

Decision Trees

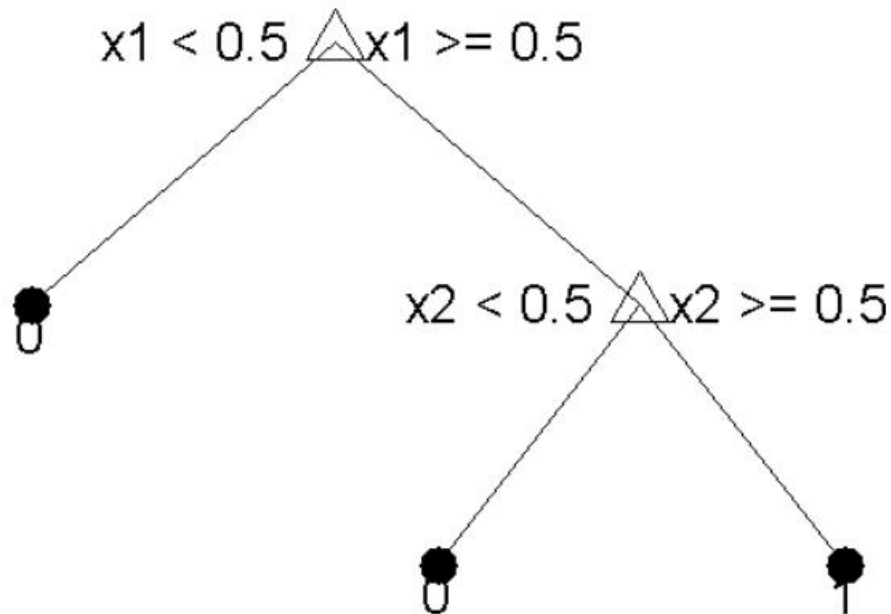
- trees and forests

Decision Trees

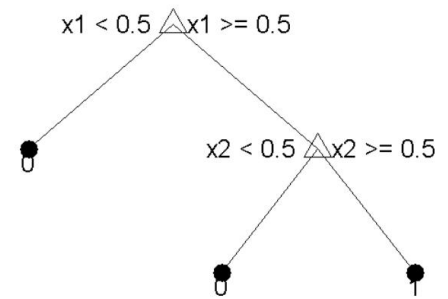
- Decision trees, or classification trees and regression trees, predict responses to data. To predict a response, follow the decisions in the tree from the root (beginning) node down to a leaf node. The leaf node contains the response. Classification trees give responses that are nominal, such as 'true' or 'false'. Regression trees give numeric responses.
- Each step in a prediction involves checking the value of one predictor (variable).

Decision Trees

- For example, here is a simple classification tree:



Decision Trees



- This tree predicts classifications based on two predictors, x_1 and x_2 . To predict, start at the top node, represented by a triangle (Δ). The first decision is whether x_1 is smaller than 0.5. If so, follow the left branch, and see that the tree classifies the data as type 0.
- If, however, x_1 exceeds 0.5, then follow the right branch to the lower-right triangle node. Here the tree asks if x_2 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 tree classifies the data as type 1.

Train Classification Tree

- This example shows how to train a classification tree.
- Create a classification tree using the entire ionosphere data set.

load ionosphere % Contains X and Y variables

```
Mdl = fitctree(X,Y)
```

Mdl =

ClassificationTree

ResponseName: 'Y'

CategoricalPredictors: []

ClassNames: {'b' 'g'}

ScoreTransform: 'none'

NumObservations: 351

Properties, Methods

Train Classification Tree

- Data Set Information:

This radar data was collected by a system in Goose Bay, Labrador. This system consists of a phased array of 16 high-frequency antennas with a total transmitted power on the order of 6.4 kilowatts. The targets were free electrons in the ionosphere. "Good" radar returns are those showing evidence of some type of structure in the ionosphere. "Bad" returns are those that do not; their signals pass through the ionosphere.

- Attribute Information:

- All 34 are continuous

- The 35th attribute is either "good" or "bad" according to the definition summarized above. This is a binary classification task.

Train Regression Tree

- This example shows how to train a regression tree.
- Create a regression tree using all observation in the carsmall data set. Consider the Horsepower and Weight vectors as predictor variables, and the MPG vector as the response.

```
load carsmall % Contains Horsepower, Weight, MPG
```

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

```
Mdl = fitrtree(X,MPG)
```

```
Mdl =
```

```
RegressionTree
```

```
ResponseName: 'Y'
```

```
CategoricalPredictors: []
```

```
ResponseTransform: 'none'
```

```
NumObservations: 94
```

```
Properties, Methods
```

View Decision Tree

- This example shows how to view a classification or regression tree. There are two ways to view a tree: `view(tree)` returns a text description and `view(tree,'mode','graph')` returns a graphic description of the tree.
- Create and view a classification tree.

```
load fisheriris % load the sample data
```

```
ctree = fitctree(meas,species); % create classification tree
```


View Decision Tree

`view(ctree) % text description`

Decision tree for classification

1 if $x_3 < 2.45$ then node 2 elseif $x_3 \geq 2.45$ then node 3 else setosa

2 class = setosa

3 if $x_4 < 1.75$ then node 4 elseif $x_4 \geq 1.75$ then node 5 else versicolor

4 if $x_3 < 4.95$ then node 6 elseif $x_3 \geq 4.95$ then node 7 else versicolor

5 class = virginica

6 if $x_4 < 1.65$ then node 8 elseif $x_4 \geq 1.65$ then node 9 else versicolor

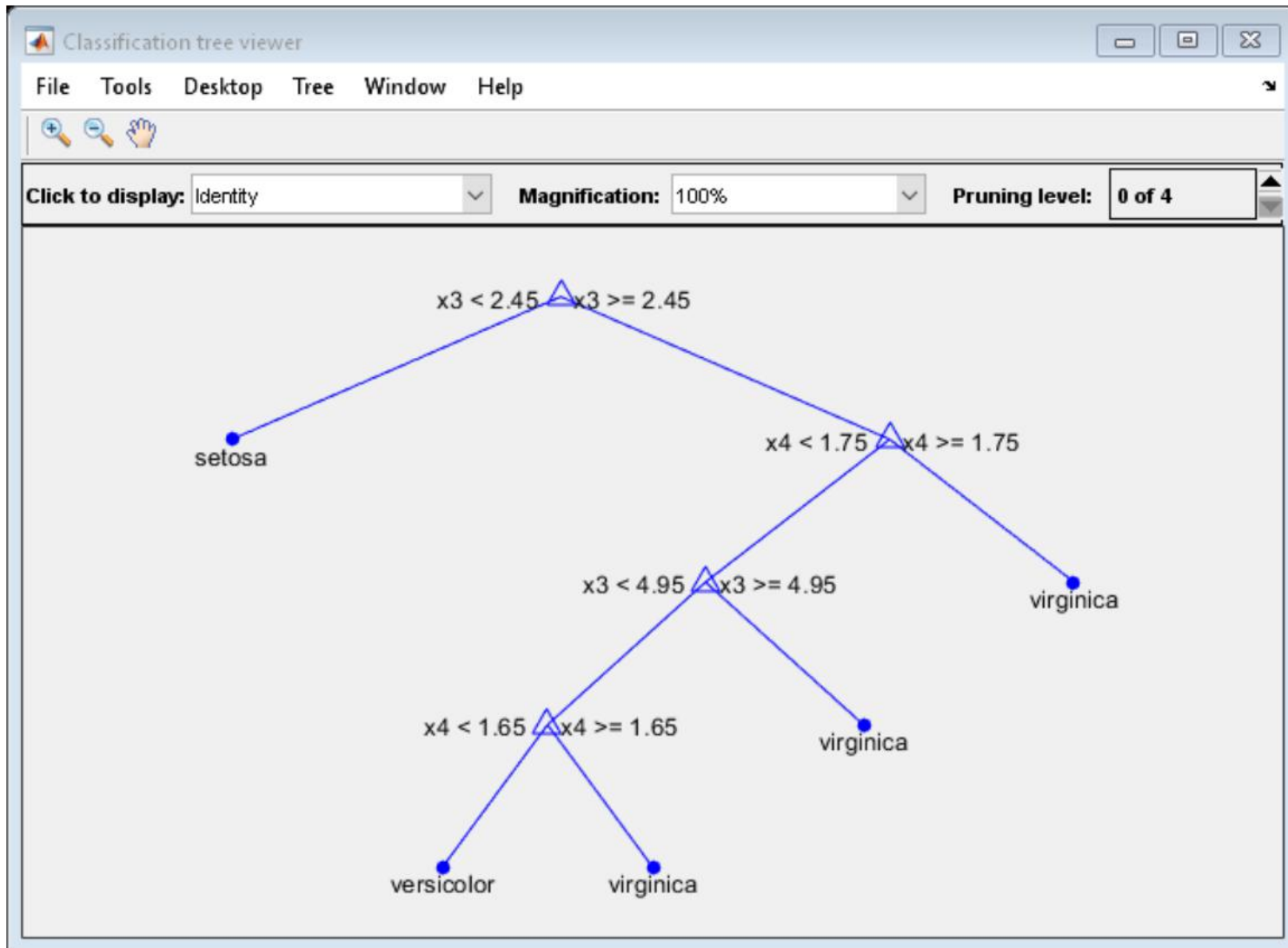
7 class = virginica

8 class = versicolor

9 class = virginica

View Decision Tree

```
view(ctree,'mode','graph') % graphic description
```



view a regression tree

- load carsmall % load the sample data, contains Horsepower, Weight, MPG

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

```
rtree = fitrtree(X,MPG,'MinParent',30); % create classification tree
```

```
view(rtree) % text description
```

