Efficient Machine Learning with R

Low-Compute Predictive Modeling with tidymodels

Simon P. Couch

Table of contents

I'm glad you're here.

Welcome to *Efficient Machine Learning with R*! This is a book about predictive modeling with tidymodels, focused on reducing the time and memory required to train machine learning models without sacrificing predictive performance.

- This book assumes familiarity with data analysis with the tidyverse as well as the basics of machine learning with tidymodels: fitting models with parsnip, resampling data with rsample, and tuning model parameters with tune. For more on tidy data analysis, see (Wickham, Çetinkaya-Rundel, and Grolemund 2023). For the basics of predictive modeling with tidymodels, see (Kuhn and Silge 2022).
- *Efficient Machine Learning with R* is staunchly empirical. While the scaling properties of many of the statistical and computational techniques described in this book are well-theorized, we'll observe that empirical timings often defy expectations based on that theory.

Outline:

- The **Introduction** demonstrates a 145-fold speedup with an applied example. By adapting a grid search on a canonical model to use a more performant modeling engine, hooking into a parallel computing framework, transitioning to an optimized search strategy, and defining the grid to search over carefully, the section shows that users can drastically cut down on tuning times without sacrificing predictive performance. The following chapters then explore those optimizations in further details.
- The **Models** chapter explore timings to resample various different modeling engines. The chapter compares implementations both within and across model types.
- **Parallelism** compares various approaches to distributing model computations across CPU cores. We'll explore two different across-model parallel computing frameworks supported by tidymodels—process forking and socket clusters—and explore their relationship to within-model parallelization.
- Then, **Search** explores various alternatives to grid search that can reduce the total number of model fits required to search a given grid space by only resampling models that seem to have a chance at being the "best."

- Finally, **Submodels** investigates approaches to designing grids that can further reduce the total number of model fits required to search a given grid space by generating predictions from one model that can be used to evaluate several.

The optimizations discussed in those aforementioned chapters can, on their own, substantially reduce the time to evaluate machine learning models with tidymodels. Depending on the problem context, some modeling workflows may benefit from more specialized optimizations. The following chapters discuss some of those use cases:

- **Preprocessing**
- **Sparsity**
- **GPU Training**
- **Stacking**

1 Introduction

1.1 Early value

To demonstrate the impact that a couple small pieces of intuition can have on execution time when evaluating machine learning models, I'll run through a quick model tuning example. On the first go, I'll lean on tidymodels' default values and a simple grid search, and on the second, I'll pull a few tricks out from my sleeves that will drastically reduce the time to evaluate models while only negligibly decreasing predictive performance.

1.1.1 Setup

First, loading a few needed packages:

```r
library(tidymodels)
library(future)
library(finetune)
library(bonsai)
```

For the purposes of this example, we'll simulate a data set of 100,000 rows and 18 columns. The first column, `class`, is a binary outcome, and the remaining variables are a mix of numerics and factors.

```r
set.seed(1)
d <- simulate_classification(1e5)
```

We'll first split the data into training and testing sets before generating a set of 10 folds from the training data for cross-validation.

```r
set.seed(1)
d_split <- initial_split(d)
d_train <- training(d_split)
d_test <- testing(d_split)
d_folds <- vfold_cv(d_train)
```

1.1.2 A first go

For my first go at tuning, I'll tune a boosted tree model using grid search. By default, tidymodels will use XGBoost as the modeling engine; I'll try out a few different values for `learn_rate`—a parameter that controls how drastically newly added trees impact predictions—and `trees`—the number of trees in the ensemble.

```r
bt <-
  boost_tree(learn_rate = tune(), trees = tune()) %>%
  set_mode("classification")
```

I'll carry out a grid search using `tune_grid()`, trying out a bunch of different pairs of values for `learn_rate` and `trees` and seeing what sticks. The argument `grid = 12` indicates that I want to try out 12 different combinations of values and will let tidymodels take care of exactly what those values are.

```r
set.seed(1)

bm_basic <-
  bench::mark(
    basic =
      tune_grid(
        object = bt,
        preprocessor = class ~ .,
        resamples = d_folds,
        grid = 12
      )
  )
```

`bench::mark()` returns, among other things, a precise timing of how long this process takes.

```r
bm_basic
```

```
# A tibble: 1 x 3
  expression   median mem_alloc
* <bch:expr> <bch:tm> <bch:byt>
1 basic         3.68h    5.43GB
```

Holy smokes! 3.68 hours is a good while. What all did `tune_grid()` do, though?

First, let's break down how many model fits actually happened. Since I've supplied `grid = 12`, we're evaluating 12 possible model configurations. Each of those model configurations is evaluated against `d_folds`, a 10-fold cross validation object, meaning that each configuration is fitted 10 times. That's 120 model fits!

Further, consider that those fits happen on 9/10ths of the training data, or 67500 rows.

With a couple small changes, though, the time to tune this model can be *drastically* decreased.

1.1.3 A speedy go

To cut down on the time to evaluate these models, I'll make a few small modifications.

First, I'll **evaluate in parallel**: Almost all modern laptops have more than one CPU core, and distributing computations across them only takes a couple lines of code with tidymodels.

```r
plan(multisession, workers = 4)
```

While this tuning process could benefit from distributing across many more cores than 4, I'll just use 4 here to give a realistic picture of the kinds of speedups possible on a typical laptop.

Then, we'll **use a clever grid**: The tidymodels framework enables something called the "submodel trick," a technique that will allow us to predict from many more models than we actually fit. Instead of just supplying `grid = 12`, I'll construct the grid myself.

```r
set.seed(1)
bt_grid <-
  bt %>%
  extract_parameter_set_dials() %>%
  grid_regular(levels = 4)
```

ℹ Note

To learn more about the submodel trick, see Chapter 3.

Next, I'll **switch out the computational engine**: Substituting XGBoost with another gradient-boosting model that can better handle some properties of this dataset will cut down on our fit time by a good bit.

```r
bt_lgb <- bt %>% set_engine("lightgbm")
```

Finally, I'll **give up early on poorly-performing models**: Rather than using grid search with `tune_grid()`, I'll use a technique called *racing* that stops evaluating models when they seem to be performing poorly using the `tune_race_anova()` function.

```r
set.seed(1)

bm_speedy <-
  bench::mark(
    speedy =
      tune_race_anova(
        object = bt_lgb,
        preprocessor = class ~ .,
        resamples = d_folds,
        grid = bt_grid
      )
  )
```

Checking out the new benchmarks:

```r
bm_speedy
```

```
# A tibble: 1 x 3
  expression   median mem_alloc
* <bch:expr> <bch:tm> <bch:byt>
1 speedy        1.52m    47.5MB
```

The total time to tune was reduced from 3.68 *hours* to 1.52 *minutes*—the second approach was 145 times faster than the first.

The first thing I'd wonder when seeing this result is how much of a penalty in predictive performance I'd suffer due to this transition. Let's evaluate both of the top models from these tuning results on the test set. First, for the basic workflow:

```r
fit_basic <-
  select_best(bm_basic$result[[1]], metric = "roc_auc") %>%
  finalize_workflow(workflow(class ~ ., bt), parameters = .) %>%
  last_fit(split = d_split)
```

```r
collect_metrics(fit_basic)
```

```
# A tibble: 3 x 4
  .metric     .estimator .estimate .config
  <chr>       <chr>          <dbl> <chr>
1 accuracy    binary         0.835 Preprocessor1_Model1
2 roc_auc     binary         0.896 Preprocessor1_Model1
3 brier_class binary         0.119 Preprocessor1_Model1
```

As for the quicker approach:

```r
fit_speedy <-
  select_best(bm_speedy$result[[1]], metric = "roc_auc") %>%
  finalize_workflow(workflow(class ~ ., bt), parameters = .) %>%
  last_fit(split = d_split)
```

```r
collect_metrics(fit_speedy)
```

```
# A tibble: 3 x 4
  .metric     .estimator .estimate .config
  <chr>       <chr>          <dbl> <chr>
1 accuracy    binary         0.835 Preprocessor1_Model1
2 roc_auc     binary         0.895 Preprocessor1_Model1
3 brier_class binary         0.120 Preprocessor1_Model1
```

Virtually indistinguishable performance results in 0.7% of the time.

1.2 Our approach

This book is intended for tidymodels users who have been waiting too long for their code to run. I generally assume that users are familiar with data manipulation and visualization with the tidyverse as well as the basics of machine learning with tidymodels, like evaluating models against resamples using performance metrics. For the former, I recommend (Wickham, Çetinkaya-Rundel, and Grolemund 2023) for getting up to speed—for the latter, (Kuhn and Silge 2022). If you're generally comfortable with the content in those books, you're ready to go.

Modern laptops are remarkable. Users of tidymodels working on many machines made in the last few years are well-prepared to interactively develop machine learning models based on tens of millions of rows of data. That said, without the right information, it's quite easy to mistakenly introduce performance issues that result in analyses on even tens of thousands of rows of data becoming too cumbersome to work with. Generally, the tidymodels framework attempts to guard users from making such mistakes and addressing them ourselves when they're in our control. At the same time, many foundational and well-used approaches in classical machine learning have well-theorized adaptations that substantially cut down on the elapsed time while preserving predictive performance. The tidymodels framework implements many such adaptations and this book aims to surface them in a holistic and coherent way. Readers will come out of having read this book with a grab bag of one-liners that can cut down on elapsed time to develop machine learning models by orders of magnitude.

1.3 The cost of slowness

All of this said, R is not known for its computational efficiency. If I really prioritize that, why am I writing a book about R?

