10

# Multivariate Methods

# Introduction to Multivariate Methods

Large, high-dimensional data sets are common in the modern era of computer-based instrumentation and electronic data storage. High-dimensional data present many challenges for statistical visualization, analysis, and modeling.

Data visualization, of course, is impossible beyond a few dimensions. As a result, pattern recognition, data preprocessing, and model selection must rely heavily on numerical methods.

A fundamental challenge in high-dimensional data analysis is the so-called *curse of dimensionality*. Observations in a high-dimensional space are necessarily sparser and less representative than those in a low-dimensional space. In higher dimensions, data over-represent the edges of a sampling distribution, because regions of higher-dimensional space contain the majority of their volume near the surface. (A $d$-dimensional spherical shell has a volume, relative to the total volume of the sphere, that approaches 1 as $d$ approaches infinity.) In high dimensions, typical data points at the interior of a distribution are sampled less frequently.

Often, many of the dimensions in a data set—the measured features—are not useful in producing a model. Features may be irrelevant or redundant. Regression and classification algorithms may require large amounts of storage and computation time to process raw data, and even if the algorithms are successful the resulting models may contain an incomprehensible number of terms.

Because of these challenges, multivariate statistical methods often begin with some type of *dimension reduction*, in which data are approximated by points in a lower-dimensional space. Dimension reduction is the goal of the methods presented in this chapter. Dimension reduction often leads to simpler models and fewer measured variables, with consequent benefits when measurements are expensive and visualization is important.

# Multidimensional Scaling

**In this section...**



## Introduction to Multidimensional Scaling

One of the most important goals in visualizing data is to get a sense of how near or far points are from each other. Often, you can do this with a scatter plot. However, for some analyses, the data that you have might not be in the form of points at all, but rather in the form of pairwise similarities or dissimilarities between cases, observations, or subjects. There are no points to plot.

Even if your data are in the form of points rather than pairwise distances, a scatter plot of those data might not be useful. For some kinds of data, the relevant way to measure how near two points are might not be their Euclidean distance. While scatter plots of the raw data make it easy to compare Euclidean distances, they are not always useful when comparing other kinds of inter-point distances, city block distance for example, or even more general dissimilarities. Also, with a large number of variables, it is very difficult to visualize distances unless the data can be represented in a small number of dimensions. Some sort of dimension reduction is usually necessary.

Multidimensional scaling (MDS) is a set of methods that address all these problems. MDS allows you to visualize how near points are to each other for many kinds of distance or dissimilarity metrics and can produce a representation of your data in a small number of dimensions. MDS does not require raw data, but only a matrix of pairwise distances or dissimilarities.

## Classical Multidimensional Scaling

## Introduction to Classical Multidimensional Scaling

The function `cmdscale` performs classical (metric) multidimensional scaling, also known as *principal coordinates analysis*. `cmdscale` takes as an input a matrix of inter-point distances and creates a configuration of points. Ideally, those points are in two or three dimensions, and the Euclidean distances between them reproduce the original distance matrix. Thus, a scatter plot of the points created by `cmdscale` provides a visual representation of the original distances.

As a very simple example, you can reconstruct a set of points from only their inter-point distances. First, create some four dimensional points with a small component in their fourth coordinate, and reduce them to distances.

```
X = [ normrnd(0,1,10,3), normrnd(0,.1,10,1) ];
D = pdist(X,'euclidean');
```

Next, use `cmdscale` to find a configuration with those inter-point distances. `cmdscale` accepts distances as either a square matrix, or, as in this example, in the vector upper-triangular form produced by `pdist`.

```
[Y,eigvals] = cmdscale(D);
```

`cmdscale` produces two outputs. The first output, `Y`, is a matrix containing the reconstructed points. The second output, `eigvals`, is a vector containing the sorted eigenvalues of what is often referred to as the "scalar product matrix," which, in the simplest case, is equal to `Y*Y'`. The relative magnitudes of those eigenvalues indicate the relative contribution of the corresponding columns of `Y` in reproducing the original distance matrix `D` with the reconstructed points.

```
format short g
[eigvals eigvals/max(abs(eigvals))]
ans =
       12.623            1
       4.3699      0.34618
       1.9307      0.15295
     0.025884    0.0020505
  1.7192e-015  1.3619e-016
  6.8727e-016  5.4445e-017
```

```
  4.4367e-017   3.5147e-018
 -9.2731e-016 -7.3461e-017
  -1.327e-015 -1.0513e-016
 -1.9232e-015 -1.5236e-016
```

If `eigvals` contains only positive and zero (within round-off error) eigenvalues, the columns of `Y` corresponding to the positive eigenvalues provide an exact reconstruction of `D`, in the sense that their inter-point Euclidean distances, computed using `pdist`, for example, are identical (within round-off) to the values in `D`.

```
maxerr4 = max(abs(D - pdist(Y))) % exact reconstruction
maxerr4 =
  2.6645e-015
```

If two or three of the eigenvalues in `eigvals` are much larger than the rest, then the distance matrix based on the corresponding columns of `Y` nearly reproduces the original distance matrix `D`. In this sense, those columns form a lower-dimensional representation that adequately describes the data. However it is not always possible to find a good low-dimensional reconstruction.

```
% good reconstruction in 3D
maxerr3 = max(abs(D - pdist(Y(:,1:3))))
maxerr3 =
     0.029728

% poor reconstruction in 2D
maxerr2 = max(abs(D - pdist(Y(:,1:2))))
maxerr2 =
      0.91641
```

The reconstruction in three dimensions reproduces `D` very well, but the reconstruction in two dimensions has errors that are of the same order of magnitude as the largest values in `D`.

```
max(max(D))
ans =
       3.4686
```

Often, `eigvals` contains some negative eigenvalues, indicating that the distances in `D` cannot be reproduced exactly. That is, there might not be any configuration of points whose inter-point Euclidean distances are given by `D`. If the largest negative eigenvalue is small in magnitude relative to the largest positive eigenvalues, then the configuration returned by `cmdscale` might still reproduce `D` well.

## Example: Multidimensional Scaling

Given only the distances between 10 US cities, `cmdscale` can construct a map of those cities. First, create the distance matrix and pass it to `cmdscale`. In this example,`D` is a full distance matrix: it is square and symmetric, has positive entries off the diagonal, and has zeros on the diagonal.

```
cities = ...
{'Atl','Chi','Den','Hou','LA','Mia','NYC','SF','Sea','WDC'};
D = [    0  587 1212  701 1936  604  748 2139 2182   543;
       587    0  920  940 1745 1188  713 1858 1737   597;
      1212  920    0  879  831 1726 1631  949 1021  1494;
       701  940  879    0 1374  968 1420 1645 1891  1220;
      1936 1745  831 1374    0 2339 2451  347  959  2300;
       604 1188 1726  968 2339    0 1092 2594 2734   923;
       748  713 1631 1420 2451 1092    0 2571 2408   205;
      2139 1858  949 1645  347 2594 2571    0  678  2442;
      2182 1737 1021 1891  959 2734 2408  678    0  2329;
       543  597 1494 1220 2300  923  205 2442 2329     0];
[Y,eigvals] = cmdscale(D);
```

Next, look at the eigenvalues returned by `cmdscale`. Some of these are negative, indicating that the original distances are not Euclidean. This is because of the curvature of the earth.

```
format short g
[eigvals eigvals/max(abs(eigvals))]
ans =
  9.5821e+006            1
  1.6868e+006      0.17604
       8157.3    0.0008513
       1432.9   0.00014954
       508.67  5.3085e-005
       25.143   2.624e-006
```

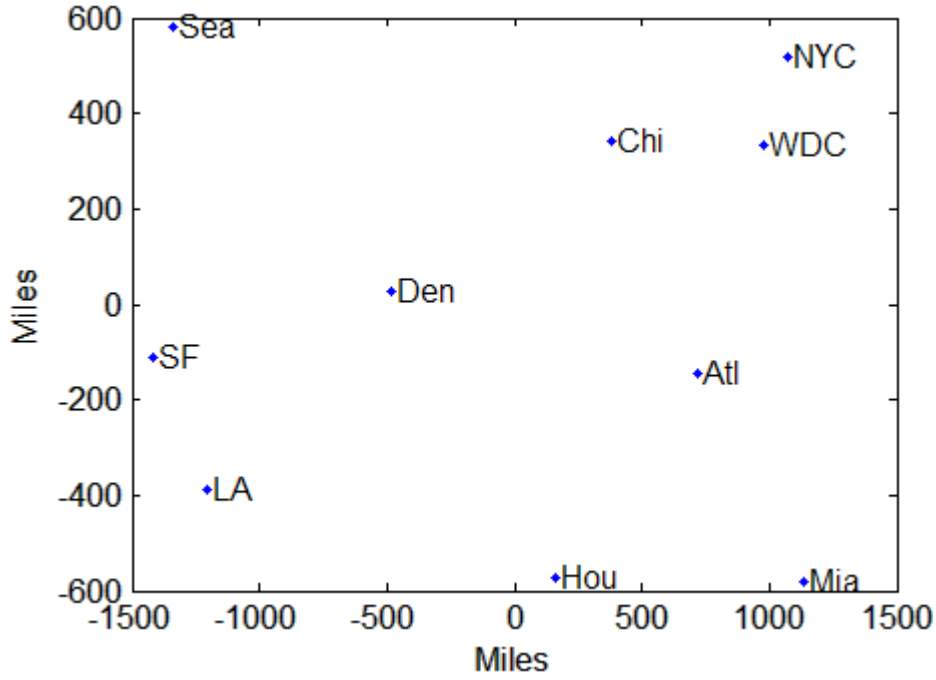
```
  5.3394e-010  5.5722e-017
       -897.7 -9.3685e-005
      -5467.6   -0.0005706
       -35479   -0.0037026
```

However, in this case, the two largest positive eigenvalues are much larger in magnitude than the remaining eigenvalues. So, despite the negative eigenvalues, the first two coordinates of `Y` are sufficient for a reasonable reproduction of `D`.

```
Dtriu = D(find(tril(ones(10),-1)))';
maxrelerr = max(abs(Dtriu-pdist(Y(:,1:2))))./max(Dtriu)
maxrelerr =
    0.0075371
```

Here is a plot of the reconstructed city locations as a map. The orientation of the reconstruction is arbitrary. In this case, it happens to be close to, although not exactly, the correct orientation.

```
plot(Y(:,1),Y(:,2),'.')
text(Y(:,1)+25,Y(:,2),cities)
xlabel('Miles')
ylabel('Miles')
```




## Nonclassical Multidimensional Scaling

The function `mdscale` performs nonclassical multidimensional scaling. As with `cmdcale`, you use `mdscale` either to visualize dissimilarity data for which no "locations" exist, or to visualize high-dimensional data by reducing its dimensionality. Both functions take a matrix of dissimilarities as an input and produce a configuration of points. However, `mdscale` offers a choice of different criteria to construct the configuration, and allows missing data and weights.

