Gaussian Naïve Bayes Classification

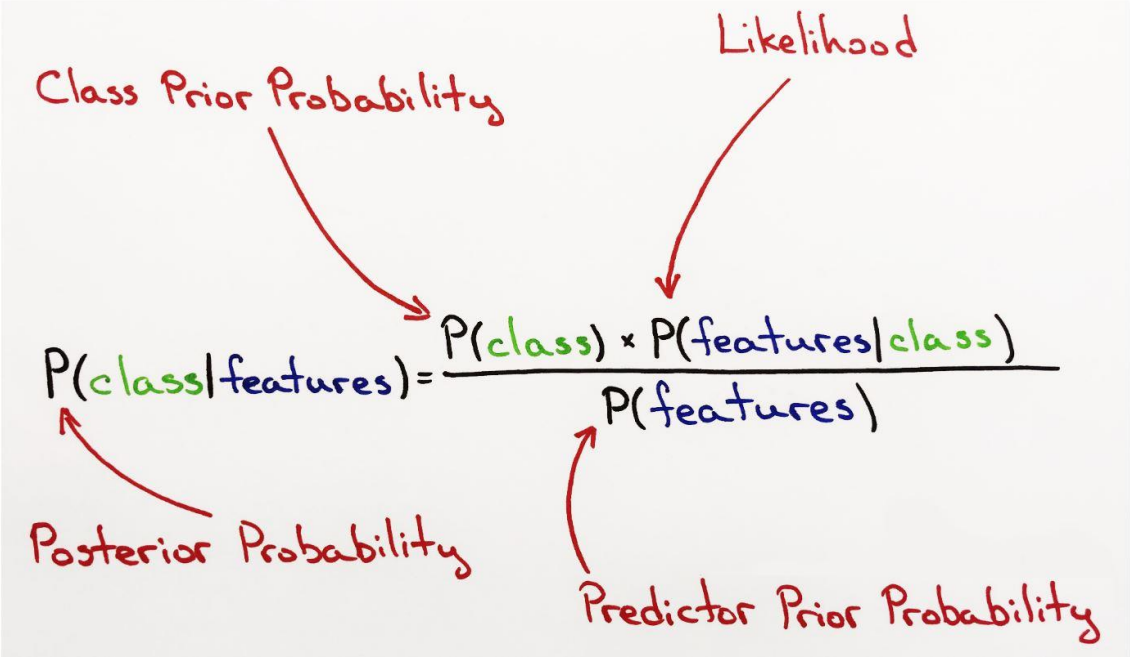
Naive Bayes models are a group of fast and simple classification algorithms that are often suitable for high-dimensional datasets. Because they are fast and have few tunable parameters, they are also used as a baseline for classification problems.

Bayesian Classification

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes's theorem.

Bayes Theorem provides a principled way for calculating a conditional probability. The theorem allows us to update the prior belief of an event X given a new piece of information Y. The result P(X|Y) is the probability of the event occuring given the new information Y.

In a classification problem we want to predict the class or label[1] of a sample given the observed features which we write as p(class|features).



Posterior Probability

- This is the updated belief given the new data, namely the features of the sample.

[1] In this tutorial the terms *class* and *label* are used interchangeably.

Class Prior Probability

- This is the prior belief; the probability of the class before updating with the new data.

Likelihood

- Likelihood is the product of all Normal Probability Density Functions[2] (assuming independence, ergo the "Naivete"). The Normal PDF is calculated using the Gaussian Distribution. Hence, the name Gaussian Naive Bayes.
- We will use the Normal PDF to calculate the normal probability value for each feature given the class.
- There's an important distinction to keep in mind between Likelihood and Probability.
    - Normal Probability is calculated for each feature given the class and is always between 0 and 1.
    - Likelihood is the product of all Normal Probability values.
    - Since there will always be features that could be added, the product of all Normal Probabilities is not the probability but the Likelihood.

Predictor Prior Probability

- Predictor Prior Probability is another way of saying Marginal Probability.
- It is the probability given the new data under all possible features for each class.
- It is not necessary for the Naive Bayes Classifier to calculate this because we are looking for the prediction and not the exact probability.

Normal PDF Formula

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

The normal distribution allows us to determine the probability of every feature given the class.

