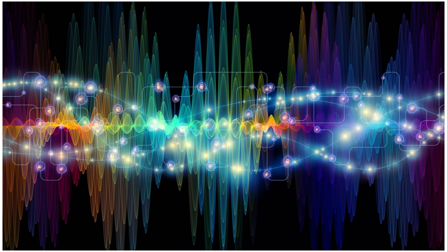
Important Learners in ML



Learning goals

- General idea of important ML algorithms
- Overview of strengths and weaknesses

CONTENTS

1 *k*-Nearest Neighbors (*k*-NN)

2 Generalized Linear Models (GLM)

3 Classification & Regression Trees (CART)

4 Random Forests

5 Gradient Boosting

6 Linear Support Vector Machines (SVM)

7 Nonlinear Support Vector Machines

8 Gaussian Processes (GP)

9 Neural Networks (NN)

K-NN – METHOD SUMMARY

REGRESSION CLASSIFICATION NONPARAMETRIC WHITE-BOX

General idea

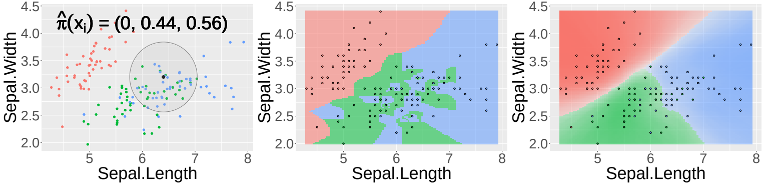
- **similarity** in feature space (w.r.t. certain **distance metric** $d(\mathbf{x}^{(i)}, \mathbf{x})$) $\rightsquigarrow$ similarity in target space
- **Prediction** for $\mathbf{x}$: construct k**-neighborhood** $N_k(\mathbf{x})$ from k points closest to $\mathbf{x}$ in $\mathcal{X}$, then predict
  - (weighted) mean target for **regression**: $\hat{y} = \frac{1}{\sum\limits_{i: \mathbf{x}^{(i)} \in N_k(\mathbf{x})} w_i} \sum\limits_{i: \mathbf{x}^{(i)} \in N_k(\mathbf{x})} w_i y^{(i)}$ with $w_i = \frac{1}{d(\mathbf{x}^{(i)}, \mathbf{x})}$

    $\rightarrow$ optional: higher weights w_i for close neighbors
  - most frequent class for **classification**: $\hat{y} = \underset{\ell \in \{1, \ldots, g\}}{\arg\max} \sum\limits_{i: \mathbf{x}^{(i)} \in N_k(\mathbf{x})} \mathbb{I}(y^{(i)} = \ell)$

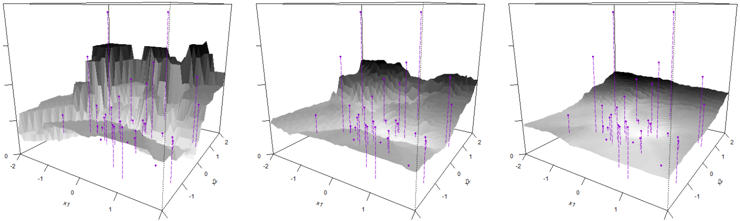
    $\Rightarrow$ Estimating posterior probabilities as $\hat{\pi}_\ell(\mathbf{x}^{(i)}) = \frac{1}{k} \sum\limits_{i: \mathbf{x}^{(i)} \in N_k(\mathbf{x})} \mathbb{I}(y^{(i)} = \ell)$
- **Nonparametric** behavior: parameters = training data; no compression of information
- Not immediately interpretable, but inspection of neighborhoods can be revealing

K-NN – METHOD SUMMARY

Hyperparameters Neighborhood **size** k (locality), **distance** metric (next page)



Classification
Left: Neighborhood for exemplary observation in `iris`, $k = 50$
Middle: Prediction surface for $k = 1$
Right: Prediction surface for $k = 50$



Regression
Left: Prediction surface for $k = 3$
Middle: Prediction surface for $k = 7$
Right: Prediction surface for $k = 15$

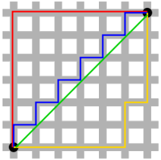
- Small k ⇒ very local, "wiggly" decision boundaries
- Large k ⇒ rather global, smooth decision boundaries

K-NN – METHOD SUMMARY

Popular distance metrics

- Numerical feature space:

  $\Rightarrow$ 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 $\rightarrow d(\mathbf{x}, \tilde{\mathbf{x}}) = \sqrt{\sum_j (x_j - \tilde{x}_j)^2}$
  - Visualization: Manhattan (red, blue, yellow) vs. Euclidean (green)

- Mixed feature space:
  - **Gower distance** can handle numerical and categorical features, and missing data:

    - numerical: $d(x_i, x_j) = \dfrac{|x_i - x_j|}{\max(x) - \min(x)}$

    - categorical: $d(x_i, x_j) = \begin{cases} 1, \text{if } x_i \neq x_j \\ 0, \text{if } x_i = x_j \end{cases}$

    - Gower distance as average over individual scores

- Optional **weighting** to account for beliefs about varying feature importance

Figure Source: https://es.m.wikipedia.org/wiki/Archivo:Manhattan_distance.svg

K-NN – IMPLEMENTATION & PRACTICAL HINTS

Preprocessing Features should be standardized or normalized

Implementation

- **R:** `mlr3` learners (calling `kknn::kknn()`)
  - **Classification:**
    - `LearnerClassifKKNN`
    - `fnn::knn()`
  - **Regression:**
    - `LearnerRegrKKNN`
    - `fnn::knn.reg()`
  - Nearest Neighbour Search in $\mathcal{O}(N \log N)$: `RANN::nn2()`
- **Python:** From package `sklearn.neighbors`
  - **Classification:**
    - `KNeighborsClassifier()`
    - `RadiusNeighborsClassifier()` as alternative if data not uniformly sampled
  - **Regression:**
    - `KNeighborsRegressor()`
    - `RadiusNeighborsRegressor()` as alternative if data not uniformly sampled

