12

Parametric Classification

Introduction to Parametric Classification

Models of data with a categorical response are called *classifiers*. A classifier is built from *training data*, for which classifications are known. The classifier assigns new *test data* to one of the categorical levels of the response.

Parametric methods, like "Discriminant Analysis" on page 12-3, fit a parametric model to the training data and interpolate to classify test data.

Nonparametric methods, like "Classification Trees and Regression Trees" on page 13-31, use other means to determine classifications. In this sense, classification methods are analogous to the methods discussed in "Nonlinear Regression" on page 9-130.

Discriminant Analysis

In this section...

What Is Discriminant Analysis?

Discriminant analysis is a classification method. It assumes that different classes generate data based on different Gaussian distributions.

- To train (create) a classifier, the fitting function estimates the parameters of a Gaussian distribution for each class (see "How the ClassificationDiscriminant.fit Method Creates a Classifier" on page 12-5).
- To predict the classes of new data, the trained classifier finds the class with the smallest misclassification cost (see "How the predict Method Classifies" on page 12-6).

To learn how to prepare your data for discriminant analysis and create a classifier, see "Steps in Supervised Learning (Machine Learning)" on page 13-2.

Linear discriminant analysis is also known as the Fisher discriminant, named for its inventor, Sir R. A. Fisher [2].

Example: Create Discriminant Analysis Classifiers

To create the basic types of discriminant analysis classifiers for the Fisher iris data:

1. Load the data:

```
load fisheriris;
```

2. Create a default (linear) discriminant analysis classifier:

```
linclass = ClassificationDiscriminant.fit(meas,species);
```

   To visualize the classification boundaries of a 2-D linear classification of the data, see Linear Discriminant Classification — Fisher Training Data on page 12-13.

3. Classify an iris with average measurements:

```
meanmeas = mean(meas);
meanclass = predict(linclass,meanmeas)

meanclass =
    'versicolor'
```

4. Create a quadratic classifier:

```
quadclass = ClassificationDiscriminant.fit(meas,species,...
    'discrimType','quadratic');
```

   To visualize the classification boundaries of a 2-D quadratic classification of the data, see Quadratic Discriminant Classification — Fisher Training Data on page 12-15.

5. Classify an iris with average measurements using the quadratic classifier:

```
meanclass2 = predict(quadclass,meanmeas)

meanclass2 =
    'versicolor'
```

How the ClassificationDiscriminant.fit Method Creates a Classifier

The model for discriminant analysis is:

- Each class (`Y`) generates data (`X`) using a multivariate normal distribution. In other words, the model assumes `X` has a Gaussian mixture distribution (`gmdistribution`).
  - For linear discriminant analysis, the model has the same covariance matrix for each class; only the means vary.
  - For quadratic discriminant analysis, both means and covariances of each class vary.

Under this modeling assumption, `ClassificationDiscriminant.fit` infers the mean and covariance parameters of each class.

- For linear discriminant analysis, it computes the sample mean of each class. Then it computes the sample covariance by first subtracting the sample mean of each class from the observations of that class, and taking the empirical covariance matrix of the result.
- For quadratic discriminant analysis, it computes the sample mean of each class. Then it computes the sample covariances by first subtracting the sample mean of each class from the observations of that class, and taking the empirical covariance matrix of each class.

The `fit` method does not use prior probabilities or costs for fitting.

Weighted Observations

The `fit` method constructs weighted classifiers using the following scheme. Suppose M is an N-by-K class membership matrix:

$M_{nk} = 1$ if observation n is from class k
$M_{nk} = 0$ otherwise.

The estimate of the class mean for unweighted data is

$$\hat{\mu}_k = \frac{\sum_{n=1}^{N} M_{nk} x_n}{\sum_{n=1}^{N} M_{nk}}.$$

For weighted data with positive weights w_n, the natural generalization is

$$\hat{\mu}_k = \frac{\sum_{n=1}^{N} M_{nk} w_n x_n}{\sum_{n=1}^{N} M_{nk} w_n}.$$

The unbiased estimate of the pooled-in covariance matrix for unweighted data is

$$\hat{\Sigma} = \frac{\sum_{n=1}^{N}\sum_{k=1}^{K} M_{nk} \left(x_n - \hat{\mu}_k\right)\left(x_n - \hat{\mu}_k\right)^T}{N - K}.$$

For quadratic discriminant analysis, the `fit` method uses $K = 1$.

For weighted data, assuming the weights sum to 1, the unbiased estimate of the pooled-in covariance matrix is

$$\hat{\Sigma} = \frac{\sum_{n=1}^{N}\sum_{k=1}^{K} M_{nk} w_n \left(x_n - \hat{\mu}_k\right)\left(x_n - \hat{\mu}_k\right)^T}{1 - \sum_{k=1}^{K} \frac{W_k^{(2)}}{W_k}},$$

where

- $W_k = \sum_{n=1}^{N} M_{nk} w_n$ is the sum of the weights for class k.
- $W_k^{(2)} = \sum_{n=1}^{N} M_{nk} w_n^2$ is the sum of squared weights per class.

How the predict Method Classifies

There are three elements in the `predict` classification algorithm:

- "Posterior Probability" on page 12-7
- "Prior Probability" on page 12-8
- "Cost" on page 12-8

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \underset{y=1,\ldots,K}{\arg\min} \sum_{k=1}^{K} \hat{P}(k \mid x) C(y \mid k),$$

where

- $\hat{y}$ is the predicted classification.
- K is the number of classes.
- $\hat{P}(k \mid x)$ is the posterior probability of class k for observation x.
- $C(y \mid k)$ is the cost of classifying an observation as y when its true class is k.

The space of `X` values divides into regions where a classification `Y` is a particular value. The regions are separated by straight lines for linear discriminant analysis, and by conic sections (ellipses, hyperbolas, or parabolas) for quadratic discriminant analysis. For a visualization of these regions, see "Example: Creating and Visualizing a Discriminant Analysis Classifier" on page 12-9.

Posterior Probability

The posterior probability that a point x belongs to class k is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean μ_k and covariance Σ_k at a point x is

$$P(x \mid k) = \frac{1}{\left(2\pi |\Sigma_k|\right)^{1/2}} \exp\left(-\frac{1}{2} (x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k) \right),$$

where $|\Sigma_k|$ is the determinant of Σ_k, and Σ_k^{-1} is the inverse matrix.

Let $P(k)$ represent the prior probability of class k. Then the posterior probability that an observation x is of class k is

$$\hat{P}(k \mid x) = \frac{P(x \mid k)P(k)}{P(x)},$$

where $P(x)$ is a normalization constant, namely, the sum over k of $P(x \mid k)P(k)$.

Prior Probability

The prior probability is one of three choices:

- `'uniform'` — The prior probability of class `k` is 1 over the total number of classes.
- `'empirical'` — The prior probability of class `k` is the number of training samples of class `k` divided by the total number of training samples.
- A numeric vector — The prior probability of class `k` is the `j`th element of the `prior` vector. See `ClassificationDiscriminant.fit`.

After creating a classifier `obj`, you can set the prior using dot addressing:

```
obj.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

Cost

There are two costs associated with discriminant analysis classification: the true misclassification cost per class, and the expected misclassification cost per observation.

True Misclassification Cost per Class. `Cost(i,j)` is the cost of classifying an observation into class `j` if its true class is `i`. By default, `Cost(i,j)=1` if `i~=j`, and `Cost(i,j)=0` if `i=j`. In other words, the cost is `0` for correct classification, and `1` for incorrect classification.

You can set any cost matrix you like when creating a classifier. Pass the cost matrix in the `Cost` name-value pair in `ClassificationDiscriminant.fit`.

After you create a classifier `obj`, you can set a custom cost using dot addressing:

```
obj.Cost = B;
```

`B` is a square matrix of size `K`-by-`K` when there are `K` classes. You do not need to retrain the classifier when you set a new cost.

Expected Misclassification Cost per Observation. Suppose you have `Nobs` observations that you want to classify with a trained discriminant analysis classifier `obj`. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label,score,cost] = predict(obj,Xnew)
```

returns, among other outputs, a cost matrix of size `Nobs`-by-`K`. Each row of the cost matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^{K} \hat{P}(i \mid Xnew(n)) C(k \mid i),$$

where

- K is the number of classes.
- $\hat{P}(i \mid Xnew(n))$ is the posterior probability of class i for observation $Xnew(n)$.
- $C(k \mid i)$ is the cost of classifying an observation as k when its true class is i.

