Exercise 11 – Tuning

Introduction to Machine Learning

Hint: Useful libraries for this exercise sheet

R

```
# Consider the following libraries for this exercise sheet:

library(mlr3)
library(mlr3learners)
library(mlr3tuning)

# for visualization
library(mlr3viz)
library(ggplot2)
```

Python

```
# Consider the following libraries for this exercise sheet:

# general
import pandas as pd
import numpy as np
import math

# plots
import matplotlib.pyplot as plt

# sklearn
from sklearn.neighbors import KNeighborsClassifier
```

```python
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import validation_curve
```

Exercise 1: Random search

Learning goals

1. Learn to write pseudo-code
2. Implement basic tuning procedure

Random search (RS) is a simple yet effective tuning procedure. We will implement RS from scratch to find the optimal degree $d \in \mathbb{N}$ in a polynomial regression problem.

Consider the following skeleton of pseudo-code:

Algorithm: Random Search

```
***Requires*** ...
< main body >
***Returns*** ...
```

What should this algorithm return as a result?

Solution

The minimum required output will be the optimal degree d^*. (Additional info might enhance user experience in a real-world implementation.)

What should be the required user input? Add the inputs to the pseudo-code.

Hint: Use a single hold-out split in evaluation.

Solution

We need (at least) the following input:

- A search space for d
- The number of RS trials (budget)
- Data to train and evaluate the learner on + the proportion to use as training data (aternatively, we might require separate training and test datasets)
- An evaluation criterion

Let's add that to the pseudo-code:

```
Algorithm: Random Search

Requires search space $\tilde{\Lambda} \subset \mathbb{N}$, budget $B \in \mathbb{N}$, dataset $\mathcal{D} \in (\mathcal{X} \times \mathcal{Y})^n$, train set proportion $s \in (0, 1)$, evaluation criterion $\rho$
< main body >
Returns $d^* \in \mathbb{N}$
```

Start to implement the main body by

- defining elements that allow you to store the currently optimal candidate,
- performing a holdout split on the data, and
- setting up a loop for evaluation of each candidate.

Solution

Tracking optimal candidates

- We'll need a variable that is set to the optimal degree in any given iteration, to be updated when a candidate performs better than previous ones.
- Likewise, we'll need to store the estimated generalization error associated with the optimal candidate so we can compare it to new candidates (we'll assume that smaller ρ means lower GE)

Holdout split

Simple one-liner.

Loop

- The first thing to do is defining the set of candidate values, which we'll obtain by drawing as many random samples from the search space as our budget allows.
- Afterwards, we can run a `for` loop over this candidate set.

There's no official pseudo-code language–you should phrase your code such that a human with knowledge in any suitable programming language can unambiguously translate the pseudo-code into that programming language.

```
Algorithm: Random Search

Requires search space $\tilde{\Lambda} \subset \mathbb{N}$, budget $B \in \mathbb{N}$, dataset $\mathcal{D} \in (\mathcal{X} \times \mathcal{Y})^n$, train set proportion $s \in (0,1)$, evaluation criterion $\rho$
set_seed // crucial to make the procedure reproducible
$d^* \leftarrow 0$ // any starting value okay
$\mathrm{GE}^* \leftarrow \infty$ // ensures that first candidate is optimal until a better one comes along
candidates $\leftarrow$ sample_uniform(from $= \tilde{\Lambda}$, number $= B$, replacement $=$ False)
$\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}} \leftarrow$ split_holdout(data $= \mathcal{D}$, train_proportion $= s$)
for $d$ in candidates do
  ...
end for
Returns $d^* \in \mathbb{N}$
```

Finalize the pseudo-code by adding steps for training & evaluation and a rule to update the optimal candidate.

