EBDS20102 - Supervised Machine Learning - *Exercise 3*

Comparison of Kernel Ridge Regressions for Predicting Airline Data

Mathijs de Jong	Chao Liang	Yuchou Peng	Eva Mynott
380891	588140	481487	495602

For this assignment, we apply ridge regression using kernel transformation to predict airline data. We created the package `mlkit`[1] for storing the implementations used in this course.

The following implementation of a linear ridge regression that supports kernel transformation has been implemented.

```
#' Fitting Linear Models with Ridge Penalty using Dual Solution
#'
#' @description Implementation of the dual analytical solution for a linear
#' regression model with a Ridge penalty term.
#'
#' @param formula an object of class \code{\link[stats]{formula}}: a symbolic
#' description of the model to be fitted following the standard of
#' \code{\link[stats]{lm}}.
#' @param data an optional data frame, list or environment (or object coercible
#' by \code{\link[base]{as.data.frame}} to a data frame) containing the
#' variables in the model. If not found in data, the variables are taken from
#' environment (\code{formula}), typically the environment from which this
#' function is called.
#' @param lambda penalty term scaling hyperparameter.
#' @param intercept optional boolean indicating whether to fit an intercept. If
#' \code{TRUE}, \code{standardize} is ignored. Default is \code{FALSE}.
#' @param standardize optional boolean indicating whether to return results for
#' standardized data. If \code{intercept} is \code{TRUE}, this argument is
#' ignored. Default is \code{FALSE}.
#' \code{TRUE}, \code{standardize} is ignored. Default is \code{FALSE}.
#' @param kernel optional kernel to use in the ridge regression model. By
#' default the linear kernel with constant zero is used, that is, no kernel
#' transformation is applied. See the details section for more details on
#' available kernel transformations.
#' @param const optional constant parameter for the kernel transformation.
#' Default is \code{0}.
#' @param degree optional degree parameter in the kernel transformation. Default
#' is \code{NULL}.
#' @param scale optional scale parameter in the kernel transformation. Default
#' is \code{NULL}.
#'
#' @return \code{dual.ridge.lm} returns an object of \code{\link{class}}
#' \code{mlkit.dual.ridge.fit}. An object of class \code{mlkit.dual.ridge.fit}
#' is a list containing at least the following components:
#' \item{coefficients}{a named vector of optimal coefficients.}
#' \item{alpha}{L1-weight hyperparameter in elastic net penalty term.}
#' \item{lambda}{penalty term scaling hyperparameter.}
#' \item{r2}{coefficient of determination for optimal coefficients.}
```

[1] `mlkit` is available on Github: https://github.com/Accelerytics/mlkit

```
#' \item{x}{matrix containing the explanatory variables used in estimation.}
#' \item{ker.mat}{kernel matrix used in estimation.}
#' \item{kernel}{kernel transformation.}
#' \item{const}{constant parameter in the kernel transformation.}
#' \item{degree}{degree parameter in the kernel transformation.}
#' \item{scale}{scale parameter in the kernel transformation.}
#'
#' @export
#'
dual.ridge.lm = function(formula, data, lambda, intercept=F, standardize=F,
  kernel='lin', const=0, degree=NULL, scale=NULL) {

  # Extract dependent variable and explanatory variables
  x = stats::model.matrix(formula, data)
  y = data.matrix(data[, all.vars(formula)[1]]);

  # Store scaling parameters
  if (!intercept & !standardize) {
    x.mean = colMeans(x); x.sd = apply(x, 2, stats::sd)
    y.mean = mean(y); y.sd = stats::sd(y)
  }

  # Add intercept or standardize data if necessary
  x = create.x(x, intercept, standardize)
  y = create.y(y, intercept, standardize)

  # Generate transformed explanatory variables matrix k
  k = create.k(x, kernel, const, degree, scale)

  # Estimate model
  n = nrow(x)
  j = diag(n) - 1 / n
  k.til = j %*% k %*% j
  eig = eigen(k.til, symmetric = TRUE); quad.eig.vals = eig$value ^ 2
  inv.mat = diag(quad.eig.vals / (quad.eig.vals + lambda))
  w0 = mean(y)
  q.til = as.vector(eig$vectors %*% inv.mat %*% t(eig$vectors) %*% j %*% y)

  # Construct output
  coefficients = c('w0'=w0, q.til)
  y.hat = coefficients[1] + coefficients[-1]
  rss = sum((y - y.hat) ^ 2)
  r2 = 1 - rss / sum((y - mean(y)) ^ 2)

  # Return mlfit object
  res = list('coefficients'=coefficients, 'alpha'=0, 'lambda'=lambda, 'r2'=r2,
    'x'=x, 'ker.mat'=k, 'kernel'=kernel, 'const'=const, 'degree'=degree,
    'scale'=scale)
  class(res) = 'mlkit.dual.ridge.fit'
  return(res)
}
```

