Lecture 8: Naive Bayes, LDA, and QDA

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

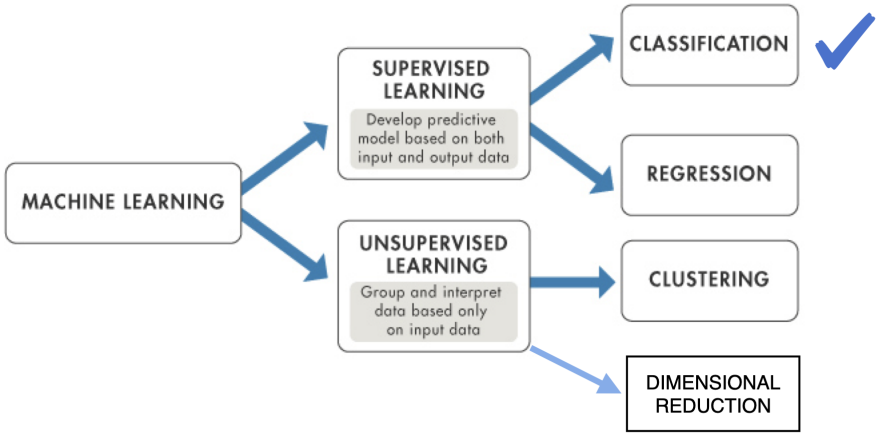
March 24, 2023

1 Naive Bayes, LDA, and QDA

In today's lecture[1], we introduce a number of probabilistic *classification algorithms*. These classification algorithms are the **Naive Bayes**, the **Linear Discriminant Analysis (LDA)**, and the **Quadratic Discriminant Analysis (QDA)**. All these classification algorithms are constructed upon the Bayes' theorem in statistics.[2]

1.1 Background

So far, we have introduced two classification algorithms, namely the logistic regression and the support vector machine. Recall that classification algorithms belong to the realm of supervised machine learning where the target is a categorical variable.



[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

[2] Thomas Bayes was not only a statistician but also a philosopher, you can read his original letter "An essay towards solving a problem in the doctrine of chances," at https://doi.org/10.1098/rstl.1763.0053.

In previous weeks, we introduced the logistic regression and the support vector machine algorithms for classification problems. As we saw, the former took a probabilistic approach toward the classification problem, whereas the latter took a geometric approach. The classification algorithms we will introduce today (*i.e.* naive Bayes, LDA, and QDA) all take a probabilistic approach toward the classification problem. This approach is based upon an important theorem in statistics, namely the Bayes' theorem.

Question: We already introduced a probabilistic classification algorithm, namely the logistic regression. Why are we interested in other probabilistic approaches toward classification problems?

Answer: There are several good reasons to look for other probabilistic approaches toward classification problems:

- Logistic regression cannot handle categorical features, whereas algorithms based on Bayes' theorem can include categorical features.
- The parameter estimates for the logistic regression model are surprisingly unstable when there are substantial separations between different classes. The classification methods we will introduce today do not suffer from this problem.
- If the distribution of the features (approximately) follows a normal distribution for each class, and the size of the dataset is not too large, then the probabilistic approaches we will introduce today typically turn more accurate results than the logistic regression.

1.2 Naive Bayes Classifier

To introduce the naive Bayes classifier, let's review basic formulas from the probability theory. Let X and Y represent two events. We say events X and Y are *independent* if the occurrence of one does not affect the probability of occurrence of the other. We use notation $X \cap Y$ to indicate event X and Y. If X and Y are independent events, then

$$p(X \cap Y) = p(X)\, p(Y)\,. \tag{1}$$

Moreover, for two events X and Y, we use the notation $X \cup Y$ to indicate the event X or Y. It is easily shown that the probability of event X or Y can be expressed in terms of probabilities of events X, Y, and $X \cap Y$ as follows

$$p(X \cup Y) = p(X) + p(Y) - p(X \cap Y)\,. \tag{2}$$

Another crucial concept in probability theory is the notion of **conditional probability**. For two events X and Y, $p(Y|X)$ denotes the *probability of the occurrence of* Y, *given that event* X *has already occurred*. It is shown that $p(Y|X)$ can be calculated as follows.

$$p(Y|X) = \frac{p(X \cap Y)}{p(X)}\,. \tag{3}$$

A fundamental element of probability theory and statistics is the Bayes' theorem which provides a way to calculate conditional probabilities provided that certain information has been given. Let us state Bayes' theorem and see how it works through an example.

Bayes' Theorem: Let X and Y be two events. Then $p(Y|X) = \dfrac{p(X|Y)\,p(Y)}{p(X)}$.

Using (3), one can immediately prove the statement of the Bayes' theorem. Equation (3) states that $p(X \cap Y) = p(Y|X)\,p(X)$, and at the same time $p(Y \cap X) = p(X|Y)\,p(Y)$. Since $p(X \cap Y) = p(Y \cap X)$, the result follows immediately:

$$\underbrace{p(Y|X)}_{\text{posterior}} = \frac{\overbrace{p(X|Y)}^{\text{likelihood}}\;\overbrace{p(Y)}^{\text{prior}}}{\underbrace{p(X)}_{\text{evidence}}}\,. \tag{4}$$

The posterior $p(Y|X)$ is the quantity of interest in Bayesian statistics because it expresses exactly what we are interested in (*i.e.* what we know about Y after having observed X).

Example 1: Suppose that we have 100 lasers. We know that 30 of these lasers are unstable and rest are stable lasers. Moreover we found out that 40 lasers, 10 of them are unstable, have poor fiber coupling. What is the probability for a laser to be unstable given that it has poor fiber coupling?

Answer: Let us define to events X and Y to be lasers with poor fiber coupling and unstable lasers, respectively. Then Bayes' theorem (4) states

$$p(\text{unstable}\,|\,\text{poor fiber coupling}) = \frac{p(\text{poor fiber coupling}\,|\,\text{unstable})\,p(\text{unstable})}{p(\text{poor fiber coupling})}\,. \tag{5}$$

Since 30 lasers are unstable, we have $p(\text{unstable}) = \frac{30}{100} = 0.30$. Moreover, 40 lasers have poor fiber coupling problem. Hence, $p(\text{poor fiber coupling}) = \frac{40}{100} = 0.40$. The likelihood can be easily calculated by noting that out of 40 unstable lasers, 10 of them have poor fiber coupling. Therefore, $p(\text{poor fiber coupling}\,|\,\text{unstable}) = \frac{10}{40} = 0.25$.

Putting things together, we find

$$p(\text{unstable}\,|\,\text{poor fiber coupling}) = \frac{0.25 \times 0.3}{0.4} = 0.1875\,. \tag{6}$$

This result means that if we get another batch of lasers, where 100 of them have poor fiber coupling problem, approximately 19 of those 100 lasers are expected to have a stability problem.[3]

Now that we have stated the Bayes' theorem, we can introduce the naive Bayes classifier. In here, we consider two major naive Bayes classifiers, namely the **classical naive Bayes classifier** and the **Gaussian naive Bayes classifier**. The former is used when the features are *categorical variables*, whereas the latter concerns the case where the features are *continuous*.

1.2.1 Classical Naive Bayes Classifier

Suppose we have d categorical features x_1, x_2, $\cdots$, x_d, and a categorical target variable y. We use $X = (x_1, x_2, \cdots, x_d)$ to denote the collection of the d categorical features. Moreover, suppose the total number of possible distinct outcomes for y is K (*i.e.* We are dealing with a classification

[3] Most of the time, when we deal with unstable lasers, the source of the problem would be poor construction or improper heat-sinking. Temperature or current related instabilities are also very common.

problem with K classes). We are interested in probability $p(y = k|X = X^*)$ of each class given that $X = X^*$, where X^* denotes the observed values of features. We would like to calculate these probabilities for all $k = 1, 2, \cdots, K$. According to Bayes' theorem, we have

$$p(y = k|X = X^*) = \frac{p(X = X^*|y = k)\, p(y = k)}{p(X = X^*)}\,. \tag{7}$$

After calculating $p(y = k|X = X^*)$ through (7), the predicted class, $\hat{y}$, associated with $X = X^*$ will be the one with highest probability. In other words, we have

$$\hat{y} = \underset{i\in\{1,2\cdots,K\}}{\text{argmax}} \frac{p(X = X^*|y = i)\, p(y = i)}{p(X = X^*)} = \underset{i\in\{1,2\cdots,K\}}{\text{argmax}} \left(p(X = X^*|y = i)\, p(y = i)\right). \tag{8}$$

Note that since the denominator in (8) is the same for all classes (*i.e.* it is independent of i), we can drop it, as it will have no role in finding the value of $\hat{y}$. We are now in position to explain what the term *naive* in naive Bayes refers to. In general, calculating the likelihood

$$p(X = X^*|y = i) = p(x_1 = x_1^* \cap x_2 = x_2^* \cap \cdots \cap x_d = x_d^*|y = i) \tag{9}$$

is a cumbersome task. However, the calculation can be significantly simplified under an assumption. The naive Bayes classifier assumes that $x_i = x_i^*$ and $x_j = x_j^*$ are **independent** events when $i \neq j$. Under this mutual independence assumption, the likelihood (9) can be simplified using (1)

$$p(X = X^*|y = i) = p(x_1 = x_1^*|y = i)\, p(x_2 = x_2^*|y = i) \cdots p(x_d = x_d^*|y = i) = \prod_{j=1}^{d} p(x_j = x_j^*|y = i)\,. \tag{10}$$

Substituting (10) into (8), we find the predicted class by the **naive classifier** to be

$$\hat{y} = \underset{i\in\{1,2\cdots,K\}}{\text{argmax}} \left(p(y = i) \prod_{j=1}^{d} p(x_j = x_j^*|y = i)\right). \tag{11}$$

As usual, in machine learning we prefer to minimize functions. This can be easily done by inserting a minus sign. Also, to get rid of the product, we can take the logarithm to ease the necessary calculation. In summary, we have

$$\boxed{\hat{y} = \underset{i\in\{1,2\cdots,K\}}{\text{argmin}} \; -\left[\log\left(p(y = i)\right) + \sum_{j=1}^{d} \log\left(p(x_j = x_j^*|y = i)\right)\right]}\,. \tag{12}$$

Remark: The naive assumption of the naive classifier (*i.e.* independence of features for assuming their values) is determined on case by case basis for each dataset. A priori, one cannot claim whether the naive assumption is valid before considering the dataset.