Decision tree for regression

1 if $x_2 < 3085.5$ then node 2 elseif $x_2 \geq 3085.5$ then node 3 else 23.7181

2 if $x_1 < 89$ then node 4 elseif $x_1 \geq 89$ then node 5 else 28.7931

3 if $x_1 < 115$ then node 6 elseif $x_1 \geq 115$ then node 7 else 15.5417

4 if $x_2 < 2162$ then node 8 elseif $x_2 \geq 2162$ then node 9 else 30.9375

5 fit = 24.0882

6 fit = 19.625

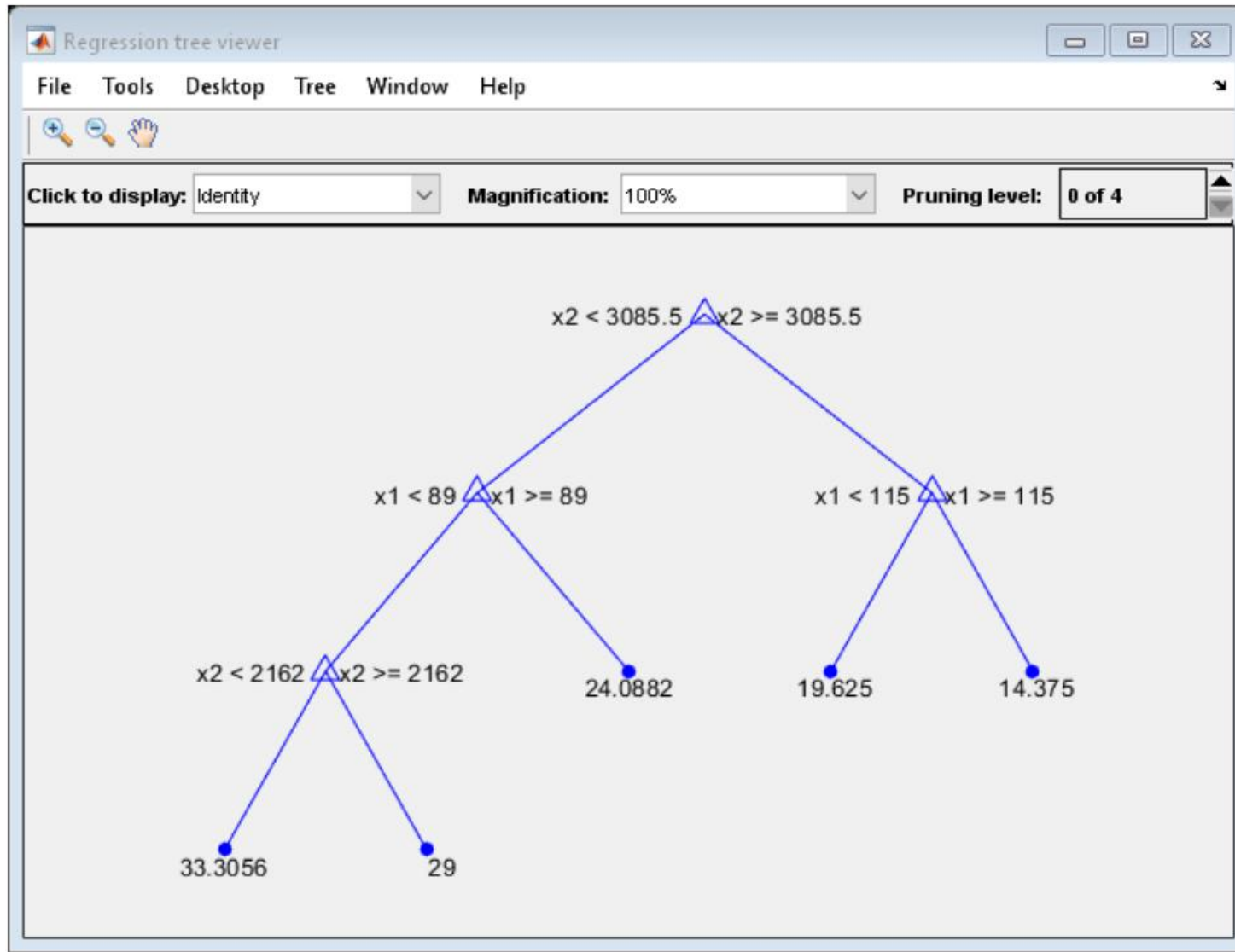
7 fit = 14.375

8 fit = 33.3056

9 fit = 29

view a regression tree

```
view(rtree,'mode','graph') % graphic description
```



Growing Decision Trees

By default, `fitctree` and `fitrtree` use the standard CART algorithm to create decision trees. That is, they perform the following steps:

- 1 Start with all input data, and examine all possible binary splits on every predictor.
- 2 Select a split with best optimization criterion.
- 3 Impose the split.
- 4 Repeat recursively for the two child nodes.

Growing Decision Trees

- 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)
- 'deviance'
- 'twoing'

Growing Decision Trees

- Impurity

ClassificationTree splits nodes based on either impurity or node error. Impurity means one of several things, depending on your choice of the ***SplitCriterion*** name-value pair argument:

- Gini's Diversity Index (gdi) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes i at the node, and $p(i)$ is the observed fraction of classes with class i that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

Growing Decision Trees

- Impurity

ClassificationTree splits nodes based on either impurity or node error. Impurity means one of several things, depending on your choice of the ***SplitCriterion*** name-value pair argument:

- Deviance ('deviance') — With $p(i)$ defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

Growing Decision Trees

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let $L(i)$ denote the fraction of members of class i in the left child node after a split, and $R(i)$ denote the fraction of members of class i in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R)\left(\sum_i |L(i) - R(i)|\right)^2,$$

where $P(L)$ and $P(R)$ are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and therefore similar to the parent node. The split did not increase node purity.

Prediction Using Trees

- This example shows how to predict class labels or responses using trained classification and regression trees.
- After creating a tree, you can easily predict responses for new data. Suppose X_{new} 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 (Mdl) and the new data, enter

$Y_{\text{new}} = \text{predict}(Mdl, X_{\text{new}})$

- For each row of data in X_{new} , `predict` runs through the decisions in Mdl and gives the resulting prediction in the corresponding element of Y_{new} .

Prediction Using Trees

- For example, find the predicted classification of a point at the mean of the ionosphere data.

```
load ionosphere
```

```
CMdl = fitctree(X,Y);
```

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

```
Ynew = 1x1 cell array
```

```
{'g'}
```

- Find the predicted MPG of a point at the mean of the carsmall data.

```
load carsmall
```

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

```
RMdl = fitrtree(X,MPG);
```

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

```
Ynew = 28.7931
```

Predict Out-of-Sample Responses

- This example shows how to predict out-of-sample responses of regression trees, and then plot the results.
- Load the carsmall data set. Consider Weight as a predictor of the response MPG.

```
load carsmall
```

```
idxNaN = isnan(MPG + Weight);
```

```
X = Weight(~idxNaN);
```

```
Y = MPG(~idxNaN);
```

```
n = numel(X);
```

- Partition the data into training (50%) and validation (50%) sets.

```
rng(1) % For reproducibility
```

```
idxTrn = false(n,1);
```

```
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices
```

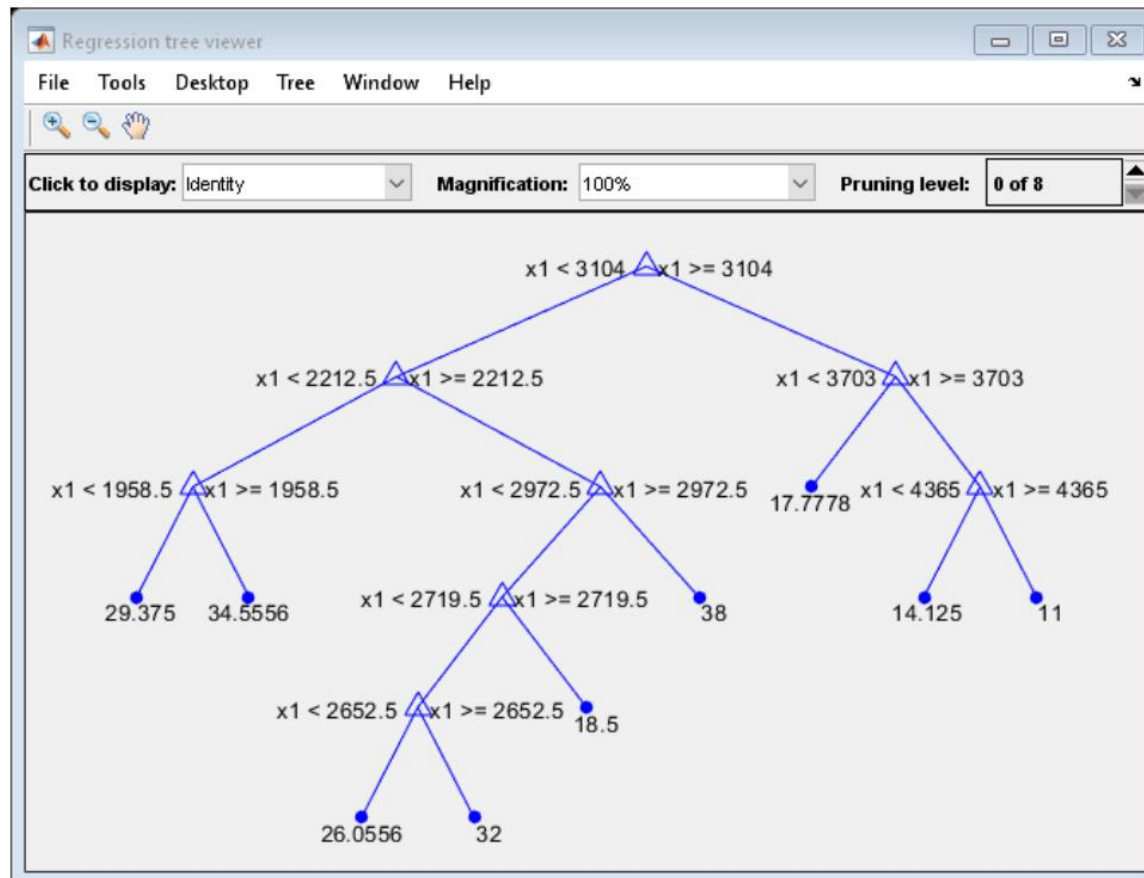
```
idxVal = idxTrn == false; % Validation set logical indices
```

Predict Out-of-Sample Responses

- Grow a regression tree using the training observations.

```
Mdl = fitrtree(X(idxTrn),Y(idxTrn));
```

```
view(Mdl,'Mode','graph')
```



Predict Out-of-Sample Responses

- Compute fitted values of the validation observations for each of several subtrees.

```
m = max(Mdl.PruneList);  
pruneLevels = 0:2:m; % Pruning levels to consider  
z = numel(pruneLevels);  
Yfit = predict(Mdl,X(idxVal),'SubTrees',pruneLevels);
```

- Yfit is an n-by- z matrix of fitted values in which the rows correspond to observations and the columns correspond to a subtree.