K-NN – PROS & CONS

Advantages

- + Algorithm **easy** to explain and implement
- + No distributional or functional **assumptions** → able to model data of **arbitrary complexity**
- + No **training** or **optimization** required
- + **local model** → **nonlinear** decision boundaries
- + Easy to **tune** (few hyperparameters) → number of neighbors *k*, distance metric
- + **Custom** distance metrics can often be easily designed to incorporate domain knowledge

Disadvantages

- – Sensitivity w.r.t. **noisy** or **irrelevant** features and outliers due to dependency on distance measure
- – Heavily affected by **curse of dimensionality**
- – Bad performance when feature **scales** are not consistent with feature relevance
- – Poor handling of data **imbalances** (worse for more global model, i.e., large *k*)

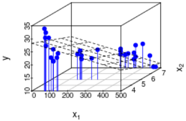
GENERALIZED LINEAR MODELS – METHOD SUMMARY

REGRESSION CLASSIFICATION PARAMETRIC WHITE-BOX FEATURE SELECTION

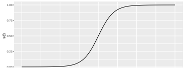
General idea Represent target as function of linear predictor $\boldsymbol{\theta}^\top \mathbf{x}$ (weighted sum of features)
$\rightarrow$ **Interpretation:** if feature x_j increases by 1 unit, the linear predictor changes by θ_j units

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

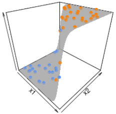
- **Linear Regression**: $\mathcal{Y} = \mathbb{R}$, ϕ identity
- **Logistic Regression**: $\mathcal{Y} = \{0, 1\}$, logistic sigmoid $\phi(\boldsymbol{\theta}^\top \mathbf{x}) = \frac{1}{1+\exp(-\boldsymbol{\theta}^\top \mathbf{x})} =: \pi(\mathbf{x} \mid \boldsymbol{\theta})$
  $\Rightarrow$ Decision rule: Linear hyperplane



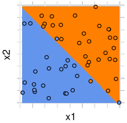
Linear regression hyperplane



Logistic sigmoid function



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



Corresponding separating hyperplane

GENERALIZED LINEAR MODELS – METHOD SUMMARY

Loss functions

- **Lin. Regr.**:
  - Typically, based on **quadratic** loss (OLS estimation):

$$L(y, f) = (y - f)^2$$

- **Log. Regr.**: Based on **bernoulli** / **log** / **cross-entropy** loss
  - Loss based on scores

$$\begin{aligned} L(y, f) &= \ln(1 + \exp(-y \cdot f)) \quad \text{for } y \in \{-1, +1\} \\ L(y, f) &= -y \cdot f + \log(1 + \exp(f)) \quad \text{for } y \in \{0, 1\} \end{aligned}$$

  - Loss based on probabilities:

$$\begin{aligned} L(y, \pi) &= \ln(1 + \exp(-y \cdot \log(\pi))) \quad \text{for } y \in \{-1, +1\} \\ L(y, \pi) &= -y \log(\pi) - (1 - y) \log(1 - \pi) \quad \text{for } y \in \{0, 1\} \end{aligned}$$

GENERALIZED LINEAR MODELS – METHOD SUMMARY

Optimization

- Minimization of the empirical risk
- For **OLS**: analytical solution $\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$
- For other loss functions:
  - **Log. Regr.**: Convex problem, solvable via second-order optimization methods (e.g. BFGS)
  - **Else**: Numerical optimization

Multi-class extension of logistic regression

- Estimate **class-wise** scoring functions: $\Rightarrow \pi : \mathcal{X} \to [0,1]^g,\ \pi(\mathbf{x}) = (\pi_1(\mathbf{x}), \dots, \pi_g(\mathbf{x})),\ \sum\limits_{k=1}^{g} \pi_k(\mathbf{x}) = 1$
- Achieved through **softmax** transformation: $\pi_k(\mathbf{x} \mid \boldsymbol{\theta}) = \exp(\boldsymbol{\theta}_k^\top \mathbf{x}) \big/ \sum_{j=1}^{g} \exp(\boldsymbol{\theta}_j^\top \mathbf{x})$
- Multi-class log-loss: $L\left(y, \pi(\mathbf{x})\right) = -\sum\limits_{k=1}^{g} \mathbb{I}_{\{y=k\}} \log(\pi_k(\mathbf{x}))$
- Predict class with maximum score (or use thresholding variant)

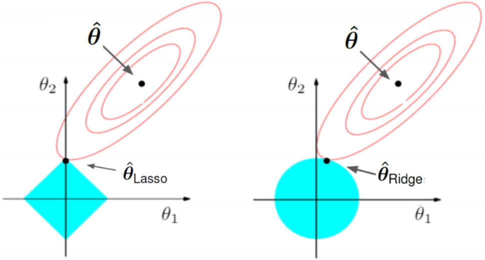
GENERALIZED LINEAR MODELS – REGULARIZATION

General idea

- Unregularized LM: risk of **overfitting** in high-dimensional space with only few observations
- **Goal**: avoidance of overfitting by adding **penalty term**

Regularized empirical risk

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


$\hat{\boldsymbol{\theta}}$
θ_2
θ_1
$\hat{\boldsymbol{\theta}}_{\text{Lasso}}$
$\hat{\boldsymbol{\theta}}$
θ_2
θ_1
$\hat{\boldsymbol{\theta}}_{\text{Ridge}}$

Optimization under regularization

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

Choice of regularization parameter

- Standard hyperparameter optimization problem
- E.g., choose λ with minimum mean cross-validated error

