

Hands-on Machine Learning with Python - Module 2

Hands-on webinar

Katrien Antonio & Jonas Crevecoeur & Roel Henckaerts

[hands-on-machine-learning-Python-module-2](#) | March, 2023

Prologue

Introduction

Course

 <https://github.com/katrienantonio/hands-on-machine-learning-Python-module-2>

The course repo on GitHub, where you can find the data sets, lecture sheets, Google Colab links and Python notebooks.


Us

 <https://katrienantonio.github.io/> & LinkedIn profile Jonas & LinkedIn profile Roel

 katrien.antonio@kuleuven.be & jonas.crevecoeur@kuleuven.be & roel.henckaerts@kuleuven.be

 (Katrien) Professor in insurance data science

 (Jonas) PhD in insurance data science, now data science consultant at UHasselt and KULeuven

 (Roel) PhD in insurance data science, now senior data scientist at [Prophecy Labs](#)

Why this course?

The goals of this course

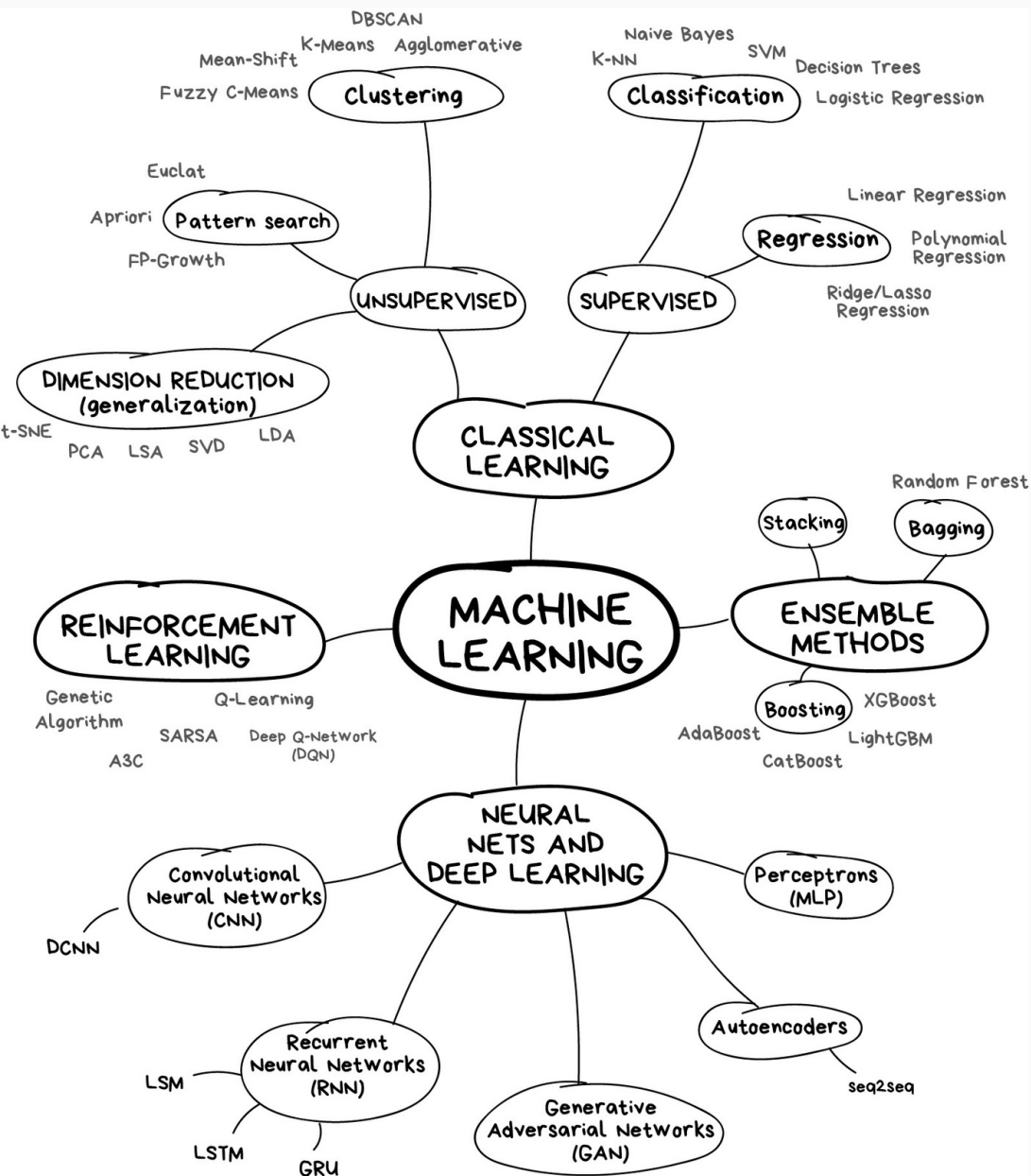
- develop practical **machine learning (ML) foundations in Python**
- **fill in the gaps** left by traditional training in actuarial science or econometrics
- focus on the use of ML methods for the **analysis of frequency + severity data**, but also **non-standard data** such as images
- **explore** a substantial range of **methods (and data types)** (from GLMs to deep learning), but - most importantly - **build foundation** so that you can explore other methods (and data types) yourself.

"In short, we will cover things that we wish someone had taught us in our undergraduate programs."

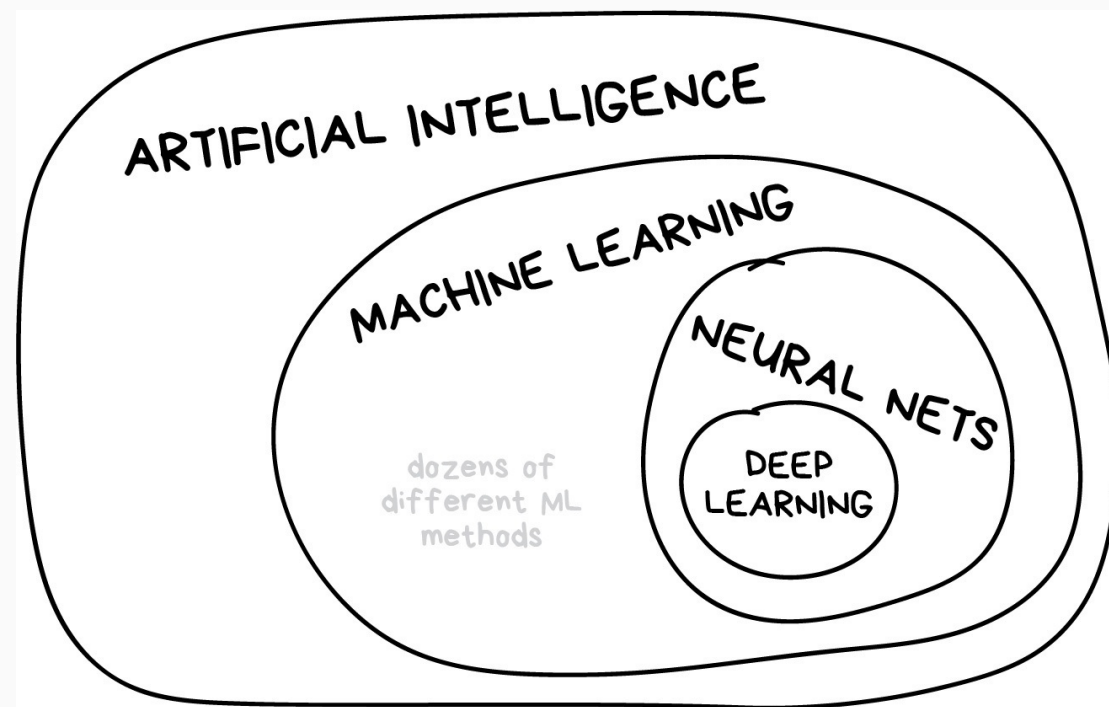
This quote is from the **Data science for economists course** by Grant McDermott.

