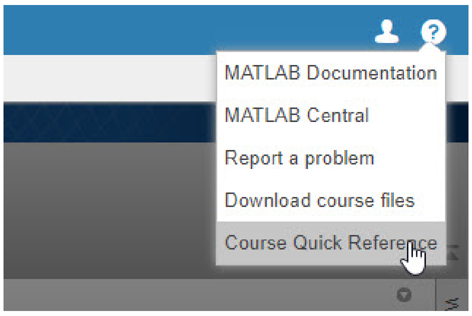
Getting Started

Review - Machine Learning Onramp

Summary: Machine Learning Onramp

These functions will be used throughout this course. Click any function to visit the documentation.

View this information in the course quick reference guide any time you need a reference.



Import and Process Data

The readtable function creates a table in MATLAB from a data file.

```
allStats = readtable("bballStats.txt");
playerInfo = readtable("bballPlayers.txt");
```

The categorical function creates a categorical array from data.

```
positions = ["G","G-F","F-G","F","F-C","C-F","C"];
playerInfo.pos = categorical(playerInfo.pos,positions);
```

Assigning the empty array removes rows or columns. The rmmissing function removes any row with missing or undefined elements.

```
allStats(19:end) = [];
playerInfo = rmmissing(playerInfo);
```

The groupsummary function calculates statistics grouped according to a grouping variable.

```
playerStats = groupsummary(allStats,"playerID","sum");
```

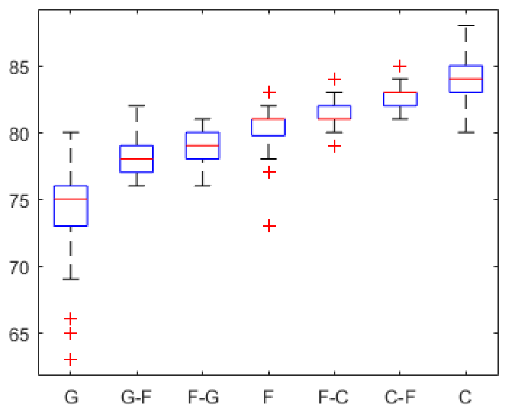
The innerjoin function merges two tables, retaining only the common key variable observations.

```
data = innerjoin(playerInfo,playerStats);
```

Visualize and Engineer Features

The boxplot function can create separate box plots based on a grouping variable.

```
boxplot(data.height,data.pos)
```

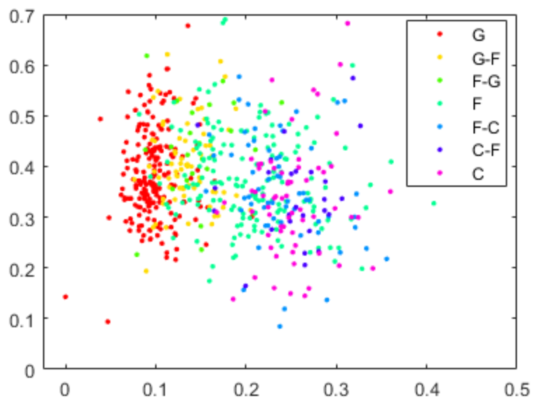


You can use indexing and element-wise division to scale variables in a table.

```
data{:,8:22} = data{:,8:22}./data.minutes;
data.minutes = data.minutes./data.GP;
```

You can use the gscatter function to create a grouped scatter plot.

```
gscatter(data.rebounds,data.points,data.pos)
```



Train and Evaluate a Model

The fitcknn function fits a k-nearest neighbors classification model.

```
knnmodel = fitcknn(data,"pos");
```

You can use property name-value pairs to modify model options.

```
knnmodel = fitcknn(data,"pos","NumNeighbors",5,...
      "Standardize",true);
```

You can calculate the misclassification rate for a data set using the loss function.

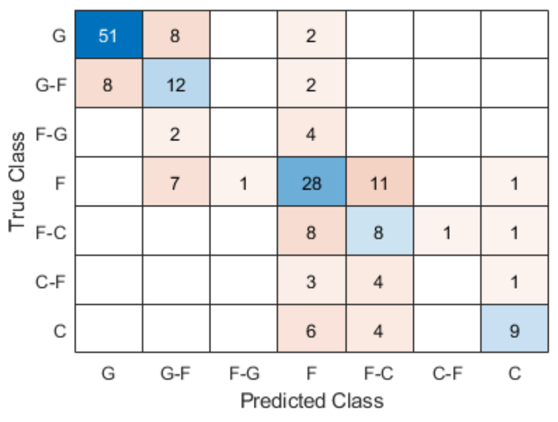
```
mdlLoss = loss(knnmodel,dataTest)
mdlLoss =
    0.4085
```

The predict function uses a classification model to predict classes for observations.

```
predPos = predict(knnmodel,dataTest);
```

You can use the confusionchart function to visually compare true classes and predicted classes.

```
confusionchart(data.pos,predPos);
```



Finding Natural Patterns in Data

Low Dimensional Visualization

Multidimensional Scaling

Step 1 - Calculate pairwise distances

You can use the function `pdist` to calculate the pairwise distance between the observations. Note that the input should be a numeric matrix.

```
>> D = pdist(data,"distance")
```

Outputs

`D`	A distance or dissimilarity vector containing the distance between each pair of observations. `D` is of length *m*(*m*-1)/2.

Inputs

`data`	An *m*-by-*n* numeric matrix containing the data. Each of the *m* rows is considered an observation.
`"distance"`	An optional input that indicates the method of calculating the distance or dissimilarity. Commonly used methods are `"euclidean"` (default), `"cityblock"`, and `"correlation"`.

Step 2 - Perform multidimensional scaling

You can now use the dissimilarity vector as an input to the function `cmdscale`.

```
>> [x,e] = cmdscale(D)
```

Outputs

`x`	*m*-by-*q* matrix of the reconstructed coordinates in *q*-dimensional space. *q* is the minimum number of dimensions needed to achieve the given pairwise distances.
`e`	Eigenvalues of the matrix `x*x'`.

Inputs

`D`	A distance or dissimilarity vector.

You can use the eigenvalues `e` to determine if a low-dimensional approximation to the points in `x` provides a reasonable representation of the data. If the first *p* eigenvalues are significantly larger than the rest, the points are well approximated by the first *p* dimensions (that is, the first *p* columns of `x`).

Principal Component Analysis

Another commonly used method for dimensionality reduction is principal component analysis (PCA). Use the function `pca` to perform principal component analysis.

```
>> [pcs,scrs,~,~,pexp] = pca(data)
```

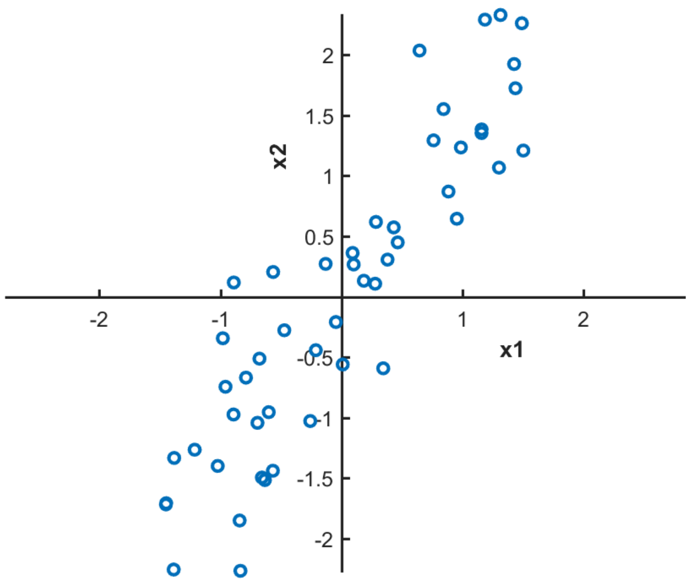
Outputs

`pcs`	A *n*-by-*n* matrix of principal components.
`scrs`	An *m*-by-*n* matrix containing the data transformed using the linear coordinate transformation matrix `pcs` (first output).
`pexp`	A vector of length *n* containing the percentage of variance explained by each principal component.

Inputs

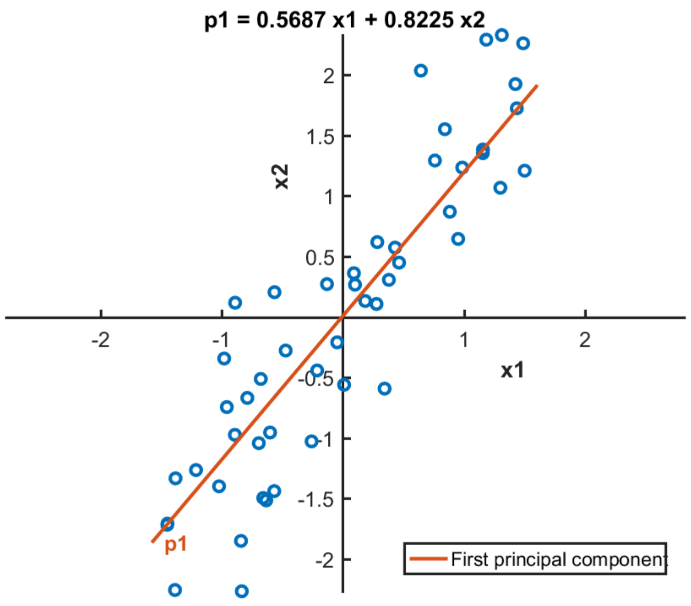
`data`	An *m*-by-*n* numeric matrix. The *n* columns correspond to *n* observed variables. Each of the *m* rows corresponds to an observation.

