index

March 17, 2022

1 Gaussian Naive Bayes - Lab

1.1 Introduction

Now that you've seen how to employ multinomial Bayes for classification, its time to practice implementing the process yourself. You'll also get a chance to investigate the impacts of using true probabilities under the probability density function as opposed to the point estimate on the curve itself.

1.2 Objectives

You will be able to:

- Independently code and implement the Gaussian Naive Bayes algorithm

1.3 Load the dataset

To get started, load the dataset stored in the file `'heart.csv'`. The dataset contains various measurements regarding patients and a `'target'` feature indicating whether or not they have heart disease. You'll be building a GNB classifier to help determine whether future patients do or do not have heart disease. As reference, this dataset was taken from Kaggle. You can see the original data post here: https://www.kaggle.com/ronitf/heart-disease-uci.

[1]:
```
# Your code here
# Load the dataset
import pandas as pd
import numpy as np

df = pd.read_csv("heart.csv")
```

1.4 Define the problem

As discussed, the dataset contains various patient measurements along with a `'target'` variable indicating whether or not the individual has heart disease. Define `X` and `y` below:

[2]:
```
# Your code here
y = df["target"]
X = df.drop("target", axis = 1)
y.value_counts(normalize=True)
```

[2]:
```
1    0.544554
0    0.455446
Name: target, dtype: float64
```

1.5 Perform a Train-test split

While not demonstrated in the previous lesson, you've seen from your work with regression that an appropriate methodology to determine how well your algorithm will generalize to new data is to perform a train-test split.

> Note: Set `random_state` to 22 and `test_size` to 0.25 to have your results match those of the solution branch provided.

[3]:
```
# Your code here
# Perform a train-test split
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.25,
                                                    random_state = 22)
```

1.6 Calculate the mean & standard deviation of each feature for each class in the training set

Now, calculate the mean and standard deviation for each feature within each of the target class groups. This will serve as your a priori distribution estimate to determine the posterior likelihood of an observation belonging to one class versus the other.

[4]:
```
# Your code here
# Calculate the mean and standard deviation for each feature within each class
# for the training set
train = pd.concat([X_train, y_train], axis = 1)
aggs_train = train.groupby("target").agg(["mean", "std"])
aggs_train
```

[4]:
```
              age                  sex                   cp                \
             mean       std       mean       std       mean       std
target
0       57.281553  8.009085  0.796117  0.404853  0.466019  0.916253
1       52.322581  9.995567  0.564516  0.497832  1.427419  0.972578

          trestbps                  chol                ...     exang                \
              mean        std        mean        std    ...      mean       std
target                                                  ...
0       134.067961  18.919469  251.543689  52.341596    ...  0.543689  0.500523
1       129.137097  16.589415  240.516129  46.683240    ...  0.161290  0.369291

         oldpeak                slope                   ca                  thal   \
            mean       std       mean       std       mean       std       mean
```

```
target
0       1.600971  1.310253  1.184466  0.555676  1.213592  1.025656  2.582524
1       0.592742  0.749544  1.580645  0.612827  0.354839  0.818422  2.072581


             std
target
0       0.602678
1       0.444906

[2 rows x 26 columns]
```

1.7 Define a function to calculate the point estimate for the conditional probability of a feature value for a given class

Recall that the point estimate is given by the probability density function of the normal distribution:

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

> Note: Feel free to use the built-in function from SciPy to do this as demonstrated in the lesson. Alternatively, take the time to code the above formula from scratch.

[5]:
```python
# Your code here
from scipy import stats

def point_estimate(row, target, feature, aggs):

    mean = aggs[feature]["mean"][target]
    mu   = aggs[feature]["std"][target]

    point = row[feature]
    point_est = stats.norm.pdf(point, loc = mean, scale = mu)
    return point_est
```

1.8 Define a prediction function

Define a prediction function that will return a predicted class value for a particular observation. To do this, calculate the point estimates for each of the features using your function above. Then, take the product of these point estimates for a given class and multiply it by the probability of that particular class. Take the class associated with the largest probability output from these calculations as your prediction.

