

Naive Bayes Classification

Amelia Henriksen

December 14, 2016

A Naive Bayes classifier is a simple and accessible model that has numerous applications: from text classification to medical diagnostics. Naive Bayes has essentially two parts: application of Bayes' theorem and a strong independence assumption. To describe the algorithm and how it works, we will discuss it in the context of a simple classification problem.

The Problem

Suppose we want to build a classifier based on the “seeds” dataset found in the UCI Machine Learning repository [1]. This dataset consists of 210 randomly selected samples of wheat kernels with seven measurements for associated geometric features. These real valued, continuous features indicate the area, perimeter, compactness, length, width, asymmetry coefficient, and length of the kernel groove for each kernel sample. Furthermore, each kernel is classified by one of three varieties of wheat: Kama, Rosa, or Canadian. We call these three classifications our “label set.”

Given a single wheat kernel's feature measurements, we want to predict its associated label. Let $x = (x_1, x_2, \dots, x_n)$ be a sample with n features (note that for the seeds dataset, $n = 7$). Let $Y = \{y_1, \dots, y_m\}$ be the label set (for the seeds dataset, $m = 3$). For each label y_j in the label set, we want to compute the probability of the sample x having a given label. Naive Bayes finds the label y_j that maximizes $P(y_j|x_1, x_2, \dots, x_n)$ for all $y_j \in Y$ [2].

We use Bayes' theorem to compute

$$P(y_j|x_1, x_2, \dots, x_n) = \frac{P(x_1, x_2, \dots, x_n|y_j)P(y_j)}{P(x_1, x_2, \dots, x_n)}$$

. Here is where the independence assumption comes into play. We assume that the random variables associated with each feature i are independent for all $i \in 1, \dots, n$. This allows us to compute

$$P(y_j|x_1, x_2, \dots, x_n) = \frac{P(y_j) \prod_{i=1}^n P(x_i|y_j)}{P(x_1, x_2, \dots, x_n)}$$

Finally, we can compute the predicted label for our sample:

$$y = \arg \max_{y_j \in Y} \left[\frac{P(y_j) \prod_{i=1}^n P(x_i|y_j)}{P(x_1, x_2, \dots, x_n)} \right] = \arg \max_{y_j \in \text{label set}} \left[P(y_j) \prod_{i=1}^n P(x_i|y_j) \right]$$

We avoid underflow by equivalently solving

$$y = \arg \max_{y_j \in K} \left[\log(P(y_j)) + \sum_{i=1}^n \log(P(x_i|y_j)) \right]$$

The assumption that the features of our dataset are independent of one another may seem rather strong—it is the reason Naive Bayes is considered “naive.” Indeed, when we take a look at the features in the wheat dataset, we see a clear dependence between the area parameter and the length and width parameters. However, as we will later see, the Naive Bayes classifier consistently yields surprisingly good results, even when the independence condition is violated. This, in conjunction with the relative simplicity of the algorithm, is what makes the method so powerful.

Solving for $P(y_j)$ and $P(x_i|y_j)$

We solve for $P(y_j)$ and $P(x_i|y_j)$ using a training dataset X , that is, using a set of samples with known labels. We calculate $P(y_j)$ to be the probability that a sample of label y_j appears in the training set. For example, in our seeds dataset, there are 70 samples of Kama wheat kernels. Thus $P(Kama) = \frac{70}{210} = \frac{1}{3}$.

Since each feature in the seeds dataset is continuous and real valued, we decide to use the Gaussian distribution to model $P(x_i|y_j)$ for each $i \in \{1, \dots, n\}, j \in \{1, \dots, m\}$. Note that although we will be demonstrating the efficacy of the Gaussian Naive Bayes classifier in this example, Multivariate and Bernoulli Naive Bayes classifiers are equally popular (indeed, Multinomial Naive Bayes is a prominent method in text classification). We calculate each $P(x_i|y_j)$ by using the maximum likelihood estimator to determine the appropriate parameters for each label. Thus, if there are l samples in the training set with label y_j , and $\{X_{ki}\}_{k=1}^l$ is the set of measurements for feature i on this set of samples, then

$$\mu_{ji} = \frac{1}{l} \sum_{k=1}^l X_{ki}$$

and

$$\sigma_{ji}^2 = \frac{1}{l} \sum_{k=1}^l (X_{ki} - \mu_{ji})^2$$

. We can carry out similar MLE computations to find the estimated parameters for other distributions.

