Naive Bayes Classifier From Scratch

This notebook demonstrates how to create, fit, and evaluate a *Naive Bayes Classifier* from scratch in Python. It explains the background and implementation to provide an understanding of the use cases of a Naive Bayes Classifier and how they work.

Background

A **Naive Bayes Classifier** is a very simple machine learning algorithm used for *classification*, or predicting the class of a given data point. It is based on the *Bayes Theorem*, an equation that updates the probability of a result as more information becomes available.

Bayes' Theorem

The Bayes Theorem defines the following formula:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where:

- A and B are events
- $P(B) \neq 0$ to ensure the denominator is valid
- $P(A|B)$ is the *posterior probability* or the probability of A happening if B is true
- $P(B|A)$ is the *likelihood* or the probability of seeing B if A is true
- $P(A)$ is the *prior probability* of A or the belief before seeing evidence
- $P(B)$ is the *marginal probability* of B or the total probability of observing B

The Naive Assumption

Naive Bayes assumes that every feature is unrelated to the others as long as the class label is known. This is represented mathematically as:

$$P(x_1, x_2, \ldots, x_n|y) = P(x_1|y) \cdot P(x_2|y) \cdot \ldots \cdot P(x_n|y)$$

While this assumption of independence is rarely true in the real world, it has several use cases, such as

- sentiment analysis
- text classification
- image recognition

Or real world examples including:

- Fraud Detection
- Medical Diagnosis
- Content Recommendation

Types of Naive Bayes Classifiers

Depending on the type of input data, a different variant of Naive Bayes is used. The three most common types are:

- **Gaussian**: assuming the features are distributed normally
- **Multinomial**: assuming features are counts or frequencies
- **Bernoulli**: assuming features are binary or boolean

This notebook demonstrates how to create a Gaussian Naive Bayes Classifier and fit it on synthetic data.

Step 1: Data Generation

The code below creates the `generate_data` function with the following parameters:

- `n_samples` : the number of samples or data points to generate
- `n_features` : the number of features or columns each sample has
- `n_classes` : the number of classes or categories

This function generates a synthetic dataset we can use to train our model on. You can change the length and complexity of the dataset by calling it with different values. The function does the following to generate data for each class:

- the mean of each feature is randomly selected
- variability is added using a diagonal covariance matrix
- random data points around the mean are generated
- noise is added to the samples
- samples are labeled based on class

After data is generated for each class, the function returns the features as `X` and the labels as `y` .

In [1]:
```
import numpy as np # import NumPy for math and arrays

def generate_data(n_samples=1000, n_features=3, n_classes=3):
    X = [] # store features
    y = [] # store labels

    # generate for each class
    for i in range(n_classes):
        mean = np.random.uniform(-5, 5, size=n_features)              # random mean
        covariance = np.eye(n_features) * np.random.uniform(0.5, 1.5) # randomly sc
```

```
        # generate samples
        samples = np.random.multivariate_normal(mean, covariance, size=n_samples //
        noise = np.random.normal(0, 0.1, samples.shape) # calculate noise to add
        samples += noise                                # add noise
        X.append(samples)                               # append to features
        y.extend([i] * (n_samples // n_classes))        # add labels
    return np.vstack(X), np.array(y)                    # return data
```

We also want to be able to train and test our model using different data. To do this, we will define the function `split_data`, to split our features and labels into train and test sets. The function takes `X` and `y` as inputs as well as `test_size`, used to determine the amount of samples in the testing data.

In [2]:
```
def split_data(X, y, test_size=0.2):
    # sample indices
    indices = [i for i in range(len(X))]        # list indices corresponding to sam
    np.random.shuffle(indices)                  # shuffle indices for random split
    # where to split data
    split_index = int(len(X) * (1 - test_size)) # where to split the data
    train_index = indices[:split_index]         # select indices for training set
    test_index = indices[split_index:]          # select indices for testing seu
    # train samples
    X_train = X[train_index]                    # get training features
    y_train = y[train_index]                    # get training labels
    # test samples
    X_test = X[test_index]                      # get testing features
    y_test = y[test_index]                      # get testing labels
    return X_train, X_test, y_train, y_test     # return train/test set
```

Step 2: Model Fitting

Naive Bayes classifiers are trained differently than regression models. Instead of optimizing parameters over many iterations, they learn instantly by summarizing the training data by calculating various statistics for each class.

