# Machine Learning with R at LRZ

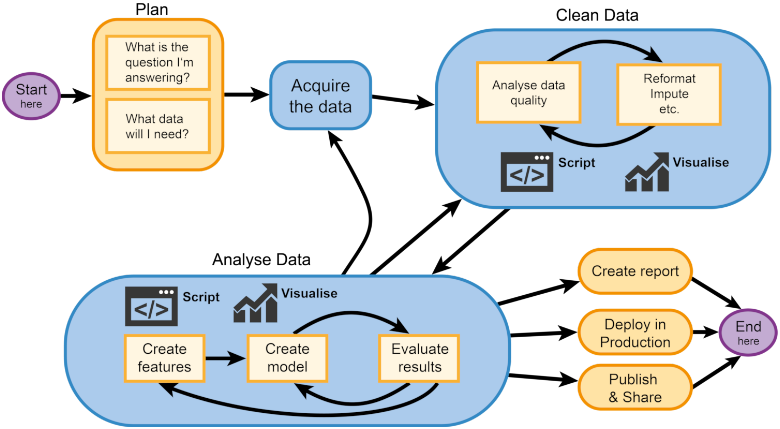
Introduction to mlr

www.essentialds.de

2019-10-11

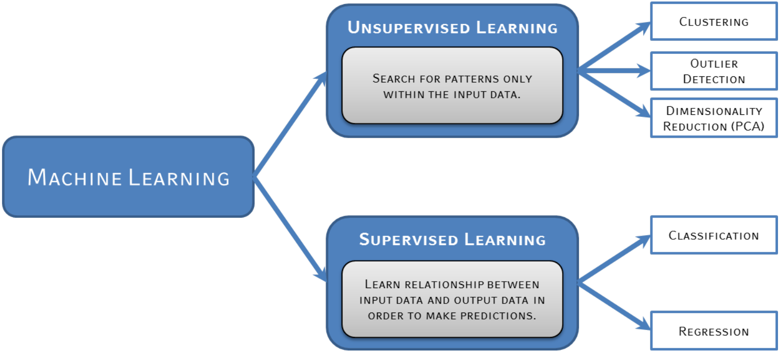
# WHAT IS MACHINE LEARNING

# Typical Workflow



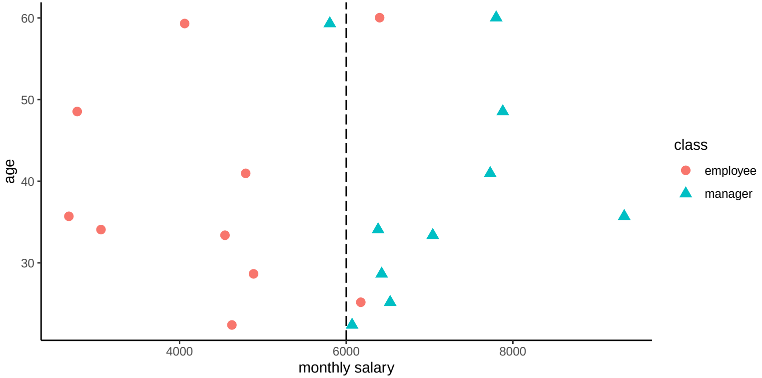
Essential Data Science Training

Machine learning (ML) can be seen as the intersection between computer science and computational statistics in which computer algorithms learn to solve different tasks based on data (e.g. making predictions, finding groups, . . . ).



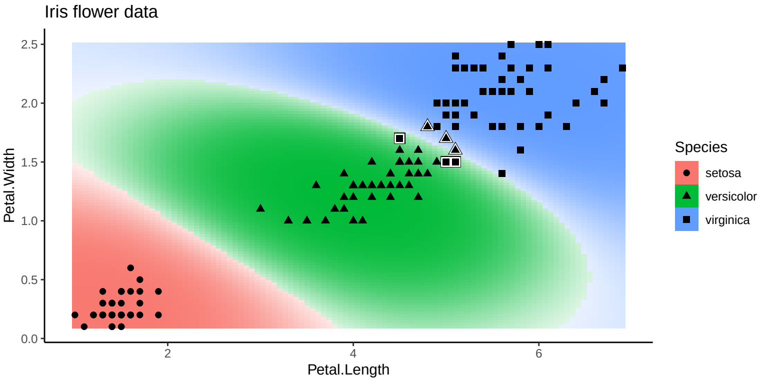
# Binary Classification Task

- $y$ is a categorical variable (with two values)
- E.g., sick/healthy, or credit/no credit
- **Goal**: Predict a class (or membership probabilities)



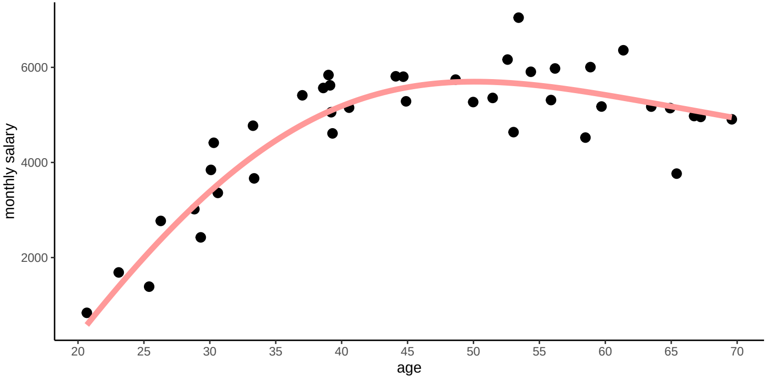
# Multiclass Classification Task

- $y$ is a categorical variable with $> 2$ unordered values
- Each instance belongs to only one class
- **Goal**: Predict a class (or membership probabilities)