For example, the cereal data include measurements on 10 variables describing breakfast cereals. You can use `mdscale` to visualize these data in two dimensions. First, load the data. For clarity, this example code selects a subset of 22 of the observations.

```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber ...
    Carbo Sugars Shelf Potass Vitamins];
```

```
% Take a subset from a single manufacturer
mfg1 = strcmp('G',cellstr(Mfg));
X = X(mfg1,:);
size(X)
ans =
    22 10
```

Then use `pdist` to transform the 10-dimensional data into dissimilarities. The output from `pdist` is a symmetric dissimilarity matrix, stored as a vector containing only the (23*22/2) elements in its upper triangle.

```
dissimilarities = pdist(zscore(X),'cityblock');
size(dissimilarities)
ans =
     1   231
```

This example code first standardizes the cereal data, and then uses city block distance as a dissimilarity. The choice of transformation to dissimilarities is application-dependent, and the choice here is only for simplicity. In some applications, the original data are already in the form of dissimilarities.

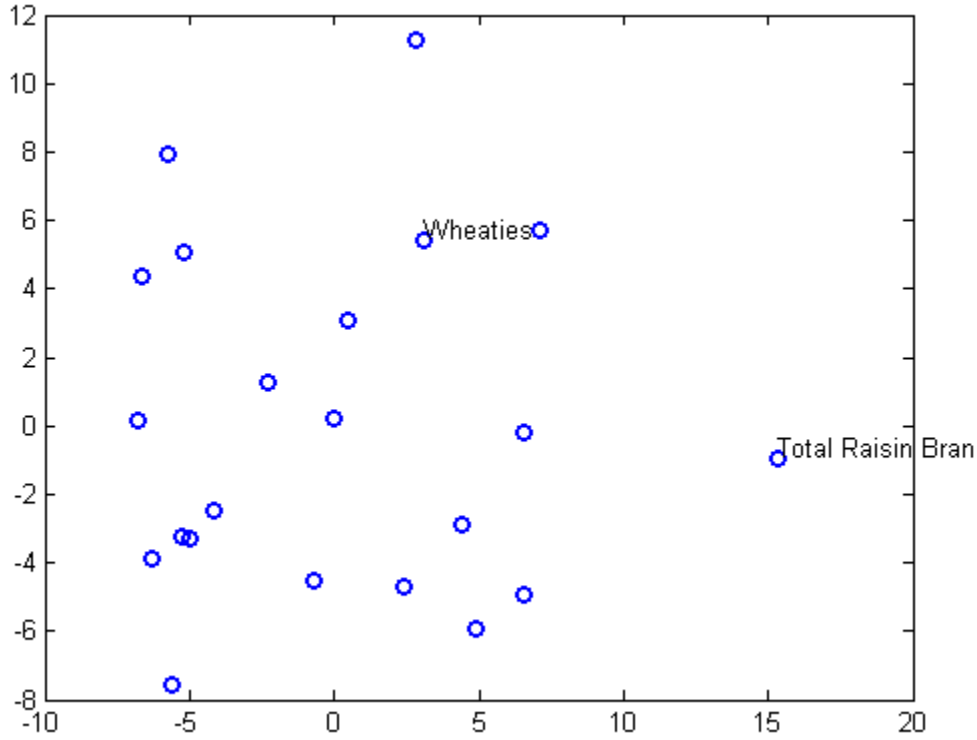
Next, use `mdscale` to perform metric MDS. Unlike `cmdscale`, you must specify the desired number of dimensions, and the method to use to construct the output configuration. For this example, use two dimensions. The metric STRESS criterion is a common method for computing the output; for other choices, see the `mdscale` reference page in the online documentation. The second output from `mdscale` is the value of that criterion evaluated for the output configuration. It measures the how well the inter-point distances of the output configuration approximate the original input dissimilarities:

```
[Y,stress] =...
mdscale(dissimilarities,2,'criterion','metricstress');
stress
stress =
    0.1856
```

A scatterplot of the output from `mdscale` represents the original 10-dimensional data in two dimensions, and you can use the `gname` function to label selected points:

```
plot(Y(:,1),Y(:,2),'o','LineWidth',2);
```

```
gname(Name(mfg1))
```




## Nonmetric Multidimensional Scaling

Metric multidimensional scaling creates a configuration of points whose inter-point distances approximate the given dissimilarities. This is sometimes too strict a requirement, and non-metric scaling is designed to relax it a bit. Instead of trying to approximate the dissimilarities themselves, non-metric scaling approximates a nonlinear, but monotonic, transformation of them. Because of the monotonicity, larger or smaller distances on a plot of the output will correspond to larger or smaller dissimilarities, respectively. However, the nonlinearity implies that `mdscale` only attempts to preserve the

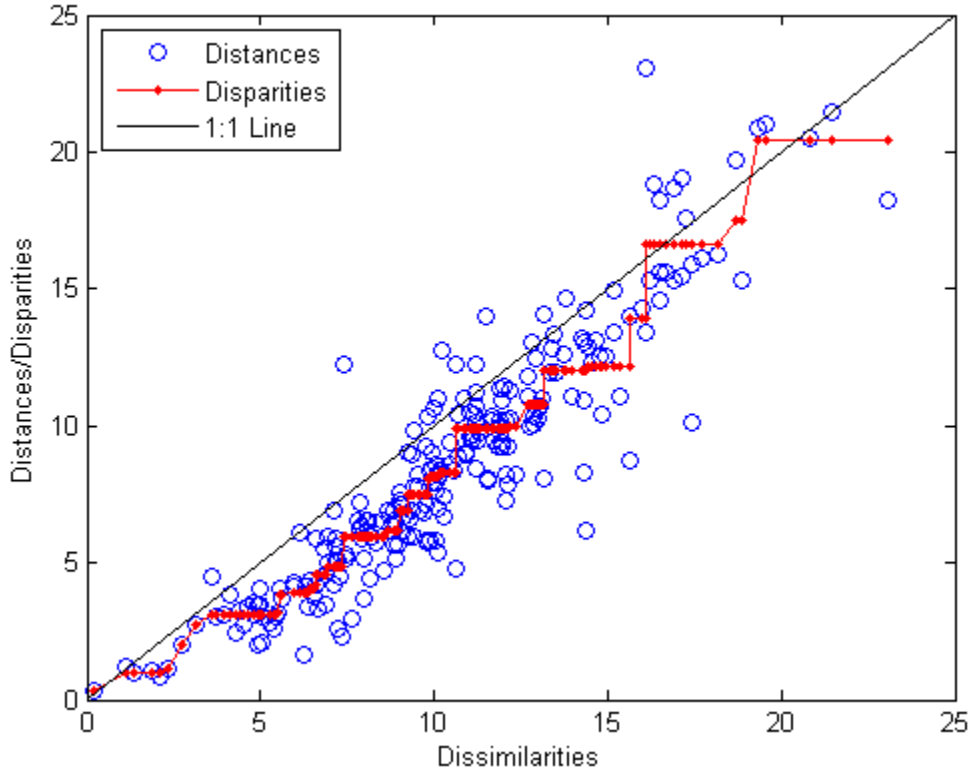
ordering of dissimilarities. Thus, there may be contractions or expansions of distances at different scales.

You use `mdscale` to perform nonmetric MDS in much the same way as for metric scaling. The nonmetric STRESS criterion is a common method for computing the output; for more choices, see the `mdscale` reference page in the online documentation. As with metric scaling, the second output from `mdscale` is the value of that criterion evaluated for the output configuration. For nonmetric scaling, however, it measures the how well the inter-point distances of the output configuration approximate the disparities. The disparities are returned in the third output. They are the transformed values of the original dissimilarities:

```
[Y,stress,disparities] = ...
mdscale(dissimilarities,2,'criterion','stress');
stress
stress =
    0.1562
```

To check the fit of the output configuration to the dissimilarities, and to understand the disparities, it helps to make a Shepard plot:

```
distances = pdist(Y);
[dum,ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities,distances,'bo', ...
     dissimilarities(ord),disparities(ord),'r.-', ...
     [0 25],[0 25],'k-')
xlabel('Dissimilarities')
ylabel('Distances/Disparities')
legend({'Distances' 'Disparities' '1:1 Line'},...
       'Location','NorthWest');
```




This plot shows that `mdscale` has found a configuration of points in two dimensions whose inter-point distances approximates the disparities, which in turn are a nonlinear transformation of the original dissimilarities. The concave shape of the disparities as a function of the dissimilarities indicates that fit tends to contract small distances relative to the corresponding dissimilarities. This may be perfectly acceptable in practice.

`mdscale` uses an iterative algorithm to find the output configuration, and the results can often depend on the starting point. By default, `mdscale` uses `cmdscale` to construct an initial configuration, and this choice often leads to a globally best solution. However, it is possible for `mdscale` to stop at a configuration that is a local minimum of the criterion. Such

cases can be diagnosed and often overcome by running `mdscale` multiple times with different starting points. You can do this using the `'start'` and `'replicates'` parameters. The following code runs five replicates of MDS, each starting at a different randomly-chosen initial configuration. The criterion value is printed out for each replication; `mdscale` returns the configuration with the best fit.

```
opts = statset('Display','final');
[Y,stress] =...
mdscale(dissimilarities,2,'criterion','stress',...
'start','random','replicates',5,'Options',opts);

35 iterations, Final stress criterion = 0.156209
31 iterations, Final stress criterion = 0.156209
48 iterations, Final stress criterion = 0.171209
33 iterations, Final stress criterion = 0.175341
32 iterations, Final stress criterion = 0.185881
```

Notice that `mdscale` finds several different local solutions, some of which do not have as low a stress value as the solution found with the `cmdscale` starting point.

# Procrustes Analysis

**In this section...**



## Comparing Landmark Data

The `procrustes` function analyzes the distribution of a set of shapes using Procrustes analysis. This analysis method matches landmark data (geometric locations representing significant features in a given shape) to calculate the best shape-preserving Euclidian transformations. These transformations minimize the differences in location between compared landmark data.

Procrustes analysis is also useful in conjunction with multidimensional scaling. In "Example: Multidimensional Scaling" on page 10-6 there is an observation that the orientation of the reconstructed points is arbitrary. Two different applications of multidimensional scaling could produce reconstructed points that are very similar in principle, but that look different because they have different orientations. The `procrustes` function transforms one set of points to make them more comparable to the other.

## Data Input

The `procrustes` function takes two matrices as input:

- The target shape matrix $X$ has dimension $n \times p$, where $n$ is the number of landmarks in the shape and $p$ is the number of measurements per landmark.
- The comparison shape matrix $Y$ has dimension $n \times q$ with $q \leq p$. If there are fewer measurements per landmark for the comparison shape than the target shape ($q < p$), the function adds columns of zeros to $Y$, yielding an $n \times p$ matrix.

The equation to obtain the transformed shape, $Z$, is

$$Z = bYT + c \qquad \textbf{(10-1)}$$

where:

- $b$ is a scaling factor that stretches ($b > 1$) or shrinks ($b < 1$) the points.
- $T$ is the orthogonal rotation and reflection matrix.
- $c$ is a matrix with constant values in each column, used to shift the points.