Module 2's Outline

- Prologue
- Decision tree
 - what is tree-based machine learning?
 - tree basics: structure, terminology, growing process
 - examples on regression and classification
 - tuning via grid search and cross-validation
 - modelling claim frequency and severity data with trees
 - with `scikit-Learn`
- Interpretation tools
 - feature importance
 - partial dependence plot
 - with `scikit-Learn`
- Bagging
 - from a single tree to Bootstrap Aggregating
 - out-of-bag error
 - with `scikit-Learn`
- Random forest
 - from bagging to random forests
 - tuning
 - with `scikit-Learn`
- Gradient boosting
 - (stochastic) gradient boosting with trees
 - training process and tuning parameters
 - modelling claim frequencies and severities
 - with `scikit-Learn` and `xgboost`

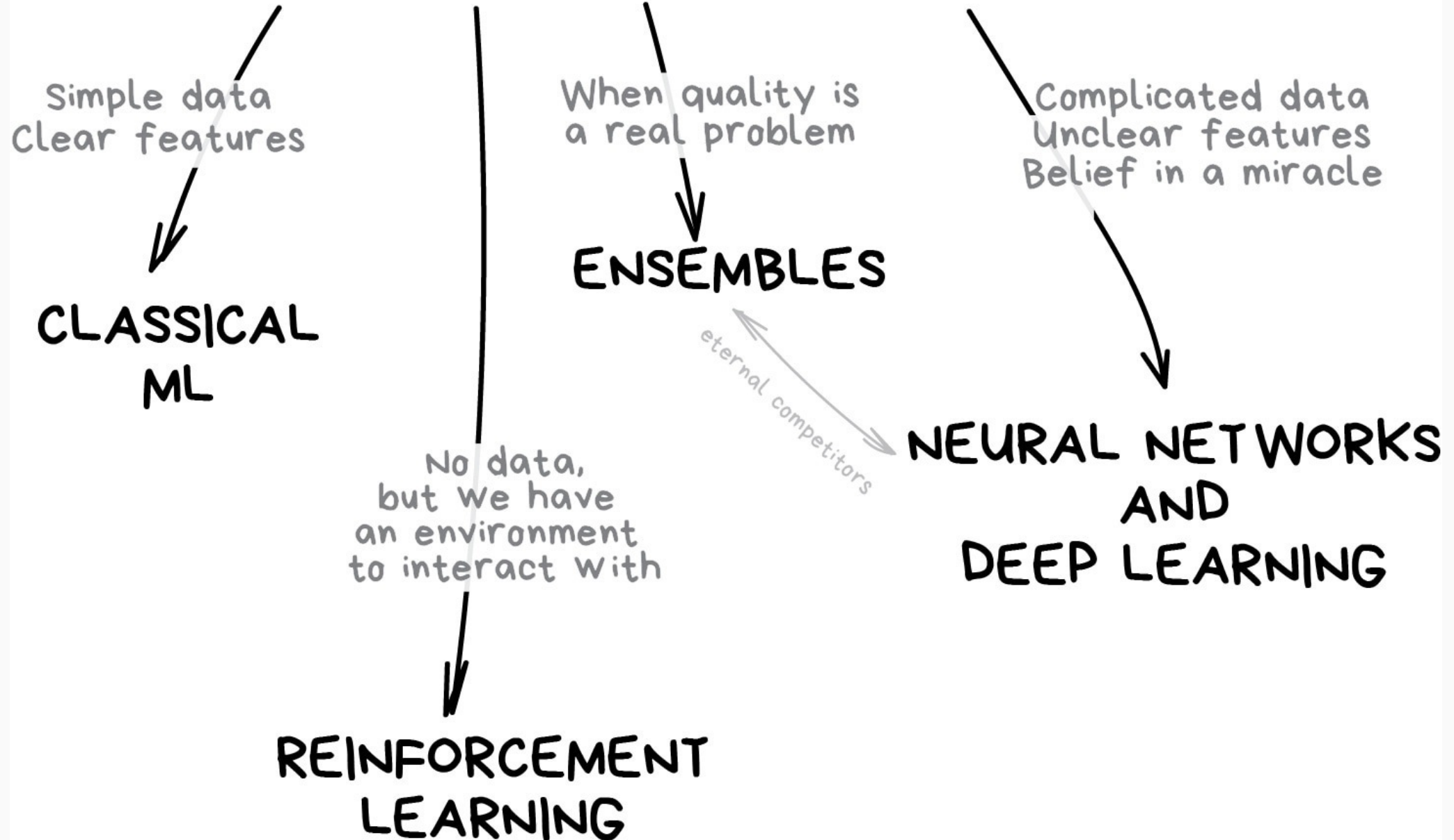


Some roadmaps to explore the ML landscape...



Source: [Machine Learning for Everyone In simple words. With real-world examples. Yes, again.](#)

THE MAIN TYPES OF MACHINE LEARNING



Background reading



Henckaerts et al. (2020) paper on [Boosting insights in insurance tariff plans with tree-based machine learning methods](#)

- full algorithmic details of regression trees, bagging, random forests and gradient boosting machines
- with focus on claim frequency and severity modelling
- including interpretation tools (VIP, PDP, ICE, H-statistic)
- model comparison (GLMs, GAMs, trees, RFs, GBMs)
- managerial tools (e.g. loss ratio, discrimination power).