[2] Probability density function (PDF) is a statistical expression that defines a probability distribution (the likelihood of an outcome) for a discrete random variable (e.g., a stock) as opposed to a continuous random variable.

Building Generative Models

In a classification problem we want to build a model by which we can compute P(features|Ci) for every class Ci. Such a model is called a generative model because it specifies the hypothetical random process that generates the data. Specifying this generative model for each class is the main piece of the training of a Bayesian classifier. The general version of such a training step is a very difficult task, but we can make it simpler through the use of some simplifying assumptions about the form of this model. This is where the word naive in "naive Bayes" comes in; if we make naive assumptions about the generative model, then we can find an approximation of it, and we can then proceed with the Bayesian classification.

Different types of naive Bayes classifiers are built upon the different naive assumptions about the data.

Multiple Classifications

A classification model can handle both binary and multiple classifications. When predicting a class, the model calculates the posterior probability for all classes and selects the class with the largest posterior probability as the predicted class. This value is referred to as the ***Maximum A Posterior*** (MAP).

Note that when making a prediction among multiple classes we can simplify the MAP calculation. Suppose we are trying to decide between two classes; C1 and C2. We can compute the ratio of the posterior probabilities for each class:

$$\frac{P(C1 \mid features)}{P(C2 \mid features)} = \frac{P(features \mid C1)\ P(C1)}{P(features \mid C2)\ P(C2)}$$

and predict the class with higher posterior probability. Doing so will allows us to avoid to calculate P(features).

When to Use Naive Bayes

Because naive Bayesian classifiers make such stringent assumptions about data, they will generally not perform as well as a more complicated model. However they have several advantages:

- They are extremely fast for both training and prediction
- They provide straightforward probabilistic prediction
- They are often easily interpretable
- They have few (if any) tunable parameters

These advantages make a naive Bayesian classifier a good choice as an initial baseline classification. If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem. If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

Naive Bayes classifiers tend to perform especially well in the following situations:

- When the naive assumptions actually match the data (rare in practice)
- For well-separated categories, when model complexity is less important
- For high-dimensional data, when the model complexity is less important

The last two points seem distinct, but they actually are related: as the dimension of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in every single dimension to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information. For this reason, simplistic classifiers like naive Bayes tend to work as well or better than more sophisticated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

Gaussian Naive Bayes in SciKit-learn

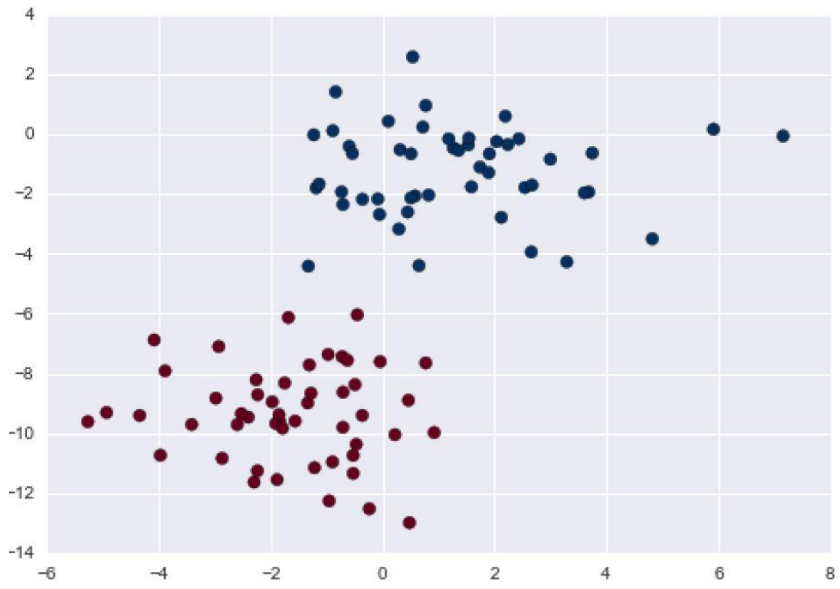
Perhaps the easiest naive Bayes classifier to understand is Gaussian naive Bayes. In this classifier, the assumption is that data from each label is drawn from a simple Gaussian distribution.