(i use R for many reasons. you evidently do, too. it is true that many modeling engines only implement some performance optimizations / hardware accelators for their python interfaces—at the same time, modeling engines in R are often interfaces to the same compiled code as that used from python.)

1.4 The hard part

To better understand how to cut down on the time to evaluate models with tidymodels, we need to understand a bit about how tidymodels works.

Like many other "unifying frameworks" for ML (mlr3, caret, scikit-learn(?)), the tidymodels framework itself does not implement the algorithms to train and predict from models. Instead, tidymodels provides a common interface to modeling *engines*: packages (or functions from packages) that provide the methods to `fit()` and `predict()`.

The process of "translating" between the tidymodels and engine formats is illustrated in Figure 1.1. When fitting and predicting with tidymodels, some portion of the elapsed time to run code is due to the "translation" of the inputted unified code to the specific syntax that the engine expects, and some portion of it is due to the translation of what the engine returns to the unified output returned by tidymodels; these portions are in the tidymodels team's control. The rest of the elapsed time occurs inside of the modeling engine's code.

The portions of the elapsed time that are in the tidymodels team's control are shown in green, and I'll refer to them in this book as "overhead." The overhead of tidymodels in terms of elapsed time is relatively constant with respect to the size of training data. This overhead consists of tasks like checking data types, handling errors and warnings, and—most importantly—programmatically assembling calls to engine functions.

The portion of the elapsed time shown in orange represents the actual training of (or predicting from) the model. This portion is implemented by the modeling *engine* and is thus not in the tidymodels team's control. In contrast to overhead, the elapsed time of this code is very much sensitive to the size of the inputted data; depending on the engine, increases in the number of rows or columns of training or testing data may drastically increase the time to train or predict from a given model.

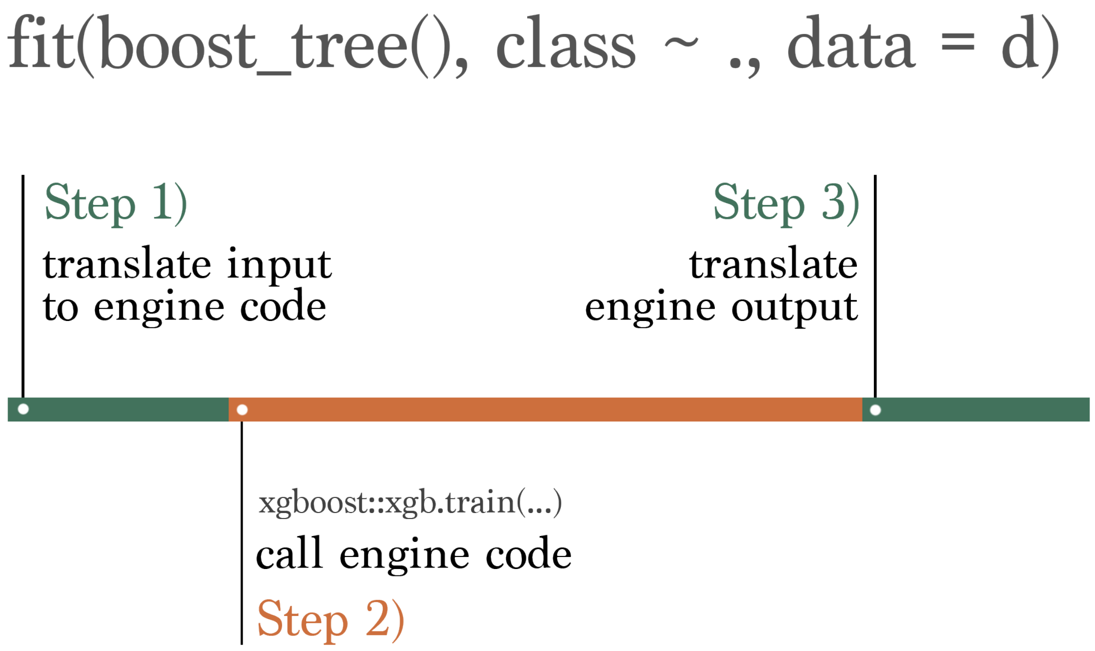

fit(boost_tree(), class ~ ., data = d)
Step 1)
translate input to engine code
Step 3)
translate engine output
xgboost::xgb.train(...)
call engine code
Step 2)

Figure 1.1

Note

The algorithmic complexity of the models implemented by these engines is well-understood in many cases. At the same time, the behavior of elapsed time for some engine implementations often differs greatly from what theory would lead one to believe. Regressions in modeling code may introduce undue slowdowns and, conversely, performance optimizations that lead to elapsed times that scale better than theory would suggest may be the very reason for the existence of some engines.

As shown in Figure 1.2, the proportion of elapsed time that overhead is responsible for depends on how quickly the engine can fit or predict for a given dataset.

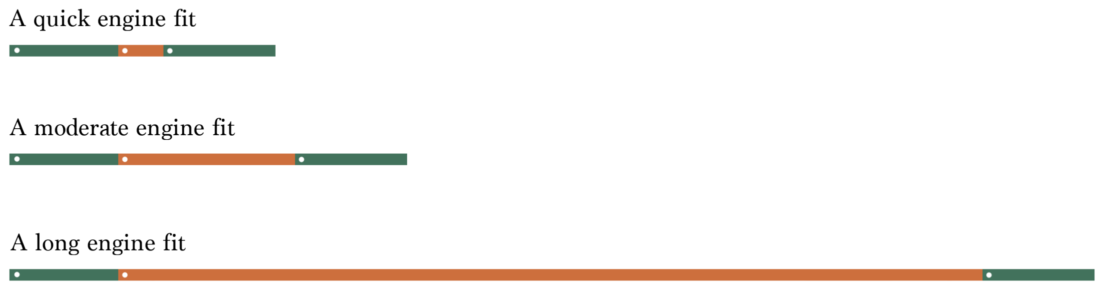


Figure 1.2

Since the absolute overhead of tidymodels' translation is relatively constant, overhead is only a substantial portion of elapsed time when models fit or predict *very* quickly. For a linear model fitted on 30 data points with `lm()`, this overhead is continuously benchmarked to remain under 2/3. That is, absolute worst-case, fitting a model with tidymodels takes three times longer than using the engine interface itself. However, this overhead approaches fractions of a percent for fits on even 10,000 rows for many engines. Thus, a focus on reducing the elapsed time of overhead is valuable in the sense that the framework ought not to unintentionally introduce regressions that cause overhead to scale with the size of training data, but in general, the hard part of reducing elapsed time when evaluating models is reducing the elapsed time for computations carried out by the modeling engine.

The next question is then *how could tidymodels cut down on elapsed time for modeling engines that it doesn't own?* To answer this question, let's revisit the applied example from Section 1.1.2. In that first example, the code does some translation to the engine's syntax, sets up some error handling, and then fits and predicts from 120 models.

Figure 1.3 depicts this process, where we evaluate all '120 models in order. Each white dot in the engine portion of the elapsed time represents another round of fitting and predicting

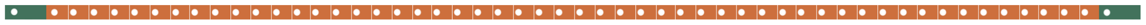


Figure 1.3

with engine. Remember that in reality, for even modest dataset sizes, the green portions representing tidymodels overhead are much smaller by proportion than represented.

In Section 1.1.3, the first thing I did was introduce a parallel backend. Distributing engine fits across available cores is itself a gamechanger, as illustrated in Figure 1.4.

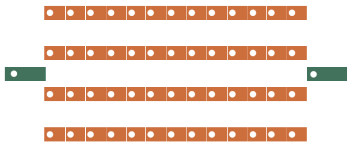
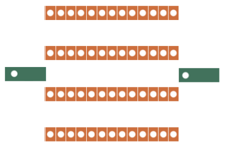
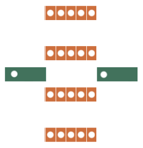


Figure 1.4

Then, switching out computational engines for a more performant alternative further reduces elapsed time, as shown in **?@fig-parallel-resample-opt**.



Finally, as depicted in **?@fig-parallel-resample-opt2**, the submodel trick described in Chapter 3 and racing described in **?@sec-search** eliminate a substantial portion of the engine fits.



The tidymodels team devotes substantial energy to ensuring support for the most performant parallelization technologies, modeling engines, model-specific optimizations, and search techniques. This book will demonstrate how to best make use of these features to reduce the time needed to evaluate machine learning models.

1.5 Datasets

In Section 1.1.1, I used a function `simulate_classification()` to generate data. This is one of two functions, the other being `simulate_regression()`, that create the data underlying many of the experiments in this book.

These two functions are adaptations of their similarly named friends `sim_classification()` and `sim_regression()` from the modeldata package. They make small changes to those function—namely, introducing factor predictors with some tricky distributions—that surface slowdowns with some modeling engines.

