Naive Bayes

Supervised ML Algorithm

Sahasra Ranjan

April 2020

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of class variable.

What actually Bayes' theorem is?

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1}$$

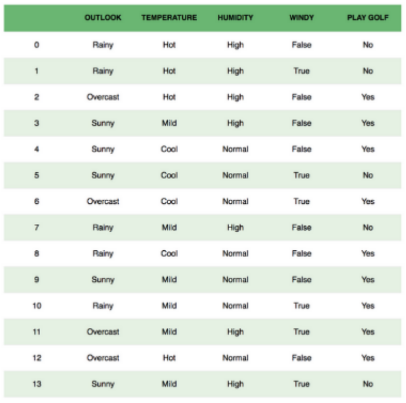
$P(A|B)$ is the probability of **A** happening, given that **B** has occured.

Why "Naive"? Because the presence of one particular feature does not affect the other.

Without going to deep, let's see an example:

The Golf Match Problem

Consider the problem of playing golf, dataset for the same:



	OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY GOLF
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No

Figure 1: Dataset for possiblity of a golf match

Bayes theorem for this example can be rewritten as:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)} \tag{2}$$

The variable **y** is the class variable (play golf), which represents if it is suitable to play golf or not given the conditions. Variable **X** is a matrix representing the parameters/features.

$\mathbf{X} = (x_1, x_2, x_3, ..., x_n)$ $x_1, x_2, ...x_n$ represent the features[1].

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)} \tag{3}$$

These values can be obtained by looking at the dataset and substituting them into equation will give us the result.

In our case, the the class variable($\mathbf{y}$) has only two outcomes, yes or no. Therefore, we need to find class $\mathbf{y}$ with maximum probability.

$$y = argmax_y P(y) \prod_{i=1}^{n} P(x_i|y) \tag{4}$$

Types of Naive Bayes classifier:

Miltinomial Naive Bayes:

This is mostly used for document classification problem, i.e whether a document belongs to the category of sports, politics, technology etc. The features/predictors used by the classifier are the frequency of the words present in the document.

Bernoulli Naive Bayes

This is similar to the multinomial naive bayes but the predictors are boolean variables. The parameters that we use to predict the class variable take up only values yes or no, for example if a word occurs in the text or not.

Gaussian Naive Bayes

When the predictors take up a continuous value and are not discrete, we assume that these values are sampled from a gaussian distribution.

The formula for conditional probability changes to:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} e^{-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}} \tag{5}$$

[1]temperature, humidity and windy (here)

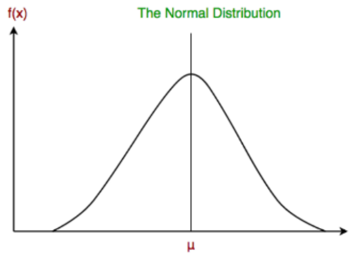


Figure 2: Gaussian Distribution(Normal Distribution)

Laplace Smoothing

It is problematic when a frequency-based probability is zero, because it will wipe out all the information in the other probabilities.

A solution would be **Laplace smoothing** , which is a technique for smoothing categorical data. A small-sample correction, or **pseudo-count**, will be incorporated in every probability estimate. Consequently, no probability will be zero. this is a way of regularizing Naive Bayes, and when the pseudo-count is zero, it is called Laplace smoothing. While in the general case it is often called **Lidstone smoothing.**

$$P_{i,\ \alpha-smoothed} = \frac{x_i + \alpha}{N + \alpha d} \tag{6}$$

where, $\alpha > 0$ the "pseudocount" is a smoothing parameter. And, $1/d$ is the Uniform Probability.

Note

In practice, we use logs to represent probabilities:

$$\log\left(P(x_1|y)P(x_2|y)...P(x_n|y)\right) = \log P(x_1|y) + \log P(x_2|y) + ... + \log P(x_n|y) \tag{7}$$

Applications

Naive Bayes algorithms are mostly used in sentiment analysis, spam filtering, recommendation systems etc. They are fast and easy to implement but their biggest disadvantage is that the requirement of predictors to be independent. In most of the real life cases, the predictors are dependent, this hinders the performance of the classifier.

Implementation

```python
import numpy as np
import pandas as pd

mush = pd.read_csv("./mushrooms.csv")
# This data consist of missing data with '?' as value. All
    the missing values are from single column.

# To remove it.
mush.replace('?',np.nan,inplace=True)
print(len(mush.columns),"columns, after dropping NA,",len(
    mush.dropna(axis=1).columns))

mush.dropna(axis=1,inplace=True)

target = 'class'
features = mush.columns[mush.columns != target]
classes = mush[target].unique()

test = mush.sample(frac=0.3)
mush = mush.drop(test.index)

# Probabilties Calculation
probs = {}
probcl = {}
for x in classes:
    mushcl = mush[mush[target]==x][features]
    clsp = {}
    tot = len(mushcl)
    for col in mushcl.columns:
        colp = {}
        for val,cnt in mushcl[col].value_counts().iteritems()
    :
            pr = cnt/tot
            colp[val] = pr
        clsp[col] = colp
    probs[x] = clsp
    probcl[x] = len(mushcl)/len(mush)


def probabs(x):
    #X - pandas Series with index as feature
    if not isinstance(x,pd.Series):
        raise IOError("Arg must of type Series")
    probab = {}
    for cl in classes:
        pr = probcl[cl]
        for col,val in x.iteritems():
            try:
                pr *= probs[cl][col][val]
            except KeyError:
                pr = 0
        probab[cl] = pr
    return probab


def classify(x):
    probab = probabs(x)
    mx = 0
    mxcl = ''
    for cl,pr in probab.items():
        if pr > mx:
            mx = pr
            mxcl = cl
    return mxcl

#Train data
b = []
for i in mush.index:
    b.append(classify(mush.loc[i,features]) == mush.loc[i,
   target])
print("Test 1")
print(sum(b),"correct of",len(mush))
print("Accuracy:", sum(b)/len(mush))


#Test data
b = []
for i in test.index:
    b.append(classify(test.loc[i,features]) == test.loc[i,
   target])
print("Test 2")
print(sum(b),"correct of",len(test))
print("Accuracy:",sum(b)/len(test))


'''
Output:

```

```
23 columns, after dropping NA, 22
Test 1
5671 correct of 5687
Accuracy: 0.997186565851943
Test 2
2433 correct of 2437
Accuracy: 0.9983586376692655
'''
```

Listing 1: Python Implementation for Naive Bayes