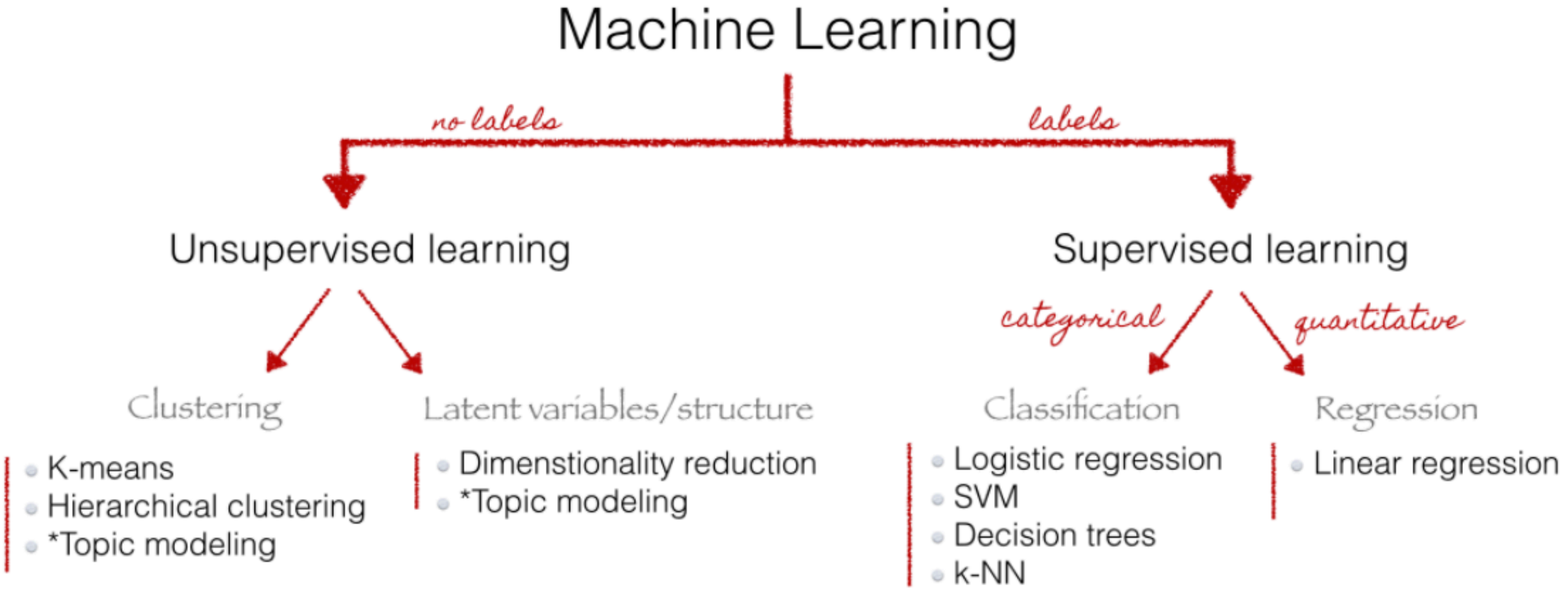
Supervised Learning

UC3M/IC3JM Graduate Workshop, Fall 2024

Patrick Kraft

2024-12-05

Introduction



Source: Christine Doig 2015

Universal machine learning workflow (Chollet and Allaire 2018)

1. **Define the problem at hand and the data on which you'll be training.**
   Collect this data, or annotate it with labels if need be.

2. **Choose how you'll measure success on your problem.**
   Which metrics will you monitor on your validation data?

3. **Determine your evaluation protocol.**
   Which portion of the data should you use for validation? K-fold validation?

4. **Preparing/preprocess your data**

5. **Start by developing a first model that does better than a basic baseline.**
   Then revise the model to the point that it overfits

6. **Regularize your model and tune its hyperparameters.**
   Evaluate performance based on the validation data.

7. **Final training (on all training + validation data) and model testing on unseen test dataset**

Accuracy in Classification Tasks (James et al. 2013, ch. 4.4.3)

		Predicted class		
		− / Null / 0	+ / Non-null / 1	Total
True class	− / Null / 0	True Neg. (TN)	False Pos. (FP)	N
	+ / Non-null / 1	False Neg. (FN)	True Pos. (TP)	P
	Total	N*	P*	

Name	Definition	Synonyms
False Pos. rate	FP/N	Type I error, 1−Specificity
True Pos. rate	TP/P	1−Type II error, power, sensitivity, recall
Pos. Pred. value	TP/P*	Precision, 1−false discovery proportion
Neg. Pred. value	TN/N*	

TABLE 4.7. *Important measures for classification and diagnostic testing, derived from quantities in Table 4.6.*

- **Global performance metrics:**
  - Accuracy = $\frac{TP+TN}{TP+FP+TN+FN}$
  - Error Rate = 1 − Accuracy)
  - No Information rate = % modal category
- **Row / column performance metrics:**
  - Specificity = $\frac{TN}{TN+FP}$
  - Sensitivity/Recall = $\frac{TP}{TP+FN}$
  - Precision = $\frac{TP}{TP+FP}$

Accuracy in Classification Tasks (James et al. 2013, ch. 4.4.3)

- **Specificity**: How many of the actual negative cases are correctly identified?
  - Interpretation: Specificity matters when it is crucial to correctly identify negative cases, such as in a test where false alarms are costly or burdensome, like unnecessary medical tests.

- **Sensitivity/Recall**: How many of the actual positive cases are correctly identified?
  - Interpretation: Recall is critical when missing a true positive has significant consequences, such as in medical diagnostics where you don't want to miss a disease diagnosis.

- **Precision**: How many of the predicted positive results are truly positive?
  - Interpretation: Precision is important when the cost of false positives is high, for example, in spam email detection, where you want to avoid marking a legitimate email as spam.

