1 How to visualize decision trees

Brought to you by explained.ai

Terence Parr and Prince Grover

(Terence is a tech lead at Google and ex-Professor of computer/data science in University of San Francisco's MS in Data Science program and Prince is an alumnus. You might know Terence as the creator of the ANTLR parser generator.)

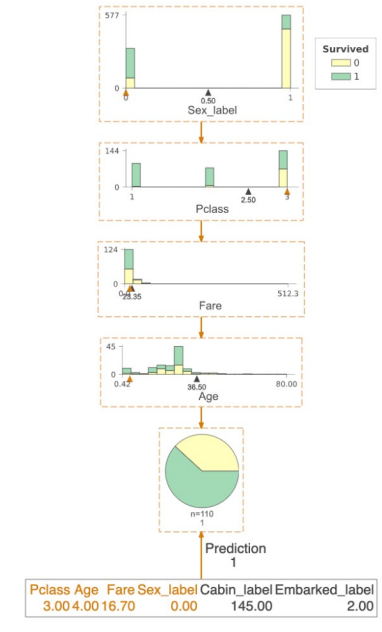
Please send comments, suggestions, or fixes to Terence.

Contents

Update July 2020 Tudor Lapusan has become a major contributor to `dtreeviz` and, thanks to his work, `dtreeviz` can now visualize XGBoost and Spark decision trees as well as sklearn. Beyond what is described in this article, the library now also includes the following features. See dtreeviz_sklearn_visualisations.ipynb for examples.

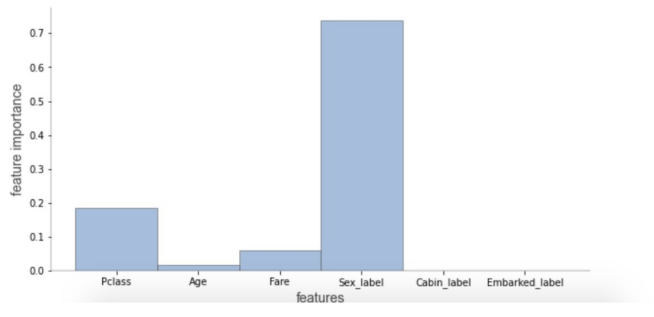
A visualization of just the path from the root to a decision tree leaf.

```
trees.dtreeviz(sk_dtree, show_just_path=True, X = dataset[features].iloc[10])
```

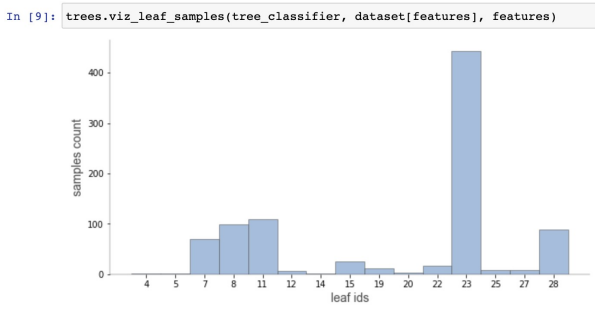


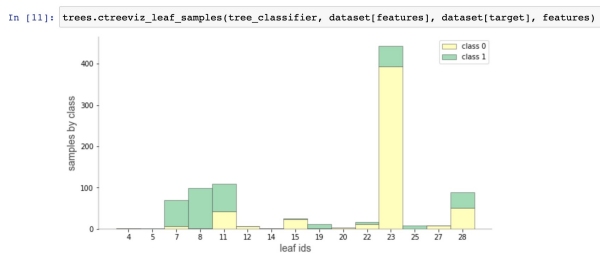
An explanation in English how a decision tree makes a prediction for a specific record.

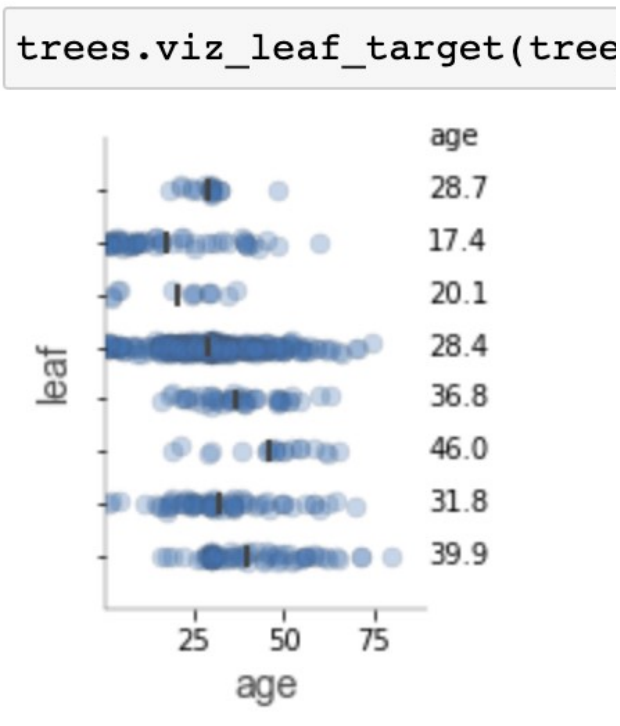
```
In [17]: print(trees.explain_prediction_path(tree_classifier, X, feature_names=features, explanation_type="plain_english"))
2.5 <= Pclass
Age < 36.5
Fare < 23.35
Sex_label < 0.5

In [18]: trees.explain_prediction_path(tree_classifier, X, feature_names=features, explanation_type="sklearn_default")
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x13578bfd0>
```



Visualizations for purity and distributions for individual leaves.



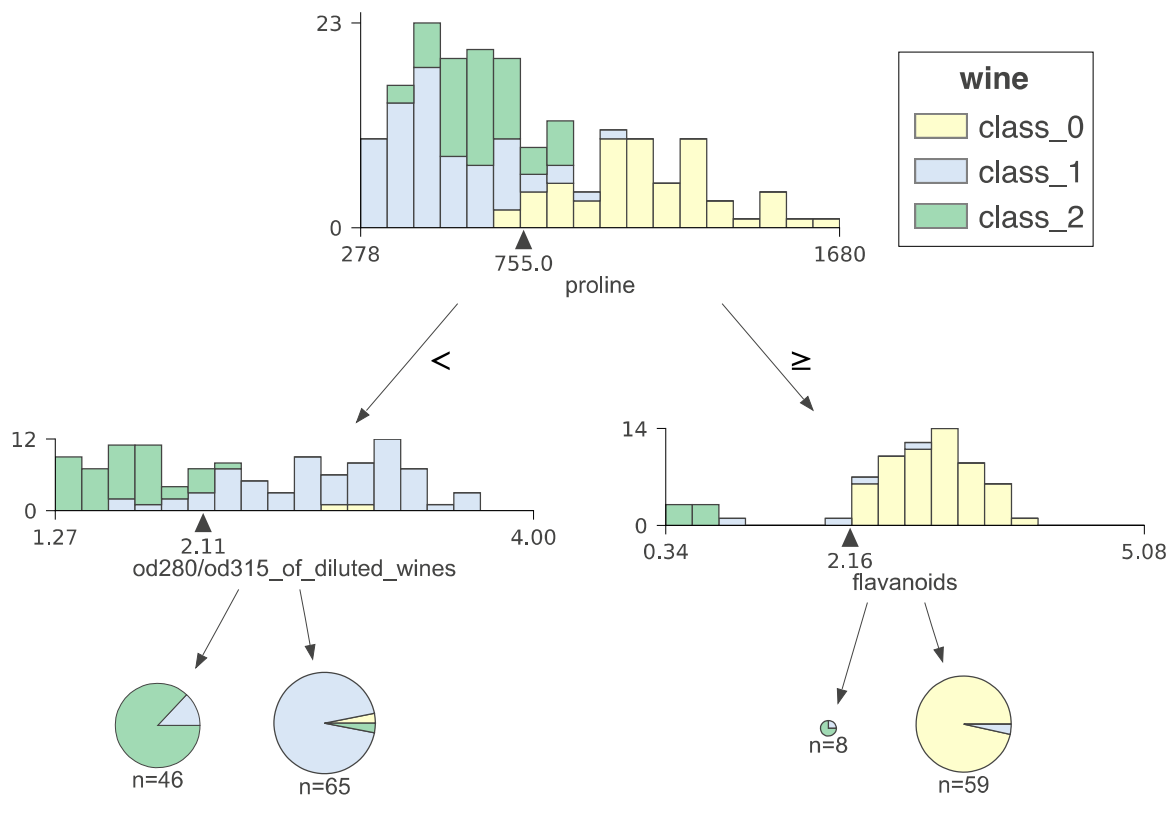




1.1 Introduction

Decision trees are the fundamental building block of gradient boosting machines and Random Forests™, probably the two most popular machine learning models for structured data. Visualizing decision trees is a tremendous aid when learning how these models work and when interpreting models. Unfortunately, current visualization packages are rudimentary and not immediately helpful to the novice. For example, we couldn't find a library that visualizes how decision nodes split up the feature space. It is also uncommon for libraries to support visualizing a specific feature vector as it weaves down through a tree's decision nodes; we could only find one image showing this.

So, we've created a general package for scikit-learn decision tree visualization and model interpretation, which we'll be using heavily in an upcoming machine learning book (written with Jeremy Howard). Here's a sample visualization for a tiny decision tree (click to enlarge):



This article demonstrates the results of this work, details the specific choices we made for visualization, and outlines the tools and techniques used in the implementation. The visualization software is part of a nascent Python machine learning library called dtreeviz. We assume you're familiar with the basic mechanism of decision trees if you're interested in visualizing them, but let's start with a brief summary so that we're all using the same terminology. (If you're not familiar with

decision trees, check out fast.ai's Introduction to Machine Learning for Coders MOOC.)

1.2 Decision tree review

A decision tree is a machine learning model based upon binary trees (trees with at most a left and right child). A decision tree learns the relationship between observations in a training set, represented as feature vectors $\mathbf{x}$ and target values y, by examining and condensing training data into a binary tree of interior nodes and leaf nodes. (Notation: vectors are in bold and scalars are in italics.)

Each leaf in the decision tree is responsible for making a specific prediction. For regression trees, the prediction is a value, such as price. For classifier trees, the prediction is a target category (represented as an integer in scikit), such as cancer or not-cancer. A decision tree carves up the feature space into groups of observations that share similar target values and each leaf represents one of these groups. For regression, similarity in a leaf means a low variance among target values and, for classification, it means that most or all targets are of a single class.

Any path from the root of the decision tree to a specific leaf predictor passes through a series of (internal) decision nodes. Each decision node compares a single feature's value in $\mathbf{x}$, x_i, with a specific *split point* value learned during training. For example, in a model predicting apartment rent prices, decision nodes would test features such as the number of bedrooms and number of bathrooms. (See **Section 1.6.3** *Visualizing tree interpretation of a single observation*.) Even in a classifier with discrete target values, decision nodes still compare numeric *feature* values because scitkit's decision tree implementation assumes that all features are numeric. Categorical variables must be one hot encoded, binned, label encoded, etc...

To train a decision node, the model examines a subset of the training observations (or the full training set at the root). The node's feature and split point within that feature space are chosen during training to split the observations into left and right buckets (subsets) to maximize similarity as defined above. (This selection process is generally done through exhaustive comparison of features and feature values.) The left bucket has observations whose x_i feature values are all less than the split point and the right bucket has observations whose x_i is greater than the split point. Tree construction proceeds recursively by creating decision nodes for the left bucket and the right bucket. Construction stops when some stopping criterion is reached, such as having less than five observations in the node.

1.3 The key elements of decision tree visualization

Decision tree visualizations should highlight the following important elements, which we demonstrate below.

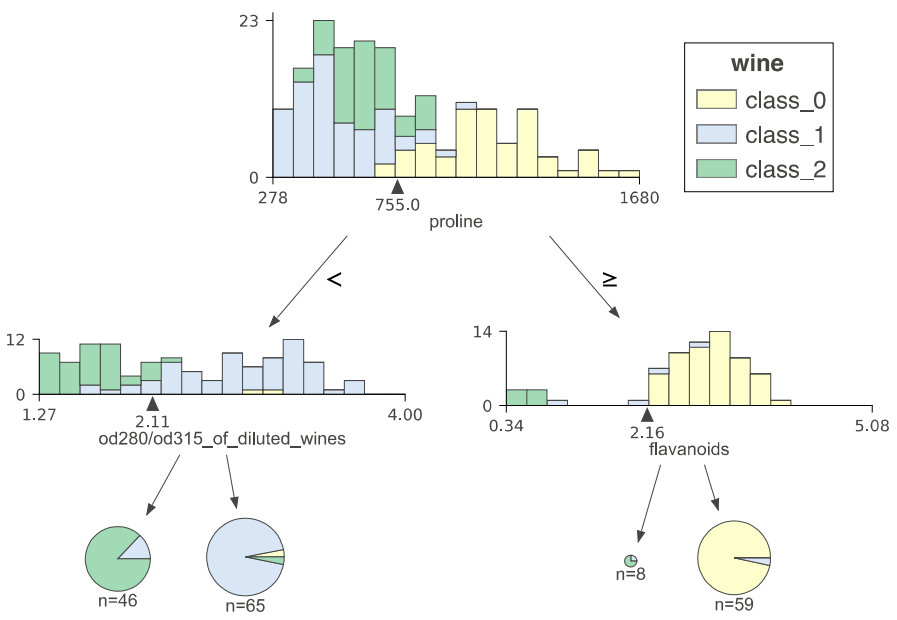
- Decision node **feature versus target value** distributions (which we call feature-target space in this article). We want to know how separable the target values are based upon the feature and a split point.
- Decision node **feature name and feature split value**. We need to know which feature each decision node is testing and where in that space the nodes splits the observations.
- **Leaf node purity**, which affects our prediction confidence. Leaves with low variance among the target values (regression) or an overwhelming majority target class (classification) are much more reliable predictors.
- **Leaf node prediction value**. What is this leaf actually predicting from the collection of target values?
- **Numbers of samples in decision nodes**. Sometimes it's useful to know where all most of the samples are being routed through the decision nodes.
- **Numbers of samples in leaf nodes**. Our goal is a decision tree with fewer, larger and purer leaves. Nodes with too few samples are possible indications of overfitting.
- How a specific feature vector is **run down the tree** to a leaf. This helps explain why a particular

feature vector gets the prediction it does. For example, in a regression tree predicting apartmen rent prices, we might find a feature vector routed into a high predicted price leaf because of a decision node that checks for more than three bedrooms.

