13

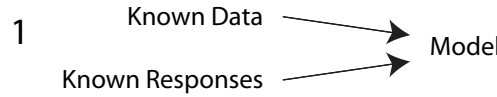
Supervised Learning

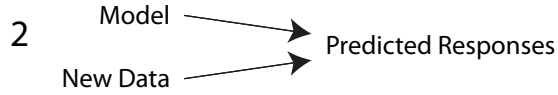
Supervised Learning (Machine Learning) Workflow and Algorithms

In this section...

Steps in Supervised Learning (Machine Learning)

Supervised learning (machine learning) takes a known set of input data and known responses to the data, and seeks to build a predictor model that generates reasonable predictions for the response to new data.





For example, suppose you want to predict if someone will have a heart attack within a year. You have a set of data on previous people, including their ages, weight, height, blood pressure, etc. You know if the previous people had heart attacks within a year of their data measurements. So the problem is combining all the existing data into a model that can predict whether a new person will have a heart attack within a year.

Supervised learning splits into two broad categories:

- Classification for responses that can have just a few known values, such as `'true'` or `'false'`. Classification algorithms apply to nominal, not ordinal response values.

- Regression for responses that are a real number, such as miles per gallon for a particular car.

You can have trouble deciding whether you have a classification problem or a regression problem. In that case, create a regression model first—regression models are often more computationally efficient.

While there are many Statistics Toolbox algorithms for supervised learning, most use the same basic workflow for obtaining a predictor model:

Prepare Data

All supervised learning methods start with an input data matrix, usually called `X` in this documentation. Each row of `X` represents one observation. Each column of `X` represents one variable, or predictor. Represent missing entries with `NaN` values in `X`. Statistics Toolbox supervised learning algorithms can handle `NaN` values, either by ignoring them or by ignoring any row with a `NaN` value.

You can use various data types for response data `Y`. Each element in `Y` represents the response to the corresponding row of `X`. Observations with missing `Y` data are ignored.

- For regression, `Y` must be a numeric vector with the same number of elements as the number of rows of `X`.
- For classification, `Y` can be any of these data types. The table also contains the method of including missing entries.

Data Type	Missing Entry
Numeric vector	`NaN`
Categorical vector	`<undefined>`
Character array	Row of spaces
Cell array of strings	`''`
Logical vector	(not possible to represent)

Choose an Algorithm

There are tradeoffs between several characteristics of algorithms, such as:

- Speed of training
- Memory utilization
- Predictive accuracy on new data
- Transparency or interpretability, meaning how easily you can understand the reasons an algorithm makes its predictions

Details of the algorithms appear in "Characteristics of Algorithms" on page 13-6. More detail about ensemble algorithms is in "Choose an Applicable Ensemble Method" on page 13-53.

Fit a Model

The fitting function you use depends on the algorithm you choose.

- For classification trees or regression trees, use `ClassificationTree.fit` or `RegressionTree.fit`.
- For classification or regression trees using an older toolbox function, use `classregtree`.
- For classification or regression ensembles, use `fitensemble`.
- For classification or regression ensembles in parallel, or to use specialized `TreeBagger` functionality such as outlier detection, use `TreeBagger`.

Choose a Validation Method

The three main methods for examining the accuracy of the resulting fitted model are:

- Examine resubstitution error. For examples, see:
  - "Example: Resubstitution Error of a Classification Tree" on page 13-33

    "Example: Cross Validating a Regression Tree" on page 13-34

    "Example: Test Ensemble Quality" on page 13-59
- Examine the cross-validation error. For examples, see:
  - "Example: Cross Validating a Regression Tree" on page 13-34
  - "Example: Test Ensemble Quality" on page 13-59
  - "Example: Classification with Many Categorical Levels" on page 13-71
- Examine the out-of-bag error for bagged decision trees. For examples, see:
  - "Example: Test Ensemble Quality" on page 13-59
  - "Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger" on page 13-97
  - "Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger" on page 13-106

Examine Fit; Update Until Satisfied

After validating the model, you might want to change it for better accuracy, better speed, or to use less memory.

- Change fitting parameters to try to get a more accurate model. For examples, see:
  - "Example: Tuning RobustBoost" on page 13-92
  - "Example: Unequal Classification Costs" on page 13-66
- Change fitting parameters to try to get a smaller model. This sometimes gives a model with more accuracy. For examples, see:
  - "Example: Selecting Appropriate Tree Depth" on page 13-35
  - "Example: Pruning a Classification Tree" on page 13-38

- "Example: Surrogate Splits" on page 13-76
- "Example: Regularizing a Regression Ensemble" on page 13-82
- "Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger" on page 13-97
- "Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger" on page 13-106

- Try a different algorithm. For applicable choices, see:
  - "Characteristics of Algorithms" on page 13-6
  - "Choose an Applicable Ensemble Method" on page 13-53

When you are satisfied with the model, you can trim it using the appropriate `compact` method (`compact` for classification trees, `compact` for classification ensembles, `compact` for regression trees, `compact` for regression ensembles). `compact` removes training data and pruning information, so the model uses less memory.

Use Fitted Model for Predictions

To predict classification or regression response for most fitted models, use the `predict` method:

```
Ypredicted = predict(obj,Xnew)
```

- `obj` is the fitted model object.
- `Xnew` is the new input data.
- `Ypredicted` is the predicted response, either classification or regression.

For `classregtree`, use the `eval` method instead of `predict`.

Characteristics of Algorithms

This table shows typical characteristics of the various supervised learning algorithms. The characteristics in any particular case can vary from the listed ones. Use the table as a guide for your initial choice of algorithms, but be aware that the table can be inaccurate for some problems. SVM is available if you have a Bioinformatics Toolbox™ license.

Characteristics of Supervised Learning Algorithms

Algorithm	Predictive Accuracy	Fitting Speed	Prediction Speed	Memory Usage	Easy to Interpret	Handles Categorical Predictors
Trees	Low	Fast	Fast	Low	Yes	Yes
Boosted Trees	High	Medium	Medium	Medium	No	Yes
Bagged Trees	High	Slow	Slow	High	No	Yes
SVM	High	Medium	*	*	*	No
Naive Bayes	Low	**	**	**	Yes	Yes
Nearest Neighbor	***	Fast***	Medium	High	No	Yes***

* — SVM prediction speed and memory usage are good if there are few support vectors, but can be poor if there are many support vectors. When you use a kernel function, it can be difficult to interpret how SVM classifies data, though the default linear scheme is easy to interpret.

** — Naive Bayes speed and memory usage are good for simple distributions, but can be poor for kernel distributions and large data sets.

*** — Nearest Neighbor usually has good predictions in low dimensions, but can have poor predictions in high dimensions. For linear search, Nearest Neighbor does not perform any fitting. For *k*d-trees, Nearest Neighbor does perform fitting. Nearest Neighbor can have either continuous or categorical predictors, but not both.

Classification Using Nearest Neighbors

In this section...

Pairwise Distance

Categorizing query points based on their distance to points in a training dataset can be a simple yet effective way of classifying new points. You can use various metrics to determine the distance, described next. Use `pdist2` to find the distance between a sets of data and query points.

Distance Metrics

Given an *mx*-by-*n* data matrix `X`, which is treated as *mx* (1-by-*n*) row vectors $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_{mx}$, and *my*-by-*n* data matrix `Y`, which is treated as *my* (1-by-*n*) row vectors $\mathtt{y}_1$, $\mathtt{y}_2$, ...,$\mathtt{y}_{my}$, the various distances between the vector $\mathtt{x}_s$ and $\mathtt{y}_t$ are defined as follows:

- Euclidean distance

  $$d_{st}^2 = (x_s - y_t)(x_s - y_t)'$$

  The Euclidean distance is a special case of the Minkowski metric, where `p` = 2.

- Standardized Euclidean distance

  $$d_{st}^2 = (x_s - y_t)V^{-1}(x_s - y_t)'$$

  where `V` is the *n*-by-*n* diagonal matrix whose *j*th diagonal element is $\mathtt{S}(j)^2$, where `S` is the vector containing the inverse weights.

- Mahalanobis distance

  $$d_{st}^2 = (x_s - y_t)C^{-1}(x_s - y_t)'$$

  where `C` is the covariance matrix.

- City block metric

$$d_{st} = \sum_{j=1}^{n} \left| x_{sj} - y_{tj} \right|$$

  The city block distance is a special case of the Minkowski metric, where p = 1.

- Minkowski metric

$$d_{st} = \sqrt[p]{\sum_{j=1}^{n} \left| x_{sj} - y_{tj} \right|^{p}}$$

  For the special case of p = 1, the Minkowski metric gives the city block metric, for the special case of p = 2, the Minkowski metric gives the Euclidean distance, and for the special case of p = ∞, the Minkowski metric gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \left\{ \left| x_{sj} - y_{tj} \right| \right\}$$

  The Chebychev distance is a special case of the Minkowski metric, where p = ∞.

- Cosine distance

$$d_{st} = \left(1 - \frac{x_s y_t'}{\sqrt{(x_s x_s')(y_t y_t')}} \right)$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(y_t - \bar{y}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)'} \sqrt{(y_t - \bar{y}_t)(y_t - \bar{y}_t)'}}$$

  where

$$\bar{x}_s = \frac{1}{n}\sum_j x_{sj}$$

and

$$\bar{y}_t = \frac{1}{n}\sum_j y_{tj}$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq y_{tj}) / n)$$

- Jaccard distance

$$d_{st} = \frac{\#\left[\left(x_{sj} \neq y_{tj}\right) \cap \left(\left(x_{sj} \neq 0\right) \cup \left(y_{tj} \neq 0\right)\right)\right]}{\#\left[\left(x_{sj} \neq 0\right) \cup \left(y_{tj} \neq 0\right)\right]}$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)'}\sqrt{(r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

  where

  - r_{sj} is the rank of x_{sj} taken over x_{1j}, x_{2j}, ...$x_{mx,j}$, as computed by `tiedrank`.
  - r_{tj} is the rank of y_{tj} taken over y_{1j}, y_{2j}, ...$y_{my,j}$, as computed by `tiedrank`.
  - r_s and r_t are the coordinate-wise rank vectors of x_s and y_t, i.e., $r_s = (r_{s1}, r_{s2}, \ldots r_{sn})$ and $r_t = (r_{t1}, r_{t2}, \ldots r_{tn})$.

  - $\bar{r}_s = \frac{1}{n}\sum_j r_{sj} = \frac{(n+1)}{2}$.

  - $\bar{r}_t = \frac{1}{n}\sum_j r_{tj} = \frac{(n+1)}{2}$.

k-Nearest Neighbor Search

Given a set X of n points and a distance function `D`, k-nearest neighbor (kNN) search lets you find the k closest points in X to a query point or set of points. The kNN search technique and kNN-based algorithms are widely used as benchmark learning rules—the relative simplicity of the kNN search technique makes it easy to compare the results from other classification techniques to kNN results. They have been used in various areas such as bioinformatics, image processing and data compression, document retrieval, computer vision, multimedia database, and marketing data analysis. You can use kNN search for other machine learning algorithms, such as kNN classification, local weighted regression, missing data imputation and interpolation, and density estimation. You can also use kNN search with many distance-based learning functions, such as K-means clustering.

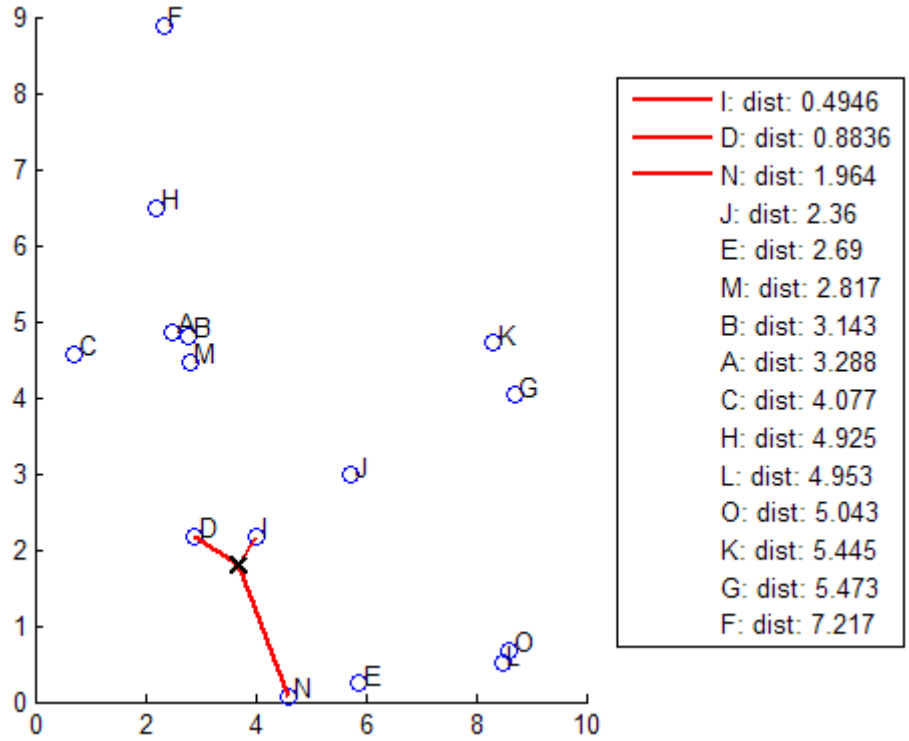
k-Nearest Neighbor Search Using Exhaustive Search

When your input data meets any of the following criteria, `knnsearch` uses the exhaustive search method by default to find the k-nearest neighbors:

- The number of columns of `X` is more than 10.
- `X` is sparse.
- The distance measure is either:
  - `'seuclidean'`
  - `'mahalanobis'`
  - `'cosine'`
  - `'correlation'`
  - `'spearman'`
  - `'hamming'`
  - `'jaccard'`
  - A custom distance function

`knnsearch` also uses the exhaustive search method if your search object is an `ExhaustiveSearcher` object. The exhaustive search method finds the distance from each query point to every point in `X`, ranks them in ascending

order, and returns the k points with the smallest distances. For example, this diagram shows the $k = 3$ nearest neighbors.



k-Nearest Neighbor Search Using a *k*d-Tree

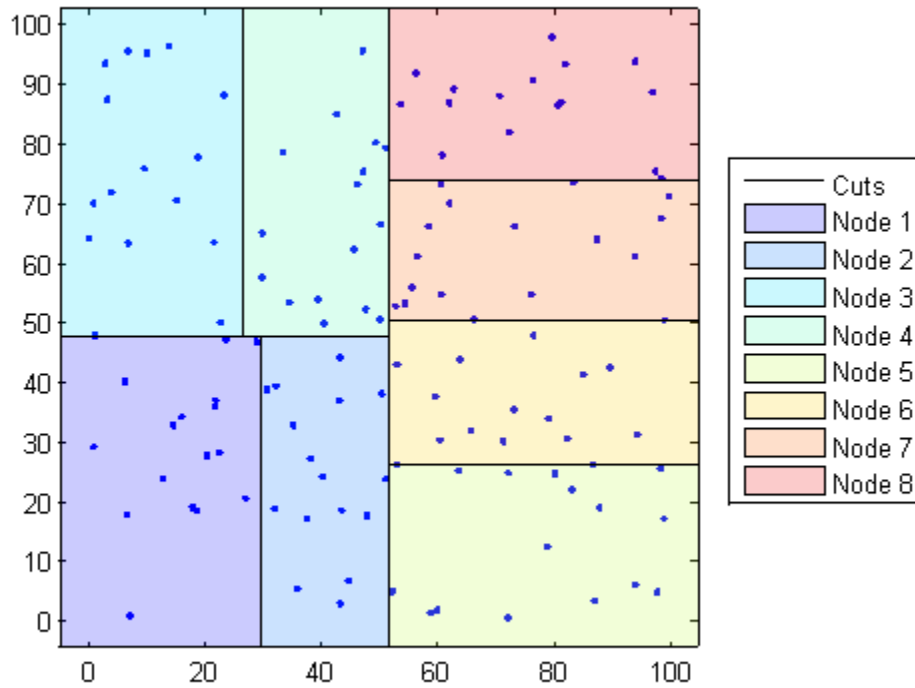
When your input data meets all of the following criteria, `knnsearch` creates a *k*d-tree by default to find the *k*-nearest neighbors:

- The number of columns of `X` is less than 10.
- `X` is not sparse.
- The distance measure is either:
  - `'euclidean'` (default)
  - `'cityblock'`

- `'minkowski'`
- `'chebychev'`

`knnsearch` also uses a *k*d-tree if your search object is a `KDTreeSearcher` object.

*k*d-trees divide your data into nodes with at most `BucketSize` (default is 50) points per node, based on coordinates (as opposed to categories). The following diagrams illustrate this concept using `patch` objects to color code the different "buckets."



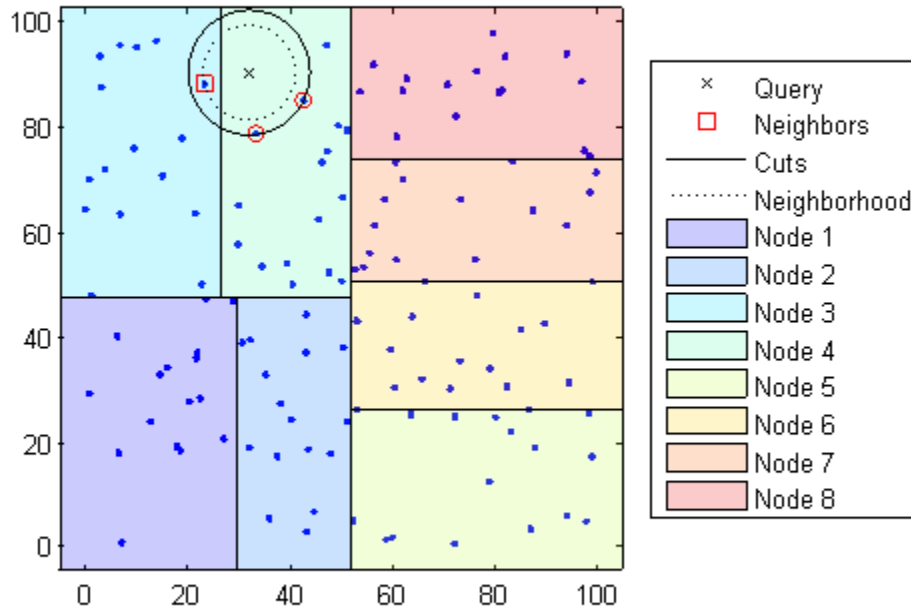
When you want to find the *k*-nearest neighbors to a given query point, `knnsearch` does the following:

1. Determines the node to which the query point belongs. In the following example, the query point (32,90) belongs to Node 4.

2. Finds the closest k points within that node and its distance to the query point. In the following example, the points in red circles are equidistant from the query point, and are the closest points to the query point within Node 4.

3. Chooses all other nodes having any area that is within the same distance, in any direction, from the query point to the kth closest point. In this example, only Node 3 overlaps the solid black circle centered at the query point with radius equal to the distance to the closest points within Node 4.

4. Searches nodes within that range for any points closer to the query point. In the following example, the point in a red square is slightly closer to the query point than those within Node 4.



Using a *k*d-tree for large datasets with fewer than 10 dimensions (columns) can be much more efficient than using the exhaustive search method, as `knnsearch` needs to calculate only a subset of the distances. To maximize the efficiency of *k*d-trees, use a `KDTreeSearcher` object.

What Are Search Objects?

Basically, objects are a convenient way of storing information. Classes of related objects (for example, all search objects) have the same properties with values and types relevant to a specified search method. In addition to storing information within objects, you can perform certain actions (called *methods*) on objects.

All search objects have a `knnsearch` method specific to that class. This lets you efficiently perform a *k*-nearest neighbors search on your object for that specific object type. In addition, there is a generic `knnsearch` function that searches without creating or using an object.

To determine which type of object and search method is best for your data, consider the following:

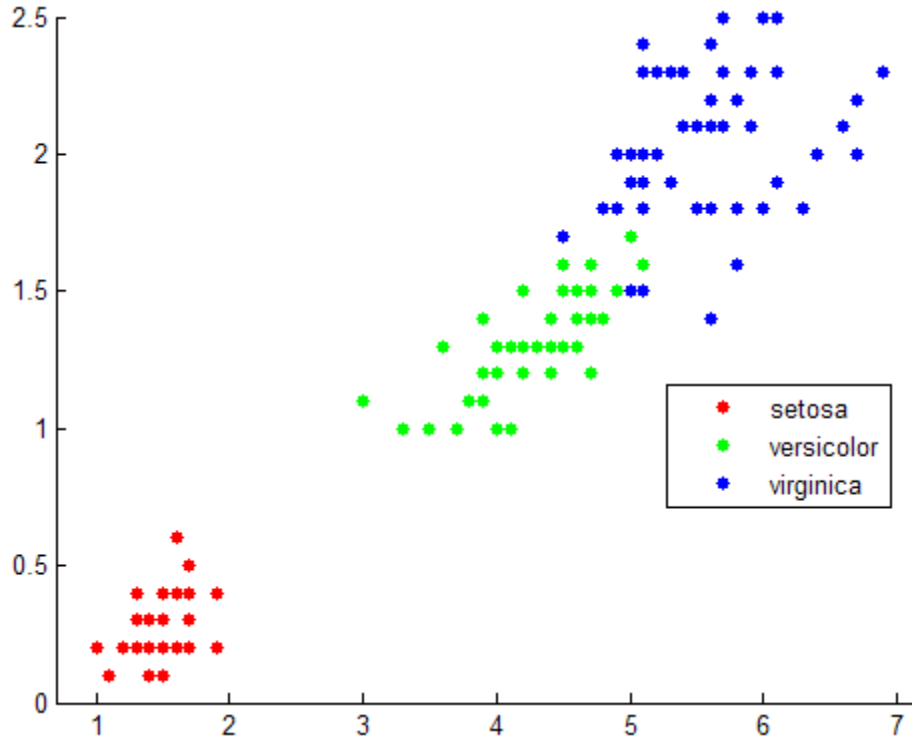
- Does your data have many columns, say more than 10? The `ExhaustiveSearcher` object may perform better.
- Is your data sparse? Use the `ExhaustiveSearcher` object.
- Do you want to use one of these distance measures to find the nearest neighbors? Use the `ExhaustiveSearcher` object.
  - `'seuclidean'`
  - `'mahalanobis'`
  - `'cosine'`
  - `'correlation'`
  - `'spearman'`
  - `'hamming'`
  - `'jaccard'`
  - A custom distance function
- Is your dataset huge (but with fewer than 10 columns)? Use the `KDTreeSearcher` object.
- Are you searching for the nearest neighbors for a large number of query points? Use the `KDTreeSearcher` object.

For more detailed information on object-oriented programming in MATLAB, see *Object-Oriented Programming*.

