ML CARET WAY - LAB

Enrique J. De La Hoz D.

Data Science - UTB

In-sample RMSE

In the R environment is included the diamonds dataset, which is a classic dataset from the ggplot2 package. The dataset contains physical attributes of diamonds as well as the price they sold for. One interesting modeling challenge is predicting diamond price based on their attributes using something like a linear regression.

Recall that to fit a linear regression, you use the lm() function in the following format:

```
mod <- lm(y ~ x, my_data)
```

To make predictions using mod on the original data, you call the predict() function:

```
pred <- predict(mod, my_data)
```

- Fit a linear model on the diamonds dataset predicting price using all other variables as predictors (i.e. price ~ .). Save the result to model.
- Make predictions using model on the full original dataset and save the result to p.
- Compute errors using the formula $errors = predicted - actual$. Save the result to error.
- Compute RMSE using the formula you learned previously and print it to the console. You can also use the function rmse() from the package metrics.

Cross-validation approaches

As better approach to validating models is to use multiple systematic test sets, rather than a single random train/test split. Fortunately, the caret package makes this very easy to do:

Caret supports many types of cross-validation, and you can specify which type of cross-validation and the number of cross-validation folds with the trainControl() function, which you pass to the trControl argument in train():

```
model <- train(
  y ~ ., my_data,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 10,
    verboseIter = TRUE
  )
)
```

It's important to note that you pass the method for modeling to the main train() function and the method for cross-validation to the trainControl() function.

- Fit a linear regression to model price using all other variables in the diamonds dataset as predictors. Use the train() function and 10-fold cross-validation.
- Print the model to the console and examine the results.

n x n-fold cross-validation

You can do more than just one iteration of cross-validation. Repeated cross-validation gives you a better estimate of the test-set error. You can also repeat the entire cross-validation procedure. This takes longer, but gives you many more out-of-sample datasets to look at and much more precise assessments of how well the model performs.

One of the awesome things about the train() function in caret is how easy it is to run very different models or methods of cross-validation just by tweaking a few simple arguments to the function call. For example, you could repeat your entire cross-validation procedure 5 times for greater confidence in your estimates of the model's out-of-sample accuracy, e.g.:

```
trControl = trainControl(
  method = "cv", number = 5,
  repeats = 5, verboseIter = TRUE
)
```

- Re-fit the linear regression model to the Diamonds dataset.
- Use 5 repeats of 5-fold cross-validation.
- Print the model to the console.
- Check the difference in the RMSE

Fitting and evaluating models' performance with Caret

Fit a logistic regression model

Once you have your random training and test sets you can fit a logistic regression model to your training set using the glm() function. glm() is a more advanced version of lm() that allows for more varied types of regression models, aside from plain vanilla ordinary least squares regression.

Be sure to pass the argument family = "binomial" to glm() to specify that you want to do logistic (rather than linear) regression. For example:

- Create a Train and test dataset with 60-40 proportion.
- Fit a logistic regression called model to predict Class using all other variables as predictors. Use the training set for Sonar.
- Predict on the test set using that model. Call the result p like you've done before.
- Use ifelse() to create a character vector, m_or_r that is the positive class, "M", when p is greater than 0.5, and the negative class, "R", otherwise.
- Convert m_or_r to be a factor, p_class, with levels the same as those of test[["Class"]].
- Make a confusion matrix with confusionMatrix(), passing p_class and the "Class" column from the test dataset.
- Use ifelse() to create a character vector, m_or_r that is the positive class, "M", when p is greater than 0.9, and the negative class, "R", otherwise.
- Convert m_or_r to be a factor, p_class, with levels the same as those of test[["Class"]].
- Make a confusion matrix with confusionMatrix(), passing p_class and the "Class" column from the test dataset.
- Use ifelse() to create a character vector, m_or_r that is the positive class, "M", when p is greater than 0.1, and the negative class, "R", otherwise.
- Convert m_or_r to be a factor, p_class, with levels the same as those of test[["Class"]].

- Make a confusion matrix with confusionMatrix(), passing p_class and the "Class" column from the test dataset.

Plot an ROC curve

An ROC curve is a really useful shortcut for summarizing the performance of a classifier over all possible thresholds. This saves you a lot of tedious work computing class predictions for many different thresholds and examining the confusion matrix for each.