GENERALIZED LINEAR MODELS – REGULARIZATION

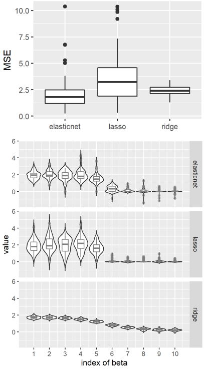
Ridge vs. LASSO

- **Ridge**
  - Global shrinkage ⇒ overall smaller but still dense $\boldsymbol{\theta}$
  - Applicable with large number of influential features, handling correlated variables by shrinking their coefficients by equal amount
- **LASSO**
  - Actual variable selection by shrinking coefficients for irrelevant features all the way to zero
  - Suitable for sparse problems, ineffective with correlated features (randomly selecting one)
- Neither overall better ⇒ compromise: **elastic net**
  - Weighted combination of Ridge and LASSO
  - Introducing additional penalization coefficient: $\mathcal{R}_{\text{reg}}(\boldsymbol{\theta}) = \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) + \lambda \cdot P_\alpha(\boldsymbol{\theta})$, with $P_\alpha(\boldsymbol{\theta}) = [\alpha \cdot \|\boldsymbol{\theta}\|_1 + (1-\alpha) \cdot \frac{1}{2} \cdot \|\boldsymbol{\theta}\|_2^2]$

Ridge performs better for correlated features:

$\boldsymbol{\beta} = (\underbrace{2, \ldots, 2}_{5}, \underbrace{0, \ldots, 0}_{5})$

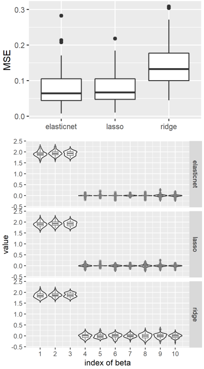
$\text{cor}(\mathbf{X}_i, \mathbf{X}_j) = 0.8^{|i-j|}, \forall i, j$



Lasso performs better for uncorrelated features:

$\boldsymbol{\beta} = (2, 2, 2, \underbrace{0, \ldots, 0}_{7})$

$\text{cor}(\mathbf{X}_i, \mathbf{X}_j) = 0, \forall i \neq j$



GENERALIZED LINEAR MODELS – IMPLEMENTATION

Implementation

- **R:**
  - **Unregularized:** `mlr3` learner `LearnerRegrLM`, calling `stats::lm()` / `mlr3` learner `LearnerClassifLogReg`, calling `stats::glm()`
  - **Regularized** / **ElasticNet:** `mlr3` learners `LearnerClassifGlmnet` / `LearnerRegrGlmnet`, calling `glmnet::glmnet()`
  - For **large classification** data: `mlr3` learner `LearnerClassifLiblineaR`, calling `LiblineaR::LiblineaR()` uses fast coordinate descent
- **Python:** From package `sklearn.linear_model`
  - **Unregularized:**
    - `LinearRegression()`
    - `LogisticRegression(penalty = None)`
  - **Regularized:**
    - *Linear regression:* `Lasso(),Ridge(),ElasticNet()`
    - *Logistic regression:* `LogisticRegression(penalty = {‘l1’, ‘l2’, ‘elasticnet’})`
  - Package for advanced **statistical** models: `statsmodels.api`

GENERALIZED LINEAR MODELS – PROS & CONS

Advantages

- + **Simple and fast** implementation
- + **Analytical** solution for L2 loss
- + Applicable for any **dataset size**, as long as number of observations $\gg$ number of features
- + Flexibility **beyond linearity** with polynomials, trigonometric transformations, interaction terms etc.
- + Intuitive **interpretability** via feature effects
- + Statistical hypothesis **tests** for effects available

Disadvantages

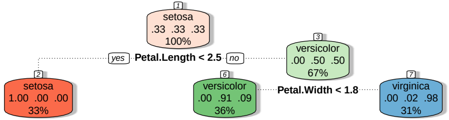
- — **Nonlinearity** of many real-world problems
- — Further restrictive **assumptions**: linearly independent features, homoskedastic residuals, normality of conditional response
- — **Sensitivity** w.r.t. outliers and noisy data (especially with L2 loss)
- — Also a LM can **overfit** (e.g., many features and few observations)
- — Feature **interactions** must be handcrafted $\rightarrow$ practically infeasible for higher orders

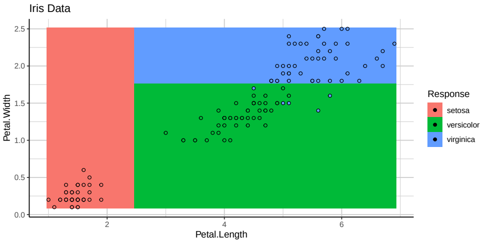
CART – METHOD SUMMARY

REGRESSION | CLASSIFICATION | NONPARAMETRIC | WHITE-BOX | FEATURE SELECTION

General idea (CART – Classification and Regression Trees)

- Start at root node containing all data
- Perform repeated **axis-parallel binary splits** in feature space to obtain **rectangular partitions** at terminal nodes $Q_1, \dots, Q_M$
- Splits based on reduction of node **impurity**
  → empirical risk minimization (**ERM**)
- In each step:
  - Find **optimal split** (feature-threshold combination)
    → greedy search
  - Assign constant prediction c_m to all obs. in Q_m
    → Regression: c_m is average of y
    → Classif.: c_m is majority class (or class proportions)
  - Stop when a pre-defined criterion is reached
    → See Complexity control

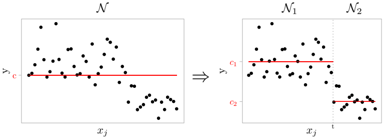




Hypothesis space
$$\mathcal{H} = \left\{ f(\mathbf{x}) : f(\mathbf{x}) = \sum_{m=1}^{M} c_m \mathbb{I}(\mathbf{x} \in Q_m) \right\}$$

CART – METHOD SUMMARY

Empirical risk



- Splitting **feature** x_j **at split point** t divides a parent node $\mathcal{N}$ into two child nodes:

$$\mathcal{N}_1 = \{(\mathbf{x}, y) \in \mathcal{N} : x_j \leq t\} \text{ and } \mathcal{N}_2 = \{(\mathbf{x}, y) \in \mathcal{N} : x_j > t\}$$

- Compute empirical risks in child nodes and minimize their sum to find best split (impurity reduction):

$$\arg\min_{j,t} \mathcal{R}(\mathcal{N}, j, t) = \arg\min_{j,t} \mathcal{R}(\mathcal{N}_1) + \mathcal{R}(\mathcal{N}_2)$$

