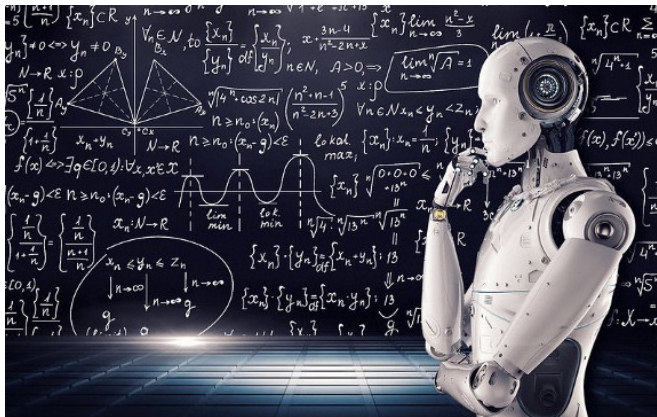


# Common Machine Learning Algorithms



# CONTENTS

- 1 Linear Models (LM)
- 2 Linear Support Vector Machines (SVM)
- 3 Nonlinear Support Vector Machines
- 4  $k$ -Nearest Neighbors ( $k$ -NN)
- 5 Classification & Regression Trees (CART)
- 6 Random Forests
- 7 Gradient Boosting
- 8 Neural Networks (NN)

# LINEAR MODELS (LM)

# LINEAR MODELS – METHOD SUMMARY

REGRESSION

CLASSIFICATION

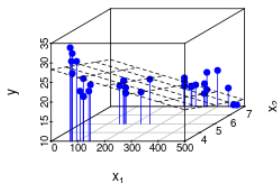
PARAMETRIC

WHITE-BOX

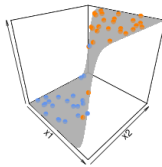
**General idea** Represent target as function of linear predictor  $\theta^T \mathbf{x}$

**Hypothesis space**  $\mathcal{H} = \{f : \mathcal{X} \rightarrow \mathbb{R} \mid f(\mathbf{x}) = \phi(\theta^T \mathbf{x})\}$ , with suitable transformation  $\phi(\cdot)$ , e.g.,

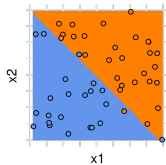
- Identity  $\phi(\theta^T \mathbf{x}) = \theta^T \mathbf{x} \rightarrow$  **linear regression**
- Logistic sigmoid function  $\phi(\theta^T \mathbf{x}) = \frac{1}{1 + \exp(-\theta^T \mathbf{x})} =: \pi(\mathbf{x} \mid \theta) \rightarrow$  **(binary) logistic regression**
  - Probability  $\pi(\mathbf{x} \mid \theta) = \mathbb{P}(y = 1 \mid \mathbf{x})$  of belonging to one of two classes
  - Separating hyperplane via decision rule (e.g.,  $\hat{y} = 1 \Leftrightarrow \pi(\mathbf{x}) > 0.5$ )



Linear regression hyperplane



Logistic function for bivariate input and loss-minimal  $\theta$



Corresponding separating hyperplane

# LINEAR MODELS – METHOD SUMMARY

## Empirical risk

- **Linear regression**

- Typically, based on **quadratic** loss:  $\mathcal{R}_{\text{emp}}(\theta) = \sum_{i=1}^n \left( y^{(i)} - f(\mathbf{x}^{(i)} | \theta) \right)^2$   
⇒ corresponding to ordinary-least-squares (OLS) estimation
- Alternatives: e.g., **absolute** or **Huber** loss (both improving robustness)

- **Logistic regression**: based on **Bernoulli/log/cross-entropy** loss

$$\Rightarrow \mathcal{R}_{\text{emp}}(\theta) = \sum_{i=1}^n -y^{(i)} \log \left( \pi \left( \mathbf{x}^{(i)} \right) \right) - (1 - y^{(i)}) \log \left( 1 - \pi \left( \mathbf{x}^{(i)} \right) \right)$$