In order to illustrate how the classical naive classifier works in practice, we do the following simple example by hand.

Example 2:

Suppose we are given the following dataset in which there are 3 categorical features (x_1 = Stability, x_2 = Energy Efficiency, x_3 = Lifetime). The target variable is the binary categorical variable y = Buy. The possible outcomes for Stability are Stable, Unstable, and Stable at low powers, for Energy Efficiency are High, Medium, and Low, and for Lifetime are High and Medium. The possible outcome (*i.e.* classes) for the target Buy are Yes and No.

Device	Stability	Energy Efficiency	Lifetime	Buy
1	Stable	High	High	Yes
2	Stable	High	Medium	Yes
3	Stable	Low	Medium	No
4	Stable	Medium	High	Yes
5	Unstable	Medium	Medium	No
6	Unstable	High	High	No
7	Unstable	Low	Medium	No
8	Stable at low powers	High	High	No
9	Stable at low powers	High	Medium	Yes
10	Stable at low powers	Medium	Medium	No

Using the classical naive Bayes classifier and the dataset provided above, predict the class $\hat{y}$ of $X^* = (\text{Stable}, \text{Medium}, \text{Medium})$. Shall we buy this photonic device or not?

Answer: We can use formula (11) to solve this problem. First, note that we need to calculate the likelihoods of individual features for each class. This problem is a binary classification, and there are only two classes (Yes and No). Individual likelihoods can be easily calculated by counting frequencies in each class. The following table summarizes all individual likelihoods.

Features	$p(x_i\|Y)$	$p(x_i\|Y)$
Stability	Buy	Don't Buy
Stable	$\frac{3}{4}$	$\frac{1}{6}$
Unstable	$\frac{0}{4}$	$\frac{3}{6}$
Stable at low powers	$\frac{1}{4}$	$\frac{2}{6}$
Energy Efficiency	Buy	Don't Buy
Long	$\frac{3}{4}$	$\frac{2}{6}$
Medium	$\frac{1}{4}$	$\frac{2}{6}$
Low	$\frac{0}{4}$	$\frac{2}{6}$
Lifetime	Buy	Don't Buy
High	$\frac{2}{4}$	$\frac{2}{6}$
Medium	$\frac{2}{4}$	$\frac{4}{6}$

Now, we have to calculate the quantity inside argmax on the right hand side of (11) for $X^* = (\text{Stable}, \text{Medium}, \text{Medium})$, and pick the class that leads to the greater value:

$$\begin{aligned} p(y = \text{Yes}) \prod_{j=1}^{3} p(x_j = x_j^* | y = \text{Yes}) &= p(y = \text{Yes}) p(x_1 = \text{Stable} | y = \text{Yes}) \\ &\quad \times p(x_2 = \text{Medium} | y = \text{Yes}) p(x_3 = \text{Medium} | y = \text{Yes}) \\ &= \frac{4}{10} \times \frac{3}{4} \times \frac{1}{4} \times \frac{2}{4} = \frac{3}{80}, \\ p(y = \text{No}) \prod_{j=1}^{3} p(x_j = x_j^* | y = \text{No}) &= p(y = \text{No}) p(x_1 = \text{Stable} | y = \text{No}) \\ &\quad \times p(x_2 = \text{Medium} | y = \text{No}) p(x_3 = \text{Medium} | y = \text{No}) \\ &= \frac{6}{10} \times \frac{1}{6} \times \frac{2}{6} \times \frac{4}{6} = \frac{2}{90}. \end{aligned} \tag{13}$$

Since $\frac{3}{80} > \frac{2}{90}$, the argmax function in (11) picks the Yes class. Hence, for $X^* = (\text{Stable}, \text{Medium}, \text{Medium})$, the predicted class is $\hat{y} = \text{Yes}$, we can buy this product.

Question: How is the *independence* assumption of the naive Bayes classifier justified in this example?

1.2.2 Gaussian Naive Bayes Classifier

In the previous section (*i.e.* classical naive Bayes classifier), the features were all assumed to be categorical. However, it is often the case that the features of a classification problem are continuous variables. How can we promote the naive Bayes classifier to work with continuous features? This is the question we would like to answer in this section.

The very first technical point that we need to be aware of is the fact that probability functions in the case of continuous variables are replaced by **probability density functions**. In addition, when dealing with continuous variables, a typical assumption is that the continuous values associated with each class are distributed according to a *distribution*. As is indicated by its name, the Gaussian naive Bayes classifier assumes that each (continuous) feature follows a Gaussian (*i.e.* normal) distribution. What we discussed about the naive Bayes classifier in the previous section holds for the Gaussian naive Bayes classifier as well. In the context of the Gaussian naive Bayes classifier, a feature assuming a specific value has no effect on the probabilities of other features assuming their own values (*i.e.* the naive/independence assumption). Each (continuous) feature follows a Gaussian distribution, and the likelihood in the Bayes formula are calculated by

$$p(x_i = x_i^* | y = C_k) = \frac{1}{\sqrt{2\pi\,\sigma_{i,k}^2}} \exp\left(- \frac{(x_i^* - \mu_{i,k})^2}{2\,\sigma_{i,k}^2} \right), \tag{14}$$

where $\mu_{i,k}$ and $\sigma_{i,k}^2$ are the mean and the variance of the i-th feature in the k-th class, and they are explicitly given by

$$\mu_{i,k} = \frac{1}{n_k} \sum_{\{j|\, y^{(j)}=k\}} x_i^{(j)} \,,$$
$$\sigma_{i,k}^2 = \frac{1}{n_k - 1} \sum_{\{j|\, y^{(j)}=k\}} (x_i^{(j)} - \mu_{i,k})^2 \,, \tag{15}$$

where n_k is the size of the k-th class. To perform the Gaussian naive Bayes classification, one can then maximize the same quantity as (11) in which the individual likelihoods are calculated through (14).

Question: In either the classical or the Gaussian naive Bayes classifications, is it possible to give the probabilities (in addition to the predicted class) associated with each class, similar to the case of logistic regression?

Hint: To answer this question, look at equations (7) and (8) carefully!

1.3 LDA Classifier

Similar to the naive Bayes case, the Linear Discriminant Analysis (LDA) is based upon the Bayes' theorem in statistics. LDA classification method can be used when all features are continuous. Let $\vec{x} \in \mathbb{R}^d$ denote the features and y be the categorical variable with K classes. Rewriting the Bayes formula (7) the probability density function, we have

$$p(y = k|\vec{x} = \vec{x}^*) = \frac{p(y = k)\, p(\vec{x} = \vec{x}^*|y = k)}{\sum_{\ell=1}^{K} p(y = \ell)\, p(\vec{x} = \vec{x}^*|y = \ell)} \,, \tag{16}$$

where the denominator is nothing but $p(\vec{x} = \vec{x}^*)$. In the case of Gaussian naive Bayes classifier, we assumed that the d features, $\vec{x} \in \mathbb{R}^d$, are drawn from d Gaussian distribution, one corresponding to each feature x_i for $i = 1, 2, \cdots, d$. This originated from the crucial independence assumption we made. For LDA classifier, however, we do not assume that making observations on different features are independent of one another. Instead, we assume that the values for $\vec{x} \in \mathbb{R}^d$ are drawn from *one single multivariate Gaussian distribution*. The multivariate Gaussian distribution used in LDA classifier is given by

$$p(\vec{x} = \vec{x}^*|y = k) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\Big(-\frac{1}{2}(\mathrm{x}^* - \boldsymbol{\mu}_k)^\mathsf{T} \cdot \mathbf{\Sigma}^{-1} \cdot (\mathrm{x}^* - \boldsymbol{\mu}_k)\Big) \,, \tag{17}$$

where $\mathrm{x}^\mathsf{T} = (x_1\, x_2\, \cdots\, x_d)$, $\mathbf{\Sigma}$ is the $d \times d$ covariance matrix, and $\boldsymbol{\mu}_k$ is the mean vector associated with the k-th class

$$\boldsymbol{\mu}_k = \frac{1}{n_k} \sum_{\{j|\, y^{(j)}=k\}} \mathrm{x}^{(j)} \,, \tag{18}$$

with n_k being the size of the k-th class.

Remark: If the covariance matrix $\mathbf{\Sigma}$ is diagonal, then (17) reduces to the product of d separate single variate Gaussian distribution.

In order to predict the class of an observation, we must maximize (16) with (17) substituted. It can be shown in a straightforward manner that this problem is equivalent to maximizing a different, but simpler function $\delta_k(\vec{x})$ defined by

$$\delta_k(\vec{x}) = \mathrm{x}^{\mathsf{T}} \cdot \boldsymbol{\Sigma}^{-1} \cdot \boldsymbol{\mu}_k - \frac{1}{2}\boldsymbol{\mu}_k^{\mathsf{T}} \cdot \boldsymbol{\Sigma}^{-1} \cdot \boldsymbol{\mu}_k + \log(p(y=k)) \,. \tag{19}$$

In other words, LDA classifier assigns the class of an observation $\vec{x} = \vec{x}^*$ by

$$\boxed{\hat{y} = \underset{k\in\{1,2\cdots,K\}}{\operatorname{argmax}} \ \delta_k(\vec{x}^*)} \,. \tag{20}$$

Remark: Note that x appears linearly in $\delta_k(\vec{x})$ (see equation (19)). That is why the term linear appears in the name Linear Discriminant Analysis!