Example: Classifying Query Data Using knnsearch

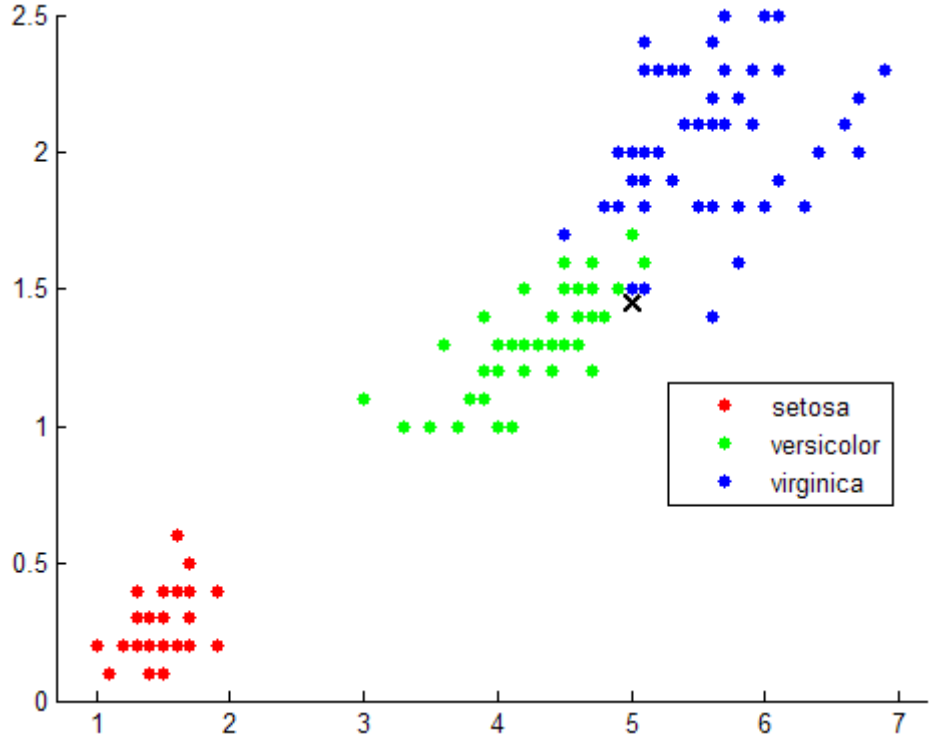
1 Classify a new point based on the last two columns of the Fisher iris data. Using only the last two columns makes it easier to plot:

```
load fisheriris
x = meas(:,3:4);
gscatter(x(:,1),x(:,2),species)
set(legend,'location','best')
```



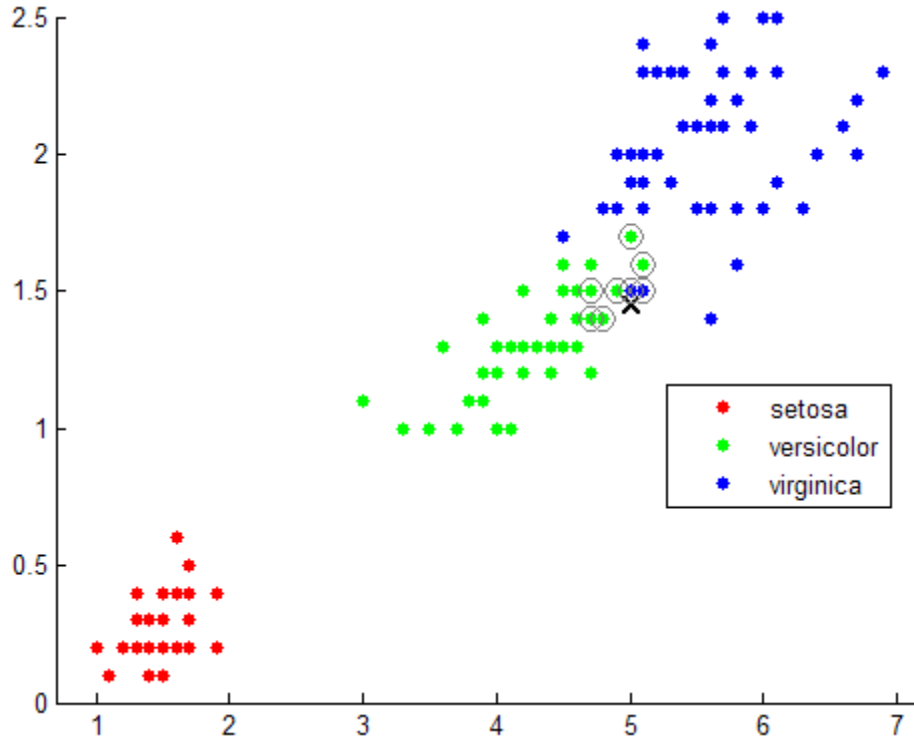
2 Plot the new point:

```
newpoint = [5 1.45];
line(newpoint(1),newpoint(2),'marker','x','color','k',...
   'markersize',10,'linewidth',2)
```



3 Find the 10 sample points closest to the new point:

```
[n,d] = knnsearch(x,newpoint,'k',10)
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...
    'linestyle','none','markersize',10)
```



4 It appears that `knnsearch` has found only the nearest eight neighbors. In fact, this particular dataset contains duplicate values:

```
x(n,:)

ans =

    5.0000    1.5000
    4.9000    1.5000
    4.9000    1.5000
    5.1000    1.5000
    5.1000    1.6000
    4.8000    1.4000
    5.0000    1.7000
    4.7000    1.4000
    4.7000    1.4000
```

```
    4.7000    1.5000
```

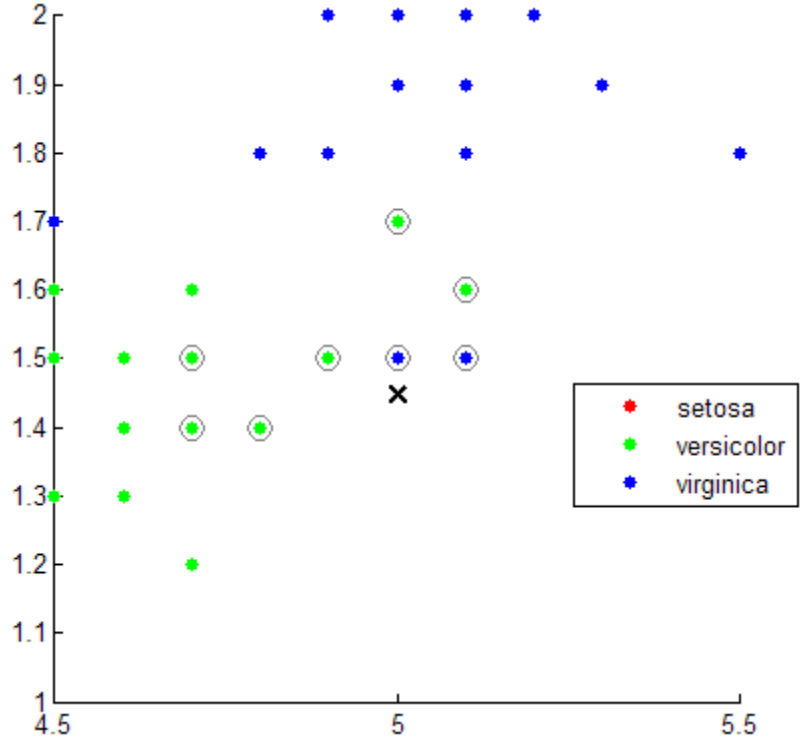
5 To make duplicate values visible on the plot, use the following code:

```
% jitter to make repeated points visible
xj = x + .05*(rand(150,2)-.5);
gscatter(xj(:,1),xj(:,2),species)
```

The jittered points do not affect any analysis of the data, only the visualization. This example does not jitter the points.

6 Make the axes equal so the calculated distances correspond to the apparent distances on the plot axis equal and zoom in to see the neighbors better:

```
set(gca,'xlim',[4.5 5.5],'ylim',[1 2]); axis square
```



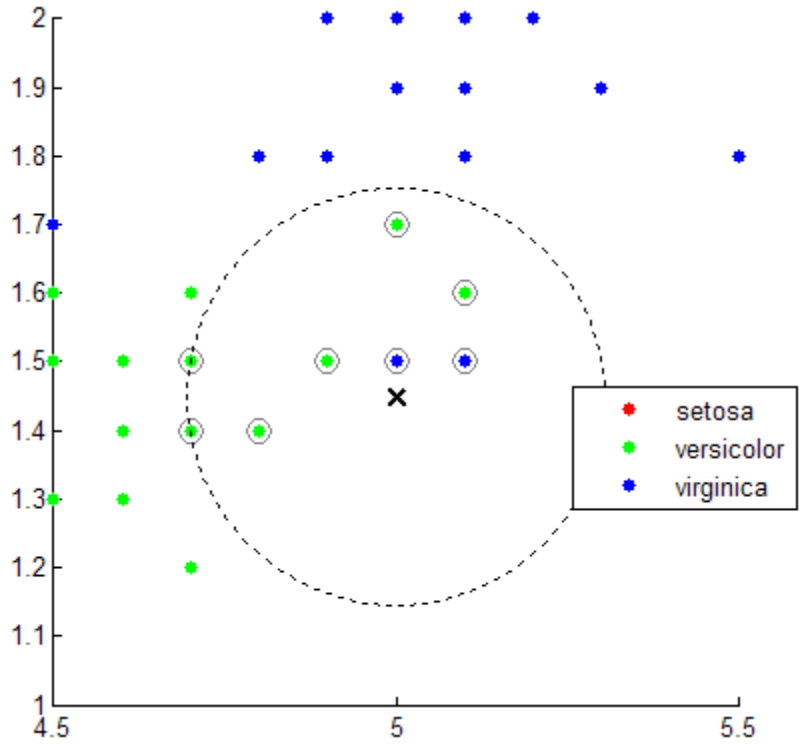
7 Find the species of the 10 neighbors:

```
tabulate(species(n))

       Value    Count   Percent
   virginica        2     20.00%
  versicolor        8     80.00%
```

Using a rule based on the majority vote of the 10 nearest neighbors, you can classify this new point as a versicolor.

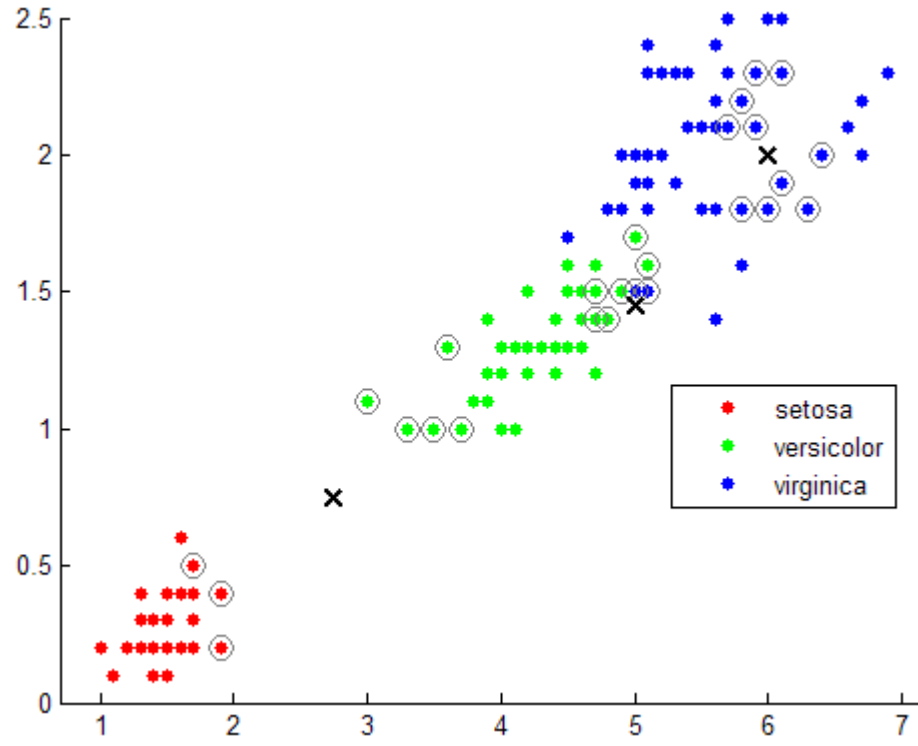
8 Visually identify the neighbors by drawing a circle around the group of them:

```
% Define the center and diameter of a circle, based on the
% location of the new point:
ctr = newpoint - d(end);
diameter = 2*d(end);
% Draw a circle around the 10 nearest neighbors:
h = rectangle('position',[ctr,diameter,diameter],...
   'curvature',[1 1]);
set(h,'linestyle',':')
```



9 Using the same dataset, find the 10 nearest neighbors to three new points:

```
figure
newpoint2 = [5 1.45;6 2;2.75 .75];
gscatter(x(:,1),x(:,2),species)
legend('location','best')
[n2,d2] = knnsearch(x,newpoint2,'k',10);
line(x(n2,1),x(n2,2),'color',[.5 .5 .5],'marker','o',...
   'linestyle','none','markersize',10)
line(newpoint2(:,1),newpoint2(:,2),'marker','x','color','k',...
   'markersize',10,'linewidth',2,'linestyle','none')
```



10 Find the species of the 10 nearest neighbors for each new point:

```
tabulate(species(n2(1,:)))
       Value    Count   Percent
   virginica        2     20.00%
  versicolor        8     80.00%

tabulate(species(n2(2,:)))
      Value    Count   Percent
  virginica       10    100.00%

tabulate(species(n2(3,:)))
       Value    Count   Percent
  versicolor        7     70.00%
      setosa        3     30.00%
```

For further examples using `knnsearch` methods and function, see the individual reference pages.

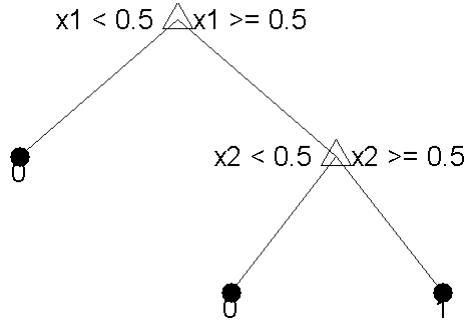
Classification Trees and Regression Trees

In this section...

What Are Classification Trees and Regression Trees?

Classification trees and regression trees predict responses to data. To predict a response, follow the decisions in the tree from the root (beginning) node down to a leaf node. The leaf node contains the response. Classification trees give responses that are nominal, such as `'true'` or `'false'`. Regression trees give numeric responses.

Statistics Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor (variable). For example, here is a simple classification tree:



This tree predicts classifications based on two predictors, `x1` and `x2`. To predict, start at the top node, represented by a triangle (Δ). The first decision is whether `x1` is smaller than `0.5`. If so, follow the left branch, and see that the tree classifies the data as type `0`.

If, however, x1 exceeds 0.5, then follow the right branch to the lower-right triangle node. Here the tree asks if x2 is smaller than 0.5. If so, then follow the left branch to see that the tree classifies the data as type 0. If not, then follow the right branch to see that the that the tree classifies the data as type 1.

Creating Classification Trees and Regression Trees

1. Collect your known input data into a matrix X. Each row of X represents one observation. Each column of X represents one variable (also called a predictor). Use NaN to represent a missing value.

2. Collect the responses to X in a response variable Y. Each entry in Y represents the response to the corresponding row of X. Represent missing values as shown in Response Data Types on page 13-26.

   - For regression, Y must be a numeric vector with the same number of elements as the number of rows of X.
   - For classification, Y can be any of the following data types; the table also contains the method of including missing entries:

   Response Data Types

Data Type	Missing Entry
Numeric vector	NaN
Categorical vector	<undefined>
Character array	Row of spaces
Cell array of strings	''
Logical vector	(not possible to represent)

   For example, suppose your response data consists of three observations in this order: true, false, true. You could express Y as:

   - [1;0;1] (numeric vector)
   - nominal({'true','false','true'}) (categorical vector)
   - [true;false;true] (logical vector)

- `['true ';'false';'true ']` (character array, padded with spaces so each row has the same length)
- `{'true','false','true'}` (cell array of strings)

Use whichever data type is most convenient.

3 Create a tree using one of these methods:

- For a classification tree, use `ClassificationTree.fit`:

```
tree = ClassificationTree.fit(X,Y);
```

- For a regression tree, use `RegressionTree.fit`:

```
tree = RegressionTree.fit(X,Y);
```

Example: Creating a Classification Tree

To create a classification tree for the `ionosphere` data:

```
load ionosphere % contains X and Y variables
ctree = ClassificationTree.fit(X,Y)

ctree =

ClassificationTree:
           PredictorNames: {1x34 cell}
    CategoricalPredictors: []
             ResponseName: 'Y'
               ClassNames: {'b'  'g'}
           ScoreTransform: 'none'
            NObservations: 351
```

Example: Creating a Regression Tree

To create a regression tree for the `carsmall` data based on the `Horsepower` and `Weight` vectors for data, and `MPG` vector for response:

```
load carsmall % contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = RegressionTree.fit(X,MPG)
```

```
rtree =

RegressionTree:
           PredictorNames: {'x1'  'x2'}
    CategoricalPredictors: []
             ResponseName: 'Y'
        ResponseTransform: 'none'
            NObservations: 94
```

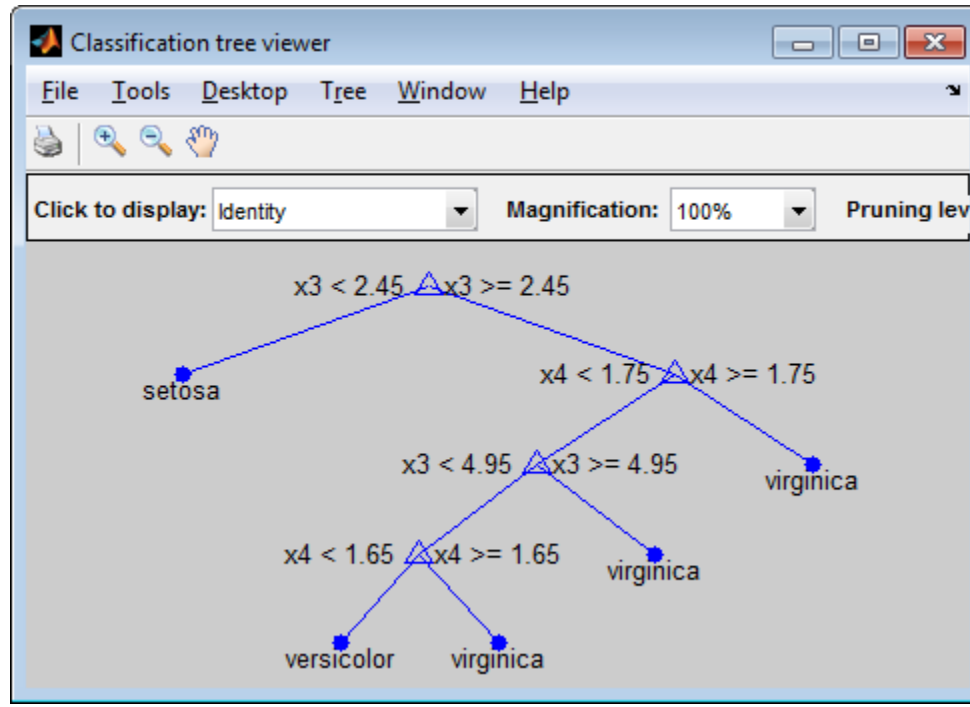
Viewing a Tree

There are two ways to view a tree:

- `view(tree)` returns a text description of the tree.
- `view(tree,'mode','graph')` returns a graphic description of the tree.

"Example: Creating a Classification Tree" on page 13-27 has the following two views:

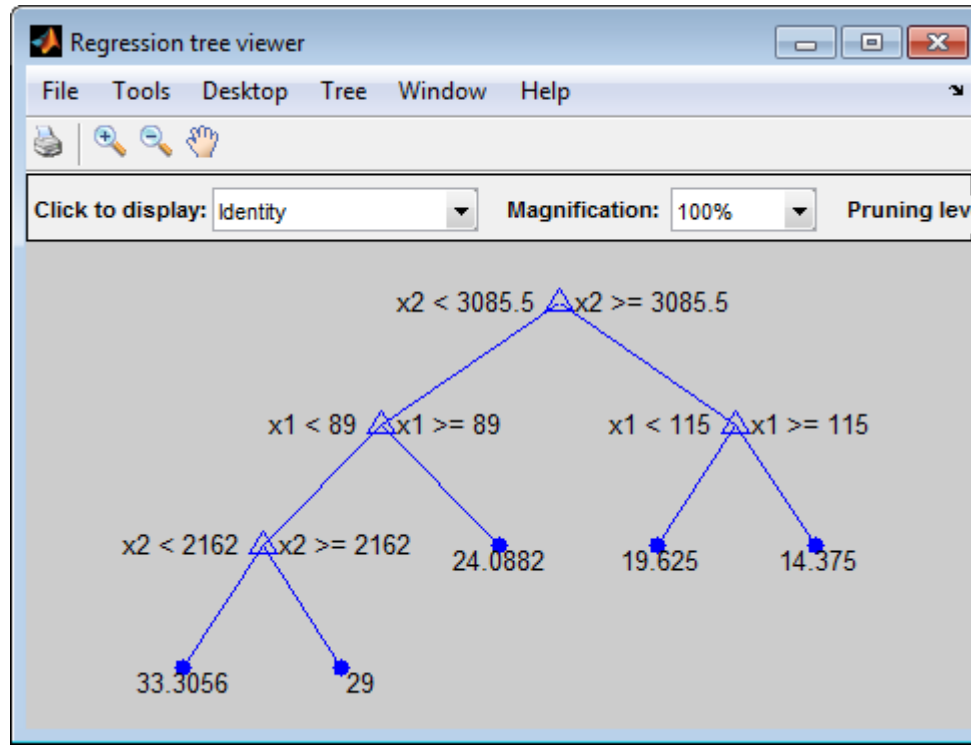
```
load fisheriris
ctree = ClassificationTree.fit(meas,species);
view(ctree)

Decision tree for classification
1  if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else setosa
2  class = setosa
3  if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else versicolor
4  if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else versicolor
5  class = virginica
6  if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(ctree,'mode','graph')
```



Similarly, "Example: Creating a Regression Tree" on page 13-27 has the following two views:

```
load carsmall % contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = RegressionTree.fit(X,MPG,'MinParent',30);
view(rtree)

Decision tree for regression
1  if x2<3085.5 then node 2 elseif x2>=3085.5 then node 3 else 23.7181
2  if x1<89 then node 4 elseif x1>=89 then node 5 else 28.7931
3  if x1<115 then node 6 elseif x1>=115 then node 7 else 15.5417
4  if x2<2162 then node 8 elseif x2>=2162 then node 9 else 30.9375
5  fit = 24.0882
6  fit = 19.625
7  fit = 14.375
```

```
8   fit = 33.3056
9   fit = 29

view(rtree,'mode','graph')
```



How the Fit Methods Create Trees

The `ClassificationTree.fit` and `RegressionTree.fit` methods perform the following steps to create decision trees:

1. Start with all input data, and examine all possible binary splits on every predictor.
2. Select a split with best optimization criterion.

- If the split leads to a child node having too few observations (less than the `MinLeaf` parameter), select a split with the best optimization criterion subject to the `MinLeaf` constraint.

3 Impose the split.

4 Repeat recursively for the two child nodes.

The explanation requires two more items: description of the optimization criterion, and stopping rule.

Stopping rule: Stop splitting when any of the following hold:

- The node is *pure*.
  - For classification, a node is pure if it contains only observations of one class.
  - For regression, a node is pure if the mean squared error (MSE) for the observed response in this node drops below the MSE for the observed response in the entire data multiplied by the tolerance on quadratic error per node (`qetoler` parameter).
- There are fewer than `MinParent` observations in this node.
- Any split imposed on this node would produce children with fewer than `MinLeaf` observations.

Optimization criterion:

- Regression: mean-squared error (MSE). Choose a split to minimize the MSE of predictions compared to the training data.
- Classification: One of three measures, depending on the setting of the `SplitCriterion` name-value pair:
  - `'gdi'` (Gini's diversity index, the default)
  - `'twoing'`
  - `'deviance'`

  For details, see `ClassificationTree` "Definitions" on page 20-203.

For a continuous predictor, a tree can split halfway between any two adjacent unique values found for this predictor. For a categorical predictor with L levels, a classification tree needs to consider $2^{L-1}-1$ splits. To obtain this formula, observe that you can assign L distinct values to the left and right nodes in 2^L ways. Two out of these 2^L configurations would leave either left or right node empty, and therefore should be discarded. Now divide by 2 because left and right can be swapped. A classification tree can thus process only categorical predictors with a moderate number of levels. A regression tree employs a computational shortcut: it sorts the levels by the observed mean response, and considers only the $L-1$ splits between the sorted levels.

Predicting Responses With Classification Trees and Regression Trees

After creating a tree, you can easily predict responses for new data. Suppose `Xnew` is new data that has the same number of columns as the original data `X`. To predict the classification or regression based on the tree and the new data, enter

```
Ynew = predict(tree,Xnew);
```

For each row of data in `Xnew`, `predict` runs through the decisions in `tree` and gives the resulting prediction in the corresponding element of `Ynew`. For more information for classification, see the classification `predict` reference page; for regression, see the regression `predict` reference page.

For example, to find the predicted classification of a point at the mean of the `ionosphere` data:

```
load ionosphere % contains X and Y variables
ctree = ClassificationTree.fit(X,Y);
Ynew = predict(ctree,mean(X))

Ynew =
    'g'
```

To find the predicted `MPG` of a point at the mean of the `carsmall` data:

```
load carsmall % contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = RegressionTree.fit(X,MPG);
```

```
Ynew = predict(rtree,mean(X))

Ynew =
   28.7931
```

Improving Classification Trees and Regression Trees

You can tune trees by setting name-value pairs in `ClassificationTree.fit` and `RegressionTree.fit`. The remainder of this section describes how to determine the quality of a tree, how to decide which name-value pairs to set, and how to control the size of a tree:

Examining Resubstitution Error

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

Example: Resubstitution Error of a Classification Tree. Examine the resubstitution error of a default classification tree for the Fisher iris data:

```
load fisheriris
ctree = ClassificationTree.fit(meas,species);
resuberror = resubLoss(ctree)

resuberror =
    0.0200
```

The tree classifies nearly all the Fisher iris data correctly.

Cross Validation

To get a better sense of the predictive accuracy of your tree for new data, cross validate the tree. By default, cross validation splits the training data into 10 parts at random. It trains 10 new trees, each one on nine parts of the data. It then examines the predictive accuracy of each new tree on the data not included in training that tree. This method gives a good estimate of the predictive accuracy of the resulting tree, since it tests the new trees on new data.

Example: Cross Validating a Regression Tree. Examine the resubstitution and cross-validation accuracy of a regression tree for predicting mileage based on the `carsmall` data:

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
rtree = RegressionTree.fit(X,MPG);
resuberror = resubLoss(rtree)

resuberror =
    4.7188
```

The resubstitution loss for a regression tree is the mean-squared error. The resulting value indicates that a typical predictive error for the tree is about the square root of 4.7, or a bit over 2.

Now calculate the error by cross validating the tree:

```
cvrtree = crossval(rtree);
cvloss = kfoldLoss(cvrtree)

cvloss =
   23.4808
```

The cross-validated loss is almost 25, meaning a typical predictive error for the tree on new data is about 5. This demonstrates that cross-validated loss is usually higher than simple resubstitution loss.

Control Depth or "Leafiness"

When you grow a decision tree, consider its simplicity and predictive power. A deep tree with many leaves is usually highly accurate on the training data.

However, the tree is not guaranteed to show a comparable accuracy on an independent test set. A leafy tree tends to overtrain, and its test accuracy is often far less than its training (resubstitution) accuracy. In contrast, a shallow tree does not attain high training accuracy. But a shallow tree can be more robust — its training accuracy could be close to that of a representative test set. Also, a shallow tree is easy to interpret.

If you do not have enough data for training and test, estimate tree accuracy by cross validation.

For an alternative method of controlling the tree depth, see "Pruning" on page 13-38.

Example: Selecting Appropriate Tree Depth. This example shows how to control the depth of a decision tree, and how to choose an appropriate depth.

1 Load the `ionosphere` data:

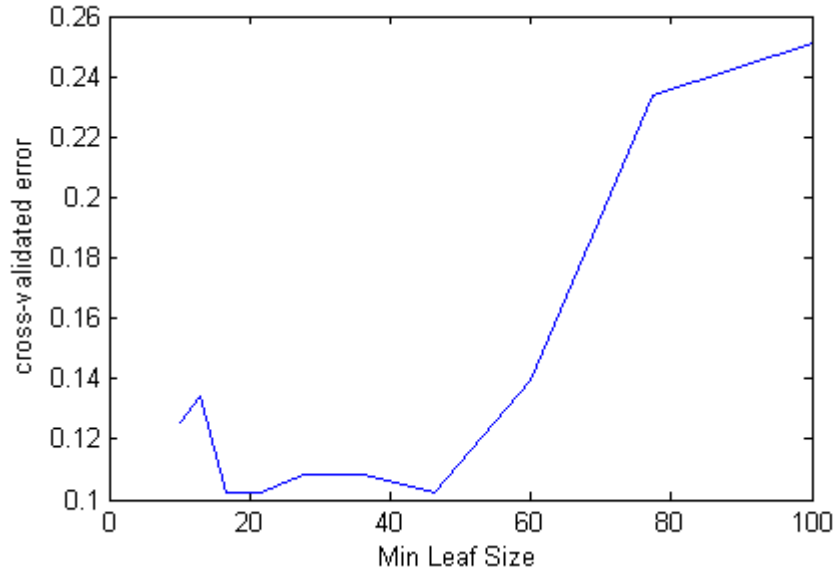
```
load ionosphere
```

2 Generate minimum leaf occupancies for classification trees from `10` to `100`, spaced exponentially apart:

```
leafs = logspace(1,2,10);
```

3 Create cross validated classification trees for the `ionosphere` data with minimum leaf occupancies from `leafs`:

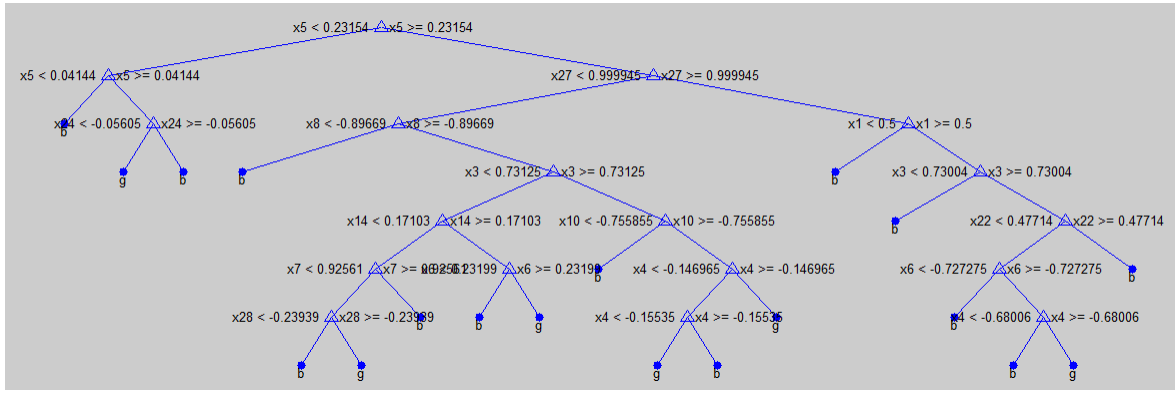
```
N = numel(leafs);
err = zeros(N,1);
for n=1:N
    t = ClassificationTree.fit(X,Y,'crossval','on',...
        'minleaf',leafs(n));
    err(n) = kfoldLoss(t);
end
plot(leafs,err);
xlabel('Min Leaf Size');
ylabel('cross-validated error');
```



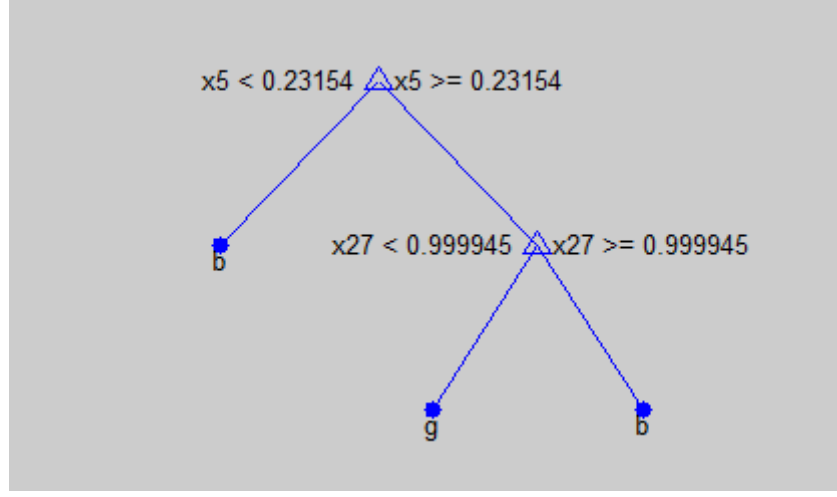
The best leaf size is between about 20 and 50 observations per leaf.

4 Compare the near-optimal tree with at least 40 observations per leaf with the default tree, which uses 10 observations per parent node and 1 observation per leaf.

