Implementing Gaussian Naive Bayes with Coronary Heart Disease Data

December 13, 2024

Roshan Parikh, Wali Siddiqui, Danielle Whisnant, Rio Wombacher

GitHub: https://github.com/roshanparikh/GNBforCoronaryHeartDisease.git

1 Gaussian Naive Bayes Classification (GNB)

Gaussian Naive Bayes (GNB) is a generative classification model. It assumes that the data for each feature is conditionally independent given the class label and that these features follow Gaussian (normal) distributions. Unlike standard Naive Bayes, which may handle discrete features, GNB specifically models continuous features using the normal distribution. The model makes predictions by combining probabilities from all features for each class. For a given input, the probabilities are calculated for each class, and the final classification is assigned to the class with the highest posterior probability. Formal equation:

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

1.1 Key Assumptions and Applications

Due to our assumption that our features come from normal distributions, Gaussian Naive Bayes Classification is best suited for data with exclusively continuous variables, and where the features are not strongly correlated. This model also has some shortcomings, namely, it is vulnerable to datasets with outlier values that may greatly affect the mean and variance of the data, and large complex datasets where more complex models will typically perform better.

1.2 Model Parameter Estimation

Gaussian Naive Bayes, being a **generative model**, does not use an optimizer function. Instead, it capitalizes on the naive assumption that our data comes from conditional independent normal distributions and uses **closed-form Maximum Likelihood Estimation (MLE)** to estimate parameters. MLE determines the parameters $ _y, ^2_y, P(y) $ that maximize the likelihood of the observed data. This is equivalent to minimizing the log loss. Formally:

$$\arg\min_\theta \sum_{i=1}^{m} -\log\left[P_\theta(x_i, y_i)\right]$$

1.2.1 Parameters Estimated:

- **Class Priors:** $ P(y) $, the proportion of observations in each class.
- **Feature Means:** $ _y $, the mean of each feature $ x_i $ given class $ y $.
- **Feature Variances:** $ ^2_y $, the variance of each feature $ x_i $ given class $ y $.

  Note: Unlike other Naive Bayes classifiers, GNB does not use Laplace smoothing because it works with continuous features. Instead, **variance smoothing** is applied by adding a very small constant (e.g., $ 10^{-6} $) to the variance to avoid instability when variance approaches zero.

1.3 Prediction Process

Now to make our predictions, we utilize our assumption that our data comes from normal distributions to calculate our predicted probabilities for each class. We calculate the predicted probabilities for each class $ y $ using the conditional probabilities for each feature $ x_i $, assuming normal distributions:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right) \Rightarrow \log P(x_i, y) = -\frac{1}{2}\log(2\pi\sigma_y^2) - \frac{(x_i-\mu_y)^2}{2\sigma_y^2}$$

1.3.1 Steps:

1. **Compute Conditional Probabilities:** For each feature $ x_i $ and class $ y $, calculate $ P(x_i y) $ using the Gaussian probability density function.

2. **Get Joint Probabilities:** For each each class in $ y $, compute the joint probability: $\prod_{i=1}^{d} P(x_i \mid y)$

3. **Calculate Postiers:** Multiply our joint probabilities by the priors, then use logarithms for ease of computation:

$$P(y \mid x) \propto P(y)\prod_{i=1}^{d} P(x_i \mid y) \Rightarrow \log(Py \mid x) \propto \log P(y) + \log\sum_{i=1} P(x_i \mid y)$$

4. **Normalize Probabilities:** Convert the joint probabilities into valid probabilities by normalizing them to sum to 1.

5. **Assign Class:** Select the class $ y $ with the highest posterior probability as the prediction.

