Applied Machine Learning - Regression Models

Max Kuhn (RStudio)

Loading

```r
library(tidymodels)
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

```
## ── Attaching packages ──────────────────────────────────────────── tidymodels 0.0.2 ──
```

```
## ✔ broom     0.5.1          ✔ purrr     0.3.3
## ✔ dials     0.0.3.9002     ✔ recipes   0.1.7.9001
## ✔ dplyr     0.8.3          ✔ rsample   0.0.5
## ✔ ggplot2   3.2.1          ✔ tibble    2.1.3
## ✔ infer     0.4.0          ✔ yardstick 0.0.4
## ✔ parsnip   0.0.4
```

```
## ── Conflicts ──────────────────────────────────────────────── tidymodels_conflicts() ──
## ✖ dplyr::combine()    masks gridExtra::combine()
## ✖ purrr::discard()    masks scales::discard()
## ✖ dplyr::filter()     masks stats::filter()
## ✖ recipes::fixed()    masks stringr::fixed()
## ✖ dplyr::group_rows() masks kableExtra::group_rows()
## ✖ dplyr::lag()        masks stats::lag()
## ✖ ggplot2::margin()   masks dials::margin()
## ✖ dials::offset()     masks stats::offset()
## ✖ recipes::step()     masks stats::step()
```

```r
library(tune)
```

Outline

- Example Data
- Regularized Linear Models
- Multivariate Adaptive Regression Splines
- Parallel Processing
- Bayesian Optimization

Example Data: Train Ridership

These data are used in our Feature Engineering and Selection book.

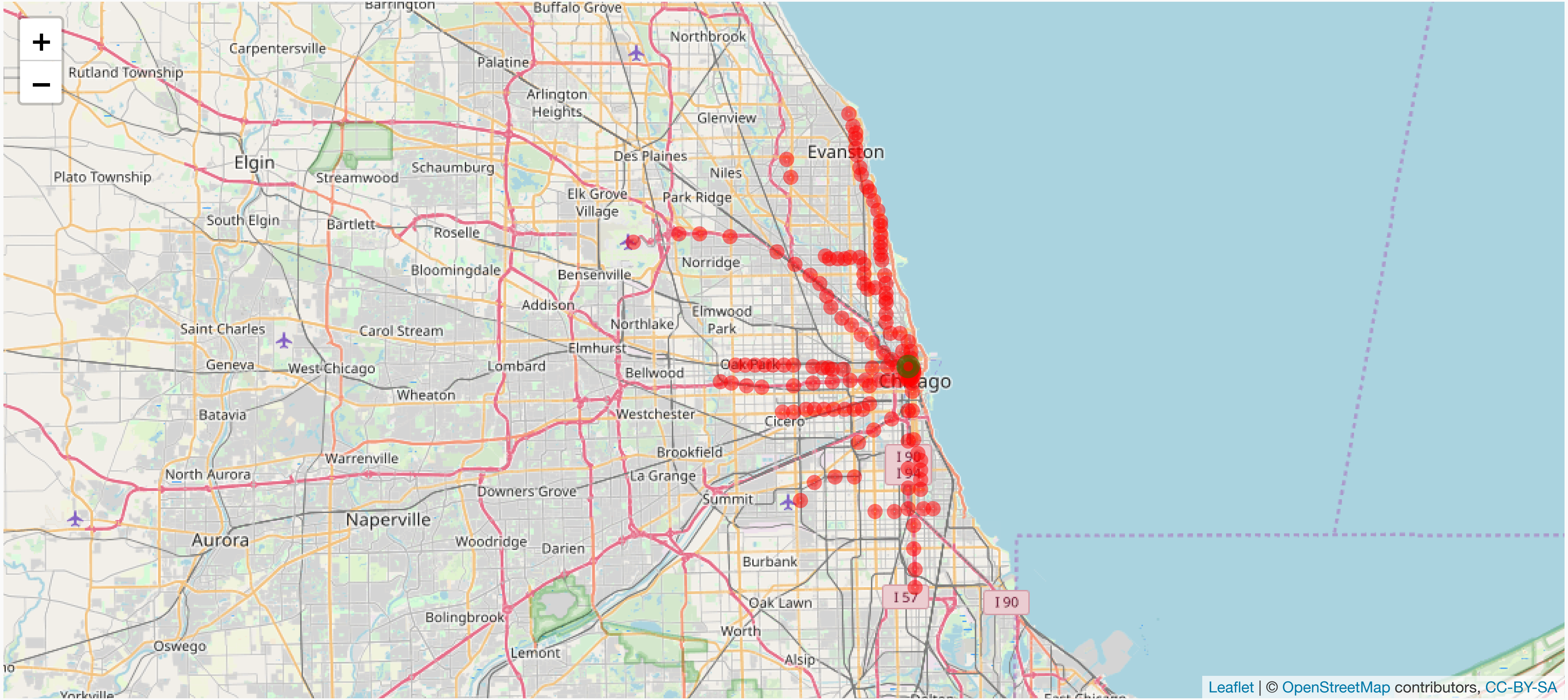
Several years worth of data were assembled to try to predict the daily number of people entering the Clark and Lake elevated ("L") train station in Chicago.

For predictors,

- the 14-day lagged ridership at this and other stations (units: thousands of rides/day)
- weather data
- home/away game schedules for Chicago teams
- the date

The data are in `dials`. See `?Chicago`.

L Train Locations



Hands-On: Explore the Data

Take a look at these data for a few minutes and see if you can find any interesting characteristics in the predictors or the outcome.

10:00

How Should Features Be Encoded/Engineered?

Should the ridership data be transformed?

How should we encode the date?

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago)
```