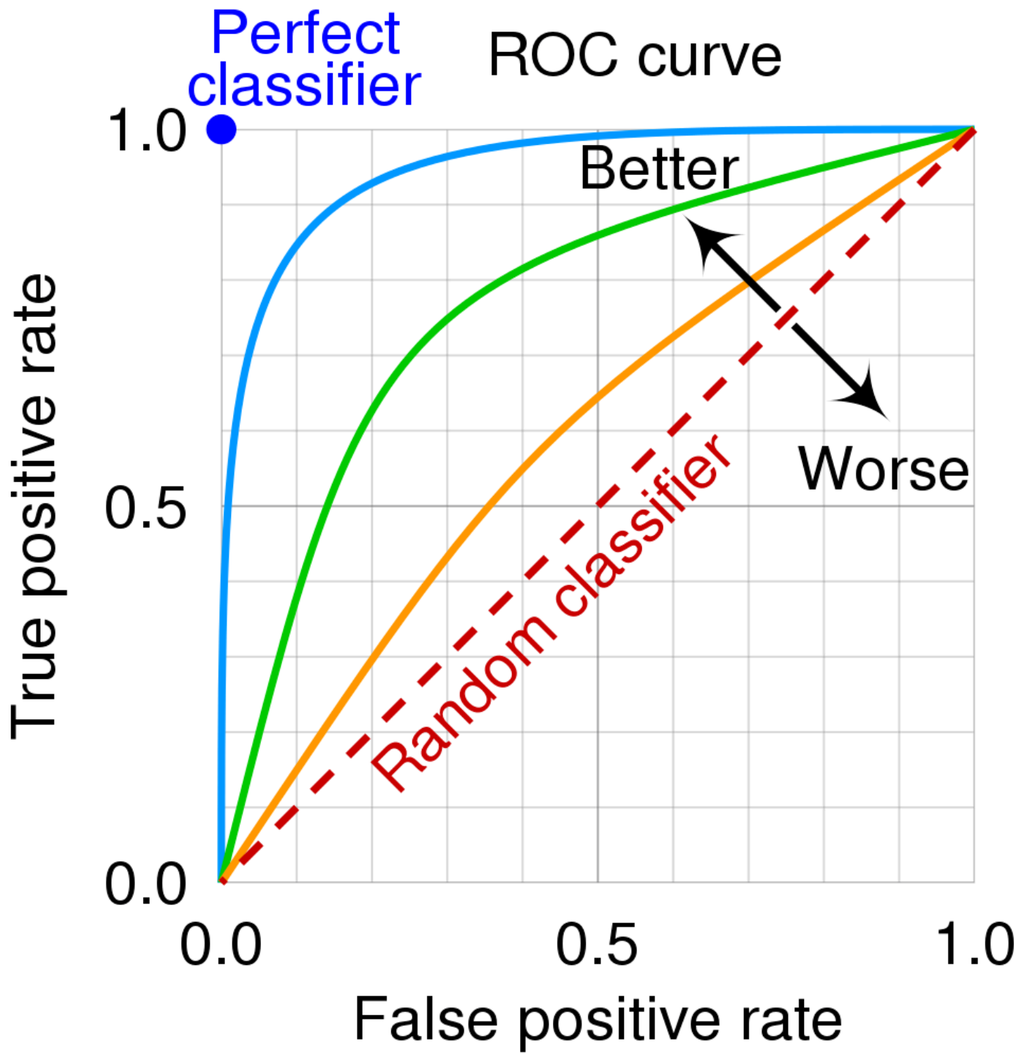
Accuracy in Classification Tasks (James et al. 2013, ch. 4.4.3)

- **F1-Score**: Combining both **precision** and **recall** into single score
  - $F1 = 2 \times \frac{precision \times recall}{precision + recall}$
  - Highest possible value of an F-score is 1.0, indicating perfect precision and recall, and the lowest possible value is 0, if either precision or recall are zero

- Why?
  - Useful when caring about both reducing false positives (high precision) and false negatives (high recall)
  - **Handles class imbalance better**: considers both false positives and false negatives, providing a better sense of model's performance with respect to the minority class
  - Provides **single metric** for comparison

Error rates and ROC curve

- **Problem**: error rates depend on the classification threshold
- The **ROC curve** displays two types of errors (false positives, true positives rate) for all possible thresholds
  - FPR = 1 - specificity
  - TPR = sensitivity/recall
- The **Area under the (ROC) curve (AUC)** provides the overall performance of a classifier, summarized over **all possible thresholds**



Outline for Today

1. Preprocessing data: recipes & workflows
2. Resampling & cross-validation
3. Feature selection & regularization
4. Tree-based methods
5. Tuning models

1. Preprocessing data: recipes & workflows

Preprocessing data & feature engineering

- **Preprocessing** = transforming raw data into a format that can be used by the model
  - Data cleaning, feature selection, data normalization (e.g., scale/transform variables to similar ranges, etc.)

- **Feature engeneering**: process of creating new features or transforming existing features to improve model's performance and interpretability
  - **Goal**: create features that better capture underlying patterns and relationships in the data to provide more accurate/robust predictions (e.g, **encoding** of text to numbers)

- Tidymodel's **recipes** provide *pipeable steps for feature engineering and data preprocessing to prepare for modeling*

- **Different preprocessing steps** may be necessary for **different datasets, outcomes and models**!

Feature engineering steps

Why preprocessing data?

- Sources: KDNuggest; García, Luengo, and Herrera (2015)
- **Poor model performance**: Algorithm may not understand badly cleaned/incorrectly formatted data (missings, outliers, irrelevant data) which can lead to poor model performance
- **Overfitting**: Non-cleaned data contain irrelevant or redundant information that can lead to overfitting (i.e., model adapts to much to training data and performs badly in unseen test data)
- **Longer training times**: Algorithm may be faster if data is clean (has to process less data itself, e.g., doesn't have to create all those dummies)
- **Interpretability of model**: Preprocessed data may make model more interpretable (e.g., think of variables in linear regression model)
- **Biased results**: Non-preprocessed data may contain errors or biases that can lead to unfair or inaccurate results (e.g., again outliers etc.)

