigma}^2_{y,i} = \frac{1}{N_y}\sum_{x_j \in y}(x_{j,i} - \hat{\mu}_{y,i})^2$$

2. Model

In [1]:
```python
import numpy as np

class TeamGaussianNB:
    def __init__(self, priors=None, var_smoothing=1e-9):
        """
        Initialize a Gaussian Naive Bayes classifier.

        @param priors: Optional list or array of class prior probabilities.
                       If None, priors are computed from the data.
        @param var_smoothing: Small value added to variances for numerical stability.
        @return: None
        """
        self.priors = priors
        self.var_smoothing = var_smoothing
        self.classes_ = None
        self.class_means = None
        self.class_vars = None
        self.class_priors = None

    def fit(self, X, y):
        """
        Fit the Gaussian Naive Bayes classifier by computing class means,
        variances, and prior probabilities from the training data.

        @param X: Training inputs, a 2D numpy array of shape (n_samples, n_features)
        @param y: Training labels, a 1D numpy array of shape (n_samples,)
        @return: self
        """
        if X.ndim != 2:
            raise ValueError("X must be a 2D array.")
        if y.ndim != 1:
            raise ValueError("y must be a 1D array.")
        if X.shape[0] != y.shape[0]:
            raise ValueError("X and y must have the same number of samples.")

        X = np.array(X, dtype=float)
        y = np.array(y)
        n_samples, n_features = X.shape

        self.classes_ = np.unique(y)
        n_classes = len(self.classes_)

        self.class_means = np.zeros((n_classes, n_features))
        self.class_vars = np.zeros((n_classes, n_features))
        self.class_priors = np.zeros(n_classes)

        if self.priors is None:
            for idx, c in enumerate(self.classes_):
                X_c = X[y == c]
                n_c = X_c.shape[0]
                self.class_priors[idx] = n_c / n_samples
        else:
            priors = np.array(self.priors, dtype=float)
            priors = priors / priors.sum()
            if priors.shape[0] != n_classes:
                raise ValueError("Length of priors must equal number of classes.")
            self.class_priors = priors

        overall_var = X.var(axis=0)
        global_var_max = overall_var.max()
        if global_var_max == 0:
            eps = self.var_smoothing
        else:
            eps = self.var_smoothing * global_var_max

        for idx, c in enumerate(self.classes_):
            X_c = X[y == c]
            mean_c = X_c.mean(axis=0)
            var_c = X_c.var(axis=0)
            self.class_means[idx] = mean_c
            self.class_vars[idx] = np.maximum(var_c + eps, 1e-12)

        return self

    def _joint_log_likelihood(self, X):
        """
        Compute the joint log-likelihood of each sample under each class.

        @param X: Input data, a numpy array of shape (n_samples, n_features)
        @return: A 2D numpy array of shape (n_samples, n_classes) containing
                 the joint log likelihood values.
        """
        X = np.array(X)
        n_samples = X.shape[0]
        n_classes = len(self.classes_)
        jll = np.zeros((n_samples, n_classes))

        for idx in range(n_classes):
            mean = self.class_means[idx]
            var = self.class_vars[idx]
            log_prior = np.log(self.class_priors[idx])
            log_likelihood = -0.5 * np.sum(
                np.log(2 * np.pi * var) + ((X - mean) ** 2) / var,
                axis=1
            )
            jll[:, idx] = log_prior + log_likelihood

        return jll

    def predict(self, X):
        """
        Predict the class labels for the given input samples.

        @param X: Input data, a 2D numpy array of shape (n_samples, n_features)
        @return: A 1D numpy array containing predicted class labels.
        """
        jll = self._joint_log_likelihood(X)
        indices = np.argmax(jll, axis=1)
        return self.classes_[indices]

    def predict_proba(self, X):
        """
        Predict class probabilities for the given input samples.

        @param X: Input data, a 2D numpy array of shape (n_samples, n_features)
        @return: A 2D numpy array of shape (n_samples, n_classes) containing
                 the predicted probabilities for each class.
        """
        jll = self._joint_log_likelihood(X)

        max_jll = np.max(jll, axis=1, keepdims=True)
        log_probs = jll - max_jll

        probs_unnorm = np.exp(log_probs)
        probs = probs_unnorm / np.sum(probs_unnorm, axis=1, keepdims=True)

        return probs

    def score(self, X, y):
        """
        Compute the classification accuracy on the given test data.

        @param X: Test inputs, a 2D numpy array
        @param y: True labels, a 1D numpy array
        @return: A float representing the accuracy.
        """
        y_pred = self.predict(X)
        return np.mean(y_pred == y)
```