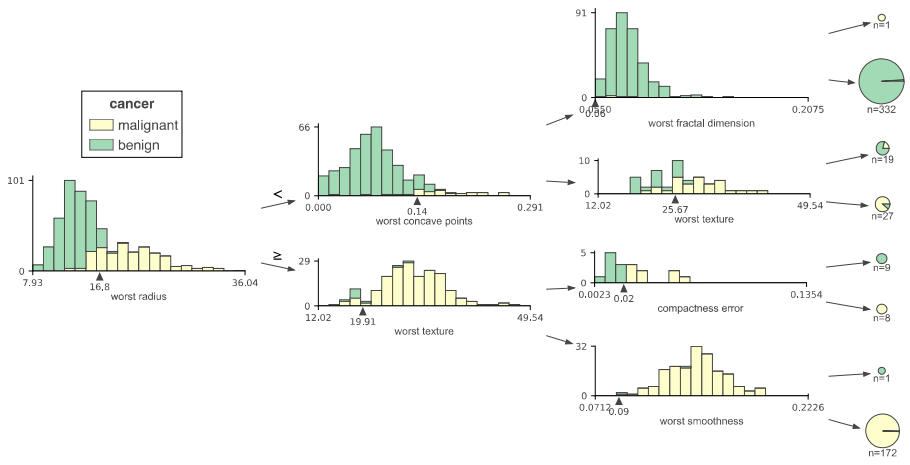
1.4 Gallery of decision tree visualizations

Before digging into the previous state-of-the-art visualizations, we'd like to give a little spoiler to show what's possible. This section highlights some samples visualizations we built from scikit regression and classification decision trees on a few data sets. You can also check out the full gallery and code to generate all samples.