We begin with the standard imports:

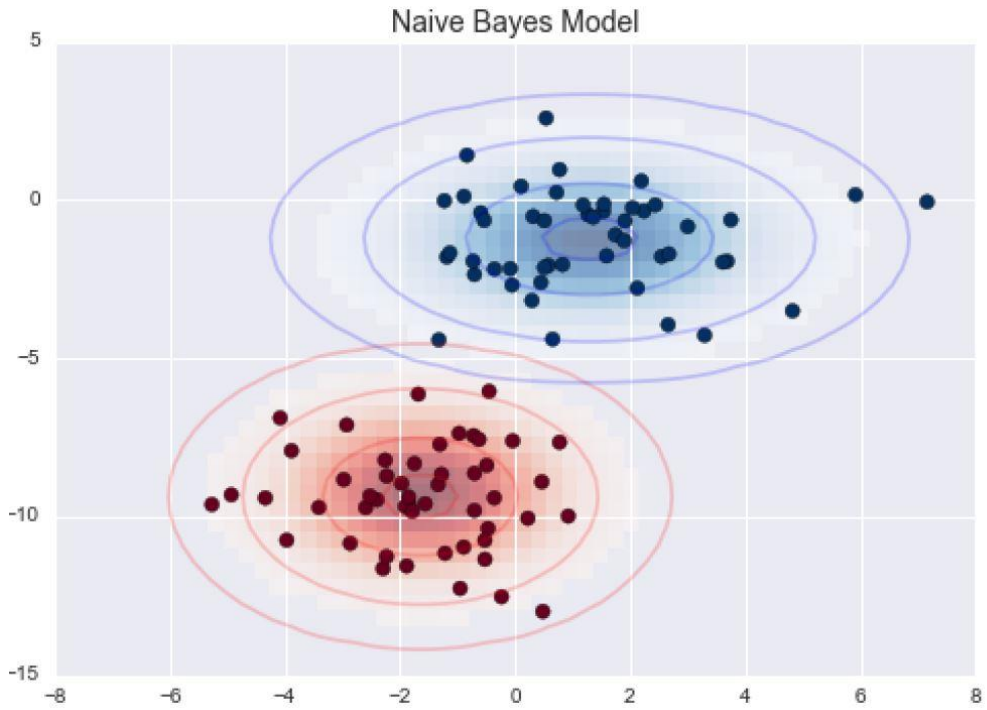
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Imagine that you have the following data:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='RdBu')
```



One fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by simply finding the mean and standard deviation of the points within each label, which is all you need to define such a distribution. The result of this naive Gaussian assumption is shown in the following figure:



The ellipses here represent the Gaussian generative model for each label with larger probability toward the center of the ellipses. With this generative model in place for each class, we have a simple recipe to compute the likelihood P(features|L1) for any data point, and thus we can quickly compute the posterior ratio and determine which label is the most probable for a given point.

This procedure is implemented with the estimator sklearn.naive_bayes.GaussianNB:

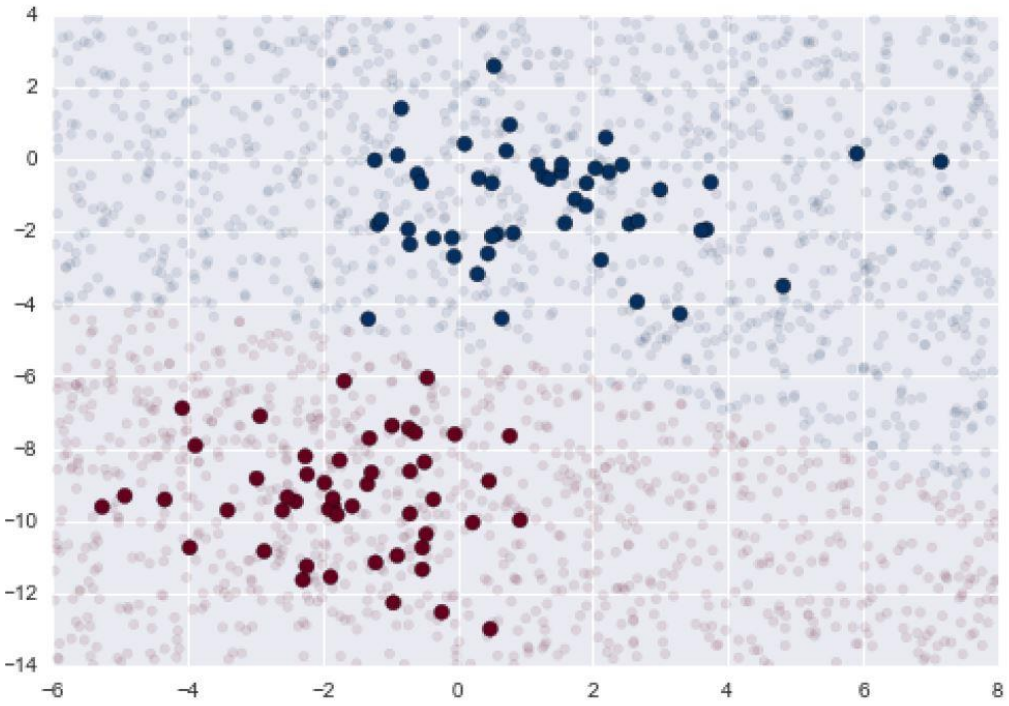
```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y)
```

Now let's generate some new data and predict the label:

