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

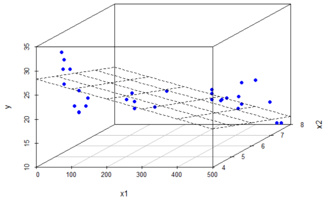
SUPERVISED PARAMETRIC WHITE-BOX

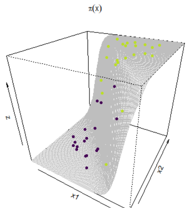
General idea A linear model (LM) depends on a weighted sum of the input variables.

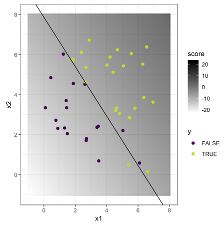
Hypothesis space $\mathcal{H} = \left\{ f : \mathcal{X} \to \mathbb{R} \mid f(\mathbf{x}) = \phi(\boldsymbol{\theta}^\top \mathbf{x}) \right\}$, where $\phi(\cdot) : \mathbb{R} \to \Phi \subseteq \mathbb{R}$ is some transformation function.

For different $\phi(\cdot)$ it results in different models, e.g.:

- identity function $\phi(\boldsymbol{\theta}^\top \mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x}$: **linear regression** - a continuous value is fitted by a **hyperplane** $\theta_0 + \boldsymbol{\theta}^T \mathbf{x}$ (regression task).
- logistic sigmoid function $\phi(\boldsymbol{\theta}^\top \mathbf{x}) = \frac{1}{1+\exp(-\boldsymbol{\theta}^\top \mathbf{x})} = \pi(\mathbf{x})$: **logistic regression** - the probability $\pi(\mathbf{x}) = \mathbb{P}(y = 1 \mid \mathbf{x})$ is fitted by a wave based on the logit link (inverse of logistic sigmoid function). Applying a decision rule (> 0.5) results in a separating hyperplane (classification task, linear classifier).







LINEAR MODEL – FUNCTIONALITY

Empirical risk

- Typically, in **linear regression** the **ordinary least squares (OLS)** with a squared loss function is used for regression: $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(y^{(i)} - \boldsymbol{\theta}^T \mathbf{x}^{(i)} \right)^2$
- Alternatively, the **absolute loss** or the **Huber loss** can be used.
- For **logistic regression** the risk function is based on the **Bernoulli loss** $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -y^{(i)} \log \left(\pi \left(\mathbf{x}^{(i)} \right) \right) - (1 - y^{(i)}) \log \left(1 - \pi \left(\mathbf{x}^{(i)} \right) \right)$.

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 $\mathcal{O}(p^2 \cdot n + p^3)$ for *n* observations and *p* 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

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)
- 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:** `mlr3` learner `LearnerRegrLM`, calling `stats::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.

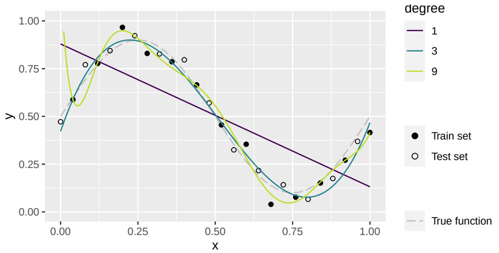
REGULARIZED LM – FUNCTIONALITY

SUPERVISED PARAMETRIC WHITE-BOX

General idea

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

Hypothesis space $\mathcal{H} = \left\{ f : \mathcal{X} \to \mathbb{R} \mid f(\mathbf{x}) = \phi(\boldsymbol{\theta}^\top \mathbf{x}) \right\}$, where $\phi(\cdot) : \mathbb{R} \to \boldsymbol{\Phi} \subseteq \mathbb{R}$ is some transformation function.



REGULARIZED LM – FUNCTIONALITY

Empirical risk

- Therefore, we minimize the empirical risk function $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ **plus the a complexity measure** $J(\boldsymbol{\theta})$: $\mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \cdot J(\boldsymbol{\theta})$.
- **Ridge regression** uses the L2-penalty as a complexity measure 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** uses a convex combination of Ridge and LASSO with $J(\boldsymbol{\theta}) = +\lambda_1\|\boldsymbol{\theta}\|_1 + \lambda_2\|\boldsymbol{\theta}\|_2^2$.

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 [or λ_1 and λ_2 for elastic net]

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

REGULARIZED LM – PRO'S & CON'S

ALMOST SAME LIKE LINEAR MODEL?–> does this make sense?