[6]:
```python
# Your code here
def prediction(y, X, row):

    data = pd.concat([X, y], axis = 1)
    aggs = data.groupby("target").agg(["mean", "std"])

    prob = []

#     target = list(pd.unique(y_train))
    target = range(2)

    for i in target:
        p = len(y[y == i])/len(y)

        for col in X.columns:
            p *= point_estimate(row, i, col, aggs)
        prob.append(p)

    return np.argmax(prob)
```

1.9 Apply your prediction function to the training and test sets

[7]:
```
# Your code here

item = prediction(y_train, X_train, row = X_train.iloc[0])
item
```

[7]:
```
0
```

[8]:
```
df.iloc[0]
```

[8]:
```
age          63.0
sex           1.0
cp            3.0
trestbps    145.0
chol        233.0
fbs           1.0
restecg       0.0
thalach     150.0
exang         0.0
oldpeak       2.3
slope         0.0
ca            0.0
thal          1.0
target        1.0
Name: 0, dtype: float64
```

1.10 Calculate the training and test accuracy

```
[9]: # Your code here
     y_train_l = list(y_train)
     train_accu = pd.DataFrame()
     train_accu["Pred"]=[prediction(y_train,X_train,X_train.iloc[row])
                         for row in range(len(X_train))]
     train_accu["true"] = [y_train_l[row] for row in range(len(y_train_l))]
     train_accu["Correct"] = (train_accu["Pred"] == train_accu["true"]).astype(int)
```

```
[10]: train_accu["Correct"].value_counts(normalize=True)
```

```
[10]: 1    0.85022
      0    0.14978
      Name: Correct, dtype: float64
```

```
[11]: y_test_l = list(y_test)
      l = range(len(X_test.index))
      test_accu = pd.DataFrame()
      prediction(y_test, X_test, X_test.iloc[l[0]])
      test_accu["Pred"] = [prediction(y_test, X_test, X_test.iloc[row]) for row in l]
      test_accu["true"] = [y_test_l[row] for row in range(len(y_test_l))]
      test_accu["Correct"] = (test_accu["Pred"] == test_accu["true"]).astype(int)
```

```
[12]: test_accu["Correct"].value_counts(normalize=True)
```

```
[12]: 1    0.802632
      0    0.197368
      Name: Correct, dtype: float64
```

1.11 Level up (Optional)

1.11.1 Adapting point estimates for the conditional probability into true probability estimates

As discussed, the point estimate from the probability density function is not a true probability measurement. Recall that the area under a probability density function is 1, representing the total probability of all possible outcomes. Accordingly, to determine the probability of a feature measurement occurring, you would need to find the area under some portion of the PDF. Determining appropriate bounds for this area however, is a bit tricky and arbitrary. For example, when generating a class prediction, you would want to know the probability of a patient having a resting blood pressure of 145 given that they had heart disease versus the probability of having a resting blood pressure of 145 given that the did not have heart disease. Previously, you've simply used the point where x=145 on the PDF curve to do this. However, the probability of any single point is actually 0. To calculate the actual probability, you would have to create a range around the observed value such as "what is the probability of having a resting blood pressure between 144 and 146 inclusive?" Alternatively, you could narrow the range and rewrite the problem as "what is the probability of having a resting blood pressure between 144.5 and 145.5?" Since defining these bounds is arbitrary, a potentially interesting research question is how various band methods might impact output predictions and the overall accuracy of the algorithm.

