Classification Trees and Network-Based Prediction

Machine Learning with Tree-Based Models in R

Prof. Asc. Endri Raco, Ph.D.

Department of Mathematical Engineering
Polytechnic University of Tirana

November 2025

Machine Learning with Tree-Based Models in R

Chapter 1: Classification trees

Chapter 2: Regression trees, cross-validation, bias-variance tradeoff

Chapter 3: Hyperparameter tuning, bagging, random forests

Chapter 4: Boosted trees

Decision trees work like flowcharts that help classify data by asking a series of questions.

Example: **Classifying Animals**

- Can live in trees? $\rightarrow$ Yes/No
- Has scales? $\rightarrow$ Yes/No
- Has feathers? $\rightarrow$ Yes/No

Each question splits the data until we reach a final classification.

{Concept from: Australian Academy of Science}

- Easy to explain and understand
- Possible to capture non-linear relationships
- Require no normalization or standardization of numeric features
- No need to create dummy indicator variables
- Robust to outliers
- Fast for large datasets

- Hard to interpret if large, deep, or ensembled
- High variance, complex trees are prone to overfitting

Key packages in tidymodels:

- **rsample:** Data splitting and resampling
- **recipes:** Data preprocessing
- **parsnip:** Model specifications
- **workflows:** Modeling workflows
- **tune:** Hyperparameter tuning
- **yardstick:** Model performance metrics
- **broom:** Model tidying
- **dials:** Hyperparameter tuning dials

```r
library(tidymodels)
```

```
-- Attaching packages ------------ tidymodels 0.1.4 --
v parsnip   0.2.1      v rsample  0.1.1
v dplyr     1.0.9      v tibble   3.1.7
v yardstick 0.0.9      v tune     0.1.6
```

Step 1: Pick a Model Class

```
library(tidymodels)

decision_tree()
```

```
Decision Tree Model Specification (unknown)
```

Step 2: Set the Engine

```
library(tidymodels)

decision_tree() %>%
  set_engine("rpart")
```

```
Decision Tree Model Specification (unknown)

Computational engine: rpart
```

Step 3: Set the Mode

```
library(tidymodels)

decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification")
```

```
Decision Tree Model Specification (classification)

Computational engine: rpart
```

Specification is a skeleton and needs data to be trained with

```
library(tidymodels)

tree_spec <- decision_tree() %>%
             set_engine("rpart") %>%
             set_mode("classification")
```

```r
# A model specification is fit using a formula
# to training data
tree_spec %>%
  fit(formula = outcome ~ age + bmi,
      data = diabetes)
```

```
parsnip model object
Fit time: 19 ms
n = 652
```

Let's build a model!

How to grow your tree

```r
head(diabetes)
```

```
# A tibble: 6 x 9
  outcome pregnancies glucose blood_pressure
    <fct>       <int>   <int>          <int>
1 yes             6     148             72
2 no              1      85             66
3 yes             8     183             64

  skin_thickness insulin   bmi   age
           <int>   <int> <dbl> <int>
1             35       0  33.6    50
2             29       0  26.6    31
3              0       0  23.3    32
```

Problem: Used all data for training - no data left to test

All Data $\rightarrow$ Decision Tree $\rightarrow$ Performance Check?

If we use all our data for training, we have nothing left to evaluate how well the model performs on new, unseen data.

Solution: Split into training and test sets

Data $\rightarrow$ **Training Set** (build model) + **Test Set** (evaluate)

Decision Tree (trained on training set) $\rightarrow$ **Performance Check** (on test set)

Three common approaches:

1. **Sequential split:** First 75% train, last 25% test
2. **Random block split:** Random contiguous blocks
3. **Random split:** Random observations throughout

Recommendation: Use random split with stratification for classification

Splits data randomly into single training and single test set

```
# Split data proportionally (default: 0.75)
diabetes_split <- initial_split(diabetes,
                                prop = 0.9)
diabetes_split
```

```
<Analysis/Assess/Total>
<692/76/768>
```

{from the rsample package}

Extract training and test sets from a data split

```
diabetes_train <- training(diabetes_split)
diabetes_test  <- testing(diabetes_split)
```

```r
# Verify the split proportion
nrow(diabetes_train) / nrow(diabetes)
```

```
[1] 0.9007812
```

{from rsample}

```r
# Training count of 'yes' and 'no' outcomes
counts_train <- table(diabetes_train$outcome)
counts_train
```

```
no yes
490 86
```

```r
# Training proportion of 'yes' outcome
prop_yes_train <- counts_train["yes"] /
                  sum(counts_train)
prop_yes_train
```

```
0.15
```

```r
# Test data count of 'yes' and 'no' outcomes
counts_test <- table(diabetes_test$outcome)
counts_test
```

```
no yes
28  48
```

```r
# Test data proportion of 'yes' outcome
prop_yes_test <- counts_test["yes"] /
                 sum(counts_test)
prop_yes_test
```

```
0.63
```

Problem: Very different proportions in train (15%) vs test (63%)!

```
initial_split(diabetes,
              prop = 0.9,
              strata = outcome)
```

Ensures random split with similar distribution of `outcome` variable

Let's split!

Predict and evaluate

General Call:

```
predict(model, new_data, type)
```

Arguments:

1. Trained model
2. Dataset to predict on
3. Prediction type: labels or probabilities

```r
predict(model,
        new_data = test_data,
        type = "class")
```

```
  .pred_class
        <fct>
1          no
2          no
3         yes
4          no
```

```r
predict(model,
        new_data = test_data,
        type = "prob")
```

```
  .pred_no .pred_yes
     <dbl>      <dbl>
1    0.866      0.134
2    0.956      0.044
3    0.672      0.328
4    0.877      0.123
```

Reveals how confused a model is

Structure:

- Rows: Predicted classes
- Columns: Actual (truth) classes
- Diagonal: Correct predictions
- Off-diagonal: Incorrect predictions

True Positives (TP): Prediction is yes, truth is yes

True Negatives (TN): Prediction is no, truth is no

False Positives (FP): Prediction is yes, truth is no

False Negatives (FN): Prediction is no, truth is yes

Step 1: Combine predictions and truth values

```
# Combine predictions and truth values
pred_combined <- predictions %>%
  mutate(true_class = test_data$outcome)

pred_combined
```

```
  .pred_class true_class
        <fct>      <fct>
1          no         no
2          no        yes
3          no         no
4         yes        yes
```

Step 2: Calculate the confusion matrix

```
# Calculate the confusion matrix
conf_mat(data = pred_combined,
         estimate = .pred_class,
         truth = true_class)
```

```
            Truth
Prediction   no  yes
       no   116   31
      yes    12   40
```

$$\text{accuracy} = \frac{\text{n of correct predictions}}{\text{n of total predictions}}$$

Function name: `accuracy()`

Same arguments as `conf_mat()`

- `data`, `estimate`, and `truth`
- Common structure in yardstick

```
accuracy(pred_combined,
         estimate = .pred_class,
         truth = true_class)
```

```
  .metric  .estimate
    <chr>       <dbl>
1 accuracy      0.708
```

This means 70.8% of predictions were correct.

Let's evaluate!

Continuous outcomes

```r
head(chocolate, 5)
```

```
# A tibble: 5 x 6
  final_grade review_date cocoa_percent company_location
        <dbl>       <int>         <dbl> <fct>
1         3.0        2009          0.80 U.K.
2        3.75        2012          0.70 Guatemala
3        2.75        2009          0.75 Colombia
4         3.5        2014          0.74 Zealand
5        3.75        2011          0.72 Australia

  bean_type                broad_bean_origin
  <fct>                    <fct>
  "Criollo, Trinitario" "Madagascar"
  "Trinitario"          "Madagascar"
  "Forastero"           "Colombia"
  ""                    "Papua New Guinea"
  ""                    "Bolivia"
```

Step 1: Create model specification

```
spec <- decision_tree() %>%
  set_mode("regression") %>%
  set_engine("rpart")

print(spec)
```

```
Decision Tree Model Specification (regression)

Computational engine: rpart
```

Step 2: Fit the model

```
model <- spec %>%
  fit(formula = final_grade ~ .,
      data = chocolate_train)

print(model)
```

```
parsnip model object
Fit time: 20ms
n = 1437

node), split, n, deviance, yval
  * denotes terminal node
```

```r
# Model predictions on new data
predict(model, new_data = chocolate_test)
```

```
      .pred
      <dbl>
1  3.281915
2  3.435234
3  3.281915
4  3.833931
5  3.281915
6  3.514151
7  3.273864
8  3.514151
```

How decision trees work:

1. Start with all data at the root node
2. Find the best split that minimizes variance
3. Create child nodes based on the split
4. Recursively repeat for each child node
5. Stop when stopping criteria are met

This creates a hierarchical tree structure where each leaf contains observations with similar target values.

