Geocomputation with R

⚔

Spatial cross-validation with **mlr**

Jannes Muenchow

GeoStats 2018

Find the slides and code

https://github.com/geocompr/geostats_18

Please install following packages:

```
list_of_packages = c("sf", "raster", "mlr", "ranger", "RQGIS",
                     "parallelMap")
install.packages(list_of_packages)
```

Contents of the tutorial

1. Study area, data and aim

Contents of the tutorial

1. Study area, data and aim
2. Introduction to **(spatial) cross-validation**

Contents of the tutorial

1. Study area, data and aim
2. Introduction to **(spatial) cross-validation**
3. **mlr** building blocks

Contents of the tutorial

1. Study area, data and aim
2. Introduction to **(spatial) cross-validation**
3. **mlr** building blocks
4. Random forest modeling with **mlr**

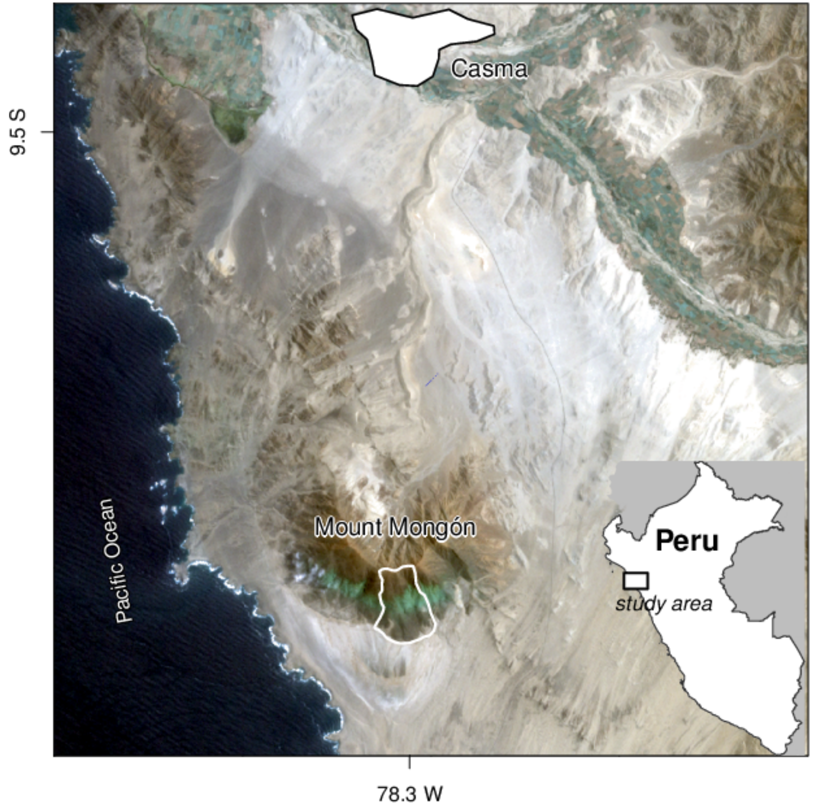
Contents of the tutorial

1. Study area, data and aim
2. Introduction to **(spatial) cross-validation**
3. **mlr** building blocks
4. Random forest modeling with **mlr**

Study area, data and aim

Study area

Where are we? Mount Mongón near Casma in northern Peru.