Interpreting the outputs



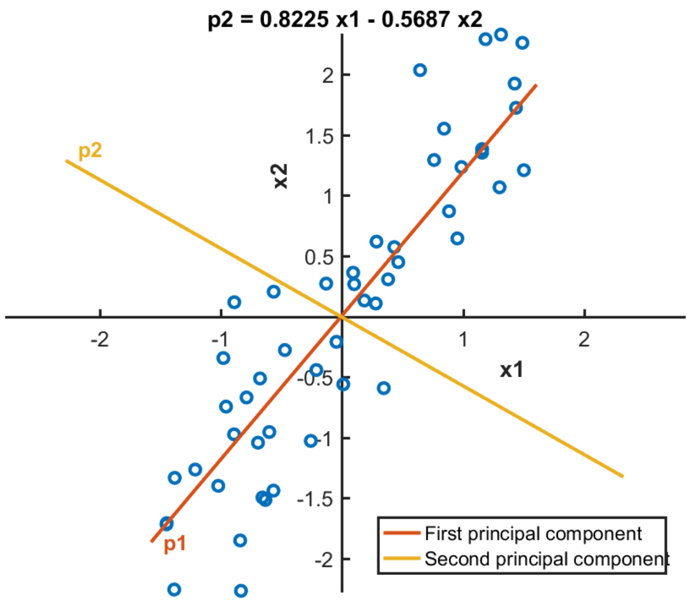
Suppose that the input matrix `data` has two columns which contain values of the observed variables x1 and x2.

```
[P,scrs,~,~,pexp] = pca(data)
P =
    0.5687    0.8225
    0.8225   -0.5687
```



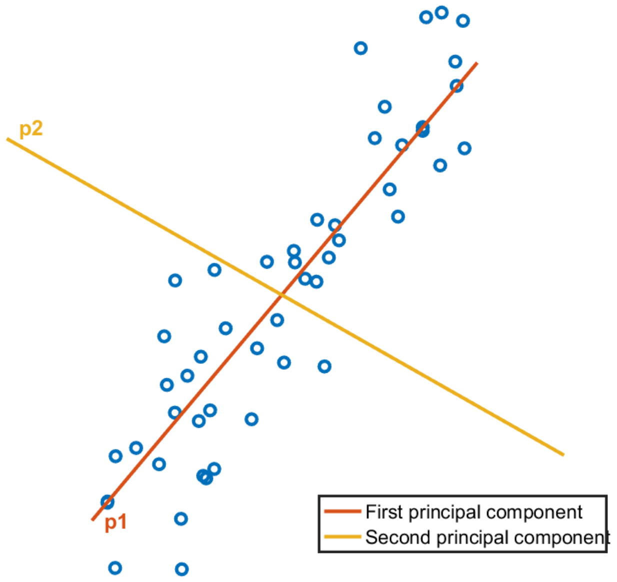
After performing PCA, the first column of the output matrix `P` contains the coefficients of the first principal component.

```
[P,scrs,~,~,pexp] = pca(data)
P =
    0.5687    0.8225
    0.8225   -0.5687
```



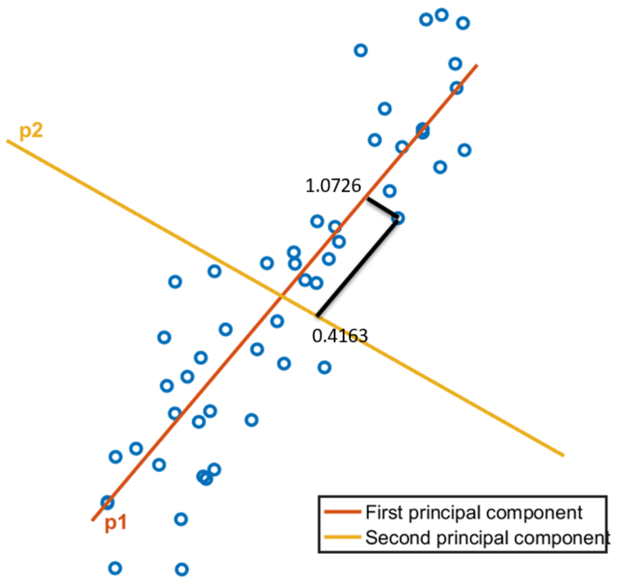
The second column of the output matrix `P` contains the coefficients of the second principal component.

```
[P,scrs,~,~,pexp] = pca(data);
```



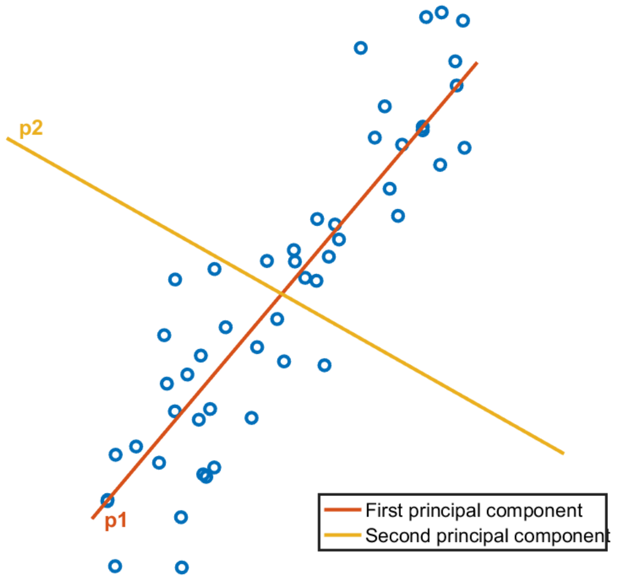
The second output `scrs` is a matrix containing the observations in `data` expressed in the coordinate space of the principal components.

```
[P,scrs,~,~,pexp] = pca(data);
scrs(42,:)
ans =
    1.0726    0.4163
```



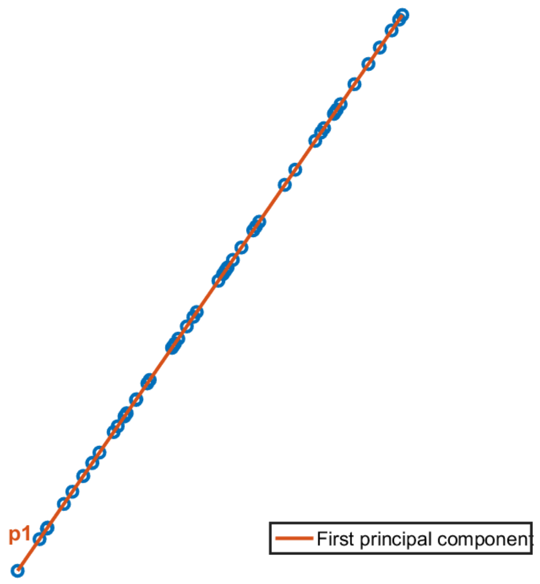
For example, a single data point and its coordinates in the transformed space are shown here.

```
[P,scrs,~,~,pexp] = pca(data);
pexp
ans =
    95.6706    4.3294
```



The last output `pexp` is a vector containing the percent variance explained by each principal component. Here, most of the variance is explained by the first principal component.

```
scrs(:,1)
```



We can use only the first column of the transformed data, reducing the dimension of the data from 2 to 1.

Gaussian Mixture Models

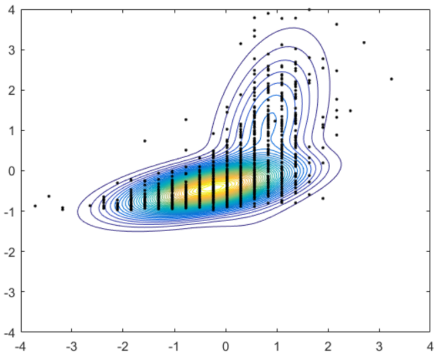
GMM Clustering

Step 1 - Fit Gaussian Mixture Model

You can use the function fitgmdist to fit several multidimensional gaussian (normal) distributions.

The command shown fits a mixture `gm` of two distributions.

```
gm = fitgmdist(X,2);
```



Step 2 - Identify Clusters

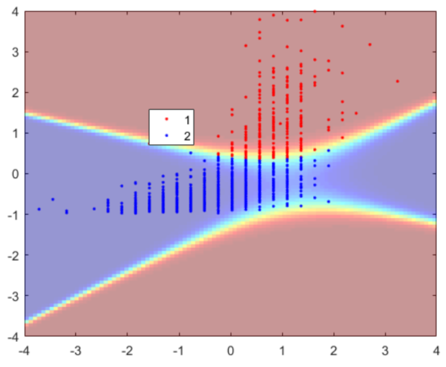
Now, the data can be clustered probabilistically, by calculating each observation's posterior probability for each component.

```
g = cluster(gm,X);
```

You can also return the individual probabilities used to determine the clusters.