Then, if x_i is the sample measurement for feature i , we can compute

$$P(x_i|y_j) = f(x_i|\mu_{ji}, \sigma_{ji}^2) = \frac{1}{\sqrt{2\sigma_{ji}^2\pi}} e^{-\frac{(x_i - \mu_{ji})^2}{2\sigma_{ji}^2}}$$

Finally, we predict the label for our sample x :

$$\begin{aligned}
y &= \arg \max_{y_j \in Y} \left[\log(P(y_j)) + \sum_{i=1}^n \log(P(x_i|y_j)) \right] \\
&= \arg \max_{y_j \in Y} \left[\log(P(y_j)) + \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\sigma_{ji}^2\pi}} e^{-\frac{(x_i - \mu_{ji})^2}{2\sigma_{ji}^2}} \right) \right] \\
&= \arg \max_{y_j \in Y} \left[\log(P(y_j)) + \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\sigma_{ji}^2\pi}} \right) + \log \left(e^{-\frac{(x_i - \mu_{ji})^2}{2\sigma_{ji}^2}} \right) \right] \\
&= \arg \max_{y_j \in Y} \left[\log(P(y_j)) + \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\sigma_{ji}^2\pi}} \right) + \left(-\frac{(x_i - \mu_{ji})^2}{2\sigma_{ji}^2} \right) \right]
\end{aligned}$$

This generates the following algorithm for predicting the label of a sample x using a Gaussian Naive Bayes classifier built with training data X (where X is a training set of l samples with n features with a known label for each sample) The Multinomial and Bernoulli Naive Bayes algorithms follow similarly.

Gaussian Naive Bayes

- Fitting the training data:

1. Group the training data X by label.
2. For each label $y_j \in Y$
 - (a) set $P(y_j) = \frac{\text{number of samples with label } y_j}{l}$
 - (b) For each feature $i \in \{1, \dots, n\}$, calculate the MLE for the distribution parameters.

$$\mu_{ji} = \frac{1}{l} \sum_{k=1}^l X_{ki}, \quad (1)$$

$$\sigma_{ji}^2 = \frac{1}{l} \sum_{k=1}^l (X_{ki} - \mu_{ji})^2 \quad (2)$$

- Predicting the label for a sample $x = (x_1, \dots, x_n)$

1. For each label $y_j \in Y$, calculate $\log(P(y_j)) + \sum_{i=1}^n \log P(x_i|y_j)$.
Calculate:

$$\log(P(y_j)) + \sum_{i=1}^n \log \left(\frac{1}{\sqrt{2\sigma_{ji}^2\pi}} \right) + \left(-\frac{(x_i - \mu_{ji})^2}{2\sigma_{ji}^2} \right)$$

2. Return the label that maximizes the above sum

As we can see here, building a Naive Bayes classifier consists of three simple and straightforward pieces:

1. Initializing a classifier object for a given label set
2. Fitting the classifier to a training dataset with samples and labels
3. Predicting the labels for a test set of samples.

These pieces are what we use to computationally build up our Gaussian Naive Bayes class, found in “NaiveBayes.py”. The array functionality available in NumPy is particularly well suited to this problem, as NumPy’s efficient array broadcasting allows us to compute the distribution parameters and prediction probabilities using cohesive array manipulations rather than costly for loops. Fancy indexing and boolean masks also make returning and manipulating the labels for training data more effective and efficient. Details are included in the documentation for “NaiveBayes.py”

Results for GaussNaiveBayes class

We now want to test our GaussNaiveBayes class for the “seeds” problem. These tests can be found in “NaiveBayesTest.ipynb.” We test the accuracy of our classifier by splitting the seeds dataset into a training set and a testing set for various test sizes and comparing the predicted test labels to the actual test labels (see self.accuracy in “NaiveBayes.py”).