Tidymodels: Use `recipes` for preprocessing data

- Recipes require definitions of… (see here)
  - **variables**: the original (raw) data columns in a data frame
  - **roles**: how variables are used in the model
  - **terms**: synonymous with features (partly happens automatically)

- `recipe()`: function to create a recipe for preprocessing data (see here for an overview)
  - `step_*()`: functions for different preprocessing steps (see here for an overview)
  - `update_role(respondent_id, new_role = "ID")`

Overview of some `step()` functions

- `step_normalize()`: Scales numeric data to have mean zero and a standard deviation of one.
- `step_center()`: Centers numeric data to have mean zero.
- `step_poly()`: create new columns that are basis expansions of variables using orthogonal polynomials.
- `step_scale()`: Scales numeric data to have unit variance.
- `step_log()`: Applies a natural logarithm transformation to numeric data. This can be useful for handling skewed data.
- `step_sqrt()`: Applies a square root transformation to numeric data, which can also help with skewed distributions.
- `step_boxcox()`: Transforms numeric data using Box-Cox transformation to make data more normal-like. This step requires positive values.
- `step_inverse()`: Applies an inverse transformation to numeric data, potentially useful for heavy-tailed distributions.
- `step_dummy()`: Converts categorical variables into dummy/one-hot encoded variables.

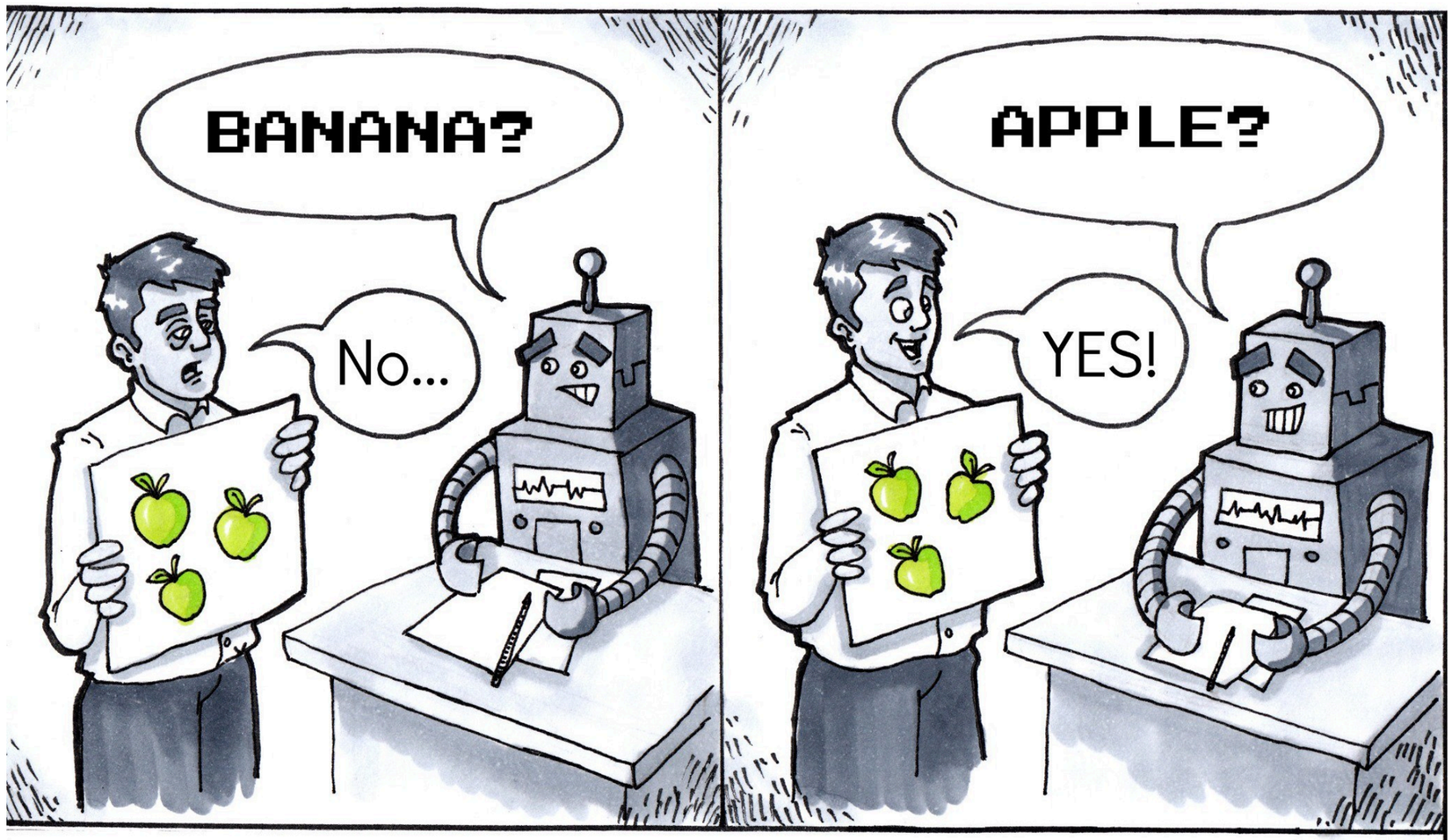
Tidymodels: Use `workflows` for modeling pipelines

- Workflows are used to specify end-to-end modeling pipelines, including preprocessing and model fitting.

- **Key Functions**
  - `workflow()`: Create a workflow object.
  - `add_recipe()`: Add a recipe to the workflow.
  - `add_model()`: Add a model to the workflow.
  - `fit()`: Train the model(s) in the workflow.
  - `predict()`: Generate predictions using the fitted model.

- **Advantage**: We can define different workflows (based on the same or different recipes/models) and compare them with each other.