The matrix p has two columns, one for each of the two clusters.

```
[g,~,p] = cluster(gm,X);
```



Interpreting the Clusters

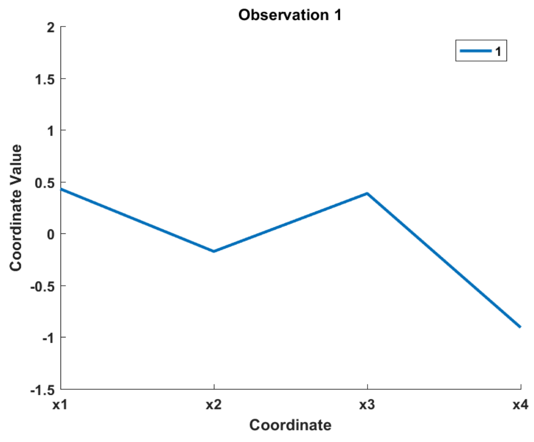
Visualizing Observations in Clusters

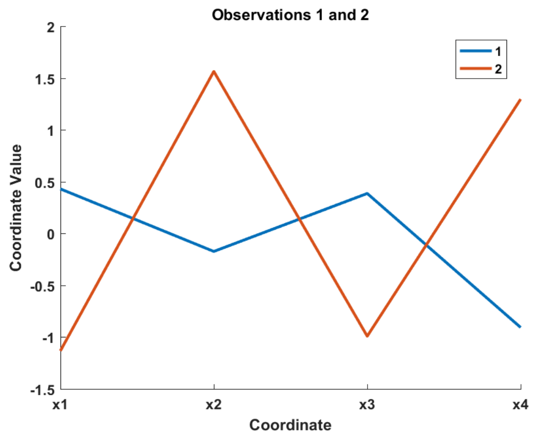
With high-dimensional data, it is difficult to visualize the groups as points in space. How can you interpret the groups given by a clustering method?

Parallel Coordinates

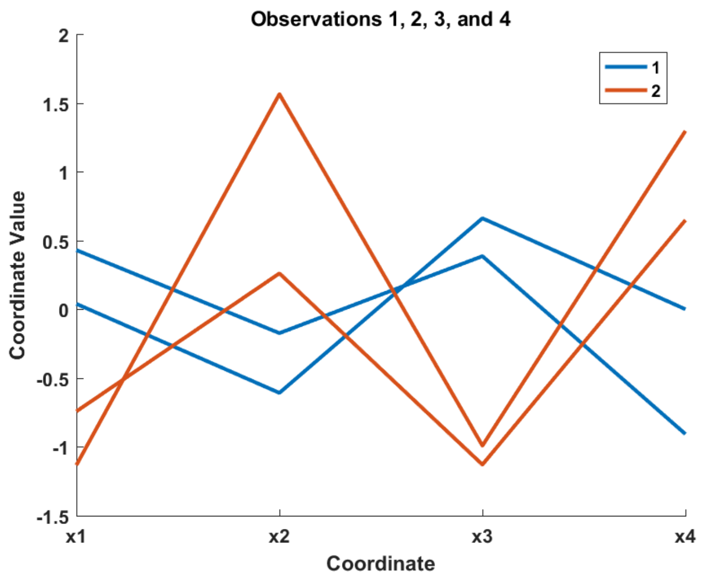
Consider a data set in which each observation has 4 variables (measurements) x1, x2, x3, and x4. Suppose that you have created two clusters.

You can visualize the first observation by plotting its variable value on the y-axis and the variable number on the x-axis. Similarly, you can visualize the second observation. If the second observation is in a different cluster, visualize it using a different color.





After visualizing several observations, you can see that cluster 1 has higher values of x1 and x3 whereas cluster 2 has higher values of x2 and x4.



Instead of visualizing each observation one-by-one, use the function `parallelcoords`, which creates the graph shown above.

```
>> parallelcoords(X,"Group",g)
```

Inputs

X	Data, specified as a numeric matrix.
`"Group"`	Property name.
g	A vector containing the observations' group or cluster identifiers.

Evaluating Cluster Quality

When using clustering techniques such as *k*-means and Gaussian mixture models, you have to specify the number of clusters. However, for high-dimensional data, it is difficult to determine the optimum number of clusters.

You can use the *silhouette* values to judge the quality of the clusters. An observation's silhouette value is a normalized measure (between -1 and +1) of how close that observation is to other observations in the same cluster, compared to the observations in different clusters.

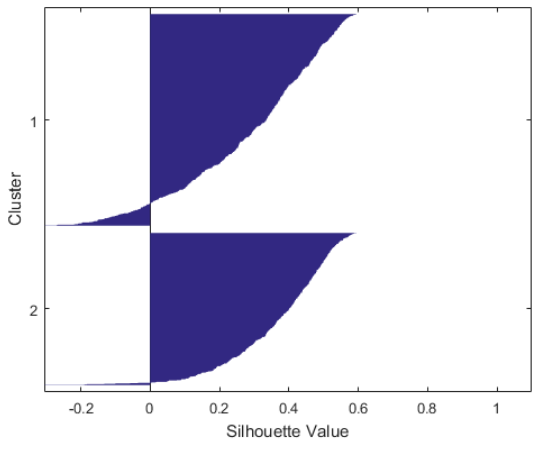
Silhouette Plots

A silhouette plot shows the silhouette value of each observation, grouped by cluster. Clustering schemes in which most of the observations have high silhouette value are desirable.

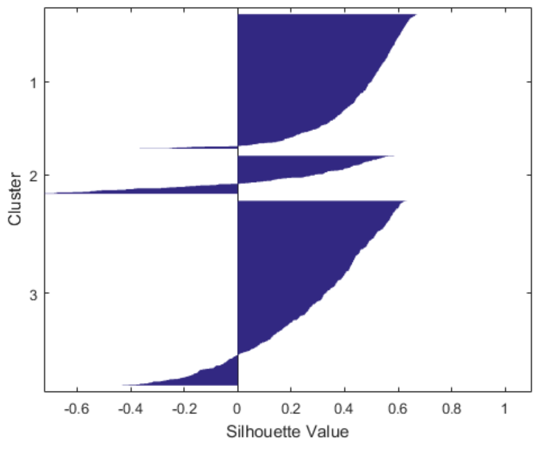
Use the `silhouette` function in MATLAB to create a silhouette plot.

Below are two silhouette plots for the same data set. On the left, the observations are divided into two clusters, and on the right they are divided into three clusters.

```
[grp,c] = kmeans(X,2);
silhouette(X,grp)
```



```
[grp,c] = kmeans(X,3);
silhouette(X,grp)
```



In this case, dividing the data into 2 clusters instead of 3 produces better quality clusters. The silhouette plot of two clusters shows fewer negative silhouette values, and those negative values are of smaller magnitude than the negative values in the silhouette plot of three clusters.

Automate Cluster Quality Evaluation

Instead of manually experimenting with different numbers of clusters, you can automate the process with the `evalclusters` function.

The following function call creates 2, 3, 4, and 5 clusters using *k*-means clustering, and calculates the silhouette value for each clustering scheme.

```
clustev = evalclusters(X,"kmeans","silhouette","KList",2:5)
```

The output variable, `clustev`, contains detailed information about the evaluation including the optimum number of clusters.

```
kbest = clustev.OptimalK
```

In place of `"silhouette"`, you can use other evaluation criteria such as `"CalinskiHarabasz"`, `"DaviesBouldin"`, and `"gap"`. Refer to the documentation for further details.

Hierarchical Clustering

Hierarchical Clustering

Step 1 – Determine Hierarchical Structure

Finding the hierarchical structure involves calculating the distance between each pair of points and then using these distances to link together pairs of “neighboring” points.

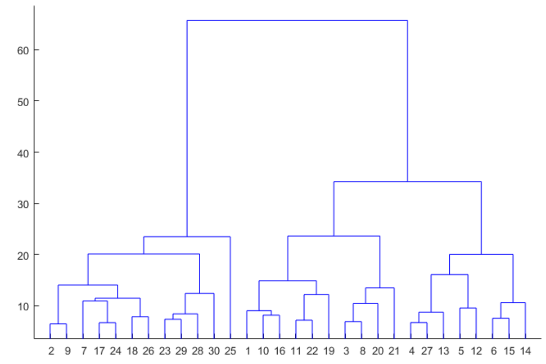
Use the `linkage` function to create the hierarchical tree.

The optional second and third inputs specify the methods for calculating the distance between clusters (default: `"single"`) and calculating the distance between points (default: `"euclidean"`).

```
Z = linkage(X,"ward","cosine");
```

You can use the `dendrogram` function to visualize the hierarchy.

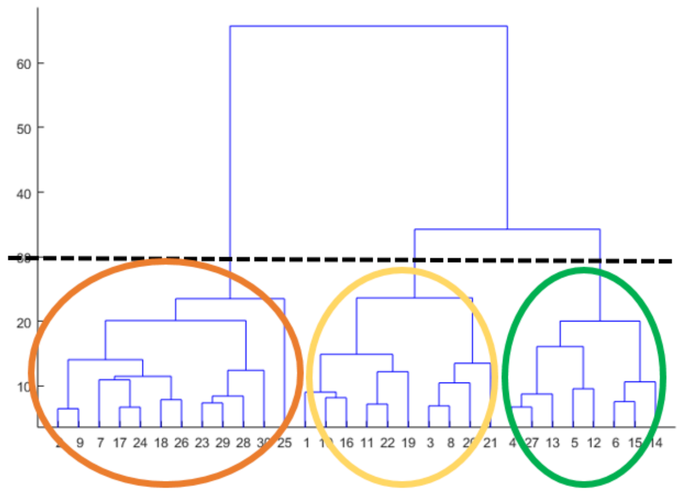
```
dendrogram(Z)
```



Step 2 – Divide Hierarchical Tree into Clusters

You can use the `cluster` function to assign observations into groups, according to the linkage distances `Z`.

```
Z = linkage(X,"centroid","cosine");
dendrogram(Z)
grp = cluster(Z,"maxclust",3)
```



Classification Methods

Nearest Neighbor Classification

k-Nearest Neighbor Overview

Function

`fitcknn`

Performance

Fit Time	Prediction Time	Memory Overhead
• Fast	• Fast • $\propto$ (Data Size)^2	• Small

Common Properties

`"NumNeighbors"` – Number of neighbors used for classification. (Default: 1)
`"Distance"` – Metric used for calculating distances between neighbors.
`"DistanceWeight"` – Weighting given to different neighbors.