# Regression Task

- **Goal**: Predict a continuous output
- $y$ is a metric variable (with values in $\mathbb{R}$)
- Regression model can be constructed by different methods, e.g., linear regression, trees or splines



# INTRODUCTION

# Motivation: Machine Learning in R

The **good** news:

- CRAN serves hundreds of packages for machine learning
- Often compliant to the unwritten interface definition:

```
model = fit(target ~ ., data = train.data, ...)
predictions = predict(model, newdata = test.data, ...)
```

The **bad** news:

- Some packages' API is "just different"
- Functionality is always package or model-dependent, even though the procedure might be general
- No meta-information available or buried in docs

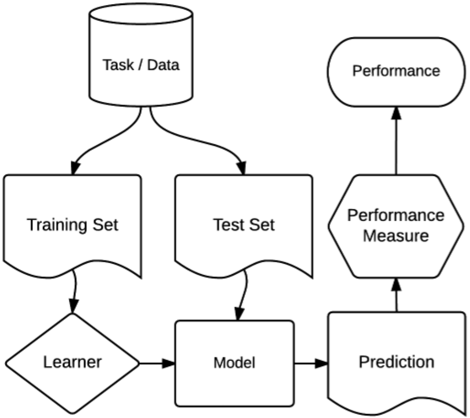
**Our goal: A domain-specific language for ML concepts!**

# Motivation: MLR

- Project home page: https://github.com/mlr-org/mlr
  - Cheatsheet for an quick overview
  - Tutorial for mlr documentation with many code examples
  - Ask questions in the GitHub issue tracker
- 8-10 main developers, quite a few contributors, 4 GSOC projects in 2015/16 and one in 2017
- About 30K lines of code, 8K lines of unit tests

# Motivation: MLR

- Unified interface for the basic building blocks: tasks, learners, hyperparameters, . . .



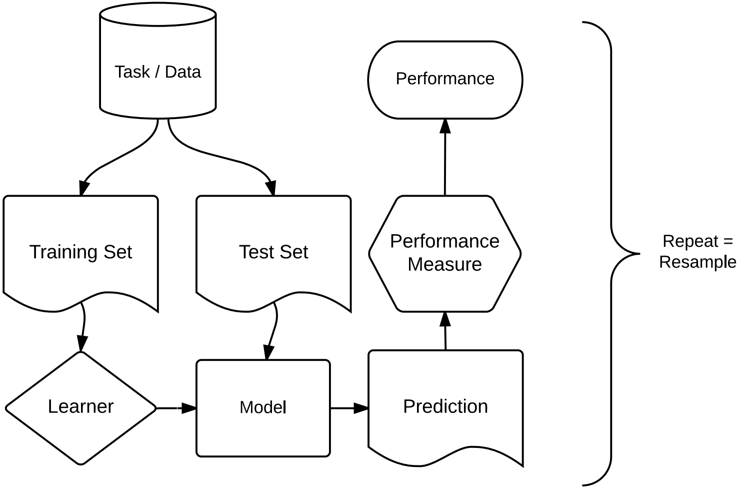
# Basic Features of MLR



- Tasks and Learners
- Train, Test, Resample
- Performance
- Benchmarking
- Hyperparameter Tuning
- Nested Resampling
- Parallelization

# Building blocks






- Core objects: tasks, learners, measures, resampling instances.

# Task Abstraction

- Tasks encapsulate data and meta-information about it.
- Regression, classification, clustering, survival tasks.

```
data(BostonHousing, package = "mlbench")
task = makeRegrTask(data = BostonHousing, target = "medv")
print(task)

## Supervised task: BostonHousing
## Type: regr
## Target: medv
## Observations: 506
## Features:
##    numerics     factors     ordered functionals
##          12           1           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
```

# Learner Abstraction

- Internal structure of learners:
  - wrappers around `fit()` and `predict()` of the package
  - description of the parameter set
  - annotations
- Naming convention: `<tasktype>.<functionname>`

```
makeLearner("regr.lm")
makeLearner("classif.randomForest")
makeLearner("classif.knn", k = 2)
```

- Adding custom learners is covered in the tutorial

# Learner Abstraction

```r
lrn = makeLearner("classif.knn", k = 2)
print(lrn)

## Learner classif.knn from package class
## Type: classif
## Name: k-Nearest Neighbor; Short name: knn
## Class: classif.knn
## Properties: twoclass,multiclass,numerics
## Predict-Type: response
## Hyperparameters: k=2

getParamSet(lrn)

##              Type len   Def    Constr Req Tunable Trafo
## k         integer   -     1  1 to Inf   -    TRUE     -
## l         numeric   -     0  0 to Inf   -    TRUE     -
## prob      logical   - FALSE         -   -   FALSE     -
## use.all   logical   -  TRUE         -   -    TRUE     -
```

# Learn more



- Extensive Tutorial covers *all* features in mlr:
  https://mlr-org.github.io/mlr/
  - Tuning
  - Resampling (with blocking)
  - Visualization Topics
  - Multilabel Classification, Survival Analysis, Clustering
  - Handling Spatial Data
  - Functional Data
  - Create Custom Learners and Measures
  - ...

# Getting Help

- Ask questions on Stackoverflow: https://stackoverflow.com/questions/tagged/mlr
- Found bugs? Report them: https://github.com/mlr-org/mlr/issues

You want to contribute? - Open a PR on github and join our slack: https://mlr-org.slack.com/

# FIRST DATA ANALYSIS

# First Classification Analysis

1. Peek into the iris data set
2. Define a classification task
3. Fit a $k$-NN classification model
4. Predict labels

# Iris Data Set

The iris dataset was introduced by the statistician Ronald Fisher and is one of the most frequent used datasets. Originally it was designed for linear discriminant analysis.