To get a sense for the average accuracy of our predictions, we test 100 trials for each testsize, and in turn test 100 equally incremented testsizes between 0.01 and 0.98 (1% of the data to 98% of the data). To test the accuracy of our classifier against its industrial counterpart, we run the same procedure using the sklearn.naive_bayes.GaussianNB object. The results are shown in Figure 1, below.

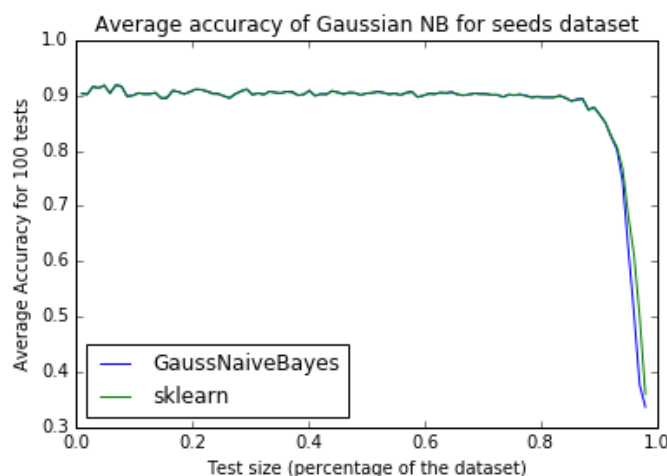


Figure 1: Accuracy of Gaussian Naive Bayes for the seeds dataset, for various test sizes.

This figure demonstrates some interesting results. First, recall that many of the features in our seeds dataset violated the independence assumption from which our classifier was derived. Yet our GaussNaiveBayes class yields about 90% accuracy—a respectable score. Even more impressive is the consistency of this accuracy—our model attains about 90% average accuracy for classifiers trained on less than 20% of the dataset (before dropping as the training dataset becomes insufficiently small, as we would expect to see).

We also note that our personal implementation for Naive Bayes matches the accuracy of the sklearn implementation exactly up to about .9 test size, and closely in the test sizes following. This testifies to the simplicity and efficacy of the test size.

In figure two, we take a closer look at the individual accuracies for each test run. As we can see here, the accuracy for the classifier has higher variation for trials of extremely small test sets—this could be explained by over-fitting of the model. Similarly, we see that the accuracy for the classifier has higher variation for trials of extremely large test sets—this could be explained by under-fitting of the model. Note that these accuracy scatter-plots match almost exactly between the industrial and personal Naive Bayes objects, with the industrial counterpart compensating slightly better for very small training datasets.

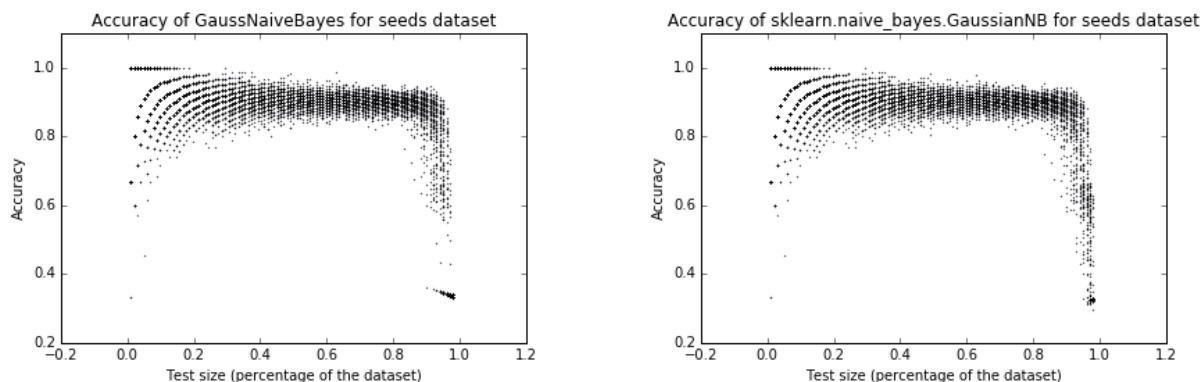


Figure 2: Accuracy of Gaussian Naive Bayes for the seeds dataset, for various test sizes.