The paper comes with two notebooks, see [examples tree-based paper](#) and [severity modelling](#).

The paper comes with an R package {distRforest} for fitting random forests on insurance data, see [distRforest](#).

What is tree-based machine learning?

Machine learning (ML) according to [Wikipedia](#):

*"Machine learning algorithms build a **mathematical model** based on sample data, known as training data, in order to make predictions or decisions without being explicitly programmed to perform the task."*

This definition goes all the way back to [Arthur Samuel](#), who coined the term "machine learning" in 1959.

Tree-based ML makes use of a **tree** as building block for the mathematical model.



Single tree

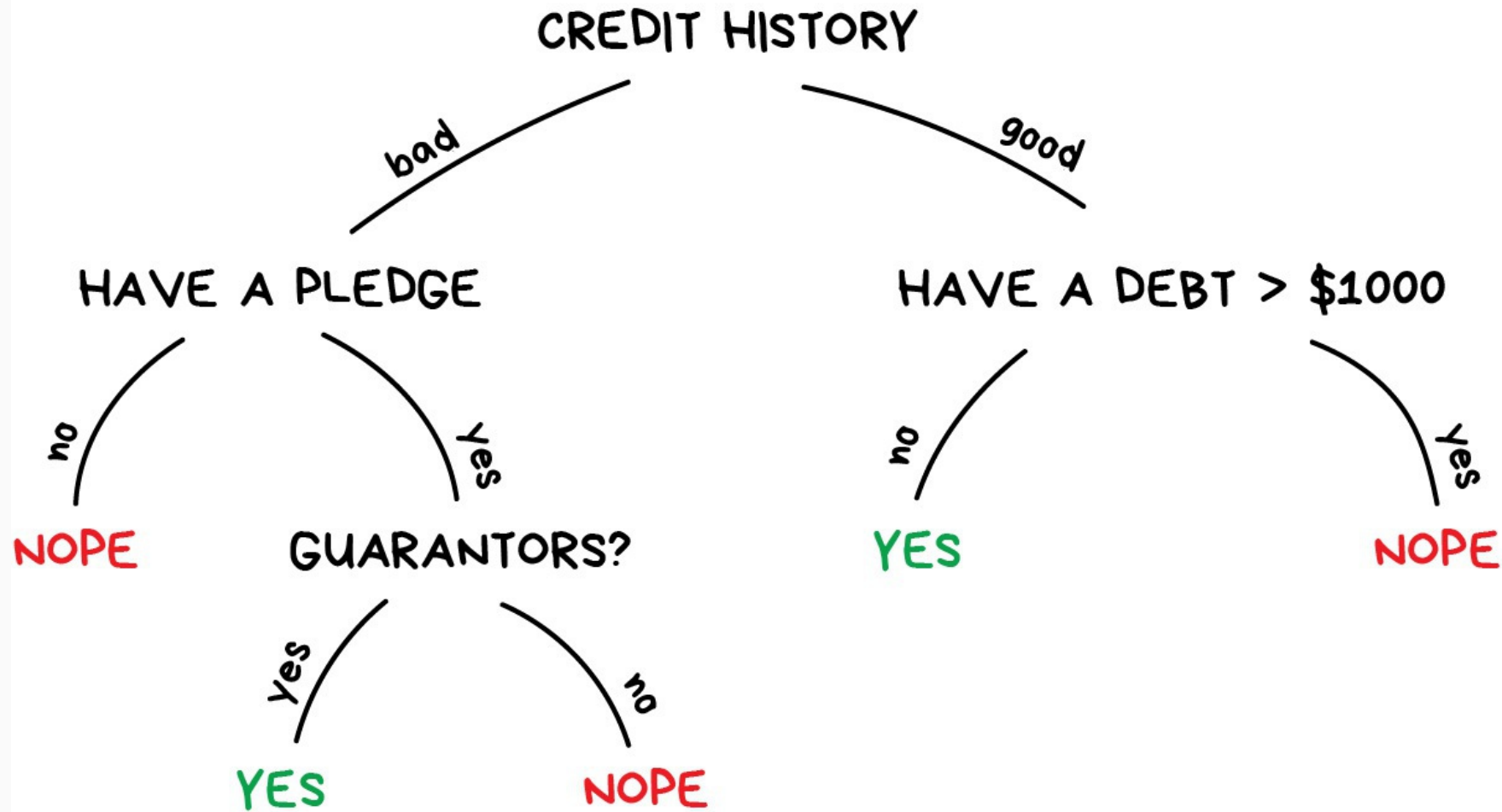


Ensemble of trees

So, a natural question to start from is: what is a **tree**?

Tree basics

GIVE A LOAN?



DECISION TREE

Tree structure and terminology

The top of the tree contains all available training observations: the **root node**.

We **partition** the data into homogeneous non-overlapping subgroups: the **nodes**.

We create subgroups via **simple yes-no questions**.

A tree then predicts the output in a **leaf node** as follows:

- average of the response for regression
- majority voting for classification.

Tree structure and terminology

The top of the tree contains all available training observations: the **root node**.