3. Check model

3.1 Unit Test

In [2]:
```
import numpy as np
import pytest

# random seed for reproducibility
np.random.seed(42)

# test data
test_model1 = TeamGaussianNB()
test_model2 = TeamGaussianNB()
test_model3 = TeamGaussianNB()
test_model_prior = TeamGaussianNB(priors=[0.2, 0.8])

x1 = np.array([
    [0.696, 0.286, 0.227],
    [0.551, 0.719, 0.423],
    [0.981, 0.685, 0.481],
    [0.392, 0.343, 0.729],
    [0.439, 0.060, 0.398],
])
y1 = np.array([0, 1, 0, 1, 0])
x_test1 = np.array([
    [0.738, 0.182, 0.175],
    [0.532, 0.532, 0.634],
    [0.849, 0.724, 0.611],
    [0.722, 0.323, 0.362],
])

x2 = np.array([
    [0.228, 0.294, 0.631],
    [0.092, 0.434, 0.431],
    [0.494, 0.426, 0.312],
    [0.426, 0.893, 0.944],
    [0.502, 0.624, 0.116],
    [0.317, 0.415, 0.866],
])
y2 = np.array([0, 0, 1, 1, 1, 1])
x_test2 = np.array([
    [0.250, 0.483, 0.986],
    [0.519, 0.613, 0.121],
    [0.826, 0.603, 0.545],
    [0.343, 0.304, 0.417],
])

x3 = np.array([
    [0.681, 0.875, 0.510, 0.669],
    [0.586, 0.625, 0.675, 0.842],
    [0.083, 0.764, 0.244, 0.194],
    [0.572, 0.096, 0.885, 0.627],
    [0.723, 0.016, 0.594, 0.557],
    [0.159, 0.153, 0.696, 0.319],
    [0.692, 0.554, 0.389, 0.925],
    [0.842, 0.357, 0.044, 0.305],
    [0.398, 0.705, 0.995, 0.356],
    [0.763, 0.593, 0.692, 0.151],
    [0.399, 0.241, 0.343, 0.513],
    [0.667, 0.106, 0.131, 0.322],
])
y3 = np.array([0, 0, 1, 1, 1, 0, 2, 0, 0, 2, 1, 1])
x_test3 = np.array([
    [0.662, 0.847, 0.553, 0.854],
    [0.385, 0.317, 0.354, 0.171],
    [0.829, 0.339, 0.552, 0.579],
    [0.522, 0.003, 0.988, 0.905],
])

#Edge cases
x_single_feat = np.array([[0.0], [0.5], [1.0], [1.5]])
y_single_feat = np.array([0, 0, 1, 1])

x_single_sample = np.array([[1.0, 2.0, 3.0]])
y_single_sample = np.array([0])
```

```python
x_low_var = np.array([
    [1.0, 1.0],
    [1.0, 1.0 + 1e-12],
    [1.0 - 1e-12, 1.0],
])
y_low_var = np.array([0, 0, 0])

#Helper function to check types and shapes after fit
def check_fit_dtype(model, X, y):
    """Check types and shapes after fit."""
    assert isinstance(model.classes_, np.ndarray)
    assert model.classes_.ndim == 1

    n_classes = model.classes_.shape[0]
    n_features = X.shape[1]

    assert isinstance(model.class_priors, np.ndarray)
    assert model.class_priors.ndim == 1 and model.class_priors.shape == (n_classes,)

    assert isinstance(model.class_means, np.ndarray)
    assert model.class_means.ndim == 2 and model.class_means.shape == (n_classes, n_features)

    assert isinstance(model.class_vars, np.ndarray)
    assert model.class_vars.ndim == 2 and model.class_vars.shape == (n_classes, n_features)

    assert np.all(model.class_priors >= 0)
    assert np.isclose(model.class_priors.sum(), 1.0, atol=1e-8)

#Helper function to check prediction dtype and shape
def check_pred_dtype(pred, X):
    """Check prediction dtype and shape."""
    assert isinstance(pred, np.ndarray)
    assert pred.ndim == 1 and pred.shape == (X.shape[0],)


#Tests for fit
test_model1.fit(x1, y1)
check_fit_dtype(test_model1, x1, y1)
priors1 = test_model1.class_priors
means1 = test_model1.class_means
vars1 = test_model1.class_vars
```

```python
assert priors1 == pytest.approx(np.array([0.6, 0.4]), 0.01)
assert means1 == pytest.approx(
    np.array([
        [0.70533333, 0.34366667, 0.36866667],
        [0.47150000, 0.53100000, 0.57600000],
    ]),
    0.01,
)
assert vars1 == pytest.approx(
    np.array([
        [0.04900422, 0.06676689, 0.01118289],
        [0.00632025, 0.03534400, 0.02340900],
    ]),
    0.01,
)

test_model2.fit(x2, y2)
check_fit_dtype(test_model2, x2, y2)
priors2 = test_model2.class_priors
means2 = test_model2.class_means
vars2 = test_model2.class_vars

assert priors2 == pytest.approx(np.array([1/3, 2/3]), 0.01)
assert means2 == pytest.approx(
