WHAT IS MACHINE LEARNING
Typical Workflow

Plan
- What is the question I'm answering?
- What data will I need?

Acquire the data

Clean Data
- Analyse data quality
- Reformat Impute etc.

Analyse Data
- Create features
- Create model
- Evaluate results

Script
Visualise

Create report
Deploy in Production
Publish & Share

End here
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, ...).
- $y$ 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)

Iris flower data

Species
- setosa
- versicolor
- virginica
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
The **good** news:

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

```r
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 a 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
Core objects: tasks, learners, measures, resampling instances.
Task Abstraction

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

```r
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
```r
lnr = makeLearner("classif.knn", k = 2)
print(lnr)
## Learner classif.knn from package class
## Type: classif
## Name: k-Nearest Neighbor; Short name: knn
## Class: classif.knn
## Properties: twoclass,multiclass,numeric
## Predict-Type: response
## Hyperparameters: k=2

getParamSet(lnr)
## 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 -
```
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
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](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:

```r
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",...
```
Iris Data Set

Cor : −0.118
setosa: 0.743
versicolor: 0.526
virginica: 0.457

Cor : 0.872
setosa: 0.267
versicolor: 0.754
virginica: 0.281

Cor : −0.428
setosa: 0.178
versicolor: 0.561
virginica: 0.401

Cor : 0.818
setosa: 0.278
versicolor: 0.546
virginica: 0.281

Cor : −0.366
setosa: 0.233
versicolor: 0.664
virginica: 0.538

Cor : 0.963
setosa: 0.332
versicolor: 0.787
virginica: 0.322

Sepal.Length
Sepal.Width
Petal.Length
Petal.Width
Species

Cor : −0.428
setosa: 0.178
versicolor: 0.561
virginica: 0.401

Cor : −0.366
setosa: 0.233
versicolor: 0.664
virginica: 0.538

Cor : 0.963
setosa: 0.332
versicolor: 0.787
virginica: 0.322

Cor : −0.118
setosa: 0.743
versicolor: 0.526
virginica: 0.457

Cor : 0.872
setosa: 0.267
versicolor: 0.754
virginica: 0.281

Cor : −0.428
setosa: 0.178
versicolor: 0.561
virginica: 0.401

Cor : 0.818
setosa: 0.278
versicolor: 0.546
virginica: 0.281

Cor : −0.366
setosa: 0.233
versicolor: 0.664
virginica: 0.538

Cor : 0.963
setosa: 0.332
versicolor: 0.787
virginica: 0.322

Setosa
Versicolor
Virginica
Define a mlr task for the iris data using the target column Species:

```r
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
- ...

FIRST DATA ANALYSIS www.essentialds.de
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.

<table>
<thead>
<tr>
<th>Class / Short Name / Name</th>
<th>Packages</th>
<th>Num.</th>
<th>Fac.</th>
<th>Ord.</th>
<th>NAs</th>
<th>Weights</th>
<th>Props</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>classif.ada ada</td>
<td>ada</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>prob</td>
<td>xval has been set to 0 by default for speed.</td>
</tr>
<tr>
<td>ada Boosting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>classif.adaboostm1 adaboostm1</td>
<td>RWeka</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>prob</td>
<td></td>
</tr>
<tr>
<td>ada Boosting M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>twoclass</td>
<td></td>
</tr>
<tr>
<td>classification bartMachine</td>
<td>bartMachine</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>prob</td>
<td></td>
</tr>
<tr>
<td>bartmachine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>twoclass</td>
<td></td>
</tr>
<tr>
<td>Bayesian Additive Regression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>classification binomial</td>
<td>stats</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>prob</td>
<td></td>
</tr>
<tr>
<td>binomial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>twoclass</td>
<td></td>
</tr>
<tr>
<td>Binomial Regression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>classification boosting</td>
<td>adabag</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>prob</td>
<td></td>
</tr>
<tr>
<td>adabag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>twoclass</td>
<td></td>
</tr>
<tr>
<td>Adabag Boosting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>multiclass</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>featImp</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xval has been set to 0 by default for speed.</td>
</tr>
</tbody>
</table>
What Learners are available?

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

```r
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:

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

```r
getParamSet(lrn.knn)
```