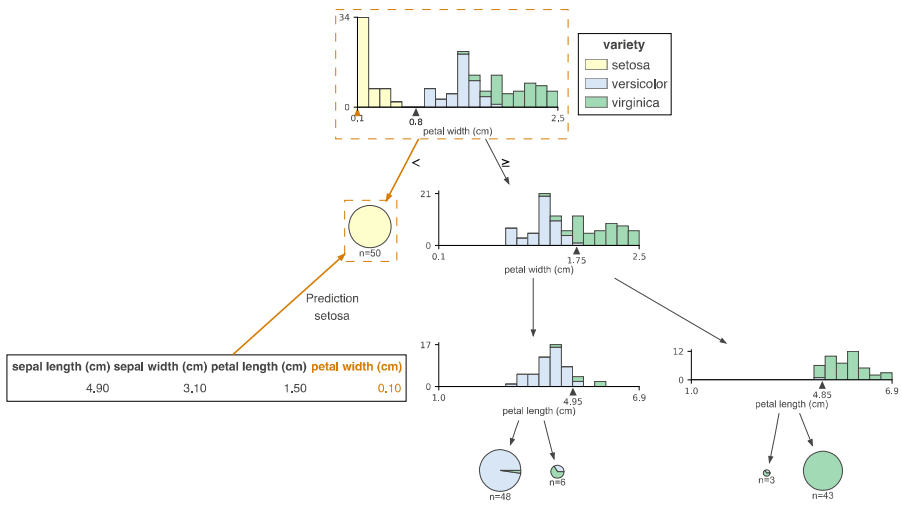
Wine 3-class top-down orientation



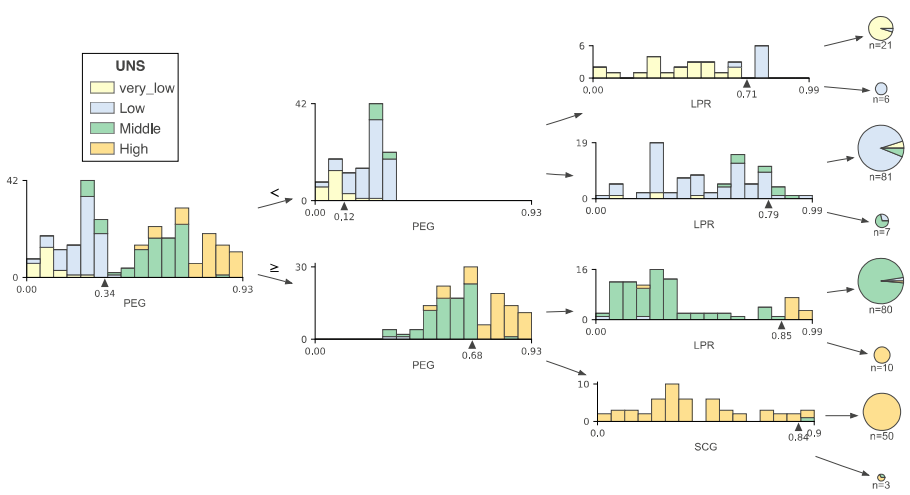
Breast cancer 2-class left-to-right



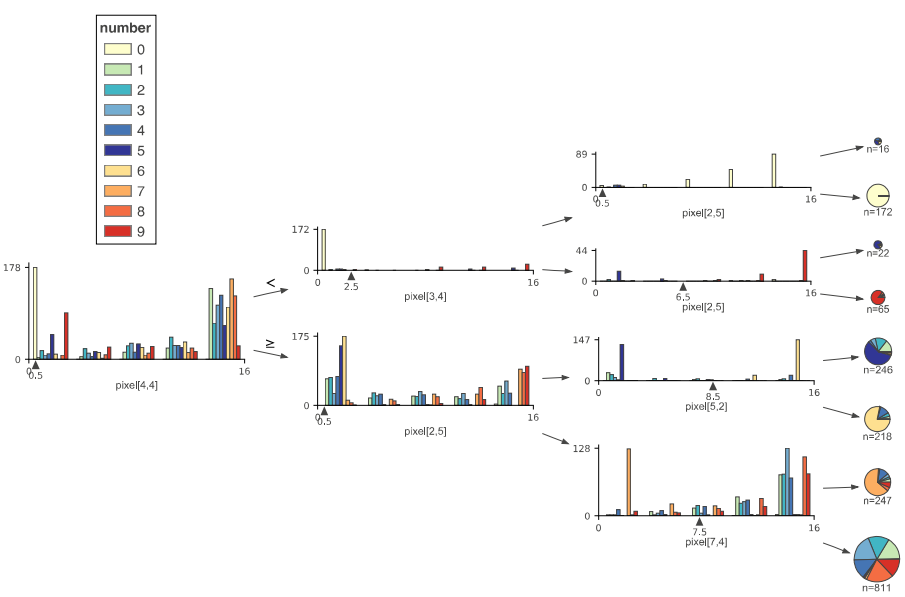
Iris 3-class showing a prediction



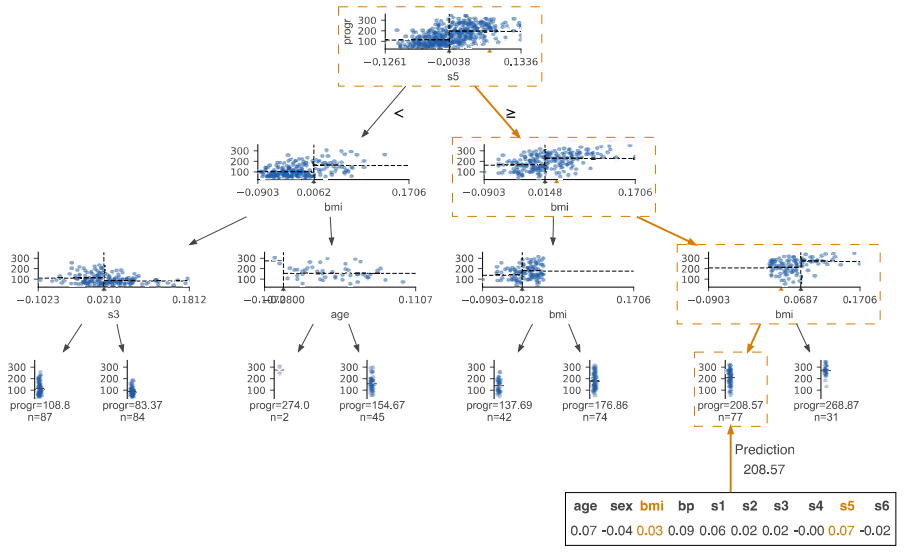
User knowledge rating 4-class



Digits 10-class

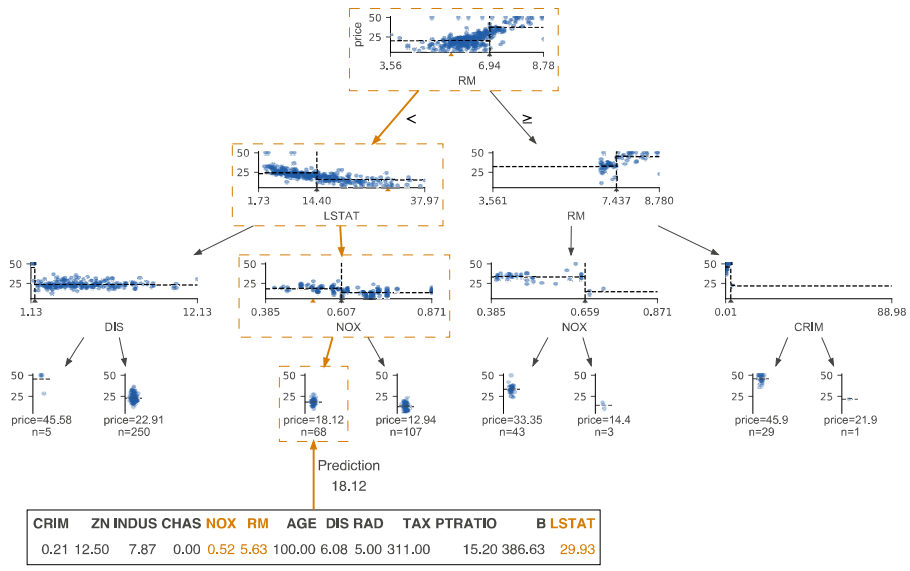


Diabetes showing a prediction

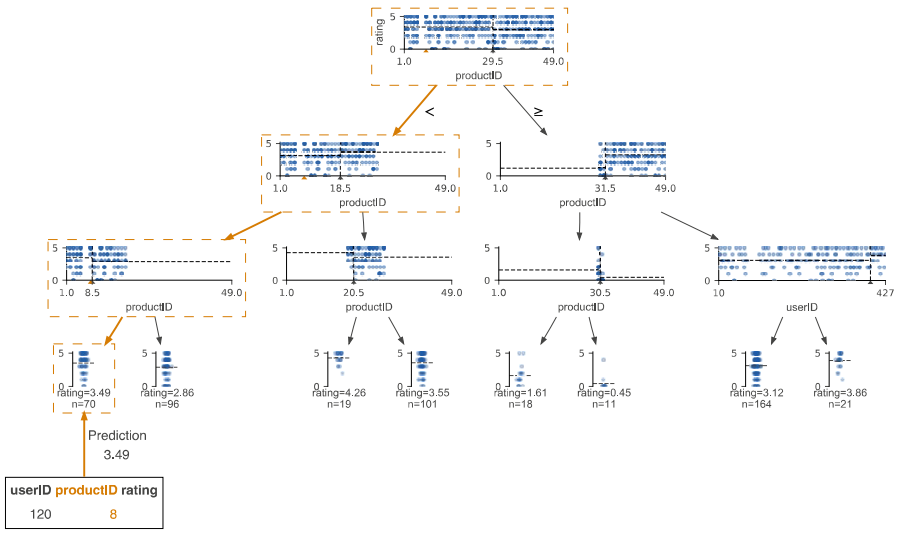


Boston showing a prediction

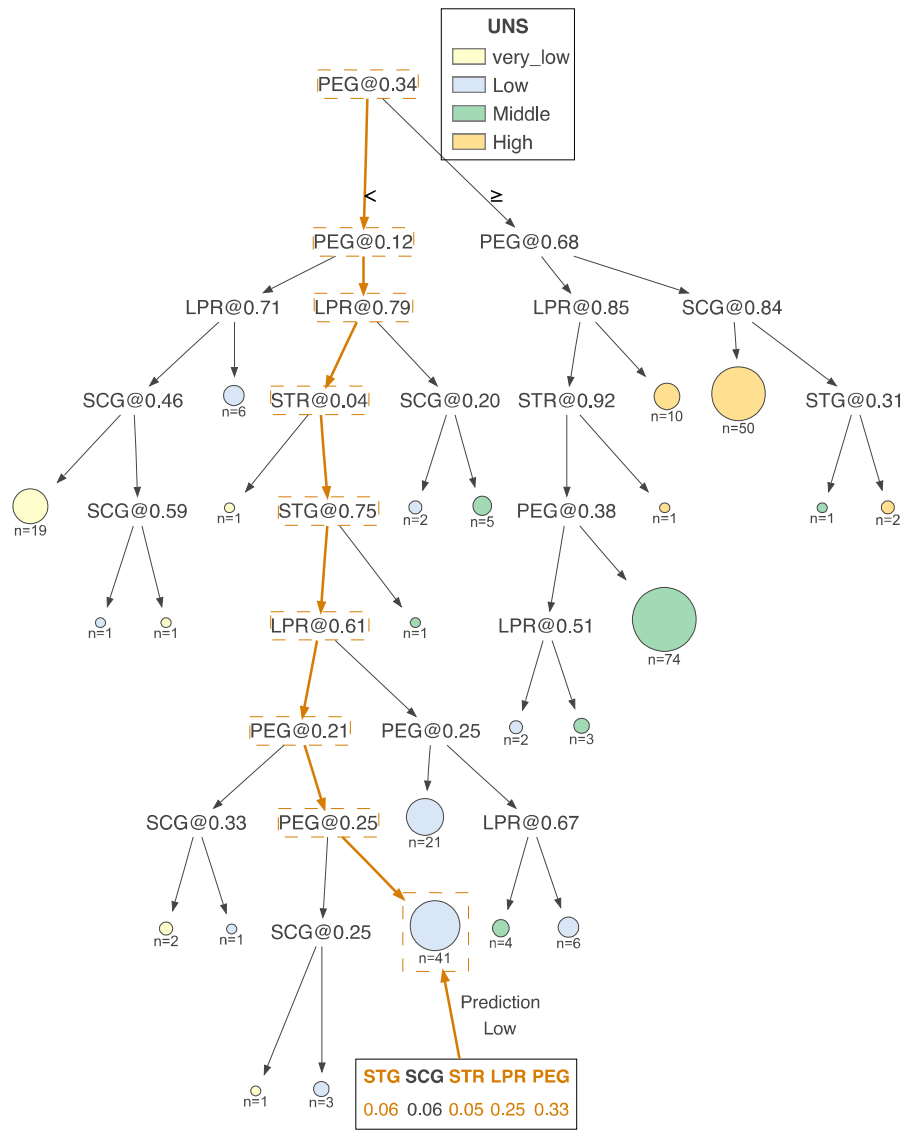
Sweets showing a prediction

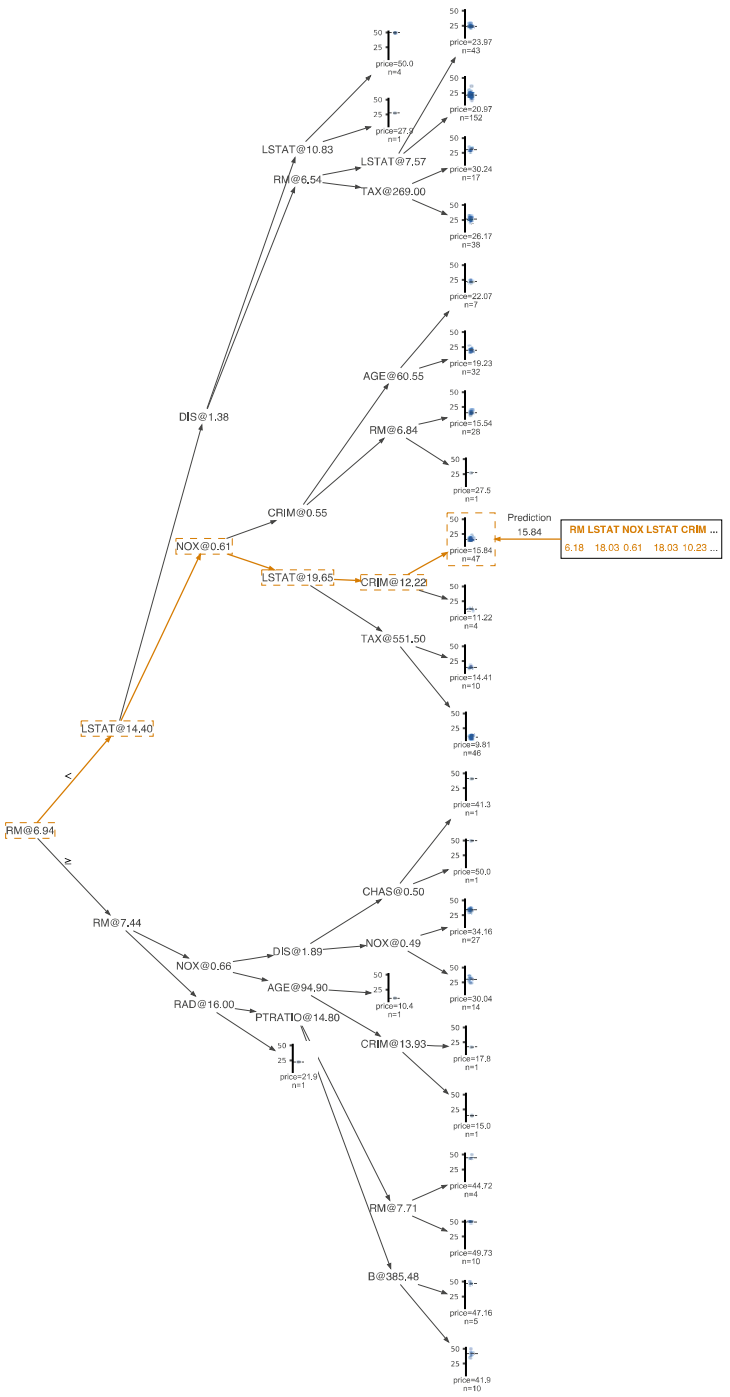


User knowledge rating 4-class non-fancy



Diabetes non-fancy





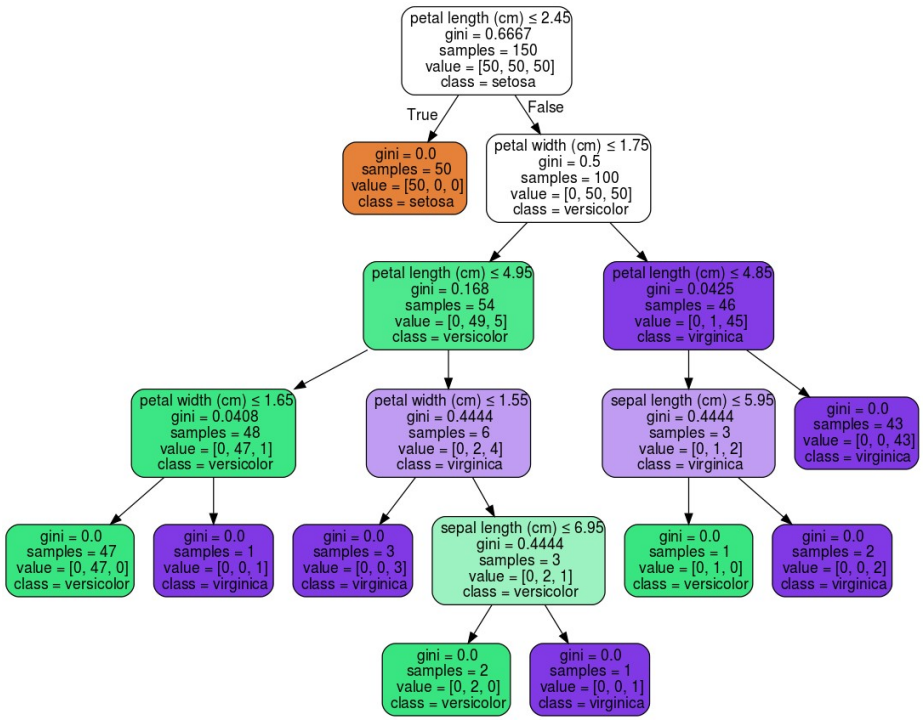
1.5 A comparison to previous state-of-the-art visualizations

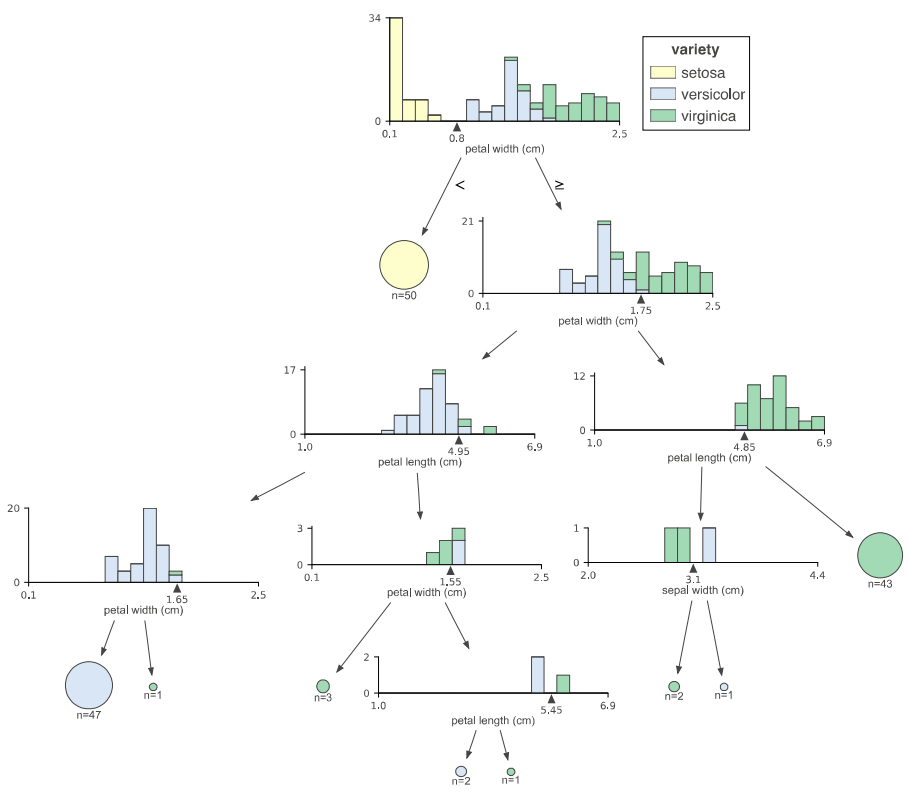
If you search for “visualizing decision trees” you will quickly find a **Python** solution provided by the awesome scikit folks: sklearn.tree.export_graphviz. With more work, you can find visualizations for R and even SAS and IBM. In this section, we collect the various decision tree visualizations we could find and compare them to the visualizations made by our `dtreeviz` library. We give a more detailed discussion of our visualizations in the next section.

Let's start with the default scitkit visualization of a decision tree on the well-known Iris data set (click on images to enlarge).

Default scikit Iris visualization

Our `dtreeviz` Iris visualization

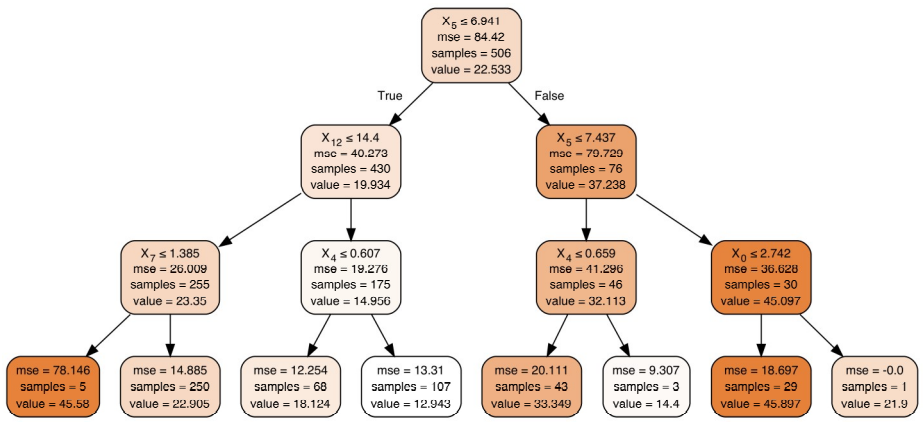




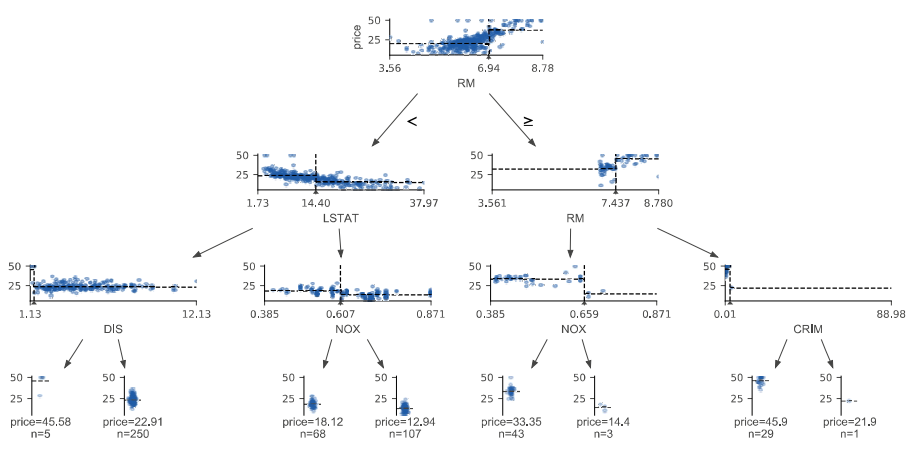
The scikit tree does a good job of representing the tree structure, but we have a few quibbles. The colors aren't the best and it's not immediately obvious why some of the nodes are colored and some aren't. If the colors represent predicted class for this classifier, one would think just the leaves would be colored because only leaves have predictions. (It turns out the non-colored nodes have no majority prediction.) Including the gini coefficient (certainty score) costs space and doesn't help with interpretation. The count of samples of the various target classes in each node is somewhat useful, but a histogram would be even better. A target class color legend would be nice. Finally, using true and false as the edge labels isn't as clear as, say, labels $<$ and $\geq$. The most obvious difference is that our decision nodes show feature distributions as overlapping stacked-histograms, one histogram per target class. Also, our leaf size is proportional to the number of samples in that leaf.

Scikit uses the same visualization approach for decision tree regressors. For example, here is scikit's visualization using the Boston data set, with `dtreeviz`'s version for comparison (click to enlarge images):

