Introduction and Application of Likelihood Estimation and Naive Bayes Classifier

Anqiao Auyeung

October 29, 2023

Introduction

One of the most widely used and effective methods for classification tasks is the Naive Bayes classifier. The underlying principle of Naive Bayes is the application of Bayes' Theorem, with the key assumption that the features of the data are conditionally independent given the class label. This simplicity allows it to perform efficiently, even on complex datasets.

At the core of the Naive Bayes classifier is the concept of likelihood estimation. In the case of Naive Bayes, likelihood estimation helps to determine the probability distribution of the features conditioned on the class, which in turn aids in the classification process. This is an overview of the concepts of likelihood estimation and the Naive Bayes classifier. We will explore their theoretical foundations and their implementation.

This article assumes a basic understanding of middle school mathematics and programming logic, with a focus on Haskell for implementation. However, prior knowledge of derivatives, calculus, and linear algebra is recommended for a deeper understanding of the material. For those less familiar with these concepts, it may be useful to review them before proceeding.

Oh, and by the way, this is a rough draft that yours truly (Anqiao Ouyang) was jotting down during my Chinese language class on October 28, 2023—You can blame my madness on the boring test-oriented middle school. They failed to restrain my genius brain!

1 Probability and Likelihood

1.1 Classical Probability

Classical probability is the most basic probability model in probability theory. It originated from studying probabilities through activities like coin tossing and dice rolling. Its characteristic lies in having a finite sample space, such as a coin with only two possibilities (heads or tails) or a six-sided die. Classical probability assumes that all possible outcomes are equally likely, with each outcome having an equal probability of occurring.

Calculating classical probability based on the frequency of event occurrences is straightforward. In the case of mutually exclusive events (i.e., events that cannot occur simultaneously), if we want to calculate the probability of event A, where the favorable outcomes for A are m and the total sample space size is n, then:

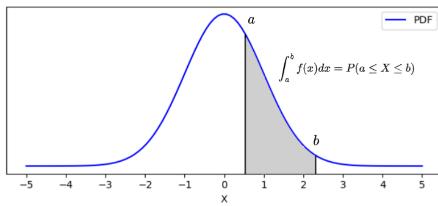
$$P(A) = \frac{m}{n}$$

This represents a typical prior probability, estimated based on previous experience or fundamental principles before considering any observations or evidence. In classical probability, the probability distribution is based on theoretical analysis of random experiments or events, without considering any new data or observational results.

1.2 Probability Distribution Function

Before delving into the likelihood function, it might be helpful to briefly understand some functions related to probability distributions.

We can describe the probability distribution of continuous random variables using Probability Density Function (PDF).



For example, the graph above illustrates the probability density function of a Gaussian distribution. If X is a continuous random variable, considering the probability density function $f(x)$, we can select a certain interval to represent the probability of the variable falling within that interval. We use definite integration to represent the area under the curve within a certain interval, expressed as:

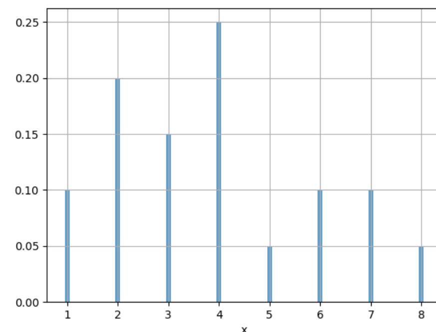
$$\int_a^b f(x)dx = P(a \leq X \leq b)$$

The probability density function possesses some fundamental properties. Firstly, it's non-negative over its entire domain. Secondly, it exhibits normalization, meaning the integral over the entire domain equals 1, i.e., $\int f(x)dx = 1$, as the total probability sums up to 1.

For discrete random variables, we can describe their probability distribution using Probability Mass Function (PMF). Unlike PDF, which describes the probability distribution of continuous random variables, Probability Mass Function directly represents the probability of each value. Unlike PDF, which requires specifying an interval and integrating to obtain the probability (because the probability is the area under the curve, which is 0 for a single point), PMF provides the probability of each individual value.