Predict Out-of-Sample Responses

- Plot Yfit and Y against X.

```
figure;
```

```
sortDat = sortrows([X(idxVal) Y(idxVal) Yfit],1); % Sort all data with respect to X
```

```
plot(sortDat(:,1),sortDat(:,2),'*');
```

```
hold on;
```

```
plot(repmat(sortDat(:,1),1,size(Yfit,2)),sortDat(:,3:end));
```

```
lev = cellstr(num2str((pruneLevels),'Level %d MPG'));
```

```
legend(['Observed MPG'; lev])
```

```
title 'Out-of-Sample Predictions'
```

```
xlabel 'Weight (lbs)';
```

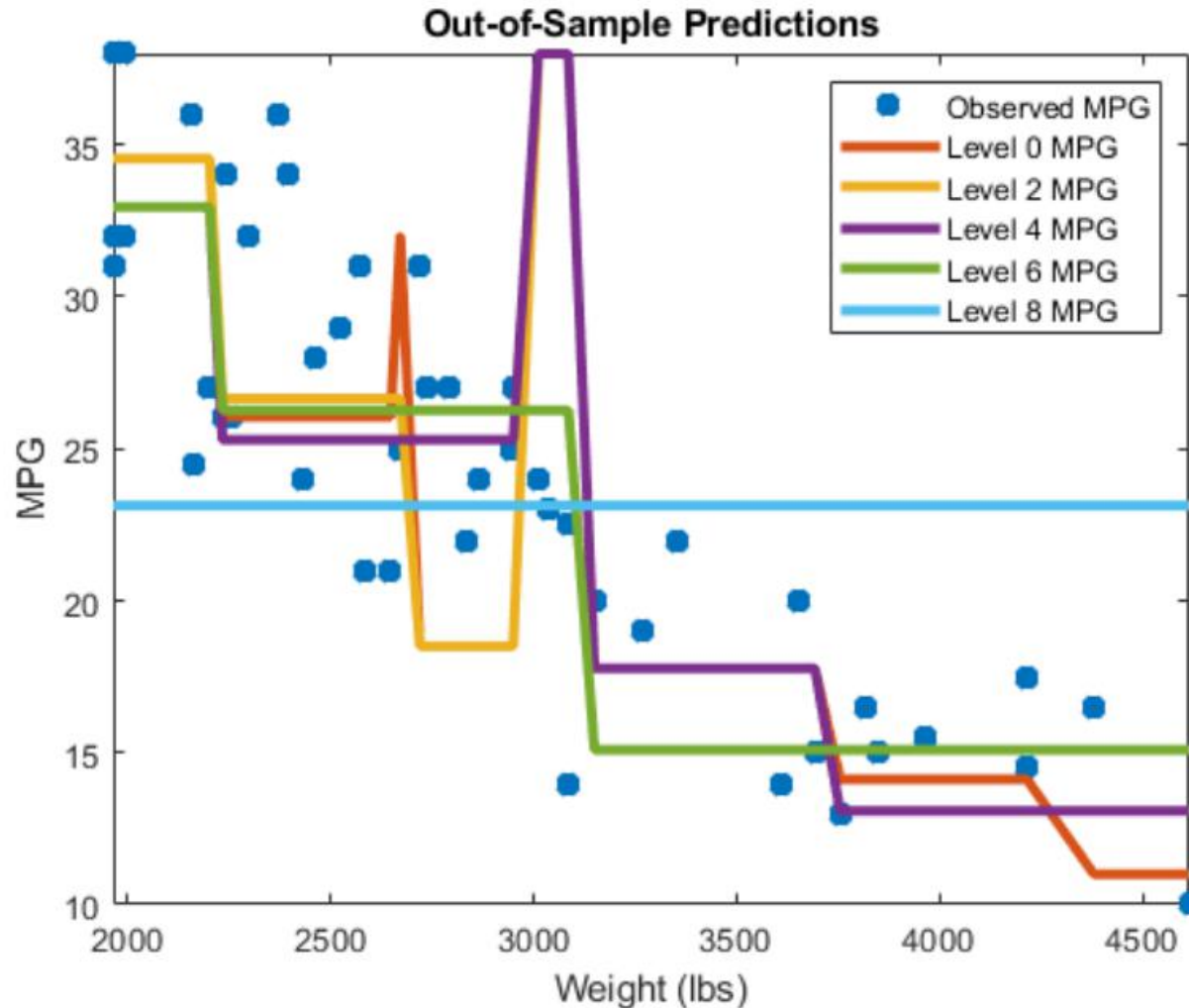
```
ylabel 'MPG';
```

```
h = findobj(gcf);
```

```
axis tight;
```

```
set(h(4:end),'LineWidth',3) % Widen all lines
```

Predict Out-of-Sample Responses



The values of Y_{fit} for lower pruning levels tend to follow the data more closely than higher levels. Higher pruning levels tend to be flat for large X intervals.

Improving Trees

- 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.

Improving Trees

- Examining Resubstitution Error

- This example shows how to examine the resubstitution error of a classification tree.

- Load Fisher's iris data.

```
load fisheriris
```

- Train a default classification tree using the entire data set.

```
Mdl = fitctree(meas,species);
```

- Examine the resubstitution error.

```
resuberror = resubLoss(Mdl)
```

```
resuberror = 0.0200
```

- The tree classifies nearly all the Fisher iris data correctly.

Improving Trees

● 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.

Improving Trees

● Cross Validation

- This example shows how to examine the resubstitution and cross-validation accuracy of a regression tree for predicting mileage based on the carsmall data.
- Load the carsmall data set. Consider acceleration, displacement, horsepower, and weight as predictors of MPG.

```
load carsmall
```

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

- Grow a regression tree using all of the observations.

```
rtree = fitrtree(X,MPG);
```

Improving Trees

- Cross Validation

- Compute the in-sample error.

```
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.

Improving Trees

- Cross Validation

- Estimate the cross-validation MSE.

```
rng 'default';  
cvrtree = crossval(rtree);  
cvloss = kfoldLoss(cvrtree)  
cvloss = 25.6450
```

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.

Improving Trees

- Split Predictor Selection

The standard CART algorithm tends to select continuous predictors that have many levels. Sometimes, such a selection can be spurious and can also mask more important predictors that have fewer levels, such as categorical predictors. That is, the **predictor selection** process at each node is biased. Also, standard CART tends to miss the important interactions between pairs of predictors and the response.

Improving Trees

- Split Predictor Selection

To mitigate selection bias and increase detection of important interactions, you can specify usage of the curvature or interaction tests using the **'PredictorSelection'** name-value pair argument. Using the curvature or interaction test has the added advantage of producing better predictor importance estimates than standard CART.

Improving Trees

Technique	'Predictor Selection' Value	Description	Training speed	When to specify
Standard CART [1]	Default	Selects the split predictor that maximizes the split-criterion gain over all possible splits of all predictors.	Baseline for comparison	Specify if any of these conditions are true: <ul style="list-style-type: none">• All predictors are continuous• Predictor importance is not the analysis goal• For boosting decision trees

[1] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and Regression Trees. Boca Raton, FL: Chapman & Hall, 1984.

Technique	'PredictorSelection' Value	Description	Training speed	When to specify
Curvature test [2][3]	'curvature'	Selects the split predictor that minimizes the p -value of chi-square tests of independence between each predictor and the response.	Comparable to standard CART	Specify if any of these conditions are true: <ul style="list-style-type: none"> The predictor variables are heterogeneous Predictor importance is an analysis goal Enhance tree interpretation
Interaction test [3]	'interaction-curvature'	Chooses the split predictor that minimizes the p -value of chi-square tests of independence between each predictor and the response (that is, conducts curvature tests), and that minimizes the p -value of a chi-square test of independence between each pair of predictors and response.	Slower than standard CART, particularly when data set contains many predictor variables.	Specify if any of these conditions are true: <ul style="list-style-type: none"> The predictor variables are heterogeneous You suspect associations between pairs of predictors and the response Predictor importance is an analysis goal Enhance tree interpretation

Improving Trees

- Control Depth

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 (or overfit), 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.

Improving Trees

● Control Depth

`fitctree` and `fitrtree` have three name-value pair arguments that control the depth of resulting decision trees:

- **MaxNumSplits** — The maximal number of branch node splits is `MaxNumSplits` per tree. Set a large value for `MaxNumSplits` to get a deep tree. The default is $\text{size}(X,1) - 1$.
- **MinLeafSize** — Each leaf has at least `MinLeafSize` observations. Set small values of `MinLeafSize` to get deep trees. The default is 1.
- **MinParentSize** — Each branch node in the tree has at least `MinParentSize` observations. Set small values of `MinParentSize` to get deep trees. The default is 10.

Improving Trees

- Control Depth

- If you specify `MinParentSize` and `MinLeafSize`, the learner uses the setting that yields trees with larger leaves (i.e., shallower trees):

$$\text{MinParent} = \max(\text{MinParentSize}, 2 * \text{MinLeafSize})$$

- If you supply `MaxNumSplits`, the software splits a tree until one of the three splitting criteria is satisfied.

Improving Trees

- Select Appropriate Tree Depth

- This example shows how to control the depth of a decision tree, and how to choose an appropriate depth.
- Load the ionosphere data.

load ionosphere

- Generate an exponentially spaced set of values from 10 through 100 that represent the minimum number of observations per leaf node.

leafs = logspace(1,2,10);

Improving Trees

- Select Appropriate Tree Depth

