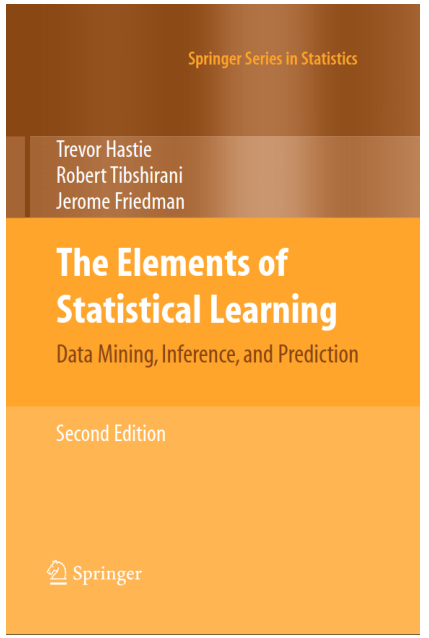
Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Supervised ML 🧑

[Dr. Hanjo Odendaal]

The bible of machine learning

- Focuses on understanding the relationship between input variables (features) and output variables (responses).
    - **Supervised:** Learning with labeled data (e.g., regression, classification).
    - **Unsupervised:** Learning from unlabeled data (e.g., clustering, dimensionality reduction).
- Key Techniques:
    - Linear Methods (e.g., Linear Regression, Logistic Regression)
    - Nonlinear Methods (e.g., Decision Trees, SVMs, Neural Networks)
    - Model Assessment & Selection (e.g., Cross-Validation, Bias-Variance Tradeoff)
    - Ensemble Methods (e.g., Bagging, Boosting, Random Forests)



Introduction to prediction problems

In statistical modeling and machine learning, the core goal is often predicting an outcome y given inputs X:

$$y \sim f(X)$$

This can be unknown and complex (nonlinear, flexible) or specified with assumptions (linear, parametric).

Classical statistics - ("Is β different from zero?"):

- Focus on parameter estimation and inference.
- Models are often prescriptive: impose assumptions (e.g., linearity, normality).

Machine learning - "Can I predict y accurately from X?":

- Focus on prediction accuracy rather than interpretability.
- Models are often descriptive or agnostic: flexible function fitting without strong assumptions.

Introduction to prediction problems

In both classical statistics and machine learning, the ultimate goal is to find a good approximation $\hat{f}(X)$ to the unknown true function $f(X)$. Prediction error can be decomposed into three parts:

$$\mathbb{E}\left[(y - \hat{f}(X))^2\right] = \underbrace{\left[\text{Bias}(\hat{f}(X))\right]^2}_{\text{Systematic error}} + \underbrace{\text{Variance}(\hat{f}(X))}_{\text{Model instability}} + \underbrace{\text{Irreducible error}}_{\text{Noise in } y}$$

- **Bias:** Error from wrong assumptions about $f(X)$ (e.g., assuming linear when it is nonlinear).
- **Variance:** Error from model sensitivity to training data (e.g., overfitting).
- **Irreducible error:** Noise or randomness in yy that no model can eliminate.

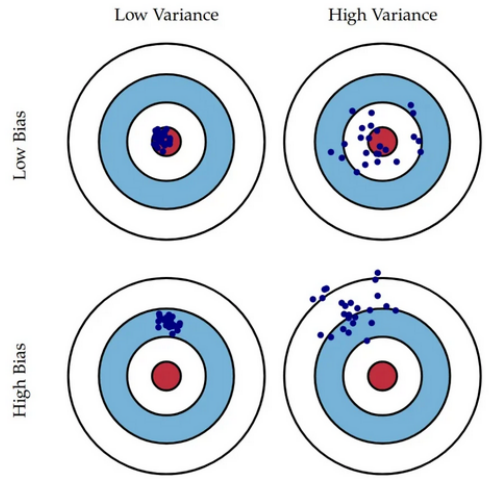


Fig. 1 Graphical illustration of bias and variance.

Introduction to prediction problems

In both classical statistics and machine learning, the ultimate goal is to find a good approximation $\hat{f}(X)$ to the unknown true function $f(X)$. Prediction error can be decomposed into three parts:

$$\mathbb{E}\left[(y - \hat{f}(X))^2\right] = \underbrace{\left[\text{Bias}(\hat{f}(X))\right]^2}_{\text{Systematic error}} + \underbrace{\text{Variance}(\hat{f}(X))}_{\text{Model instability}} + \underbrace{\text{Irreducible error}}_{\text{Noise in } y}$$

- **Bias:** Error from wrong assumptions about $f(X)$ (e.g., assuming linear when it is nonlinear).
- **Variance:** Error from model sensitivity to training data (e.g., overfitting).
- **Irreducible error:** Noise or randomness in yy that no model can eliminate.

Simple models (high bias, low variance) vs flexible models (low bias, high variance):

- Bias and variance are *modelable* - we can trade them off.
- Irreducible error is *unmodelable* - it sets a floor on how well any model can ever perform.

Tradeoff between underfitting and overfitting. Need validation techniques (cross-validation, regularization).

Two types of problems

Regression

- Objective: Predict a *continuous* target (house price, salary, GDP)

$$y \in \mathbb{R}$$

- Model:

$$y = f(X) + \varepsilon, \quad \text{with } \mathbb{E}[\varepsilon] = 0$$

- Loss Function (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}\left(X_i\right)\right)^2}$$

- **Goal**: Minimize prediction error on real-valued outcomes

Classification

- Objective: Predict a categorical target (Default, Customer Cluster, Recession)

$$y \in \{1, 2, \ldots, K\}$$

- Model (e.g., logistic):

$$P(y = 1 \mid X) = \sigma(f(X)) = \frac{1}{1 + e^{-f(X)}}$$

- Loss Function (Log Loss):

$$-\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)\right]$$

- **Goal**: Maximize classification accuracy or likelihood

Hyperparameter tuning and the Bias–Variance Trade-off

🎯 Why does tuning matter?

- We've learned there's a trade-off between **bias** and **variance**.
- Many models allow us to adjust this trade-off using **hyperparameters**.

🔧 What is a hyperparameter?