Example: Creating and Visualizing a Discriminant Analysis Classifier

This example shows both linear and quadratic classification of the Fisher iris data. The example uses only two of the four predictors to enable simple plotting.

1 Load the data:

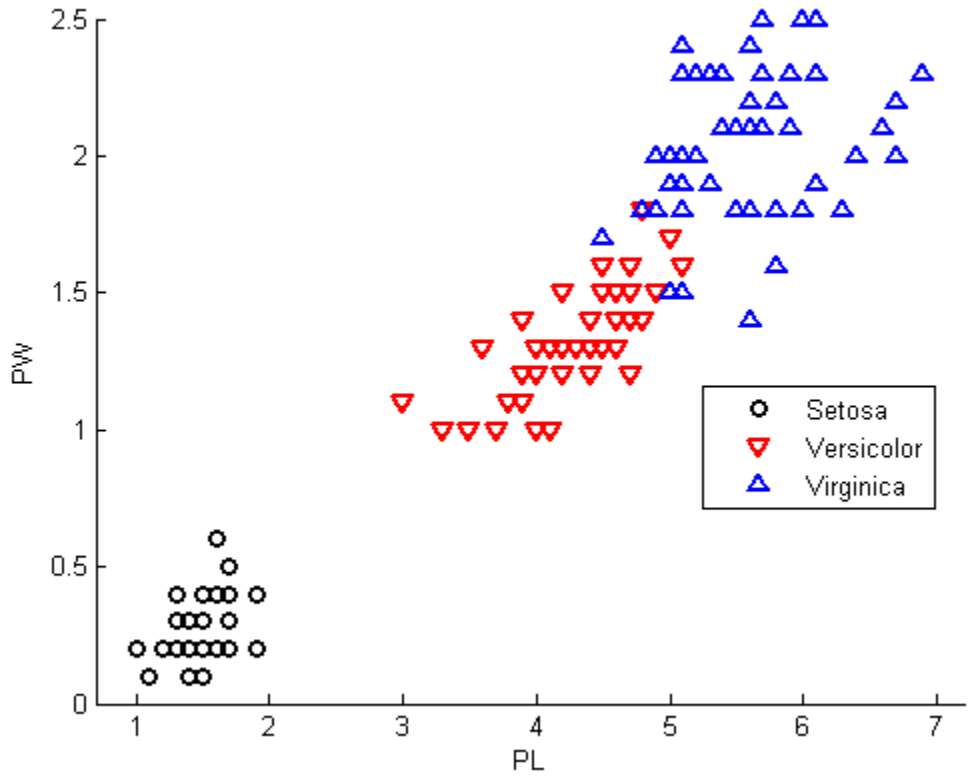
```
load fisheriris
```

2 Use the petal length (PL) and petal width (PW) measurements:

```
PL = meas(:,3);
PW = meas(:,4);
```

3 Plot the data, showing the classification:

```
h1 = gscatter(PL,PW,species,'krb','ov^',[],'off');
set(h1,'LineWidth',2)
legend('Setosa','Versicolor','Virginica',...
       'Location','best')
```



4 Create a linear classifier:

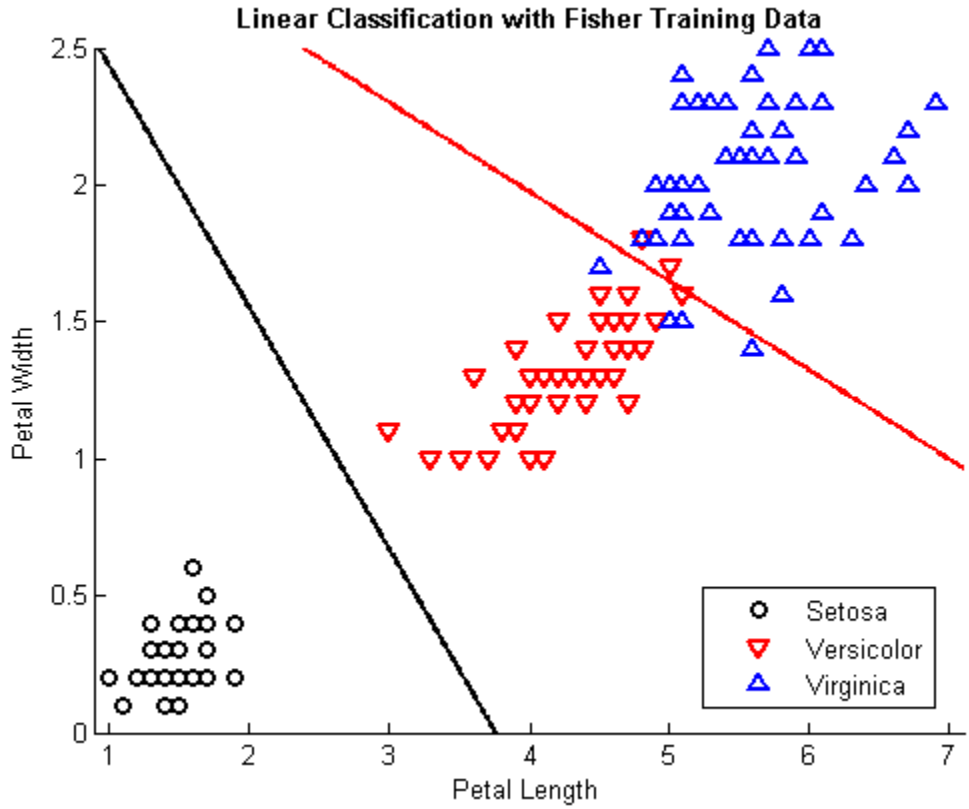
```
X = [PL,PW];
cls = ClassificationDiscriminant.fit(X,species);
```

5 Plot the classification boundaries:

```
hold on
K = cls.Coeffs(2,3).Const;
L = cls.Coeffs(2,3).Linear;
% Plot the curve K + [x,y]*L  = 0:
f = @(x1,x2) K + L(1)*x1 + L(2)*x2;
h2 = ezplot(f,[.9 7.1 0 2.5]);
```

```
set(h2,'Color','r','LineWidth',2)

K = cls.Coeffs(1,2).Const;
L = cls.Coeffs(1,2).Linear;
% Plot the curve K + [x1,x2]*L  = 0:
f = @(x1,x2) K + L(1)*x1 + L(2)*x2;
h3 = ezplot(f,[.9 7.1 0 2.5]);
set(h3,'Color','k','LineWidth',2)
axis([.9 7.1 0 2.5])
xlabel('Petal Length')
ylabel('Petal Width')
title('{\bf Linear Classification with Fisher Training Data}')
```