Solution

```
Algorithm: Random Search

Requires search space $\tilde{\Lambda} \subset \mathbb{N}$, budget $B \in \mathbb{N}$, dataset $\mathcal{D} \in (\mathcal{X} \times \mathcal{Y})^n$, train set proportion $s \in (0,1)$, evaluation criterion $\rho$
set_seed // crucial to make the procedure reproducible
$d^* \leftarrow 0$ // any starting value okay
$\mathrm{GE}^* \leftarrow \infty$ // ensures that first candidate is optimal until a better one comes along
candidates $\leftarrow$ sample_uniform(from $= \tilde{\Lambda}$, number $= B$)
$\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}} \leftarrow$ split_holdout(data $= \mathcal{D}$, train_proportion $= s$, replacement $=$ False)
for $d$ in candidates do
    learner $\leftarrow$ polynomial_regression(degree $= d$)
    train(learner = learner, data $= \mathcal{D}_{\text{train}}$)
    GE $\leftarrow$ evaluate(learner = learner, data $= \mathcal{D}_{\text{test}}$, criterion $= \rho$)
    if $\mathrm{GE} < \mathrm{GE}^*$ do
```

```
        $d^* \leftarrow d$
        $\mathrm{GE}^* \leftarrow \mathrm{GE}$
      end **if**
  end **for**
  ***Returns*** $d^* \in \mathbb{N}$
```

Describe how you could implement a more flexible resampling strategy.

Solution

A more general way to define the user input might be to accept sets of training and test datasets, respectively. We could then add a `for` loop over those to compute the GE associated with each candidate d.

Exercise 2: Basic tuning techniques

Learning goals

Understand difference between grid and random search from different perspectives

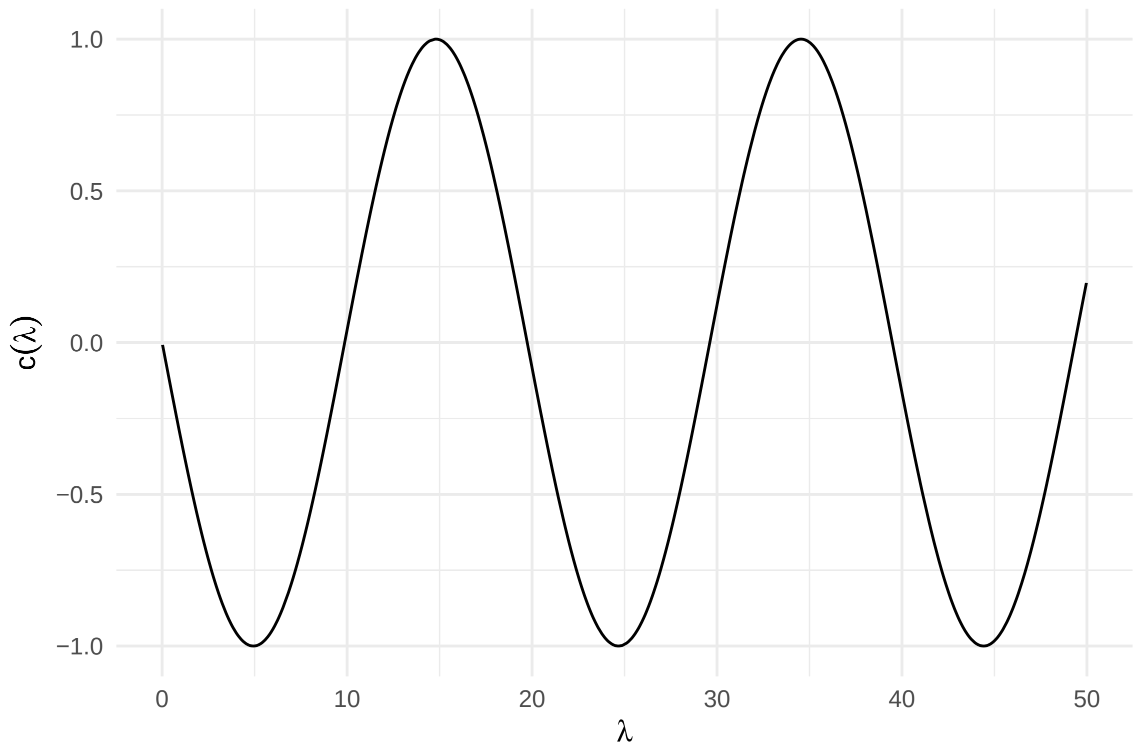
Explain the difference in implementation between random search (RS) and grid search (GS).

