Exercise 12 – Nested Resampling

Introduction to Machine Learning

Hint: Useful libraries

R

```
# Consider the following libraries for this exercise sheet:

library(mlr3verse)
library(mlr3tuning)
```

Python

```
# Consider the following libraries for this exercise sheet:

# general
import numpy as np
import pandas as pd
# sklearn
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.compose import make_column_selector
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.metrics import balanced_accuracy_score
```

```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.feature_selection import VarianceThreshold
```

Exercise 1: Tuning Principles

> Learning goals
>
> 1. Understand model fitting procedure in nested resampling
> 2. Discuss bias and variance in nested resampling

Suppose that we want to compare four different learners:

Learner	Tuning required
Logistic regression (lm)	no
CART (rpart)	yes
k-NN (kknn)	yes
LDA (lda)	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?

Solution

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.

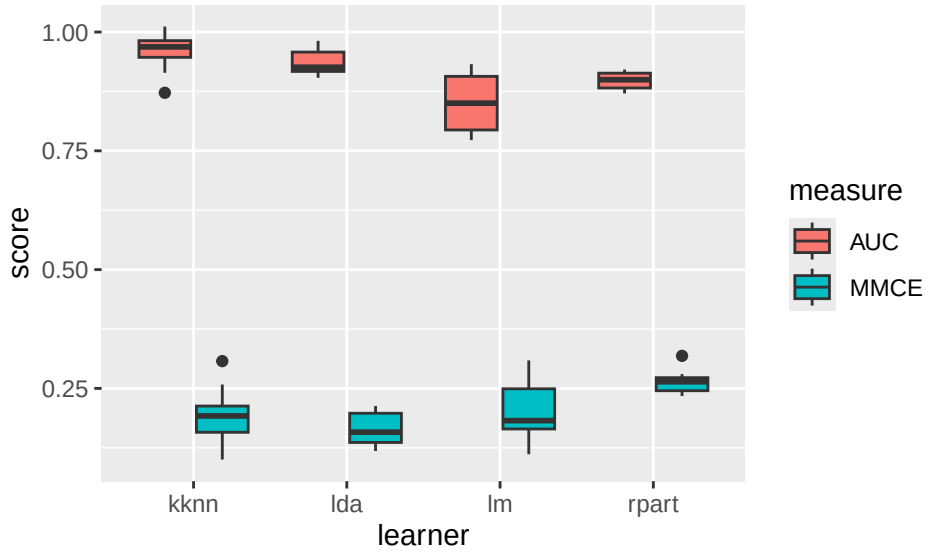
- So, in total:

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

- 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:

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

Giving the following benchmark result, which learner performs best? Explain your decision.



Solution

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

Recap briefly what is meant by the **bias-variance trade-off** in resampling.

Solution

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

Are the following statements true or not? Explain your answer in one sentence.

i. The bias of the generalization error estimate for 3-CV is higher than for 10-CV.

ii. Every outer loss can also be used as inner loss, assuming standard gradient-based optimization.

Solution

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).

Exercise 2: AutoML

> Learning goals
>
> Build autoML pipeline with R/Python

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

R Exercise

`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.

Note

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).

Create a task object in `mlr3` (the problem is pre-specified under the ID "pima").

Solution

```
(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
```

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`.

Solution

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

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`).

Solution

```
# Create a pipeline with preprocessing steps
ppl_preproc <- ppl("robustify", task = task, factors_to_numeric = TRUE)
```

Create another `ppl`, of type `branch` this time, to enable selection between your learners.

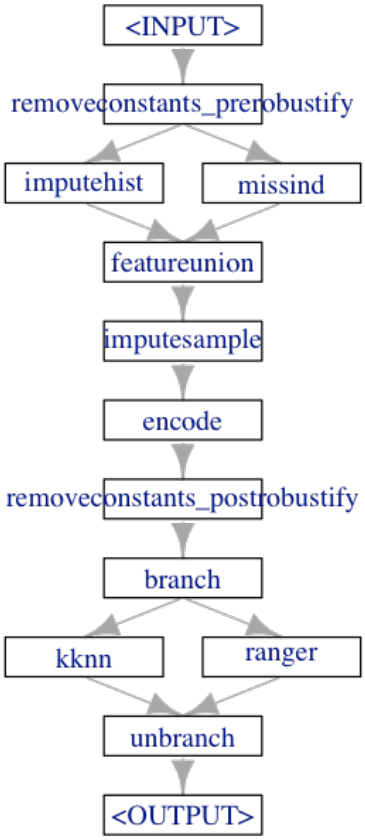
Solution

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

Chain both pipelines using the double pipe and plot the resulting graph. Next, convert it into a graph learner with `as_learner()`.

