classiFunc: Classification of Functional Data

Thomas Maierhofer

2017-05-29

This vignette gives a quick introduction to the key features and functions included in the `classiFunc` package. Please use the Project Page to suggest new features or report bugs. This package offers an extensible and efficient implementation of `k` nearest neighbor classification for functional data.

The following chunk gives a quick introduction to the usage of the `classiFunc` package.

```
library("classiFunc")

# classification of the ArrowHead data set
data("ArrowHead", package = "classiFunc")
classes = ArrowHead[,"target"]

set.seed(123)
# use 80% of data as training set and 20% as test set
train_inds = sample(1:nrow(ArrowHead), size = 0.8 * nrow(ArrowHead), replace = FALSE)
test_inds = (1:nrow(ArrowHead))[!(1:nrow(ArrowHead)) %in% train_inds]

# create functional data as matrix with observations as rows
fdata = ArrowHead[,!colnames(ArrowHead) == "target"]

# create a k = 3 nearest neighbor classifier with Euclidean distance (default) of the
# first order derivative of the data
mod = classiKnn(classes = classes[train_inds], fdata = fdata[train_inds,],
nderiv = 1L, knn = 3L)
# or create a kernel estimator
mod2 = classiKernel(classes = classes[train_inds], fdata = fdata[train_inds,])

# predict the model for the test set
# matrix with the prediction probabilities for the three classes
pred = predict(mod, newdata =  fdata[test_inds,], predict.type = "prob")
```

All functionality of this package can also be accessed through the `mlr` package Project Page. For an introduction on how to use `mlr` check out the Online Tutorial. Currently, the learners are not merged into the Master branch of `mlr`. If you want to use the development version, please download the package from the Project Page. The following chunck gives a quick introduction on how to use the `classiFunc` learners in `mlr`.

```
# download and install the mlr branch containing the classiFunc learners
# devtools::install_github("maierhofert/mlr",
#                          ref = "fda_pull1_task")
```

```
library("mlr")

# classification of the ArrowHead data set
data("ArrowHead", package = "classiFunc")

# create the classiKnn learner for classification of functional data
lrn = makeLearner("fdaclassif.classiKnn")
# create a task from the training data set
task = makeFDAClassifTask(data = ArrowHead[train_inds,], target = "target")
# train the model on the training data task
m.mlr = train(lrn, task)

# predict the test data set
pred = predict(m.mlr, newdata = ArrowHead[test_inds,])
```

By using the `mlr` interface for this package, a multitude of new possibilities are available. One of the key features to be added by the `mlr` package is automatic hyperparameter tuning. In the following chunk a kernel estimator is created that automatically chooses its band width by cross validation.

```
# create the classiKernel learner for classification of functional data
lrn.kernel = makeLearner("fdaclassif.classiKernel", predict.type = "prob")

# create parameter set
parSet.bandwidth = makeParamSet(
  makeNumericParam(id = "h", lower = 0, upper = 10)
)

# control for tuning hyper parameters
# Use higher resolution in application
ctrl = makeTuneControlGrid(resolution = 11L)

# tuned learners
lrn.bandwidth.tuned = makeTuneWrapper(learner = lrn.kernel,
                                      resampling = makeResampleDesc("CV", iters = 5),
                                      measures = mmce,
                                      par.set = parSet.bandwidth,
                                      control = ctrl)

# train the model on the training data task
m = train(lrn.bandwidth.tuned, task)

# predict the test data set
pred = predict(m, newdata = ArrowHead[test_inds,])
```

The Brier score optimal ensemble proposed in Fuchs et al. (2015), Nearest neighbor ensembles for functional data with interpretable feature selection, can also be reproduced using the implementation in `mlr`. A newly implemented stacked learner aggregates the individual base learners to an ensemble learner by creating a weighted mean of their individual predictions. Other ensemble learners can easily be created.

```r
# create the base learners
b.lrn1 = makeLearner("fdaclassif.classiKnn",
                     id = "Manhattan.lrn",
                     par.vals = list(metric = "Manhattan"),
                     predict.type = "prob")
b.lrn2 = makeLearner("fdaclassif.classiKnn",
                     id = "mean.lrn",
                     par.vals = list(metric = "mean"),
                     predict.type = "prob")
b.lrn3 = makeLearner("fdaclassif.classiKnn",
                     id = "globMax.lrn",
                     par.vals = list(metric = "globMax"),
                     predict.type = "prob")

# create an ensemble learner as porposed in Fuchs et al. (2015)
# the default uses leave-one-out CV to estimate the weights of the base learners as
proposed in the original paper
# set resampling to CV for faster run time
ensemble.lrn = makeStackedLearner(base.learners = list(b.lrn1, b.lrn2, b.lrn3),
                                  predict.type = "prob",
                                  resampling = makeResampleDesc("CV", iters = 5L),
                                  method = "classif.bs.optimal")

# create another ensemble learner
ensemble.rf.lrn = makeStackedLearner(base.learners = list(b.lrn1, b.lrn2, b.lrn3),
                                     super.learner = "classif.randomForest",
                                     predict.type = "prob",
                                     method = "stack.cv")

# train the model on the training data task
ensemble.m = train(ensemble.lrn, task)

# predict the test data set
pred = predict(ensemble.m, newdata = ArrowHead[test_inds,])
```