## Optimization

- For **OLS**: analytically with  $\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$  ( $\mathbf{X} \in \mathbb{R}^{n \times p}$  : matrix of feature vectors)
- For **other loss functions**: numerical optimization

Hyperparameters    None

# LINEAR MODELS – PRO'S & CON'S

## Advantages

- + **Simple and fast** implementation
- + **Analytical** solution for quadratic loss
- + **Cheap** computation
- + Applicable for any **dataset size**, as long as number of observations  $\gg$  number of features
- + Flexibility **beyond linearity** with polynomials, trigonometric transformations etc.
- + Intuitive **interpretability** via feature effects
- + Statistical hypothesis **tests** for effects available

## Disadvantages

- **Nonlinearity** of many real-world problems
- Further restrictive **assumptions**: linearly independent features, homoskedastic residuals, normality of conditional response
- **Sensitivity** w.r.t. outliers and noisy data (especially with L2 loss)
- Risk of **overfitting** in higher dimensions
- Feature **interactions** must be handcrafted, so higher orders practically infeasible
- No handling of **missing** data

Simple, highly interpretable method, but with strong assumptions, practical limitations, and risk of overfitting

# LINEAR MODELS – REGULARIZATION

## Idea

- Unregularized LM: risk of **overfitting** in high-dimensional space with only few observations
- **Goal**: find compromise between model fit and generalization by adding **penalty term**

## Regularized empirical risk

- Empirical risk function **plus complexity penalty**  $J(\theta)$ , controlled by shrinkage parameter  $\lambda > 0$ :  
 $\mathcal{R}_{\text{reg}}(\theta) := \mathcal{R}_{\text{emp}}(\theta) + \lambda \cdot J(\theta)$ .
- Popular regularizers
  - **Ridge** regression: L2 penalty  $J(\theta) = \|\theta\|_2^2$
  - **LASSO** regression: L1 penalty  $J(\theta) = \|\theta\|_1$

## Optimization under regularization

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

# LINEAR MODELS – REGULARIZATION

## Choice of regularization parameter

- Standard hyperparameter optimization problem
- E.g., choose  $\lambda$  with minimum mean cross-validated error (default in R package `glmnet`)

## Ridge vs. LASSO

- **Ridge**

- Overall smaller, but still dense  $\theta$
- Applicable with large number of influential features, handling correlated variables by shrinking their coefficients to equal degree

- **LASSO**

- Actual variable selection
- Suitable for sparse problems, ineffective with correlated features (randomly selecting one)
- Neither overall better → compromise: **elastic net** = weighted combination of Ridge and LASSO



# LINEAR MODELS – PRACTICAL HINTS

## Implementation

- **R:**
  - **Unregularized:** `mlr3 learner LearnerRegrLM`, calling `stats::lm()` / `mlr3 learner LearnerClassifLogReg`, calling `stats::glm()`
  - **Regularized:** `mlr3 learners LearnerClassifGlmnet / LearnerRegrGlmnet`, calling `glmnet::glmnet()`
- **Python:** `LinearRegression` from package `sklearn.linear_model`, package for advanced statistical parameters `statsmodels.api`

# LINEAR SUPPORT VECTOR MACHINES (SVM)

# LINEAR SVM – METHOD SUMMARY

CLASSIFICATION

PARAMETRIC

BLACK-BOX

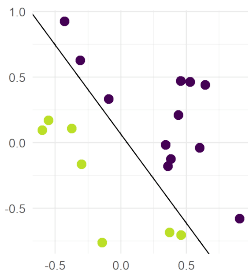
## General idea

- Find linear decision boundary (**separating hyperplane**) that best discriminates classes
  - **Hard-margin** SVM: maximize distance (**margin**  $\gamma > 0$ ) to closest points (**support vectors, SVs**) on each side of decision boundary
  - **Soft-margin** SVM: relax separation to allow for margin violations  $\rightarrow$  maximize margin while minimizing violations
- 3 types of training points
  - **non-SVs** with no impact on decision boundary
  - **SVs** located exactly on decision boundary
  - **margin violators**

