Models in Machine Learning

An Overview

PREFACE

@Bernd: pls fill with wisdom

In machine learning, there's something called the "No Free Lunch" theorem. In a nutshell, it states that no one algorithm works best for every problem, and it's especially relevant for supervised learning (i.e. predictive modeling). For example, you can't say that neural networks are always better than decision trees or vice-versa. There are many factors at play, such as the size and structure of your dataset. As a result, you should try many different algorithms for your problem, while using a hold-out "test set" of data to evaluate performance and select the winner.

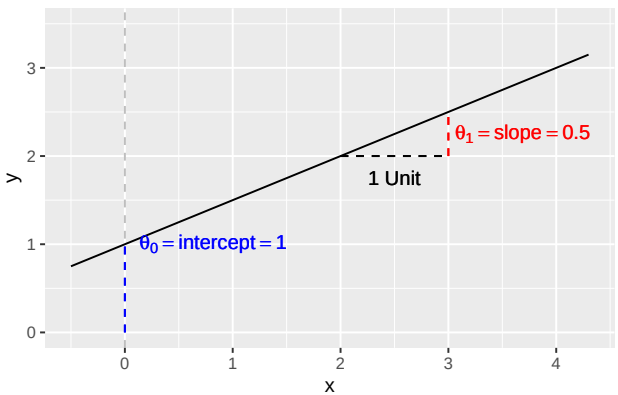
LINEAR MODEL – FUNCTIONALITY

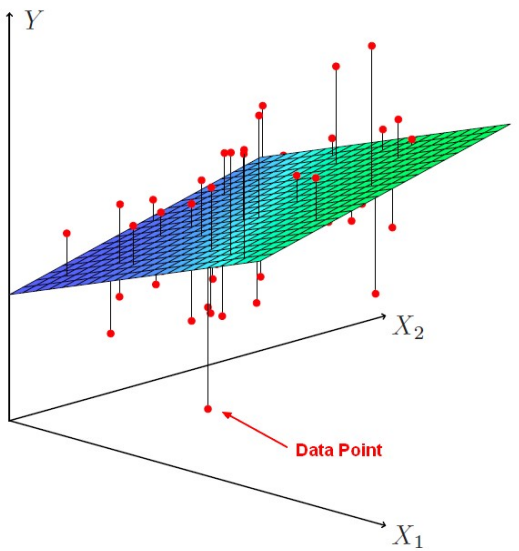
SUPERVISED | PARAMETRIC | LINEAR | WHITE-BOX

Caro alter Tanzbesen

General idea A linear model (LM) fits a **hyperplane** $\theta_0 + \boldsymbol{\theta}^T \mathbf{x}$ to minimize the distance between the data points and its closed point on the hyperplane.

Hypothesis space $\mathcal{H} = \{\theta_0 + \boldsymbol{\theta}^T \mathbf{x} \mid (\theta_0, \boldsymbol{\theta}) \in \mathbb{R}^{p+1}\}$





LINEAR MODEL – FUNCTIONALITY

Empirical risk

- Typically, **ordinary least squares (OLS)** with a squared loss function is used for regression: $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y^{(i)} - \boldsymbol{\theta}^T \mathbf{x}^{(i)} \right)^2$
- Sometimes the empirical risk function is based on the **absolute loss** or the **Huber loss**.
- For **logistic regression** the ERM is based on the **log loss** $L\left(y, f(\mathbf{x})\right) = \log\left[1 + \exp\left(-yf(\mathbf{x})\right)\right]$.
- In this case, the hyperplane can represent the decision boundary between two classes (**classification**).

Optimization

- for **OLS**: analytically with $\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$
- for **other loss functions**: numerical optimization

Hyperparameters None

Runtime behavior $\mathcal{O}(d^2 \cdot n + d^3)$ for n observations and d features

LINEAR MODEL – PRO'S & CON'S

Advantages

+ **simple and fast** implementation; cheap computational costs
+ intuitive **interpretability**: mean influence of features on the output and feature importance
+ fits **linearly** separable data sets very well
+ works well **independent of data size**
+ basis for many machine learning algorithms
+ boo

Disadvantages

- not suitable for data based on a **non-linear** data generating process $\rightarrow$ **strong simplification** of real-world problems
- **strong assumptions**: data is independent (multi-collinearity must be removed and normal distributed residuals ??
- tend to **overfit** (can be reduced by regularization)
- **sensitive to outliers and noisy data**

Simple method with good interpretability for linear problems, but strong assumptions and simplification of real-world problems

LINEAR MODEL – PRACTICAL HINTS

Check assumptions???????