Provided a number of rows, `sim_classification()` generates a tibble with that many rows and 16 columns:

```r
d_class <- simulate_classification(1000)

d_class
```

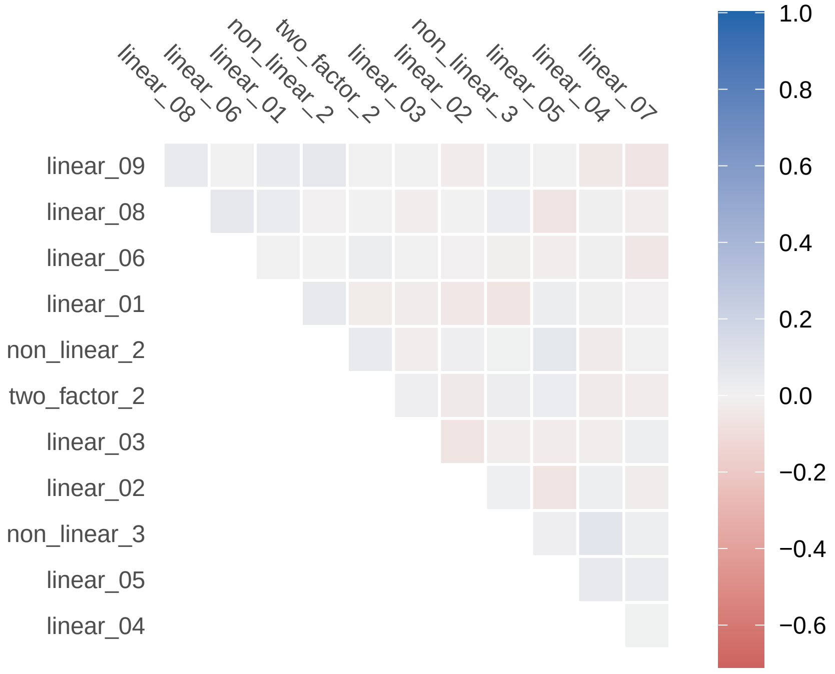
```
# A tibble: 1,000 x 18
   class   two_factor_1 two_factor_2 non_linear_1 non_linear_2 non_linear_3
   <fct>   <fct>               <dbl> <fct>               <dbl>        <dbl>
 1 class_1 level_2          -0.133   level_3            0.0355       0.770
 2 class_1 level_2           0.894   level_3            0.356        0.690
 3 class_2 level_2          -1.59    level_4            0.249        0.650
 4 class_1 level_1           2.17    level_3            0.879        0.0747
 5 class_1 level_2           0.464   level_1            0.318        0.903
 6 class_2 level_2          -2.04    level_3            0.321        0.133
 7 class_2 level_2           1.11    level_3            0.848        0.211
 8 class_2 level_1          -0.183   level_3            0.381        0.155
 9 class_2 level_1           0.00202 level_3            0.275        0.0545
10 class_2 level_2           0.198   level_3            0.918        0.715
# i 990 more rows
# i 12 more variables: linear_01 <dbl>, linear_02 <dbl>, linear_03 <dbl>,
#   linear_04 <dbl>, linear_05 <dbl>, linear_06 <dbl>, linear_07 <dbl>,
#   linear_08 <dbl>, linear_09 <dbl>, linear_10 <fct>, linear_11 <fct>,
#   linear_12 <fct>
```

The leftmost column, `class`, is a binary outcome variable, and the remaining columns are predictor variables. The predictors throw a few curveballs at the modeling functions we'll benchmark in this book.

For one, the predictors are moderately correlated. Correlated predictors can lead to unstable parameter estimates and can make the model more sensitive to small changes in the data.

Further, correlated predictors may lead to slower convergence in gradient descent processes (like those driving gradient-boosted trees like XGBoost and LightGBM), as the resulting elongated and narrow surface of loss functions causes the algorithm to zigzag towards the optimum value, significantly increasing the training time along the way.

```
Non-numeric variables removed from input: `class`, `two_factor_1`, `non_linear_1`, `linear_1
Correlation computed with
* Method: 'pearson'
* Missing treated using: 'pairwise.complete.obs'
```



Secondly, there are a number of factor predictors. Some modeling engines experience slower training times with many factor predictors for a variety of reasons. For one, most modeling engines ultimately implement training routines on numeric matrices, requiring that factor predictors are somehow encoded as numbers. Most often in R, this is in the form of treatment contrasts, where an n-length factor with l levels is represented as an $n\ x\ l-1$ matrix composed of zeroes and ones. Each column is referred to as a dummy variable. The first column has value 1 when the i-th entry of the factor is the second level, zero otherwise. The second column has value 1 when the i-th entry of the factor is the third level, zero otherwise. We know that the i-th entry of the first took its first level if all of the entries in the ith column of the resulting matrix are zero. While this representation of a factor is relatively straightforward, it's quite memory intensive; a factor with 100 levels ultimately will require a 99-column matrix to be allocated in order to be included in a model. While many modeling engines in R assume that factors will be encoded as treatment constrasts, different modeling engines have different approaches to processing factor variables, some more efficient than others. More on this in Chapter 4, in particular.

Original factor

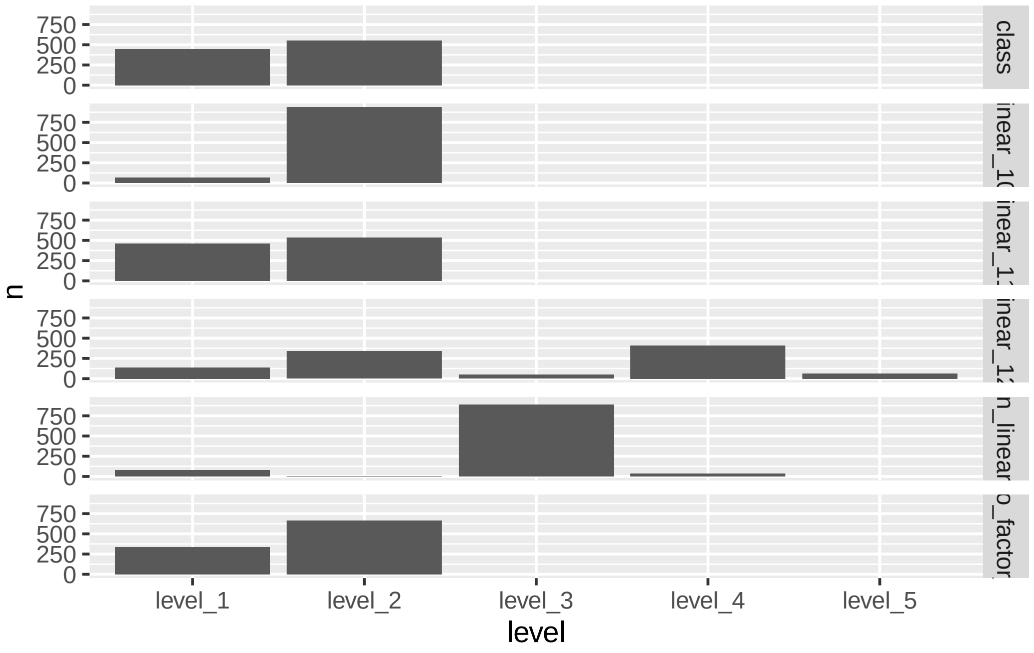
```
  x
1 a
2 b
3 c
```

Factor with treatment contrasts

```
  xb xc
1  0  0
2  1  0
3  0  1
```

Further, many of those factor variables have a class imbalance; that is, some levels of the factor occur much more often than others. Some models may struggle to learn from the less frequently-occurring classes, potentially requiring more iterations of descent processes for some models to converge. Even when this is not the case, it may be varyingly "worth it" in terms of memory usage to allocate a dummy variable to a factor level that only appears a couple times in a dataset with many rows.

```
Warning: Removed 13 rows containing missing values or values outside the scale range
(`geom_col()`).
```



The regression dataset looks quite similar.

```r
d_reg <- simulate_regression(1000)

d_reg
```

```
# A tibble: 1,000 x 16
   outcome predictor_01 predictor_02 predictor_03 predictor_04 predictor_05
     <dbl>        <dbl>        <dbl>        <dbl>        <dbl>        <dbl>
 1    6.36        -2.45        2.46          2.85        2.16         -1.28
 2   -2.11        -4.08        1.64         -3.49        3.21         -3.18
 3   26.0         -4.95        1.77         -2.21        3.35          5.29
 4   -1.31        -3.36       -2.95         -4.05       -0.911         5.88
 5   53.8          0.604      -6.35         -4.89       -5.96         -7.86
 6  -32.1          3.53       -6.56          2.06       -0.965        -2.11
 7   24.8          2.59        1.68         -1.88        3.10         -1.35
 8   11.7          3.79       -1.46         -2.23       -0.467        -0.662
 9    7.19         4.88        1.08          4.17        3.01         -4.15
10    9.43         2.78       -0.441         1.62        0.452         1.74
# i 990 more rows
# i 10 more variables: predictor_06 <dbl>, predictor_07 <dbl>,
#   predictor_08 <dbl>, predictor_09 <dbl>, predictor_10 <fct>,
#   predictor_11 <fct>, predictor_12 <fct>, predictor_13 <fct>,
#   predictor_14 <fct>, predictor_15 <fct>
```

The left-most column, `outcome`, is a numeric outcome, and the remaining 15 columns are a mix of numeric and factor. The same story related to correlation and tricky factor imbalances goes for the regression dataset. Demonstrating that is homework.

2 Parallel computing

...

Above all, my question is: how can I reduce the time to train machine learning models by utilizing the handful of CPU cores (and maybe even the snazzy GPU) on my laptop?

2.1 Across-models

In Chapter 1, one of the changes I made to greatly speed up that resampling process was to introduce an across-models parallel backend. By "across-models," I mean that each individual model fit happens on a single CPU core, but I allot each of the CPU cores I've reserved for training a number of model fits to take care of.

Phrased another way, in the case of "sequential" training, all 120 model fits happened one after the other.

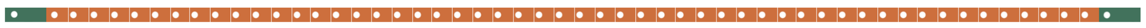


Figure 2.1

While the one CPU core running my R process works itself to the bone, the other remaining 9 are mostly sitting idle (besides keeping my many browser tabs whirring). CPU parallelism is about somehow making use of more cores than the one my main R process is running on; in theory, if n times as many cores are working on fitting models, the whole process could take $1/n$ of the time.

