Gaussian Naive Bayes

Introduction

Expanding Bayes theorem to account for multiple observations and conditional probabilities drastically increases predictive power. In essence, it allows you to develop a belief network taking into account all of the available information regarding the scenario. In this lesson, you'll take a look at one particular implementation of a multinomial naive Bayes algorithm: Gaussian Naive Bayes.

Objectives

You will be able to:

- Explain the Gaussian Naive Bayes algorithm
- Implement the Gaussian Naive Bayes (GNB) algorithm using SciPy and NumPy

Theoretical background

Multinomial Bayes expands upon Bayes' theorem to multiple observations.

Recall that Bayes' theorem is:

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

Expanding to multiple features, the multinomial Bayes' formula is:

$$P(y|x_1, x_2, \ldots, x_n) = \frac{P(y)\prod_i^n P(x_i|y)}{P(x_1, x_2, \ldots, x_n)}$$

Here y is an observation class while x_1 through x_n are various features of the observation. Similar to linear regression, these features are assumed to be linearly independent. The motivating idea is that the various features $x_1, x_2, \ldots x_n$ will help inform which class a particular observation belongs to. This could be anything from 'Does this person have a disease?' to 'Is this credit card purchase fraudulent' or 'What marketing audience does this individual fall into?'. In this lesson you will work with classic iris dataset. This dataset includes various measurements of a flower's anatomy and the specific species of the flower. For that dataset, y would be the flower species while x_1 through x_n would be the various measurements for a given flower. As such, the equation for Multinomial Bayes, given above, would allow you to calculate the probability that a given flower belongs to a specific category of species.

With that, let's dig into the formula a little more to get a deeper understanding. In the numerator, you multiply the product of the conditional probabilities $P(x_i|y)$ by the probability of the class y. The denominator is the overall probability (across all classes) for the observed values of the various features. In practice, this can be difficult or impossible to calculate. Fortunately, doing so is typically not required, as you will simply be comparing the relative probabilities of the various classes—do you believe this flower is of species A, species B or species C?

To calculate each of the conditional probabilities in the numerator, $P(x_i|y)$, the Gaussian Naive Bayes algorithm traditionally uses the Gaussian probability density function to give a relative estimate of the probability of the feature observation, x_i, for the class y. Some statisticians don't agree with this as the probability of any point on a PDF curve is actually 0. As you've seen in z-tests and t-tests, only ranges of values have a probability, and these are calculated by taking the area under the PDF curve for the given range. While true, these point estimates can be loosely used as 'the relative probability for values near x_i'.

With that, you have:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{\frac{-(x-\mu_i)^2}{2\sigma_i^2}}$$

Where μ_i is the mean of feature x_i for class y and σ_i^2 is the variance of feature x_i for class y.

From there, each of the relative posterior probabilities are calculated for each of the classes. The largest of these is the class which is the most probable for the given observation.

With that, let's take a look in practice to try to make this process a little clearer.

Load the dataset

First, let's load in the Iris dataset to use to demonstrate the Gaussian Naive Bayes algorithm:

In [1]:
```
from sklearn import datasets
import pandas as pd
import numpy as np

iris = datasets.load_iris()

X = pd.DataFrame(iris.data)
X.columns = iris.feature_names

y = pd.DataFrame(iris.target)
y.columns = ['Target']

df = pd.concat([X, y], axis=1)
df.head()
```

Out[1]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

It's always a good idea to briefly examine the data. In this case, let's check how many observations there are for each flower species:

In [2]:
```
df['Target'].value_counts()
```

Out[2]:
```
2    50
1    50
0    50
Name: Target, dtype: int64
```

Calculate the mean and standard deviation of each feature for each class

Next, you calculate the mean and standard deviation within a class for each of the features. You'll then use these values to calculate the conditional probability of a particular feature observation for each of the classes.