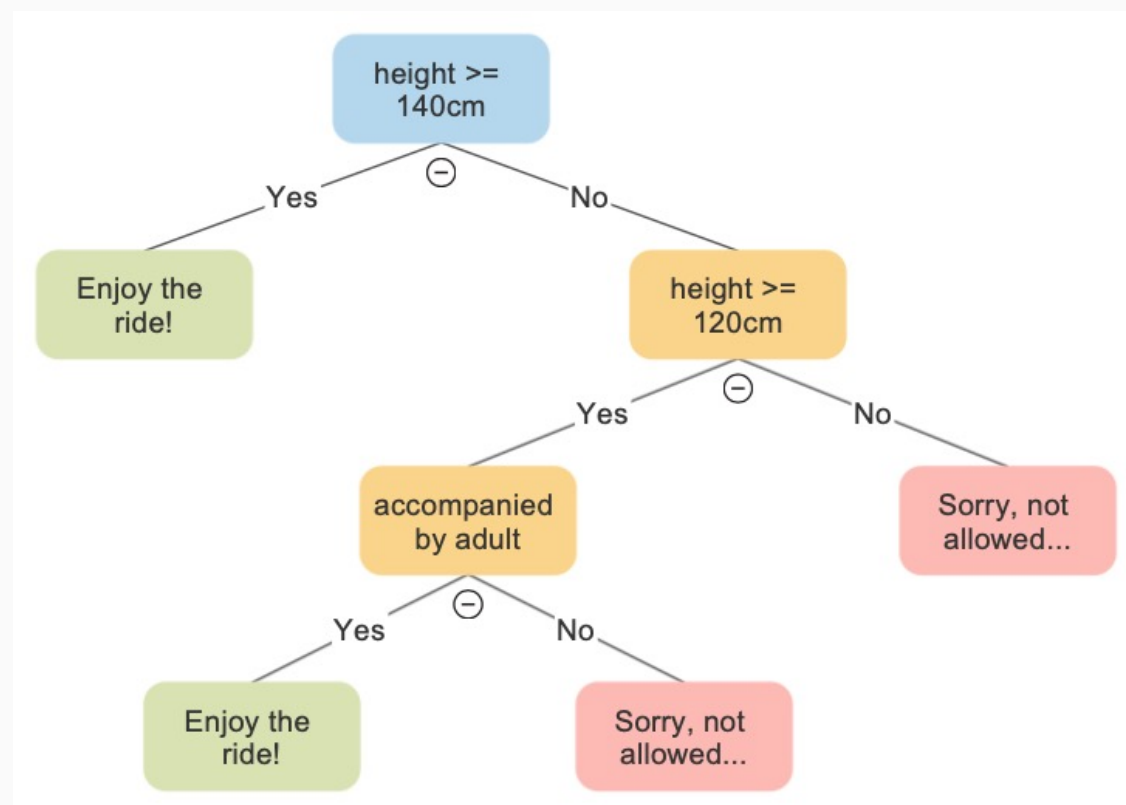
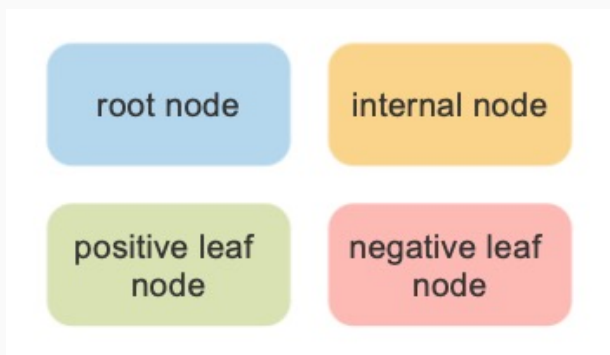
We **partition** the data into homogeneous non-overlapping subgroups: the **nodes**.

We create subgroups via **simple yes-no questions**.

A tree then predicts the output in a **leaf node** as follows:

- average of the response for regression
- majority voting for classification.

Different types of nodes:



Tree growing process

A golden standard is the Classification And Regression Tree algorithm: **CART** (Breiman et al., 1984).

CART uses **binary recursive partitioning** to split the data in subgroups.

In each node, we search for the best feature to **partition** the data into two regions: R_1 and R_2 (hence, **binary**).

Take-away - what is **best**?

Minimize the **overall loss** between observed responses and leaf node prediction





- overall loss = loss in region R_1 + loss in region R_2
- for regression: mean squared or absolute error, deviance,...
- for classification: cross-entropy, Gini index,...

After splitting the data, this process is repeated for region R_1 and R_2 separately (hence, **recursive**).

Repeat until **stopping criterion** is satisfied, e.g., maximum depth of a tree or minimum loss improvement.

How deep should a tree be?

The **bias-variance trade off**:

- a **shallow** tree will underfit:
bias  and variance 
- a **deep** tree will overfit:
bias  and variance 
- find right **balance** between bias and variance!





Typical approach to get the right fit:

- fit an overly complex **deep tree**
- **prune** the tree to find the **optimal subtree**.

How to **prune**?

How deep should a tree be?

The **bias-variance trade off**:

- a **shallow** tree will underfit:
bias  and variance 
- a **deep** tree will overfit:
bias  and variance 
- find right **balance** between bias and variance!

Typical approach to get the right fit:

- fit an overly complex **deep tree**
- **prune** the tree to find the **optimal subtree**.

How to **prune**?

Look for the smallest subtree that minimizes a **penalized loss function**:

$$\min\{f_{\text{loss}} + \alpha \cdot |T|\}$$

- loss function f_{loss}
- complexity parameter α
- number of leaf nodes $|T|$.

A shallow tree results when α is large and a deep tree when α is small.

Perform **cross-validation** on the complexity parameter:

- `ccp_alpha` is the complexity parameter in {sklearn}
- `ccp_alpha` is α divided by f_{loss} evaluated in root node.

Cfr. tuning of the regularization parameter in lasso regression from **Module 1**.

Decision trees with scikit-learn

Let's go to our Colab notebook and learn how we can fit decision trees in Python with `scikit-learn`.

We will explore the following aspects:

- fit simple decision trees to regression and classification toy examples
- try out different parameter settings and check the effect on our tree
- tune towards the optimal parameter values by grid search cross-validation

Claim frequency and severity modeling with trees

Claim frequency prediction on the MTPL data

Recall the MTPL data set introduced in Module 1.

The **Poisson GLM** is a classic approach for modelling **claim frequency** data.

How to deal with claim counts in a decision tree?

Use the **Poisson deviance** as **loss function**:

$$D^{\text{Poi}} = 2 \cdot \sum_{i=1}^n y_i \cdot \ln \frac{y_i}{\text{expo}_i \cdot \hat{f}(x_i)} - \{y_i - \text{expo}_i \cdot \hat{f}(x_i)\},$$

with **expo** the exposure measure.

And what about claim severities?

Actuarial trees with scikit-learn



Let's go to our Colab notebook and learn how we can fit claim frequency and severity trees in Python with `scikit-learn`.

Interpretation tools

Interpreting a tree model

Interpretability depends on the **size of the tree**

- is easy with a **shallow** tree but hard with a **deep** tree
- luckily there are some **tools** to help you.

Feature importance

- identify the most **important** features
- implemented in the package {vip}.

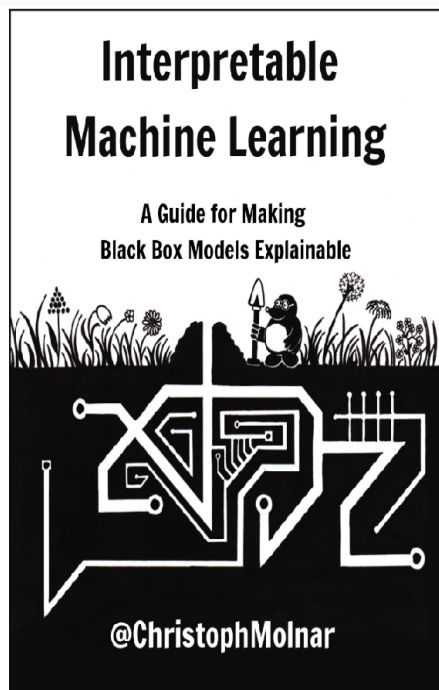
Partial dependence plot

- measure the **marginal effect** of a feature
- implemented in the package {pdp}.