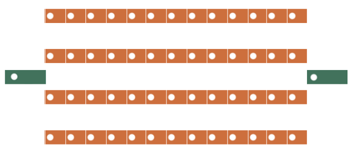


Figure 2.2

Why do I say “in theory”? The orchestration of splitting up that work is actually a very, very difficult problem, for two main reasons:

- **Getting data from one place to another**: Each of us has likely spent minutes or even hours waiting for a call to `load()` on a big old `.RData` object to complete. In that situation, data is being brought in from our hard disk or a remote database (or even elsewhere in memory) into memory allocated by our R process. R is a single-process program, so in order to train multiple models simultaneously, we need multiple R processes, each able to access the training data. We then have two options: one would be to somehow allow each of those R processes to share one copy of the data (this is the idea behind forking, described in Section 2.1.2), the other to send a copy of the data to each process (the idea behind socket clusters, described in Section 2.1.3). The former *sounds* nice but can become a headache quite quickly. The latter *sounds* computationally expensive but, with enough memory and sufficiently low latency in copying data (as would be the case with a set of clusters living on one laptop), can often outperform forking for local workflows.
- **Load balancing**: Imagine I have some machine learning model with a hyperparameter p, and that the computational complexity of that model is such that, with hyperparameter value p, the model takes p minutes to train. I am trying out values of p in $1, 2, 3, ..., 40$ and distributing model training across 5 cores. Without the knowledge of how p affects training times, I might send models with p in $1, 2, 3, ...8$ off to the first core, $9, 10, 11, ..., 16$ off to the second core, and so on. In this case, the first core would finish up all of its fits in a little over half an hour while the last would take almost 5 hours. In this example, I’ve taken the penalty on overhead of sending all of the training data off to each core, but in the end, one core ends up doing the majority of the work anyway. In this case, too, we were lucky that the computational complexity of model fits relative to this parameter were roughly linear—it’s not uncommon for model fit times to have a quadratic or geometric relationship with the values of important hyperparameters. A critical reader might have two questions. The first: if the computational complexity relative to this parameter is known, why don’t you just batch the values of p up such that each worker will take approximately the same amount of time? This is a reasonable question, and it relates to the hard problem of *chunking*. In some situations, related to individual parameters, it really is just about this simple to determine the relationship between parameter values and fit times. In reality, those relationships tend not to be quite so clear-cut, and even when they are, the implications of that parameter value for fit times often depend on the values of other parameters; a pairing of some parameter value p with some other value of a different parameter q might cause instability in some gradient descent process or otherwise, making the problem of estimating the fit time of a model given some set of parameter values a pretty difficult problem for some model types. The second question: couldn’t you just send each of the cores a single parameter value and have them let the parent R process know they’re done, at which point they’ll receive another parameter value to get to work on evaluating? That way, the workers

that happen to end up with a quicker-fitting values earlier on won't sit idle waiting for other cores to finish. This approach is called *asynchronous* (or "async") and, in some situations, can be quite helpful. Remember, though, that this requires getting data (in the form of the communication that a given worker is done evaluating a model, and maybe passing along some performance metric values) back and forth much more often. If the overhead of that communication exceeds the time that synchronous workers had spent idle, waiting for busier cores to finish running, then the asynchronous approach will result in a net slowdown.

There are two dominant approaches to distributing model fits across local cores that I've hinted at already: forking and socket clusters. We'll delve further into the weeds of each of those approaches in the coming subsections. At a high level, though, the folk knowledge is that forking is subject to less overhead in sending data back and forth, but has some quirks that make it less portable (more plainly, it's not available on Windows) and a bit unstable thanks to a less-than-friendly relationship with R's garbage collector. As for load balancing, the choice between these two parallelism techniques isn't really relevant.

Before I spend time experimenting with these techniques, I want to quickly situate the terminology I'm using here in the greater context of discussions of parallel computing with R. "Sequential," "forking," and "socket clusters" are my preferred terms for the techniques I'll now write about, but there's quite a bit of diversity in the terminology folks use to refer to them. I've also called out keywords (as in, functions or packages) related to these techniques in various generations of parallel computing frameworks in R. In "base," I refer to functions in the parallel package, building on popular packages multicore (first on CRAN in 2009, inspiring `mclapply()`) and snow (first on CRAN in 2003, inspiring `parLapply()`) and included in base installations of R from 2011 onward (R Core Team 2024).

Technique	future	foreach	base	Synonyms
Sequential	`sequential()`			Serial
Forking	`multicore()`	doMC	`mclapply()`	Process forking
Socket Clusters	`multisession()`	doParallel	`parLapply()`	Parallel Socket Clusters (PSOCK), Cluster, Socket

2.1.1 Sequential

Generally, in this chapter, I'm writing about various approaches to parallel computing. I'll compare each of those approaches to each, but also to the sequential (or "not parallel") approach. Sequential evaluation means evaluating model fits *in sequence*, or one after the other.

To demonstrate the impacts of different parallelism approaches throughout this chapter, we'll always start the conversation with a short experiment. I'll define a function that tracks the elapsed time to resample a boosted tree ensemble against simulated data, given a number of rows to simulate and a parallelism approach.

```r
time_resample_bt <- function(n_rows, plan) {
  # simulate data with n_rows rows
  set.seed(1)
  d <- simulate_regression(n_rows)

  # set up a parallelism plan
  # set `workers = 4`, which will be ignored for `plan = "sequential"`.
  if (plan == "multicore") {
    rlang::local_options(parallelly.fork.enable = TRUE)
  }
  if (plan == "multisession") {
    rlang::local_options(future.globals.maxSize = 1024*1024^2) # 1gb
  }
  suppressWarnings(
    plan(plan, workers = 4)
  )

  # track the elapsed time to...
  bench::mark(
    resample =
      fit_resamples(
        # ...evaluate a boosted tree ensemble...
        boost_tree("regression"),
        # ...modeling the outcome using all predictors...
        outcome ~ .,
        # ...against a 10-fold cross-validation of `d`.
        vfold_cv(d, v = 10)
      ),
    memory = FALSE
  )
}
```

Here's a quick example, simulating 100 rows of data and evaluating its resamples sequentially:

```r
t_seq <- time_resample_bt(100, "sequential")

t_seq
```

```
t_seq
```

```
# A tibble: 1 x 3
  expression   median mem_alloc
* <bch:expr> <bch:tm> <bch:byt>
1 resample      201ms    2.19MB
```

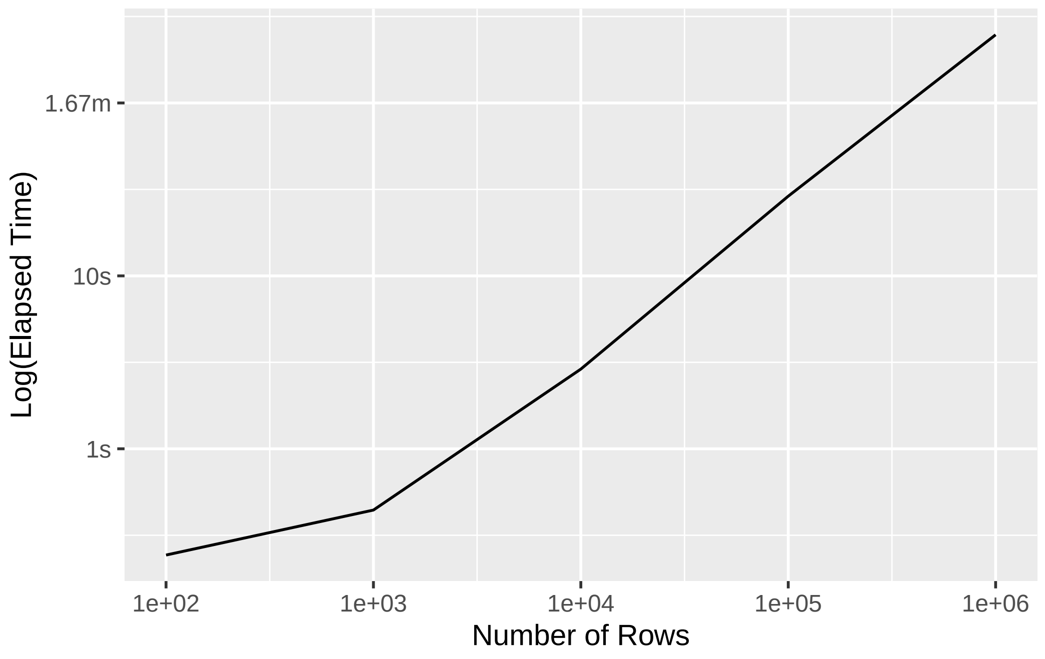
In total, the whole process took 0.2 seconds on my laptop. This expression, among other things, fits a model for each resample on $n * \frac{v-1}{v} = 100 * \frac{9}{10} = 90$ rows, meaning that even if the model fits took up 100% of the evaluation time in total, they take 0.02 seconds each. In other words, these fits are quite fast. As such, the overhead of distributing computations across cores would have to be quite minimal in order to see speedups with computations done in parallel. Scaling up the number of rows in the training data, though, results in elapsed times becoming a bit more cumbersome; in the following code, we'll resample models on datasets with 100 to a million rows, keeping track of the elapsed time for each iteration.

```
press_seq <-
  bench::press(
    time_resample_bt(n_rows, "sequential"),
    n_rows = 10^(2:6)
  )
```

For now, the graph we can put together with this data isn't super interesting:

```
ggplot(press_seq) +
  aes(x = n_rows, y = median) +
  scale_x_log10() +
  geom_line() +
  labs(y = "Log(Elapsed Time)", x = "Number of Rows")
```

```
Warning: The `trans` argument of `continuous_scale()` is deprecated as of ggplot2 3.5.0.
i Please use the `transform` argument instead.
```



More rows means a longer fit time—what a thrill! In the following sections, we'll compare this timing to those resulting from different parallelism approaches.