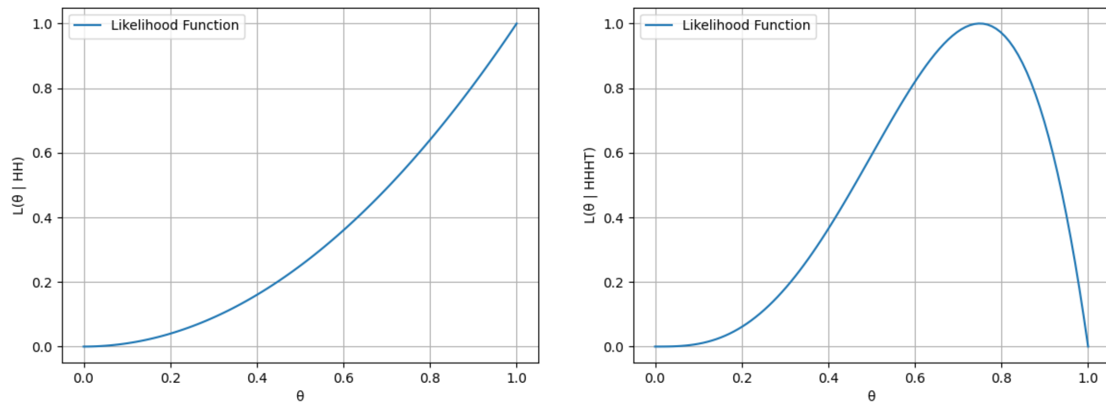
In the case of a Probability Mass Function (PMF) as shown in the graph, each discrete variable x corresponds to a probability, and their sum equals 1, i.e.,

$$\sum P(X = x) = 1$$



1.3 Likelihood Function

In less formal terms, likelihood might be confused with the concept of probability, but let's move past that now and understand it anew.



Probability, generally speaking, refers to the likelihood of an event occurring given a set of known parameters. Likelihood, on the other hand, pertains to the likelihood of model parameters given a set of data. For instance, consider a fair coin tossed twice resulting in two heads. What's the credibility of saying the probability of getting heads is 50

The likelihood function represents the distribution function of the likelihood concept concerning model parameters. Typically denoted as $L(\theta|X)$, where:

- L: represents the likelihood function - θ: represents the model parameters (e.g., 50- X: represents the observed data (e.g., "two heads" in the example)

If we denote H as heads and T as tails for a coin, observe the estimates for tossing two heads HH and tossing three heads and one tail $HHHT$ in the graph below:

We can observe that when we toss two heads, because we're estimating parameters based on outcomes (which is the reverse), if we have no prior knowledge of the coin's bias and obtain such results, if I say "the probability of getting heads is 1", the credibility is 1. However, if I say "the probability of getting heads is 0.5", the credibility of such a statement would be lower given these results. Now, looking at the right side, when we toss three heads and one tail, the function changes. In this case, saying "the probability of getting heads is 1" holds no credibility anymore, as there's one occurrence of tails.

The likelihood function for parameter θ is the result of multiplying together the conditional probabilities $f(x_i|\theta)$ from 1 to n for all observed data points:

$$L(\theta) = \prod_{i=1}^{n} f(x_i|\theta)$$

In many cases, for the convenience of subsequent integration, we can take the logarithm, which results in the logarithm of the likelihood function. Because logarithms satisfy $log(ab) = log(a) + log(b)$, we can easily transform multiplication into addition using logarithms without affecting the location of function extrema, i.e.,

$$\log(L(\theta)) = \log\left(\prod_{i=1}^{n} f(x_i|\theta)\right) = \sum_{i=1}^{n} \log(f(x_i|\theta))$$

Taking the logarithm of the likelihood function typically transforms the product operation into a sum operation, making calculations easier, especially when optimization algorithms are used.

1.4 Maximum Likelihood Estimation

So, what does maximizing likelihood mean? It means finding the probability at which an event is most likely to occur (note: not the probability itself). We're interested in finding the maximum value of the function graph. As mentioned earlier, we can transform the likelihood function into a logarithmic likelihood function, which is often necessary in maximum likelihood estimation. This is because the logarithmic function is monotonically increasing, and its rate of increase decreases as x increases. Thus, it doesn't affect the position of the extremum, and we can more conveniently find the maximum likelihood.

It's quite simple, really. We all know that when the derivative is 0, that's where the local maximum of the function lies. Since the logarithmic likelihood function is monotonically increasing, we don't have to worry about local issues. We just need to set the derivative of the function to 0 like $log'(L(\theta)) = 0$.