Warning: There exists an unsupervised algorithm with the same acronym, namely LDA which stands for *Latent Dirichlet Allocation*. Latent Dirichlet Allocation is an unsupervised algorithm that is primarily used for topic modeling. Make sure to recognize the correct algorithm from the context whenever encountering the acronym LDA!

1.4 QDA Classifier

As discussed in the previous section, the LDA classifier assumes that observations within each class are drawn from a multivariate Gaussian distribution with a class-specific mean and a *common covariance matrix for all K classes*. Quadratic Discriminant Analysis (QDA) offers an alternative approach. Similar to LDA in order to calculate likelihoods in the Bayes' theorem, the QDA classifier assumes that the observations from each class are drawn from a multivariate Gaussian distribution. However, unlike LDA, *QDA assumes that each class possesses its own covariance matrix*. In the context of QDA, we calculate K covariance matrices, one associated with each class, and equation (17) is replaced by

$$p(\vec{x} = \vec{x}^* | y = k) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}_k|^{1/2}} \exp\Big(-\frac{1}{2}(\mathrm{x}^* - \boldsymbol{\mu}_k)^{\mathsf{T}} \cdot \boldsymbol{\Sigma}_k^{-1} \cdot (\mathrm{x}^* - \boldsymbol{\mu}_k)\Big) \,. \tag{21}$$

In order to predict the class of an observation, we must maximize (16) with (21) substituted. It can be shown in a straightforward manner that this problem is equivalent to maximizing a different, but simpler function $\tilde{\delta}_k(\vec{x})$ defined by

$$\tilde{\delta}_k(\vec{x}) = -\frac{1}{2}(\mathrm{x} - \boldsymbol{\mu}_k)^{\mathsf{T}} \cdot \boldsymbol{\Sigma}_k^{-1} \cdot (\mathrm{x} - \boldsymbol{\mu}_k) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| + \log(p(y=k)) \,. \tag{22}$$

In other words, QDA classifier assigns the class of an observation $\vec{x} = \vec{x}^*$ by

$$\boxed{\hat{y} = \underset{k\in\{1,2\cdots,K\}}{\operatorname{argmax}} \ \tilde{\delta}_k(\vec{x}^*)} \,. \tag{23}$$

Remark: Note that x appears quadratically in $\tilde{\delta}_k(\vec{x})$ (see equation (22)). That is why the term quadratic appears in the name Quadratic Discriminant Analysis!

Question: How do you compare LDA and QDA? How should we decide which one is more appropriate to use for a given case? You will explore this question in your Homework 2!

2 Coding for Naive Bayes, LDA, and QDA

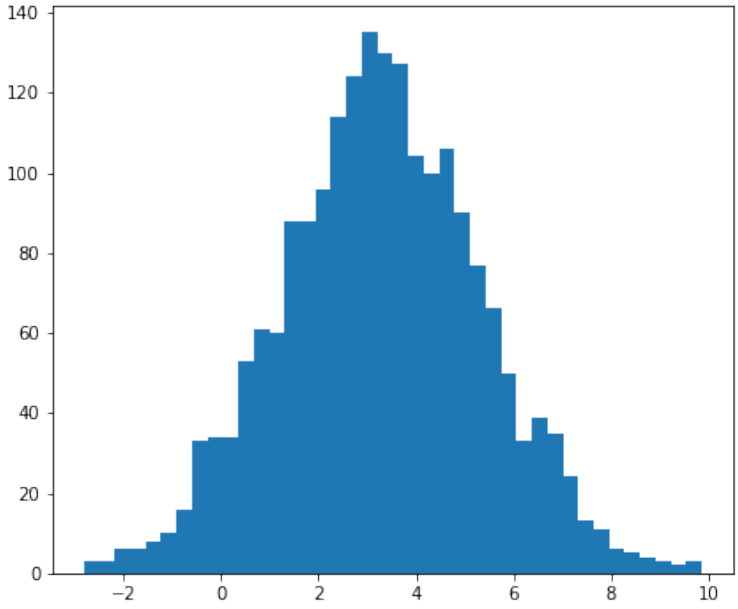
In here, we present three examples of classification problems to be solved by the algorithms we introduced in today's lecture (*i.e.* Naive Bayes, LDA, and QDA).

2.1 A Toy Dataset

In this example, we construct a dataset whose samples are drawn from Gaussian distributions for each continuous feature. The goal is to apply the Gaussian Naive Bayes classifier to this problem. We can easily employ the numpy built command np.random.normal(μ, σ, n) to generate a sample of size n from a Gaussian distribution with mean μ and standard deviation σ.

[1]:
```
# Importing basic libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

[2]:
```
# Defining a function to generate a sample from a gaussian distribution with␣
 ↪mean 'mu' and
# standard deviation 'sigma' with some random noise with strength␣
 ↪'noise_strength'

def gaussian_data(mu, sigma, size, noise_strength):
    noise = np.array([noise_strength*np.random.uniform(mu-5*sigma, mu+5*sigma)␣
 ↪for _ in range(size)])
    return np.random.normal(mu, sigma, size) + noise


fig, axs = plt.subplots(1, 1, figsize =(6, 5), tight_layout = True)

axs.hist(gaussian_data(3, 2, 2000, 0.1), bins = 40)  # Plotting the histogram of␣
 ↪the sample
plt.show()
```



The bell-shaped histogram above confirms that the sample is drawn from a Gaussian distribution. Now, we define a function to create a dataset with a number of features each of which is drawn from a Gaussian distribution. The categorical target variable of the dataset is defined by

$$y^{(i)} = \text{sign}\big((x_1^{(i)} - \mu_1)(x_2^{(i)} - \mu_2)\cdots(x_d^{(i)} - \mu_d)\big) = \text{sign}\Big(\prod_{j=1}^{d}(x_j^{(i)} - \mu_j)\Big)\,. \tag{24}$$

[3]:
```
# Defining the function to create the above described dataset

def gaussian_df(mu_list, sigma_list, size, noise_strength):
    df = pd.DataFrame()
    for i in range(len(mu_list)):
        df['x%d' %(i+1)] = gaussian_data(mu_list[i], sigma_list[i], size,␣
 ↪noise_strength)
    df['Class'] = 1
    for i in range(len(mu_list)):
        df['Class'] = df['Class'] * (df['x%d' %(i+1)] - mu_list[i])
    df['Class'] = df.Class.apply(lambda x: 1 if x>0 else -1)
    return df
```

[4]:
```
# Creating dataset 'toy_df' of size 2000 with 5 continuous features drawn from␣
 ↪gaussian distributions with
# means [1, 19, 42, 57, 79] and standard deviations [2, 1, 4, 1, 2]

mu_list = [1, 19, 42, 57, 79]
sigma_list = [2, 1, 4, 1, 2]
```

```
toy_df = gaussian_df(mu_list, sigma_list, 2000, 0.3)
toy_df.sample(5)
```

[4]:

	x1	x2	x3	x4	x5	Class
760	5.420327	24.564265	48.439731	74.784136	101.540775	1
1867	0.691078	26.468673	45.805522	72.964394	105.169581	-1
1922	-1.355500	23.199682	56.503149	77.049220	102.298742	-1
1436	1.913738	27.490268	52.182476	74.232957	100.511689	1
1506	-0.095893	23.434577	54.656859	72.128787	102.924426	-1

[5]:
```
# Finding the size of each class

print('The size of +1 Class:', len(toy_df[toy_df['Class']==1]))
print('The size of -1 Class:', len(toy_df[toy_df['Class']==-1]))
```

```
The size of +1 Class: 1044
The size of -1 Class: 956
```

The above sizes show that we deal with a fairly *balanced* binary classification problem.

[6]:
```
# Defining the features and the target of the model

X = toy_df[toy_df.columns[:-1]].values      # Features
y = toy_df[toy_df.columns[-1]].values       # Target
```

[7]:
```
# Breaking the data into train and test subsets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=3)
```

2.1.1 Gaussian Naive Bayes Classifier

[8]:
```
# Importing 'GaussianNB' from naive_bayes module

from sklearn.naive_bayes import GaussianNB

gnb_clf = GaussianNB()                # Instantiating the gaussian naive Bayes␣
 ↪classifier
gnb_clf.fit(X_train, y_train)         # Fitting the training data
```

[8]:
```
GaussianNB()
```

[9]:
```
# Finding the predictions of the model for the train and test subsets
```

```
train_y_pred = gnb_clf.predict(X_train)
test_y_pred = gnb_clf.predict(X_test)
```

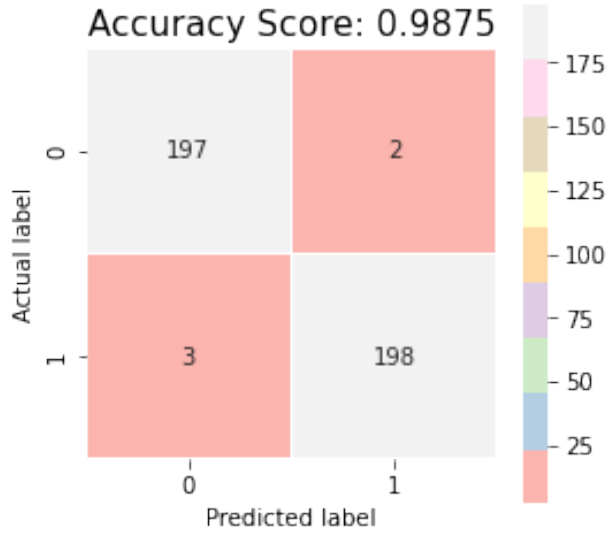
[10]:
```
# Computing various classification evaluation metrics and presenting the␣
 ↪classification report

from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report

train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Gaussian NB Train Classification Report: \n\n', train_report,'\n\n')
print('Gaussian NB Test Classification Report: \n\n', test_report)
```