2.1.2 Forking

Process forking is a mechanism where an R process creates an exact copy of itself, called a "child" process (or "worker.") Initially, workers share memory with the original ("parent") process, meaning that there's no overhead resulting from creating multiple copies of training data to send out to workers. So, in the case of tidymodels, each worker needs to have some modeling packages loaded and some training data available to get started on evaluating a model workflow against resample; those packages and data are already available in the parent process, so tidymodels should see very little overhead in shipping data off to workers with forking.

There are a few notable drawbacks of process forking:

- There's no direct way to "fork" a process from one machine to another, so forking is available only on a single machine. To distribute computations across multiple machines, practitioners will need to make use of socket clusters (described in the following section Section 2.1.3).
- Forking is based on the operating system command `fork`, available only on Unix-alikes (i.e. macOS and Linux). Windows users are out of luck.

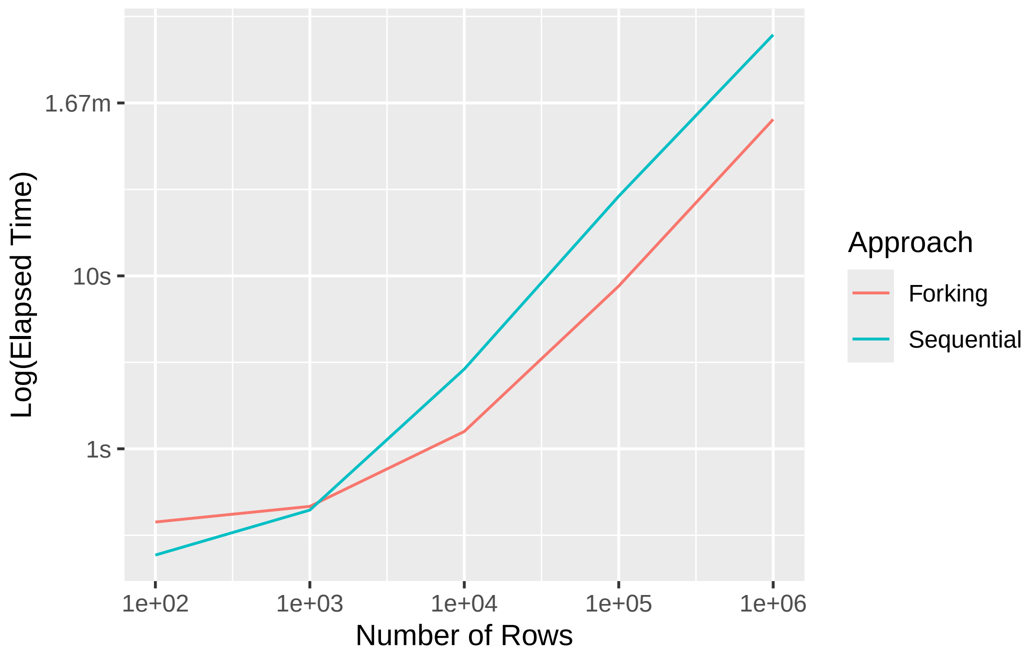
- In practice, memory that is initially shared often ends up ultimately copied due to R's garbage collection. "[I]f the garbage collector starts running in one of the forked [workers], or the [parent] process, then that originally shared memory can no longer be shared and the operating system starts copying memory blocks into each [worker]. Since the garbage collector runs whenever it wants to, there is no simple way to avoid this" (Bengtsson 2021).

Let's rerun that experiment from the previous section using forking and compare timings.

```r
press_fork <-
  bench::press(
    time_resample_bt(n_rows, "multicore"),
    n_rows = 10^(2:6)
  )
```

We can make the plot from the last section a bit more interesting now.

```r
bind_rows(
  mutate(press_seq, Approach = "Sequential"),
  mutate(press_fork, Approach = "Forking")
) %>%
  ggplot() +
  aes(x = n_rows, y = median, col = Approach) +
  scale_x_log10() +
  geom_line() +
  labs(y = "Log(Elapsed Time)", x = "Number of Rows")
```

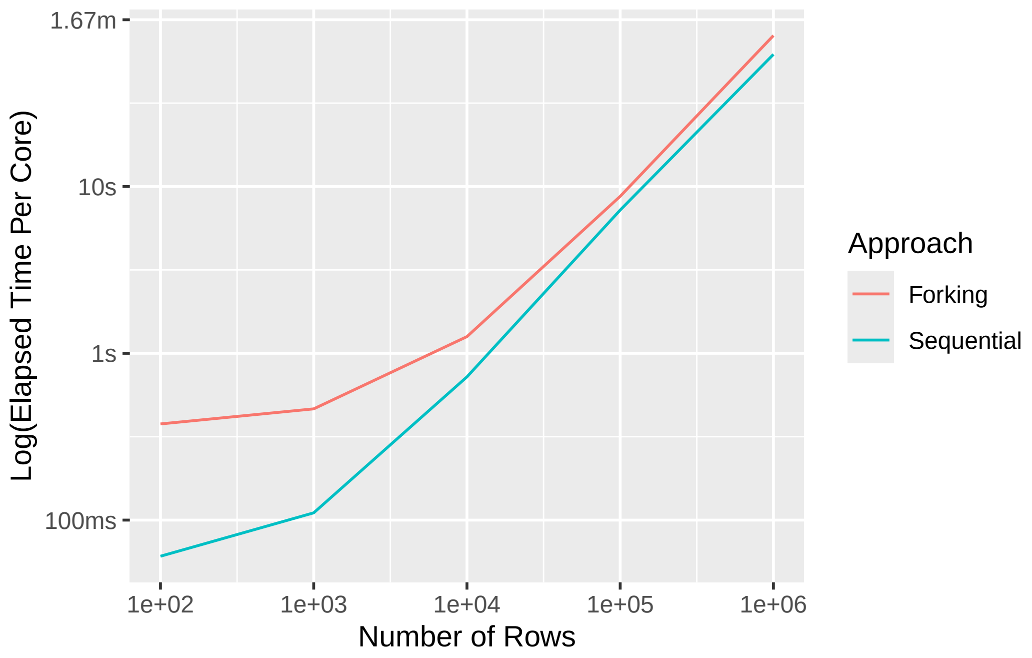


Well, if that ain't by the book! For the smallest training dataset, `n = 100`, distributing computations across cores resulted in a new slowdown. Very quickly, though, forking meets up with the sequential approach in elapsed time, and by the time we've made it to more realistic dataset sizes, forking almost always wins. In case the log scale is tripping you up, here are the raw timings for the largest dataset:

```
# A tibble: 2 x 3
    median Approach    n_rows
* <bch:tm> <chr>        <dbl>
1    4.13m Sequential 1000000
2    1.34m Forking    1000000
```

Another angle from which to poke at this is *how much time did we lose to overhead*? Said another way, if I distributed these computations across 4 cores, then I should see a 4-fold speedup in a perfect world. If I divide the time it takes to resample this model sequentially by 4, how does it compare to the timing I observe with forking?

```
bind_rows(
  mutate(press_seq, Approach = "Sequential"),
  mutate(press_fork, Approach = "Forking")
) %>%
  mutate(median_adj = case_when(
    Approach == "Sequential" ~ median / 4,
    .default = median
  )) %>%
  ggplot() +
  aes(x = n_rows, y = median_adj, col = Approach) +
  scale_x_log10() +
  geom_line() +
  labs(y = "Log(Elapsed Time Per Core)", x = "Number of Rows")
```



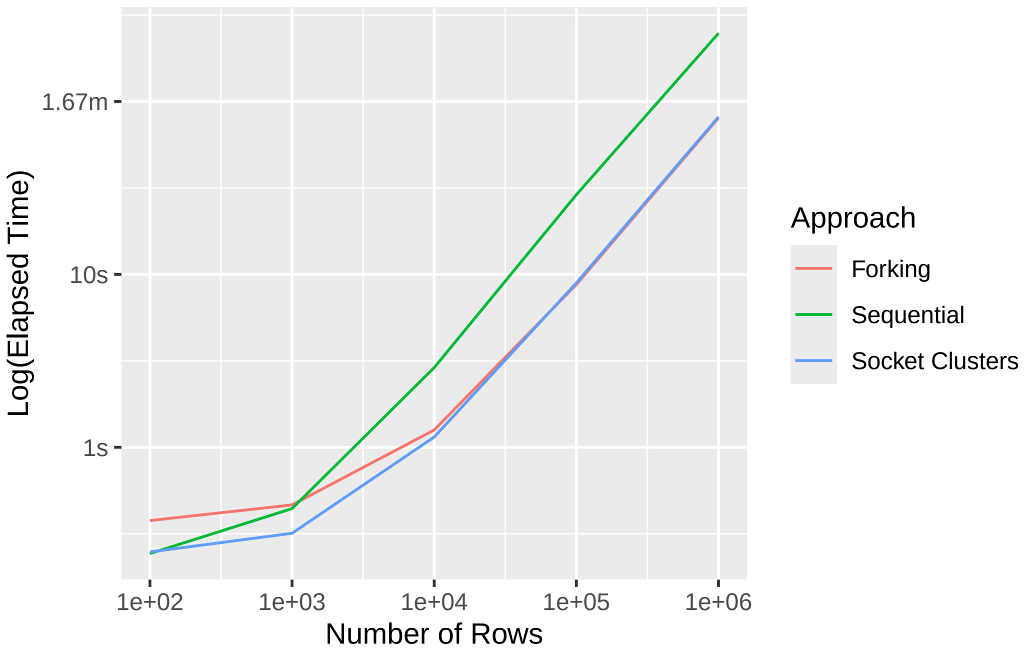
Per core, no parallel approach will ever be faster than sequential. (If it is, there's a performance bug in your code!) As the computations that happen in workers take longer and longer, though, the overhead shrinks as a proportion of the total elapsed time.