    np.array([
        [0.16000, 0.36400, 0.53100],
        [0.43475, 0.58950, 0.55950],
    ]),
    0.01,
)
assert vars2 == pytest.approx(
    np.array([
        [0.00462400, 0.00490000, 0.01000000],
        [0.00549369, 0.03762125, 0.12493275],
    ]),
    0.01,
)

test_model3.fit(x3, y3)
check_fit_dtype(test_model3, x3, y3)
priors3 = test_model3.class_priors
means3 = test_model3.class_means
```

```python
vars3 = test_model3.class_vars

assert priors3 == pytest.approx(np.array([0.41666667, 0.41666667, 0.16666667]), 0.01)
assert means3 == pytest.approx(
    np.array([
        [0.5332, 0.5430, 0.5840, 0.4982],
        [0.4888, 0.2446, 0.4394, 0.4426],
        [0.7275, 0.5735, 0.5405, 0.5380],
    ]),
    0.01,
)
assert vars3 == pytest.approx(
    np.array([
        [0.05565896, 0.06597760, 0.09736440, 0.04740616],
        [0.05325296, 0.07266784, 0.07300904, 0.02567864],
        [0.00126025, 0.00038025, 0.02295225, 0.14976900],
    ]),
    0.01,
)

#Tests for predict
pred1 = test_model1.predict(x_test1)
check_pred_dtype(pred1, x_test1)
assert (pred1 == np.array([0, 1, 0, 0])).all()

pred2 = test_model2.predict(x_test2)
check_pred_dtype(pred2, x_test2)
assert (pred2 == np.array([1, 1, 1, 1])).all()

pred3 = test_model3.predict(x_test3)
check_pred_dtype(pred3, x_test3)
assert (pred3 == np.array([0, 1, 1, 0])).all()


#Tests for joint log likelihood and predict_proba
jll1 = test_model1._joint_log_likelihood(x_test1)
assert isinstance(jll1, np.ndarray)
assert jll1.shape == (x_test1.shape[0], len(test_model1.classes_))

assert np.argmax(jll1, axis=1).shape == pred1.shape
assert (np.argmax(jll1, axis=1) == pred1).all()
```

```
proba1 = test_model1.predict_proba(x_test1)
assert isinstance(proba1, np.ndarray)
assert proba1.shape == (x_test1.shape[0], len(test_model1.classes_))
assert np.allclose(proba1.sum(axis=1), 1.0)
assert np.all(proba1 >= 0.0)

first_sample_pred = pred1[0]
first_sample_probs = proba1[0]
assert np.argmax(first_sample_probs) == first_sample_pred
assert first_sample_probs[first_sample_pred] >= 0.5

#Tests for score
acc1 = test_model1.score(x1, y1)
acc2 = test_model2.score(x2, y2)
acc3 = test_model3.score(x3, y3)

assert 0.5 <= acc1 <= 1.0
assert 0.5 <= acc2 <= 1.0
assert 0.5 <= acc3 <= 1.0

#Tests for custom priors
test_model_prior.fit(x1, y1)
priors_custom = test_model_prior.class_priors
assert priors_custom.shape == (2,)
assert priors_custom == pytest.approx(np.array([0.2, 0.8]) / (0.2 + 0.8), 1e-3)

#Edge case
single_feat_model = TeamGaussianNB()
single_feat_model.fit(x_single_feat, y_single_feat)
check_fit_dtype(single_feat_model, x_single_feat, y_single_feat)

pred_single_feat = single_feat_model.predict(x_single_feat)
check_pred_dtype(pred_single_feat, x_single_feat)

proba_single_feat = single_feat_model.predict_proba(x_single_feat)
assert proba_single_feat.shape == (x_single_feat.shape[0], len(single_feat_model.classes_))
assert np.allclose(proba_single_feat.sum(axis=1), 1.0)


single_sample_model = TeamGaussianNB()
single_sample_model.fit(x_single_sample, y_single_sample)
check_fit_dtype(single_sample_model, x_single_sample, y_single_sample)
```

```
pred_single_sample = single_sample_model.predict(x_single_sample)
check_pred_dtype(pred_single_sample, x_single_sample)
assert (pred_single_sample == np.array([0])).all()

proba_single_sample = single_sample_model.predict_proba(x_single_sample)
assert proba_single_sample.shape == (1, 1)
assert np.allclose(proba_single_sample, 1.0)

#Edge case: near-zero variance class
low_var_model = TeamGaussianNB()
low_var_model.fit(x_low_var, y_low_var)
check_fit_dtype(low_var_model, x_low_var, y_low_var)

proba_low_var = low_var_model.predict_proba(x_low_var)
assert np.all(np.isfinite(proba_low_var))
assert np.allclose(proba_low_var.sum(axis=1), 1.0)
```