```
DefaultTree = ClassificationTree.fit(X,Y);
view(DefaultTree,'mode','graph')
```




```
OptimalTree = ClassificationTree.fit(X,Y,'minleaf',40);
view(OptimalTree,'mode','graph')
```




```
resubOpt = resubLoss(OptimalTree);
lossOpt = kfoldLoss(crossval(OptimalTree));
resubDefault = resubLoss(DefaultTree);
lossDefault = kfoldLoss(crossval(DefaultTree));
resubOpt,resubDefault,lossOpt,lossDefault

resubOpt =
    0.0883
```

```
resubDefault =
    0.0114

lossOpt =
    0.1054

lossDefault =
    0.1026
```

The near-optimal tree is much smaller and gives a much higher resubstitution error. Yet it gives similar accuracy for cross-validated data.

Pruning

Pruning optimizes tree depth (leafiness) is by merging leaves on the same tree branch. “Control Depth or “Leafiness”” on page 13-34 describes one method for selecting the optimal depth for a tree. Unlike in that section, you do not need to grow a new tree for every node size. Instead, grow a deep tree, and prune it to the level you choose.

Prune a tree at the command line using the `prune` method (classification) or `prune` method (regression). Alternatively, prune a tree interactively with the tree viewer:

```
view(tree,'mode','graph')
```

To prune a tree, the tree must contain a pruning sequence. By default, both `ClassificationTree.fit` and `RegressionTree.fit` calculate a pruning sequence for a tree during construction. If you construct a tree with the `'Prune'` name-value pair set to `'off'`, or if you prune a tree to a smaller level, the tree does not contain the full pruning sequence. Generate the full pruning sequence with the `prune` method (classification) or `prune` method (regression).

Example: Pruning a Classification Tree. This example creates a classification tree for the `ionosphere` data, and prunes it to a good level.

1 Load the `ionosphere` data:

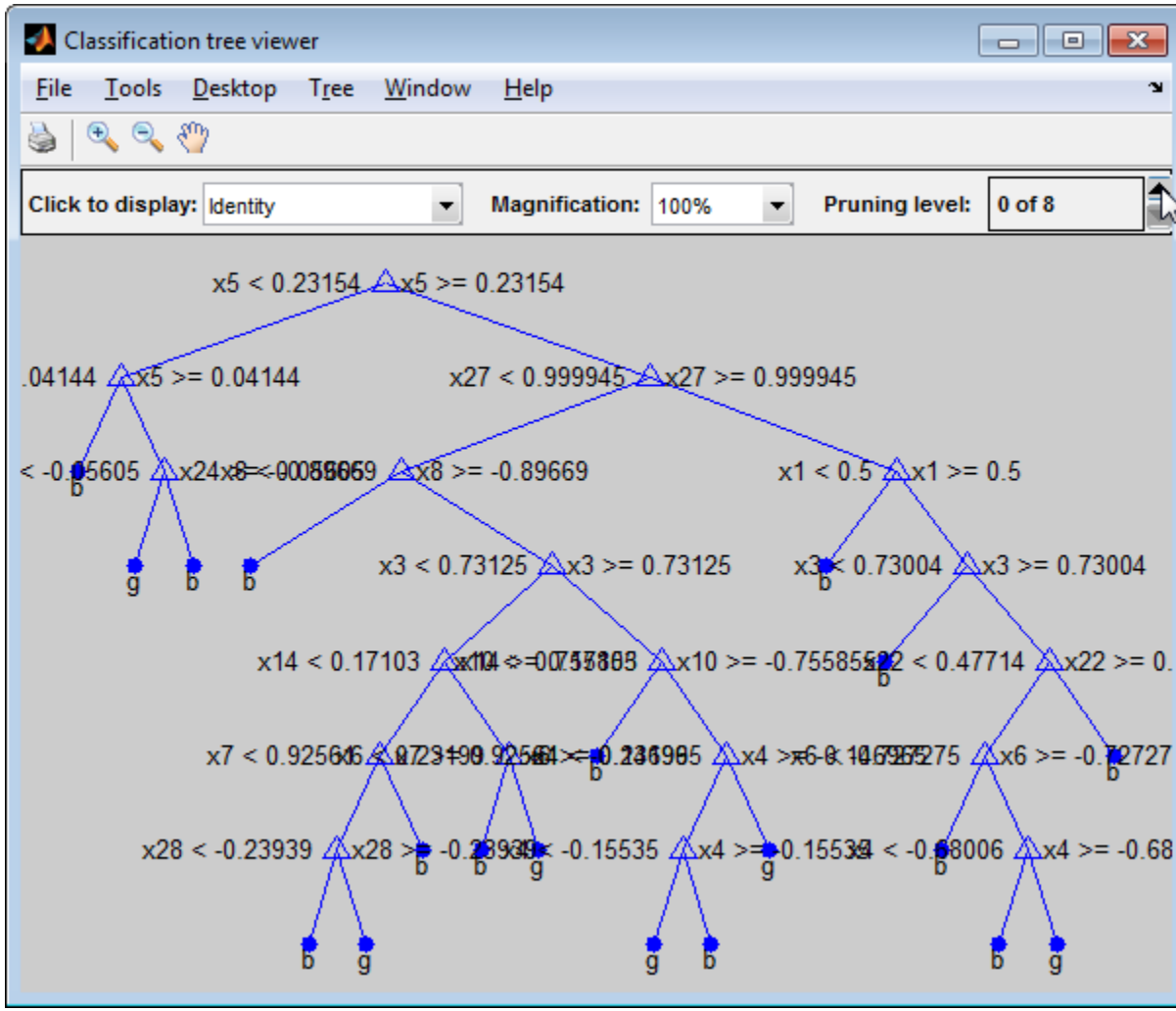
```
load ionosphere
```

2 Construct a default classification tree for the data:

```
tree = ClassificationTree.fit(X,Y);
```

3 View the tree in the interactive viewer:

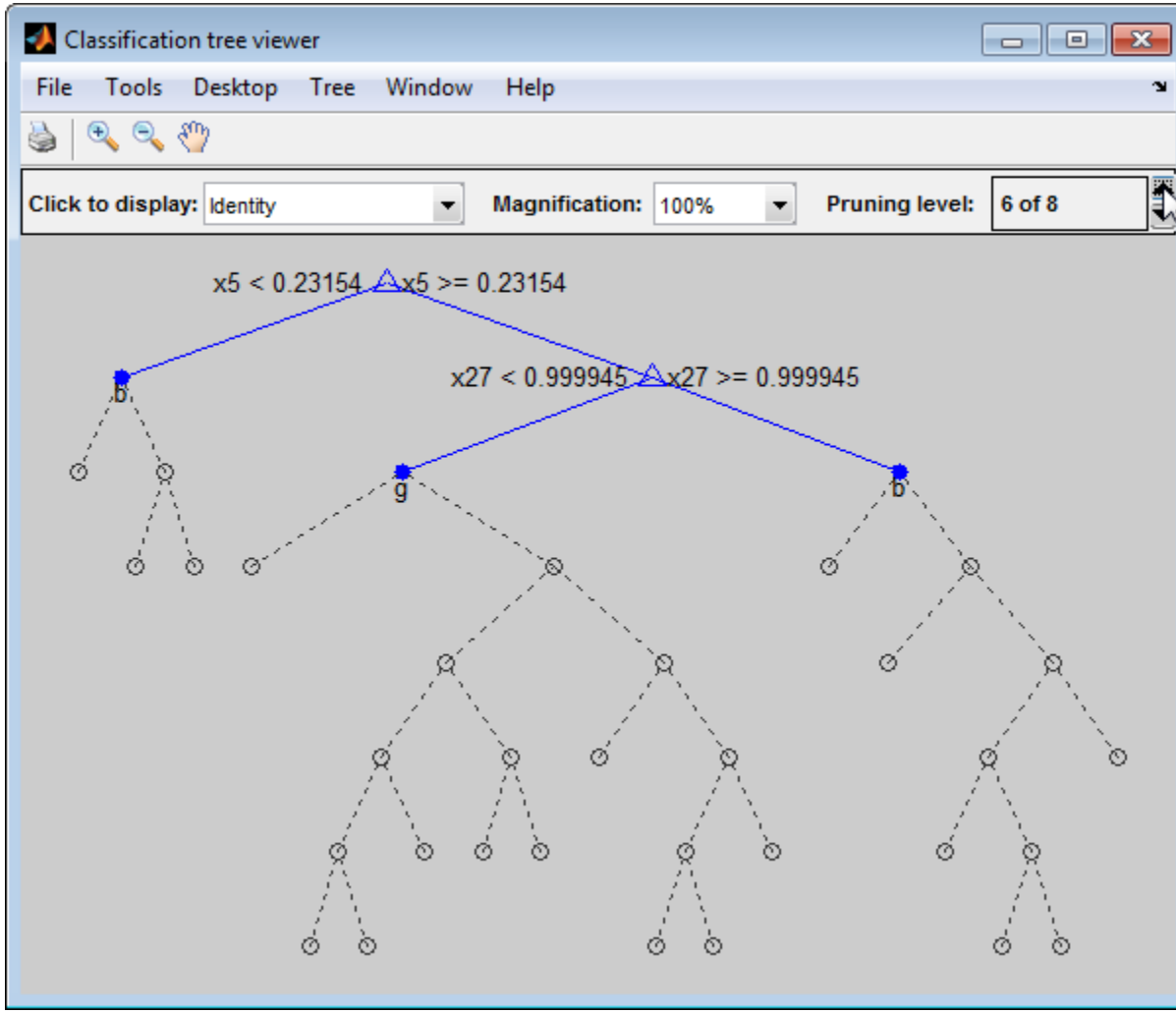
```
view(tree,'mode','graph')
```



4 Find the optimal pruning level by minimizing cross-validated loss:

```
[~,~,~,bestlevel] = cvLoss(tree,...
    'subtrees','all','treesize','min')

bestlevel =
     6
```

5 Prune the tree to level 6 in the interactive viewer:



The pruned tree is the same as the near-optimal tree in “Example: Selecting Appropriate Tree Depth” on page 13-35.

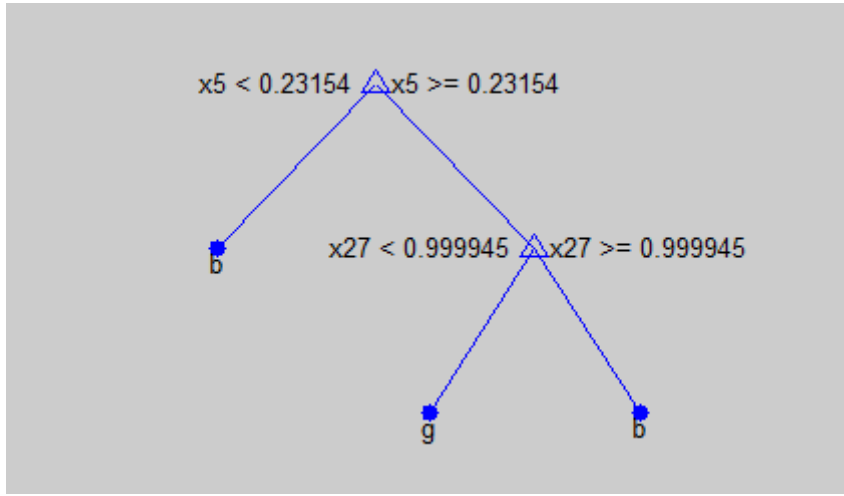
6 Set `'treesize'` to `'se'` (default) to find the maximal pruning level for which the tree error does not exceed the error from the best level plus one standard deviation:

```
[~,~,~,bestlevel] = cvLoss(tree,'subtrees','all')

bestlevel =
     6
```

In this case the level is the same for either setting of `'treesize'`.

7 Prune the tree to use it for other purposes:

```
tree = prune(tree,'Level',6);
view(tree,'mode','graph')
```



Alternative: classregtree

The `ClassificationTree` and `RegressionTree` classes are new in MATLAB R2011a. Previously, you represented both classification trees and regression trees with a `classregtree` object. The new classes provide all the functionality of the `classregtree` class, and are more convenient when used in conjunction with “Ensemble Methods” on page 13-50.

Before the `classregtree` class, there were `treefit`, `treedisp`, `treeval`, `treeprune`, and `treetest` functions. Statistics Toolbox software maintains these only for backward compatibility.

Example: Creating Classification Trees Using classregtree

This example uses Fisher's iris data in `fisheriris.mat` to create a classification tree for predicting species using measurements of sepal length, sepal width, petal length, and petal width as predictors. Here, the predictors are continuous and the response is categorical.

1. Load the data and use the `classregtree` constructor of the `classregtree` class to create the classification tree:

```
load fisheriris

t = classregtree(meas,species,...
                 'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

   `t` is a `classregtree` object and can be operated on with any class method.

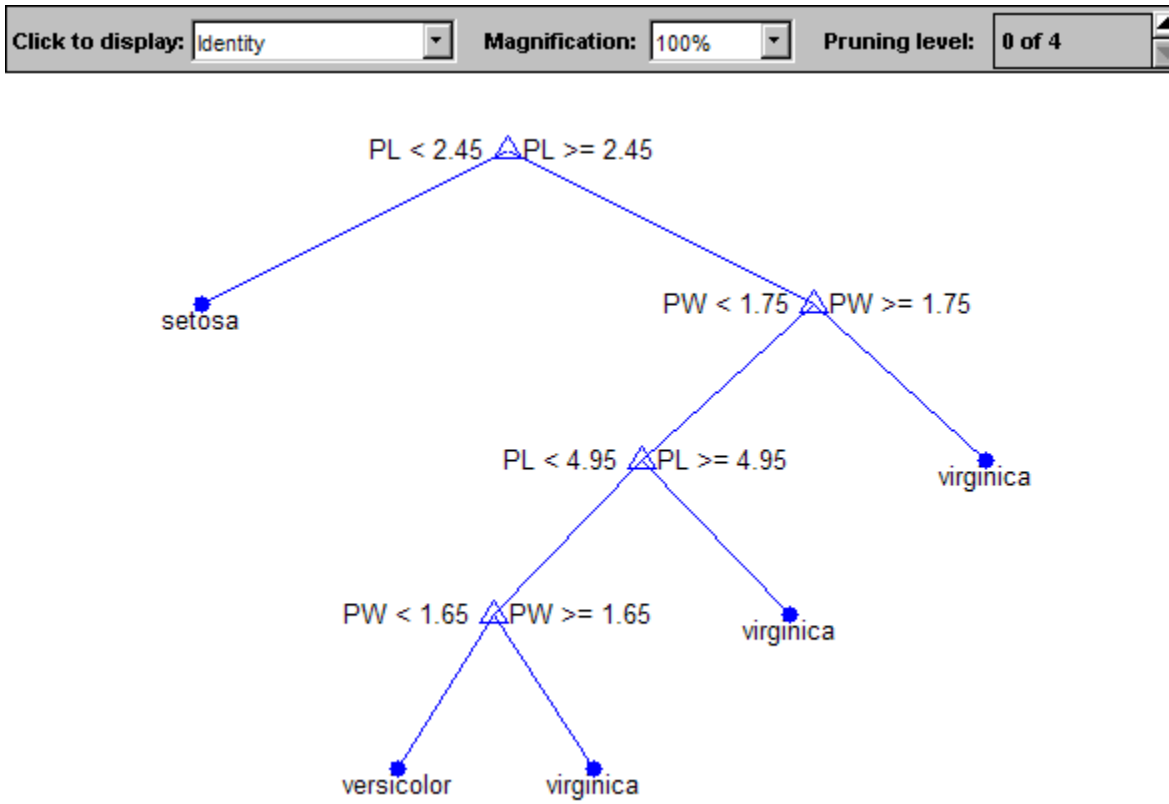
2. Use the `type` method of the `classregtree` class to show the type of the tree:

```
treetype = type(t)
treetype =
classification
```

   `classregtree` creates a classification tree because `species` is a cell array of strings, and the response is assumed to be categorical.

3. To view the tree, use the `view` method of the `classregtree` class:

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the iris at the triangular branching nodes. A `true` answer to any question follows the branch to the left. A `false` follows the branch to the right.

4 The tree does not use sepal measurements for predicting species. These can go unmeasured in new data, and you can enter them as `NaN` values for predictions. For example, to use the tree to predict the species of an iris with petal length `4.8` and petal width `1.6`, type:

```
predicted = t([NaN NaN 4.8 1.6])
predicted =
    'versicolor'
```

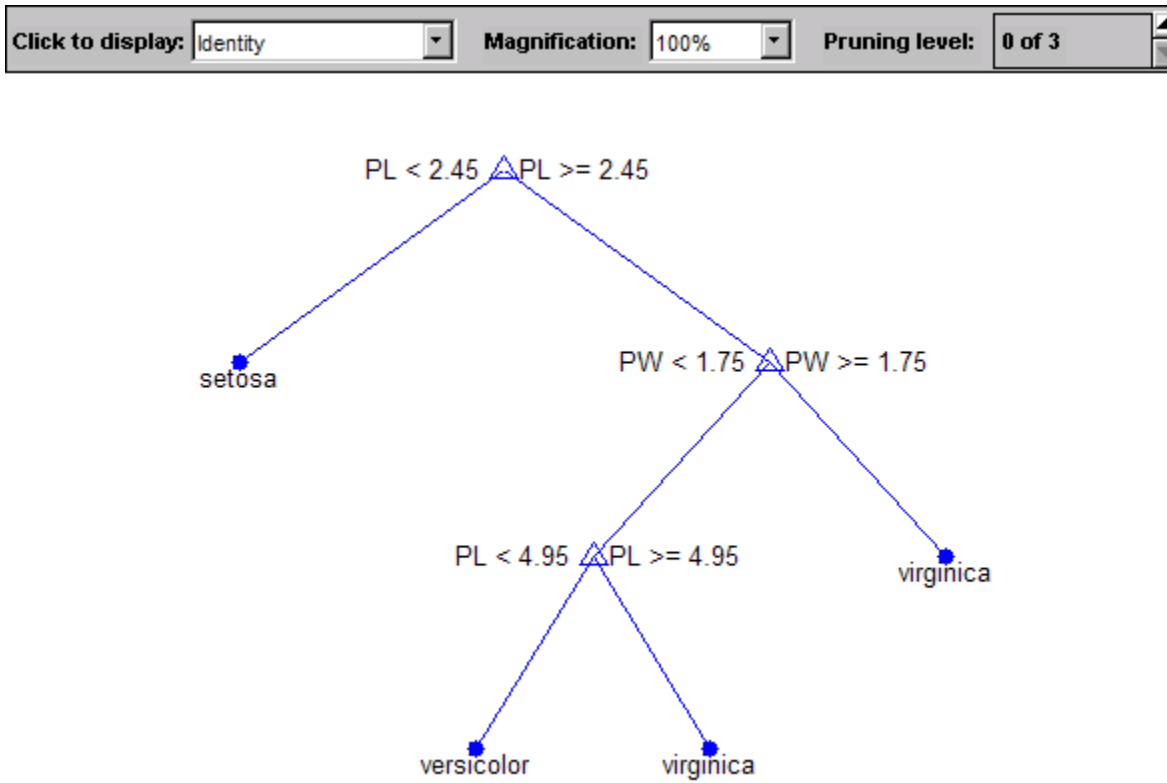
The object allows for functional evaluation, of the form `t(X)`. This is a shorthand way of calling the `eval` method of the `classregtree` class. The predicted species is the left leaf node at the bottom of the tree in the previous view.

5 You can use a variety of methods of the `classregtree` class, such as `cutvar` and `cuttype` to get more information about the split at node 6 that makes the final distinction between `versicolor` and `virginica`:

```
var6 = cutvar(t,6) % What variable determines the split?
var6 =
    'PW'

type6 = cuttype(t,6) % What type of split is it?
type6 =
    'continuous'
```

6 Classification trees fit the original (training) data well, but can do a poor job of classifying new values. Lower branches, especially, can be strongly affected by outliers. A simpler tree often avoids overfitting. You can use the `prune` method of the `classregtree` class to find the next largest tree from an optimal pruning sequence:

```
pruned = prune(t,'level',1)
pruned =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  class = versicolor
7  class = virginica

view(pruned)
```



To find the best classification tree, employing the techniques of resubstitution and cross validation, use the `test` method of the `classregtree` class.

Example: Creating Regression Trees Using classregtree

This example uses the data on cars in `carsmall.mat` to create a regression tree for predicting mileage using measurements of weight and the number of cylinders as predictors. Here, one predictor (weight) is continuous and the other (cylinders) is categorical. The response (mileage) is continuous.

1 Load the data and use the `classregtree` constructor of the `classregtree` class to create the regression tree:

```
load carsmall

t = classregtree([Weight, Cylinders],MPG,...
                 'cat',2,'splitmin',20,...
                 'names',{'W','C'})

t =

Decision tree for regression
 1  if W<3085.5 then node 2 elseif W>=3085.5 then node 3 else 23.7181
 2  if W<2371 then node 4 elseif W>=2371 then node 5 else 28.7931
 3  if C=8 then node 6 elseif C in {4 6} then node 7 else 15.5417
 4  if W<2162 then node 8 elseif W>=2162 then node 9 else 32.0741
 5  if C=6 then node 10 elseif C=4 then node 11 else 25.9355
 6  if W<4381 then node 12 elseif W>=4381 then node 13 else 14.2963
 7  fit = 19.2778
 8  fit = 33.3056
 9  fit = 29.6111
10  fit = 23.25
11  if W<2827.5 then node 14 elseif W>=2827.5 then node 15 else 27.2143
12  if W<3533.5 then node 16 elseif W>=3533.5 then node 17 else 14.8696
13  fit = 11
14  fit = 27.6389
15  fit = 24.6667
16  fit = 16.6
17  fit = 14.3889
```

`t` is a `classregtree` object and can be operated on with any of the methods of the class.

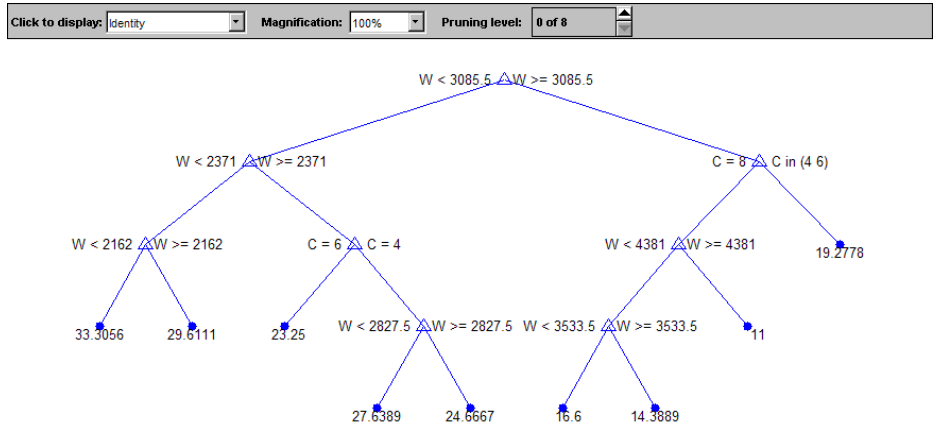
2 Use the `type` method of the `classregtree` class to show the type of the tree:

```
treetype = type(t)
treetype =
regression
```

`classregtree` creates a regression tree because `MPG` is a numerical vector, and the response is assumed to be continuous.

3 To view the tree, use the `view` method of the `classregtree` class:

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the car at the triangular branching nodes. A `true` answer to any question follows the branch to the left; a `false` follows the branch to the right.