The set is a typical test case for many statistical classification techniques and has its own **wikipedia page.**

Setosa

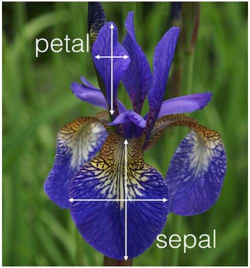
Versicolor

Virginica

Source: https://en.wikipedia.org/wiki/Iris_flower_data_set

# Iris Data Set

- 150 iris flowers
- 3 different species (50 setosa, 50 versicolor, 50 virginica) to be predicted.
- Sepal length / width and petal length / width in [cm].

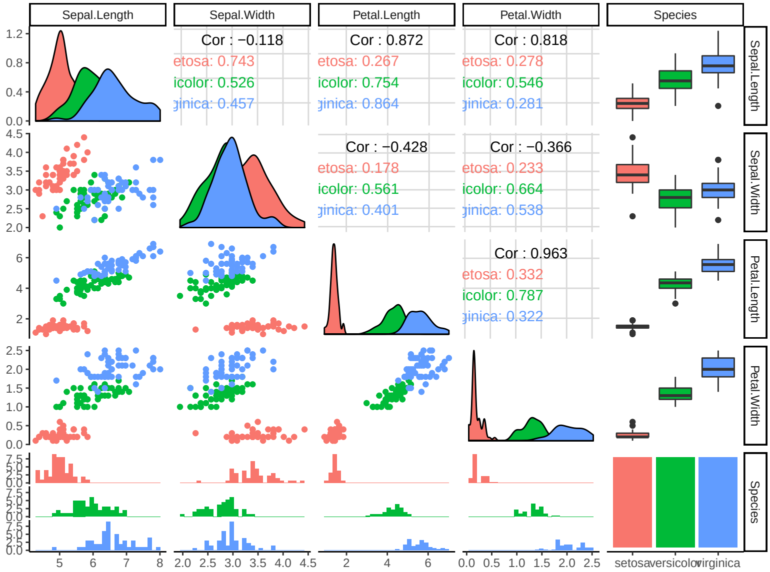



Source: https://holgerbrandl.github.io/kotlin4ds_kotlin_night_frankfurt//krangl_example_report.html

A peek into the data:

```
data("iris", package = "datasets")
str(iris)

## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1..
```

# Iris Data Set



Define a `mlr` task for the iris data using the target column `Species`:

```
library(mlr)
task.iris = makeClassifTask(id = "iris", data = iris, target = "Species")
print(task.iris) # Gives you an overview of the task

## Supervised task: iris
## Type: classif
## Target: Species
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
## Classes: 3
##     setosa versicolor  virginica
##         50         50         50
## Positive class: NA
```

Functions prefixed with `getTask[...]` allow to extract information, i.e., `getTaskData` extracts the `data.frame`.

# What Learners are available?



## Classification

- LDA, QDA, RDA, MDA
- Trees and forests
- Boosting (different variants)
- SVMs (different variants)
- (Deep) Neural Networks
- ...

## Regression

- Linear, lasso and ridge
- Boosting
- Trees and forests
- Gaussian processes
- (Deep) Neural Networks
- ...

## Clustering

- K-Means
- EM
- DBscan
- X-Means
- ...

## Survival

- Cox-PH
- Cox-Boost
- Random survival forest
- Penalized regression
- ...

We can explore them on the *webpage*, e.g. Classification learners:

## Classification (82)

For classification the following additional learner properties are relevant and shown in column **Props**:

- *prob*: The method can predict probabilities,
- *oneclass*, *twoclass*, *multiclass*: One-class, two-class (binary) or multi-class classification problems be handled,
- *class.weights*: Class weights can be handled.

| Class / Short Name / Name | Packages | Num. | Fac. | Ord. | NAs | Weights | Props | Note |
|---|---|---|---|---|---|---|---|---|
| **classif.ada**<br>*ada*<br><br>ada Boosting | ada<br>rpart | X | X | | | | prob<br>twoclass | `xval` has been set to `0` by default for speed. |
| **classif.adaboostm1**<br>*adaboostm1*<br><br>ada Boosting M1 | RWeka | X | X | | | | prob<br>twoclass<br>multiclass | NAs are directly passed to WEKA with `na.action = na.pass`. |
| **classif.bartMachine**<br>*bartmachine*<br><br>Bayesian Additive Regression Trees | bartMachine | X | X | | X | | prob<br>twoclass | `use_missing_data` has been set to `TRUE` by default to allow missing data support. |
| **classif.binomial**<br>*binomial*<br><br>Binomial Regression | stats | X | X | | | X | prob<br>twoclass | Delegates to `glm` with freely choosable binomial link function via learner parameter `link`. We set 'model' to FALSE by default to save memory. |
| **classif.boosting**<br>*adabag*<br><br>Adabag Boosting | adabag<br>rpart | X | X | | X | | prob<br>twoclass<br>multiclass<br>featimp | `xval` has been set to `0` by default for speed. |

# What Learners are available?

Or use `listLearners` to find appropriate learners for the given task:

```
tab = listLearners(task.iris, warn.missing.packages = FALSE)
tab[1:5, c("class", "package")]

##                    class       package
## 1 classif.adaboostm1         RWeka
## 2   classif.boosting adabag,rpart
## 3        classif.C50           C50
## 4    classif.cforest         party
## 5      classif.ctree         party
```

This is possible because

- the task contains relevant data/task characteristics (e.g., missings) and
- the learner checks if it can handle data/tasks with these characteristics (learner properties)