```
Gaussian NB Train Classification Report:

              precision    recall  f1-score   support
          -1       0.97      0.99      0.98       757
           1       0.99      0.97      0.98       843

    accuracy                           0.98      1600
   macro avg       0.98      0.98      0.98      1600
weighted avg       0.98      0.98      0.98      1600


Gaussian NB Test Classification Report:

              precision    recall  f1-score   support
          -1       0.98      0.99      0.99       199
           1       0.99      0.99      0.99       201

    accuracy                           0.99       400
   macro avg       0.99      0.99      0.99       400
weighted avg       0.99      0.99      0.99       400
```

The above report indicates a strong classifier with no signs of overfitting!

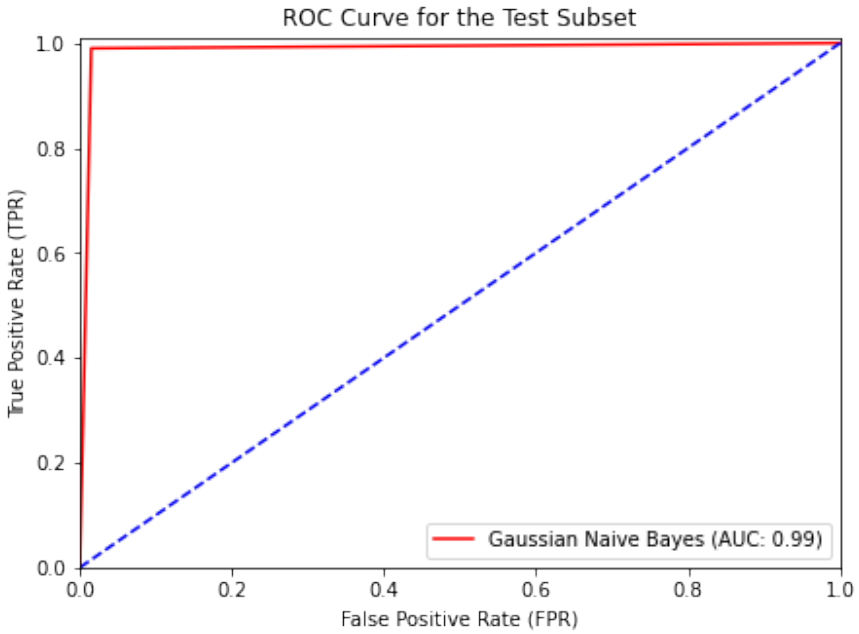
[11]:
```
# Computing the confusion matrix

gnb_c_matrix = confusion_matrix(y_test, test_y_pred)
```

```
plt.figure(figsize=(4,4))
sns.heatmap(gnb_c_matrix, annot=True, fmt=".0f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()
```

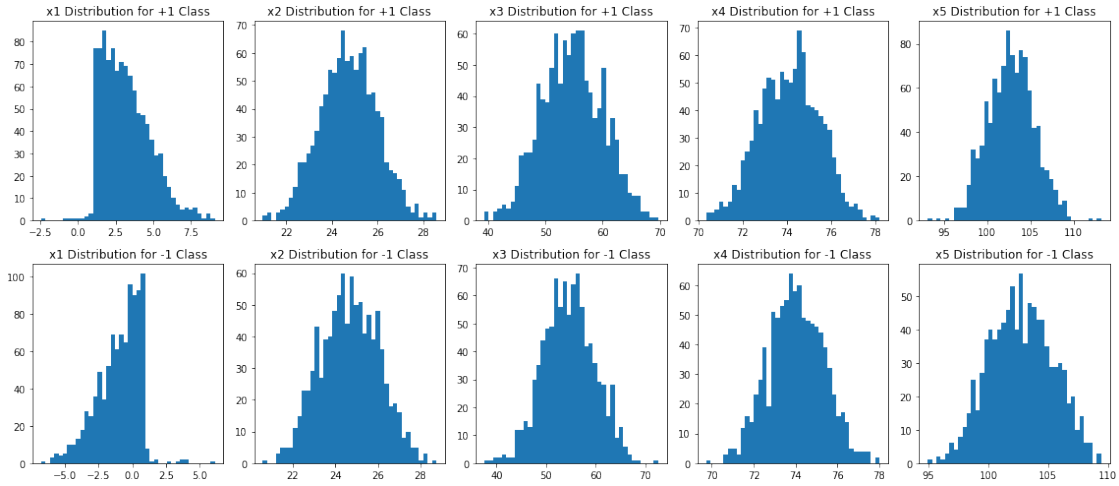


[12]:
```
# Plotting the ROC curve for the test subset

from sklearn.metrics import roc_curve, auc   # Importing 'roc_curve' and 'auc'␣
 ↪from sklearn

fpr, tpr, thresholds = roc_curve(test_y_pred, y_test)  # Computing ROC for the␣
 ↪test subset
auc(fpr, tpr)                                          # Computing AUC for the␣
 ↪test subset

plt.figure(figsize=(7, 5))
plt.plot(fpr, tpr, color='red', label='Gaussian Naive Bayes (AUC: %.2f)'
% auc(fpr, tpr))
plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.01])
plt.title('ROC Curve for the Test Subset')
plt.xlabel('False Positive Rate (FPR)')
```

```
plt.ylabel('True Positive Rate (TPR)')
plt.legend()
plt.show()
```



[13]:
```
# Checking the distribution of each feature in each class (+1 and -1 classes)

fig, axs = plt.subplots(2, len(mu_list), figsize=(16, 7), tight_layout=True)
for j in range(len(mu_list)):
    x_plus_arr = toy_df[toy_df['Class']==1]['x%d' %(j+1)].values
    x_minus_arr = toy_df[toy_df['Class']==-1]['x%d' %(j+1)].values
    axs[0, j].hist(x_plus_arr, bins = 40)
    axs[0, j].set_title('x%d Distribution for +1 Class' %(j+1))
    axs[1, j].hist(x_minus_arr, bins = 40)
    axs[1, j].set_title('x%d Distribution for -1 Class' %(j+1))
plt.show()
```



The above histograms show that the features follow a Gaussian distribution in each class.

2.1.2 Logistic Regression Classifier

It would be interesting to compare the performance of the naive Bayes classifier above with the performance of the logistic regression.

[14]:
```
# Converting classes from -1 and 1 to 0 and 1
# (Note for binary logistic regression, the classes must be labeled by 0 and 1)

y_train = np.where(y_train!=1, 0, y_train)
y_test = np.where(y_test!=1, 0, y_test)
```

[15]:
```
from sklearn.linear_model import LogisticRegression    # Importing Logistic␣
 ↪Regression from sklearn

logreg = LogisticRegression(penalty = 'none', max_iter = 10000) # Instantiating␣
 ↪logistic regression

logreg.fit(X_train, y_train)  # Fitting the train data to 'logreg'

# Finding predictions of the logistic regression model for train and test subsets
train_y_pred = logreg.predict_proba(X_train).argmax(axis=1)
test_y_pred = logreg.predict_proba(X_test).argmax(axis=1)
```

[16]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
```

```
test_report = classification_report(y_test, test_y_pred)      # Generate␣
 ↪classification report for test data

print('Logistic Regression Train Classification Report: \n\n',␣
 ↪train_report,'\n\n')
print('Logistic Regression Test Classification Report: \n\n', test_report)
```

```
Logistic Regression Train Classification Report:

              precision    recall  f1-score   support
           0       0.98      0.99      0.98       757
           1       0.99      0.98      0.99       843

    accuracy                           0.98      1600
   macro avg       0.98      0.98      0.98      1600
weighted avg       0.98      0.98      0.98      1600



Logistic Regression Test Classification Report:

              precision    recall  f1-score   support
           0       0.99      0.98      0.99       199
           1       0.99      1.00      0.99       201

    accuracy                           0.99       400
   macro avg       0.99      0.99      0.99       400
weighted avg       0.99      0.99      0.99       400
```

As observed above, logistic regression performs pretty strong as well!

2.1.3 A Tiny Change in the Dataset

Let us make a tiny change in the toy datset, and rerun the Gaussian naive Bayes classifier!

[17]:
```
# Only the middle element of mu_list has changed (from 42 to 2)

mu_list_2 = [1, 19, 2, 57, 79]
sigma_list_2 = [2, 1, 4, 1, 2]

toy_df_2 = gaussian_df(mu_list_2, sigma_list_2, 2000, 0.3)
toy_df_2.sample(5)
```

[17]:

	x1	x2	x3	x4	x5	Class
157	5.029451	24.713040	3.565125	75.481771	104.132930	1
1973	3.084532	25.961450	13.942581	76.217394	98.161303	1
179	-0.090687	24.513726	4.943059	72.678270	101.853009	-1

```
913   3.386350  25.292237   1.416019  71.416674   99.772383     -1
1306  0.114479  26.885134   1.470550  72.455226  101.864316      1
```

[18]:
```
# Checking the size of each class

print('The size of +1 Class:', len(toy_df_2[toy_df_2['Class']==1]))
print('The size of -1 Class:', len(toy_df_2[toy_df_2['Class']==-1]))
```

```
The size of +1 Class: 1007
The size of -1 Class: 993
```

[19]:
```
# Defining the features and the target of the model

X = toy_df_2[toy_df_2.columns[:-1]].values      # Features
y = toy_df_2[toy_df_2.columns[-1]].values       # Target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=3)
```

[20]:
```
gnb_clf = GaussianNB()                # Instantiating the gaussian naive Bayes␣
 ↪classifier
gnb_clf.fit(X_train, y_train)         # Fitting the training data
```