This model is very effective, if the following assumptions are fulfilled:

- **linearity**: the relationship between the mean of predicted value and the features
- **homoscedasticity**: The variance of residuals is equal for all features.
- **independence**: All observations are independent of each other.
- **normality**: Y is normally distributed for any fixed value of the features

Implementation

- **R:** function `lm`
- **Python:** `LinearRegression` from package `sklearn.linear_model`, package for advanced statistical parameters `statsmodels.api`

Regularization

In practice, we often use regularized models in order to **prevent overfitting** or perform feature selection. More details will follow in the subsequent chapter.

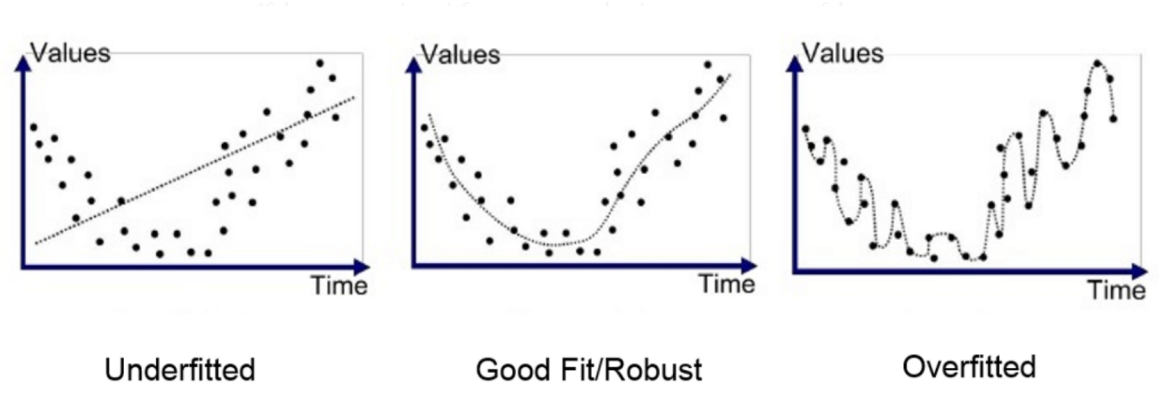
REGULARIZED LM – FUNCTIONALITY

SUPERVISED | PARAMETRIC | LINEAR | WHITE-BOX

General idea

- Linear model (LM) can **overfit** if we operate in high-dimensional space with not that many oberservations.
- When features are highly correlated, the least-squares estimate becomes highly sensitive to random errors in the observed response, producing a **large variance in the fit**.
- If we fit a linear model, we can find a compromise between generalizing the model (simple model, underfitted) and correspond closely to the data (complex model, overfitted).

Hypothesis space $\mathcal{H} = \{\theta_0 + \boldsymbol{\theta}^T \mathbf{x} \mid (\theta_0, \boldsymbol{\theta}) \in \mathbb{R}^{p+1}\}$



REGULARIZED LM – FUNCTIONALITY

Empirical risk

- Therefore, we minimize the empirical risk function $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ **plus the a complexity measure** $J(\boldsymbol{\theta})$:

$$\mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \cdot J(\boldsymbol{\theta}).$$

- We can use the L2-penalty for the complexity measure (**ridge regression**) with $J(\boldsymbol{\theta}) = |\boldsymbol{\theta}\|_2^2$.
- Alternativly, **LASSO** (least absolute shrinkage and selection operator) uses the L1-penalty ($J(\boldsymbol{\theta}) = |\boldsymbol{\theta}\|_1$).
- Whereas both regularization methods shrink the coefficients of the model, LASSO also performs **feature selection**.
- Elastic net as a convex combination of Ridge and LASSO ???

Optimization

- for **Ridge** regression: analytically with $\hat{\boldsymbol{\theta}}_{\text{Ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda \boldsymbol{I})^{-1}\mathbf{X}^T\mathbf{y}$
- for **LASSO regression**: e. g. (sub)-gradient descent

Hyperparameters **Shrinkage parameter** λ [and α for Elastic net??]