Linear Discriminant Classification — Fisher Training Data

6 Create a quadratic discriminant classifier:

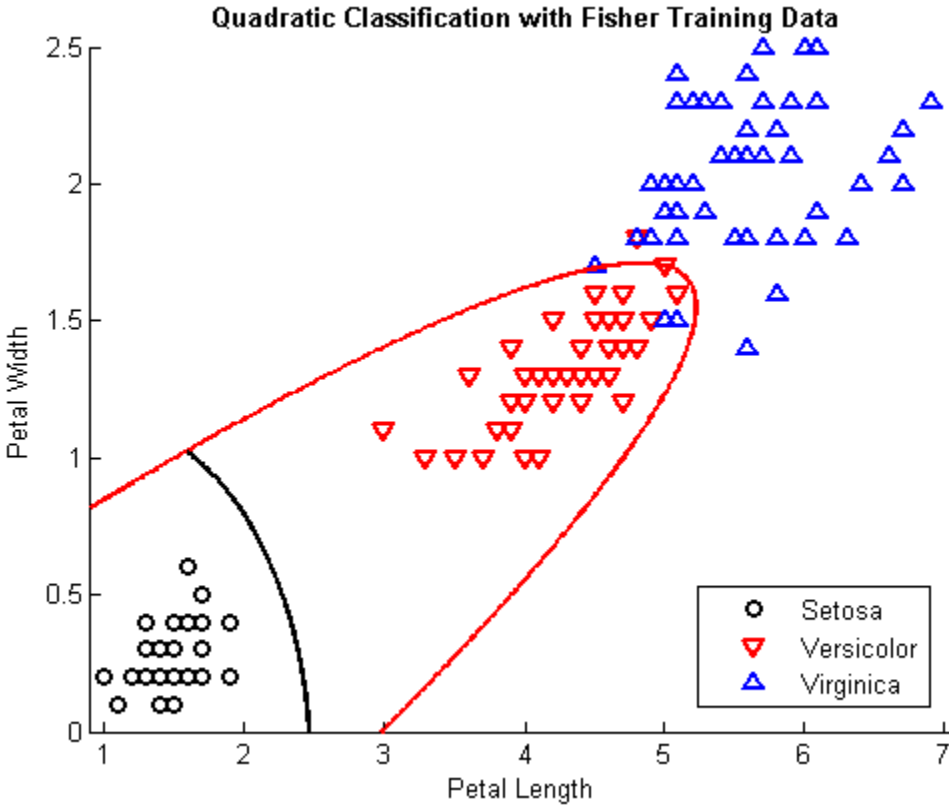
```
cqs = ClassificationDiscriminant.fit(X,species,...
    'DiscrimType','quadratic');
```

7 Plot the classification boundaries:

```
delete(h2); delete(h3) % remove the linear plots
K = cqs.Coeffs(2,3).Const;
L = cqs.Coeffs(2,3).Linear;
Q = cqs.Coeffs(2,3).Quadratic;
% Plot the curve K + [x1,x2]*L + [x1,x2]*Q*[x1,x2]'=0:
```

```
f = @(x1,x2) K + L(1)*x1 + L(2)*x2 + Q(1,1)*x1.^2 + ...
    (Q(1,2)+Q(2,1))*x1.*x2 + Q(2,2)*x2.^2;
h2 = ezplot(f,[.9 7.1 0 2.5]);
set(h2,'Color','r','LineWidth',2)

K = cqs.Coeffs(1,2).Const;
L = cqs.Coeffs(1,2).Linear;
Q = cqs.Coeffs(1,2).Quadratic;
% Plot the curve K + [x1,x2]*L + [x1,x2]*Q*[x1,x2]'=0:
f = @(x1,x2) K + L(1)*x1 + L(2)*x2 + Q(1,1)*x1.^2 + ...
    (Q(1,2)+Q(2,1))*x1.*x2 + Q(2,2)*x2.^2;
h3 = ezplot(f,[.9 7.1 0 1.02]); % plot the relevant
                                % portion of the curve
set(h3,'Color','k','LineWidth',2)
axis([.9 7.1 0 2.5])
xlabel('Petal Length')
ylabel('Petal Width')
title('{\bf Quadratic Classification with Fisher Training Data}')
hold off
```



Quadratic Discriminant Classification — Fisher Training Data

Improving a Discriminant Analysis Classifier

Deal with Singular Data

Discriminant analysis needs data sufficient to fit Gaussian models with invertible covariance matrices. If your data is not sufficient to fit such a model uniquely, `ClassificationDiscriminant.fit` fails. This section shows methods for handling failures.

Tip To obtain a discriminant analysis classifier without failure, set the `DiscrimType` name-value pair to `'pseudoLinear'` or `'pseudoQuadratic'` in `ClassificationDiscriminant.fit`.

"Pseudo" discriminants never fail, because they use the pseudoinverse of the covariance matrix Σ_k (see `pinv`).

Example: Singular Covariance Matrix. When the covariance matrix of the fitted classifier is singular, `ClassificationDiscriminant.fit` can fail:

```
load popcorn
X = popcorn(:,[1 2]);
X(:,3) = 0; % a zero-variance column
Y = popcorn(:,3);
ppcrn = ClassificationDiscriminant.fit(X,Y);

Error using ClassificationDiscriminant (line 635)
Predictor x3 has zero variance. Either exclude this predictor or set 'discrimType' to
'pseudoLinear' or 'diagLinear'.

Error in classreg.learning.FitTemplate/fit (line 243)
            obj = this.MakeFitObject(X,Y,W,this.ModelParams,fitArgs{:});

Error in ClassificationDiscriminant.fit (line 296)
            this = fit(temp,X,Y);
```

To proceed with linear discriminant analysis, use a `pseudoLinear` or `diagLinear` discriminant type:

```
ppcrn = ClassificationDiscriminant.fit(X,Y,...
    'discrimType','pseudoLinear');
meanpredict = predict(ppcrn,mean(X))
```

```
meanpredict =
    3.5000
```

Choose a Discriminant Type

There are six types of discriminant analysis classifiers: linear and quadratic, with *diagonal* and *pseudo* variants of each type.

Tip To see if your covariance matrix is singular, set `discrimType` to `'linear'` or `'quadratic'`. If the matrix is singular, the `ClassificationDiscriminant.fit` method fails for `'quadratic'`, and the `Gamma` property is nonzero for `'linear'`.

To obtain a quadratic classifier even when your covariance matrix is singular, set `discrimType` to `'pseudoQuadratic'` or `'diagQuadratic'`.

```
obj = ClassificationDiscriminant.fit(X,Y,...
    'discrimType','pseudoQuadratic') % or 'diagQuadratic'
```

Choose a classifier type by setting the `discrimType` name-value pair to one of:

- `'linear'` (default) — Estimate one covariance matrix for all classes.
- `'quadratic'` — Estimate one covariance matrix for each class.
- `'diagLinear'` — Use the diagonal of the `'linear'` covariance matrix, and use its pseudoinverse if necessary.
- `'diagQuadratic'` — Use the diagonals of the `'quadratic'` covariance matrices, and use their pseudoinverses if necessary.
- `'pseudoLinear'` — Use the pseudoinverse of the `'linear'` covariance matrix if necessary.
- `'pseudoQuadratic'` — Use the pseudoinverses of the `'quadratic'` covariance matrices if necessary.

`ClassificationDiscriminant.fit` can fail for the `'linear'` and `'quadratic'` classifiers. When it fails, it returns an explanation, as shown in "Deal with Singular Data" on page 12-16.

`ClassificationDiscriminant.fit` always succeeds with the diagonal and pseudo variants. For information about pseudoinverses, see `pinv`.

You can set the discriminant type using dot addressing after constructing a classifier:

```
obj.DiscrimType = 'discrimType'
```

You can change between linear types or between quadratic types, but cannot change between a linear and a quadratic type.

Examine the Resubstitution Error and Confusion Matrix

The *resubstitution error* is the difference between the response training data and the predictions the classifier makes of the response based on the input training data. If the resubstitution error is high, you cannot expect the predictions of the classifier to be good. However, having low resubstitution error does not guarantee good predictions for new data. Resubstitution error is often an overly optimistic estimate of the predictive error on new data.

The *confusion matrix* shows how many errors, and which types, arise in resubstitution. When there are `K` classes, the confusion matrix `R` is a `K`-by-`K` matrix with

`R(i,j)` = the number of observations of class `i` that the classifier predicts to be of class `j`.

Example: Resubstitution Error of a Discriminant Analysis Classifier.

