Introduction to Machine Learning

ML-Basics: Losses & Risk Minimization

HOW TO EVALUATE MODELS

OVERVIEW

No Free Lunch 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. Hypothesis space + Risk + Optimization

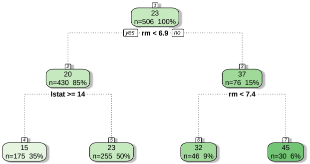
CART FUNCTIONALITY

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.

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

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



Full tree

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*** $\mathcal{R}(\mathcal{N}_t) = \sum_{(\mathbf{x},y)\in\mathcal{N}_t} \sum_{k=1}^{g} (\mathbb{I}(y=k) - \pi_k(\mathbf{x}))^2$
    - 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

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 APPLICATION

For applications of CART, note the following:

Pruning / early stopping

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

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

Implementation

R: package $\texttt{rpart}$
Python: $\texttt{DecisionTreeClassifier}$ / $\texttt{DecisionTreeRegressor}$ from package $\texttt{scikit-learn}$
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***.

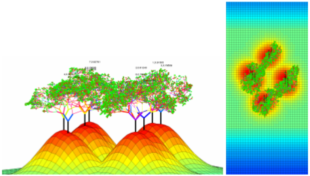
RANDOM FORESTS FUNCTIONALITY

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

General idea Random forests are ***bagging ensembles***: they combine multiple CART (base learners) to form a strong learner. They use **complex** trees with low bias and compensate for single trees' high variance by aggregating M of them in a **decorrelated** manner.

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

- Training on bootstrap samples of the data and only on a random subset of features to incur variability
- Aggregation via averaging (regression) or majority voting (classification)



RANDOM FORESTS **FUNCTIONALITY**

Empirical risk Applicable with **any** kind of loss function (just like tree base learners)
$\rightarrow$ 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

RANDOM FORESTS **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 FORESTS APPLICATION

For applications of random forests, note the following:

Pre-processing

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

Implementation

R: package $\texttt{ranger}$
Python: $\texttt{RandomForestClassifier}$ / $\texttt{RandomForestRegressor}$ from package $\texttt{scikit-learn}$

Tuning

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

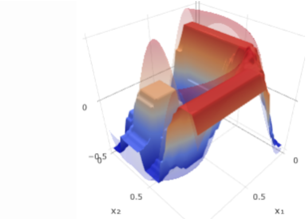
GRADIENT BOOSTING **FUNCTIONALITY**

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. The final model is a weighted sum of base learners $b(\mathbf{x}, \theta^{[m]})$ with weights $\beta^{[m]}$.

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

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



GRADIENT BOOSTING **FUNCTIONALITY**

Empirical risk

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

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

Hyperparameters

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

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 **APPLICATION**

For applications of random forests, note the following:

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
$\rightarrow$ Available in all major programming languages

Stochastic gradient boosting (SGB)

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

Implementation

R: packages gbm, $\mathrm{xgboost}$
Python: $\mathrm{GradientBoostingClassifier}$ / $\mathrm{GradientBoostingRegressor}$ from package $\mathrm{scikit\text{-}learn}$, $\mathrm{XGBClassifier}$ / $\mathrm{XGBRegressor}$ from package $\mathrm{xgboost}$

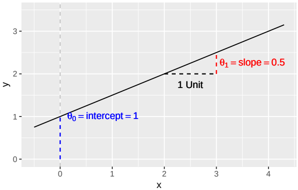
Tuning

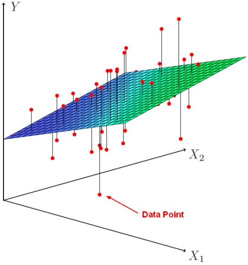
FUNCTIONALITY

SUPERVISED PARAMETRIC WHITE-BOX

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

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





LINEAR MODEL **FUNCTIONALITY**

Empirical risk

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

Optimization

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

Hyperparameters none

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 and normal distributed residuals ??
- - tend to **overfit** (can be reduced by regularization)
- - **sensitive to outliers and noisy data**

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

For applications of linear models, note the following:

Check assumptions???????

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

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

Implementation

R: function lm
Python: $\mathrm{LinearRegression}$ from package $\mathrm{sklearn.linear\_model}$, package for advanced statistical parameters $\mathrm{statsmodels.api}$

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.

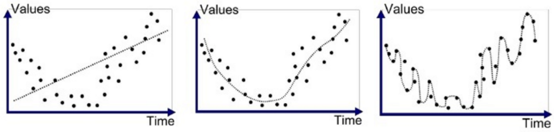
 FUNCTIONALITY

SUPERVISED PARAMETRIC WHITE-BOX

General idea

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

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



Underfitted Good Fit/Robust Overfitted

REGULARIZED LM **FUNCTIONALITY**

Empirical risk

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

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

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

Optimization

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

Hyperparameter **shrinkage parameter** λ [and α for Elastic net??]

REGULARIZED LM **PRO'S & CON'S**

SAME LIKE LINEAR MODEL?

Advantages

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

Disadvantages

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

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

APPLICATION

For applications of linear models, note the following:

Choice of regularization parameter λ

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

Implementation

R: package for regularized linear model glmnet
Python: $\mathrm{LinearRegression}$ from package $\mathrm{sklearn.linear\_model}$, package for advanced statistical parameters $\mathrm{statsmodels.api}$