Special Notes

For normalizing the data, use the `"Standardize"` option.

The `"cosine"` distance metric works well for "wide" data (more predictors than observations) and data with many predictors.

Classification Trees

Decision Trees Overview

Function

`fitctree`

Performance

Fit Time	Prediction Time	Memory Overhead
• $\propto$ Size of the data	• Fast	• Small

Common Properties

`"SplitCriterion"` – Formula used to determine optimal splits at each level.
`"MinLeafSize"` – Minimum number of observations in each leaf node.
`"MaxNumSplits"` – Maximum number of splits allowed in the decision tree.

Special Notes

Trees are a good choice when there is a significant amount of missing data.

Naive Bayes Classification

Naïve Bayes Overview

Function

`fitcnb`

Performance

Fit Time	Prediction Time	Memory Overhead
• Normal Dist. - Fast • Kernel Dist. - Slow	• Normal Dist. - Fast • Kernel Dist. - Slow	• Normal Dist. - Small • Kernel Dist. - Moderate to large

Common Properties	"DistributionNames" – Distribution used to calculate probabilities. "Width" – Width of the smoothing window (when "DistributionNames" is set to "kernel"). "Kernel" – Type of kernel to use (when "DistributionNames" is set to "kernel").
Special Notes	Naive Bayes is a good choice when there is a significant amount of missing data.

Discriminant Analysis

Fitting Discriminant Analysis Models

Linear Discriminant Analysis

By default, the covariance for each response class is assumed to be the same. This results in linear boundaries between classes.

```
daModel = fitcdiscr(dataTrain,"ResponseVarName")
```

Quadratic Discriminant Analysis

Removing the assumption of equal covariances results in a quadratic boundary between classes. Use the `"DiscrimType"` option to do this:

```
daModel = fitcdiscr(dataTrain,"ResponseVarName","DiscrimType","quadratic")
```

Discriminant Analysis (DA) Review

Function	fitcdiscr		
Performance	**Fit Time** • Fast • ∝ size of the data	**Prediction Time** • Fast • ∝ size of the data	**Memory Overhead** • Linear DA - Small • Quadratic DA - Moderate to large • ∝ number of predictors
Common Properties	"DiscrimType" - Type of boundary used. "Delta" - Coefficient threshold for including predictors in a linear boundary. (Default: 0) "Gamma" - Regularization to use when estimating the covariance matrix for linear DA.		
Special Notes	Linear discriminant analysis works well for “wide” data (more predictors than observations).		

Support Vector Machines

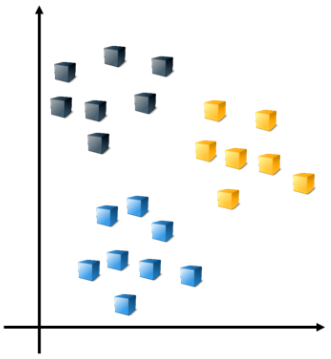
Support Vector Machines Review

Function	fitcsvm

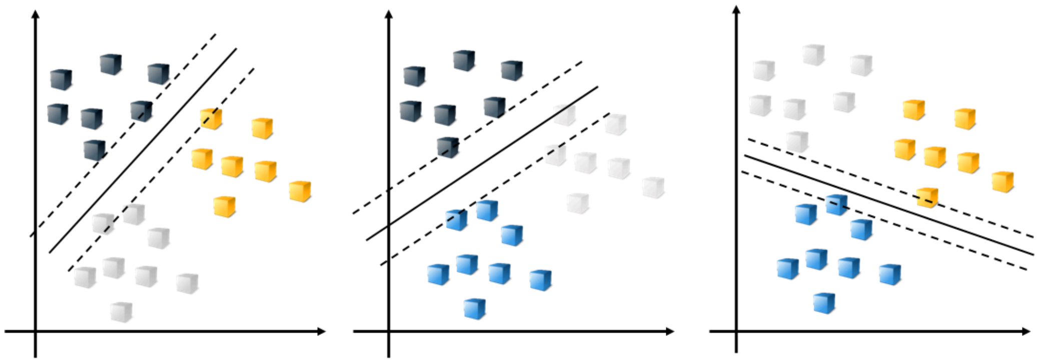
Performance	**Fit Time** • Fast • ∝ square of the size of the data	**Prediction Time** • Very Fast • ∝ square of the size of the data	**Memory Overhead** • Moderate
Common Properties	`"KernelFunction"` – Variable transformation to apply. `"KernelScale"` – Scaling applied before the kernel transformation. (Default value: 1) `"BoxConstraint"` – Regularization parameter controlling the misclassification penalty. (Default value: 1)		
Special Notes	SVMs use a distance based algorithm. For data that is not normalized, use the `"Standardize"` option. Linear SVMs work well for “wide” data (more predictors than observations). Gaussian SVMs often work better on “tall” data (more observations than predictors).		

Multiclass Support Vector Machine Models

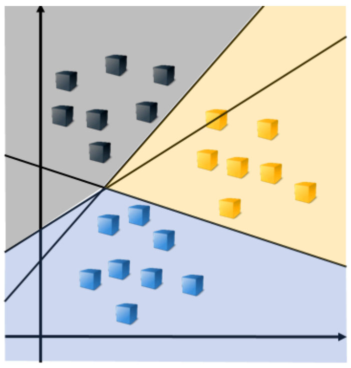
The underlying calculations for classification with SVMs are binary by nature. You can perform multiclass SVM classification by creating an error-correcting output codes (ECOC) classifier.



Suppose there are three classes in the data.



By default, the ECOC model reduces the model to multiple, binary classifiers using the one-vs-one design.



The combination of the resulting classifiers is used for prediction.

Creating Multiclass Support Vector Machines

Creating a multiclass SVM model is a two-step process.

1. Create a template for a binary classifier – Create a template for a binary SVM using the function `templateSVM`.

```
>> template = templateSVM("PropertyName",PropertyValue)
```

Outputs	
`template`	An SVM classifier.

Inputs	
`"PropertyName"`	Optional property name, e.g., `"KernelFunction"`.
`PropertyValue`	Property value, e.g., `"polynomial"`. You can provide multiple property name-value pairs.

2. Create multiclass SVM classifier – Use the function `fitcecoc` to create a multiclass SVM classifier.

```
>> ecocModel = fitcecoc(dataTrain,"y","Learners",template)
```

Outputs	
`ecocModel`	ECOC classifier.

Inputs	
`dataTrain`	Training data.
`"y"`	Response variable name.
`"Learners"`	Property name for specifying the binary classifier.
`template`	Binary classifier.

Classification with Neural Networks

Neural Networks

Function	`fitcnet`		
Performance	**Fit Time** • Slow • ∝ Network & data size	**Prediction Time** • Slow • ∝ Network & data size	**Memory Overhead** • Medium • ∝ Network & data size
Common Properties	`"LayerSizes"` – Sizes of fully connected layers. (Default: 10) `"Activations"` – Activation functions for fully connected layers. (Default: ReLU) `"ValidationData"` – Validation data for training convergence detection.		
Special Notes	Neural networks require data to be normalized. For normalizing the data, use the `"Standardize"` option. To train a neural network with more than one hidden layer specify the number of neurons per hidden layer as a vector. For example, `"LayerSizes",[20,15]` creates a network with two hidden layers with 20 and 15 neurons accordingly. Neural networks work well for "tall" data (more observations than predictors).		

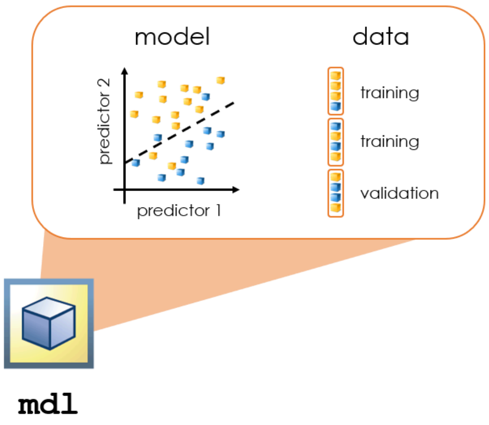
Improving Predictive Models

Cross Validation

Using Cross Validation

When you create a model, the model variable contains information about both the mathematical model and the data set.

The information about the partitions for cross validation will also be stored in the model variable.



Fitting Cross-Validated Models

To create a model with cross validation, provide one of the following options in the model-creation function.

```
        mdl = fitcknn(data,"Response","PropertyName",PropertyValue)
```

Property Name	Property Value	Type of validation
`"CrossVal"`	`"on"`	10-fold cross validation
`"Holdout"`	scalar from 0 to 1	Holdout with the given fraction reserved for validation.
`"KFold"`	*k* (scalar)	*k*-fold cross validation
`"Leaveout"`	`"on"`	Leave-one-out cross validation

If you already have a partition created using the `cvpartition` function, you can provide that to the fitting function instead.

```
  cvpt = cvpartition(data.Response,"KFold",k)
  mdl = fitcknn(data,"Response","CVPartition",cvpt)
```

Calculating the Loss

To evaluate a cross-validated model, use the `kfoldLoss` function.

```
  mdlLoss = kfoldLoss(mdl)
```

Reducing Predictors - Feature Transformation

Principal Component Analysis

Principal component analysis (PCA) transforms an *n*-dimensional feature space into a new *n*-dimensional space of orthogonal components. The components are ordered by the variation explained in the data.

```
>> [pcs,scrs,~,~,pctExp] = pca(X)
```

Outputs

`pcs`	Principal components coefficients. The contribution of each predictor to the *n* principal components. (*n*-by-*n* matrix)
`scrs`	Principal component scores. The representations of the *m* observations in `X` in the principal component space. (*m*-by-*n* matrix)
`pctExp`	Percent of the variance in the data explained by each principal component. (1-by-*n* vector)

Inputs

`X`	Data matrix with *m* observations and *n* predictors. (*m*-by-*n* matrix)

The transformed variables contain the same amount of information as the original data. However, assuming that the data contains some amount of noise, the components that contain the last few percent of explained variance are likely to represent noise more than information.