In practice, setting the derivative to 0 works even for likelihood functions with convexity, as we can directly determine the local maximum. However, we rarely do this in practice because the logarithmic likelihood function possesses more desirable properties, including numerical stability in computer implementations. The concept is simple, so let's try it out.

1.5 Practical: Estimation of Binomial Distribution Parameters

We often use the binomial distribution to model the probability distribution of multiple independent binary experiments. That is, each experiment has two mutually exclusive outcomes, which we consider as "success" and "failure." Assuming the total number of experiments is n, the probability of success is p, and the number of successes is k. Since the result of the test is discrete, we use the probability mass function to represent it. If the random variable x follows a binomial distribution with parameters n and p, then the probability of obtaining k successes after n experiments is given by:

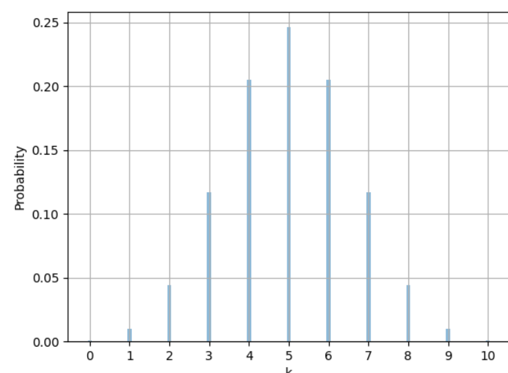
$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

Here, the binomial coefficient $\binom{n}{k}$ represents the number of ways to choose k objects out of n in n experiments, much like choosing k books out of n books, indicating how many ways you can select these n books. Its calculation formula is:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n - k)!}$$

What we are primarily concerned about is the probability of success p, where $1 - p$ represents the probability of failure. p^k represents the probability of k successes, and $p^k(1-p)^{n-k}$ represents the probability of failing in the remaining $n - k$ experiments.

For example, the figure illustrates that if attempted 10 times with a success probability of 0.5, the probability of five successes is the highest but still close to 0.25. Some readers may notice here that the difference between the binomial distribution and the Bernoulli distribution lies only in considering multiple experiments; if $n = 1$, it becomes the Bernoulli distribution.



This is just a distribution pattern, and what we need to do is, of course, not that simple—nor too complicated. What we need to do is simply to judge "the likelihood that this set of data follows a binomial distribution" given a set of given data and probabilities.

In our code, we default to 1 for success and 0 for failure. We'll first define the function signature, the operation process, factorial, and numerical values:

```
binomialLikelihood :: [Int] -> Double -> Double
  binomialLikelihood observations p = sum [log (choose n k) +
      fromIntegral k * log p + fromIntegral (n - k) * log (1 - p) | k <-
      observations]
  where
      choose :: Int -> Int -> Double
      choose n k = factorial n / (factorial k * factorial (n - k))
      n = fromIntegral $ length observations
      factorial 0 = 1
      factorial m = fromIntegral m * factorial (m - 1)
```

Given a list of integers, although we don't check whether they're 0s or 1s, this will affect the result. Each occurrence of 1 represents a success, so you can also pass integers like 3 or 4. Here, 'choose' denotes $\binom{n}{k}$, and we define the factorial operation below.

Then, we take the logarithm of the likelihood, perform calculations, and sum them up (line 2 of the code). The resulting value is the likelihood. Next, we define a function 'FindMLE' to obtain the maximum likelihood. It iterates from 0.01 to 0.99 over possible probability values to find the maximum value of the likelihood function, obtaining the most likely estimated parameter. Here, we use 'maximumBy' to compare likelihood function values under observed data:

```
findMLE :: [Int] -> Double
findMLE xs = maximumBy (\p1 p2 -> compare (likelihoodForP xs p1)
    (likelihoodForP xs p2)) [0.01, 0.02 .. 0.99]
```

You can test it yourself. Here's some simple testing code. In the simplest case, the result should be 0.25:

```
main :: IO ()
main = do
    let observations = [1, 0]
    let likelihood = BinomialDistribution.findMLE observations
    putStrLn $ "Likelihood: " ++ show likelihood
```

Now that we have gained some understanding of basic concepts like probability, likelihood, and Bayes' theorem, we'll delve into the application of Bayesian statistics in Naive Bayes later on.