Finally, we want to compare the timing between the industrial implementation of Gaussian Naive Bayes and our personal class. Using the magic `%%timeit` function in the ipython notebook, we find that the GaussNaiveBayes class predict method is actually faster than it's sklearn counterpart (it takes between 1/3 and 2/3 the time of the industrial classifier). For example, for a test size of 0.2, we have the following results:

```
train, test = train_test_split(np.arange(y.size), test_size=.2)
Xtr, Xt, ytr, yt = X[train], X[test], y[train], y[test]
NB.fit(Xtr, ytr)
sklNB.fit(Xtr, ytr)
%%timeit
NB.fit(Xtr, ytr)
#>>> 10000 loops, best of 3: 192 micro s per loop

%%timeit
sklNB.fit(Xtr, ytr)
#>>> 1000 loops, best of 3: 381 micro s per loop

%%timeit
p1 = NB.predict(Xt)
```

```
#>>>10000 loops, best of 3: 41.8 micro s per loop

%%timeit
p2 = sklNB.predict(Xt)
#>>>10000 loops, best of 3: 112 micro s per loop
```

Though this is likely due to some preprocessing and assertions on the part of the more robust sklearn function, which become important at higher dimensions, this still speaks highly to the efficiency of our implementation.

Improvements and observations

We have empirically shown the good performance of the Naive Bayes algorithm for our dataset. We now consider how Naive Bayes might be improved. The first consideration, of course, is to find a way to improve the accuracy of Naive Bayes for a problem that violates the independence assumption. Yet in 1997, Pedro Domingos and Michael Pazzani published a pivotal paper that showed that the Bayesian classifier is globally optimal for learning disjunctions and conjunctions of literals (that is, features that satisfy independence for class 0 and not class 1, or vice versa)[3]. Essentially, their work significantly extended the known class of functions for which Naive Bayes is optimal to a far greater class than problems satisfying independence.

Our next consideration might be to address the set of problems where this looser set of assumptions is still violated. We might suggest that a good way to improve the accuracy of the Naive Bayes classifier would be to perform some sort of model selection to minimize error due to dependence by selecting features that minimize this error. Yet, in the same paper mentioned above, Domingos and Pazzani also empirically demonstrated that the Naive Bayes classifier may still perform better than classifiers that extend naive bayes by detecting feature dependence [3]. Indeed, in a 2005 paper, Webb, Boughton and Wang note that using model selection increases the expected variance of the model, since selecting between models (for example, removing features that satisfy some “dependence” assumption) will cause the model to more closely fit the corresponding training data [4]. The classifier is then more susceptible to changes in the training data, whereas the Naive Bayes classifier’s lack of model selection actually makes changes in the model “more gradual” [4].

Obviously, this is not to say that Naive Bayes cannot be and has not been improved. In 2000, Zheng and Webb introduced the LBR algorithm (Lazy Learning of Bayesian Rules) which applies “lazy learning techniques to Bayesian tree induction” obtains significantly lower error rates than simple Naive Bayes [5]. In 2005, Webb, Boughton and Wang introduced an algorithm that averages all of a constrained class of classifiers to obtain a scheme that has similar accuracy to this *LBR* method and greater computational efficiency [4]. In 2012, Jiang, Cai, Zhange, and Wang introduced yet another improved Naive Bayes method that uses a random search through the space of attributes for an improved selection scheme for Naive Bayes [6]. Indeed, research on improvements to Naive Bayes and its application areas are ongoing and yielding new and powerful results. However, it is important to observe that in the case of the Naive Bayes classifier, the base algorithm is not nearly so naive as its name implies.

References

- [1] M. Lichman. UCI machine learning repository, 2013.
- [2] Naive bayes. 2014.
- [3] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2):103–130, 1997.
- [4] Geoffrey I. Webb, Janice R. Boughton, and Zhihai Wang. Not so naive bayes: Aggregating one-dependence estimators. *Machine Learning*, 58(1):5–24, 2005.
- [5] Zijian Zheng and Geoffrey I. Webb. Lazy learning of bayesian rules. *Machine Learning*, 41(1):53–84, 2000.
- [6] Liangxiao Jiang, Zhihua Cai, Harry Zhang, and Dianhong Wang. Not so greedy: Randomly selected naive bayes. *Expert Systems with Applications*, 39(12):11022 – 11028, 2012.