```
rng('default')
```

```
N = numel(leafs);
```

```
err = zeros(N,1);
```

```
for n=1:N
```

```
    t = fitctree(X,Y,'CrossVal','On',...
```

```
    'MinLeafSize',leafs(n));
```

```
    err(n) = kfoldLoss(t);
```

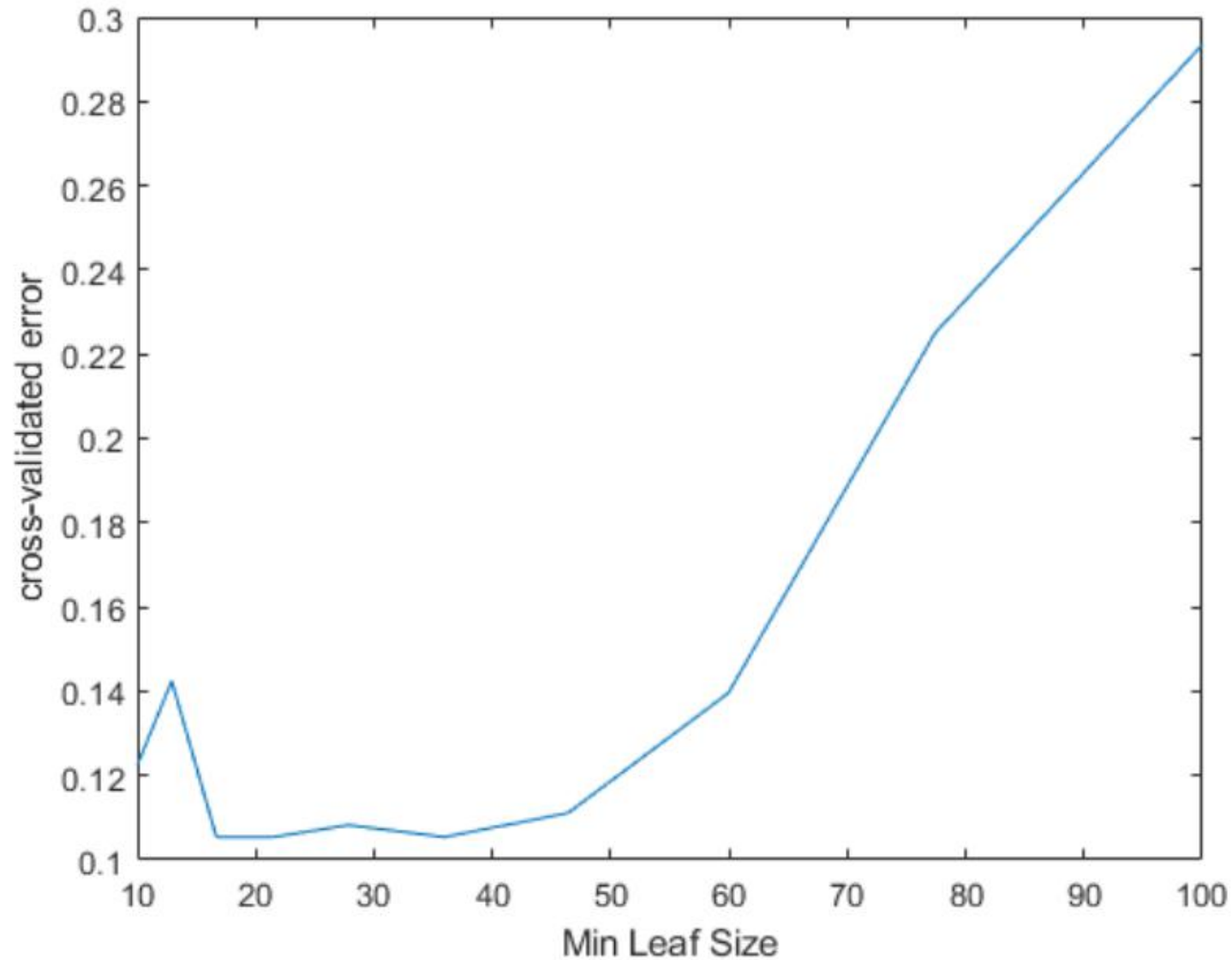
```
end
```

```
plot(leafs,err);
```

```
xlabel('Min Leaf Size');
```

```
ylabel('cross-validated error');
```

Improving Trees



Improving Trees

- Select Appropriate Tree Depth

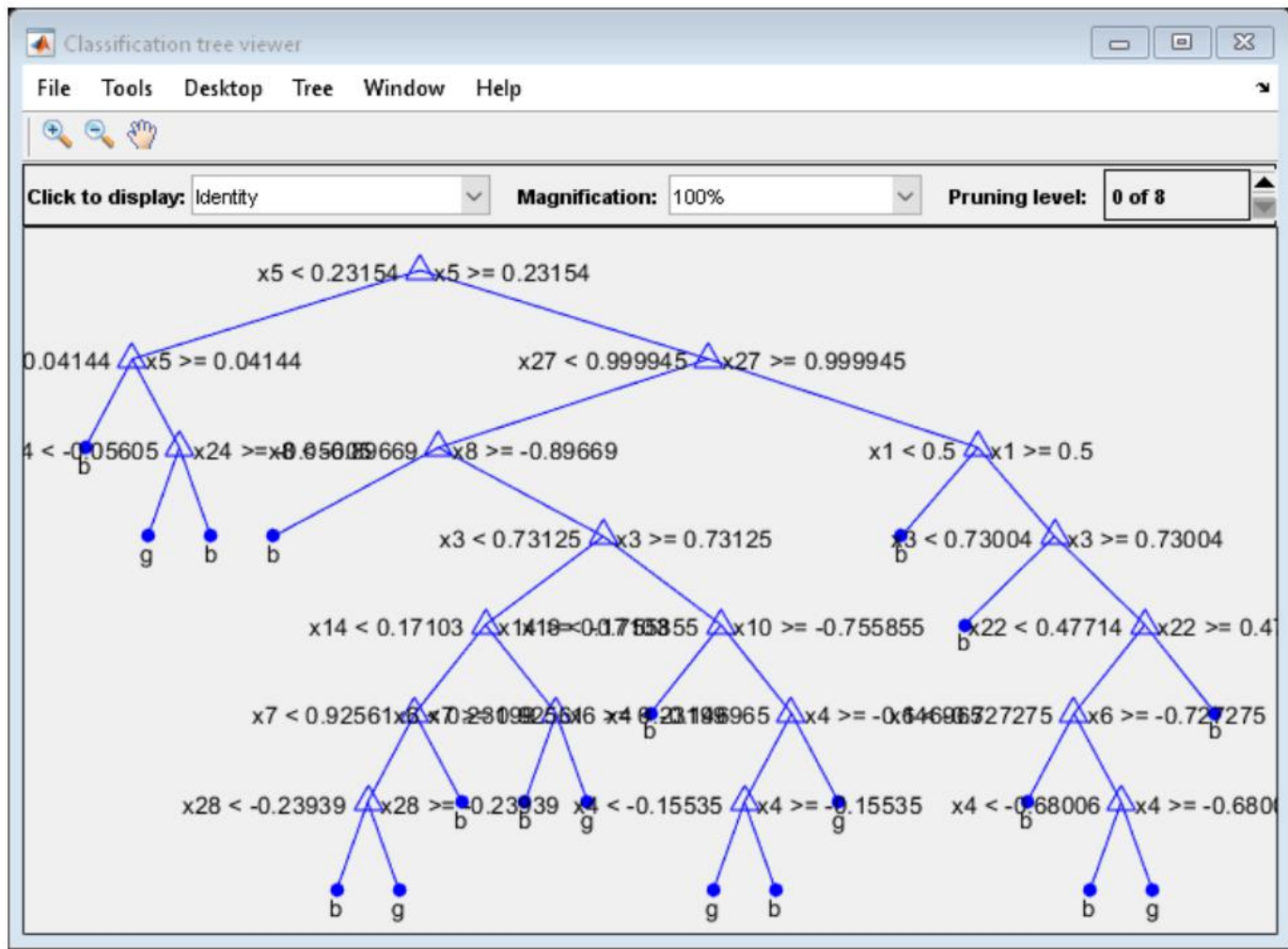
Compare the near-optimal tree with at least 40 observations per leaf with the default tree, which uses 10 observations per parent node and 1 observation per leaf.