Solution

```
ppl_combined <- ppl_preproc %>>% ppl_learners
plot(ppl_combined)
graph_learner <- as_learner(ppl_combined)
```



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).

Solution

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

# 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"
```

A data.table: 10 x 11

id <chr>	class <chr>	lower <dbl>	upper <dbl>	levels <list>	nlevels <dbl>	is_bounded <lgl>	special_vals <list>	default <list>	storage_type <chr>	type <list>
ranger.sample.fraction	ParamDbl	0	1	NULL	Inf	TRUE	NULL	<environment: 0x162f09158>	numeric	train
ranger.save.memory	ParamLgl	NA	NA	TRUE, FALSE	2	TRUE	NULL	FALSE	logical	train
ranger.scale.permutation.importance	ParamLgl	NA	NA	TRUE, FALSE	2	TRUE	NULL	FALSE	logical	train
ranger.seed	ParamInt	NA	NA	jack , inf-jack	2	TRUE	NULL	infjack	character	predict

id <chr>	class <chr>	lower <dbl>	upper <dbl>	levels <list>	nlevels <dbl>	is_bounded <lgl>	special_vals <list>	default <list>	storage_type <chr>	tags <list>
ranger.seed	ParamInt	-Inf	Inf	NULL	Inf	FALSE	NULL	NULL	integer	train , predict
ranger.split	ParamUty	NA	NA	NULL	Inf	FALSE	NULL	NULL	list	train
ranger.splitrule	ParamFct	NA	NA	gini , extratrees, hellinger	3	TRUE	NULL	gini	character	train
ranger.verbose	ParamLgl	NA	NA	TRUE, FALSE	2	TRUE	NULL	TRUE	logical	train , predict
ranger.write.forest	ParamLgl	NA	NA	TRUE, FALSE	2	TRUE	NULL	TRUE	logical	train
branch.selection	ParamFct	1	2	NULL	2	TRUE	NULL	<environment: 0x17990cd08>	integer	train , predict , required

---

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.

Solution

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

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()`.

Solution

```
rr$score()
rr$aggregate()
```

A rr_score: 3 x 9

task <list>	task_id <chr>	learner <list>	learner_id <chr>	resampling <list>	resampling_id <chr>	iteration <int>	prediction <list>	classif.ce <dbl>
<environment: 0x17984ce48>	pima	<environment: 0x1655992d8>	graph_learner	<environment: 0x17a5a7248>	cv	1	<environment: 0x166bceba8>	0.2695312
<environment: 0x17984ce48>	pima	<environment: 0x165789c70>	graph_learner	<environment: 0x17a5a7248>	cv	2	<environment: 0x166926e28>	0.2617188
<environment: 0x17984ce48>	pima	<environment: 0x1631c8e38>	graph_learner	<environment: 0x17a5a7248>	cv	3	<environment: 0x16e16daa8>	0.1953125

classif.ce: 0.2421875

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

Python Exercise

`sklearn.pipeline.Pipeline` makes 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.

The purpose of the pipeline is to assemble several steps of transformation and a final estimator that can be cross-validated together while setting different parameters. So to speak, the pipeline estimator can be treated as any other estimator.

Load the data set pima, encode the target `diabetes` as 0-1-vector and perform a stratified `train_test_split`.

Solution

```
# load data
pima = pd.read_csv("../data/pima.csv")
X_pima = pima.copy()
y_pima = X_pima.pop("diabetes")

# encode the target as 0-1 vector
le = LabelEncoder()
y_pima = le.fit_transform(y_pima)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_pima, y_pima, test_size=0.2, stratif
```

As part of our modeling process, we want to perform certain preprocessing steps. While this step is highly customizable, we want to include at least One-Hot-Encoding of categorical features, and imputing of missing values.

Instance a `ColumnTransformer` object and include these two steps for a dynamic choice of columns.

Hint

Strings are considered as `dtype = object`.

Solution

