Model tuning

STAT30270 – Statistical Machine Learning

Contents

1 Model tuning via the `caret` package

In some of the past lab examples, we performed model comparison and tuning by manually implementing the cross-validation procedure. The R package `caret` provides an extensive and integrated collection of functionalities that we can use for data splitting and model tuning using resampling-based methods. A manual describing the many functionalities of `caret` is available here:

- `https://topepo.github.io/caret/`

The package can be used in conjunction with a large collection of supervised learning methods to tune the corresponding hyperparameters via cross-validation. A model can be specified via the `method` tag, where each model has associated complexity hyperparameters. See `?train` for details and the webpage `https://topepo.github.io/caret/train-models-by-tag.html` for further information. The table below lists the models of this module and some of their associated hyperparameters:

Model	**Method**	**Hyperparameter**
Logistic regression	`glm`	Variables/terms included in the model formulation
	`glmStepAIC`	Stepwise variable selection with AIC
SVM - Polynomial	`svmPoly`	`degree` (polynomial degree), `scale` (scaling coefficient), `C` (cost)
SVM - GRBF	`svmRadial`	`sigma` (sigma scaling coefficient), `C` (cost)
Classification trees	`rpart`	`cp` (tree complexity parameter)
Random forests	`rf`	`mtry` (number predictors for split)

Note that not all the hyperparameters of a supervised learning method can be automatically tuned using `caret`. For example, tuning the classification threshold for a probabilistic classifier (like logistic regression) would require some ad-hoc implementation, which can still leverage some of the functionalities of `caret`.

1.1 Tuning SVM with GRBF kernel

We consider an application to data eye scans for the purpose of computer-assisted diagnosis of diabetic retinopathy. The data is available in the file `data_rethinopaty_messidor.csv`. This dataset contains features extracted from the Messidor image data set to predict whether an image contains signs of diabetic retinopathy or not. The target variable is `class`, which indicates the presence or absence of diabetic retinopathy. All input features represent either a detected lesion (`ma`), a descriptive numerical feature of a blood vessel leak (`exudate`), or an image-level descriptor (`macula_opticdisc_distance` and `opticdisc_diameter`). Three additional binary features concern quality of the assessment, screening, and an additional categorization (`quality`, `pre-screening`, `am_fm_classification`). Additional details are available here:

- `https://archive.ics.uci.edu/dataset/329/diabetic+retinopathy+debrecen`.

- `https://www.adcis.net/en/third-party/messidor/`

We remove the binary features, as they are related to pre-screening, quality, additional categorization and not relevant to the purpose of predicting retinopathy as a function of image-derived numerical features. All the other numerical input features are kept. We also convert the target label variable to `factor`.

We then split the data into test and training+validation sets. The test data will be set aside for evaluating the generalized predictive performance. We set aside 20% of the data for testing. The remaining portion of the data will be used for model training and tuning. The function `createDataPartition` can be used to easily split the data into a stratified random sample to construct the test set and the training+validation sets.

To avoid information leakage (see lab 3), we standardize the training+validation data and the test data using the summary statistic computed from the training+validation data – See Lab 3 on model evaluation. This can be achieved using the function `preProcess` and setting the argument `method = c("center", "scale")`. The function estimates the transformation parameter from the data set in input, these can be applied using the function `predict` to any data set with the same variables.

```r
# load
library(caret)

# load data, rename and convert class column
data_messidor <- read.csv("data/data_retinopathy_messidor.csv")
data_messidor <- data_messidor[,-c(1,2,19)]           # remove binary variables
data_messidor$class <- factor(data_messidor$class)    # convert to factor

# check range of values
range(data_messidor[,-17])

# split
train_val <- createDataPartition(data_messidor$class, p = 0.80, list = FALSE)
data <- data_messidor[train_val,]
data_test <- data_messidor[-train_val,]

# standardize
pre_scale <- preProcess(data, method = c("center", "scale"))
data <- predict(pre_scale, data)
data_test <- predict(pre_scale, data_test)
#
# # check me
# colMeans(data[,-17])
# colMeans(data_test[,-17])
```

Now that we split our data, we will implement k-fold cross-validation to perform hyperparameter tuning. The function `trainControl` is employed to specify the cross-validation procedure implemented. The argument `method` specifies what cross-valdation approach to use, here we set it to `repeatedcv` to perform repeated cross-validation. The arguments `number` define the number of folds, while the argument `repeats` is used to specify the number of replications.