The `procrustes` function chooses $b$, $T$, and $c$ to minimize the distance between the target shape $X$ and the transformed shape $Z$ as measured by the least squares criterion:

$$\sum_{i=1}^{n}\sum_{j=1}^{p}(X_{ij} - Z_{ij})^2$$

## Preprocessing Data for Accurate Results

Procrustes analysis is appropriate when all `p` measurement dimensions have similar scales. The analysis would be inaccurate, for example, if the columns of $Z$ had different scales:

- The first column is measured in milliliters ranging from 2,000 to 6,000.
- The second column is measured in degrees Celsius ranging from 10 to 25.
- The third column is measured in kilograms ranging from 50 to 230.

In such cases, standardize your variables by:

1. Subtracting the sample mean from each variable.
2. Dividing each resultant variable by its sample standard deviation.

Use the `zscore` function to perform this standardization.

## Example: Comparing Handwritten Shapes

In this example, use Procrustes analysis to compare two handwritten number threes. Visually and analytically explore the effects of forcing size and reflection changes as follows:



### Step 1: Load and Display the Original Data

Input landmark data for two handwritten number threes:

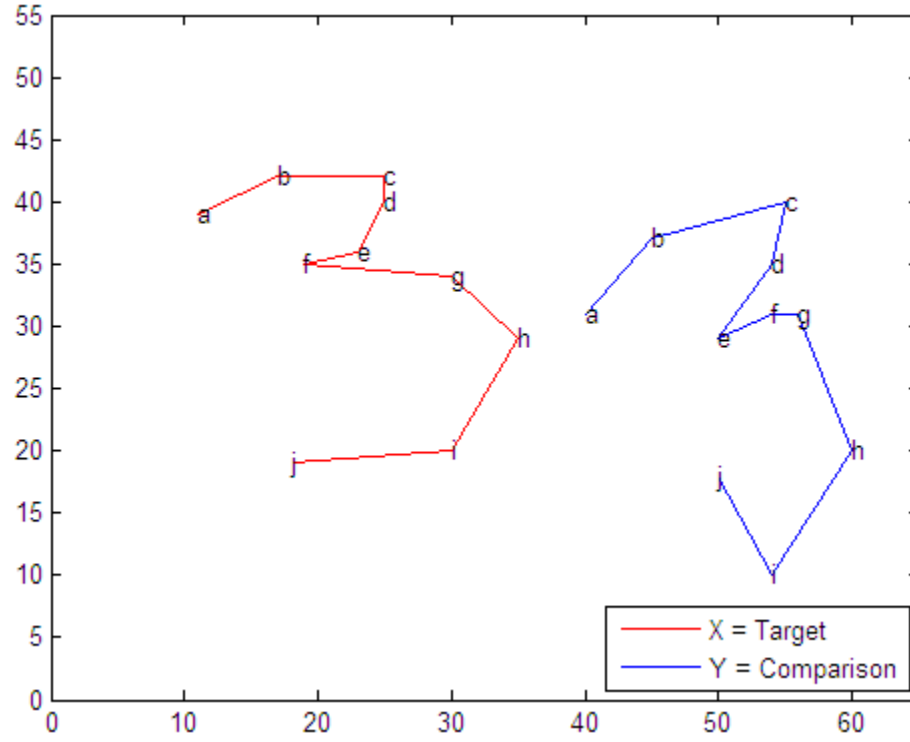
```
A = [11 39;17 42;25 42;25 40;23 36;19 35;30 34;35 29;...
30 20;18 19];
B = [15 31;20 37;30 40;29 35;25 29;29 31;31 31;35 20;...
29 10;25 18];
```

Create `X` and `Y` from `A` and `B`, moving `B` to the side to make each shape more visible:

```
X = A;
Y = B + repmat([25 0], 10,1);
```

Plot the shapes, using letters to designate the landmark points. Lines in the figure join the points to indicate the drawing path of each shape.

```
plot(X(:,1), X(:,2),'r-', Y(:,1), Y(:,2),'b-');
text(X(:,1), X(:,2),('abcdefghij')')
text(Y(:,1), Y(:,2),('abcdefghij')')
legend('X = Target','Y = Comparison','location','SE')
set(gca,'YLim',[0 55],'XLim',[0 65]);
```




## Step 2: Calculate the Best Transformation

Use Procrustes analysis to find the transformation that minimizes distances between landmark data points.

Call `procrustes` as follows:

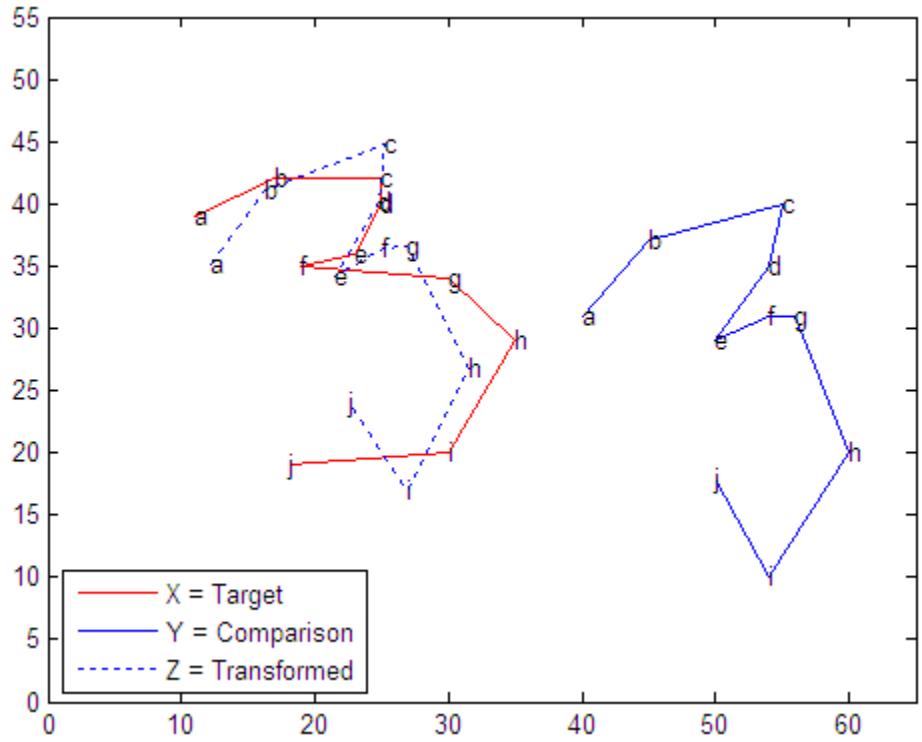
```
[d, Z, tr] = procrustes(X,Y);
```

The outputs of the function are:

- `d` – A standardized dissimilarity measure.)
- `Z` – A matrix of the transformed landmarks.
- `tr` – A structure array of the computed transformation with fields `T`, `b`, and `c` which correspond to the transformation equation, Equation 10-1.

Visualize the transformed shape, Z, using a dashed blue line:

```
plot(X(:,1), X(:,2),'r-', Y(:,1), Y(:,2),'b-',...
Z(:,1),Z(:,2),'b:');
text(X(:,1), X(:,2),('abcdefghij')')
text(Y(:,1), Y(:,2),('abcdefghij')')
text(Z(:,1), Z(:,2),('abcdefghij')')
legend('X = Target','Y = Comparison',...
'Z = Transformed','location','SW')
set(gca,'YLim',[0 55],'XLim',[0 65]);
```




## Step 3: Examine the Similarity of the Two Shapes

Use two different numerical values to assess the similarity of the target shape and the transformed shape.

**Dissimilarity Measure d.** The dissimilarity measure `d` gives a number between 0 and 1 describing the difference between the target shape and the transformed shape. Values near 0 imply more similar shapes, while values near 1 imply dissimilarity. For this example:

```
d =
     0.1502
```

The small value of `d` in this case shows that the two shapes are similar.

`procrustes` calculates `d` by comparing the sum of squared deviations between the set of points with the sum of squared deviations of the original points from their column means:

```
numerator = sum(sum((X-Z).^2))
numerator =

  166.5321

denominator = sum(sum(bsxfun(@minus,X,mean(X)).^2))
denominator =

  1.1085e+003

ratio = numerator/denominator
ratio =

    0.1502
```

---

**Note** The resulting measure `d` is independent of the scale of the size of the shapes and takes into account only the similarity of landmark data. "Examining the Scaling Measure b" on page 10-19 shows how to examine the size similarity of the shapes.

---

**Examining the Scaling Measure b.** The target and comparison threes in the previous figure visually show that the two numbers are of a similar size. The closeness of calculated value of the scaling factor `b` to 1 supports this observation as well:

```
tr.b
ans =
    0.9291
```

The sizes of the target and comparison shapes appear similar. This visual impression is reinforced by the value of `b` = 0.93, which implies that the best transformation results in shrinking the comparison shape by a factor .93 (only 7%).

### Step 4: Restrict the Form of the Transformations

Explore the effects of manually adjusting the scaling and reflection coefficients.

**Fixing the Scaling Factor `b = 1`.** Force `b` to equal 1 (set `'Scaling'` to `false`) to examine the amount of dissimilarity in size of the target and transformed figures:

```
ds = procrustes(X,Y,'Scaling',false)
ds =
    0.1552
```

In this case, setting `'Scaling'` to `false` increases the calculated value of `d` only 0.0049, which further supports the similarity in the size of the two number threes. A larger increase in `d` would have indicated a greater size discrepancy.