1.12 Rewriting the conditional probability formula

Rewrite your conditional probability formula above to take a feature observation, a given class, and a range width and calculate the actual probability beneath the PDF curve of an observation falling within the range of the given width centered at the given observation value. For example, taking the previous example of resting blood pressure, you might calculate the probability of having a resting blood pressure within 1bp of 145 given that a patient has heart disease. In this case, the range width would be 2bp (144bp to 146bp) and the corresponding area under the PDF curve for the normal distribution would look like this:

With that, write such a function below:

[15]:
```python
import scipy.stats as stats
def p_band_x_given_class(obs_row, feature, target, range_width_std, aggs):
    """obs_row is the observation in question
    feature is the feature of the observation row for which you are
    calculating a conditional probability c is the class flag for the
    conditional probability range_width_std is the range in standard
    deviations of the feature variable to calculate the integral under
    the PDF curve for"""
    # Your code here

    point = obs_row[feature]
    mean = aggs[feature]["mean"][target]
    mu   = aggs[feature]["std"][target]

    upper_point = point + range_width_std*mu / 2
    lower_point = point - range_width_std*mu / 2

    p_x_given_y = (stats.norm.cdf(upper_point,loc=mean,scale=mu)
                   - stats.norm.cdf(lower_point,loc=mean,scale=mu))


    return p_x_given_y
```

1.13 Update the prediction function

Now, update the `predict_class()` function to use this new conditional probability function. Be sure that you can pass in the range width variable to this wrapper function.

[17]:
```python
# Your code here
# Update the prediction function

def updated_prediction(y, X, row, range_width_std):

    data = pd.concat([X, y], axis = 1)
    aggs = data.groupby("target").agg(["mean", "std"])

    prob = []

    target = list(pd.unique(y_train))
#     target = range(2)

    for i in target:
        p = len(y[y == i])/len(y)

        for col in X.columns:
            p *= p_band_x_given_class(row, col, i, range_width_std, aggs)
#               p *= point_estimate(row, i, col, aggs)
        prob.append(p)

    return np.argmax(prob)
```

1.14 Experiment with the impact of various range-widths

Finally, write a `for` loop to measure the impact of varying range-widths on the classifier's test and train accuracy. Iterate over various range-widths from 0.1 standard deviations to 2 standard deviations. For each of these, store the associated test and train accuracies. Finally, plot these on a graph. The x-axis should be the associated range-width (expressed in standard deviations; each feature will have a unique width applicable to the specific scale). The y-axis will be the associated accuracy. Be sure to include a legend for train accuracy versus test accuracy.

Note: Expect your code to take over two minutes to run.

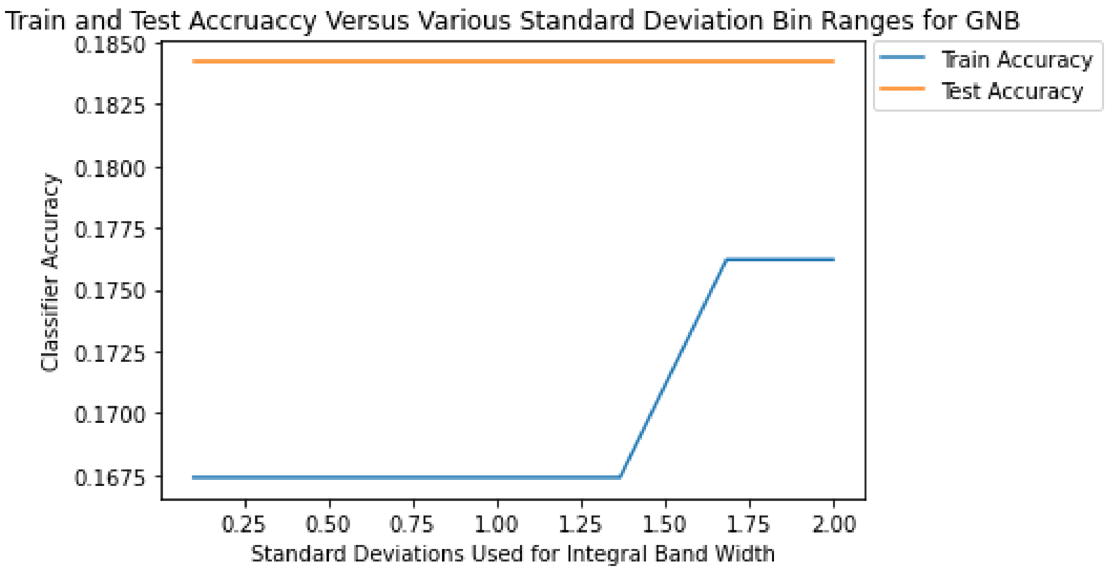
[19]:
```
# Your code here

# Your code here
import matplotlib.pyplot as plt
%matplotlib inline
train_accs = []
test_accs = []
range_stds = np.linspace(0.1, 2, num=7)
for range_std in range_stds:

    y_hat_train = [updated_prediction(y_train, X_train, X_train.iloc[row],
                                       range_width_std = range_std
                                      ) for row in  range(len(X_train))]
    y_hat_test = [updated_prediction(y_test, X_test, X_test.iloc[row],
                                      range_width_std = range_std
                                     ) for row in  range(len(X_test))]

    residuals_train = y_hat_train == y_train
    acc_train = residuals_train.sum()/len(residuals_train)
    residuals_test = y_hat_test == y_test
    acc_test = residuals_test.sum()/len(residuals_test)

    train_accs.append(acc_train)
    test_accs.append(acc_test)
plt.plot(range_stds, train_accs, label='Train Accuracy')
plt.plot(range_stds, test_accs, label='Test Accuracy')
plt.title('Train and Test Accruaccy Versus Various Standard Deviation Bin␣
 ↪Ranges for GNB')
plt.ylabel('Classifier Accuracy')
plt.xlabel('Standard Deviations Used for Integral Band Width')
plt.legend(loc=(1.01,.85));
```