Runtime behavior $\mathcal{O}(d^2 \cdot n + d^3)$ for n observations and d features

REGULARIZED LM – PRO'S & CON'S

SAME LIKE LINEAR MODEL?

Advantages

+ simple and fast implementation; cheap computational costs
+ intuitive interpretability: mean influence of features on the output and feature importance
+ fits linearly separable data sets very well
+ works well independent of data size
+ basis for many machine learning algorithms
+ prevents **overfitting**

Disadvantages

- not suitable for data based on a non-linear data generating process $\rightarrow$ strong simplification of real-world problems
- strong assumptions: data is independent (multi-collinearity must be removed and normal distributed residuals ????
- sensitive to outliers and noisy data

Simple method with good interpretability for linear problems, but strong assumptions and simplification of real-world problems.

REGULARIZED LM – PRACTICAL HINTS

Choice of regularization parameter λ

Choose λ with e. g. smallest sum of squared residuals through cross-validation.
In the R package `glmnet` `lambda.min` is the value of λ that gives minimum mean cross-validated error.

Implementation

- **R:** package for regularized linear model `glmnet`
- **Python:** `LinearRegression` from package `sklearn.linear_model`, package for advanced statistical parameters `statsmodels.api`

SVM – FUNCTIONALITY

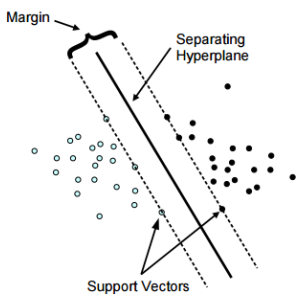
SUPERVISED | |NON| PARAMETRIC | |NON| LINEAR | WHITE-BOX

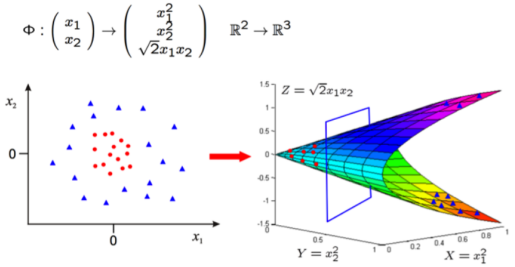
General idea

- Support vector machines (SVMs) construct separating **hyperplanes** in a multi-dimenstional space.
- The SVM algorithm finds a decision boundary (**separating hyperplane**) that maximizes the distance (**margin**) between the closest members (**support vectors**) of the separate classes. (**Linear SVM**)
- If the data is not linearly separable, **kernels** transform the input space into a higher dimensional space. (**Non-linear SVM**)

Hypothesis space

$$\mathcal{H} = \{\text{sign} \sum_{i=1}^{n} \alpha_i y^{(i)} k(\mathbf{x}^{(i)}, \mathbf{x}) + \theta_0 | \ (\theta_0, \boldsymbol{\theta}) \in \mathbb{R}^{p+1}\}$$





SVM – FUNCTIONALITY

Empirical risk
N

Optimization XX

Hyperparameters

- **C**: penalization for missclassified data points
- **kernel parameters**: depending if and which kernel is used (e. g. degree of the polynomial kernel or width of RBF kernel)

Runtime behavior $\mathcal{O}(n^2 \cdot d + n^3)$ for n observations and d features

SVM – PRO'S & CON'S

Advantages

+ can learn non-linear decision boundaries
+ often sparse solution
+ robust against overfitting; especially in high-dimensional space
+ computational costs are low ?????

Disadvantages

- not easy tunable as it is highly important to choose the right kernel
- does not scale well to larger data sets

XX

SVM – PRACTICAL HINTS

Kernels

For projecting the input data into another onto a higher dimension, mainly these three types of kernels are used:

- **Linear kernel**: the dot product of the given observations
- **Polynomial kernel**: curved lines in the input space
- **Radial basis function (RBF) kernel**: complex regions in the input space (e. g. spirals)

Implementation

- **R:** package for regularized linear model `XXX`
- **Python:** `XX` from package `XXX`, package for XXXX `statsmodels.api`

