Exercise 11 – Tuning

Introduction to Machine Learning

Hint: Useful libraries for this exercise sheet

R

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

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

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

Python

```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
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?

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

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

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.

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

Describe how you could implement a more flexible resampling strategy.

Exercise 2: Basic tuning techniques

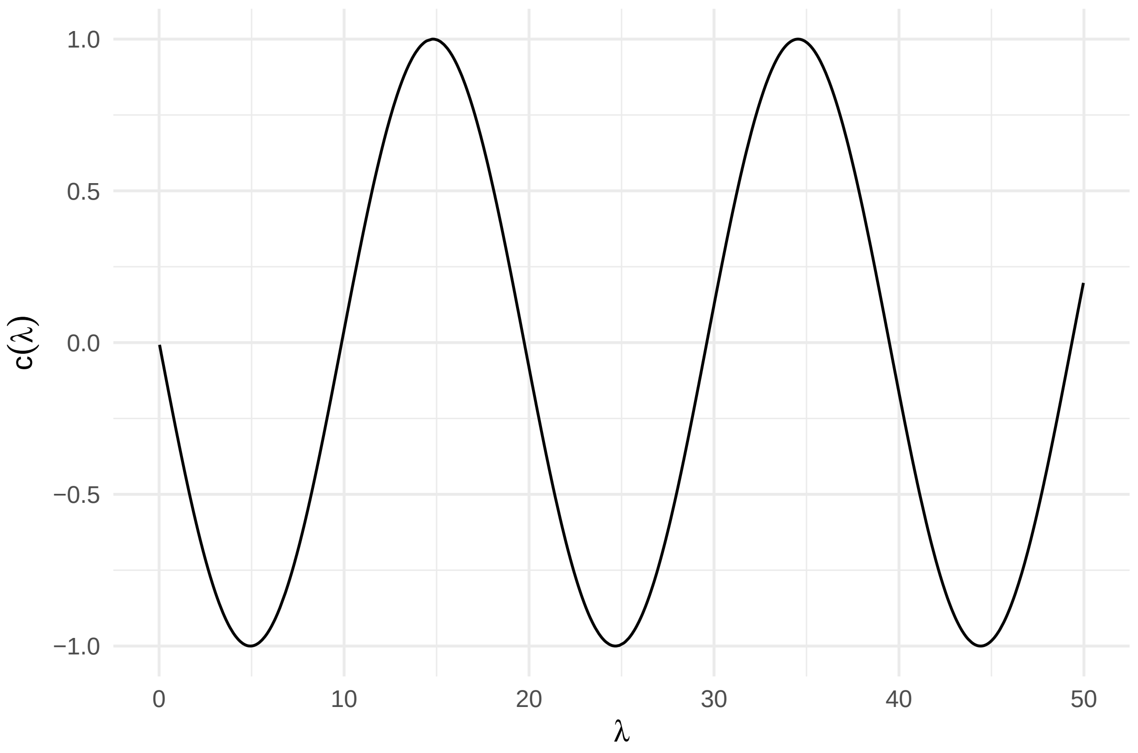
Learning goals

Understand difference between grid and random search from different perspectives