R Session



2. Resampling and Cross-Validation

Validation Set Approach (training vs. validation vs. test data)

To avoid overfitting we can estimate our model based on a subsample and evaluate performance based on a validation set

k-Fold Cross-Validation

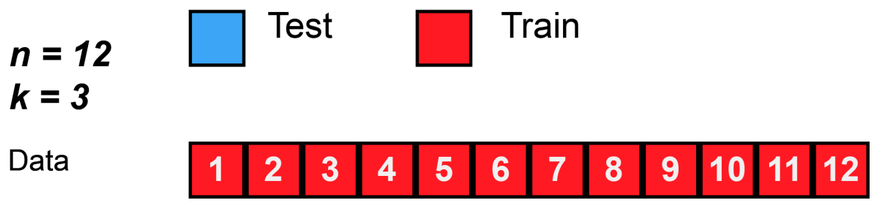


Figure 1: K-fold cross-validation: With n = 12 observations and k = 3 folds, data will be shuffled and a total of 3 models trained and tested (Source: MBanuelos22/Wikipedia (CC BY-SA 4.0))

- Data set split into k parts. Each of the k folds is used for assessment (validation dataset) once, while the remaining $k - 1$ parts are used for the analysis (training dataset). Estimation is repeated k times.
- Error rate: $CV_{(k)} = \frac{1}{k} \sum_{i=1}^{k} Err_i$
  - *Advantages*
    - Offers a balance between bias and variance.
    - Efficient use of data, with all observations used for both training and validation.
    - More computationally efficient than LOOCV.
  - *Disadvantages*
    - The choice of (k) can affect performance estimates.
    - Variability in performance estimates due to randomness in splits.

Leave-one-out cross-validation (LOOCV)

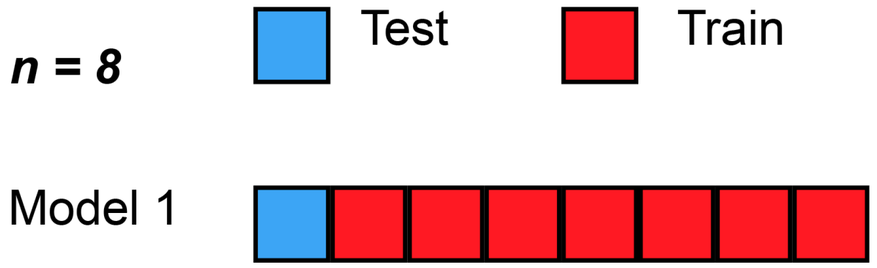


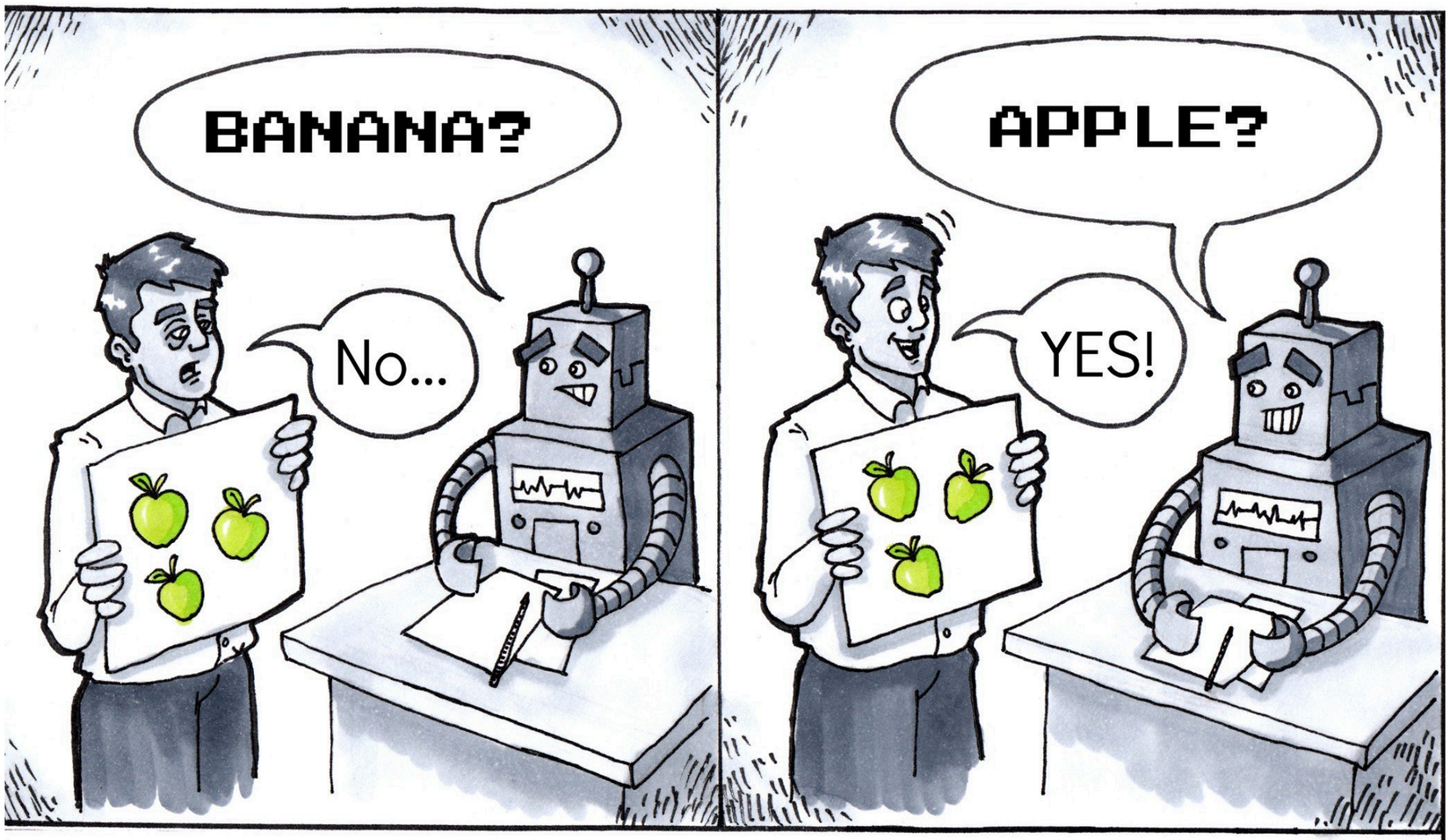
Figure 2: Leave-one-out cross-validation (LOOCV): With n = 8 observations a total of 8 models will be trained and tested (Source: MBanuelos22/Wikipedia (CC BY-SA 4.0))

- LOOCV is a special case of k-Fold Cross-Validation where k is set to equal n
- *Advantages*
  - Low bias as almost all data is used for training.
  - Every observation is used for both training and validation, ensuring a thorough assessment.
  - Reproducibility: Performing LOOCV repeatedly will always produces the same results
- *Disadvantages*
  - Computationally intensive for large datasets.
  - Can have high variance in performance estimates.