Solution

- The main difference lies in how we obtain the candidate values: RS samples them uniformly from the search space, while GS creates a multi-dimensional grid from the search space and tries all configurations in it.
- In the algorithm from Exercise 1, we'd thus need to adjust the `candidates` $\leftarrow$ ... part (and slightly adapt the user input).

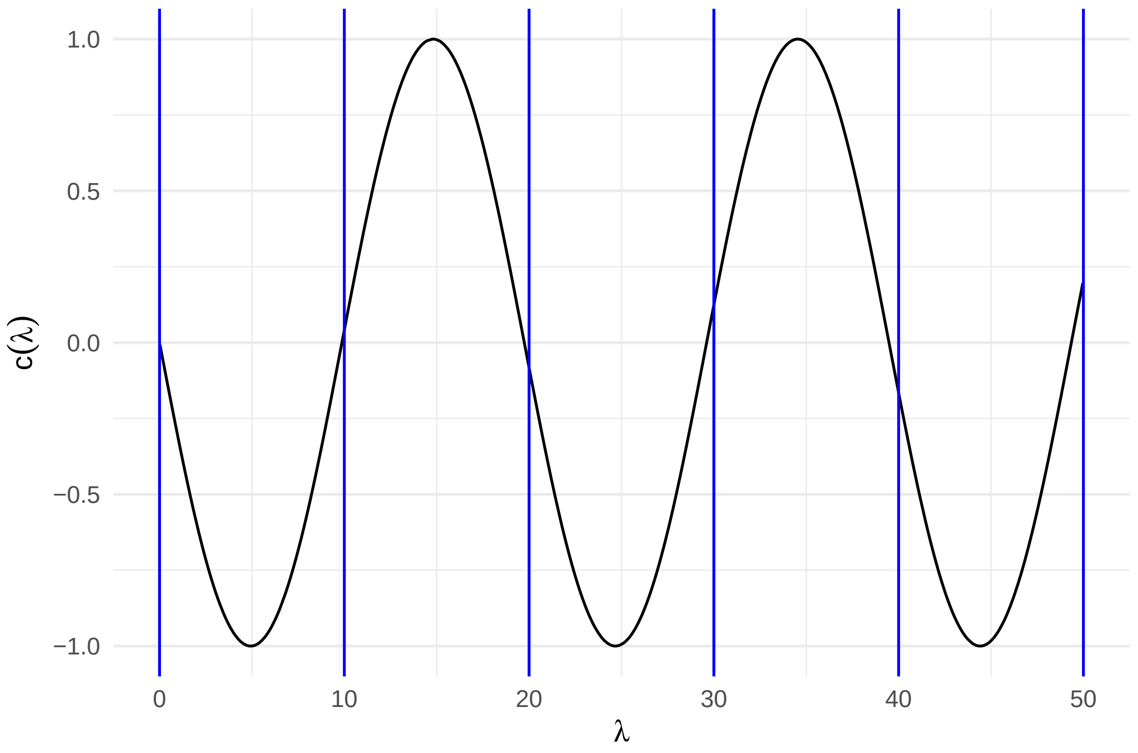
Consider the following objective function $c(\lambda)$ with a single hyperparameter $\lambda \in \mathbb{R}$. The objective is to be minimized. Explain whether RS or GS will be more suited to find the optimal value λ^*, given

- a search space of $\lambda \in [0, 50]$, and
- a budget of 6 evaluations.



Solution

- In this (stylized) example, the discretization of GS is quite harmful: Choosing the grid as the lower and upper bounds of the search space plus 4 equidistant values within, as is common, will prevent us from ever exploring promising values for λ.