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>len</th>
<th>Def</th>
<th>Constr</th>
<th>Req</th>
<th>Tunable</th>
<th>Trafo</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>integer</td>
<td>-</td>
<td>7</td>
<td>1 to Inf</td>
<td>-</td>
<td>TRUE</td>
<td>-</td>
</tr>
<tr>
<td>distance</td>
<td>numeric</td>
<td>-</td>
<td>2</td>
<td>0 to Inf</td>
<td>-</td>
<td>TRUE</td>
<td>-</td>
</tr>
<tr>
<td>kernel</td>
<td>discrete</td>
<td>-</td>
<td>optimal rectangular, triangular</td>
<td>-</td>
<td>TRUE</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td>logical</td>
<td>-</td>
<td>TRUE</td>
<td>-</td>
<td>-</td>
<td>TRUE</td>
<td>-</td>
</tr>
</tbody>
</table>
A model is usually trained on a subset of the data - the remaining part is used to evaluate its performance.

```r
n = getTaskSize(task.iris)
train.ind = sample(n, n/2)
test.ind = setdiff(1:n, train.ind)
str(train.ind)
## int [1:75] 14 50 118 43 150 148 90 91 ... 

str(test.ind)
## int [1:75] 1 2 3 5 8 10 11 12 ... 
```
# 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.

```r
# 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
## 1 1 setosa 0.957 0.0426 0 setosa
## 2 2 setosa 0.927 0.0733 0 setosa
## 3 3 setosa 0.938 0.0617 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
## 1 setosa 0.957 0.0426 0 setosa
## 2 setosa 0.927 0.0733 0 setosa
## 3 setosa 0.938 0.0617 0 setosa
```
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
## 1 0.957 0.0426 0
## 2 0.927 0.0733 0
## 3 0.938 0.0617 0

head(truth, 3)
## [1] setosa setosa setosa
## Levels: setosa versicolor virginica
First Classification Analysis: Evaluation

# total number of errors
sum(pred.class != truth)
## [1] 7

# mean misclassification error (MMCE)
mean(pred.class != truth)
## [1] 0.0933

# percentage of accurate predictions (ACC, accuracy)
mean(pred.class == truth)
## [1] 0.907
### First Classification Analysis: Evaluation

```r
calculateConfusionMatrix(preds)

## predicted
## true  setosa versicolor virginica -err.-
## setosa    27          0          0          0
## versicolor  0         22          6          6
## virginica   0          1         19          1
## -err.-      0          1          6          7
```
First Classification Analysis: Evaluation

```r
performance(preds, measures = mmce)  # mean misclassification error
## mmce
## 0.0933

performance(preds, measures = list(mmce, acc))
## mmce  acc
## 0.0933 0.9067

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

Predictions for learner fitted on two features.
EXERCISE 1
RESAMPLING
task = iris.task
n = getTaskSize(task)
ratio = \( \frac{2}{3} \)

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
## 0.04
# Define learner:
```r
lrn = makeLearner("classif.randomForest", predict.type = "prob")
```

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

```r
print(r)
## Resample Result
## Task: spam-example
## Learner: classif.randomForest
## Aggr perf: acc.test.mean=0.950, auc.test.mean=0.986
## Runtime: 16.4351
```
Cross-validation in mlr

```r
head(r$measures.test)
## iter acc auc
## 1 1 0.950 0.985
## 2 2 0.947 0.984
## 3 3 0.952 0.990

head(as.data.frame(r$pred))
## id truth prob.nonspam prob.spam response iter set
## 1 1816 nonspam 1.000 0.000 nonspam 1 test
## 2 1819 nonspam 0.990 0.010 nonspam 1 test
## 3 1820 nonspam 0.992 0.008 nonspam 1 test
## 4 1824 nonspam 0.902 0.098 nonspam 1 test
## 5 1828 nonspam 0.988 0.012 nonspam 1 test
## 6 1829 nonspam 0.978 0.022 nonspam 1 test
```
## Parameter of `makeResamplingDesc`

<table>
<thead>
<tr>
<th>Methods</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CV</td>
<td>iters</td>
<td>Number of iterations</td>
</tr>
<tr>
<td>LOO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RepCV</td>
<td>reps</td>
<td>Repeats for repeated CV</td>
</tr>
<tr>
<td></td>
<td>folds</td>
<td>Folds in the repeated CV</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>iters</td>
<td>Number of iterations</td>
</tr>
<tr>
<td>Subsample</td>
<td>iters</td>
<td>Number of iterations</td>
</tr>
<tr>
<td></td>
<td>split</td>
<td>Proportion of training cases</td>
</tr>
<tr>
<td>Holdout</td>
<td>split</td>
<td>Proportion of training cases</td>
</tr>
</tbody>
</table>
1. Explicitly define resampling:

```r
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)
```

```r
res1
```

## Resample Result
## Task: iris-example
## Learner: classif.randomForest
## Aggr perf: mmce.test.mean=0.053
## Runtime: 0.297201

Other pre defined objects are cv2, cv3 and cv5.
2. Use `crossval`:

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

```r
## Resample Result
## Task: iris-example
## Learner: classif.randomForest
## Aggr perf: mmce.test.mean=0.047
## Runtime: 0.315572
```

Similar functions are `repcv`, `holdout`, `subsample`, `bootstrapOOB`, `bootstrapB632` and `bootstrapB632plus`. 
# quick way to compare learners with identical train/test splits