- A **hyperparameter** is a tuning knob - it controls how flexible or stable the model is.
- It **changes the behavior of the learner** (the algorithm itself).
- Not all models have hyperparameters:
  - Example: **Ordinary Least Squares (OLS)** has no tuning parameters.
  - That's why this idea might seem new.

Example: Random Forest

- The Random Forest model has **one key hyperparameter**:

$$\texttt{mtry} = \text{number of features considered at each split}$$

- **Small** `mtry`:
  - More randomness across trees
  - Higher variance in splits
  - Lower correlation between trees → better averaging
- **Large** `mtry`:
  - Trees become more similar
  - Lower variance per tree, but higher correlation across the forest
- The **best value of** `mtry` lies somewhere in between... but how do we choose `mtry`?

Cross-validation

> That's where **cross-validation** comes in - it helps us estimate which value gives the best generalization.

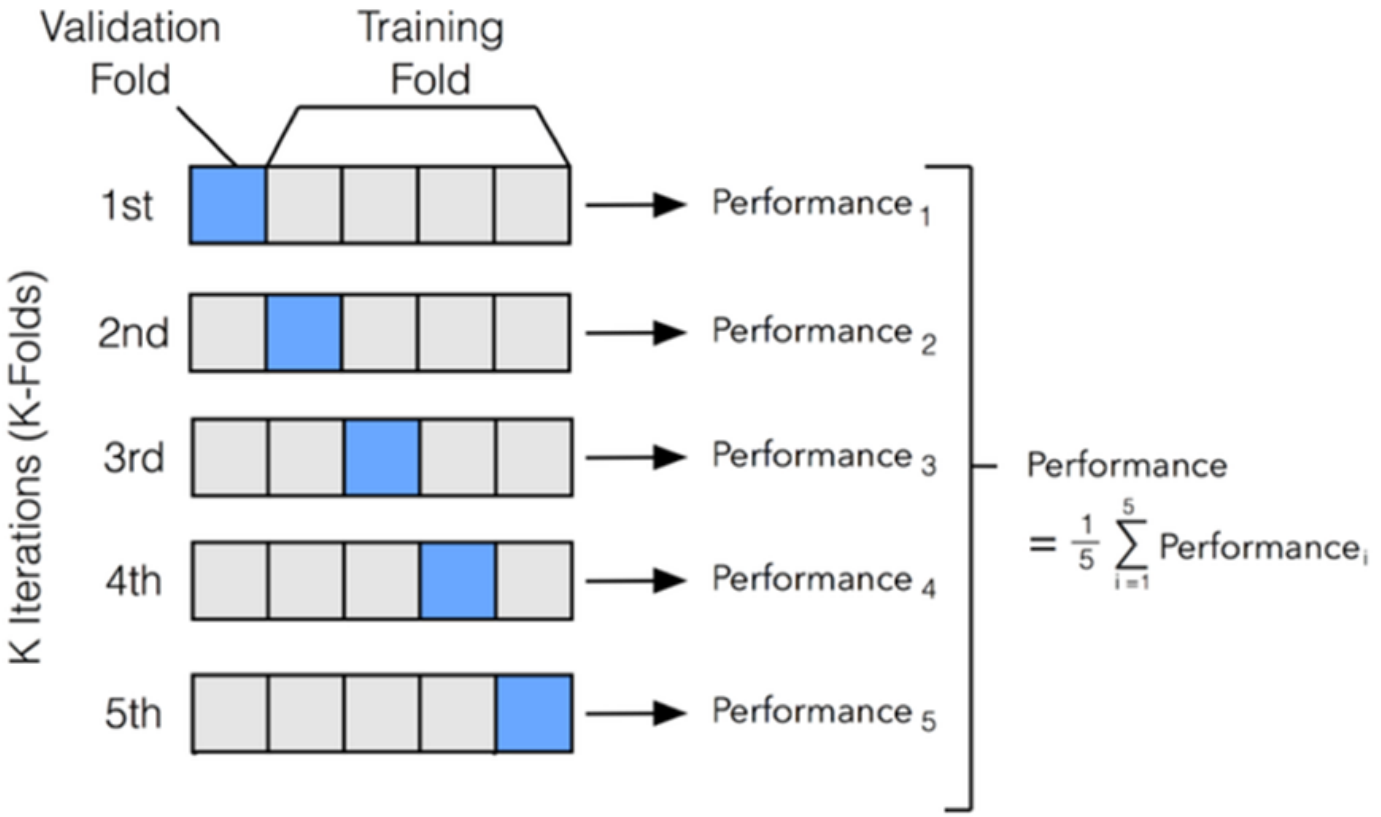
Why do we need resampling?

- Training error often **underestimates** the true error.
- A model that performs well on training data may not generalize to **new data**.
- We need a method to **assess performance reliably** on unseen data.

K-fold Cross-Validation:

- Split data into k equal-sized **folds**
- For each fold $i = 1, \ldots, k$:
  - Fit model on $k - 1$ folds
  - Evaluate on the i-th fold
- Get k test errors: $\varepsilon_1, \ldots, \varepsilon_k$
- Average them: $\mathrm{CV}_{(k)} = \frac{1}{k} \sum_{i=1}^{k} \varepsilon_i$