Examine the resubstitution error of the default discriminant analysis classifier for the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200
```

The resubstitution error is very low, meaning `obj` classifies nearly all the Fisher iris data correctly. The total number of misclassifications is:

```
resuberror * obj.NObservations

ans =
    3.0000
```

To see the details of the three misclassifications, examine the confusion matrix:

```
R = confusionmat(obj.Y,resubPredict(obj))

R =
    50     0     0
     0    48     2
     0     1    49

obj.ClassNames

ans =
    'setosa'
    'versicolor'
    'virginica'
```

- `R(1,:) = [50 0 0]` means `obj` classifies all 50 setosa irises correctly.
- `R(2,:) = [0 48 2]` means `obj` classifies 48 versicolor irises correctly, and misclassifies two versicolor irises as virginica.
- `R(3,:) = [0 1 49]` means `obj` classifies 49 virginica irises correctly, and misclassifies one virginica iris as versicolor.

Cross Validation

Typically, discriminant analysis classifiers are robust and do not exhibit overtraining when the number of predictors is much less than the number of observations. Nevertheless, it is good practice to cross validate your classifier to ensure its stability.

Example: Cross Validating a Discriminant Analysis Classifier. Try five-fold cross validation of a quadratic discriminant analysis classifier:

1 Load the Fisher iris data:

```
load fisheriris
```

2 Create a quadratic discriminant analysis classifier for the data:

```
quadisc = ClassificationDiscriminant.fit(meas,species,...
    'DiscrimType','quadratic');
```

3 Find the resubstitution error of the classifier:

```
qerror = resubLoss(quadisc)

qerror =
    0.0200
```

The classifier does an excellent job. Nevertheless, resubstitution error can be an optimistic estimate of the error when classifying new data. So proceed to cross validation.

4 Create a cross-validation model:

```
cvmodel = crossval(quadisc,'kfold',5);
```

5 Find the cross-validation loss for the model, meaning the error of the out-of-fold observations:

```
cverror = kfoldLoss(cvmodel)

cverror =
    0.0333
```

The cross-validated loss is nearly as low as the original resubstitution loss. Therefore, you can have confidence that the classifier is reasonably accurate.

Change Costs and Priors

Sometimes you want to avoid certain misclassification errors more than others. For example, it might be better to have oversensitive cancer detection instead of undersensitive cancer detection. Oversensitive detection gives more false positives (unnecessary testing or treatment). Undersensitive detection gives more false negatives (preventable illnesses or deaths). The

consequences of underdetection can be high. Therefore, you might want to set costs to reflect the consequences.

Similarly, the training data `Y` can have a distribution of classes that does not represent their true frequency. If you have a better estimate of the true frequency, you can include this knowledge in the classification `prior` property.

Example: Setting Custom Misclassification Costs. Consider the Fisher iris data. Suppose that the cost of classifying a versicolor iris as virginica is 10 times as large as making any other classification error. Create a classifier from the data, then incorporate this cost and then view the resulting classifier.

1 Load the Fisher iris data and create a default (linear) classifier as in "Example: Resubstitution Error of a Discriminant Analysis Classifier" on page 12-18:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200

R = confusionmat(obj.Y,resubPredict(obj))

R =
    50     0     0
     0    48     2
     0     1    49

obj.ClassNames

ans =
    'setosa'
    'versicolor'
    'virginica'
```

`R(2,:) = [0 48 2]` means `obj` classifies 48 versicolor irises correctly, and misclassifies two versicolor irises as virginica.

2 Change the cost matrix to make fewer mistakes in classifying versicolor irises as virginica:

```
obj.Cost(2,3) = 10;
R2 = confusionmat(obj.Y,resubPredict(obj))

R2 =
    50     0     0
     0    50     0
     0     7    43
```

`obj` now classifies all versicolor irises correctly, at the expense of increasing the number of misclassifications of virginica irises from 1 to 7.

Example: Setting Alternative Priors. Consider the Fisher iris data. There are 50 irises of each kind in the data. Suppose that, in a particular region, you have historical data that shows virginica are five times as prevalent as the other kinds. Create a classifier that incorporates this information.

1 Load the Fisher iris data and make a default (linear) classifier as in "Example: Resubstitution Error of a Discriminant Analysis Classifier" on page 12-18:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200

R = confusionmat(obj.Y,resubPredict(obj))

R =
    50     0     0
     0    48     2
     0     1    49

obj.ClassNames

ans =
    'setosa'
    'versicolor'
```

```
    'virginica'
```

`R(3,:)  = [0 1 49]` means `obj` classifies 49 virginica irises correctly, and misclassifies one virginica iris as versicolor.

2 Change the prior to match your historical data, and examine the confusion matrix of the new classifier:

```
obj.Prior = [1 1 5];
R2 = confusionmat(obj.Y,resubPredict(obj))

R2 =
    50     0     0
     0    46     4
     0     0    50
```

The new classifier classifies all virginica irises correctly, at the expense of increasing the number of misclassifications of versicolor irises from 2 to 4.

Regularize a Discriminant Analysis Classifier

To make a more robust and simpler model, try to remove predictors from your model without hurting its predictive power. This is especially important when you have many predictors in your data. Linear discriminant analysis uses the two regularization parameters, `Gamma` and `Delta`, to identify and remove redundant predictors. The `cvshrink` method helps you identify appropriate settings for these parameters.

1. Load data and create a classifier.

Create a linear discriminant analysis classifier for the `ovariancancer` data. Set the `SaveMemory` and `FillCoeffs` options to keep the resulting model reasonably small.

```
load ovariancancer
obj = ClassificationDiscriminant.fit(obs,grp,...
    'SaveMemory','on','FillCoeffs','off');
```

2. Cross validate the classifier.

Use 30 levels of `Gamma` and 30 levels of `Delta` to search for good parameters. This search is time consuming. Set `Verbose` to `1` to view the progress.

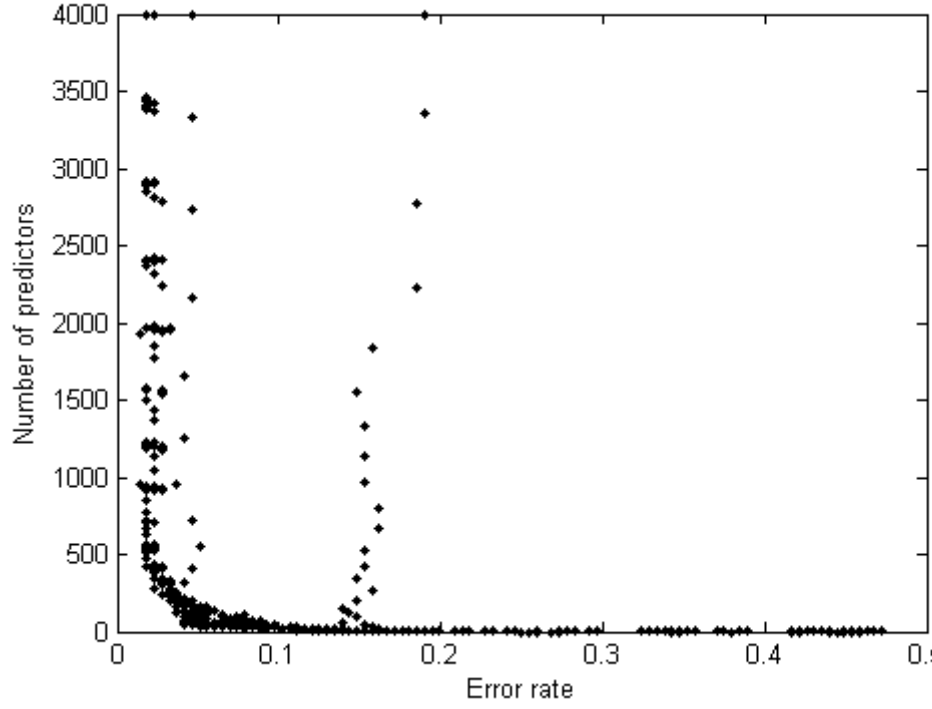
```
rng(8000,'twister') % for reproducibility
[err,gamma,delta,numpred] = cvshrink(obj,...
    'NumGamma',29,'NumDelta',29,'Verbose',1);
```

