

Solution 1: Tuning Principles

a) Benchmark result:

(i) Total number of models trained:

$$\underbrace{4 \cdot 10}_{\text{outer resampling}} + 2 \cdot \underbrace{10 \cdot \underbrace{5 \cdot 200}_{\text{one tuning iteration}}}_{\substack{\text{all outer folds in one tuning procedure} \\ \text{all tuning procedures}}} = 20,040.$$

(ii) Since we evaluate on AUC, we select k -NN with the best average result in that respect.

b) Less data for training leads to higher bias, less data for evaluation leads to higher variance.

c) Statements:

- i) True – 3-CV leads to smaller train sets, therefore we are not able to learn as well as in, e.g., 10-CV.
- ii) False – we are relatively flexible in choosing the outer loss, but the inner loss needs to be suitable for empirical risk minimization, which encompasses differentiability in most cases (i.e., whenever optimization employs derivatives).

Solution 2: AutoML with mlr3

This exercise is a compact version of a tutorial on mlr3gallery. Feel free to explore the additional steps and explanations featured in the original (there is also a bunch of other useful code demos).

```
a) library(mlr3verse)

## Loading required package: mlr3

library(mlr3tuning)

## Loading required package: paradox

(task <- tsk("pima"))

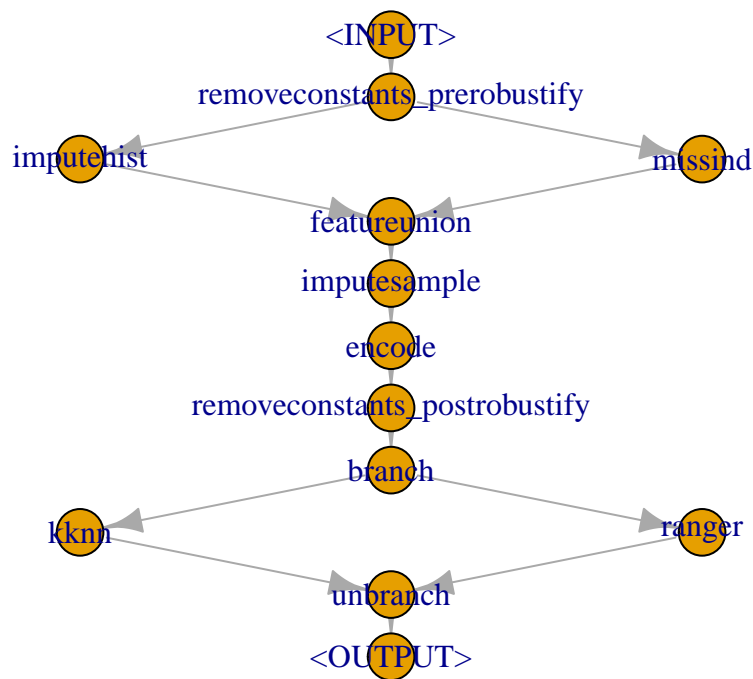
## <TaskClassif:pima> (768 x 9): Pima Indian Diabetes
## * Target: diabetes
## * Properties: twoclass
## * Features (8):
##   - dbl (8): age, glucose, insulin, mass, pedigree, pregnant, pressure,
##     triceps
```

```
b) learners <- list(
  po(lrn("classif.kknn", id = "kknn")),
  po(lrn("classif.ranger", id = "ranger")))
```

```
c) ppl_preproc <- ppl("robustify", task = task, factors_to_numeric = TRUE)
```

```
d) ppl_learners <- ppl("branch", learners)
```

```
e) ppl_combined <- ppl_preproc %>>% ppl_learners  
plot(ppl_combined)
```



```
graph_learner <- as_learner(ppl_combined)
```

```
f) # check available hyperparameters for tuning (converting to data.table for
# better readability)
tail(as.data.table(graph_learner$param_set), 10)
```