1.4 Evaluation

The performance of Gaussian Naive Bayes can be evaluated using standard classification metrics, such as: - **Accuracy** - **Precision/Recall** - **Log Loss**

For this explanation, **accuracy** is used as the primary evaluation metric.

```
[1]: import pandas as pd
     import numpy as np

     class GaussianNaiveBayes(object):
         """Gaussian Naive Bayes model

         @attributes:
             n_classes: number of classes
                 for our dataset, we will have 2 classes (yes heart disease and no␣
      ↪heart disease)
             class_means: NumPy array of the means for each class
             class_vars: NumPy array of the variances for each class
             label_priors: NumPy array of the priors distribution (1-D array)
         """

         def __init__(self, n_classes):
             '''
             Notes here
             '''
             self.n_classes = n_classes
             self.label_priors = None
             self.class_means = None
             self.class_vars = None

         def train(self, X_train, y_train, alpha=1e-9):
             '''
             Trains the model. Calculates label priors. Calculates mean, and␣
      ↪variance for each feature for each class.
             @params:
                 X_train: a 2D (n_examples x n_attributes) numpy array
                 y_train: a 1D (n_examples) numpy array
                 alpha: adjustment for variance (scalar)
             @return (to be used for unit tests):
                 label_priors: a 1-D numpy array with the prior distribution for␣
      ↪each class
                 class_means: a 1-D numpy array with the means for each class
                 class_vars: a 1-D numpy array with the variances for each class
             '''
             # Input Validation
             if not isinstance(X_train, np.ndarray):
                 raise TypeError("X_train must be a NumPy array.")
             if not isinstance(y_train, np.ndarray):
                 raise TypeError("y_train must be a NumPy array.")
             if X_train.shape[0] != y_train.shape[0]:
                 raise ValueError("Number of samples in X_train and y_train must be␣
      ↪equal.")
             if X_train.size == 0:
```

```
            raise ValueError("X_train is empty.")
        if y_train.size == 0:
            raise ValueError("y_train is empty.")

        # Number of features
        self.n_attributes = X_train.shape[1]

        # Creating empty arrays of length n_classes to store class means and␣
↪variances
        class_means = np.zeros([self.n_classes, self.n_attributes])
        class_vars = np.zeros([self.n_classes, self.n_attributes])

        # Calculating the prior probability: P(y)
        label_priors = np.bincount(y_train, minlength=self.n_classes)/
↪len(y_train)

        for n_class in range(self.n_classes):
            # Getting X examples for specific classes
            X_class = X_train[y_train == n_class]

            if X_class.size == 0:
                # Assign a small variance (alpha) to prevent computational␣
↪issues if no data for a class
                class_means[n_class] = np.zeros(self.n_attributes)
                class_vars[n_class] = np.full(self.n_attributes, alpha)
            else:
                # Calculating mean (mu) and variance (sigma^2)
                class_means[n_class] = (np.mean(X_class, axis = 0))
                class_vars[n_class] = (np.var(X_class, axis = 0))

        class_vars += alpha

        self.label_priors = label_priors
        self.class_means = class_means
        self.class_vars = class_vars

        return label_priors, class_means, class_vars


    def predict(self, X_test):
        '''
        @params:
            X_test: 2-D array
        @return:
            predictions: 1-D array of length X_test
        '''
        predictions = np.zeros(X_test.shape[0])
        # Log(P(y))
        log_priors = np.log(self.label_priors)

        for i,x in enumerate(X_test):
            # P(y|x); np array of length n_classes
            posteriors = np.zeros_like(log_priors)

            for n_class in range(self.n_classes):
                log_prior = log_priors[n_class]
                mean = self.class_means[n_class]
                var = self.class_vars[n_class]

                log_likelihood = self.log_likelihood_func(x, mean, var)

                # Calculates posterior distribution for this example x
                posteriors[n_class] = (log_prior + log_likelihood)

            # Predicts the class by determining the index (class) with the␣
↪highest posterior probability
            predictions[i] = np.argmax(posteriors)

        return predictions


    def log_likelihood_func(self, x, mean, var):
        '''
        @params:
            x: 1-D array (example)
            mean: scalar
            var: scalar
        @returns:
            log likelihood for example
        '''

        epsilon = 1e-10
        var = np.maximum(var, epsilon)
        log_likelihood = -0.5 * np.sum(
        np.log(2 * np.pi * var) +  # log of variance term
        ((x - mean) ** 2) / var    # squared difference normalized by variance
        )

        return log_likelihood


    def accuracy(self, X_test, y_test):
        '''
        Calculate 0-1 loss over predictions.
        @params:
            X_test: 2D array (n_examples x n_attributes) where each row is an␣
 ↪example and each column is a feature/attribute
            _test: 1D array where each entry corresponds to the label for a row␣
 ↪in X
        @return:
            0-1 loss for the input data and associated labels
        '''
        predictions = self.predict(X_test)
        return np.mean(predictions == y_test)
```