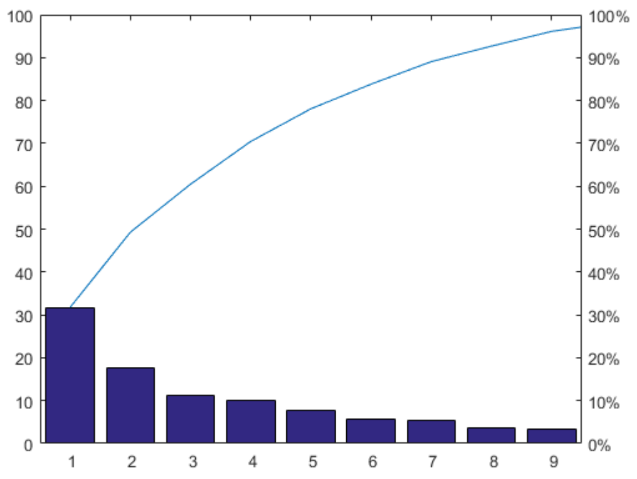
Reducing the Number of Predictors

PCA can be used for dimensionality reduction by discarding the principal components beyond a chosen threshold of explained variance. The `pareto` function can be used to visualize the variance explained by the principal components.

In the following example, the input `X` has 11 columns but the first 9 principal components explain more than 95% of variance.

```
[pcs,scrs,~,~,pctExp] = pca(X);
pareto(pctExp)

Xreduced = scrs(:,1:9);
```



Interpreting Principal Components

The principal components by themselves have no physical meaning. However, the coefficients of the linear transformation indicate the *contribution* of each variable in the principal component.

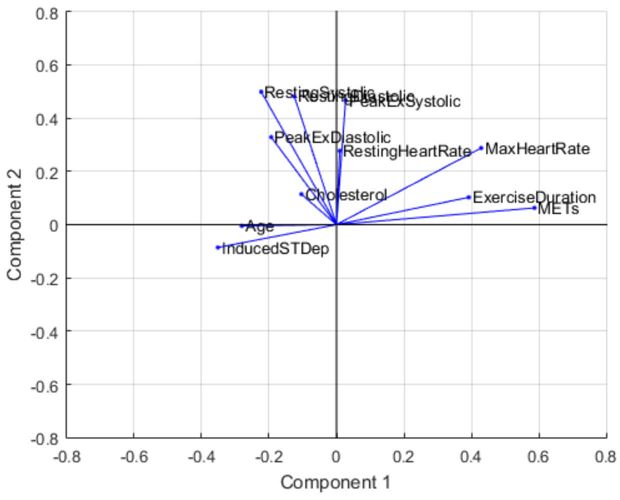
For example, if the coefficients of the first principal component are 0.8, 0.05, and 0.3, the first variable has the largest contribution followed by the third and the second variable.

Biplot

You can visualize any two principal components using the function `biplot`. It's commonly used to visualize the first two principal components, which explain the greatest amount of variance in the data.

In the following biplot, you can see that the predictive variables `Age`, `InducedSTDep`, `METS`, and `ExerciseDuration` contribute heavily to the first principal component, but not to the second principal component.

```
biplot(pcs(:,1:2),"VarLabels",varnames)
```

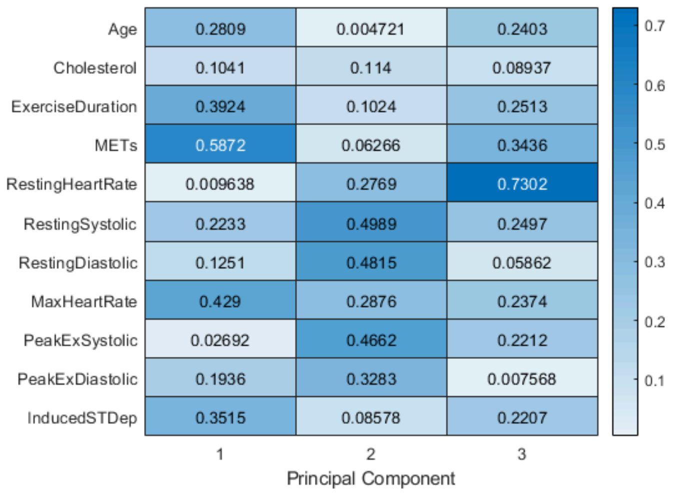


Heat Map

You can also visualize the contributions of each variable to the principal components as a heat map.

For example, the following heat map visualizes the first three principal components. You can see that `METS`, `MaxHeartRate`, and `ExerciseDuration` contribute heavily to the first principal component.

```
heatmap(abs(pcs(:,1:3)),...
    "YDisplayLabels",varnames);
xlabel("Principal Component")
```



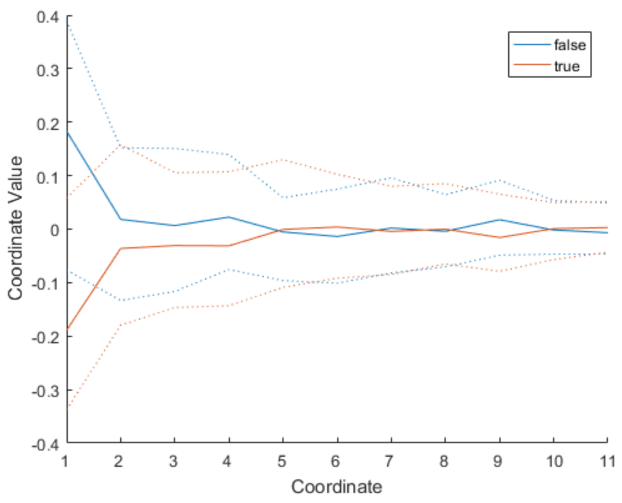
	1	2	3
Age	0.2809	0.004721	0.2403
Cholesterol	0.1041	0.114	0.08937
ExerciseDuration	0.3924	0.1024	0.2513
METs	0.5872	0.06266	0.3436
RestingHeartRate	0.009638	0.2769	0.7302
RestingSystolic	0.2233	0.4989	0.2497
RestingDiastolic	0.1251	0.4815	0.05862
MaxHeartRate	0.429	0.2876	0.2374
PeakExSystolic	0.02692	0.4662	0.2212
PeakExDiastolic	0.1936	0.3283	0.007568
InducedSTDep	0.3515	0.08578	0.2207

Parallel Coordinates Plot

PCA can be performed independent of the response variable. However, when the data has a response variable that has multiple categories (for example, true and false), a parallel coordinates plot of the principal component scores can be useful.

In the following plot, notice that the observations from one group (false) have high values of the first principal component and the observations from the second group (true) have low values.

```
parallelcoords(scrs,"Group",y,"Quantile",0.25)
```



Reducing Predictors - Feature Selection

Feature Ranking Algorithms

Feature ranking algorithms assign scores to features based on how relevant they are to the model according to a given metric. Some common algorithms include Chi-Square, Minimum Redundancy Maximum Relevance (MRMR), and Neighborhood Component Analysis.

Use the `fs__` functions to apply feature ranking algorithms to data. Most feature ranking algorithms return the positions and scores for all the features sorted from highest to lowest.

```
>> [idx,scores] = fs__(tblData,ResponseVarName)
```

Outputs

`idx`	Indices of predictors ordered by predictor importance.
`scores`	Predictor scores.

Inputs

`tblData`	Table data with predictor variables and a response variable.
`ResponseVarName`	Name of the response variable.

You can then use linear indexing to select a subset of features and use them to train the model.

```
toKeep  = idx(1:nFeatures);
selected = [tblData(:,toKeep),tblData(:,"ResponseVarName")];
mdl = fitcknn(selected,"ResponseVarName");
```

Feature ranking algorithms in MATLAB

Algorithm	Function	Task
Chi-Square	fscchi2	Classification
Minimum redundancy maximum relevance (MRMR)	fscmrmr / fsrmrmr	Classification/Regression
Neighborhood component analysis (NCA)	fscnca / fsrnca	Classification/Regression
F-tests	fsrftest	Regression
Laplacian	fsulaplacian	Clustering
ReliefF/RReliefF	relieff	Classification/Regression

Sequential Feature Selection

The function sequentialfs performs sequential feature selection.

```
>> toKeep = sequentialfs(fun,X,y)
```

Outputs

toKeep	Logical vector indicating which predictors are included in the final model.

Inputs

fun	Function handle for a function that fits a model and calculates the loss.
X	Numeric matrix with *m* observations and *n* predictors.
y	Numeric vector of *m* response values.

Note that the first input is a *handle* to the error function.

```
tokeep = sequentialfs(@errorFun,X,y)
```

You can use the optional property "cv" to specify the cross validation method. For example, you can specify a 7-fold cross validation.

```
tokeep = sequentialfs(@errorFun,X,y,"cv",7)
```

Structure of the Error Function

Sequential feature selection requires an error function that builds a model and calculates the prediction error. The error function must have the following structure.

- Four numeric inputs:
  - two for training the model (predictor matrix and response vector)
  - two for evaluating the model (predictor matrix and response vector)
- One scalar output representing the prediction error.

errorFun.m

```
function error = errorFun(Xtrain,ytrain,Xtest,ytest)

% Create the model with the learning method of your choice
mdl = fitcsvm(Xtrain,ytrain);

% Calculate the number of test observations misclassified
ypred = predict(mdl,XTest);
error = nnz(ypred ~= ytest);

end
```

Note You do not need to create the training and test data sets. The `sequentialfs` function will internally partition the data before calling the error function.