```
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
ynew = model.predict(Xnew)
```

Now we can plot this new data to get an idea of where the decision boundary is:

```
plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:,0], Xnew[:,1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
plt.axis(lim)
```



We see a slightly curved boundary in the classifications—in general, the boundary in Gaussian naive Bayes is quadratic.

A nice property of the Bayesian formalism is that it naturally allows for probabilistic classification, which we can compute using the predict_proba method:

```
yprob = model.predict_proba(Xnew)
yprob[-8:].round(2)
```

```
array([ [ 0.89, 0.11],
        [ 1. , 0. ],
        [ 1. , 0. ],
        [ 1. , 0. ],
        [ 1. , 0. ],
        [ 1. , 0. ],
        [ 0. , 1. ],
        [ 0.15, 0.85]])
```

The columns give the posterior probabilities of the first and second label, respectively. If you are looking for estimates of uncertainty in your classification, Bayesian approaches like this can be a useful approach.

Of course, the final classification will only be as good as the model assumptions that lead to it, which is why Gaussian naive Bayes often does not produce very good results. Still, in many cases — especially as the number of features becomes large—this assumption is not harmful enough to prevent Gaussian naive Bayes from being a useful method.

Unfolding Gaussian Naive Bayes Classification

We will using Naive Bayes and the Gaussian Distribution to build a classifier that will predict flower species based on petal and sepal features. We will be working with the iris data set, a collection of 4 dimensional features that define 3 different types of flower species.

The **Iris data set** is a classic and is widely used when explaining classification models. The data set has 4 independent variables and 1 dependent variable. The table below shows 5 row samples. The first 4 columns are the independent variables (features) espreesed in cm. The 5th column is the dependent variable (class/label).

Independent variables

- *sepal length* (cm)
- *sepal width* (cm)
- *petal length* (cm)
- *petal width* (cm)

Dependent variable

- *class*
  - *Iris Setosa*
  - *Iris Versicolour*
  - *Iris Virginica*

Random 5 Row Sample

sepal length	sepal width	petal length	petal width	class
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
7.0	3.2	4.7	1.4	Iris-versicolor
6.3	2.8	5.1	1.5	Iris-virginica
6.4	3.2	4.5	1.5	Iris-versicolor

Python implementation

To create a classification model for the Iris data set we go through a three steps process.

1. Prepare Data
2. Build Model
3. Test Model

The complete code GaussNB.py is available on the course website.

1. Prepare data

First import the necessary libraries.

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
import statistics
from math import pi
from math import e
```

Load the Iris data set and extract the features and labels (target) of every sample. The variable target is a vector containing the label of every sample. The data set unique labels are placed in the vector target_values. Finally, split the data into a train_set and a test_set.

```
iris = datasets.load_iris()
data=iris.data
target=iris.target
target_values=np.unique(target)
X_train,X_test,y_train,y_test=train_test_split(data,target,test_size=0.3)
```

Group the data with respect their class. That is, place all the individual instances of each class in the same group.