Excellent source on interpretable machine learning:

Interpretable Machine Learning book by Christophe Molnar.

Feature importance and partial dependence



With **feature importance**:

- sum improvements in loss function over all splits on a variable x_ℓ
- important variables appear high and often in a tree.

With **partial dependence**:

- univariate

$$\bar{f}_\ell(x_\ell) = \frac{1}{n} \sum_{i=1}^n f_{\text{tree}}(x_\ell, \mathbf{x}_{-\ell}^i)$$

- bivariate

$$\bar{f}_{k,\ell}(x_k, x_\ell) = \frac{1}{n} \sum_{i=1}^n f_{\text{tree}}(x_k, x_\ell, \mathbf{x}_{-k,\ell}^i)$$

- marginal effects, interactions can stay hidden!

Interpretation tools with scikit-learn



Let's go to our Colab notebook and learn how we can explain decision trees in Python with `scikit-learn`.

That's a wrap on single trees!

Advantages 😊

- Shallow tree is easy to **explain** graphically.
- Closely mirrors the human **decision-making** process.
- Handles all types of features **without** pre-processing.
- **Fast** and very scalable to big data.
- **Automatic** variable selection.
- Surrogate splits can handle **missing** data.

That's a wrap on single trees!

Advantages 😊

- Shallow tree is easy to **explain** graphically.
- Closely mirrors the human **decision-making** process.
- Handles all types of features **without** pre-processing.
- **Fast** and very scalable to big data.
- **Automatic** variable selection.
- Surrogate splits can handle **missing** data.

Disadvantages 😞

- Tree uses **step** functions to approximate the effect.
- Greedy heuristic approach chooses **locally** optimal split (i.e., based on all previous splits).
- Data becomes **smaller** and smaller down the tree.
- All this results in **high variance** for a tree model...
- ... which harms **predictive performance**.

From a single tree to ensembles of trees

Ensembles of trees

Remember: prediction error = bias + variance + irreducible error.

Good **predictive performance** requires low bias **AND** low variance.

Two popular **ensemble** algorithms (that can be applied to any type of model, not just trees) are:

Bagging (Breiman, 1996)

- low **bias** via detailed individual models
- (think: deep trees)
- low **variance** via averaging of those models
- (think: in parallel)

Boosting (Friedman, 2001)

- low **variance** via simple individual models
- (think: stumps)
- low **bias** by incrementing the model sequentially
- (think: sequentially).

Random forest (Breiman, 2001) is then a modification on bagging for trees to further improve the variance reduction.

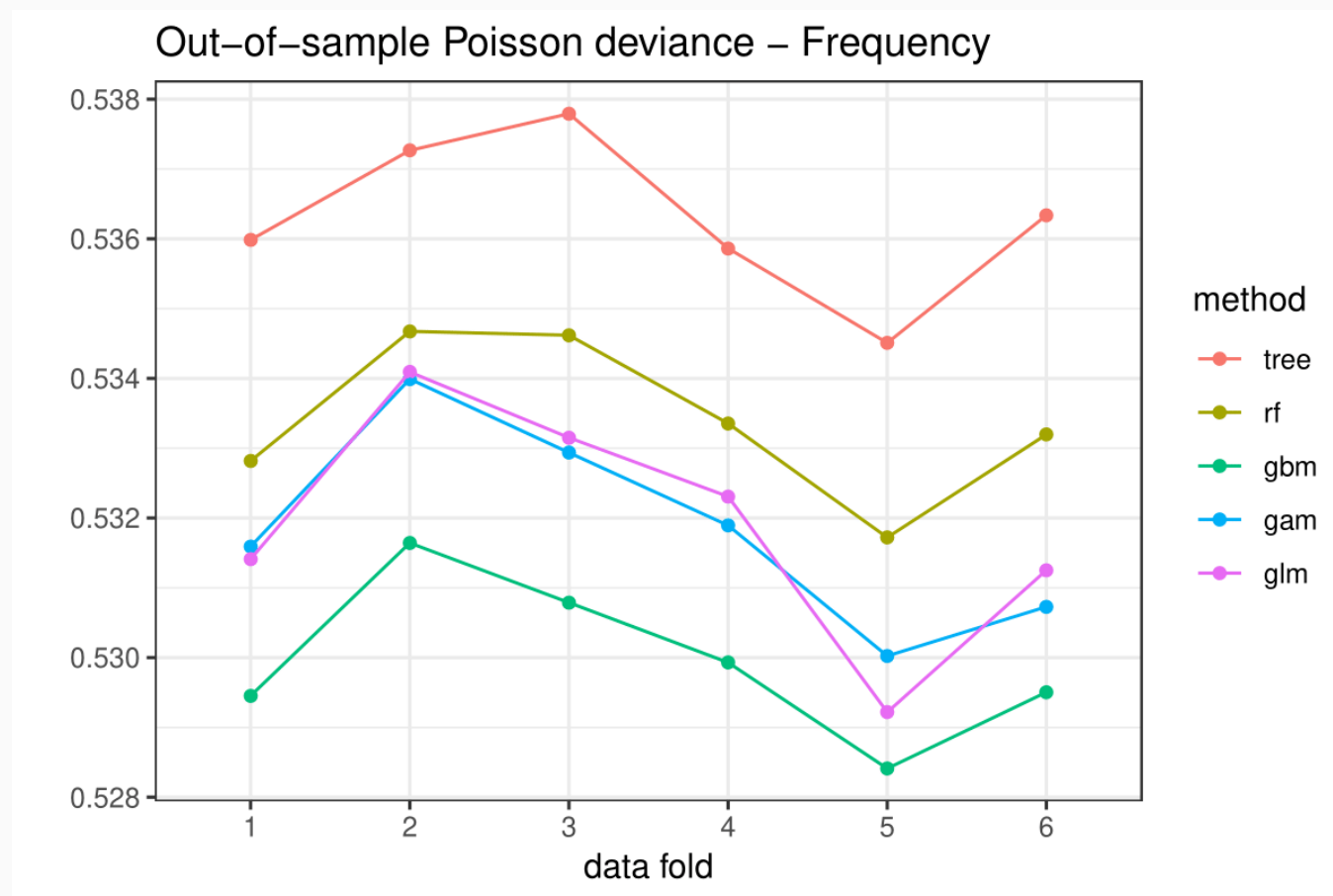
Model comparison on claim frequency data

Detailed discussion in our North American Actuarial Journal (2021) paper.