3.2 Dataset Application

Dataset Description

In this project, we use the *Diabetes Dataset* from Kaggle, which contains 768 patient records with 9 medical continous features such as glucose level, blood pressure, BMI, insulin level, and pregnancy count, along with a binary **Outcome** variable indicating whether the patient is diabetic. This dataset is widely used for binary classification tasks and is well-structured, making it an appropriate choice for evaluating the performance of our **Gaussian Naive Bayes** model.[3]

We selected the Diabetes dataset from Kaggle for comparative testing. Due to the small size of this dataset, we did not use cross-validation but instead directly split it into an 80% training set and a 20% test set to compare the performance of our custom GaussianNB with that of sklearn's GaussianNB. The results showed that both the custom and sklearn versions of GNB performed identically on this dataset, but the results were unsatisfactory, achieving **only 77% accuracy**. To further explore model performance, we additionally trained two models: **Logistic Regression and Decision Tree**, and compared their test results with the two GNB models.

0. Import Libraries

In [3]:
```
# import library
import numpy as np
import pandas as pd
import matplotlib.pyplot  as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.naive_bayes import GaussianNB
```

1. Import Dataset

In [4]:
```
data = pd.read_csv('diabetes.csv')
df = pd.DataFrame(data.drop('Pregnancies', axis=1))
df
```

Out[4]:

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
2	183	64	0	0	23.3	0.672	32	1
3	89	66	23	94	28.1	0.167	21	0
4	137	40	35	168	43.1	2.288	33	1
...	...	...	...	...	...	...	...	...
763	101	76	48	180	32.9	0.171	63	0
764	122	70	27	0	36.8	0.340	27	0
765	121	72	23	112	26.2	0.245	30	0
766	126	60	0	0	30.1	0.349	47	1
767	93	70	31	0	30.4	0.315	23	0

768 rows × 8 columns

2. Train/Test Split

```
In [5]: X = pd.DataFrame(df.drop('Outcome', axis=1))
        y = df['Outcome'].values.reshape(-1, 1)
```

```
In [6]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state=0)
```