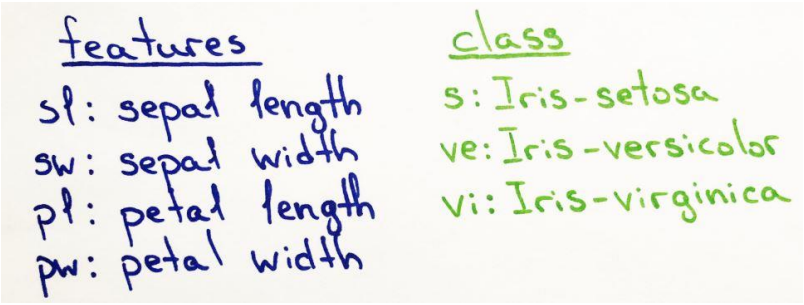
```
def group_by_class(self, data, target):
    """
    :param data: Training set
    :param target: the list of class labels labelling data
    :return:
    Separate the data by their target class; that is, create one group
    for every value of the target class. It returns all the groups
     """
  separated = [[x for x, t in zip(data, target) if t == c]
       for c in self.target_values]
  groups=[np.array(separated[0]),np.array(separated[1]),
       np.array(separated[2])]
  return np.array(groups)
```

2. Build model

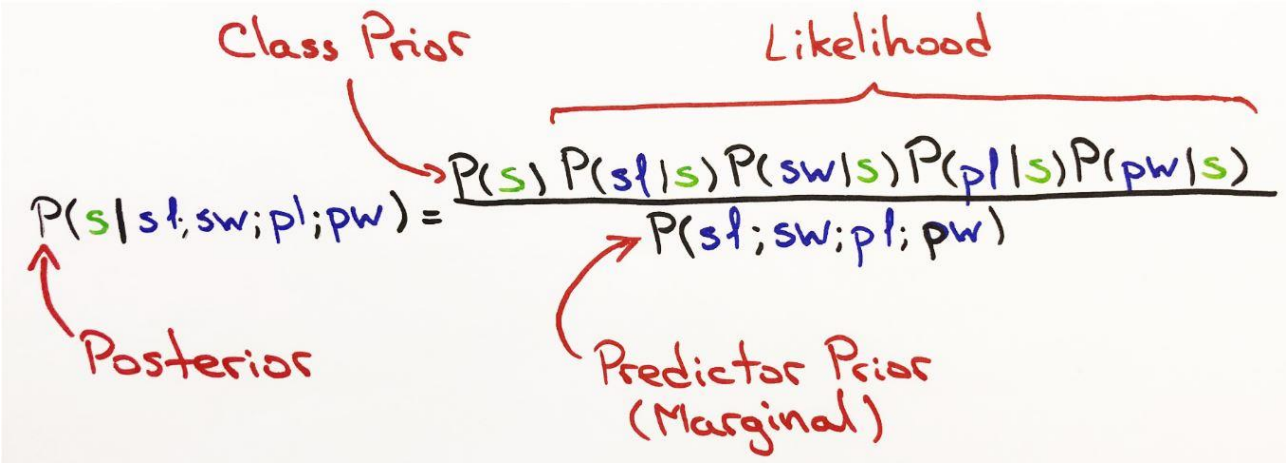
Here we build the functions to calculate the Bayes Theorem:

1. Prior Probability
2. Likelihood
3. Joint Probability
4. Marginal Probability
5. Posterior Probability

Features and Class



Using Iris-setosa as an example.



1. **Prior Probability**

$$P(class)$$

$$P(s)$$

Prior Probability is what we know about each class before considering the new data. It's the probability of each class occurring.

```
"""
The probability of each group of instances (that is the class) with
respect to the total number of instances
"""
len(group)/len(data)
```

Train

This is where we learn from the train set by calculating the mean and the standard deviation. Using the grouped classes, calculate the (mean, standard deviation) combination for each feature of each class.

```
def train(self, data, target):
    """
    :param data: a dataset
    :param target: the list of class labels labelling data
    :return:
    For each target class:
      1. yield prior_prob: the probability of each class
      2. yield summary: list of {'mean': 0.0,'stdev': 0.0}
         for every feature in data
    """
    groups = self.group_by_class(data, target)
    for index in range(groups.shape[0]):
       group=groups[index]
       self.summaries[self.target_values[index]] = {
         'prior_prob': len(group)/len(data),
         'summary': [i for i in self.summarize(group)]
       }
```