**Forcing a Reflection in the Transformation.** This example requires only a rotation, not a reflection, to align the shapes. You can show this by observing that the determinant of the matrix `T` is 1 in this analysis:

```
det(tr.T)
ans =
    1.0000
```

If you need a reflection in the transformation, the determinant of `T` is -1. You can force a reflection into the transformation as follows:

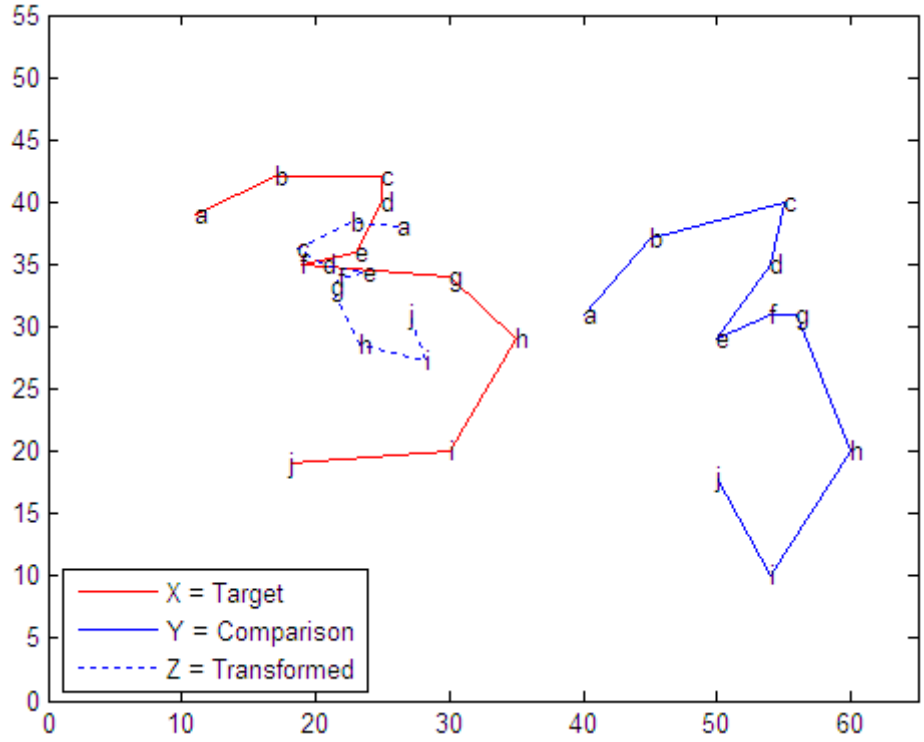
```
[dr,Zr,trr] = procrustes(X,Y,'Reflection',true);
dr
dr =
```

```
    0.8130
```

The `d` value increases dramatically, indicating that a forced reflection leads to a poor transformation of the landmark points. A plot of the transformed shape shows a similar result:

- The landmark data points are now further away from their target counterparts.
- The transformed three is now an undesirable mirror image of the target three.

```
plot(X(:,1), X(:,2),'r-', Y(:,1), Y(:,2),'b-',...
Zr(:,1),Zr(:,2),'b:');
text(X(:,1), X(:,2),('abcdefghij')')
text(Y(:,1), Y(:,2),('abcdefghij')')
text(Zr(:,1), Zr(:,2),('abcdefghij')')
legend('X = Target','Y = Comparison',...
'Z = Transformed','location','SW')
set(gca,'YLim',[0 55],'XLim',[0 65]);
```




It appears that the shapes might be better matched if you flipped the transformed shape upside down. Flipping the shapes would make the transformation even worse, however, because the landmark data points would be further away from their target counterparts. From this example, it is clear that manually adjusting the scaling and reflection parameters is generally not optimal.

# Feature Selection

| In this section... |
| --- |
|  |
|  |

## Introduction to Feature Selection

*Feature selection* reduces the dimensionality of data by selecting only a subset of measured features (predictor variables) to create a model. Selection criteria usually involve the minimization of a specific measure of predictive error for models fit to different subsets. Algorithms search for a subset of predictors that optimally model measured responses, subject to constraints such as required or excluded features and the size of the subset.

Feature selection is preferable to feature transformation when the original units and meaning of features are important and the modeling goal is to identify an influential subset. When categorical features are present, and numerical transformations are inappropriate, feature selection becomes the primary means of dimension reduction.

## Sequential Feature Selection



### Introduction to Sequential Feature Selection

A common method of feature selection is *sequential feature selection*. This method has two components:

- An objective function, called the *criterion*, which the method seeks to minimize over all feasible feature subsets. Common criteria are mean squared error (for regression models) and misclassification rate (for classification models).
- A sequential search algorithm, which adds or removes features from a candidate subset while evaluating the criterion. Since an exhaustive

comparison of the criterion value at all $2^n$ subsets of an $n$-feature data set is typically infeasible (depending on the size of $n$ and the cost of objective calls), sequential searches move in only one direction, always growing or always shrinking the candidate set.

The method has two variants:

- *Sequential forward selection* (*SFS*), in which features are sequentially added to an empty candidate set until the addition of further features does not decrease the criterion.
- *Sequential backward selection* (*SBS*), in which features are sequentially removed from a full candidate set until the removal of further features increase the criterion.

Stepwise regression is a sequential feature selection technique designed specifically for least-squares fitting. The functions `stepwise` and `stepwisefit` make use of optimizations that are only possible with least-squares criteria. Unlike generalized sequential feature selection, stepwise regression may remove features that have been added or add features that have been removed.

The Statistics Toolbox function `sequentialfs` carries out sequential feature selection. Input arguments include predictor and response data and a function handle to a file implementing the criterion function. Optional inputs allow you to specify SFS or SBS, required or excluded features, and the size of the feature subset. The function calls `cvpartition` and `crossval` to evaluate the criterion at different candidate sets.

## Example: Sequential Feature Selection

For example, consider a data set with 100 observations of 10 predictors. The following generates random data from a logistic model, with a binomial distribution of responses at each set of values for the predictors. Some coefficients are set to zero so that not all of the predictors affect the response:

```
n = 100;
m = 10;
X = rand(n,m);
b = [1 0 0 2 .5 0 0 0.1 0 1];
Xb = X*b';
p = 1./(1+exp(-Xb));
```

```
N = 50;
y = binornd(N,p);
```

The `glmfit` function fits a logistic model to the data:

```
Y = [y N*ones(size(y))];
[b0,dev0,stats0] = glmfit(X,Y,'binomial');

% Display coefficient estimates and their standard errors:
model0 = [b0 stats0.se]
model0 =
    0.3115    0.2596
    0.9614    0.1656
   -0.1100    0.1651
   -0.2165    0.1683
    1.9519    0.1809
    0.5683    0.2018
   -0.0062    0.1740
    0.0651    0.1641
   -0.1034    0.1685
    0.0017    0.1815
    0.7979    0.1806

% Display the deviance of the fit:
dev0
dev0 =
  101.2594
```

This is the full model, using all of the features (and an initial constant term). Sequential feature selection searches for a subset of the features in the full model with comparative predictive power.

First, you must specify a criterion for selecting the features. The following function, which calls `glmfit` and returns the deviance of the fit (a generalization of the residual sum of squares) is a useful criterion in this case:

```
function dev = critfun(X,Y)

[b,dev] = glmfit(X,Y,'binomial');
```

You should create this function as a file on the MATLAB path.

The function sequentialfs performs feature selection, calling the criterion function via a function handle:

```
maxdev = chi2inv(.95,1);
opt = statset('display','iter',...
              'TolFun',maxdev,...
              'TolTypeFun','abs');

inmodel = sequentialfs(@critfun,X,Y,...
                       'cv','none',...
                       'nullmodel',true,...
                       'options',opt,...
                       'direction','forward');
```

```
Start forward sequential feature selection:
Initial columns included:  none
Columns that can not be included:  none
Step 1, used initial columns, criterion value 309.118
Step 2, added column 4, criterion value 180.732
Step 3, added column 1, criterion value 138.862
Step 4, added column 10, criterion value 114.238
Step 5, added column 5, criterion value 103.503
Final columns included:  1 4 5 10
```

The iterative display shows a decrease in the criterion value as each new feature is added to the model. The final result is a reduced model with only four of the original ten features: columns 1, 4, 5, and 10 of X. These features are indicated in the logical vector inmodel returned by sequentialfs.

The deviance of the reduced model is higher than for the full model, but the addition of any other single feature would not decrease the criterion by more than the absolute tolerance, maxdev, set in the options structure. Adding a feature with no effect reduces the deviance by an amount that has a chi-square distribution with one degree of freedom. Adding a significant feature results in a larger change. By setting maxdev to chi2inv(.95,1), you instruct sequentialfs to continue adding features so long as the change in deviance is more than would be expected by random chance.

The reduced model (also with an initial constant term) is:

```
[b,dev,stats] = glmfit(X(:,inmodel),Y,'binomial');
```

```
% Display coefficient estimates and their standard errors:
model = [b stats.se]
model =
    0.0784    0.1642
    1.0040    0.1592
    1.9459    0.1789
    0.6134    0.1872
    0.8245    0.1730
```

# Feature Transformation

**In this section...**



## Introduction to Feature Transformation

*Feature transformation* is a group of methods that create new features (predictor variables). The methods are useful for dimension reduction when the transformed features have a descriptive power that is more easily ordered than the original features. In this case, less descriptive features can be dropped from consideration when building models.

Feature transformation methods are contrasted with the methods presented in "Feature Selection" on page 10-23, where dimension reduction is achieved by computing an optimal subset of predictive features measured in the original data.

The methods presented in this section share some common methodology. Their goals, however, are essentially different:

- Nonnegative matrix factorization is used when model terms must represent nonnegative quantities, such as physical quantities.
- Principal component analysis is used to summarize data in fewer dimensions, for example, to visualize it.
- Factor analysis is used to build explanatory models of data correlations.

## Nonnegative Matrix Factorization

## Introduction to Nonnegative Matrix Factorization

*Nonnegative matrix factorization* (*NMF*) is a dimension-reduction technique based on a low-rank approximation of the feature space. Besides providing a reduction in the number of features, NMF guarantees that the features are nonnegative, producing additive models that respect, for example, the nonnegativity of physical quantities.

Given a nonnegative $m$-by-$n$ matrix $X$ and a positive integer $k < \min(m,n)$, NMF finds nonnegative $m$-by-$k$ and $k$-by-$n$ matrices $W$ and $H$, respectively, that minimize the norm of the difference $X - WH$. $W$ and $H$ are thus approximate nonnegative factors of $X$.

The $k$ columns of $W$ represent transformations of the variables in $X$; the $k$ rows of $H$ represent the coefficients of the linear combinations of the original $n$ variables in $X$ that produce the transformed variables in $W$. Since $k$ is generally smaller than the rank of $X$, the product $WH$ provides a compressed approximation of the data in $X$. A range of possible values for $k$ is often suggested by the modeling context.

The Statistics Toolbox function `nnmf` carries out nonnegative matrix factorization. `nnmf` uses one of two iterative algorithms that begin with random initial values for $W$ and $H$. Because the norm of the residual $X - WH$ may have local minima, repeated calls to `nnmf` may yield different factorizations. Sometimes the algorithm converges to a solution of lower rank than $k$, which may indicate that the result is not optimal.

## Example: Nonnegative Matrix Factorization

For example, consider the five predictors of biochemical oxygen demand in the data set `moore.mat`:

```
load moore
X = moore(:,1:5);
```

The following uses `nnmf` to compute a rank-two approximation of `X` with a multiplicative update algorithm that begins from five random initial values for `W` and `H`:

```
opt = statset('MaxIter',10,'Display','final');
[W0,H0] = nnmf(X,2,'replicates',5,...
                   'options',opt,...
```

```
                    'algorithm','mult');
    rep iteration    rms resid    |delta x|
      1       10      358.296   0.00190554
      2       10      78.3556  0.000351747
      3       10      230.962    0.0172839
      4       10      326.347   0.00739552
      5       10      361.547   0.00705539
Final root mean square residual = 78.3556
```

The `'mult'` algorithm is sensitive to initial values, which makes it a good choice when using `'replicates'` to find `W` and `H` from multiple random starting values.