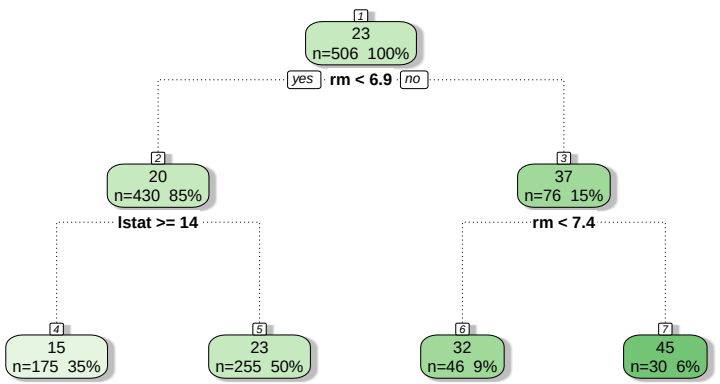
CART – FUNCTIONALITY

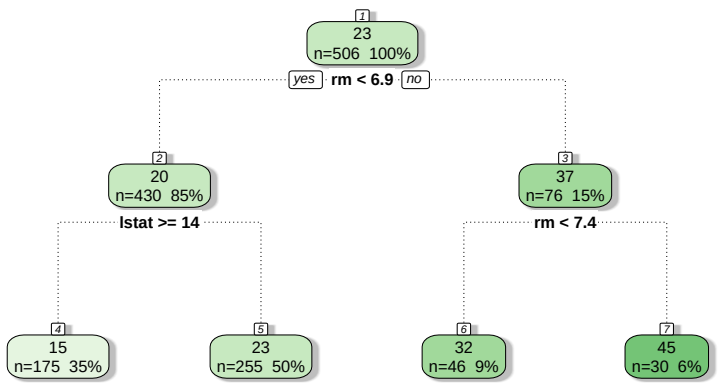
SUPERVISED | NON-PARAMETRIC | NON-LINEAR | WHITE-BOX | FEATURE SELECTION

General idea

- Starting from a root node, ***classification & regression trees (CART)*** perform repeated **binary splits** of the data according to feature values, thereby subsequently dividing the input space $\mathcal{X}$ into T **rectangular partitions** Q_t.
- Observations are passed along until each ends up in exactly one leaf node (unless ***stopped early*** or ***pruned***).
- In each step, CART find the optimal feature-threshold combination to split by.
- Leaf node t is assigned response c_t.

Hypothesis space $\mathcal{H} = \left\{ f(\mathbf{x}) : f(\mathbf{x}) = \sum_{t=1}^{T} c_t \mathbb{I}(\mathbf{x} \in Q_t) \right\}$





CART – FUNCTIONALITY

Empirical risk

- Empirical risk is calculated for each potential terminal node $\mathcal{N}_t$ of a split.
- In general, trees can handle any type of loss function. Typical choices are:
  - Classification (for g classes):
    - Using ***Brier score*** $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x},y) \in \mathcal{N}_t} \sum_{k=1}^{g} \left(\mathbb{I}(y = k) - \pi_k(\mathbf{x}) \right)^2$
    - Using ***Bernoulli loss*** $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x},y) \in \mathcal{N}_t} \sum_{k=1}^{g} \mathbb{I}(y = k) \cdot \log(\pi_k(\mathbf{x}))$
  - Regression: Using ***quadratic loss*** $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x},y) \in \mathcal{N}_t} (y - c_t)^2$

Optimization

Exhaustive search over all (randomly selected) split candidates in each node to minimize empirical risk in the child nodes (greedy optimization)

Hyperparameters **Complexity**, i.e., number of leaves T

Runtime behavior $\mathcal{O}(n^2 \cdot d)$ for n observations and d features

CART – PRO'S & CON'S

Advantages

+ **Easy** to understand, interpret & visualize
+ Automatic handling of **non-numerical** features
+ Built-in **feature selection**
+ Automatic handling of **missings**
+ **Interaction** effects between features easily possible, even of higher orders
+ **Fast** computation and good scalability
+ High **flexibility** (custom split criteria or leaf-node prediction rules)

Disadvantages

- Rather **low accuracy** (at least, without bagging or boosting)
- High **variance/instability**: strong dependence on training data
- Therefore, poor generalization & risk of **overfitting**
- Several steps required for modeling **linear** relationships
- In presence of categorical features, **bias** towards features with **many categories**

Simple and good with feature selection, but not the best predictor

CART – PRACTICAL HINTS

Pruning / early stopping

Unless interrupted, splitting will go on until each leaf node contains a single observation (expensive + overfitting!)