Accommodating Categorical Data

Some algorithms and functionality (e.g., `sequentialfs`) requires predictors in the form of a numeric matrix. If your data contains categorical predictors, how can you include these predictors in the model?

One option is to assign a number to each category. However, this may impose a false numerical structure on the observations.

For example, say you assign the numbers 1 through 4 to four categories in a predictor. This implies that the distance between categories 1 and 4 is three times larger than the distance between categories 3 and 4. In reality, the categories are probably equidistant from each other.

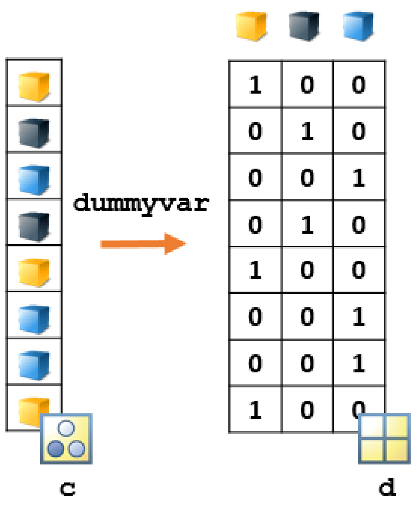
Creating Dummy Variables

A better approach is to create new *dummy predictors*, with one dummy predictor for each category.

You can create a matrix of dummy variables using the function `dummyvar`.

```
d = dummyvar(c)
```

Each column of `d` represents one of the categories in `c`. Each row corresponds to an observation, and has one element with value `1` and all other elements equal to `0`. The `1` appears in the column corresponding to that observation's assigned category.



This matrix can now be used in a machine learning model in place of the categorical vector `c`, with each column being treated as a separate predictor variable that indicates the presence (`1`) or absence (`0`) of that category of `c`.

Concatenate this matrix with a matrix containing the numeric predictors.

Hyperparameter Optimization

Performing Hyperparameter Optimization

You can use the `"OptimizeHyperparameters"` property name-value pair to choose which model properties to optimize. Most model-creation functions accept the `"OptimizeHyperparameters"` option.

```
>> mdl = fitcknn(data,"ResponseName","OptimizeHyperparameters",params)
```

Outputs

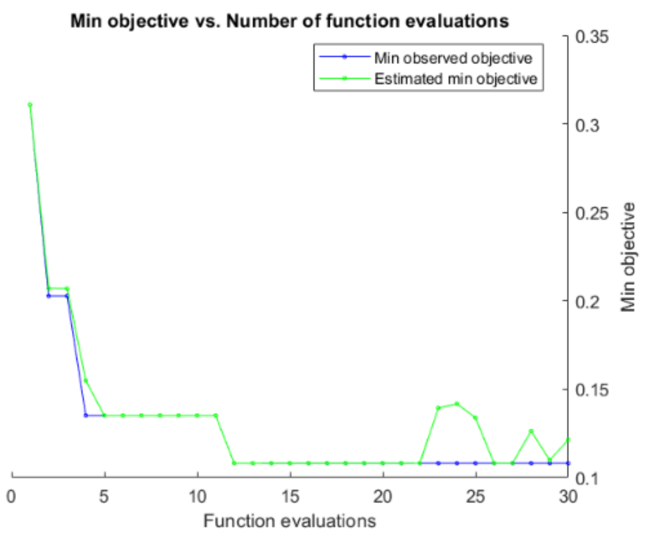
`mdl`	Model fit using optimized property values.

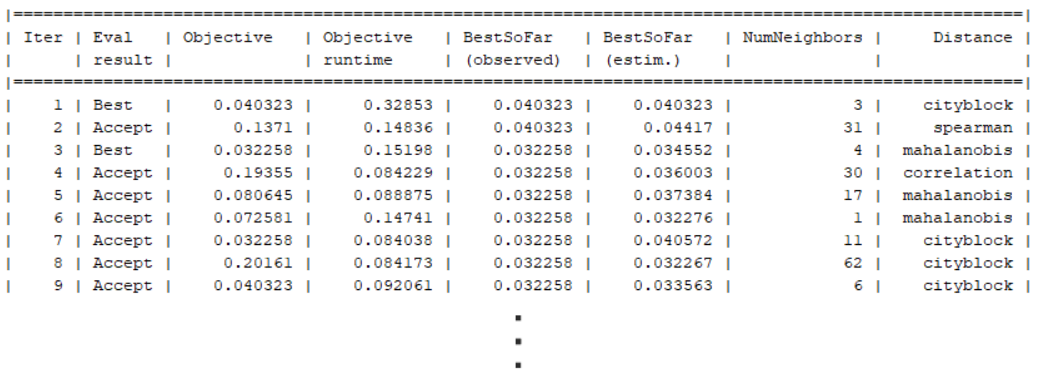
Inputs

`data,"ResponseName"`	Table of predictors and response values and response variable name.
`"OptimizeHyperparameters"`	Optional property for hyperparameter optimization.
`params`	Model properties to optimize, specified as a string array or cell array. Using `"auto"` and `"all"` optimizes pre-selected properties.

During the optimization, iterative updates are displayed, as well as a plot with the best objective function value against the iteration number.

```
mdl = fitcknn(data,"y","OptimizeHyperparameters","auto")
```





```
|=====================================================================================================|
| Iter | Eval   | Objective   | Objective   | BestSoFar   | BestSoFar   | NumNeighbors |     Distance |
|      | result |             | runtime     | (observed)  | (estim.)    |              |              |
|=====================================================================================================|
|    1 | Best   |    0.040323 |     0.32853 |    0.040323 |    0.040323 |            3 |    cityblock |
|    2 | Accept |      0.1371 |     0.14836 |    0.040323 |     0.04417 |           31 |     spearman |
|    3 | Best   |    0.032258 |     0.15198 |    0.032258 |    0.034552 |            4 |  mahalanobis |
|    4 | Accept |     0.19355 |    0.084229 |    0.032258 |    0.036003 |           30 |  correlation |
|    5 | Accept |    0.080645 |    0.088875 |    0.032258 |    0.037384 |           17 |  mahalanobis |
|    6 | Accept |    0.072581 |     0.14741 |    0.032258 |    0.032276 |            1 |  mahalanobis |
|    7 | Accept |    0.032258 |    0.084038 |    0.032258 |    0.040572 |           11 |    cityblock |
|    8 | Accept |     0.20161 |    0.084173 |    0.032258 |    0.032267 |           62 |    cityblock |
|    9 | Accept |    0.040323 |    0.092061 |    0.032258 |    0.033563 |            6 |    cityblock |
                                                   .
                                                   .
                                                   .
```

Setting the `"OptimizeHyperparameters"` property value to `"auto"` will optimize a typical set of hyperparameters. The properties optimized differ depending on the model type. For example, for Nearest Neighbor classification, the optimized properties are `"Distance"` and `"NumNeighbors"`.

Optimization Options

By default, hyperparameter optimization uses Bayesian optimization and tries to minimize the 5-fold cross-validation loss. You can change these settings with the `"HyperparameterOptimizationOptions"` property name-value pair.

Specify the optimization options using a structure. To use 10-fold cross validation, create a cross-validation partition and then create a structure containing option name-value pairs.

```
part = cvpartition(y,"KFold",10);
opt = struct("CVPartition",part);
mdl = fitcknn(data,"y","OptimizeHyperparameters","auto","HyperparameterOptimizationOptions",opt);
```

You can set many optimization options in the structure. For example, you can hide the plots and set the maximum number of objective function evaluations.

```
opt = struct("ShowPlots",false,"MaxObjectiveEvaluations",50);
```

To see the available options for a particular model-creation function, view the function's documentation.

Ensemble Learning

Fitting Ensemble Models

The `fitcensemble` function creates a classification ensemble of weak learners. Similarly, the `fitrensemble` function creates a regression ensemble. Both functions have identical syntax.

```
>> mdl = fitcensemble(data,"ResponseName")
```

Outputs

`mdl`	Ensemble model variable.

Inputs

`data`	Table containing the predictors and response values.
`"ResponseName"`	Response variable name.

Commonly Used Options

- "Method" - Bagging (**b**ootstrap **agg**regation) and boosting are the two most common approaches used in ensemble modeling. The `fitcensemble` function provides several bagging and boosting methods. For example, use the "Bag" method to create a random forest.

  ```
  mdl - fitcensemble(data,"Y","Method","Bag")
  ```

  The default method depends on if it is a binary or multiclass classification problem, as well as the type of learners in the ensemble.

- "Learners" - You can specify the type of weak-learner to use in the ensemble: "tree", "discriminant", or "knn". The default learner type depends on the method specified: the method "Subspace" has default learner "knn", and all other methods have default learner "tree".

  ```
  mdl = fitcensemble(data,"Y","Learners","knn")
  ```

  The `fitcensemble` function uses the default settings for each learner type. To customize learner properties, use a weak-learner template.

  ```
  mdl = fitcensemble(data,"Y","Learners",templateKNN("NumNeighbors",3))
  ```

  You can use a cell vector of learners to create an ensemble composed of more than one type of learner. For example, an ensemble could consist of two types of kNN learners.

  ```
  lnrs = {templateKNN("NumNeighbors",3),templateKNN("NumNeighbors",5)}
  mdl = fitcensemble(data,"Y","Learners",lnrs)
  ```

- "NumLearningCycles" - At every learning cycle, one weak learner is trained for each learner specified in "Learners". The default number of learning cycles is 100. If "Learners" contains only one learner (as is usually the case), then by default 100 learners are trained. If "Learners" contains two learners, then by default 200 learners are trained (two learners per learning cycle).