Default scikit Boston visualization

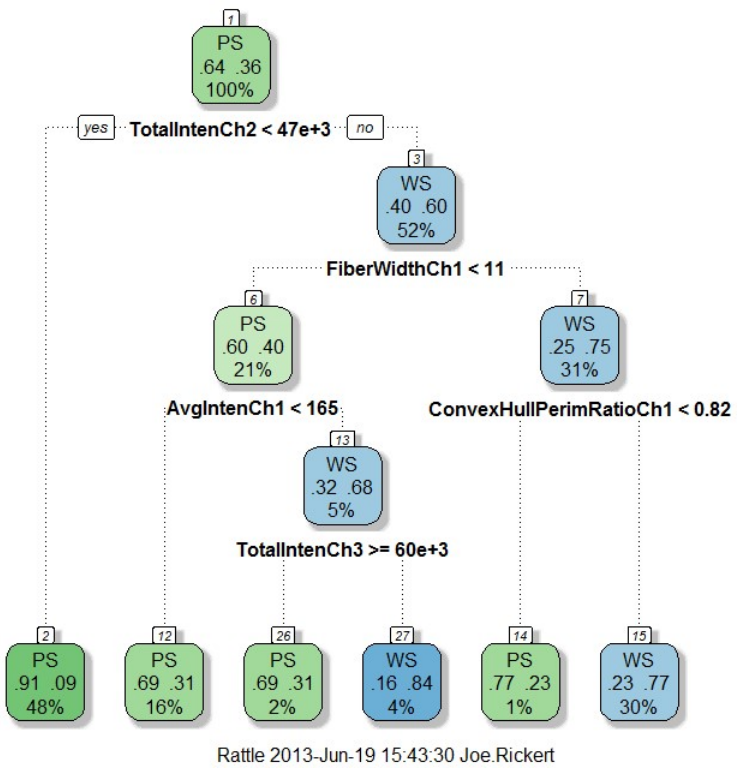


Our `dtreeviz` Boston visualization



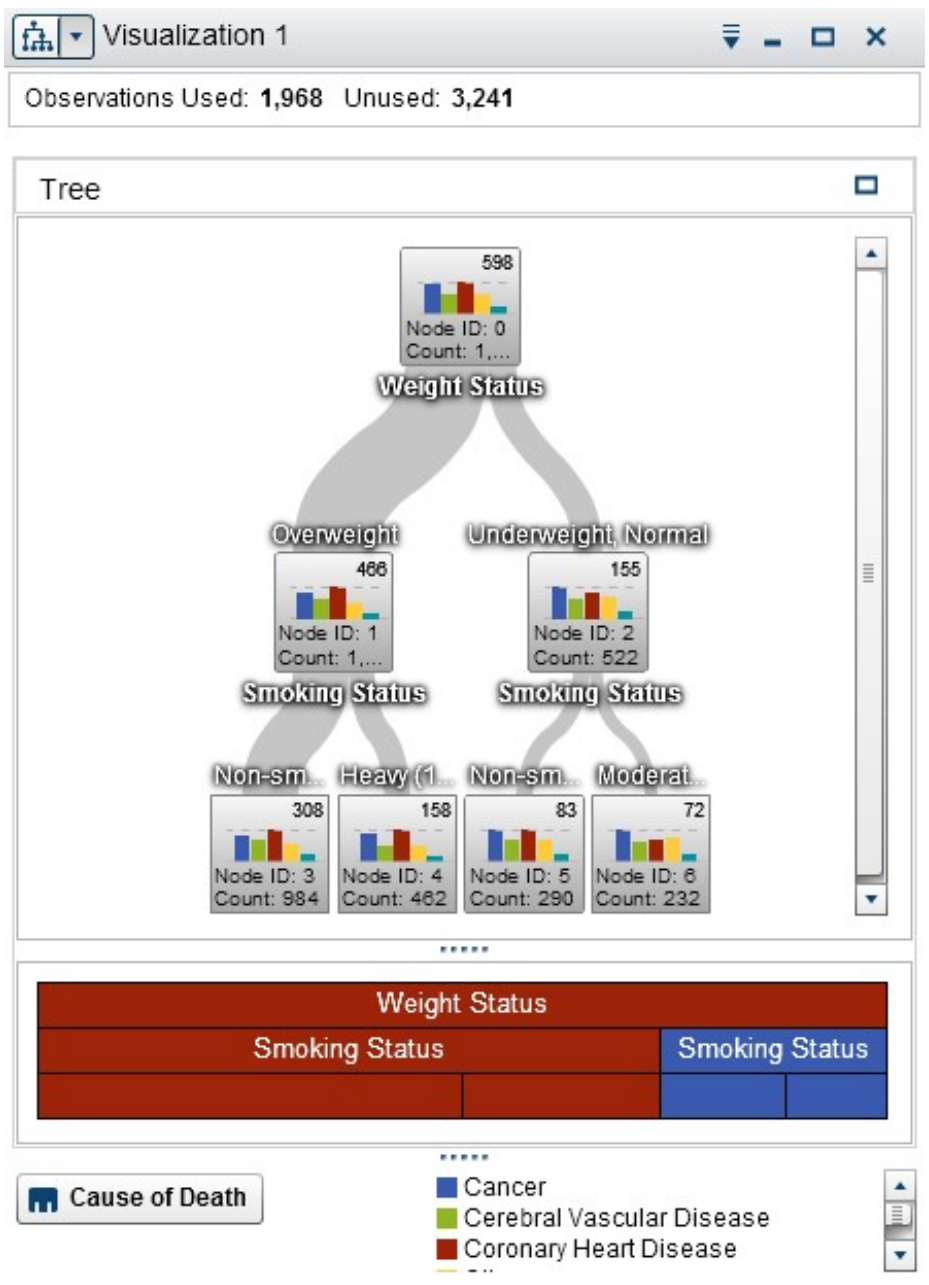
In the scikit tree, it's not immediately clear what the use of color implies, but after studying the image, darker images indicate higher predicted target values. As before, our decision nodes show the feature space distribution, this time using a feature versus target value scatterplot. The leaves use strip plots to show the target value distribution; leaves with more dots naturally have a higher number of samples.

R programmers also have access to a package for visualizing decision trees, which gives similar results to scikit but with nicer edge labels:

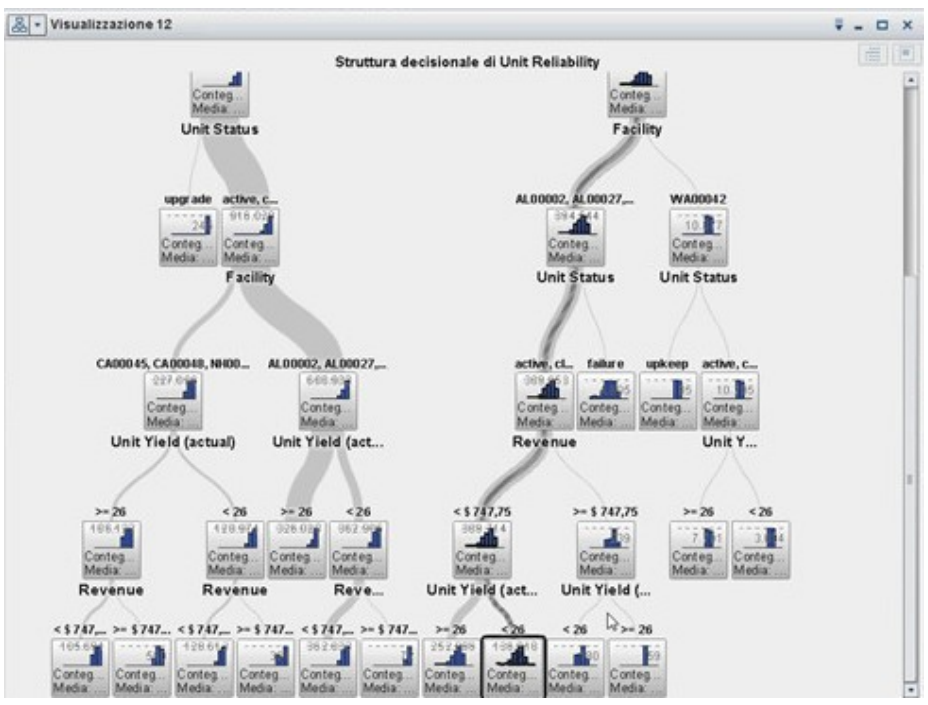


SAS and **IBM** also provide (non-Python-based) decision tree visualizations. Starting with SAS, we see that their decision nodes include a bar chart related to the node's sample target values and other details:

SAS visualization



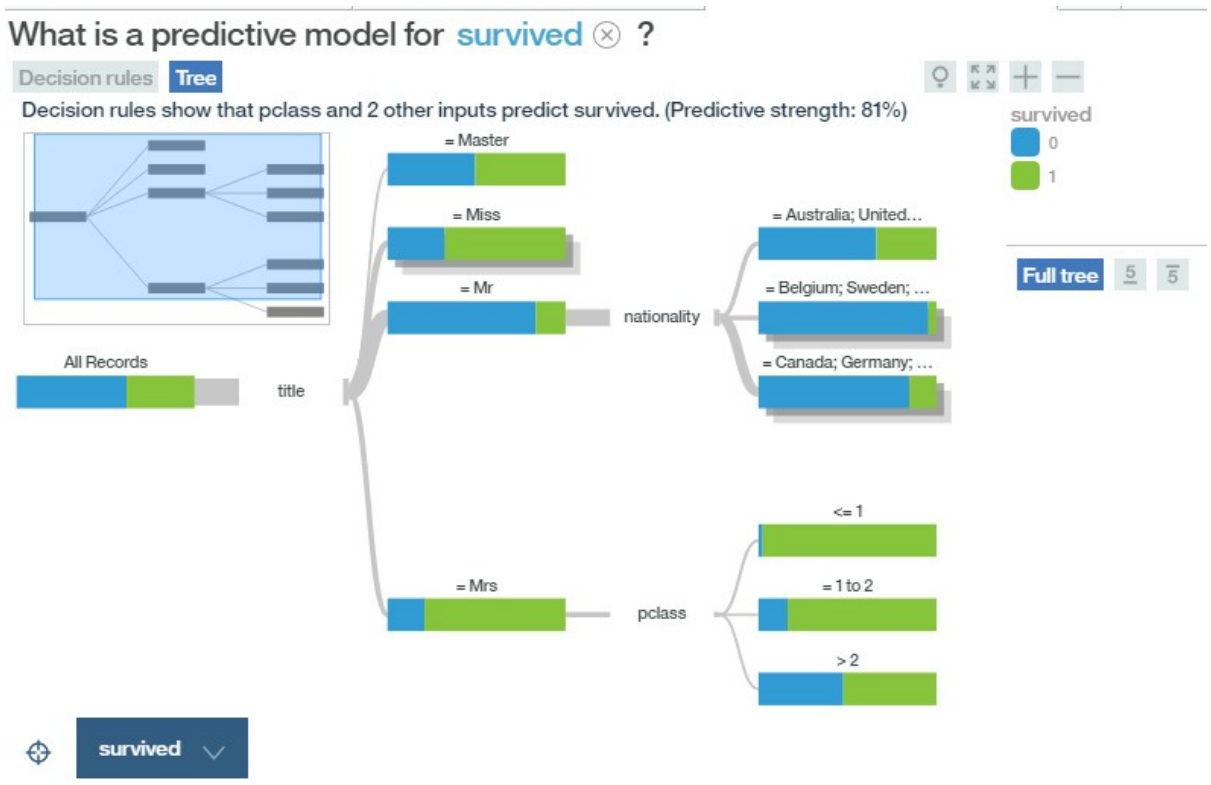
SAS visualization (best image quality we could find with numeric features)



Indicating the size of the left and right buckets via edge width is a nice touch. But, those bar charts are hard to interpret because they have no horizontal axis. Decision nodes testing categorical variables (left image) have exactly one bar per category, so they must represent simple category counts, rather than feature distributions. For numeric features (right image), SAS decision nodes show a histogram of either target or feature space (we can't tell from the image). SAS node bar charts / histograms appear to illustrate just target values, which tells us nothing about how the feature space was split.

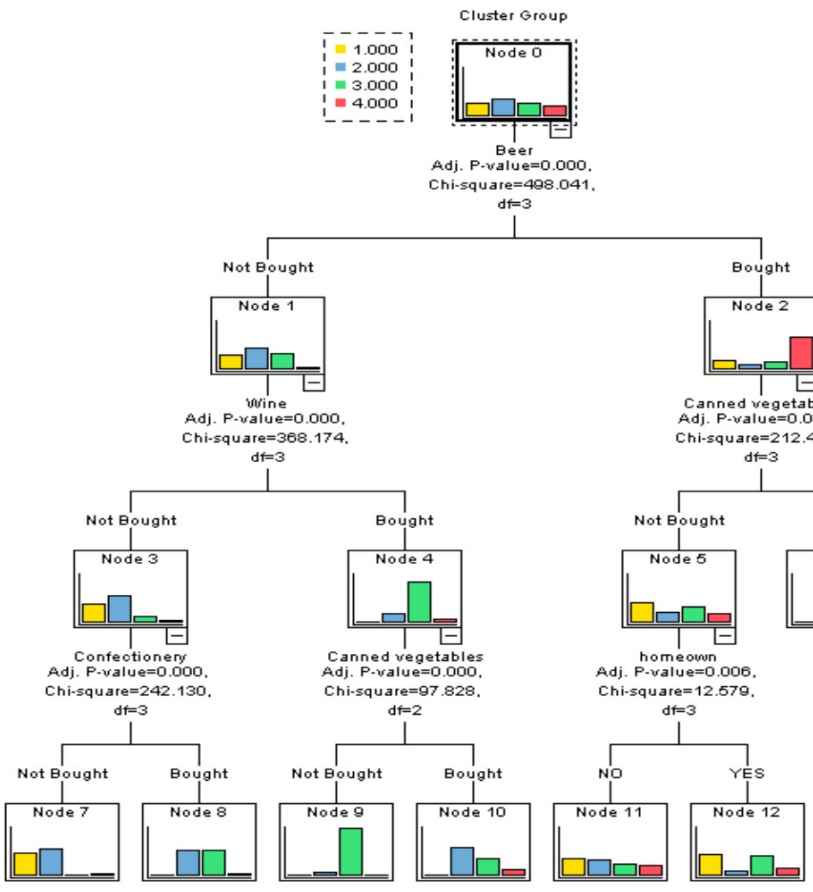
The SAS tree on the right appears to highlight a path through the decision tree for a specific unknown feature vector, but we couldn't find any other examples from other tools and libraries. The ability to visualize a specific vector run down the tree does not seem to be generally available.

Moving on to IBM software, here is a nice visualization that also shows decision node category counts as bar charts, from IBM's Watson analytics (on the TITANIC data set):

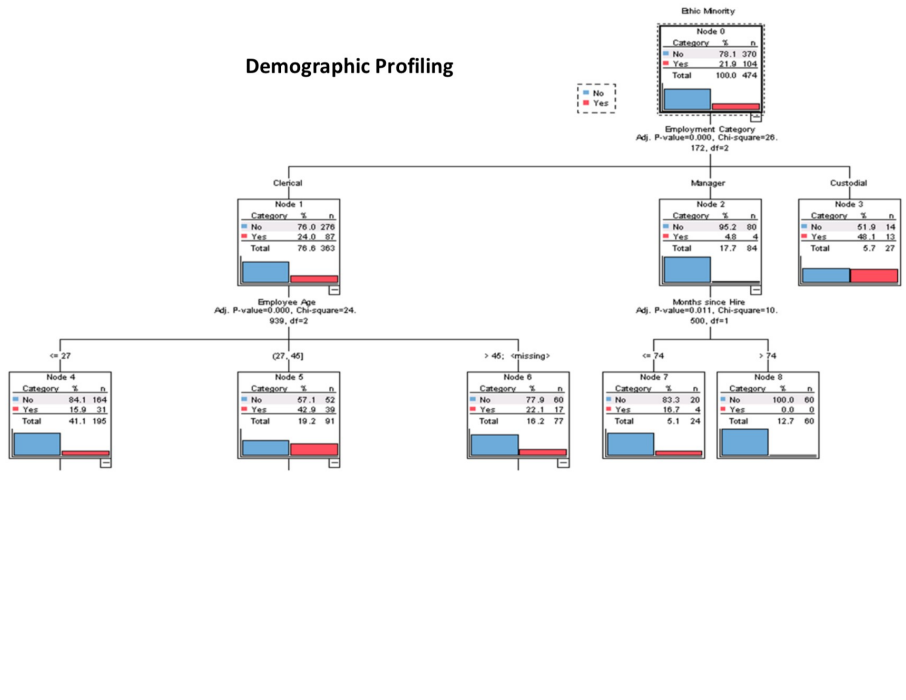


IBM's earlier **SPSS** product also had decision tree visualizations:

SPSS visualization

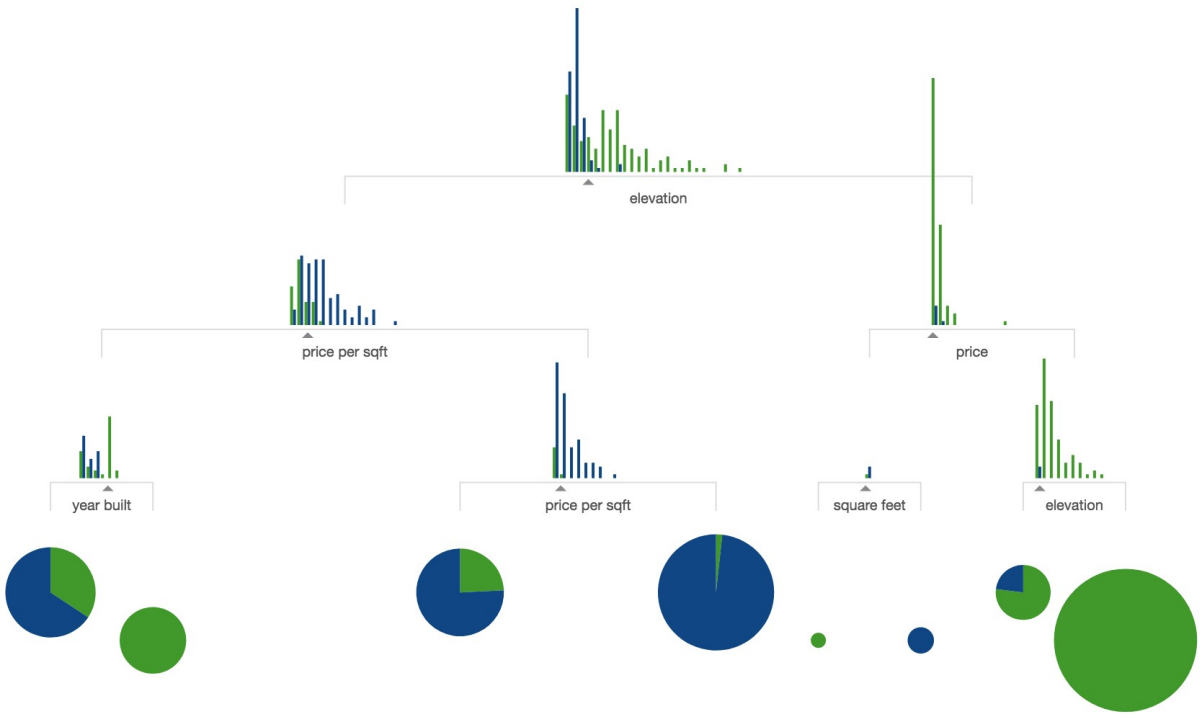


SPSS visualization



These SPSS decision nodes seem to give the same SAS-like bar chart of sample target class counts.