```
# Define the column transformer
preprocessor = ColumnTransformer(
    transformers=[
        # One-hot encode categorical columns, like strings
        ('onehot', OneHotEncoder(drop='first', handle_unknown='ignore'), make_column_selec
        # Impute missing values for numerical columns
        ('imputer', SimpleImputer(strategy = 'median'), make_column_selector(dtype_include
    ])
```

Next, both pipelines for the k-NN and random forest are created. Like this you can create estimators with highly individual preprocessing steps. Include the previously created `ColumnTransformer`, a `VarianceThreshold` to remove constant columns and the corresponding estimator as a final step. Additionally, scale the columns for the k-NN estimator.

Solution

```
# Create a pipeline with preprocessing and modeling steps
# for knn
clf_knn = Pipeline([
    ('preprocessor', preprocessor),
    ('constant', VarianceThreshold()), # remove variables with constant values across all
    ('scaler', StandardScaler()),  # Data scaling
    ('classifier', KNeighborsClassifier())  # KNN model
])

# for random forest
clf_randomforest = Pipeline([
    ('preprocessor', preprocessor),
    ('constant', VarianceThreshold()), # remove variables with constant values across all
    ('classifier', RandomForestClassifier(random_state=42))  # Random Forest
])
```

A very common ensembling technique is to predict according to the decisions of numerous estimators. This is refered to as `VotingClassifier` and enables you to predict the class label based on the argmax of the sums of the predicted probabilities. Instanciate a `VotingClassifier` with the two classifier pipelines for k-NN and random forest.

Hint

Set the parameters `voting = "soft"` and `n_jobs = -1` for parallel computation.

Solution

```
# combine both classifiers with a soft voting ensembling
clf_voting = VotingClassifier(
    estimators=[('knn', clf_knn), ('random_forest', clf_randomforest)],
    voting = "soft",
    n_jobs = -1
```

```
)
```

Now you have an estimator object just like any other. Take a look at its tunable hyperparameters. You will optimize 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:

```
param_grid_voting = [{"<voting_estimator1>__<pipelie1_estimator>__<hyperparameter>":
                          list(<parameter_range>)},
                     {"<voting_estimator2>__<pipelie2_estimator>__<hyperparameter>":
                          list(<parameter_range>)}]
```

Please note, that the estimator names should be on par with the labels given in the `VotingClassifier`, the `Pipeline` and, of course, the hyperparameter of the used estimator in the pipeline. Each level of hyperparameters of our created ensemble estimator is accessable through the seperation `__` (double underscore).

Solution

```
# define a parameter grid for the tuning process
param_grid_voting = [{"knn__classifier__n_neighbors": list(range(1,11))},
                     {"random_forest__classifier__max_features": list(range(1,6))}]
```

Nested Resampling is a method to avoid the so called `optimization bias` by tuning parameters and evaluation performance on different subsets of your training data.

Use

- Stratified 3-CV in both inner and outer loop.
- accuracy as inner performance measure,
- grid search as tuning algorithm.

You may use the following, incomplete code to compute the nested resampling:

```
NUM_OUTER_FOLDS = <...>
nested_scores_voting = np.zeros(NUM_OUTER_FOLDS) # initalize scores with 0
# Choose cross-validation techniques for the inner and outer loops,
# independently of the dataset.
# E.g "GroupKFold", "LeaveOneOut", "LeaveOneGroupOut", etc.
inner_cv = <...>(n_splits=<...>, shuffle=True, random_state=42)
outer_cv = <...>(n_splits=<...>, shuffle=True, random_state=42)

for i, (train_index, val_index) in enumerate(outer_cv.split(X_train, y_train)):
  # Nested CV with parameter optimization for ensemble pipeline
  clf_gs_voting = <...>(
      estimator=<...>,
      param_grid=<...>,
      cv=<...>,
      n_jobs=-1
  )
  clf_gs_voting.fit(X_train.iloc[<...>], y_train[<...>])
  nested_scores_voting[i] = clf_gs_voting.score(X_train.iloc[<...>], y_train[<...>])
```

Solution

Define resampling strategies

