Black box on 'sick' dataset

Mateusz Bąkała

2020, 29 of April

Preprocessing repeated

To keep it fair, necessary code was replicated from previous report in order to recreate original modified dataset.

```r
# extraction
sick_dataset <- getOMLDataSet(data.id = 38)
data <- sick_dataset$data
target <- sick_dataset$target.features
indices <- read.table("indeksy_treningowe.txt")
train_length <- nrow(indices)
test_length <- nrow(data) - train_length
# preprocessing
data <- data %>%
  select(-c(TBG_measured, TBG, hypopituitary)) %>%
  mutate(age = ifelse(age == 455, 45, age))
# data split
data_train <- slice(data, indices$x)
data_test <- slice(data, -indices$x)
# imputation
data_train <- knnImputation(data_train, k = 15)
data_test <- knnImputation(data_test, k = 15, distData = data_train)
```

Models tried

Several black-box models were used to acquire a wider range of results. These were restricted to the ones available in `mlr3` package, though. Amongst them these were suited for classification task: `classif.glmnet`, `classif.kknn`, `classif.lda`, `classif.log_reg`, `classif.naive_bayes`, `classif.qda`, `classif.ranger`, `classif.rpart`, `classif.svm` and `classif.xgboost`. If we exclude interpretable models, only the following are left: `classif.ranger` and `classif.svm`. XGBoost was excluded due to the existence of an R package.

To be fair, there are extension packages containing more `mlr3` learners, but they are still poorly documented and mostly auto-imported from `mlr`, so not really reliable. Thus I opted for sticking to these two readily available.

Define variables

As `mlr3` presents object-oriented idea of programming, it will be useful to declare necessary variables first, such as task and learners.

```r
task <- TaskClassif$new(id = "data",
                        backend = rbind(data_train, data_test),
                        target = target,
                        positive = "sick")
```

```r
learner_ranger <- lrn("classif.ranger", predict_type = "prob")
learner_svm <- lrn("classif.svm", predict_type = "prob")
```

As we later noticed, svm demands no categorical columns. Thus one-hot encoding had to be performed. Luckily, `mlr3pipelines` are exactly what we need here:

```r
poe <- po("encode")
task <- poe$train(list(task))$output
```