All of the visualizations we encountered from the major players were useful, but we were most inspired by the eye-popping visualizations in A visual introduction to machine learning, which shows an (animated) decision tree like this:



This visualization has three unique characteristics over previous work, aside from the animation:

- the decision nodes show how the feature space is split
- the split points for decision nodes are shown visually (as a wedge) in the distribution
- the leaf size is proportional to the number of samples in that leaf

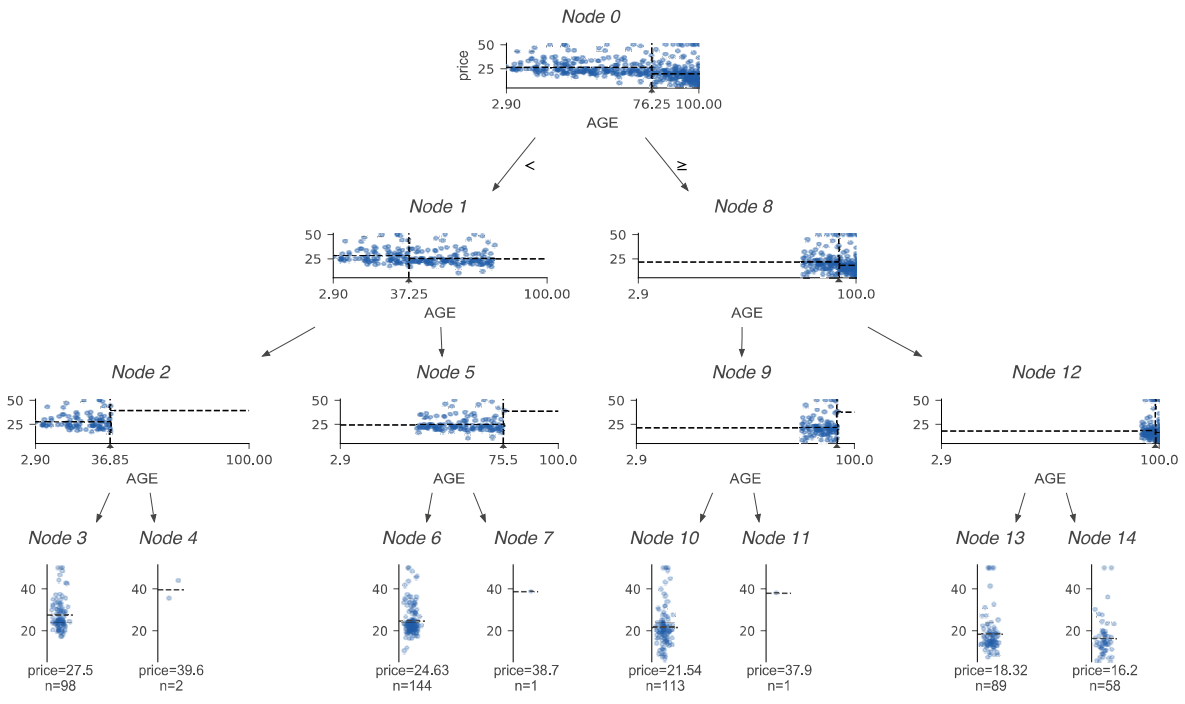
While that visualization is a hardcoded animation for educational purposes, it points in the right direction.

1.6 Our decision tree visualizations

Other than the educational animation in A visual introduction to machine learning, we couldn't find a decision tree visualization package that illustrates how the feature space is split up at decision nodes (feature-target space). This is the critical operation performed during decision tree model training and is what newcomers should focus on, so we'll start by examining decision node visualizations for both classification and regression trees.

1.6.1 Visualizing feature-target space

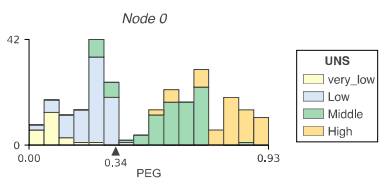
Training of a decision node chooses feature x_i and split value within x_i's range of values (feature space) to group samples with similar target values into two buckets. Just to be clear, training involves examining the relationship between features and target values. Unless decision nodes show feature-target space in some way, the viewer cannot see how and why training arrived at the split value. To highlight how decision nodes carve up the feature space, we trained a regressor and classifier with a single (AGE) feature (code to generate images). Here's a regressor decision tree trained using a single feature from the Boston data, AGE, and with node ID labeling turned on for discussion purposes here:

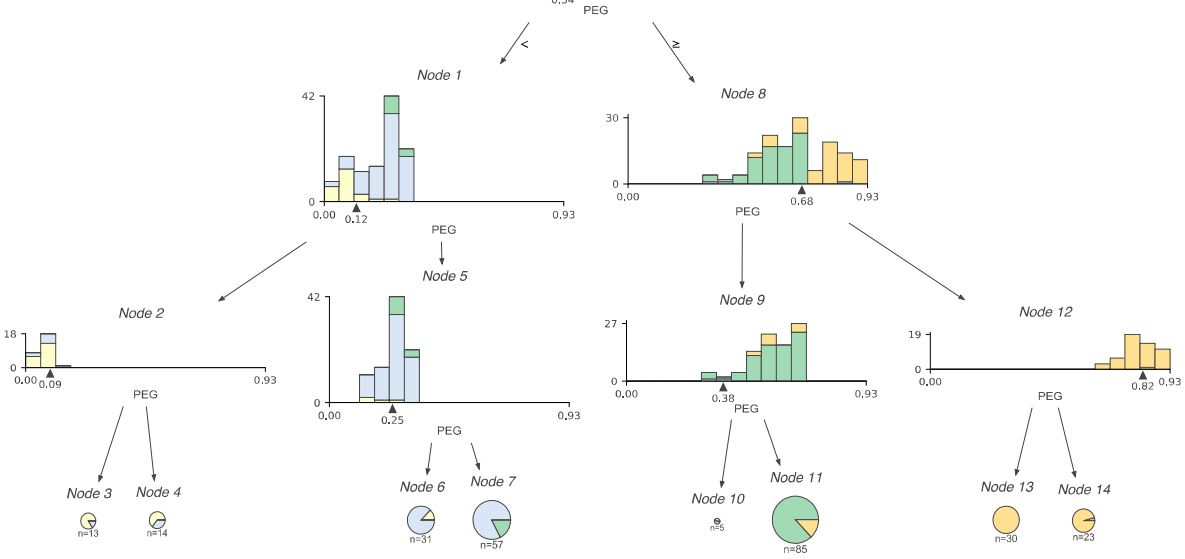


Horizontal dashed lines indicate the target mean for the left and right buckets in decision nodes; a vertical dashed line indicates the split point in feature space. The black wedge highlights the split point and identifies the exact split value. Leaf nodes indicate the target prediction (mean) with a dashed line.

As you can see, each AGE feature axis uses the same range, rather than zooming in, to make it easier to compare decision nodes. As we descend through decision nodes, the sample AGE values are boxed into narrower and narrower regions. For example, the AGE feature space in node 0 is split into the regions of AGE future space shown in nodes 1 and 8. Node 1's space is then split into the chunks shown in nodes 2 and 5. The prediction leaves are not very pure because training a model on just a single variable leads to a poor model, but this restricted example demonstrates how decision trees carve up feature space.

While a decision tree implementation is virtually identical for both classifier and regressor decision trees, the way we interpret them is very different, so our visualizations are distinct for the two cases. For a regressor, showing feature-target space is best done with a scatterplot of feature versus target. For classifiers, however, the target is a category, rather than a number, so we chose to illustrate feature-target space using histograms as an indicator of feature space distributions. Here's a classifier tree trained on the USER KNOWLEDGE data, again with a single feature (PEG) and with nodes labeled for discussion purposes:



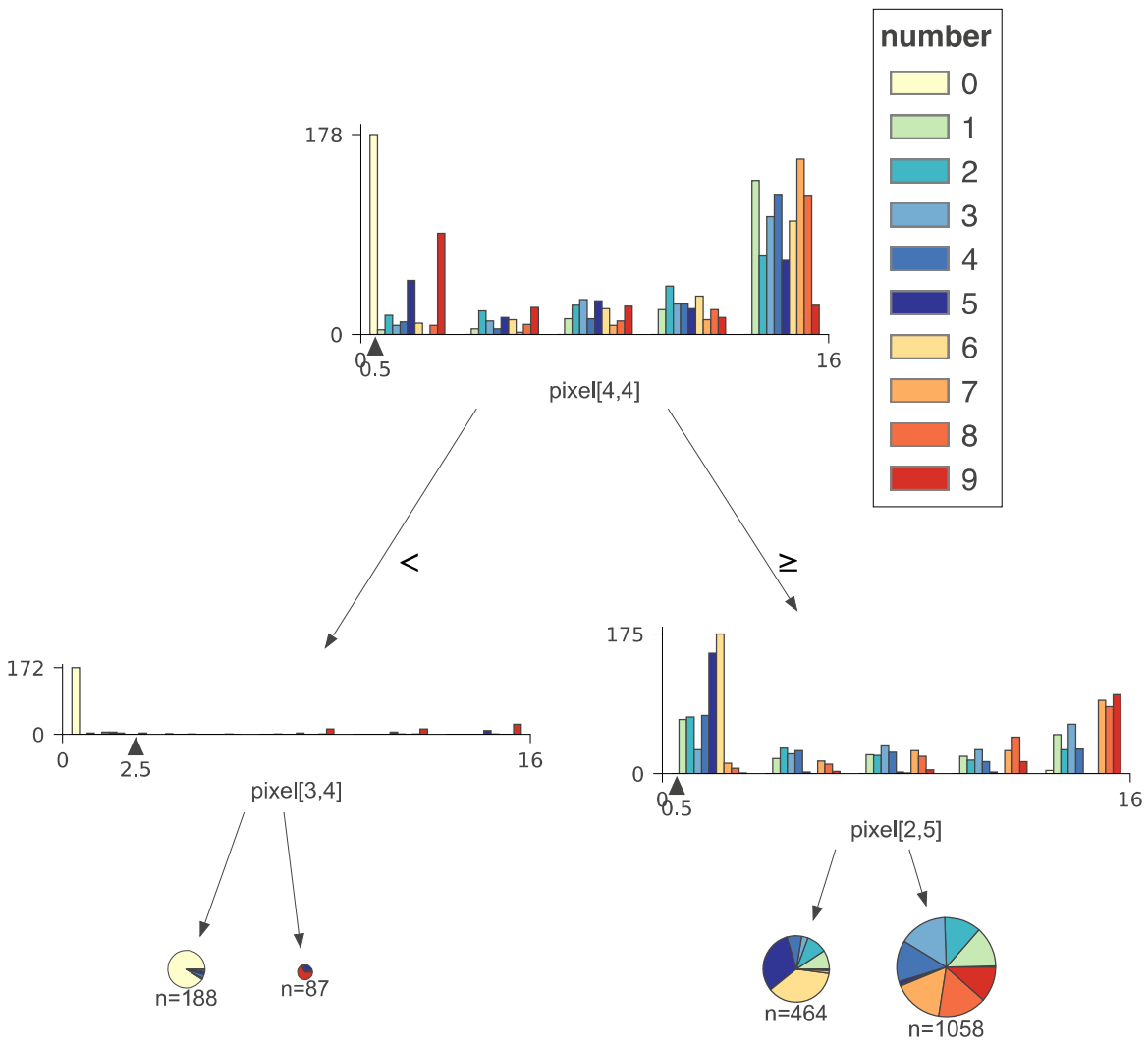


Ignoring color, the histogram shows the PEG feature space distribution. Adding color gives us an indication of the relationship between feature space and target class. For example, in node 0 we can see that samples with `very_low` target class are clustered at the low end of PEG feature space and samples with `High` target class are clustered at the high-end. As with the regressor, the feature space of a left child is everything to the left of the parent's split point in the same feature space; similarly for the right child. For example, combining the histograms of nodes 9 and 12 yields the histogram of node 8. We

force the horizontal axis range to be the same for all PEG decision nodes so that decision nodes lower in the tree clearly box in narrower regions that are more and more pure.

We use a stacked histogram so that overlap is clear in the feature space between samples with different target classes. Note that the height in the Y axis of the stacked histogram is the total number of samples from all classes; multiple class counts are stacked on top of each other.

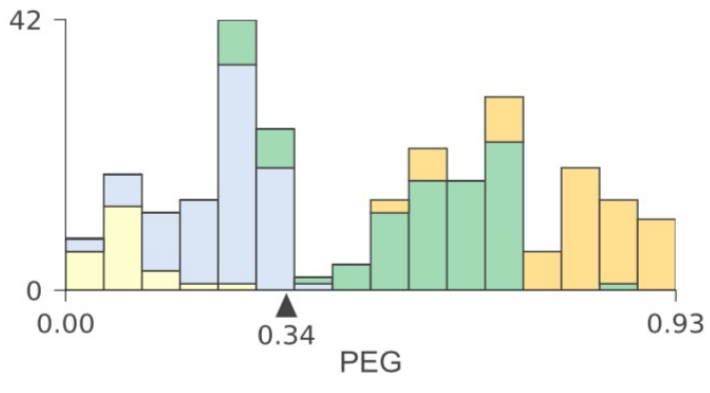
When there are more than four or five classes, the stacked histograms are difficult to read, so we recommend setting the histogram type parameter to `bar` not `barstacked` in this case. With high cardinality target categories, the overlapping distributions are harder to visualize and things break down, so we set a limit of 10 target classes. Here's a shallow tree example using the 10-class Digits data set using non-stacked histograms:

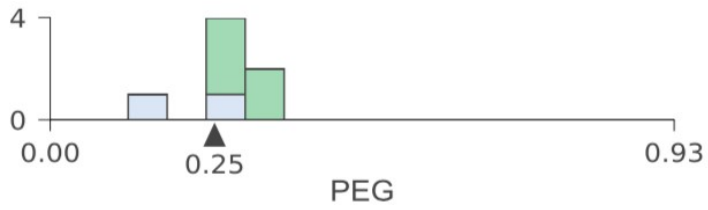


1.6.2 It's all about the details

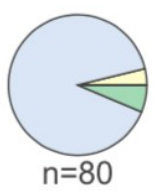
Thus far we've skipped over many of the visual cues and details that we obsessed over during construction of the library and so we hit the key elements here.