```
Done building cross-validated model.
Processing Gamma step 1 out of 30.
Processing Gamma step 2 out of 30.
Processing Gamma step 3 out of 30.
%%% (many lines removed) %%%
Processing Gamma step 28 out of 30.
Processing Gamma step 29 out of 30.
Processing Gamma step 30 out of 30.
```

3. Examine the quality of the regularized classifiers.

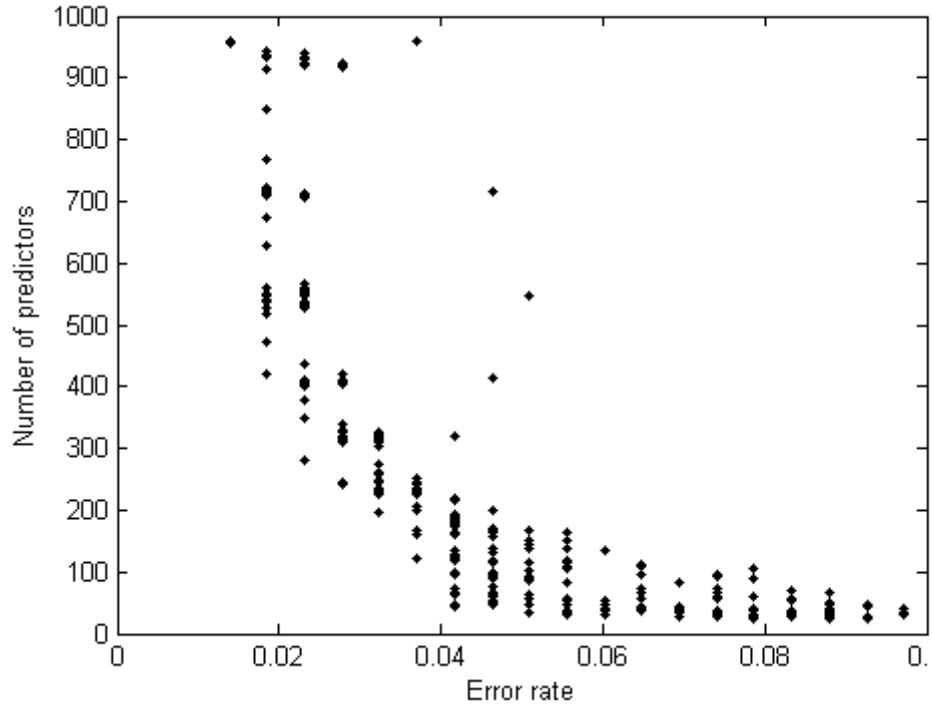
Plot the number of predictors against the error.

```
figure;
plot(err,numpred,'k.')
xlabel('Error rate');
ylabel('Number of predictors');
```



Examine the lower-left part of the plot more closely.

```
axis([0 .1 0 1000])
```



There is a clear tradeoff between lower number of predictors and lower error.

4. Choose an optimal tradeoff between model size and accuracy.

Find the values of `Gamma` and `Delta` that give minimal error.

```
minerr = min(min(err));
[p,q] = find(err == minerr)
```

```
p =
    24
    25

q =
     8
     8
```

Two points have the same minimal error: `[24,8]` and `[25,8]`, which correspond to

```
[gamma(p(1)),delta(p(1),q(1))]

ans =
    0.8436    0.1463

[gamma(p(2)),delta(p(2),q(2))]

ans =
    0.8697    0.1425
```

These points correspond to about a quarter of the total predictors having nonzero coefficients in the model.

```
numpred(p(1),q(1))

ans =
   957

numpred(p(2),q(2))

ans =
   960
```

To further lower the number of predictors, you must accept larger error rates. For example, to choose the `Gamma` and `Delta` that give the lowest error rate with 250 or fewer predictors:

```
low250 = min(min(err(numpred <= 250)))

low250 =

    0.0278

lownum = min(min(numpred(err == low250)))

lownum =

   243
```

You need 243 predictors to achieve an error rate of 0.0278, and this is the lowest error rate among those that have 250 predictors or fewer. The `Gamma` and `Delta` that achieve this error/number of predictors:

```
[r,s] = find((err == low250) & (numpred == lownum));
gamma(r)

ans =

    0.7133

delta(r,s)

ans =

    0.2960
```

5. Set the regularization parameters.

To set the classifier with these values of `Gamma` and `Delta`, use dot addressing.

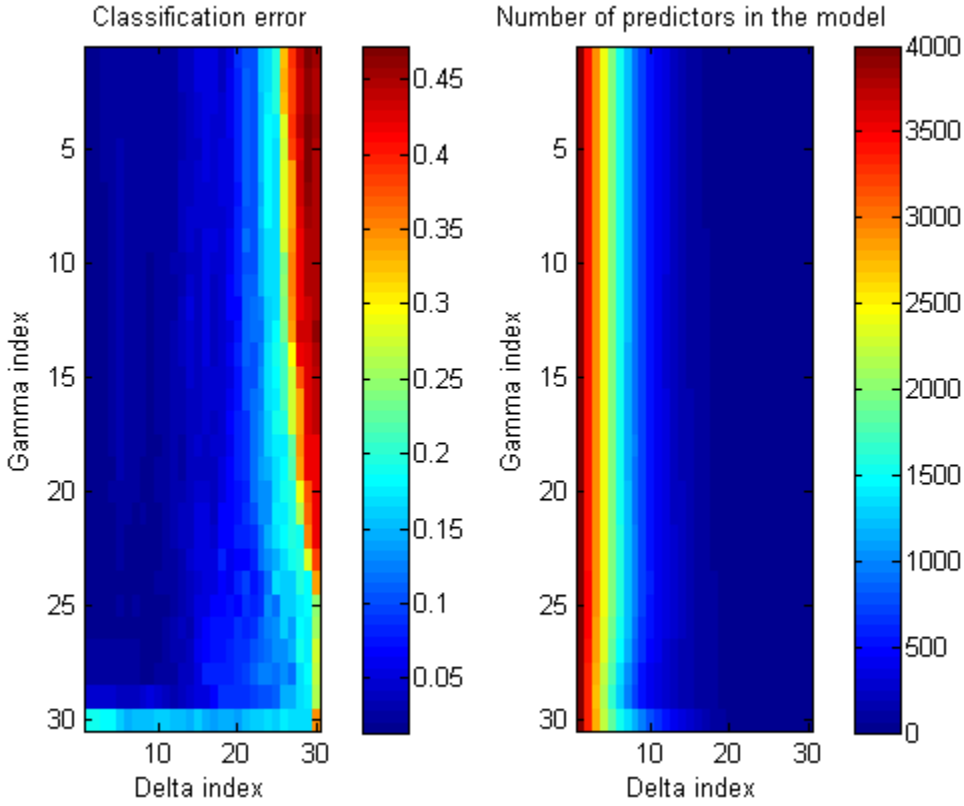
```
obj.Gamma = gamma(r);
obj.Delta = delta(r,s);
```

6. Heat map plot.

To compare the `cvshrink` calculation to that in Guo, Hastie, and Tibshirani [3], plot heat maps of error and number of predictors against `Gamma` and the index of the `Delta` parameter. (The `Delta` parameter range depends on the value of the `Gamma` parameter. So to get a rectangular plot, use the `Delta` index, not the parameter itself.)

```
% First create the Delta index matrix
indx = repmat(1:size(delta,2),size(delta,1),1);
figure
subplot(1,2,1)
imagesc(err);
colorbar;
title('Classification error');
xlabel('Delta index');
ylabel('Gamma index');
```

```
subplot(1,2,2)
imagesc(numpred);
colorbar;
title('Number of predictors in the model');
xlabel('Delta index');
ylabel('Gamma index');
```



You see the best classification error when `Delta` is small, but fewest predictors when `Delta` is large.

Examining the Gaussian Mixture Assumption

Discriminant analysis assumes that the data comes from a Gaussian mixture model (see "How the ClassificationDiscriminant.fit Method Creates a Classifier" on page 12-5). If the data appears to come from a Gaussian mixture model, you can expect discriminant analysis to be a good classifier. Furthermore, the default linear discriminant analysis assumes that all class covariance matrices are equal. This section shows methods to check these assumptions:

Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis

The Bartlett test (see Box [1]) checks equality of the covariance matrices of the various classes. If the covariance matrices are equal, the test indicates that linear discriminant analysis is appropriate. If not, consider using quadratic discriminant analysis, setting the `DiscrimType` name-value pair to `'quadratic'` in `ClassificationDiscriminant.fit`.

The Bartlett test assumes normal (Gaussian) samples, where neither the means nor covariance matrices are known. To determine whether the covariances are equal, compute the following quantities:

- Sample covariance matrices per class σ_i, $1 \le i \le k$, where k is the number of classes.
- Pooled-in covariance matrix σ.
- Test statistic V:

$$V = (n-k)\log(|\Sigma|) - \sum_{i=1}^{k}(n_i - 1)\log(|\Sigma_i|)$$

  where n is the total number of observations, and n_i is the number of observations in class i, and $|\Sigma|$ means the determinant of the matrix Σ.

- Asymptotically, as the number of observations in each class n_i become large, V is distributed approximately χ^2 with $kd(d + 1)/2$ degrees of freedom, where d is the number of predictors (number of dimensions in the data).

The Bartlett test is to check whether V exceeds a given percentile of the χ^2 distribution with $kd(d + 1)/2$ degrees of freedom. If it does, then reject the hypothesis that the covariances are equal.

Example: Bartlett Test for Equal Covariance Matrices. Check whether the Fisher iris data is well modeled by a single Gaussian covariance, or whether it would be better to model it as a Gaussian mixture.