```
DefaultTree = fitctree(X,Y);
```

```
view(DefaultTree,'Mode','Graph')
```

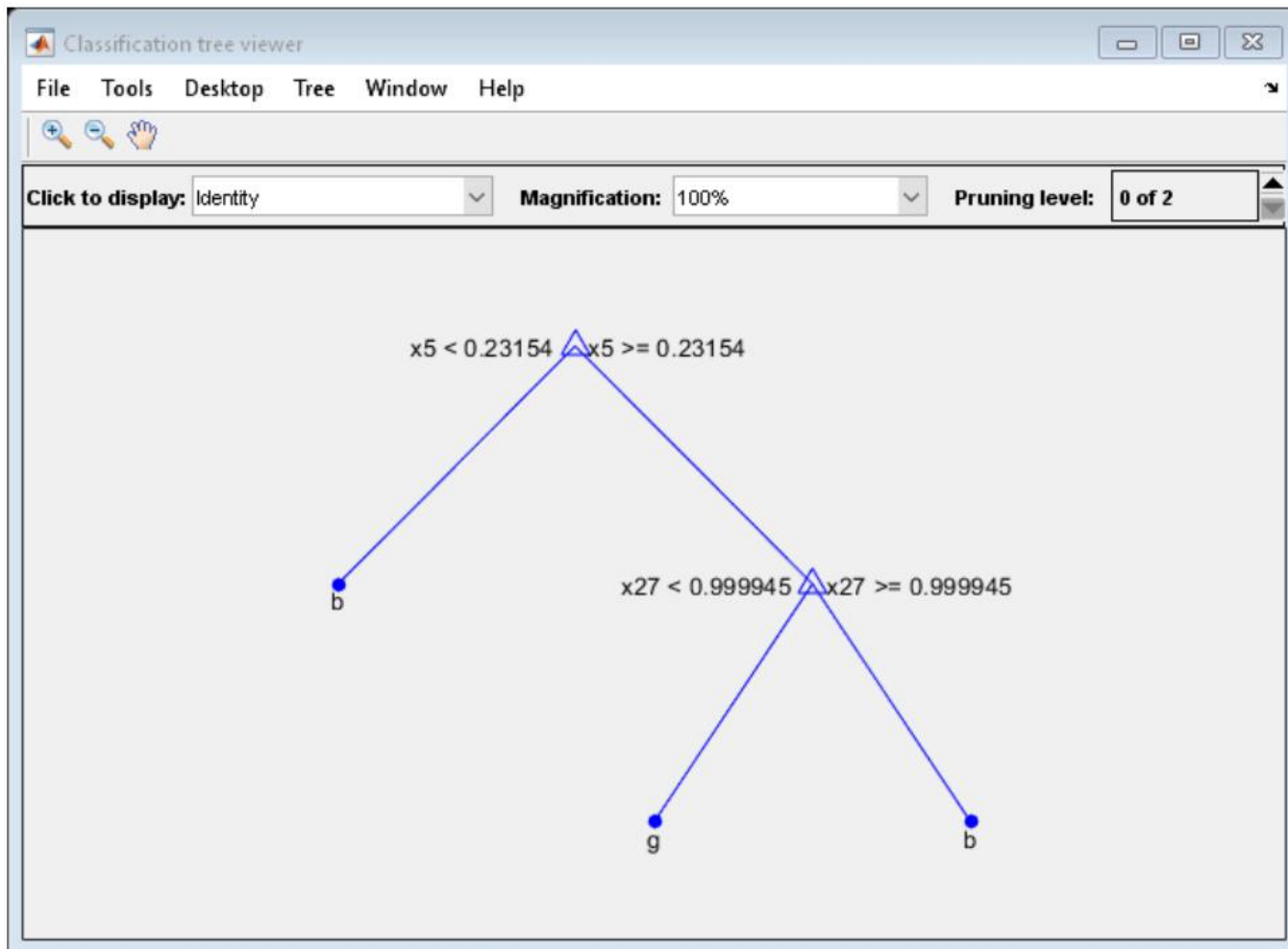
Improving Trees

```
DefaultTree = fitctree(X,Y);  
view(DefaultTree,'Mode','Graph')
```



Improving Trees

```
OptimalTree = fitctree(X,Y,'MinLeafSize',40);  
view(OptimalTree,'mode','graph')
```



Improving Trees

```
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 = 0.1111
```

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

Improving Trees

● Pruning

Pruning optimizes tree depth (leafiness) by merging leaves on the same tree branch. “Control Depth or “Leafiness”” on page 19-21 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. Alternatively, prune a tree interactively with the tree viewer:

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

Improving Trees

● Pruning

To prune a tree, the tree must contain a pruning sequence. By default, both `fitctree` and `fitrtree` 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.

Improving Trees

- Prune a Classification Tree

This example creates a classification tree for the ionosphere data, and prunes it to a good level.

- Load the ionosphere data:

```
load ionosphere
```

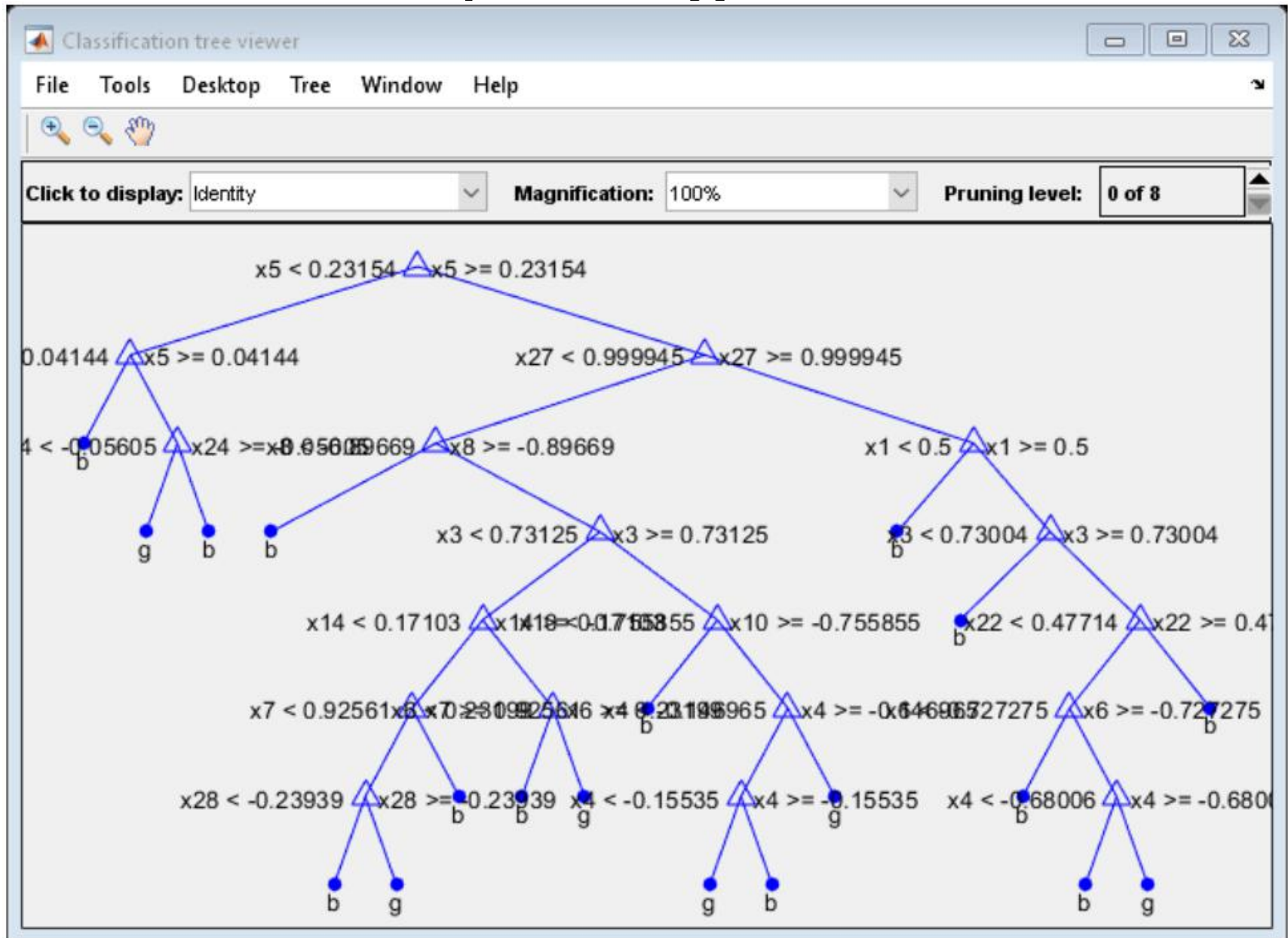
- Construct a default classification tree for the data:

```
tree = fitctree(X,Y);
```

- View the tree in the interactive viewer:

```
view(tree,'Mode','Graph')
```

Improving Trees



Improving Trees

- Prune a Classification Tree

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