  Note: If $\mathcal{R}$ is the average instead of the sum of loss functions, we need to reweight: $\frac{|\mathcal{N}_t|}{|\mathcal{N}|}\mathcal{R}(\mathcal{N}_t)$

- In general, compatible with arbitrary losses – typical choices:
  - g-way classification:

Brier score → **Gini** impurity	**Bernoulli** loss → **entropy** impurity
$\mathcal{R}(\mathcal{N}) = \sum\limits_{(\mathbf{x},y)\in\mathcal{N}} \sum\limits_{k=1}^{g} \hat{\pi}_k^{(\mathcal{N})}(1 - \hat{\pi}_k^{(\mathcal{N})})$	$\mathcal{R}(\mathcal{N}) = - \sum\limits_{(\mathbf{x},y)\in\mathcal{N}} \sum\limits_{k=1}^{g} \hat{\pi}_k^{(\mathcal{N})} \log \hat{\pi}_k^{(\mathcal{N})}$

  - Regression (**quadratic** loss): $\mathcal{R}(\mathcal{N}) = \sum\limits_{(\mathbf{x},y)\in\mathcal{N}} (y - c)^2$ with $c = \frac{1}{|\mathcal{N}|} \sum\limits_{(\mathbf{x},y)\in\mathcal{N}} y$

Optimization

- **Exhaustive** search over all split candidates, choice of risk-minimal split
- In practice: reduce number of split candidates (e.g., using quantiles instead of all observed values)

CART – IMPLEMENTATION & PRACTICAL HINTS

Hyperparameters and complexity control

- Unless interrupted, splitting continues until we have pure leaf nodes (costly + overfitting)
- Hyperparameters: Complexity (i.e., number of terminal nodes) controlled via tree depth, minimum number of observations per node, maximum number of leaves, minimum risk reduction per split, ...
- Limit tree growth / complexity via
  - **Early stopping:** stop growth prematurely
    → hard to determine good stopping point before actually trying all combinations
  - **Pruning:** grow deep trees and cut back in risk-optimal manner afterwards

Implementations

- **R:**
  - **CART**: `mlr3` learners `LearnerClassifRpart` / `LearnerRegrRpart`, calling `rpart::rpart()`
  - **Conditional inference trees**: `partykit::ctree()`
    mitigates overfitting by controlling tree size via p-value-based splitting
  - **Model-based recursive partitioning**: `partykit::mob()`
    fits a linear model within each terminal node of the decision tree
  - **Rule-based models**: `Cubist::cubist()` for regression and `C50::C5.0()` for classification; more flexible frameworks for fitting various types of models (e.g., GLMs) within a tree's terminal nodes
- **Python:** `DecisionTreeClassifier` / `DecisionTreeRegressor` from package `scikit-learn`

CART – PROS & CONS

Dual purpose of CART

- **Exploration purpose** to obtain interpretable decision rules (here: performance/tuning is secondary)
- **Prediction model**: CART as base learner in **ensembles** (bagging, random forest, boosting) can improve stability and performance (if tuned properly), but becomes less interpretable

Advantages

- + **Easy** to understand & visualize (**interpretable**)
- + Built-in **feature selection**
  → e.g., when features are not used for splitting
- + Applicable to **categorical** features
  → e.g., 2^m possible binary splits for m categories
  → trick for regr. with L2-loss and binary classif.: categories can be sorted ⇒ $m - 1$ binary splits
- + Handling of **missings** possible via surrogate splits
- + Models **interactions**, even of higher order
- + **Fast** computation and good scalability
- + High **flexibility** with custom split criteria or leaf-node prediction rules

Disadvantages

- — Rather **poor generalization**
- — High **variance/instability**: model can change a lot when training data is minimally changed
- — Can **overfit** if tree is grown too deep
- — Not well-suited to model **linear** relationships
- — **Bias** toward features with many unique values or categories

RANDOM FORESTS – METHOD SUMMARY

REGRESSION CLASSIFICATION NONPARAMETRIC BLACK-BOX FEATURE SELECTION

General idea

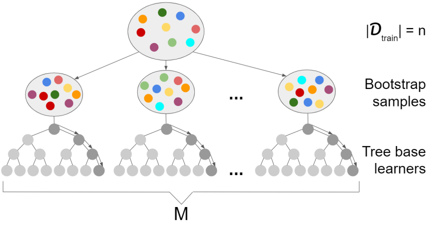
- **Bagging ensemble** of M tree **base learners** fitted on **bootstrap** data samples

  $\Rightarrow$ Reduce **variance** by ensembling while slightly increasing **bias** by bootstrapping

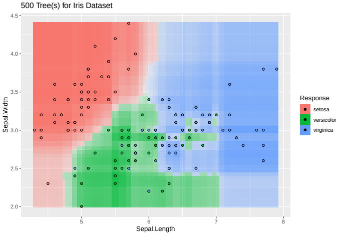
  - Use unstable, **high-variance** base learners by letting trees grow to full size
  - Promoting **decorrelation** by random subset of candidate features for each split
- **Predict** via averaging (regression) or majority vote (classification) of base learners

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 & Optimization Just like tree base learners

Out-of-bag (OOB) error

- Ensemble prediction for obs. outside individual trees' bootstrap training sample ⇒ unseen test sample
- Use resulting loss as unbiased estimate of **generalization error**
- Mainly useful for tuning and less for model comparison as we usually compare all models uniformly by CV

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

Hyperparameters

- **Ensemble size**, i.e., number of trees
- **Complexity** of base learners, e.g., tree depth, min-split, min-leaf-size
- **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

RANDOM FORESTS – IMPLEMENTATION & PRACTICAL HINTS

Extremely Randomized Trees

- Variance of trees can be further increased by **randomizing split points** instead of using the optimal one
- Alternatively consider *k* random splits and pick the best one according to impurity

Tuning

- **Ensemble size** should not be tuned as it only decreases variance $\longrightarrow$ choose sufficiently large ensemble
- While default values for **number of split points** is often good, tuning it can still improve performance
- Tuning the **minimum samples in leafs** and **minimum samples for splitting** can be benificial but no huge performance increases are to be expected