1.5 Check Model

[2]:
```
#Testing base cases
import pytest
#Set random seed for reproducibility
np.random.seed(0)

# Creates Test Models with 2 classes (yes heart disease and no heart disease)
test_model1 = GaussianNaiveBayes(2)
test_model2 = GaussianNaiveBayes(2)

#Creates Data for test_model1 - base case
x1 = np.array([
    [1.0, 2.1, 3.2],
    [1.1, 2.0, 3.1],
    [1.2, 2.2, 3.0],
    [4.0, 5.1, 6.2],
    [4.1, 5.0, 6.1]
])
y1 = np.array([0, 0, 0, 1, 1])
x_test1 = np.array([
    [1.05, 2.05, 3.05],
    [4.05, 5.05, 6.05],
    [2.5, 3.5, 4.5]
])
y_test1 = np.array([0, 1, 0])

#Creates data for test_model2 - base case
x2 = np.array([
    [2.0, 3.1, 4.2],
    [2.1, 3.0, 4.1],
    [2.2, 3.2, 4.0],
    [5.0, 6.1, 7.2],
    [5.1, 6.0, 7.1],
    [5.2, 6.2, 7.0]
```

```
])
y2 = np.array([0, 0, 0, 1, 1, 1])
x_test2 = np.array([
    [2.05, 3.05, 4.05],
    [5.05, 6.05, 7.05],
    [3.5, 4.5, 5.5]
])
y_test2 = np.array([0, 1, 0])

# Test Models
def check_train_dtype(model, label_priors, class_means, class_vars, x_train):
    assert isinstance(class_means, np.ndarray)
    assert class_means.ndim == 2 and class_means.shape == (model.n_classes,␣
 ↪x_train.shape[1])
    assert isinstance(class_vars, np.ndarray)
    assert class_vars.ndim == 2 and class_vars.shape == (model.n_classes,␣
 ↪x_train.shape[1])
    assert isinstance(label_priors, np.ndarray)
    assert label_priors.ndim == 1 and label_priors.shape == (model.n_classes,)

# Test Model 1 training
label_priors1, class_means1, class_vars1 = test_model1.train(x1, y1)
check_train_dtype(test_model1, label_priors1, class_means1, class_vars1, x1)

# Expected values for Model 1 (calculated manually)
expected_label_priors1 = np.array([0.6, 0.4])
expected_class_means1 = np.array([
    [1.1, 2.1, 3.1],  # Mean of class 0
    [4.05, 5.05, 6.15]  # Mean of class 1
])
expected_class_vars1 = np.array([
    [0.0066667, 0.0066667, 0.0066667],  # Variance of class 0
    [0.0025, 0.0025, 0.0025]  # Variance of class 1
])

assert np.allclose(label_priors1, expected_label_priors1, atol=1e-2)
assert np.allclose(class_means1, expected_class_means1, atol=1e-2)
assert np.allclose(class_vars1, expected_class_vars1, atol=1e-2)

# Test Model 2 training
label_priors2, class_means2, class_vars2 = test_model2.train(x2, y2)
check_train_dtype(test_model2, label_priors2, class_means2, class_vars2, x2)

# Expected values for Model 2
expected_label_priors2 = np.array([0.5, 0.5])
expected_class_means2 = np.array([
    [2.1, 3.1, 4.1],  # Mean of class 0
    [5.1, 6.1, 7.1]  # Mean of class 1
])
expected_class_vars2 = np.array([
    [0.0066667, 0.0066667, 0.0066667],  # Variance of class 0
    [0.0066667, 0.0066667, 0.0066667]  # Variance of class 1
])

assert np.allclose(label_priors2, expected_label_priors2, atol=1e-2)
assert np.allclose(class_means2, expected_class_means2, atol=1e-2)
assert np.allclose(class_vars2, expected_class_vars2, atol=1e-2)

# helper to confirm data types and shapes of predictions
def check_test_dtype(predictions, x_test):
    assert isinstance(predictions, np.ndarray)
    assert predictions.ndim == 1 and predictions.shape == (x_test.shape[0],)

# Test Model 1 predictions
predictions1 = test_model1.predict(x_test1)
check_test_dtype(predictions1, x_test1)
expected_predictions1 = y_test1
assert np.array_equal(predictions1, expected_predictions1)

# Test Model 2 predictions
predictions2 = test_model2.predict(x_test2)
check_test_dtype(predictions2, x_test2)
expected_predictions2 = y_test2
assert np.array_equal(predictions2, expected_predictions2)

# Test Model 1 accuracy
accuracy1 = test_model1.accuracy(x_test1, y_test1)
assert accuracy1 == pytest.approx(1.0, 0.01)

# Test Model 2 accuracy
accuracy2 = test_model2.accuracy(x_test2, y_test2)
assert accuracy2 == pytest.approx(1.0, 0.01)

#Test Model 1 log-likelihood
for idx, x in enumerate(x_test1):
    for n_class in range(test_model1.n_classes):