Define the classification learner:

```
lrn.knn = makeLearner("classif.kknn", k = 30, predict.type = "prob")

## Loading required package: kknn

print(lrn.knn) # learner will predict classes and probabilities

## Learner classif.kknn from package kknn
## Type: classif
## Name: k-Nearest Neighbor; Short name: kknn
## Class: classif.kknn
## Properties: twoclass,multiclass,numerics,factors,prob
## Predict-Type: prob
## Hyperparameters: k=30
```

# First Classification Analysis: Learner

The learner contains information about all parameters that can be specified:

```
# list available hyperparameters + defaults, constraints, dependencies, ...
getParamSet(lrn.knn)

##               Type len     Def               Constr Req Tunable Trafo
## k          integer   -       7             1 to Inf   -    TRUE     -
## distance   numeric   -       2             0 to Inf   -    TRUE     -
## kernel    discrete   - optimal rectangular,trian...   -    TRUE     -
## scale      logical   -    TRUE                    -   -    TRUE     -
```

# First Classification Analysis: Training

A model is usually trained on a subset of the data - the remaining part is used to evaluate its performance.

```
n = getTaskSize(task.iris)
train.ind = sample(n, n/2)
test.ind = setdiff(1:n, train.ind)
str(train.ind)

##  int [1:75] 44 118 61 130 138 7 77 128 ...

str(test.ind)

##  int [1:75] 2 4 8 10 11 13 19 21 ...
```

# First Classification Analysis: Training



```r
# train model with mlr
mod = train(lrn.knn, task = task.iris, subset = train.ind)
print(mod)

## Model for learner.id=classif.kknn; learner.class=classif.kknn
## Trained on: task.id = iris; obs = 75; features = 4
## Hyperparameters: k=30

# retrieve model as returned from the third party package
knn.mod = getLearnerModel(mod)
```

The prediction is then applied on the unseen test data.

```
# predict using the task
preds = predict(mod, task = task.iris, subset = test.ind)
head(as.data.frame(preds), 3)

##   id  truth prob.setosa prob.versicolor prob.virginica response
## 2  2 setosa       0.979        0.021452              0   setosa
## 4  4 setosa       0.987        0.012807              0   setosa
## 8  8 setosa       0.999        0.000838              0   setosa

# predict using data set observations
preds = predict(mod, newdata = iris[test.ind, ])
head(as.data.frame(preds), 3)

##    truth prob.setosa prob.versicolor prob.virginica response
## 2 setosa       0.979        0.021452              0   setosa
## 4 setosa       0.987        0.012807              0   setosa
## 8 setosa       0.999        0.000838              0   setosa
```

# First Classification Analysis: Predictions

```r
pred.class = getPredictionResponse(preds) # predicted classes
pred.prob = getPredictionProbabilities(preds) # predicted probabilities
truth = getPredictionTruth(preds) # true classes
head(pred.class, 3)

## [1] setosa setosa setosa
## Levels: setosa versicolor virginica

head(pred.prob, 3)

##   setosa versicolor virginica
## 2  0.979   0.021452         0
## 4  0.987   0.012807         0
## 8  0.999   0.000838         0

head(truth, 3)

## [1] setosa setosa setosa
## Levels: setosa versicolor virginica
```

# First Classification Analysis: Evaluation

```r
# total number of errors
sum(pred.class != truth)

## [1] 8

# mean misclassification error (MMCE)
mean(pred.class != truth)

## [1] 0.107

# percentage of accurate predictions (ACC, accuracy)
mean(pred.class == truth)

## [1] 0.893
```

# First Classification Analysis: Evaluation

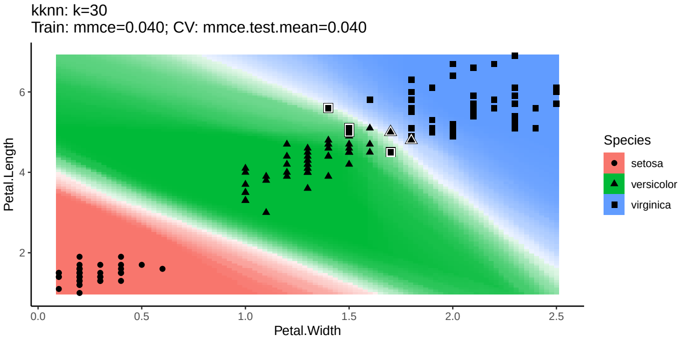
Essential Data Science Training

```
calculateConfusionMatrix(preds)

##             predicted
## true         setosa versicolor virginica -err.-
##   setosa         21          0         0      0
##   versicolor      0         22         6      6
##   virginica       0          2        24      2
##   -err.-          0          2         6      8
```

# First Classification Analysis: Evaluation

```r
performance(preds, measures = mmce) # mean misclassification error

##  mmce
## 0.107

performance(preds, measures = list(mmce, acc))

##  mmce   acc
## 0.107 0.893

listMeasures(task.iris)

##  [1] "featperc"          "mmce"              "lsr"
##  [4] "bac"               "qsr"               "timeboth"
##  [7] "multiclass.aunp"   "timetrain"         "multiclass.aunu"
## [10] "ber"               "timepredict"       "multiclass.brier"
## [13] "ssr"               "acc"               "logloss"
## [16] "wkappa"            "multiclass.au1p"   "multiclass.au1u"
## [19] "kappa"
```

```r
plotLearnerPrediction(lrn.knn, task.iris,
  features = c("Petal.Width", "Petal.Length"))
```




Predictions for learner fitted on two features.