4 Use the tree to predict the mileage for a 2000-pound car with either 4, 6, or 8 cylinders:

```
mileage2K = t([2000 4; 2000 6; 2000 8])
mileage2K =
   33.3056
   33.3056
   33.3056
```

The object allows for functional evaluation, of the form `t(X)`. This is a shorthand way of calling the `eval` method of the `classregtree` class.

5 The predicted responses computed above are all the same. This is because they follow a series of splits in the tree that depend only on weight, terminating at the left-most leaf node in the view above. A 4000-pound car, following the right branch from the top of the tree, leads to different predicted responses:

```
mileage4K = t([4000 4; 4000 6; 4000 8])
```

```
mileage4K =
   19.2778
   19.2778
   14.3889
```

6 You can use a variety of other methods of the `classregtree` class, such as `cutvar`, `cuttype`, and `cutcategories`, to get more information about the split at node 3 that distinguishes the 8-cylinder car:

```
var3 = cutvar(t,3) % What variable determines the split?
var3 =
    'C'

type3 = cuttype(t,3) % What type of split is it?
type3 =
    'categorical'

c = cutcategories(t,3) % Which classes are sent to the left
                       % child node, and which to the right?
c =
    [8]    [1x2 double]
c{1}
ans =
     8
c{2}
ans =
     4     6
```

Regression trees fit the original (training) data well, but may do a poor job of predicting new values. Lower branches, especially, may be strongly affected by outliers. A simpler tree often avoids over-fitting. To find the best regression tree, employing the techniques of resubstitution and cross validation, use the `test` method of the `classregtree` class.

Ensemble Methods

In this section...

Framework for Ensemble Learning

You have several methods for melding results from many weak learners into one high-quality ensemble predictor. These methods follow, as closely as possible, the same syntax, so you can try different methods with only minor changes in your commands.

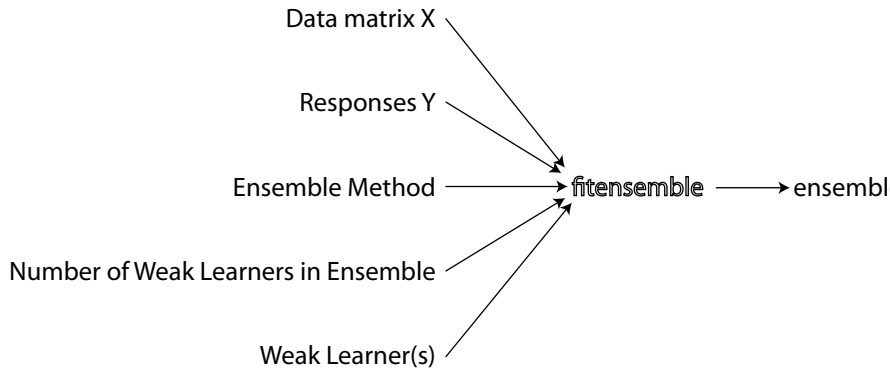
Create an ensemble with the `fitensemble` function. The syntax of `fitensemble` is

```
ens = fitensemble(X,Y,model,numberens,learners)
```

- `X` is the matrix of data, each row containing one observation, each column contains one predictor variable.
- `Y` is the responses, with the same number of observations as rows in `X`.
- `model` is a string naming the type of ensemble.
- `numberens` is the number of weak learners in `ens` from each element of `learners`. So the number of elements in `ens` is `numberens` times the number of elements in `learners`.

- `learners` is a string naming a weak learner, is a weak learner template, or is a cell array of such templates.

Pictorially, here is the information you need to create an ensemble:



For all classification or nonlinear regression problems, follow these steps to create an ensemble:

1 "Put Predictor Data in a Matrix" on page 13-51

2 "Prepare Response Data" on page 13-52

3 "Choose an Applicable Ensemble Method" on page 13-53

4 "Set the Number of Ensemble Members" on page 13-54

5 "Prepare the Weak Learners" on page 13-54

6 "Call fitensemble" on page 13-55

Put Predictor Data in a Matrix

All supervised learning methods start with a data matrix, usually called `X` in this documentation. Each row of `X` represents one observation. Each column of `X` represents one variable, or predictor.

Currently, you can use only decision trees as learners for ensembles. Decision trees can handle `NaN` values in `X`. Such values are called "missing." If you have

some missing values in a row of X, a decision tree finds optimal splits using nonmissing values only. If an entire row consists of NaN, fitensemble ignores that row. If you have data with a large fraction of missing values in X, use surrogate decision splits. For examples of surrogate splits, see "Example: Unequal Classification Costs" on page 13-66 and "Example: Surrogate Splits" on page 13-76.

Prepare Response Data

You can use a wide variety of data types for response data.

- For regression ensembles, Y must be a numeric vector with the same number of elements as the number of rows of X.
- For classification ensembles, Y can be any of the following data types. The table also contains the method of including missing entries.

Data Type	Missing Entry
Numeric vector	NaN
Categorical vector	<undefined>
Character array	Row of spaces
Cell array of strings	''
Logical vector	(not possible to represent)

fitensemble ignores missing values in Y when creating an ensemble.

For example, suppose your response data consists of three observations in the following order: true, false, true. You could express Y as:

- [1;0;1] (numeric vector)
- nominal({'true','false','true'}) (categorical vector)
- [true;false;true] (logical vector)
- ['true ';'false';'true '] (character array, padded with spaces so each row has the same length)
- {'true','false','true'} (cell array of strings)

Use whichever data type is most convenient. Since you cannot represent missing values with logical entries, do not use logical entries when you have missing values in `Y`.

Choose an Applicable Ensemble Method

`fitensemble` uses one of these algorithms to create an ensemble.

- For classification with two classes:
  - `'AdaBoostM1'`
  - `'LogitBoost'`
  - `'GentleBoost'`
  - `'RobustBoost'`
  - `'Bag'`
- For classification with three or more classes:
  - `'AdaBoostM2'`
  - `'Bag'`
- For regression:
  - `'LSBoost'`
  - `'Bag'`

Since `'Bag'` applies to all methods, indicate whether you want a classifier or regressor with the `type` name-value pair set to `'classification'` or `'regression'`.

For descriptions of the various algorithms, and aid in choosing which applies to your data, see “Ensemble Algorithms” on page 13-118. The following table gives characteristics of the various algorithms. In the table titles:

- `Regress.` — Regression
- `Classif.` — Classification
- `Preds.` — Predictors
- `Estim.` — Estimate

- `Gen.` — Generalization
- `Pred.` — Prediction
- `Mem.` — Memory usage

Algorithm	Regress.	Binary Classif.	Binary Classif. Multi-Level Preds.	Classif. 3+ Classes	Auto Estim. Gen. Error	Fast Train	Fast Pred.	Low Mem.
`Bag`	×	×		×	×			
`AdaBoostM1`		×				×	×	×
`AdaBoostM2`				×		×	×	×
`LogitBoost`		×	×			×	×	×
`GentleBoost`		×	×			×	×	×
`RobustBoost`		×					×	×
`LSBoost`	×					×	×	×

Set the Number of Ensemble Members

Choosing the size of an ensemble involves balancing speed and accuracy.

- Larger ensembles take longer to train and to generate predictions.
- Some ensemble algorithms can become overtrained (inaccurate) when too large.

To set an appropriate size, consider starting with several dozen to several hundred members in an ensemble, training the ensemble, and then checking the ensemble quality, as in "Example: Test Ensemble Quality" on page 13-59. If it appears that you need more members, add them using the `resume` method (classification) or the `resume` method (regression). Repeat until adding more members does not improve ensemble quality.

Prepare the Weak Learners

Currently there is one built-in weak learner type: `'Tree'`. To create an ensemble with the default tree options, pass in `'Tree'` as the weak learner.

To set a nondefault classification tree learner, create a classification tree template with the `ClassificationTree.template` method.

Similarly, to set a nondefault regression tree learner, create a regression tree template with the `RegressionTree.template` method.

While you can give `fitensemble` a cell array of learner templates, the most common usage is to give just one weak learner template.

For examples using a template, see "Example: Unequal Classification Costs" on page 13-66 and "Example: Surrogate Splits" on page 13-76.

Common Settings for Weak Learners.

- The depth of the weak learner tree makes a difference for training time, memory usage, and predictive accuracy. You control the depth with two parameters:
  - `MinLeaf` — Each leaf has at least `MinLeaf` observations. Set small values of `MinLeaf` to get a deep tree.
  - `MinParent` — Each branch node in the tree has at least `MinParent` observations. Set small values of `MinParent` to get a deep tree.

  If you supply both `MinParent` and `MinLeaf`, the learner uses the setting that gives larger leaves:

  ```
  MinParent = max(MinParent,2*MinLeaf)
  ```

- `Surrogate` — Grow decision trees with surrogate splits when `Surrogate` is `'on'`. Use surrogate splits when your data has missing values.

  Note Surrogate splits cause training to be slower and use more memory.

Call fitensemble

The syntax of `fitensemble` is

```
ens = fitensemble(X,Y,model,numberens,learners)
```

- `X` is the matrix of data. Each row contains one observation, and each column contains one predictor variable.
- `Y` is the responses, with the same number of observations as rows in `X`.
- `model` is a string naming the type of ensemble.
- `numberens` is the number of weak learners in `ens` from each element of `learners`. So the number of elements in `ens` is `numberens` times the number of elements in `learners`.
- `learners` is a string naming a weak learner, a weak learner template, or a cell array of such strings and templates.

The result of `fitensemble` is an ensemble object, suitable for making predictions on new data. For a basic example of creating a classification ensemble, see "Creating a Classification Ensemble" on page 13-57. For a basic example of creating a regression ensemble, see "Creating a Regression Ensemble" on page 13-58.

Where to Set Name-Value Pairs. There are several name-value pairs you can pass to `fitensemble`, and several that apply to the weak learners (`ClassificationTree.template` and `RegressionTree.template`). To determine which option (name-value pair) is appropriate, the ensemble or the weak learner:

- Use template name-value pairs to control the characteristics of the weak learners.
- Use `fitensemble` name-value pairs to control the ensemble as a whole, either for algorithms or for structure.

For example, to have an ensemble of boosted classification trees with each tree deeper than the default, set the `ClassificationTree.template` name-value pairs (`MinLeaf` and `MinParent`) to smaller values than the defaults. This causes the trees to be leafier (deeper).

To name the predictors in the ensemble (part of the structure of the ensemble), use the `PredictorNames` name-value pair in `fitensemble`.

Basic Ensemble Examples

Creating a Classification Ensemble

Create a classification ensemble for the Fisher iris data, and use it to predict the classification of a flower with average measurements.

1 Load the data:

```
load fisheriris
```

2 The predictor data `X` is the `meas` matrix.

3 The response data `Y` is the `species` cell array.

4 The only boosted classification ensemble for three or more classes is `'AdaBoostM2'`.

5 For this example, arbitrarily take an ensemble of `100` trees.

6 Use a default tree template.

7 Create the ensemble:

```
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree')

ens =
classreg.learning.classif.ClassificationEnsemble:
           PredictorNames: {'x1'  'x2'  'x3'  'x4'}
    CategoricalPredictors: []
             ResponseName: 'Y'
                ClassNames: {'setosa'  'versicolor'  'virginica'}
           ScoreTransform: 'none'
            NObservations: 150
                 NTrained: 100
                   Method: 'AdaBoostM2'
            LearnerNames: {'Tree'}
     ReasonForTermination: [1x77 char]
                  FitInfo: [100x1 double]
       FitInfoDescription: [2x83 char]
```

8 Predict the classification of a flower with average measurements:

```
flower = predict(ens,mean(meas))

flower =
    'versicolor'
```

Creating a Regression Ensemble

Create a regression ensemble to predict mileage of cars based on their horsepower and weight, trained on the `carsmall` data. Use the resulting ensemble to predict the mileage of a car with 150 horsepower weighing 2750 lbs.

1 Load the data:

```
load carsmall
```

2 Prepare the input data.

```
X = [Horsepower Weight];
```

3 The response data `Y` is `MPG`.

4 The only boosted regression ensemble type is `'LSBoost'`.

5 For this example, arbitrarily take an ensemble of `100` trees.

6 Use a default tree template.

7 Create the ensemble:

```
ens = fitensemble(X,MPG,'LSBoost',100,'Tree')

ens =
classreg.learning.regr.RegressionEnsemble:
           PredictorNames: {'x1'  'x2'}
    CategoricalPredictors: []
             ResponseName: 'Y'
        ResponseTransform: 'none'
            NObservations: 94
                 NTrained: 100
                   Method: 'LSBoost'
             LearnerNames: {'Tree'}
```

```
    ReasonForTermination: [1x77 char]
                 FitInfo: [100x1 double]
       FitInfoDescription: [2x83 char]
           Regularization: []
```

8 Predict the mileage of a car with 150 horsepower weighing 2750 lbs:

```
mileage = ens.predict([150 2750])

mileage =
   22.6735
```

Test Ensemble Quality

Usually you cannot evaluate the predictive quality of an ensemble based on its performance on training data. Ensembles tend to "overtrain," meaning they produce overly optimistic estimates of their predictive power. This means the result of `resubLoss` for classification (`resubLoss` for regression) usually indicates lower error than you get on new data.

To obtain a better idea of the quality of an ensemble, use one of these methods:

- Evaluate the ensemble on an independent test set (useful when you have a lot of training data).
- Evaluate the ensemble by cross validation (useful when you don't have a lot of training data).
- Evaluate the ensemble on out-of-bag data (useful when you create a bagged ensemble with `fitensemble`).

Example: Test Ensemble Quality

This example uses a bagged ensemble so it can use all three methods of evaluating ensemble quality.

1 Generate an artificial dataset with 20 predictors. Each entry is a random number from 0 to 1. The initial classification:

$Y = 1$ when $X(1) + X(2) + X(3) + X(4) + X(5) > 2.5$
$Y = 0$ otherwise.

```
rng(1,'twister') % for reproducibility
X = rand(2000,20);
Y = sum(X(:,1:5),2) > 2.5;
```

In addition, to add noise to the results, randomly switch 10% of the classifications:

```
idx = randsample(2000,200);
Y(idx) = ~Y(idx);
```

2 Independent Test Set

Create independent training and test sets of data. Use 70% of the data for a training set by calling `cvpartition` with the `holdout` option:

```
cvpart = cvpartition(Y,'holdout',0.3);
Xtrain = X(training(cvpart),:);
Ytrain = Y(training(cvpart),:);
Xtest = X(test(cvpart),:);
Ytest = Y(test(cvpart),:);
```

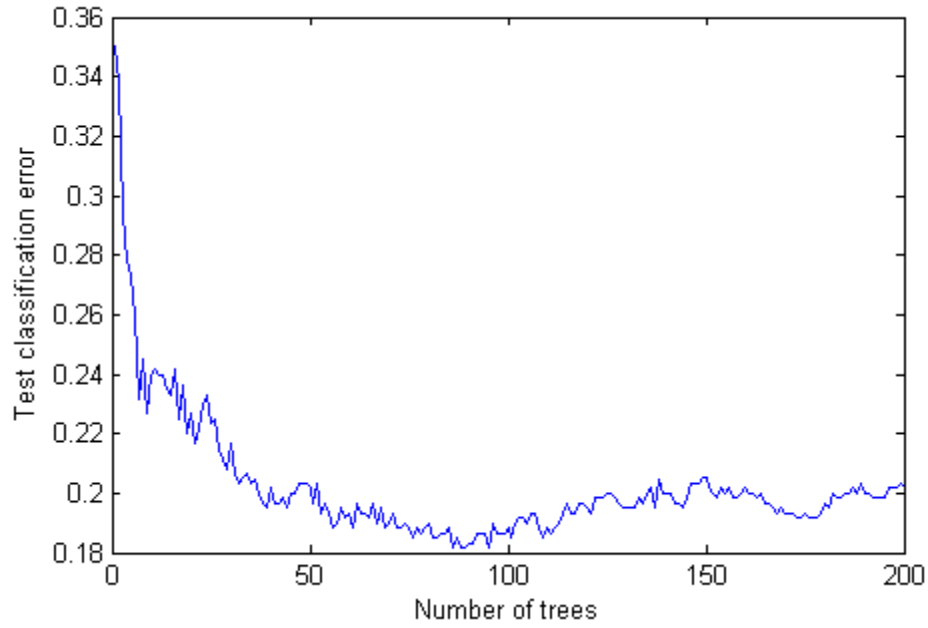
3 Create a bagged classification ensemble of 200 trees from the training data:

```
bag = fitensemble(Xtrain,Ytrain,'Bag',200,'Tree',...
    'type','classification')

bag =

classreg.learning.classif.ClassificationBaggedEnsemble:
           PredictorNames: {1x20 cell}
    CategoricalPredictors: []
             ResponseName: 'Y'
                ClassNames: [0 1]
           ScoreTransform: 'none'
            NObservations: 1400
                 NTrained: 200
                   Method: 'Bag'
             LearnerNames: {'Tree'}
     ReasonForTermination: [1x77 char]
                  FitInfo: []
       FitInfoDescription: 'None'
```

```
                FResample: 1
                  Replace: 1
       UseObsForLearner: [1400x200 logical]
```

4 Plot the loss (misclassification) of the test data as a function of the number of trained trees in the ensemble:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
xlabel('Number of trees');
ylabel('Test classification error');
```



5 Cross validation

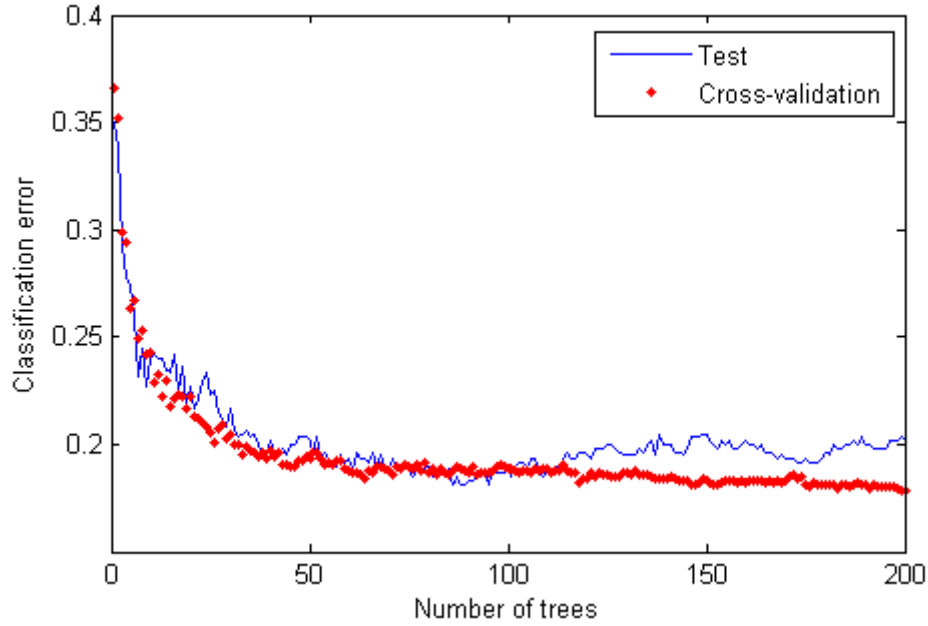
Generate a five-fold cross-validated bagged ensemble:

```
cv = fitensemble(X,Y,'Bag',200,'Tree',...
    'type','classification','kfold',5)
```

```
cv =

classreg.learning.partition.ClassificationPartitionedEnsemble:
      CrossValidatedModel: 'Bag'
           PredictorNames: {1x20 cell}
    CategoricalPredictors: []
             ResponseName: 'Y'
            NObservations: 2000
                    KFold: 5
                Partition: [1x1 cvpartition]
           NTrainedPerFold: [200 200 200 200 200]
               ClassNames: [0 1]
           ScoreTransform: 'none'
```

6 Examine the cross validation loss as a function of the number of trees in the ensemble:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
hold
plot(kfoldLoss(cv,'mode','cumulative'),'r.');
hold off;
xlabel('Number of trees');
ylabel('Classification error');
legend('Test','Cross-validation','Location','NE');
```

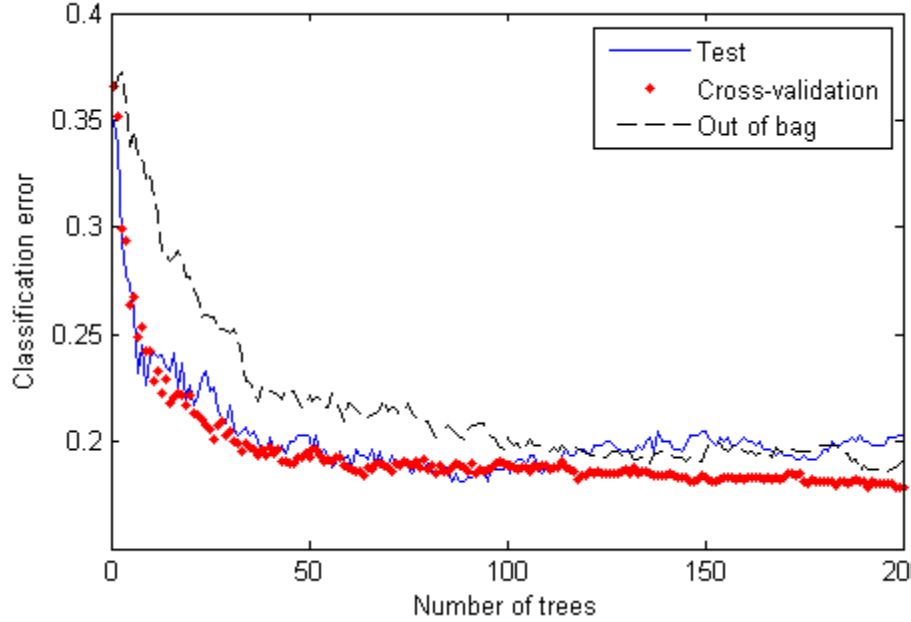


Cross validating gives comparable estimates to those of the independent set.

7 Out-of-Bag Estimates

Generate the loss curve for out-of-bag estimates, and plot it along with the other curves:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
hold
plot(kfoldLoss(cv,'mode','cumulative'),'r.');
plot(oobLoss(bag,'mode','cumulative'),'k--');
hold off;
xlabel('Number of trees');
ylabel('Classification error');
legend('Test','Cross-validation','Out of bag','Location','NE');
```



The out-of-bag estimates are again comparable to those of the other methods.

Classification: Imbalanced Data or Unequal Misclassification Costs

In many real-world applications, you might prefer to treat classes in your data asymmetrically. For example, you might have data with many more observations of one class than of any other. Or you might work on a problem in which misclassifying observations of one class has more severe consequences than misclassifying observations of another class. In such situations, you can use two optional parameters for `fitensemble`: `prior` and `cost`.

By using `prior`, you set prior class probabilities (that is, class probabilities used for training). Use this option if some classes are under- or overrepresented in your training set. For example, you might obtain your training data by simulation. Because simulating class `A` is more expensive than class `B`, you opt to generate fewer observations of class `A` and more

observations of class `B`. You expect, however, that class `A` and class `B` are mixed in a different proportion in the real world. In this case, set prior probabilities for class `A` and `B` approximately to the values you expect to observe in the real world. `fitensemble` normalizes prior probabilities to make them add up to `1`; multiplying all prior probabilities by the same positive factor does not affect the result of classification.

If classes are adequately represented in the training data but you want to treat them asymmetrically, use the `cost` parameter. Suppose you want to classify benign and malignant tumors in cancer patients. Failure to identify a malignant tumor (false negative) has far more severe consequences than misidentifying benign as malignant (false positive). You should assign high cost to misidentifying malignant as benign and low cost to misidentifying benign as malignant.

You must pass misclassification costs as a square matrix with nonnegative elements. Element `C(i,j)` of this matrix is the cost of classifying an observation into class `j` if the true class is `i`. The diagonal elements `C(i,i)` of the cost matrix must be `0`. For the example above, you can choose malignant tumor to be class 1 and benign tumor to be class 2. Then you can set the cost matrix to

$$\begin{bmatrix} 0 & c \\ 1 & 0 \end{bmatrix}$$

where $c > 1$ is the cost of misidentifying a malignant tumor as benign. Costs are relative—multiplying all costs by the same positive factor does not affect the result of classification.

If you have only two classes, `fitensemble` adjusts their prior probabilities using $\tilde{P}_i = C_{ij} P_i$ for class $i = 1,2$ and $j \neq i$. P_i are prior probabilities either passed into `fitensemble` or computed from class frequencies in the training data, and $\tilde{P}_i$ are adjusted prior probabilities. Then `fitensemble` uses the default cost matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and these adjusted probabilities for training its weak learners. Manipulating the cost matrix is thus equivalent to manipulating the prior probabilities.

If you have three or more classes, `fitensemble` also converts input costs into adjusted prior probabilities. This conversion is more complex. First, `fitensemble` attempts to solve a matrix equation described in Zhou and Liu [15]. If it fails to find a solution, `fitensemble` applies the "average cost" adjustment described in Breiman et al. [5]. For more information, see .Zadrozny, Langford, and Abe [14]

Example: Unequal Classification Costs

This example uses data on patients with hepatitis to see if they live or die as a result of the disease. The data is described at `http://archive.ics.uci.edu/ml/datasets/Hepatitis`.

1. Load the data into a file named `hepatitis.txt`:

```
s = urlread(['http://archive.ics.uci.edu/ml/' ...
  'machine-learning-databases/hepatitis/hepatitis.data']);
fid = fopen('hepatitis.txt','w');
fwrite(fid,s);
fclose(fid);
```

2. Load the data `hepatitis.txt` into a dataset, with variable names describing the fields in the data:

```
VarNames = {'die_or_live' 'age' 'sex' 'steroid' 'antivirals' 'fatigue' ...
    'malaise' 'anorexia' 'liver_big' 'liver_firm' 'spleen_palpable' ...
    'spiders' 'ascites' 'varices' 'bilirubin' 'alk_phosphate' 'sgot' ...
    'albumin' 'protime' 'histology'};
ds = dataset('file','hepatitis.txt','VarNames',VarNames,...
    'Delimiter',',','ReadVarNames',false,'TreatAsEmpty','?',...
    'Format','%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f');
```

`ds` is a dataset with 155 observations and 20 variables:

```
size(ds)

ans =
```

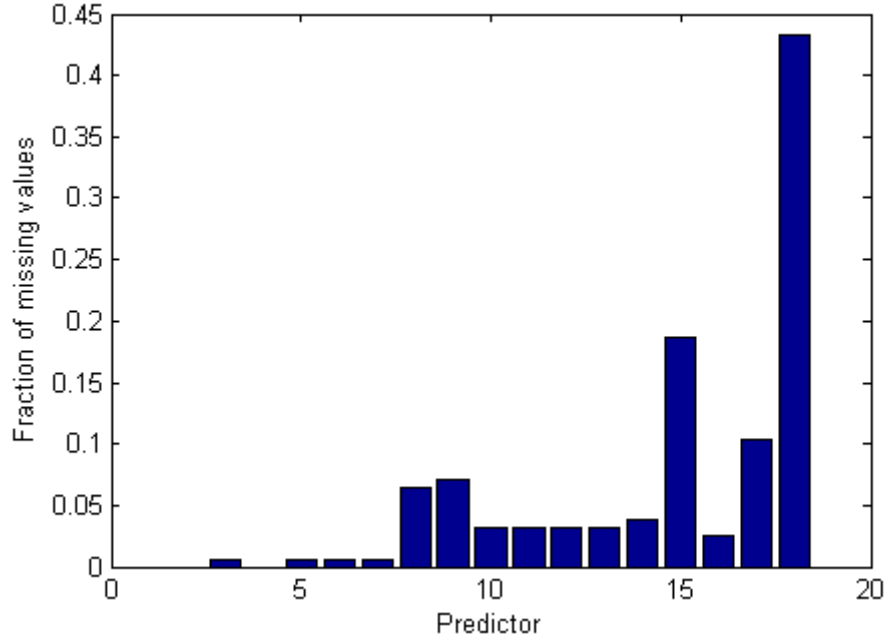
```
    155     20
```

3 Convert the data in the dataset to the format for ensembles: a numeric matrix of predictors, and a cell array with outcome names: `'Die'` or `'Live'`. The first field in the dataset has the outcomes.