Tidymodels: Resampling

- `rsample`: for sample splitting (e.g. train/test or cross-validation)
  - provides functions to create different types of resamples and corresponding classes for their analysis
- `initial_split`: Use this to split the data into training and test data (with arguments `prop`, `strata`)
  - `prop`-argument: Specify share of training data observations
  - `strata`-argument: Conduct stratified sampling on the dependent variable (better if classes are imbalanced!)
- `initial_validation_split(data, prop = c(0.6, 0.2))`: Split into training (60%), validation (20%), and test data (20%)
- `vfold_cv(data_train, v = 10)`: randomly splits the data into $v = k$ groups of roughly equal size (called "folds")
  - Use after `initial_split`, i.e., split into training/test data
  - `fit_resamples()` (`tune` package): computes a set of performance metrics across one or more resamples

R Session



3. Feature selection & regularization

Concepts

- **Feature selection**
  - also known as **variable selection**, **attribute selection** or **variable subset selection**, is the process of selecting a subset of relevant features (variables, predictors) for use in model construction (Wikipedia)
- **Regularization**
  - = either choice of the model or modifications to the algorithm
  - Penalizing the complexity of a model to reduce overfitting
- **Theory vs. algorithm feature selection**
  - ML largely focuses on automated selection, albeit theory may play an important role

Methods for feature/variable selection

- **Shrinkage**: involves fitting a model involving all p predictors but shrink coefficients to zero relative to the least squares estimates
  - This shrinkage (also known as **regularization**) has the effect of reducing variance
  - Depending on what type of shrinkage is performed, some coefficients may be estimated to be exactly zero,i.e., shrinkage methods can perform variable selection
  - e.g., ridge regression (James et al. 2013, Ch. 6.2.1), the lasso (James et al. 2013, Ch. 6.2.2)

Ridge regression (RR)

- **Least squares (LS) estimator**: least squares fitting procedure estimates $\beta_0, \beta_1, \ldots, \beta_p$ using the values that minimize RSS

- **Ridge regression (RR)**: seeks coefficient estimates that fit data well (minimize RSS) + *shrinkage penalty* $\lambda \sum_{j=1}^{p} \beta_j^2$
  - **Tuning parameter** λ serves to control the relative impact of these two terms (RSS and shrinking penalty) on the regression coefficient estimates
    - When $\lambda = 0$ the penalty term has no effect and ridge regression will produce least squares estimates
    - As $\lambda \rightarrow \infty$, impact of the shrinkage penalty grows, and the ridge regression coefficient estimates will approach zero
    - Penality is only applied to coefficients not the intercept

- Ridge regression will produce a different set of coefficient estimates $\hat{\beta}_\lambda^R$ for each λ
  - **Hyperparameter tuning**: fit many models with different λs to find the model the performs “best”