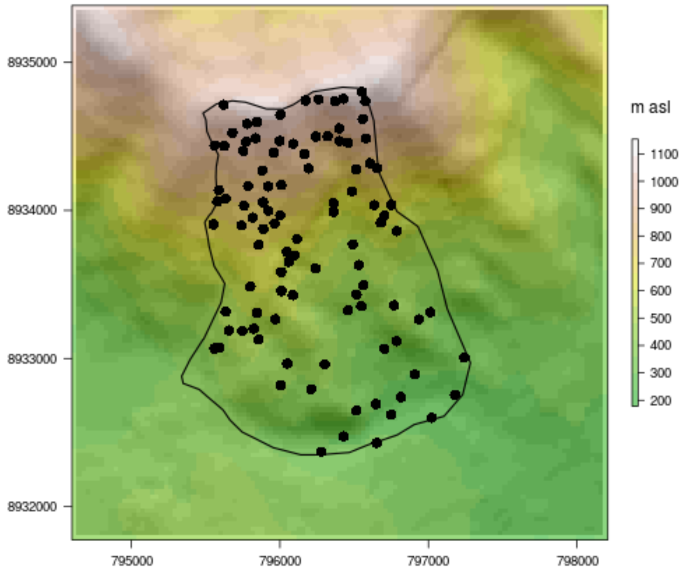
Austral summer

Mount Mongón in summer

Mount Mongón in austral winter

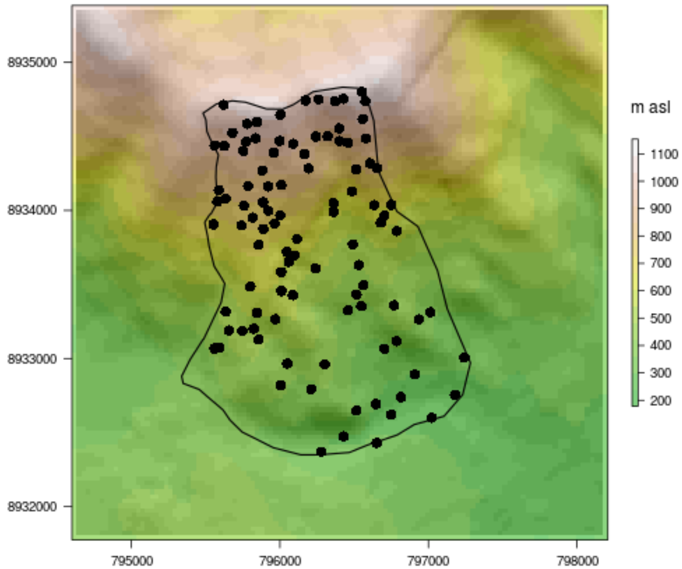
Data

- 100 randomly distributed plots
- coverage of all vascular plants in each plot
- First NMDS axis represents the main gradient (our response, see Chapter 14 of *Geocomputation with R* (Lovelace, Nowosad, and Muenchow, 2018)



Aim

- model the floristic gradient as a function of environmental predictors using a random forest model
- spatial cross-validation to retrieve a bias-reduced estimate of the model's performance
- tune hyperparameters for the predictive mapping of the floristic gradient
- but before we do that, we will introduce the **mlr** building blocks to easily do spatial cross-validation with a simple `lm` (though this definitely is not be the most appropriate model for our data...)