```
X = double(ds(:,2:end));
ClassNames = {'Die' 'Live'};
Y = ClassNames(ds.die_or_live);
```

4 Inspect the data for missing values:

```
figure;
bar(sum(isnan(X),1)/size(X,1));
xlabel('Predictor');
ylabel('Fraction of missing values');
```



Most predictors have missing values, and one has nearly 45% of missing values. Therefore, use decision trees with surrogate splits for better accuracy. Because the dataset is small, training time with surrogate splits should be tolerable.

5 Create a classification tree template that uses surrogate splits:

```
rng(0,'twister') % for reproducibility
t = ClassificationTree.template('surrogate','on');
```

6 Examine the data or the description of the data to see which predictors are categorical:

```
X(1:5,:)

ans =

  Columns 1 through 6

   30.0000    2.0000    1.0000    2.0000    2.0000    2.0000
   50.0000    1.0000    1.0000    2.0000    1.0000    2.0000
   78.0000    1.0000    2.0000    2.0000    1.0000    2.0000
   31.0000    1.0000       NaN    1.0000    2.0000    2.0000
   34.0000    1.0000    2.0000    2.0000    2.0000    2.0000

  Columns 7 through 12

    2.0000    1.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    1.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000

  Columns 13 through 18

    2.0000    1.0000   85.0000   18.0000    4.0000       NaN
    2.0000    0.9000  135.0000   42.0000    3.5000       NaN
    2.0000    0.7000   96.0000   32.0000    4.0000       NaN
    2.0000    0.7000   46.0000   52.0000    4.0000   80.0000
    2.0000    1.0000       NaN  200.0000    4.0000       NaN
```

```
  Column 19

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

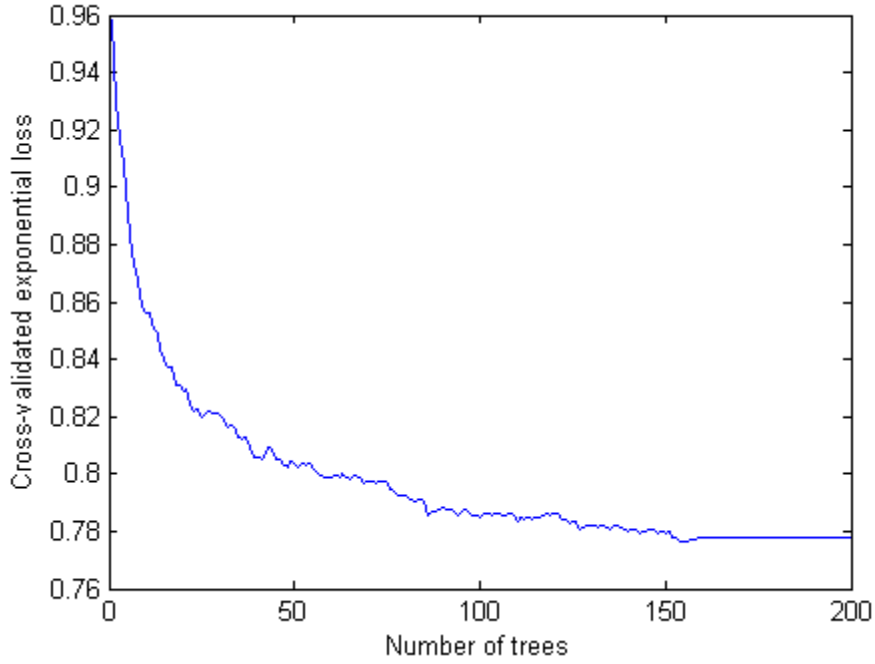
It appears that predictors 2 through 13 are categorical, as well as predictor 19. You can confirm this inference with the dataset description at http://archive.ics.uci.edu/ml/datasets/Hepatitis.

7 List the categorical variables:

```
ncat = [2:13,19];
```

8 Create a cross-validated ensemble using 200 learners and the GentleBoost algorithm:

```
a = fitensemble(X,Y,'GentleBoost',200,t,...
  'PredictorNames',VarNames(2:end),'LearnRate',0.1,...
  'CategoricalPredictors',ncat,'kfold',5);
figure;
plot(kfoldLoss(a,'mode','cumulative','lossfun','exponential'));
xlabel('Number of trees');
ylabel('Cross-validated exponential loss');
```



9 Inspect the confusion matrix to see which people the ensemble predicts correctly:

```
[Yfit,Sfit] = kfoldPredict(a); %
confusionmat(Y,Yfit,'order',ClassNames)

ans =
    16    16
    10   113
```

Of the 123 people who live, the ensemble predicts correctly that 113 will live. But for the 32 people who die of hepatitis, the ensemble only predicts correctly that half will die of hepatitis.

10 There are two types of error in the predictions of the ensemble:

- Predicting that the patient lives, but the patient dies

- Predicting that the patient dies, but the patient lives

Suppose you believe that the first error is five times worse than the second. Make a new classification cost matrix that reflects this belief:

```
cost.ClassNames = ClassNames;
cost.ClassificationCosts = [0 5; 1 0];
```

11 Create a new cross-validated ensemble using `cost` as misclassification cost, and inspect the resulting confusion matrix:

```
aC = fitensemble(X,Y,'GentleBoost',200,t,...
  'PredictorNames',VarNames(2:end),'LearnRate',0.1,...
  'CategoricalPredictors',ncat,'kfold',5,...
  'cost',cost);
[YfitC,SfitC] = kfoldPredict(aC);
confusionmat(Y,YfitC,'order',ClassNames)

ans =
    19    13
     9   114
```

As expected, the new ensemble does a better job classifying the people who die. Somewhat surprisingly, the new ensemble also does a better job classifying the people who live, though the result is not statistically significantly better. The results of the cross validation are random, so this result is simply a statistical fluctuation. The result seems to indicate that the classification of people who live is not very sensitive to the cost.

Example: Classification with Many Categorical Levels

Generally, you cannot use classification with more than 31 levels in any categorical predictor. However, two boosting algorithms can classify data with many categorical levels: `LogitBoost` and `GentleBoost`. For details, see “LogitBoost” on page 13-125 and “GentleBoost” on page 13-126.

This example uses demographic data from the U.S. Census, available at `http://archive.ics.uci.edu/ml/machine-learning-databases/adult/`. The objective of the researchers who posted the data is predicting whether an individual makes more than $50,000/year, based on a set of characteristics.

You can see details of the data, including predictor names, in the `adult.names` file at the site.

1. Load the `'adult.data'` file from the UCI Machine Learning Repository:

```
s = urlread(['http://archive.ics.uci.edu/ml/' ...
   'machine-learning-databases/adult/adult.data']);
```

2. `'adult.data'` represents missing data as `'?'`. Replace instances of missing data with the blank string `''`:

```
s = strrep(s,'?','');
```

3. Put the data into a MATLAB dataset array:

```
fid = fopen('adult.txt','w');
fwrite(fid,s);
fclose(fid);
clear s;
VarNames = {'age' 'workclass' 'fnlwgt' 'education' 'education_num' ...
    'marital_status' 'occupation' 'relationship' 'race' ...
    'sex' 'capital_gain' 'capital_loss' ...
    'hours_per_week' 'native_country' 'income'};
ds = dataset('file','adult.txt','VarNames',VarNames,...
    'Delimiter',',','ReadVarNames',false,'Format',...
   '%u%s%u%s%u%s%s%s%s%s%u%u%u%s%s');
cat = ~datasetfun(@isnumeric,ds(:,1:end-1)); % Logical indices
%                                             of categorical variables
catcol = find(cat); % indices of categorical variables
```

4. Many predictors in the data are categorical. Convert those fields in the dataset array to `nominal`:

```
ds.workclass = nominal(ds.workclass);
ds.education = nominal(ds.education);
ds.marital_status = nominal(ds.marital_status);
ds.occupation = nominal(ds.occupation);
ds.relationship = nominal(ds.relationship);
ds.race = nominal(ds.race);
ds.sex = nominal(ds.sex);
```

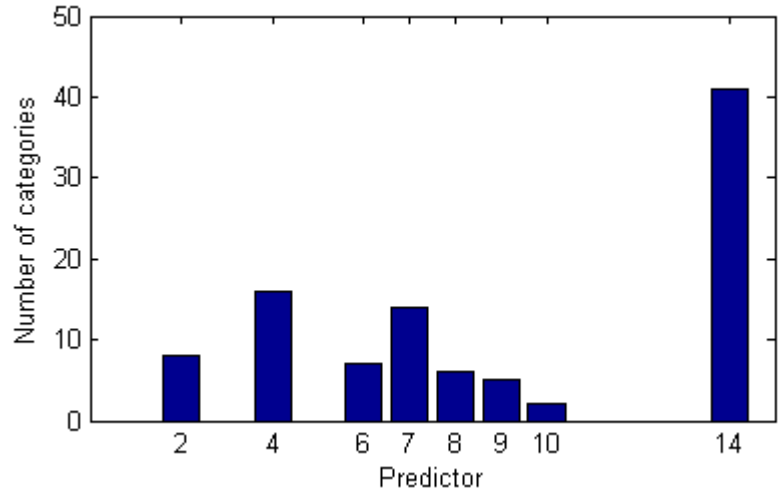
```
ds.native_country = nominal(ds.native_country);
ds.income = nominal(ds.income);
```

5 Convert the dataset array into numerical variables for `fitensemble`:

```
X = double(ds(:,1:end-1));
Y = ds.income;
```

6 Some variables have many levels. Plot the number of levels of each predictor:

```
ncat = zeros(1,numel(catcol));
for c=1:numel(catcol)
    [~,gn] = grp2idx(X(:,catcol(c)));
    ncat(c) = numel(gn);
end
figure;
bar(catcol,ncat);
xlabel('Predictor');
ylabel('Number of categories');
```



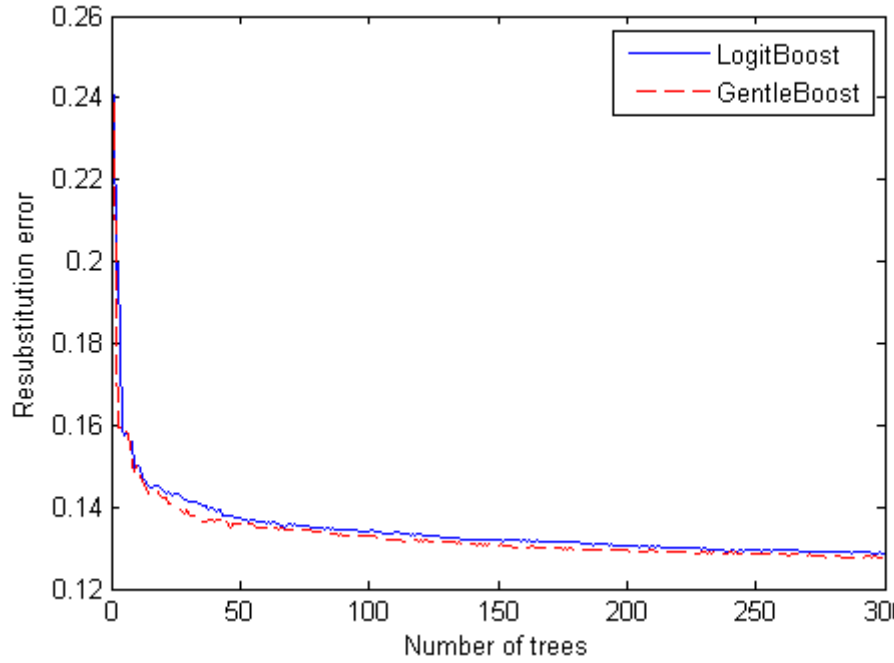
Predictor 14 (`'native_country'`) has more than 40 categorical levels. This is too many levels for any method except `LogitBoost` and `GentleBoost`.

7 Create classification ensembles using both `LogitBoost` and `GentleBoost`:

```
lb = fitensemble(X,Y,'LogitBoost',300,'Tree','CategoricalPredictors',cat,...
    'PredictorNames',VarNames(1:end-1),'ResponseName','income');
gb = fitensemble(X,Y,'GentleBoost',300,'Tree','CategoricalPredictors',cat,...
    'PredictorNames',VarNames(1:end-1),'ResponseName','income');
```

8 Examine the resubstitution error for the two ensembles:

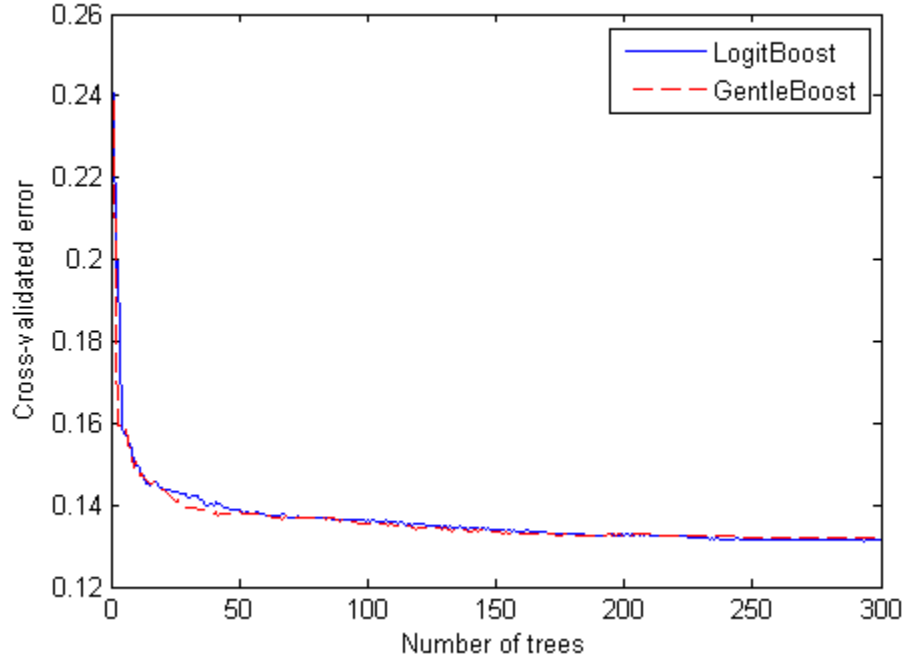
```
figure;
plot(resubLoss(lb,'mode','cumulative'));
hold on
plot(resubLoss(gb,'mode','cumulative'),'r--');
hold off
xlabel('Number of trees');
ylabel('Resubstitution error');
legend('LogitBoost','GentleBoost','Location','NE');
```



The algorithms have similar resubstitution error.

9 Estimate the generalization error for the two algorithms by cross validation.

```
lbcv = crossval(lb,'kfold',5);
gbcv = crossval(gb,'kfold',5);
figure;
plot(kfoldLoss(lbcv,'mode','cumulative'));
hold on
plot(kfoldLoss(gbcv,'mode','cumulative'),'r--');
hold off
xlabel('Number of trees');
ylabel('Cross-validated error');
legend('LogitBoost','GentleBoost','Location','NE');
```



The cross-validated loss is nearly the same as the resubstitution error.

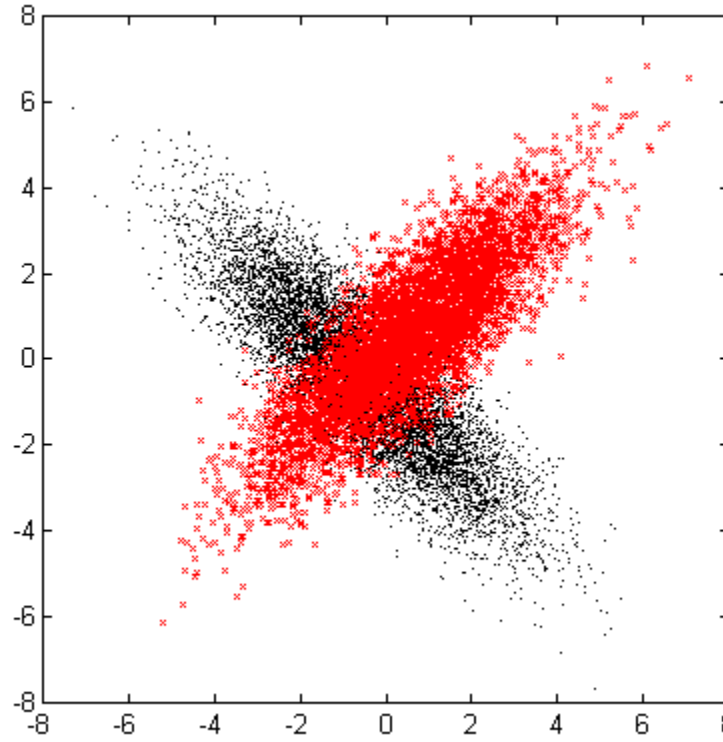
Example: Surrogate Splits

When you have missing data, trees and ensembles of trees give better predictions when they include surrogate splits. Furthermore, estimates of predictor importance are often different with surrogate splits. Eliminating unimportant predictors can save time and memory for predictions, and can make predictions easier to understand.

This example shows the effects of surrogate splits for predictions for data containing missing entries in both training and test sets. There is a redundant predictor in the data, which the surrogate split uses to infer missing values. While the example is artificial, it shows the value of surrogate splits with missing data.

1 Generate and plot two different normally-distributed populations, one with 5000 members, one with 10,000 members:

```
rng(1,'twister') % for reproducibility
N = 5000;
N1 = 2*N; % number in population 1
N2 = N; % number in population 2
mu1 = [-1 -1]/2; % mean of population 1
mu2 = [1 1]/2; % mean of population 2
S1 = [3   -2.5;...
   -2.5    3]; % variance of population 1
S2 = [3   2.5;...
   2.5    3]; % variance of population 2
X1 = mvnrnd(mu1,S1,N1); % population 1
X2 = mvnrnd(mu2,S2,N2); % population 2
X = [X1; X2]; % total population
Y = ones(N1+N2,1); % label population 1
Y(N1+1:end) = 2; % label population 2

figure
plot(X1(:,1),X1(:,2),'k.','MarkerSize',2)
hold on
plot(X2(:,1),X2(:,2),'rx','MarkerSize',3);
hold off
axis square
```



There is a good deal of overlap between the data points. You cannot expect perfect classification of this data.

2 Make a third predictor that is the same as the first component of `X`:

```
X = [X X(:,1)];
```

3 Remove half the values of predictor `1` at random:

```
X(rand(size(X(:,1))) < 0.5,1) = NaN;
```

4 Partition the data into a training set and a test set:

```
cv = cvpartition(Y,'holdout',0.3); % 30% test data
```

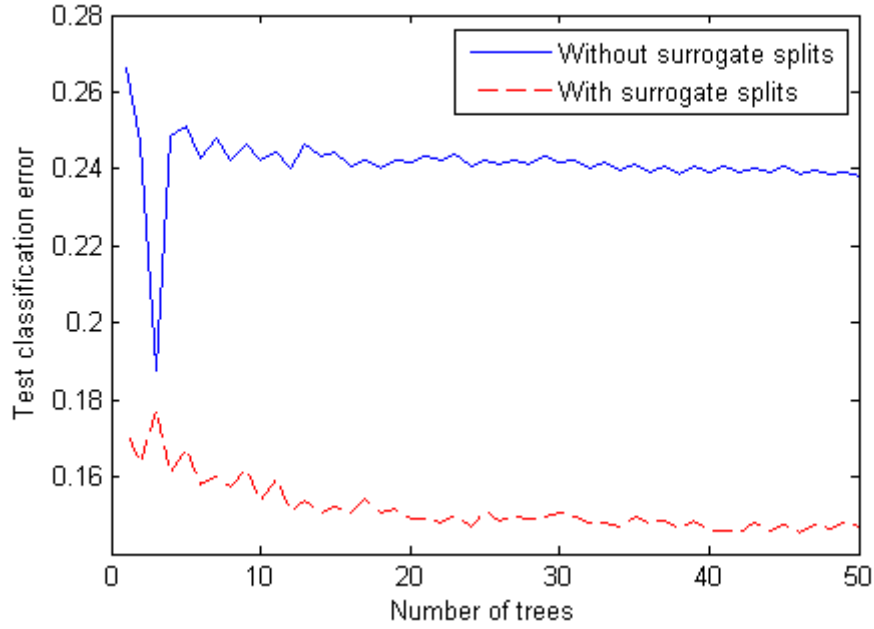
```
Xtrain = X(training(cv),:);
Ytrain = Y(training(cv));
Xtest = X(test(cv),:);
Ytest = Y(test(cv));
```

5 Create two `Bag` ensembles: one with surrogate splits, one without. First create the template for surrogate splits, then train both ensembles:

```
templS = ClassificationTree.template('surrogate','on');
bag = fitensemble(Xtrain,Ytrain,'Bag',50,'Tree',...
    'type','class','nprint',10);

Training Bag...
Grown weak learners: 10
Grown weak learners: 20
Grown weak learners: 30
Grown weak learners: 40
Grown weak learners: 50

bagS = fitensemble(Xtrain,Ytrain,'Bag',50,templS,...
    'type','class','nprint',10);

Training Bag...
Grown weak learners: 10
Grown weak learners: 20
Grown weak learners: 30
Grown weak learners: 40
Grown weak learners: 50
```

6 Examine the accuracy of the two ensembles for predicting the test data:

```
figure
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
hold on
plot(loss(bagS,Xtest,Ytest,'mode','cumulative'),'r--');
hold off;
legend('Without surrogate splits','With surrogate splits');
xlabel('Number of trees');
ylabel('Test classification error');
```



The ensemble with surrogate splits is obviously more accurate than the ensemble without surrogate splits.

7 Check the statistical significance of the difference in results with the McNemar test:

```
Yfit = predict(bag,Xtest);
YfitS = predict(bagS,Xtest);
N10 = sum(Yfit==Ytest & YfitS~=Ytest);
N01 = sum(Yfit~=Ytest & YfitS==Ytest);
mcnemar = (abs(N10-N01) - 1)^2/(N10+N01);
pval = 1 - chi2cdf(mcnemar,1)

pval =
     0
```

The extremely low *p*-value indicates that the ensemble with surrogate splits is better in a statistically significant manner.

Ensemble Regularization

Regularization is a process of choosing fewer weak learners for an ensemble in a way that does not diminish predictive performance. Currently you can regularize regression ensembles.

The `regularize` method finds an optimal set of learner weights α_t that minimize

$$\sum_{n=1}^{N} w_n g\left(\left(\sum_{t=1}^{T}\alpha_t h_t\left(x_n\right)\right), y_n\right) + \lambda\sum_{t=1}^{T}|\alpha_t|.$$

Here

- $\lambda \geq 0$ is a parameter you provide, called the lasso parameter.
- h_t is a weak learner in the ensemble trained on N observations with predictors x_n, responses y_n, and weights w_n.
- $g(f,y) = (f - y)^2$ is the squared error.

The ensemble is regularized on the same (x_n,y_n,w_n) data used for training, so

$$\sum_{n=1}^{N} w_n g\left(\left(\sum_{t=1}^{T}\alpha_t h_t\left(x_n\right)\right), y_n\right)$$

is the ensemble resubstitution error (MSE).

If you use $\lambda = 0$, `regularize` finds the weak learner weights by minimizing the resubstitution MSE. Ensembles tend to overtrain. In other words, the resubstitution error is typically smaller than the true generalization error. By making the resubstitution error even smaller, you are likely to make the ensemble accuracy worse instead of improving it. On the other hand, positive values of λ push the magnitude of the α_t coefficients to 0. This often improves the generalization error. Of course, if you choose λ too large, all the optimal coefficients are 0, and the ensemble does not have any accuracy. Usually you can find an optimal range for λ in which the accuracy of the regularized ensemble is better or comparable to that of the full ensemble without regularization.

A nice feature of lasso regularization is its ability to drive the optimized coefficients precisely to zero. If a learner's weight a_t is 0, this learner can be excluded from the regularized ensemble. In the end, you get an ensemble with improved accuracy and fewer learners.

Example: Regularizing a Regression Ensemble

This example uses data for predicting the insurance risk of a car based on its many attributes.

1 Load the `imports-85` data into the MATLAB workspace:

```
load imports-85;
```

2 Look at a description of the data to find the categorical variables and predictor names:

```
Description

Description =
1985 Auto Imports Database from the UCI repository
http://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.names
Variables have been reordered to place variables with numeric values (referred
to as "continuous" on the UCI site) to the left and categorical values to the
right. Specifically, variables 1:16 are: symboling, normalized-losses,
wheel-base, length, width, height, curb-weight, engine-size, bore, stroke,
compression-ratio, horsepower, peak-rpm, city-mpg, highway-mpg, and price.
Variables 17:26 are: make, fuel-type, aspiration, num-of-doors, body-style,
drive-wheels, engine-location, engine-type, num-of-cylinders, and fuel-system.
```

The objective of this process is to predict the "symboling," the first variable in the data, from the other predictors. "symboling" is an integer from `-3` (good insurance risk) to `3` (poor insurance risk). You could use a classification ensemble to predict this risk instead of a regression ensemble. As stated in "Steps in Supervised Learning (Machine Learning)" on page 13-2, when you have a choice between regression and classification, you should try regression first. Furthermore, this example is to show regularization, which currently works only for regression.

