mlr3 Learners: An Introduction

Natalie Foss

2022-09-19

Introduction

The purpose of this post is to introduce you to `mlr3` learners and give an example of how to train a model (showing the best practice methods), using the `palmer penguins` dataset. `palmer penguins` is a built in task described as "classification data to predict the species of penguins". Read more about the dataset here.

This blog post picks up from the previous post If you do not wish to read that post, this code chunk will get you up to speed (though we recommend you skim it).

```
library("mlr3")
task = tsk("penguins")
learner = lrn("classif.rpart")
```

Train and Test Split

As discussed in the last post, when training a model and evaluating, it is important to consider data leakage. Data leakage occurs when a model is tested on some of all of the same data it was trained on. It results in overly optimistic performance results. For this reason we subset the data into train and test splits.

```
# setting a seed will ensure your output matches the post output!
set.seed(3)
```

We will create two variables:

- splits$train - randomly samples 80% of the data points (rows) in the task (meaning it will hold the row indexes for 80% of the rows)
- splits$test - holds the remaining 20% of the row ids

`partition()` is a function that allows us to create balanced subsets of our data. we specify `cat_col` as "species" so that `partition()` balances the factors in each split. Meaning that the ratio of each species is the same in the train and test set. Read more about `partition()` here.

```
splits = partition(task, ratio = 0.8, cat_col="species")
print(str(splits))
```

```
## List of 2
##  $ train: int [1:275] 1 2 3 4 5 6 7 8 9 10 ...
##  $ test : int [1:69] 31 32 35 39 42 44 49 52 53 55 ...
## NULL
```

Now that we have data set aside for testing we can actually train the model.

Training the Model

```
learner$train(task, splits$train)
learner$model
```

```
## n= 275
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 275 153 Adelie (0.44363636 0.19636364 0.36000000)
##   2) flipper_length< 206.5 169  49 Adelie (0.71005917 0.28994083 0.00000000)
##     4) bill_length< 43 119   2 Adelie (0.98319328 0.01680672 0.00000000) *
##     5) bill_length>=43 50   3 Chinstrap (0.06000000 0.94000000 0.00000000) *
##   3) flipper_length>=206.5 106   7 Gentoo (0.01886792 0.04716981 0.93396226)
##     6) island=Dream,Torgersen 7   2 Chinstrap (0.28571429 0.71428571 0.00000000) *
##     7) island=Biscoe 99   0 Gentoo (0.00000000 0.00000000 1.00000000) *
```

Now we have this model (decision tree) that was trained on 80% of the data and we can now see it make some predictions on the test set.

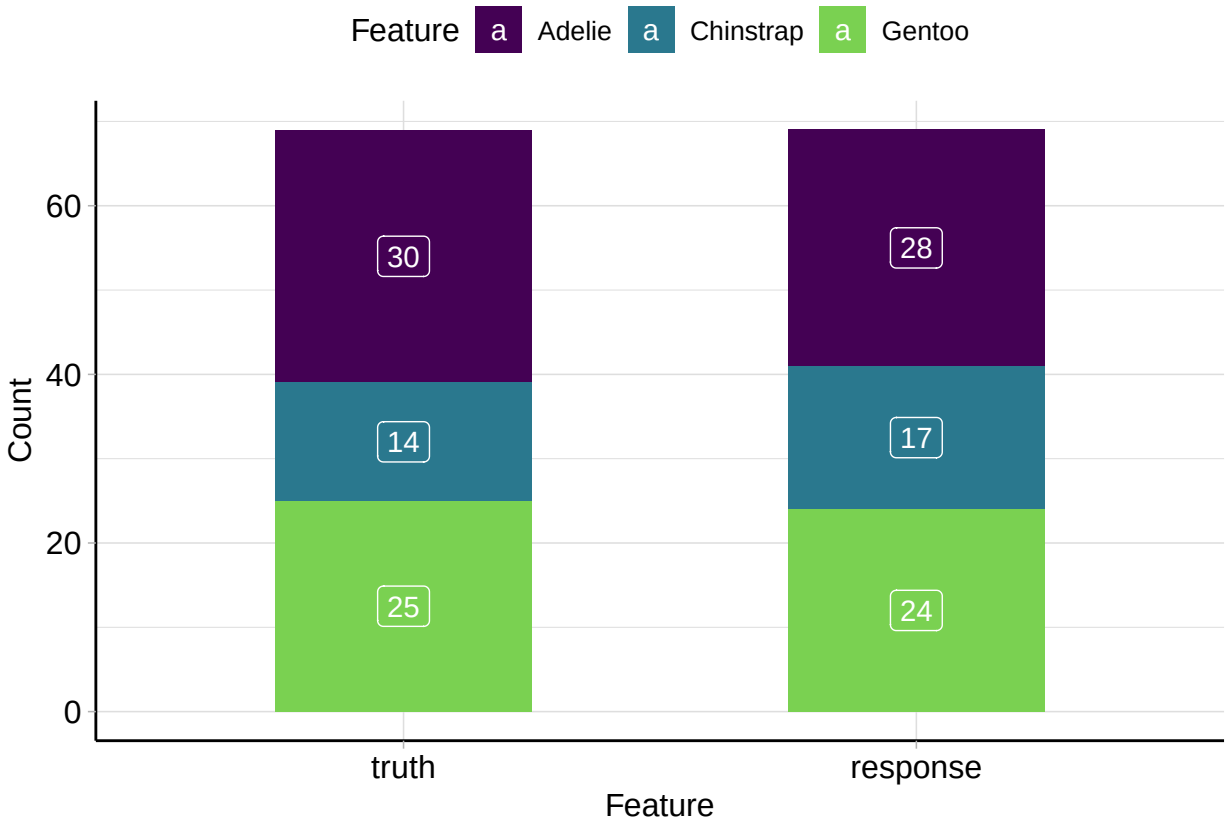
```
prediction = learner$predict(task, splits$test)
```