Regression Methods

Linear Models

Fitting Linear Regression Models

Use the function `fitlm` to fit a linear regression model.

```
>> mdl = fitlm(data,"modelspec")
```

Outputs

`mdl`	A regression model variable containing the coefficients and other information about the model.

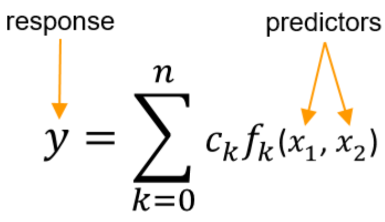
Inputs

`data`	A table containing the data used to fit the regression model. See below for details.
"modelspec"	Specification of the regression model. See below for details.

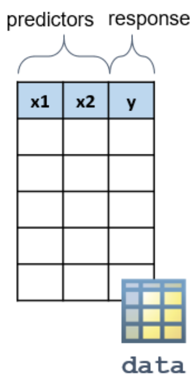
How to Organize the Data

In a regression model, the relationship between the predictors and the response can be described by the

The first input to `fitlm` is a table containing the predictors and the response. By default, `fitlm` uses the last column as following formula.

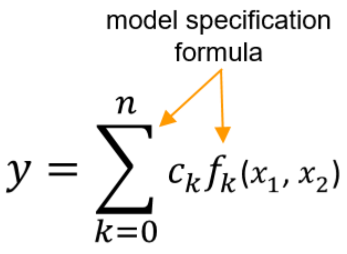


the response and all other columns as predictors.



How to Specify the Regression Model

When modeling a linear regression, you can apply different functions to the predictive variables. As the second input to `fitlm`, you can use one of the predefined models or you can specify a model by providing a formula in *Wilkinson–Rogers* notation.


model specification formula
$y = \sum_{k=0}^{n} c_k f_k(x_1, x_2)$

Common predefined models:

Model name	Meaning
`"linear"`	Intercept and linear terms for each predictor.
`"interactions"`	Intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
`"quadratic"`	Intercept, linear terms, interactions, and squared terms.

Wilkinson–Rogers notation: `responseVar ~ terms`

Operator	Meaning	Example
+	Include this term.	`"y ~ x1+x2"` includes the intercept term, x1, and x2: $y = c_0 + c_1 x_1 + c_2 x_2$
-	Exclude this term.	`"y ~ x1+x2-1"` excludes the intercept term: $y = c_1 x_1 + c_2 x_2$
*****	Include product and all lower-order terms.	`"y ~ x1*x2"` includes the intercept term, x1, x2, and x1*x2: $y = c_0 + c_1 x_1 + c_2 x_2 + c_3 x_1 x_2$
:	Includes the product term only.	`"y ~ x1:x2"` includes only x1*x2: $y = c_1 x_1 x_2$
^	Include power and all lower-	`"y ~ (x1^2)+(x2^2)"` includes the intercept, x1, x2, x1^2, and x2^2: $y = c_0 + c_1 x_1 + c_2 x_2 + c_3 x_1^2 + c_4 x_2^2$

Operator	Meaning	Example
	order terms.	

Fitting Models Using Data Stored in Arrays

If your data is stored in numeric arrays instead of a single table, you can provide the `fitlm` function the predictor and response values as numeric arrays.

```
>> mdl = fitlm(X,y)
```

Outputs

`mdl`	Linear regression model variable.

Inputs

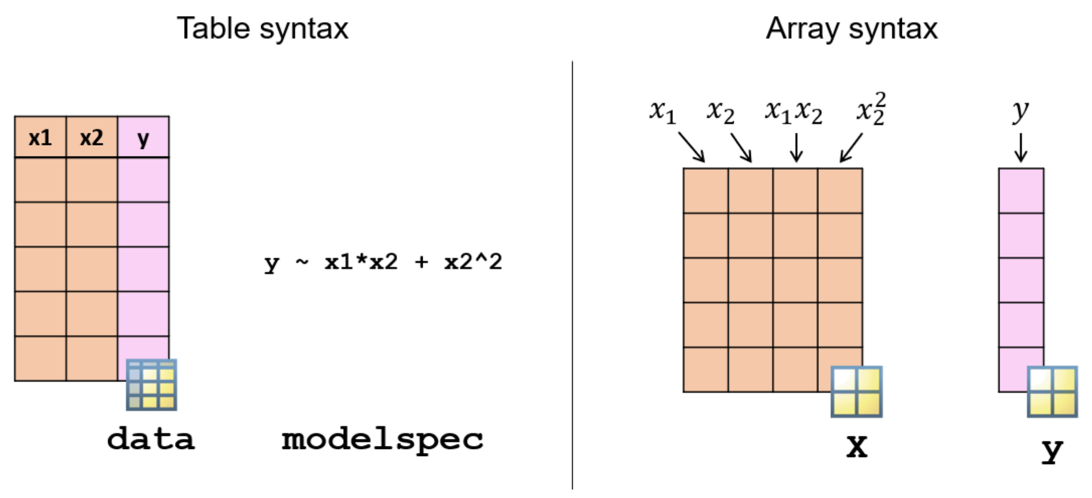
X	Predictor values, specified as a numeric matrix.
y	Response values, specified as a vector.

How to Specify the Regression Model

Each column of the predictor matrix `X` is treated as one predictor variable. By default, `fitlm` will fit a model with an intercept and a linear term for each predictor (column).

To fit a different regression formula, you have two options. You can store the predictors and the response in a table and provide the model specification separately. Alternatively, you can create a matrix with a column for each term in the regression formula. This matrix is called a *design matrix*.

$$y = c_0 + c_1x_1 + c_2x_2 + c_3x_1x_2 + c_4x_2^2$$



Stepwise Fitting

Fitting Stepwise Linear Regression

Use the function `stepwiselm` to fit a stepwise linear regression model.

```
>> stepwiseMdl = stepwiselm(data,"modelspec")
```

Outputs

`stepwiseMdl`	A linear model variable containing the coefficients and other information about the model.

Inputs

`data`	A table containing the data used to fit the regression model. See below for details.
`"modelspec"`	Starting model for the stepwise regression (see below).

As with `fitlm`, `stepwiselm` uses the last column of `data` as the response and all other columns as predictors.

How to Specify a Model

`stepwiselm` *chooses* the model for you. However, you can provide the following inputs to control the model selection process.

- `"modelspec"` - The second input to the function specifies the *starting model*. `stepwiselm` starts with this model and adds or removes terms based on certain criteria.

  Commonly used starting values: `"constant"`, `"linear"`, `"interactions"` (constant, linear, and interaction terms), `"quadratic"` (constant, linear, interaction, and quadratic terms).

- `"Lower"` and `"Upper"` - If you want to limit the complexity of the model, use these properties. For example, the following model will definitely contain the intercept and the linear terms but will not contain any terms with a degree of three or more.

  ```
  mdl = stepwiselm(data,"Lower","linear","Upper","quadratic")
  ```

  By default, `stepwiselm` considers models as simple as a constant term only and as complex as an interaction model.

How a Model is Chosen

`stepwiselm` iteratively adds and subtracts terms from the starting model, if the modified model is better than the previous iteration.

“Better” is judged according to the value of the `"Criterion"` property. The default value is `"sse"` — an *F*-test on the sum of squared error. You can change this measure by setting the `"Criterion"` property.

```
mdl = stepwiselm(data,"Criterion","rsquared")
```

Available values: `"sse"` (default), `"aic"`, `"bic"`, `"rsquared"`, `"adjrsquared"`

Regularized Linear Models

Ridge and Lasso Regression

In linear regression, the coefficients are chosen by minimizing the mean squared error (MSE). Mean squared error is the squared difference between the observed and the predicted response value.

$$\min_{c_0,cj}\left(\frac{1}{n}\sum_{i=1}^{n}\left(\underset{\text{Observed response}}{y_i} - \overset{\text{Predicted response}}{\boxed{c_0 - \sum_{j=1}^{p} x_{ij}c_j}}\right)^2\right)$$

In ridge and lasso regression, a penalty term is added to MSE.

This penalty term is composed of the fit coefficient values and a tuning parameter λ. The larger the value of λ, the greater the penalty and, therefore, the more the coefficients are “shrunk” towards zero.

Ridge Regression

$$\min_{c_0,cj}\left(\frac{1}{2n}\left(\sum_{i=1}^{n}\left(y_i - c_0 - \sum_{j=1}^{p} x_{ij}c_j\right)^2 + \overset{\text{Penalty term}}{\boxed{\lambda\sum_{j=1}^{p} c_j^2}}\right)\right)$$

Lasso

$$\min_{c_0,cj}\left(\frac{1}{2n}\sum_{i=1}^{n}\left(y_i - c_0 - \sum_{j=1}^{p} x_{ij}c_j\right)^2 + \overset{\text{Penalty term}}{\boxed{\lambda\sum_{j=1}^{p} |c_j|}}\right)$$

The Penalty Term

The difference between the two methods is how the penalty term is calculated. Ridge regression uses an L^2 norm of the coefficients. Lasso uses an L^1 norm.

The use of different norms provides different regularization behavior.

- Ridge regression shrinks coefficients continuously, and keeps all predictors.
- Lasso allows coefficients to be set to zero, reducing the number of predictors included in the model.

Lasso can be used as a form of feature selection, however feature selection may not be appropriate for cases with similar, highly correlated variables. It may result in loss of information which could impact accuracy and the interpretation of results. Ridge regression maintains all features, but the model may still be very complex if there is a large number of predictors.