Our classifier tree visualizations use node size to give visual cues about the number of samples associated with each node. Histograms get proportionally shorter as the number of samples in the node decrease and leaf node diameters get smaller. The feature space (horizontal axis) is always the same width and the same range for a given feature, which makes it much easier to compare the feature-target spaces of different nodes. The bars of all histograms are the same width in pixels. We use just the start/stop range labels for both horizontal and vertical axes to reduce clutter.

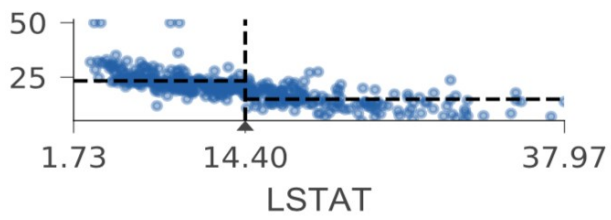




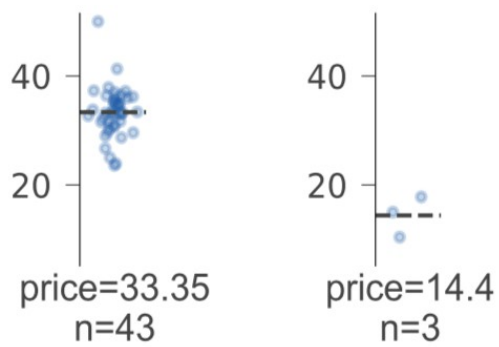
We use a pie chart for classifier leaves, despite their bad reputation. For the purpose of indicating purity the viewer only needs an indication of whether there is a single strong majority category. The viewer does not need to see the exact relationship between elements of the pie chart, which is one key area where pie charts fail. The color of the pie chart majority slice gives the leaf prediction.



Turning to regressor trees now, we make sure that the target (vertical) axis of all decision nodes is the same height and the same range to make comparing nodes easier. Regressor feature space (horizontal axis) is always the same width and the same range for a given feature. We set a low alpha for all scatterplot dots so that increased target value density corresponds to darker color.



Regressor leaves also show the same range vertically for the target space. We use a strip plot rather than, say, a box plot, because the strip plot shows the distribution explicitly while implicitly showing the number of samples by the number of dots. (We also write out the number of samples in text for leaves.) The leaf prediction is the distribution center of mass (mean) of the strip plot, which we highlight with a dashed line.



There are also a number of miscellaneous details that we think improve the quality of the diagrams:

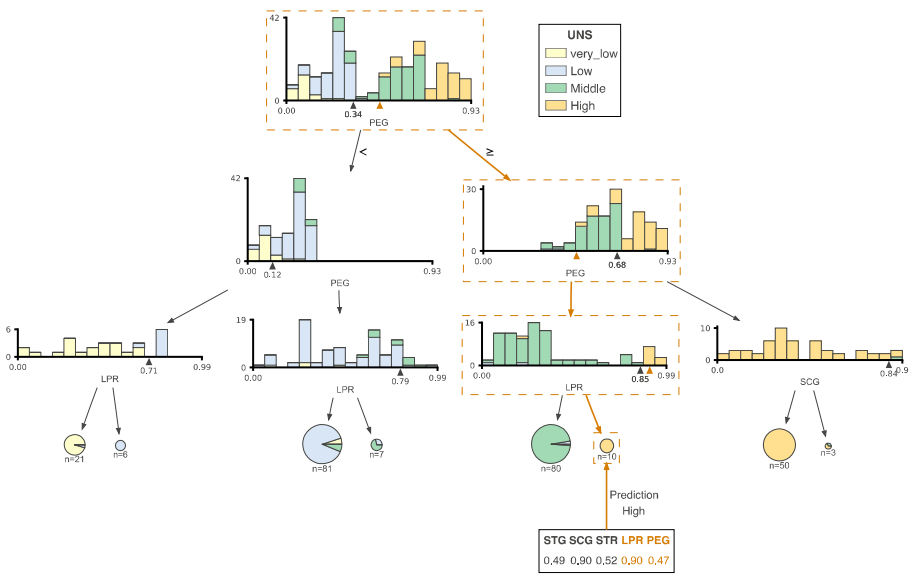
- Classifiers include a legend
- All colors were handpicked from colorblind safe palettes, one handpicked palette per number o target categories (2 through 10)
- We use a gray rather than black for text because it's easier on the eyes
- Lines are hairlines
- We draw outlines of bars in bar charts and slices in pie charts

1.6.3 Visualizing tree interpretation of a single observation

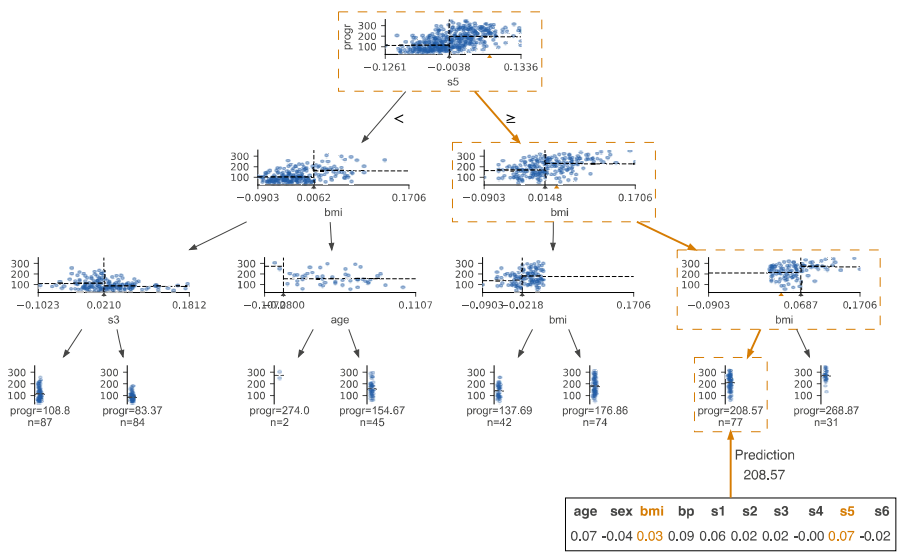
To figure out how model training arrives at a specific tree, all of the action is in the feature-space splits of the decision nodes, which we just discussed. Now, let's take a look at visualizing how a specific feature vector yields a specific prediction. The key here is to examine the decisions taken along the path from the root to the leaf predictor node.

Decision-making within a node is straightforward: take the left path if feature x_i in test vector **x** is less than the split point, otherwise take the right path. To highlight the decision-making process, we have to highlight the comparison operation. For decision nodes along the path to the leaf predictor node, we show an orange wedge at position x_i in the horizontal feature space. This makes the comparison easy tc see; if the orange wedge is to the left of the black wedge, go left else go right. Decision nodes involved in the prediction process are surrounded by boxes with dashed lines and the child edges are thicker and colored orange. Here are two sample trees showing test vectors (click on images to expand):

KNOWLEDGE data with test vector



Diabetes data with test vector

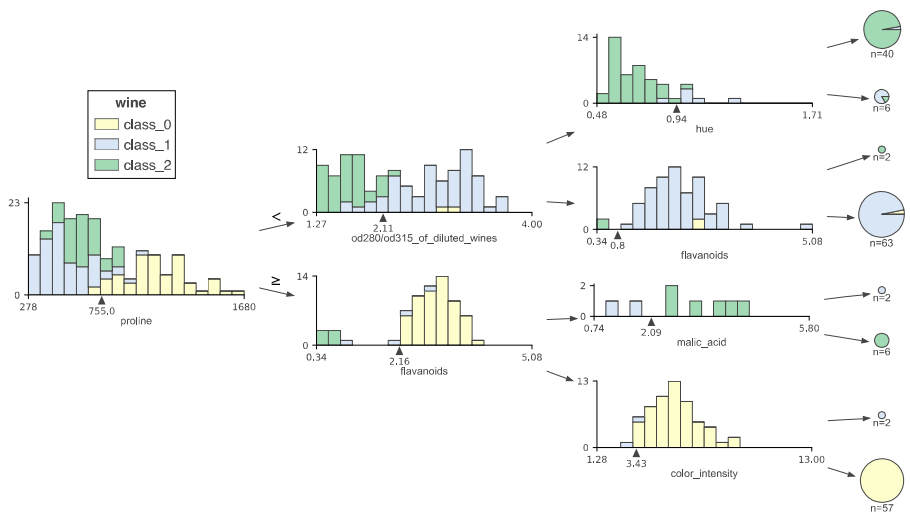


The test vector **x** with feature names and values appears below the leaf predictor node (or to the right ir left-to-right orientation). The test vector highlights the features used in one or more decision nodes. When the number of features reaches a threshold of 20 (10 for left-to-right orientation), test vectors do not show unused features to avoid unwieldly test vectors.

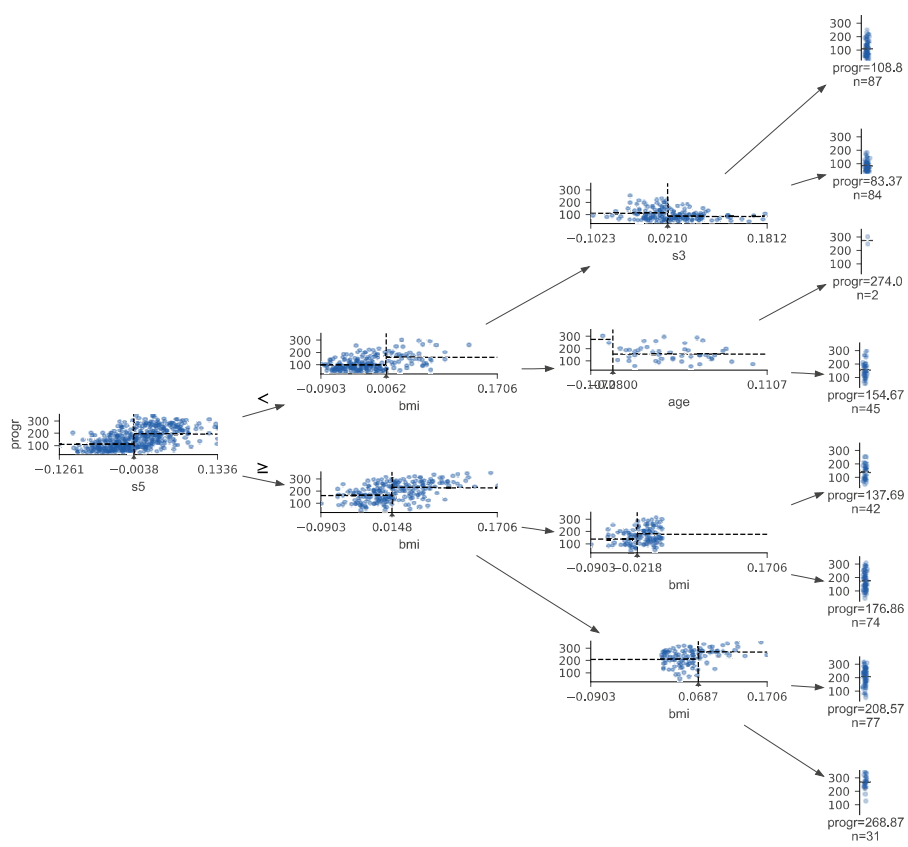
1.6.4 Left-to-right orientation

Some users have a preference for left-to-right orientation instead of top-down and sometimes the nature of the tree simply flows better left-to-right. Sample feature vectors can still be run down the tree with the left-to-right orientation. Here are some examples (click on the images to enlarge):