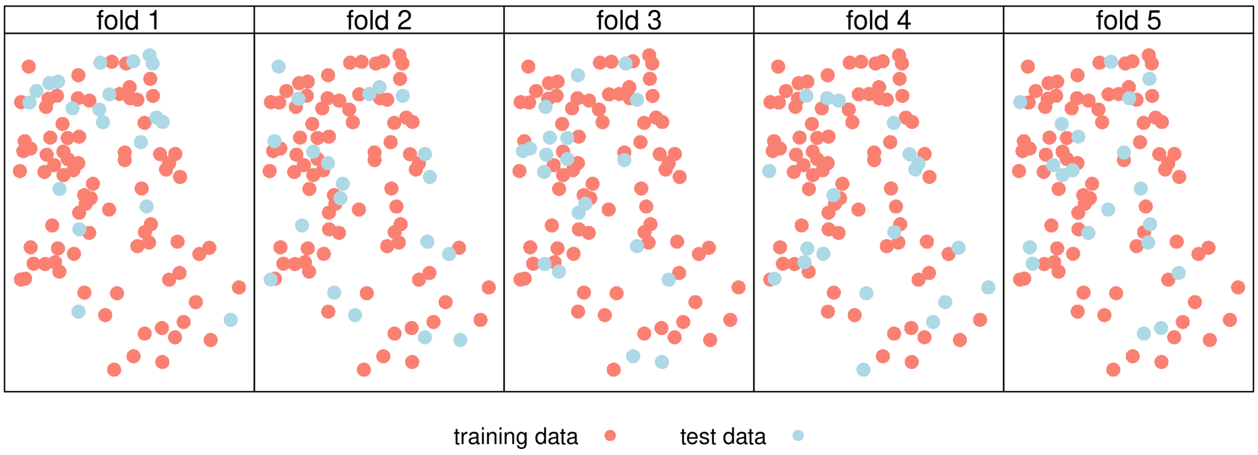


Introduction to (spatial) cross-validation

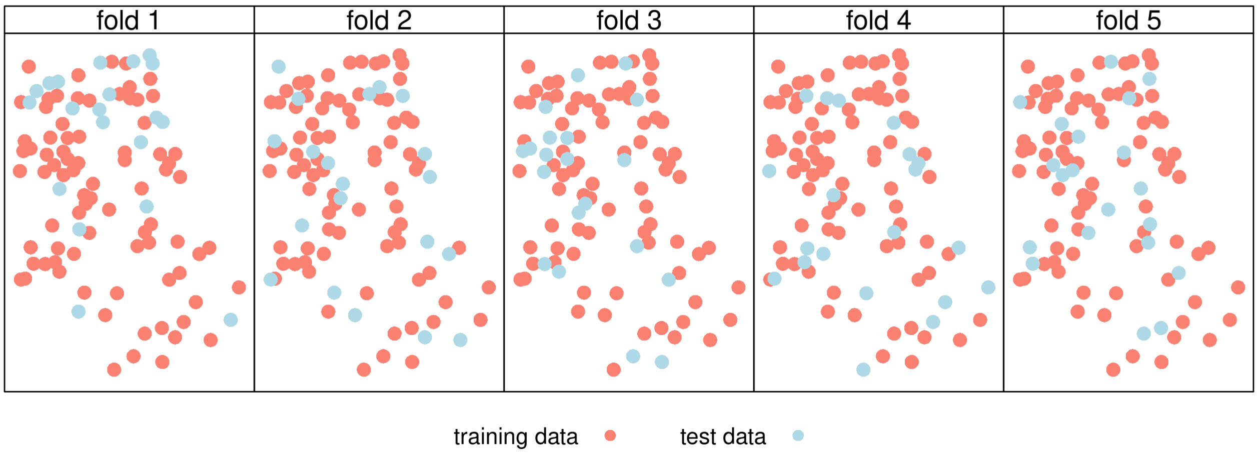
Cross-validation

Aim of spatial cross-validation is to find out how generalizable a model is. Fitting to closely the input data, including its noise, leads to a bad predictive performance on unseen data (overfitting).

Random partitioning



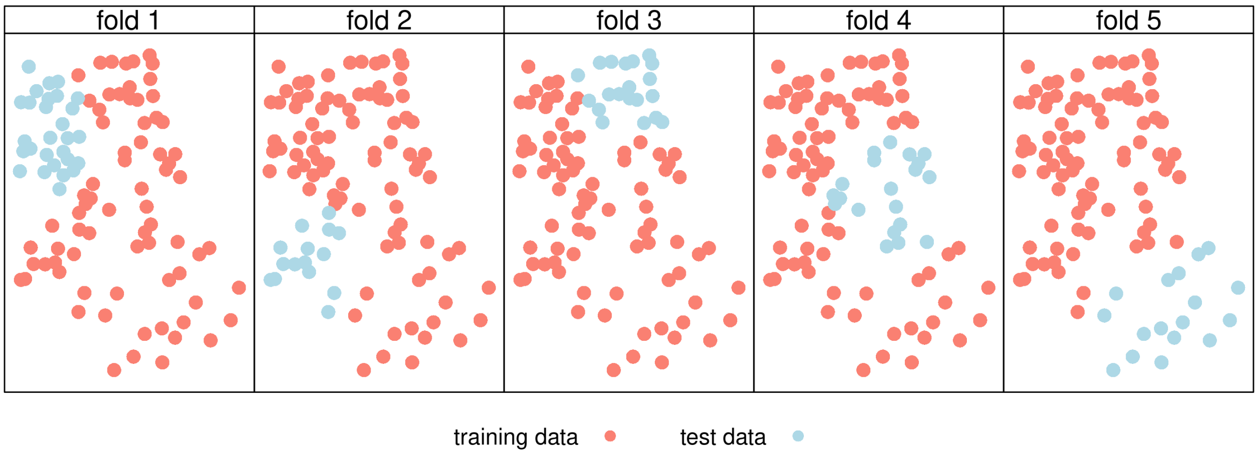
Random partitioning



Problems with conventional random partitioning when using spatial data:

- violation of the fundamental independence assumption in cross-validation
- which subsequently leads to overoptimistic, i.e. biased results
- Solution: use spatial partitioning for a bias-reduced assessment of a **model's performance**