In [3]:
```
aggs = df.groupby('Target').agg(['mean', 'std'])
aggs
```

Out[3]:

	sepal length (cm)		sepal width (cm)		petal length (cm)		petal width (cm)	
	mean	std	mean	std	mean	std	mean	std
Target								
0	5.006	0.352490	3.428	0.379064	1.462	0.173664	0.246	0.105386
1	5.936	0.516171	2.770	0.313798	4.260	0.469911	1.326	0.197753
2	6.588	0.635880	2.974	0.322497	5.552	0.551895	2.026	0.274650

Calculate conditional probability point estimates

Take another look at how to implement point estimates for the conditional probabilities of a feature for a given class. To do this, you'll simply use the PDF of the normal distribution. (Again, there can be some objection to this method as the probability of a specific point for a continuous distribution is 0. Some statisticians bin the continuous distribution into a discrete approximation to remedy this, but doing so requires additional work and the width of these bins is an arbitrary value which will potentially impact results.)

$$ \Large P(x_i|y) = \frac{1}{\sqrt{2 \pi \sigma_i^2}}e^{\frac{-(x-\mu_i)^2}{2\sigma_i^2}}$$

In [4]:
```
from scipy import stats

def p_x_given_class(obs_row, feature, class_):
    mu = aggs[feature]['mean'][class_]
    std = aggs[feature]['std'][class_]

    # A single observation
    obs = df.iloc[obs_row][feature]

    p_x_given_y = stats.norm.pdf(obs, loc=mu, scale=std)
    return p_x_given_y

# Notice how this is not a true probability; you can get values > 1
p_x_given_class(0, 'petal length (cm)', 0)
```

Out[4]:
```
2.1553774365786804
```

Multinomial Bayes

In [5]:
```
row = 100
c_probs = []
for c in range(3):
    # Initialize probability to relative probability of class
    p = len(df[df['Target'] == c])/len(df)
    for feature in X.columns:
        p *= p_x_given_class(row, feature, c)
        # Update the probability using the point estimate for each feature
        c_probs.append(p)

c_probs
```

Out[5]:
```
[0.0004469582872647558,
 0.00044432855867026464,
 5.436807559640758e-152,
 9.529514999027405e-251,
 0.20091323410933296,
 0.06135077392562668,
 5.488088968636944e-05,
 2.460149009916488e-12,
 0.1887425821931875,
 0.140076102721696,
 0.0728335779635225,
 0.023861042537402642]
```

Calculating class probabilities for observations

While you haven't even attempted to calculate the denominator for the original equation,

$$P(y|x_1,x_2,...x_n) = \frac{P(y)\prod_{i}^{n}P(x_i|y)}{P(x_1,x_2,...x_n)}$$

you don't really have to.

That is, the probability $P(x_1, x_2, ..., x_n)$ is the probability of the given observation across all classes; it is not a function of class at all. As such, it will be a constant across all of these posterior class probabilities. Since you are simply interested in the most likely class for each observation, you can simply pick the class with the largest numerator. With that, let's adapt the code snippet above to create a function which predicts a class for a given row of data.

In [6]:
```
def predict_class(row):
    c_probs = []
    for c in range(3):
        # Initialize probability to relative probability of class
        p = len(df[df['Target'] == c])/len(df)
        for feature in X.columns:
            p *= p_x_given_class(row, feature, c)
        c_probs.append(p)
    return np.argmax(c_probs)
```

Let's also take an example row to test this new function:

In [7]:
```
row = 0
df.iloc[row]
```

Out[7]:
```
sepal length (cm)    5.1
sepal width (cm)     3.5
petal length (cm)    1.4
petal width (cm)     0.2
Target               0.0
Name: 0, dtype: float64
```

In [8]:
```
predict_class(row)
```

Out[8]:
```
0
```

Nice! It appears that this `predict_class()` function has correctly predicted the class for this first row! Now it's time to take a look at how accurate this function is across the entire dataset!

Calculating accuracy

In order to determine the overall accuracy of your newly minted Gaussian Naive Bayes classifier, you'll need to generate predictions for all of the rows in the dataset. From there, you can then compare these predictions to the actual class values stored in the 'Target' column. Take a look:

In [9]:
```
df['Predictions'] =  [predict_class(row) for row in df.index]
df['Correct?'] = df['Target'] == df['Predictions']
df['Correct?'].value_counts(normalize=True)
```

Out[9]:
```
True     0.96
False    0.04
Name: Correct?, dtype: float64
```

Summary

Nicely done! You're well on your way to using Bayesian statistics in the context of machine learning! In this lesson, you saw how to adapt Bayes theorem along with your knowledge of the normal distribution to create a machine learning classifier known as Gaussian Naive Bayes.