Implementation

- **R:** `mlr3` learners `LearnerClassifRanger` / `LearnerRegrRanger`, calling `ranger::ranger()` as a highly efficient and flexible implementation
- **Python:** `RandomForestClassifier` / `RandomForestRegressor` from package `scikit-learn`

RANDOM FORESTS – PROS & CONS

Advantages

- + Retains most of **trees'** advantages (e.g., feature selection, feature interactions)
- + Fairly **good predictor**: mitigating base learners' variance through bagging
- + Quite **robust** w.r.t. small changes in data
- + Good with **high-dimensional** data, even in presence of noisy features
- + Easy to **parallelize**
- + Robust to its hyperparameter configuration
- + Intuitive measures of **feature importance**

Disadvantages

- — Loss of individual trees' **interpretability**
- — Can be suboptimal for **regression** when extrapolation is needed
- — **Bias** toward selecting features with many categories (same as CART)
- — Rather large model size and slow inference time for large ensembles
- — Typically inferior in **performance** to tuned gradient tree boosting.

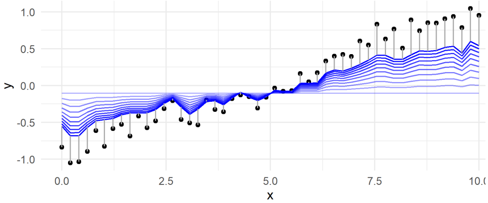
GRADIENT BOOSTING – METHOD SUMMARY

REGRESSION CLASSIFICATION (NON)PARAMETRIC BLACK-BOX FEATURE SELECTION

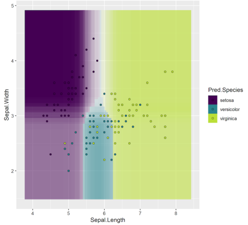
General idea

- **Sequential ensemble** of M **base learners** by greedy forward stagewise additive modeling
  - In each iteration a base learner is fitted to current **pseudo residuals** $\Rightarrow$ one boosting iteration is one approximate **gradient step in function space**
  - Base learners are typically **trees**, **linear regressions** or **splines**
- **Predict** via (weighted) sum of base learners

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



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

- In general, compatible with any **differentiable** loss
- Base learner in iteration m is fitted on **Pseudo residuals**: $\tilde{r}^{(i)} = -\frac{\partial L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right)}{\partial f\left(\mathbf{x}^{(i)}\right)}$ by minimizing the **L2-loss**: $\sum_{i=1}^{n}(\tilde{r}^{(i)} - b(\mathbf{x}^{(i)}, \boldsymbol{\theta}))^2$

Optimization

- Same optimization procedure as base learner, while keeping the current ensemble $\hat{f}^{[m-1]}$ fixed
  $\Rightarrow$ Efficient and generally applicable since *inner* loss is always L2
- $\beta^{[m]}$ is found via **line search** or fixed to a **small constant value** and combined with the leaf values $c_t^{[m]}$ for tree base learners: $\tilde{c}_t^{[m]} = \beta^{[m]} \cdot c_t^{[m]}$

Hyperparameters

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

GRADIENT BOOSTING – PRACTICAL HINTS

Scalable Gradient Boosting

- **Feature and data subsampling** for each base learner fit
- **Parallelization** and **approximate split finding** for tree base learners
- GPU accelaration

Explainable / Componentwise Gradient Boosting

- Base learners of **simple linear regression** models or **splines**, selecting a single feature in each iteration
- Allows **feature selection** and creates an **interpretable** model since uni- and bivariate effects can be visualized directly.
- Feature interactions can be learned via ranking techniques (e.g., GA^{2}M FAST)

Tuning

- Use **early-stopping** to determine ensemble size
- Various **regularization parameters**, e.g., L1/L2, number of leaves, ... that need to be carefully tuned
- Tune learning rate and base learner complexity hyperparameters on **log-scale**

GRADIENT BOOSTING – IMPLEMENTATION

Gradient Tree Boosting

- **R:** `mlr3` learners `LearnerClassifXgboost` / `LearnerRegrXgboost`, `LearnerClassifLightGBM` / `LearnerRegrLightGBM`
- **Python:** `GradientBoostingClassifier` / `GradientBoostingRegressor` from package `scikit-learn`, `XGBClassifier` / `XGBRegressor` from package `xgboost`, `lgb.train` from package `lightgbm`

⇒ `LightGBM` current state-of-the-art but slightly more complicated to use than `xgboost`

Componentwise Gradient Boosting

- **R:** `mboost` from package `mboost`, `boostLinear` / `boostSplines` from package `compboost`
- **Python:** /

⇒ `mboost` very flexible but slow while `compboost` is much faster with limited features

GRADIENT BOOSTING – PROS & CONS

Advantages

+ Retains of most of **base learners'** advantages
+ Very **good predictor** due to aggressive loss minimization, typically only outperformed by heterogenous **stacking ensembles**
+ High **flexibility** via custom loss functions and choice of base learner
+ Highly efficient implementations exist (`lightgbm` / `xgboost`) that work well on large (distributed) data sets
+ Componentwise boosting: Good combination of (a) high performance (b) interpretable model and (c) feature selection

Disadvantages

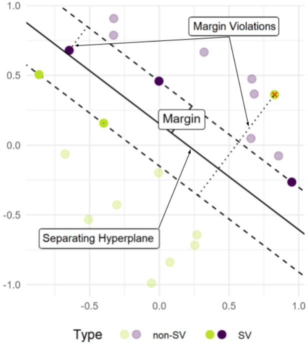
– Loss of base learners' potential **interpretability**
– **Many hyperparameters** to be carefully tuned
– Hard to **parallelize** ($\leadsto$ solved by efficient implementation)

LINEAR SVM – METHOD SUMMARY

CLASSIFICATION REGRESSION PARAMETRIC WHITE-BOX

General idea (Soft-margin SVM)