Comment: Not a wild difference from our point estimates obtained by using points from the PDF itself, but there is some impact. **Interestingly, these graphs will differ substantially in shape depending on the initial train-test split used.** The recommendation would be to use the point estimates from the PDF itself, or a modest band-width size.

1.15 Appendix: Plotting PDFs and probability integrals

Below, feel free to take a look at the code used to generate the PDF graph image above.

```
[ ]: temp = df[df['target'] == 1]['trestbps']
     aggs = temp.agg(['mean', 'std'])
     aggs
```

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     from matplotlib.patches import Polygon
     import seaborn as sns
     import scipy.stats as stats
     sns.set_style('white')
```

```
[ ]: x = np.linspace(temp.min(), temp.max(), num=10**3)
     pdf = stats.norm.pdf(x, loc=aggs['mean'], scale=aggs['std'])
     xi = 145
     width = 2
     xi_lower = xi - width/2
     xi_upper = xi + width/2

     fig, ax = plt.subplots()

     plt.plot(x, pdf)

     # Make the shaded region
     ix = np.linspace(xi_lower, xi_upper)
     iy = stats.norm.pdf(ix, loc=aggs['mean'], scale=aggs['std'])
     verts = [(xi_lower, 0), *zip(ix, iy), (xi_upper, 0)]
     poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
     ax.add_patch(poly);

     plt.plot((145, 145), (0, stats.norm.pdf(145, loc=aggs['mean'],␣
      ↪scale=aggs['std'])), linestyle='dotted')
     p_area = stats.norm.cdf(xi_upper, loc=aggs['mean'], scale=aggs['std']) - stats.
      ↪norm.cdf(xi_lower, loc=aggs['mean'], scale=aggs['std'])
     print('Probability of Blood Pressure Falling withing Range for the Given Class:␣
      ↪{}'.format(p_area))
     plt.title('Conditional Probability of Resting Blood Pressure ~145 for Those␣
      ↪With Heart Disease')
     plt.ylabel('Probability Density')
     plt.xlabel('Resting Blood Pressure')
```

Comment: See https://matplotlib.org/gallery/showcase/integral.html for further details on plotting shaded integral areas under curves.

1.16 Summary

Well done! In this lab, you implemented the Gaussian Naive Bayes classifier from scratch, and used it to generate classification predictions and validated the accuracy of the model.