2.1.3 Socket Clusters

Let's see how this plays out in practice. Returning to the same experiment from before:

```
press_sc <-
  bench::press(
    time_resample_bt(n_rows, "multisession"),
    n_rows = 10^(2:6)
  )
```

```
bind_rows(
  mutate(press_seq, Approach = "Sequential"),
  mutate(press_fork, Approach = "Forking"),
  mutate(press_sc, Approach = "Socket Clusters")
) %>%
  ggplot() +
  aes(x = n_rows, y = median, col = Approach) +
  scale_x_log10() +
  geom_line() +
  labs(y = "Log(Elapsed Time)", x = "Number of Rows")
```



2.2 Within-models

I've only written so far about across-model parallelism, where multiple CPU cores are used to train a set of models but individual model fits happen on a single core. For most machine learning models available via tidymodels, this is the only type of parallelism possible. However, some of the most well-used modeling engines in tidymodels, like XGBoost and LightGBM, allow for distributing the computations to train a *single* model across several CPU cores (or even on GPUs). This begs the question, then, of whether tidymodels users should always stick to across-model parallelism, to within-model parallelism for the models it's available for, or to a hybrid of both. We'll focus first on the former two options in this subsection and then explore their interactions in Section 2.3.

2.2.1 CPU

2.2.2 GPU

XGBoost parallelizes at a finer level (individual trees and split finding), while LightGBM parallelizes at a coarser level (features and data subsets).

XGBoost

- arg `device` is `cpu` or `cuda` (or `gpu`, but `cuda` is the only supported `device`).
- arg `nthread` is integer
- uses openMP for cpu (?)
- `gpu_hist` is apparently pretty ripping?

LightGBM

- `device_type` is `cpu`, `gpu`, or `cuda` (fastest, but requires GPUs supporting CUDA)
- `cuda` is fastest but only available on Linux with NVIDIA GPUs with compute capability 6.0+
- `gpu` is based on OpenCL… M1 Pro is a "built-in" / "integrated"
- `cpu` uses OpenMP for CPU parallelism
- set to real CPU cores (i.e. not threads)
- refer to Installation Guide to build LightGBM with GPU or CUDA support
- are `num_threads` is integer
- Dask available to Python users

aorsf

- `n_thread` implements OpenMP CPU threading

keras (MLP)

- CPU or GPU
- cuda (available only for jax backend?)
- also available with tensorflow

h2o stuff

baguette

- `control_bag(allow_parallel)`? does this respect nested parallelism?

While GPU support is available for both libraries in R, it's not as straightforward to use as in Python. The R ecosystem generally has less robust GPU support compared to Python, which can make GPU-accelerated machine learning in R more challenging to set up and use.

2.3 Within and Across

2.4 Strategy

Choosing n cores and `parallel_over`…

2.5 Distributed Computing

So far in this chapter, I've mostly focused on the performance considerations for distributing computations across cores on a single computer. Distributed computing "in the cloud," where data is shipped off for processing on several different computers, is a different ball-game. That conversation is mostly outside of the scope of this book, but I do want to give some high-level intuition on how local parallelism differs from distributed parallelism.

The idea that will get you the most mileage in reasoning about distributed parallelism is this: the overhead of sending data back and forth is much more substantial in distributed computing than it is in the local context.

In order for me to benefit from distributing these computations across cores, the overhead of sending data out to workers has to be so minimal that it doesn't overtake the time saved in distributing model fits across cores. Let's see how this plays out on my laptop, first:

```
t_par <- time_resample_bt(100, "multisession")

t_par
```

```
t_par
```

```
# A tibble: 1 x 3
  expression   median mem_alloc
* <bch:expr> <bch:tm> <bch:byt>
1 resample      242ms      15MB
```

No dice! As we know from earlier on, though, we'll start to see a payoff when model fits take long enough to outweigh the overhead of sending data back and forth between workers. But, but! Remember the high-mileage lesson: this overhead is greater for distributed systems. Let's demonstrate this.

I'll first run this experiment for numbers of rows $100, 1000, ..., 1,000,000$ both sequentially and via socket clusters on my laptop, recording the timings as I do so. Then, I'll do the same thing on a popular data science hosted service, and we'll compare results.

```
bench::press(
  time_resample_bt(n_rows, plan),
  n_rows = 10^(2:6),
  plan = c("sequential", "multisession")
)
```

The resulting timings are in the object `timings` and look like this:

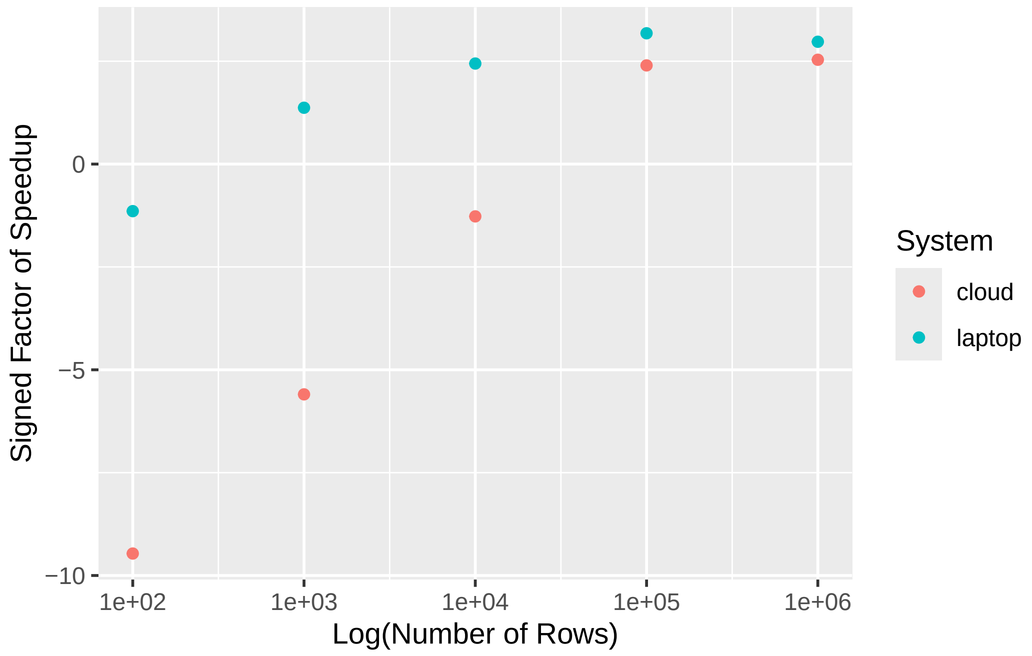
```
timings %>%
  select(n_rows, plan, median, system) %>%
  head()
```

```
# A tibble: 6 x 4
   n_rows plan           median system
*   <dbl> <chr>        <bch:tm> <chr>
1     100 sequential   216.08ms laptop
2    1000 sequential   442.77ms laptop
3   10000 sequential      2.81s laptop
4  100000 sequential      27.8s laptop
5 1000000 sequential      4.03m laptop
6     100 multisession 247.13ms laptop
```

There's one timing per unique combination of number of rows, parallelism plan, and system (`"laptop"` vs `"cloud"`). Based on these timings, we can calculate the factor of speedup for sequential evaluation versus its parallel analogue.

```
timings %>%
  select(n_rows, plan, median, system) %>%
  pivot_wider(names_from = plan, values_from = median) %>%
  mutate(
    speedup = as.numeric(sequential / multisession),
    speedup = if_else(speedup < 1, -1/speedup, speedup)
  ) %>%
```

```r
ggplot() +
aes(x = n_rows, y = speedup, col = system) +
geom_point() +
scale_x_log10() +
labs(
  x = "Log(Number of Rows)",
  y = "Signed Factor of Speedup",
  col = "System"
)
```



In this plot, a signed speedup value of 2 would mean that the socket cluster (i.e., parallel) approach was twice as fast, while a value of -2 would mean that the sequential approach ran twice as fast as the socket cluster approach. In general, larger numbers of rows (and thus longer-running model fits) tend to be associated with greater speedups as a result of switching to parallel computing. For local clusters on my laptop, the overhead of passing data around is small enough that I start to see a payoff when switching to parallel computing for only 1000 rows. As for the cloud system, though, model fits have to take a *long* time before switching to parallel computing begins to reduce elapsed times.

The conclusion to draw here is not that one ought not to use hosted setups for parallel computing. Aside from the other many benefits of doing data analysis in hosted environments, some of these environments enable “massively parallel” computing, or, in other words, a ton of cores. In the example shown in this chapter, we fitted a relatively small number of models, so we wouldn’t necessarily benefit (and would likely see a slowdown) from scaling up the number of cores utilized. If we had instead wanted to tune that boosted tree over 1,000 proposed hyper-parameter combinations—requiring 10,000 model fits with a 10-fold resampling scheme—we would likely be much better off utilizing a distributed system with higher latency (and maybe even less performant individual cores) and 1,000 or 10,000 cores.