Spatial partitioning



mlr building blocks

Input data

We are already in possession of the following data

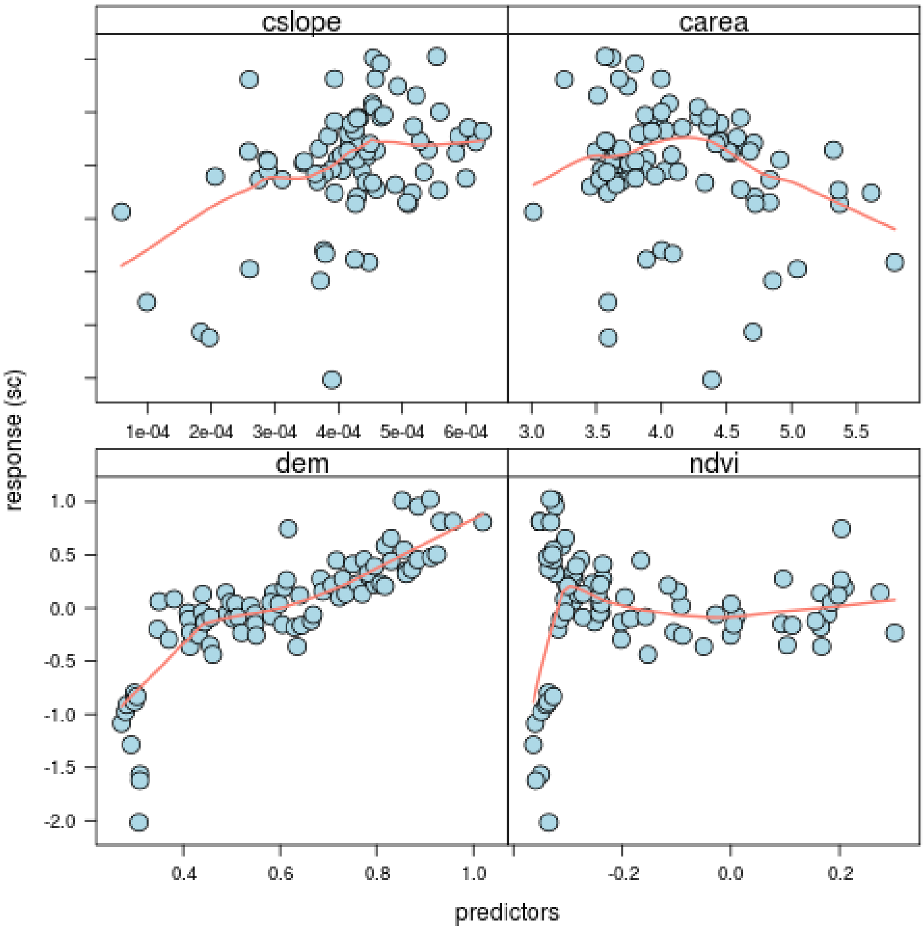
```r
library(mlr)
library(dplyr)
library(sf)
# response-predictor dataframe
head(rp, 2)
```

```
##     dem       ndvi       cslope    carea         sc
## 1 0.272 -0.3603059 0.0003712326 4.854817 -1.0843143
## 2 0.280 -0.3487794 0.0002598672 5.043667 -0.9752411
```

```r
# coordinates
head(coords, 2)
```

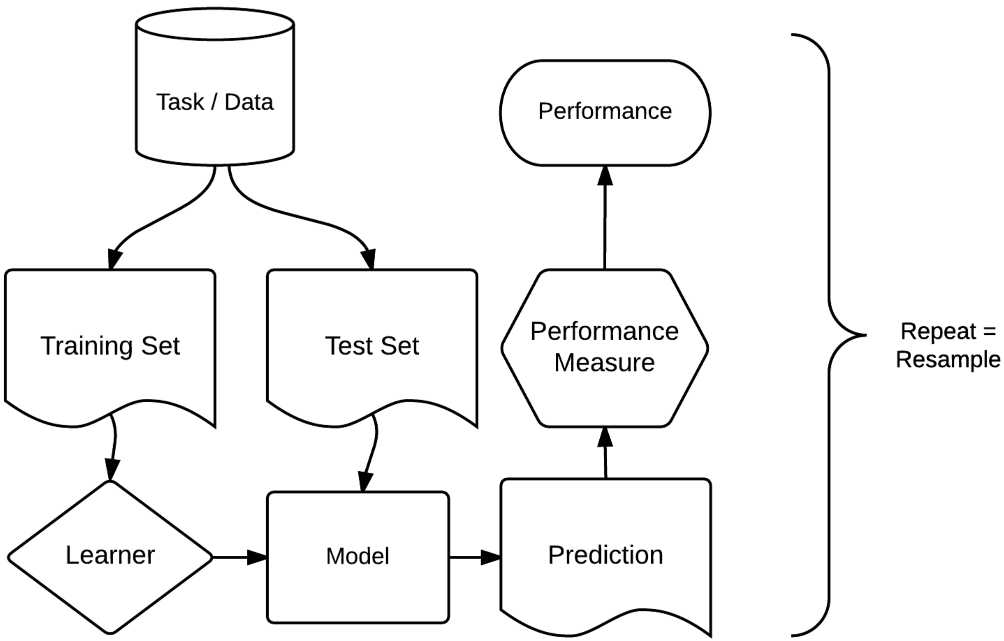
```
##          x       y
## 1 797178.6 8932755
## 2 796749.3 8932621
```

Little data exploration



Building blocks

mlr is a metapackage that lets you combine hundreds of modeling algorithms of many different package within a single framework (Bischl, Lang, Kotthoff, Schiffner, Richter, Studerus, Casalicchio, and Jones, 2016)



Source: openml.github.io

Create a task

```r
library(mlr)
# create task
task = makeRegrTask(data = rp, target = "sc",
                    coordinates = coords)
```

Learner

To find out which learners are available for a specific task run:

```
lrns = listLearners(task, warn.missing.packages = FALSE)
dplyr::select(lrns, class, name, short.name, package)
```

Learner

To find out which learners are available for a specific task run:

```
lrns = listLearners(task, warn.missing.packages = FALSE)
dplyr::select(lrns, class, name, short.name, package)
```

We already know that there is a learner named `regr.lm` for running a simple linear model.

