Solution 1: Tuning Principles

a) Benchmark result:

(i) Total number of models trained:

- Let's first consider the 2 learners for which we conduct no further tuning. We train each of them 10 times within the outer resampling (10-CV) loop (on 90% of the data, respectively.) This gives us the desired performance estimate.
- For the 2 learners with additional hyperparameter tuning, the picture is a little more involved: We also train each of them 10 times for the outer loop (on 90% of the data), where we endow them with the optimal hyperparameter configuration, to get the performance estimate. In order to get this optimal configuration, though, we first need to run the inner tuning loop, where we determine the (estimated) performance of each of the 200 candidate configurations via 5-CV, meaning we train with each configuration 5 times on $80\% \cdot 90\% = 72\%$ of the data.
- So, in total:

$$\underbrace{2 \cdot 10}_{\text{log reg \& LDA}} + \underbrace{2 \cdot 10}_{k\text{-NN \& CART}} \cdot (\overbrace{5 \cdot 200}^{\text{find optimal } \boldsymbol{\lambda}} \overbrace{+1}^{\text{train with optimal } \boldsymbol{\lambda}}) = 20,040.$$

- We see: Without the inner loop for 2 learners, it would just be $2 \cdot 10 + 2 \cdot 10 \cdot 1 = 40$. The inner tuning adds $5 \cdot 200$ model fits for each learner (2) and outer fold (10).
- Alternatively, we can thus look at this from an inner-to-outer perspective:

$$\underbrace{2 \cdot 10 \cdot 5 \cdot 200}_{\text{tuning for } k\text{-NN \& CART in each fold (1)}} + \underbrace{4 \cdot 10}_{\text{outer loop (2) with fold-specific } \boldsymbol{\lambda} \text{ from (1)}}$$

(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)
```

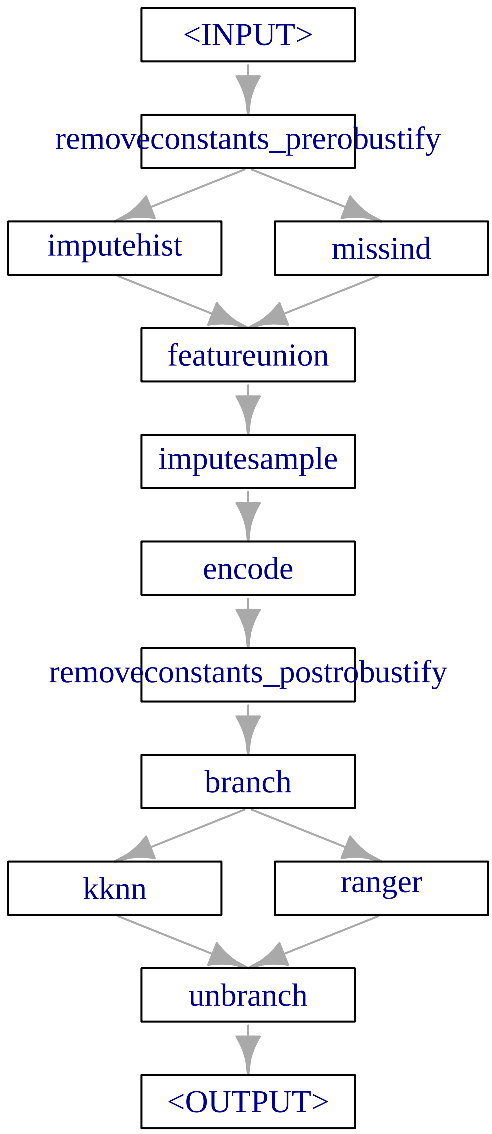
```
## Loading required package:  paradox

(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)
```




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

f)
```r
# 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
##   1:               ranger.sample.fraction ParamDbl     0     1
```

```
##  2:                  ranger.save.memory ParamLgl    NA    NA
##  3: ranger.scale.permutation.importance ParamLgl    NA    NA
##  4:                    ranger.se.method ParamFct    NA    NA
##  5:                         ranger.seed ParamInt  -Inf   Inf
##  6:          ranger.split.select.weights ParamUty    NA    NA
##  7:                    ranger.splitrule ParamFct    NA    NA
##  8:                      ranger.verbose ParamLgl    NA    NA
##  9:                 ranger.write.forest ParamLgl    NA    NA
## 10:                    branch.selection ParamInt     1     2
##                    levels nlevels is_bounded special_vals        default
##  1:                           Inf       TRUE    <list[0]> <NoDefault[3]>
##  2:                TRUE,FALSE       2       TRUE    <list[0]>          FALSE
##  3:                TRUE,FALSE       2       TRUE    <list[0]>          FALSE
##  4:              jack,infjack       2       TRUE    <list[0]>        infjack
##  5:                           Inf      FALSE    <list[1]>
##  6:                           Inf      FALSE    <list[0]>
##  7: gini,extratrees,hellinger       3       TRUE    <list[0]>           gini
##  8:                TRUE,FALSE       2       TRUE    <list[0]>           TRUE
##  9:                TRUE,FALSE       2       TRUE    <list[0]>           TRUE
## 10:                             2       TRUE    <list[0]> <NoDefault[3]>
##     storage_type                    tags
##  1:      numeric                   train
##  2:      logical                   train
##  3:      logical                   train
##  4:    character                 predict
##  5:      integer           train,predict
##  6:         list                   train
##  7:    character                   train
##  8:      logical           train,predict
##  9:      logical                   train
## 10:      integer 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
tuner <- tnr("random_search")

rr <- tune_nested(
  tuner = tuner,
  task = task,
  learner = graph_learner,
  inner_resampling = rsmp("cv", folds = 3),
  outer_resampling = rsmp("cv", folds = 3),
```

```r
  measure = msr("classif.ce"),
  term_evals = 3)
```

h)
```r
rr$score()
```

```
##    task_id          learner_id resampling_id iteration classif.ce
## 1:    pima graph_learner.tuned            cv         1  0.2695312
## 2:    pima graph_learner.tuned            cv         2  0.2617188
## 3:    pima graph_learner.tuned            cv         3  0.1953125
## Hidden columns: task, learner, resampling, prediction
```

```r
rr$aggregate()
```

```
## classif.ce
##  0.2421875
```

The performance estimate for our tuned learner then amounts to an MCE of around 0.24.

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.