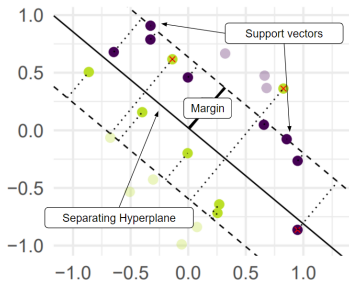
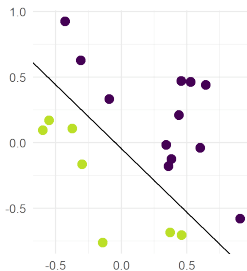
**Hypothesis space**  $\mathcal{H} = \{f : \mathcal{X} \rightarrow \mathbb{R} \mid f(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x} + \theta_0\}$  **separator intercept notwendig?**

**was ist mit regression, auch in nonlinear svm?**

# LINEAR SVM – METHOD SUMMARY



Hard-margin SVM: margin is maximized by boundary on the right



Soft-margin SVM with margin violations

## Dual problem

$$\max_{\alpha \in \mathbb{R}^n} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y^{(i)} y^{(j)} \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C \quad \forall i \in \{1, \dots, n\} \quad (C = \infty \text{ for hard-margin SVM}),$$

$$\sum_{i=1}^n \alpha_i y^{(i)} = 0$$

# LINEAR SVM – METHOD SUMMARY

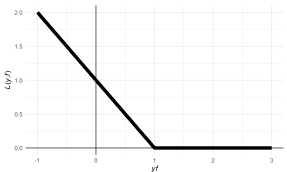
## Empirical risk

Soft-margin SVM also interpretable as **L2-regularized ERM**:

$$\frac{1}{2} \|\boldsymbol{\theta}\|_2^2 + C \sum_{i=1}^n L(y^{(i)}, f(\mathbf{x}^{(i)}))$$

with

- $\|\boldsymbol{\theta}\| = 1/\gamma$ ,
- $C > 0$ : penalization for missclassified data points
- $L(y, f) = \max(1 - yf, 0)$ : **hinge** loss  
→ other loss functions applicable (e.g., **Huber** loss)



## Optimization

- Typically, tackling **dual** problem (though feasible in corresponding primal) via **quadratic programming**
- Popular: **sequential minimal optimization** → iterative algorithm based on breaking down objective into bivariate quadratic problems with analytical solutions

**Hyperparameters**    Cost parameter  $C$

# LINEAR SVM – PRO'S & CON'S

## Advantages

- + Often **sparse** solution
- + Robust against overfitting (**regularized**); especially in high-dimensional space
- + **Stable** solutions, as non-SV do not influence decision boundary

## Disadvantages

- **Costly** implementation; long training times
- **Limited scalability** to larger data sets ??
- Confined to **linear separation**
- Poor **interpretability**
- No handling of **missing** data

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

# LINEAR SVM – PRACTICAL HINTS

**Preprocessing** Features must be rescaled before applying SVMs.

**Tuning** Cost parameter  $C$  must be tuned and has strong influence on resulting separating hyperplane.

## Implementation

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

# NONLINEAR SUPPORT VECTOR MACHINES



# NONLINEAR SVM – METHOD SUMMARY

CLASSIFICATION

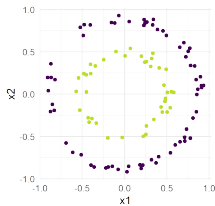
NONPARAMETRIC

BLACK-BOX

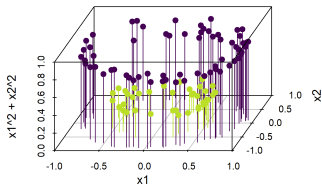
## General idea

- Move **beyond linearity** by mapping data to transformed space where they are linearly separable
- **Kernel trick** (based on Mercer's theorem, existence of reproducing kernel Hilbert space):
  - Replace two-step operation feature map  $\phi \rightsquigarrow$  inner product by **kernel**  $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ , s.t.  
 $\langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle = k(\mathbf{x}, \tilde{\mathbf{x}})$
  - No need for explicit construction of feature maps; very fast and flexible