Goal for regression trees:

Low variance or deviation from the mean within groups

Design decisions:

- `min_n`: Number of data points in a node needed for further split (default: 20)
- `tree_depth`: Maximum depth of a tree (default: 30)
- `cost_complexity`: Penalty for complexity (default: 0.01)

Set them in the very first step:

```
decision_tree(tree_depth = 4,
              cost_complexity = 0.05) %>%
  set_mode("regression")
```

These parameters control how the tree grows and help prevent overfitting.

Model with tree_depth = 1

```
decision_tree(tree_depth = 1) %>%
  set_mode("regression") %>%
  set_engine("rpart") %>%
  fit(formula = final_grade ~ .,
      data = chocolate_train)
```

```
parsnip model object
Fit time: 1ms
n = 1000

node), split, n, yval
1) root                       1000 2.347450
  2) cocoa_percent>=0.905       16 2.171875 *
  3) cocoa_percent<0.905       984 3.190803 *
```

Root node: - n = 1000 samples - mean grade: 2.347450

Split: cocoa_percent >= 0.905?

Leaf node 2: - n = 16 samples - mean grade: 2.171875

Leaf node 3: - n = 984 samples
- mean grade: 3.190803

Let's do regression!

Performance metrics for regression trees

Classification problems:

Accuracy (confusion matrix) - binary correctness

Regression problems:

"Correct" is relative, no binary correctness

$\Rightarrow$ Measure how far predictions are away from truth

Two main metrics:

1. Mean Absolute Error (MAE)
2. Root Mean Square Error (RMSE)

Both measure prediction error, but in different ways.

Visual representation:

- Plot shows true values vs predictions
- Red vertical bars represent errors
- MAE = average length of the red bars

Interpretation:

Average absolute distance between predictions and actual values

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|actual_i - predicted_i|$$

Interpretation:

"Sum of absolute deviations divided by the number of predictions"

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(actual_i - predicted_i)^2$$

Interpretation:

"Mean squared error"

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(actual - predicted)^2}$$

Interpretation:

"Root of the mean squared error"

Key difference from MAE:

Large errors get higher weight due to squaring

```r
# parsnip and yardstick are included in tidymodels
library(tidymodels)

# Make predictions and add to test data
predictions <- predict(model,
                       new_data = chocolate_test) %>%
  bind_cols(chocolate_test)
```

```
# A tibble: 358 x 7
    .pred final_grade review_date cocoa_percent
    <dbl>       <dbl>       <int>          <dbl>
1    2.5         2.75        2013           0.7
2   3.64         3.25        2014           0.8
3    3.3          3.5        2012           0.7
4   3.25          3.5        2011          0.72

  company_location bean_type  broad_bean_origin
  <fct>            <fct>      <fct>
  France           ...        ...
  France           ...        ...
  France           ...        ...
  Fiji             ...        ...
# ... with 354 more rows
```

```r
# Evaluate using mae()
mae(predictions,
    estimate = .pred,
    truth = final_grade)
```

```
# A tibble: 1 x 2
  .metric .estimate
    <chr>      <dbl>
1     mae      0.363
```

On average, predictions are 0.363 points away from true values.

```r
# Evaluate using rmse()
rmse(predictions,
     estimate = .pred,
     truth = final_grade)
```

```
# A tibble: 1 x 2
  .metric .estimate
    <chr>      <dbl>
1    rmse      0.457
```

RMSE is higher than MAE because it penalizes larger errors more.

Let's evaluate!

Cross-validation

Process:

1. Split training data into k equal folds
2. For each fold:
   - Use fold as validation set
   - Use remaining k-1 folds as training set
   - Train model and calculate error
3. Average errors across all k folds

Benefits: More robust performance estimate

Initial setup:

- **Training data:** Rows 1-500
- **Test data:** Rows 501-600
- **Create 5 folds** from training data

Each fold contains approximately 100 observations ($500/5 = 100$)

Fold 1 as test:

- **Test:** Rows 1-100 (Fold 1)
- **Train:** Rows 101-500 (Folds 2-5)

Result: Model 1, MAE 1

Fold 2 as test:

- **Train:** Rows 1-100, 201-500 (Folds 1, 3-5)
- **Test:** Rows 101-200 (Fold 2)

Result: Model 2, MAE 2

Fold 3 as test:

- **Train:** Rows 1-200, 301-500 (Folds 1-2, 4-5)
- **Test:** Rows 201-300 (Fold 3)

Result: Model 3, MAE 3

Fold 4 as test:

- **Train:** Rows 1-300, 401-500 (Folds 1-3, 5)
- **Test:** Rows 301-400 (Fold 4)

Result: Model 4, MAE 4

Fold 5 as test:

- **Train:** Rows 1-400 (Folds 1-4)
- **Test:** Rows 401-500 (Fold 5)

Result: Model 5, MAE 5

Final metric:

$$\text{Cross-validated MAE} = \frac{MAE_1 + MAE_2 + ... + MAE_5}{5}$$

This gives a more robust estimate of model performance than a single train-test split.

After cross-validation:

1. Use CV results to select best hyperparameters
2. Train final model on **all training data**
3. Evaluate on held-out test set

Training data (all 5 folds) $\rightarrow$ **Final Model**

```r
# Random seed for reproducibility
set.seed(100)

# Create 10 folds of the dataset
chocolate_folds <- vfold_cv(chocolate_train,
                            v = 10)
```

```
# 10-fold cross-validation
# A tibble: 10 x 2
   splits             id
1  <split [1293/144]> Fold01
2  <split [1293/144]> Fold02
3  <split [1293/144]> Fold03
4  ...
```

```r
# Fit a model for every fold and calculate MAE and RMSE
fits_cv <- fit_resamples(tree_spec,
                         final_grade ~ .,
                         resamples = chocolate_folds,
                         metrics = metric_set(mae, rmse))
```

```
# Resampling results
# 10-fold cross-validation
# A tibble: 10 x 4
   splits             id      .metrics
   <list>             <chr>   <list>
1  <split [1293/144]> Fold01  <tibble [2 x 4]>
2  <split [1293/144]> Fold02  <tibble [2 x 4]>
3  <split [1293/144]> Fold03  <tibble [2 x 4]>
4  ...
```

Each fold produces two metrics (MAE and RMSE).

```r
# Collect raw errors of all model runs
all_errors <- collect_metrics(fits_cv,
                              summarize = FALSE)
print(all_errors)
```

```
# A tibble: 20 x 3
   id       .metric .estimate
   <chr>    <chr>       <dbl>
1  Fold01   mae         0.362
2  Fold01   rmse        0.442
3  Fold02   mae         0.385
4  Fold02   rmse        0.504
5  ...
```

```r
library(ggplot2)

ggplot(all_errors,
       aes(x = .estimate, fill = .metric)) +
  geom_histogram()
```

This shows the distribution of MAE and RMSE values across all 10 folds, helping identify variability in model performance.

```r
# Collect and summarize errors of all model runs
collect_metrics(fits_cv)
```

```
# A tibble: 2 x 3
  .metric   mean     n
  <chr>    <dbl> <int>
1 mae      0.383    10
2 rmse     0.477    10
```

Interpretation: Average MAE of 0.383 and RMSE of 0.477 across all folds.

Let's cross-validate!

Tuning hyperparameters

Model parameters whose values control model complexity and are set prior to model training

Hyperparameters influence the shape and complexity of trees.

Hyperparameters in parsnip decision trees:

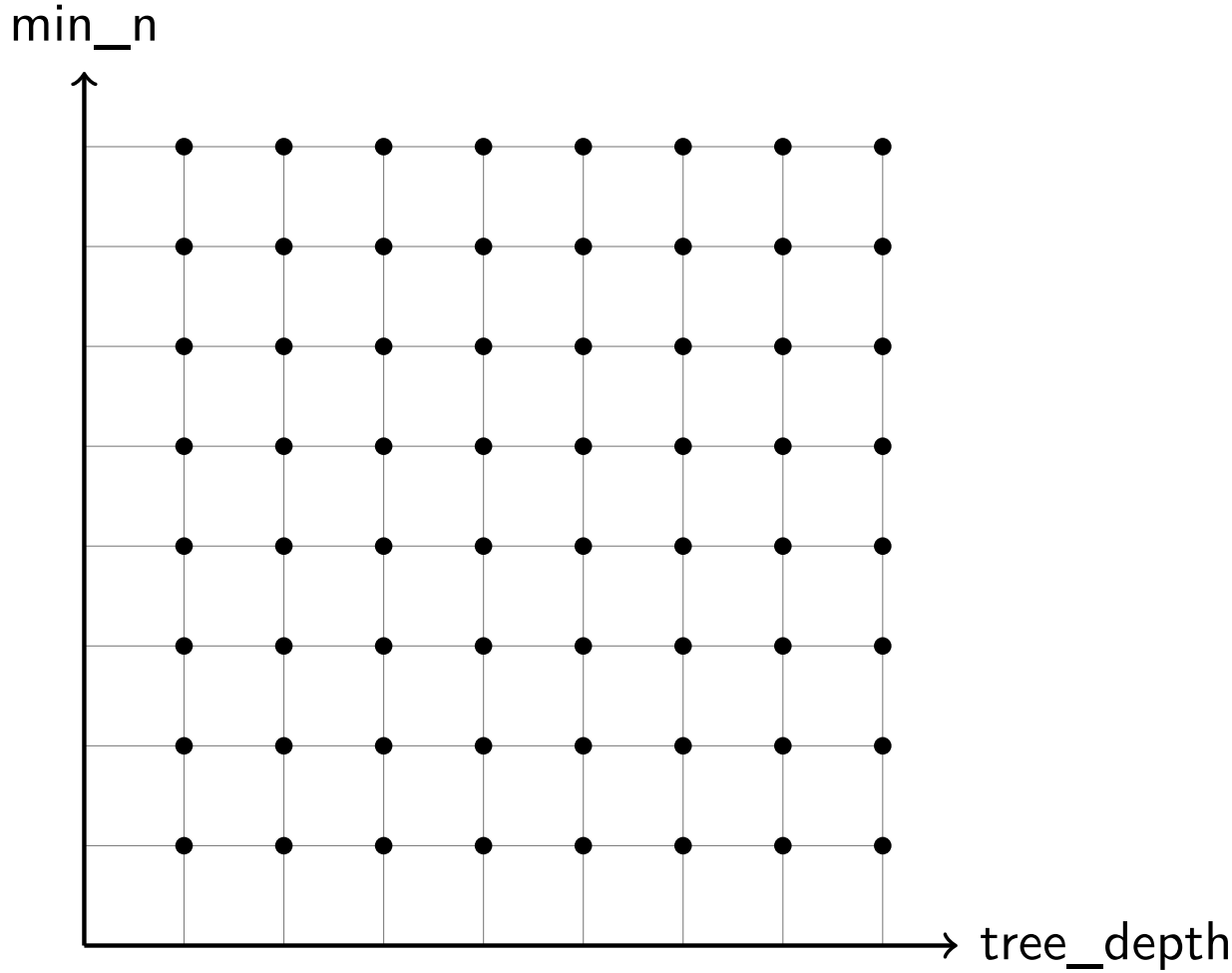
- `min_n`: Minimum number of samples required to split a node
- `tree_depth`: Maximum allowed depth of the tree
- `cost_complexity`: Penalty for tree complexity

Default values set by `parsnip`:

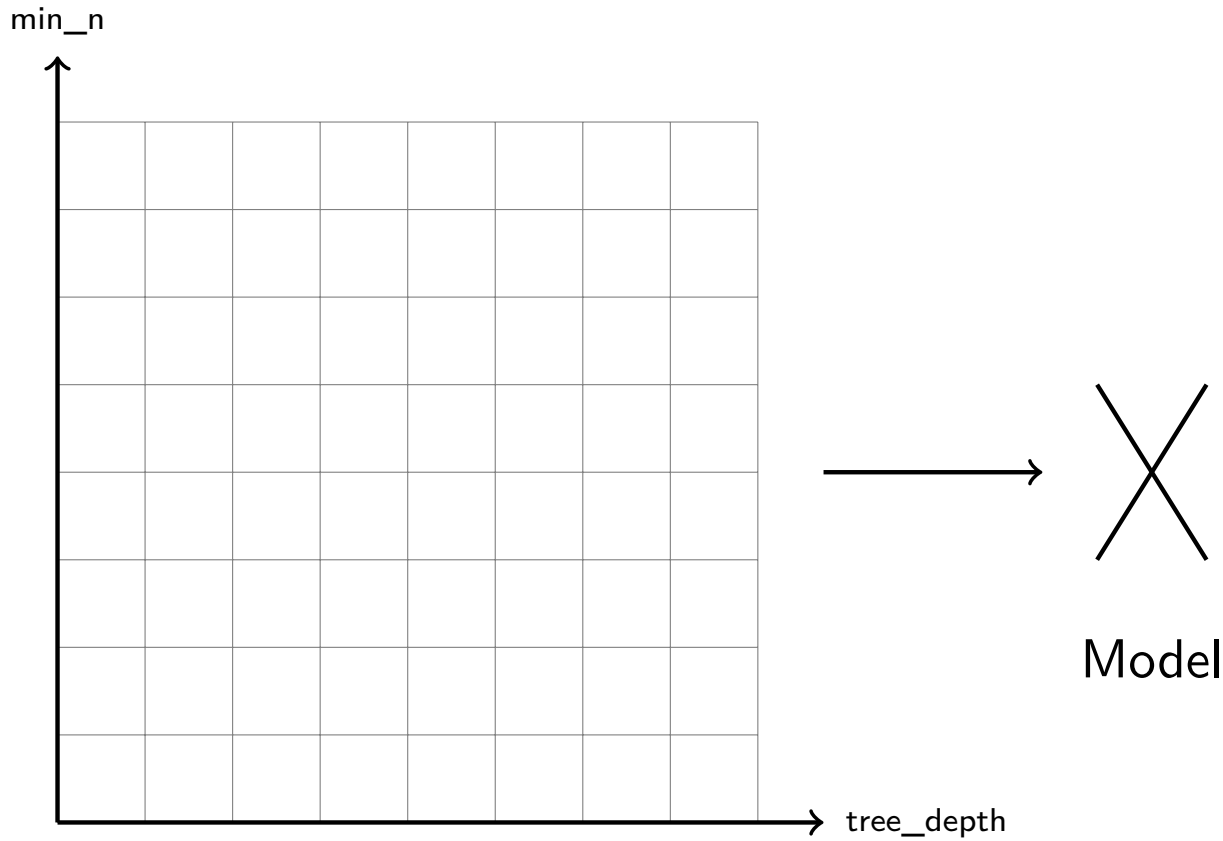
```
decision_tree(min_n = 20,
              tree_depth = 30,
              cost_complexity = 0.01)
```

These work well in many cases, but may not be optimal for all datasets.

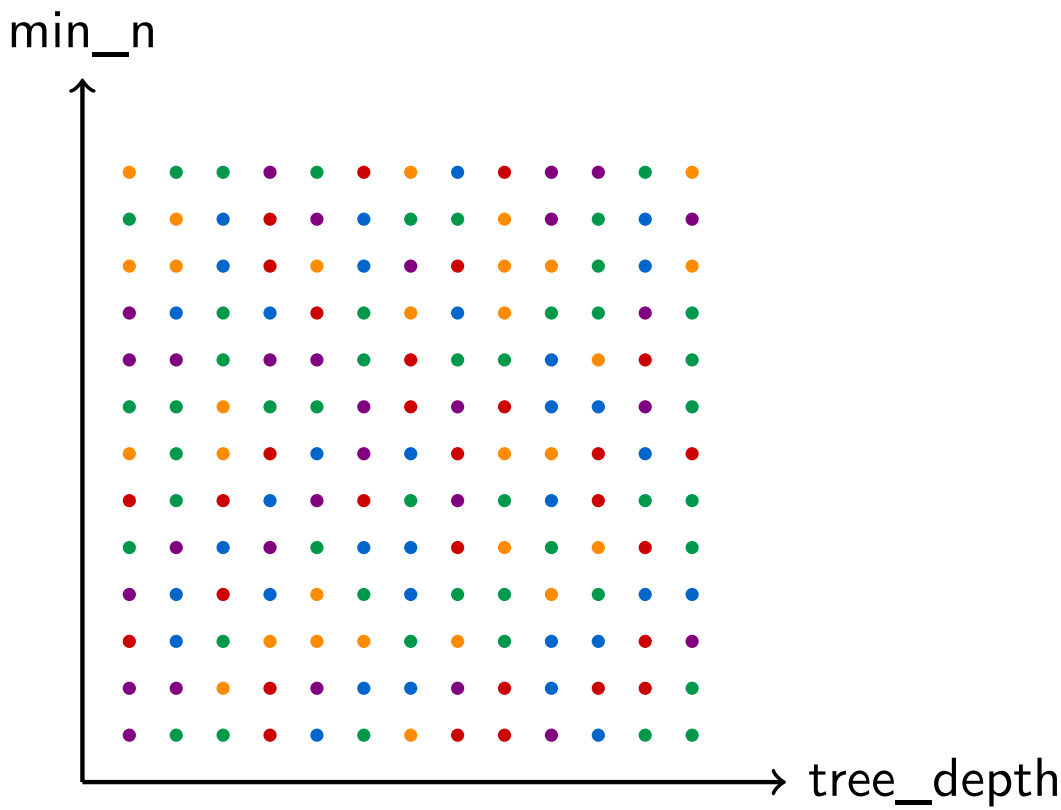
Goal of hyperparameter tuning: Find the optimal set of hyperparameter values for your specific data.