To be able to use the function above for predictions, it has to support the generic `predict` function. The following function has been defined to ensure this is the case.

```r
#' Model Predictions for Dual Ridge Regression Model
#'
#' @description Custom implementation for prediction using model fits using the
#' \code{\link{dual.ridge.lm}} method.
#'
#' @method predict mlkit.dual.ridge.fit
#'
#' @param object \code{mlkit.dual.ridge.fit} object generated by a call to the
#' \code{\link{dual.ridge.lm}} method that is used for prediction.
#' @param newdata optional matrix of explanatory variables to use in prediction.
#' Default is \code{NULL} in which case the in-sample predictions are returned.
#' @param ... additional arguments affecting the predictions produced.
#'
#' @return Atomic vector containing predictions based on the given model and
#' explanatory variables.
#'
#' @export
#'
predict.mlkit.dual.ridge.fit = function(object, newdata=NULL, ...) {

  # Generate transformed explanatory variables matrix k
  k = create.k(object$x, object$kernel, object$const, object$degree,
    object$scale, length(object$yhat), y=newdata)

  # Generate predictions and return them in an atomic vector
  yhat = object$coefficients[1] + k %*% MASS::ginv(object$ker.mat) %*%
    object$coefficients[-1]
  return(yhat)
}
```

Next, we can use the `grid.search.cross.validation` function for hyperparameter tuning.

```r
#' Grid search K-fold cross-validation
#'
#'@description
#' Implementation of the grid search approach using K-fold cross-validation for
#' hyperparameter tuning of a given \code{estimator}.
#'
#' @param formula an object of class \code{\link[stats]{formula}}: a symbolic
#' description of the model to be fitted following the standard of
#' \code{\link[stats]{lm}}.
#' @param data an optional data frame, list or environment (or object coercible
#' by \code{\link[base]{as.data.frame}} to a data frame) containing the
#' variables in the model. If not found in data, the variables are taken from
#' environment (\code{formula}), typically the environment from which this
#' function is called.
#' @param estimator estimator function that has arguments \code{formula} and
#' \code{data} and returns a list containing parameters estimates in the
#' component \code{coefficients}.
#' @param params.list list or vector containing hyperparameters and their
#' respective values to consider
#' @param n.folds optional number of folds (K) in cross-validation. Default is
#' 5.
#' @param ind.metric metric function taking in a numerical vector of
```

```
#' predictions for the dependent variable together with a numerical vector of
#' actual outcomes of the dependent variable that returns a performance metric
#' for the individual folds.
#' @param comb.metric optional function used to combine individual fold
#' metrics. Default is \code{\link[base]{mean}}.
#' @param fold.id optional vector containing numerical fold identifiers for
#' each row in the data. If \code{NULL}, \code{n.folds} is used and random fold
#' identifiers are constructed divided the observations equally over K folds.
#' If provided, \code{n.folds} is ignored. Default is \code{NULL}.
#' @param force optional boolean indicating whether or not to allow for errors
#' due to singularity when applying the estimator. If \code{TRUE}, the
#' individual metric of folds with hyperparameter combinations for which the
#' estimator is not able to estimate coefficients due to singularity, are set to
#' \code{Inf} and hence the errors are ignored. Default is \code{FALSE}.
#' @param verbose optional boolean indicating whether to show a progress bar.
#' Default is \code{FALSE}.
#' @param plot optional boolean indicating whether to generate heatmaps of
#' performance on the grid and displaying the estimated coefficients. Default
#' is \code{FALSE}.
#' @param contour.scale optional vector containing the hyperparameters and
#' their scales for the axes in the contour plots. Default is \code{NULL}.
#' @param coef.lims optimal limits of the coefficients plot. Default is
#' \code{NULL}.
#' @param coef.names optional names of coefficients, used in estimates barplot.
#' Default is \code{NULL} in which case the column names of the explanatory
#' variables are used.