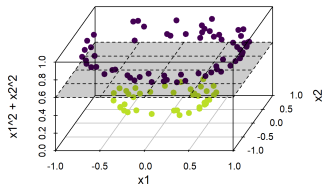
Hypothesis space  $\mathcal{H} = \left\{ f(\mathbf{x}) = \sum_{i=1}^n \alpha_i y^{(i)} k(\mathbf{x}^{(i)}, \mathbf{x}) + \theta_0 \mid \theta_0, \alpha_i \in \mathbb{R} \right\}$



Nonlinear problem in original space



Mapping to 3D space and subsequent linear separation – implicitly handled by kernel in nonlinear SVM



# NONLINEAR SVM – METHOD SUMMARY

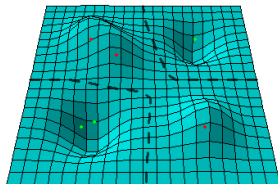
**Dual problem**    **Kernelize** dual (soft-margin) SVM problem, replacing all inner products by kernels:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y^{(i)} y^{(j)} k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}), \quad \text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^n \alpha_i y^{(i)} = 0.$$

**Hyperparameters**    Cost  $C$  of margin violations, kernel hyperparameters (e.g., width of RBF kernel)

**Interpretation as basis function approach**

- **Representer theorem:** dual soft-margin SVM problem expressible through  $\theta = \sum_{j=1}^n \beta_j \phi(\mathbf{x}^{(j)})$
- Sparse, weighted sum of **basis functions** with  $\beta_j = 0$  for non-SVs
- Result: **local** model with smoothness depending on kernel properties



RBF kernel as mixture of Gaussian basis functions, forming bumpy, nonlinear decision surface to discern red and green points

# NONLINEAR SVM – PRO'S & CON'S

## Advantages

- + high **accuracy**
- + can learn **nonlinear 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** → possible with nonlinear SVMs which are explained in the following slides.
- poor **interpretability**
- **not easy tunable** as it is highly important to choose the right kernel
- No handling of **missing data**

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

# NONLINEAR SVM – PRACTICAL HINTS

## Common kernels

- **Linear** kernel: dot product of given observations  $\rightarrow k(\mathbf{x}, \tilde{\mathbf{x}}) = \mathbf{x}^\top \tilde{\mathbf{x}} \Rightarrow$  linear SVM
- **Polynomial** kernel of degree  $d \in \mathbb{N}$ : monomials (i.e., feature interactions) up to  $d$ -th order  $\rightarrow k(\mathbf{x}, \tilde{\mathbf{x}}) = (\mathbf{x}^\top \tilde{\mathbf{x}} + b)^d, b \geq 0$
- **Radial basis function (RBF)** kernel: infinite-dimensional feature space, allowing for perfect separation of all finite datasets  $\rightarrow k(\mathbf{x}, \tilde{\mathbf{x}}) = \exp(-\gamma \|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2)$  with bandwidth parameter  $\gamma > 0$

## Tuning

- High sensitivity w.r.t. hyperparameters, especially those of kernel  $\rightarrow$  **tuning** very important
- For RBF kernels, use **RBF sigma heuristic** to determine bandwidth