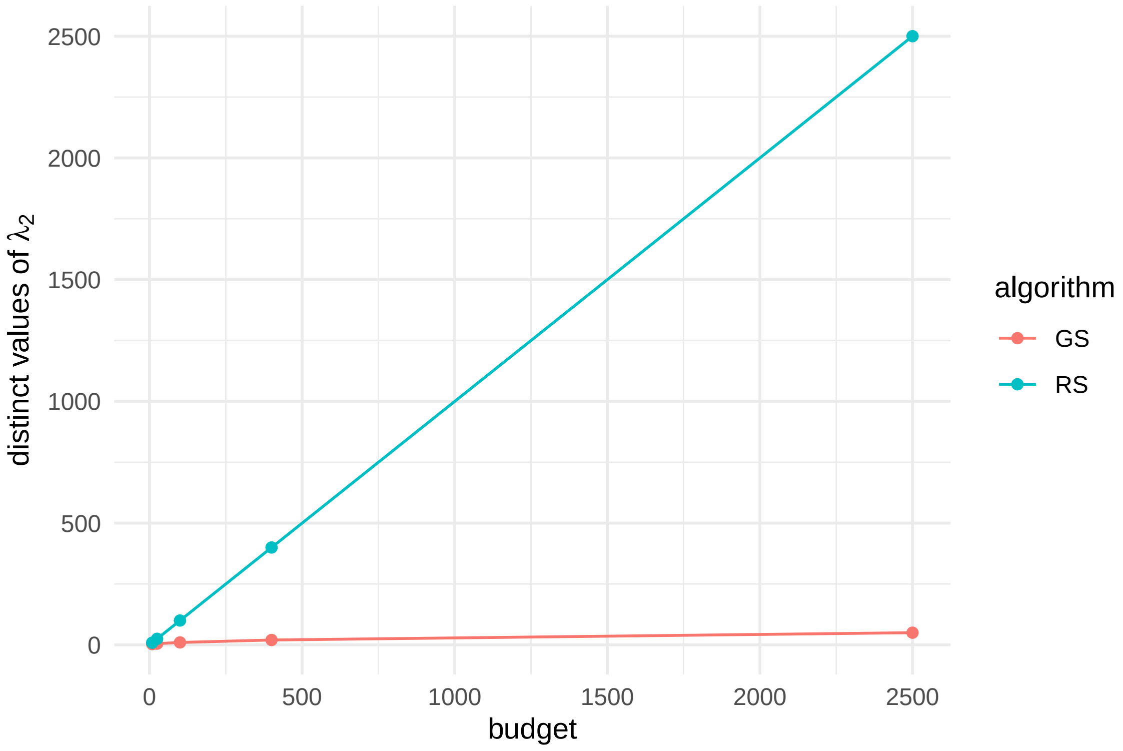


- With RS, every value in $[0, 50]$ is equally likely to be tried, so we have at least a chance to find one of the good values $\rightsquigarrow$ RS is preferable here.

Consider now a bivariate objective function $c(\lambda_1, \lambda_2)$ with $\lambda_1, \lambda_2 \in \mathbb{R}$. The objective is to be minimized. Suppose that λ_2 is vastly more important for the objective than λ_1.

Visualize the number of different values of λ_2 that RS and (exhaustive) GS are expected to explore for a budget B of 9, 25, 100, 400, 2500 evaluations.

Solution



- Since GS comes with discretization, distributing the budget across both hyperparameter means that λ_1 receives $\sqrt{B}$ candidate values even though it doesn't matter for the objective. Consequently, we only have $\sqrt{B}$ values for λ_2 left to explore.
- RS, on the other hand, will (in expectation) evaluate B different values of λ_2 for every budget, making it much more efficient in this situation.

Exercise 3: Interpreting tuning results

Learning goals

1. Implement tuning procedure
2. Interpret effect of hyperparameters

In this exercise we will perform hyperparameter optimization (HPO) for the task of classifying the `credit risk` data with a k-NN classifier. We use `mlr3` (tuning chapter in the book); see Jupyter notebook for a similar case study in Python.

The `kknn` implementation used by `mlr3` contains several hyperparameters, three of which are to be tuned for our prediction:

- k (number of neighbors)

- `kernel`
- `scale`

Details about the hyperparameters

- k: determines the size of the neighborhood and thus influences the locality of the model. Smaller neighborhoods reflect the belief that only very similar (close) neighbors should be allowed to weigh into the prediction of a new observation, and predictions may change strongly for only small changes of the input variables. If k is chosen too small, we may encounter overfitting. Conversely, larger neighborhoods produce a more global model with larger parts of the input space receiving the same prediction. Setting k too large may result in underfitting.