2 Naive Bayes

Naive Bayes is one application of Bayesian statistics used as a machine learning algorithm for text classification and natural language processing.

Real-world situations are often complex, so Naive Bayes makes a simple assumption that features are independent of each other. Although this assumption may not always hold true, Naive Bayes still performs remarkably well, hence the term "naive." Let's assume $x = \{x_1, x_2, \cdots, x_n\}$ represents a set of features, and the class is denoted by c. The independence assumption of Naive Bayes implies:

$$P(x|c) = P(x_1|c)P(x_2|c)\cdots P(x_n|c) = \prod_{i=1}^{n} P(x_i|c)$$

Substituting back, we can see that a part of this expression is the prior probability $\frac{P(x|c)P(x)}{P(x)}$, while the rest is the product of feature conditional probabilities, represented directly using the multiplication symbol:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)} = \frac{P(c)}{P(x)} \prod_{i=1}^{n} P(x_i|c)$$

Here, we're concerned with estimating the prior probability of each class c in the classifier without any feature information. Here, $|D_c|$ represents the number of samples belonging to class c in the training set D, while $|D|$ represents the total number of samples in the entire training set, including samples from all classes. We can straightforwardly calculate the prior probability of class c using a simple ratio.

$$P(c) = \frac{|D_c|}{|D|}$$

On the other hand, Naive Bayes classifiers also involve estimating the likelihood probability. $|D_{c,x_i}|$ represents the number of samples in the training set D that have feature x_i and belong to class c. The formula for estimating the likelihood probability $P(x_i|c)$ is:

$$P(x_i|c) = \frac{|D_{c,x_i}|}{|D_c|}$$

¿ I choose to omit the subscript of class c, as is commonly done in most reference materials. This is because discussing a single class simplifies the expression in Bayesian classifiers. Essentially, this doesn't have much impact, so understanding is sufficient. ¿ **Just be aware that in Naive Bayes, there are often more than one class, so in practical operations, we need to distinguish between multiple classes (while in theory, we usually discuss a single class).**

2.1 Laplace Smoothing

A common method to optimize the calculation of our prior probabilities is Laplace smoothing, also known as add-one smoothing. Its focus is straightforward: add one to the numerator of the probability estimate to ensure that all events have nonzero probabilities, and add the possible number of classes N in the training set D to the denominator as a correction term.

$$P(c) = \frac{|D_c| + 1}{|D| + N}$$

$$P(x_i|c) = \frac{|D_{c,x_i}| + 1}{|D_c| + N}$$

Using Laplace smoothing is largely to prevent the probability of certain events from being zero, especially in sparse data situations, as this could lead to extreme biases in model predictions (e.g., events deemed impossible). On the other hand, it improves the robustness of the model, making it more stable and less sensitive to minor fluctuations, reducing the model's tendency to overfit noisy data, and so on.

2.2 Dependency Estimation

During the theoretical stage, Naive Bayes classifiers opt for simplicity by assuming independence to generalize various situations. While reality often deviates from this assumption, using it for classification in most cases doesn't affect its excellent performance. However, it's wise not to be overly flexible, so we consider some relatively complex variants that take dependencies into account, generally referred to as semi-Naive Bayes.

Dependency estimation allows for at most one dependency on another attribute in each attribute classification. We add a parent attribute pa_i for attribute x_i, called the parent attribute of x_i. We should rewrite the probability formula concerning x as:

$$P(x|c) = \prod_{i=1}^{n} P(x_i|c, pa_i)$$

For a given observed data x, the posterior probability of class c_i is represented as:

$$P(c|x) \propto P(c) \prod_{i=1}^{n} P(x_i|c, pa_i)$$

Here, $\propto$ denotes "proportional to," meaning we only need to consider its relative values, and we can omit $P(x)$ from the original formula $\frac{P(c)}{P(x)}$ and treat it as a normalization factor.

Considering the conditional probability of the parent attribute pa_i, it can be expressed as:

$$P(x_i|c, pa_i) = \frac{|D_{c,x_i,pa_i}|}{|D_{c_i,pa_i}|}$$

For the approach of dependency estimation, there are several algorithms with different dependency relationships.