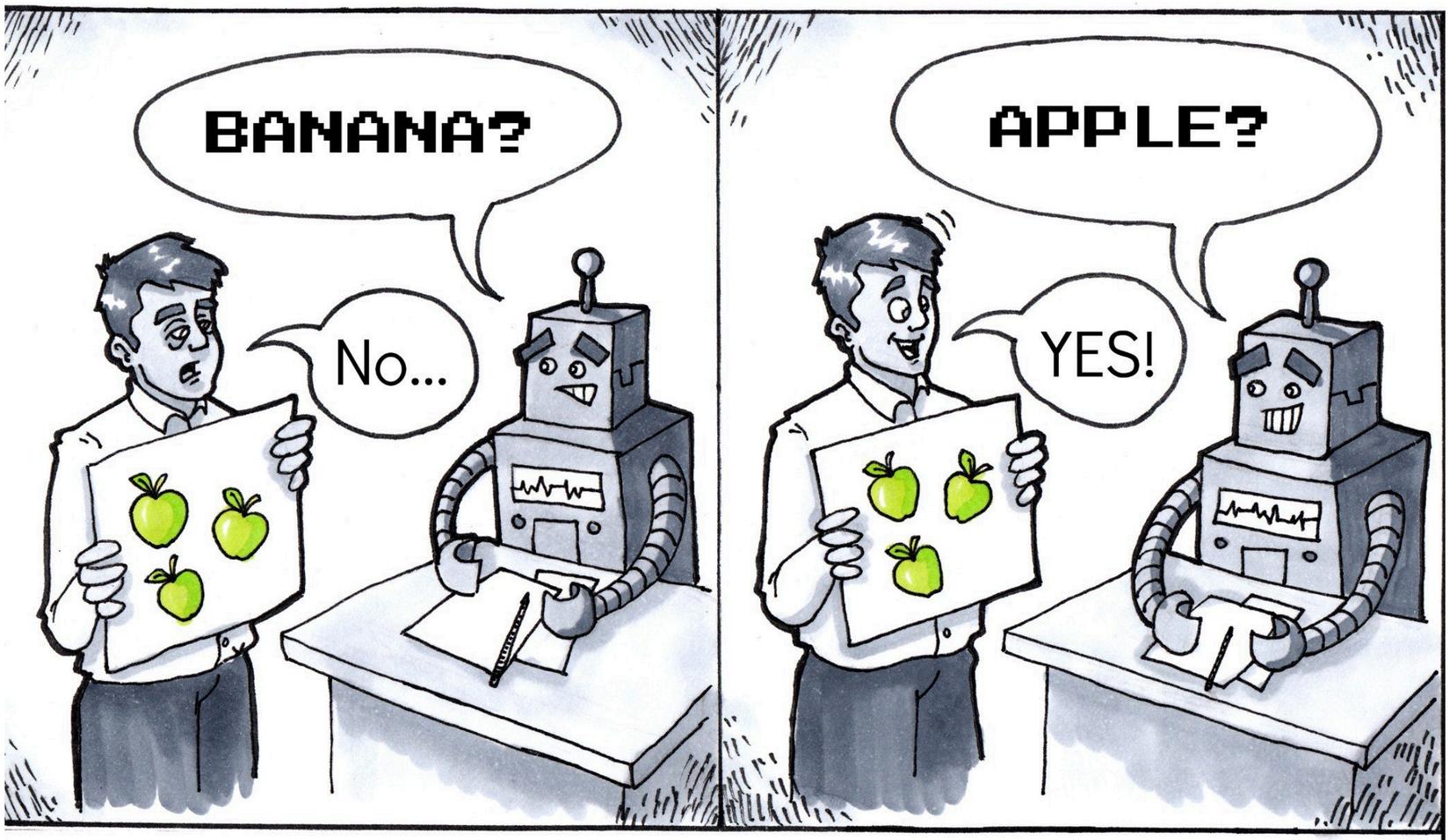
The Lasso

- **Ridge regression (RR)** has one obvious disadvantage
  - Unlike subset selection methods (best subset, forward stepwise, and backward stepwise selection) that select models that involve a variable subset, RR will include all p predictors in final model
  - Penalty $\lambda \sum_{j=1}^{p} \beta_j^2$ will shrink all coefficients towards zero, but no set them exactly to 0 (unless $\lambda = \infty$)
  - No problem for accuracy, but challenging for model interpretation when p is large

Comparing the Lasso and Ridge Regression

- **Advantage of Lasso**: produces simpler and more interpretable models involving subset of predictors
- However, which method leads to better prediction accuracy?
  - Neither ridge regression nor lasso universally dominates the other
  - Lasso works better when few predictors have significant coefficients, others have small/zero coefficients
  - Ridge regression performs better with many predictors of roughly equal coefficient size
  - Cross-validation helps determine the better approach for a specific data set
  - Both methods can reduce variance at the cost of slight bias increase, improving prediction accuracy

R Session

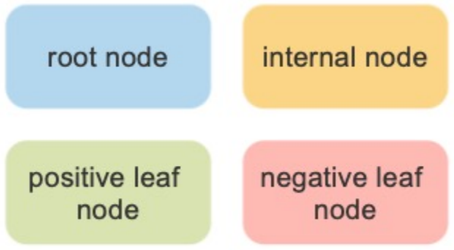


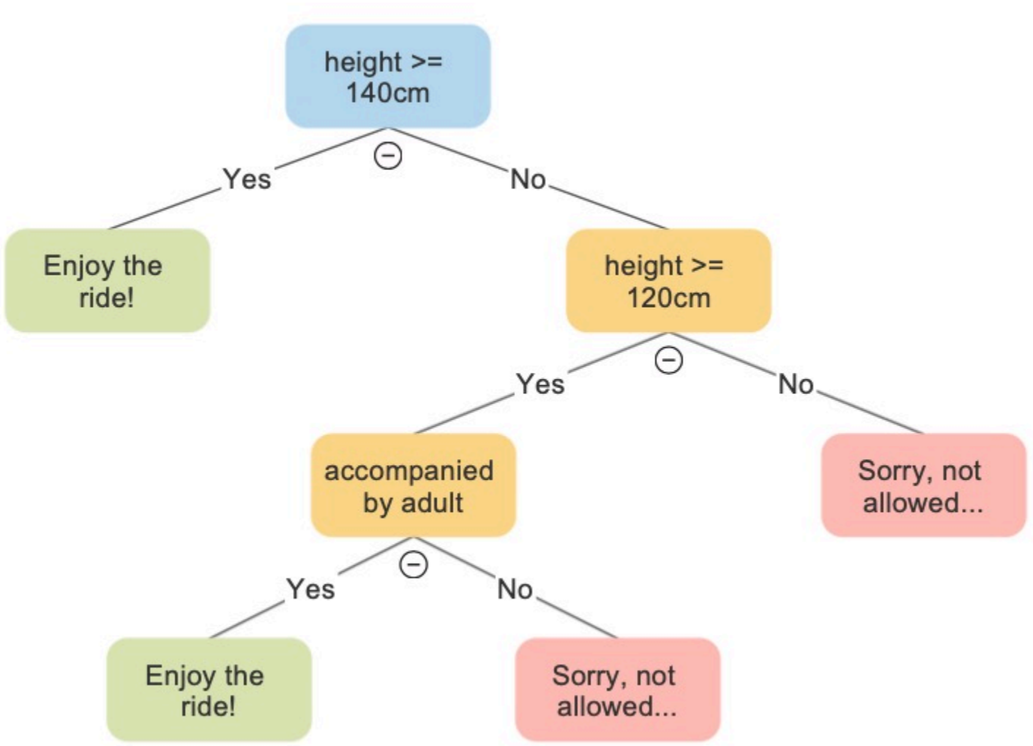
4. Tree-based methods

Tree structure and terminology

- Top of the tree contains all available training observations: **root node**
- **Partition** the data into homogeneous non-overlapping subgroups: **nodes**
- Subgroups formed via **simple yes-no questions**

- Tree predicts the output in a **leaf node** as follows:
  - average of the response for regression
  - majority voting for classification
- Different types of nodes:





Advantages and Disadvantages of (single) Trees

- Pro (James et al. 2013, Ch. 8.1.4)
  - Trees are very **easy to explain** to people (even easier than linear regression)!
  - Decision trees more closely **mirror human decision-making** than do other regression and classification approaches
  - Trees can be displayed graphically, and are easily **interpreted even by a non-expert** (especially if they are small).
  - Trees can easily **handle qualitative predictors** without the need to create dummy variables.
- Contra (James et al. 2013, Ch. 8.1.4)
  - Trees generally do not have the **same level of predictive accuracy as** some **other** regression and classification **approaches**
  - Trees can be **very non-robust**: a small change in the data can cause a large change in the final estimated tree.