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
+ less tendency to **overfit**

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. the 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.

Ridge vs. LASSO

- neither is overall better $\rightarrow$ elastic net as a compromise
- **Ridge** works better, if there are many influential and high correlated features.
- In contrast, **LASSO** is more suitable if the underlying structure is sparse (only a few features influence the output y).
- LASSO can set coefficients to zero, thus performing **variable selection**.

Implementation

- **R:** `mlr3` learners `LearnerClassifGlmnet` / `LearnerRegrGlmnet`, calling `glmnet::glmnet()`
- **Python:** `LinearRegression` from package `sklearn.linear_model`, package for advanced statistical parameters `statsmodels.api`

LINEAR SVM – FUNCTIONALITY

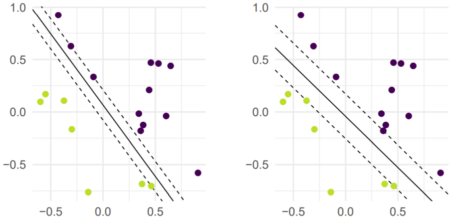
SUPERVISED PARAMETRIC BLACK-BOX

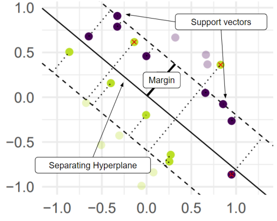
General idea

- The support vector machine (SVM) algorithm finds a decision boundary (**separating hyperplane**) that maximizes the distance (**margin** γ) between the closest members (**support vectors, SV**) of the separate classes. (**hard margin**)
- In a **soft-margin SVM** also "violations" of the margin are allowed $\rightarrow$ not only the margin should be maximized, but also the margin violations minimized.
- There are three different types of training points: **Non-SVs** which do not influence the hyperplane, **SVs** which are exactly on the hyperplane, and **margin violators**.

Hypothesis space

$$\mathcal{H} = \{ f : \mathcal{X} \to \mathbb{R} \mid f(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x} + \theta_0 \}$$





LINEAR SVM – FUNCTIONALITY

Empirical risk

Soft-margin SVMs can also be interpreted as **L2-regularized ERM**:

$$\frac{1}{2}\|\boldsymbol{\theta}\|^2 + C\sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right); \; L(y, f) = \max(1 - yf, 0),$$

with $L(y, f)$ as the hinge loss, $\|\boldsymbol{\theta}\| = 1/\gamma$ and C as cost parameter for margin violation.

Optimization As the problem is **not differentiable**, there are the following solutions:

1. Use a smoothed loss (squared hinge, huber), then do gradient descent.
   NB: Will not create a sparse SVM if we do not add extra tricks.
2. Use **subgradient** methods.
3. Do stochastic subgradient descent.
   Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.

Hyperparameters **C** : penalization for missclassified data points

Runtime behavior $\mathcal{O}(n^3)$ for n observations

LINEAR SVM – PRO'S & CON'S

Advantages

+ high **accuaracy**
+ often **sparse** solution
+ robust against overfitting (**regularized**); especially in high-dimensional space
+ **stable** solutions, as the non-SV do not influence the separating hyperplane

Disadvantages

- **costly implementation**; long training times
- does not scale well to **larger data sets** **??**
- only **linear separation** $\rightarrow$ possible with non-linear SVMs which are explained in the following slides.
- poor **interpretability**

Very accurate solution for high-dimensional data that is linear separable

LINEAR SVM – PRACTICAL HINTS

Prepocessing

Features must be rescaled before applying SVMs.

Tuning

The cost parameter *C* must be tuned, as it has a strong influence on the resulting separating hyperplane.

Implementation

- **R:** `mlr3` learners `LearnerClassifSVM` / `LearnerRegrSVM`, calling `e1071::svm()`
- **Python:** `sklearn.svm.SVC` from package `scikit-learn` / package `libSVM`

NON-LINEAR SVM – FUNCTIONALITY

SUPERVISED | NON PARAMETRIC | BLACK-BOX

General idea

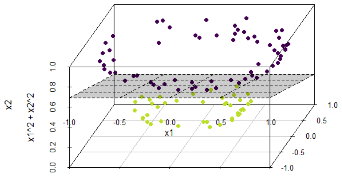
- Non-linear SVMs construct a separating hyperplane in a **higher dimensional dimension**
- **Kernels = feature map + inner product** $k(\mathbf{x}, \tilde{\mathbf{x}})$ transform the input space into a higher dimensional space and calculates the inner product.