Define a few holidays from the `timeDate` package to be used later.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago)
```

`ridership` at Clark and Lake is the outcome.

All other columns are predictors.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago) %>%
  step_holiday(date, holidays = us_hol)
```

Make indicator variables for the 20 US holidays identified in `us_hol`.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago) %>%
  step_holiday(date, holidays = us_hol) %>%
  step_date(date)
```

Make factor variables from the `date` column, such as `dow`, `month`, and `year`.

These are not automatically converted to dummy variables.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago) %>%
  step_holiday(date, holidays = us_hol) %>%
  step_date(date) %>%
  step_rm(date)
```

We've made all of our date-based predictors, so remove the `date` column from the data.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago) %>%
  step_holiday(date, holidays = us_hol) %>%
  step_date(date) %>%
  step_rm(date) %>%
  step_dummy(all_nominal())
```

Make dummy variables out of all of the factor or character columns in the data.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago) %>%
  step_holiday(date, holidays = us_hol) %>%
  step_date(date) %>%
  step_rm(date) %>%
  step_dummy(all_nominal()) %>%
  step_zv(all_predictors())
```

In case there are column with only a single unique value (perhaps due to resampling), remove them.

A Recipe

```r
library(stringr)

# define a few holidays

us_hol <-
  timeDate::listHolidays() %>%
  str_subset("(^US)|(Easter)")

chi_rec <-
  recipe(ridership ~ ., data = Chicago) %>%
  step_holiday(date, holidays = us_hol) %>%
  step_date(date) %>%
  step_rm(date) %>%
  step_dummy(all_nominal()) %>%
  step_zv(all_predictors())
  # step_normalize(one_of(!!stations))
  # step_pca(one_of(!!stations), num_comp = tune())
```

The ridership between stations is highly correlated.

If we use a model that would be harded by this, we *could* extract the principal components for these columns.

Resampling

If your job were to model these data, you would probably take historical data as your training set and use the most recent data as the test set.

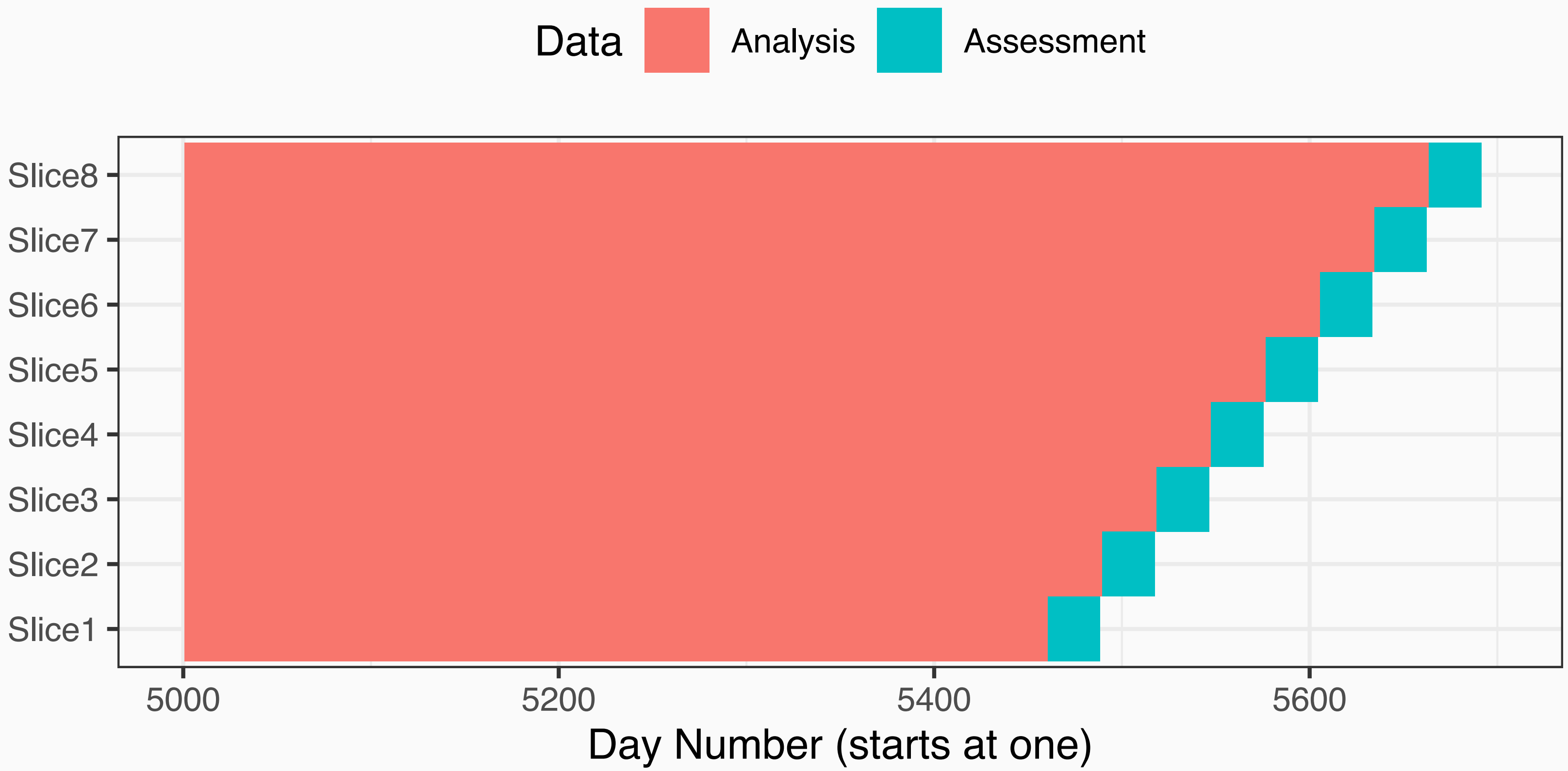
Our resampling scheme will emulate this using rolling forecasting origin resampling with