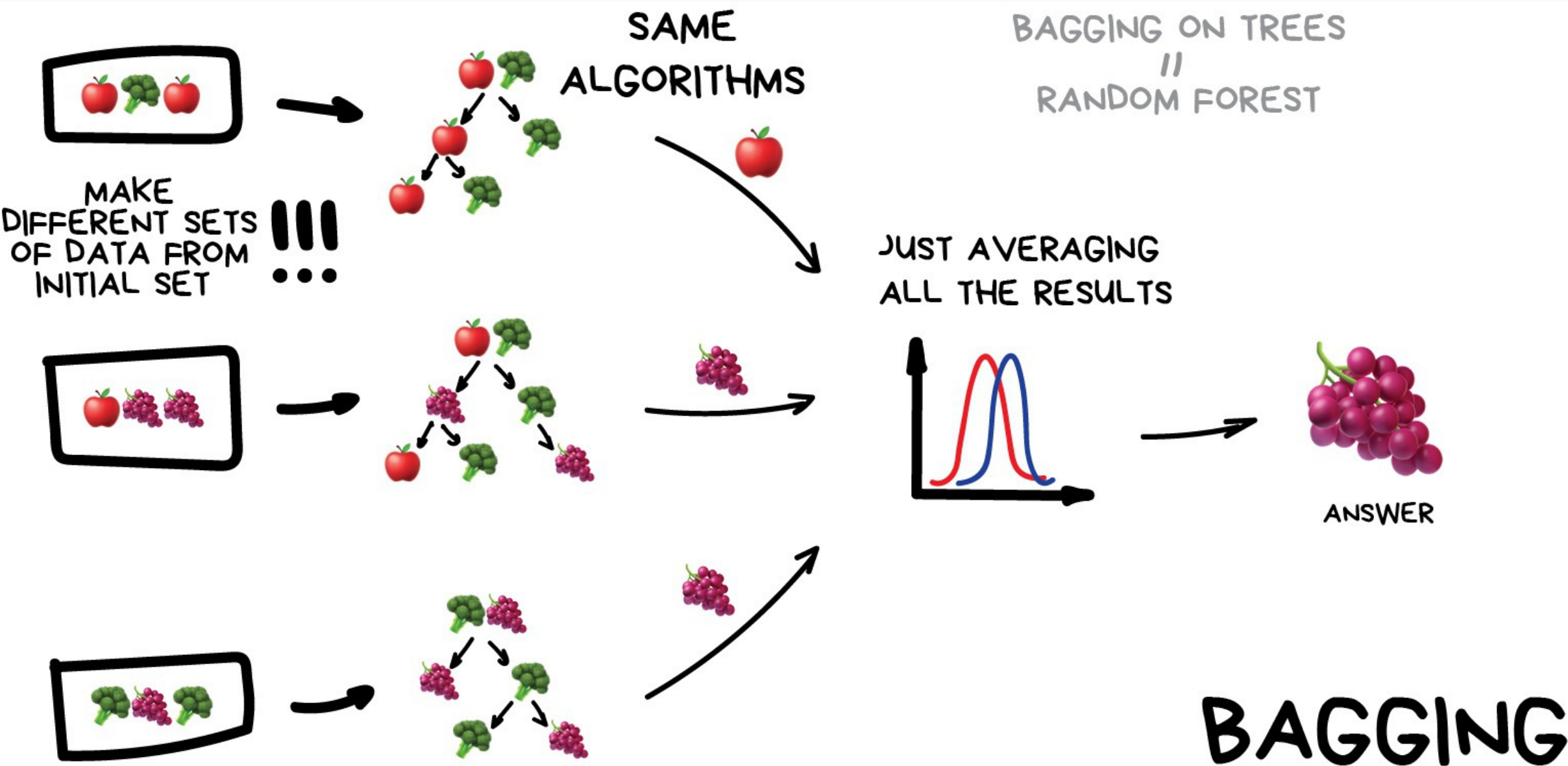
Analyzing frequency as well as severity data.

Picture taken from [Henckaerts et al. \(2021\)](#).

Boosting > Random forest > Bagging > Decision tree



Introducing bagging



Bagging or Bootstrap AGGregatING

- build a lot of different **base learners** on bootstrapped samples of the data
- **combine** their predictions
- model **averaging** helps to:
 - reduce variance
 - avoid overfitting.
- bagging works best for **base learners** with:
 - **low bias** and **high variance**
 - for example: deep decision trees.

Bagging with trees?

- do **B** times:
 - create bootstrap sample by drawing with replacement from the original data
 - fit a deep tree to the bootstrap sample.
- **combine** the predictions of these B trees
 - **average** prediction for regression
 - **majority** vote for classification.

Out-of-bag (OOB) error

Bootstrap samples are constructed **with** replacement.

Some observations are not present in a bootstrap sample:

- they are called the **out-of-bag** observations
- use those to calculate the out-of-bag (OOB) error
- measures **hold-out** error like cross-validation does.

Advantage of OOB over cross-validation?

- the OOB error comes **for free** with bagging.

Bagging trees with scikit-learn



Let's go to our Colab notebook and learn how we can perform bagging with decision trees in Python with `scikit-learn`.

From bagging to random forests

Problem of dominant features



A downside of bagging is that **dominant features** can cause individual trees to have a **similar structure**.

This is known as **tree correlation**.

Remember the **feature importance** results discussed earlier for the MTPL data?

- `bm` is a very dominant variable
- `ageph` was rather important
- `power` also, but to a lesser degree.

Problem?

- bagging gets its predictive performance from **variance reduction**
- however, this reduction  when tree correlation 
- dominant features therefore **hurt** the predictive performance of a bagged ensemble!

Random forest

Random forest is a modification on bagging to get an ensemble of **de-correlated** trees.

Process is very similar to bagging, with one small **trick**:

- before each split, select a **subset of features** at random as candidate features for splitting
- this essentially decorrelates the trees in the ensemble, improving predictive performance
- the number of candidates is typically considered a tuning parameter.

Bagging introduces randomness in the **rows** of the data.