Elastic Net

You can also use a penalty term that uses a weighted average of both. This is *elastic net* regression, which introduces another parameter – the weighting between ridge (L^2 norm) and lasso (L^1 norm).

Fitting Ridge Regression Models

You can use the function `ridge` to fit a ridge regression model.

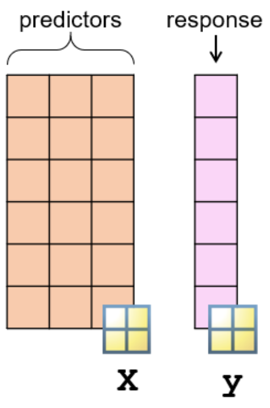
```
>> b = ridge(y,X,lambda,scaled)
```

Outputs

b	Ridge regression coefficients.

Inputs

y	Response values, specified as a vector.
X	Predictor values, specified as a numeric matrix.
lambda	Regularization parameter.
scaled	A {0,1}-valued flag to determine if the coefficient estimates in `b` are restored to the scale of the original data.



Notes

- The matrix `X` is a numeric *design matrix* with columns representing the terms in the regression formula. If the original data contains two predictive variables x1 and x2, but the desired regression model formula contains the terms x1, x2, and x1*x2, the matrix `X` should have 3 columns: x1, x2, and x1*x2.

- The ridge parameter `lambda` is a non-negative number. In a later section, you will try to estimate the optimum value of `lambda`.

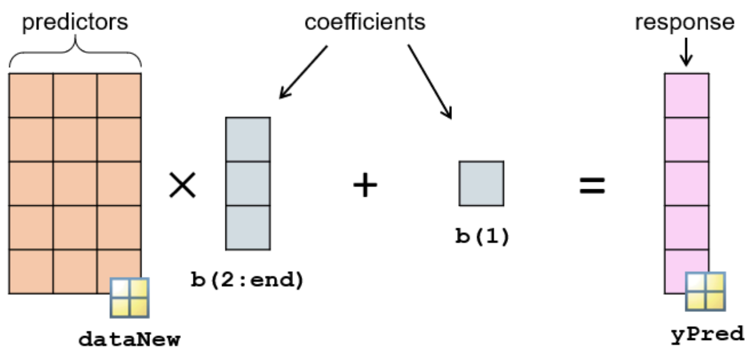
- The `ridge` function normalizes the predictors before fitting the model. Therefore, by default, the regression coefficients correspond to the normalized data. Set the `scaled` flag to `0` to restore the coefficients to the scale of the original data.

Predicting Response using Ridge Regression Models

When using a ridge regression model for predictions, you will want the regression coefficients in the scale of the original data. In this case (the `scaled` flag set to `0`), the coefficient vector `b` will contain *n*+1 coefficients for a model with *n* predictors. The first element of `b` corresponds with the intercept term.

You can predict the response by multiplying the matrix containing the predictors and the last *n* elements of the coefficient vector. Add the first element of the coefficient vector to incorporate the intercept in the calculation.

```
yPred = dataNew*b(2:end) + b(1);
```



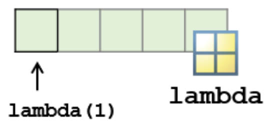
Determining the Regression Parameter

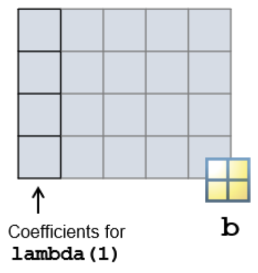
When modeling data using ridge regression, how can you determine the regression parameter, `lambda`?

When you provide a vector of λ values to the `ridge` function, the output `b` is a matrix of coefficients.

```
b = ridge(y,X,lambda,scaled);
```

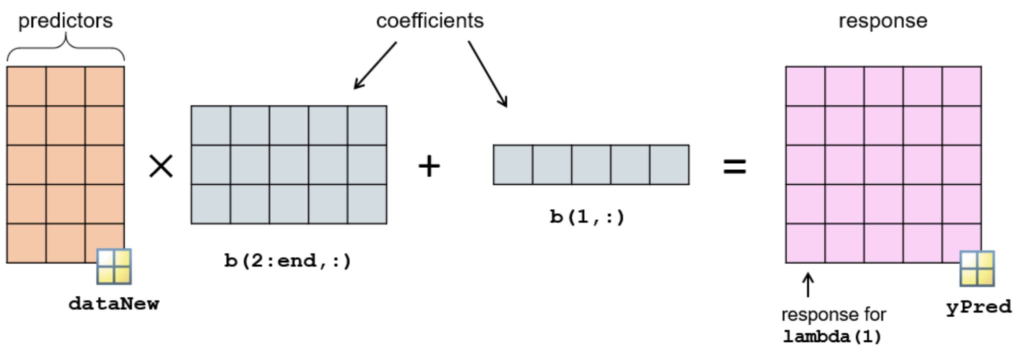
The columns of `b` contain the coefficient values for each parameter in the vector `lambda`.





You can now use each column of the matrix `b` as regression coefficients and predict the response.

```
yPred = dataNew*b(2:end,:) + b(1,:);
```



The response `yPred` is a matrix where each column is the predicted response for the corresponding value of `lambda`. You can use `yPred` to calculate the mean squared error (MSE), and choose the coefficients which minimize MSE.

Fitting Lasso Regression Models

Lasso (**L**east **A**bsolute **S**hrinkage and **S**election **O**perator) regression models are fit using the `lasso` function.

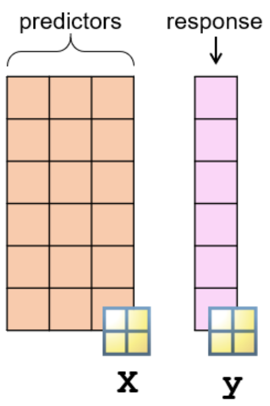
```
>> [b,fitInfo] = lasso(X,y,"Lambda",lambda)
```

Outputs

`b`	Lasso coefficients.
`fitInfo`	A structure containing information about the model.

Inputs

`X`	Predictor values, specified as a numeric matrix.
`y`	Response values, specified as a vector.
`"Lambda"`	Property name for specifying the regularization parameter.
`lambda`	Regularization parameter value.



Notes

- Like in ridge, the matrix `X` is a design matrix with columns representing the terms in the regression formula.
- The `"Lambda"` property is optional. If not specified, `lasso` uses a geometric sequence of λ values based on the data.
- `ridge` and `lasso` implement their penalty terms slightly differently, and as a result, use different scalings for `lambda`. To use λ values in `lasso` which have the same interpretation as for `ridge`, scale `lambda` in `lasso` by the number of observations.
- Use the optional property `"Alpha"` with a value between 0 and 1 to create an elastic net. Recall that elastic net regression uses a penalty term which is a weighted average of the ridge (L^2) and lasso (L^1) penalty terms. `"Alpha"` values near 1 are closer to lasso, and `"Alpha"` values near 0 are closer to ridge.

Predicting Response using Lasso Model

You can predict the response by multiplying the matrix containing the predictors by the coefficient vector.

Note that the intercept term is **not** included in the output coefficients. Instead, it is a field in the output structure `fitInfo`.

```
yPred = dataNew*b + fitInfo.Intercept
```

SVMs, Trees and Neural Networks

Regression using SVMs, Decision Trees, and Neural Networks

Linear regression techniques like `fitlm` are parametric, which means predictions are based on a finite set of parameters that are estimated from the training data.

A *parametric regression* model:

- Assumes a relationship that can be specified using a formula.
- Is easy to interpret – you can measure the change in the response per unit change in the predictor.

$$y = c_0 + c_1 x_1 + c_2 x_2 + c_3 (x_3)^3$$

You may not need a model with a specific interpretable formula if its primary purpose is simply to predict the response for unknown observations. In this case, you can use a nonparametric model.

A *nonparametric regression* model:

- Does not fit the regression model based on a given formula.
- Can provide more accurate predictions, but is more difficult to interpret.

$$No\ Formula$$

Support vector machines (SVMs), decision trees, and neural networks are some of the nonparametric techniques you can use for regression.

`fitrtree`	*Fit binary regression decision tree.*
`fitrsvm`	*Fit a support vector machine regression model.*
`fitrnet`	*Fit a neural network for regression.*

These techniques are covered in detail in the Classification Methods chapter. You can find information specific to regression in the documentation.

Gaussian Process Regression

Fitting and Predicting Using Gaussian Process Regression

Fitting GPR Models

You can fit a GPR model by using the `fitrgp` function.

```
>> mdl = fitrgp(data,"ResponseVarName")
```

Outputs

`mdl`	A GPR model variable.

Inputs

`data`	A table containing the predictor and response values.
`"ResponseVarName"`	Name of the response variable.

Use the `"KernelFunction"` property to change the kernel to one of the predefined options.

```
mdl = fitrgp(data,"ResponseVarName","KernelFunction","exponential")
```

Predicting the Response

In addition to the predicted response values, the `predict` function for GPR models can also return the standard deviation and prediction intervals for the predicted values.

```
>> [yPred,yStd,yInt] = predict(mdl,dataNew)
```

Outputs

`yPred`	Predicted response value(s).
`yStd`	Standard deviation for each predicted value.
`yInt`	Matrix whose columns contain the lower and upper limits of the 95% prediction interval for each predicted value.

Inputs

`mdl`	A GPR model variable.
`dataNew`	Predictor values for one or more new observations.

You can change the significance level of the prediction intervals by setting the `"Alpha"` property to a value between 0 and 1. The default value is 0.05.

```
[yPred,yStd,yInt] = predict(mdl,dataNew,"Alpha",0.01)
```