[20]:
```
GaussianNB()
```

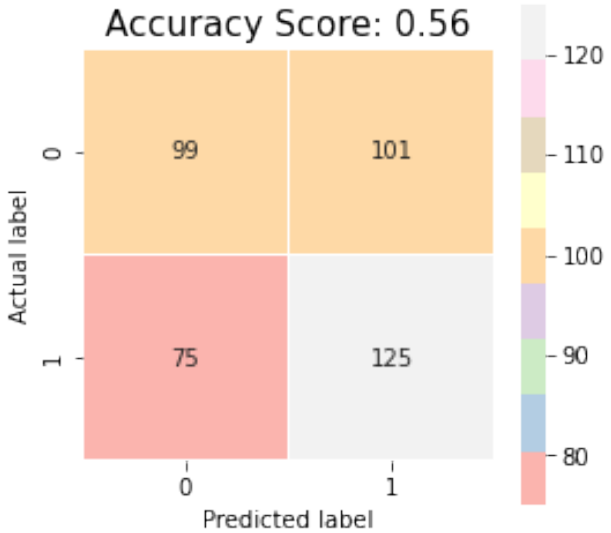
[21]:
```
# Finding the predictions of the model for the train and test subsets

train_y_pred = gnb_clf.predict(X_train)
test_y_pred = gnb_clf.predict(X_test)
```

[22]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Train Classification Report: \n\n', train_report,'\n\n')
print('Test Classification Report: \n\n', test_report)
```

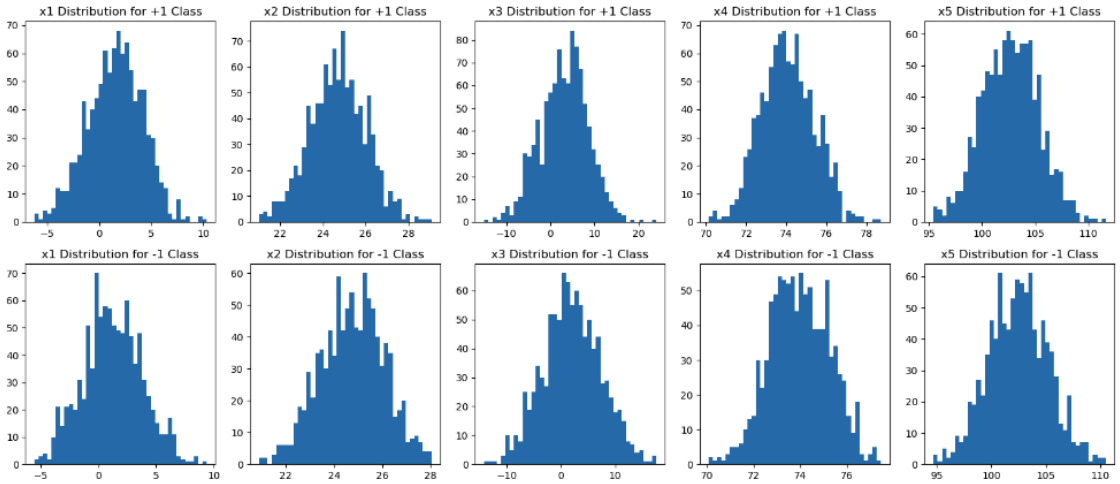
```
Train Classification Report:

              precision    recall  f1-score   support

          -1       0.57      0.52      0.55       793
           1       0.57      0.61      0.59       807
```

```
    accuracy                           0.57      1600
   macro avg       0.57      0.57      0.57      1600
weighted avg       0.57      0.57      0.57      1600


Test Classification Report:

              precision    recall  f1-score   support
          -1       0.57      0.49      0.53       200
           1       0.55      0.62      0.59       200

    accuracy                           0.56       400
   macro avg       0.56      0.56      0.56       400
weighted avg       0.56      0.56      0.56       400
```

[23]:
```
# Computing the confusion matrix

gnb_c_matrix = confusion_matrix(y_test, test_y_pred)

plt.figure(figsize=(4,4))
sns.heatmap(gnb_c_matrix, annot=True, fmt=".0f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()
```



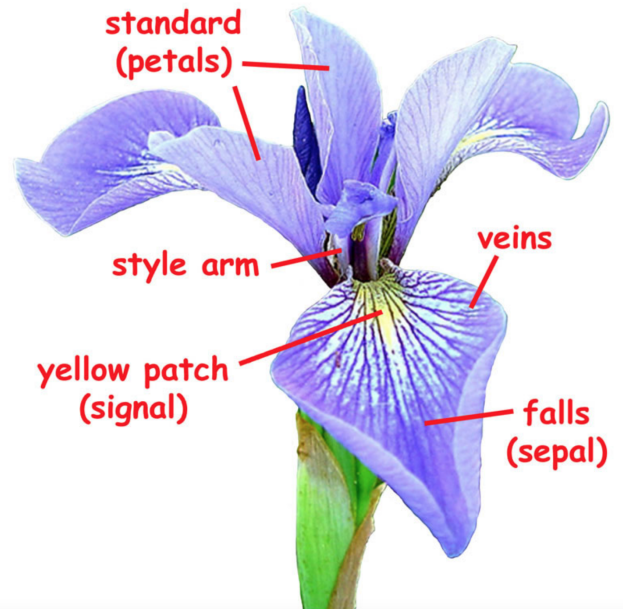
[24]:
```
# Checking the distribution of each feature in each class (+1 and -1 classes)

fig, axs = plt.subplots(2, len(mu_list_2), figsize=(16, 7), tight_layout=True)
for j in range(len(mu_list_2)):
    x_plus_arr = toy_df_2[toy_df_2['Class']==1]['x%d' %(j+1)].values
    x_minus_arr = toy_df_2[toy_df_2['Class']==-1]['x%d' %(j+1)].values
    axs[0, j].hist(x_plus_arr, bins = 40)
    axs[0, j].set_title('x%d Distribution for +1 Class' %(j+1))
    axs[1, j].hist(x_minus_arr, bins = 40)
    axs[1, j].set_title('x%d Distribution for -1 Class' %(j+1))
plt.show()
```



Question: What happened?! Why did the performance of the Gaussian Naive Bayes Classifier become so weak (more or less at the level of the performance of the monkey classifier) under this change?

2.2 Iris Dataset

This example uses a famous dataset which aims to classify different types of iris flower based on the sizes of sepals and petals. This dataset already exists among scikit-learn datasets.



[25]:
```
# Loading isis dataset
from sklearn.datasets import load_iris

iris = load_iris()
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------
**Data Set Characteristics:**
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:
    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
```

```
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

[26]:
```
# Storing iris dataset in a dataframe

iris_df = pd.DataFrame(iris.data)       # Defining dataframe boston_df
iris_df.columns = iris.feature_names    # Defining the headers of the dataframe
iris_df['Class'] = iris.target          # Adding Price column

iris_df.sample(5)                       # Viewing the first few rows of dataframe
```

[26]:
```
    sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
95                5.7               3.0                4.2               1.2
68                6.2               2.2                4.5               1.5
45                4.8               3.0                1.4               0.3
94                5.6               2.7                4.2               1.3
71                6.1               2.8                4.0               1.3

    Class
95      1
68      1
45      0
94      1
71      1
```

[27]:
```
# Finding the sizes of the classes

print('Size of class 0:', len(iris_df[iris_df['Class']==0]))
print('Size of class 1:', len(iris_df[iris_df['Class']==1]))
print('Size of class 2:', len(iris_df[iris_df['Class']==2]))
print('Size of the dataset:', len(iris_df))
```

```
Size of class 0: 50
Size of class 1: 50
Size of class 2: 50
Size of the dataset: 150
```

As observed above, the iris classification problem is *perfectly balanced*.

[28]:
```
# Defining the features and the target of the model

X = iris_df[iris_df.columns[:-1]].values     # Features
y = iris_df[iris_df.columns[-1]].values      # Target
```

[29]:
```
# Breaking the data into train and test subsets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,␣
 ↪random_state=3)
```

[30]:
```
# Loading GaussianNB from naive_bayes module

from sklearn.naive_bayes import GaussianNB

gnb_clf = GaussianNB()            # Instantiating the gaussian naive Bayes␣
 ↪classifier
gnb_clf.fit(X_train, y_train)     # Fitting the training data
```

[30]:
```
GaussianNB()
```

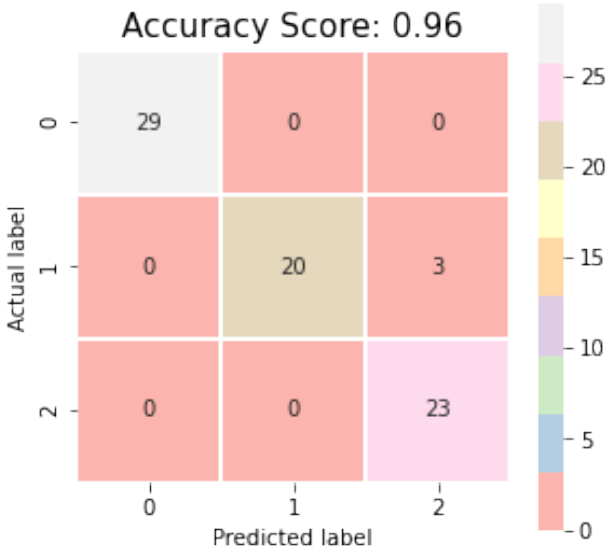
[31]:
```
# Finding predictions of the gaussian naive Bayes classifier for the train and␣
 ↪test subsets

train_y_pred = gnb_clf.predict(X_train)
test_y_pred = gnb_clf.predict(X_test)
```

[32]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Gaussian NB Train Classification Report: \n\n', train_report,'\n\n')
print('Gaussian NB Test Classification Report: \n\n', test_report)
```

```
Gaussian NB Train Classification Report:

              precision    recall  f1-score   support
           0       1.00      1.00      1.00        21
           1       0.89      0.93      0.91        27
           2       0.92      0.89      0.91        27

    accuracy                           0.93        75
   macro avg       0.94      0.94      0.94        75
weighted avg       0.93      0.93      0.93        75