```
load fisheriris;
prednames = {'SepalLength','SepalWidth',...
    'PetalLength','PetalWidth'};
L = ClassificationDiscriminant.fit(meas,species,...
    'PredictorNames',prednames);
Q = ClassificationDiscriminant.fit(meas,species,...
    'PredictorNames',prednames,'DiscrimType','quadratic');
D = 4; % Number of dimensions of X
Nclass = [50 50 50];
N = L.NObservations;
K = numel(L.ClassNames);
SigmaQ = Q.Sigma;
SigmaL = L.Sigma;
logV = (N-K)*log(det(SigmaL));
for k=1:K
    logV = logV - (Nclass(k)-1)*log(det(SigmaQ(:,:,k)));
end
nu = (K-1)*D*(D+1)/2;
pval = 1-chi2cdf(logV,nu)

pval =
     0
```

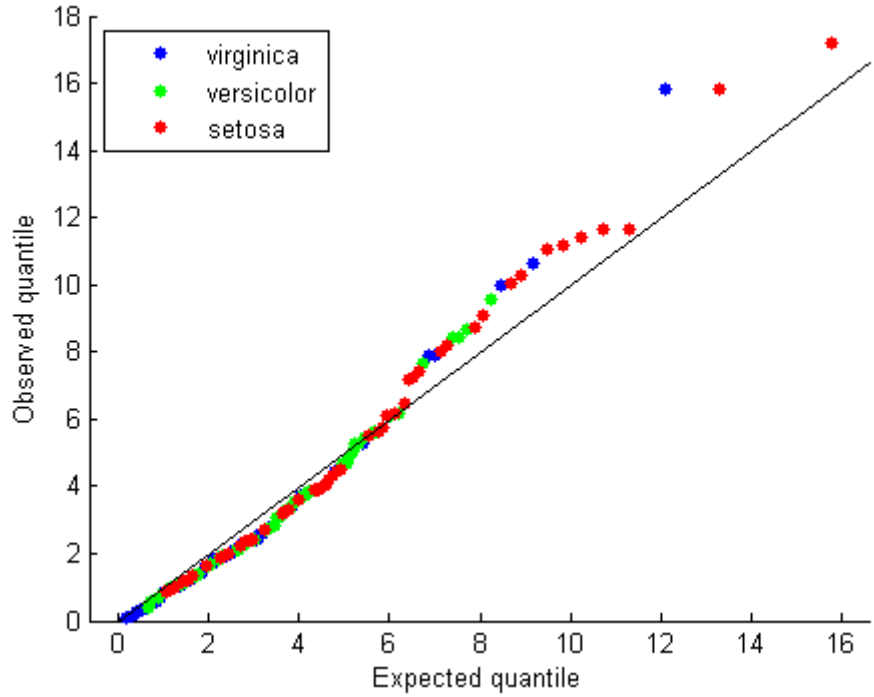
The Bartlett test emphatically rejects the hypothesis of equal covariance matrices. If `pval` had been greater than `0.05`, the test would not have rejected the hypothesis. The result indicates to use quadratic discriminant analysis, as opposed to linear discriminant analysis.

Q-Q Plot

A Q-Q plot graphically shows whether an empirical distribution is close to a theoretical distribution. If the two are equal, the Q-Q plot lies on a 45° line. If not, the Q-Q plot strays from the 45° line.

Check Q-Q Plots for Linear and Quadratic Discriminants. For linear discriminant analysis, use a single covariance matrix for all classes.

```
load fisheriris;
prednames = {'SepalLength','SepalWidth',...
    'PetalLength','PetalWidth'};
L = ClassificationDiscriminant.fit(meas,species,...
    'PredictorNames',prednames);
N = L.NObservations;
K = numel(L.ClassNames);
mahL = mahal(L,L.X,'ClassLabels',L.Y);
D = 4;
expQ = chi2inv(((1:N)-0.5)/N,D); % expected quantiles
[mahL,sorted] = sort(mahL); % sorted obbserved quantiles
figure;
gscatter(expQ,mahL,L.Y(sorted),'bgr',[],[],'off');
legend('virginica','versicolor','setosa','Location','NW');
xlabel('Expected quantile');
ylabel('Observed quantile');
line([0 20],[0 20],'color','k');
```

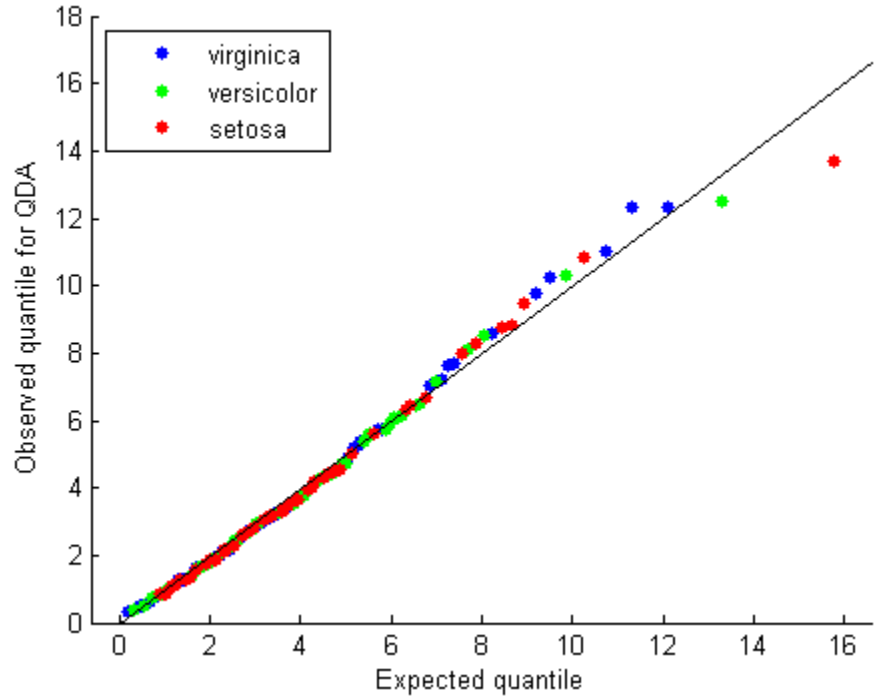


Overall, the agreement between the expected and observed quantiles is good. Look at the right half of the plot. The deviation of the plot from the 45° line upward indicates that the data has tails heavier than a normal distribution. There are three possible outliers on the right: two observations from class `'setosa'` and one observation from class `'virginica'`.

As shown in “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 12-30, the data does not match a single covariance matrix. Redo the calculations for a quadratic discriminant.