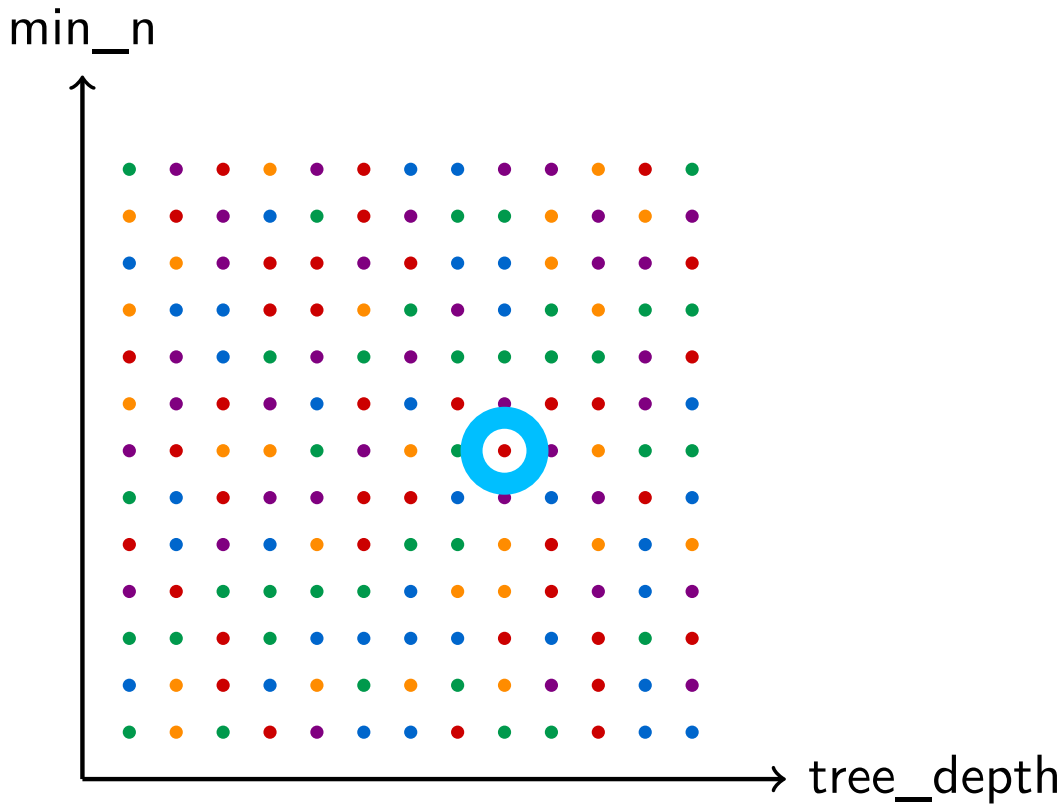
Grid search: Systematically evaluate model performance across combinations of hyperparameter values.



Each point on the grid represents a unique model to train and evaluate.



Different colors represent different performance levels across the hyperparameter space.



The goal is to identify the hyperparameter combination that maximizes performance.

Step 1: Create Placeholders with tune()

```
# Mark hyperparameters for tuning
spec_untuned <- decision_tree(
  min_n = tune(),
  tree_depth = tune()
) %>%
  set_engine("rpart") %>%
  set_mode("classification")
```

```
Decision Tree Model Specification (classification)

Main Arguments:
  tree_depth = tune()
  min_n = tune()
```

Purpose: Labels parameters for tuning

Effect: Tells `tidymodels` which hyperparameters should be optimized rather than fixed

The rest of the model specification follows the usual pattern: - Set the engine - Set the mode - Later, fit with data

Using grid_regular()

```r
# Create a regular grid of hyperparameter values
tree_grid <- grid_regular(
  parameters(spec_untuned),
  levels = 3
)

print(tree_grid)
```

```
# A tibble: 9 x 2
  min_n tree_depth
  <int>      <int>
1     2          1
2    21          1
3    40          1
4     2          8
5    21          8
6    40          8
7     2         15
8    21         15
9    40         15
```

Function: Creates a complete factorial grid

Arguments:

- `parameters(spec_untuned)`: Extracts tunable parameters from model spec
- `levels = 3`: Number of values to try for each hyperparameter

With 2 hyperparameters and 3 levels each: $3 \times 3 = 9$ combinations

```
tree_grid
```

min_n	tree_depth
2	1
21	1
40	1
2	8
21	8
40	8
2	15
21	15
40	15

Each row represents one model configuration to evaluate.

Using tune_grid()

```
# Tune the model across all grid combinations
tune_results <- tune_grid(
  spec_untuned,
  outcome ~ .,
  resamples = my_folds,
  grid = tree_grid,
  metrics = metric_set(accuracy)
)
```

What happens:

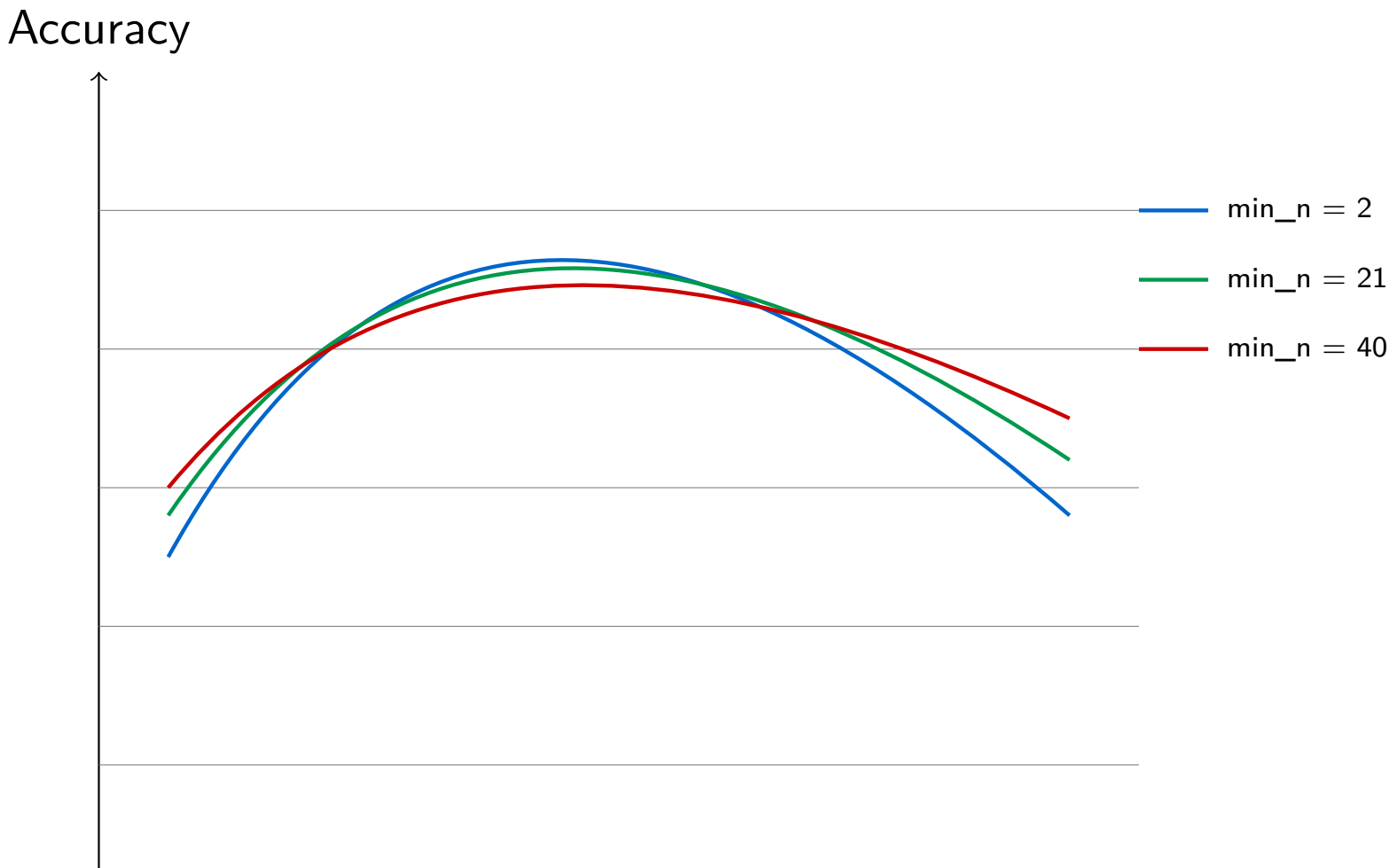
1. Builds a model for every grid point
2. Evaluates each model using cross-validation
3. Records performance metrics for each combination