Now perform the factorization using an alternating least-squares algorithm, which converges faster and more consistently. Run 100 times more iterations, beginning from the initial `W0` and `H0` identified above:

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,2,'w0',W0,'h0',H0,...
                 'options',opt,...
                 'algorithm','als');
    rep iteration    rms resid    |delta x|
      1        3      77.5315 3.52673e-005
Final root mean square residual = 77.5315
```

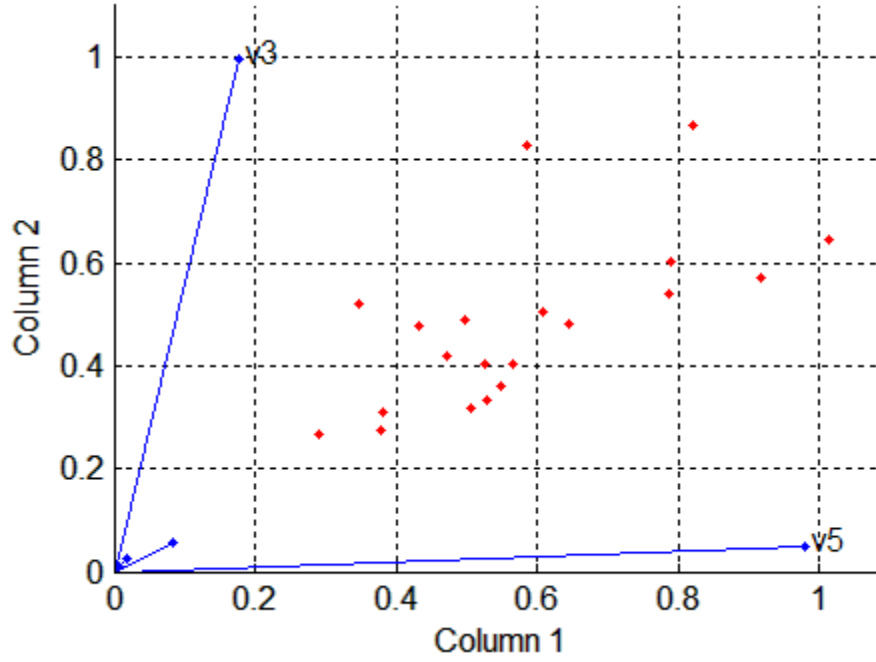
The two columns of `W` are the transformed predictors. The two rows of `H` give the relative contributions of each of the five predictors in `X` to the predictors in `W`:

```
H
H =
    0.0835  0.0190  0.1782  0.0072  0.9802
    0.0558  0.0250  0.9969  0.0085  0.0497
```

The fifth predictor in `X` (weight `0.9802`) strongly influences the first predictor in `W`. The third predictor in `X` (weight `0.9969`) strongly influences the second predictor in `W`.

Visualize the relative contributions of the predictors in `X` with a `biplot`, showing the data and original variables in the column space of `W`:

```
biplot(H','scores',W,'varlabels',{'','','v3','','v5'});
axis([0 1.1 0 1.1])
xlabel('Column 1')
ylabel('Column 2')
```




## Principal Component Analysis (PCA)



### Introduction to Principal Component Analysis (PCA)

One of the difficulties inherent in multivariate statistics is the problem of visualizing data that has many variables. The MATLAB function `plot` displays a graph of the relationship between two variables. The `plot3` and `surf` commands display different three-dimensional views. But when

there are more than three variables, it is more difficult to visualize their relationships.

Fortunately, in data sets with many variables, groups of variables often move together. One reason for this is that more than one variable might be measuring the same driving principle governing the behavior of the system. In many systems there are only a few such driving forces. But an abundance of instrumentation enables you to measure dozens of system variables. When this happens, you can take advantage of this redundancy of information. You can simplify the problem by replacing a group of variables with a single new variable.

Principal component analysis is a quantitatively rigorous method for achieving this simplification. The method generates a new set of variables, called *principal components*. Each principal component is a linear combination of the original variables. All the principal components are orthogonal to each other, so there is no redundant information. The principal components as a whole form an orthogonal basis for the space of the data.

There are an infinite number of ways to construct an orthogonal basis for several columns of data. What is so special about the principal component basis?

The first principal component is a single axis in space. When you project each observation on that axis, the resulting values form a new variable. And the variance of this variable is the maximum among all possible choices of the first axis.

The second principal component is another axis in space, perpendicular to the first. Projecting the observations on this axis generates another new variable. The variance of this variable is the maximum among all possible choices of this second axis.

The full set of principal components is as large as the original set of variables. But it is commonplace for the sum of the variances of the first few principal components to exceed 80% of the total variance of the original data. By examining plots of these few new variables, researchers often develop a deeper understanding of the driving forces that generated the original data.

You can use the function `princomp` to find the principal components. To use `princomp`, you need to have the actual measured data you want to analyze. However, if you lack the actual data, but have the sample covariance or correlation matrix for the data, you can still use the function `pcacov` to perform a principal components analysis. See the reference page for `pcacov` for a description of its inputs and outputs.

## Example: Principal Component Analysis



**Computing Components.** Consider a sample application that uses nine different indices of the quality of life in 329 U.S. cities. These are climate, housing, health, crime, transportation, education, arts, recreation, and economics. For each index, higher is better. For example, a higher index for crime means a lower crime rate.

Start by loading the data in `cities.mat`.

```
load cities
whos
  Name             Size         Bytes  Class
  categories       9x14           252  char array
  names          329x43         28294  char array
  ratings        329x9          23688  double array
```

The `whos` command generates a table of information about all the variables in the workspace.

The `cities` data set contains three variables:

- `categories`, a string matrix containing the names of the indices

- `names`, a string matrix containing the 329 city names
- `ratings`, the data matrix with 329 rows and 9columns

The `categories` variable has the following values:

```
categories
categories =
   climate
   housing
   health
   crime
   transportation
   education
   arts
   recreation
   economics
```

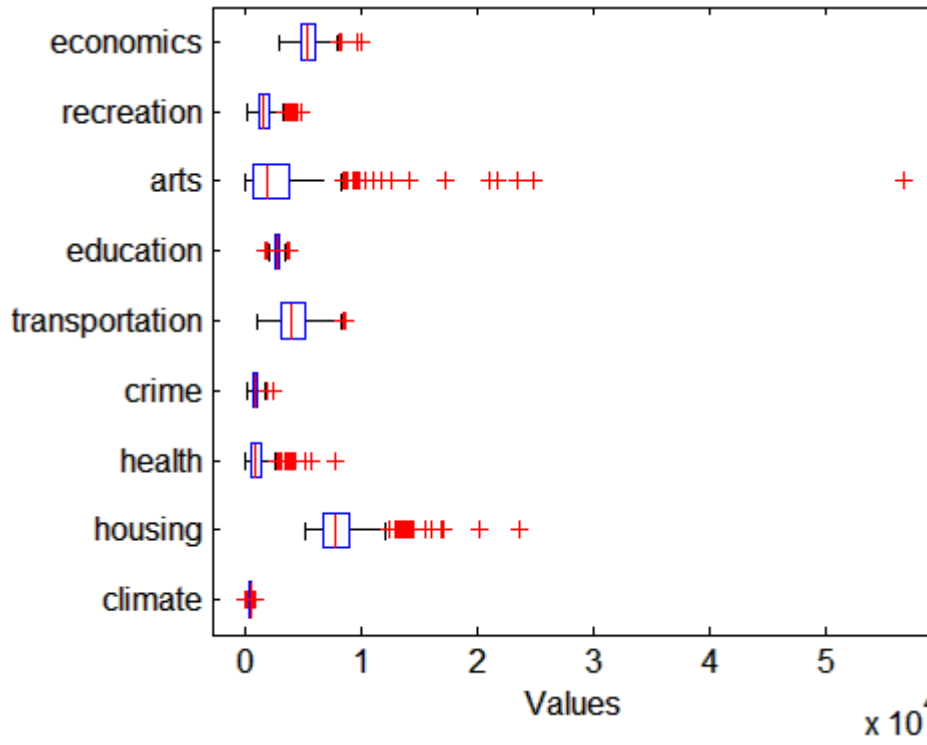
The first five rows of `names` are

```
first5 = names(1:5,:)
first5 =
   Abilene, TX
   Akron, OH
   Albany, GA
   Albany-Troy, NY
   Albuquerque, NM
```

To get a quick impression of the ratings data, make a box plot.

```
boxplot(ratings,'orientation','horizontal','labels',categories)
```

This command generates the plot below. Note that there is substantially more variability in the ratings of the arts and housing than in the ratings of crime and climate.




Ordinarily you might also graph pairs of the original variables, but there are 36two-variable plots. Perhaps principal components analysis can reduce the number of variables you need to consider.

Sometimes it makes sense to compute principal components for raw data. This is appropriate when all the variables are in the same units. Standardizing the data is often preferable when the variables are in different units or when the variance of the different columns is substantial (as in this case).

You can standardize the data by dividing each column by its standard deviation.

```
stdr = std(ratings);
sr = ratings./repmat(stdr,329,1);
```

Now you are ready to find the principal components.

```
[coefs,scores,variances,t2] = princomp(sr);
```

The following sections explain the four outputs from `princomp`.

**Component Coefficients.** The first output of the `princomp` function, `coefs`, contains the coefficients of the linear combinations of the original variables that generate the principal components. The coefficients are also known as *loadings*.

The first three principal component coefficient vectors are:

```
c3 = coefs(:,1:3)
c3 =
    0.2064    0.2178   -0.6900
    0.3565    0.2506   -0.2082
    0.4602   -0.2995   -0.0073
    0.2813    0.3553    0.1851
    0.3512   -0.1796    0.1464
    0.2753   -0.4834    0.2297
    0.4631   -0.1948   -0.0265
    0.3279    0.3845   -0.0509
    0.1354    0.4713    0.6073
```

The largest coefficients in the first column (first principal component) are the third and seventh elements, corresponding to the variables `health` and `arts`. All the coefficients of the first principal component have the same sign, making it a weighted average of all the original variables.

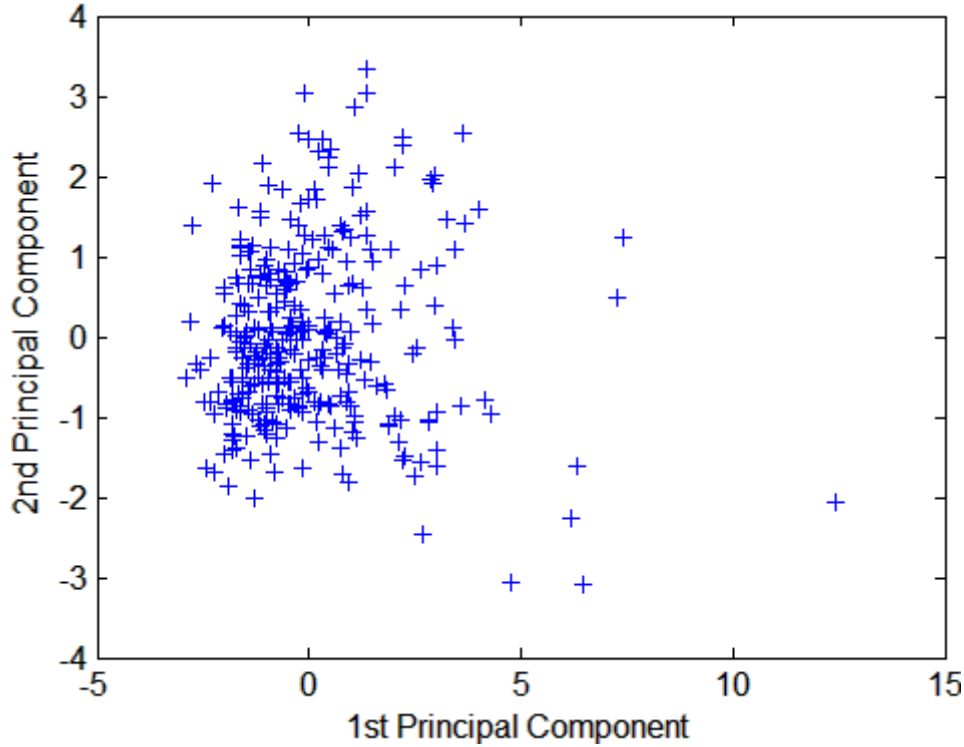
The principal components are unit length and orthogonal:

```
I = c3'*c3
I =
    1.0000   -0.0000   -0.0000
   -0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

**Component Scores.** The second output, `scores`, contains the coordinates of the original data in the new coordinate system defined by the principal components. This output is the same size as the input data matrix.

