Lecture 11: Machine Learning

Tree-Based Machine Learning Methods

James Sears*
AFRE 891 SS24
Michigan State University

*Parts of these slides are adapted from **"Prediction and Machine-Learning in Econometrics"** by Ed Rubin, used under **CC BY-NC-SA 4.0**.

Table of Contents

Prologue

Packages we'll use today:

```
pacman::p_load(grf, janitor, magrittr, parsnip, ranger, rpart, rpart.plot
```

As well, let's use two datasets:

- The ISL default data again
- Heart disease data

```
default_df ← ISLR::Default

heart_df ← read_csv("data/Heart.csv") %>%
  dplyr::select(-1) %>%
  rename(HeartDisease = AHD) %>%
  clean_names()
```

Prologue

Throughout our econometric training, we've become experts in predicting an outcome by writing down a model with a **specific functional form** for the relationship between $\mathbf{y}$ and $\mathbf{X}$.

Decision Trees (or classification and regression trees, CART) are a supervised learning method that offers an alternative, **data-driven** way of generating predictions by partitioning the covariate space into groups and using the grouping to generate predictions.

After working through decision trees, we'll shift our focus to **forest methods** that combine many, many trees to overcome drawbacks of individual trees and yield predictions of alternate objects - including heterogeneous treatment effects.

Decision Trees

Decision Trees: Punchline

Goal

- Split the *predictor space* (our $\mathbf{X}$) into regions (partitions)
- Predict the most-common value within a region

Attributes of Decision Trees

1. Work for **both classification and regression**
2. Are inherently **nonlinear**
3. Are relatively **simple** and **interpretable**
4. Often **underperform** relative to competing methods
5. easily extend to **very competitive ensemble methods** (forests with *many* trees)🌲

🌲 Though the ensembles will be much less interpretable.

Growing Decision Trees

1. Divide the predictor space into J regions (using predictors $\mathbf{x}_1, \ldots, \mathbf{x}_p$)

- **Regression Trees:** Choose the regions to **minimize RSS** across all J regions

$$\sum_{j=1}^{J} \left(y_i - \hat{y}_{R_j} \right)^2$$

Growing Decision Trees: Steps

1. Divide the predictor space into J regions (using predictors $\mathbf{x}_1, \ldots, \mathbf{x}_p$)

- **Classification Trees:** Choose the regions to **minimize Gini index** or **Entropy**

$$G = \sum_{k=1}^{K} \hat{p}_{mk} (1 - \hat{p}_{mk}) \qquad Entropy = -\sum_{k=1}^{K} \hat{p}_{mk} \log(\hat{p}_{mk})$$

- Keep splitting until a specified threshold is reached (e.g. at most 5 observations per region)

Classification Tree

Q: Why are we using the Gini index or entropy (*vs.* error rate)?

A: The error rate isn't sufficiently sensitive to grow good trees.
The Gini index and entropy tell us about the **composition** of the leaf.

Consider two different leaves in a three-level classification.

Leaf 1

- A: 51, B: 49, C: 00
- **Error rate:** 49%
- **Gini index:** 0.4998
- **Entropy:** 0.6929

Leaf 2

- A: 51, B: 25, C: 24
- **Error rate:** 49%
- **Gini index:** 0.6198
- **Entropy:** 1.0325

The **Gini index** and **entropy** tell us about the distribution.

Growing Decision Trees: Steps

2. Make predictions using the regions' mean outcome.

For region R_j predict $\hat{y}_{R_j}$ where

$$\hat{y}_{R_j} = \frac{1}{n_j} \sum_{i \in R_j} y$$

Growing Decision Trees

Let's visualize this by growing a classification tree for predicting credit card defaults based on income and card balances.

Growing Decision Trees

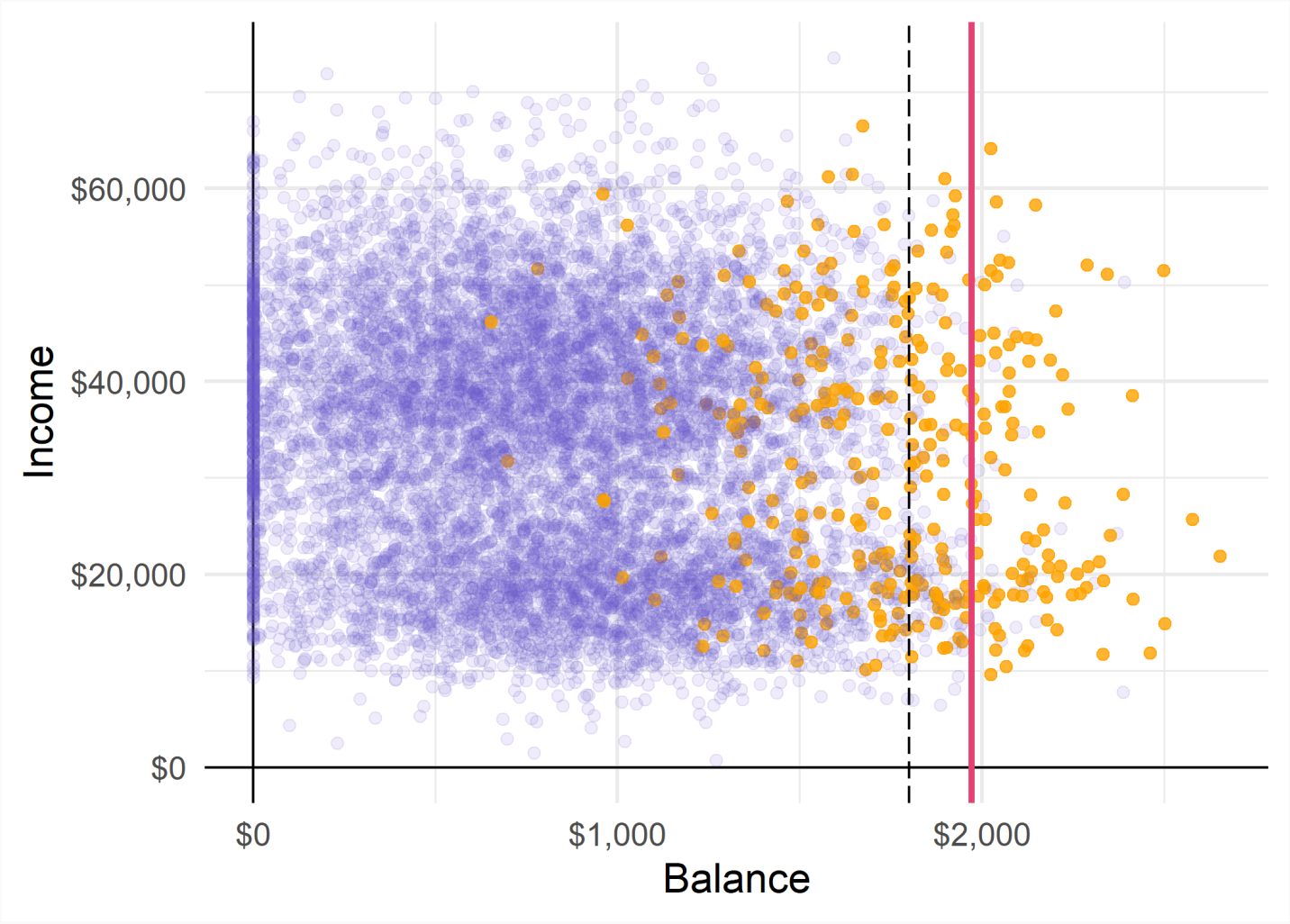
Consider our two-dimensional covariate space:

Growing Decision Trees

The **first partition** splits balance at $1,800.

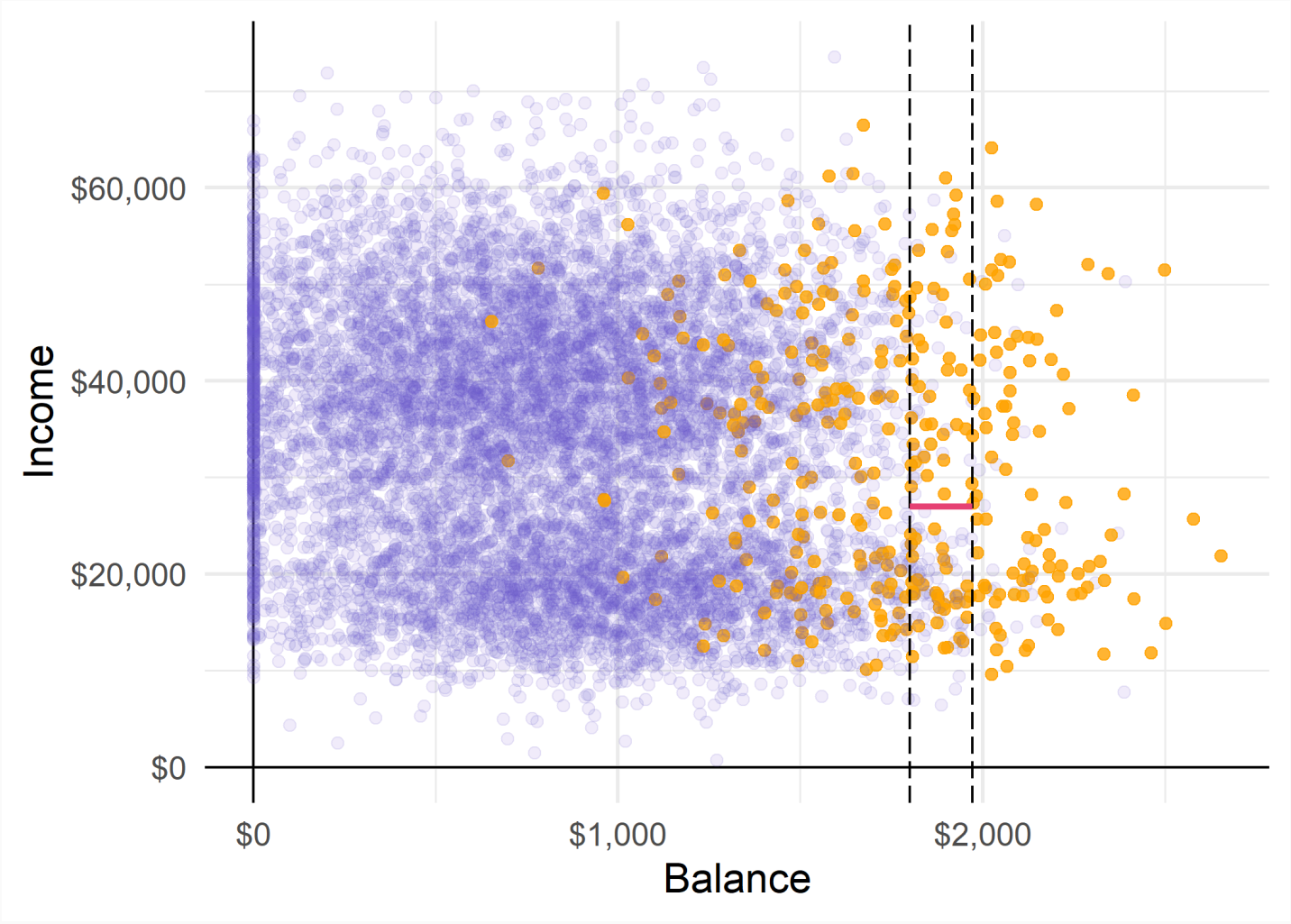
Growing Decision Trees

The **second partition** splits balance at \$1,972, (*conditional on bal. > \$1,800*).