# EXERCISE 1

# RESAMPLING

# Hold-out in mlr

Essential Data Science Training

```r
task = iris.task
n = getTaskSize(task)
ratio = 2/3

set.seed(123)
train.inds = sample(1:n, n * ratio)
test.inds = setdiff(1:n, train.inds)

lrn.knn1 = makeLearner("classif.knn", k = 1)
mod = train(lrn.knn1, task, subset = train.inds)
preds = predict(mod, task, subset = test.inds)
preds = predict(mod, newdata = iris[test.inds, ]) # alternative
mlr::performance(preds, mmce)

## mmce
## 0.08
```

# Cross-validation in mlr

```r
# Define learner:
lrn = makeLearner("classif.randomForest", predict.type = "prob")

# Define resampling strategy:
rdesc = makeResampleDesc("CV", iters = 3, stratify = TRUE)
r = resample(lrn, spam.task, rdesc,
  measure = list(mlr::acc, mlr::auc))
print(r)

## Resample Result
## Task: spam-example
## Learner: classif.randomForest
## Aggr perf: acc.test.mean=0.952,auc.test.mean=0.985
## Runtime: 15.258
```

```r
head(r$measures.test)

##   iter   acc   auc
## 1    1 0.949 0.982
## 2    2 0.956 0.988
## 3    3 0.950 0.986

head(as.data.frame(r$pred))

##     id   truth prob.nonspam prob.spam response iter  set
## 1 1814 nonspam        0.684     0.316  nonspam    1 test
## 2 1818 nonspam        0.960     0.040  nonspam    1 test
## 3 1822 nonspam        0.922     0.078  nonspam    1 test
## 4 1828 nonspam        0.982     0.018  nonspam    1 test
## 5 1829 nonspam        0.958     0.042  nonspam    1 test
## 6 1843 nonspam        0.948     0.052  nonspam    1 test
```

# Parameter of makeResamplingDesc



| Methods | Parameter | Description |
|---|---|---|
| `CV` | `iters` | Number of iterations |
| `LOO` | | |
| `RepCV` | `reps` | Repeats for repeated CV |
| | `folds` | Folds in the repeated CV |
| `Bootstrap` | `iters` | Number of iterations |
| `Subsample` | `iters` | Number of iterations |
| | `split` | Proportion of training cases |
| `Holdout` | `split` | Proportion of training cases |

# Possible Ways to use Cross Validation

1. Explicitly define resampling:

```
rdesc = makeResampleDesc("CV", iters = 10)
rdesc = cv10

res1 = resample("classif.randomForest", iris.task, resampling = rdesc,
  show.info = FALSE)
res2 = resample("classif.randomForest", iris.task, resampling = cv10,
  show.info = FALSE)

res1
```

```
## Resample Result
## Task: iris-example
## Learner: classif.randomForest
## Aggr perf: mmce.test.mean=0.047
## Runtime: 0.237422
```

Other pre defined objects are `cv2`, `cv3` and `cv5`.

# Possible Ways to use Cross Validation

2. Use `crossval`:

```
res3 = crossval("classif.randomForest", iris.task, iters = 10,
  show.info = FALSE)
res3
```

```
## Resample Result
## Task: iris-example
## Learner: classif.randomForest
## Aggr perf: mmce.test.mean=0.053
## Runtime: 0.377968
```

Similar functions are `repcv`, `holdout`, `subsample`, `bootstrapOOB`, `bootstrapB632` and `bootstrapB632plus`.

# Classification Analysis: Benchmarking

```
# quick way to compare learners with identical train/test splits
task = iris.task
learners = list(
  makeLearner("classif.knn", k = 3),
  makeLearner("classif.lda"),
  makeLearner("classif.naiveBayes")
)
benchmark(learners, task, resamplings = cv3)

##         task.id         learner.id mmce.test.mean
## 1 iris-example        classif.knn         0.0533
## 2 iris-example        classif.lda         0.0200
## 3 iris-example classif.naiveBayes         0.0400
```

# Classification Analysis: Benchmarking

```r
tasks = list(iris.task, sonar.task, pid.task)
bm = benchmark(learners, tasks, resampling = cv3)
print(bm)

##                        task.id         learner.id mmce.test.mean
## 1                 iris-example       classif.knn         0.0467
## 2                 iris-example       classif.lda         0.0200
## 3                 iris-example classif.naiveBayes        0.0400
## 4 PimaIndiansDiabetes-example       classif.knn         0.2930
## 5 PimaIndiansDiabetes-example       classif.lda         0.2357
## 6 PimaIndiansDiabetes-example classif.naiveBayes        0.2500
## 7                Sonar-example       classif.knn         0.1877
## 8                Sonar-example       classif.lda         0.2788
## 9                Sonar-example classif.naiveBayes        0.3028
```

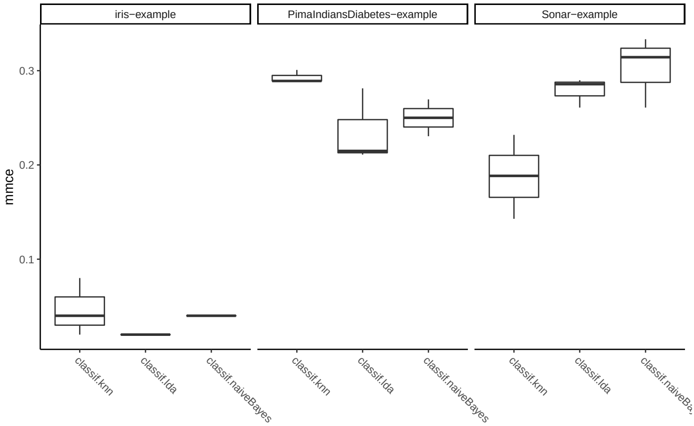
Essential Data Science Training

```
# aggregated data:
getBMRAggrPerformances(bm, as.df = TRUE)

##                          task.id         learner.id mmce.test.mean
## 1                   iris-example        classif.knn         0.0467
## 2                   iris-example        classif.lda         0.0200
## 3                   iris-example classif.naiveBayes         0.0400
## 4 PimaIndiansDiabetes-example        classif.knn         0.2930
## 5 PimaIndiansDiabetes-example        classif.lda         0.2357
## 6 PimaIndiansDiabetes-example classif.naiveBayes         0.2500
## 7                  Sonar-example        classif.knn         0.1877
## 8                  Sonar-example        classif.lda         0.2788
## 9                  Sonar-example classif.naiveBayes         0.3028
```