Gaussian NB Test Classification Report:

              precision    recall  f1-score   support
           0       1.00      1.00      1.00        29
           1       1.00      0.87      0.93        23
           2       0.88      1.00      0.94        23

    accuracy                           0.96        75
   macro avg       0.96      0.96      0.96        75
weighted avg       0.96      0.96      0.96        75
```

As observed from above reports, the Gaussian naive Bayes classifier performs pretty strong.

[33]:
```
# Computing the confusion matrix

gnb_c_matrix = confusion_matrix(y_test, test_y_pred)

plt.figure(figsize=(4.5,4.5))
sns.heatmap(gnb_c_matrix, annot=True, fmt=".0f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()
```



[34]:
```
# Calculating the probabilities associated with test instances

gnb_clf.predict_proba(X_test[:10])
```

[34]:
```
array([[1.00000000e+000, 9.61557332e-020, 5.34866299e-023],
       [1.00000000e+000, 9.07018600e-019, 2.46887826e-022],
       [1.00000000e+000, 5.78673006e-016, 7.89931955e-020],
       [1.00000000e+000, 2.30426387e-017, 1.89268416e-021],
       [1.00000000e+000, 2.35111912e-018, 1.33057651e-020],
       [2.63956449e-295, 6.35814342e-017, 1.00000000e+000],
       [8.13504889e-066, 9.99913715e-001, 8.62849574e-005],
       [1.00000000e+000, 4.14719870e-019, 2.41468874e-022],
       [3.96186457e-165, 2.55209029e-004, 9.99744791e-001],
       [2.83453419e-088, 9.57210759e-001, 4.27892405e-002]])
```

3 Laser Stability Dataset

In this dataset we have 3531 lasers. The output power of the laser beam intensity is measured T hours after it starts lasing, where T = 2, 20, 40, 60 hours. We also know whether is made with material 1 or 2 and whether it has a heatsink or not. The last column indicates whether the laser still outputs a stable beam after 1000 hours of operation. Our goal is predicting this outcome from the short-term measuments (the first four column of the dataset), material type, and existence of heatsink.

[1]:
```
# Importing the libraries that will be used in this notebook
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
```

[2]:
```
# Loading the dataset
df = pd.read_csv('https://raw.githubusercontent.com/simsekergun/ENEE691/main/
 ↪week08/laser_stability.csv')     # Reading the csv source file as a dataframe
df.shape
```

[2]:
```
(3531, 7)
```

[3]:
```
# Displaying a sample of 5 rows
df.sample(5)
```

[3]:

	2h	20h	40h	60h	material	heatsink	StableOrNot
2428	1.0001	0.99853	0.98988	0.99393	2	0	0
3060	1.0028	1.00070	1.00540	0.99974	1	1	1
317	1.0025	0.99409	1.00140	0.99897	1	1	1
2467	1.0026	1.00020	1.00440	1.00380	1	1	1
2423	1.0001	0.99915	1.00400	1.00240	1	1	1

[4]:
```
# Finding the size of each class
print('Size of class 0:', len(df[df.StableOrNot==0]))
print('Size of class 1:', len(df[df.StableOrNot==1]))
print('Size of the dataset:', len(df))
```

```
Size of class 0: 1847
Size of class 1: 1684
Size of the dataset: 3531
```

[5]:
```
# Breaking features to continuous and categorical ones
cont_features = ['2h', '20h', '40h', '60h']
cat_features = ['material', 'heatsink']
```

3.1 LDA Classifier

We first perform classification employing the LDA classifier as follows.

[6]:
```
# Defining the continuous features and the target of the model
X = df[cont_features].values      # Features
y = df.StableOrNot.values        # Target
```

[7]:
```
# Breaking the data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=3)
```

[8]:
```
lda_clf = LinearDiscriminantAnalysis()      # Instantiating the lda classifier
lda_clf.fit(X_train, y_train)               # Fitting the train data
```

[8]:
```
LinearDiscriminantAnalysis()
```

[9]:
```
# Finding the predictions of the lda classifier for train and test subsets

train_y_pred = lda_clf.predict(X_train)
test_y_pred = lda_clf.predict(X_test)
```

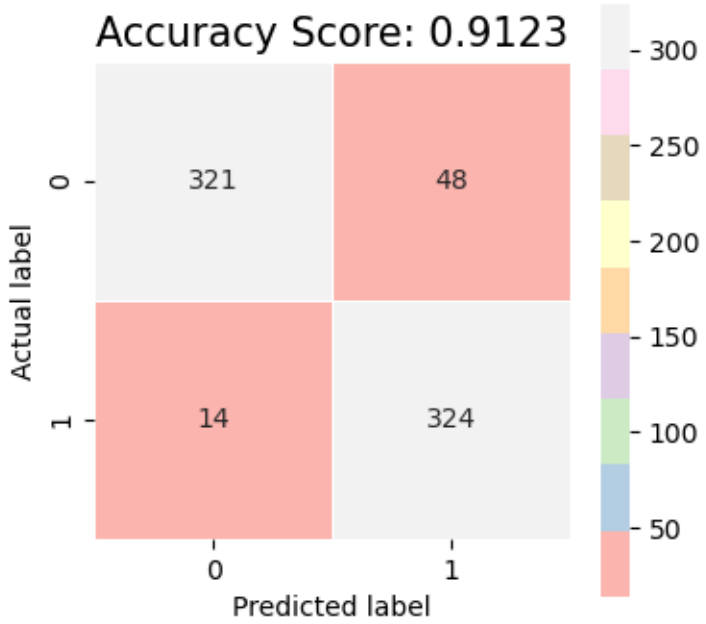
[10]:
```
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report

train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('LDA Train Classification Report: \n\n', train_report,'\n\n')
print('LDA Test Classification Report: \n\n', test_report)
```

```
LDA Train Classification Report:

              precision    recall  f1-score   support

           0       0.95      0.88      0.91      1478
           1       0.88      0.95      0.91      1346
```

```
    accuracy                           0.91      2824
   macro avg       0.91      0.91      0.91      2824
weighted avg       0.91      0.91      0.91      2824
```

```
LDA Test Classification Report:

              precision    recall  f1-score   support

           0       0.96      0.87      0.91       369
           1       0.87      0.96      0.91       338

    accuracy                           0.91       707
   macro avg       0.91      0.91      0.91       707
weighted avg       0.92      0.91      0.91       707
```

[11]:
```
# Computing lda confusion matrix
lda_c_matrix = confusion_matrix(y_test, test_y_pred)

plt.figure(figsize=(4,4))
sns.heatmap(lda_c_matrix, annot=True, fmt=".0f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()
```

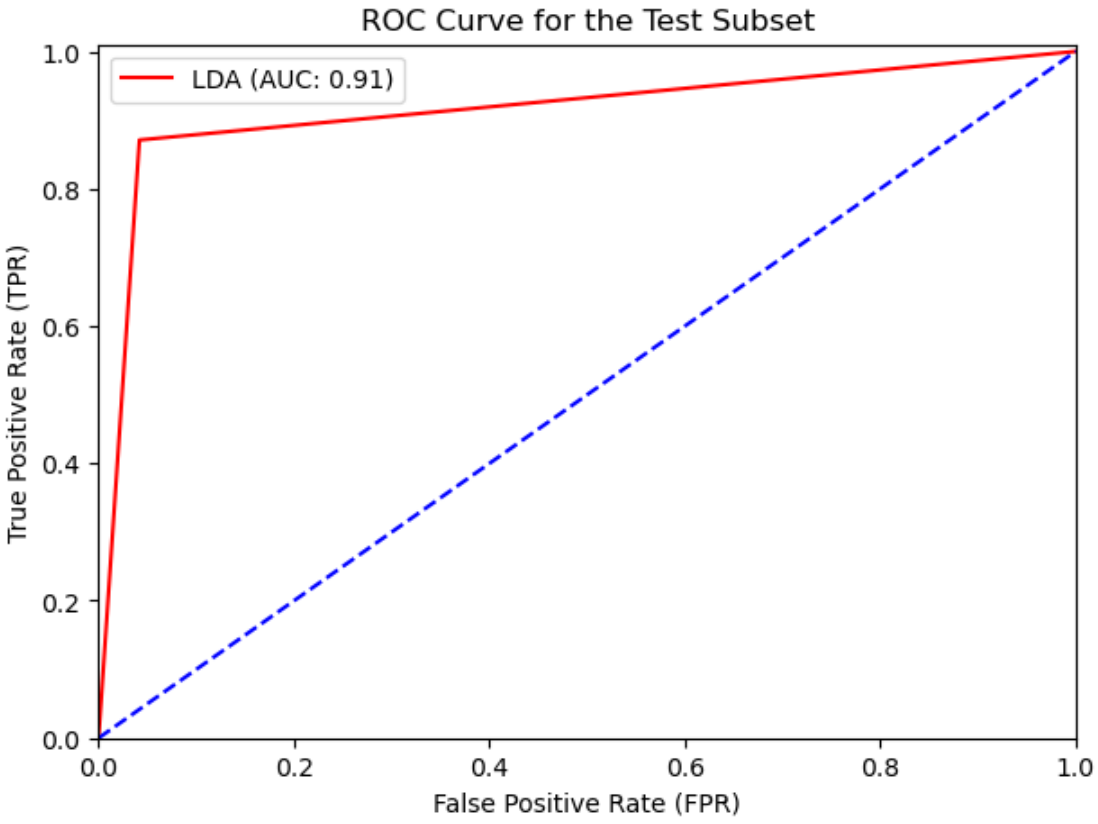


[12]:
```
# Finding probabilities of test instances predicted by lda classifier
lda_clf.predict_proba(X_test[:10])
```

[12]:
```
array([[9.77944127e-01, 2.20558730e-02],
       [5.91543659e-02, 9.40845634e-01],
       [9.75714414e-01, 2.42855859e-02],
       [9.66934640e-01, 3.30653603e-02],
       [6.91993288e-03, 9.93080067e-01],
       [5.61740703e-02, 9.43825930e-01],
       [9.99715437e-01, 2.84562707e-04],
       [7.60215373e-01, 2.39784627e-01],
       [3.59803802e-01, 6.40196198e-01],
       [4.42063116e-02, 9.55793688e-01]])
```

[13]:
```
# Plotting the ROC curve for the test subset
fpr, tpr, thresholds = roc_curve(test_y_pred, y_test)  # Computing ROC for the␣
 ↪test subset
auc(fpr, tpr)                                          # Computing AUC for the␣
 ↪test subset

plt.figure(figsize=(7, 5))
plt.plot(fpr, tpr, color='red', label='LDA (AUC: %.2f)'
% auc(fpr, tpr))
plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.01])
```