One of the best package for computing ROC curves is caTools, which contains a function called colAUC(). This function is very user-friendly and can actually calculate ROC curves for multiple predictors at once. In this case, you only need to calculate the ROC curve for one predictor, e.g.:

```
colAUC(predicted_probabilities, actual, plotROC = TRUE)
```

The function will return a score called AUC (more on that later) and the plotROC = TRUE argument will return the plot of the ROC curve for visual inspection.

Using the objects model, test, and train from the last exercise using the sonar data:

- Predict probabilities (i.e. type = "response") on the test set, then store the result as p.
- Make an ROC curve using the predicted test set probabilities.

Customizing trainControl

The area under the ROC curve is a very useful, single-number summary of a model's ability to discriminate the positive from the negative class (e.g. mines from rocks). An AUC of 0.5 is no better than random guessing, an AUC of 1.0 is a perfectly predictive model, and an AUC of 0.0 is perfectly anti-predictive (which rarely happens).

This is often a much more useful metric than simply ranking models by their accuracy at a set threshold, as different models might require different calibration steps (looking at a confusion matrix at each step) to find the optimal classification threshold for that model.

You can use the trainControl() function in caret to use AUC (instead of acccuracy), to tune the parameters of your models. The twoClassSummary() convenience function allows you to do this easily.

When using twoClassSummary(), be sure to always include the argument classProbs = TRUE or your model will throw an error! (You cannot calculate AUC with just class predictions. You need to have class probabilities as well.)

- Customize the trainControl object to use twoClassSummary rather than defaultSummary.
- Use 10-fold cross-validation.
- Be sure to tell trainControl() to return class probabilities.

```
 # Create trainControl object: myControl
myControl <- trainControl(
  method = "cv",
  number = 10,
  summaryFunction = ___,
  classProbs = ___, # IMPORTANT!
  verboseIter = ___
)
```

Using custom trainControl

Now that you have a custom trainControl object, it's easy to fit caret models that use AUC rather than accuracy to tune and evaluate the model. You can just pass your custom trainControl object to the train() function via the trControl argument, e.g.:

```
train(<standard arguments here>, trControl = myControl)
```

This syntax gives you a convenient way to store a lot of custom modeling parameters and then use them across multiple different calls to train(). - Use train() to fit a glm model (i.e. method = "glm") to Sonar using your custom trainControl object, myControl. You want to predict Class from all other variables in the data (i.e. Class ~ .). Save the result to model.

- Print the model to the console and examine its output.

Applied case: Churn Rate

The churn dataset contains real world data on a variety of telecom customers and the modeling challenge is to predict which customers will cancel their service (or churn).

- Install the packages C50 and load the data churn.
- Use createFolds() to create 5 CV folds on churn_y, your target variable for this exercise.
- Pass them to trainControl() to create a reusable trainControl for comparing models.

Fit the baseline model

Now that you have a reusable trainControl object called myControl, you can start fitting different predictive models to your churn dataset and evaluate their predictive accuracy.

- Fit a glmnet model to the churn dataset called model_glmnet. Make sure to use myControl, which you created in the last exercise.

Fit a Random Forest model

- Fit a random forest model to the churn dataset. Be sure to use myControl as the trainControl like you've done before and implement the "ranger" method.

Create a resamples object

Now that you have fit two models to the churn dataset, it's time to compare their out-of-sample predictions and choose which one is the best model for your dataset.

You can compare models in caret using the resamples() function, provided they have the same training data and use the same trainControl object with preset cross-validation folds. resamples() takes as input a list of models and can be used to compare dozens of models at once (though in this case you are only comparing two models).

- Create a list() containing the glmnet model as item1 and the ranger model as item2.
- Pass this list to the resamples() function and save the resulting object as resamples.
- Summarize the results by calling summary() on resamples.

- Pass the resamples object to the bwplot() function to make a box-and-whisker plot. Look at the resulting plot and note which model has the higher median ROC statistic. Be sure to specify which metric you want to plot.
- Pass the resamples object to the xyplot() function. Look at the resulting plot and note how similar the two models' predictions are (or are not) on the different folds. Be sure to specify which metric you want to plot.
- **Optional**: Create an ensemble model and compare the results obtained.