## Implementation

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

# ***K*-NEAREST NEIGHBORS (*K*-NN)**

# K-NN – METHOD SUMMARY

REGRESSION

CLASSIFICATION

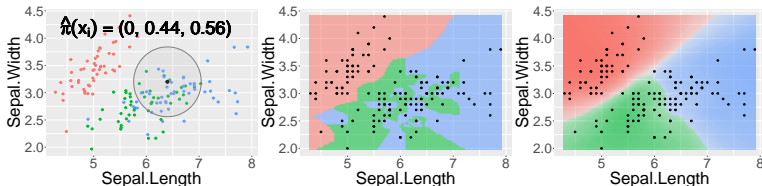
NONPARAMETRIC

WHITE-BOX

## General idea

- Rationale: **similarity** in feature space  $\rightsquigarrow$  similarity in target space w.r.t. certain **metric**
- **Prediction** for  $\mathbf{x}$ : construct  **$k$ -neighborhood**  $N_k(\mathbf{x})$  from  $k$  points closest to  $\mathbf{x}$  in  $\mathcal{X}$ , then predict
  - (weighted) mean target for **regression**:  $\hat{y} = 1 / (\sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} w_i) \sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} w_i y^{(i)}$
  - most frequent class for **classification**:  $\hat{y} = \arg \max_{\ell \in \{1, \dots, g\}} \sum_{i:\mathbf{x}^{(i)} \in N_k(\mathbf{x})} \mathbb{I}(y^{(i)} = \ell)$
- No distributional or functional **assumptions**
- **Nonparametric** behavior: parameters = training data; no compression of information

## Hyperparameters Neighborhood size $k$ (locality), distance measure



Left: Neighborhood for exemplary observation in iris,  $k = 50$   
Right: Prediction surfaces for  $k \in \{1, 50\}$

# K-NN – PRO'S & CON'S

## Advantages

- + **Easy** to explain and implement
- + No functional **assumptions** – therefore (in theory) able to model data situations of **arbitrary complexity**
- + No **training** period
- + No **optimization** required
- + Constant evolvement with **new data**
- + Ability to learn **nonlinear** decision boundaries
- + Easy to **tune**

## Disadvantages

- Sensitivity w.r.t. **noisy** or **irrelevant** features and outliers due to utter reliance on distances
- Bad performance when feature **scales** not consistent with importance
- Heavily afflicted by **curse of dimensionality**
- No handling of **missing** data
- Poor handling of data **imbalances** (worse for more global model, i.e., large  $k$ )
- High **memory** consumption of distance computation

Easy and intuitive for small, well-behaved datasets with meaningful feature space distances

# K-NN – PRACTICAL HINTS

## Popular distance measures

- Numerical features: typically, **Minkowski** distances  $d(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|_q = \left( \sum_j |x_j - \tilde{x}_j|^q \right)^{\frac{1}{q}}$ 
  - $q = 1$ : **Manhattan** distance  $\rightarrow d(\mathbf{x}, \tilde{\mathbf{x}}) = \sum_j |x_j - \tilde{x}_j|$
  - $q = 2$ : **Euclidean** distance  $\rightarrow d(\mathbf{x}, \tilde{\mathbf{x}}) = \sqrt{\sum_j (x_j - \tilde{x}_j)^2}$
- In presence of categorical features: **Gower** distance
- **Custom** distance measures applicable
- Optional **weighting** to account for beliefs about varying feature importance

## Implementation

- **R**: `mlr3` learners `LearnerClassifKNN` / `LearnerRegrKNN`, calling `knnc : knnc()`
- **Python**: `KNeighborsClassifier` / `KNeighborsRegressor` from package `scikit-learn`



# **CLASSIFICATION & REGRESSION TREES (CART)**

# CART – METHOD SUMMARY

REGRESSION

CLASSIFICATION

NONPARAMETRIC

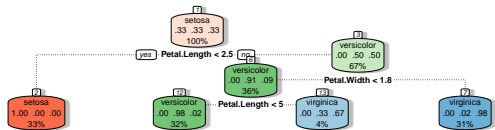
WHITE-BOX

FEATURE SELECTION

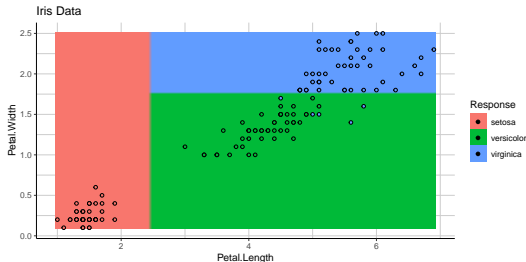
## General idea

- Starting from root node containing all data, perform repeated **binary splits**, subsequently dividing input space into **rectangular partitions**  $Q_t$ 
  - In each step, find **optimal split** (feature-threshold combination) → greedy search
  - Assign same response  $c_t$  to all observations in terminal region  $Q_t$
- Splits based on node **impurity**, equivalently interpretable as **ERM**