```
plt.title('ROC Curve for the Test Subset')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend()
plt.show()
```



The performance of LDA classifier is pretty strong. No sings of overfitting.

3.2 QDA Classifier

Now, we perform classification employing the QDA classifier (for the continuous features).

[14]:
```
qda_clf = QuadraticDiscriminantAnalysis()       # Instantiating the QDA classifier
qda_clf.fit(X_train, y_train)                   # Fitting the training data
```

[14]:
```
QuadraticDiscriminantAnalysis()
```

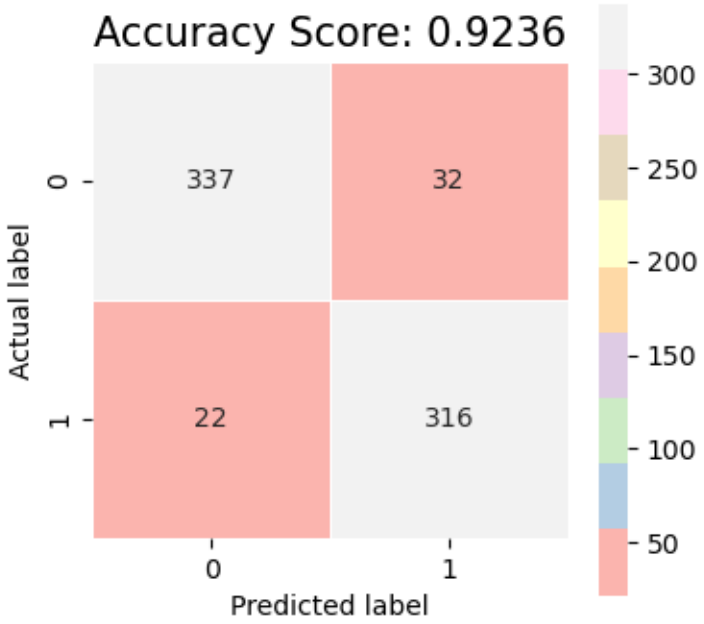
[15]:
```
# Finding predictions of qda classifier for the train and test subsets

train_y_pred = qda_clf.predict(X_train)
test_y_pred = qda_clf.predict(X_test)
```

[16]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('QDA Train Classification Report: \n\n', train_report,'\n\n')
print('QDA Test Classification Report: \n\n', test_report)
```

```
QDA Train Classification Report:

              precision    recall  f1-score   support

           0       0.95      0.91      0.93      1478
           1       0.90      0.94      0.92      1346

    accuracy                           0.92      2824
   macro avg       0.92      0.93      0.92      2824
weighted avg       0.93      0.92      0.92      2824



QDA Test Classification Report:

              precision    recall  f1-score   support

           0       0.94      0.91      0.93       369
           1       0.91      0.93      0.92       338

    accuracy                           0.92       707
   macro avg       0.92      0.92      0.92       707
weighted avg       0.92      0.92      0.92       707
```

[17]:
```
# Computing qda confusion matrix

qda_c_matrix = confusion_matrix(y_test, test_y_pred)

plt.figure(figsize=(4,4))
sns.heatmap(qda_c_matrix, annot=True, fmt=".0f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(round(test_score, 4))
```

```
plt.title(all_sample_title, size = 15);
plt.show()
```

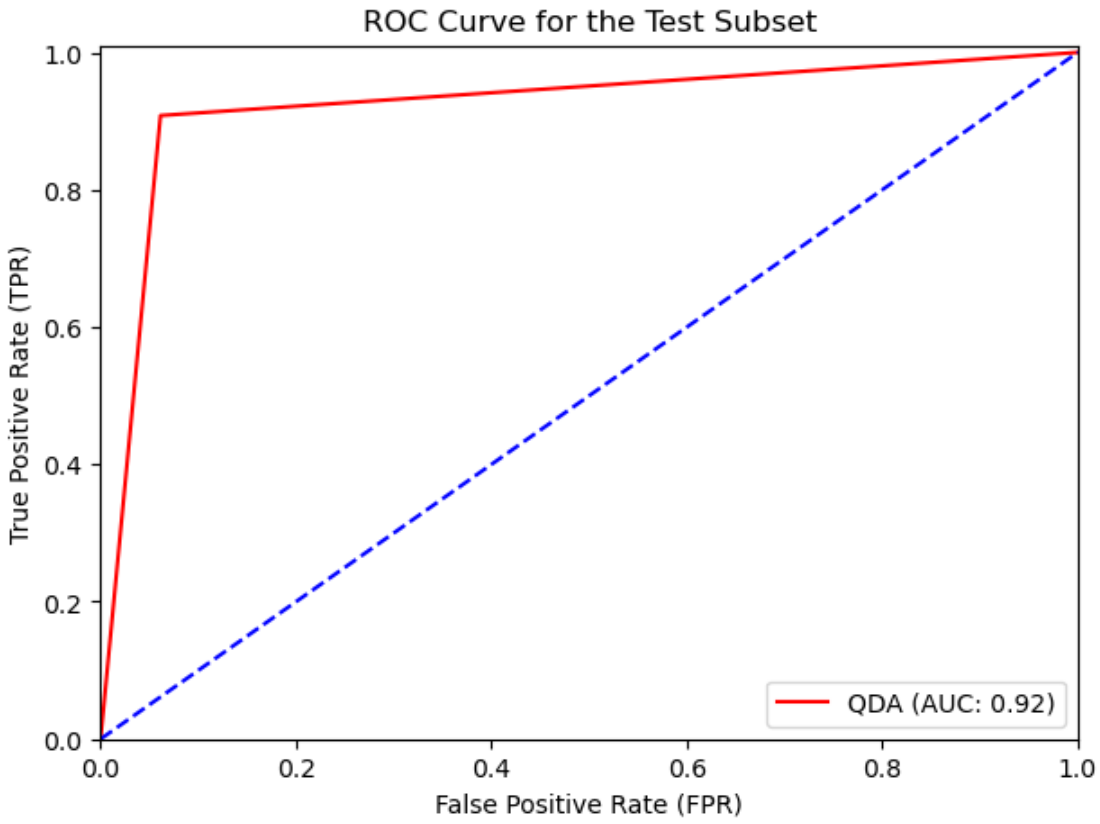


[18]:
```
# Finding probabilities of test instances predicted by qda classifier

qda_clf.predict_proba(X_test[:10])
```

[18]:
```
array([[9.99998438e-01, 1.56220877e-06],
       [1.13525268e-02, 9.88647473e-01],
       [9.98291193e-01, 1.70880651e-03],
       [9.92462672e-01, 7.53732794e-03],
       [1.91054345e-02, 9.80894566e-01],
       [5.40437832e-03, 9.94595622e-01],
       [9.99999999e-01, 1.33907772e-09],
       [9.27795160e-01, 7.22048401e-02],
       [4.87395104e-01, 5.12604896e-01],
       [2.16468534e-02, 9.78353147e-01]])
```

[19]:
```
# Plotting the ROC curve for the test subset

from sklearn.metrics import roc_curve, auc    # Importing 'roc_curve' and 'auc'␣
 ↪from sklearn

fpr, tpr, thresholds = roc_curve(test_y_pred, y_test)  # Computing ROC for the␣
 ↪test subset
```

```
auc(fpr, tpr)                                                    # Computing AUC for the␣
 ↪test subset

plt.figure(figsize=(7, 5))
plt.plot(fpr, tpr, color='red', label='QDA (AUC: %.2f)'
% auc(fpr, tpr))
plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.01])
plt.title('ROC Curve for the Test Subset')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend()
plt.show()
```



QDA performs slightly better than LDA. Again, no overfitting is observed.

3.3 Logistic Regression

We now apply logistic regression to the same classification problem to compare the results.

[20]:
```
from sklearn.linear_model import LogisticRegression   # Importing Logistic␣
 ↪Regression from sklearn

logreg = LogisticRegression(penalty = 'none', max_iter = 10000) # Instantiating␣
 ↪logistic regression

logreg.fit(X_train, y_train)  # Fitting the train data to 'logreg'

train_y_pred = logreg.predict_proba(X_train).argmax(axis=1)  # Predicting the␣
 ↪class for train set
test_y_pred = logreg.predict_proba(X_test).argmax(axis=1)    # Predicting the␣
 ↪class for test set
```

[21]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)   # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)      # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)   # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)      # Generate␣
 ↪classification report for test data

print('Logistic Regression Train Classification Report: \n\n',␣
 ↪train_report,'\n\n')
print('Logistic Regression Test Classification Report: \n\n', test_report)
```

```
Logistic Regression Train Classification Report:

              precision    recall  f1-score   support

           0       0.93      0.91      0.92      1478
           1       0.90      0.92      0.91      1346

    accuracy                           0.91      2824
   macro avg       0.91      0.91      0.91      2824
weighted avg       0.91      0.91      0.91      2824



Logistic Regression Test Classification Report:

              precision    recall  f1-score   support

           0       0.93      0.92      0.92       369
           1       0.91      0.92      0.91       338

    accuracy                           0.92       707
```

```
   macro avg       0.92      0.92      0.92       707
weighted avg       0.92      0.92      0.92       707
```

The overall performance of the logistic regression is not very different than the LDA or QDA classifier.

3.4 Gaussian Naive Bayes Classifier

We now apply Gaussian naive Bayes to the same classification problem to compare the results.

[22]:
```
gnb_clf = GaussianNB()              # Instantiating the gaussian naive Bayes␣
 ↪classifier
gnb_clf.fit(X_train, y_train)      # Fitting the training data
```

[22]:
```
GaussianNB()
```

[23]:
```
# Finding the predictions of gaussian naive Bayes for train and test subsets

train_y_pred = gnb_clf.predict(X_train)
test_y_pred = gnb_clf.predict(X_test)
```

[24]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Gaussian NB Train Classification Report: \n\n', train_report,'\n\n')
print('Gaussian NB Test Classification Report: \n\n', test_report)
```

```
Gaussian NB Train Classification Report:

              precision    recall  f1-score   support

           0       0.92      0.88      0.90      1478
           1       0.87      0.92      0.90      1346

    accuracy                           0.90      2824
   macro avg       0.90      0.90      0.90      2824
weighted avg       0.90      0.90      0.90      2824



Gaussian NB Test Classification Report:
```

```
                precision    recall  f1-score   support

           0       0.92      0.89      0.91       369
           1       0.89      0.91      0.90       338

    accuracy                           0.90       707
   macro avg       0.90      0.90      0.90       707
weighted avg       0.90      0.90      0.90       707
```

The performance of Gaussian NB is slightly poorer than others.