Model Visualization

We can visualize the predictions with `autoplot()`:

```
#install.packages("mlr3viz")
library("mlr3viz")

autoplot(prediction)
```



Model Performance

Since this is a classification problem we can get the confusion matrix which will give us an idea of how well the model performed. The confusion matrix shows how many data points were correctly and incorrectly classified. The diagonal values denote the correct classifications and all other values denote incorrect classifications. Read more about confusion matrices here.

```
prediction$confusion
```

```
##            truth
## response    Adelie Chinstrap Gentoo
##   Adelie        26         2      0
##   Chinstrap      4        12      1
##   Gentoo         0         0     24
```

While the confusion matrix shows the model's performance generally, we can quantify the predictive performance of our model by calling `prediction$score()`. There are several different scoring methods to choose from (filtering here by classification measures).

```
as.data.table(mlr_measures$keys("classif"))
```

```
##                       V1
##  1:         classif.acc
```

```
##  2:         classif.auc
##  3:        classif.bacc
##  4:      classif.bbrier
##  5:          classif.ce
##  6:       classif.costs
##  7:         classif.dor
##  8:       classif.fbeta
##  9:         classif.fdr
## 10:          classif.fn
## 11:         classif.fnr
## 12:        classif.fomr
## 13:          classif.fp
## 14:         classif.fpr
## 15:     classif.logloss
## 16:   classif.mauc_au1p
## 17:   classif.mauc_au1u
## 18:   classif.mauc_aunp
## 19:   classif.mauc_aunu
## 20:      classif.mbrier
## 21:         classif.mcc
## 22:         classif.npv
## 23:         classif.ppv
## 24:       classif.prauc
## 25:   classif.precision
## 26:      classif.recall
## 27: classif.sensitivity
## 28: classif.specificity
## 29:          classif.tn
## 30:         classif.tnr
## 31:          classif.tp
## 32:         classif.tpr
##                      V1
```

Note: to see other types of learners replace "classif" with one of: "clust", "dens", "regr", or "surv".

For this example we will use classification error, read more about this measure in the documentation.

```
measure = msr("classif.ce")
prediction$score(measure)
```

```
## classif.ce
##  0.1014493
```

Why Resampling is Important

Compare the performance of the previous model to this one, which is identically prepared except for a different seed for randomness.

```
set.seed(22)
splits = partition(task, ratio = 0.8, cat_col="species")
learner$train(task, splits$train)
prediction = learner$predict(task, splits$test)
prediction$score(measure)
```

```
## classif.ce
## 0.01449275
```

This is a massive difference in classification error. In the first model roughly 10% of the test set was incorrectly classified and in the second model only ~1.5% of the test set was incorrectly classified. Now we have no idea which result is an accurate representation of the model performance on this task. The solution to this problem is resampling.

Read about how to use resampling methods in `mlr3` in the next post.