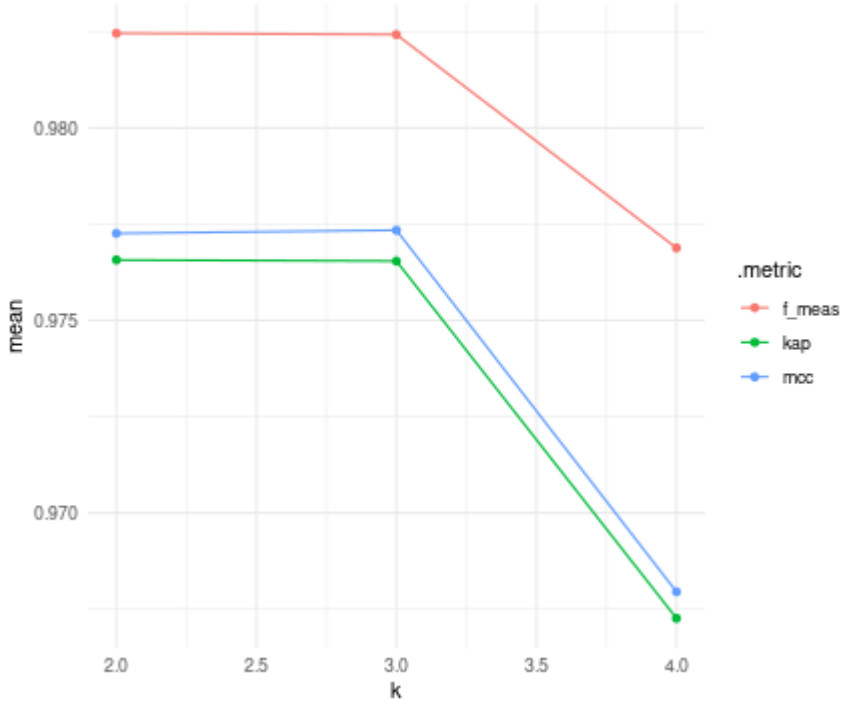
Cross-validation



https://medium.com/@ompramod9921/cross-validation-623620ff84c2

Hyperparameter tuning

```r
library(tidymodels)
library(ranger)

mtry ← c(2, 3, 4)
split  ← initial_split(penguins, prop = 0.7)  # Split the dataset
train  ← training(split)  # Training set
test   ← testing(split)  # Test set
penguins_cv ← vfold_cv(penguins, v = 10)

pred_penguin ← function(k){

  run_penguin ← function(df){
    training ← analysis(df)
    testing ← assessment(df)
    fit ← ranger(species ~ ., mtry = k, data = training, num.trees
    probs ← predict(fit, testing)$predictions

    testing_probs ← testing %>%
      select(species) %>%
      bind_cols(as_tibble(probs))

    bind_rows(
      f_meas(testing_probs, truth = species, estimate = value),
      mcc(testing_probs, truth = species, estimate = value),
      kap(testing_probs, truth = species, estimate = value)
    )
  }

  penguins_cv %>% mutate(res = map(splits, run_penguin)) %>%
    select(-splits) %>%
    mutate(k = k, .before = 1)
```



Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Algo friends ⚒

Introduction to all the algo friends

We will not dive into deep learning yet. Start with linear models and trees, gradually build to more complex ensemble methods.

Linear Methods:

- Assume a linear relationship between inputs and outputs: OLS, Lasso, Ridge

Tree-Based Methods:

- Partition the input space into regions and fit simple models locally: CART, RF and Boosting

Support Vector Machines (SVMs):

- Find hyperplanes that best separate classes with maximum margin. Powerful for both linear and nonlinear classification.

> All of these models are going to need **hyperparameter** tuning. More on this later.

Regularization (Elastic Band)

Ridge regression (Elastic Band): You want a *good fit*, but you're penalised for large coefficients. You're trying to fit the best line, but each slope (coefficient) is tied to zero with a rubber band. The more you stretch a beta, the more the rubber band pulls back.

> *Keep everything, just shrink it toward average.*

For Ridge Regression, we have to solve an optimization objective (Closed-form):

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = \arg\min_{\boldsymbol{\beta}} \underbrace{\sum_{i=1}^{n} (y_i - \mathbf{X}_i^\top \boldsymbol{\beta})^2}_{\text{fit to data}} + \underbrace{\lambda \sum_{j=1}^{p} \beta_j^2}_{\text{rubber band penalty}}$$

- The first term is just **Ordinary Least Squares**: make predictions close to the observed data.
- The second term is the ℓ_2 penalty: the sum of *squares* of the coefficients:
- $\lambda = 0 \rightarrow$ no rubber band $\rightarrow$ pure OLS
- $\lambda = 10 \rightarrow$ strong pull $\rightarrow$ coefficients shrink closer to zero

Regularization (Fly Trap)

Lasso regression (Fly Trap): You want to only keep those coefficients that have something to offer. Little pushes don't move the coefficient - it gets stuck at zero. But big, important ones break free.

> *Instead of a rubber band, the coefficient is moving on a fly trap.*

$$\hat{\boldsymbol{\beta}}_{\text{lasso}} = \arg\min_{\boldsymbol{\beta}} \underbrace{\sum_{i=1}^{n} (y_i - \mathbf{X}_i^\top \boldsymbol{\beta})^2}_{\text{fit to data}} + \underbrace{\lambda \sum_{j=1}^{p} |\beta_j|}_{\text{FlyTrap penalty}}$$

- Still a trade-off between **fit** and **simplicity**
- But now the penalty is the ℓ_1 norm - the sum of *absolute* values

🪰 Why is it "sticky"?

- The *absolute value* has a kink at 0 (non-differentiable).
- This is why Lasso can perform **variable selection**
  - It doesn't just **shrink**, it **zeros out**

Elastic Nets: Fly Trap + Rubber Band

Elastic Net combines **Lasso** and **Ridge** penalties:

> *It's like a fly trap with rubber bands. Some coefficients stick (Lasso), others stretch gently (Ridge).*

(1) In high-dimensional settings ($p > n$), Lasso can behave erratically and (2) If predictors are **highly correlated**, Lasso tends to pick one and ignore the others.

$$\hat{\boldsymbol{\beta}}_{\text{EN}} = \arg\min_{\boldsymbol{\beta}} \underbrace{\sum_{i=1}^{n} (y_i - \mathbf{X}_i^\top \boldsymbol{\beta})^2}_{\text{fit to data}} + \lambda \left[\underbrace{(1-\alpha) \sum_{j=1}^{p} \beta_j^2}_{\text{Rubber Band}} + \underbrace{\alpha \sum_{j=1}^{p} |\beta_j|}_{\text{Fly Trap}} \right]$$

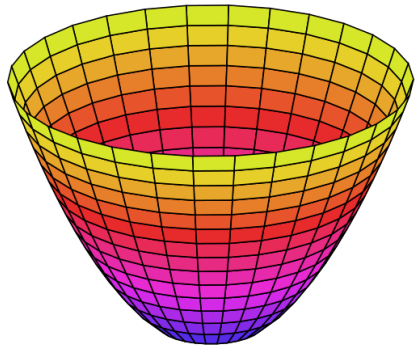
- λ: overall penalty strength
- $\alpha \in [0, 1]$: balance between Ridge and Lasso
  - $\alpha = 1$: pure Lasso, $\alpha = 0$: pure Ridge, $0 < \alpha < 1$: Elastic Net blend