#' @param seed optimal seed to specify. Default is \code{NULL}.
#' @param ... additional arguments to be passed to the \code{estimator}
#' function.
#'
#' @return \code{grid.search.cross.validation} returns an object of
#' \code{\link{class}} \code{"gscv"}. An object of class \code{"gscv"} is a
#' list containing at least the following components:
#' \item{coefficients}{a named vector of optimal coefficients.}
#' \item{metric}{metric of the optimal hyperparameters.}
#' \item{params}{a named vector of optimal hyperparameters.}
#' @export
#'
grid.search.cross.validation = function(formula, data, estimator, params.list,
  n.folds=5, ind.metric, comb.metric=mean, fold.id=NULL, force=F, verbose=F,
  plot=F, contour.scale=NULL, coef.lims=NULL, coef.names=NULL, seed=NULL, ...) {

  # Define constants
  y = data.matrix(data[, all.vars(formula)[1]]); set.seed(seed)
  x = stats::model.matrix(formula, data); N = nrow(x); opt.metric = Inf

  # Specify fold ids if not given and initialize metrics and ids vector
  if(is.null(fold.id)) fold.id = ((1:N) %% n.folds + 1)[sample(N, N)]
  else n.folds =length(unique(fold.id)); metrics = rep(NULL, n.folds);
  test.ids = matrix(nrow=n.folds, ncol=N)
  for (fold in 1:n.folds) test.ids[fold, ] = (fold.id == fold)

  # Create grid for cross validation search
```

```
grid = expand.grid(params.list, stringsAsFactors=F)
n.combs = nrow(grid); metric = rep(NULL, n.combs)

# If verbose, initialize progress bar
if (verbose) pb = progress::progress_bar$new(total = n.combs * n.folds)

# Apply grid search
for (i in 1:n.combs) {

  # Apply cross validation and if verbose, update progress bar
  for (fold in 1:n.folds) { if (verbose) pb$tick()

    # Compute individual metric on fold
    metrics[fold] = tryCatch(
      ind.metric(stats::predict(do.call(estimator, c(list(formula=formula,
        data=data[!test.ids[fold, ], ]), as.list(grid[i, ]), list(...))),
        Xu=x[test.ids[fold, ], ]), y[test.ids[fold, ]]),
      error = function(e) {
        warning(paste('Failed for', paste(names(params.list), '=', grid[i, ],
          collapse=', ')))
        if (force & grepl('.*(singular)|(infinite).*', e$message)) return(Inf)
        stop(e)
      }
    )
  }

  # Combine performances on folds to overall performance
  metric[i] = comb.metric(metrics)
}

# Extract optimal hyperparameters
opt.id = which.min(metric)
opt.metric = metric[opt.id]; opt.params = grid[opt.id, ]

# Estimate best beta
opt.beta = do.call(estimator, c(list(formula=formula, data=data),
  as.list(opt.params), list(...)))$coefficients

# Plot heatmaps and coefficients if required
if (plot) {
  grid$metric = metric

  # Contour plots if more than 1 hyperparameter
  if (length(params.list) > 1) {
    combs = utils::combn(names(params.list), 2);
    for (i in 1:ncol(combs)) {
      col.x = combs[1, i]; col.y = combs[2, i]
      p = ggplot2::ggplot(data = grid, ggplot2::aes_string(x=col.x, y=col.y,
        z='metric')) + ggplot2::geom_contour_filled() +
        ggplot2::ylab(latex2exp::TeX(paste0('$\\', col.y, '$'))) +
        ggplot2::xlab(latex2exp::TeX(paste0('$\\', col.x, '$')))
      if (!is.null(contour.scale))
        p = p + ggplot2::scale_y_continuous(trans=contour.scale[col.y]) +
```

```
          ggplot2::scale_x_continuous(trans=contour.scale[col.x])
      }
    } else {
      col.x = names(params.list)[1]; p = ggplot2::ggplot(data = grid,
        ggplot2::aes_string(x=col.x, y=metric)) + ggplot2::geom_line() +
        ggplot2::ylab('metric') +
        ggplot2::xlab(latex2exp::TeX(paste0('$\\', col.x, '$')))
      if (!is.null(contour.scale))
        p = p + ggplot2::scale_x_continuous(trans=contour.scale[col.x])
    }
    print(p)

    # Plots beta estimates
    if (is.null(coef.names)) coef.names = c('(Intercept)', colnames(x))
    p = ggplot2::ggplot(data.frame(y=coef.names,
      beta=as.vector(opt.beta)), ggplot2::aes(beta, y)) + ggplot2::geom_col() +
      ggplot2::ylab('Expl. variable') +
      ggplot2::xlab(latex2exp::TeX('$\\beta$'))
    if (!is.null(coef.lims)) p = p + ggplot2::xlim(coef.lims)
    print(p)
  }

  # Return gscv object
  res = list('coefficients'=opt.beta, 'metric'=opt.metric,
    'params'=c(opt.params))
  class(res) = 'gscv'
  return(res)
}
```