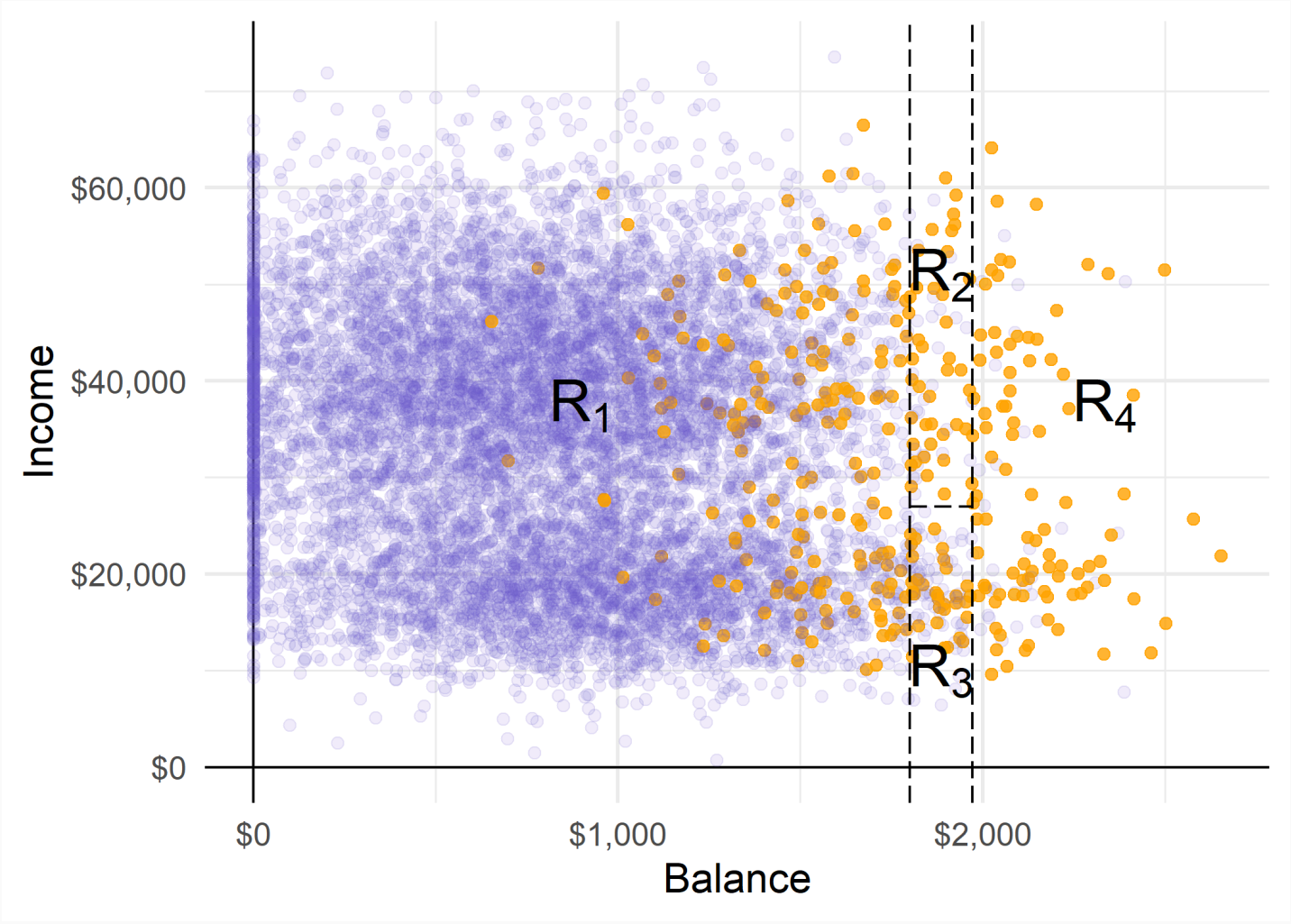
Growing Decision Trees

The **third partition** splits income at $27K for bal. between $1,800 and $1,972.



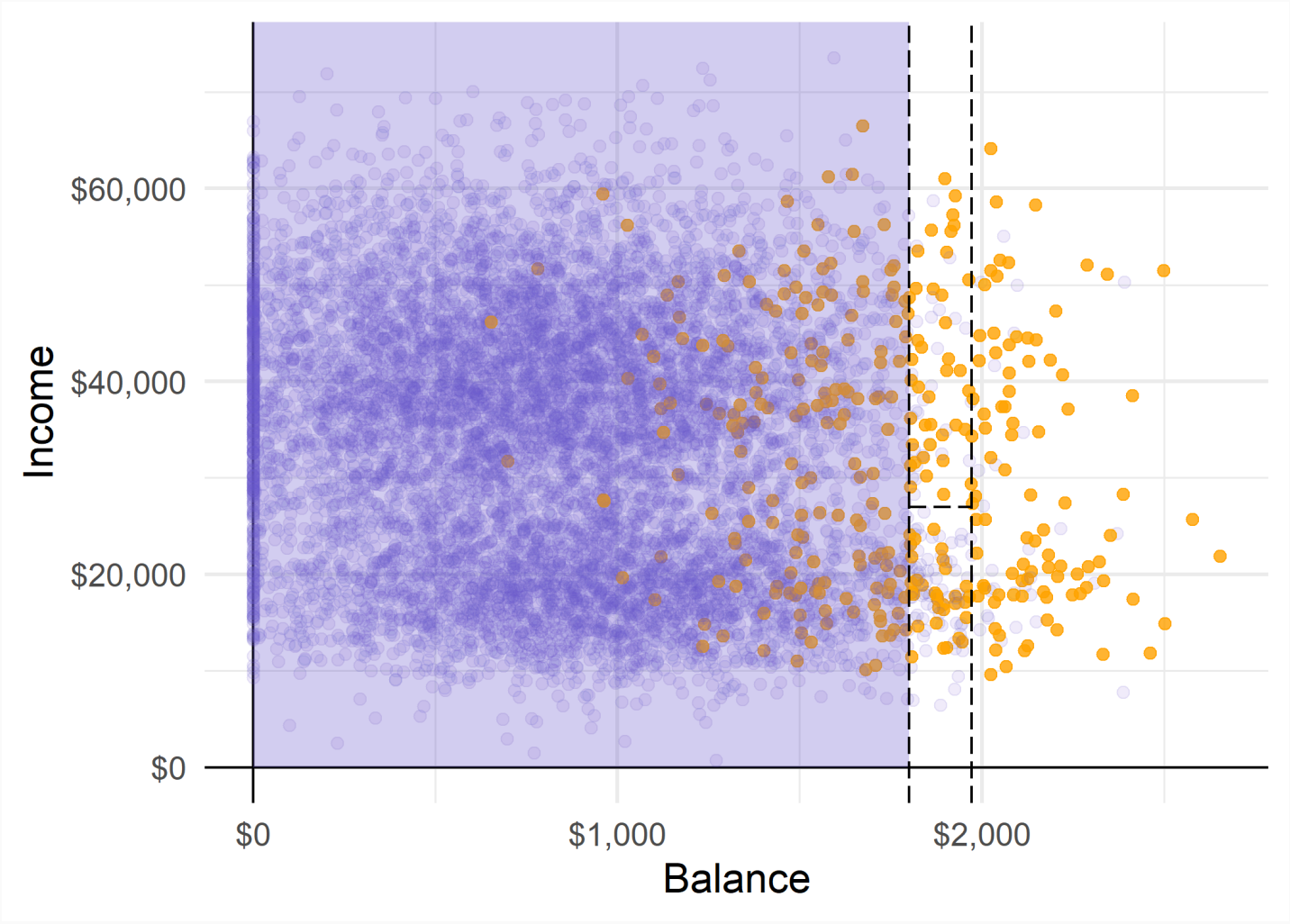
Growing Decision Trees

These three partitions give us four **regions**...



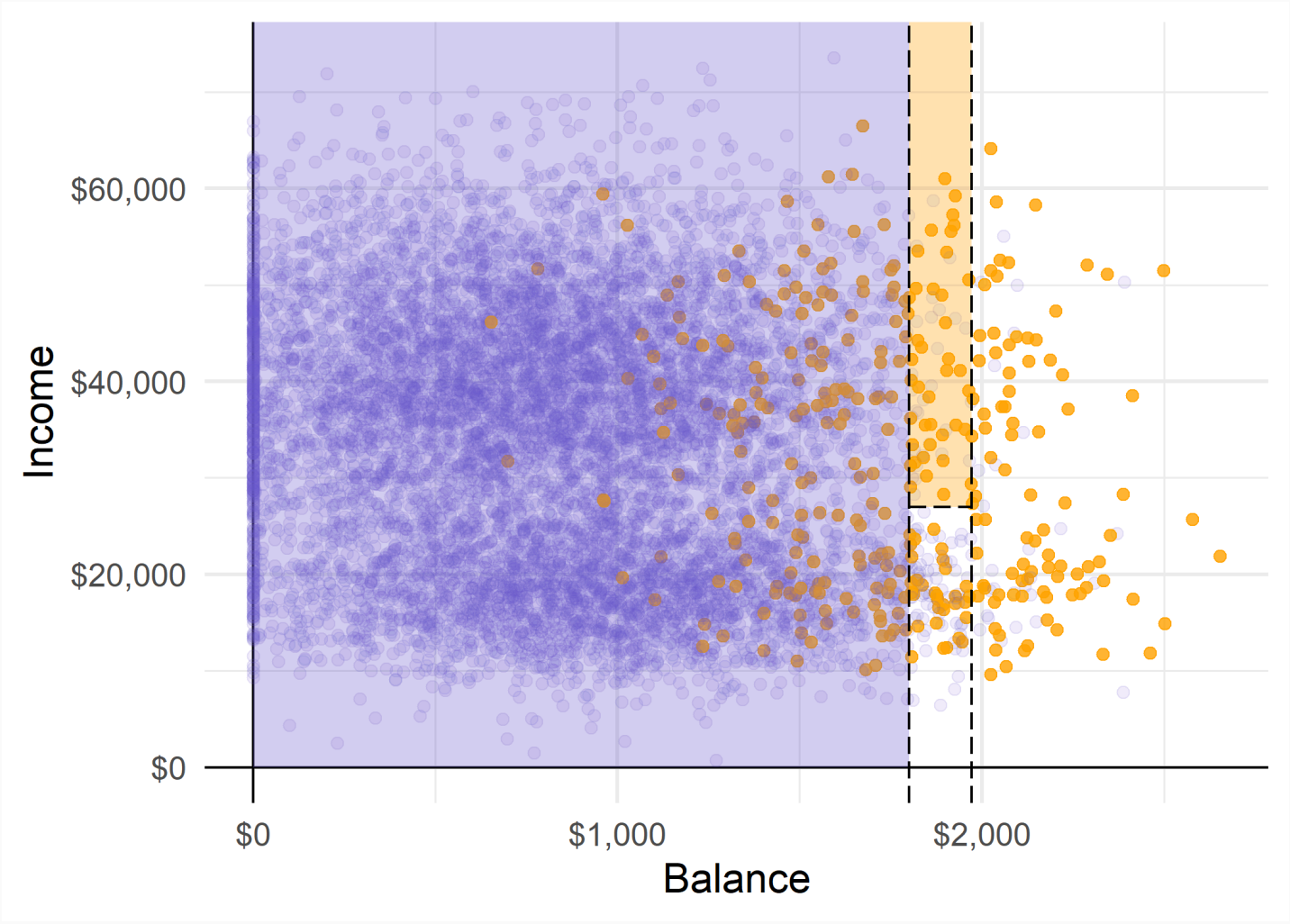
Growing Decision Trees

Predictions cover each region (e.g. using the region's **most common class**).



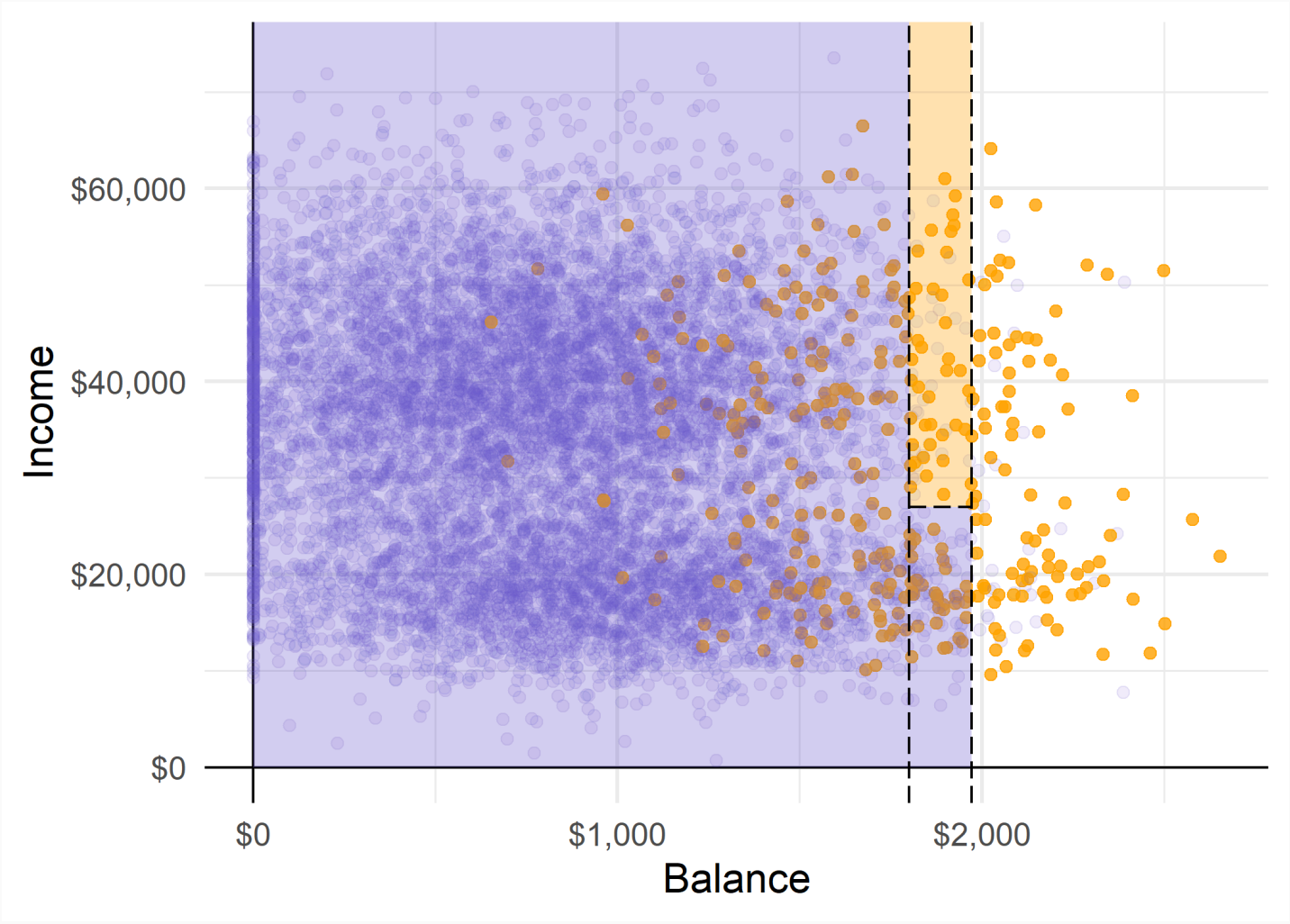
Growing Decision Trees

Predictions cover each region (e.g. using the region's **most common class**).