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



Classification tree for iris data after 3 splits



Corresponding prediction surface with axis-aligned boundaries

# CART – METHOD SUMMARY

## Empirical risk

- Calculated for each potential terminal node  $\mathcal{N}_t$  of a split
- In general, compatible with arbitrary losses – typical choices:

- $g$ -way classification:

- **Brier score**  $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x}, y) \in \mathcal{N}_t} \sum_{k=1}^g \hat{\pi}_k^{(\mathcal{N}_t)} (1 - \hat{\pi}_k^{(\mathcal{N}_t)}) \rightarrow$  **Gini** impurity

- **Bernoulli loss**  $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x}, y) \in \mathcal{N}_t} - \sum_{k=1}^g \hat{\pi}_k^{(\mathcal{N}_t)} \log \hat{\pi}_k^{(\mathcal{N}_t)} \rightarrow$  **entropy** impurity

- Regression: **quadratic** loss  $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x}, y) \in \mathcal{N}_t} (y - c_t)^2$

## Optimization

- **Exhaustive** search over all split candidates, choice of risk-minimal split
- In practice: limit number of candidates, use tricks to avoid combinatorial explosion

**Hyperparameters**    **Complexity**, i.e., number of leaves  $T$  (controlled indirectly, see *Implementation*)

# CART – PRO'S & CON'S

## Advantages

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

## Disadvantages

- Rather **poor generalization** when used stand-alone
- High **variance/instability**: strong dependence on training data
- Substantial risk of **overfitting**
- Not well-suited for modeling **linear** relationships
- **Bias** toward features with many categories

Simple, good with feature selection and highly interpretable, but not the most performant learner

# CART – PRACTICAL HINTS

## Complexity control

- Unless interrupted, splitting continues until we have one observation per leaf node (costly + overfitting)
- Limit tree growth via
  - **Early stopping:** stop growth prematurely  
→ hard to determine good stopping point before actually trying all combinations
  - **Pruning:** grow to large size and cut back in risk-optimal manner

**Bagging / boosting** As CART are highly **instable** predictors on their own, they are typically used as base learners in bagging (random forest) or boosting ensembles.

## 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, ...

# RANDOM FORESTS

# RANDOM FORESTS – METHOD SUMMARY

REGRESSION

CLASSIFICATION

NONPARAMETRIC

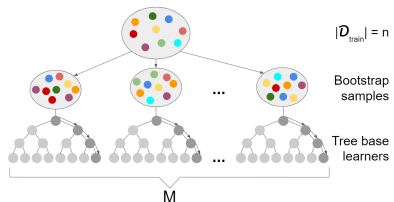
BLACK-BOX

FEATURE SELECTION

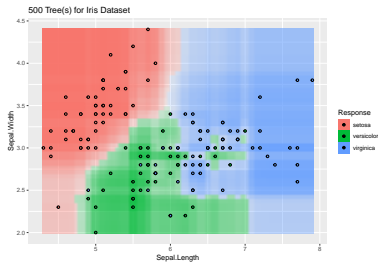
## General idea

- Combine  $M$  tree **base learners** into **bagging ensemble**, fitting same learner on **bootstrap** data samples
  - Use unstable, **high-variance** base learners  $\rightarrow$  let trees grow to full size
  - Mitigate individual trees' bias by promoting **decorrelation**  $\rightarrow$  use random subset of candidate features for each split
- **Predict** via averaging (regression) or majority vote (classification)