- Find linear decision boundary (**separating hyperplane**) that
  - maximizes distance (**margin** γ) to closest points (**support vectors, SVs**) on each side of decision boundary
  - while minimizing margin violations (points either on **wrong side of hyperplane** or **between dashed margin line and hyperplane**)
- 3 types of training points
  - **non-SVs** with no impact on decision boundary
  - **SVs that are margin violators** and affect decision boundary
  - **SVs located exactly on dashed margin lines** and affect decision boundary



Soft-margin SVM with margin violations

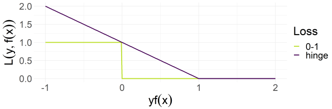
Hypothesis space (primal) $\mathcal{H} = \left\{ f(\mathbf{x}) \ : \ f(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^\top \mathbf{x} + \theta_0) \right\}$

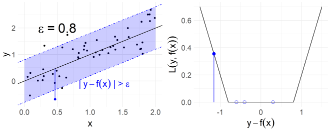
LINEAR SVM – METHOD SUMMARY

Empirical risk Soft-margin SVM as **L2-regularized ERM**:

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

- $\|\boldsymbol{\theta}\| = 1/\gamma$ ($\hat{=}$ maximizing margin)
- $C > 0$: penalization for margin violations
- Loss aims at minimizing margin violations
  $\rightarrow$ Classif. (**hinge** loss): $L(y, f) = \max(1 - yf, 0)$
  $\rightarrow$ Regr. (ϵ**-insensitive** loss): $L(y, f) = \max(|y - f| - \epsilon, 0)$





Dual problem SVMs as a constraint optimization (primal) problem (maximize margin s.t. constraints on obs. to limit margin violations) can be formulated as a Lagrangian dual problem with Lagrange multipliers $\alpha_i \geq 0$:

$$\max_{\boldsymbol{\alpha} \in \mathbb{R}^n} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y^{(i)} y^{(j)} \left\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \right\rangle \quad \text{s.t.} \quad 0 \leq \alpha_i \leq C \;\; \forall i \in \{1, \dots, n\} \text{ and } \sum_{i=1}^{n} \alpha_i y^{(i)} = 0$$

Hypothesis space (dual) Non-SVs have $\alpha_i = 0$ as they do not affect the hyperplane

$$\mathcal{H} = \left\{ f(\mathbf{x}) \;:\; f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y^{(i)} \langle \mathbf{x}^{(i)}, \mathbf{x} \rangle + \theta_0 \right) \;|\; \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \right\}$$

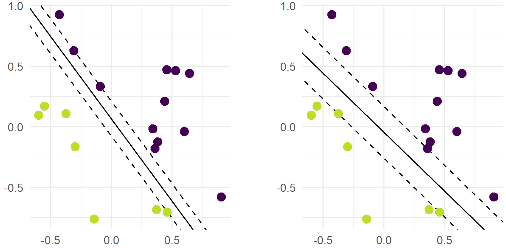
LINEAR SVM – METHOD SUMMARY

Optimization

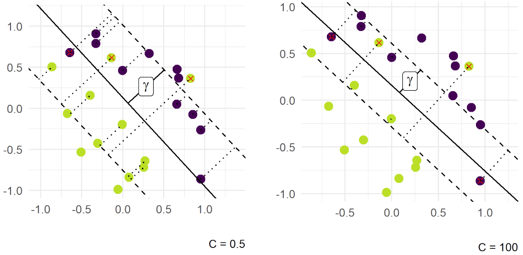
- Typically, tackling **dual** problem (though feasible in corresponding primal) via **quadratic programming**
- Popular: **sequential minimal optimization** ⇒ iterative algorithm based on breaking down objective into bivariate quadratic problems with analytical solutions

Hyperparameters

Cost parameter C to control maximization of the margin vs. minimizing margin violations



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



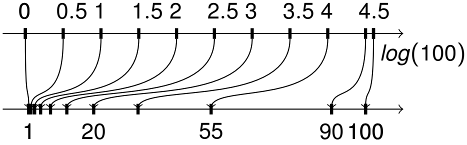
Soft-margin SVM: large margin and few margin violations on the right (best trade-off)

LINEAR SVM – IMPLEMENTATION & PRACTICAL HINTS

Preprocessing Features should be scaled before applying SVMs (applies generally to regularized models)

Tuning

- Tuning of cost parameter C advisable
  $\Rightarrow$ strong influence on resulting hyperplane
- C it is often tuned on a log-scale grid for optimal and space-filling search space



Implementation

- **R:** `mlr3` learners `LearnerClassifSVM` / `LearnerRegrSVM`, calling `e1071::svm()` with linear kernel (`libSVM` interface). Further implementations in `mlr3extralearners` based on
  - `kernlab::ksvm()` allowing custom kernels
  - `LiblineaR::LiblineaR()` for a fast implementation with linear kernel
- **Python:** `sklearn.svm.SVC` from package `scikit-learn` / package `libSVM`

NONLINEAR SVM – METHOD SUMMARY

CLASSIFICATION REGRESSION NONPARAMETRIC BLACK-BOX

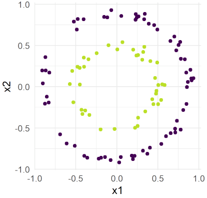
General idea

- Move **beyond linearity** by mapping data to transformed space where they are linearly separable
- **Kernel trick**
  - No need for explicit construction of feature maps
  - Replace inner product of feature map $\phi : \mathcal{X} \to \Phi$ by **kernel**: $\langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle = k(\mathbf{x}, \tilde{\mathbf{x}})$

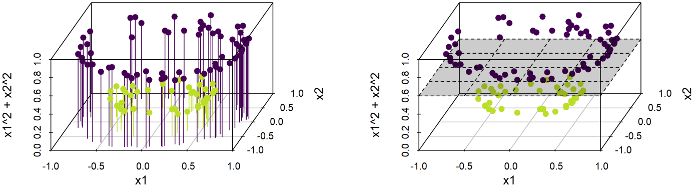
Hypothesis space

$$\mathcal{H} = \left\{ f(\mathbf{x}) \ : \ f(\mathbf{x}) = \text{sign}\left(\boldsymbol{\theta}^\top \phi(\mathbf{x}) + \theta_0\right) \right\} \text{ (primal)}$$

$$\mathcal{H} = \left\{ f(\mathbf{x}) \ : \ f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y^{(i)} k(\mathbf{x}^{(i)}, \mathbf{x}) + \theta_0 \right) \ | \ \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y^{(i)} = 0 \right\} \text{ (dual)}$$