Using all functions defined above, we can now use the following code to compare different implementations of linear kernel ridge regressions. Note that the hyperparameters considered in grid search are arbitrary chosen. Furthermore, the standard `dsmle::krr` function is replaced by a function that supports formulas so it can be used in the grid search cross validation method.

As expected, we find similar results for the different implementations of the kernel ridge regressions.

```
################################################################################
# Initialize local settings
################################################################################

# Specify working directory
WEEK = 'Week 3'
setwd(paste0(BASE.DIR, '/Supervised Machine Learning/', WEEK, '/Assignment'))

# Specify options
options(scipen=999)
set.seed(42)

################################################################################
# Load dependencies
################################################################################

# Install and load packages
install.packages(setdiff(c('car', 'caret', 'devtools', 'e1071', 'Ecdat',
  'ISLR', 'plotrix'), installed.packages()))
```

```r
# Install and load latest version of own package
devtools::install_github('Accelerytics/mlkit', upgrade='always', force=T)
library(mlkit)

# Install and load dsmle package
install.packages("dsmle_1.0-4.tar.gz", repos=NULL, type="source")
library(dsmle)

################################################################################
# Pre-process data
################################################################################

# Load data
df = Ecdat::Airline
formula = output ~ 0 + .

# Specify hyperparameter values to consider
params.length = 10
lambdas = c(0, 10 ^ seq(-5, 5, length.out=params.length - 1))
kernels = c('lin', 'pol', 'rbf')
consts = c(0, 0.5, 1, 2, 10, 100, 1000)
degrees = c(0.5, 1, 2, 10, 100, 1000)
scales = c(0.5, 1, 2, 10, 100, 1000)
params.list = list(
  'lambda' = lambdas,
  'kernel' = kernels,
  'const' = consts,
  'degree' = degrees,
  'scale' = scales
)

# Specify fold ids
N = nrow(df); n.folds = 5; fold.id = ((1:N) %% n.folds + 1)[sample(N, N)]

################################################################################
# Comparison of different implementations
################################################################################

# Define metric functions
rmse = function(y.hat, y) sqrt(mean((y.hat - y) ^ 2))

# Hyperparameter tuning using estimator based on mlkit implementation
gscv.mlkit = mlkit::grid.search.cross.validation(formula,
  as.data.frame(scale(df)), mlkit::dual.ridge.lm, params.list, ind.metric=rmse,
  fold.id=fold.id, verbose=T, force=T)
print(gscv.mlkit)

# Hyperparameter tuning using estimator based on dsmle implementation
krr.function = function(formula, data, lambda, kernel, const, degree, scale) {
  y = data.matrix(data[, all.vars(formula)[1]]);
  X = stats::model.matrix(formula, data)
  kernel.type = ifelse(kernel == 'lin', 'linear', ifelse(kernel == 'rbf',
    'RBF', 'nonhompolynom'))
```

```
  return(dsmle::krr(y, X, lambda, kernel.type, degree, scale, center=F,
    scale=F))
}
gscv.dsmle = mlkit::grid.search.cross.validation(formula,
 as.data.frame(scale(df)), krr.function, params.list, ind.metric=rmse,
 fold.id=fold.id, verbose=T, force=T)
print(gscv.dsmle)
```