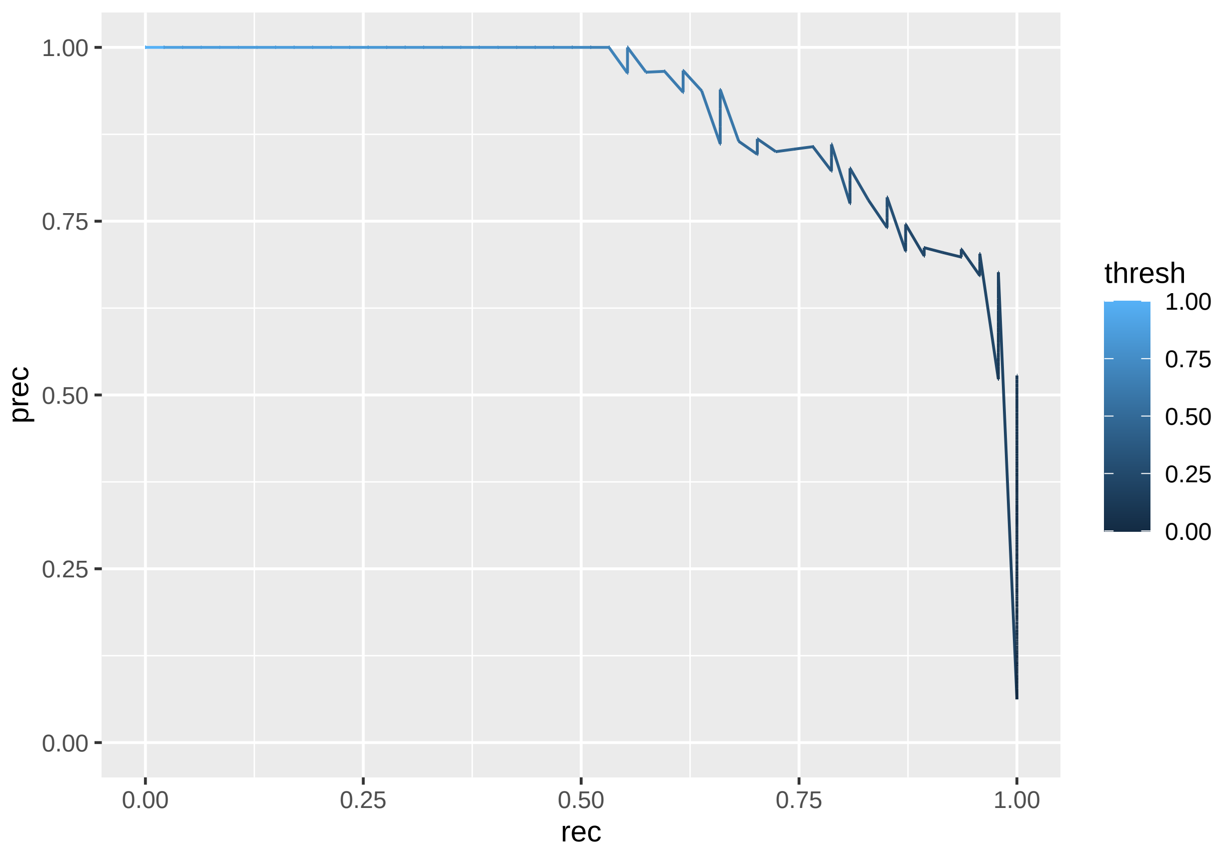
Before tuning

Experience gained from previous report clearly states that this dataset has a significant risk of model being overfitted. Thus for each model two scores shall be computed – before and after tuning.

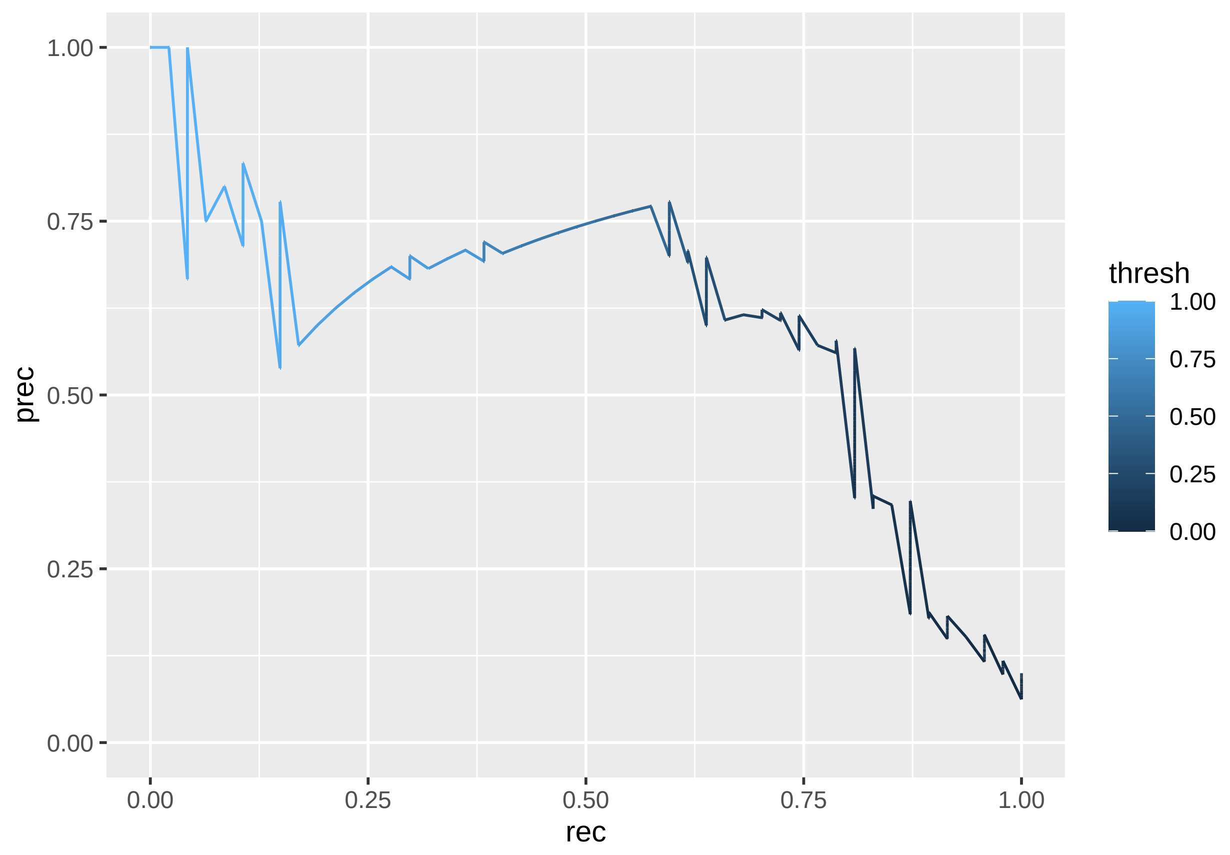
```r
prediction_ranger <- learner_ranger$train(
  task, row_ids = 1:train_length)$predict(
    task, row_ids = train_length + 1:test_length)
prediction_svm <- learner_svm$train(
  task, row_ids = 1:train_length)$predict(
    task, row_ids = train_length + 1:test_length)
auprc_ranger <- auprc(prediction_ranger$prob[, "sick"], data_test$Class, "sick")
auprc_svm <- auprc(prediction_svm$prob[, "sick"], data_test$Class, "sick")
knitr::kable(data.frame(learner = c("classif.ranger", "classif.svm"),
                        auprc = c(auprc_ranger, auprc_svm)))
```

learner	auprc
classif.ranger	0.8954025
classif.svm	0.5791360

Let's create AUPRC curve for ranger.