3.5 Multinomial Naive Bayes Classifier

Now, we would like to see what we can learn from the categorical features. We apply the classical naive Bayes classifier to the heart disease dataset with categorical features.

[25]:
```
# Defining the features and the target of the model
X = df[cat_features].values      # Features
y = df.StableOrNot.values       # Target
```

[26]:
```
# Breaking the data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=3)
```

[27]:
```
mnb_clf = MultinomialNB()         # Instantiating the multinomial naive Bayes␣
 ↪classifier
mnb_clf.fit(X_train, y_train)     # Fitting the training data
```

[27]:
```
MultinomialNB()
```

[28]:
```
# Finding the predictions of multinomial naive Bayes for train and test subsets
train_y_pred = mnb_clf.predict(X_train)
test_y_pred = mnb_clf.predict(X_test)
```

[29]:
```
train_score = metrics.accuracy_score(y_train, train_y_pred)    # Compute train␣
 ↪accuracy
test_score = metrics.accuracy_score(y_test, test_y_pred)       # Compute test␣
 ↪accuracy
train_report = classification_report(y_train, train_y_pred)    # Generate␣
 ↪classification report for train data
test_report = classification_report(y_test, test_y_pred)       # Generate␣
 ↪classification report for test data

print('Multinomial NB Train Classification Report: \n\n', train_report,'\n\n')
print('Multinomial NB Test Classification Report: \n\n', test_report)
```

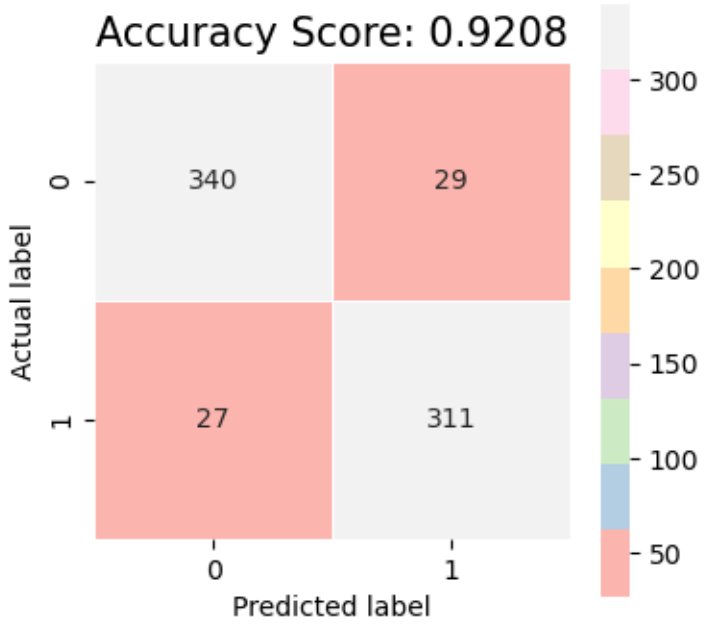
```
Multinomial NB Train Classification Report:

              precision    recall  f1-score   support

           0       0.91      0.89      0.90      1478
           1       0.89      0.90      0.89      1346

    accuracy                           0.90      2824
   macro avg       0.90      0.90      0.90      2824
weighted avg       0.90      0.90      0.90      2824



Multinomial NB Test Classification Report:

              precision    recall  f1-score   support

           0       0.93      0.92      0.92       369
           1       0.91      0.92      0.92       338

    accuracy                           0.92       707
   macro avg       0.92      0.92      0.92       707
weighted avg       0.92      0.92      0.92       707
```

[30]:
```
# Finding probabilities of test instances predicted by multinomial naive Bayes␣
 ↪classifier
mnb_clf.predict_proba(X_test[:10])
```

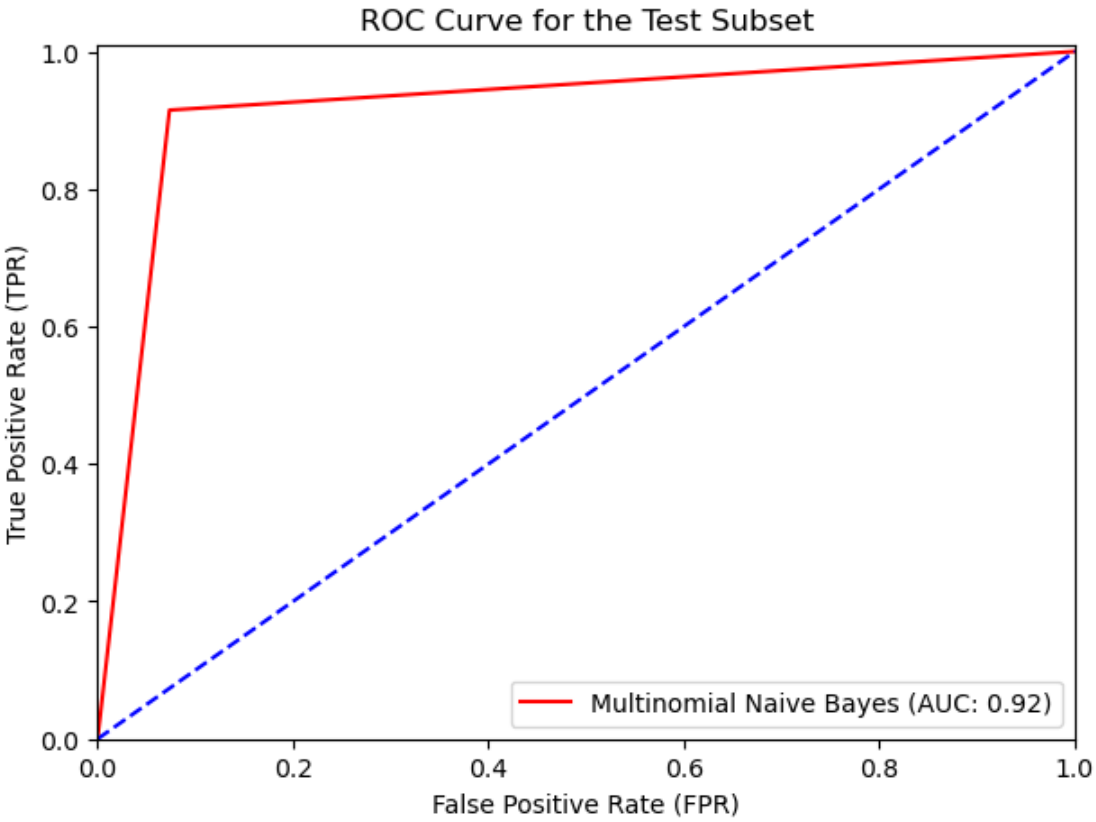
[30]:
```
array([[0.76462079, 0.23537921],
       [0.18305454, 0.81694546],
       [0.65381868, 0.34618132],
       [0.76462079, 0.23537921],
       [0.18305454, 0.81694546],
       [0.65381868, 0.34618132],
       [0.76462079, 0.23537921],
       [0.27818683, 0.72181317],
       [0.65381868, 0.34618132],
       [0.18305454, 0.81694546]])
```

[31]:
```
# Computing multinomial naive Bayes confusion matrix

mnb_c_matrix = confusion_matrix(y_test, test_y_pred)

plt.figure(figsize=(4,4))
sns.heatmap(mnb_c_matrix, annot=True, fmt=".0f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
```

```
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()
```



[32]:
```
# Plotting the ROC curve for the test subset
fpr, tpr, thresholds = roc_curve(test_y_pred, y_test)  # Computing ROC for the␣
 ↪test subset
auc(fpr, tpr)                                          # Computing AUC for the␣
 ↪test subset

plt.figure(figsize=(7, 5))
plt.plot(fpr, tpr, color='red', label='Multinomial Naive Bayes (AUC: %.2f)'
% auc(fpr, tpr))
plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.01])
plt.title('ROC Curve for the Test Subset')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend()
plt.show()
```



As observed above, the multinomial naive Bayes classifier performs as strong as the previous classifiers. This indicates that perhaps the categorical features in this dataset are as significant as the continuous features for determining the existence of instability issue.

Question: How can we take the advantage of the both categorical and continuous features simultaneously to perform a classification based on the naive Bayes approach? Is there a *hybrid method* which would allow for both categorical and continuous features?

3.6 Further Reading

1. Mitchell, T (1997). Machine Learning, ch. 6, McGraw Hill.
2. Ng, A.Y. & Jordan, M. I. (2002). On Discriminative vs. Generative Classifiers: A comparison of Logistic Regression and Naive Bayes, Neural Information Processing Systems, Ng, A.Y., and Jordan, M. (2002).