Wine

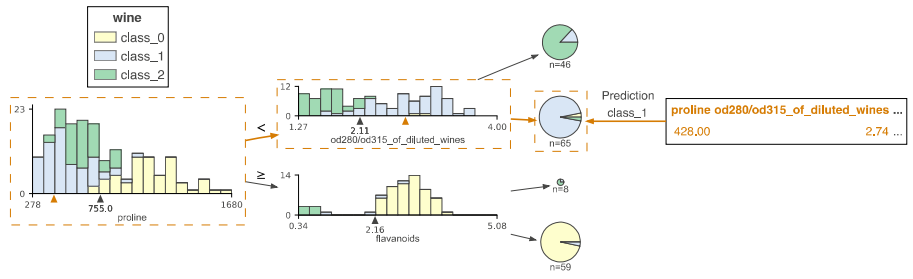


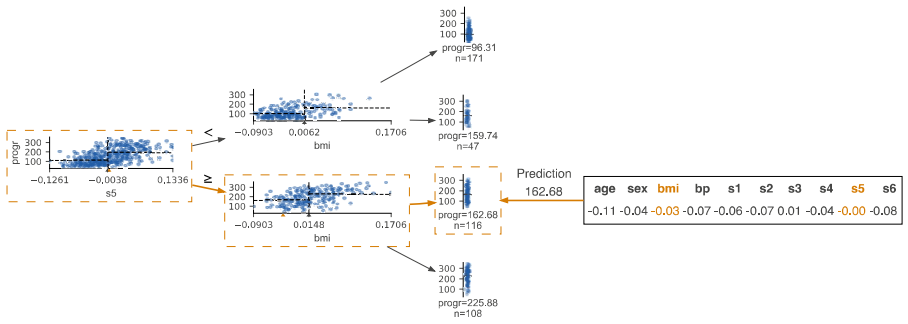
Diabetes



Wine showing a prediction

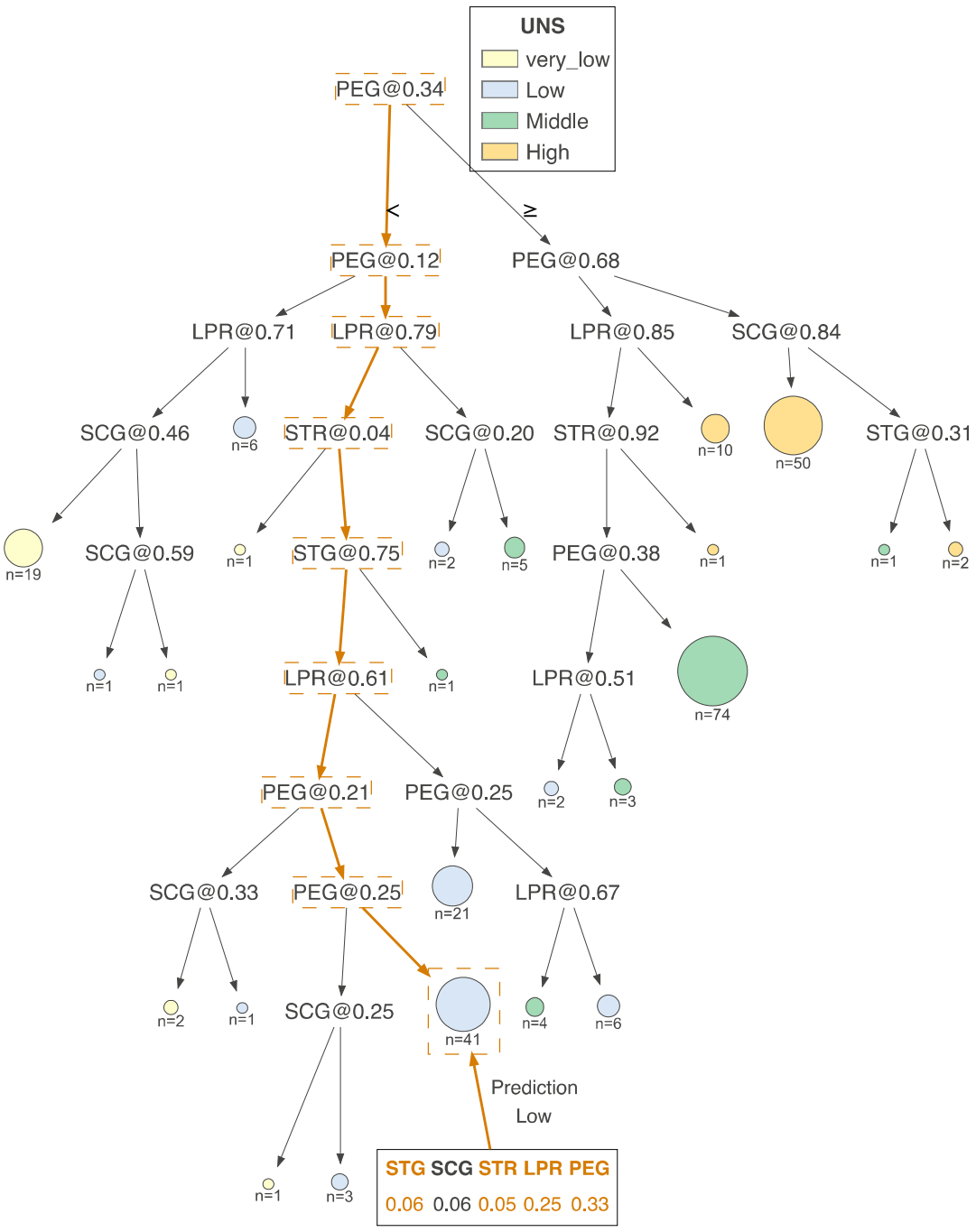
Diabetes showing a prediction

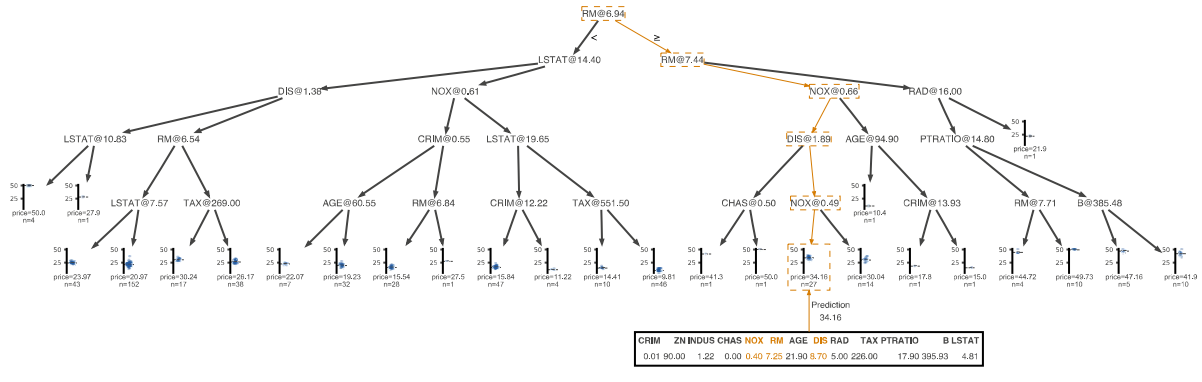




1.6.5 Simplified non-fancy layout

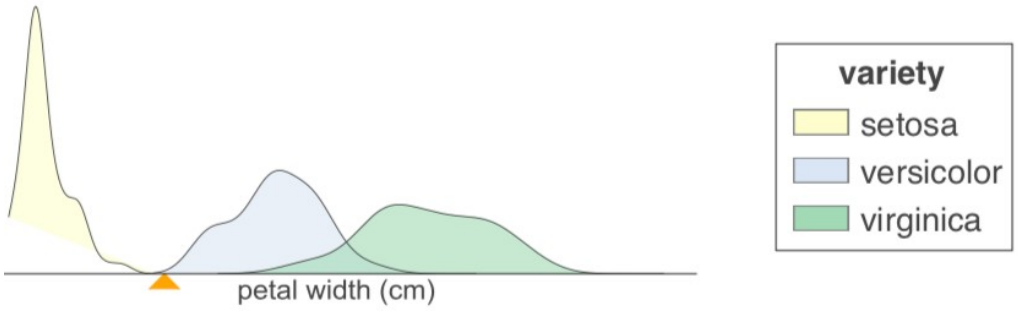
To evaluate the generality of a decision tree, if it often helps to get a high-level overview of the tree. This generally means examining things like tree shape and size, but more importantly, it means looking at the leaves. We'd like to know how many samples each leaf has, how pure the target values are, and just generally where most of the weight of samples falls. Getting an overview is harder when the visualization is too large and so we provide a "non-fancy" option that generates smaller visualizations while retaining key leaf information. Here are a sample classifier and a regressor in non-fancy mode with top-down orientation:



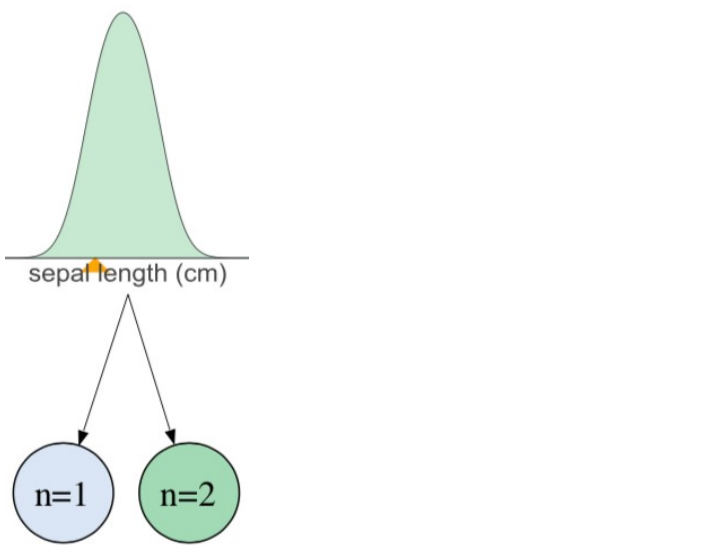


1.7 What we tried and rejected

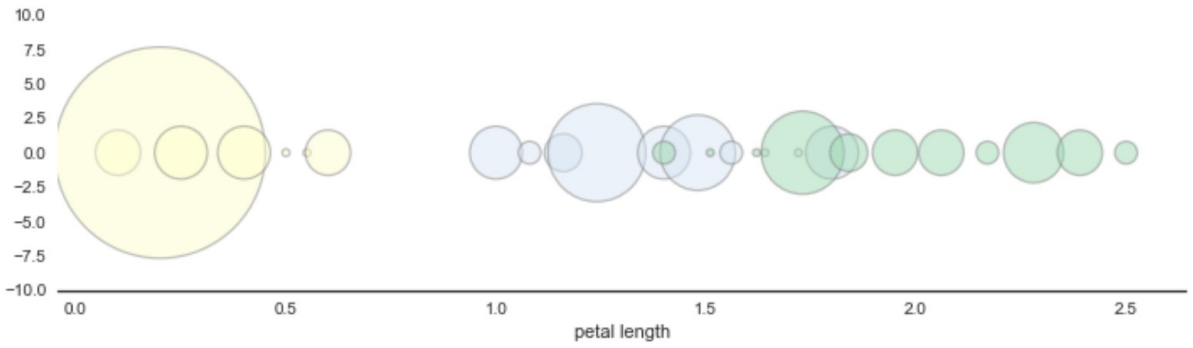
Those interested in these tree visualizations from a design point of view might find it interesting to read about what we tried and rejected. Starting with classifiers, we thought that the histograms were a bit blocky and perhaps kernel density estimates would give a more accurate picture. We had decision nodes that looked like this:



The problem is that decision nodes with only one or two samples gave extremely misleading distributions:



We also experimented using bubble charts instead of histograms for classifier decision nodes:



These look really cool but, in the end, histograms are easier to read.

Turning to regression trees, we considered using box plots to show the distribution of prediction values and also used a simple bar chart to show the number of samples:

This dual plot for each leaf is less satisfying than the strip plot we use now. The box plot also doesn't show the distribution of target values nearly as well as a strip plot. Before the strip plot, we just laid out the target values using the sample index value as the horizontal axis:

That is misleading because the horizontal axis is usually feature space. We scrunched that into a strip plot.

1.8 Code sample

This section gives a sample visualization for the Boston regression data set and the Wine classification data set. You can also check out the full gallery of sample visualizations and the code to generate the samples.

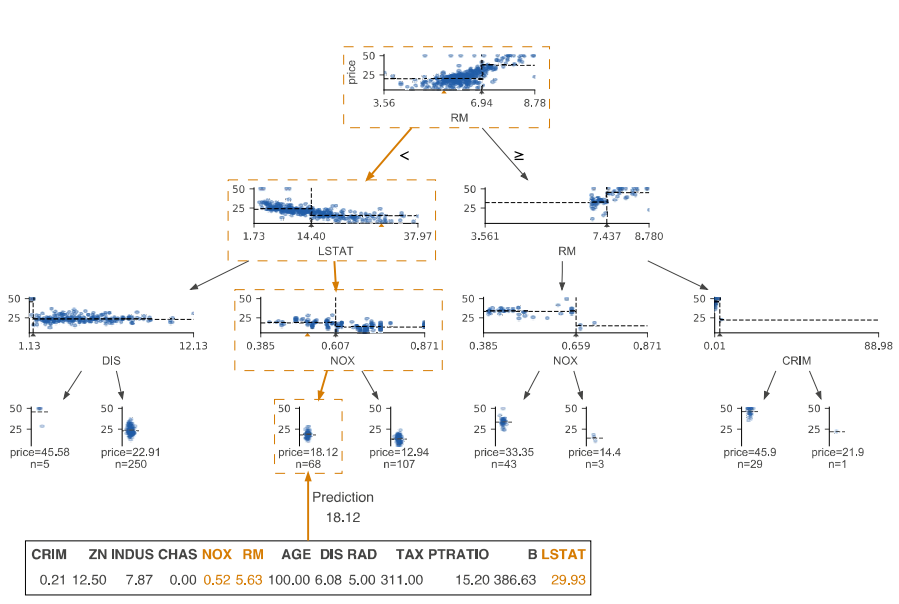
1.8.1 Boston regression tree visualization

Here is a code snippet to load the Boston data and train a regression tree with a maximum depth of three decision nodes:

```
boston = load_boston()

X_train = boston.data
y_train = boston.target
testX = X_train[5,:]

regr = tree.DecisionTreeRegressor(max_depth=3)
regr = regr.fit(X_train, y_train)
```

The code to visualize the tree involves passing the tree model, the training data, feature and target names, and a test vector (if desired):

```
viz = dtreeviz(regr, X_train, y_train, target_name='price',
               feature_names=boston.feature_names,
               X = testX)
viz.save("boston.svg") # suffix determines the generated image format
viz.view()             # pop up window to display image
```



1.8.2 Wine classification tree visualization

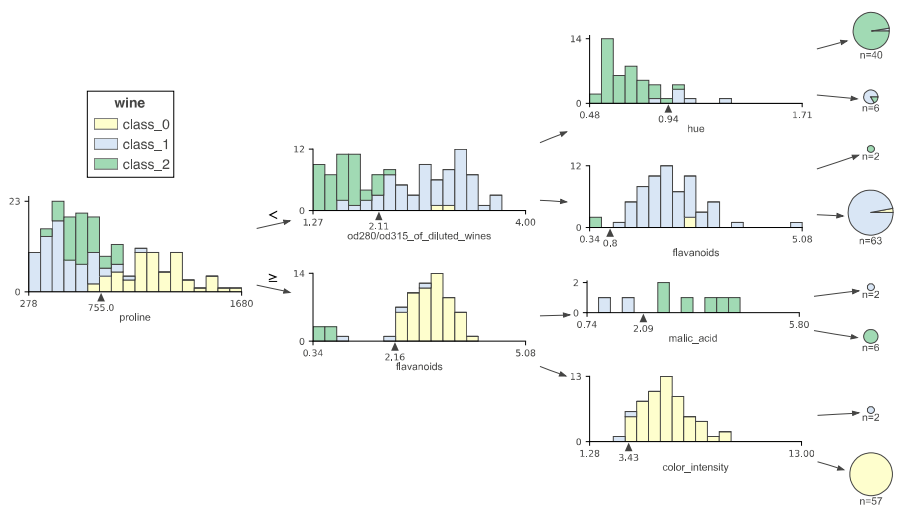
Here is a code snippet to load the Wine data and train a classifier tree with a maximum depth of three decision nodes:

```
clf = tree.DecisionTreeClassifier(max_depth=3)
```

```
wine = load_wine()

clf.fit(wine.data, wine.target)
```

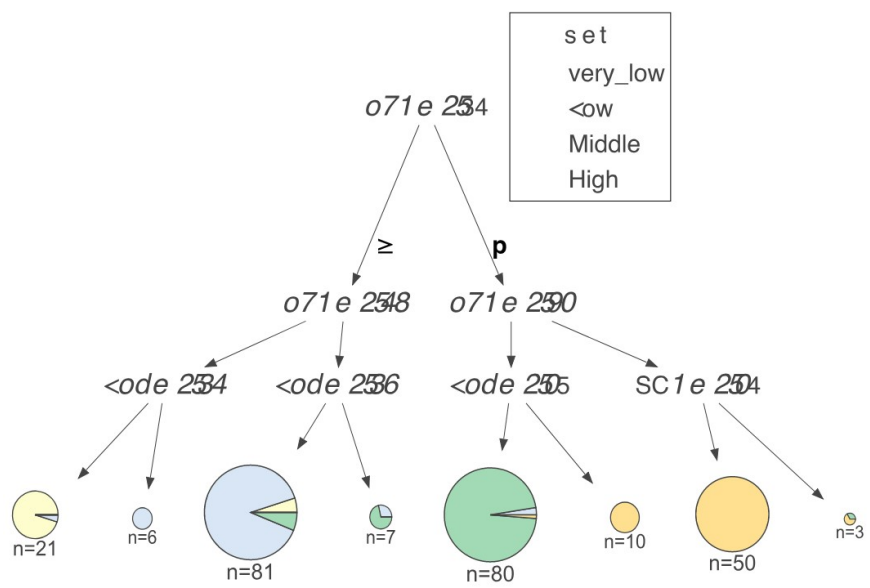
Visualizing a classifier is the same as visualizing a regressor, except that the visualization needs the target class names:

```
viz = dtreeviz(clf, wine.data, wine.target, target_name='wine',
               feature_names=wine.feature_names,
               class_names=list(wine.target_names))
viz.view()
```



In Jupyter notebooks, the object returned from `dtreeviz()` has a `_repr_svg_()` function that Jupyter uses to display the object automatically. See notebook of examples.