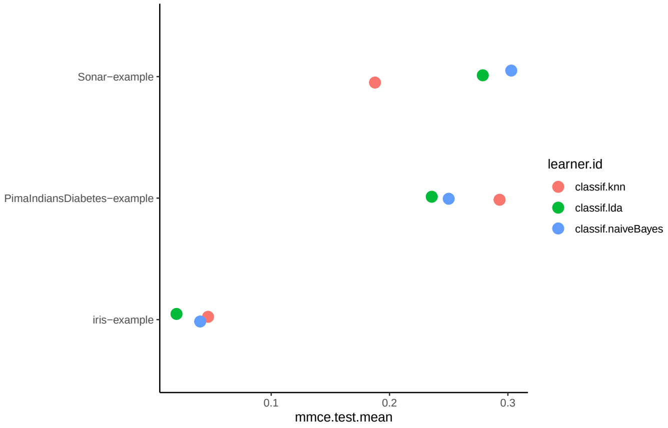
```
# complete data:
head(as.data.frame(bm), 10)

##                            task.id          learner.id iter  mmce
## 1                     iris-example         classif.knn    1 0.040
## 2                     iris-example         classif.knn    2 0.020
## 3                     iris-example         classif.knn    3 0.080
## 4                     iris-example         classif.lda    1 0.020
## 5                     iris-example         classif.lda    2 0.020
## 6                     iris-example         classif.lda    3 0.020
## 7                     iris-example  classif.naiveBayes    1 0.040
## 8                     iris-example  classif.naiveBayes    2 0.040
## 9                     iris-example  classif.naiveBayes    3 0.040
## 10 PimaIndiansDiabetes-example         classif.knn    1 0.301
```

# Classification Analysis: Benchmarking

```r
plotBMRBoxplots(bm, pretty.names = FALSE)
```



```r
plotBMRSummary(bm, pretty.names = FALSE)
```



# TUNING

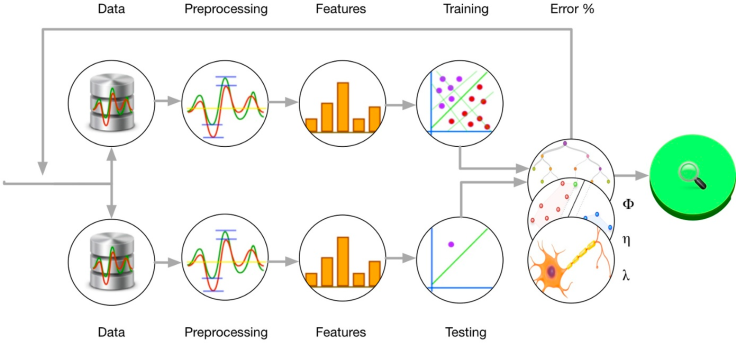
# Hyperparameter Tuning

- Many parameters or decisions for an ML algorithm are not decided by the fitting procedure
- **Model parameters** are optimized during training, by some form of loss minimization. They are an **output** of the training. E.g., the coefficients of a linear model or the optimal splits of a tree learner.
- **Hyperparameters** must be specified before the training phase. They are an **input** of the training. E.g., how small a leaf can become for a tree; $k$ and which distance measure to use for kNN

- HPs have to be set either by the user or by (smart) default values
- Our goal is to optimize these w.r.t. the estimated prediction error; this implies an independent test set, or cross-validation
- The same applies to preprocessing, feature construction and other model-relevant operations. In general we might be interested in optimizing an entire ML "pipeline"

# ML Pipeline




Data
Preprocessing
Features
Training
Error %
Φ
η
λ
Data
Preprocessing
Features
Testing

# Why tuning is important

- Hyperparameters control the complexity of a model, i.e., how flexible the model is
- If a model is too flexible so that it simply "memorizes" the training data, we will face the dreaded problem of overfitting
- Hence, control of capacity, i.e., proper setting of hyperparameters can prevent overfitting the model on the training set
- Many other factors like optimization control settings, distance functions, scaling, algorithmic variants in the fitting procedure can heavily influence model performance in non-trivial ways. It is extremely hard to guess the correct choices here.

# Types of hyperparameters

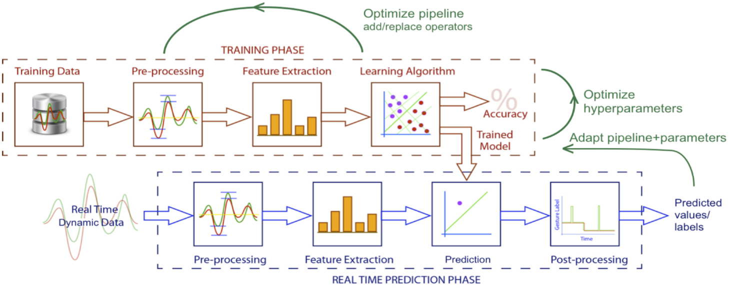
- Numerical parameters (real valued / integers)
  - *mtry* in a random forest
  - Neighborhood size $k$ for kNN
- Categorical parameters:
  - Which split criterion for classification trees?
  - Which distance measure for kNN?
- Ordinal parameters:
  - $\{\texttt{low}, \texttt{medium}, \texttt{high}\}$
- Dependent parameters:
  - If we use the Gaussian kernel for the SVM, what is its width?