```
[~,~,~,bestlevel] = cvLoss(tree,...
```

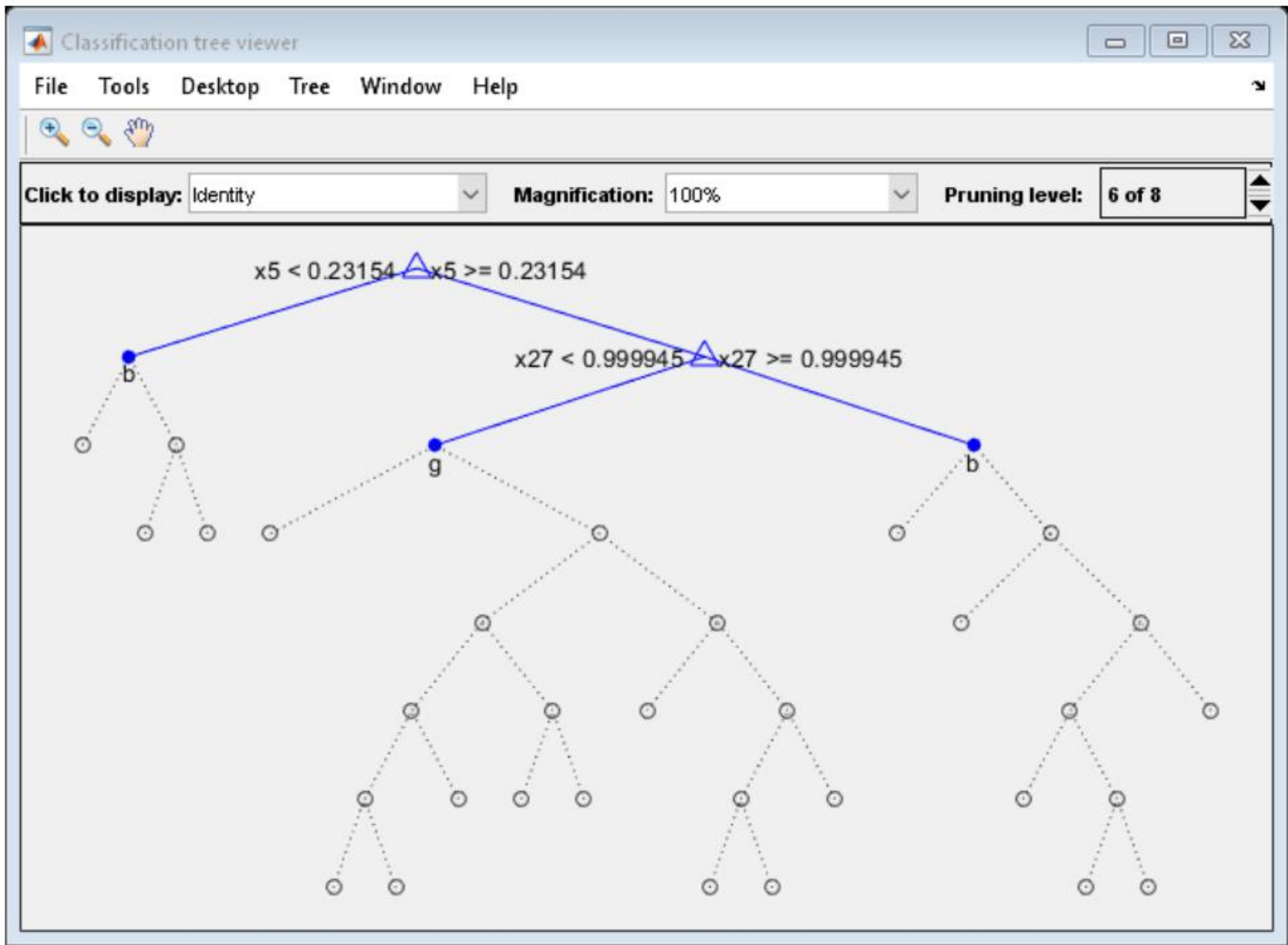
```
'SubTrees','All','TreeSize','min')
```

```
bestlevel = 6
```

- Prune the tree to level 6:

```
view(tree,'Mode','Graph','Prune',6)
```

Improving Trees



Improving Trees

- Prune a Classification Tree

- 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')
```

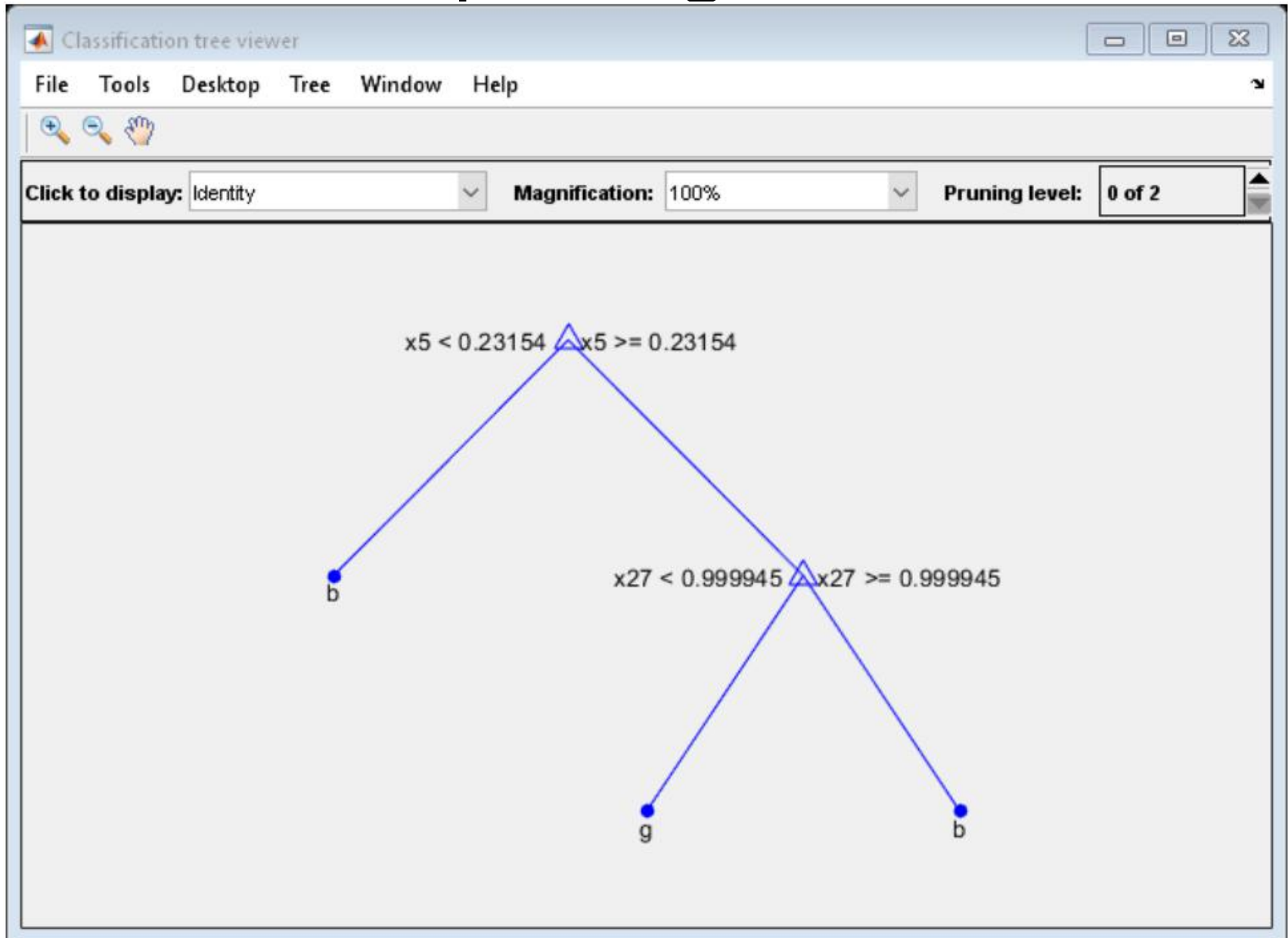
```
bestlevel = 6
```

- In this case the level is the same for either setting of 'TreeSize'.
- Prune the tree to use it for other purposes:

```
tree = prune(tree,'Level',6);
```

```
view(tree,'Mode','Graph')
```

Improving Trees



Challenges in Splitting Multilevel Predictors

When you grow a classification tree, finding an optimal binary split for a categorical predictor with many levels is more computationally challenging than finding a split for a continuous predictor. For a continuous predictor, a tree can split halfway between any two adjacent, unique values of the predictor. In contrast, to find an exact, optimal binary split for a categorical predictor with L levels, a classification tree must consider $2^{L-1}-1$ splits.

Challenges in Splitting Multilevel Predictors

To obtain this formula:

- 1 Count the number of ways to assign L distinct values to the left and right nodes. There are 2^L ways.
- 2 Divide 2^L by 2, because left and right can be swapped.
- 3 Discard the case that has an empty node.

Challenges in Splitting Multilevel Predictors

For regression problems and binary classification problems, the software uses the exact search algorithm through a computational shortcut. The tree can order the categories by mean response (for regression) or class probability for one of the classes (for classification). Then, the optimal split is one of the $L - 1$ splits for the ordered list. Therefore, computational challenges arise only when you grow classification trees for data with $K \geq 3$ classes.

Algorithms for Categorical Predictor Split

To reduce computation, the software offers several heuristic algorithms for finding a good split. You can choose an algorithm for splitting categorical predictors by using the 'AlgorithmForCategorical' name-value pair argument when you grow a classification tree using `fitctree` or when you create a classification learner using `templateTree` for a classification ensemble (`fitcensemble`) or a multiclass ECOC model (`fitcecoc`).

Algorithms for Categorical Predictor Split

If you do not specify an algorithm, the software selects the optimal algorithm for each split using the known number of classes and levels of a categorical predictor. If the predictor has at most `MaxNumCategories` levels, the software splits categorical predictors using the exact search algorithm.

Otherwise, the software chooses a heuristic search algorithm based on the number of classes and levels. The default `MaxNumCategories` level is 10. Depending on your platform, the software cannot perform an exact search on categorical predictors with more than 32 or 64 levels.

Available heuristic algorithms

- Pull Left by Purity

The pull left by purity algorithm starts with all L categorical levels on the right branch.

The algorithm then takes these actions:

- 1 Inspect the K categories that have the largest class probabilities for each class.
- 2 Move the category with the maximum value of the split criterion to the left branch.
- 3 Continue moving categories from right to left, recording the split criterion at each move, until the right child has only one category remaining.

Out of this sequence, the selected split is the one that maximizes the split criterion.

Available heuristic algorithms

- Principal Component-Based Partitioning

The principal component-based partitioning algorithm[2] finds a close-to-optimal binary partition of the L predictor levels by searching for a separating hyperplane. The hyperplane is perpendicular to the first principal component of the weighted covariance matrix of the centered class probability matrix.