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



Schematic depiction of bagging process



Prediction surface for iris data with 500-tree ensemble

# RANDOM FORESTS – METHOD SUMMARY

## 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 split candidates in each node of each tree to minimize empirical risk in child nodes (greedy optimization)

## Hyperparameters

- **Ensemble size**, i.e., number of trees
- **Complexity** of base learners
- **Number of split candidates**, i.e., number of features to be considered at each split  
→ frequently used heuristics with total of  $p$  features:  $\lfloor \sqrt{p} \rfloor$  for classification,  $\lfloor p/3 \rfloor$  for regression

## Out-of-bag (OOB) error

- Compute ensemble prediction for observations outside individual trees' bootstrap training sample  
→ unseen test points
- Use resulting loss as unbiased estimate of **generalization error**



# RANDOM FORESTS – PRO'S & CON'S

## Advantages

- + Translation of most of **trees'** advantages (e.g., feature selection, feature interactions)
- + Fairly good **good predictors**: mitigating base learners' weakness through bagging
- + Quite **stable** w.r.t. changes in data
- + Good with **high-dimensional** data, even in presence of noisy covariates
- + Easy to **parallelize**
- + Rather easy to **tune**
- + Intuitive measures of **feature importance**

## Disadvantages

- Loss of trees' **interpretability** – black-box method
- Hard to **visualize**
- Often suboptimal for **regression**
- **Bias** toward features with many categories
- Often still inferior in **performance** to other methods (e.g., boosting)

Fairly good and stable predictor with built-in feature selection, but black-box method

# RANDOM FORESTS – PRACTICAL HINTS

**Pre-processing** Inherent feature selection, but high **computational cost** for large number of features  
→ upstream feature selection (e.g., via PCA) might be advisable

## Feature importance

- Based on **improvement in split criterion**: aggregate improvements by all splits using  $j$ -th feature
- Based on **permutation**: permute  $j$ -th feature in OOB observations and compute impact on OOB error

**Tuning** Number of split candidates often more impactful than number of trees

## Implementation

- **R**: `mlr3` learners `LearnerClassifRanger` / `LearnerRegrRanger`, calling `ranger::ranger()`
- **Python**: `RandomForestClassifier` / `RandomForestRegressor` from package `scikit-learn`

# GRADIENT BOOSTING

# GRADIENT BOOSTING – METHOD SUMMARY

REGRESSION

CLASSIFICATION

(NON)PARAMETRIC

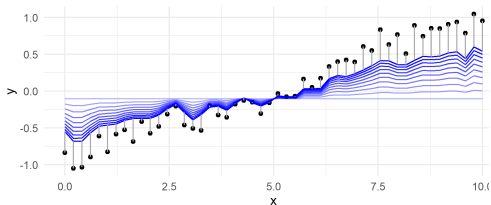
BLACK-BOX

FEATURE SELECTION

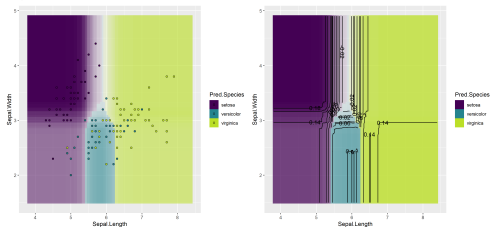
## General idea

- Create **ensemble** in **sequential**, stage-wise manner
  - In each iteration, add new model component in risk-minimal fashion
  - Final model: weighted sum of base learners (frequently, **CART**)
- Fit each base learner to current **point-wise residuals**  
→ one boosting iteration  $\hat{=}$  one approximate **gradient step in function space**

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



Boosting prediction function with GAM base learners for univariate regression problem after 10 iterations



Boosting prediction surface with tree base learners for iris data after 100 iterations (*right*: contour lines of discriminant functions)