```
# initalize scores with 0
NUM_OUTER_FOLDS = 3
nested_scores_voting = np.zeros(NUM_OUTER_FOLDS)
gsCV_nested = []

# Choose cross-validation techniques for the inner and outer loops, independently of the d
# E.g "GroupKFold", "LeaveOneOut", "LeaveOneGroupOut", etc.
inner_cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
outer_cv = StratifiedKFold(n_splits=NUM_OUTER_FOLDS, shuffle=True, random_state=43)
```

Run loop

```
for i, (train_index, val_index) in enumerate(outer_cv.split(X_train, y_train)):
    # Nested CV with parameter optimization for ensemble pipeline
    clf_gs_voting = GridSearchCV(
        estimator=clf_voting,
        param_grid=param_grid_voting,
        cv=inner_cv,
        n_jobs=-1
```

```
    )
    #gsCV_nested.append(clf_gs_voting)
    clf_gs_voting.fit(X_train.iloc[train_index], y_train[train_index])
    nested_scores_voting[i] = clf_gs_voting.score(X_train.iloc[val_index], y_train[val_ind
```

Extract performance estimates per outer fold and overall (as mean). According to your results, determine the best classifier object.

Solution

per fold

```
# print performance per outer fold
print(nested_scores_voting)
```

```
[0.75121951 0.71707317 0.75490196]
```

aggregated

```
# print performance aggregated over all folds
print(nested_scores_voting.mean())
```

```
0.7410648812370476
```

detailed

```
# Nested CV with parameter optimization for ensemble pipeline
clf_gs_voting = GridSearchCV(
    estimator=clf_voting,
    param_grid=param_grid_voting,
    cv=inner_cv,
    n_jobs=-1
)
clf_gs_voting.fit(X_train, y_train)
```

```
GridSearchCV(cv=StratifiedKFold(n_splits=3, random_state=42, shuffle=True),
             estimator=VotingClassifier(estimators=[('knn',
                                                     Pipeline(steps=[('preprocessor',
                                                                      ColumnTransformer(trans
                                                                                        ('constant',
                                                                                         VarianceThreshold()),
                                                                                        ('classifier',
                                                                                         RandomForestClassifier
                                    n_jobs=-1, voting='soft'),
             n_jobs=-1,
             param_grid=[{'knn__classifier__n_neighbors': [1, 2, 3, 4, 5, 6, 7,
                                                           8, 9, 10]},
                         {'random_forest__classifier__max_features': [1, 2, 3,
                                                                      4, 5]}])
```

Lastly, evaluate the performance on the test set. Think about the imbalance of your data set and how this is affecting the performance measurement accuracy. Try to find a better metric and compare these two.

Solution

```
# evaluate performace on test set with accuracy
test_scores_voting = clf_gs_voting.score(X_test, y_test)
print(test_scores_voting)
```

```
0.7467532467532467
```

Accuracy does not account for imbalanced data! Let's check how the test data is distributed:

```
unique, counts = np.unique(y_test, return_counts=True)
table = pd.DataFrame(data = dict(zip(unique, counts)), index=[0]) #index necassary because
table
```

	0	1
0	100	54

Confusion matrix

```
pred_test = clf_gs_voting.predict(X_test)
conf_matrix = pd.DataFrame(confusion_matrix(pred_test, y_test))
conf_matrix
```

	0	1
0	84	23
1	16	31

The distribution shows a shift towards 'false' with 2/3 of all test observations.

```
# evaluate performace on test set with balanced accuracy to account for imbalances data se
# Balanced accuracy = (Sensitivity + Specificity) / 2
balanced_accuracy = balanced_accuracy_score(y_test, pred_test)
print(balanced_accuracy)
```

```
0.707037037037037
```

The balanced accuracy is lower than a normal accuracy score, as it accounts seperatly for the lower Sensitivity.

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

> Learning goals
>
> Apply course contents to real-world problem

Make yourself familiar with the Titanic Kaggle challenge.

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 mlr3 or scikit-learn).
- How do you fare compared to the public leaderboard?

mlr3 Hint

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

Solution

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.