A plot of the first two columns of `scores` shows the ratings data projected onto the first two principal components. `princomp` computes the scores to have mean zero.

```
plot(scores(:,1),scores(:,2),'+')
xlabel('1st Principal Component')
ylabel('2nd Principal Component')
```




Note the outlying points in the right half of the plot.

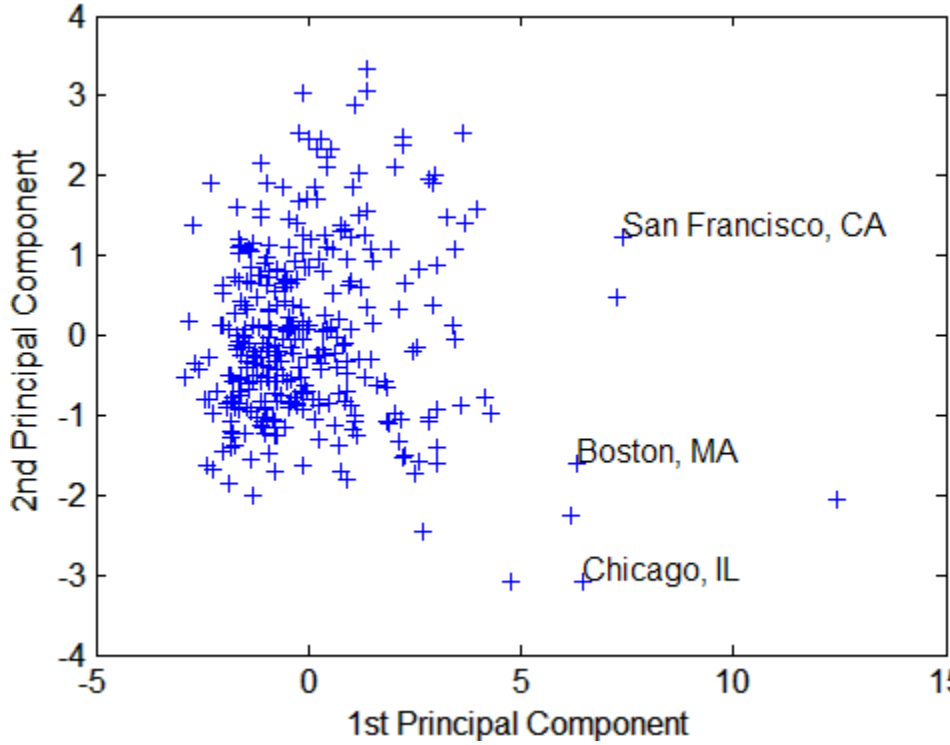
While it is possible to create a three-dimensional plot using three columns of `scores`, the examples in this section create two-dimensional plots, which are easier to describe.

The function `gname` is useful for graphically identifying a few points in a plot like this. You can call `gname` with a string matrix containing as many case

labels as points in the plot. The string matrix `names` works for labeling points with the city names.

```
gname(names)
```

Move your cursor over the plot and click once near each point in the right half. As you click each point, it is labeled with the proper row from the `names` string matrix. Here is the plot after a few clicks:




When you are finished labeling points, press the **Return** key.

The labeled cities are some of the biggest population centers in the United States. They are definitely different from the remainder of the data, so perhaps they should be considered separately. To remove the labeled cities from the data, first identify their corresponding row numbers as follows:

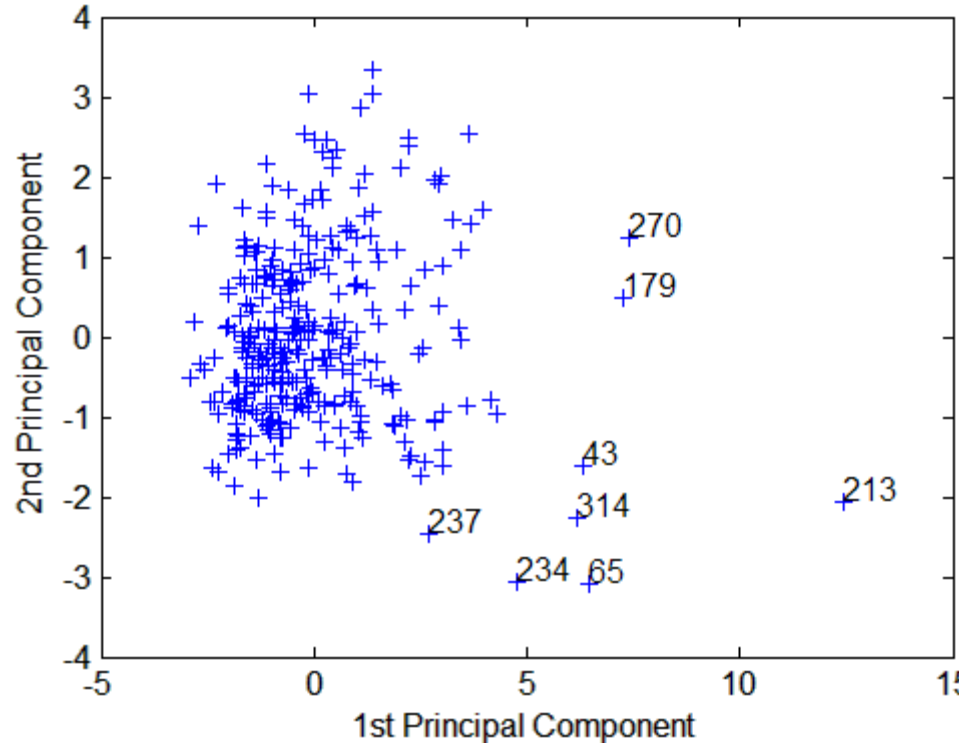
1. Close the plot window.

2. Redraw the plot by entering

```
plot(scores(:,1),scores(:,2),'+')
xlabel('1st Principal Component');
ylabel('2nd Principal Component');
```

3. Enter `gname` without any arguments.

4. Click near the points you labeled in the preceding figure. This labels the points by their row numbers, as shown in the following figure.




Then you can create an index variable containing the row numbers of all the metropolitan areas you choose.

```
metro = [43 65 179 213 234 270 314];
names(metro,:)
ans =
   Boston, MA
   Chicago, IL
   Los Angeles, Long Beach, CA
   New York, NY
   Philadelphia, PA-NJ
   San Francisco, CA
   Washington, DC-MD-VA
```

To remove these rows from the ratings matrix, enter the following.

```
rsubset = ratings;
nsubset = names;
nsubset(metro,:) = [];
rsubset(metro,:) = [];
size(rsubset)
ans =
   322     9
```

**Component Variances.** The third output, `variances`, is a vector containing the variance explained by the corresponding principal component. Each column of `scores` has a sample variance equal to the corresponding element of `variances`.

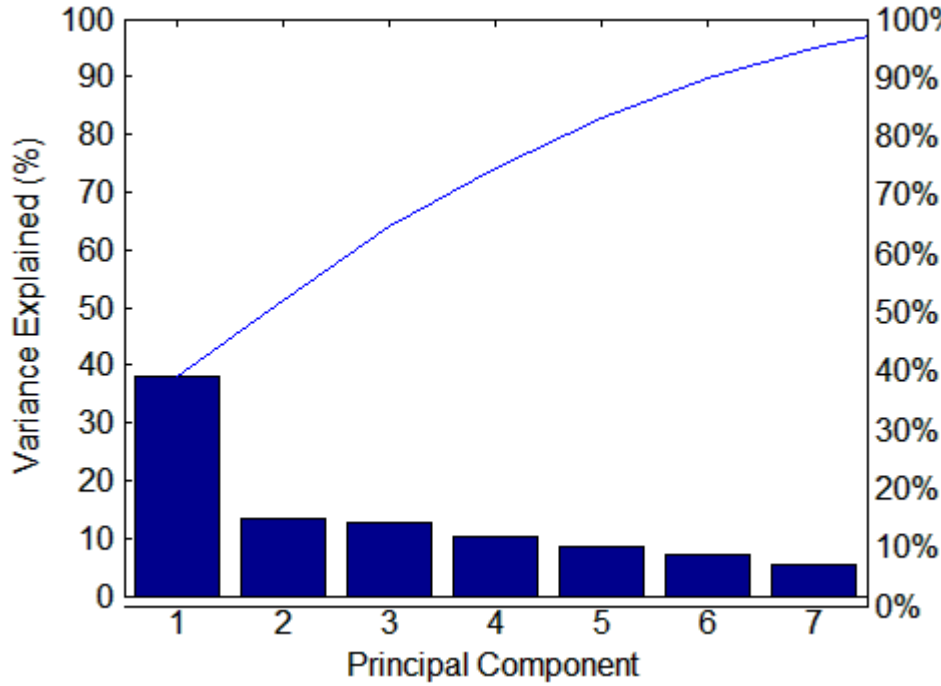
```
variances
variances =
    3.4083
    1.2140
    1.1415
    0.9209
    0.7533
    0.6306
    0.4930
    0.3180
    0.1204
```

You can easily calculate the percent of the total variability explained by each principal component.

```
percent_explained = 100*variances/sum(variances)
percent_explained =
   37.8699
   13.4886
   12.6831
   10.2324
    8.3698
    7.0062
    5.4783
    3.5338
    1.3378
```

Use the `pareto` function to make a *scree plot* of the percent variability explained by each principal component.

```
pareto(percent_explained)
xlabel('Principal Component')
ylabel('Variance Explained (%)')
```



The preceding figure shows that the only clear break in the amount of variance accounted for by each component is between the first and second components. However, that component by itself explains less than 40% of the variance, so more components are probably needed. You can see that the first three principal components explain roughly two-thirds of the total variability in the standardized ratings, so that might be a reasonable way to reduce the dimensions in order to visualize the data.