BUG IN JUPYTER NOTEBOOK
Jupyter notebooks currently, as of September 2018, do not display the SVG generated by this library properly. The fonts etc... are all messed up:



The good news is that github displays them properly as does JupyterLab.

Use `Image(viz.topng())` to display (poorly) in Juypter notebook or simply call `viz.view()`, which will pop up a window that displays things properly.

1.9 Our implementation

This project was very frustrating with lots of programming deadends, fiddling with parameters, working around bugs/limitations in tools and libraries, and creatively mashing up a bunch of existing tools. The only fun part was the (countless) sequence of experiments in visual design. We pushed through because it seemed likely that the machine learning community would find these visualization

as useful as we will. This project represents about two months of trudging through stackoverflow, documentation, and hideous graphics programming.

At the highest level, we used matplotlib to generate images for decision and leaf nodes and combined them into a tree using the venerable graphviz. We also used HTML labels extensively in the graphviz tree description for layout and font specification purposes. The single biggest headache was convincing all components of our solution to produce high-quality vector graphics.

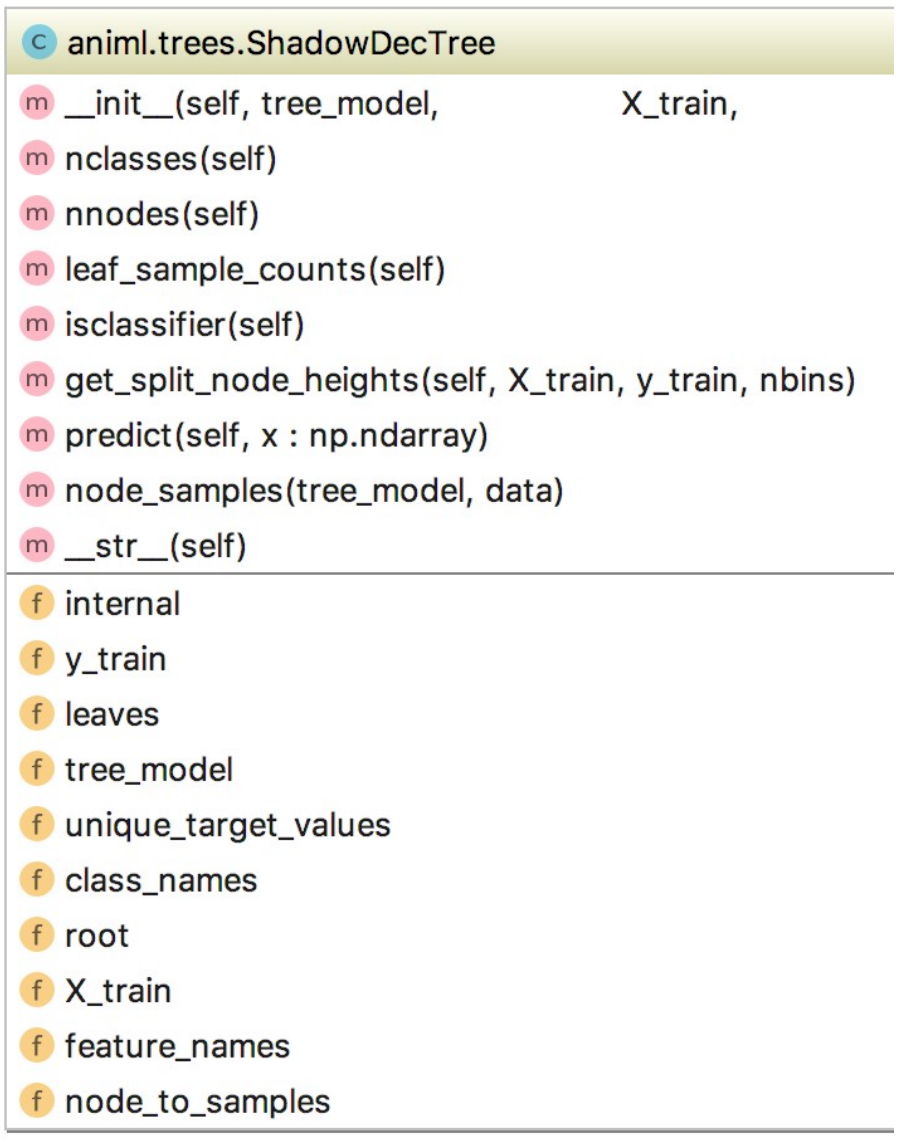
Our initial coding experiments led us to create a shadow tree wrapping the decision trees created by scikit, so let's start with that.

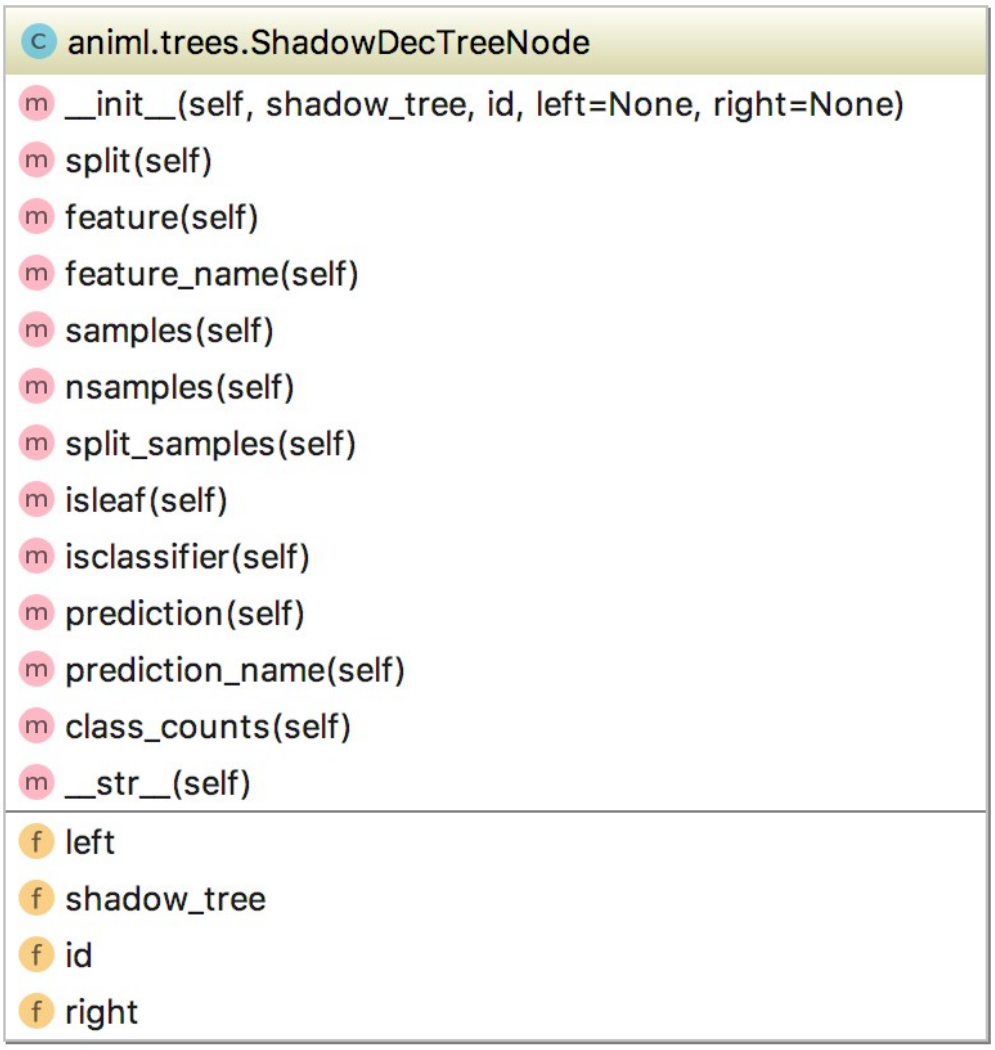
1.9.1 Shadow trees for scikit decision trees

The decision trees for classifiers and regressors from scikit-learnare built for efficiency, not necessarily ease of tree walking or extracting node information. We created dtreeviz.shadow.ShadowDecTree and `dtreeviz.shadow.ShadowDecTreeNode` classes as an easy-to-use (traditional binary tree) wrapper for all tree information. Here's how to create a shadow tree from a scikit classifier or regressor tree model:

```
shadow_tree = ShadowDecTree(tree_model, X_train, y_train, feature_names, class_names)
```

The shadow tree/node classes have a lot of methods that could be useful to other libraries and tools that need to walk scikit decision trees. For example, `predict()` not only runs a feature vector through the tree but also returns the path of visited nodes. The samples associated with any particular node can be had through `node_samples()`.





1.9.2 Tool mashup

Graphviz dot tree layout language is very useful for getting decent tree layouts if you know all of the tricks, such as getting left children to appear to the left of right children with interconnecting hidden graph edges. For example, if you have two leaves, `leaf4` and `leaf5`, that must appear left to right on the same level, here is the graphviz magic:

```
LSTAT3 -> leaf4 [penwidth=0.3 color="#444443" label=<>]
LSTAT3 -> leaf5 [penwidth=0.3 color="#444443" label=<>]
{
    rank=same;
    leaf4 -> leaf5 [style=invis]
}
```

We usually use HTML labels on graphviz nodes rather than just text labels because they give much more control over text display and provide an ability to show tabular data as actual tables. For example, when displaying a test vector run down the tree, the test vector is shown using an HTML table:

STG	SCG	STR	LPR	PEG
0.39	0.42	0.83	0.65	0.19

To generate generate images from graphviz files, we use the `graphviz` python package, which ends up `exec`ing the `dot` binary executable using one of its utility routines (`run()`). Occasionally, we used slightly different parameters on the `dot` command and so we just directly call `run()` like this for flexibility:

```
cmd = ["dot", "-Tpng", "-o", filename, dotfilename]
stdout, stderr = run(cmd, capture_output=True, check=True, quiet=False)
```

We also use the `run()` function to execute the `pdf2svg` (PDF to SVG conversion) tool, as described in the next section.

1.9.3 Vector graphics via SVG

We use matplotlib to generate the decision and leaf nodes and, to get the images into a graphviz/dot image, we use HTML graphviz labels and then reference the generated images via `img` tags like this:

```
<img src="/tmp/node3_94806.svg"/>
```

The 94806 number is the process ID, which helps isolate multiple instances of `dtreeviz` running on the same machine. Without this, it's possible for multiple processes to overwrite the same temporary files.

Because we wanted scalable, vector graphics, we tried importing SVG images initially but we could not get graphviz to accept those files (pdf neither). It took us four hours to figure out that generating and importing SVG were two different things and we needed the following magic incantation on OS X using `--with-librsvg`:

```
$ brew install graphviz --with-librsvg --with-app --with-pango
```

Originally, when we resorted to generating PNG files from matplotlib, we set the dots per inch (dpi) to be 450 so that they looked okay on high resolution screens like the iMac. Unfortunately, that meant we had to specify the actual size we wanted for the overall tree using an HTML table in graphviz using `width` and `height` parameters on `<td>` tags. That cause a lot of trouble because we had to figure out what the aspect ratio was coming out of matplotlib. Once we moved to SVG files, we unnecessarily parsed the SVG files to get the size for use in the HTML; as we wrote this document we realized extracting the size information from SVG files was unnecessary.

Unfortunately, graphviz's SVG output simply referenced the node files that we imported, rather than embedding the node images within the overall tree image. This is a very inconvenient form because sending a single tree visualization means sending a zip of files rather than a single file. We spent the time to parse SVG XML and embed all referenced images within a single large meta-SVG file. That worked great and there was much celebration.

Then we noticed that graphviz does not properly handle text in HTML labels when generating SVG. For example, the text of classifier tree legends was cut off and overlapping. Rats.

What finally worked to get a single clean SVG file was first generating a PDF file from graphviz and then converting the PDF to SVG with `pdf2svg` (`pdf2cairo` also appears to work).

Then we noticed that Jupyter notebook has a bug where it does not display those SVG files properly (see above). Jupyter lab does handle the SVG properly as does github. We added a `topng()` method so users of Jupyter notebook can use `Image(viz.topng())` to get inline images. Better yet, call `viz.view()`, which will pop up a window that displays images properly.

1.10 Lessons learned

Sometimes solving a programming problem is less about algorithms and more about working within the constraints and capabilities of the programming ecosystem, such as tools and libraries. That is definitely the case with this decision tree visualization software. The programming was not hard; it was more a matter of fearlessly bashing our way to victory through an appropriate mashup of graphics tools and libraries.

Designing the actual visualization also required a seemingly infinite number of experiments and tweaks. Generating high quality vector-based images also required pathological determination and a

trail of dead code left along the circuitous path to success.

We are definitely not visualization aficionados, but for this specific problem we banged on it until we got effective diagrams. In Edward Tufte's seminar I learned that you can pack a lot of information into a rich diagram, as long as it's not an arbitrary mishmash; the human eye can resolve lots of details. We used a number of elements from the design palette to visualize decision trees: color, line thickness, line style, different kinds of plots, size (area, length, graph height, ...), color transparency (alpha), text styles (color, font, bold, italics, size), graph annotations, and visual flow. All visual elements had to be motivated. For example, we didn't use color just because colors are nice. We used color to highlight an important dimension (target category) because humans quickly and easily pick out color differences. Node size differences should also be easily picked out by humans. (is that a kitty cat or lion?), so we used that to indicate leaf size.

1.11 Future work

The visualizations described in this document are part of the dtreeviz machine learning library, which is just getting started. We'll likely moved the rfpimp permutation importance library into `dtreeviz` soon. At this point, we haven't tested the visualizations on anything but OS X. We'd welcome instructions from programmers on other platforms so that we could include those installation steps in the documentation.

There are a couple of tweaks we like to do, such as bottom justifying the histograms and classifier trees so that it's easier to compare notes. Also, some of the wedge labels overlap with the axis labels. Finally, it would be interesting to see what the trees look like with incoming edge thicknesses proportional to the number of samples in that node.