$\rightarrow$ Use **pruning** and **stopping criteria** to limit complexity

Implementation

R: package `rpart`
Python: `DecisionTreeClassifier` / `DecisionTreeRegressor` from package `scikit-learn`
Complexity controlled via tree depth, minimum number of observations per node, maximum number of leaves, minimum risk reduction per split, ...

Bagging / boosting

Since CART are instable predictors on their own, they are typically ensembled to form a ***random forest*** (***bagging***) or used in combination with ***boosting***.

RANDOM FOREST – FUNCTIONALITY

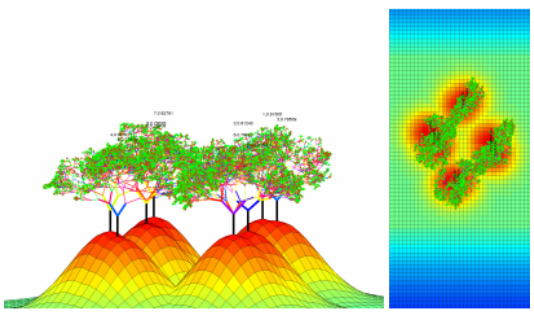
SUPERVISED | NON-PARAMETRIC | NON-LINEAR | BLACK-BOX | FEATURE SELECTION

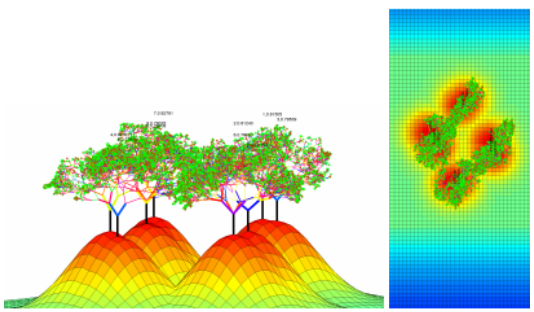
General idea

- ***Random forests (RF)*** are **bagging ensembles**: they combine *M* CART (base learners) to form a strong learner.
- They use **complex** trees with low bias and compensate for single trees' high variance by aggregating them in a **decorrelated** manner.
- Each tree is trained a on a **bootstrap sample** of the data and only on a random **subset of features** to incur variability.
- Aggregation takes place via **averaging** (regression) or **majority voting** (classification).

Hypothesis space

$$\mathcal{H} = \left\{ f(\mathbf{x}) : f(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} \sum_{t=1}^{T^{[m]}} c_t^{[m]} \mathbb{I}(\mathbf{x} \in Q_t^{[m]}) \right\}$$





RANDOM FOREST – FUNCTIONALITY

Empirical risk

- Applicable with **any** kind of loss function (just like tree base learners)
- Computation of empirical risk for all potential child nodes in all trees

Optimization

Exhaustive search over all (randomly selected) split candidates in each node of each tree to minimize empirical risk in the child nodes (greedy optimization)

Hyperparameters

- **Ensemble size**, i.e., number of trees
- **Complexity**, i.e., number of leaves T of each base learner
- **Number of split candidates**, i.e., number of features to be considered as splitting variables at each split
  $\rightarrow$ Frequently used heuristics: $\lfloor \sqrt{p} \rfloor$ for classification and $\lfloor p/3 \rfloor$ for regression

Runtime behavior $\mathcal{O}(M \cdot n^2 \cdot d)$ for M trees, n observations and d features

RANDOM FOREST – PRO'S & CON'S

Advantages

+ Translation of most of **base learners'** advantages (e.g., inherent variable selection, handling of missing data)
+ Fairly good at **prediction**: improved accuracy through bagging
+ Inherent computation of **feature importance**
+ Quite **stable** wrt changes in the data
+ Good with **high-dimensional** data, even in presence of noisy covariates
+ Applicable to **unbalanced** data
+ Easy to **parallelize**
+ Rather easy to **tune**

Disadvantages

- Translation of some of **base learners'** disadvantages (e.g., trouble to model linear relations, bias towards features with many categories)
- Loss of single trees' **interpretability** – black-box method
- Hard to **visualize**
- Often suboptimal for **regression**
- Often still inferior in **performance** to other methods (e.g., boosting)

Fairly good predictor, but black-box method

RANDOM FOREST – PRACTICAL HINTS

Pre-processing

Inherent feature selection of random forests, but high **computational costs** for large number of features
$\rightarrow$ Upstream feature selection (e.g., via PCA) might be advisable