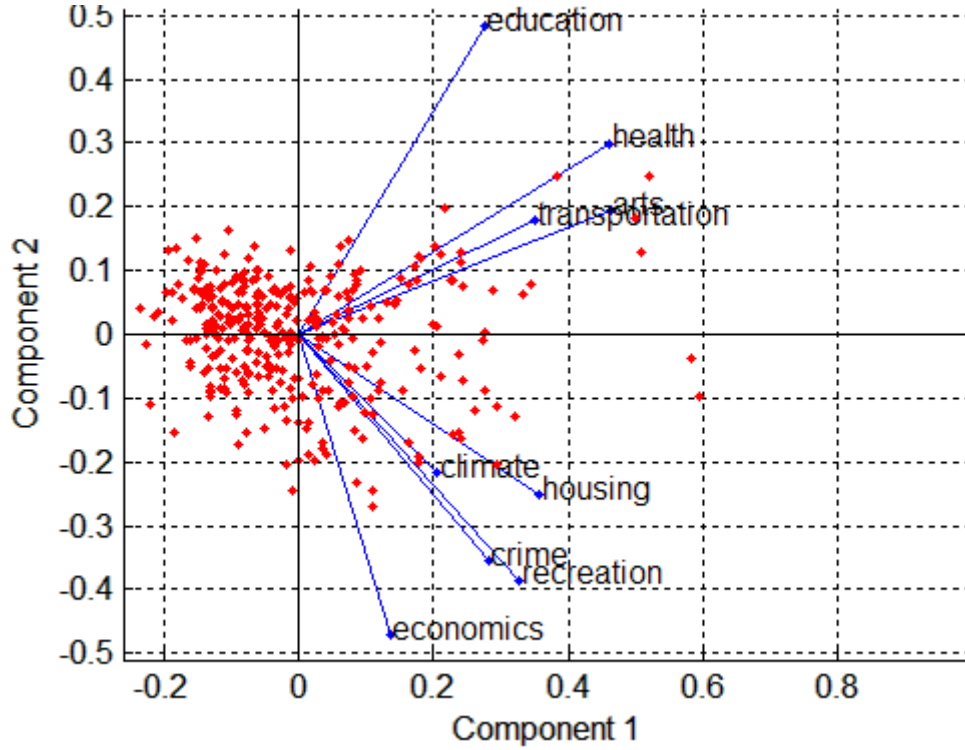
**Hotelling's T2.** The last output of the `princomp` function,`t2`, is Hotelling's$T^2$, a statistical measure of the multivariate distance of each observation from the center of the data set. This is an analytical way to find the most extreme points in the data.

```
[st2, index] = sort(t2,'descend'); % Sort in descending order.
extreme = index(1)
extreme =
   213
names(extreme,:)
ans =
   New York, NY
```

It is not surprising that the ratings for New York are the furthest from the average U.S. town.

**Visualizing the Results.** Use the `biplot` function to help visualize both the principal component coefficients for each variable and the principal component scores for each observation in a single plot. For example, the following command plots the results from the principal components analysis on the cities and labels each of the variables.

```
biplot(coefs(:,1:2), 'scores',scores(:,1:2),...
'varlabels',categories);
axis([-.26 1 -.51 .51]);
```




Each of the nine variables is represented in this plot by a vector, and the direction and length of the vector indicates how each variable contributes to the two principal components in the plot. For example, you have seen that the first principal component, represented in this biplot by the horizontal axis, has positive coefficients for all nine variables. That corresponds to the nine vectors directed into the right half of the plot. You have also seen that the second principal component, represented by the vertical axis, has positive coefficients for the variables education, health, arts, and transportation, and negative coefficients for the remaining five variables. That corresponds to vectors directed into the top and bottom halves of the plot, respectively. This indicates that this component distinguishes between cities that have high values for the first set of variables and low for the second, and cities that have the opposite.

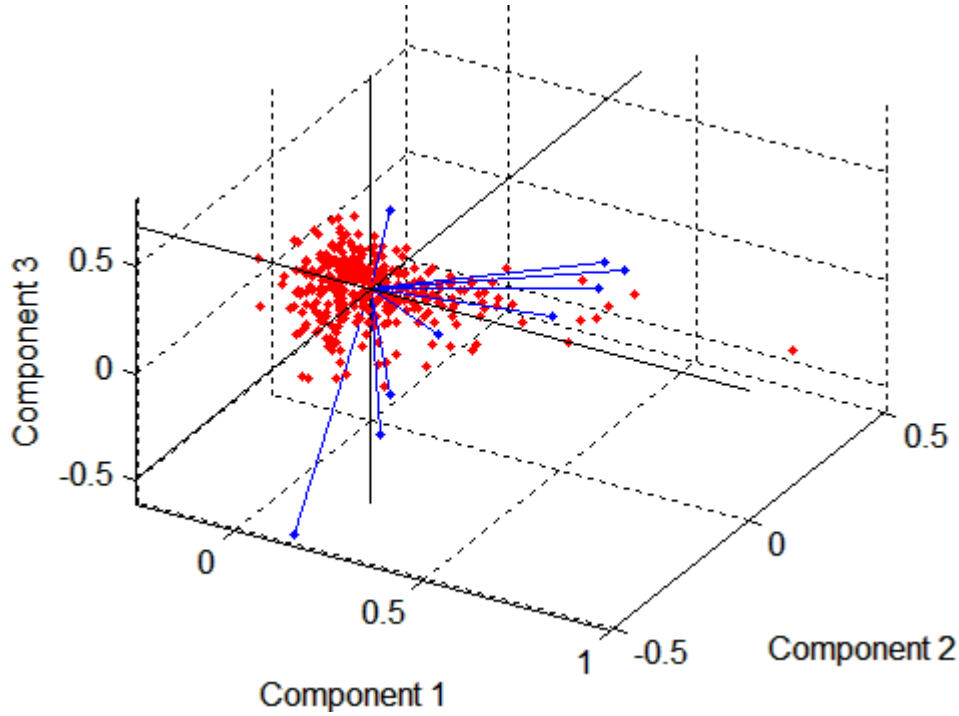
The variable labels in this figure are somewhat crowded. You could either leave out the `VarLabels` parameter when making the plot, or simply select and drag some of the labels to better positions using the Edit Plot tool from the figure window toolbar.

Each of the 329 observations is represented in this plot by a point, and their locations indicate the score of each observation for the two principal components in the plot. For example, points near the left edge of this plot have the lowest scores for the first principal component. The points are scaled to fit within the unit square, so only their relative locations may be determined from the plot.

You can use the `Data Cursor`, in the **Tools** menu in the figure window, to identify the items in this plot. By clicking on a variable (vector), you can read off that variable's coefficients for each principal component. By clicking on an observation (point), you can read off that observation's scores for each principal component.

You can also make a biplot in three dimensions. This can be useful if the first two principal coordinates do not explain enough of the variance in your data. Selecting `Rotate 3D` in the **Tools** menu enables you to rotate the figure to see it from different angles.

```
biplot(coefs(:,1:3), 'scores',scores(:,1:3),...
'obslabels',names);
axis([-.26 1 -.51 .51 -.61 .81]);
view([30 40]);
```




## Factor Analysis



### Introduction to Factor Analysis

Multivariate data often includes a large number of measured variables, and sometimes those variables overlap, in the sense that groups of them might be dependent. For example, in a decathlon, each athlete competes in 10 events, but several of them can be thought of as speed events, while others can be thought of as strength events, etc. Thus, you can think of a competitor's 10 event scores as largely dependent on a smaller set of three or four types of athletic ability.

Factor analysis is a way to fit a model to multivariate data to estimate just this sort of interdependence. In a factor analysis model, the measured variables depend on a smaller number of unobserved (latent) factors. Because each factor might affect several variables in common, they are known as *common factors*. Each variable is assumed to be dependent on a linear combination of the common factors, and the coefficients are known as *loadings*. Each measured variable also includes a component due to independent random variability, known as *specific variance* because it is specific to one variable.

Specifically, factor analysis assumes that the covariance matrix of your data is of the form

$$\sum_x = \Lambda\Lambda^T + \Psi$$

where $\Lambda$ is the matrix of loadings, and the elements of the diagonal matrix $\Psi$ are the specific variances. The function `factoran` fits the Factor Analysis model using maximum likelihood.

## Example: Factor Analysis



**Factor Loadings.** Over the course of 100 weeks, the percent change in stock prices for ten companies has been recorded. Of the ten companies, the first four can be classified as primarily technology, the next three as financial, and the last three as retail. It seems reasonable that the stock prices for companies that are in the same sector might vary together as economic conditions change. Factor Analysis can provide quantitative evidence that companies within each sector do experience similar week-to-week changes in stock price.

In this example, you first load the data, and then call `factoran`, specifying a model fit with three common factors. By default, `factoran` computes rotated estimates of the loadings to try and make their interpretation simpler. But in this example, you specify an unrotated solution.

```
load stockreturns

[Loadings,specificVar,T,stats] = ...
  factoran(stocks,3,'rotate','none');
```

The first two `factoran` return arguments are the estimated loadings and the estimated specific variances. Each row of the loadings matrix represents one of the ten stocks, and each column corresponds to a common factor. With unrotated estimates, interpretation of the factors in this fit is difficult because most of the stocks contain fairly large coefficients for two or more factors.

```
Loadings
Loadings =
       0.8885    0.2367   -0.2354
       0.7126    0.3862    0.0034
       0.3351    0.2784   -0.0211
       0.3088    0.1113   -0.1905
       0.6277   -0.6643    0.1478
       0.4726   -0.6383    0.0133
       0.1133   -0.5416    0.0322
       0.6403    0.1669    0.4960
       0.2363    0.5293    0.5770
       0.1105    0.1680    0.5524
```

---

**Note** "Factor Rotation" on page 10-48 helps to simplify the structure in the `Loadings` matrix, to make it easier to assign meaningful interpretations to the factors.

---

From the estimated specific variances, you can see that the model indicates that a particular stock price varies quite a lot beyond the variation due to the common factors.

```
specificVar
specificVar =
       0.0991
       0.3431
       0.8097
       0.8559
       0.1429
```

```
        0.3691
        0.6928
        0.3162
        0.3311
        0.6544
```

A specific variance of 1 would indicate that there is *no* common factor component in that variable, while a specific variance of 0 would indicate that the variable is *entirely* determined by common factors. These data seem to fall somewhere in between.