Fly Trap vs Rubber Band (Intuition)

The OLS loss function is:

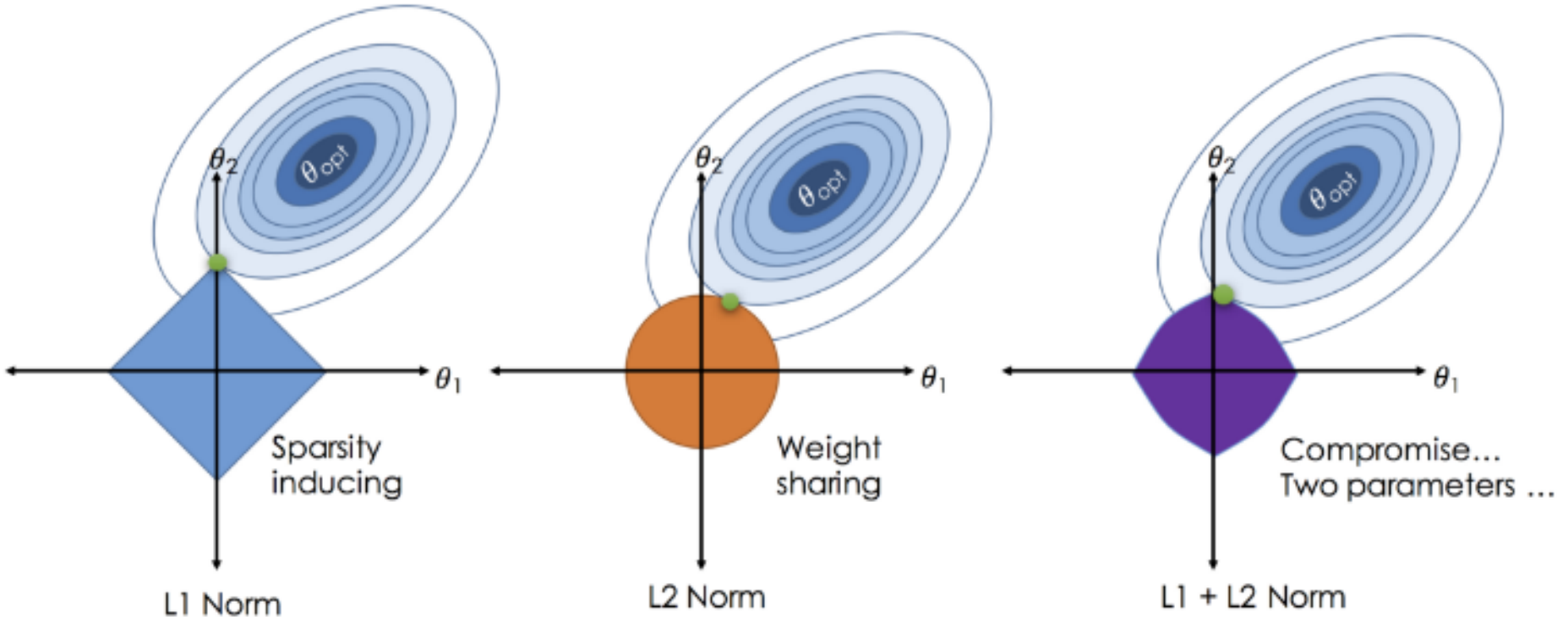
$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = \arg\min_{\boldsymbol{\beta}} \underbrace{\sum_{i=1}^{n} (y_i - \mathbf{X}_i^\top \boldsymbol{\beta})^2}_{\text{fit to data}}$$

In 2D (β_1, β_2), the level sets (i.e., constant-value slices) of this quadratic function are ellipses. For OLS, loss is a quadratic bowl in β-space - shaped like a paraboloid. If you slice the bowl horizontally, you get elliptical contours. The center of the ellipses is the OLS solution (unpenalized), where loss is minimized.



λ controls the size of the diamond (or circle for Ridge). So it's not that the diamond is pushing into the ellipses, we are sliding larger and larger ellipses outward until we hit the constraint. Better fit, more flexibility (small λ, large diamond) vs. Simple, worse fit (large λ, small diamond)

Fly Trap vs Rubber Band



https://medium.com/data-science/from-linear-regression-to-ridge-regression-the-lasso-and-the-elastic-net-4eaecaf5f7e6

Comparison with Bayesian

Ridge = Gaussian Prior

- Ridge regression adds an ℓ_2 penalty:

$$\lambda \sum_{j=1}^{p} \beta_j^2$$

- **Bayesian view**: This is equivalent to assuming

$$\beta_j \sim \mathcal{N}(0, \tau^2)$$

Coefficients are likely to be **small**, but can be nonzero. Gaussian prior → smooth penalty → no sparsity.

Lasso = Laplace Prior

- Lasso regression adds an ℓ_1 penalty:

$$\lambda \sum_{j=1}^{p} |\beta_j|$$

- **Bayesian view**: This is equivalent to assuming

$$\beta_j \sim \text{Laplace}(0, b)$$

Coefficients are expected to be close to zero, but the prior has **sharp peaks** → pushes many $\beta_j = 0$. Laplace prior encourages sparsity.

Tree methods

What is a decision tree (CART) 🌳?

A decision tree is a *recursive partitioning* method:

- Splits the input space X into *rectangular regions*
- Makes *piecewise constant predictions* within each region

How Does It Work?