```
load fisheriris;
prednames = {'SepalLength','SepalWidth',...
    'PetalLength','PetalWidth'};
Q = ClassificationDiscriminant.fit(meas,species,...
    'PredictorNames',prednames,'DiscrimType','quadratic');
Nclass = [50 50 50];
```

```
N = L.NObservations;
K = numel(L.ClassNames);
mahQ = mahal(Q,Q.X,'ClassLabels',Q.Y);
expQ = chi2inv(((1:N)-0.5)/N,D);
[mahQ,sorted] = sort(mahQ);
figure;
gscatter(expQ,mahQ,Q.Y(sorted),'bgr',[],[],'off');
legend('virginica','versicolor','setosa','Location','NW');
xlabel('Expected quantile');
ylabel('Observed quantile for QDA');
line([0 20],[0 20],'color','k');
```



The Q-Q plot shows a better agreement between the observed and expected quantiles. There is only one outlier candidate, from class `'setosa'`.

Mardia Kurtosis Test of Multivariate Normality

The Mardia kurtosis test (see Mardia [4]) is an alternative to examining a Q-Q plot. It gives a numeric approach to deciding if data matches a Gaussian mixture model.

In the Mardia kurtosis test you compute M, the mean of the fourth power of the Mahalanobis distance of the data from the class means. If the data is normally distributed with constant covariance matrix (and is thus suitable for linear discriminant analysis), M is asymptotically distributed as normal with mean $d(d + 2)$ and variance $8d(d + 2)/n$, where

- d is the number of predictors (number of dimensions in the data).
- n is the total number of observations.

The Mardia test is two sided: check whether M is close enough to $d(d + 2)$ with respect to a normal distribution of variance $8d(d + 2)/n$.

Example: Mardia Kurtosis Test for Linear and Quadratic Discriminants. Check whether the Fisher iris data is approximately normally distributed for both linear and quadratic discriminant analysis. According to "Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis" on page 12-30, the data is not normal for linear discriminant analysis (the covariance matrices are different). "Check Q-Q Plots for Linear and Quadratic Discriminants" on page 12-32 indicates that the data is well modeled by a Gaussian mixture model with different covariances per class. Check these conclusions with the Mardia kurtosis test:

```
load fisheriris;
prednames = {'SepalLength','SepalWidth',...
    'PetalLength','PetalWidth'};
L = ClassificationDiscriminant.fit(meas,species,...
    'PredictorNames',prednames);
mahL = mahal(L,L.X,'ClassLabels',L.Y);
D = 4;
N = L.NObservations;
obsKurt = mean(mahL.^2);
expKurt = D*(D+2);
varKurt = 8*D*(D+2)/N;
[~,pval] = ztest(obsKurt,expKurt,sqrt(varKurt))
```

```
pval =
    0.0208
```

The Mardia test indicates to reject the hypothesis that the data is normally distributed.

Continuing the example with quadratic discriminant analysis:

```
Q = ClassificationDiscriminant.fit(meas,species,...
    'PredictorNames',prednames,'DiscrimType','quadratic');
mahQ = mahal(Q,Q.X,'ClassLabels',Q.Y);
obsKurt = mean(mahQ.^2);
[~,pval] = ztest(obsKurt,expKurt,sqrt(varKurt))