```r
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 0.04
## 2 iris-example classif.lda 0.02
## 3 iris-example classif.naiveBayes 0.04
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 0.0400
## 2 iris-example classif.lda 0.0200
## 3 iris-example classif.naiveBayes 0.0467
## 4 PimaIndiansDiabetes-example classif.knn 0.3034
## 5 PimaIndiansDiabetes-example classif.lda 0.2331
## 6 PimaIndiansDiabetes-example classif.naiveBayes 0.2500
## 7 Sonar-example classif.knn 0.1734
## 8 Sonar-example classif.lda 0.2883
## 9 Sonar-example classif.naiveBayes 0.2981
## aggregated data:

```r
getBMRAggrPerformances(bm, as.df = TRUE)
```

<table>
<thead>
<tr>
<th>task.id</th>
<th>learner.id</th>
<th>mmce.test.mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iris-example classif.knn</td>
<td>0.0400</td>
</tr>
<tr>
<td>2</td>
<td>iris-example classif.lda</td>
<td>0.0200</td>
</tr>
<tr>
<td>3</td>
<td>iris-example classif.naiveBayes</td>
<td>0.0467</td>
</tr>
<tr>
<td>4</td>
<td>PimaIndiansDiabetes-example classif.knn</td>
<td>0.3034</td>
</tr>
<tr>
<td>5</td>
<td>PimaIndiansDiabetes-example classif.lda</td>
<td>0.2331</td>
</tr>
<tr>
<td>6</td>
<td>PimaIndiansDiabetes-example classif.naiveBayes</td>
<td>0.2500</td>
</tr>
<tr>
<td>7</td>
<td>Sonar-example classif.knn</td>
<td>0.1734</td>
</tr>
<tr>
<td>8</td>
<td>Sonar-example classif.lda</td>
<td>0.2883</td>
</tr>
<tr>
<td>9</td>
<td>Sonar-example classif.naiveBayes</td>
<td>0.2981</td>
</tr>
</tbody>
</table>
Classification Analysis: Benchmarking

# 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 0.080
## 3 iris-example classif.knn 3 0.000
## 4 iris-example classif.lda 1 0.040
## 5 iris-example classif.lda 2 0.020
## 6 iris-example classif.lda 3 0.000
## 7 iris-example classif.naiveBayes 1 0.040
## 8 iris-example classif.naiveBayes 2 0.080
## 9 iris-example classif.naiveBayes 3 0.020
## 10 PimaIndiansDiabetes-example classif.knn 1 0.324
Classification Analysis: Benchmarking

plotBMRBoxplots(bm, pretty.names = FALSE)
Classification Analysis: Benchmarking

\[ \text{plotBMRSummary}(\text{bm}, \text{pretty.names} = \text{FALSE}) \]
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 $k$NN.
- 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 %

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:**
  - \{low, medium, 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 Diagram](image-url)

Test Accuracy Heatmap

Parameter 1 vs Parameter 2 with Test Accuracy values.
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
Random search

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>n.trees</td>
<td>integer</td>
<td>3</td>
<td>500</td>
</tr>
<tr>
<td>shrinkage</td>
<td>numeric</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>interaction</td>
<td>integer</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>bag.fraction</td>
<td>numeric</td>
<td>0.2</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Advanced Tuning Techniques

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

```r
lrn = makeLearner("classif.gbm", predict.type="prob", shrinkage = 0.1)
lrn = setHyperPars(lrn, distribution = "bernoulli", shrinkage = 0.2)
```
Create a set of parameters

Here we optimize boosting

```r
par.set = makeParamSet(
    makeIntegerParam("n.trees", lower = 3, upper = 20),
    makeNumericParam("shrinkage", lower = 0, upper = 0.2)
)
```
Optimize the hyperparameter of learner

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

\[
\begin{align*}
\text{tr} \times \\
\text{## $n.trees$} \\
\text{## [1] 5} \\
\text{## $shrinkage$} \\
\text{## [1] 0.2} \\
\text{tr} \times y \\
\text{## auc.test.mean} \\
\text{## 0.849} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{## n.trees shrinkage auc.test.mean exec.time} \\
\text{## 1 6 0.181 0.792 0.05} \\
\text{## 2 6 0.121 0.806 0.03} \\
\text{## 3 10 0.134 0.813 0.06} \\
\end{align*}
\]
```r
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

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

## Type len Def Constr Req Tunable Trafo

```r
## 1 numericvector 3 -1 to 1 - TRUE -
```
PARALLEL MLR
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, ...
We use parallelMap in mlr an abstraction for all backends

- Initialize with parallelStart()

- Parallelize function call with
  parallelMap() / parallelLapply() / ...

- Stop with parallelStop()

```r
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; elements

parallelStop()

## Stopped parallelization. All cleaned up.
Parallelization

Parallelize the bootstrap instead:

```r
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=0.063
##
## Task: iris-example, Learner: classif.svm
## 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=0.042
##
parallelStop()
## Stopped parallelization. All cleaned up.
```
EXERCISE 2