Implementation

R: package `ranger`
Python: `RandomForestClassifier` / `RandomForestRegressor` from package `scikit-learn`

Tuning

Overall **limited tunability**
Number of split candidates often more impactful than number of trees

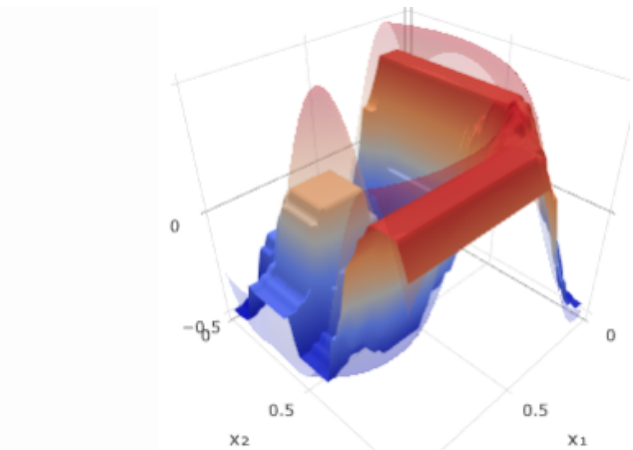
GRADIENT BOOSTING – FUNCTIONALITY

SUPERVISED | NON-PARAMETRIC | NON-LINEAR | BLACK-BOX | FEATURE SELECTION

General idea Gradient boosting (GB) is an ***ensemble*** method that constructs a strong learner from weak base learners (frequently, CART).
As opposed to ***bagging***, however, base learners are assembled in a **sequential, stage-wise** manner: in each iteration, GB improves the current model by adding a new component that minimizes empirical risk. The final model is a weighted sum of base learners $b(\mathbf{x}, \theta^{[m]})$ with weights $\beta^{[m]}$.

Hypothesis space $\mathcal{H} = \left\{ f(\mathbf{x}) : f(\mathbf{x}) = \sum_{m=1}^{M} \beta^{[m]} b(\mathbf{x}, \theta^{[m]}) \right\}$

- Finding next additive component $\widehat{=}$ fitting base learner to current point-wise residuals
- One boosting iteration $\widehat{=}$ one approximate gradient step in function space
- Fitting of each base learner using information from previously added ones



GRADIENT BOOSTING – FUNCTIONALITY

Empirical risk

- **Outer loss:** Loss used to compute pseudo-residuals – how large is the error of the current model fit?
  $\rightarrow$ Arbitrary **differentiable** loss function
- **Inner loss:** Loss used to fit next base learner component to current pseudo-residuals
  $\rightarrow$ Typically, ***quadratic loss*** (desirable optimization properties)

Optimization ***Functional gradient descent*** for outer optimization loop, procedure for inner one depending on inner loss

Hyperparameters

- **Ensemble size**, i.e., number of base learners
- **Learning rate**, i.e., impact of single base learner
- **Complexity** of base learners (depending on type used)

Runtime behavior $\mathcal{O}(M \cdot n \cdot d)$ for M base learners, n observations and d features

GRADIENT BOOSTING – PRO'S & CON'S

Advantages

+ Powerful **off-the-shelf** method for supercharging weak learners' performance
+ Translation of most of **base learners'** advantages (e.g., for tree boosting: inherent variable selection, handling of missing data)
+ High predictive **accuracy** that is hard to outperform
+ High **flexibility** (custom loss functions, many tuning options)
+ Applicable to **unbalanced** data

Disadvantages

- Hardly **interpretable** – black-box method
- Hard to **visualize**
- Prone to **overfitting**
- Sensitive to **outliers**
- Hard to **tune** (high sensitivity to variations in hyperparameter values)
- Rather **slow** in training
- Hard to **parallelize**

High-performing predictor, but rather delicate to handle

GRADIENT BOOSTING – PRACTICAL HINTS

XGBoost (extreme gradient boosting)

Fast, efficient implementation of gradient-boosted decision trees that has become **state of the art** for many machine learning problems
$\rightarrow$ Clever modeling techniques + computational speed

Stochastic gradient boosting (SGB)

Faster, approximate version of GB that performs each iteration only on **random subset** of the data
$\rightarrow$ Potentially preferrable for high-dimensional data sets