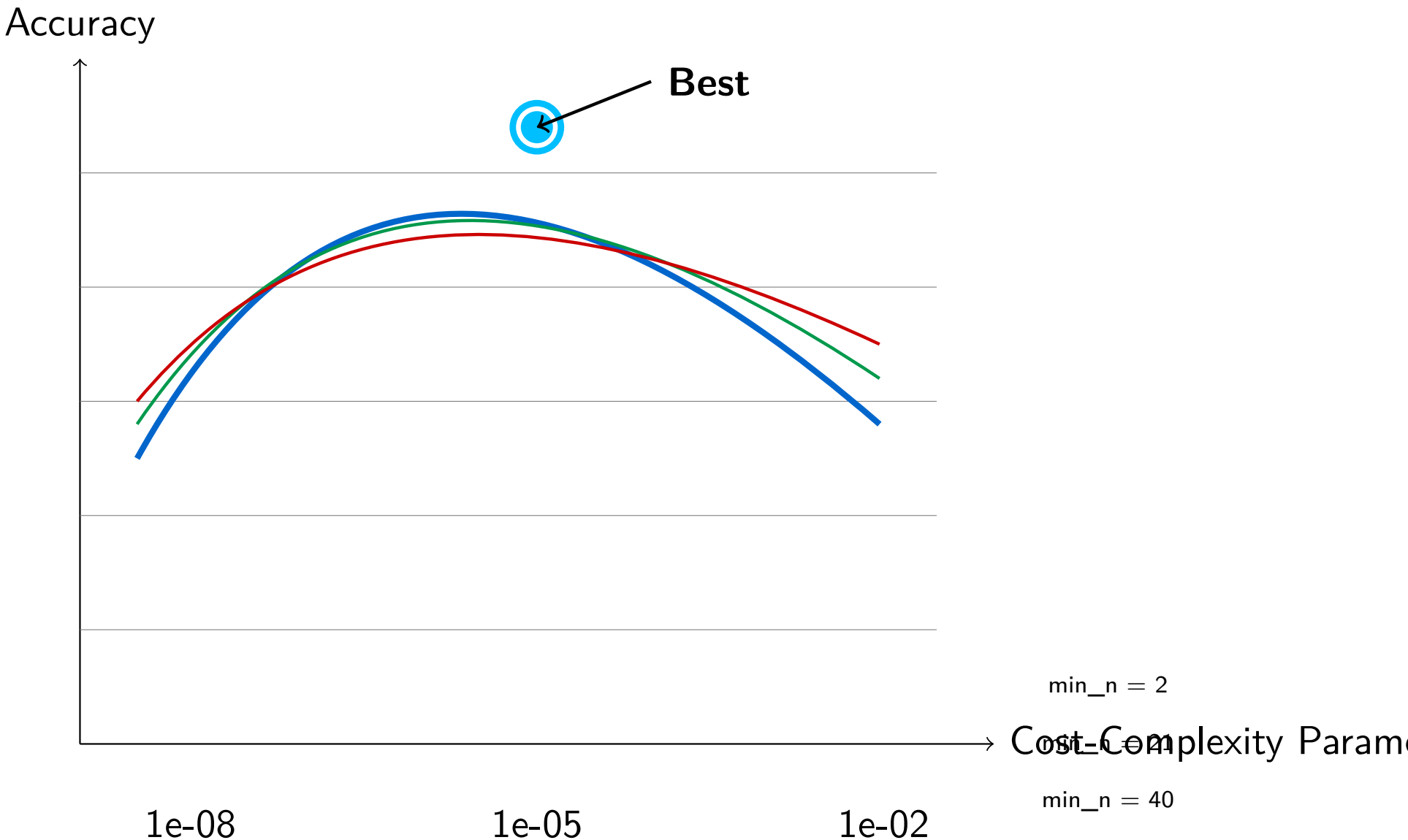
```r
tune_grid(
  spec_untuned,              # Untuned model specification
  outcome ~ .,               # Model formula
  resamples = my_folds,      # CV folds
  grid = tree_grid,          # Hyperparameter grid
  metrics = metric_set(accuracy)  # Evaluation metrics
)
```

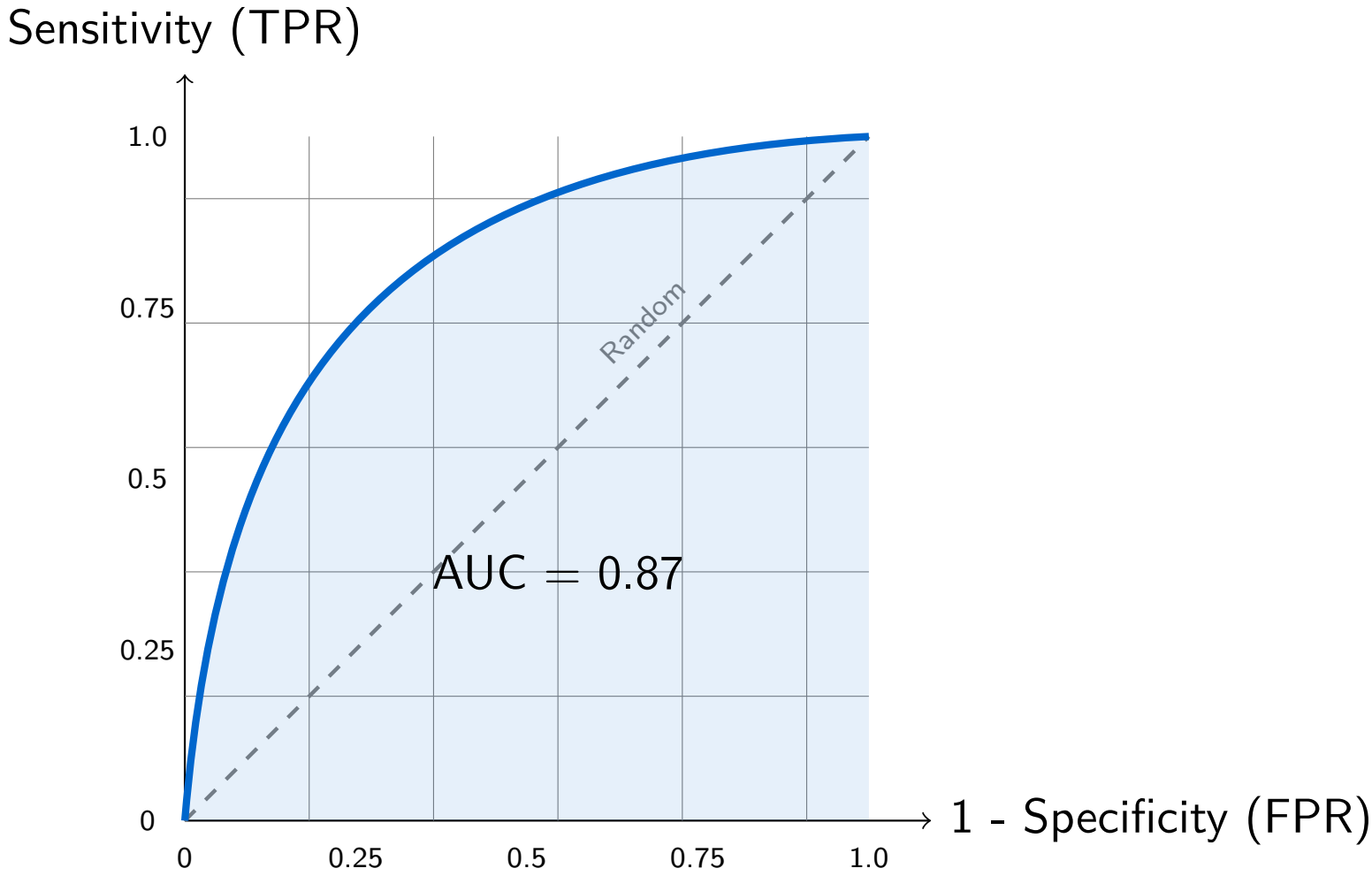
Computational cost: Fits models $\times$ folds total models