##		id	class	lower	upper	levels
## 1:	ranger.sample.fraction	ParamDbl		0	1	
## 2:	ranger.save.memory	ParamLgl		NA	NA	TRUE,FALSE
## 3:	ranger.scale.permutation.importance	ParamLgl		NA	NA	TRUE,FALSE
## 4:	ranger.se.method	ParamFct		NA	NA	jack,infjack
## 5:	ranger.seed	ParamInt		-Inf	Inf	
## 6:	ranger.split.select.weights	ParamUty		NA	NA	
## 7:	ranger.splitrule	ParamFct		NA	NA	gini,extratrees
## 8:	ranger.verbose	ParamLgl		NA	NA	TRUE,FALSE
## 9:	ranger.write.forest	ParamLgl		NA	NA	TRUE,FALSE
## 10:	branch.selection	ParamInt		1	2	

##	nlevels	is_bounded	special_vals	default	storage_type
## 1:	Inf	TRUE	<list[0]>	<NoDefault[3]>	numeric
## 2:	2	TRUE	<list[0]>	FALSE	logical
## 3:	2	TRUE	<list[0]>	FALSE	logical
## 4:	2	TRUE	<list[0]>	infjack	character
## 5:	Inf	FALSE	<list[1]>		integer
## 6:	Inf	FALSE	<list[0]>		list
## 7:	2	TRUE	<list[0]>	gini	character
## 8:	2	TRUE	<list[0]>	TRUE	logical
## 9:	2	TRUE	<list[0]>	TRUE	logical
## 10:	2	TRUE	<list[0]>	<NoDefault[3]>	integer


```
##
## tags
## 1: train
## 2: train
## 3: train
## 4: predict
## 5: train,predict
## 6: train
## 7: train
## 8: train,predict
## 9: train
## 10: train,predict,required

# seeing all our hyperparameters of interest are of type int, we specify the
# tuning objects accordingly, and dependencies for k and mtry
graph_learner$param_set$values$branch.selection <-
  to_tune(p_int(1, 2))
graph_learner$param_set$values$kknn.k <-
  to_tune(p_int(3, 10, depends = branch.selection == 1))
graph_learner$param_set$values$ranger.mtry <-
  to_tune(p_int(1, 5, depends = branch.selection == 2))

# rename learner (otherwise, mlr3 will display a lengthy chain of operations
# in result tables)
graph_learner$id <- "graph_learner"
```

```
g) # make sure to set a seed for reproducible results
set.seed(123)
```

```
# perform nested resampling, terminating after 3 evaluations
rr <- tune_nested(
  method = "random_search",
  task = task,
  learner = graph_learner,
  inner_resampling = rsmp("cv", folds = 3),
  outer_resampling = rsmp("cv", folds = 3),
  measure = msr("classif.ce"),
  term_evals = 3)

```

h) `rr$score()`

```
##           task task_id          learner      learner_id
## 1: <TaskClassif[50]>   pima <AutoTuner[42]> graph_learner.tuned
## 2: <TaskClassif[50]>   pima <AutoTuner[42]> graph_learner.tuned
## 3: <TaskClassif[50]>   pima <AutoTuner[42]> graph_learner.tuned
##           resampling resampling_id iteration      prediction
## 1: <ResamplingCV[20]>           cv           1 <PredictionClassif[20]>
## 2: <ResamplingCV[20]>           cv           2 <PredictionClassif[20]>
## 3: <ResamplingCV[20]>           cv           3 <PredictionClassif[20]>
##      classif.ce
## 1:  0.2460938
## 2:  0.2421875
## 3:  0.2109375

rr$aggregate()

## classif.ce
##  0.2330729

```

The performance estimate for our tuned learner then amounts to an MCE of around 0.23.

Solution 3: Kaggle Challenge

We do not provide an explicit solution here, but have a look at the [tuning code demo](#), which covers some parts, and take inspiration from the public contributions on Kaggle.