- `kernel`: determines the importance weights in the k-neighborhood.

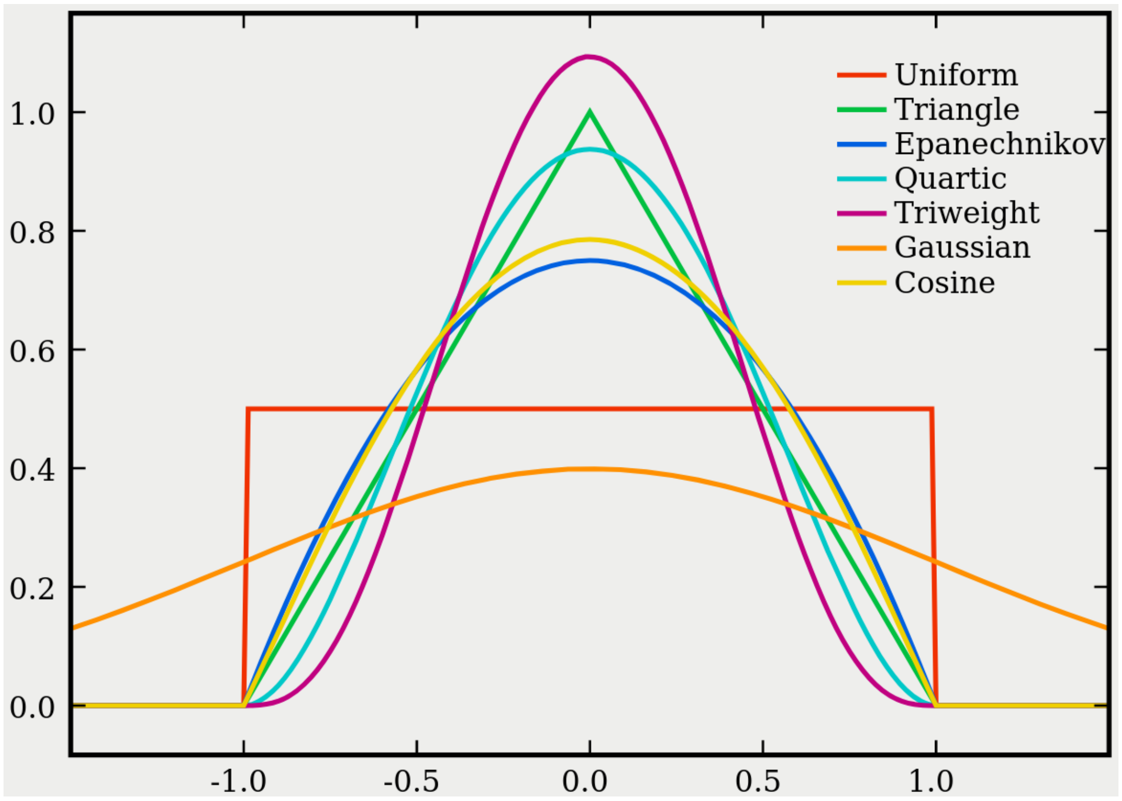
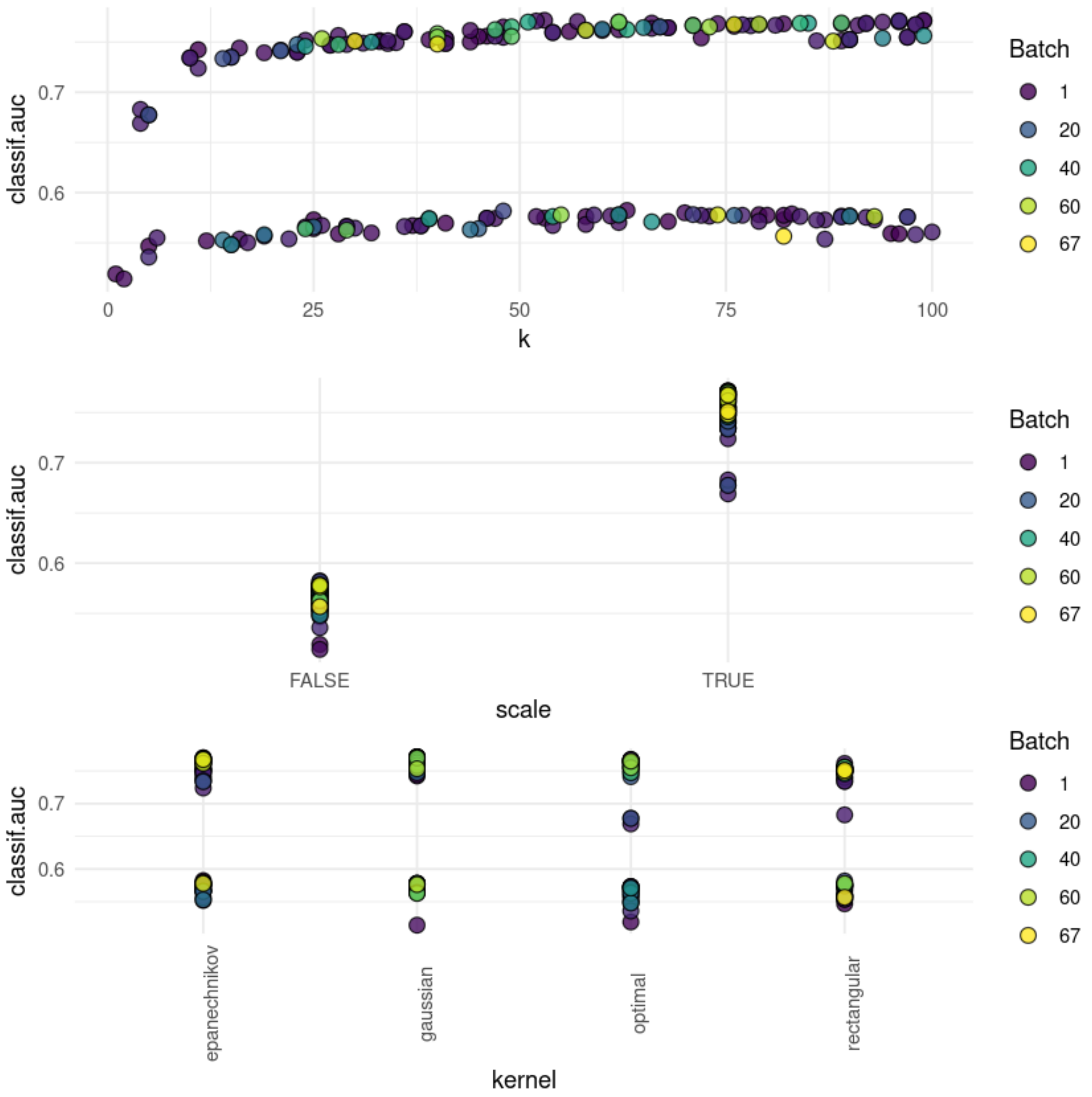
  