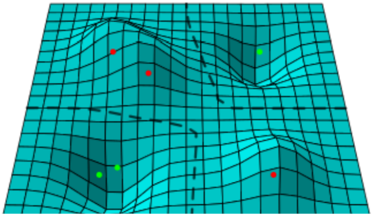
Hypothesis space

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

$$\phi(x_1, x_2) = (x_1, x_2, x_1^2 + x_2^2)$$

$$k(\mathbf{x}, \tilde{\mathbf{x}}) = \exp(-\gamma\|\mathbf{x} - \tilde{\mathbf{x}}\|^2)$$





NON-LINEAR SVM – FUNCTIONALITY

Empirical risk

SVMs can also be interpreted as **L2-regularized ERM**:

$$\frac{1}{2}\|\boldsymbol{\theta}\|^2 + C\sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right);\ L(y,f) = \max(1 - yf, 0),$$

with $L(y, f)$ as the hinge loss, $\|\boldsymbol{\theta}\| = 1/\gamma$ and C as cost parameter for margin violation.

Optimization As the problem is **not differentiable**, there are the following solutions:

1. Use a smoothed loss (squared hinge, huber), then do gradient descent.
   NB: Will not create a sparse SVM if we do not add extra tricks.
2. Use **subgradient** methods.
3. Do stochastic subgradient descent.
   Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.

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 $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ for n observations depending on the kernel

NON-LINEAR SVM – PRO'S & CON'S

Advantages

+ high **accuaracy**
+ can learn **non-linear decision boundaries**
+ often **sparse** solution
+ robust against overfitting (**regularized**); especially in high-dimensional space
+ **stable** solutions, as the non-SV do not influence the separating hyperplane

Disadvantages

- **costly implementation**; long training times
- does not scale well to **larger data sets ??**
- only **linear separation** $\rightarrow$ possible with non-linear SVMs which are explained in the following slides.
- poor **interpretability**
- **not easy tunable** as it is highly important to choose the right kernel

Non-linear SVMs perform very well for non-linear separable data, but are hard to interpret and need a lot of tuning.

NON-LINEAR SVM – PRACTICAL HINTS

Kernels

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)

Tuning

- SVMs are sensitive to its hyperparameters and **should always be tuned**.
- For the RBF kernel the **RBF sigma heuristic** is used.

Implementation

- **R:** `mlr3` learners `LearnerClassifSVM` / `LearnerRegrSVM`, calling `e1071::svm()`
- **Python:** `sklearn.svm.SVC` from package `scikit-learn` / package `libSVM`

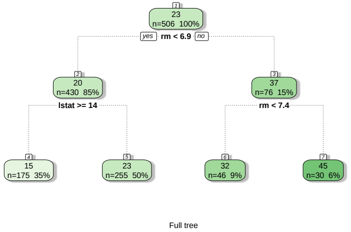
CART – FUNCTIONALITY

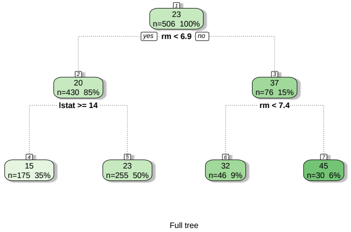
SUPERVISED NON-PARAMETRIC WHITE-BOX FEATURE SELECTION

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 $\mathcal{O}(n^2 \cdot p)$ for n observations and p 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:** `mlr3` learners `LearnerClassifRpart` / `LearnerRegrRpart`, calling `rpart::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**.

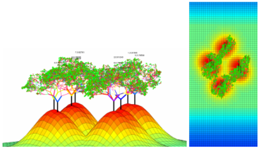
RANDOM FOREST – FUNCTIONALITY

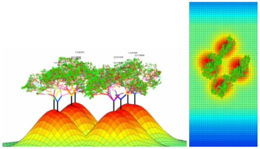
SUPERVISED | NON-PARAMETRIC | BLACK-BOX | FEATURE SELECTION

General idea

- **Random forests (RF)** perform **bagging**: they combine M trees (base learners) to form a strong **ensemble** learner.
- They use **complex** trees with low bias and compensate for the resulting variance by aggregating them in a **decorrelated** manner.
- Each tree is trained on a **bootstrap sample** of the data and only on a random **subset of features** to incur variability.
- Aggregation 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 $\mathcal{O}(M \cdot n^2 \cdot p)$ for M trees, n observations and p 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:** `mlr3` learners `LearnerClassifRanger` / `LearnerRanger`, calling `ranger::ranger()`
- **Python:** `RandomForestClassifier` / `RandomForestRegressor` from package `scikit-learn`