Sweet! Now, another one for svm.



As we can see, ranger has already quite formidable score, while svm… as usual, needs extensive tuning first and foremost. Now, it will be only fair to recompute rpart score, as one-hot had to be added. So:

```
learner_rpart <- lrn("classif.rpart", predict_type = "prob")
prediction_rpart <- learner_rpart$train(
  task, row_ids = 1:train_length)$predict(
    task, row_ids = train_length + 1:test_length)
auprc_rpart <- auprc(prediction_rpart$prob[, "sick"], data_test$Class, "sick")
knitr::kable(data.frame(learner = "classif.rpart",
                        auprc = auprc_rpart))
```

learner	auprc
classif.rpart	0.7763511

Indeed, rpart fares much worse than ranger. It's understandable, as ranger is essentially a lot of rparts cramped together, mixed and stirred. Now, let's train our models, especially svm.

Tuning

We already defined task and learners, so we only need to declare some other used by tuning instance and hyperparameter sets for each learner.

```
resampling <- rsmp("cv", folds = 5)
measure <- msr("classif.auprc")
terminator = term("evals", n_evals = 40)
tuner <- tnr("random_search")
```

```r
# svm hyperparameters
tune_ps_svm <- ParamSet$new(list(
  ParamDbl$new("coef0", lower = -1, upper = 3),
  ParamFct$new("kernel", levels = c("sigmoid", "polynomial"))
))
instance_svm <- TuningInstance$new(
  task = task,
  learner = learner_svm,
  resampling = resampling,
  measures = measure,
  param_set = tune_ps_svm,
  terminator = terminator
)
tuner$tune(instance_svm)
learner_svm$param_set$values <- instance_svm$result$params

# rpart hyperparameters
tune_ps_rpart <- ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.05),
  ParamInt$new("maxdepth", lower = 3, upper = 10)
))
instance_rpart <- TuningInstance$new(
  task = task,
  learner = learner_rpart,
  resampling = resampling,
  measures = measure,
  param_set = tune_ps_rpart,
  terminator = terminator
)
tuner$tune(instance_rpart)
learner_rpart$param_set$values <- instance_rpart$result$params

# ranger hyperparameters
tune_ps_ranger <- ParamSet$new(list(
  ParamInt$new("mtry", lower = 3, upper = 10),
  ParamInt$new("num.trees", lower = 400, upper = 1500)
))
instance_ranger <- TuningInstance$new(
  task = task,
  learner = learner_ranger,
  resampling = resampling,
  measures = measure,
  param_set = tune_ps_ranger,
  terminator = terminator
)
tuner$tune(instance_ranger)
learner_ranger$param_set$values <- instance_ranger$result$params
```

Now that tuning is (finally!) done, let's compute our scores.

```r
prediction_rpart <- learner_rpart$train(
  task, row_ids = 1:train_length)$predict(
    task, row_ids = train_length + 1:test_length)
prediction_ranger <- learner_ranger$train(
  task, row_ids = 1:train_length)$predict(
    task, row_ids = train_length + 1:test_length)
prediction_svm <- learner_svm$train(
  task, row_ids = 1:train_length)$predict(
    task, row_ids = train_length + 1:test_length)
auprc_rpart <- auprc(prediction_rpart$prob[, "sick"], data_test$Class, "sick")
auprc_ranger <- auprc(prediction_ranger$prob[, "sick"], data_test$Class, "sick")
auprc_svm <- auprc(prediction_svm$prob[, "sick"], data_test$Class, "sick")
knitr::kable(data.frame(learner = c("classif.rpart", "classif.ranger", "classif.svm"),
                        auprc = c(auprc_rpart, auprc_ranger, auprc_svm)))
```

learner	auprc
classif.rpart	0.6223753
classif.ranger	0.8243149
classif.svm	0.2817664

Ouch, these tuned models behaved as if they were minimizing AUPRC. Anyways, ranger could achieve quite nice scores, as some tuning CVs returned AUPRC above 0.94, whereas svm managed… about 0.7. Supposedly, svm isn't just suited to imbalanced classification type of tasks. Ranger obviously is, as single trees are as well. Rpart scored 0.88 twice out of 40 tries, so it's not that far off from ranger.

In conclusion, this comparison has shown us… that we could try and create an interpretable random (or semi-random?) forest, as it could give some solid scores. Svm just doesn't make it, unless one prepares data with this algorithm in mind, which wasn't the case here.