Define the learner

```
lrn = makeLearner(cl = "regr.lm", predict.type = "response")
```

Define the learner

```
lrn = makeLearner(cl = "regr.lm", predict.type = "response")
```

To find out more about the learner, run:

```
# simple lm of the stats package
getLearnerPackages(lrn)
helpLearner(lrn)
```

Define the learner

```
lrn = makeLearner(cl = "regr.lm", predict.type = "response")
```

To find out more about the learner, run:

```
# simple lm of the stats package
getLearnerPackages(lrn)
helpLearner(lrn)
```

Just to convince you that we are really using a simple `lm`, let us retrieve the learner model:

```
getLearnerModel(train(lrn, task))
```

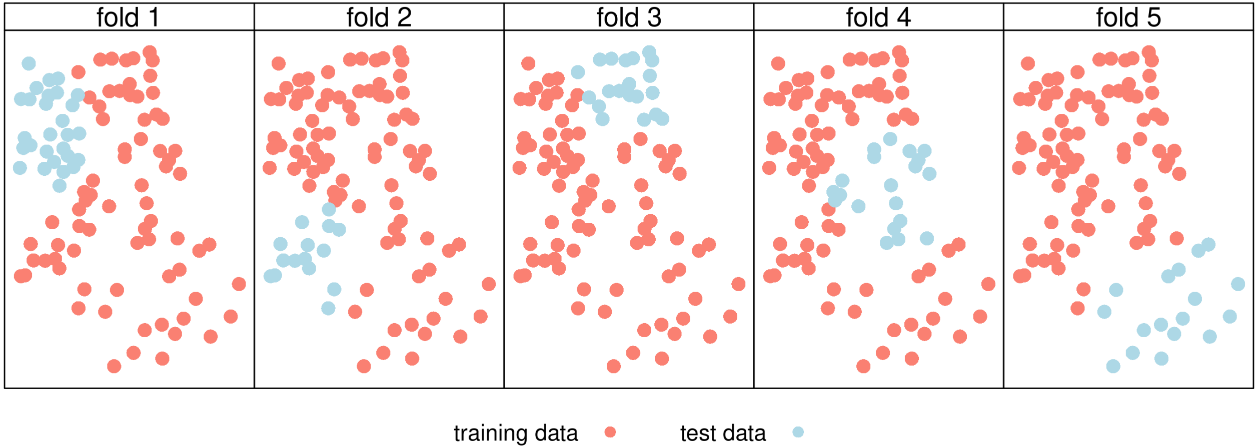
```
##
## Call:
## stats::lm(formula = f, data = d)
##
## Coefficients:
## (Intercept)          dem         ndvi       cslope        carea
##     -0.8854       2.1124       0.2853     526.8357      -0.1389
```

Define spatial partitioning

```
# performance level
perf_level = makeResampleDesc(method = "SpRepCV",
                              folds = 5,
                              reps = 100)
```

Define spatial partitioning

```
# performance level
perf_level = makeResampleDesc(method = "SpRepCV",
                              folds = 5,
                              reps = 100)
```



Execute the resampling

```
cv_sp_lm = mlr::resample(
  task = task,
  learner = lrn,
  resampling = perf_level,
  # specify the performance measure
  measures = mlr::rmse)
```

```
## Resampling: repeated spatial cross-validation

## Measures:             rmse

## [Resample] iter 1:    0.3320561

## [Resample] iter 2:    0.3523074

## [Resample] iter 3:    0.3655570

## [Resample] iter 4:    0.8188942

## [Resample] iter 5:    0.1668623

## [Resample] iter 6:    0.1640437

## [Resample] iter 7:    0.3120112
```

Have a look at the result

```
cv_sp_lm
```

```
## Resample Result
## Task: rp
## Learner: regr.lm
## Aggr perf: rmse.test.rmse=0.4280988
## Runtime: 2.51188
```

Ok, is this good or not?

Have a look at the result

```
cv_sp_lm
```

```
## Resample Result
## Task: rp
## Learner: regr.lm
## Aggr perf: rmse.test.rmse=0.4280988
## Runtime: 2.51188
```

Ok, is this good or not?

```
range(rp$sc)
```

```
## [1] -2.017518  1.023526
```

Have a look at the result

```
cv_sp_lm
```

```
## Resample Result
## Task: rp
## Learner: regr.lm
## Aggr perf: rmse.test.rmse=0.4280988
## Runtime: 2.51188
```

Ok, is this good or not?

```
range(rp$sc)
```

```
## [1] -2.017518  1.023526
```

Hence, this corresponds to a mean deviation from the true value of (%):

```
cv_sp_lm$aggr / diff(range(rp$sc)) * 100
```

```
## rmse.test.rmse
##       14.07737
```

Random forests

Random forests

Like many other machine learning algorithms, random forests have hyperparameters (James, Witten, Hastie, and Tibshirani, 2013). These hyperparameters are not estimated from the data like the coefficients of (semi-)parametric models (`lm`, `glm`, `gam`) but need to be specified before the learning begins. To find the optimal hyperparameters, one needs to run many models using random hyperparameter values. There are several approaches how to do this, here, we will use a random search with 50 iterations while we limit the tuning space to a specific range in accordance with the literature (Probst, Wright, and Boulesteix, 2018; Schratz, Muenchow, Iturritxa, Richter, and Brenning, 2018).