        mean = class_means1[n_class]
        var = class_vars1[n_class]

        # Compute expected log likelihood manually
        adjusted_var = np.maximum(var, 1e-10)
        expected_log_likelihood = -0.5 * np.sum(np.log(2 * np.pi *␣
 ↪adjusted_var) + ((x - mean) ** 2) / adjusted_var)
```

```
        # Compute log likelihood using the model's method
        computed_log_likelihood = test_model1.log_likelihood_func(x, mean, var)
        assert np.isclose(computed_log_likelihood, expected_log_likelihood,␣
 ↪atol=1e-6)

#Test Model 2 log-likelihood
for idx, x in enumerate(x_test2):
    for n_class in range(test_model2.n_classes):
        mean = class_means2[n_class]
        var = class_vars2[n_class]

        # Compute expected log likelihood manually
        adjusted_var = np.maximum(var, 1e-10)
        expected_log_likelihood = -0.5 * np.sum(np.log(2 * np.pi *␣
 ↪adjusted_var) + ((x - mean) ** 2) / adjusted_var)

        # Compute log likelihood using the model's method
        computed_log_likelihood = test_model2.log_likelihood_func(x, mean, var)
        assert np.isclose(computed_log_likelihood, expected_log_likelihood,␣
 ↪atol=1e-6)



#Testing Edge Cases:
#1. Tests Train on an empty feature set
model_emp = GaussianNaiveBayes(2)
X_train_emp = np.array([]).reshape(0, 3)  # 0 samples, 3 features
y_train_emp = np.array([], dtype=int)
with pytest.raises(ValueError):
    model_emp.train(X_train_emp, y_train_emp)

#2. Tests model with zero variance
model_var = GaussianNaiveBayes(2)
X_train_var = np.array([
    [1.0, 2.0],
    [1.0, 2.0],
    [1.0, 2.0],
    [3.0, 4.0],
    [3.0, 4.0],
    [3.0, 4.0]
    ])
y_train_var = np.array([0, 0, 0, 1, 1, 1])
label_priors_var, class_means_var, class_vars = model_var.train(X_train_var,␣
 ↪y_train_var)

expected_class_vars = np.array([
    [0.0, 0.0],  # Zero variance for class 0
    [0.0, 0.0]   # Zero variance for class 1
    ])

assert np.allclose(class_vars, expected_class_vars, atol=1e-9)

X_test_var = np.array([
    [1.0, 2.0],
    [3.0, 4.0],
    [2.0, 3.0]])

predictions_var = model_var.predict(X_test_var)
expected_predictions_var = np.array([0, 1, 0])
assert np.array_equal(predictions_var, expected_predictions_var)

#3. Tests model with more than two classes
model3 = GaussianNaiveBayes(3)
X_train3 = np.array([
        [1.0, 2.0],
        [1.1, 2.1],
        [1.2, 2.2],
        [3.0, 4.0],
        [3.1, 4.1],
        [3.2, 4.2],
        [5.0, 6.0],
        [5.1, 6.1],
        [5.2, 6.2]
    ])
y_train3 = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2], dtype=int)
model3.train(X_train3, y_train3, alpha=1e-9)

X_test3 = np.array([
        [1.05, 2.05],
        [3.05, 4.05],
        [5.05, 6.05]])
y_test3 = np.array([0, 1, 2], dtype=int)

predictions3 = model3.predict(X_test3)
expected_predictions3 = np.array([0, 1, 2])
assert np.array_equal(predictions3, expected_predictions3)

#4. Tests model when training data only has one single class
modela = GaussianNaiveBayes(n_classes=2)
X_traina = np.array([
        [1.0, 2.0],
        [1.1, 2.1],
        [1.2, 2.2]
    ])
```

```
y_traina = np.array([0, 0, 0], dtype=int)  # Only class 0

label_priorsa, class_meansa, class_varsa = modela.train(X_traina, y_traina,␣
↪alpha=1e-9)
check_train_dtype(modela, label_priorsa, class_meansa, class_varsa, X_traina)

expected_label_priorsa = np.array([1, 0])
expected_class_meansa = np.array([
        [1.1, 2.1],    # Mean of class 0
        [1e-9, 1e-9]    # Mean of class 1 (adjusted by alpha)
    ])
expected_class_varsa = np.array([
        [0.0066667, 0.0066667],  # Variance of class 0
        [1e-9, 1e-9]              # Variance of class 1 (adjusted by alpha)
    ])
assert np.allclose(label_priorsa, expected_label_priorsa, atol=1e-2)
assert np.allclose(class_meansa, expected_class_meansa, atol=1e-2)
assert np.allclose(class_varsa, expected_class_varsa, atol=1e-2)
```

