Assessed Coursework #2: Performance & Profiling

2023-12-26

In this assessed coursework, I use profiling to identify inefficiencies in the `tune` function of the `e1071` package, particularly focusing on its application to tune support vector machines. The coursework primarily focuses on profiling and common R functions such as vectorisation and parallelisation to improve performance.

Support vector machines

Support vector machines (SVMs) are a classification technique that works to find the optimal decision boundary between classes. To do so, they find the decision boundary with the thickest error margin possible. In situations where perfect classification accuracy on the training data is not possible, as with most real-world datasets, SVMs solve the soft margin problem to find the decision boundary $f(\mathbf{x}; \mathbf{w}) = 0$ in order to minimise

$$\|\mathbf{w}\|^2 + \sum_i \epsilon_i$$

subject to the constraints that, for all i,

$$y_i f(\mathbf{x}_i; \mathbf{w}) + \epsilon_i \geq 1, \epsilon_i \geq 0$$

Finding a solution to this constrained optimisation problem is difficult for high-dimensional datasets, and these constraints are relatively complex. The primal SVM problem can be reformulated into the dual problem using Lagrange multipliers. The dual problem simplifies the optimization process and allows for the computation of the support vectors without explicitly finding the high-dimensional vector $\mathbf{w}$ in the primal space. The dual problem also enables the use of kernel functions, allowing SVMs to handle non-linear separations efficiently.

SVM hyperparameters

SVMs rely on the use of several hyperparameters, depending on the use of kernel function. All SVMs use the cost hyperparameter C, which controls the trade-off between minimising errors on the training data and controlling the complexity of the model. This hyperparameter $C > 0$ comes into the primal problem such that the decision boundary minimises

$$\|\mathbf{w}\|^2 + C \sum_i \epsilon_i$$

SVMs can also use a variety of kernel functions, which can introduce additional hyperparameters. For instance, if using a radial basis kernel function

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma |\mathbf{x}_i - \mathbf{x}_j|^2)$$

one must also specify the value of the bandwidth γ.

These hyperparameters should be tuned in order to find the values that yield the best performance. There are several methods for hyperparameter tuning, most notably cross-validation.

e1071 package

The `e1071` R package is commonly used for implementing SVMs in R. Here is an example application on a dataset of penguin species. We use an SVM with a linear kernel ($C = 1$) to predict the penguin species using data about the bill length, bill depth, flipper length, and body mass:

```
library(tidyverse)
library(e1071) # svm package
library(palmerpenguins) # data package
library(caret) # ML package
library(here) # filepath package

# Filter to 2 penguin species for binary classification
adelie_gentoo <- penguins %>%
  filter(species %in% c("Adelie", "Gentoo")) %>%
  mutate(species = droplevels(species)) %>%
  filter(!is.na(bill_length_mm))

# Create training / test split
set.seed(999)
index <- createDataPartition(adelie_gentoo$species, p = 0.7, list = FALSE)
# Training data
penguin_training <- adelie_gentoo[index,] %>%
  select(species,
         ends_with("_mm"),
         body_mass_g)
penguin_testing <- adelie_gentoo[-index,] %>%
  select(species,
         ends_with("_mm"),
         body_mass_g)

# Train a Support Vector Machine (SVM) model
penguin_svm_model <- svm(species ~ ., data = penguin_training, kernel = "linear", cost = 1)

# Make predictions on the test set
penguin_svm_predictions <- predict(penguin_svm_model, newdata = penguin_testing)

# Evaluate the model performance
table(penguin_svm_predictions, penguin_testing$species)
```

```
##
## penguin_svm_predictions Adelie Gentoo
##                  Adelie     45      0
##                  Gentoo      0     36
```

We achieve 100% classification accuracy on the penguin dataset.