Example: 9 grid points $\times$ 5 folds $=$ 45 models trained

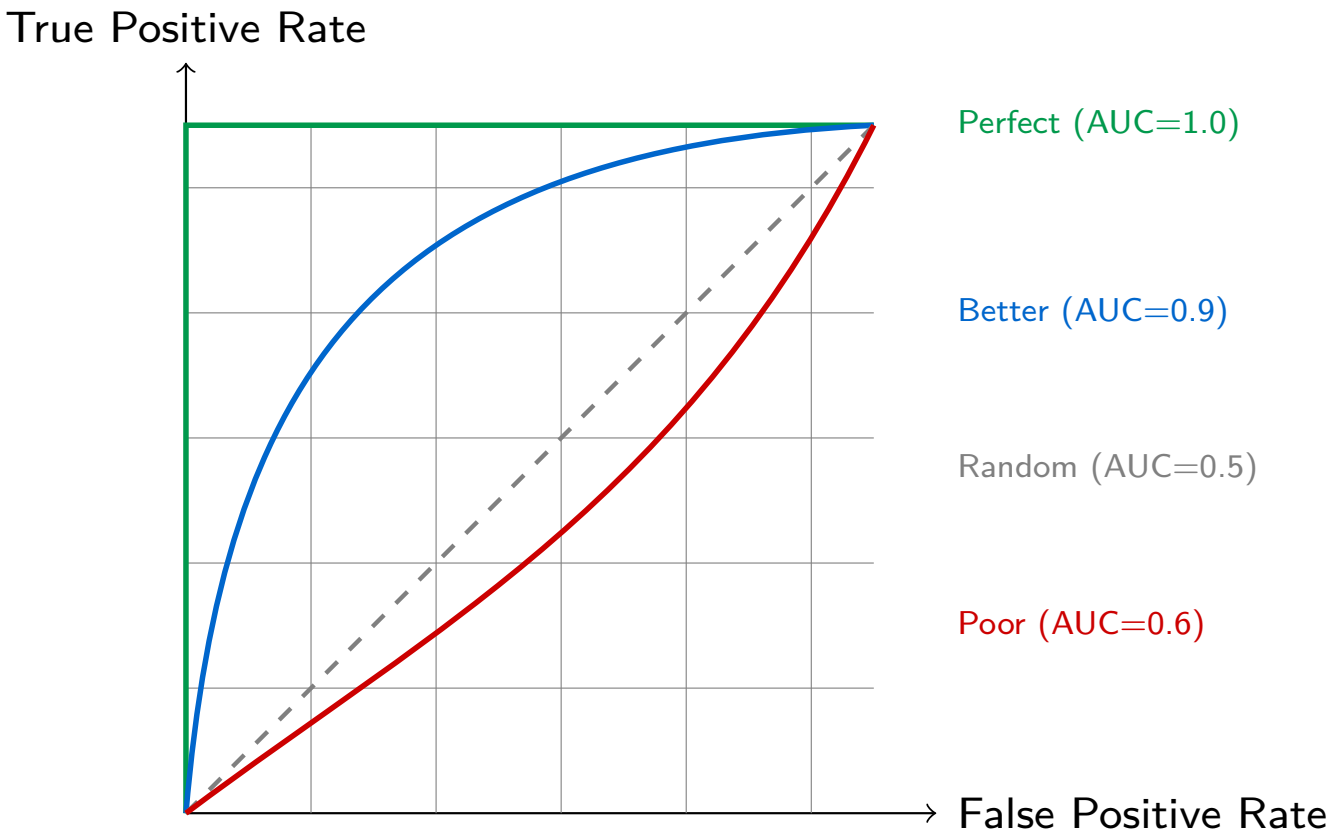
```r
# Automatically plot performance across grid
autoplot(tune_results)
```




Accuracy
Best
min_n = 2
min_n = 21
min_n = 40
Cost-Complexity Parame
1e-08
1e-05
1e-02

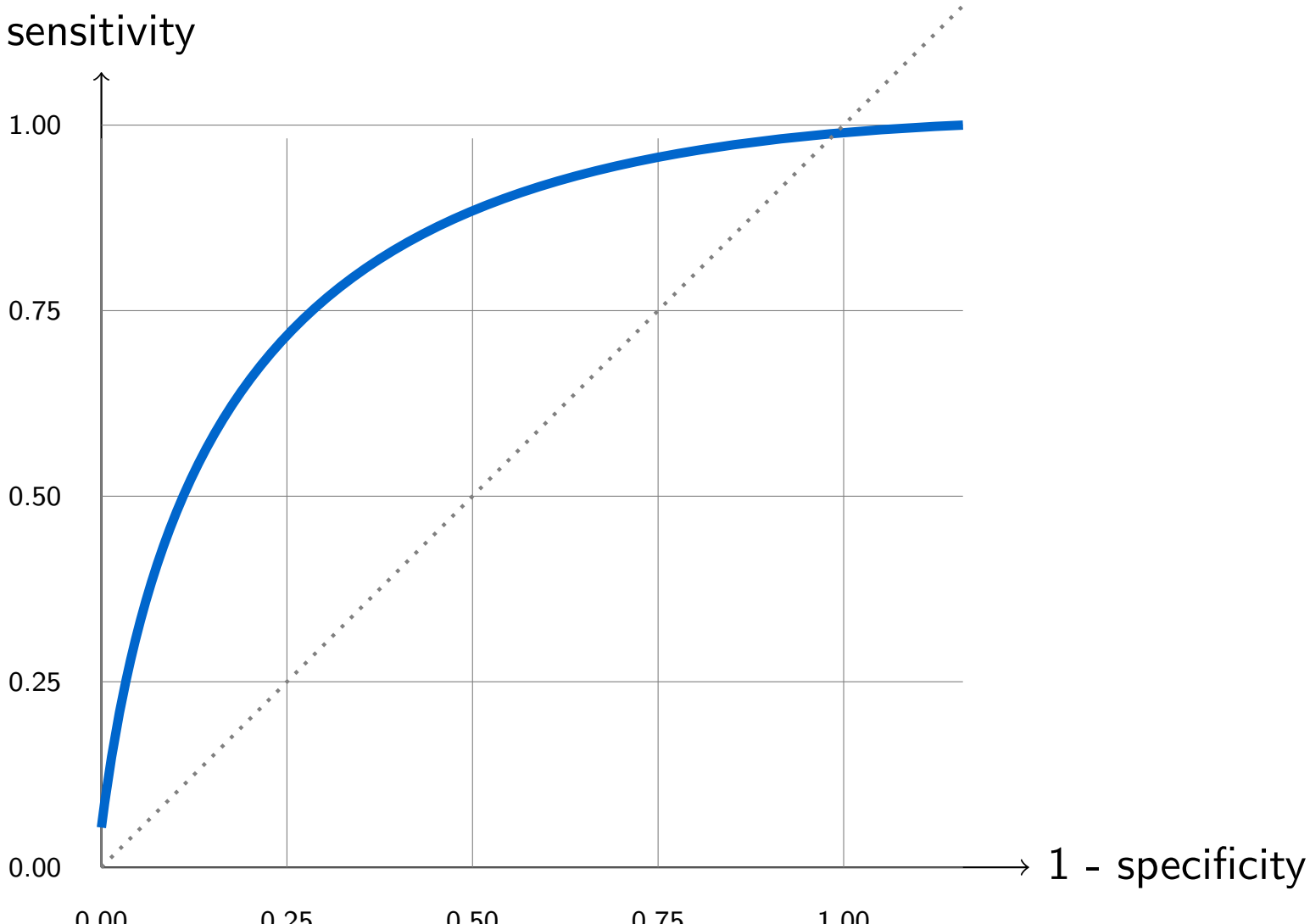


The curve shows the trade-off between sensitivity and specificity across all thresholds.



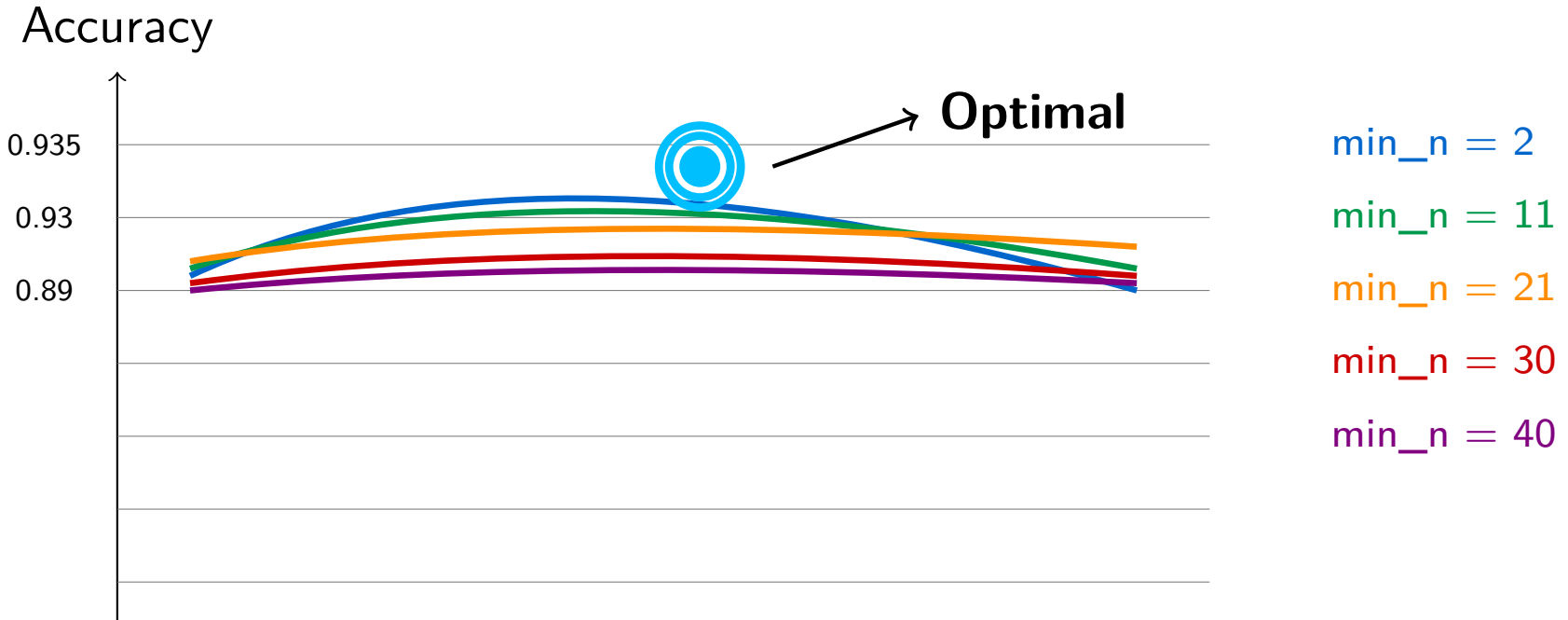
Better models have curves closer to the top-left corner with higher AUC values.

```r
# Plot the ROC curve
autoplot(roc)
```