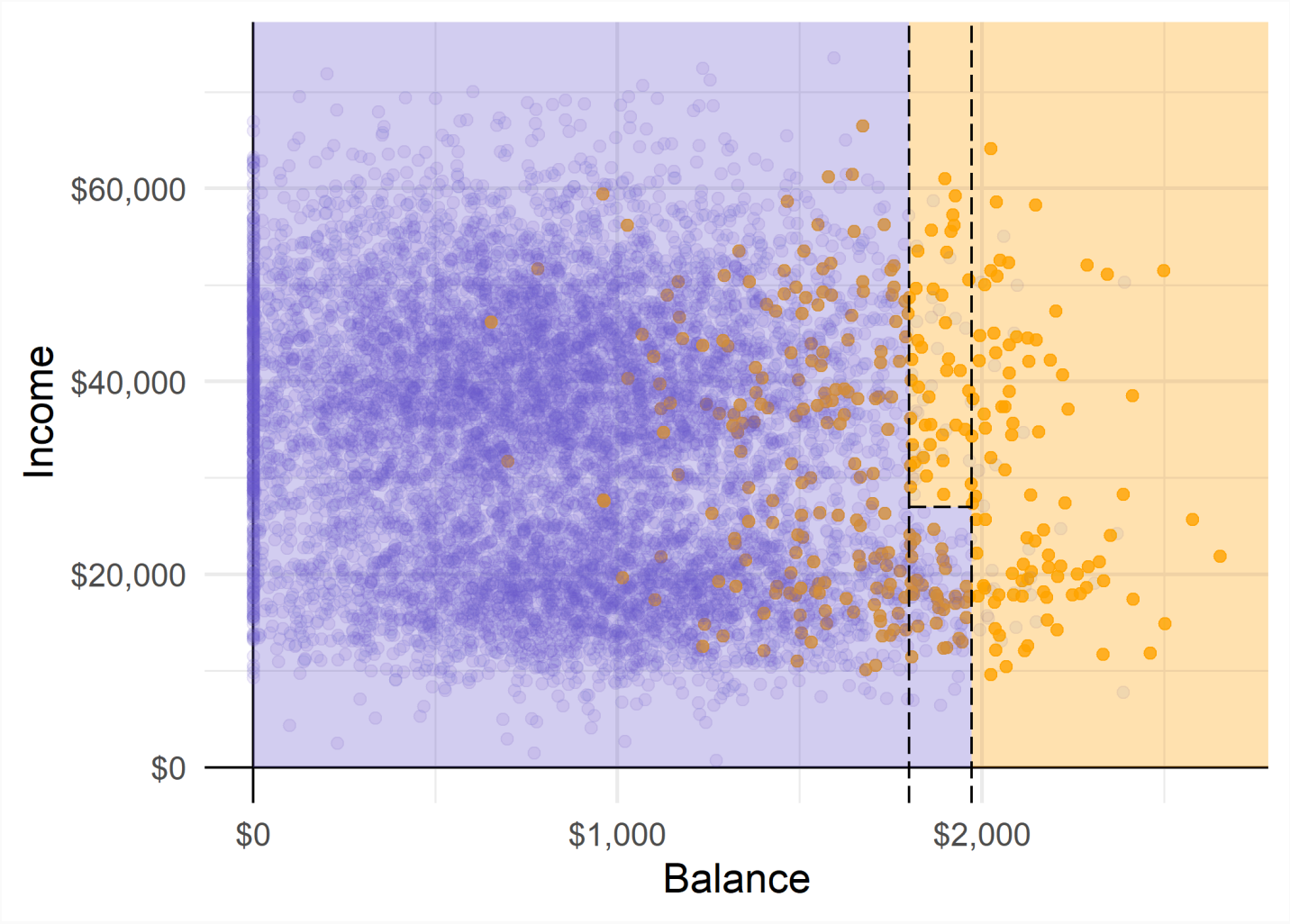
Growing Decision Trees

Predictions cover each region (e.g. using the region's **most common class**).



Growing Decision Trees

Predictions cover each region (e.g. using the region's **most common class**).



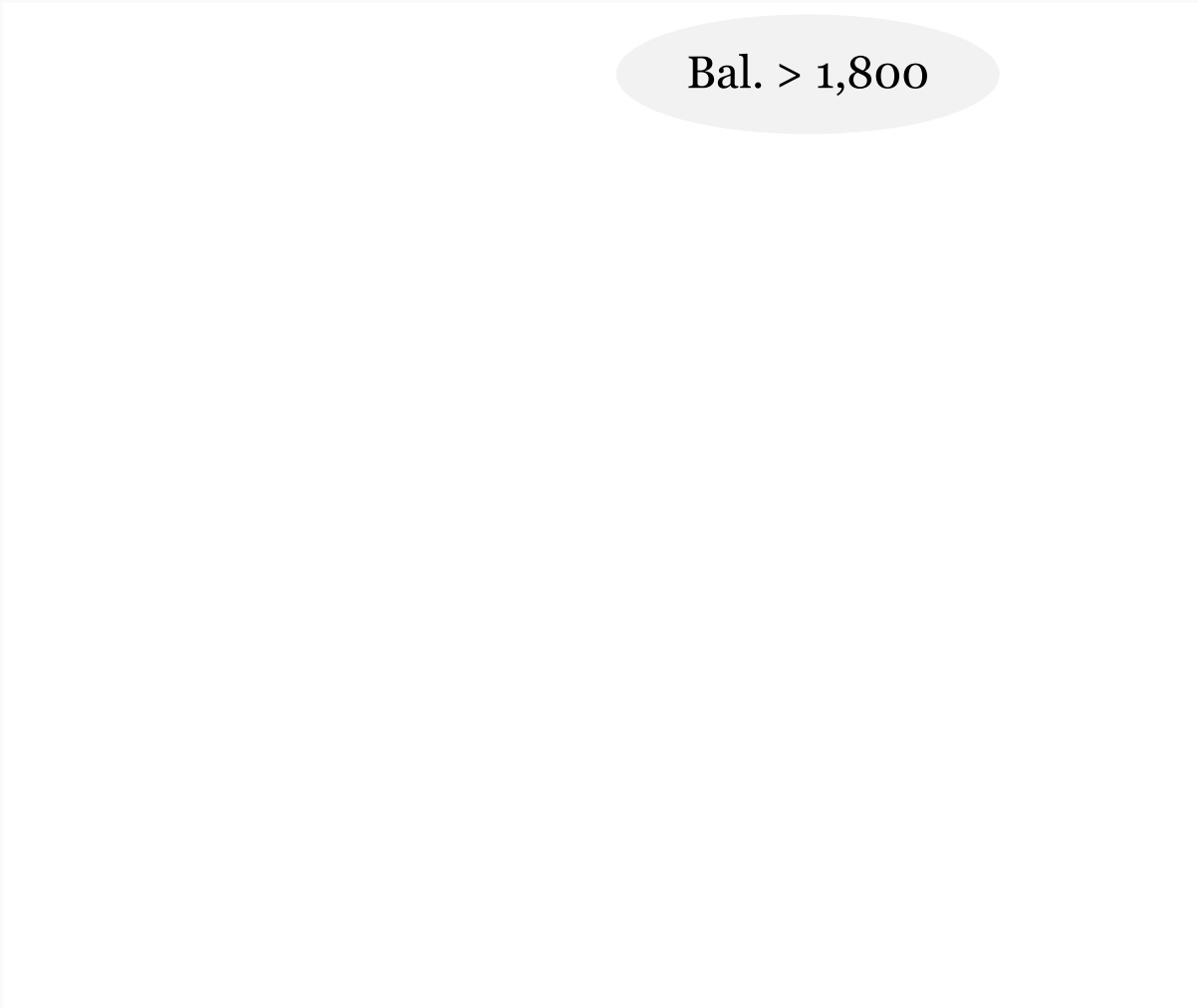
Visualizing the Decision Tree

Q: Why do we call it a tree?

A: Because we can easily represent this partitioning process in the form of a **decision tree!**

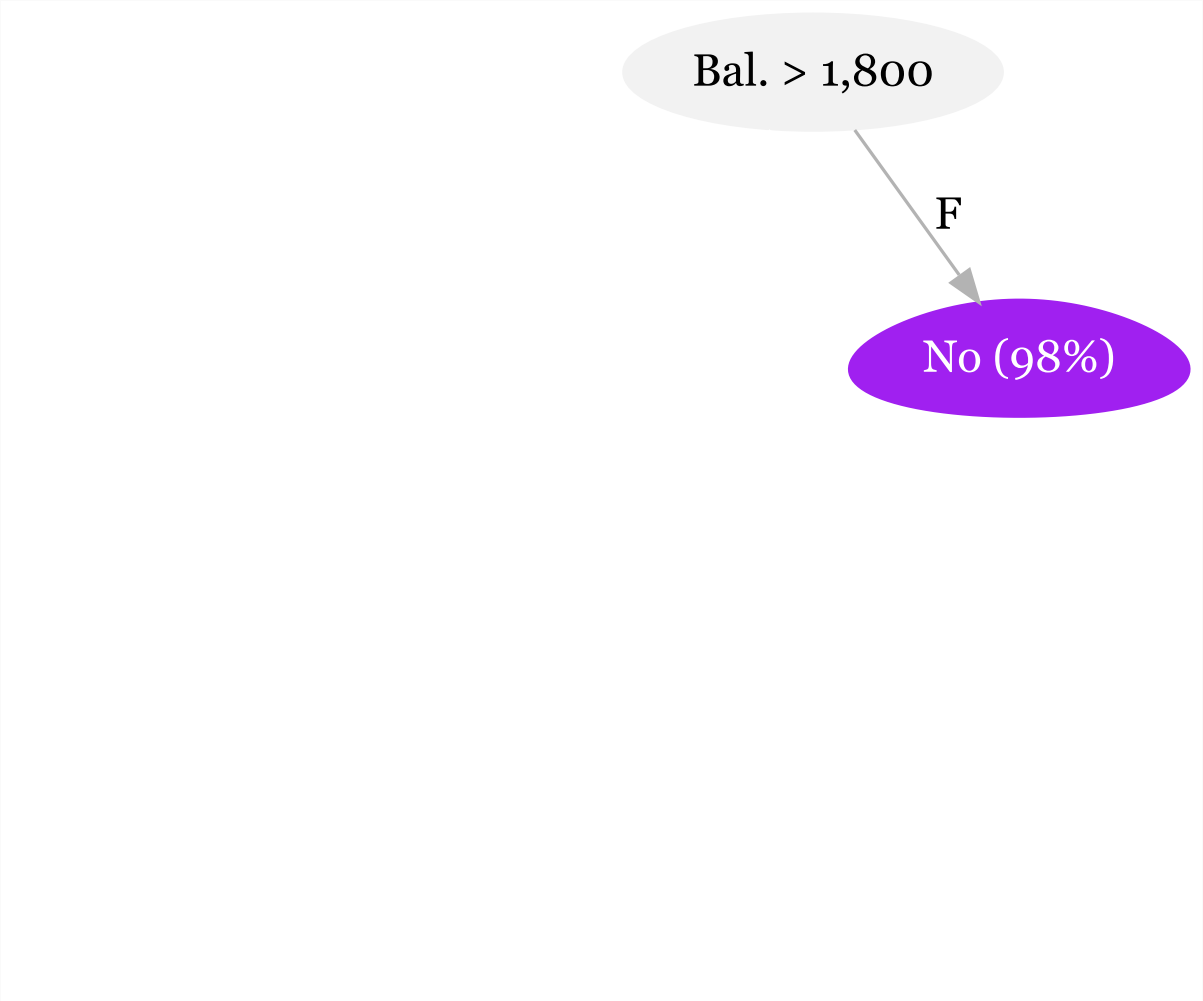
Visualizing the Decision Tree

Our first split occurs at a Balance of $1800:



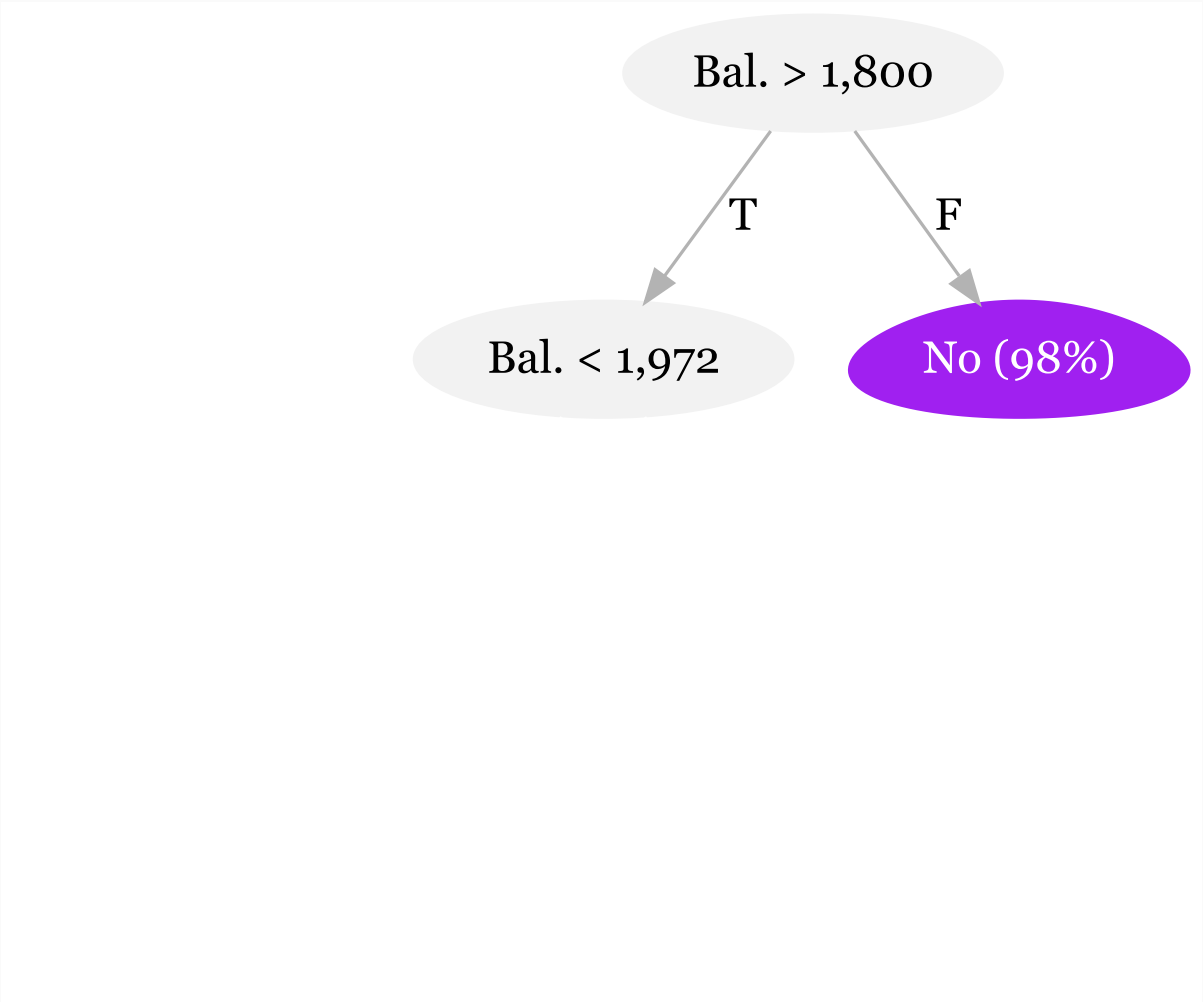
Visualizing the Decision Tree

If Balance > 1800 Is False, we get our first region:



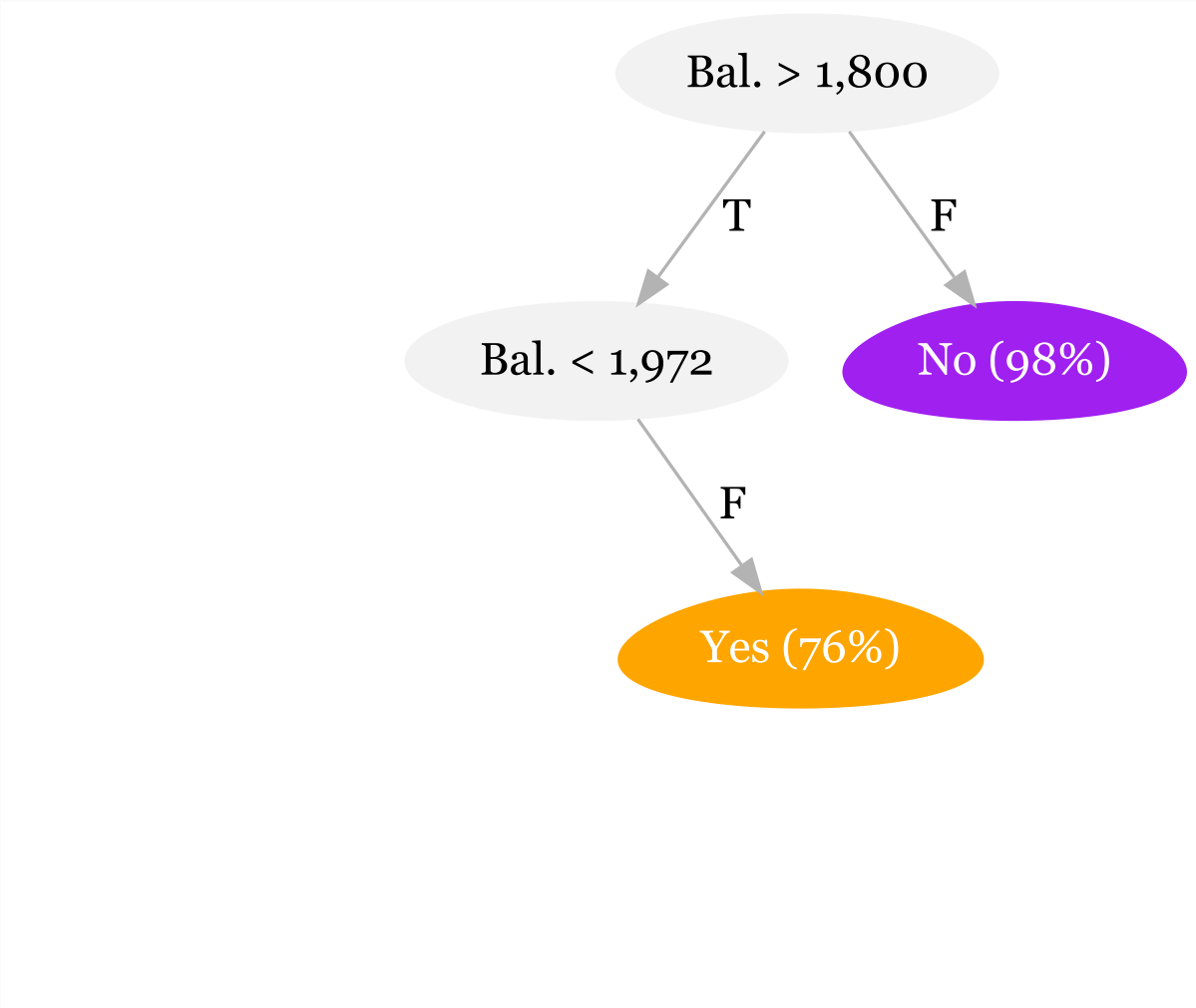
Visualizing the Decision Tree

If Balance < 1800 Is True, we next split on Balance < 1,972



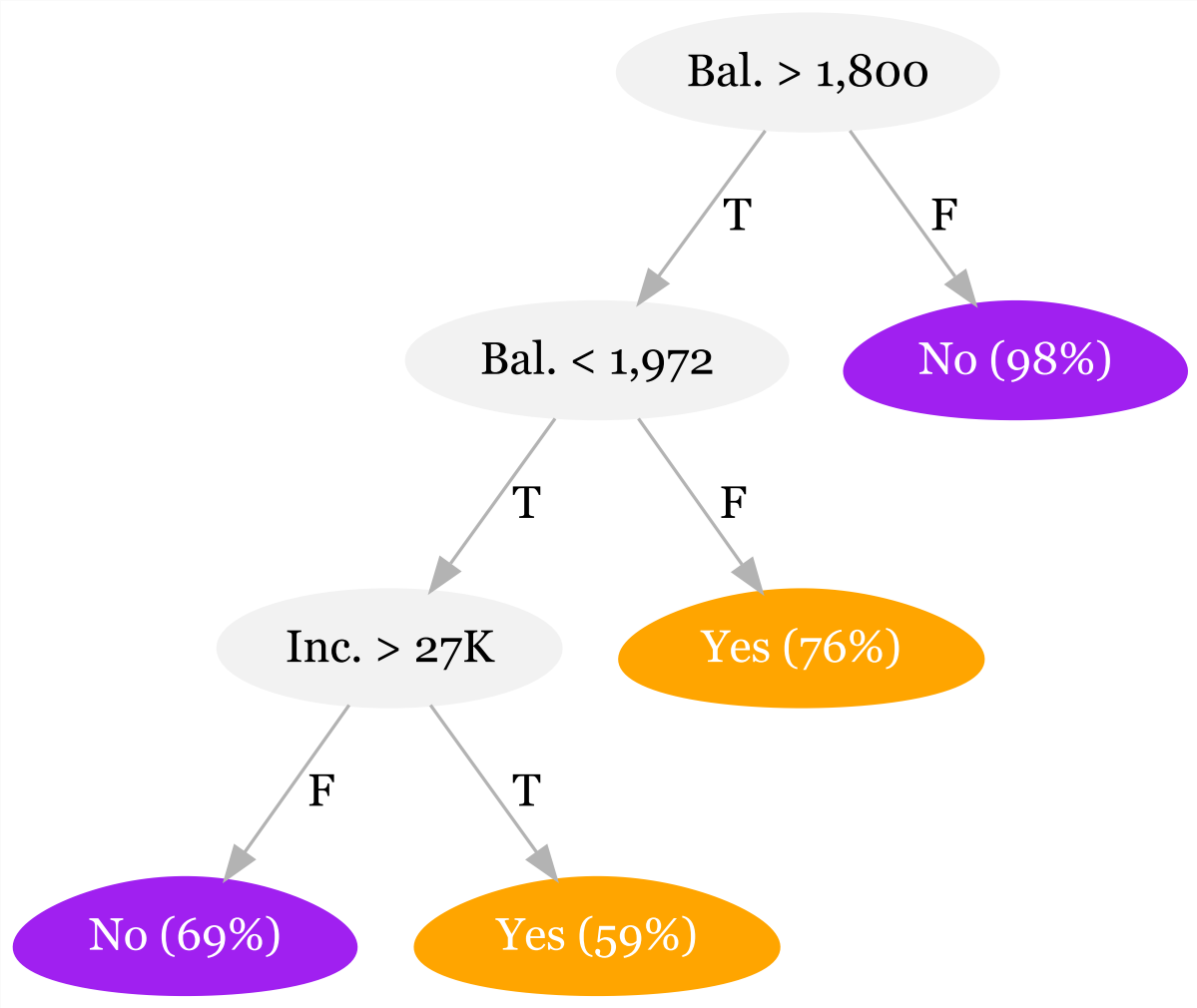
Visualizing the Decision Tree

If False, we get our second terminal node (76% default)