The function `train` is used to implement the cross-validation procedure. The function requires in input the list of settings from `trainControl`. As we use the SVM classifier with GRBF kernel, we set the `method` argument to `svmRadial`.

NOTE: The following code chunks might return several warning messages. These are not really a problem and often related to internal settings of the function used to implement the different classifiers. You can safely turn these messages off in your R markdown file using the chunk option `message=FALSE`. Alternatively, you could turn off warning messages globally by running in R `options(warn = -1)`, then use `options(warn = 0)` to turn them on again (not recommended).

```r
# 5-fold cv with 10 replications
train_ctrl <- trainControl(method = "repeatedcv", number = 5, repeats = 10)

# implement cross-validation with GRBF SVM
```

```r
fit_svm_grbf <- train(class ~ ., data = data,
                 method = "svmRadial",
                 trControl = train_ctrl)

fit_svm_grbf
```

The function implements the cross-validation procedure and it returns the estimated validation accuracy averaged over the replications and folds. By default, the function `train` considered a restricted set of values of the cost `C` and it held fixed the coefficient `sigma`. We can visualize the average accuracy as a function of the cost `C` (`sigma` is fixed).

```r
plot(fit_svm_grbf)
```

We clearly see that the range of values `C` considered by default is too restrictive, and there might larger values of `C` corresponding to higher accuracy. The optimal SVM for that default range corresponds to the upper bound of the cost, which corresponds to a sub-optimum (we cannot know if the value of the accuracy is decreasing for a larger value of `C`), hence there is no guarantee that larger values of cost might not provide a better performance. The solution is to define a custom range of values of C, which will include also larger values of this hyperparameter. In addition, we might want to specify our own set of `sigma` values. We can create a grid of (`C`, `sigma`) values and perform the tuning over this grid by providing it in input to the argument `tuneGrid`.

```r
tune_grid <- expand.grid(C = c(1, 10, 50, 100, 200),
                         sigma = c(0.001, 0.005, 0.01, 0.05, 0.1))

fit_svm_grbf <- train(class ~ ., data = data,
                      method = "svmRadial",
                      trControl = train_ctrl,
                      tuneGrid = tune_grid)

fit_svm_grbf
plot(fit_svm_grbf)
```

As supposed, a larger cost corresponds to a better predictive performance. Similarly, a `sigma` value different from the default one corresponds to an improved performance. It is always good practice to check if any of the selected hyperparameters are at the boundaries of the defined grid, since this may be indication of a sub-optimal solution that does not guarantee that lower or larger values of the hyperparameters will not yield improved performance. If this happens, the easiest approach is to expand the range.

The output from `train` is a list with many slots including various information, among which the best selected model corresponding to the optimal tuning hyperparameters - **among those considered!**. The final model is the one corresponding to the optimal hyperparameter value(s) and is estimated on the full data (training+validation, see the help file `?train` and related documentation). We can directly use the function `predict` to obtain the predicted class labels on the test data from the final model. The function `confusionMatrix` can be employed to compute the confusion matrix and extract several predictive performance metrics; see `?confusionMatrix` for details. The argument `positive` is used to set the "positive" class, since the main interest is to diagnose the presence of diabetic retinopathy, we set it to `"1"`.

```r
# extract estimated class labels
class_hat <- predict(fit_svm_grbf, newdata = data_test)

# compute metrics
confusionMatrix(class_hat, data_test$class, positive = "1")
```

1.2 Task

- Take some time to explore the different metrics in output from `confusionMatrix`.

- Replicate this example tuning a SVM with polynomial kernel. In doing so, you could also consider different cross-validation settings.

1.3 Model tuning and comparison

The functionalities of the `caret` package can be readily employed to implement model tuning within a type of model and model comparison between different types of models. Here we compare logistic regression models with different subsets of features against SVMs with polynomial kernel.

We can use the same approach described in the previous section. However, we need to make sure that that the comparison between the two models is fair. To do so, the only minor modification is that before calling `train` we need to set the same random number seed via the function `set.seed`. This ensures that the same resampling sets are used in the cross-validation process, guaranteeing that the training and evaluation is performed on the same data sets for both models. In fact, not ensuring that the classifiers are trained and compared on the same splits of the data would invalid the comparison, since in this case it would be impossible to diagnose whether the difference in performance is due (entirely or partially) to the random difference between the splits or the actual difference in predictive performance between the classifiers.