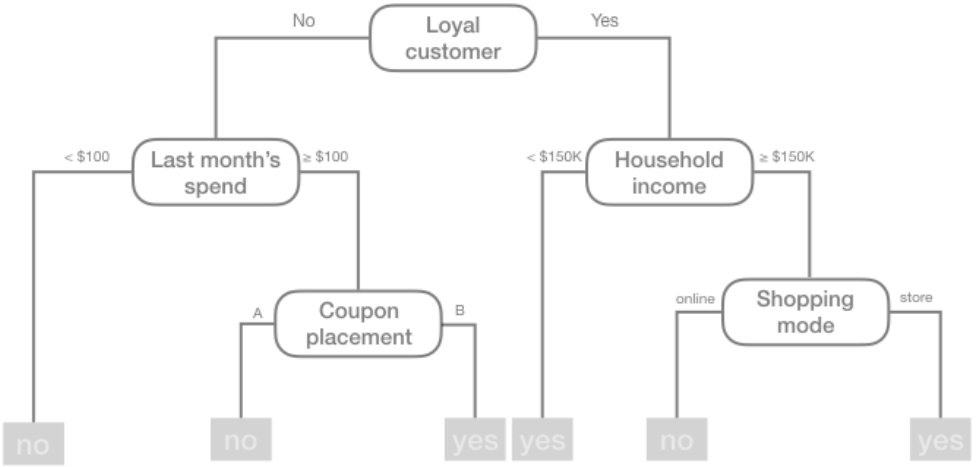
1. Find the best feature and split-point that reduce prediction error (e.g., squared error for regression).
2. Recursively repeat the split within each sub-region.
3. Stop splitting based on a **stopping rule** (e.g., minimum leaf size or depth).

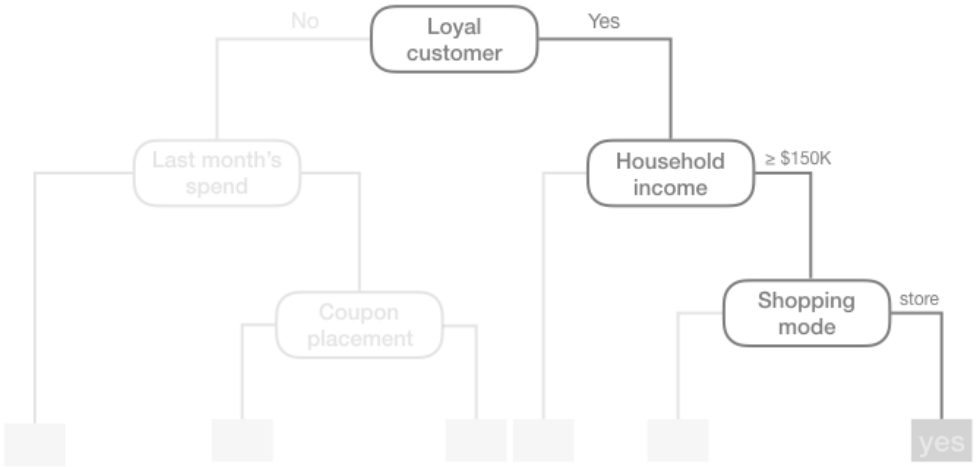
Properties

- **Non-parametric**: no assumptions about $f(X)$
- Captures interactions and **nonlinearities**
- Easy to **visualize** and **interpret**
- But: **high variance** if grown too deep

Tree methods

Below we show an example of a **CART** tree making a prediction on whether someone will use a coupon:





https://bradleyboehmke.github.io/HOML/images/exemplar-decision-tree.png

Bagging vs Boosting

Bagging (Bootstrap Aggregating): Random Forest

- Train multiple models $\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_M$ on bootstrap samples from the data.
- Final prediction:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{M} \sum_{m=1}^{M} \hat{f}_m(x)$$

- Reduces **variance**, models are **independent**, no interaction or adaptation between models

> Many weak models trained in parallel, then averaged to stabilize.

Boosting: GBM/XGBoost

- Models are trained *sequentially* to focus on the errors of the previous ones.
- At each step m, we fit:

$$\hat{f}_m(x) = \arg\min_f \sum_{i=1}^{n} L\left(y_i, \hat{f}_{m-1}(x_i) + f(x_i)\right)$$

$$\hat{f}_{\text{boost}}(x) = \sum_{m=1}^{M} \gamma_m \hat{f}_m(x)$$

- Reduces **bias**, learners **adapt** to previous errors, can overfit if not regularized

> Each new model corrects the last - like a student revising mistakes

Random Forest

1. For $m = 1$ to M (number of trees):
   - Draw a **bootstrap sample** from the training data.
   - Grow a decision tree $\hat{f}_m(x)$:
     - At each split, randomly select m_{try} predictors.
     - Choose the best split from those.
     - Grow the tree **fully** (no pruning).
2. Final prediction:
   - **Regression**: average predictions

$$\hat{f}_{\text{RF}}(x) = \frac{1}{M} \sum_{m=1}^{M} \hat{f}_m(x)$$

   - **Classification**: majority vote

⚙️ Tuning & Evaluation

Key hyperparameters:

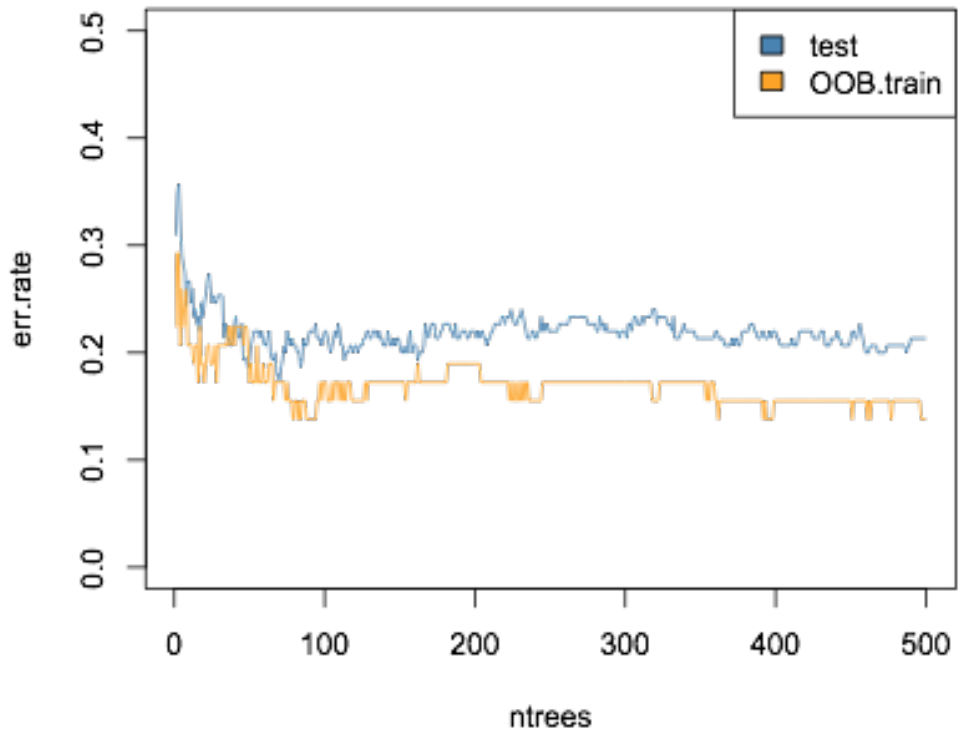
- M: number of trees
- m_{try}: number of predictors sampled at each split
- **Node size** / **max depth**