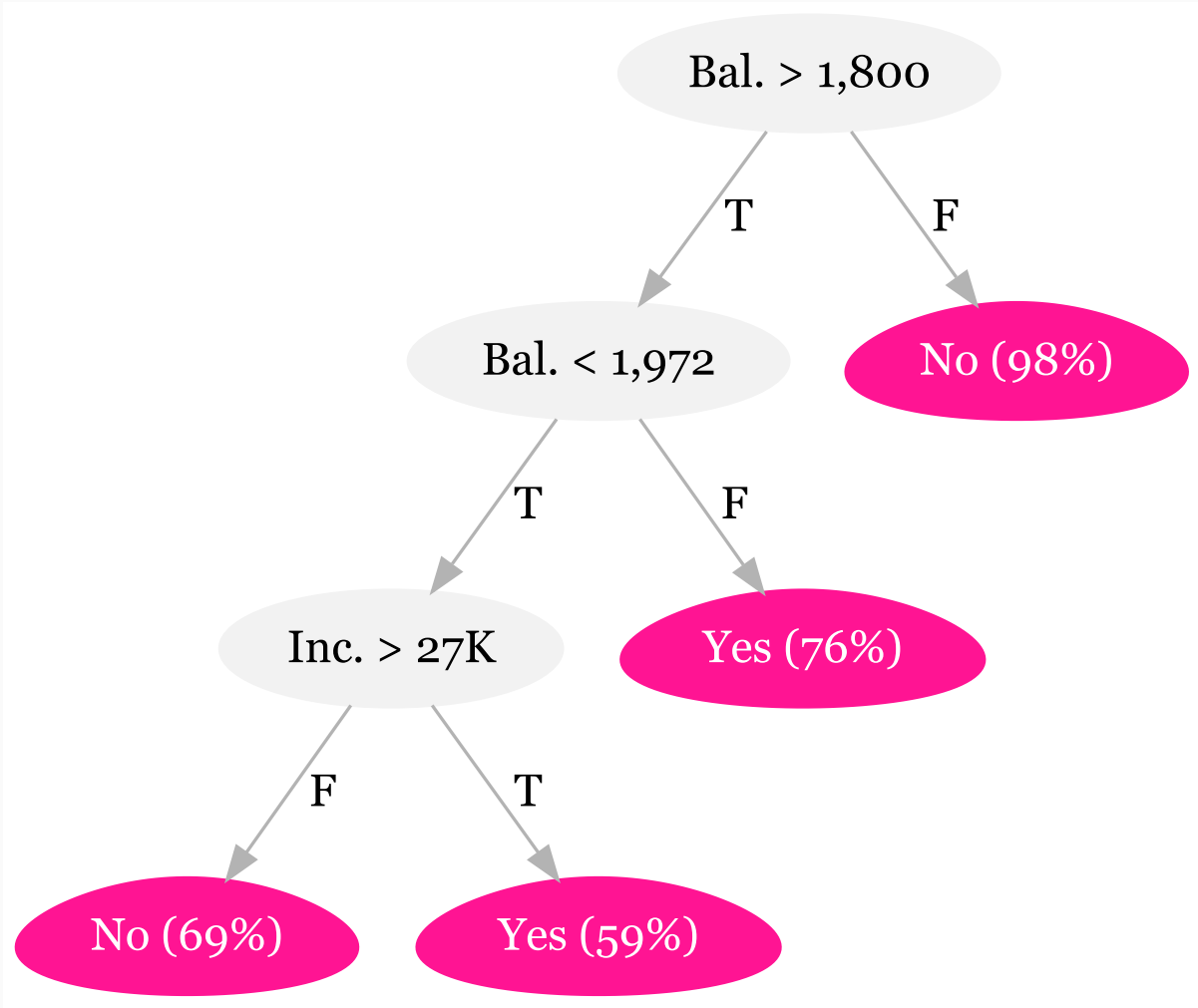
Visualizing the Decision Tree

If True, we get our final split at Income > 27,000



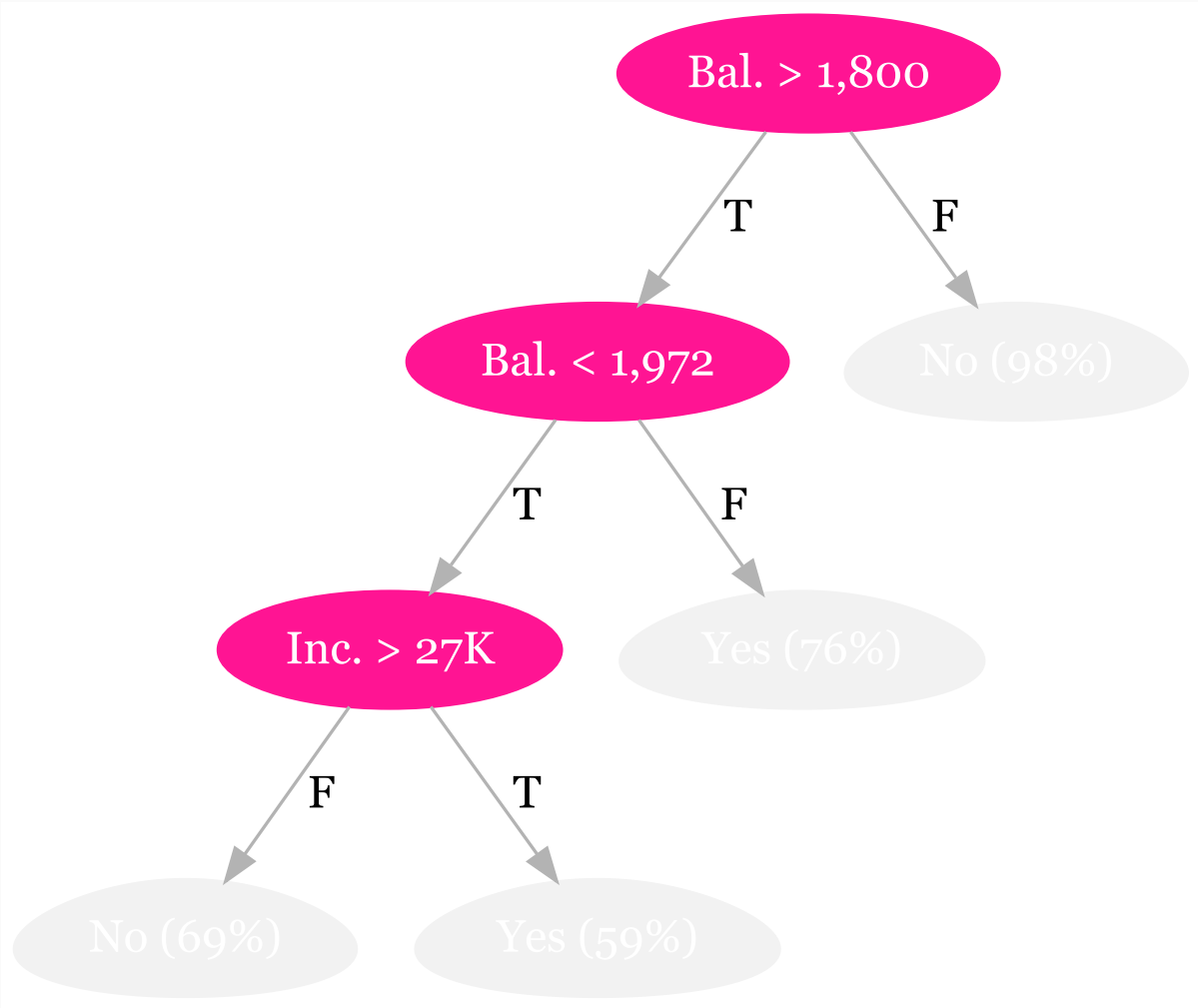
Decision Tree Anatomy

The **regions** correspond to the tree's **terminal nodes** (or **leaves**)



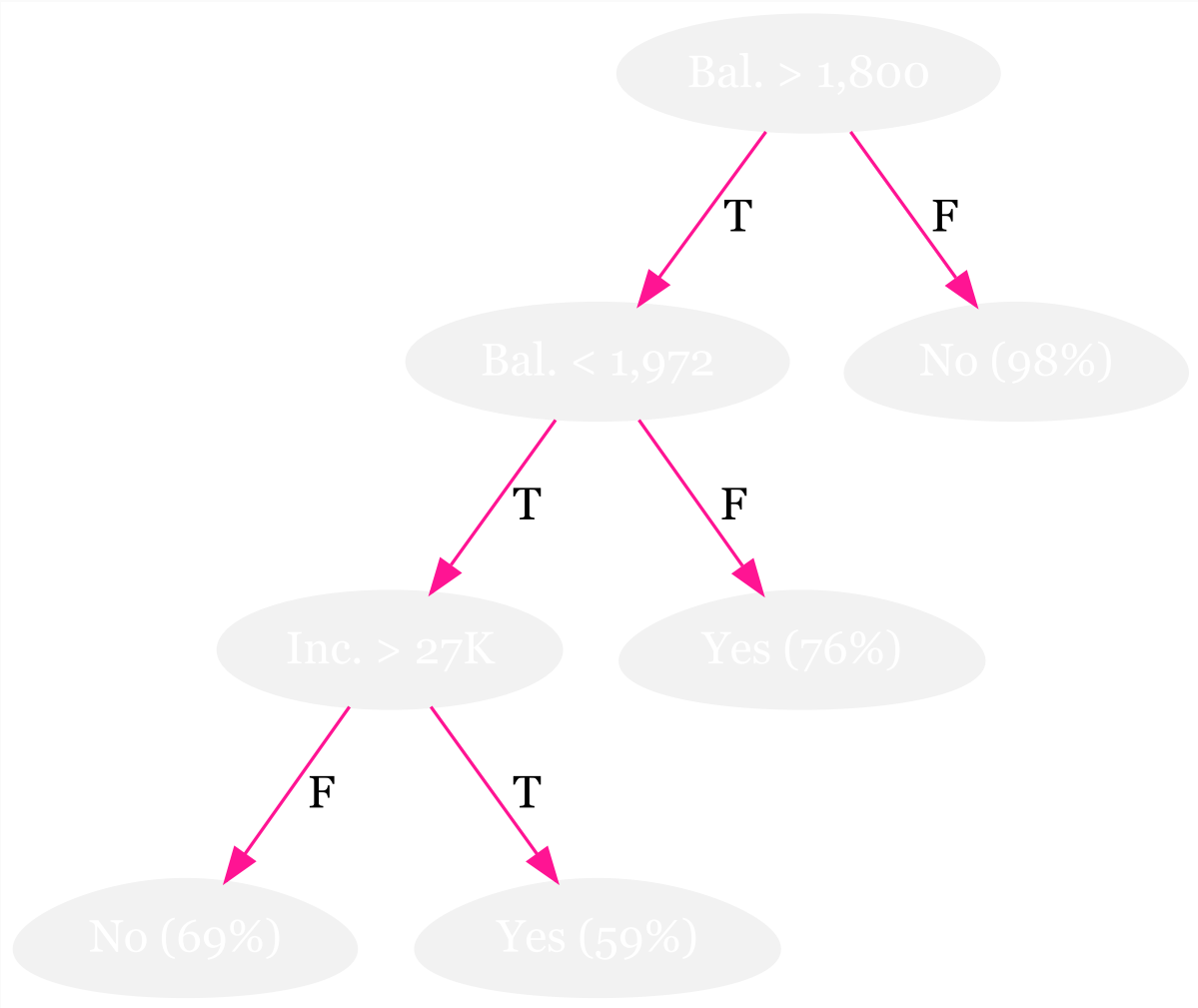
Decision Tree Anatomy

The graph's **separating lines** correspond to the tree's **internal nodes**



Decision Tree Anatomy

The segments connecting the nodes are the tree's **branches**



Growing The Tree

Problem: Examining every possible partition (every cutoff s of every predictor $\mathbf{x}_j$) is **computationally infeasible.**

Solution: a top-down, *greedy* algorithm named **recursive binary splitting**

- **Recursive:** starts with the "best" split, then find the next "best" split, etc.
- **Binary:** each split creates only two branches: "yes" and "no"
- **Greedy:** each step makes the "best" split without consideration for the overall process

Splitting Example

Consider the dataset

i	y	x1	x2
1	0	1	4
2	8	3	2
3	6	5	6

With just three observations, each variable only has two actual splits.

You can think about cutoffs as the ways we divide observations into two groups.

Splitting Example

One possible split: x_1 at 2, which yields (1) $x_1 < 2$ *vs.* (2) $x_1 \geq 2$

i	y	x1	x2
1	0	1	4
2	8	3	2
3	6	5	6

Splitting Example

One possible split: x_1 at 2, which yields (1) $x_1 < 2$ *vs.* (2) $x_1 \geq 2$

i	pred	y	x1	x2
1	**0**	0	1	4
2	**7**	8	3	2
3	**7**	6	5	6

This split yields an RSS of $0^2 + 1^2 + (-1)^2 = 2$.

Note$_1$ Splitting x_1 at 2 yields the same results as 1.5, 2.5—anything in (1, 3).

Splitting Example

An alternative split: x_1 at 4, which yields (1) $x_1 < 4$ *vs.* (2) $x_1 \geq 4$

i	pred	y	x1	x2
1	**4**	0	1	4
2	**4**	8	3	2
3	**6**	6	5	6

This split yields an RSS of $(-4)^2 + 4^2 + 0^2 = 32$.

Previous: Splitting x_1 at 4 yielded RSS = 2. *(Much better)*

Splitting Example

Another split: x_2 at 3, which yields (1) $x_1 < 3$ *vs.* (2) $x_1 \geq 3$

i	pred	y	x1	x2
1	**3**	0	1	4
2	**8**	8	3	2
3	**3**	6	5	6

This split yields an RSS of $(-3)^2 + 0^2 + 3^2 = 18$.

Splitting Example

Final split: x_2 at 5, which yields (1) $x_1 < 5$ *vs.* (2) $x_1 \geq 5$

i	pred	y	x1	x2
1	**4**	0	1	4
2	**4**	8	3	2
3	**6**	6	5	6

This split yields an RSS of $(-4)^2 + 4^2 + 0^2 = 32$.