# Components of tuning problem

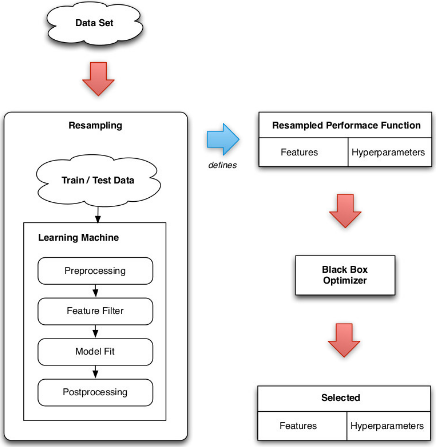
- The learner (possibly: several competing learners?)
- The performance measure. Determined by the application. Not necessarily identical to the loss function that the learner tries to minimize. We could even be interested in multiple measures simultaneously, e.g., accuracy and sparseness of our model, TPR and PPV, etc.
- A (resampling) procedure for estimating the predictive performance
- The learner's hyperparameters and their respective regions-of-interest over which we optimize

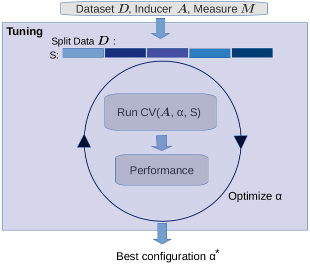
Tuner proposes configuration, eval by resampling, tuner receives performance, iterate

Tuner proposes configuration, eval by resampling, tuner receives performance, iterate



Tuner proposes configuration, eval by resampling, tuner receives performance, iterate



# Why is tuning so hard?



- Tuning is derivative-free ("black box problem"): It is usually impossible to compute derivatives of the objective (i.e., the resampled performance measure) that we optimize with regard to the HPs. All we can do is evaluate the performance for a given hyperparameter configuration.
- Every evaluation requires one or multiple train and predict steps of the learner, i.e., every evaluation is very **expensive**.
- Even worse: the answer we get from that evaluation is **not exact, but stochastic** in most settings, as we use resampling.

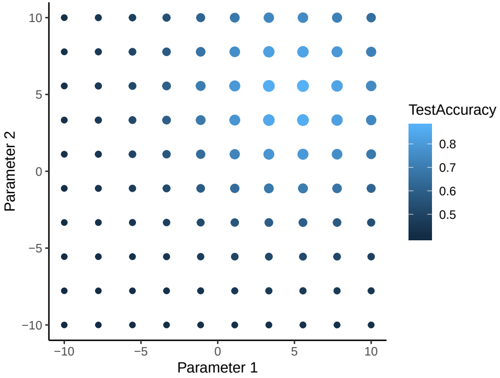
# Why is tuning so hard?



- Categorical and dependent hyperparameters aggravate our difficulties: the space of hyperparameters we optimize over has a non-metric, complicated structure.
- For large and difficult problems parallelizing the computation seems relevant, to evaluate multiple HP configurations in parallel or to speed up the resampling-based performance evaluation

# Grid search

- Simple technique which is still quite popular, tries all HP combinations on a multi-dimensional discretized grid
- For each hyperparameter a finite set of candidates is predefined
- We simply search all possible combinations in arbitrary order



# Grid search



## Advantages

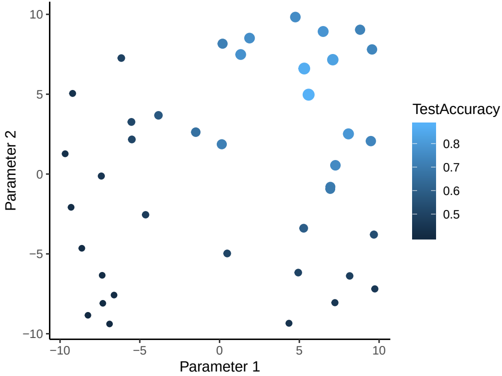
- Very easy to implement, therefore very popular
- All parameter types possible
- Parallelization is trivial

## Disadvantages

- Combinatorial explosion, inefficient
- Searches large irrelevant areas
- Which values / discretization?

# Random search

- Small variation of grid search
- Uniformly sample from the region-of-interest



### Advantages

- Very easy to implement, therefore very popular
- All parameter types possible
- Parallelization is trivial
- Anytime algorithm - we can always increase the budget when we are not satisfied
- Often better than grid search, as each individual parameter has been tried with $m$ different values, when the search budget was $m$. Mitigates the problem of discretization

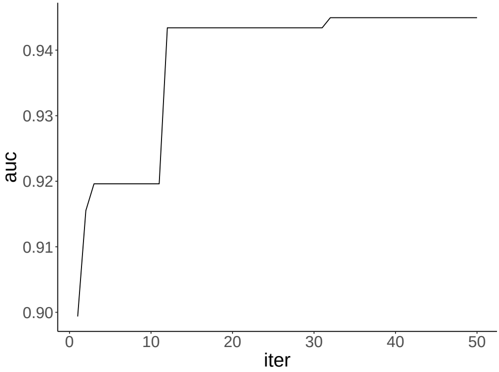
### Disadvantages

- As for grid search, many evaluations in areas with low likelihood for improvement

Essential Data Science Training

Tuning gradient boosting with random search and 5CV on the spam data set for AUC

| Parameter | Type | Min | Max |
|---|---|---|---|
| n.trees | integer | 3 | 500 |
| shrinkage | numeric | 0 | 1 |
| interaction | integer | 1 | 5 |
| bag.fraction | numeric | 0.2 | 0.9 |



# Advanced Tuning Techniques

- Stochastic local search, e.g. simulated annealing
- Genetic algorithms / CMAES
- Iterated F-Racing
- Model-based Optimization / Bayesian Optimization
- Hyperband
- ...