Out-of-Bag (OOB) Error:

- Each tree is built on a bootstrap sample ($\approx$ 63% of data).
- The remaining ~37% is OOB data - used as a **built-in validation set**.
- Compute prediction error on OOB samples.

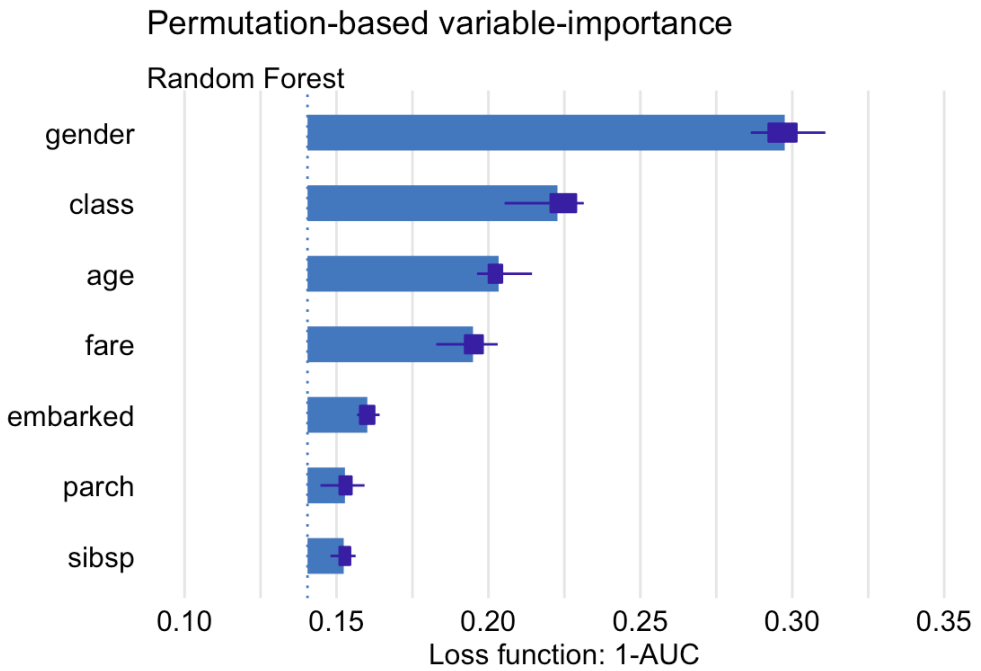
Random Forest (OOB)

- OOB error vs number of trees → check convergence



Random Forest (Variable Importance)

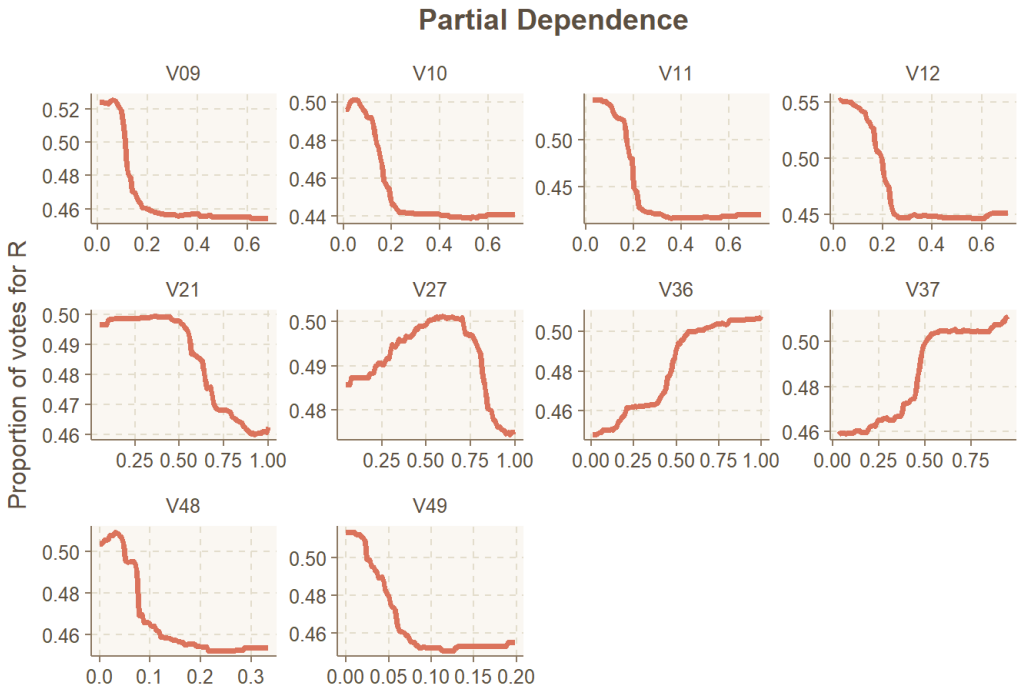
Variable importance plots



https://ema.drwhy.ai/featureImportance.html

Random Forest (Dependence Plots)

There is a whole science in "Explainable ML" or XAI, but for now, only focus on: Partial dependence plots (for interpretation). Go here https://christophm.github.io/interpretable-ml-book/overview.html#agnostic



https://sethdobson.netlify.app/2019/08/08/in-search-of-the-perfect-partial-plot/

Gradient Boosting Model

- Gradient descent is an algorithm to **minimize loss functions**. It works by **taking steps in the direction of the negative gradient** of the loss:

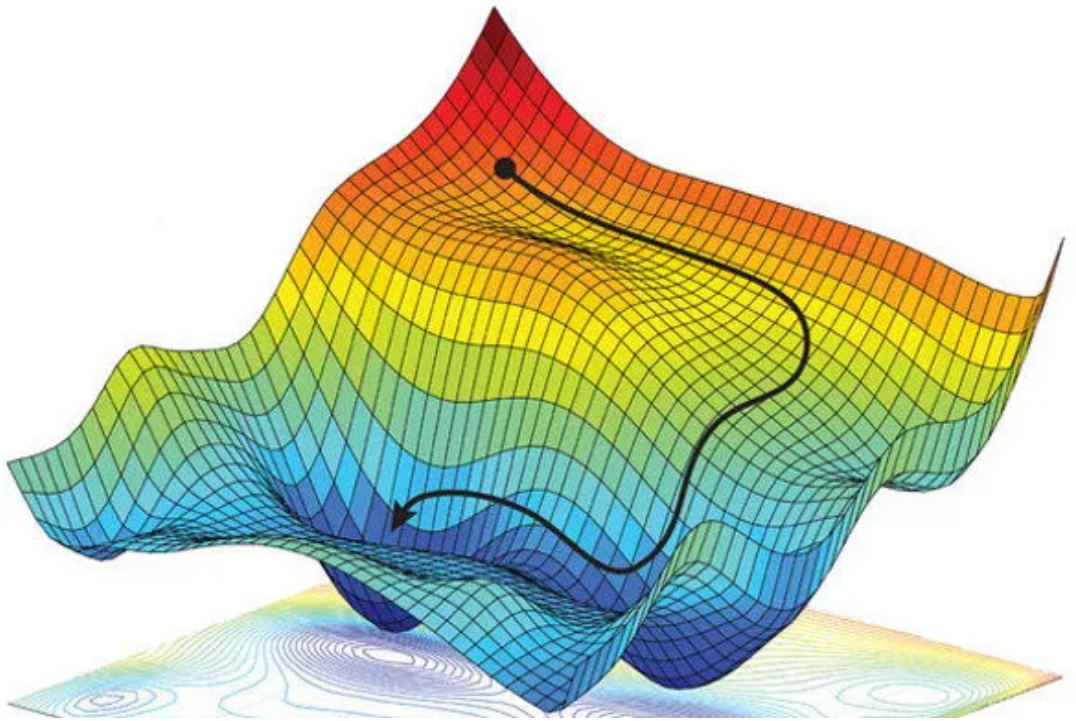
$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta L(\theta^{(t)})$$

- In boosting, we apply this idea **functionally**: each learner is a step in gradient space!

We take the derivative of the loss with respect to the prediction $\hat{y}$ (with $y = 3$ and $\hat{y} = 2$):

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}(y - \hat{y})^2 = -2(y - \hat{y})$$

$$\frac{\partial L}{\partial \hat{y}} = -2(3 - 2) = -2$$



- The gradient is -2. That means if you increase your prediction, the loss will go down. The model is underpredicting - the gradient tells you how to shift it.

Gradient Boosting Model

- Gradient descent is an algorithm to **minimize loss functions**.
- It works by **taking steps in the direction of the negative gradient** of the loss:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta L(\theta^{(t)})$$

- In boosting, we apply this idea **functionally**: each learner is a step in gradient space!

GBMs build an additive model:

$$\hat{f}(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

where each $h_m(x)$ is fit to the **negative gradient** of the loss function at step m.

⚙️ Tuning & Evaluation

- You can use **any differentiable loss function**:
  - Squared error (regression)
  - Log loss (classification)
  - Huber loss (robust regression)

Key hyperparameters:

- M: number of boosting iterations (trees)
- η: learning rate (step size for each update)
- $\mathbf{max\ depth}$: depth of each tree (controls complexity)
- $\mathbf{subsample}$: fraction of data used for each boosting round

> GBMs are very flexible but sensitive to tuning. Small learning rate + more trees = often better generalization.

Extreme Gradient Boosting Model

- XGBoost is an efficient, regularized implementation of gradient boosting. It uses **second-order optimization**: both **gradients** and **Hessians** (curvature). At each boosting step, it fits a tree to the **negative gradient**:

$$g_i = \frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}, \quad h_i = \frac{\partial^2 L(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}$$

- Each tree is chosen to **minimize a regularized objective**:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2] + \Omega(f_t)$$

where $\Omega(f_t)$ penalizes tree complexity (depth, leaf weight, etc.)

Extreme Gradient Boosting Model

⚙️ Tuning & Features

- XGBoost supports all GBM hyperparameters, plus:

Additional hyperparameters:

Parameter	Helps Control	Risk if Too Low	Risk if Too High
`lambda`	Overfitting	Underfit	May oversmooth
`gamma`	Complexity	Noisy, deep trees	No splits, underfit
`subsample`	Variance	Overfit	Underutilized data
`colsample_bytree`	Tree diversity	Overfit	Misses key features

But actually there is a reason why it has **EXTREME** in the names: https://xgboost.readthedocs.io/en/stable/parameter.html

Support Vector Machines (SVM)

SVMs are classifiers that find the **maximum-margin hyperplane** to separate classes in feature space. The goal: Maximize the distance between the decision boundary and the nearest points from each class. The decision function:

$$f(x) = \mathbf{w}^\top x + b$$

is chosen such that the margin is maximized and misclassifications are minimized.

- With **kernel tricks**, SVMs can separate non-linear data by implicitly mapping inputs into higher-dimensional space.