Splitting Example

Across our four possible splits (two variables each with two splits)

- x_1 with a cutoff of 2: RSS = 2
- x_1 with a cutoff of 4: RSS = 32
- x_2 with a cutoff of 3: RSS = 18
- x_2 with a cutoff of 5: RSS = 32

our split of x_1 at 2 generates the lowest RSS.

Splitting

Note: Categorical predictors work in exactly the same way.
We want to try all possible combinations of the categories.

Ex: For a four-level categorical predictor (levels: A, B, C, D)

- Split 1: A|B|C *vs.* D
- Split 2: A|B|D *vs.* C
- Split 3: A|C|D *vs.* B
- Split 4: B|C|D *vs.* A
- Split 5: A|B *vs.* C|D
- Split 6: A|C *vs.* B|D
- Split 7: A|D *vs.* B|C

we would need to try 7 possible splits.

Splitting

Once we make our a split, we then continue splitting, conditional on the regions from our previous splits.

So if our first split creates R_1 and R_2, then our next split searches the predictor space only in R_1 or R_2.

The tree continue to grow until it hits some specified threshold, *e.g.*, at most 5 observations in each leaf.

We are no longer searching the full space—it is conditional on the previous splits.

Too many Splits?

One can have **too many splits.**

Q: Why?

A: "More splits" means

1. more flexibility (think about the bias-variance tradeoff/overfitting)
2. less interpretability (one of the selling points for trees)

Q: So what can we do?

A: Prune your trees!

Pruning

The idea: Some regions may increase **variance** more than they reduce **bias**.

- By removing these regions, we gain in test MSE.

Candidates for trimming: Regions that do not reduce RSS very much.

Updated strategy: Grow big trees T_0 and then trim T_0 to an optimal subtree.

Updated problem: Considering all possible subtrees can get expensive.

Pruning

Updated solution: add a penalty for complexity with **cost-complexity pruning**

- Pay a penalty to become more complex with a greater number of regions $|T|$

For regression trees[1]:

$$\sum_{m=1}^{|T|} \sum_{i:x \in R_m} \left(y_i - \hat{y}_{R_m}\right)^2 + \alpha |T|$$

For any value of $\alpha (\geq 0)$, we get a subtree $T \subset T_0$.

We choose α via cross validation.

1 For classification trees we instead penalize the error rate

Decision Trees in R

To train decision trees in R, we'll use **tidymodels** (and several others) for modeling and machine learning using **tidyverse** principles[2]

Let's first set the seed and define our cross-validation with `vfold_cv` from **rsample**

- `v` the number of folds (here we'll use 5)

```
# Define our CV split
set.seed(12345)
default_cv ← default_df %>% vfold_cv(v = 5)
```

2. **tidymodels** allow you to do a *ton* of stuff that we're just scratching the surface of.

Decision Trees in R

Next, let's use the `decision_tree()` function from **parsnip** to set up the parameters of the tree:

- `mode`: `"regression"` or `"classification"`
- `cost_complexity`: the cost (penalty) paid for complexity
- `tree_depth`: *max.* tree depth (max. number of splits in a "branch")
- `min_n`: *min.* # of observations for a node to split
- Use the **rpart** "engine" to grow the tree

```
# Define the decision tree
default_tree ← decision_tree(
  mode = "classification", #
  cost_complexity = tune(),
  tree_depth = tune(),
  min_n = 10 # Arbitrarily choosing '10'
) %>% set_engine("rpart")
```

Decision Trees in R

Next, use `recipe()` to build a recipe describing the formula/data

```
# Define recipe
default_recipe ← recipe(default ~ ., data = default_df)
```

And define the workflow, first adding the model then the recipe

```
# Define the workflow
default_flow ← workflow() %>%
  add_model(default_tree) %>% add_recipe(default_recipe)
```

Decision Trees in R

Lastly, execute the workflow through cross-validation!

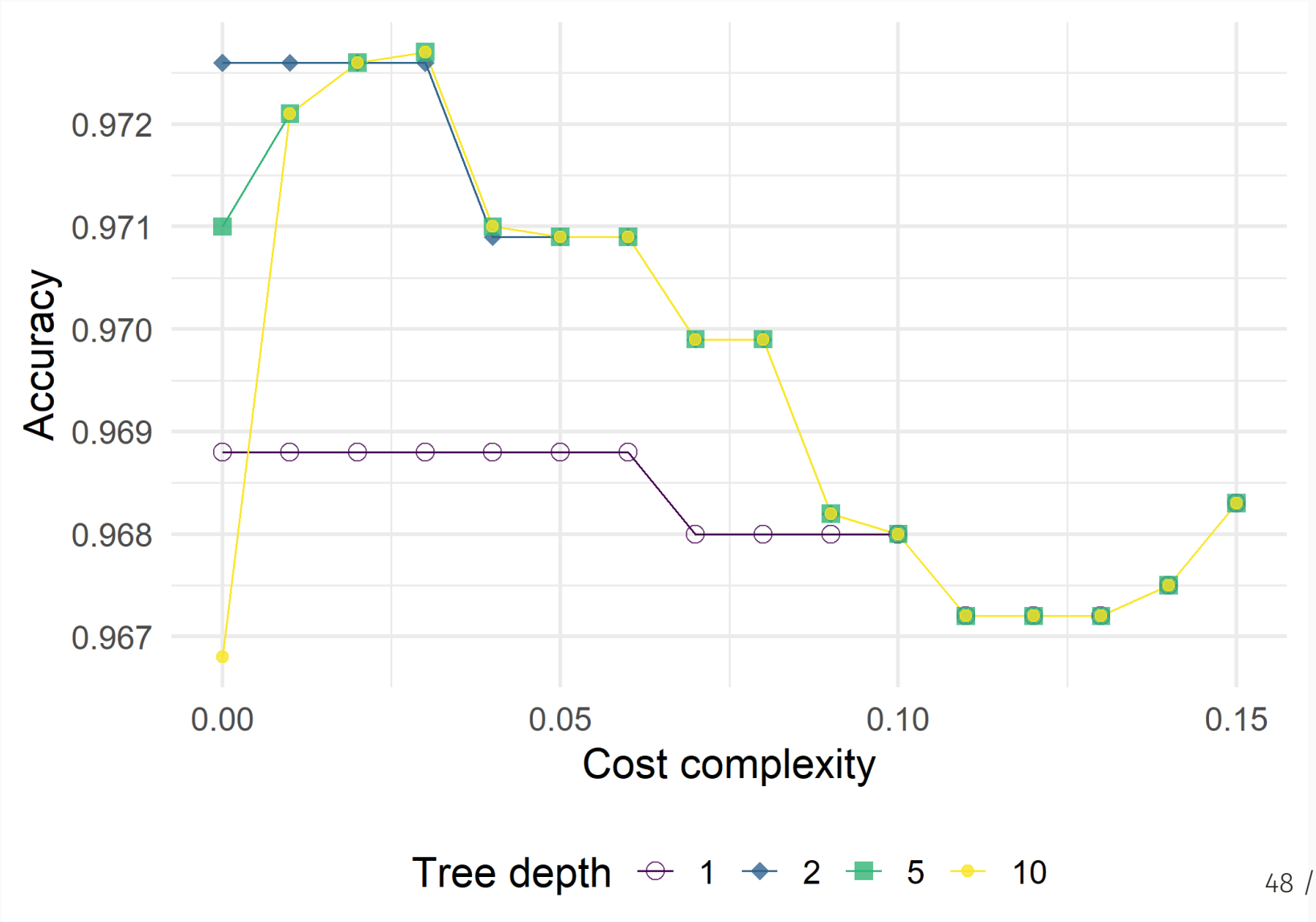
```
# Tune!
default_cv_fit ← default_flow %>% tune_grid(
  default_cv,
  grid = expand_grid(
    cost_complexity = seq(0, 0.15, by = 0.01),
    tree_depth = c(1, 2, 5, 10),
  ),
  metrics = metric_set(accuracy)
)
```

Decision Trees in R

Plotting

```
ggplot(
  data = default_cv_fit %>% collect_metrics() %>% filter(.metric == "accu
  aes(
    x = cost_complexity,
    y = mean,
    color = tree_depth %>% factor(levels = c(1,2,5,10), ordered = T),
    shape = tree_depth %>% factor(levels = c(1,2,5,10), ordered = T)
  )
) +
geom_line(linewidth = 0.4) +
geom_point(size = 3, alpha = 0.8) +
scale_y_continuous("Accuracy") +
scale_x_continuous("Cost complexity") +
scale_color_viridis_d("Tree depth") +
scale_shape_manual("Tree depth", values = c(1, 18, 15, 20)) +
theme_minimal(base_size = 16) +
theme(legend.position = "bottom")
```

Decision Trees in R



Decision Trees in R

To plot the CV-chosen tree...

1. **Fit** the chosen/best model with `finalize_workflow()` and `select_best()`

```
best_flow ←
  default_flow %>%
  finalize_workflow(select_best(default_cv_fit, metric = "accuracy")) %>%
  fit(data = default_df)
```

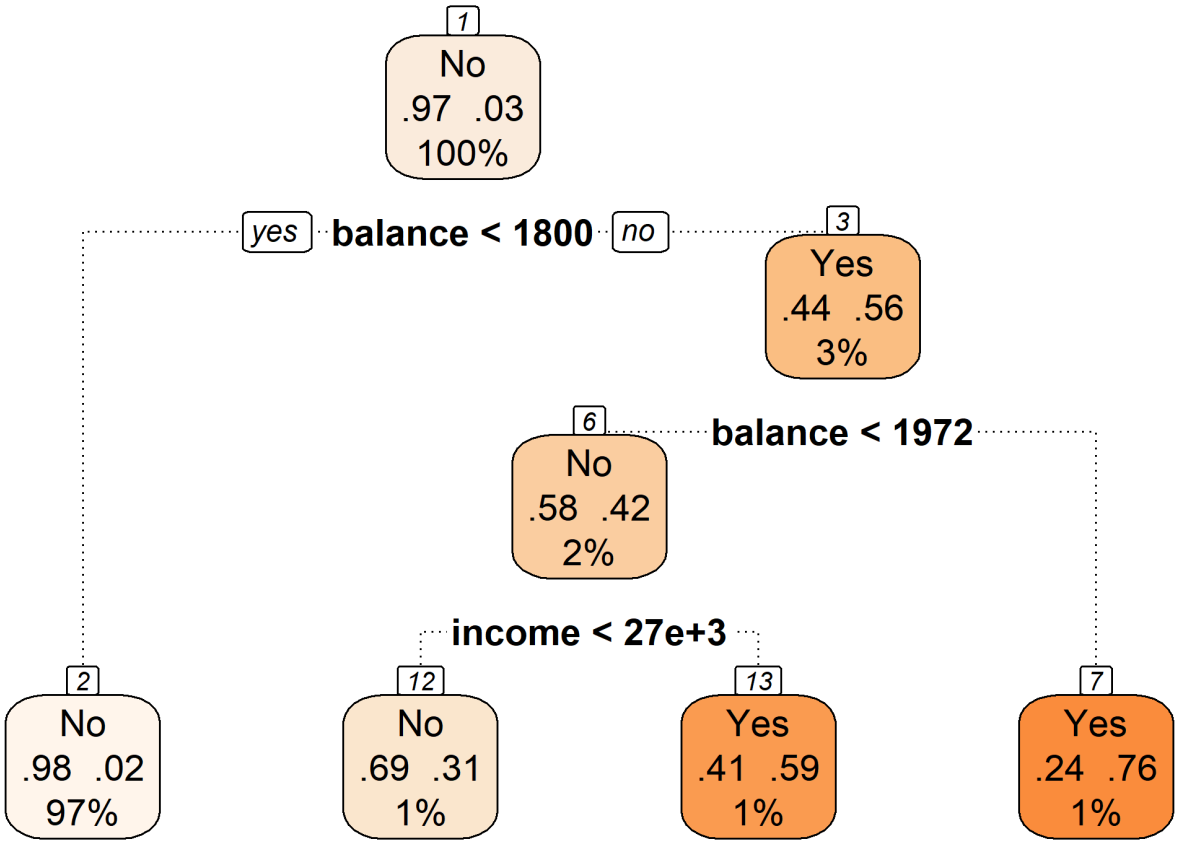
2. **Extract** the fitted model with `extract_fit_parsnip()`

```
best_tree ← best_flow %>% extract_fit_parsnip()
```

Decision Trees in R

3. **Plot** the tree with `rpart.plot()` from **rpart.plot**.

```
best_tree$fit %>% rpart.plot()
```