Again: Define a learner

We can use the already specified regression task... just in case let us repeat it here again

```
task = makeRegrTask(data = rp, target = "sc",
                    coordinates = coords)
```

Again: Define a learner

We can use the already specified regression task… just in case let us repeat it here again

```
task = makeRegrTask(data = rp, target = "sc",
                    coordinates = coords)
```

But we need to change our learner in order to use a random forest model. Here, we will use a random forest implementation of the **ranger** package, i.e. we replace `regr.lm` by `reg.ranger` (again see `listLearners(task)`).

Again: Define a learner

We can use the already specified regression task… just in case let us repeat it here again

```
task = makeRegrTask(data = rp, target = "sc",
                    coordinates = coords)
```

But we need to change our learner in order to use a random forest model. Here, we will use a random forest implementation of the **ranger** package, i.e. we replace `regr.lm` by `reg.ranger` (again see `listLearners(task)`).

```
lrn = makeLearner(cl = "regr.ranger", predict.type = "response")
```

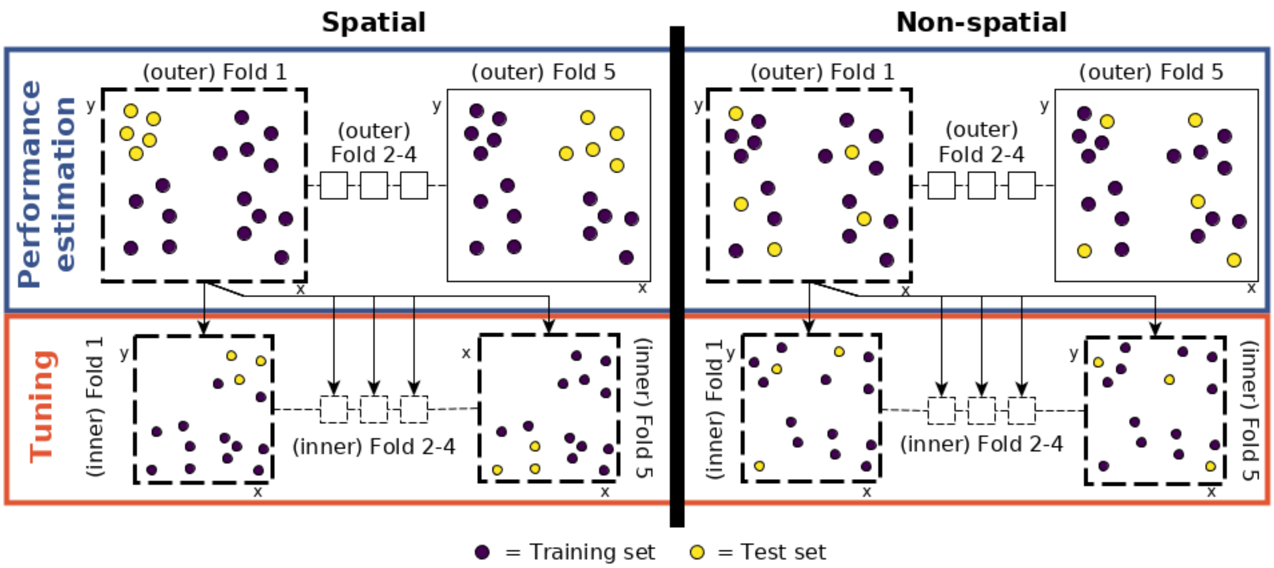
Spatial cross-validation

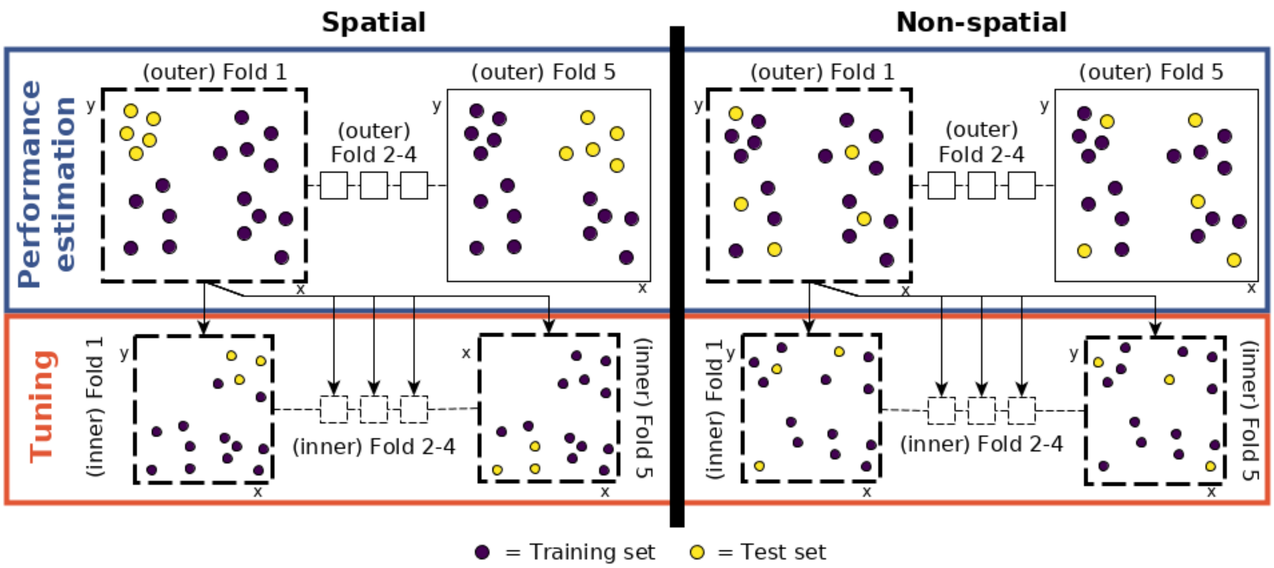
We are already familiar with the spatial cross-validation of the performance level (outer level).

```
perf_level = makeResampleDesc(method = "SpRepCV", folds = 5,
                              reps = 100)
```

Hyperparameter tuning

However, now that we use a random forest model, we need to tune its hyperparameters. And we have to do it in an inner loop using again spatial cross-validation. The inner loop is necessary because tuning the hyperparameters in the performance loop would be like cheating a bit since we then would use the same data for the performance estimation and the hyperparameter tuning. This is called nested spatial cross-validation. A visualization might help (taken from Schratz, Muenchow, Iturritxa, et al. (2018)):