  figure source

  Can you guess which kernel corresponds to unweighted k-NN?

- `scale` (logical): defines whether variables should be normalized to equal standard deviation. This is often reasonable to avoid implicit importance weighting through different natural scales (for example, recall that neighborhoods in a bivariate feature space are circular for quadratic distance – scaling either dimension will change which observations end up in the neighborhood).

You receive the following `mlr3` output from tuning k-NN with random search. Interpret the impact of each hyperparemeter on the objective.



Solution

We plot the classification performance in terms of AUC across different choices of each hyperparameter. Let's look at each one in turn:

- Increasing k initially leads to an improvement that plateaus after around 50 neighbors. However, there seem to be two quite distinct groups of candidate values.
- Scaling the variables boosts performance quite substantially.
- The choice of the kernel does not seem to have much impact. Again, we see two clusters of candidate values.

Obviously, the interpretability of these plots is limited: we only see *marginal* effects of individual hyperparameters. The fact that they really interact with each other contributes substantially to the difficulty of the tuning problem. We can clearly see this in the plot for k and `kernel`, where we have two quite distinct patterns corresponding to different values of `scale`.

Now let's look at the code that generated the above results. Start by defining the `german_credit` task, where you reserve 800 observations for training, and the `kknn` learner (the learner should output probabilities).

Solution

```
# define task and learner
task <- tsk("german_credit")
set.seed(123)
train_rows <- sample(seq_len(task$nrow), 800, replace = FALSE)
test_rows <- setdiff(seq_len(task$nrow), train_rows)
task_train <- task$clone()$filter(train_rows)
task_test <- task$clone()$filter(test_rows)
lrn_knn <- lrn("classif.kknn", predict_type = "prob")
```

Set up the search space to tune over using the `ps` function. Include choices for $k \in \{1, 2, \dots, 100\}$, `scale` $\in \{\text{yes}, \text{no}\}$, and `kernel` $\in \{\text{rectangular}, \text{epanechnikov}, \text{gaussian}, \text{optimal}\}$.

Solution

```
# set up search space
search_space <- ps(
    k = p_int(1, 100),
    scale = p_lgl(),
    kernel = p_fct(c("rectangular", "epanechnikov", "gaussian", "optimal"))
)
```

Define the stopping criterion for random search with a so-called *terminator* (`trm`). We want the tuning procedure to finish after 200 evaluations or a maximum runtime of 30 seconds.

Hint: You can define this combinded terminator via a list of individual terminators.

Solution

```
# create combined terminator object (either criterion, whichever is met first,
# can invoke the termination)
terminator_evals <- trm("evals", n_evals = 200)
terminator_runtime <- trm("run_time", secs = 30)
terminator <- trm(
    "combo",
    list(terminator_evals, terminator_runtime),
    any = TRUE
)
```

Set up a tuning instance using the function `ti`. This object combines all of the above components. Set the evaluation criterion to AUC.

Solution

```
# create tuning instance
instance <- ti(
    task = task_train,
    learner = lrn_knn,
    resampling = rsmp("cv", folds = 5),
    terminator = terminator,
    search_space = search_space,
    measure = msr("classif.auc")
)
```

Finally, define the tuner (`tnr`) of type "random_search" and run the optimization. Don't forget to make your code reproducible.

Solution

Optimization

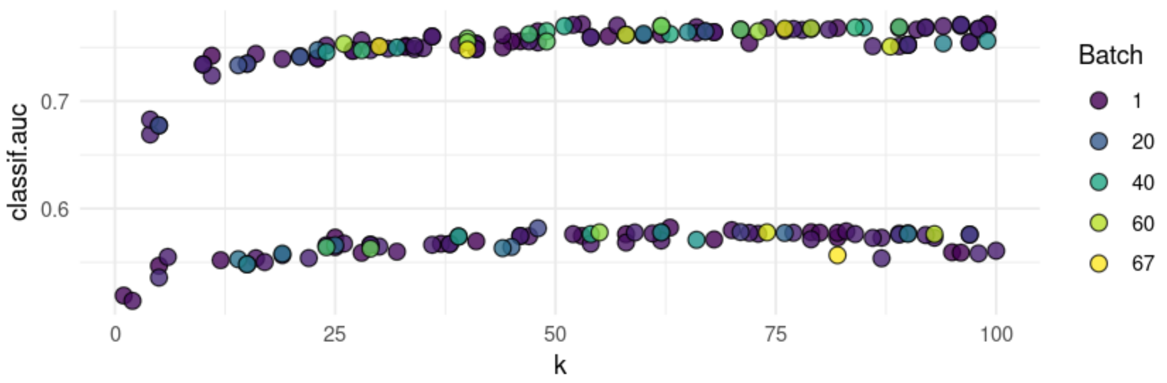
```r
lgr::get_logger("mlr3")$set_threshold("warn")
lgr::get_logger("bbotk")$set_threshold("warn")
optimizer <- tnr("random_search")
set.seed(123)
optimizer$optimize(instance)
```