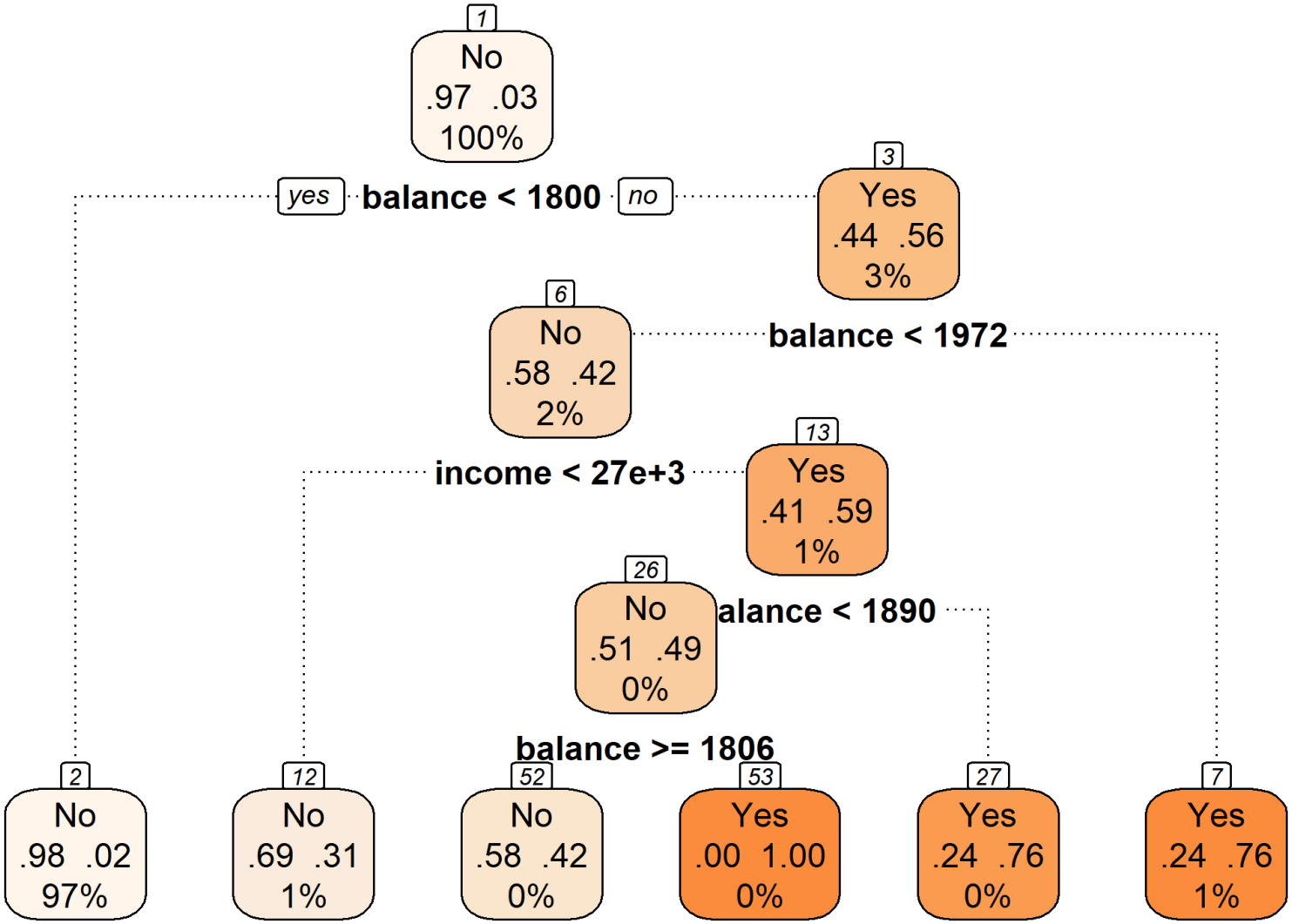
Decision Trees in R

The previous tree has cost complexity of 0.03 (and a max. depth of 5).

We can compare this "best" tree to a less pruned/penalized tree

- `cost_complexity = 0.005`
- `tree_depth = 5`

Decision Trees in R



Decision Trees vs. Linear Models

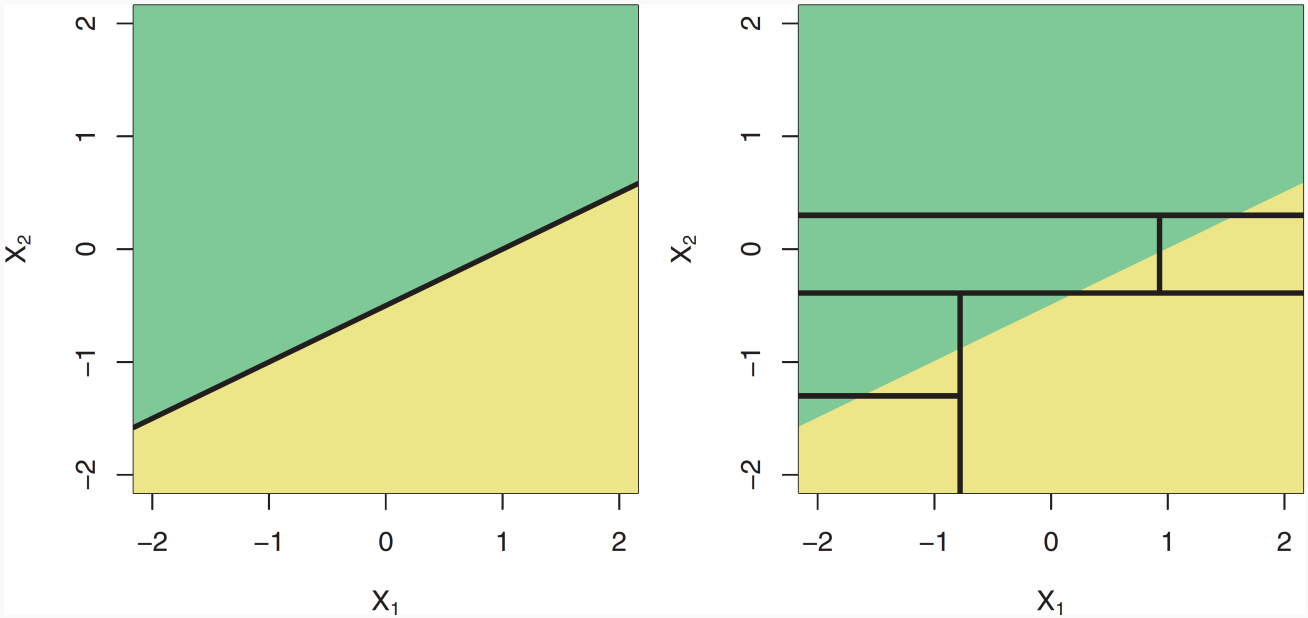
Q How do trees compare to linear models?

A It depends how linear the true boundary is.

Linear Boundary

Linear models recreate the line

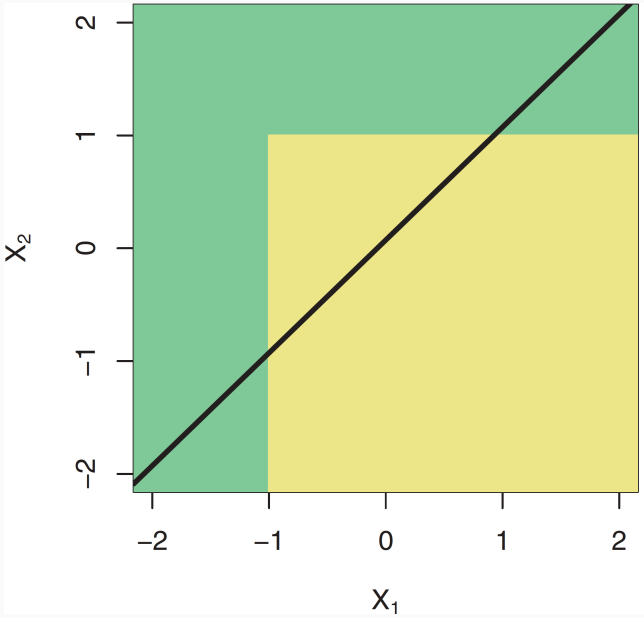
trees struggle at replicating linearity



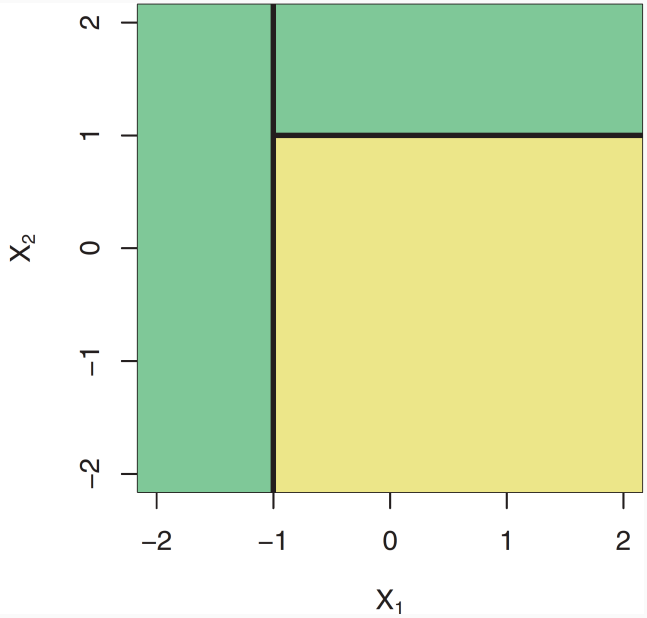
Source: ISL, p. 315

Nonlinear Bounday

Linear models struggle with nonlinear boundaries



trees easily replicate these relationships



Source: ISL, p. 315

Strengths and weaknesses

As with any method, decision trees have **tradeoffs**.

Strengths	Weaknesses
+ Easily explained/interpretted	- Outperformed by other methods
+ Include several graphical options	- Struggle with linearity
+ Mirror human decision making?	- Can be very "non-robust" - small data changes can cause huge changes in our tree
+ Handle num. or cat. on LHS/RHS	

Next: Create ensembles of trees to strengthen these weaknesses. with...

Forests! Which will also weaken some of the strengths.


Random Forests

Random Forests

The gist: combine many individual trees into a single **forest** via **bootstrap aggregation (bagging)**

Bagging:

1. Create B bootstrapped samples
2. Train an estimator (tree) $\hat{f}^b(x)$ on each of the B samples
3. Aggregate across your B bootstrapped models:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x)$$

This aggregated model $\hat{f}_{\text{bag}}(x)$ is your final model.

Bagging

When we apply bagging to decision trees,

- We typically **grow the trees deep and do not prune**
- For **regression**, we **average** across the B trees' regions
- For **classification**, we have more options—but often take **plurality**

Individual (unpruned) trees will be very **flexible** and **noisy**,
but their **aggregate** will be quite **stable**.

Out-of-Bag Error

Bagging also offers a convenient method for evaluating performance.

For any bootstrapped sample, we omit ~n/3 observations.

Out-of-bag (OOB) error estimation estimates the test error rate using observations **randomly omitted** from each bootstrapped sample.

For each observation i:

1. Find all samples S_i in which i was omitted from training.
2. Aggregate the $|S_i|$ predictions $\hat{f}^b(x_i)$, *e.g.*, using their mean or mode
3. Calculate the error, *e.g.*, $y_i - \hat{f}_{i,\text{OOB},i}(x_i)$

Out-of-Bag Error

When B is big enough, the OOB error rate will be very close to LOOCV.

Q: Why use OOB error rate?

A: When B and n are large, cross validation—with any number of folds—can become pretty computationally intensive.

Random Forests

Random forests overcome a major shortcoming of bagging: **very correlated trees**

- Since trees consider all predictors at every split of every tree, the trees will often be highly related.

Solution: decorrelate the trees!

- Only consider a **random subset** of m $(\approx \sqrt{p})$ predictors when making each split (for each tree)
- This nudges trees away from always using the same variables, increasing variation across trees, and potentially reducing the variance of our estimates
- If our predictors are very correlated, we may want to shrink m

Random Forests

Random forests thus introduce **two dimensions of random variation**

1. The **bootstrapped sample**
2. The m **randomly selected predictors** (for the split)

Everything else about random forests works just as it did with decision trees.

Random Forests in R

Let's see how to train a random forest using **tidymodels**.

Let's try and predict heart disease occurrence as a function of

- `age` Patient age
- `sex` Gender (`sex=1` for male)
- `chest_pain` Type of chest pain
- `rest_bp` Resting blood pressure
- `chol` Serum cholesterol level
- `fbs` Fasting blood sugar above 120mg/dl
- `restecg` Resting ECG results
  - 0 normal, 1 have ST-T wave abnormality, 2 probable/definite left ventricular hypertrophy
- `max_hr` Max heart rate
- `ex_ang` Exercised induced angina (pain from reduced blood flow)
  - 1 yes
- `oldpeak` ST depression induced by exercise relative to rest (observed in ECG)
- `slope` Slope of the peak exercise ST segment (observed in ECG)
  - 0 upsloping, 1 flat, 2 downsloping
- `ca` Number of major vessels colored by fluoroscopy (`ca`)
- `thal` Thalium stress test result
  - 0 normal, 1 fixed defect, 2 reversible defect, 3 not described

Random Forests in R

Let's first define our recipe and do some preprocessing.