Let's try on a larger dataset of abalone data. This example predicts sex of abalones using some variables pertaining to the dimension of their shells. We now use an RBF kernel ($C = 1, \gamma = 0.2$:

```
library(AppliedPredictiveModeling)
data("abalone")
```

```r
mf_abalone <- abalone %>%
  select(-Rings) %>%
  filter(Type %in% c("M", "F")) %>%
  mutate(Type = droplevels(Type))

# Create training / test split
set.seed(999)
index <- createDataPartition(mf_abalone$Type, p = 0.7, list = FALSE)
# Training data
abalone_training <- mf_abalone[index,]
abalone_testing <- mf_abalone[-index,]

# Train a Support Vector Machine (SVM) model
abalone_svm_model <- svm(Type ~ ., data = abalone_training, kernel = "radial", cost = 1, gamma = 0.2)

# Make predictions on the test set
abalone_svm_predictions <- predict(abalone_svm_model, newdata = abalone_testing)

# Evaluate the model performance
abalone_conf_matrix <- table(abalone_svm_predictions, abalone_testing$Type)
abalone_accuracy <- sum(diag(abalone_conf_matrix)) / sum(abalone_conf_matrix)
cat("Accuracy:", abalone_accuracy, "\n")
```

```
## Accuracy: 0.5364706
```

```r
abalone_conf_matrix
```

```
##
## abalone_svm_predictions   F   M
##                       F 111 113
##                       M 281 345
```

We have poorer performance on this larger dataset. Let's see if tuning the hyperparameters using cross-validation helps.

Tuning in `e1071`

The `e1071` package has a `tune` function that tunes hyperparameters using a grid search. We can supply several values for the hyperparameters C and γ and `tune` will yield the model with the hyperparameters that performed best, as measured by classification error:

```r
tictoc::tic()
abalone_tuned_svm <- tune(svm,
                          Type ~ .,
                          data = abalone_training,
                          kernel = "radial",
                          ranges = list(cost = c(0.5, 1, 1.5),
                                        gamma = c(0.02, 0.2, 2)),
                          tunecontrol = tune.control(sampling = "cross",
                                                     cross = 10))
tictoc::toc()
```

```
## 9.079 sec elapsed
```

Note that running `tune` on these 9 parameter combinations (3 values of C × 3 values of γ) was relatively slow. We will see why momentarily. We can now examine the parameters that gave the lowest classification error out of the combinations tried, and then test the best model on our testing data.

```
abalone_tuned_svm$best.parameters
```

```
##   cost gamma
## 9  1.5     2
```

```
# Make predictions on the test set
tuned_abalone_svm_predictions <- predict(abalone_tuned_svm$best.model, newdata = abalone_testing)

# Evaluate the model performance
tuned_abalone_conf_matrix <- table(tuned_abalone_svm_predictions, abalone_testing$Type)
tuned_abalone_accuracy <- sum(diag(tuned_abalone_conf_matrix)) / sum(tuned_abalone_conf_matrix)
cat("Accuracy:", tuned_abalone_accuracy, "\n")
```

```
## Accuracy: 0.54
```

```
tuned_abalone_conf_matrix
```

```
##
## tuned_abalone_svm_predictions   F   M
##                             F 158 157
##                             M 234 301
```

We have slightly improved accuracy on the new values of the hyperparameters, which use a higher bandwidth value, but performance is still poor. Trying more combinations of C and γ is incredibly slow and computationally intensive, however, because of the structure of the `tune` function.