The `fit` function is defined below to train our model. It starts by identifying each unique class and initializing a dict to store class statistics. It then iterates through the classes doing the following for each one:

- seperate samples labeled as the current class
- calculate the mean and variance for each feature
- calculate the prior probability, or the likelihood of the label before looking at the data
- save the classes statistics to the dictionary

Finally, the function returns the dictionary containing each classes summary.

In [3]:
```
def fit(X, y):                                  # function to fit model on data
    classes = np.unique(y)                      # get each unique class
    stats = {}                                  # dict to store stats for each c
    for i in classes:                           # iterate through each class
```

```
        class_X = X[y == i]                          # get all samples with the class
        stats[i] = {                                 # add stats for class
            "mean": class_X.mean(axis=0),            # average of each feature in cla
            "var": class_X.var(axis=0),              # variance of each feature in cl
            "prior": class_X.shape[0] / X.shape[0] # estimated probability of class
        }
    return stats                                     # return data stats
```

Step 3: Probability Density Function

Probability density functions are used to calculate the likelihood of a feature value given a class. Since our data is Gaussian, or distributed normally, we use the Gaussian Probability Density Function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{1}$$

Where:

- x is the feature we are evaluating
- μ is the average of that feature for a given class
- σ is the standard deviation of that feature for the class

In our implementation, we used variance instead of standard deviation and included an epsilon, `eps`, or a very small value just larger than zero, added to avoid division by zero. The function begins by calculating the numerator, then the denominator. Finally, it returns the quotient of the numerator and denominator.

In [4]:
```
def pdf(x, mean, var):                                  # compute probability den
    eps = 1e-6                                          # epsilon to avoid divisi
    numerator = np.exp(- (x - mean)**2 / (2 * var + eps)) # calculate numerator of
    denominator = np.sqrt(2 * np.pi * var + eps)        # calculate pdf denominat
    return numerator / denominator                      # return probability dens
```

Step 4: Predict

A Naive Bayes model classifies, or predicts, new samples by calculating the **Posterior Probability** for each class. This is done by combining:

- The *Prior Probability* of each class, or how likely it is overall, with
- The *Likelihood* of the features for that class using the probability density function

Below, we wrote the `predict` function with inputs `X` for the samples we want to classify and `stats` for the datasets summary we determined using the `fit` function.

For each sample in `X`, it does the following:

- calculates the log of the prior for each class

- calculates the log likelihood of the features for each class
- combines the log likelihoods of each feature
- calculates the log-posterior probability by taking the sum of the prior and likelihood
- selects the class with the highest log-posterior probability for the prediction

After this is done, the function returns an array of the predicted classes of `X`.

In [5]:
```
def predict(X, stats):                                                  # predict c
    predictions = []                                                    # list to s
    for i in X:                                                         # iterate t
        posteriors = []                                                 # store pos
        for j, params in stats.items():                                 # iterate t
            prior = np.log(params["prior"])                             # log of cl
            probability_density = pdf(i, params["mean"], params["var"]) # calculate
            likelihood = np.sum(np.log(probability_density))            # calculate
            posteriors.append(prior + likelihood)                       # calculate
        predictions.append(np.argmax(posteriors))                       # predict c
    return np.array(predictions)
```

Step 5: Using the Code

Now we can use our code to create and fit our model.

First, we generate and split our data:

In [6]:
```
# Generate data
X, y = generate_data()
X_train, X_test, y_train, y_test = split_data(X, y)
```

Next, we can fit the model on the data

In [7]:
```
# fit
model_stats = fit(X_train, y_train)
```

Then, we predict our testing data

In [8]:
```
# Predict
predictions = predict(X_test, model_stats)
```

Finally, we can evaluate the model

In [9]:
```
# Accuracy
accuracy = np.mean(predictions == y_test)
print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.96
```

Summary

Background

- Naive Bayes is a classifier based on the *Bayes Theorem*
- It assumes features are independent of one another given the class
- Used for text classification, sentiment analysis, and images detection

Types of Naive Bayes

- *Gaussian* or normally distributed features
- *Multinomial* or frequency based features
- *Bernoulli* or binary features

Implementation

- *Data*: synthetically generate a Gaussian dataset
- *Fitting*: calculate mean, variance, and prior probability for each class
- *Prediction*: calculate posterior probability using the gaussian probability density function
- *Evaluation*: measure models performance using accuracy

Author and Liscense

Kaggle | Github