1.6 Main

[3]:
```
#Testing the dataset using SKlearn
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import LabelEncoder

#Our Model splitting the data the same way done in SKlearn
df = pd.read_csv('processed_cleveland.csv')

df['num'] = (df['num'] > 0).astype(int)

# Categorical features to encode
categorical_features = ['cp', 'restecg', 'slope', 'thal', 'ca']

for feature in categorical_features:
    df[feature] = pd.Categorical(df[feature]).codes

# Remove any rows with NaN values
df = df.dropna()

# Extract features and target
X = df.drop(columns=['num'])
y = df['num']
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=48)
```

```python
# Initialize and train the Gaussian Naive Bayes classifier
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Make predictions
y_pred = gnb.predict(X_test)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Sklearn Gaussian Naive Bayes Accuracy: {accuracy * 100:.2f}%")
```

```
Sklearn Gaussian Naive Bayes Accuracy: 85.25%
```

[4]:
```python
#Testing our model using train_test_split (as done in SKlearn above)
X_train, X_test, y_train, y_test = train_test_split(X.values, y.values,␣
 ↪test_size=0.2, random_state=48)

gnb = GaussianNaiveBayes(n_classes=2)

gnb.train(X_train, y_train, alpha=1e-9)

accuracy = gnb.accuracy(X_test, y_test)

print(f"Accuracy of our model splitting like the dataset as done in sklearn:␣
 ↪{accuracy * 100:.2f}%")
```

```
Accuracy of our model splitting like the dataset as done in sklearn: 85.25%
```

[5]:
```python
# Manual 80/20 split in SKlearn
train_size = int(0.8 * len(X))
X_train = X.values[:train_size]
y_train = y.values[:train_size]
X_test = X.values[train_size:]
y_test = y.values[train_size:]

sklearn_gnb = GaussianNB()
sklearn_gnb.fit(X_train, y_train)
sklearn_accuracy = sklearn_gnb.score(X_test, y_test)
print(f"Scikit-learn Gaussian Naive Bayes Accuracy using 80/20 rule:␣
 ↪{sklearn_accuracy * 100:.2f}%")
```

```
Scikit-learn Gaussian Naive Bayes Accuracy using 80/20 rule: 73.77%
```