- **Recipe:** predict heart disease (Yes/No) as a function of everything else
- **Cleaning:** imput missing values (median for numeric, mode for categorical)

```
# Impute missing values
heart_recipe ← recipe(heart_disease ~ ., data = heart_df) %>%
  step_impute_median(all_predictors() & all_numeric()) %>%
  step_impute_mode(all_predictors() & all_nominal())
```

Now we can use `prep()` and `juice()` to replace our data frame with the cleaned version:

```
heart_df_clean ← heart_recipe %>% prep() %>% juice()
```

Random Forests in R

You have several **options** for training random forests with `tidymodels`.

- *E.g.* `ranger`, `randomForest`, `spark`.

`rand_forest()` accesses each of these packages via their *engines*.

- The default engine is `"ranger"` (**`ranger`** package).
- The argument `mtry` gives m, the # of predictors at each split.

You've already seen the other hyperparameters for `ranger`:

- `trees` the number of trees in your (random) forest
- `min_n` min. # of observations

And even more if you look outside of `tidymodels`.

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification
- Try 3 variables per split

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification
- Try 3 variables per split
- 100 trees in the forest

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification
- Try 3 variables per split
- 100 trees in the forest
- At least 2 obs. to split

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification
- Try 3 variables per split
- 100 trees in the forest
- At least 2 obs. to split
- Choose the `ranger` engine

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"

)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification
- Try 3 variables per split
- 100 trees in the forest
- At least 2 obs. to split
- Choose the `ranger` engine
- Set a **splitting rule**

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

Training a random forest in R using **tidymodels** and **ranger**:

- Goal: Classification
- Try 3 variables per split
- 100 trees in the forest
- At least 2 obs. to split
- Choose the `ranger` engine
- Set a **splitting rule**
- Set the method for calculating variable importance

```
# Define the random forest
heart_rf ← rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forests in R

We can then build our workflow and fit the random forest:

Interpreting Random Forests

While ensemble methods like random forests tend to **improve predictive performance**, they also tend to **reduce interpretability**.

We can illustrate **variables' importance** by considering their splits' reductions in the model's **performance metric** (RSS, Gini, entropy, *etc.*).

Note By default, many variable importance functions will scale importance.

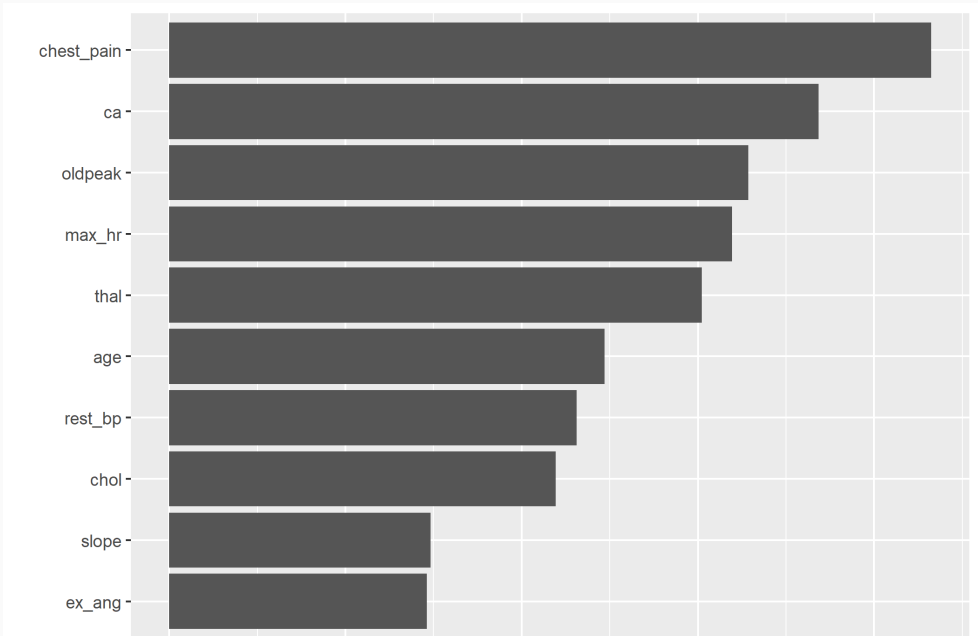
This idea isn't exclusive to forests; it also works for a single tree.

Variable Importance

Plotting the most important variables using `vip()` from **vip**:

```
# extract the fitted forest
extr ← heart_rf_fit %>%
  extract_fit_parsnip()

# plot variable importance
extr %>% vip::vip()
```



Variable Importance

Or extracting manually:

```
var_imp ← data.frame(variable = names(extr$fit$variable.importance),
                     importance = extr$fit$variable.importance %>% as.numeric() #
                     ) %>% arrange(desc(importance))
var_imp
```

variable	importance
chest_pain	21.6
ca	18.4
oldpeak	16.4
max_hr	16
thal	15.1
age	12.4

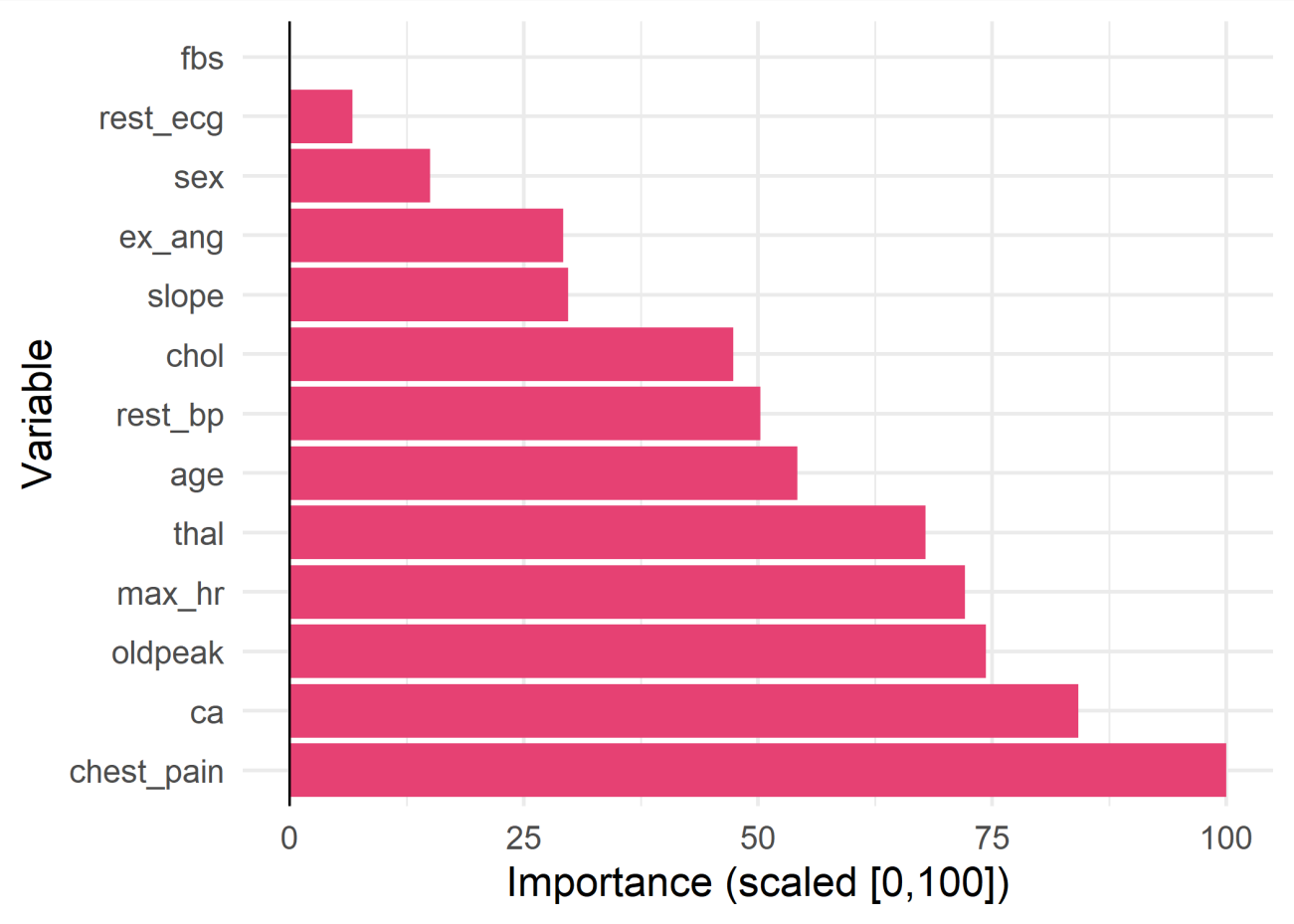
Variable Importance

Standardizing to 0-100 scale:

```
var_imp ← var_imp %>%
  # standardize importance
  mutate(imp_std = importance - min(importance),
         imp_std = 100 * imp_std/ max(imp_std))
```

Variable Importance

Plotting standardized importance:



Random Forest Error

Out-of-bag error is hidden pretty deep in the fitted model:

```
rf_error ← heart_rf_fit$fit$fit$fit$prediction.error
```

Which means we predicted 87% of heart disease cases correctly with our forest!

Random Forest Predictions

Extracting the predictions confirms another **interpretability challenge** of random forests/ensemble methods:

since we **average over all tree predictions**, our predicted classifications represent the **share of trees that assign the observation to class** s.[3]

3. The result of this averaging is more interpretable for regression forests

Random Forest Predictions

Extracting the predictions and viewing them:

```
rf_fit ← heart_rf_fit %>% extract_fit_parsnip()
rf_pred ← rf_fit$fit$predictions %>% as.data.frame()
head(rf_pred)
```

No	Yes
0.6	0.4
0.0303	0.97
0.025	0.975
0.526	0.474
1	0
0.935	0.0645

Random Forest Predictions

Like with the predicted probabilities with the Bayes classifier, we can assign each observation to the **most frequently predicted class**:

```
rf_pred ← rf_pred %>% mutate(heart_disease_hat = case_when(
  Yes > No ~ "Yes", # assign yes if more trees assign the obs to Yes
  TRUE ~ "No" # otherwise assign to No
))
```

Adding the predictions into the cleaned data frame:

```
heart_df_clean ← mutate(heart_df_clean,
                        heart_disease_hat = rf_pred$heart_disease_hat,
                        correct = ifelse(
                          heart_disease == heart_disease_hat, 1, 0))
```

Random Forest Predictions

Seeing how we did at predicting each class:

```
group_by(heart_df_clean, heart_disease) %>%
  summarise(Correct = mean(correct) %>% round(4),
            Count = n())
```

heart_disease	Correct	Count
No	0.86	164
Yes	0.748	139

So we did relatively better at predicting the more frequent "No Heart Disease" cases.[4.]

4. Note that the unconditional average would get us a different value than the model's error rate. That's because the error rate is calculated at the individual tree level while the above error is a summary/average across all the trees' predictions

Random Forest Cross-Validation

Just like with our single tree, we can **tune** our random forest parameters via **cross-validation**.

1. Partition the data into training/test samples

```
set.seed(12345)
split ← initial_split(data = heart_df_clean, prop = 0.5) # split 50/50

train_set ← training(split)
test_set ← testing(split)
```

Random Forest Cross-Validation

2. Setup the cross-validation with `vfold_cv()`

```
# 3-fold cross-validation
heart_cv ← vfold_cv(train_set, v = 3)
```

Random Forest Cross-Validation

3. Set up the random forest for cross-validating parameters

```
# Define the random forest
heart_rf_cv ← rand_forest(
  mode = "classification",
  mtry = tune(),
  trees = 100,
  min_n = tune()
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini",
  importance = "impurity"
)
```

Random Forest Cross-Validation

4. Build a grid of possible hyperparameter values

```
# Define the grid of parameter values
rf_grid ← expand_grid(
  mtry = 1:13,
  min_n = 1:15
)
```

Random Forest Cross-Validation

5. Set the workflow

```
rf_cv_wf ← workflow() %>%
  add_model(heart_rf_cv) %>% # new CV random forest setup
  add_recipe(heart_recipe) # same recipe as before
```

Random Forest Cross-Validation

6. Tune!

```
rf_tune ← rf_cv_wf %>%
  tune_grid(
    resamples = heart_cv, # the CV setup
    grid = rf_grid # the grid of hyperparameter values to try
  )
```

Random Forest Cross-Validation

Showing the 5 best-performing models:

```
show_best(x = rf_tune, metric = "accuracy", n = 5) %>%
   select(mtry:min_n, mean, std_err)
```

mtry	min_n	mean	std_err
2	1	0.847	0.0273
3	5	0.841	0.0352
4	1	0.841	0.0405
3	9	0.834	0.0183
3	6	0.834	0.0248

Random Forests

Random Forests offer a lot of improvement over single decision trees from a performance standpoint.

However, they can often be harder to interpret.

Perhaps more limiting to the types of questions we economists are interested in: it's not always a **conditional mean** that we're interested in.

What if we want to harness the benefits of **forest-based ensemble methods** for identifying **any conditional object of interest?**

Next time...

Table of Contents

Part 3: Tree-Based Methods