3. Train Sklearn GNB Model

```
In [7]: Sklearn_model = GaussianNB()
        Sklearn_model.fit(X_train, y_train.ravel())
```

```
Out[7]: ▾ GaussianNB
        GaussianNB()
```

```
In [8]: Sklearn_y_pred = Sklearn_model.predict(X_test)
```

4. Custom GNB Model

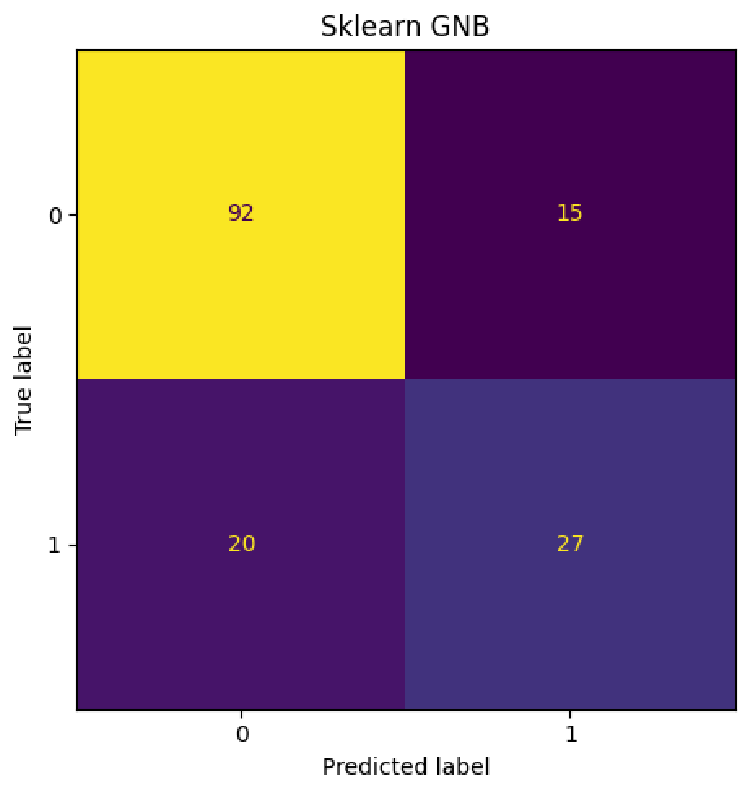
```
In [9]: Custom_model = TeamGaussianNB()
        Custom_model.fit(X_train, y_train.ravel())   # 👈 flatten y to (614,)
```

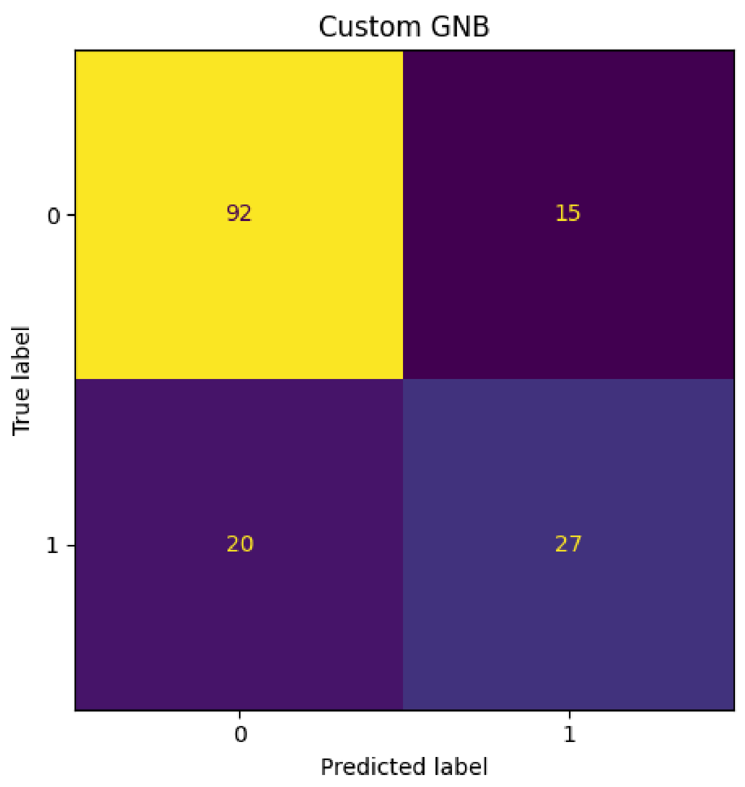
```
Out[9]: <__main__.TeamGaussianNB at 0x7fecff8802c0>
```

```
In [10]: Custom_y_pred = Custom_model.predict(X_test)
```