[6]:
```python
#80/20 Split Using Our Methods
gnb = GaussianNaiveBayes(n_classes=2)

alpha = 1e-9
gnb.train(X_train, y_train, alpha)
```

```
accuracy = gnb.accuracy(X_test, y_test)

print(f"Accuracy of our model splitting like the dataset using the 80/20 rule:␣
↪{accuracy * 100:.2f}%")
```

```
Accuracy of our model splitting like the dataset using the 80/20 rule: 73.77%
```

1.7 Comparing Models

To evaluate the performance of our Gaussian Naive Bayes model, we compared it to the GaussianNB implementation from scikit-learn, using a dataset containing features relevant to diagnosing cardiovascular issues. The dataset used in this evaluation is a heart disease dataset, which contains various clinical and diagnostic features used to predict the presence or absence of coronary heart disease in patients. It consists of both continuous and categorical attributes that are essential for medical decision-making. The information on this dataset and testing came from "Machine-Learning-Based Prediction Models of Coronary Heart Disease Using Naïve Bayes and Random Forest Algorithms," which used the Cleveland Database from the UCI Repository (Bemando, et al.).

Both models rely on the assumption that continuous features are distributed according to a Gaussian (normal) distribution, which aligns the theoretical underpinnings of our implementation with those of scikit-learn.

We began by preprocessing the dataset to ensure it was in a format suitable for analysis. Non-numeric categorical features were converted into numeric representations where necessary, and any missing values were addressed. The dataset was then split into training and test subsets to evaluate model performance consistently.

For scikit-learn's model, we utilized the GaussianNB class, which provides a ready-made implementation of the Gaussian Naive Bayes algorithm. The model was trained on the same training data as our custom implementation. After training, both models were tested on the same test set to predict the target labels.

The accuracy of each model was calculated by comparing predicted labels to the actual test set labels. This allowed us to directly compare the performance of our implementation with scikit-learn's. We compared our model with sklearn's model in two ways. The first was by splitting the dataset for both our implementation and scikit-learn's using their built-in train_test_split method. This function randomly shuffles the dataset before splitting, ensuring that the training and testing sets are representative of the overall data distribution. The random_state parameter ensures the shuffle is reproducible. We selected random_state=48 for this to ensure our results were consistent. When doing this for both our model and scikit-learn's with random splitting, we get an accuracy of 85.25% for both models. This is also consistent with the paper referenced for testing, which is getting an accuracy of 85% for their Gaussian Naive Bayes model. When we do this through a more brute-force method without shuffling, we get an accuracy of 73.77% for both of them.

This comparison ensured that our model was correctly capturing the Gaussian Naive Bayes principles and performing on par with the well-established library implementation of scikit-learn.

1.8 References

Baladram, S. (2024) Gaussian Naive Bayes, Explained: A Visual Guide with Code Examples for Beginners, Medium. Available at: https://towardsdatascience.com/gaussian-naive-bayes-explained-a-visual-guide-with-code-examples-for-beginners-04949cef383c (Accessed: 10 December 2024).

Bemando, Miranda, and Aryuni (2021) ‘Machine-Learning-Based Prediction Models of Coronary Heart disease Using Naïve Bayes and Random Forest Algorithms,’ IEEE Available at: https://ieeexplore.ieee.org/document/9537060 (Accessed 12 December 2024).

Gaussian Naive Bayes (2023) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/gaussian-naive-bayes/(Accessed: 10 December 2024).

GaussianNB (no date) scikit-learn. Available at: https://scikit-learn/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html (Accessed: 10 December 2024).

Kashishdafe (2024) ‘Gaussian Naive Bayes: Understanding the Basics and Applications’, Medium, 23 March. Available at: https://medium.com/@kashishdafe0410/gaussian-naive-bayes-understanding-the-basics-and-applications-52098087b963 (Accessed: 10 December 2024).

‘Naive Bayes classifier’ (2024) Wikipedia. Available at: https://en.wikipedia.org/w/index.php?title=Naive_Bayes_
(Accessed: 10 December 2024).