Essential Data Science Training

```
lrn = makeLearner("classif.gbm", predict.type="prob")
ps = getParamSet(makeLearner("classif.gbm"))
print(ps, constr.clip=10)

##                       Type len   Def      Constr Req Tunable Trafo
## distribution      discrete   -     -  bernoul...   -    TRUE     -
## n.trees            integer   -   100    1 to Inf   -    TRUE     -
## cv.folds           integer   -     0 -Inf to Inf   -    TRUE     -
## interaction.depth  integer   -     1    1 to Inf   -    TRUE     -
## n.minobsinnode     integer   -    10    1 to Inf   -    TRUE     -
## shrinkage          numeric   - 0.001    0 to Inf   -    TRUE     -
## bag.fraction       numeric   -   0.5      0 to 1   -    TRUE     -
## train.fraction     numeric   -     1      0 to 1   -    TRUE     -
## keep.data          logical   -  TRUE           -   -   FALSE     -
## verbose            logical   - FALSE           -   -   FALSE     -
```

# Hyperparameters in mlr

- Either set them in constructor, or change them later:

```
lrn = makeLearner("classif.gbm", predict.type="prob", shrinkage = 0.1)
lrn = setHyperPars(lrn, distribution = "bernoulli", shrinkage = 0.2)
```

# Tuning in mlr

- Create a set of parameters
- Here we optimize boosting

```
par.set = makeParamSet(
  makeIntegerParam("n.trees", lower = 3, upper = 20),
  makeNumericParam("shrinkage", lower = 0, upper = 0.2)
)
```

Essential Data Science Training

Optimize the hyperparameter of learner

```
tune.ctrl = makeTuneControlRandom(maxit = 50L)
tr = tuneParams(lrn, task = sonar.task, par.set = par.set,
  resampling = hout, control = tune.ctrl,
  measures = mlr::auc)
```

```r
tr$x

## $n.trees
## [1] 15
##
## $shrinkage
## [1] 0.0586

tr$y

## auc.test.mean
##         0.781

head(as.data.frame(tr$opt.path), 3L)[, c(1,2,3,7)]

##   n.trees shrinkage auc.test.mean exec.time
## 1      16    0.1721         0.770      0.20
## 2      15    0.1839         0.722      0.04
## 3      13    0.0893         0.729      0.01
```

# Parameter Types

```
makeNumericParam("x" ,lower = -1, upper = 1)
makeIntegerParam("x" ,lower = -1L, upper = 1L)
makeDiscreteParam("x" ,values = c("a", "b", "c"))
makeLogicalParam("x")
```

and vector-types exist for all param types

```
makeNumericVectorParam("x" , len = 3L, lower = -1, upper = 1)

##              Type len Def  Constr Req Tunable Trafo
## 1 numericvector   3   - -1 to 1   -    TRUE     -
```

# PARALLEL MLR

# Parallelization

- Many tasks in statistics are embarrassingly parallel (independence assumptions, resampling, ...)
- `R` is mostly single-threaded (matrix operations may be parallel, depending on your installation)
- Multiple backends for explicit parallelization available:
  - Multicore (packages parallel/multicore)
  - Socket and MPI cluster (packages parallel/snow/Rmpi)
  - HPC-Clusters (package `batchtools`): SLURM, Torque/PBS, SGE, LSF, Docker, SSH makeshift clusters, ...

# Parallelization



- We use `parallelMap` in `mlr` an abstraction for all backends
- Initialize with `parallelStart()`
- Parallelize function call with `parallelMap()`/`parallelLapply()`/...
- Stop with `parallelStop()`

```
parallelStartSocket(4)
parallelMap(function(x) x^2, 1:10)
parallelStop()
```

# Parallelization



- The first loop which is marked as parallel executable will be automatically parallelized
- Which loop is suited best for parallelization depends on the number of iterations
- Levels allow fine grained control over the parallelization
  - `mlr.resample`: Each resampling iteration (a train / test step) is a parallel job.
  - `mlr.benchmark`: Each experiment "run this learner on this data set" is a parallel job.
  - `mlr.tuneParams`: Each evaluation in hyperparameter space "resample with these parameter settings" is a parallel job. How many of these can be run independently in parallel depends on the tuning algorithm.
  - `mlr.selectFeatures`: Each evaluation in feature space "resample with this feature subset" is a parallel job.

```
lrns = list(makeLearner("classif.rpart"), makeLearner("classif.svm"))
rdesc = makeResampleDesc("Bootstrap", iters = 100)

parallelStartSocket(4)

## Starting parallelization in mode=socket with cpus=4.

bm = benchmark(learners = lrns, tasks = iris.task, resamplings = rdesc)

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Mapping in parallel: mode = socket; level = mlr.benchmark; cpus = 4; elemen

parallelStop()

## Stopped parallelization. All cleaned up.
```

Essential Data Science Training

## Parallelize the bootstrap instead:

```
parallelStartSocket(4, level = "mlr.resample")

## Starting parallelization in mode=socket with cpus=4.

bm = benchmark(learners = lrns, tasks = iris.task, resamplings = rdesc)

## Task: iris-example, Learner: classif.rpart

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Resampling: OOB bootstrapping

## Measures:             mmce

## Mapping in parallel: mode = socket; level = mlr.resample; cpus = 4; elements = 100.

##

## Aggregated Result: mmce.test.mean=0.060

##

## Task: iris-example, Learner: classif.svm

## Exporting objects to slaves for mode socket: .mlr.slave.options
```

# EXERCISE 2