3 Prepare the data for ensemble fitting:

```
Y = X(:,1);
X(:,1) = [];
VarNames = {'normalized-losses' 'wheel-base' 'length' 'width' 'height' ...
  'curb-weight' 'engine-size' 'bore' 'stroke' 'compression-ratio' ...
  'horsepower' 'peak-rpm' 'city-mpg' 'highway-mpg' 'price' 'make' ...
  'fuel-type' 'aspiration' 'num-of-doors' 'body-style' 'drive-wheels' ...
  'engine-location' 'engine-type' 'num-of-cylinders' 'fuel-system'};
catidx = 16:25; % indices of categorical predictors
```

4 Create a regression ensemble from the data using 300 default trees:

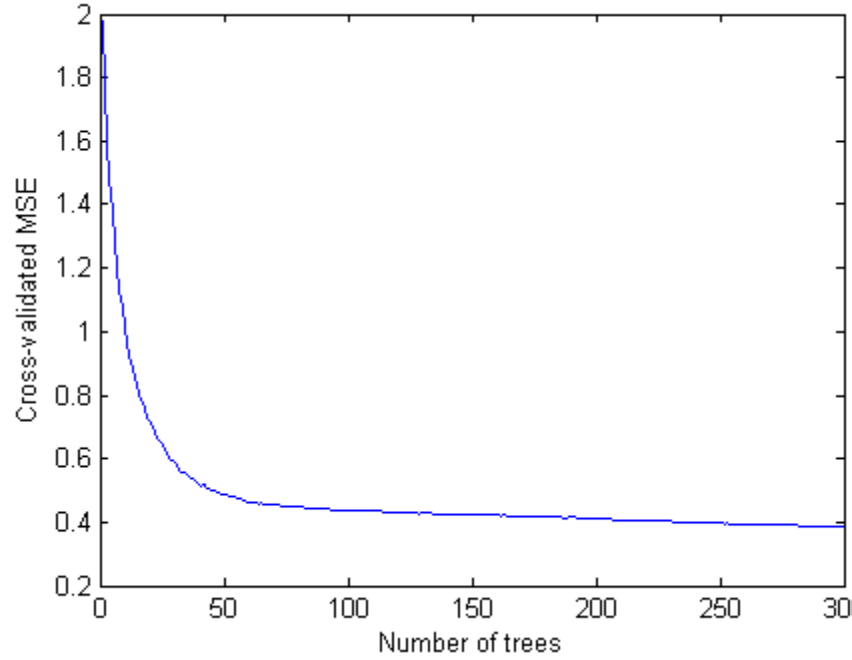
```
ls = fitensemble(X,Y,'LSBoost',300,'Tree','LearnRate',0.1,...
    'PredictorNames',VarNames,'ResponseName','symboling',...
    'CategoricalPredictors',catidx)

ls =

classreg.learning.regr.RegressionEnsemble:
           PredictorNames: {1x25 cell}
    CategoricalPredictors: [16 17 18 19 20 21 22 23 24 25]
             ResponseName: 'symboling'
        ResponseTransform: 'none'
            NObservations: 205
                 NTrained: 300
                   Method: 'LSBoost'
             LearnerNames: {'Tree'}
     ReasonForTermination: [1x77 char]
                  FitInfo: [300x1 double]
       FitInfoDescription: [2x83 char]
           Regularization: []
```

The final line, `Regularization`, is empty (`[]`). To regularize the ensemble, you have to use the `regularize` method.

5 Cross validate the ensemble, and inspect its loss curve.

```
cv = crossval(ls,'kfold',5);
figure;
plot(kfoldLoss(cv,'mode','cumulative'));
```

```
xlabel('Number of trees');
ylabel('Cross-validated MSE');
```



It appears you might obtain satisfactory performance from a smaller ensemble, perhaps one containing from 50 to 100 trees.

6 Call the `regularize` method to try to find trees that you can remove from the ensemble. By default, `regularize` examines 10 values of the lasso (`Lambda`) parameter spaced exponentially.

```
ls = regularize(ls)

ls =

classreg.learning.regr.RegressionEnsemble:
           PredictorNames: {1x25 cell}
    CategoricalPredictors: [16 17 18 19 20 21 22 23 24 25]
```

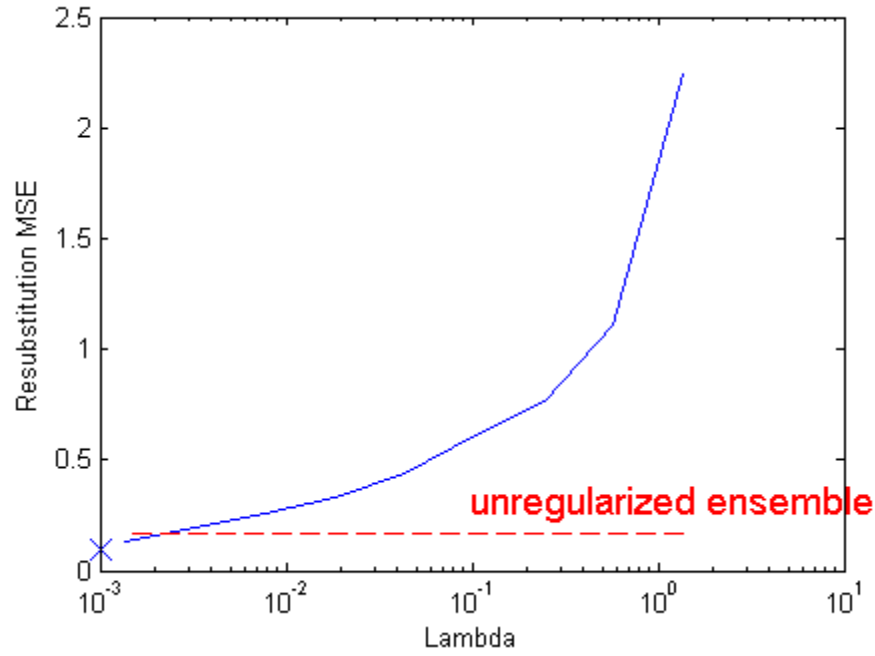
```
                ResponseName: 'symboling'
           ResponseTransform: 'none'
               NObservations: 205
                    NTrained: 300
                      Method: 'LSBoost'
                LearnerNames: {'Tree'}
      ReasonForTermination: [1x77 char]
                     FitInfo: [300x1 double]
          FitInfoDescription: [2x83 char]
              Regularization: [1x1 struct]
```

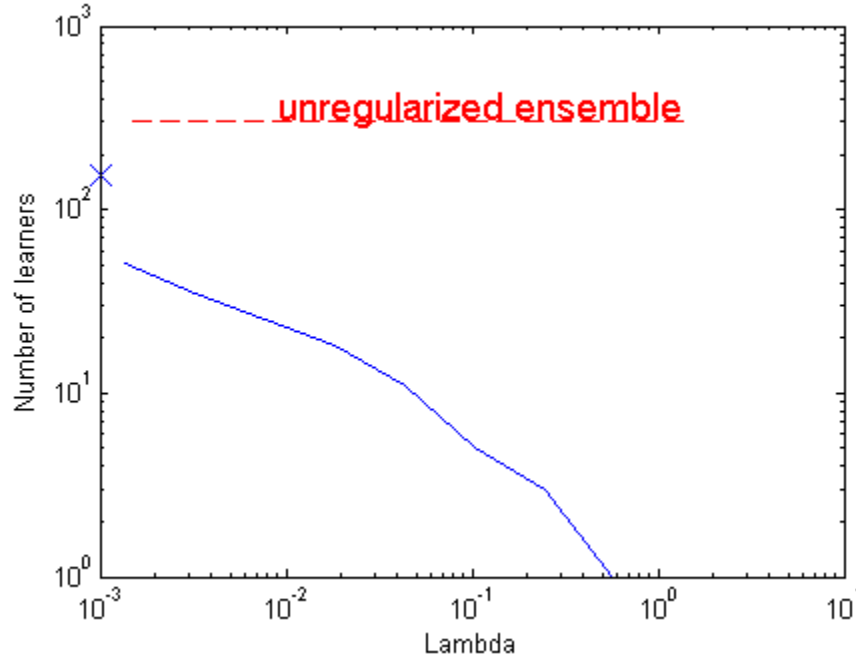
The `Regularization` property is no longer empty.

7 Plot the resubstitution mean-squared error (MSE) and number of learners with nonzero weights against the lasso parameter. Separately plot the value at `Lambda=0`. Use a logarithmic scale since the values of `Lambda` are exponentially spaced.

```
figure;
semilogx(ls.Regularization.Lambda,ls.Regularization.ResubstitutionMSE);
line([1e-3 1e-3],[ls.Regularization.ResubstitutionMSE(1) ...
     ls.Regularization.ResubstitutionMSE(1)],...
    'marker','x','markersize',12,'color','b');
r0 = resubLoss(ls);
line([ls.Regularization.Lambda(2) ls.Regularization.Lambda(end)],...
     [r0 r0],'color','r','LineStyle','--');
xlabel('Lambda');
ylabel('Resubstitution MSE');
annotation('textbox',[0.5 0.22 0.5 0.05],'String','unregularized ensemble',...
    'color','r','FontSize',14,'LineStyle','none');

figure;
loglog(ls.Regularization.Lambda,sum(ls.Regularization.TrainedWeights>0,1));
line([1e-3 1e-3],...
    [sum(ls.Regularization.TrainedWeights(:,1)>0) ...
    sum(ls.Regularization.TrainedWeights(:,1)>0)],...
    'marker','x','markersize',12,'color','b');
line([ls.Regularization.Lambda(2) ls.Regularization.Lambda(end)],...
    [ls.NTrained ls.NTrained],...
```

```
    'color','r','LineStyle','--');
xlabel('Lambda');
ylabel('Number of learners');
annotation('textbox',[0.3 0.8 0.5 0.05],'String','unregularized ensemble',...
    'color','r','FontSize',14,'LineStyle','none');
```



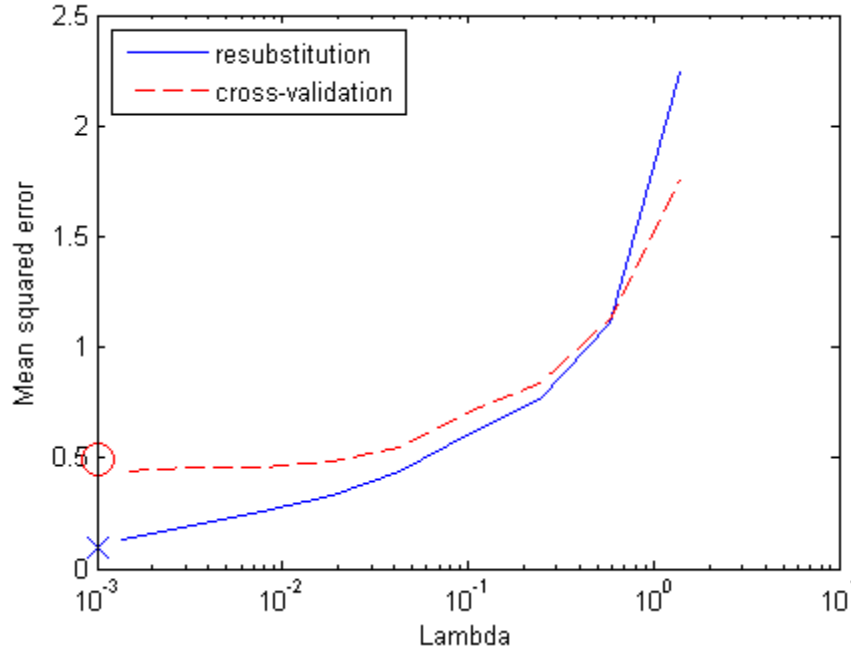


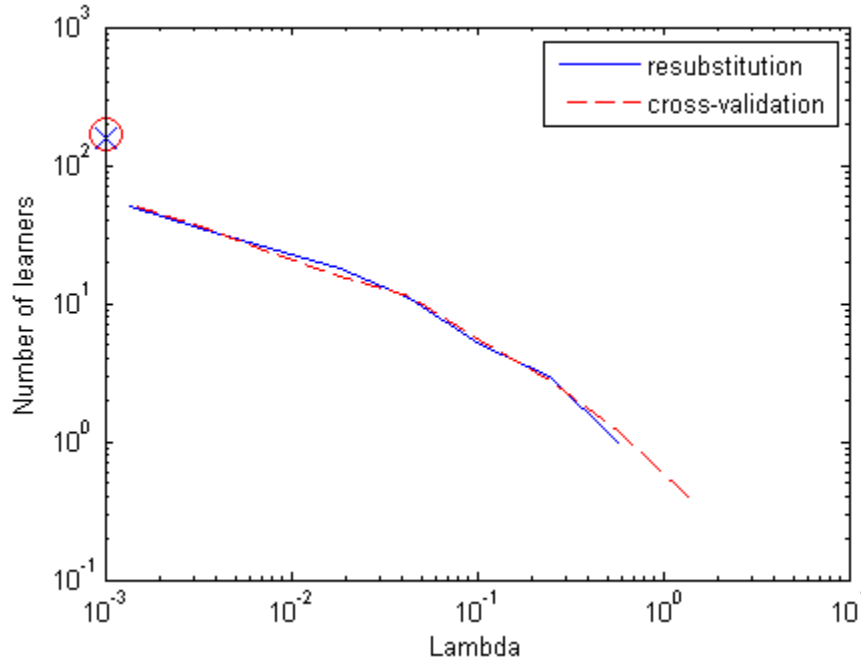
8 The resubstitution MSE values are likely to be overly optimistic. To obtain more reliable estimates of the error associated with various values of `Lambda`, cross validate the ensemble using `cvshrink`. Plot the resulting cross validation loss (MSE) and number of learners against `Lambda`.

```
rng(0,'Twister') % for reproducibility
[mse,nlearn] = cvshrink(ls,'lambda',ls.Regularization.Lambda,'kfold',5);

figure;
semilogx(ls.Regularization.Lambda,ls.Regularization.ResubstitutionMSE);
hold;
semilogx(ls.Regularization.Lambda,mse,'r--');
hold off;
xlabel('Lambda');
ylabel('Mean squared error');
legend('resubstitution','cross-validation','Location','NW');
```

```
line([1e-3 1e-3],[ls.Regularization.ResubstitutionMSE(1) ...
     ls.Regularization.ResubstitutionMSE(1)],...
    'marker','x','markersize',12,'color','b');
line([1e-3 1e-3],[mse(1) mse(1)],'marker','o',...
    'markersize',12,'color','r','LineStyle','--');

figure;
loglog(ls.Regularization.Lambda,sum(ls.Regularization.TrainedWeights>0,1));
hold;
loglog(ls.Regularization.Lambda,nlearn,'r--');
hold off;
xlabel('Lambda');
ylabel('Number of learners');
legend('resubstitution','cross-validation','Location','NE');
line([1e-3 1e-3],...
    [sum(ls.Regularization.TrainedWeights(:,1)>0) ...
    sum(ls.Regularization.TrainedWeights(:,1)>0)],...
    'marker','x','markersize',12,'color','b');
line([1e-3 1e-3],[nlearn(1) nlearn(1)],'marker','o',...
```

```
'markersize',12,'color','r','LineStyle','--');
```





Examining the cross-validated error shows that the cross-validation MSE is almost flat for `Lambda` up to a bit over `1e-2`.

9 Examine `ls.Regularization.Lambda` to find the highest value that gives MSE in the flat region (up to a bit over `1e-2`):

```
jj = 1:length(ls.Regularization.Lambda);
[jj;ls.Regularization.Lambda]

ans =

  Columns 1 through 6

    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000
         0    0.0014    0.0033    0.0077    0.0183    0.0435

  Columns 7 through 10
```

```
    7.0000    8.0000    9.0000   10.0000
    0.1031    0.2446    0.5800    1.3754
```

Element 5 of `ls.Regularization.Lambda` has value `0.0183`, the largest in the flat range.

10 Reduce the ensemble size using the `shrink` method. `shrink` returns a compact ensemble with no training data. The generalization error for the new compact ensemble was already estimated by cross validation in `mse(5)`.

```
cmp = shrink(ls,'weightcolumn',5)

cmp =

classreg.learning.regr.CompactRegressionEnsemble:
           PredictorNames: {1x25 cell}
    CategoricalPredictors: [16 17 18 19 20 21 22 23 24 25]
             ResponseName: 'symboling'
        ResponseTransform: 'none'
                 NTrained: 18
```

There are only 18 trees in the new ensemble, notably reduced from the 300 in `ls`.

11 Compare the sizes of the ensembles:

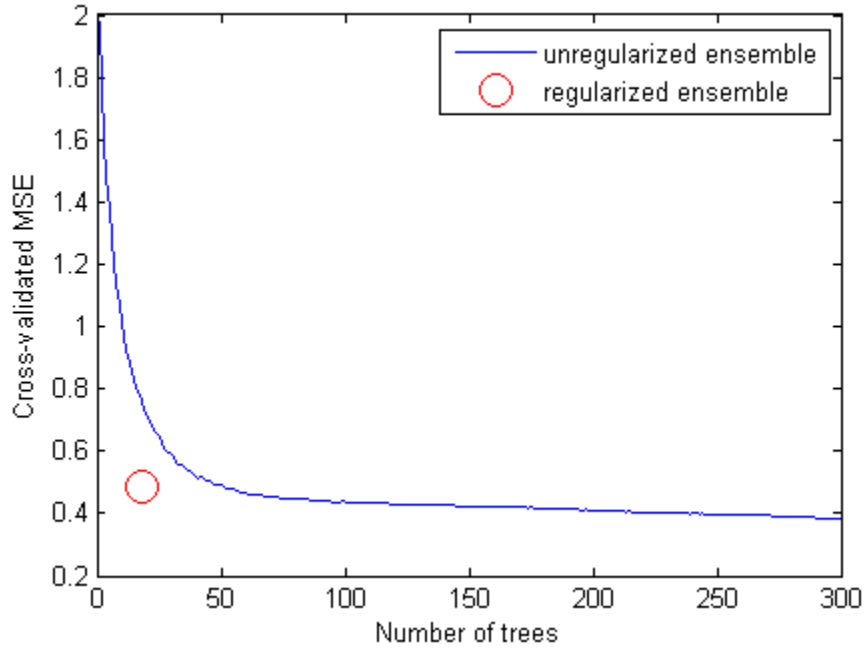
```
sz(1) = whos('cmp'); sz(2) = whos('ls');
[sz(1).bytes sz(2).bytes]

ans =
      162270     2791024
```

The reduced ensemble is about 6% the size of the original.

12 Compare the MSE of the reduced ensemble to that of the original ensemble:

```
figure;
plot(kfoldLoss(cv,'mode','cumulative'));
hold on
plot(cmp.NTrained,mse(5),'ro','MarkerSize',12);
xlabel('Number of trees');
```

```
ylabel('Cross-validated MSE');
legend('unregularized ensemble','regularized ensemble',...
    'Location','NE');
hold off
```



The reduced ensemble gives low loss while using many fewer trees.

Example: Tuning RobustBoost

The `RobustBoost` algorithm can make good classification predictions even when the training data has noise. However, the default `RobustBoost` parameters can produce an ensemble that does not predict well. This example shows one way of tuning the parameters for better predictive accuracy.

1. Generate data with label noise. This example has twenty uniform random numbers per observation, and classifies the observation as `1` if the sum of the first five numbers exceeds 2.5 (so is larger than average), and `0` otherwise:

```
rng(0,'twister') % for reproducibility
Xtrain = rand(2000,20);
Ytrain = sum(Xtrain(:,1:5),2) > 2.5;
```

2 To add noise, randomly switch 10% of the classifications:

```
idx = randsample(2000,200);
Ytrain(idx) = ~Ytrain(idx);
```

3 Create an ensemble with `AdaBoostM1` for comparison purposes:

```
ada = fitensemble(Xtrain,Ytrain,'AdaBoostM1',...
    300,'Tree','LearnRate',0.1);
```

4 Create an ensemble with `RobustBoost`. Since the data has 10% incorrect classification, perhaps an error goal of 15% is reasonable.

```
rb1 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,...
   'Tree','RobustErrorGoal',0.15,'RobustMaxMargin',1);
```

5 Try setting a high value of the error goal, `0.6`. You get an error:

```
rb2 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,'Tree','RobustErrorGoal',0.6)

??? Error using ==> RobustBoost>RobustBoost.RobustBoost at 33
For the chosen values of 'RobustMaxMargin' and 'RobustMarginSigma', you must set
'RobustErrorGoal' to a value between 0 and 0.5.
```

6 Create an ensemble with an error goal in the allowed range, `0.4`:

```
rb2 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,...
   'Tree','RobustErrorGoal',0.4);
```

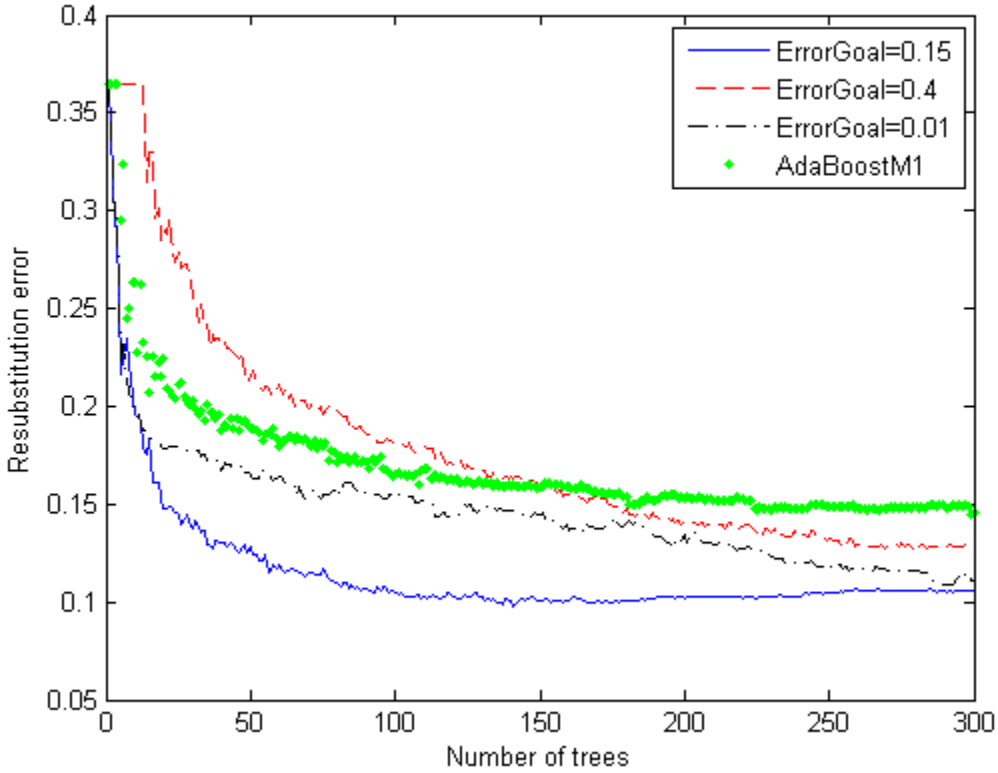
7 Create an ensemble with very optimistic error goal, `0.01`:

```
rb3 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,...
   'Tree','RobustErrorGoal',0.01);
```

8 Compare the resubstitution error of the four ensembles:

```
figure
```

```
plot(resubLoss(rb1,'mode','cumulative'));
hold on
plot(resubLoss(rb2,'mode','cumulative'),'r--');
plot(resubLoss(rb3,'mode','cumulative'),'k-.');
plot(resubLoss(ada,'mode','cumulative'),'g.');
hold off;
xlabel('Number of trees');
ylabel('Resubstitution error');
legend('ErrorGoal=0.15','ErrorGoal=0.4','ErrorGoal=0.01',...
    'AdaBoostM1','Location','NE');
```

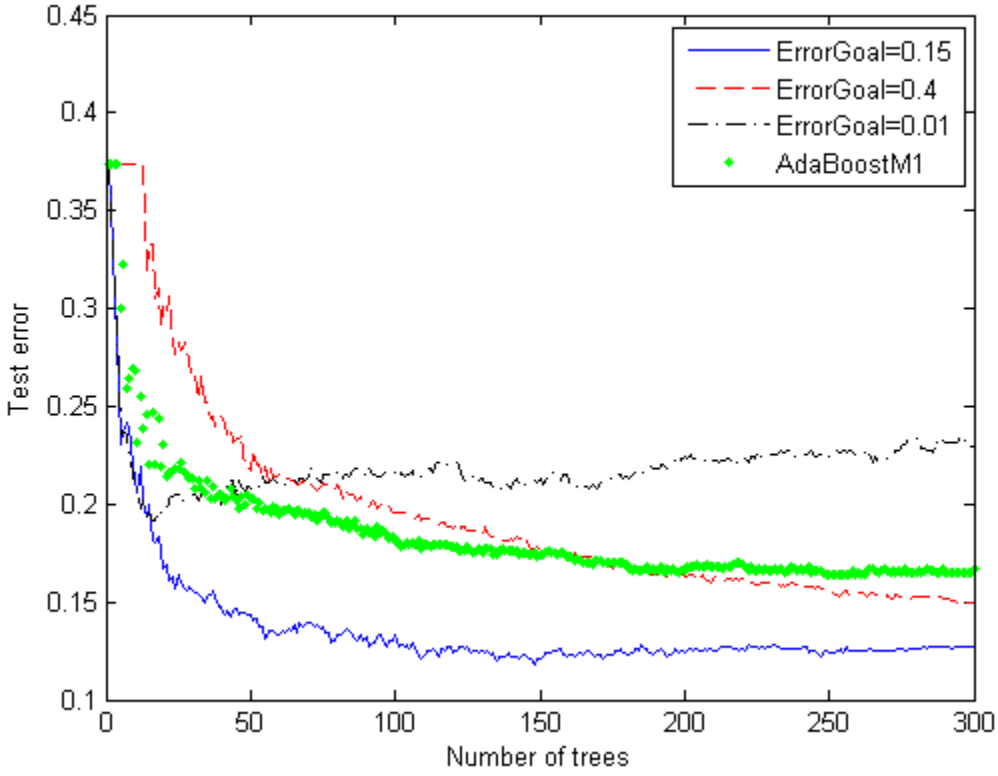


All the `RobustBoost` curves show lower resubstitution error than the `AdaBoostM1` curve. The error goal of `0.15` curve shows the lowest

resubstitution error over most of the range. However, its error is rising in the latter half of the plot, while the other curves are still descending.