The *p* value returned in the `stats` structure fails to reject the null hypothesis of three common factors, suggesting that this model provides a satisfactory explanation of the covariation in these data.

```
stats.p
ans =
       0.8144
```

To determine whether fewer than three factors can provide an acceptable fit, you can try a model with two common factors. The *p* value for this second fit is highly significant, and rejects the hypothesis of two factors, indicating that the simpler model is not sufficient to explain the pattern in these data.

```
[Loadings2,specificVar2,T2,stats2] = ...
  factoran(stocks, 2,'rotate','none');

stats2.p
ans =
      3.5610e-006
```

**Factor Rotation.** As the results illustrate, the estimated loadings from an unrotated factor analysis fit can have a complicated structure. The goal of factor rotation is to find a parameterization in which each variable has only a small number of large loadings. That is, each variable is affected by a small number of factors, preferably only one. This can often make it easier to interpret what the factors represent.

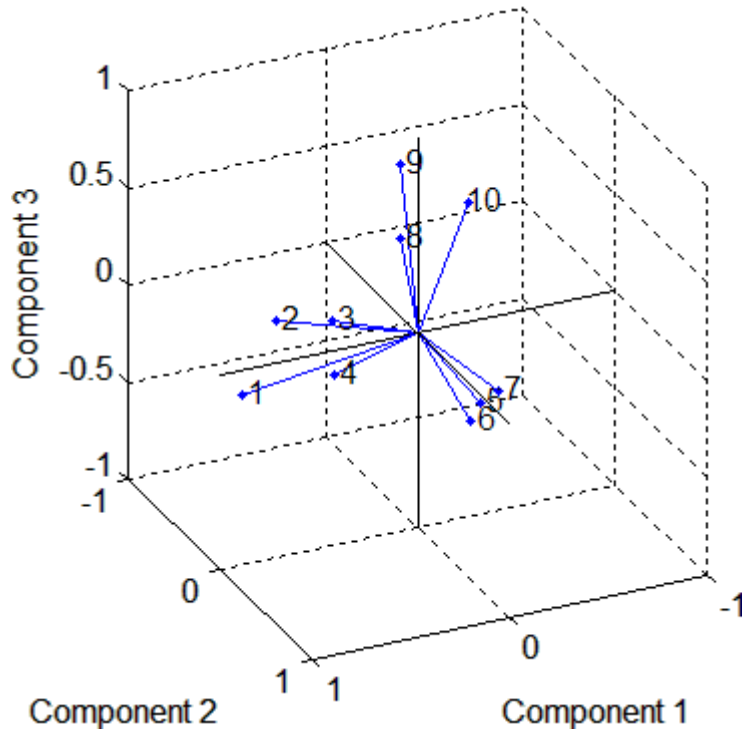
If you think of each row of the loadings matrix as coordinates of a point in *M*-dimensional space, then each factor corresponds to a coordinate axis. Factor rotation is equivalent to rotating those axes and computing new

loadings in the rotated coordinate system. There are various ways to do this. Some methods leave the axes orthogonal, while others are oblique methods that change the angles between them. For this example, you can rotate the estimated loadings by using the promax criterion, a common oblique method.

```
[LoadingsPM,specVarPM] = factoran(stocks,3,'rotate','promax');
LoadingsPM
LoadingsPM =
       0.9452    0.1214   -0.0617
       0.7064   -0.0178    0.2058
       0.3885   -0.0994    0.0975
       0.4162   -0.0148   -0.1298
       0.1021    0.9019    0.0768
       0.0873    0.7709   -0.0821
      -0.1616    0.5320   -0.0888
       0.2169    0.2844    0.6635
       0.0016   -0.1881    0.7849
      -0.2289    0.0636    0.6475
```

Promax rotation creates a simpler structure in the loadings, one in which most of the stocks have a large loading on only one factor. To see this structure more clearly, you can use the `biplot` function to plot each stock using its factor loadings as coordinates.

```
biplot(LoadingsPM,'varlabels',num2str((1:10)'));
axis square
view(155,27);
```




This plot shows that promax has rotated the factor loadings to a simpler structure. Each stock depends primarily on only one factor, and it is possible to describe each factor in terms of the stocks that it affects. Based on which companies are near which axes, you could reasonably conclude that the first factor axis represents the financial sector, the second retail, and the third technology. The original conjecture, that stocks vary primarily within sector, is apparently supported by the data.

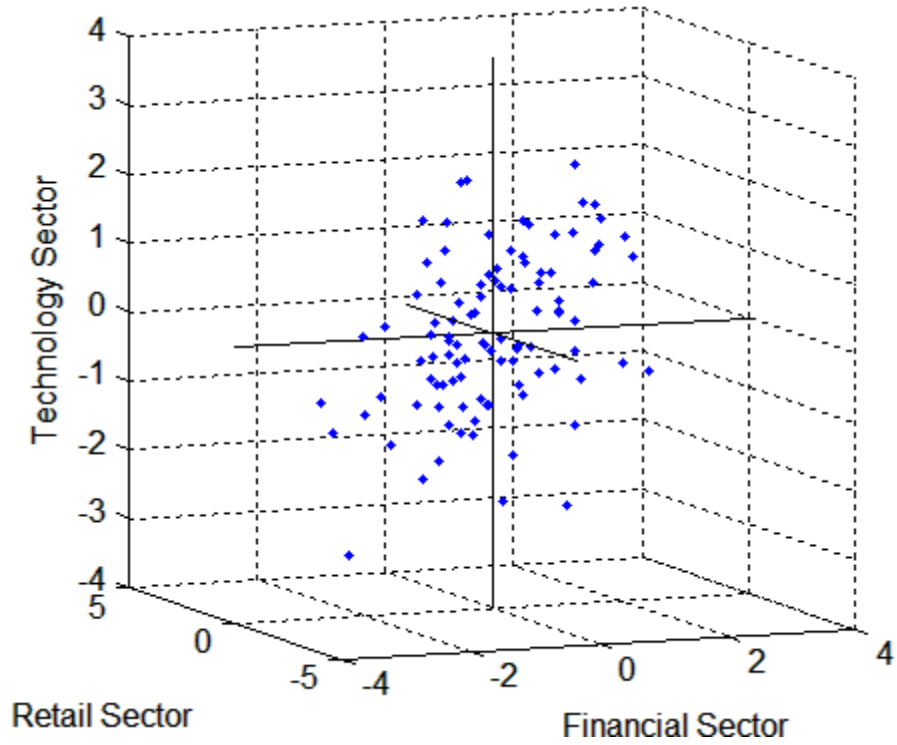
**Factor Scores.** Sometimes, it is useful to be able to classify an observation based on its factor scores. For example, if you accepted the three-factor model and the interpretation of the rotated factors, you might want to categorize each week in terms of how favorable it was for each of the three stock sectors, based on the data from the 10 observed stocks.

Because the data in this example are the raw stock price changes, and not just their correlation matrix, you can have `factoran` return estimates of the

value of each of the three rotated common factors for each week. You can then plot the estimated scores to see how the different stock sectors were affected during each week.

```
[LoadingsPM,specVarPM,TPM,stats,F] = ...
  factoran(stocks, 3,'rotate','promax');

plot3(F(:,1),F(:,2),F(:,3),'b.')
line([-4 4 NaN 0 0 NaN 0 0], [0 0 NaN -4 4 NaN 0 0],...
     [0 0 NaN 0 0 NaN -4 4], 'Color','black')
xlabel('Financial Sector')
ylabel('Retail Sector')
zlabel('Technology Sector')
grid on
axis square
view(-22.5, 8)
```

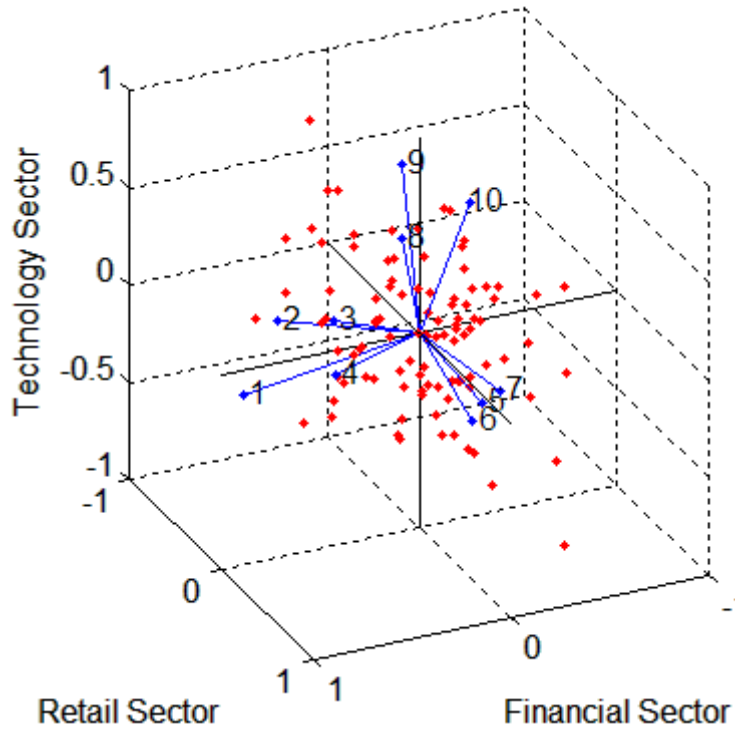


Oblique rotation often creates factors that are correlated. This plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix.

```
inv(TPM'*TPM)
ans =
       1.0000    0.1559    0.4082
       0.1559    1.0000   -0.0559
       0.4082   -0.0559    1.0000
```

**Visualizing the Results.** You can use the `biplot` function to help visualize both the factor loadings for each variable and the factor scores for each observation in a single plot. For example, the following command plots the results from the factor analysis on the stock data and labels each of the 10 stocks.

```
biplot(LoadingsPM,'scores',F,'varlabels',num2str((1:10)'))
xlabel('Financial Sector')
ylabel('Retail Sector')
zlabel('Technology Sector')
axis square
view(155,27)
```




In this case, the factor analysis includes three factors, and so the biplot is three-dimensional. Each of the 10 stocks is represented in this plot by a vector, and the direction and length of the vector indicates how each stock depends on the underlying factors. For example, you have seen that after promax rotation, the first four stocks have positive loadings on the first factor, and unimportant loadings on the other two factors. That first factor, interpreted as a financial sector effect, is represented in this biplot as one of the horizontal axes. The dependence of those four stocks on that factor corresponds to the four vectors directed approximately along that axis. Similarly, the dependence of stocks 5, 6, and 7 primarily on the second factor, interpreted as a retail sector effect, is represented by vectors directed approximately along that axis.

Each of the 100 observations is represented in this plot by a point, and their locations indicate the score of each observation for the three factors. For example, points near the top of this plot have the highest scores for the

technology sector factor. The points are scaled to fit within the unit square, so only their relative locations can be determined from the plot.

You can use the **Data Cursor** tool from the **Tools** menu in the figure window to identify the items in this plot. By clicking a stock (vector), you can read off that stock's loadings for each factor. By clicking an observation (point), you can read off that observation's scores for each factor.