The algorithm assigns a score to each of the L categories, computed as the inner product between the found principal component and the vector of class probabilities for that category. Then, the selected split is one of the $L - 1$ splits that maximizes the split criterion.

Available heuristic algorithms

- One-Versus-All by Class

The one-versus-all by class algorithm starts with all L categorical levels on the right branch. For each of the K classes, the algorithm orders the categories based on their probability for that class.

For the first class, the algorithm moves each category to the left branch in order, recording the split criterion at each move. Then the algorithm repeats this process for the remaining classes. Out of this sequence, the selected split is the one that maximizes the split criterion.

Inspect Data with Multilevel Categorical Predictors

- This example shows how to inspect a data set that includes categorical predictors with many levels (categories) and how to train a binary decision tree for classification.
- Load the census1994 file. This data set consists of demographic data from the US
- Census Bureau to predict whether an individual makes over \$50,000 a year. Specify a cell array of character vectors containing the variable names.

```
load census1994
```

```
VarNames = adultdata.Properties.VariableNames;
```

Inspect Data with Multilevel Categorical Predictors

- Some variable names in the `adulthood` table contain the `_` character. Replace instances of `_` with a space.

```
VarNames = strrep(VarNames,'_',' ');
```

- Specify the predictor data `tbl` and the response vector `Y`.

```
tbl = adulthood(:,1:end-1);
```

```
Y = categorical(adulthood.salary);
```

Inspect Data with Multilevel Categorical Predictors

- Inspect Categorical Predictors

- Some categorical variables have many levels (categories). Count the number of levels of each categorical predictor.
- Find the indexes of categorical predictors that are not numeric in the `tbl` table by using `varfun` and `isnumeric`. The `varfun` function applies the `isnumeric` function to each variable of the table `tbl`.

```
cat = ~varfun(@isnumeric,tbl,'OutputFormat','uniform');
```

- Define an anonymous function to count the number of categories in a categorical predictor using `numel` and `categories`.

```
countNumCats = @(var)numel(categories(categorical(var)));
```

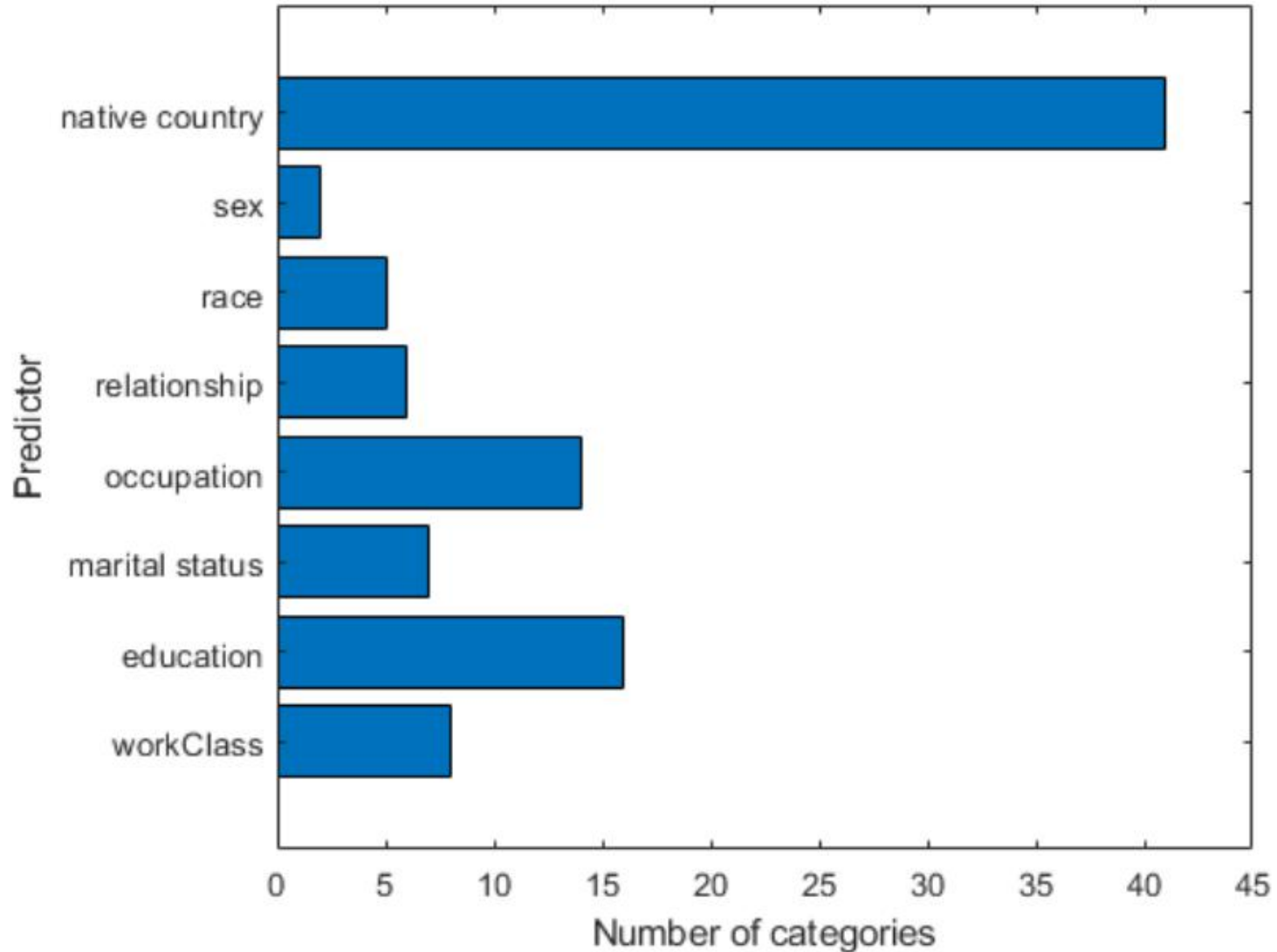
Inspect Data with Multilevel Categorical Predictors

The anonymous function `countNumCats` converts a predictor to a categorical array, then counts the unique, nonempty categories of the predictor.

Use `varfun` and `countNumCats` to count the number of categories for the categorical predictors in `tbl`.

```
numCat = varfun(@(var)countNumCats(var),tbl(:,cat),'OutputFormat','uniform');  
figure  
barh(numCat);  
h = gca;  
h.YTickLabel = VarNames(cat);  
ylabel('Predictor')  
xlabel('Number of categories')
```


Inspect Data with Multilevel Categorical Predictors



Inspect Data with Multilevel Categorical Predictors

- Train Model

For binary classification, the software uses a computational shortcut to find an optimal split for categorical predictors with many categories. For classification with more than two classes, you can choose an exact algorithm or a heuristic algorithm to find a good split by using the 'AlgorithmForCategorical' name-value pair argument of `fitctree` or `templateTree`. By default, the software selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor.

Inspect Data with Multilevel Categorical Predictors

● Train Model

Train a classification tree using `tbl` and `Y`. The response vector `Y` has two classes, so the software uses the exact algorithm for categorical predictor splits.

```
Mdl = fitctree(tbl,Y)
```

```
Mdl =
```

```
ClassificationTree
```

```
PredictorNames: {1x14 cell}
```

```
ResponseName: 'Y'
```

```
CategoricalPredictors: [2 4 6 7 8 9 10 14]
```

```
ClassNames: [<=50K >50K]
```

```
ScoreTransform: 'none'
```

```
NumObservations: 32561
```

```
Properties, Methods
```

Classification Learner

- Train Model

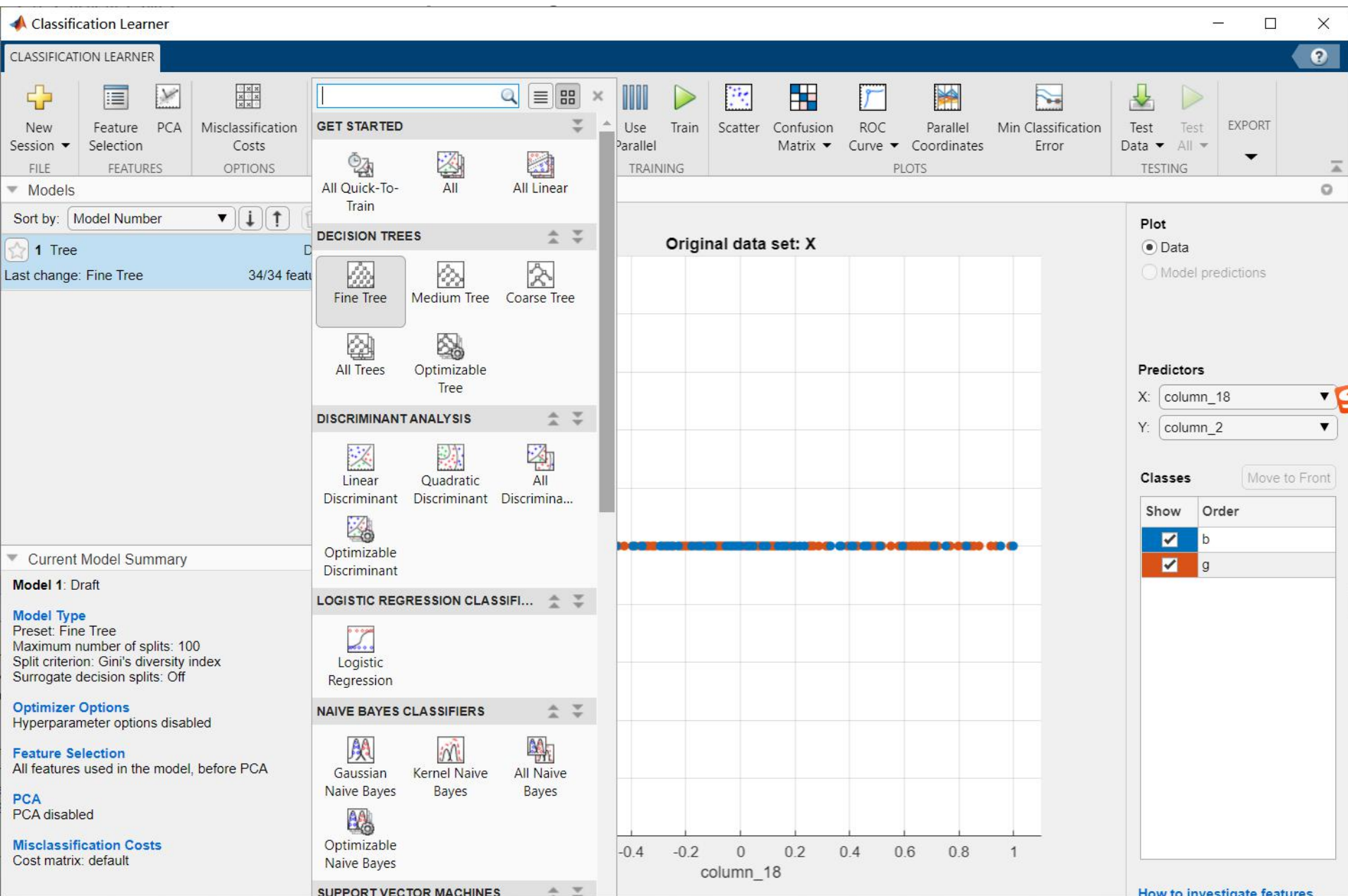
You can use Classification Learner to train models of these classifiers: decision trees, discriminant analysis, support vector machines, logistic regression, nearest neighbors, naive Bayes, and ensemble classification. In addition to training models, you can explore your data, select features, specify validation schemes, and evaluate results. You can export a model to the workspace to use the model with new data or generate MATLAB code to learn about programmatic classification.