Implementation

R: packages `gbm`, `xgboost`
Python: `GradientBoostingClassifier` / `GradientBoostingRegressor` from package `scikit-learn`, `XGBClassifier` / `XGBRegressor` from package `xgboost`

Tuning

Overall **limited tunability**
Number of split candidates often more impactful than number of trees

NEURAL NETWORK – FUNCTIONALITY

|UN| SUPERVISED | |NON| PARAMETRIC | NON-LINEAR | BLACK-BOX

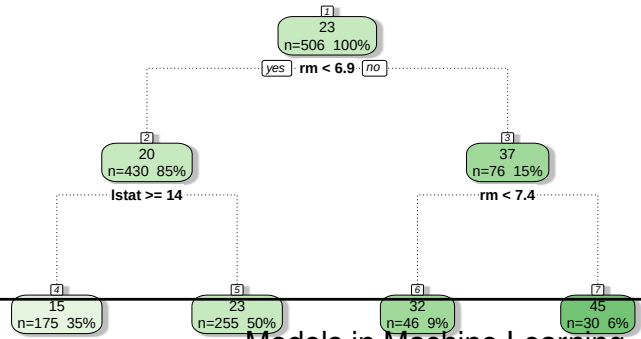
General idea

- A ***neural network (NN)*** is a model architecture loosely inspired by the human brain. It consists of various **neurons**, organized in **layers**. These layers are assembled through functional connections.
- Data enter the network in the **input layer** and sequentially pass through h **hidden layers**, each of which performs a linear **transformation** $\phi^{(j)}$ and a non-linear **activation** $\sigma^{(j)}$.
- The **output layer** receives the repeatedly transformed inputs, performs a final transformation ϕ and outputs predictions (suitably scaled by τ).

Hypothesis space

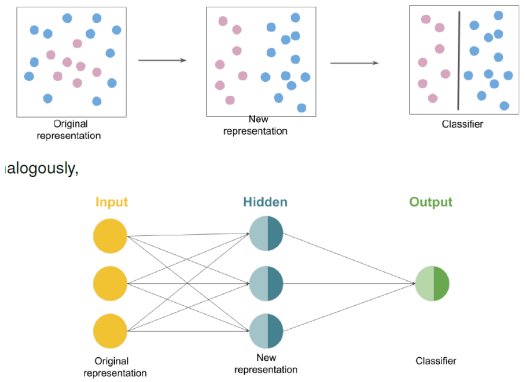
$$\mathcal{H} = \left\{ f(\mathbf{x}) : f(\mathbf{x}) = \tau \circ \phi \circ \sigma^{(h)} \circ \phi^{(h)} \circ \sigma^{(h-1)} \circ \phi^{(h-1)} \circ \ldots \circ \sigma^{(1)} \circ \phi^{(1)}(\mathbf{x}) \right\}$$

- Pass observations along until each ends up in exactly one leaf node
- In each step, find the optimal feature-threshold combination to split by
- Assign response c_t to leaf node t



NEURAL NETWORK – FUNCTIONALITY

- NNs use "hidden layers" between inputs and outputs in order to model intermediary representations of the data:



Empirical risk XX

Optimization NNs are optimised by **backpropagation** which consists of two steps:

- **Forward pass**: Predict result witch current weights and calculate empirical loss.
- **Backward pass**: Calculate error contribution of each weight and update the weights by the negative gradient descent.

Hyperparameters Number of hidden neurons, Dropout, initial weights, activation function, learning rate, number of epochs, batch size

Runtime behavior ???

NEURAL NETWORK – PRO'S & CON'S

Advantages

- \+ can solve complex, non-linear regression or classification problems
- \+ also suitable for unstructed data (e. g. image, audio and text data)
- \+ very accuarate
- \+ can easily be updated
- \+ reduces the need for feature engeneering

Disadvantages

- \- requires a huge amount of data
- \- black-model $\rightarrow$ hard to interpret or explain
- \- computationally expensive $\rightarrow$ slow to train and forecast
- \- tend to overfit
- \- requires much expertise for tuning

Computationally expensive models which can learn complex functions and are suitable for unstructred data like text or pictures.

NEURAL NETWORK – PRACTICAL HINTS

RNNS

XXXX

CNNS

XXXX

Implementation

R: package for regularized linear model `XXX`
Python: `XX` from package `XXX`, package for XXXX `statsmodels.api`