Spatial
Non-spatial
Performance estimation
Tuning
(outer) Fold 1
(outer) Fold 5
(outer) Fold 2-4
(inner) Fold 1
(inner) Fold 2-4
(inner) Fold 5
x
y
= Training set
= Test set



I know this might seem a bit overwhelming in the beginning but it is an easy concept once you get your head around it. You can reread nested spatial cross-validation including hyperparamter tuning in Chapter 11 of *Geocomputation with R* (Lovelace, Nowosad, and Muenchow, 2018)

Hyperparameter tuning

Let us define five spatially disjoint partitions in the tune level (one repetition).

```
tune_level = makeResampleDesc(method = "SpCV", iters = 5)
```

Random search

Next, we need to tell **mlr** to find the optimal hyperparameters via a random search with 50 iterations:

```
ctrl = makeTuneControlRandom(maxit = 50)
```

Random search

Next, we need to tell **mlr** to find the optimal hyperparameters via a random search with 50 iterations:

```
ctrl = makeTuneControlRandom(maxit = 50)
```

Let us limit the tuning space in accordance with the literature (Probst, Wright, and Boulesteix, 2018)

```
ps = makeParamSet(
  makeIntegerParam("mtry", lower = 1, upper = ncol(rp) - 1),
  makeNumericParam("sample.fraction", lower = 0.2, upper = 0.9),
  makeIntegerParam("min.node.size", lower = 1, upper = 10)
)
```

Random search

Next, we need to tell **mlr** to find the optimal hyperparameters via a random search with 50 iterations:

```
ctrl = makeTuneControlRandom(maxit = 50)
```

Let us limit the tuning space in accordance with the literature (Probst, Wright, and Boulesteix, 2018)

```
ps = makeParamSet(
  makeIntegerParam("mtry", lower = 1, upper = ncol(rp) - 1),
  makeNumericParam("sample.fraction", lower = 0.2, upper = 0.9),
  makeIntegerParam("min.node.size", lower = 1, upper = 10)
)
```

Recommended literature:

- James, Witten, Hastie, et al. (2013)
- Probst, Wright, and Boulesteix (2018).
- Chapter 14 of *Geocomputation with R* (Lovelace, Nowosad, and Muenchow, 2018)