9 Generate test data to see the predictive power of the ensembles. Test the four ensembles:

```
Xtest = rand(2000,20);
Ytest = sum(Xtest(:,1:5),2) > 2.5;
idx = randsample(2000,200);
Ytest(idx) = ~Ytest(idx);
figure;
plot(loss(rb1,Xtest,Ytest,'mode','cumulative'));
hold on
plot(loss(rb2,Xtest,Ytest,'mode','cumulative'),'r--');
plot(loss(rb3,Xtest,Ytest,'mode','cumulative'),'k-.');
plot(loss(ada,Xtest,Ytest,'mode','cumulative'),'g.');
hold off;
xlabel('Number of trees');
ylabel('Test error');
legend('ErrorGoal=0.15','ErrorGoal=0.4','ErrorGoal=0.01',...
    'AdaBoostM1','Location','NE');
```



The error curve for error goal 0.15 is lowest (best) in the plotted range. The curve for error goal 0.4 seems to be converging to a similar value for a large number of trees, but more slowly. `AdaBoostM1` has higher error than the curve for error goal 0.15. The curve for the too-optimistic error goal 0.01 remains substantially higher (worse) than the other algorithms for most of the plotted range.

TreeBagger Examples

`TreeBagger` ensembles have more functionality than those constructed with `fitensemble`; see TreeBagger Features Not in fitensemble on page 13-120. Also, some property and method names differ from their `fitensemble`

counterparts. This section contains examples of workflow for regression and classification that use this extra `TreeBagger` functionality.

Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger

In this example, use a database of 1985 car imports with 205 observations, 25 input variables, and one response variable, insurance risk rating, or "symboling." The first 15 variables are numeric and the last 10 are categorical. The symboling index takes integer values from –3 to 3.

1 Load the dataset and split it into predictor and response arrays:

```
load imports-85;
Y = X(:,1);
X = X(:,2:end);
```

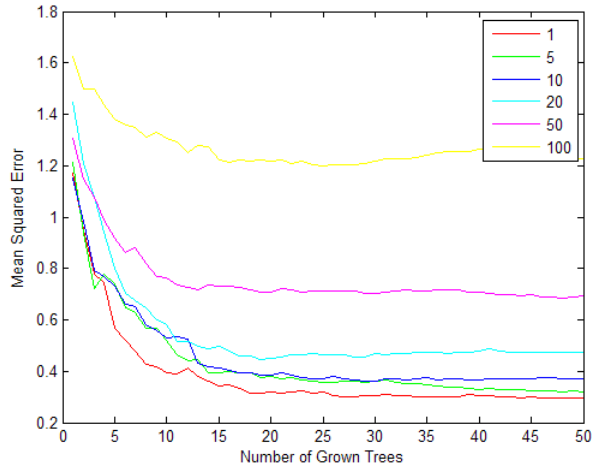
2 Because bagging uses randomized data drawings, its exact outcome depends on the initial random seed. To reproduce the exact results in this example, use the random stream settings:

```
rng(1945,'twister')
```

Finding the Optimal Leaf Size. For regression, the general rule is to set leaf size to 5 and select one third of input features for decision splits at random. In the following step, verify the optimal leaf size by comparing mean-squared errors obtained by regression for various leaf sizes. `oobError` computes MSE versus the number of grown trees. You must set `oobpred` to `'on'` to obtain out-of-bag predictions later.

```
leaf = [1 5 10 20 50 100];
col = 'rgbcmy';
figure(1);
for i=1:length(leaf)
    b = TreeBagger(50,X,Y,'method','r','oobpred','on',...
   'cat',16:25,'minleaf',leaf(i));
    plot(oobError(b),col(i));
    hold on;
end
xlabel('Number of Grown Trees');
ylabel('Mean Squared Error');
```

```
legend({'1' '5' '10' '20' '50' '100'},'Location','NorthEast');
hold off;
```



The red (leaf size 1) curve gives the lowest MSE values.

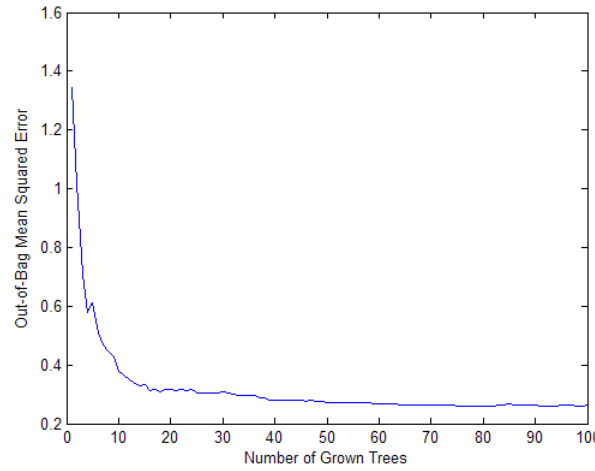
Estimating Feature Importance.

1 In practical applications, you typically grow ensembles with hundreds of trees. Only 50 trees were used in "Finding the Optimal Leaf Size" on page 13-97 for faster processing. Now that you have estimated the optimal leaf size, grow a larger ensemble with 100 trees and use it for estimation of feature importance:

```
b = TreeBagger(100,X,Y,'method','r','oobvarimp','on',...
'cat',16:25,'minleaf',1);
```

2 Inspect the error curve again to make sure nothing went wrong during training:

```
figure(2);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Mean Squared Error');
```



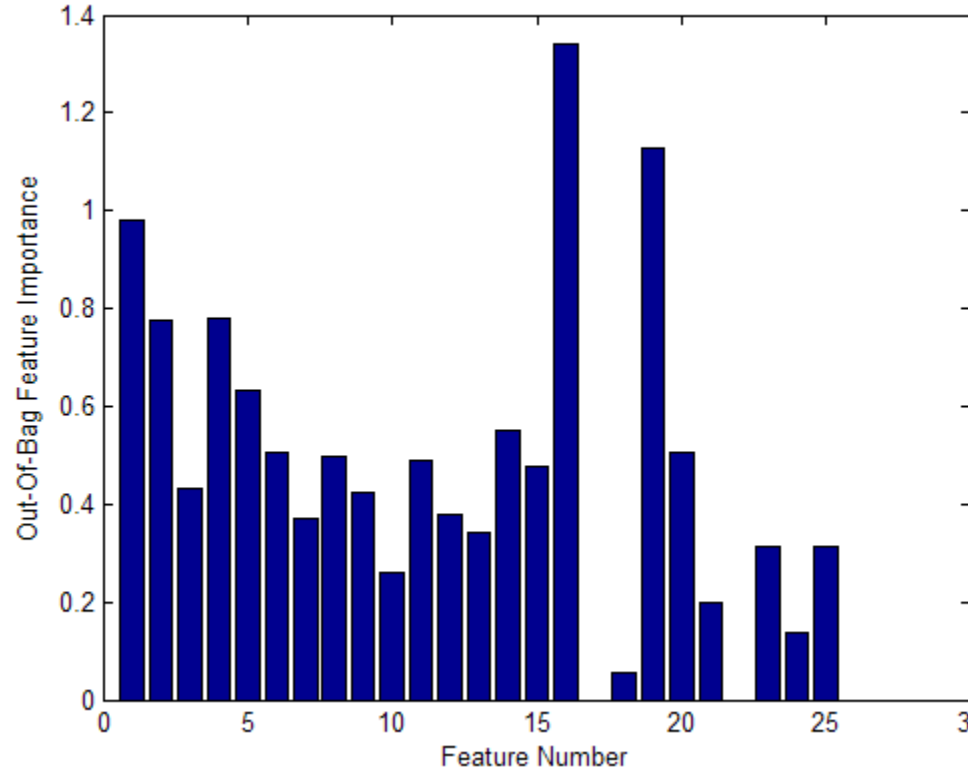
Prediction ability should depend more on important features and less on unimportant features. You can use this idea to measure feature importance.

For each feature, you can permute the values of this feature across all of the observations in the data set and measure how much worse the mean-squared error (MSE) becomes after the permutation. You can repeat this for each feature.

1 Using the following code, plot the increase in MSE due to permuting out-of-bag observations across each input variable. The `OOBPermutedVarDeltaError` array stores the increase in MSE averaged over all trees in the ensemble and divided by the standard deviation taken over the trees, for each variable. The larger this value, the more important the variable. Imposing an arbitrary cutoff at 0.65, you can select the five most important features.

```
figure(3);
bar(b.OOBPermutedVarDeltaError);
xlabel('Feature Number');
ylabel('Out-Of-Bag Feature Importance');
idxvar = find(b.OOBPermutedVarDeltaError>0.65)

idxvar =
```

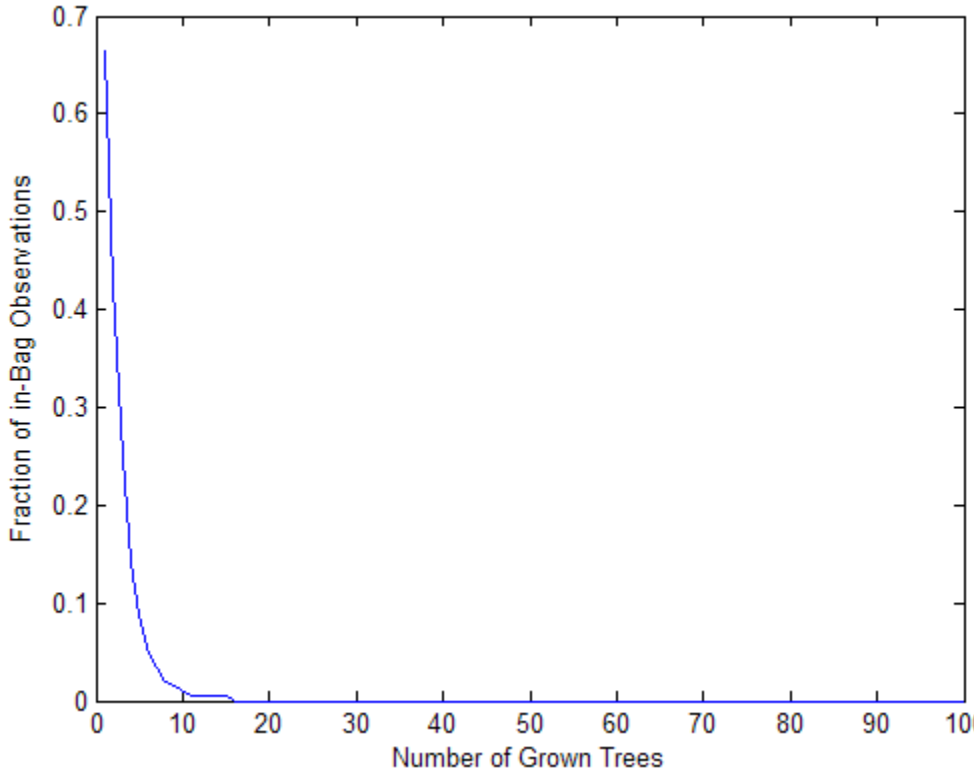
```
    1     2     4    16    19
```



2 The `OOBIndices` property of `TreeBagger` tracks which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately 2/3, the fraction of unique observations selected by one bootstrap replica, and goes down to 0 at approximately 10 trees.

```
finbag = zeros(1,b.NTrees);
for t=1:b.NTrees
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));
end
```

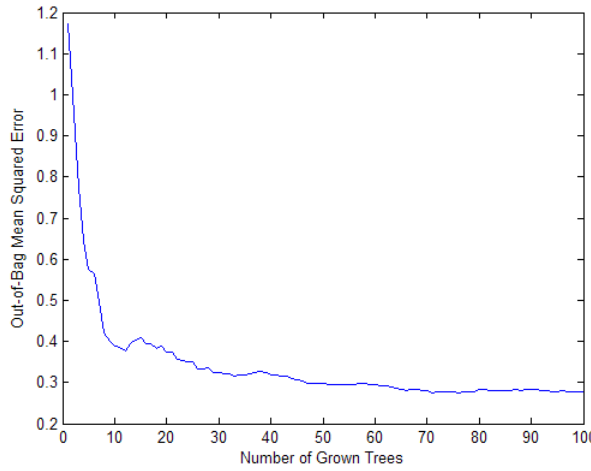
```
finbag = finbag / size(X,1);
figure(4);
plot(finbag);
xlabel('Number of Grown Trees');
ylabel('Fraction of in-Bag Observations');
```

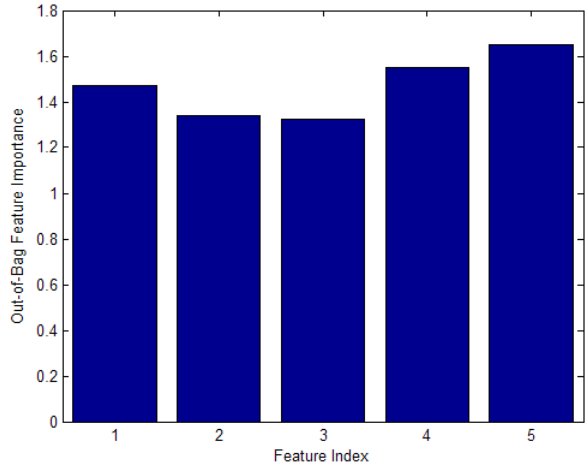


Growing Trees on a Reduced Set of Features. Using just the five most powerful features selected in "Estimating Feature Importance" on page 13-98, determine if it is possible to obtain a similar predictive power. To begin, grow 100 trees on these features only. The first three of the five selected features are numeric and the last two are categorical.

```
b5v = TreeBagger(100,X(:,idxvar),Y,'method','r',...
```

```
'oobvarimp','on','cat',4:5,'minleaf',1);
figure(5);
plot(oobError(b5v));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Mean Squared Error');
figure(6);
bar(b5v.OOBPermutedVarDeltaError);
xlabel('Feature Index');
ylabel('Out-of-Bag Feature Importance');
```





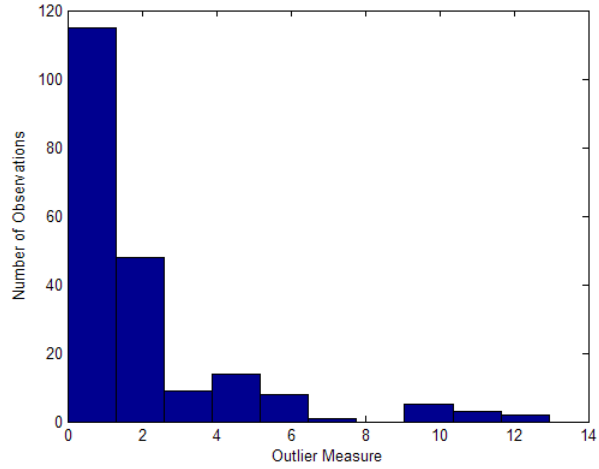
These five most powerful features give the same MSE as the full set, and the ensemble trained on the reduced set ranks these features similarly to each other. Features 1 and 2 from the reduced set perhaps could be removed without a significant loss in the predictive power.

Finding Outliers. To find outliers in the training data, compute the proximity matrix using `fillProximities`:

```
b5v = fillProximities(b5v);
```

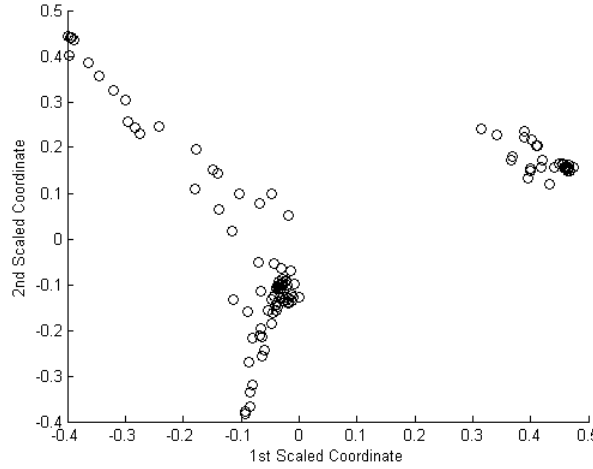
The method normalizes this measure by subtracting the mean outlier measure for the entire sample, taking the magnitude of this difference and dividing the result by the median absolute deviation for the entire sample:

```
figure(7);
hist(b5v.OutlierMeasure);
xlabel('Outlier Measure');
ylabel('Number of Observations');
```



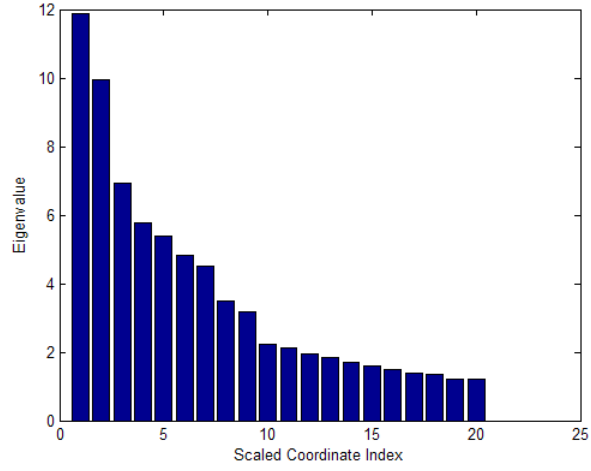
Discovering Clusters in the Data. By applying multidimensional scaling to the computed matrix of proximities, you can inspect the structure of the input data and look for possible clusters of observations. The `mdsProx` method returns scaled coordinates and eigenvalues for the computed proximity matrix. If run with the `colors` option, this method makes a scatter plot of two scaled coordinates, first and second by default.

```
figure(8);
[~,e] = mdsProx(b5v,'colors','k');
xlabel('1st Scaled Coordinate');
ylabel('2nd Scaled Coordinate');
```



Assess the relative importance of the scaled axes by plotting the first 20 eigenvalues:

```
figure(9);
bar(e(1:20));
xlabel('Scaled Coordinate Index');
ylabel('Eigenvalue');
```



Saving the Ensemble Configuration for Future Use. To use the trained ensemble for predicting the response on unseen data, store the ensemble to disk and retrieve it later. If you do not want to compute predictions for out-of-bag data or reuse training data in any other way, there is no need to store the ensemble object itself. Saving the compact version of the ensemble would be enough in this case. Extract the compact object from the ensemble:

```
c = compact(b5v)

c =

Ensemble with 100 decision trees:
Method:           regression
Nvars:                     5
```

This object can be now saved in a `*.mat` file as usual.

Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger

You can also use ensembles of decision trees for classification. For this example, use ionosphere data with 351 observations and 34 real-valued predictors. The response variable is categorical with two levels:

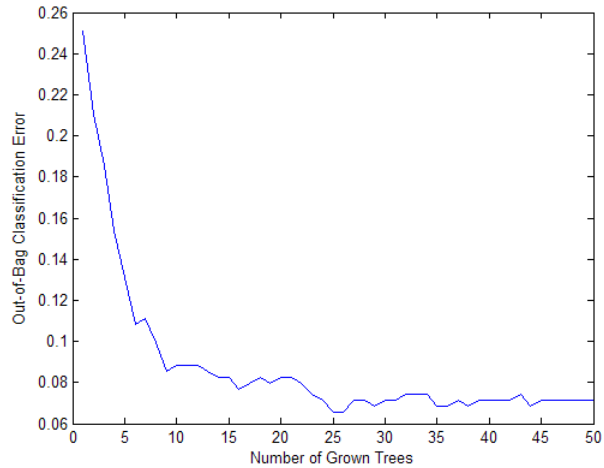
- `'g'` for good radar returns
- `'b'` for bad radar returns

The goal is to predict good or bad returns using a set of 34 measurements. The workflow resembles that for "Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger" on page 13-97.

1. Fix the initial random seed, grow 50 trees, inspect how the ensemble error changes with accumulation of trees, and estimate feature importance. For classification, it is best to set the minimal leaf size to 1 and select the square root of the total number of features for each decision split at random. These are the default settings for a `TreeBagger` used for classification.

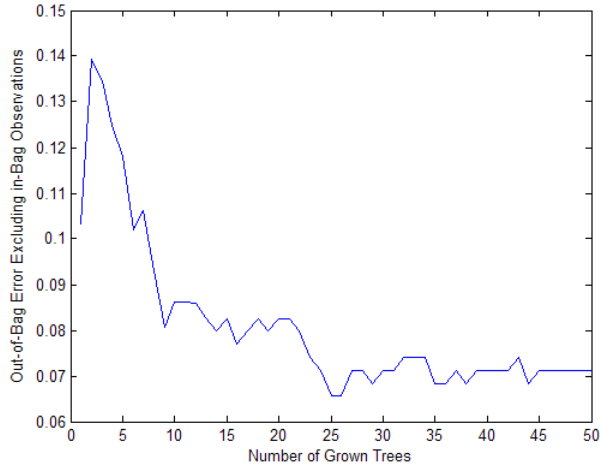
   ```
   load ionosphere;
   rng(1945,'twister')
   b = TreeBagger(50,X,Y,'oobvarimp','on');
   ```

```
figure(10);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Classification Error');
```



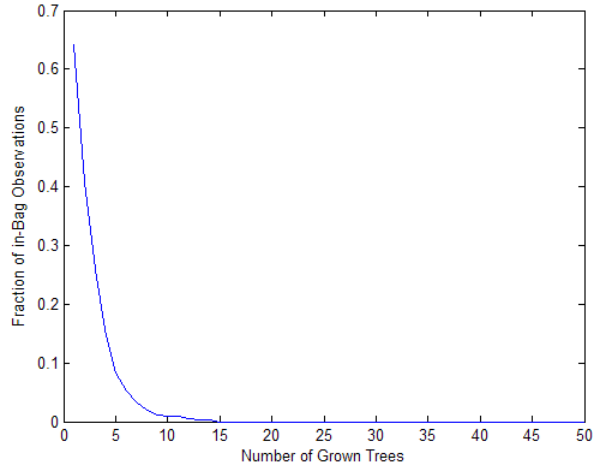
2 The method trains ensembles with few trees on observations that are in bag for all trees. For such observations, it is impossible to compute the true out-of-bag prediction, and `TreeBagger` returns the most probable class for classification and the sample mean for regression. You can change the default value returned for in-bag observations using the `DefaultYfit` property. If you set the default value to an empty string for classification, the method excludes in-bag observations from computation of the out-of-bag error. In this case, the curve is more variable when the number of trees is small, either because some observations are never out of bag (and are therefore excluded) or because their predictions are based on few trees.

```
b.DefaultYfit = '';
figure(11);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Error Excluding in-Bag Observations');
```



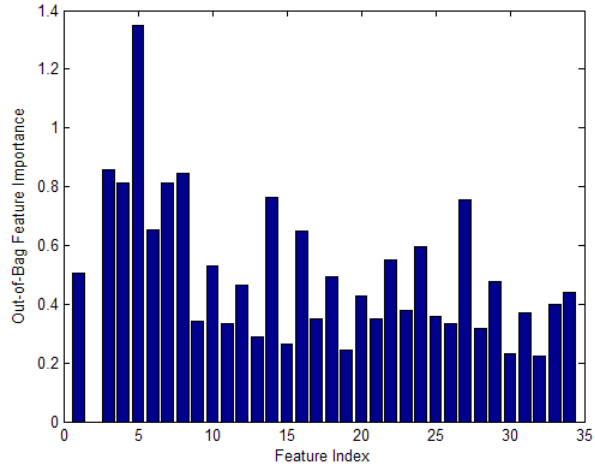
3 The `OOBIndices` property of `TreeBagger` tracks which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately 2/3, the fraction of unique observations selected by one bootstrap replica, and goes down to 0 at approximately 10 trees.

```
finbag = zeros(1,b.NTrees);
for t=1:b.NTrees
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));
end
finbag = finbag / size(X,1);
figure(12);
plot(finbag);
xlabel('Number of Grown Trees');
ylabel('Fraction of in-Bag Observations');
```



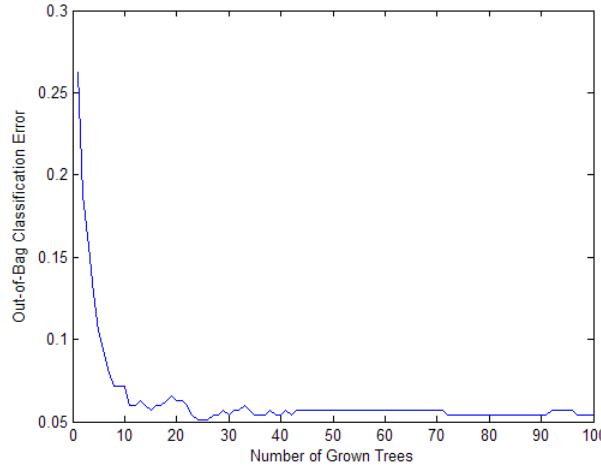
4 Estimate feature importance:

```
figure(13);
bar(b.OOBPermutedVarDeltaError);
xlabel('Feature Index');
ylabel('Out-of-Bag Feature Importance');
idxvar = find(b.OOBPermutedVarDeltaError>0.8)

idxvar =

     3     4     5     7     8
```



5 Having selected the five most important features, grow a larger ensemble on the reduced feature set. Save time by not permuting out-of-bag observations to obtain new estimates of feature importance for the reduced feature set (set `oobvarimp` to `'off'`). You would still be interested in obtaining out-of-bag estimates of classification error (set `oobpred` to `'on'`).

```
b5v = TreeBagger(100,X(:,idxvar),Y,'oobpred','on');
figure(14);
plot(oobError(b5v));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Classification Error');
```

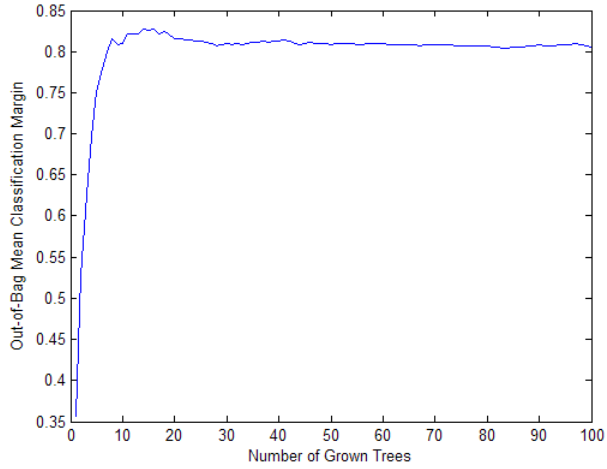


6 For classification ensembles, in addition to classification error (fraction of misclassified observations), you can also monitor the average classification margin. For each observation, the *margin* is defined as the difference between the score for the true class and the maximal score for other classes predicted by this tree. The cumulative classification margin uses the scores averaged over all trees and the mean cumulative classification margin is the cumulative margin averaged over all observations. The `oobMeanMargin` method with the `'mode'` argument set to `'cumulative'` (default) shows how the mean cumulative margin changes as the ensemble grows: every new element in the returned array represents the cumulative margin obtained by including a new tree in the ensemble. If training is successful, you would expect to see a gradual increase in the mean classification margin.