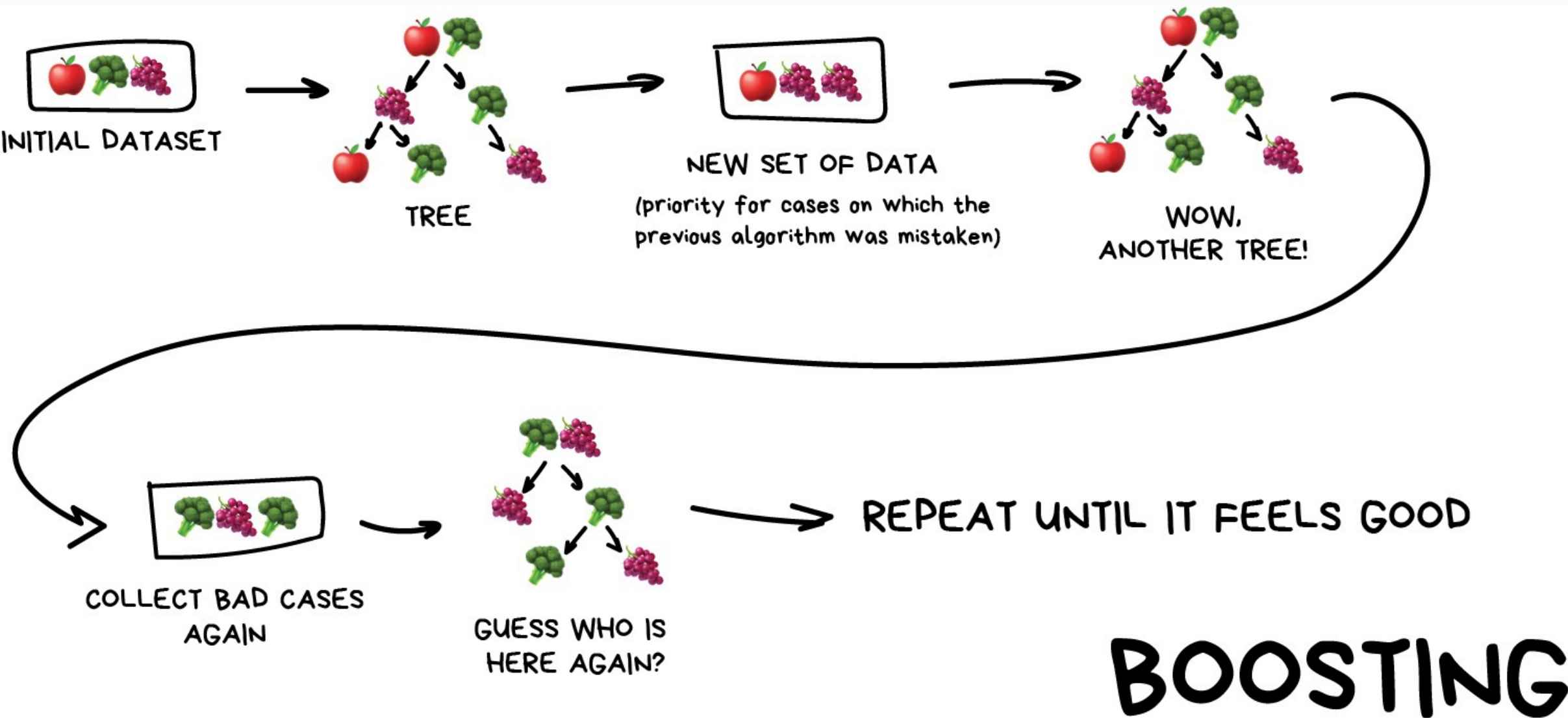
Random forest introduces randomness in the **rows** and **columns** of the data.

Random forest with scikit-learn



Let's go to our Colab notebook and learn how we can build random forests in Python with `scikit-learn`.

(Stochastic) Gradient Boosting Machines



Boosting vs. Bagging

Similar to bagging, boosting is a **general technique** to create an **ensemble** of any type of base learner.

With bagging:

- **strong base learners**
 - low bias, high variance
 - for example: deep trees
- **variance reduction** through **averaging**
- **parallel** approach
 - trees not using information from each other
 - performance thanks to **averaging**
 - low risk for overfitting.

With boosting:

- **weak base learners**
 - low variance, high bias
 - for example: stumps
- **bias reduction** in ensemble through **updating**
- **sequential** approach
 - current tree uses information from all past trees
 - performance thanks to **rectifying** past mistakes
 - high risk for overfitting.

GBM: stochastic gradient boosting with trees

We focus on GBM:

- with **decision trees**
- *stochastic* by **subsampling** in the rows (and columns) of the data
- *gradient* by optimizing the loss function via **gradient descent**.