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

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.



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.

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.

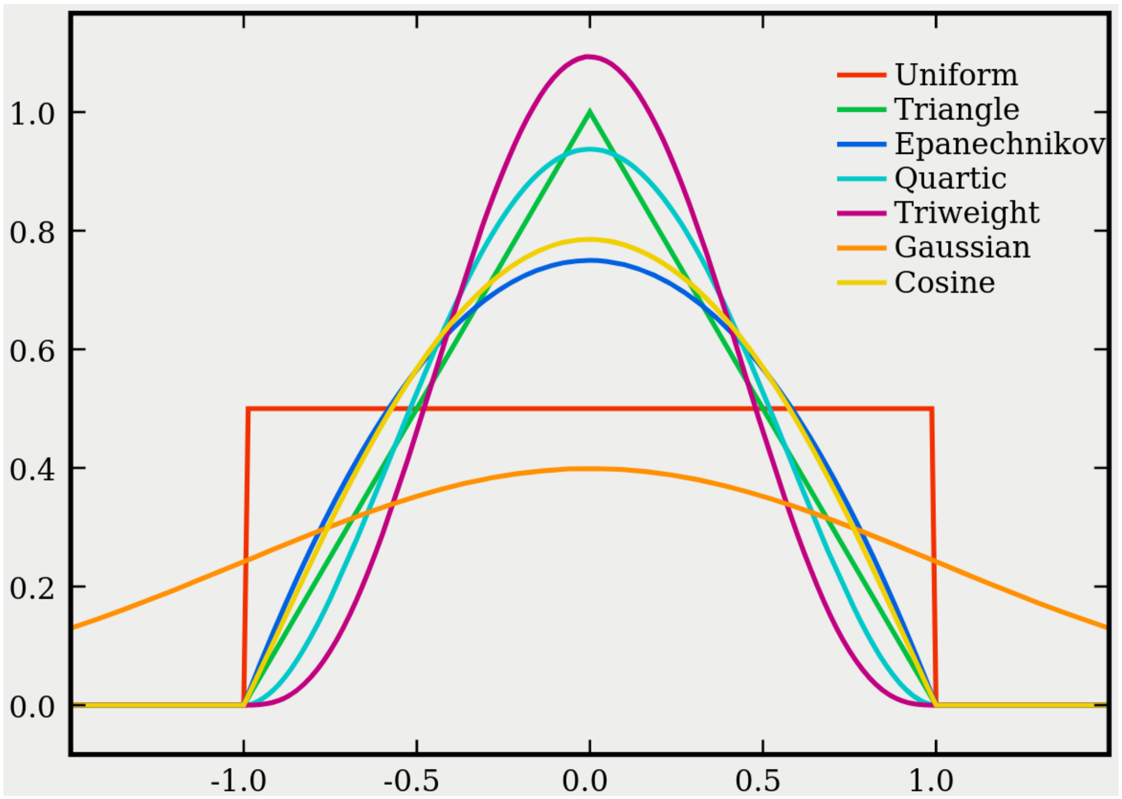
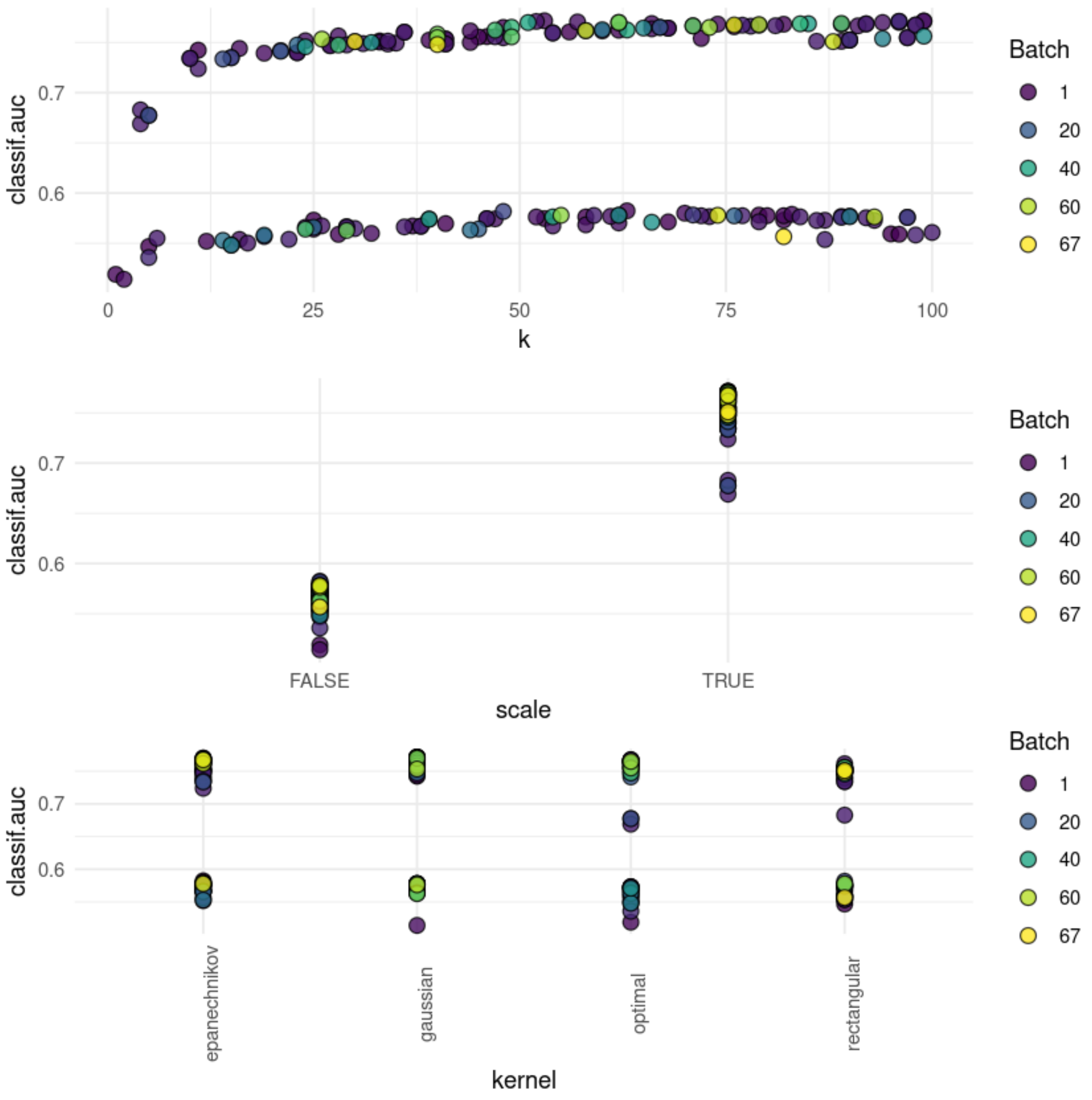


figure source

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.



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).

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` $\in \{\text{rectangular}, \text{epanechnikov}, \text{gaussian}, \text{optimal}\}$.

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.

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

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

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