Bagging

- Single decision trees may suffer from *high variance*, i.e., results could be very different for different splits
  - Q: Why?
- We want: *Low variance* procedure that yields similar results if applied repeatedly to distinct training datasets (e.g., linear regression)
- **Bagging**: construct B regression trees using B bootstrapped training sets, and average the resulting predictions
  - **Bootstrap** (also called *bootstrap aggregation*): Take repeated samples from the (single) training set, built predictive model and average predictions
  - **For Classification Problem**: For a given test observation, we can record the class predicted by each of the B trees, and take a majority vote: the overall prediction is the most commonly occurring majority vote class among the B predictions
  - **For Regression Problem**: Average across B predictions

Random forests

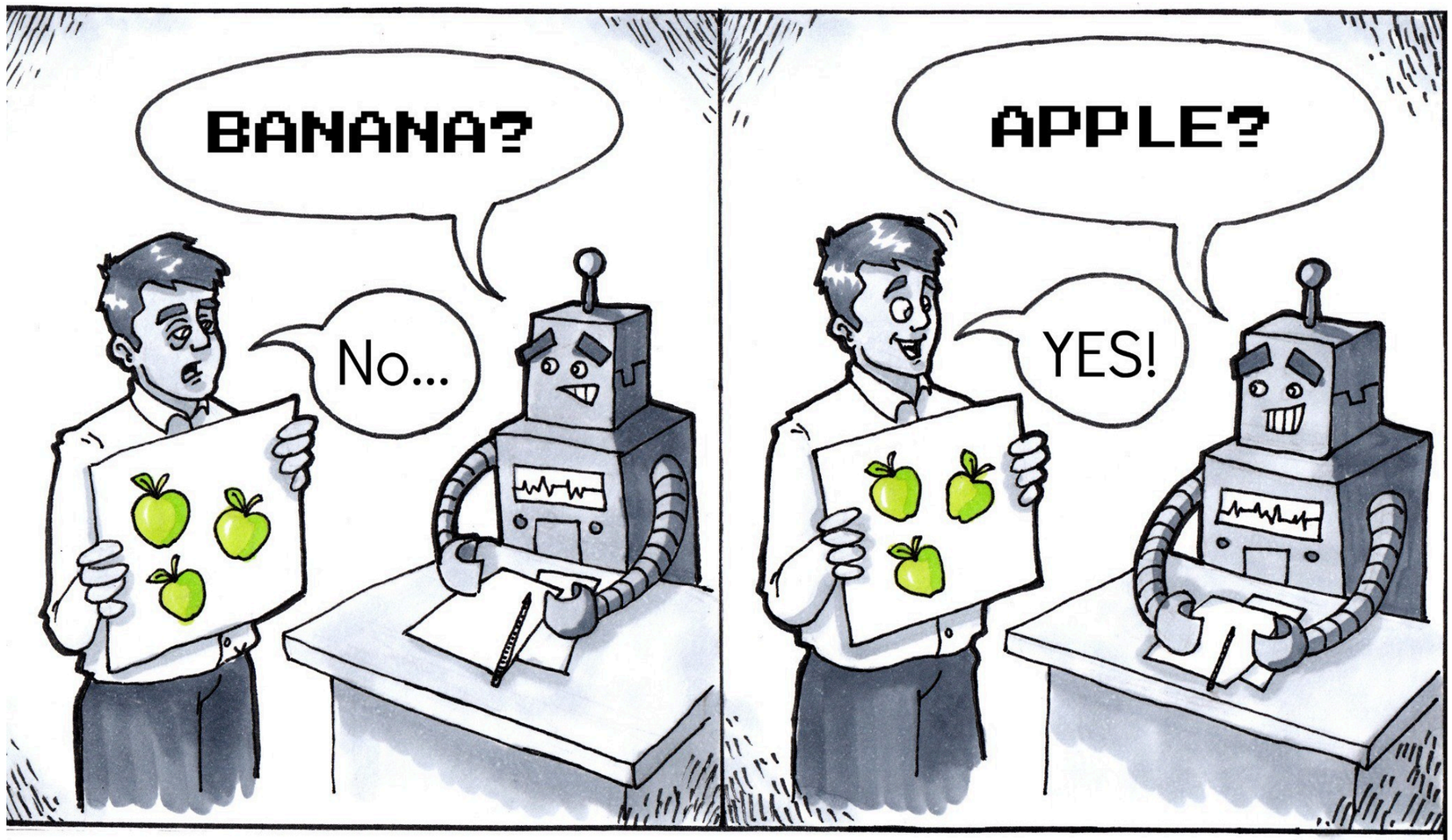
Boosting Trees: GBDT & XGBoost

- **Gradient Boosted Decision Trees (GBDT)** and **XGBoost (eXtreme Gradient Boosting)**:
  - popular machine learning algorithms that use technique called **boosting**
  - **trees built sequentially**: each new tree aims to correct errors made by previously built trees
    - model learns by fitting new predictor to residual errors made by preceding predictors and not the actual target
    - **focuses on reducing both bias and variance** by building a series of trees where each tree corrects mistakes of previous one

Key Differences between Boostind and Random Forests

- **Model Building**:
  - RF **builds trees independently and combines their predictions**, while GBDT **builds trees sequentially** (each tree **learning from** mistakes of **previous ones**)
- **Objective**
  - RF aims to **reduce variance without increasing bias**, while GBDT aims to **reduce both bias and variance through iterative corrections**
- **Data Sampling**
  - In RF **each tree** is trained on **random subset of data**; GBDT trains **each tree** on **entire dataset** but **focuses** on **instances** that were **mispredicted by previous trees**
- **Feature Sampling**
  - RF considers **random subset of features for splitting at each node**; GBDT typically **considers all features** when building each tree **focusing** more on **feature importance**
- **Error Correction**
  - GBDT directly **focuses on correcting the errors (minimizing the loss function) from the previous trees**, while RF **combines diverse trees' predictions** to **average out the errors**

R Session



5. Tuning models

Why care about hyperparameters?