Wrap it all up

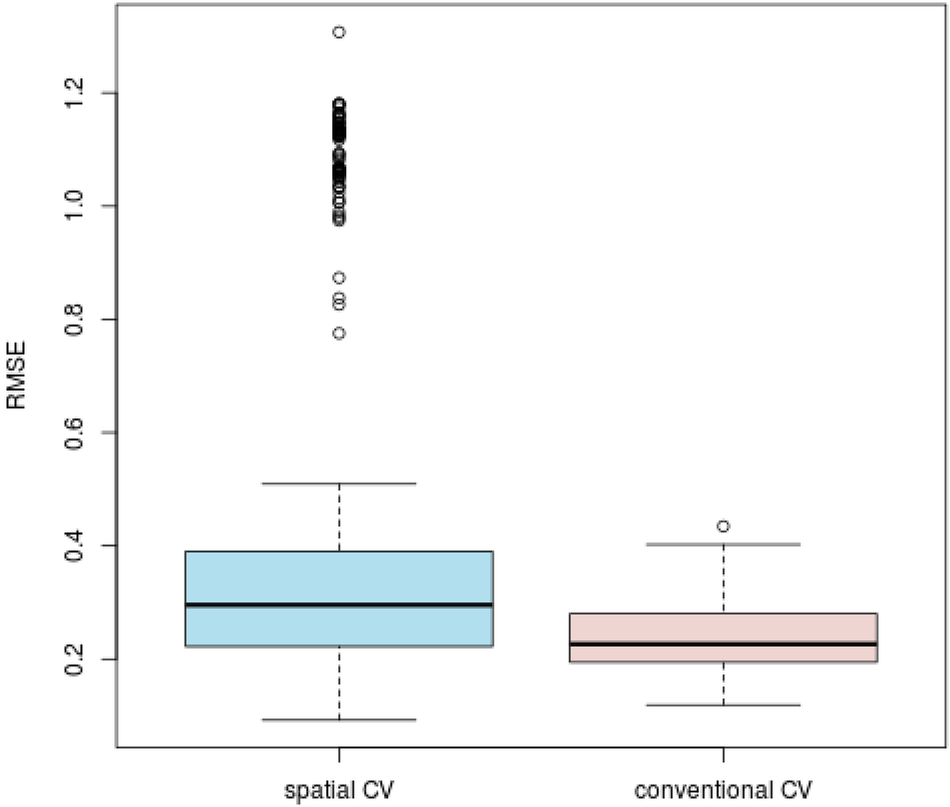
```
wrapped_lrn_rf =
  makeTuneWrapper(learner = lrn_rf,
                  # inner loop (tunning level)
                  resampling = tune_level,
                  # hyperparameter seach space
                  par.set = ps,
                  # random search
                  control = ctrl,
                  show.info = TRUE,
                  # performance measure
                  measures = mlr::rmse)
```

Resampling

Be careful, running the next code chunk takes a while since we are asking R to run `125,500` models. Parallelization might be a good idea. See `code/spatial_cv/01-mlr.R` of the `geocompr/geostats_18` repository how to set it up.

```
set.seed(12345)
cv_sp_rf = mlr::resample(learner = wrapped_lrn_rf,
                         task = task,
                         resampling = perf_level,
                         extract = getTuneResult,
                         measures = mlr::rmse)
```

Have a look at the result



Interesting...

```
cv_sp_lm$aggr
```

```
## rmse.test.rmse
##       0.4280988
```

```
cv_sp_rf$aggr
```

```
## rmse.test.rmse
##       0.4939718
```

Interesting...

```
cv_sp_lm$aggr
```

```
## rmse.test.rmse
##      0.4280988
```

```
cv_sp_rf$aggr
```

```
## rmse.test.rmse
##      0.4939718
```

The linear model is better than the random forest model...

Interesting...

```
cv_sp_lm$aggr
```

```
## rmse.test.rmse
##      0.4280988
```

```
cv_sp_rf$aggr
```

```
## rmse.test.rmse
##      0.4939718
```

The linear model is better than the random forest model...

- Relationship between response and predictors linear enough
- just four predictors, adding further environmental predictors and xy-coordinates might change the result

Predictive mapping

Find the code again in `code/spatial_cv/01-mlr.R` of the `geocompr/geostats_18`.

References

Bischl, Bernd, Michel Lang, Lars Kotthoff, et al. (2016). "Mlr: Machine Learning in R". In: *Journal of Machine Learning Research* 17.170, pp. 1-5. URL: http://jmlr.org/papers/v17/15-066.html.

James, Gareth, Daniela Witten, Trevor Hastie, et al, ed. (2013). *An Introduction to Statistical Learning: With Applications in R*. Springer texts in statistics 103. OCLC: ocn828488009. New York: Springer. 426 pp. ISBN: 978-1-4614-7137-0.

Lovelace, Robin, Jakub Nowosad and Jannes Muenchow (2018). *Geocomputation with R*. The R Series. CRC Press.

Probst, Philipp, Marvin Wright and Anne-Laure Boulesteix (2018). "Hyperparameters and Tuning Strategies for Random Forest".

arXiv: 1804.03515 [cs, stat]. URL: http://arxiv.org/abs/1804.03515 (visited on Aug. 02, 2018).

Schratz, Patrick, Jannes Muenchow, Eugenia Iturritxa, et al. (2018). "Performance Evaluation and Hyperparameter Tuning of Statistical and Machine-Learning Models Using Spatial Data".

arXiv: 1803.11266 [cs, stat]. URL: http://arxiv.org/abs/1803.11266 (visited on Jun. 18, 2018).