3 The submodel trick

In Section 1.1.3, I created a custom `grid` of parameters, rather than relying on tidymodels' default grid generation, to enable something I referred to as the "submodel trick." The submodel trick allows us to evaluate many more models than we actually fit; given that model fitting is the most computationally intensive operation when resampling models (by far), cutting out a good few of those fits results in substantial time savings. This chapter will explain what the submodel trick is, demonstrating its individual contribution to the speedup we saw in Section 1.1.3 along the way. Then, I'll explain how users can use the submodel trick in their own analyses to speed up the model development process.

3.1 Demonstration

Recall that, in Section 1.1.2, we resampled an XGBoost boosted tree model to predict whether patients would be readmitted within 30 days after an inpatient hospital stay. Using default settings with no optimizations, the model took 3.68 hours to resample. With a switch in computation engine, implementation of parallel processing, change in search strategy, *and enablement of the submodel trick*, the time to resample was knocked down to 1.52 minutes. What is the submodel trick, though, and what was its individual contribution to that speedup?

First, loading packages and setting up resamples and the model specification as before:

```r
# load packages
library(tidymodels)
library(readmission)

# load and split data:
set.seed(1)
readmission_split <- initial_split(readmission)
readmission_train <- training(readmission_split)
readmission_test <- testing(readmission_split)
readmission_folds <- vfold_cv(readmission_train)

# set up model specification
bt <-
```

```r
  boost_tree(learn_rate = tune(), trees = tune()) %>%
  set_mode("classification")
```

3.1.1 Grids

If I just pass these resamples and model specification to `tune_grid()`, as I did in Section 1.1.2, tidymodels will take care of generating the grid of parameters to evaluate itself. By default, tidymodels generates grids of parameters using an experimental design called a latin hypercube (McKay, Beckman, and Conover 2000; Dupuy, Helbert, and Franco 2015). We can use the function `grid_latin_hypercube()` from the dials package to replicate the same grid that `tune_grid()` had generated under the hood:

```r
set.seed(1)
bt_grid_latin_hypercube <-
  bt %>%
  extract_parameter_set_dials() %>%
  grid_latin_hypercube(size = 12)
```

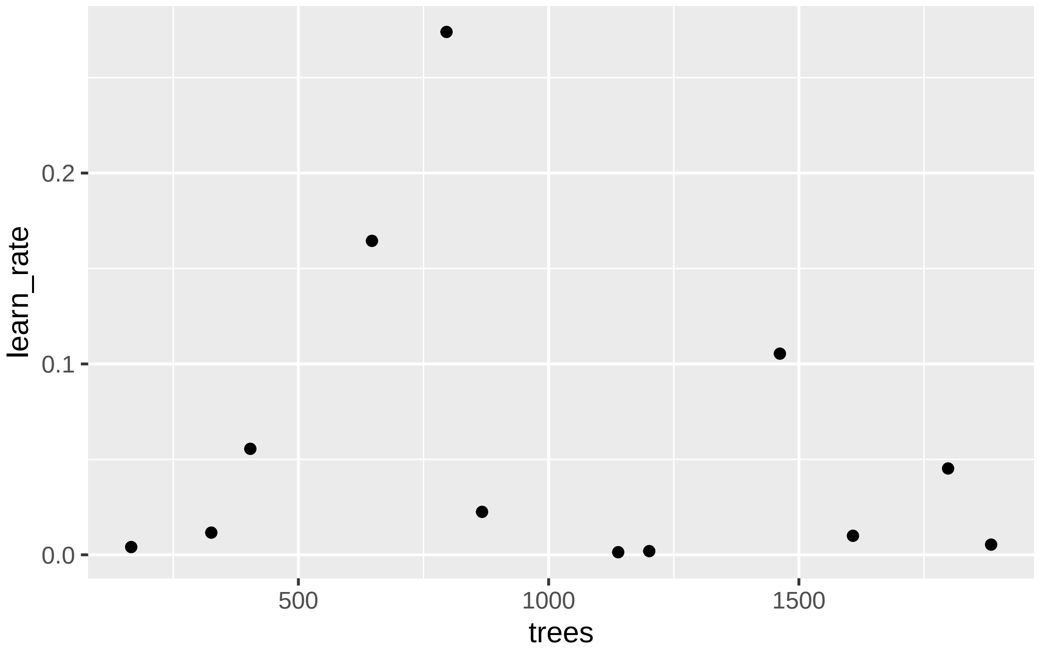
```
Warning: `grid_latin_hypercube()` was deprecated in dials 1.3.0.
i Please use `grid_space_filling()` instead.
```

```r
bt_grid_latin_hypercube
```

```
# A tibble: 12 x 2
   trees learn_rate
   <int>      <dbl>
 1   647    0.164
 2   404    0.0555
 3  1798    0.0452
 4   796    0.274
 5  1884    0.00535
 6  1139    0.00137
 7  1462    0.105
 8  1201    0.00194
 9   867    0.0225
10   166    0.00408
11   326    0.0116
12  1608    0.00997
```

Since we're working with a two-dimensional grid in this case, we can plot the resulting grid to get a sense for the distribution of values:

```
ggplot(bt_grid_latin_hypercube) +
  aes(x = trees, y = learn_rate) +
  geom_point()
```



While the details of sampling using latin hypercubes are not important to understand this chapter, note that values are not repeated in a given dimension. Said another way, we get 12 unique values for `trees`, and 12 unique values for `learn_rate`. Juxtapose this with the design resulting from `grid_regular()` used in Section 1.1.3:

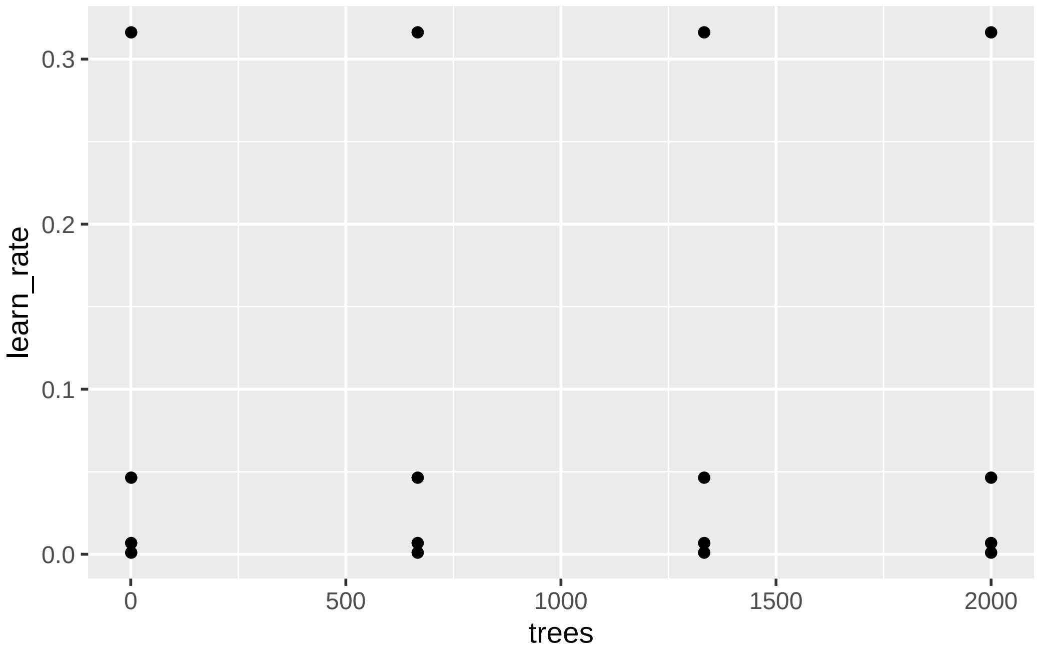
```
set.seed(1)
bt_grid_regular <-
  bt %>%
  extract_parameter_set_dials() %>%
  grid_regular(levels = 4)

bt_grid_regular
```

```
# A tibble: 16 x 2
   trees learn_rate
   <int>      <dbl>
```

```
 1     1    0.001
 2   667    0.001
 3  1333    0.001
 4  2000    0.001
 5     1    0.00681
 6   667    0.00681
 7  1333    0.00681
 8  2000    0.00681
 9     1    0.0464
10   667    0.0464
11  1333    0.0464
12  2000    0.0464
13     1    0.316
14   667    0.316
15  1333    0.316
16  2000    0.316
```

Plotting in the same way:

```
ggplot(bt_grid_regular) +
  aes(x = trees, y = learn_rate) +
  geom_point()
```



The argument `levels = 4` indicates that 4 values are generated individually for each parameter, and the resulting grid is created by pairing up each unique combination of those values.

You may have noticed that this regular grid contains even more proposed points—`4 x 4 = 16`—than the latin hypercube with `size = 12`. A reasonable question, then: how on earth would the larger grid be resampled more quickly than the smaller one? It's possible that I hid some slowdown resulting from this larger grid among the rest of the optimizations implemented in Section 1.1.3; lets test the effect of the change in grid by itself.

```r
set.seed(1)

bm_grid_regular <-
  bench::mark(
    grid_regular =
      tune_grid(
        object = bt,
        preprocessor = readmitted ~ .,
        resamples = readmission_folds,
        grid = bt_grid_regular
      )
  )
```

```r
bm_grid_regular
```

```
# A tibble: 1 x 3
  expression   median mem_alloc
* <bch:expr> <bch:tm> <bch:byt>
1 basic         1.23h    3.91GB
```

See? Have some faith in me!