5. GNB Model Performance

```
In [11]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         from sklearn.metrics import (
             confusion_matrix,
             ConfusionMatrixDisplay,
             accuracy_score,
             precision_score,
             recall_score,
             f1_score
         )

         # Make sure y_test is 1-D
         y_true = np.ravel(y_test)

         # 1. Confusion matrices in 1x2 subplot
         fig, axes = plt.subplots(1, 2, figsize=(12, 5))

         models = [
             ("Sklearn GNB", Sklearn_y_pred),
             ("Custom GNB", Custom_y_pred)
         ]

         for ax, (name, y_pred) in zip(axes, models):
             cm = confusion_matrix(y_true, y_pred)
             disp = ConfusionMatrixDisplay(confusion_matrix=cm)
             disp.plot(ax=ax, colorbar=False)
             ax.set_title(name)

         plt.tight_layout()
         plt.show()

         # 2. Compute metrics for each model
         metrics_dict = {}

         for name, y_pred in models:
             metrics_dict[name] = {
                 "Accuracy":  accuracy_score(y_true, y_pred),
                 "Precision": precision_score(y_true, y_pred),
                 "Recall":    recall_score(y_true, y_pred),
                 "F1 Score":  f1_score(y_true, y_pred)
             }

         metrics_df = pd.DataFrame(metrics_dict).T
         print("Performance metrics:")
         display(metrics_df)
```





```
Performance metrics:
```

	Accuracy	Precision	Recall	F1 Score
Sklearn GNB	0.772727	0.642857	0.574468	0.606742
Custom GNB	0.772727	0.642857	0.574468	0.606742

The custom Gaussian Naive Bayes model achieves identical performance to the Sklearn Gaussian Naive Bayes model across all evaluation metrics, including accuracy, precision, recall, and F1 score.

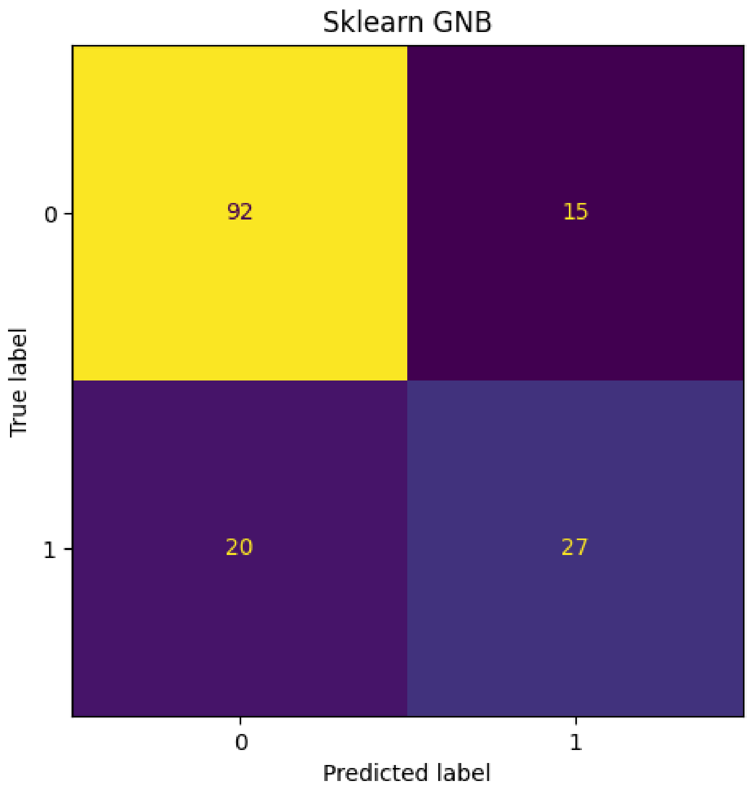
6. Model Comparison With Decision Tree and Logistic Regression