2. Likelihood

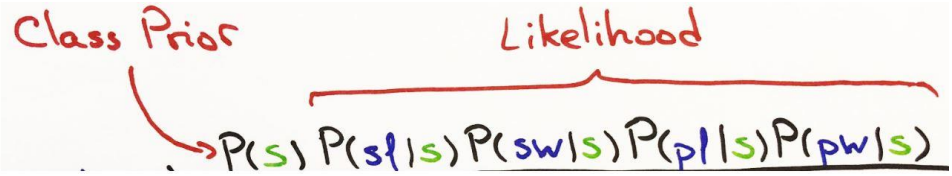
Likelihood is calculated by taking the product of all Normal Probabilities.

For each feature given the class we calculate the Normal Probability.

```
def normal_pdf(self, x, mean, stdev):
        """
        :param x: the value of a feature F
        :param mean: μ - average of F
        :param stdev: σ - standard deviation of F
        :return: Gaussian (Normal) Density function.
        N(x; μ, σ) = (1 / 2πσ) * (e ^ (x-μ)^2/-2σ^2
        """
        variance = stdev ** 2
        exp_squared_diff = (x - mean) ** 2
        exp_power = -exp_squared_diff / (2 * variance)
        exponent = e ** exp_power
        denominator = ((2 * pi) ** .5) * stdev
        normal_prob = exponent / denominator
        return normal_prob
```

3. Joint Probability

Joint Probability is calculated by taking the product of the Prior Probability and the Likelihood.



For each class:

1. Calculate the Prior Probability.
2. Use the Normal Distribution to calculate the Normal Probability of each feature; N(x; μ, σ)
3. Take the product of the Prior Probability and the Likelihood.
4. Return one Joint Probability value for each class given the new data.

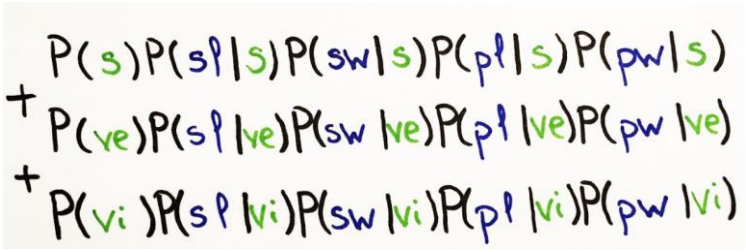
```
def joint_probabilities(self, data):
    """
    :param data: dataset in a matrix form (rows x col)
    :return:
    Use the normal_pdf(self, x, mean, stdev) to calculate the
    Normal Probability for each feature
    Yields the product of all Normal Probabilities and the
    Prior Probability of the class.
    """
    joint_probs = {}
    for y in range(self.target_values.shape[0]):
        target_v=self.target_values[y]
        item=self.summaries[target_v]
        total_features = len(item['summary'])
        likelihood = 1
        for index in range(total_features):
            feature = data[index]
            mean = self.summaries[target_v]['summary'][index]['mean']
            stdev = self.summaries[target_v]['summary'][index]['stdev']**2
            normal_prob = self.normal_pdf(feature,mean,stdev)
            likelihood *= normal_prob
        prior_prob = self.summaries[target_v]['prior_prob']
        joint_probs[target_v] = prior_prob * likelihood
    return joint_probs
```

Output:

```
Using 100 rows for training and 50 rows for testing
Grouped into 3 classes: ['Iris-virginica', 'Iris-setosa', 'Iris-versicolor']
  {
    'Iris-virginica': 7.880001356130214e-38,
    'Iris-setosa': 9.616469451152855e-230,
    'Iris-versicolor': 6.125801208117717e-68
  }
```

4. Marginal Probability

The Marginal Probability is calculated using the sum of all joint probabilities. The Marginal value, a single value, will be the same across all classes. We could think of the Marginal Probability as the total joint probability of all classes occurring given the new data.



Reminder, we're looking to predict the class by choosing the Maximum A Posterior (MAP). The prediction doesn't care about the exact posterior probability of each class and dividing by the same value is more memory intensive and does not improve the accuracy of predicting the correct class.

Thus, we really do not have to calculate the marginal probability here. See the section Multiple Classifications.

