

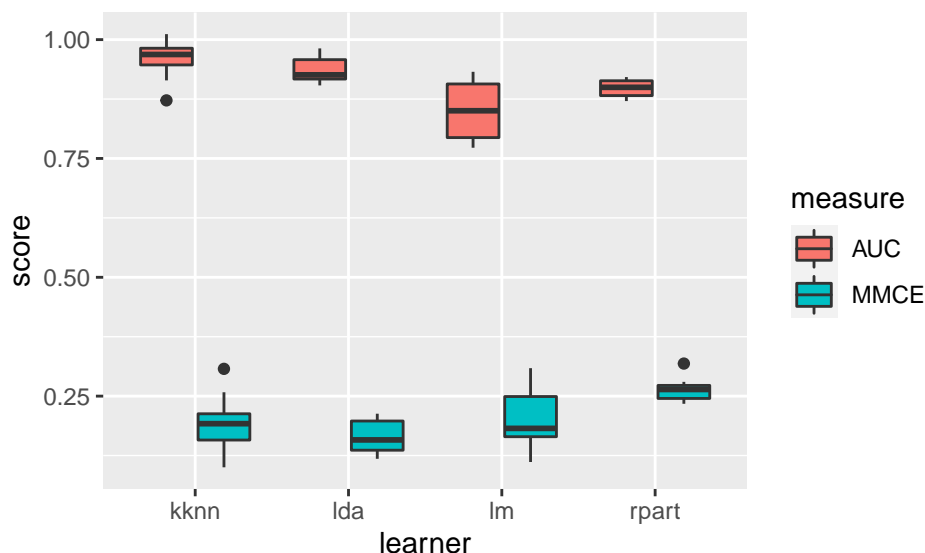
## Exercise 1: Tuning Principles

a) Suppose that we want to compare four different learners:

| Learner                                 | Tuning required |
|---|-----------------|
| Logistic regression ( <code>lm</code> ) | no              |
| CART ( <code>rpart</code> )             | yes             |
| $k$ -NN ( <code>kknn</code> )           | yes             |
| LDA ( <code>lda</code> )                | no              |

For performance evaluation and subsequent comparison, we use 10-CV as outer resampling strategy. Within the inner tuning loop, applicable to CART and  $k$ -NN, we use 5-CV in combination with random search, drawing 200 hyperparameter configurations for each model. Our measure of interest is the AUC.

- How many models need to be fitted in total to conduct the final benchmark?
- Given the following benchmark result, which learner performs best? Explain your decision.



- Recap briefly what is meant by the *bias-variance trade-off* in resampling.
- Are the following statements true or not? Explain your answer in one sentence.
  - The bias of the generalization error estimate for 3-CV is higher than for 10-CV.
  - Every outer loss can also be used as inner loss, assuming standard gradient-based optimization.

## Exercise 2: AutoML with `mlr3`

In this exercise, we build a simple automated machine learning (AutoML) system that will make data-driven choices on which learner to use and also conduct the necessary tuning.

`mlr3pipelines` make this endeavor easy, modular and guarded against many common modeling errors.

We work on the `pima` data to classify patients as diabetic and design a system that is able to choose between  $k$ -NN and a random forest, both with tuned hyperparameters.

To this end, we will use a graph learner, a "single unit of data operation" that can be trained, resampled, evaluated, ... as a whole – in other words, treated as any other learner.

- a) Create a task object in `mlr3` (the problem is pre-specified under the ID "pima").
- b) Specify the above learners, where you need to give each learner a name as input to the `id` argument. Convert each learner to a pipe operator by wrapping them in the sugar function `po()`, and store them in a `list`.
- c) Before starting the actual learning pipeline, take care of pre-processing. While this step is highly customizable, you can use an existing sequence to impute missing values, encode categorical features, and remove variables with constant value across all observations. For this, specify a pipeline (`ppl()`) of type "robustify" (setting `factors.to.numeric` to `TRUE`).
- d) Create another `ppl`, of type "branch" this time, to enable selection between your learners.
- e) Chain both pipelines using the double pipe and plot the resulting graph. Next, convert it into a graph learner with `as_learner()`.
- f) Now you have a learner object just like any other. Take a look at its tunable hyperparameters. You will optimize the learner selection, the number of neighbors in  $k$ -NN (between 3 and 10), and the number of split candidates to try in the random forest (between 1 and 5). Define the search range for each like so:

```
<learner>$param_set$values$<hyperparameter> <- to_tune(p_int(lower, upper))
```

`p_int` marks an integer hyperparameter with lower and upper bounds as defined; similar objects exist for other data types. With `to_tune()`, you signal that the hyperparameter shall be optimized in the given range.

**Hint:** You need to define dependencies, since the tuning process is defined by which learner is selected in the first place (no need to tune  $k$  in a random forest).

- g) Conveniently, there is a sugar function, `tune_nested()`, that takes care of nested resampling in one step. Use it to evaluate your tuned graph learner with
  - mean classification error as inner loss,
  - random search as tuning algorithm (allowing for 3 evaluations), and
  - 3-CV in both inner and outer loop.
- h) Lastly, extract performance estimates per outer fold (`score()`) and overall (`aggregate()`). If you want to risk a look under the hood, try `extract_inner_tuning_archives()`.

Congrats, you just designed a turn-key AutoML system that does (nearly) all the work with a few lines of code!

### Exercise 3: Kaggle Challenge

Make yourself familiar with the Titanic Kaggle challenge (<https://www.kaggle.com/c/titanic>).

Based on everything you have learned in this course, do your best to achieve a good performance in the survival challenge.

- Try out different classifiers you have encountered during the course (or maybe even something new?)
- Improve the prediction by creating new features (feature engineering).
- Tune your parameters (see: <https://mlr3book.ml-r.org.com/tuning.html>).
- How do you fare compared to the public leaderboard?

**Hint:** Use the `titanic` package to directly access the data. Use `titanic::titanic_train` for training and `titanic::titanic_test` for your final prediction.