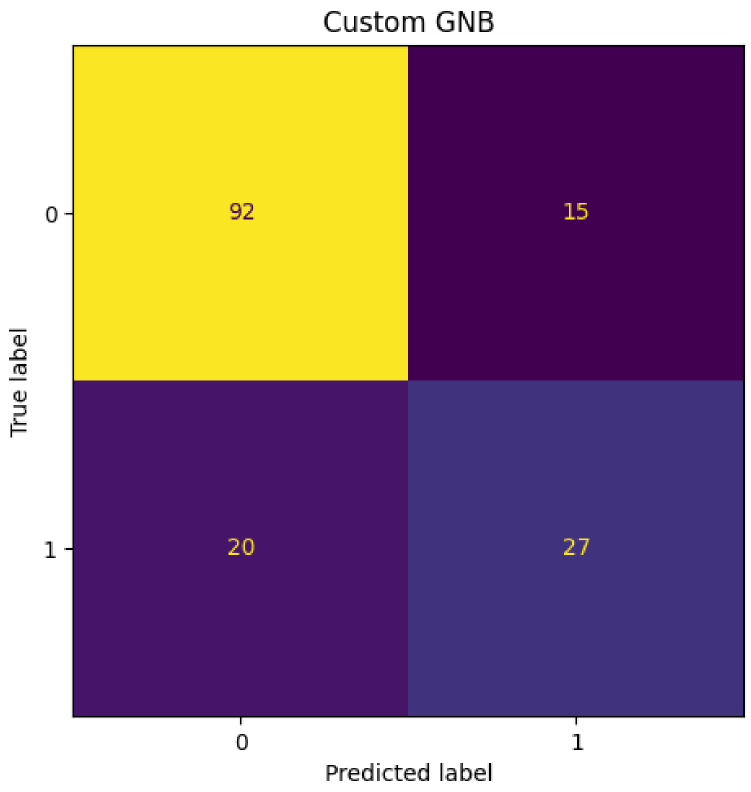
In [12]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
```

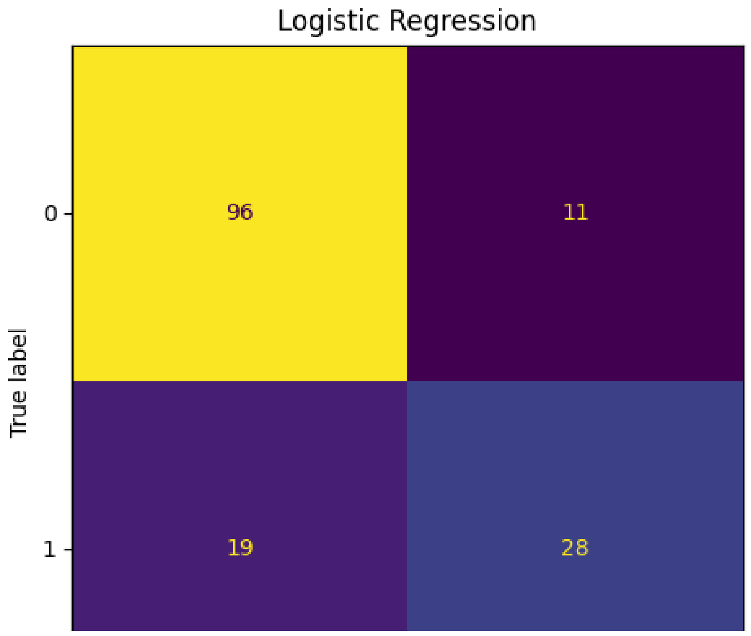
```
# Logistic Regression
log_model = LogisticRegression(max_iter=1000)
log_model.fit(X_train, y_train.ravel())
log_y_pred = log_model.predict(X_test)

