Solution 1:

a) Benchmark result:

   (i) Total number of models trained:

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

   (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:

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).

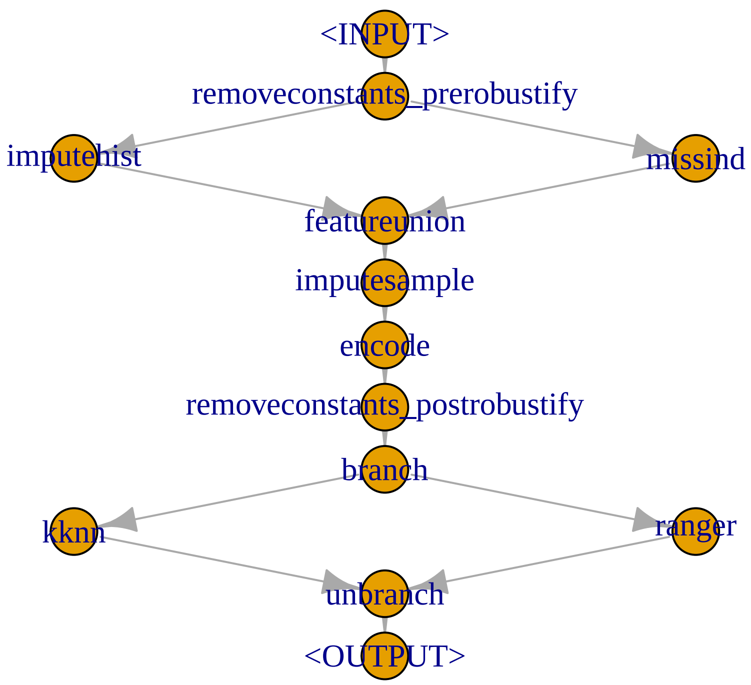
a)
```
library(mlr3verse)
library(mlr3tuning)

## Loading required package:  mlr3

## Loading required package:  paradox

(task <- tsk("pima"))

## <TaskClassif:pima> (768 x 9)
## * 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)
```r
ppl_preproc <- ppl("robustify", task = task, factors_to_numeric = TRUE)
```

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

e)
```r
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       levels nlevels
##  1:                ranger.oob.error ParamLgl    NA    NA   TRUE,FALSE       2
##  2:                ranger.max.depth ParamInt  -Inf   Inf                  Inf
##  3:                    ranger.alpha ParamDbl  -Inf   Inf                  Inf
##  4:                 ranger.min.prop ParamDbl  -Inf   Inf                  Inf
##  5:   ranger.regularization.factor ParamUty    NA    NA                  Inf
##  6: ranger.regularization.usedepth ParamLgl    NA    NA   TRUE,FALSE       2
##  7:                     ranger.seed ParamInt  -Inf   Inf                  Inf
##  8:                  ranger.minprop ParamDbl  -Inf   Inf                  Inf
##  9:                ranger.se.method ParamFct    NA    NA jack,infjack       2
## 10:                branch.selection ParamInt     1     2                    2
##     is_bounded special_vals       default storage_type                  tags
##  1:       TRUE    <list[0]>          TRUE      logical                 train
##  2:      FALSE    <list[1]>                    integer                 train
##  3:      FALSE    <list[0]>           0.5      numeric                 train
##  4:      FALSE    <list[0]>           0.1      numeric                 train
##  5:      FALSE    <list[0]>             1         list                 train
##  6:       TRUE    <list[0]>         FALSE      logical                 train
##  7:      FALSE    <list[1]>                    integer                 train
##  8:      FALSE    <list[0]>           0.1      numeric                 train
##  9:       TRUE    <list[0]>       infjack    character               predict
## 10:       TRUE    <list[0]> <NoDefault[3]>     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)
```r
# 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[49]>    pima <AutoTuner[41]> graph_learner.tuned
## 2: <TaskClassif[49]>    pima <AutoTuner[41]> graph_learner.tuned
## 3: <TaskClassif[49]>    pima <AutoTuner[41]> graph_learner.tuned
##            resampling resampling_id iteration              prediction
## 1: <ResamplingCV[19]>            cv         1 <PredictionClassif[20]>
## 2: <ResamplingCV[19]>            cv         2 <PredictionClassif[20]>
## 3: <ResamplingCV[19]>            cv         3 <PredictionClassif[20]>
##    classif.ce
## 1:  0.2500000
## 2:  0.2421875
## 3:  0.2148438

rr$aggregate()

## classif.ce
##  0.2356771
```

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

Solution 3:

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.