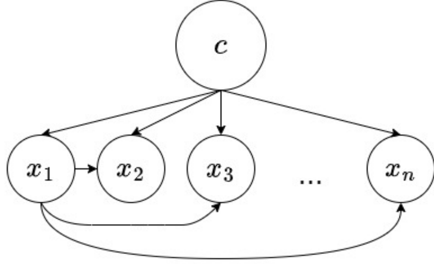
2.3 Multinomial Naive Bayes

Multinomial Naive Bayes is used for text classification and models probabilities with a multinomial distribution:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k) = \frac{n!}{x_1! x_2! \cdots x_k!} p_1^{x_1} p_2^{x_2} \cdots p_k^{x_k}.$$

2.3.1 TAN Algorithm

The simplest algorithm based on single parent one-dependence estimators is called the SPODE algorithm. It assumes that all attributes depend on the same attribute, referred to as the super-parent.

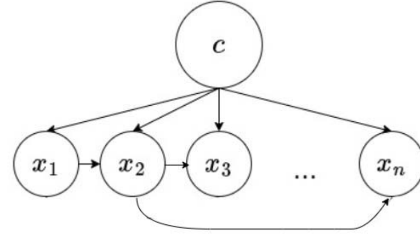


The dependency structure of SPODE is straightforward, as depicted in the diagram, where attribute x_1 serves as the super-parent.

2.3.2 SPODE Algorithm

The TAN algorithm selects a feature as the root node and employs a tree-like structure with the minimum number of edges connecting all nodes, maximizing edge lengths without forming cycles. This makes the parent node the one closest to satisfying the conditional independence assumption while also representing the dependency relationships between features.

Hence, the focus of the TAN algorithm lies in generating the maximum weighted spanning tree to establish dependencies. We can calculate the conditional mutual information between any two attributes using the following formula:



$$I(x_i, x_j|c) = \sum_{x_i, x_j, c \in c} \log \frac{P(x_i, x_j|c)}{P(x_i, c)P(x_j|c)}$$

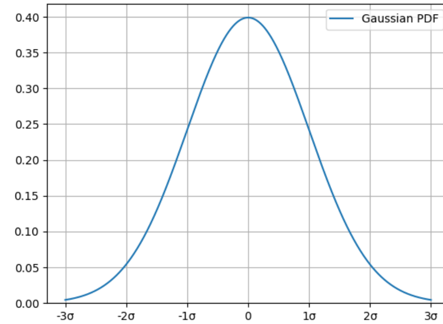
This formula measures the conditional mutual information between attributes x_i and x_j given class c by computing the logarithm of the ratio between the joint distribution of x_i and x_j given c and their respective marginal distributions given c. A higher conditional mutual information indicates a stronger correlation between x_i and x_j.

2.4 Gaussian Naive Bayes

Gaussian Naive Bayes is a variant of Naive Bayes. Typically, Naive Bayes assumes probability calculations for discrete features, but Gaussian Naive Bayes assumes that the probability distribution of feature values follows a Gaussian distribution (continuous probability distribution). As we know, the mean μ is the center or expected value of the Gaussian distribution, representing the average of the data, and the standard deviation σ is a parameter indicating the dispersion of the data points relative to the mean, representing the degree of data dispersion. The probability density function of the Gaussian distribution is:

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

Further, we use $\frac{1}{\sigma^2\sqrt{2\pi}}$ as a normalization factor to ensure the integral of the probability density function equals 1. The term $exp\left(-\frac{(x_i-\mu_c)^2}{2\sigma_c^2}\right)$ represents the probability density function of the Gaussian distribution, where the term in the exponential function, $(x_i - \mu_c)^2$, is the square of the difference between the feature value and the mean, representing the difference, and the denominator $2\sigma_c^2$ is twice the variance, which controls the shape of the curve as a measure of data dispersion.



So, in the case of a univariate Gaussian distribution model, the formula for computing the conditional probability of feature x_i belonging to class c is:

$$P(x_i|c) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x_i - \mu_c)^2}{2\sigma^2}\right)$$

For future practical applications involving estimation of multidimensional feature vectors, we need to know the probability density function of the multivariate Gaussian distribution and the conditional probability of feature vectors for classes, which are as follows:

$$f(\mathbf{x}; \mu, \sigma) = \frac{1}{(2\pi)^{n/2}|\sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^\top \sigma^{-1}(\mathbf{x} - \mu)\right)$$

Gaussian Naive Bayes models have relatively few parameters. Typically, we only need to estimate the mean and variance of each feature under each class, making it less sensitive to noise and extreme data points. It's an effective model for small datasets to prevent overfitting, and it's also a method to handle zero-frequency issues.