pval =
    0.7230
```

Because `pval` is high, you conclude the data are consistent with the multivariate normal distribution.

Bibliography

[1] Box, G. E. P. *A General Distribution Theory for a Class of Likelihood Criteria.* Biometrika 36(3), pp. 317–346, 1949.

[2] Fisher, R. A. *The Use of Multiple Measurements in Taxonomic Problems.* Annals of Eugenics, Vol. 7, pp. 179–188, 1936. Available at http://digital.library.adelaide.edu.au/dspace/handle/2440/15227.

[3] Guo, Y., T. Hastie, and R. Tibshirani. *Regularized Discriminant Analysis and Its Application in Microarray.* Biostatistics, Vol. 8, No. 1, pp. 86–100, 2007.

[4] Mardia, K. V. *Measures of multivariate skewness and kurtosis with applications.* Biometrika 57 (3), pp. 519–530, 1970.

Naive Bayes Classification

The Naive Bayes classifier is designed for use when features are independent of one another within each class, but it appears to work well in practice even when that independence assumption is not valid. It classifies data in two steps:

1. Training step: Using the training samples, the method estimates the parameters of a probability distribution, assuming features are conditionally independent given the class.

2. Prediction step: For any unseen test sample, the method computes the posterior probability of that sample belonging to each class. The method then classifies the test sample according the largest posterior probability.

The class-conditional independence assumption greatly simplifies the training step since you can estimate the one-dimensional class-conditional density for each feature individually. While the class-conditional independence between features is not true in general, research shows that this optimistic assumption works well in practice. This assumption of class independence allows the Naive Bayes classifier to better estimate the parameters required for accurate classification while using less training data than many other classifiers. This makes it particularly effective for datasets containing many predictors or features.

To learn how to prepare your data for Naive Bayes classification and create a classifier, see “Steps in Supervised Learning (Machine Learning)” on page 13-2.

Supported Distributions

Naive Bayes classification is based on estimating `P(X|Y)`, the probability or probability density of features `X` given class `Y`. The Naive Bayes classification object `NaiveBayes` provides support for normal (Gaussian), kernel, multinomial, and multivariate multinomial distributions. It is possible to use different distributions for different features.

Normal (Gaussian) Distribution

The `'normal'` distribution is appropriate for features that have normal distributions in each class. For each feature you model with a normal distribution, the Naive Bayes classifier estimates a separate normal distribution for each class by computing the mean and standard deviation of the training data in that class. For more information on normal distributions, see “Normal Distribution” on page B-83.

Kernel Distribution

The `'kernel'` distribution is appropriate for features that have a continuous distribution. It does not require a strong assumption such as a normal distribution and you can use it in cases where the distribution of a feature may be skewed or have multiple peaks or modes. It requires more computing time and more memory than the normal distribution. For each feature you model with a kernel distribution, the Naive Bayes classifier computes a separate kernel density estimate for each class based on the training data for that class. By default the kernel is the normal kernel, and the classifier selects a width automatically for each class and feature. It is possible to specify different kernels for each feature, and different widths for each feature or class.

Multinomial Distribution

The multinomial distribution (specify with the `'mn'` keyword) is appropriate when all features represent counts of a set of words or tokens. This is sometimes called the "bag of words" model. For example, an email spam classifier might be based on features that count the number of occurrences of various tokens in an email. One feature might count the number of exclamation points, another might count the number of times the word "money" appears, and another might count the number of times the recipient’s name appears. This is a Naive Bayes model under the further assumption that the total number of tokens (or the total document length) is independent of response class.

For the multinomial option, each feature represents the count of one token. The classifier counts the set of relative token probabilities separately for each class. The classifier defines the multinomial distribution for each row by the vector of probabilities for the corresponding class, and by `N`, the total token count for that row.

Classification is based on the relative frequencies of the tokens. For a row in which no token appears, `N` is 0 and no classification is possible. This classifier is not appropriate when the total number of tokens provides information about the response class.

Multivariate Multinomial Distribution

The multivariate multinomial distribution (specify with the `'mvmn'` keyword) is appropriate for categorical features. For example, you could fit a feature describing the weather in categories such as rain/sun/snow/clouds using the multivariate multinomial model. The feature categories are sometimes called the feature levels, and differ from the class levels for the response variable.

For each feature you model with a multivariate multinomial distribution, the Naive Bayes classifier computes a separate set of probabilities for the set of feature levels for each class.

Performance Curves

In this section...

Introduction to Performance Curves

After a classification algorithm such as `NaiveBayes` or `TreeBagger` has trained on data, you may want to examine the performance of the algorithm on a specific test dataset. One common way of doing this would be to compute a gross measure of performance such as quadratic loss or accuracy, averaged over the entire test dataset.

What are ROC Curves?

You may want to inspect the classifier performance more closely, for example, by plotting a Receiver Operating Characteristic (ROC) curve. By definition, a ROC curve [1,2] shows true positive rate versus false positive rate (equivalently, sensitivity versus 1–specificity) for different thresholds of the classifier output. You can use it, for example, to find the threshold that maximizes the classification accuracy or to assess, in more broad terms, how the classifier performs in the regions of high sensitivity and high specificity.

Evaluating Classifier Performance Using perfcurve

`perfcurve` computes measures for a plot of classifier performance. You can use this utility to evaluate classifier performance on test data after you train the classifier. Various measures such as mean squared error, classification error, or exponential loss can summarize the predictive power of a classifier in a single number. However, a performance curve offers more information as it lets you explore the classifier performance across a range of thresholds on its output.

You can use `perfcurve` with any classifier or, more broadly, with any method that returns a numeric score for an instance of input data. By convention adopted here,

- A high score returned by a classifier for any given instance signifies that the instance is likely from the positive class.
- A low score signifies that the instance is likely from the negative classes.

For some classifiers, you can interpret the score as the posterior probability of observing an instance of the positive class at point `X`. An example of such a score is the fraction of positive observations in a leaf of a decision tree. In this case, scores fall into the range from 0 to 1 and scores from positive and negative classes add up to unity. Other methods can return scores ranging between minus and plus infinity, without any obvious mapping from the score to the posterior class probability.

`perfcurve` does not impose any requirements on the input score range. Because of this lack of normalization, you can use `perfcurve` to process scores returned by any classification, regression, or fit method. `perfcurve` does not make any assumptions about the nature of input scores or relationships between the scores for different classes. As an example, consider a problem with three classes, `A`, `B`, and `C`, and assume that the scores returned by some classifier for two instances are as follows:

	`A`	`B`	`C`
instance 1	0.4	0.5	0.1
instance 2	0.4	0.1	0.5

If you want to compute a performance curve for separation of classes `A` and `B`, with `C` ignored, you need to address the ambiguity in selecting `A` over `B`. You could opt to use the score ratio, `s(A)/s(B)`, or score difference, `s(A)-s(B)`; this choice could depend on the nature of these scores and their normalization. `perfcurve` always takes one score per instance. If you only supply scores for class `A`, `perfcurve` does not distinguish between observations 1 and 2. The performance curve in this case may not be optimal.

`perfcurve` is intended for use with classifiers that return scores, not those that return only predicted classes. As a counter-example, consider a decision tree that returns only hard classification labels, 0 or 1, for data with two classes. In this case, the performance curve reduces to a single point because classified instances can be split into positive and negative categories in one way only.

For input, `perfcurve` takes true class labels for some data and scores assigned by a classifier to these data. By default, this utility computes a Receiver Operating Characteristic (ROC) curve and returns values of 1–specificity, or false positive rate, for `X` and sensitivity, or true positive rate, for `Y`. You can choose other criteria for `X` and `Y` by selecting one out of several provided criteria or specifying an arbitrary criterion through an anonymous function. You can display the computed performance curve using `plot(X,Y)`.

`perfcurve` can compute values for various criteria to plot either on the x- or the y-axis. All such criteria are described by a 2-by-2 confusion matrix, a 2-by-2 cost matrix, and a 2-by-1 vector of scales applied to class counts.

The `confusion` matrix, `C`, is defined as

$$\begin{pmatrix} TP & FN \\ FP & TN \end{pmatrix}$$

where

- P stands for "positive".
- N stands for "negative".
- T stands for "true".
- F stands for "false".

For example, the first row of the confusion matrix defines how the classifier identifies instances of the positive class: `C(1,1)` is the count of correctly identified positive instances and `C(1,2)` is the count of positive instances misidentified as negative.

The cost matrix defines the cost of misclassification for each category:

$$\begin{pmatrix} Cost(P \mid P) & Cost(N \mid P) \\ Cost(P \mid N) & Cost(N \mid N) \end{pmatrix}$$

where `Cost(I|J)` is the cost of assigning an instance of class `J` to class `I`. Usually `Cost(I|J)=0` for `I=J`. For flexibility, `perfcurve` allows you to specify nonzero costs for correct classification as well.

The two scales include prior information about class probabilities. `perfcurve` computes these scales by taking `scale(P)=prior(P)*N` and `scale(N)=prior(N)*P` and normalizing the sum `scale(P)+scale(N)` to 1. `P=TP+FN` and `N=TN+FP` are the total instance counts in the positive and negative class, respectively. The function then applies the scales as multiplicative factors to the counts from the corresponding class: `perfcurve` multiplies counts from the positive class by `scale(P)` and counts from the negative class by `scale(N)`. Consider, for example, computation of positive predictive value, `PPV = TP/(TP+FP)`. `TP` counts come from the positive class and `FP` counts come from the negative class. Therefore, you need to scale `TP` by `scale(P)` and `FP` by `scale(N)`, and the modified formula for `PPV` with prior probabilities taken into account is now:

$$PPV = \frac{scale(P) * TP}{scale(P) * TP + scale(N) * FP}$$

If all scores in the data are above a certain threshold, `perfcurve` classifies all instances as `'positive'`. This means that `TP` is the total number of instances in the positive class and `FP` is the total number of instances in the negative class. In this case, `PPV` is simply given by the prior:

$$PPV = \frac{prior(P)}{prior(P) + prior(N)}$$

The `perfcurve` function returns two vectors, `X` and `Y`, of performance measures. Each measure is some function of `confusion`, `cost`, and `scale` values. You can request specific measures by name or provide a function handle to compute a custom measure. The function you provide should take `confusion`, `cost`, and `scale` as its three inputs and return a vector of output values.

The criterion for `X` must be a monotone function of the positive classification count, or equivalently, threshold for the supplied scores. If `perfcurve` cannot perform a one-to-one mapping between values of the `X` criterion and score thresholds, it exits with an error message.

By default, `perfcurve` computes values of the `X` and `Y` criteria for all possible score thresholds. Alternatively, it can compute a reduced number of specific `X` values supplied as an input argument. In either case, for `M` requested values, `perfcurve` computes `M+1` values for `X` and `Y`. The first value out of these `M+1` values is special. `perfcurve` computes it by setting the `TP` instance count

to zero and setting TN to the total count in the negative class. This value corresponds to the 'reject all' threshold. On a standard ROC curve, this translates into an extra point placed at (0,0).

If there are NaN values among input scores, perfcurve can process them in either of two ways:

- It can discard rows with NaN scores.
- It can add them to false classification counts in the respective class.

That is, for any threshold, instances with NaN scores from the positive class are counted as false negative (FN), and instances with NaN scores from the negative class are counted as false positive (FP). In this case, the first value of X or Y is computed by setting TP to zero and setting TN to the total count minus the NaN count in the negative class. For illustration, consider an example with two rows in the positive and two rows in the negative class, each pair having a NaN score:

Class	Score
Negative	0.2
Negative	NaN
Positive	0.7
Positive	NaN

If you discard rows with NaN scores, then as the score cutoff varies, perfcurve computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where rows 1 and 3 are classified correctly, and rows 2 and 4 are omitted.

TP	FN	FP	TN
0	1	0	1
1	0	0	1
1	0	1	0

If you add rows with NaN scores to the false category in their respective classes, perfcurve computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where now rows

2 and 4 are counted as incorrectly classified. Notice that only the FN and FP columns differ between these two tables.

TP	FN	FP	TN
0	2	1	1
1	1	1	1
1	1	2	0

For data with three or more classes, perfcurve takes one positive class and a list of negative classes for input. The function computes the X and Y values using counts in the positive class to estimate TP and FN, and using counts in all negative classes to estimate TN and FP. perfcurve can optionally compute Y values for each negative class separately and, in addition to Y, return a matrix of size M-by-C, where M is the number of elements in X or Y and C is the number of negative classes. You can use this functionality to monitor components of the negative class contribution. For example, you can plot TP counts on the X-axis and FP counts on the Y-axis. In this case, the returned matrix shows how the FP component is split across negative classes.

You can also use perfcurve to estimate confidence intervals. perfcurve computes confidence bounds using either cross-validation or bootstrap. If you supply cell arrays for labels and scores, perfcurve uses cross-validation and treats elements in the cell arrays as cross-validation folds. If you set input parameter NBoot to a positive integer, perfcurve generates nboot bootstrap replicas to compute pointwise confidence bounds.

perfcurve estimates the confidence bounds using one of two methods:

- Vertical averaging (VA) — estimate confidence bounds on Y and T at fixed values of X. Use the XVals input parameter to use this method for computing confidence bounds.
- Threshold averaging (TA) — estimate confidence bounds for X and Y at fixed thresholds for the positive class score. Use the TVals input parameter to use this method for computing confidence bounds.

To use observation weights instead of observation counts, you can use the 'Weights' parameter in your call to perfcurve. When you use this parameter, to compute X, Y and T or to compute confidence bounds by cross-validation, perfcurve uses your supplied observation weights instead of

observation counts. To compute confidence bounds by bootstrap, `perfcurve` samples N out of N with replacement using your weights as multinomial sampling probabilities.