For the purposes of sticking to the true Bayes Theorem, we're using it here.

```
def marginal_pdf(self, joint_probabilities):
  """
  :param joint_probabilities: list of joint probabilities for each feature
  :return:
  Marginal Probability Density Function (Predictor Prior Probability)
  Joint Probability = prior * likelihood
  Marginal Probability is the sum of all joint probabilities for all classes
  """
  marginal_prob = sum(joint_probabilities.values())
  return marginal_prob
```

Output:

```
Marginal Probability: 1.29044139655
```

5. Posterior Probability

The Posterior Probability is the probability of a class occuring given the observed features.

We below calculate the posterior probability of each class with the goal of selecting MAP.

```
def posterior_probabilities(self, test_row):
  """
  :param test_row: single list of features to test; new data
  :return:
  For each feature (x) in the test_row:
     1. Calculate Predictor Prior Probability using the Normal PDF
          N(x; μ, σ). eg = P(feature | class)
     2. Calculate Likelihood by getting the product of the prior and
          the Normal PDFs
     3. Multiply Likelihood by the prior to calculate the
          Joint Probability.
  E.g.
  prior_prob: P(setosa)
  likelihood: P(sepal length | setosa) * P(sepal width | setosa) *
     P(petal length | setosa) * P(petal width | setosa)
  joint_prob: prior_prob * likelihood
  marginal_prob: predictor prior probability
  posterior_prob = joint_prob/marginal_prob
  Yields a dictionary containing the posterior probability of every class
  """
  posterior_probs = {}
  joint_probabilities = self.joint_probabilities(test_row)
  marginal_prob = self.marginal_pdf(joint_probabilities)
  for y in range(self.target_values.shape[0]):
     target_v=self.target_values[y]
     joint_prob=joint_probabilities[target_v]
     posterior_probs[target_v] = joint_prob / marginal_prob
  return posterior_probs
```

Output:

```
Using 100 rows for training and 50 rows for testing
Grouped into 3 classes: ['Iris-virginica', 'Iris-setosa', 'Iris-versicolor']
Posterior Probabilityies: {
   'Iris-virginica': 0.32379024365947745,
   'Iris-setosa': 2.5693999408505845e-158,
   'Iris-versicolor': 0.6762097563405226
}
```

3. Test model

Testing the model and predicting a class given the new data. We do this in three steps:

1. Get Maximum A Posterior
2. Predict
3. Accuracy

Get Maximum A Posterior

The get_map() method will call the posterior_probabilities() method on a single test_row eg ([6.3, 2.8, 5.1, 1.5]).

For each test_row we will calculate 3 Posterior Probabilities; one for each class. The goal is to select the Maximum A Posterior probability. The get_map() method will simply choose the Maximum A Posterior probability and return the associated class for the given test_row.

```
def get_map(self, test_row):
     """
     :param test_row: single list of features to test; new data
     :return:
     Return the target class with the largest posterior probability
     """
     posterior_probs = self.posterior_probabilities(test_row)
     target = max(posterior_probs, key=posterior_probs.get)
     return target
```

Predict

This method will return a prediction for each test_row.

Example input, list of lists:

```
[
    [5.1, 3.5, 1.4, 0.2],
    [4.9, 3.0, 1.4, 0.2],
]
```

```
def predict(self, data):
    """
    :param data: test_data
    :return:
    Predict the likeliest target for each row of data.
    Return a list of predicted targets.
    """
    predicted_targets = []
    for row in data:
        predicted = self.get_map(row)
        predicted_targets.append(predicted)
    return predicted_targets
```

Accuracy

Accuracy will test the performance of the model by taking the total number of correct predictions and divide them by the total number of predictions. This is critical in understanding the veracity of the model.

```
def accuracy(self, ground_true, predicted):
    """
    :param ground_true: list of ground true classes of test_data
    :param predicted: list of predicted classes
    :return:
    Calculate the the average performance of the classifier.
    """
    correct = 0
    for x, y in zip(ground_true, predicted):
        if x==y:
            correct += 1
    return correct / ground_true.shape[0]
```

Output:

```
Using 100 rows for training and 50 rows for testing
Grouped into 3 classes: ['Iris-virginica', 'Iris-setosa', 'Iris-versicolor']
Accuracy: 0.960
```