# GRADIENT BOOSTING – METHOD SUMMARY

## Empirical risk

- **Outer loss** used to compute pseudo-residuals – error of current model fit  
→ arbitrary **differentiable** loss function
- **Inner loss** used to fit next base learner component to current pseudo-residuals  
→ typically, **quadratic loss**

**Optimization**    **Functional gradient descent** for outer optimization loop

## Hyperparameters

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

# GRADIENT BOOSTING – PRO'S & CON'S

## Advantages

- + Powerful **off-the-shelf** method for supercharging weak learners' performance
- + High predictive **performance** that is hard to outperform
- + Translation of most of **base learners'** advantages
- + High **flexibility** (custom loss functions, many tuning options)

## Disadvantages

- Hard to **interpret** – 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 and flexible predictor, but rather delicate to handle

# GRADIENT BOOSTING – PRACTICAL HINTS

## XGBoost (extreme gradient boosting)

- Fast, efficient implementation of gradient-boosted decision trees
- **State of the art** for many machine learning problems

**Stochastic gradient boosting (SGB)** Faster, approximate version of GB that performs each iteration only on **random data subset**

Tuning [Tipps??](#)

## Implementation

- **R:** `mlr3` learners `LearnerClassifXgboost / LearnerRegrXgboost`, calling `xgboost::xgb.train()`
- **Python:** `GradientBoostingClassifier / GradientBoostingRegressor` from package `scikit-learn`, `XGBClassifier / XGBRegressor` from package `xgboost`

# NEURAL NETWORKS (NN)





# NEURAL NETWORKS – METHOD SUMMARY

**Empirical risk** Any **differentiable** loss function

## Optimization

- Variety of different optimizers, mostly based on some form of **stochastic gradient descent**
- Backbone: gradient computation for arbitrary functions via **computational graphs**

**NN types** Large variety of architectures for different purposes

- **Feedforward NNs / multi-layer perceptrons (MLPs)**: sequence of **fully-connected** layers
- **Convolutional NNs (CNNs)**: sequence of feature map extractors with spatial awareness → images
- **Recurrent NNs (RNNs)**: handling of sequential, variable-length information → times series, text, audio
- Unsupervised: **autoencoders**, **generative adversarial networks (GANs)**, ...

## Hyperparameters

- Regarding **architecture**
  - Lots of design choices → tuning problem of its own: **neural architecture search (NAS)**
  - E.g., network depth, layer types, activation functions, ...
- Regarding **optimization & regularization**
  - Crucial due to **overparameterization** and strong **nonconvexity**
  - E.g., weight initialization, choice of optimizer, learning rate, batch size, number of epochs, ...

# NEURAL NETWORKS – PRO'S & CON'S

## Advantages

- + Applicable to **complex, nonlinear** problems
- + Very **versatile** w.r.t. architectures
- + Suitable for **unstructured** data (e.g., images)
- + Strong **performance** if done right
- + Built-in **feature extraction**, obtained by intermediate representations
- + Easy handling of **high-dimensional** data
- + **Parallelizable** training

## Disadvantages

- Typically, high computational **cost**
- High demand for **training data**
- Strong tendency to **overfit**
- Requiring lots of **tuning expertise**
- **Black-box** model – hard to interpret or explain

Able to solve extremely complex tasks, but computationally expensive and hard to get right

# NEURAL NETWORKS – PRACTICAL HINTS

## Some options for regularization

- Control weight magnitude with **weight decay** (L2 regularization)
- Interrupt training when validation error starts to pick up → **early stopping**
- Use **dropout** to deactivate neurons at random, thus down-sizing network
- Expand training data and enforce invariances via **augmentation**
- ...

## Optimization tricks

- Accelerate training by incorporating **momentum**
- Control learning rate with **schedulers**, or keep it **adaptive**
- Use **batch normalization** for stability by keeping input distributions fixed throughout transformations
- ...

## Implementation

- **R:** packages `reticulate`, `neuralnet`
- **Python:** libraries `PyTorch` and `PyTorch Lightning`, `TensorFlow` (high-level API: `keras`)