Changing only the `grid` argument (and even increasing the number of proposed `grid` points), we've decreased the time to evaluate against resamples from 3.68 to 1.23 hours, or a speedup of -99.67%.

3.1.2 The trick

Passing this regular grid allowed `tune_grid()` to use what the tidymodels team refers to as the "submodel trick," where many more models can be evaluated than were actually fit.

To best understand how the submodel trick works, let's refresh on how boosted trees work. The training process begins with a simple decision tree, and subsequent trees are added iteratively, each one correcting the errors of the previous trees by focusing more on the data points associated with the greatest error. The final model is a weighted sum of all the individual trees, where each tree contributes to reducing the overall error.

So, for example, to train a boosted tree model with 2000 trees, we first need to train a boosted tree model with 1 tree. Then, we need to take that model, figure out where it made its largest errors, and train a second tree that aims to correct those errors. So on, until the, say, 667-th tree, and so on until the 1333-rd tree, and so on until, finally, the 2000-th tree. *Picking up what I'm putting down?* Along the way to training a boosted tree with 2000 trees, we happened to train a bunch of other models we might be interested in evaluating: what we call *submodels*. So, in the example of `bt_grid_regular`, for a given `learn_rate`, we only need to train the model with the maximum `trees`. In this example, that's a quarter of the model fits.

Note

You might note that we don't see a speedup nearly as drastic as 4 times. While we indeed only fit a quarter of the models, we're fitting the boosted trees with the largest number of trees, and the time to train a boosted tree scales linearly with the number of trees. Said another way, we're eliminating the need to fit only the faster-fitting models. This tends to be the case in many cases where the submodel trick applies.

To evaluate a fitted model with performance metrics, all we need are its predictions (and, usually, the true values being predicted). In pseudocode, resampling a model against performance metrics usually goes something like this:

```r
for (resample in resamples) {
  # analogue to the "training" set for the resample
  analysis <- analysis(resample)
  # analogue to the "testing" set for the resample
  assessment <- assessment(resample)

  for (model in models) {
    # the longest-running operation:
    model_fit <- fit(model, analysis)

    # usually, comparatively quick operations:
    model_predictions <- predict(model_fit, assessment)
    metrics <- c(metrics, metric(model_predictions))
  }
}
```

Note

`analysis()`, `assessment()`, `fit()`, and `predict()` are indeed actual functions in tidymodels. `metric()` is not, but an analogue could be created from the output of the function `metric_set()`.

Among all of these operations, fitting the model with `fit()` is usually the longest-running step, by far. In comparison, `predict()`ing on new values and calculating metrics takes very little time. Using the submodel trick allows us to reduce the number of `fit()`s while keeping the number of calls to `predict()` and `metric()` constant. In pseudocode, resampling with the submodel trick could look like:

```
for (resample in resamples) {
  analysis <- analysis(resample)
  assessment <- assessment(resample)

  models_to_fit <- models[unique(non_submodel_args)]

  for (model in models_to_fit) {
    model_fit <- fit(model, analysis)

    for (model_to_eval in models[unique(submodel_args)]) {
      model_to_eval <- predict(model_to_eval, assessment)
      metrics <- c(metrics, metric(model_predictions))
    }
  }
}
```

The above pseudocode admittedly requires some generosity (or mental gymnastics) to interpret, but the idea is that if `fit()`ting is indeed the majority of the time spent in resampling, and `models_to_fit` contains many fewer elements than `models` in the preceding pseudocode blocks, we should see substantial speedups.

3.1.3 At its most extreme

In the applied example above, we saw a relatively modest speedup. If we want to really show off the power of the submodel trick, in terms of time spent resampling per model evaluated, we can come up with a somewhat silly grid:

```
bt_grid_regular_go_brrr <-
  bt_grid_regular %>%
  slice_max(trees, by = learn_rate) %>%
  map(.x = c(1, seq(10, max(.$trees), 10)), .f = ~mutate(.y, trees = .x), dfr = .) %>%
  bind_rows()

bt_grid_regular_go_brrr
```

```
# A tibble: 804 x 2
   trees learn_rate
   <dbl>      <dbl>
 1     1    0.001
 2     1    0.00681
 3     1    0.0464
 4     1    0.316
 5    10    0.001
 6    10    0.00681
 7    10    0.0464
 8    10    0.316
 9    20    0.001
10    20    0.00681
# i 794 more rows
```

In this grid, we have the same number of unique values of `learn_rate`, 4, resulting in the same 4 model fits. Except that, in this case, we're evaluating every model with number of trees $1, 10, 20, 30, 40, ..., 2000$. If our hypothesis that predicting on the assessment set and generating performance metrics is comparatively fast is true, then we'll see that elapsed time *per grid point* is way lower:

```
set.seed(1)

bm_grid_regular_go_brrr <-
  bench::mark(
    grid_regular_go_brrr =
      tune_grid(
        object = bt,
        preprocessor = readmitted ~ .,
        resamples = readmission_folds,
        grid = bt_grid_regular_go_brrr
      )
  )
```

The above code took 1.44 hours to run, a bit longer than the 1.23 hours from `bm_grid_regular` (that fitted the same number of models), but comparable. *Per grid point,* that difference is huge:

```
# time per grid point for bm_grid_regular
bm_grid_regular$median[[1]] / nrow(bt_grid_regular)
```

```
[1] 4.62m
```

```
# time per grid point for bm_grid_regular_go_brrr
bm_grid_regular_go_brrr$median[[1]] / nrow(bt_grid_regular_go_brrr)
```

```
[1] 6.44s
```

Whether this difference in timing is of practical significance to a user is debatable. In the context where we generated `bm_grid_regular`, where the grid of points searched over is relatively comparable (and thus similarly likely to identify a performant model) yet the decreased number of model fits gave rise to a reasonable speedup—-99.67%—is undoubtedly impactful for many typical use cases of tidymodels. The more eye-popping per-grid-point speedups, as with `bm_grid_regular_go_brrr`, are more so a fun trick than a practical tool for most use cases.

3.2 Overview

I've demonstrated the impact of the submodel trick in the above section through one example context, tuning `trees` in an XGBoost boosted tree model. The submodel trick can be found in many places across the tidymodels framework, though.

Submodels are defined with respect to a given model parameter, e.g. `trees` in `boost_tree()`. While a given parameter often defines a submodel regardless of modeling engine for a given model type—e.g. `trees` defines submodels for `boost_tree()` models regardless of whether the model is fitted with `engine = "xgboost"` or `engine = "lightgbm"`—there are some exceptions. Many tidymodels users undoubtedly tune models using arguments that define submodels without even knowing it. Some common examples include:

- `penalty` in `linear_reg()`, `logistic_reg()`, and `multinom_reg()`, which controls the amount of regularization in linear models.
- `neighbors` in `nearest_neighbor()`, the number of training data points nearest the point to be predicted that are factored into the prediction.

Again, some modeling engines for each of these model types do not actually support prediction from submodels from the noted parameter. See Section 3.2.1 for a complete table of currently supported arguments defining submodels in tidymodels.

In the above example, we had to manually specify a grid like `bt_regular_grid` in order for tidymodels to use a grid that can take advantage of the submodel trick. This reflects the frameworks' general prioritization of predictive performance over computational performance in its design and defaults; latin hypercubes have better statistical properties when it comes to discovering performant hyperparameter combinations than a regular grid (McKay, Beckman, and Conover 2000; Stein 1987; Santner et al. 2003). However, note that in the case where a model is being tuned over only one argument, the submodel trick will kick in regardless of the sampling approach being used: regardless of how a set of univariate points are distributed, the most extreme parameter value (e.g. the max `trees`) can be used to generate predictions for values across the distribution.

3.2.1 Supported parameters

A number of tuning parameters support the submodel trick:

Model Type	Argument	Engines
boost_tree	trees	xgboost, C5.0, lightgbm
C5_rules	trees	C5.0
cubist_rules	neighbors	Cubist
discrim_flexible	num_terms	earth
linear_reg	penalty	glmnet
logistic_reg	penalty	glmnet
mars	num_terms	earth
multinom_reg	penalty	glmnet
nearest_neighbor	neighbors	kknn
pls	num_comp	mixOmics
poisson_reg	penalty	glmnet
proportional_hazards	penalty	glmnet
rule_fit	penalty	xrf

4 Sparsity

References

Bengtsson, Henrik. 2021. "A Unifying Framework for Parallel and Distributed Processing in R using Futures." *The R Journal* 13 (2): 273–91. https://doi.org/10.32614/RJ-2021-048.

Dupuy, Delphine, Céline Helbert, and Jessica Franco. 2015. "DiceDesign and DiceEval: Two r Packages for Design and Analysis of Computer Experiments." *Journal of Statistical Software* 65 (11).

Kuhn, Max, and Julia Silge. 2022. *Tidy Modeling with R.* " O'Reilly Media, Inc.".

McKay, Michael D, Richard J Beckman, and William J Conover. 2000. "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code." *Technometrics* 42 (1): 55–61.

R Core Team. 2024. *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Santner, Thomas J, Brian J Williams, William I Notz, and Brain J Williams. 2003. *The Design and Analysis of Computer Experiments.* Vol. 1. Springer.

Stein, Michael. 1987. "Large Sample Properties of Simulations Using Latin Hypercube Sampling." *Technometrics* 29 (2): 143–51.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science.* " O'Reilly Media, Inc.".