Nonlinear problem in original space



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

NONLINEAR SVM – METHOD SUMMARY

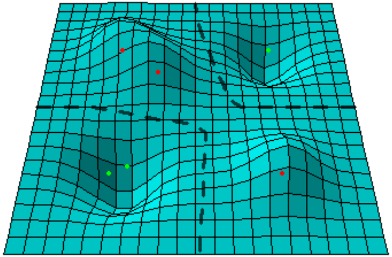
Dual problem **Kernelize** dual (soft-margin) SVM problem, replacing all inner products by kernels:

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

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

Interpretation as basis function approach

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



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

NONLINEAR SVM – IMPLEMENTATION & 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 $\Rightarrow k(\mathbf{x}, \tilde{\mathbf{x}}) = \left(\mathbf{x}^\top \tilde{\mathbf{x}} + b\right)^d,\ b \geq 0$
- **Radial basis function (RBF)** kernel: infinite-dimensional feature space, allowing for perfect separation of all finite datasets $\Rightarrow k(\mathbf{x}, \tilde{\mathbf{x}}) = \exp\left(-\gamma \|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2\right)$ 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()` with nonlinear kernel (`libSVM` interface), `kernlab::ksvm()` allowing custom kernels
- **Python:** `sklearn.svm.SVC` from package `scikit-learn` / package `libSVM`

SVM – PRO'S & CON'S

Advantages

- + Often **sparse** solution (w.r.t. observations)
- + Robust against overfitting (**regularized**); especially in high-dimensional space
- + **Stable** solutions (w.r.t. changes in train data)
  → Non-SV do not affect decision boundary
- + Convex optimization problem
  → local minimum $\hat{=}$ global minimum

Advantages (nonlinear SVM)

- + Can learn **nonlinear decision boundaries**
- + **Very flexible** due to custom kernels
  → RBF kernel yields local model
  → kernel for time series, strings etc.

Disadvantages

- – **Long** training times $\rightarrow O(n^2 p + n^3)$
- – Confined to **linear model**
- – Restricted to **continuous features**
- – Optimization can also fail or get stuck

Disadvantages (nonlinear SVM)

- – Poor **interpretability** due to complex kernel
- – **Not easy tunable** as it is highly important to choose the right kernel (which also introduces further hyperparameters)

GAUSSIAN PROCESSES (GP) – METHOD SUMMARY

REGRESSION CLASSIFICATION NONPARAMETRIC PROBABILISTIC

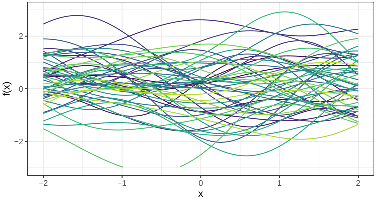
General idea

- GPs model a distribution over potential functions $\boldsymbol{f}$ that fit the observed data
- **Assumptions**:
  - n-observations follow a n-dimensional Normal distribution
  - The closer observations are, the higher they are correlated
- A **kernel** function $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ quantifies the similarity between two observations and induced the coviariance matrix of the distribution.
- **Predict** via the maximum a-posteriori (MAP) estimate.

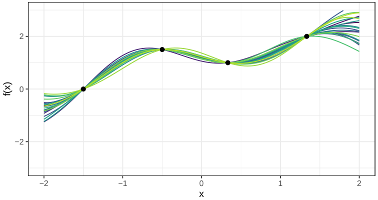
Hypothesis space

$$\mathcal{H} = \left\{ \boldsymbol{f} = \left[f\left(\mathbf{x}^{(1)}\right), \ldots, f\left(\mathbf{x}^{(n)}\right) \right] \sim \mathcal{N}\left(\boldsymbol{m}, \boldsymbol{K}\right) \mid \boldsymbol{m} \in \mathbb{R}^n, \boldsymbol{K} \in \mathbb{R}^{n \times n} \right\}$$

Functions drawn from a Gaussian process prior



Posterior process after 4 observations



GAUSSIAN PROCESSES (GP) – METHOD SUMMARY

Empirical risk

- The risk is estimated by using the posterior of a conditional Normal distribution
- Most kernels have **length scale parameters** that need to be estimated

Optimization

- The kernel parameters can be learned using **maximum likelihood** estimation
- This requires inverting the $n \times n$ -covariance matrix

Hyperparameters

- The most important hyperparameter is the choice of the kernel function $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$
- Common kernel choices for *"standard"* data are:
  - Linear or polyomial
  - Squared-exponential (infinitely differentiable)
  - Matérn (further generalization of the Squared-exponential kernel)
- Special kernels for all kind of data situation exist, e.g., a Exp-Sine-Squared kernel for periodic data
- Kernels can be composed by multiplying or addition to create more expressive structures

GP – IMPLEMENTATION & PRACTICAL HINTS

Scalable GPs for larger data

- Low-rank approximations of the covariance by using only a representative subset of **inducing points**
- Using a kernel that creates a sparse coviariance matrix

Noisy GPs

- Having an interpolator might not be suitable if the data is noisy
- A noisy GP adds a **nugget** effect to the kernel $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) + \sigma\delta_{ij}$, creating a Gaussian process regression model

Implementation

- **R:** `mlr3` learners `LearnerClassifGausspr` / `LearnerRegrGausspr`, calling `kernlab::gausspr()`
- **Python:** `GaussianProcessClassifier` / `GaussianProcessRegressor` from package `scikit-learn`, `gpytorch` for a modular, scalable, efficient and GPU accelerated implementation built on `torch`