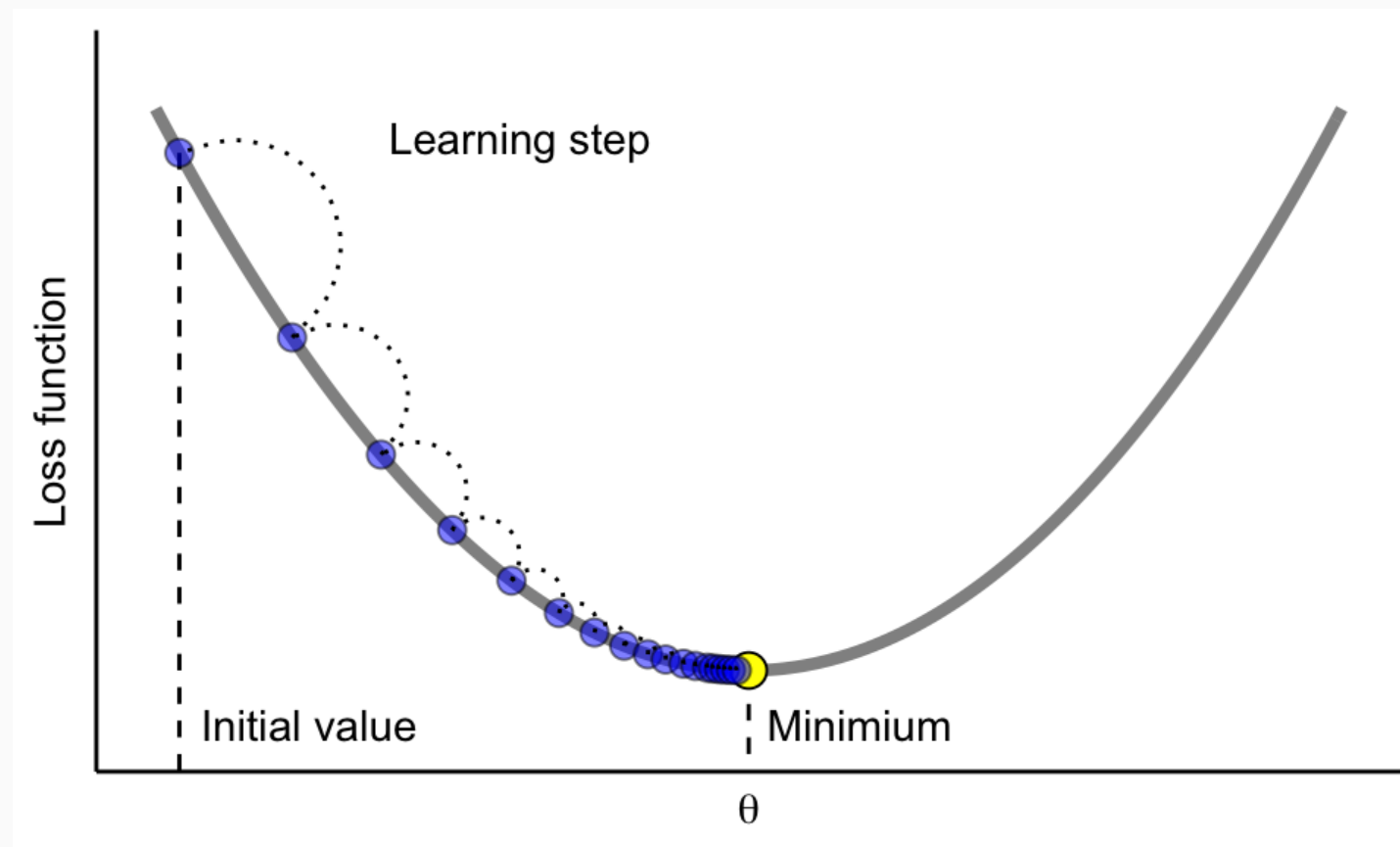


Figure 12.3 from Boehmke & Greenwell [Hands-on machine learning with R](#).

Stochastic gradient descent

The **learning rate** (also called step size) is very important in gradient descent

- if too big → likely to **overshoot** the optimal solution
- if too small → **slow** process to reach the optimal solution

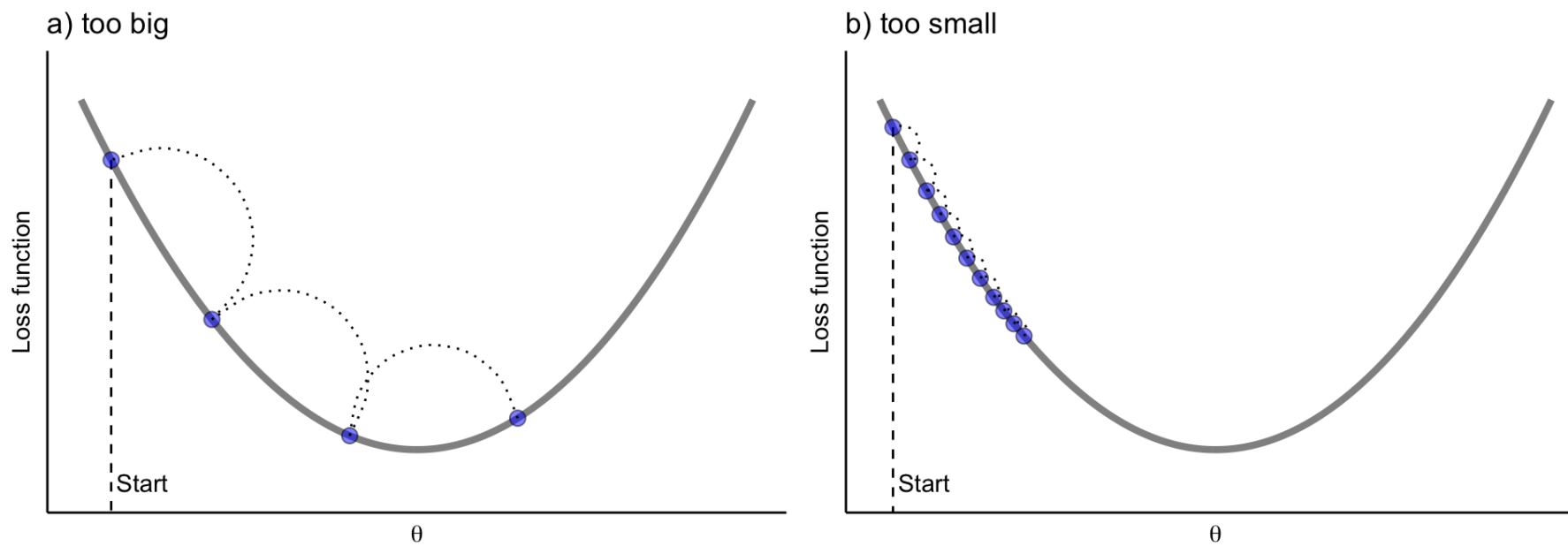


Figure 12.4 from Boehmke & Greenwell [Hands-on machine learning with R](#).

Subsampling allows to escape plateaus or local minima for **non-convex** loss functions.

GBM training process

Initialize the model fit with a global average and calculate **pseudo-residuals**.

Do the following **B** times:

- fit a tree of a pre-specified depth to the **pseudo-residuals**
- **update** the model fit and pseudo-residuals with a **shrunk** version
- shrinkage to slow down learning and **prevent** overfitting.

The model fit after **B** iterations is the **end product**.

We will explore two **popular** libraries for stochastic gradient boosting


- {sklearn}: standard for regression and classification, but not the fastest
- {xgboost}: efficient implementation with some **extra** elements, for example regularization.

GBM parameters

A lot of parameters at our disposal to **tweak** the GBM.

Some have a **big impact** on the performance and should therefore be **properly tuned**:

- number of trees: depends very much on the **use case**, ranging from 100's to 10 000's
- tree depth: **low** values are preferred for boosting to obtain weak base learners
- learning rate: typically set to the lowest possible value that is **computationally** feasible.

Rule of thumb: if learning rate  then number of trees .

GBM with scikit-learn



Let's go to our Colab notebook and learn how we can build GBMs in Python with `scikit-learn`.

XGBoost

XGBoost

XGBoost stands for eXtreme Gradient Boosting.

Optimized gradient boosting library: efficient, flexible and portable across multiple languages.

XGBoost follows the same general boosting approach as GBM, but adds some **extra elements**:

- **regularization**: extra protection against overfitting (see Lasso and glmnet on Day 1)
- **early stopping**: stop model tuning when improvement slows down
- **parallel processing**: can deliver huge speed gains
- different **base learners**: boosted GLMs are a possibility
- multiple **languages**: implemented in R, Python, C++, Java, Scala and Julia

XGBoost also allows to **subsample columns** in the data, much like the random forest did

- GBM only allowed subsampling of rows
- XGBoost therefore **unites** boosting and random forest to some extent.

Very **flexible** method with many many parameters, full list can be found [here](#).

GBM with xgboost



Let's go to our Colab notebook and learn how we can build GBMs in Python with `xgboost`.

Thanks!



Slides created with the R package `xaringan`.

Course material available via

 <https://github.com/katrienantonio/hands-on-machine-learning-Python-module-2>