2.5 Multinomial Naive Bayes

Multinomial Naive Bayes is another variant of Naive Bayes algorithm specifically designed for probability calculations involving discrete features, commonly used in document classification.

Let's start with the Multinomial distribution, which is a generalization of the Binomial distribution we discussed earlier. If we used tossing a coin as an example for the Binomial distribution, we could use rolling a die as an example for the Multinomial distribution.

In my code repository, the calculation of Gaussian probability is a separate submodule, so we'll first complete the relevant functions here.

In the previous chapter, if the random variable x follows a Binomial distribution with parameters n and p, then the probability of getting k successes after n trials is:

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

Furthermore, we can generalize this formula to the Multinomial distribution. If there are possible outcomes $x_1, x_2, \ldots, x_k$ and n trials are conducted, with each outcome having probabilities $p_1, p_2, \ldots, p_k$, then the joint probability function of the Multinomial distribution is:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k) = \frac{n!}{x_1!x_2!\cdots x_k!} p_1^{x_1} p_2^{x_2} \cdots p_k^{x_k}$$

To explain briefly, $\frac{n!}{x_1!x_2!\cdots x_k!}$ is similar to the combination coefficient $\binom{n}{k}$ we discussed earlier, but generalized for the Multinomial distribution to calculate the number of different permutations that satisfy the requirements of the occurrence frequency of each category. Then, we simply multiply the probabilities of each category occurrence, $p_1^{x_1}, p_2^{x_2}, \ldots, p_k^{x_k}$.

In practice, Multinomial Naive Bayes is particularly advantageous for text classification. One reason it left a strong impression on me is that my AI teacher in junior high school used to praise it to the skies, and not understanding something that others praised made me feel stupid. Many times, doing something yourself reveals that it's not as difficult as it seems. Our Experiment A involves implementing a Multinomial Naive Bayes for word frequency analysis and text classification.

3 Some Applications

3.1 Practical A: Word Frequency Analysis and Text Classification

We will implement a sentiment classifier for articles. For this classifier, we'll provide some samples, such as a paragraph and its corresponding sentiment (positive, neutral, negative). Through learning, the program should be able to estimate the sentiment of a sentence. This analysis is based on the word frequency of samples, so it can also be applied to text classification, as long as you can handle the word frequency statistics part, it is language-agnostic.

Firstly, let's define the data types, namely the categories of classification and the data type for training documents:

```
1 data DocumentClass = Positive | Negative | Neutral deriving (Eq, Show)
2 type Document = (DocumentClass, [String])
```

We can start by writing a function to calculate the probability of the target class based on a set of training data (i.e., documents). Here, it's quite simple: we just divide the number of documents in the specified class by the total number of documents. We can use the 'filter' function to select the documents of the target class and then divide by the total number.

```
1 calculateCategoryProbability :: [Document] -> DocumentClass -> Double
2 calculateCategoryProbability documents category =
3     let totalDocs = length documents
4         categoryDocs = length (filter (\(docClass, _) -> docClass ==
              category) documents)
5     in fromIntegral categoryDocs / fromIntegral totalDocs
```

Then we need a function to calculate the probability of each word in a given document based on the given training data.

```
1 calculateWordProbabilities :: [Document] -> DocumentClass -> [(String,
     Double)]
2 calculateWordProbabilities documents category =
3     let categoryDocs = filter (\(docClass, _) -> docClass == category)
          documents
```

```haskell
        categoryTokens = concatMap snd categoryDocs
        wordCounts = fromListWith (+) [(word, 1) | word <- categoryTokens]
        totalWords = length categoryTokens
    in [(word, fromIntegral count / fromIntegral totalWords) | (word,
        count) <- toList wordCounts]
```

The ‘categoryTokens‘ on the fourth line is obtained by extracting the list of words from each document using the ‘snd‘ function, and then applying it to each element of the list using the ‘concatMap‘ function to concatenate the results into a new list. The ‘wordCounts‘ is a list generated by ‘fromListWith (+)‘, with the second parameter generating a list of tuples containing words and counts of 1 for each word in the document using list comprehension. The function calculates the total count of each word in the document by adding the counts of the same word.