Tuning

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

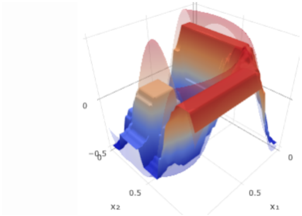
GRADIENT BOOSTING – FUNCTIONALITY

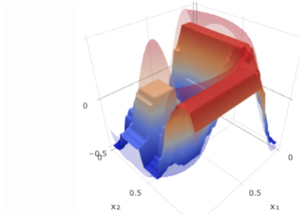
SUPERVISED | NON-PARAMETRIC | BLACK-BOX | FEATURE SELECTION

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.
- Each base learner is fitted to the current **point-wise residuals**
  $\rightarrow$ One boosting iteration $\widehat{=}$ one approximate **gradient step in function space**
- The final model is a weighted sum of base learners $b(\mathbf{x}, \theta^{[m]})$ with weights $\beta^{[m]}$.

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





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 $\mathcal{O}(M \cdot n \cdot p)$ for M base learners, n observations and p 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

Implementation

- **R:** `mlr3` learners `LearnerClassifXgboost` / `LearnerXgboost`, calling `xgboost::  xgboost::xgb.train()`
- **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

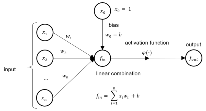
|UN| SUPERVISED |NON| PARAMETRIC BLACK-BOX

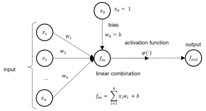
General idea

- 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 weighted functional connections.
- Batches of data enter 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** $\sigma^{(j)}$, thus creating intermediary representations of the data.
- The **output layer** receives the repeatedly transformed inputs, performs a final transformation ϕ and outputs predictions (suitably scaled by τ).

Hypothesis space

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





NEURAL NETWORK – FUNCTIONALITY

Empirical risk Any **differentiable** loss function

Optimization

NNs are optimized by **backpropagation** which consists of two steps:

- **Forward pass**: Predict result with current weights and compute empirical risk according to chosen loss function.
- **Backward pass**: Calculate error contribution of each weight by means of gradient descent – which essentially means applying the chain rule to the composition of functions applied in each layer – and update weights accordingly.

Hyperparameters

- Number of hidden **layers** (depth), number of **neurons** per layer
- **Activation** function(s)
- **Learning rate** for backpropagation
- Number of iterations (**epochs**), **batch** size
- Initial **weights**
- ...

Runtime behavior ???

NEURAL NETWORK – PRO'S & CON'S

Advantages

+ Able to solve **complex, non-linear** regression or classification problems
+ Therefore, typically very good **performance**
+ Built-in **feature extraction** - obtained by intermediary representations
+ Suitable for **unstructured** data (e. g. image, audio, text data)
+ Easy handling of **high-dimensional** or **missing** data
+ **Parallelizable** structure

Disadvantages

- Computationally **expensive** $\rightarrow$ slow to train and forecast
- Large **amounts** of data required
- **Faster-than-linear** scaling of weight matrices with increased network size
- Network architecture requiring much **expertise** in tuning
- **Black-box** model – hard to interpret or explain
- Tendency towards **overfitting**

Able to learn extremely complex functions, but computationally expensive and hard to get right

NEURAL NETWORK – PRACTICAL HINTS

Types of neural networks (RNNs, CNNs)

- **Recurrent neural networks (RNNs**: Deep NN that make use of **sequential** information with temporal **dependencies**
  $\rightarrow$ Frequently applied to **natural language processing**
- **Convolutional neural networks (CNNs)**: Regularized version of the fully connected feed-forward NN (where each neuron is connected to all neurons of the subsequent layer) that abstracts inputs to feature maps via **convolution**
  $\rightarrow$ Frequently applied to **image recognition**

Problem of neural architecture search (NAS)

NN are **not off-the-shelf** methods – the network architecture needs to be tailored to each problem anew
$\rightarrow$ Automated machine learning attempts to learn architectures

Implementation

- **R:** package `neuralnet`
- **Python:** libraries `PyTorch`, `keras`