⚙️ Tuning & Evaluation

Key hyperparameters:

- `C`: penalty for misclassification
  - Small C → wider margin, more tolerance for misclassification
  - Large C → tight margin, low tolerance
- `kernel`: transformation function (linear, RBF, poly). Enables non-linear separation
- `gamma`: controls how far influence of a point reaches (in RBF kernel)
  - Low γ → smoother boundary
  - High γ → tightly fit to training data

Tuning Random Forests via Bayesian Optimization

- Random Forests have several important hyperparameters, but two main ones are:
  - Number of features at each split (`mtry`)
  - Minimum node size / max depth (`min_n`)
- Instead of using grid or random search, we can apply **Bayesian Optimization** to **efficiently search** for the best settings. Bayesian Optimization builds a model of *performance vs. parameter settings**, and uses it to select promising values.
- This is especially useful when:
  - Evaluating the model is **costly** (e.g., cross-validation)
  - The search space is **continuous or mixed**

⚙️ Search Space (RF)

- `mtry`: integer between 1 and p (number of features)
- `min_samples_leaf`: integer (e.g., 1–20)

✅ Bayesian Optimizer Flow

1. Evaluate a few random points
2. Fit a **surrogate model** of the performance surface
3. Use an **acquisition function** to pick next point
4. Iterate until convergence

> Efficiently balances exploration and exploitation. Often outperforms grid search with far fewer evaluations.

Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Practical 💻

Tidymodels

```
library(tidymodels)

penguins ← penguins %>% drop_na()

set.seed(123)
penguin_split ← initial_split(penguins, strata = species)
penguin_train ← training(penguin_split)
penguin_folds ← vfold_cv(penguin_train, v = 5)
```

- Feature engineering

```
penguin_rec ← recipe(species ~ ., data = penguin_train) %>%
  step_normalize(all_numeric_predictors())
```

- Model

```
penguin_model ← rand_forest( mtry = tune(), min_n = tune(), trees = 100) %>% set_engine("ranger") %>%
  set_mode("classification")
```

Tidymodels: Classic tuning

- Workflow

```
penguin_wf ← workflow() %>%
  add_model(penguin_model) %>%
  add_recipe(penguin_rec)

param_final ←
  extract_parameter_set_dials(penguin_model) %>%
  finalize(penguin_train)
```

- Normal tuning

```
set.seed(123)
grid_vals ← grid_regular(param_final, levels = 5)  # or grid_random() for random search
penguin_grid ← tune_grid(
  penguin_wf,
  resamples = penguin_folds,
  grid = grid_vals,
  metrics = metric_set(accuracy),
  control = control_grid(save_pred = TRUE)
)
```

Tidymodels: Bayesian tuning

```
penguin_bayes ← tune_bayes(
  penguin_wf,
  resamples = penguin_folds,
  param_info = param_final,
  metrics = metric_set(accuracy),
  initial = 5,
  iter = 20,
  control = control_bayes(verbose = TRUE,
                          no_improve = 5)
)
```

```
penguin_grid %>% show_best(metric = "accuracy", n = 1)
# mtry min_n .metric  .estimator  mean     n std_err .cor
# <int> <int> <chr>    <chr>      <dbl> <int>   <dbl> <ch
# 1     1     2 accuracy multiclass 0.992     5 0.00495 P

penguin_bayes %>% show_best(metric = "accuracy", n = 1)
# mtry min_n .metric  .estimator  mean     n std_err .cor
# <int> <int> <chr>    <chr>      <dbl> <int>   <dbl> <ch
# 1     1    11 accuracy multiclass 0.992     5 0.00495 P
```

Tidymodels

```
final_model ← penguin_bayes %>% select_best(metric = "accuracy") %>% finalize_workflow(penguin_wf, .) %>%
    fit(data = penguin_train)
```

```
# == Workflow [trained] ==========================================================================================
# Preprocessor: Recipe
# Model: rand_forest()
#
# -- Preprocessor ----------------------------------------------------------------------------------------------
# 1 Recipe Step
#
# • step_normalize()
#
# -- Model -----------------------------------------------------------------------------------------------------
# Ranger result
#
# Call:
#  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~1L,      x), num.trees = ~100, min.node.size = min_rows(~11L, x),      num.threads = 1
#
# Type:                             Probability estimation
# Number of trees:                  100
# Sample size:                      249
# Number of independent variables:  6
# Mtry:                             1
# Target node size:                 11
# Variable importance mode:         none
# Splitrule:                        gini
# OOB prediction error (Brier s.):  0.037959
```

Tidymodels: Variable Importance

```
final_model %>% extract_fit_parsnip()
final_model %>%
  extract_fit_parsnip() %>%
  pluck("fit")

imp_tbl ← as.data.frame(rf$importance) %>%
  rownames_to_column("variable")

ggplot(imp_tbl, aes(x = reorder(variable, MeanDecre
  geom_col(fill = "steelblue") +
  coord_flip() +
  labs(
    title = "Variable Importance (Gini)",
    x = "Variable",
    y = "Mean Decrease in Gini"
  ) +
  theme_minimal()
```

```
class_imp ← imp_tbl %>%
  pivot_longer(cols = c(Adelie, Chinstrap, Gentoo), names

ggplot(class_imp, aes(x = reorder(variable, importance),
  geom_col(fill = "darkgreen") +
  coord_flip() +
  facet_wrap(~ class, scales = "free_y") +
  labs(
    title = "Class-Specific Variable Importance",
    x = "Variable",
    y = "Importance"
  ) +
  theme_minimal()
```

Tidymodels: Xbgoost

```
penguin_model ← boost_tree(
  trees = 100,               # number of boosting iterations
  tree_depth = tune(),       # max depth of a tree
  learn_rate = tune(),       # shrinkage (eta)
  loss_reduction = tune(),   # min split gain (gamma)
  sample_size = tune(),      # row subsampling
  mtry = tune()              # colsample_bytree
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

penguin_rec ← recipe(species ~ ., data = penguin_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_numeric_predictors())

penguin_wf ← workflow() %>%
  add_model(penguin_model) %>%
  add_recipe(penguin_rec)

param_final ←
  extract_parameter_set_dials(penguin_model) %>%
  finalize(penguin_train)
```

Tidymodels: Xbgoost

- Bayesian Tuning

```
penguin_bayes ← tune_bayes(
  penguin_wf,
  resamples = penguin_folds,
  param_info = param_final,
  metrics = metric_set(accuracy),
  initial = 5,
  iter = 20,
  control = control_bayes(verbose = TRUE,
                          no_improve = 5)
)
```

- Estimate model

```
final_model ← penguin_bayes %>% select_best(metric = "accuracy") %>% finalize_workflow(penguin_wf, .) %>%
    fit(data = penguin_train)
```