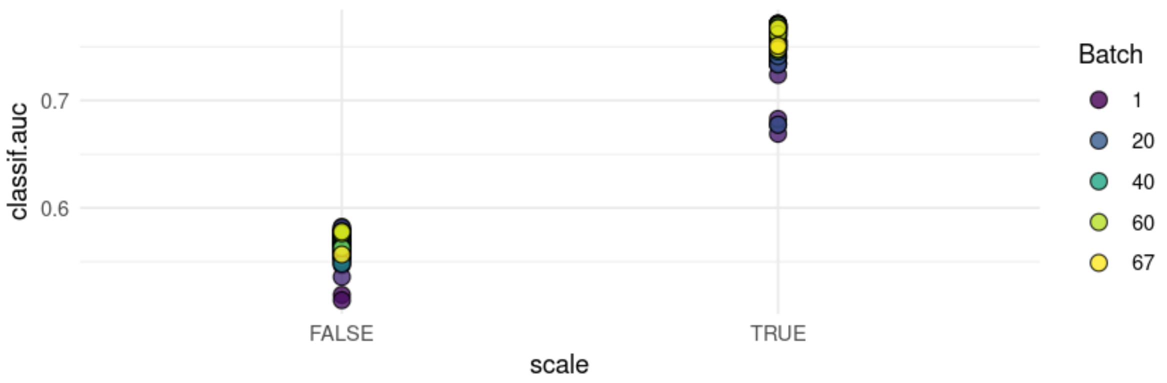
A data.table: 1 × 6

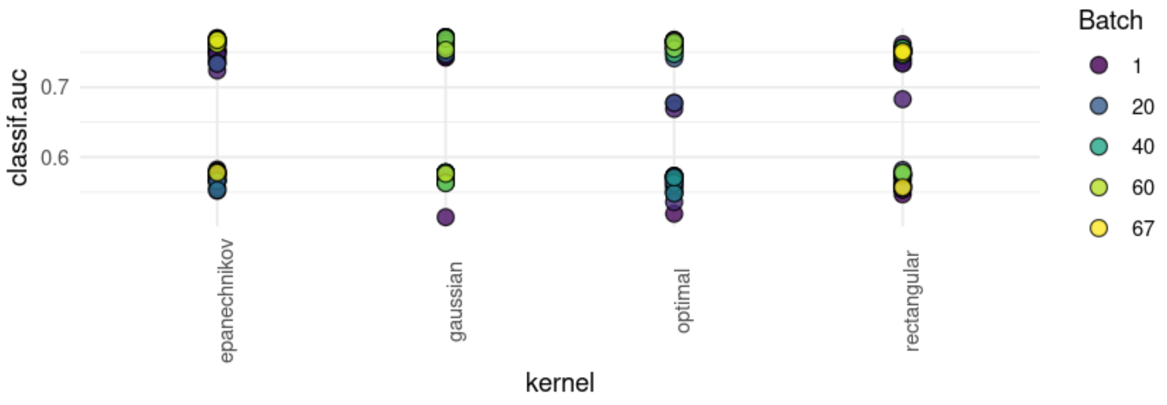
k <int>	scale <lgl>	kernel <chr>	learner_param_vals <list>	x_domain <list>	classif.auc <dbl>
53	TRUE	gaussian	53 , TRUE , gaussian	53 , TRUE , gaussian	0.7715534

Visualization

```r
p <- autoplot(instance)
p[[3]] <- p[[3]] +
    theme(axis.text.x = element_text(angle = 90))
do.call(grid.arrange, c(p, list(ncol = 1)))
```







With the hyperparameter configuration found via HPO, fit the model on all training observations and compute the AUC on your test data.

Solution

```
optimal_config <- instance$result_learner_param_vals
lrn_knn$param_set$values <- optimal_config
lrn_knn$train(task_train)
prediction <- lrn_knn$predict(task_test)
prediction$score(msr("classif.auc"))
```

classif.auc: 0.781318681318681