Key elements of the tuning visualization:

- **X-axis:** Cost-complexity parameter (displayed on log scale)
- **Y-axis:** Model accuracy
- **Multiple lines:** Each represents a different `min_n` value
- **Goal:** Identify the hyperparameter combination with peak accuracy



Using select_best()

```r
# Extract the hyperparameters with best performance
final_params <- select_best(tune_results)

print(final_params)
```

```
# A tibble: 1 x 3
  min_n tree_depth .config
  <int>      <int> <chr>
1     2          8 Model4
```

Result: The combination that achieved the highest accuracy during tuning.

Using finalize_model()

```r
# Plug best parameters into the specification
best_spec <- finalize_model(spec_untuned,
                            final_params)

print(best_spec)
```

```
Decision Tree Model Specification (classification)

Main Arguments:
  tree_depth = 8
  min_n = 2

Computational engine: rpart
```

```r
# Train final model on full training set
final_model <- best_spec %>%
  fit(outcome ~ ., data = train_data)

# Evaluate on test set
test_predictions <- predict(final_model,
                            new_data = test_data)

# Calculate final performance
accuracy(test_predictions, truth = outcome)
```

This is the model you'd deploy after confirming good test set performance.

1. Create model spec with `tune()` placeholders
2. Generate hyperparameter grid with `grid_regular()`
3. Tune models with `tune_grid()` using cross-validation
4. Visualize results with `autoplot()`
5. Select best parameters with `select_best()`
6. Finalize model spec with `finalize_model()`
7. Train final model on full training data
8. Evaluate on test set

This systematic approach prevents overfitting and finds optimal configurations.

Let's tune!

More model measures

Example: A “naive” model that always predicts “no”

In an imbalanced dataset with 98% negative samples:

- Naive accuracy: 98%
- Actual value: The model learns nothing!

Problem: Accuracy alone can be misleading when classes are imbalanced.

Solution: Use additional metrics that capture different aspects of performance.

		Truth	
		Yes	No
Prediction	Yes	TP	FP
	No	FN	TN

- **TP (True Positive):** Correctly predicted positive
- **TN (True Negative):** Correctly predicted negative
- **FP (False Positive):** Incorrectly predicted positive
- **FN (False Negative):** Incorrectly predicted negative

Definition: Proportion of actual positives correctly classified

$$\text{Sensitivity} = \frac{TP}{TP + FN}$$

		Truth	
		Yes	No
Prediction	Yes	**TP**	FP
	No	**FN**	TN

Also known as: Recall

Definition: Proportion of actual negatives correctly classified

$$\text{Specificity} = \frac{TN}{TN + FP}$$

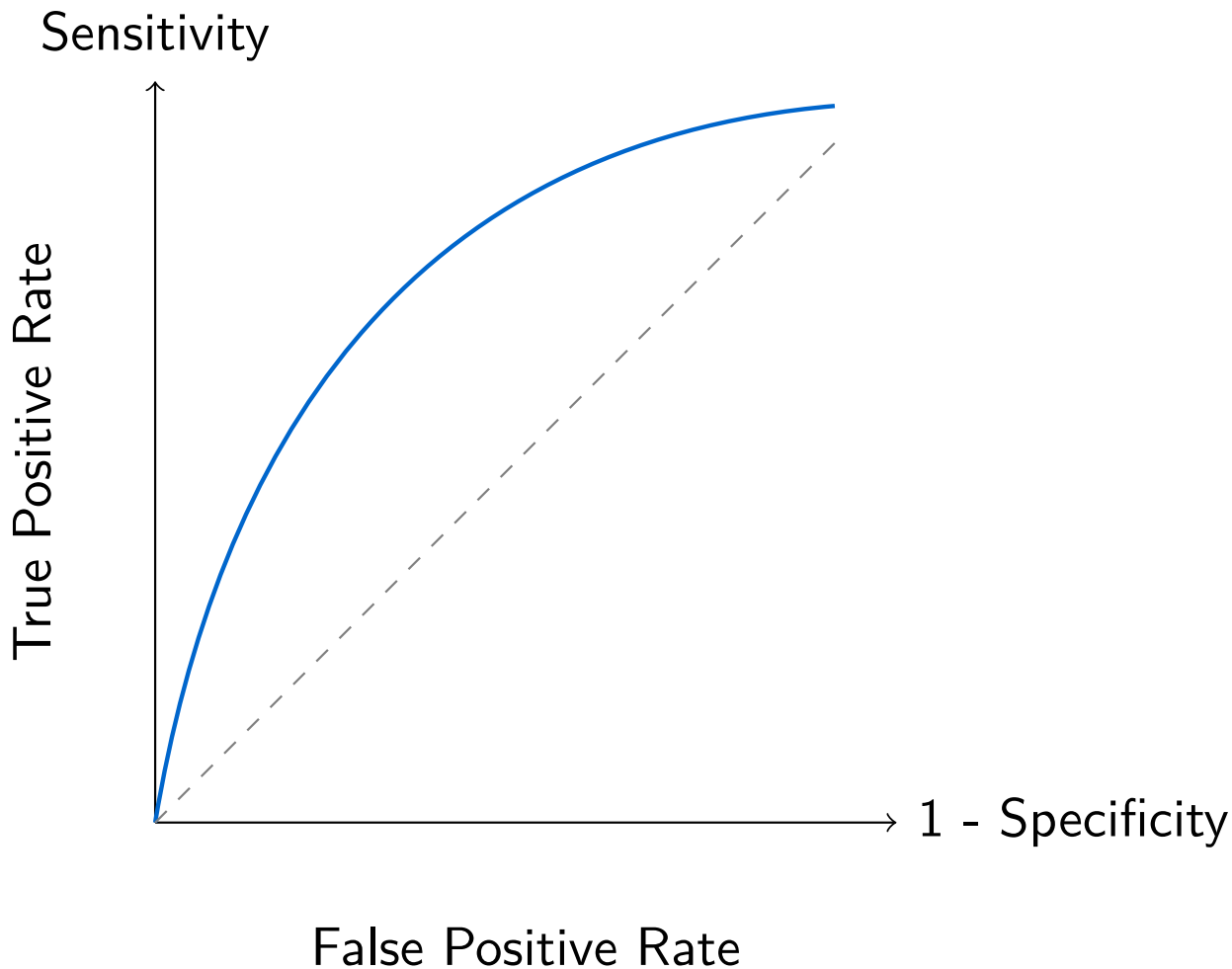
		Truth	
		Yes	**No**
Prediction	Yes	TP	**FP**
	No	FN	**TN**