Finally, here’s the function for text classification, which utilizes the two previous functions. It takes a training set and a target text as input and returns the probability of the target text corresponding to each category. We’ll start with the abstract part. The key point lies in the nested list comprehension, where each element represents the posterior probability of each category for the document. Here, ‘documentTokens‘ is a list of unique words. In the list comprehension, we first calculate the probability of the document belonging to that category, then call the ‘lookupWordProbability‘ function for each word and multiply the results to obtain the conditional probability of the document in that category. Finally, we multiply it with the conditional probability of the document and the category and store it as a tuple in the list.

```haskell
classifyDocument :: [Document] -> [String] -> [(DocumentClass, Double)]
classifyDocument documents document =
    let categories = [Positive, Negative, Neutral]
        documentTokens = nub document
    in [ (category,
        calculateCategoryProbability documents category *
        product [lookupWordProbability word category | word <-
            documentTokens])
        | category <- categories]
```

Here, we’re essentially applying Bayes’ theorem, and the formula should be:

$$P(\text{category}|\text{document}) = \frac{P(\text{category})P(\text{document}|\text{category})}{P(\text{document})}$$

Then, we define the ‘lookupWordProbability‘ function, which will call the function that calculates the probability of a word in a document. If there’s a probability, it returns it; otherwise, it returns a Laplace smoothing value, which typically occurs when a word is not present in the document.

```haskell
    where
        lookupWordProbability word category =
            case lookup word (calculateWordProbabilities documents
                category) of
                Just prob -> prob
                Nothing -> 1.0 / (fromIntegral (length (nub (map fst
                    documents))) + 1.0)
```

Since it deals with the case where the word does not appear in the document, we can apply the Laplace smoothing formula as follows, where the correction term N is 1, used as a pseudocount:

$$P(\text{word}|\text{category}) = \frac{0+1}{|\text{Documents}_{\text{category}}|+1}$$

Lastly, let's write some testing code. Here's my testing code, and after obtaining the results, we'll compare which probability is higher to directly determine the sentiment of the content. In the latter part, we'll take the category with the highest probability from the probability list and judge and output the sentiment of this sentence accordingly.

```
main :: IO ()
main = do
    let trainingData = [(Positive, ["love", "happy", "joy"]),
                        (Negative, ["hate", "sad", "angry"]),
                        (Neutral, ["neutral", "indifferent", "meh"])]
    let testSentence = words "I feel sad and angry"
    let classified = classifyDocument trainingData testSentence
    let result = maximumBy (compare `on` snd) classified
    putStrLn $ "The sentence belongs to class: " ++ show (fst result)
```

3.2 Implementation B: Gaussian Naive Bayes Classifier

We've just implemented a text classifier based on Multinomial Naive Bayes. Now, let's implement another classifier based on Gaussian data distribution to classify observations. According to the Gaussian probability density function we defined earlier, we write a function to calculate Gaussian probability, which takes the target value and a tuple containing the mean (μ) and variance (σ) of the Gaussian distribution (referred to as statistical features):

```
gaussianProbability :: Double -> (Double, Double) -> Double
gaussianProbability x (mean, variance) =
    (1.0 / (sqrt (2.0 * pi * variance))) * exp (-((x - mean) ** (2.0 ::
        Double)) / (2.0 * variance))
```

Note that this calculates the probability for univariate Gaussian, corresponding to the formula:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(x-\mu)^2}{2\sigma^2})$$

On the other hand, given a set of data assuming a Gaussian distribution, we compute their statistical features as follows:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\sigma = \frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2$$

```
calculateClassStats :: [Double] -> (Double, Double)
calculateClassStats values =
    let mean = sum values / fromIntegral (length values)
        variance = sum [(x - mean) ** (2.0 :: Double) | x <- values] /
            fromIntegral (length values)
```

```
    in (mean, variance)
```

Now, let's define some data structures for the classifier: Document represents a set of features, Class represents the class name, and ClassStats represents the statistical features corresponding to a class.

```
type Document = [Double]
type Class = String
type ClassStats = [(Double, Double)]
```