- Many machine learning models have **parameters** (learned during training) as well as **hyperparameters** (set before training).

- Hyperparameters determine what a model can learn and may therefore critically affect a model's **out-of-sample performance** (i.e., how well it generalizes to new, unseen data).

- Arnold et al. (2024:1) find that among 64 machine learning-related political science papers published between 1 January 2016 and 20 October 2021 in some of the top journals of our discipline—the American Political Science Review (APSR), Political Analysis (PA), and Political Science Research and Methods (PSRM)…
  - 36 publications (56.25 %) do not report the values of their hyper- parameters
  - 49 publications (76.56 %) do not share information their usage of tuning to find good hyperparameters
  - 13 publications (20.31 %) offer a complete account of hyperparameters/tuning

Steps in tuning (Arnold et al. 2024)

1. **Understanding the model**: How do the available hyperparameters affect the model?
2. **Choosing a performance measure**: e.g., MSE, MAE, F-1, …
3. **Defining a sensible search space**:
    - Useful starting points: hyperparameter default values in software libraries, recommendations from the literature, or own previous experience
4. **Finding the best combination in the search space**:
    - **Grid search**: try every possible combination of hyperparameters of the search space to find optimal combination
    - **Random search**: each run picks a different random set of hyperparameters from the search space
5. **Tuning under strong resource constraints**: If the model training is too involved, adaptive approaches such as sequential model-based Bayesian optimization allow for efficiently identifying and testing promising hyperparameter candidates.

- **Important!!!:** Describe whether/how hyperparameters were tuned and what final value were chosen in main body/appendix of paper

Models and their hyperparameters: Lasso & ridge regression

- See `?details_linear_reg_glmnet` for details for `glmnet` package
- These models have 2 tuning parameters
  - `penalty`: Amount of Regularization
    - The `penalty` parameter has no default and requires a single numeric value
  - `mixture`: Proportion of Lasso Penalty (default: 1)
    - A value of `mixture = 1` corresponds to a pure lasso model, while `mixture = 0` indicates ridge regression

Models and their hyperparameters: Random forest

- See `?details_rand_forest_ranger` for details for `ranger` package
- Can be used for both classification and regression
- `# = number of`
- Model has 8 tuning (hyper-)parameters
  - `mtry`: # Randomly Selected Predictors (default: depends on number of columns)
  - `trees`: # Trees (default: 500)
  - `min_n`: Minimal Node Size
    - `min_n` default depends on the mode. For regression, a value of 5 is the default. For classification, a value of 10 is used.

Models and their hyperparameters: XGBoost tuning parameters

- See `?details_boost_tree_xgboost` for details for `xgboost` package
- Can be used for both classification and regression
- `# = number of`
- Model has 8 tuning (hyper-)parameters
  - `tree_depth`: Tree depth (default: 6)[1]
  - `trees`: # trees (default: 15)[2]
  - `learn_rate`: Learning rate (default: 0.3)[3]
    - Picking lower value might be better but is slower
  - `mtry`: # randomly selected predictors among which splitting candidate is picked (default: see below)
  - `min_n`: Minimal node size before stopping splitting (default: 1)
  - `loss_reduction`: Minimum loss reduction (default: 0.0)[4]
  - `sample_size`: Proportion observations sampled (default: 1.0)[5]
  - `stop_iter`: # iterations before stopping (default: Inf)[6]

References

Arnold, Christian, Luka Biedebach, Andreas Küpfer, and Marcel Neunhoeffer. 2024. “The Role of Hyperparameters in Machine Learning Models and How to Tune Them.” *Political Science Research and Methods* 1–8.

Chollet, Francois, and J. J. Allaire. 2018. *Deep Learning with R*. 1st ed. Manning Publications.

García, Salvador, Julián Luengo, and Francisco Herrera. 2015. *Data Preprocessing in Data Mining*. Springer International Publishing.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning: With Applications in R*. Springer.

Neunhoeffer, Marcel, and Sebastian Sternberg. 2019. “How Cross-Validation Can Go Wrong and What to Do about It.” *Polit. Anal.* 27(1):101–6.

Footnotes

1.

By adjusting the “tree_depth” parameter, you control the complexity of each individual decision tree in the XGBoost ensemble. Higher values allow the tree to capture more complex patterns in the data but may also increase the risk of overfitting. Conversely, lower values restrict the complexity of the tree, potentially improving generalization but may lead to underfitting if set too low.

2.

`trees` determines the complexity and capacity of the model. More trees can potentially capture more complex patterns in the data but may also lead to overfitting if not controlled properly.

3.

`learn_rate` parameter, also known as the learning rate or eta in the context of XGBoost, is a crucial hyperparameter that influences the training process of gradient boosting algorithms, including XGBoost. The learning rate determines the step size or the rate at which the model updates its weights or parameters during the training process. It’s a scalar value that controls how much we are adjusting the weights of our network with respect to the loss gradient.

4.

The `loss_reduction` parameter plays a role in controlling the construction of individual decision trees during the boosting process. The `loss_reduction` parameter specifies the minimum amount of reduction in the loss function required to make a further partition on a leaf node of the decision tree. Essentially, it determines the threshold for splitting a node based on the improvement in the model’s performance.

5.

The `sample_size` parameter specifies the proportion of observations sampled for each tree during the training process. Gradient boosting algorithms, including XGBoost, often use a technique called “bootstrapping” or “bagging” to train individual trees. In each iteration of training, a random subset of observations is sampled with replacement from the original dataset.

6.

The `stop_iter` parameter is related to early stopping during the model training process. Early stopping is a technique used during