GAUSSIAN PROCESSES (GP) – PROS & CONS

Advantages

- + GPs allow to **quantify prediction uncertainty** induced by both intrinsic noise in the problem and errors in the parameter estimation process
- + A GP is a function **interpolator** and will predict the exact value of a training point
- + The choice of kernel function allows considerable flexibility for problem specific characteristics
- + Automatic relevance determination (ARD) determines the importance of features

Disadvantages

- — GPs are **not sparse**, i.e., they require the full training data for prediction
- — GP training requires $\mathcal{O}(n^3)$, i.e., it scales cubically in the number of observations
- — GPs cannot handle categorical features.
- — GPs are **not particularly easy to understand** conceptually

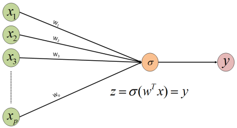
NEURAL NETWORKS – METHOD SUMMARY

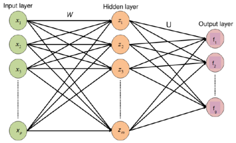
REGRESSION CLASSIFICATION (NON)PARAMETRIC BLACK-BOX

General idea

- Learn **composite function** through series of nonlinear feature transformations, represented as **neurons**, organized hierarchically in **layers**
  - Basic neuron operation: 1) affine **transformation** ϕ (weighted sum of inputs), 2) nonlinear **activation** σ
  - Combinations of simple building blocks to create a complex model
- Optimize via **mini-batch stochastic gradient descent (SGD)** variants:
  - Gradient of each weight can be infered from the **computational graph** of the network $\rightarrow$ **Automatic Differentiation** (AutoDiff)
  - Algorithm to compute weight updates based on the loss is called **Backpropagation**

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 NETWORKS – METHOD SUMMARY

Architecture

- Input layer: original features $\mathbf{x}$
- Hidden layers: nonlinear transformation of previous layer $\phi^{(h)} = \sigma^{(h-1)}(\phi^{(h-1)})$
- Output layer: number of output neurons and activation depends on problem $\tau(\phi)$
  - Regression: one output neuron, $\tau =$ identity
  - Binary classification: one output neuron, $\tau = \frac{1}{1+\exp(-\boldsymbol{\theta}^\top \mathbf{x})}$ (logistic sigmoid)
  - Multiclass Classification: g output neurons, $\tau_j = \frac{\exp(f_j)}{\sum_{j=1}^{g} \exp(f_j)}$ (softmax)

Empirical risk In general, compatible with any differentiable loss

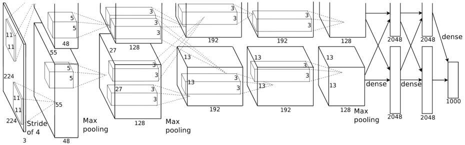
Optimization

- Variety of different optimizers, mostly based on some form of **stochastic gradient descent (SGD)**
- Improvements:
  - **(1)** Accumulation of previous gradients → **Momentum**
  - **(2)** Weight specific scaling based on previous squared gradients → **RMSProb**
    $\Rightarrow$ **ADAM** combines (1) and (2)
  - **(3)** Learning rate schedules, e.g., decaying or cyclical learning rates
- Training progress is measured in full passes over the full training data, called **epochs**
- **Batch size** is a hyperparameter and limited by input data dimension

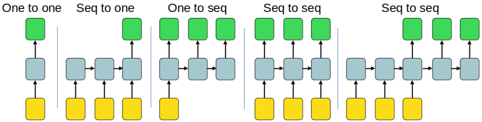
NEURAL NETWORKS – METHOD SUMMARY

Network types Large variety of architectures for different data modelities

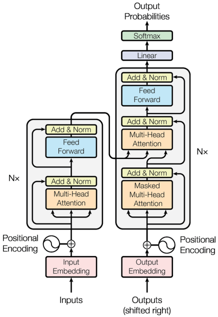
- **Feedforward NNs / multi-layer perceptrons (MLPs):** sequence of **fully-connected** layers ⇒ tabular data
- **Convolutional NNs (CNNs):** sequence of feature map extractors with spatial awareness ⇒ images, time series
- **Recurrent NNs (RNNs):** handling of sequential, variable-length information ⇒ times series, text, audio
- **Transformers:** Learning invariances from data, handling multiple/any data modalities



Convolutional network architecture



Recurrent network architecture



Transformer network architecture

NEURAL NETWORKS – METHOD SUMMARY

Hyperparameters

- **Architecture**:
  - Lots of design choices ⇒ tuning problem of its own.
  - Typically: hierachical optimization of components (cells) and macro structure of network → **Neural Architecture Search (NAS)**
  - Many predifined (well working) architectures exist for standard tasks
- **Training**:
  - Initial learning rate and various regularization parameters
  - Number of epochs is determined by **early-stopping**
  - **Data-augmentation**, e.g., applying random rotations to input images

Foundation models

- **Enormous** models trained on vast amounts of (general) data, e.g., all of wikipedia, in **self-supervised** fashion
- Used as starting point (**pre-trained**) and fine-tuned via **transfer** or **few-shot** learning for other tasks requiring little data
- Examples: GPT-3 for language, CLIP for vision-language, . . .

NEURAL NETWORKS – IMPLEMENTATION & PRACTICAL HINTS

General hints

- Instead of NAS, use a standard architecture and tune training hyperparameters
- Training pipeline (data-augmentation, training schedules, ...) is more crucial than the specific architecture
- While NNets are state-of-the-art for **computer vision (CV)** and **natural language processing (NLP)**, we recommend not to use them for tabular data because alternatives perform better
- Computational efforts for training (and inference) can be very high, requiring specific hardware.
  → Using a service (esp. for foundation models) can be more cost efficient

Implementation

- **R:** Use python libraries (below) via `reticulate`, but not really recommended except for toy applications.
- **Python libraries:**
  - `keras` for simple high level API
  - `PyTorch` for flexible design with a focus on research
  - `TensorFlow` for flexible design with a focus on deployment / industry
  - `huggingface` for pre-trained / foundation models

NEURAL NETWORKS – PROS & CONS

Advantages

- + Applicable to **complex, nonlinear** problems
- + Very **versatile** w.r.t. architectures
- + State-of-the-art for CV and NLP
- + 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