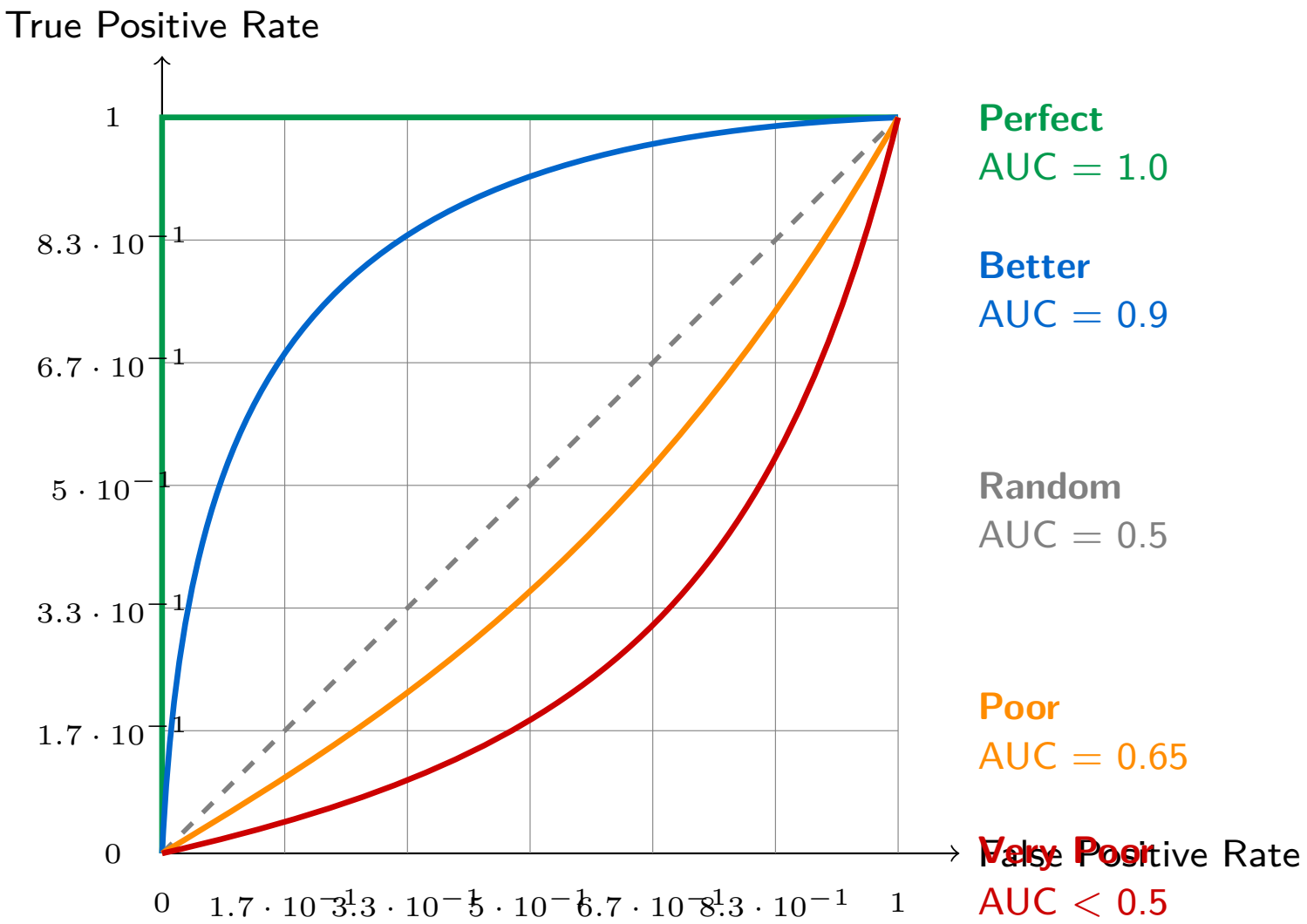
Pred Prob	T = 0.3	T = 0.5	T = 0.75
0.8	yes	yes	yes
0.5	yes	yes	no
0.7	yes	yes	no
0.6	yes	yes	no
0.3	yes	no	no
0.5	yes	yes	no
0.3	yes	no	no

Different thresholds produce different confusion matrices and performance metrics.

Receiver Operating Characteristic Curve

Visualizes model performance across all possible thresholds.





Key insights:

AUC summarizes ROC curve into a single number:

- **AUC = 1.0:** Perfect classification
- **AUC = 0.9-1.0:** Excellent
- **AUC = 0.8-0.9:** Good
- **AUC = 0.7-0.8:** Fair
- **AUC = 0.6-0.7:** Poor
- **AUC = 0.5:** No better than random
- **AUC $<$ 0.5:** Worse than random (predictions inverted)

```r
# Calculate sensitivity
sens(predictions,
     estimate = .pred_class,
     truth = true_class)
```

```
# A tibble: 1 x 2
  .metric     .estimate
  <chr>           <dbl>
1 sensitivity     0.872
```

Arguments:

- `predictions`: Data frame with predictions and truth
- `estimate`: Column with predicted classes
- `truth`: Column with actual classes

```r
# Predict probabilities (not classes)
predictions <- predict(model,
                       data_test,
                       type = "prob") %>%
  bind_cols(data_test)

# Calculate ROC curve for all thresholds
roc <- roc_curve(predictions,
                 estimate = .pred_yes,
                 truth = still_customer)

# Plot the curve
autoplot(roc)
```

```r
# Calculate area under the ROC curve
roc_auc(predictions,
        estimate = .pred_yes,
        truth = still_customer)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc binary         0.872
```

Same arguments as `roc_curve()`, but returns single summary statistic.

Let's measure!

Section 1

Model Comparison Framework

Comparing multiple modeling approaches requires consistent evaluation metrics and a structured comparison framework.

Models to Compare:

- Single decision tree (baseline)
- Bagged trees (bootstrap aggregation)
- Random forest (feature randomization)
- Gradient boosting (sequential learning)

Comparison Metrics:

- AUC-ROC for overall discrimination ability
- ROC curves for threshold analysis
- Training time and computational efficiency
- Model complexity and interpretability

Each model is trained on identical training data and evaluated on the same held-out test set to ensure fair comparison.

Section 2

Combining Predictions

We consolidate predictions from all models into a single dataset with the true outcomes for systematic comparison.

```
# Combine predictions from all models
preds_combined <- bind_cols(
  preds_tree,       # Decision tree predictions
  preds_bagging,    # Bagged tree predictions
  preds_forest,     # Random forest predictions
  preds_boosting,   # Boosted tree predictions
  test_data %>% select(outcome)
)
```

```
# A tibble: 1,011 × 5

   preds_tree preds_bagging preds_forest preds_boosting outcom
        <dbl>         <dbl>        <dbl>          <dbl> <fct>
 1      0.144         0.115       0.0333          0.136 no
```

Section 3

Computing AUC for Each Model

We calculate AUC-ROC for each model to quantify discrimination ability on the test set.

```
# Calculate AUC for all models
auc_comparison <- bind_rows(
  decision_tree = roc_auc(preds_combined, outcome, preds_tree)
  bagged_trees  = roc_auc(preds_combined, outcome, preds_baggi
  random_forest = roc_auc(preds_combined, outcome, preds_fores
  boosted_trees = roc_auc(preds_combined, outcome, preds_boost
  .id = "model"
)
```

```
# A tibble: 4 × 3

  model         .metric .estimate

  <chr>         <chr>       <dbl>

1 decision_tree roc_auc     0.911
```

Section 4

ROC Curve Comparison

ROC curves provide comprehensive visualization of model discrimination across all classification thresholds.

```
# Reshape predictions to long format
predictions_long <- tidyr::pivot_longer(
  preds_combined,
  cols = starts_with("preds_"),
  names_to = "model",
  values_to = "predictions"
)

# Calculate ROC curves for each model
roc_curves <- predictions_long %>%
  group_by(model) %>%
  roc_curve(truth = outcome, estimate = predictions)

# Visualize
autoplot(roc_curves)
```

Section 5

Interpreting Model Comparison

The comparison reveals clear performance hierarchy and provides insights for model selection decisions.

Performance Ranking:

1. Gradient boosting (AUC = 0.984) - Best overall discrimination
2. Random forest (AUC = 0.974) - Strong performance, more stable
3. Bagged trees (AUC = 0.936) - Moderate improvement over single tree
4. Single tree (AUC = 0.911) - Baseline performance

Key Insights:

Ensemble methods consistently outperform single trees. Boosting achieves highest performance but requires careful tuning. Random forests offer excellent performance with fewer hyperparameters. The performance gain from bagging to random forests demonstrates feature randomization benefits.

Section 6

Let's Compare!

Let's compare!

Section 7

Lecture Summary

We explored gradient boosting algorithms and systematic model optimization techniques for achieving state-of-the-art predictive performance.

Core Concepts:

- AdaBoost introduces adaptive weighting for sequential learning
- Gradient boosting optimizes loss functions through gradient descent
- Boosted models require careful hyperparameter tuning across multiple dimensions
- Systematic tuning workflow: specify, grid, validate, select, finalize, fit
- Model comparison requires consistent metrics and evaluation frameworks

Practical Skills:

You can now implement gradient boosted models with XGBoost, conduct comprehensive hyperparameter tuning using grid search and cross-validation,

Section 8

Course Wrap-Up

Chapter 1: Classification trees - Decision trees, data splitting, confusion matrices

Chapter 2: Regression trees - MAE, RMSE, cross-validation, bias-variance tradeoff

Chapter 3: Hyperparameter tuning - Grid search, bagging, random forests

Chapter 4: Boosting - AdaBoost, gradient boosting, model comparison

You now have a complete toolkit for building, tuning, and deploying tree-based machine learning models in R!

Section 9

Thank You!

Thank you!

Machine Learning with

Tree-Based Models in R