For decision trees, a classification score is the probability of observing an instance of this class in this tree leaf. For example, if the leaf of a grown decision tree has five `'good'` and three `'bad'` training observations in it, the scores returned by this decision tree for any observation fallen on this leaf are 5/8 for the `'good'` class and 3/8 for the `'bad'` class. These probabilities are called `'scores'` for consistency with other classifiers that might not have an obvious interpretation for numeric values of returned predictions.

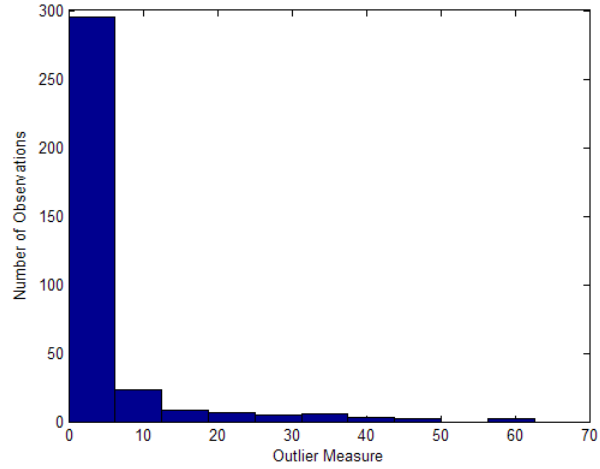
```
figure(15);
```

```
plot(oobMeanMargin(b5v));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Mean Classification Margin');
```



7 Compute the matrix of proximities and look at the distribution of outlier measures. Unlike regression, outlier measures for classification ensembles are computed within each class separately.

```
b5v = fillProximities(b5v);
figure(16);
hist(b5v.OutlierMeasure);
xlabel('Outlier Measure');
ylabel('Number of Observations');
```



8 All extreme outliers for this dataset come from the `'good'` class:

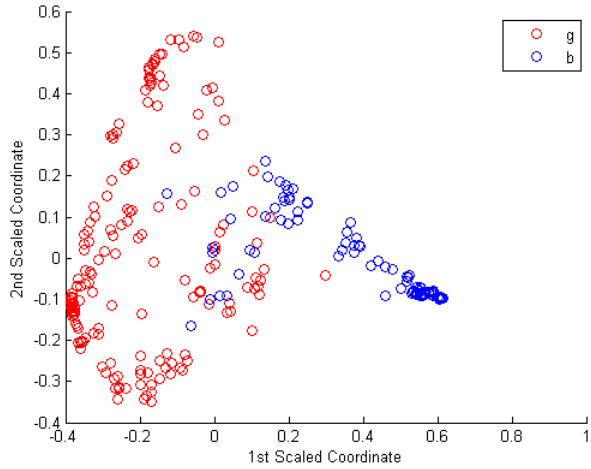
```
b5v.Y(b5v.OutlierMeasure>40)

ans =

    'g'
    'g'
    'g'
    'g'
    'g''
```

9 As for regression, you can plot scaled coordinates, displaying the two classes in different colors using the `colors` argument of `mdsProx`. This argument takes a string in which every character represents a color. To find the order of classes used by the ensemble, look at the `ClassNames` property:

```
b5v.ClassNames

ans =

    'g'
    'b'
```

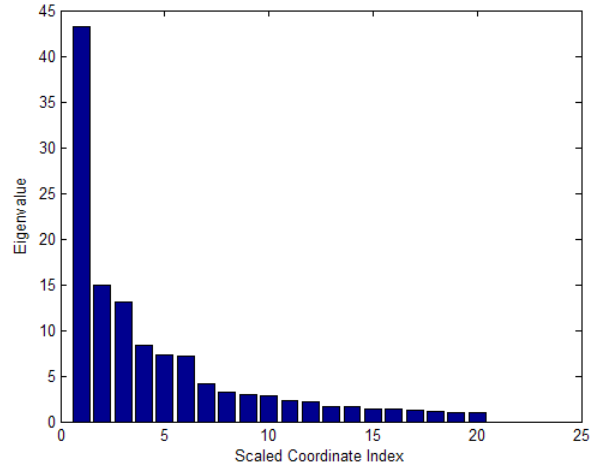
The 'good' class is first and the 'bad' class is second. Display scaled coordinates using red for 'good' and blue for 'bad' observations:

```
figure(17);
[s,e] = mdsProx(b5v,'colors','rb');
xlabel('1st Scaled Coordinate');
ylabel('2nd Scaled Coordinate');
```



10 Plot the first 20 eigenvalues obtained by scaling. The first eigenvalue in this case clearly dominates and the first scaled coordinate is most important.

```
figure(18);
bar(e(1:20));
xlabel('Scaled Coordinate Index');
ylabel('Eigenvalue');
```



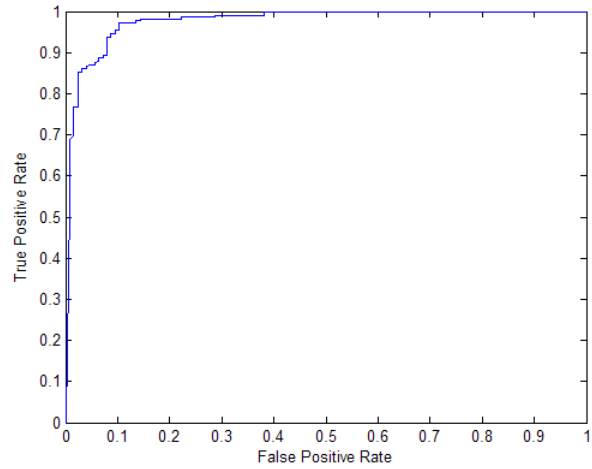
Plotting a Classification Performance Curve. Another way of exploring the performance of a classification ensemble is to plot its Receiver Operating Characteristic (ROC) curve or another performance curve suitable for the current problem. First, obtain predictions for out-of-bag observations. For a classification ensemble, the `oobPredict` method returns a cell array of classification labels (`'g'` or `'b'` for ionosphere data) as the first output argument and a numeric array of scores as the second output argument. The returned array of scores has two columns, one for each class. In this case, the first column is for the `'good'` class and the second column is for the `'bad'` class. One column in the score matrix is redundant because the scores represent class probabilities in tree leaves and by definition add up to 1.

```
[Yfit,Sfit] = oobPredict(b5v);
```

Use the `perfcurve` utility (see "Performance Curves" on page 12-9) to compute a performance curve. By default, `perfcurve` returns the standard ROC curve, which is the true positive rate versus the false positive rate. `perfcurve` requires true class labels, scores, and the positive class label for input. In this case, choose the `'good'` class as positive. The scores for this class are in the first column of `Sfit`.

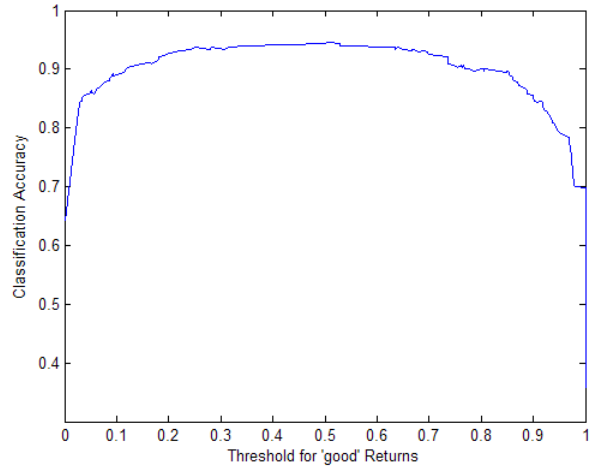
```
[fpr,tpr] = perfcurve(b5v.Y,Sfit(:,1),'g');
figure(19);
plot(fpr,tpr);
```

```
xlabel('False Positive Rate');
ylabel('True Positive Rate');
```



Instead of the standard ROC curve, you might want to plot, for example, ensemble accuracy versus threshold on the score for the `'good'` class. The `ycrit` input argument of `perfcurve` lets you specify the criterion for the *y*-axis, and the third output argument of `perfcurve` returns an array of thresholds for the positive class score. Accuracy is the fraction of correctly classified observations, or equivalently, 1 minus the classification error.

```
[fpr,accu,thre] = perfcurve(b5v.Y,Sfit(:,1),'g','ycrit','accu');
figure(20);
plot(thre,accu);
xlabel('Threshold for ''good'' Returns');
ylabel('Classification Accuracy');
```



The curve shows a flat region indicating that any threshold from 0.2 to 0.6 is a reasonable choice. By default, the function assigns classification labels using 0.5 as the boundary between the two classes. You can find exactly what accuracy this corresponds to:

```
i50 = find(accu>=0.50,1,'first')
accu(abs(thre-0.5)<eps)
```

returns

```
i50 =
    2

ans =
    0.9430
```

The maximal accuracy is a little higher than the default one:

```
[maxaccu,iaccu] = max(accu)
```

returns

```
maxaccu =
    0.9459
```

```
iaccu =
    91
```

The optimal threshold is therefore:

```
thre(iaccu)

ans =
    0.5056
```

Ensemble Algorithms

Bagging

Bagging, which stands for "bootstrap aggregation", is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, generate many bootstrap replicas of this dataset and grow decision trees on these replicas. Obtain each bootstrap replica by randomly selecting `N` observations out of `N` with replacement, where `N` is the dataset size. To find the predicted response of a trained ensemble, take an average over predictions from individual trees.

Bagged decision trees were introduced in MATLAB R2009a as `TreeBagger`. The `fitensemble` function lets you bag in a manner consistent with boosting. An ensemble of bagged trees, either `ClassificationBaggedEnsemble` or `RegressionBaggedEnsemble`, returned by `fitensemble` offers almost the same functionally as `TreeBagger`. Discrepancies between `TreeBagger` and

the new framework are described in detail in TreeBagger Features Not in fitensemble on page 13-120.

Bagging works by training learners on resampled versions of the data. This resampling is usually done by bootstrapping observations, that is, selecting *N* out of *N* observations with replacement for every new learner. In addition, every tree in the ensemble can randomly select predictors for decision splits—a technique known to improve the accuracy of bagged trees.

By default, the minimal leaf sizes for bagged trees are set to `1` for classification and `5` for regression. Trees grown with the default leaf size are usually very deep. These settings are close to optimal for the predictive power of an ensemble. Often you can grow trees with larger leaves without losing predictive power. Doing so reduces training and prediction time, as well as memory usage for the trained ensemble.

Another important parameter is the number of predictors selected at random for every decision split. This random selection is made for every split, and every deep tree involves many splits. By default, this parameter is set to a square root of the number of predictors for classification, and one third of predictors for regression.

Several features of bagged decision trees make them a unique algorithm. Drawing `N` out of `N` observations with replacement omits on average 37% of observations for each decision tree. These are "out-of-bag" observations. You can use them to estimate the predictive power and feature importance. For each observation, you can estimate the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. You can then compare the computed prediction against the observed response for this observation. By comparing the out-of-bag predicted responses against the observed responses for all observations used for training, you can estimate the average out-of-bag error. This out-of-bag average is an unbiased estimator of the true ensemble error. You can also obtain out-of-bag estimates of feature importance by randomly permuting out-of-bag data across one variable or column at a time and estimating the increase in the out-of-bag error due to this permutation. The larger the increase, the more important the feature. Thus, you need not supply test data for bagged ensembles because you obtain reliable estimates of the predictive power and feature importance in the process of training, which is an attractive feature of bagging.

Another attractive feature of bagged decision trees is the proximity matrix. Every time two observations land on the same leaf of a tree, their proximity increases by 1. For normalization, sum these proximities over all trees in the ensemble and divide by the number of trees. The resulting matrix is symmetric with diagonal elements equal to 1 and off-diagonal elements ranging from 0 to 1. You can use this matrix for finding outlier observations and discovering clusters in the data through multidimensional scaling.

For examples using bagging, see:

- "Example: Test Ensemble Quality" on page 13-59
- "Example: Surrogate Splits" on page 13-76
- "Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger" on page 13-97
- "Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger" on page 13-106

For references related to bagging, see Breiman [2], [3], and [4].

Comparison of TreeBagger and Bagged Ensembles. `fitensemble` produces bagged ensembles that have most, but not all, of the functionality of `TreeBagger` objects. Additionally, some functionality has different names in the new bagged ensembles.

TreeBagger Features Not in fitensemble

Feature	TreeBagger Property	TreeBagger Method
Computation of proximity matrix	`Proximity`	`fillProximities`, `mdsProx`
Computation of outliers	`OutlierMeasure`	
Out-of-bag estimates of predictor importance	`OOBPermutedVarDeltaError`, `OOBPermutedVarDeltaMeanMargin`, `OOBPermutedVarCountRaiseMargin`	

TreeBagger Features Not in fitensemble (Continued)

Feature	TreeBagger Property	TreeBagger Method
Merging two ensembles trained separately		`append`
Parallel computation for creating ensemble	Set the `UseParallel` name-value pair to `'always'`; see Chapter 17, "Parallel Statistics"	

Differing Names Between TreeBagger and Bagged Ensembles

Feature	TreeBagger	Bagged Ensembles
Split criterion contributions for each predictor	`DeltaCritDecisionSplit` property	First output of `predictorImportance` (classification) or `predictorImportance` (regression)
Predictor associations	`VarAssoc` property	Second output of `predictorImportance` (classification) or `predictorImportance` (regression)
Error (misclassification probability or mean-squared error)	`error` and `oobError` methods	`loss` and `oobLoss` methods (classification); `loss` and `oobLoss` methods (regression)
Train additional trees and add to ensemble	`growTrees` method	`resume` method (classification); `resume` method (regression)
Mean classification margin per tree	`meanMargin` and `oobMeanMargin` methods	`edge` and `oobEdge` methods (classification);

In addition, two important changes were made to training and prediction for bagged classification ensembles:

- If you pass a misclassification cost matrix to `TreeBagger`, it passes this matrix along to the trees. If you pass a misclassification cost matrix to `fitensemble`, it uses this matrix to adjust the class prior probabilities.

`fitensemble` then passes the adjusted prior probabilities and the default cost matrix to the trees. The default cost matrix is `ones(K)-eye(K)` for `K` classes.

- Unlike the `loss` and `edge` methods in the new framework, the `TreeBagger` `error` and `meanMargin` methods do not normalize input observation weights of the prior probabilities in the respective class.

AdaBoostM1

`AdaBoostM1` is a very popular boosting algorithm for binary classification. The algorithm trains learners sequentially. For every learner with index t, `AdaBoostM1` computes the weighted classification error

$$\varepsilon_t = \sum_{n=1}^{N} d_n^{(t)} \mathbb{I}\left(y_n = h_t\left(x_n\right)\right)$$

- x_n is a vector of predictor values for observation n.
- y_n is the true class label.
- h_t is the prediction of learner (hypothesis) with index t.
- $\mathbb{I}$ is the indicator function.
- $d_n^{(t)}$ is the weight of observation n at step t.

`AdaBoostM1` then increases weights for observations misclassified by learner t and reduces weights for observations correctly classified by learner t. The next learner $t + 1$ is then trained on the data with updated weights $d_n^{(t+1)}$.

After training finishes, `AdaBoostM1` computes prediction for new data using

$$f\left(x\right) = \sum_{t=1}^{T} \alpha_t h_t\left(x\right),$$

where

$$\alpha_t = \frac{1}{2}\log\frac{1-\varepsilon_t}{\varepsilon_t}$$

are weights of the weak hypotheses in the ensemble.

Training by `AdaBoostM1` can be viewed as stagewise minimization of the exponential loss:

$$\sum_{n=1}^{N} w_n \exp\left(-y_n f\left(x_n\right)\right)$$

where

- $y_n \in \{-1,+1\}$ is the true class label.
- w_n are observation weights normalized to add up to 1.
- $f(x_n) \in (-\infty,+\infty)$ is the predicted classification score.

The observation weights w_n are the original observation weights you passed to `fitensemble`.

The second output from the `predict` method of an `AdaBoostM1` classification ensemble is an N-by-2 matrix of classification scores for the two classes and N observations. The second column in this matrix is always equal to minus the first column. `predict` returns two scores to be consistent with multiclass models, though this is redundant since the second column is always the negative of the first.

Most often `AdaBoostM1` is used with decision stumps (default) or shallow trees. If boosted stumps give poor performance, try setting the minimal parent node size to one quarter of the training data.

By default, the learning rate for boosting algorithms is `1`. If you set the learning rate to a lower number, the ensemble learns at a slower rate, but can converge to a better solution. `0.1` is a popular choice for the learning rate. Learning at a rate less than `1` is often called “shrinkage”.

For examples using `AdaBoostM1`, see "Example: Tuning RobustBoost" on page 13-92.

For references related to `AdaBoostM1`, see Freund and Schapire [8], Schapire et al. [13], Friedman, Hastie, and Tibshirani [10], and Friedman [9].

AdaBoostM2

`AdaBoostM2` is an extension of `AdaBoostM1` for multiple classes. Instead of weighted classification error, `AdaBoostM2` uses weighted pseudo-loss for N observations and K classes:

$$\varepsilon_t = \frac{1}{2}\sum_{n=1}^{N}\sum_{k\neq y_n} d_{n,k}^{(t)}\left(1 - h_t\left(x_n, y_n\right) + h_t\left(x_n, k\right)\right)$$

where

- $h_t(x_n,k)$ is the confidence of prediction by learner at step t into class k ranging from 0 (not at all confident) to 1 (highly confident).
- $d_{n,k}^{(t)}$ are observation weights at step t for class k.
- y_n is the true class label taking one of the K values.
- The second sum is over all classes other than the true class y_n.

Interpreting the pseudo-loss is harder than classification error, but the idea is the same. Pseudo-loss can be used as a measure of the classification accuracy from any learner in an ensemble. Pseudo-loss typically exhibits the same behavior as a weighted classification error for `AdaBoostM1`: the first few learners in a boosted ensemble give low pseudo-loss values. After the first few training steps, the ensemble begins to learn at a slower pace, and the pseudo-loss value approaches 0.5 from below.

For examples using `AdaBoostM2`, see "Creating a Classification Ensemble" on page 13-57.

For references related to `AdaBoostM2`, see Freund and Schapire [8].

LogitBoost

`LogitBoost` is another popular algorithm for binary classification. `LogitBoost` works similarly to `AdaBoostM1`, except it minimizes binomial deviance

$$\sum_{n=1}^{N} w_n \log\left(1+\exp\left(-2y_n f\left(x_n\right)\right)\right),$$

where

- $y_n \in \{-1,+1\}$ is the true class label.
- w_n are observation weights normalized to add up to 1.
- $f(x_n) \in (-\infty,+\infty)$ is the predicted classification score.

Binomial deviance assigns less weight to badly misclassified observations (observations with large negative values of $y_n f(x_n)$). `LogitBoost` can give better average accuracy than `AdaBoostM1` for data with poorly separable classes.

Learner t in a `LogitBoost` ensemble fits a regression model to response values

$$\tilde{y}_n = \frac{y_n^* - p_t\left(x_n\right)}{p_t\left(x_n\right)\left(1-p_t\left(x_n\right)\right)}$$

where

- $y^*_n \in \{0,+1\}$ are relabeled classes (0 instead of –1).
- $p_t(x_n)$ is the current ensemble estimate of the probability for observation x_n to be of class 1.

Fitting a regression model at each boosting step turns into a great computational advantage for data with multilevel categorical predictors. Take a categorical predictor with L levels. To find the optimal decision split on such a predictor, classification tree needs to consider $2^{L-1} - 1$ splits. A regression tree needs to consider only $L - 1$ splits, so the processing time

can be much shorter. `LogitBoost` is recommended for categorical predictors with many levels.

`fitensemble` computes and stores the mean-squared error

$$\sum_{n=1}^{N} d_n^{(t)} \left(\tilde{y}_n - h_t(x_n) \right)^2$$

in the `FitInfo` property of the ensemble object. Here

- $d_n^{(t)}$ are observation weights at step t (the weights add up to 1).
- $h_t(x_n)$ are predictions of the regression model h_t fitted to response values $\tilde{y}_n$.

Values y_n can range from $-\infty$ to $+\infty$, so the mean-squared error does not have well-defined bounds.

For examples using `LogitBoost`, see "Example: Classification with Many Categorical Levels" on page 13-71.

For references related to `LogitBoost`, see Friedman, Hastie, and Tibshirani [10].

GentleBoost

`GentleBoost` (also known as Gentle AdaBoost) combines features of `AdaBoostM1` and `LogitBoost`. Like `AdaBoostM1`, `GentleBoost` minimizes the exponential loss. But its numeric optimization is set up differently. Like `LogitBoost`, every weak learner fits a regression model to response values $y_n \in \{-1,+1\}$. This makes `GentleBoost` another good candidate for binary classification of data with multilevel categorical predictors.

`fitensemble` computes and stores the mean-squared error

$$\sum_{n=1}^{N} d_n^{(t)} \left(\tilde{y}_n - h_t(x_n) \right)^2$$

in the `FitInfo` property of the ensemble object, where

- $d_n^{(t)}$ are observation weights at step t (the weights add up to 1).
- $h_t(x_n)$ are predictions of the regression model h_t fitted to response values y_n.

As the strength of individual learners weakens, the weighted mean-squared error approaches 1.

For examples using `GentleBoost`, see "Example: Unequal Classification Costs" on page 13-66, "Example: Classification with Many Categorical Levels" on page 13-71.

For references related to `GentleBoost`, see Friedman, Hastie, and Tibshirani [10].

RobustBoost

Boosting algorithms such as `AdaBoostM1` and `LogitBoost` increase weights for misclassified observations at every boosting step. These weights can become very large. If this happens, the boosting algorithm sometimes concentrates on a few misclassified observations and neglects the majority of training data. Consequently the average classification accuracy suffers.

In this situation, you can try using `RobustBoost`. This algorithm does not assign almost the entire data weight to badly misclassified observations. It can produce better average classification accuracy.

Unlike `AdaBoostM1` and `LogitBoost`, `RobustBoost` does not minimize a specific loss function. Instead, it maximizes the number of observations with the classification margin above a certain threshold.

`RobustBoost` trains based on time evolution. The algorithm starts at $t = 0$. At every step, `RobustBoost` solves an optimization problem to find a positive step in time Δt and a corresponding positive change in the average margin for training data Δm. `RobustBoost` stops training and exits if at least one of these three conditions is true:

- Time t reaches 1.

- `RobustBoost` cannot find a solution to the optimization problem with positive updates Δt and Δm.
- `RobustBoost` grows as many learners as you requested.

Results from `RobustBoost` can be usable for any termination condition. Estimate the classification accuracy by cross validation or by using an independent test set.

To get better classification accuracy from `RobustBoost`, you can adjust three parameters in `fitensemble`: `RobustErrorGoal`, `RobustMaxMargin`, and `RobustMarginSigma`. Start by varying values for `RobustErrorGoal` from 0 to 1. The maximal allowed value for `RobustErrorGoal` depends on the two other parameters. If you pass a value that is too high, `fitensemble` produces an error message showing the allowed range for `RobustErrorGoal`.

For examples using `RobustBoost`, see "Example: Tuning RobustBoost" on page 13-92

For references related to `RobustBoost`, see Freund [7].

LSBoost

You can use least squares boosting (`LSBoost`) to fit regression ensembles. At every step, the ensemble fits a new learner to the difference between the observed response and the aggregated prediction of all learners grown previously. The ensemble fits to minimize mean-squared error.

You can use `LSBoost` with shrinkage by passing in `LearnRate` parameter. By default this parameter is set to `1`, and the ensemble learns at the maximal speed. If you set `LearnRate` to a value from `0` to `1`, the ensemble fits every new learner to $y_n - \eta f(x_n)$, where

- y_n is the observed response.
- $f(x_n)$ is the aggregated prediction from all weak learners grown so far for observation x_n.
- η is the learning rate.

For examples using `LSBoost`, see "Creating a Regression Ensemble" on page 13-58, "Example: Regularizing a Regression Ensemble" on page 13-82

For references related to `LSBoost`, see Hastie, Tibshirani, and Friedman [11], Chapters 7 (Model Assessment and Selection) and 15 (Random Forests).

Bibliography

[1] Bottou, L., and Chih-Jen Lin. *Support Vector Machine Solvers*. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.4209&rep=rep1&type=pdf`.

[2] Breiman, L. *Bagging Predictors.* Machine Learning 26, pp. 123–140, 1996.

[3] Breiman, L. *Random Forests.* Machine Learning 45, pp. 5–32, 2001.

[4] Breiman, L. `http://www.stat.berkeley.edu/~breiman/RandomForests/`

[5] Breiman, L., et al. *Classification and Regression Trees.* Chapman & Hall, Boca Raton, 1993.

[6] Christianini, N., and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, Cambridge, UK, 2000.

[7] Freund, Y. *A more robust boosting algorithm.* arXiv:0905.2138v1, 2009.

[8] Freund, Y. and R. E. Schapire. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting.* J. of Computer and System Sciences, Vol. 55, pp. 119–139, 1997.

[9] Friedman, J. *Greedy function approximation: A gradient boosting machine.* Annals of Statistics, Vol. 29, No. 5, pp. 1189–1232, 2001.

[10] Friedman, J., T. Hastie, and R. Tibshirani. *Additive logistic regression: A statistical view of boosting.* Annals of Statistics, Vol. 28, No. 2, pp. 337–407, 2000.

[11] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. Springer, New York, 2008.

[12] Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*. Available at `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[13] Schapire, R. E. et al. *Boosting the margin: A new explanation for the effectiveness of voting methods.* Annals of Statistics, Vol. 26, No. 5, pp. 1651–1686, 1998.

[14] Zadrozny, B., J. Langford, and N. Abe. *Cost-Sensitive Learning by Cost-Proportionate Example Weighting.* CiteSeerX. [Online] 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.9780

[15] Zhou, Z.-H. and X.-Y. Liu. *On Multi-Class Cost-Sensitive Learning.* CiteSeerX. [Online] 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.9999