# Decision Tree
dt_model = DecisionTreeClassifier(random_state=0)
dt_model.fit(X_train, y_train.ravel())
dt_y_pred = dt_model.predict(X_test)
```

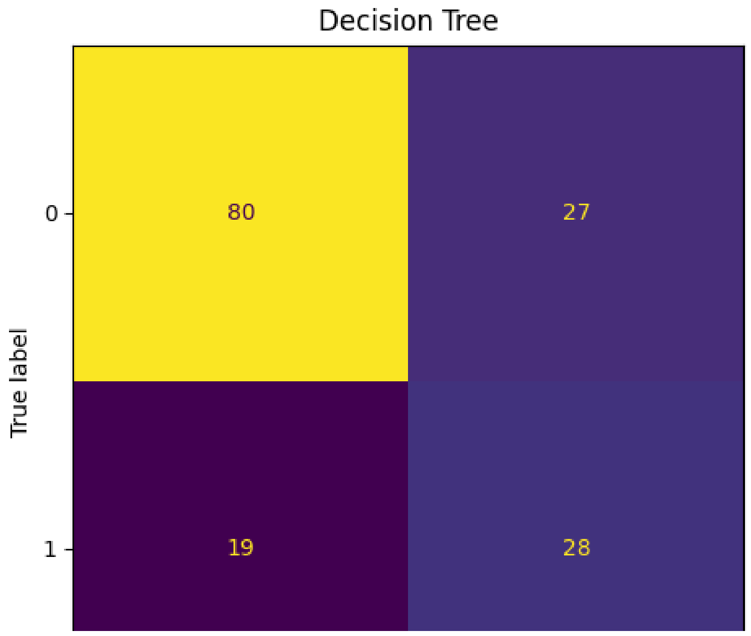
In [13]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

y_true = np.ravel(y_test)

models = [
    ("Sklearn GNB", Sklearn_y_pred),
    ("Custom GNB", Custom_y_pred),
    ("Logistic Regression", log_y_pred),
    ("Decision Tree", dt_y_pred)
]

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

for ax, (name, y_pred) in zip(axes.ravel(), models):
    cm = confusion_matrix(y_true, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot(ax=ax, colorbar=False)
    ax.set_title(name)

plt.tight_layout()
plt.show()
```


Sklearn GNB
92
15
20
27
True label
Predicted label
0
1
0
1


Custom GNB
92
15
20
27
True label
Predicted label
0
1
0
1


Logistic Regression
96
11
19
28
True label
0
1


Decision Tree
80
27
19
28
True label
0
1

In [14]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

metrics_dict = {}

for name, y_pred in models:
    metrics_dict[name] = {
        "Accuracy": accuracy_score(y_true, y_pred),
        "Precision": precision_score(y_true, y_pred),
        "Recall": recall_score(y_true, y_pred),
        "F1 Score": f1_score(y_true, y_pred)
    }

metrics_df = pd.DataFrame(metrics_dict).T
print("Model Performance Comparison:")
display(metrics_df)
```

```
Model Performance Comparison:
```

	Accuracy	Precision	Recall	F1 Score
Sklearn GNB	0.772727	0.642857	0.574468	0.606742
Custom GNB	0.772727	0.642857	0.574468	0.606742
Logistic Regression	0.805195	0.717949	0.595745	0.651163
Decision Tree	0.701299	0.509091	0.595745	0.549020

Based on the test set evaluation, Logistic Regression achieves the best overall performance among all models, with the highest accuracy, precision, recall, and F1 score. The Gaussian Naive Bayes models outperform the Decision Tree in terms of accuracy and F1 score but do not surpass Logistic Regression on this dataset. This suggests that while GNB is effective and computationally efficient, it does not provide the best predictive performance for this specific diabetes classification task.

4. References

[1] Scikit-learn (n.d.) *Naive Bayes*. Available at:
https://scikit-learn.org/stable/modules/naive_bayes.html
(Accessed: 5 December 2025).

[2] GeeksforGeeks (n.d.) *Naive Bayes Classifiers*. Available at:
https://www.geeksforgeeks.org/machine-learning/naive-bayes-classifiers/
(Accessed: 5 December 2025).

[3].Kaggle (n.d.) *Diabetes Dataset*. Available at:
https://www.kaggle.com/datasets/hashemi221022/diabetes/data
(Accessed: 5 December 2025).