Classification Learner

- Train Model

In Classification Learner, the Model Type gallery shows as available the classifier types that support your selected data.

Classifier	All predictors numeric	All predictors categorical	Some categorical, some numeric
Decision Trees	Yes	Yes	Yes
Discriminant Analysis	Yes	No	No
Logistic Regression	Yes	Yes	Yes
SVM	Yes	Yes	Yes
Nearest Neighbor	Euclidean distance only	Hamming distance only	No
Ensembles	Yes	Yes, except Subspace Discriminant	Yes, except any Subspace
Naive Bayes	Yes	Yes	Yes






Classification Learner

Decision trees are easy to interpret, fast for fitting and prediction, and low on memory usage, but they can have low predictive accuracy. Try to grow simpler trees to prevent overfitting. Control the depth with the Maximum number of splits setting.

In the Model Type gallery click All Trees to try each of the nonoptimizable decision tree options. Train them all to see which settings produce the best model with your data. Select the best model in the History list. To try to improve your model, try feature selection, and then try changing some advanced options.

Classification Learner

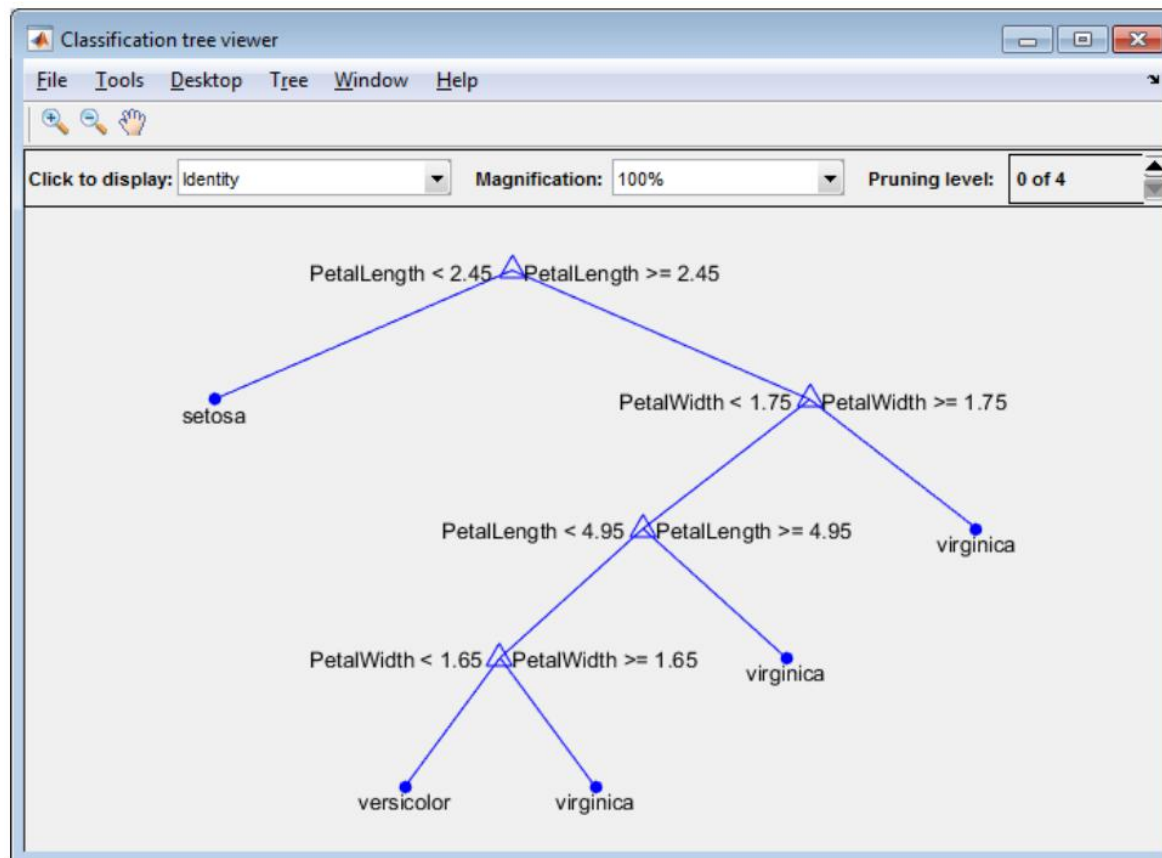
Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Coarse Tree 	Fast	Small	Easy	Low Few leaves to make coarse distinctions between classes (maximum number of splits is 4).
Medium Tree 	Fast	Small	Easy	Medium Medium number of leaves for finer distinctions between classes (maximum number of splits is 20).
Fine Tree 	Fast	Small	Easy	High Many leaves to make many fine distinctions between classes (maximum number of splits is 100).

Classification Learner

You can visualize your decision tree model by exporting the model from the app, and then entering:

```
view(trainedModel.ClassificationTree,'Mode','graph')
```

The figure shows an example fine tree trained with the fisheriris data.

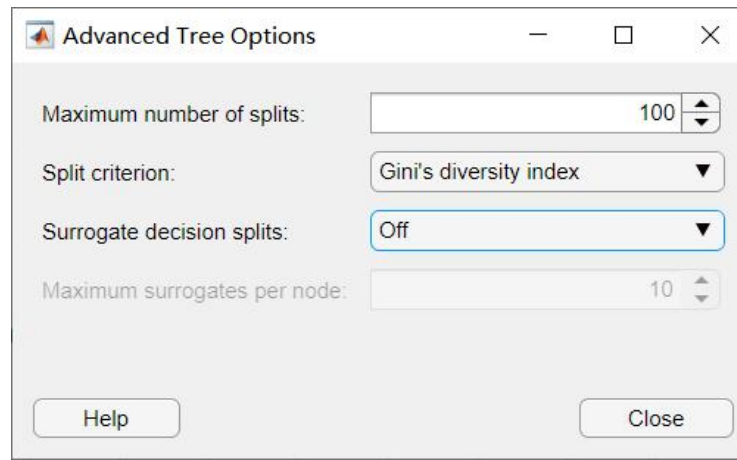


Classification Learner

Classification trees in Classification Learner use the `fitctree` function. You can set these options:

- Maximum number of splits

Specify the maximum number of splits or branch points to control the depth of your tree. When you grow a decision tree, consider its simplicity and predictive power. To change the number of splits, click the buttons or enter a positive integer value in the Maximum number of splits box.



Classification Learner

- A fine tree with many leaves is usually highly accurate on the training data. However, the tree might not show comparable accuracy on an independent test set. A leafy tree tends to overtrain, and its validation accuracy is often far lower than its training (or resubstitution) accuracy.
- In contrast, a coarse tree does not attain high training accuracy. But a coarse tree can be more robust in that its training accuracy can approach that of a representative test set. Also, a coarse tree is easy to interpret.

Classification Learner

- Split criterion

Specify the split criterion measure for deciding when to split nodes. Try each of the three settings to see if they improve the model with your data.

Split criterion options are Gini's diversity index, Twoing rule, or Maximum deviance reduction (also known as cross entropy).

The classification tree tries to optimize to pure nodes containing only one class. Gini's diversity index (the default) and the deviance criterion measure node impurity. The twoing rule is a different measure for deciding how to split a node, where maximizing the twoing rule expression increases node purity

Classification Learner

- Surrogate decision splits — Only for missing data.

Specify surrogate use for decision splits. If you have data with missing values, use surrogate splits to improve the accuracy of predictions.

When you set Surrogate decision splits to On, the classification tree finds at most 10 surrogate splits at each branch node. To change the number, click the buttons or enter a positive integer value in the Maximum surrogates per node box.

When you set Surrogate decision splits to Find All, the classification tree finds all surrogate splits at each branch node. The Find All setting can use considerable time and memory.