The tuning procedure can be painfully slow, especially if we consider a large grid of hypeparameter values and a significant number of folds and replications. To speed up the computations, we can use the parallel computing functionalities of `caret`. By default, the function `trainControl` allows to run computations in parallel on multiple processors (see argument `allowParallel`). Hence, to run the computations in parallel we simply need to activate the parallel backend. To this purpose we used the package `doParallel`. The function `makeCluster` is used to set the clusters and the number of processors to be employed. Function `registerDoParallel` is used to activate the parallel backend. After we run the computations, we switch off the backend using the function `stopCluster`. We tune the SVM first.

```r
library(doParallel)
cl <- makeCluster(6)      # I have 8 cores on my machine (keep 2 free)
registerDoParallel(cl)    # start parallel computing backend

# just to implement three-fold, i.e. 33% for validation, 66% for training
train_ctrl <- trainControl(method = "repeatedcv", number = 3, repeats = 10)

# SVM with polynomial kernel
#
# set grid
tune_grid <- expand.grid(C = c(100, 200, 500),
                         scale = c(0.1, 0.25, 0.5),
                         degree = c(1, 2, 3, 4))
#
# NOTE : THIS WILL TAKE TIME TO RUN!
# change the cross-validation settings if required
set.seed(7719)   # change it to your favorite number
fit_svm_poly <- train(class ~ ., data = data,
                      method = "svmPoly",
                      trControl = train_ctrl,
                      tuneGrid = tune_grid)
plot(fit_svm_poly)
```

We now tune and compare the collection of logistic regression models. We consider three models:

- `lr1` - Logistic regression with all the input features.
- `lr2` - Logistic regression with `ma1`, `exudate1`, `macula_opticdisc_distance`, `opticdisc_diameter`. From inspection of the data the `ma` and the `exudate` features tend to be correlated, hence we consider one of each type, plus the optic disc features.
- `lr3` - Logistic regression with automatic backward feature selection using AIC. This is implemented using the method `glmStepAIC`. Different models are automatically compared in sequence using the AIC, whereby each variable is removed in turn.

```r
# logistic regression with all variables
set.seed(7719)
fit_lr1 <- train(class ~ ., data = data,
                method = "glm",
                family = "binomial",
                trControl = train_ctrl)

# logistic regression with some pre-selected variables
set.seed(7719)
fit_lr2 <- train(class ~ ma1 + exudate1 +
                   macula_opticdisc_distance + opticdisc_diameter,
                 data = data,
                 method = "glm",
                 family = "binomial",
                 trControl = train_ctrl)

# logistic regression with automatic variable selection
set.seed(7719)
fit_lr3 <- train(class ~ ., data = data,
                 method = "glmStepAIC",
                 family = "binomial",
                 direction = "backward",
                 trControl = train_ctrl)

# the variables selected by glmStepAIC can be inspected
# by looking at the slot `finalModel`
summary(fit_lr3$finalModel)


stopCluster(cl)    # turn off parallel computing -- don't forget to do that!
```

We can use the function `resamples` to easily compare the validation predictive performance of the models across the folds and replications. The SVM classifier performs slightly better than logistic regression `lr1` and `lr3`, although there is no significant difference between these models (as indicated by their overlapping boxes). They also have similar levels of variability.

```r
comp <- resamples(list(svm_poly = fit_svm_poly, lr1 = fit_lr1, lr2 = fit_lr2, lr3 = fit_lr3))
summary(comp)

# visual comparison
cols <- c("darkorange3", "purple3", "forestgreen", "deepskyblue3")
acc_values <- comp$values[,c(2,4,6,8)]
boxplot( acc_values, col = adjustcolor(cols, 0.4), border = cols)
points(1:4, colMeans(acc_values), pch = 15, cex = 1.5, col = cols)
```

In any case, the SVM with polynomial kernel performs better than logistic regression, so we employ it to predict the test data labels and assess its predictive performance.

```r
# extract estimated class labels
class_hat <- predict(fit_svm_poly, newdata = data_test)

# compute metrics
confusionMatrix(class_hat, data_test$class, positive = "1")
```

1.4 Task

- What are the optimal hyperparameters values of the SVM? What is the logistic regression model selected by `glmStepAIC`?
- Explore the output and the returned metrics. Function `resamples` also return the elapsed computing time, inspect it. What do you notice?
- Replicate this example by adding to the collection of models SVM with GRBF kernel and alternative logistic regression models with different terms (you could consider quadratic terms, interactions, implement forward selection, etc.). Try also different cross-validation procedures and settings.