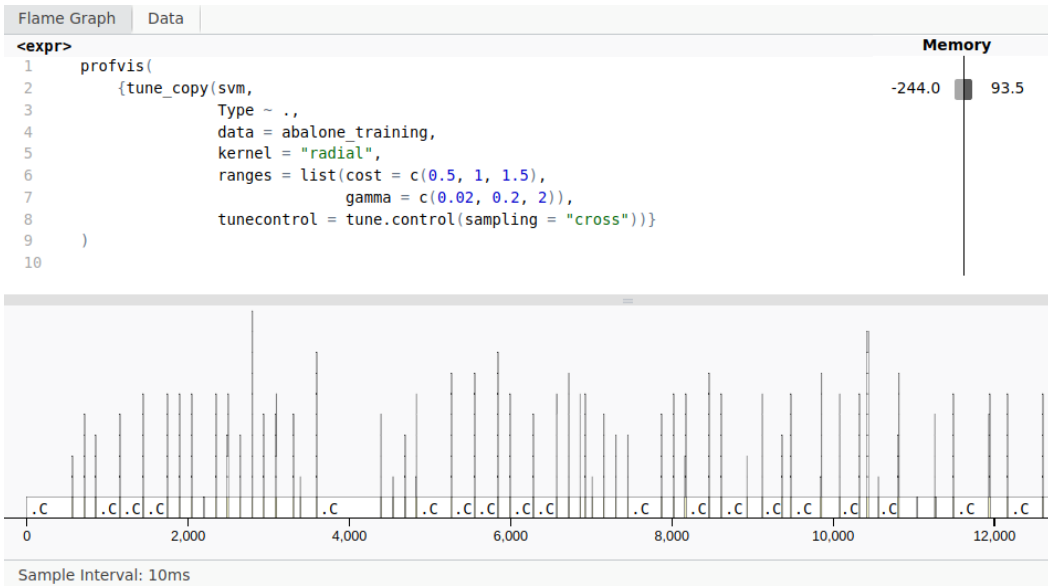
Improving `tune` performance

Profiling

We can use profiling to determine what is causing the slow performance of `tune`. I copied the source code for `tune` from here into `e1071_tune_copy.R` to explore what's going on within the function:

```
library(profvis)

profvis(
  {tune_copy(svm,
             Type ~ .,
             data = abalone_training,
             kernel = "radial",
             ranges = list(cost = c(0.5, 1, 1.5),
                           gamma = c(0.02, 0.2, 2)),
             tunecontrol = tune.control(sampling = "cross",
                                        cross = 10))}
)
```



The profiling results look like this:

Flame Graph | Data | Options ▾

Code	File	Memory (MB)		Time (ms)
.C		-312.0	460.4	14140
▾ tune_copy	<expr>	-244.0	93.5	730
▾ svm.formula		-132.5	80.8	530
▾ svm.default		-132.5	79.1	520
▸ data.frame		0	51.6	290
▸ predict.svm		-126.2	4.4	140
▸ scale_data_frame		0	17.4	50
t.default		-6.3	0	10
is.numeric		0	3.8	10
na.omit.default		0	0.1	10
▸ na.omit.data.frame		0	1.8	10
▸ predict.svm		0	2.6	70
[.data.frame		0	2.0	40
▸ model.frame.default		0	3.7	30
▸ resp		0	1.7	20
▸ table		0	1.8	20
tunecontrol$repeat.aggregate		0	0.9	10
▸ cbind		-111.5	0	10
is.pairlist		0	0.9	10
anyNA		0	2.1	10
.External2		0	0.5	10
as.integer		0	0.9	10

Sample Interval: 10ms 14910ms

Removing for-loops

By digging into the source code, we can see that `tune` contains 3 for loops nested within one another:

1. Looping over each parameter combination (9 in total in the example above)
2. Looping over each training sample (10 in the example since we performed 10-fold cross-validation)
3. Looping over the number of times training is repeated (1 in our example since `nrepeat = 1` by default)

We can eliminate the last loop by using the `replicate` function, which I have applied to create `tune_optimised`:

```
source(here("R/tune_optimized.R"))
tictoc::tic()
abalone_tuned_svm_optimised <- tune_optimised(svm,
                                              Type ~ .,
                                              data = abalone_training,
                                              kernel = "radial",
                                              ranges = list(cost = c(0.5, 1, 1.5),
                                                            gamma = c(0.02, 0.2, 2)),
                                              tunecontrol = tune.control(sampling = "cross",
                                                                         cross = 10))
tictoc::toc()
```

```
## 9.149 sec elapsed
```

The results are slightly faster, but not by much. We can examine the profiling:

```
profvis(
  {
    tune_optimised(svm,
                   Type ~ .,
                   data = abalone_training,
                   kernel = "radial",
                   ranges = list(cost = c(0.5, 1, 1.5),
                                 gamma = c(0.02, 0.2, 2)),
                   tunecontrol = tune.control(sampling = "cross"))
  }
)
```

Parallelization

```
library(foreach)
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
library(doParallel)
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
source(here("R/tune_parallel.R"))
tictoc::tic()
tune_parallel(svm,
```

```
                    Type ~ .,
                    data = abalone_training,
                    kernel = "radial",
                    ranges = list(cost = c(0.5, 1, 1.5),
                                  gamma = c(0.02, 0.2, 2)),
                    tunecontrol = tune.control(sampling = "cross"))
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1     2
##
## - best performance: 0.4317522
```

```
tictoc::toc()
```

```
## 1.839 sec elapsed
```