We define a function to calculate the prior probability of the target class given a set of classes:

```
priorProbability :: [Class] -> Class -> Double
priorProbability categories category =
    let totalDocs = length categories
        categoryCount = length (filter (== category) categories)
    in fromIntegral categoryCount / fromIntegral totalDocs
```

Then, we define a function to train the model based on the given training set and a specified class. It computes the statistical features of all the data belonging to the specified class in the training set.

```
trainModel :: [(Class, [Document])] -> Class -> ClassStats
trainModel trainingData category =
    let documents = concatMap snd $ filter (\(c, _) -> c == category)
       trainingData
        numFeatures = length (head documents)
    in [calculateClassStats [feature !! i | feature <- documents] | i <-
       [0..numFeatures - 1]]
```

In this way, we can calculate the maximum possible class for a set of data under the Gaussian Naive Bayes model by providing a training set. Firstly, we calculate the prior probability, corresponding to the priorProbability (map fst classStats) category segment, which gives the prior probability of the data belonging to the target class. Then, on the other side, by iterating through the contents of the training dataset using the map function, i.e., calculating the posterior probability of each tuple in the list under the corresponding Gaussian probability and prior probability. Specifically, we use map fst classStats to extract lists of all categories, and then multiply the Gaussian probability of each data's feature values under the corresponding class's statistical features, obtaining the product of the probability densities of all features. Multiplying these two results gives us the posterior probability we need.

Once we have the posterior probabilities for each class for this set of data, we simply select the maximum value as the most likely class it belongs to and return it.

```
predictCategory :: [Double] -> [(Class, ClassStats)] -> Class
predictCategory document classStats =
    let posteriorProbs = map (\(category, stats) ->
                            (category, priorProbability (map fst classStats)
                               category *
                               product (zipWith (\feature stat ->
                                   gaussianProbability feature stat)
                                       document stats)))
                            classStats
```

```
8         (maxCategory, _) = maximumBy (\(_, p1) (_, p2) -> compare p1 p2)
              posteriorProbs
9     in maxCategory
```

Now, with the trained model, we can calculate the most likely class for a set of data under this Gaussian Naive Bayes model. We first compute the prior probability, which is priorProbability (map fst classStats) category, representing the prior probability of the data belonging to the target category. Then, on the other side, we iterate over the training dataset's contents using the map function, calculating the posterior probability under the corresponding Gaussian and prior probabilities. Specifically, for each tuple in the list, we extract a list of all categories using map fst classStats, then calculate the product of Gaussian probabilities for each feature value under the corresponding class's statistical features. This gives us the product of probability densities for all features, and multiplying it by the prior probability gives us the posterior probability we need.

```
1  main :: IO ()
2  main = do
3      let trainingData = [("Class1", [[1.2], [1.4], [1.5], [1.3]]),
4                          ("Class2", [[2.0], [2.1], [2.2], [2.3]]),
5                          ("Class3", [[3.0], [3.1], [3.2], [3.3]])]
6
7      let classStats = map (\(category, _) -> (category, trainModel
           trainingData category)) trainingData
8      let testDocument = [1.34]
9      let predictedCategory = predictCategory testDocument classStats
10     putStrLn $ "Predicted Category: " ++ predictedCategory
```

The following test code represents a bivariate Gaussian distribution. The output result is "Class 3". Additionally, if I change the data to [2.2, 4.2], the output result will be "Class 2". This is intuitive, so feel free to try it out yourself.

```
1  main :: IO ()
2  main = do
3      let trainingData = [("Class1", [[1.2, 2.3], [1.4, 2.9], [1.5, 2.7],
           [1.3, 2.6]]),
4                              ("Class2", [[2.0, 3.1], [2.1, 3.5], [2.2, 3.3],
                                   [2.3, 3.4]]),
5                              ("Class3", [[3.0, 4.1], [3.1, 4.5], [3.2, 4.3],
                                   [3.3, 4.4]])]
6
7      let classStats = map (\(category, _) -> (category, trainModel
           trainingData category)) trainingData
8      let testDocument = [3.2, 4.2]
9      let predictedCategory = predictCategory testDocument classStats
10     putStrLn $ "Predicted Category: " ++ predictedCategory
```

References

[1] Murphy, K. P. (2012). *Introduction to machine learning*. MIT Press.

[2] Gelman, A., Carlin, J. B., Stern, H., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian data analysis* (3rd ed.). CRC Press.