- Moving analysis sets of 15 years moving over 28 day periods
- An assessment set of the most recent 28 days of data

```
data_folds <- rolling_origin(Chicago, initial = 364 * 15, assess = 7 * 4, skip = 7 * 4, cumulative = FALSE)
data_folds %>% nrow()
```

```
## [1] 8
```

Resampling Graphic



Linear Models

Linear Regression Analysis

We'll start by fitting linear regression models to these data.

As a reminder, the "linear" part means that the model is linear in the *parameters*; we can add nonlinear terms to the model (e.g. `x^2` or `log(x)`) without causing issues.

The most start might be with `lm` and the formula method.

```
lm(ridership ~ . - date, data = Chicago)
```

We know that there are a lot of features that we'd miss out on though (e.g. holidays, day-of-the-week, etc.).

Potential Issues with Linear Regression

We'll look at the L train data and examine a few different models to illustrate some more complex models and approaches to optimizing them. We'll start with linear models.

However, some potential issues with linear methods:

- They do not automatically do *feature selection* and including irrelevant predictors may degrade performance.
- Linear models are sensitive to situations where the predictors are *highly correlated* (aka collinearity). This isn't too big of an issue for these data though.

To mitigate these two scenarios, *regularization* will be used. This approach adds a penalty to the regression parameters.

- In order to have a large slope in the model, the predictor will need to have a large impact on the model.

There are different types of regularization methods.

Effect of Collinearity

As an example of collinearity, our data set has two predictors that have a correlation above 0.95: `Irving_Park` and `Belmont`.

What happens when we fit models with both predictors versus one-at-a-time?

	Coefficients			
Term	**Belmont Only**	**Irving Park Only**	**Both Predictors**	**Variance Inflation**
Irving Park	---	4.974	4.109	26.842
Belmont	4.433	---	0.795	25.112

The coefficients can drastically change depending on what is in the model and their corresponding variances can also be artificially large and may flip signs.

Regularized Linear Regression

Now suppose we want to see if *regularizing* the regression coefficients will result in better fits.

The `glmnet` model can be used to build a linear model using L_1 or L_2 regularization (or a mixture of the two).

- The general formulation minimizes: $\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + penalty$.
- An L_1 penalty (penalty is $\lambda_1 \sum |\beta_j|$) can have the effect of setting coefficients to zero.
- L_2 regularization ($\lambda_2 \sum \beta_j^2$) is basically ridge regression where the magnitude of the coefficients are dampened to avoid overfitting.

For a `glmnet` model, we need to determine the total amount regularization (called `lambda`) and the mixture of L_1 and L_2 (called `alpha`).

- `alpha` = 1 is a *lasso model* while `alpha` = 0 is *ridge regression* (aka weight decay).

Predictors require centering/scaling before being used in a `glmnet`, lasso, or ridge regression model.

Technical bits can be found in Statistical Learning with Sparsity.

Harmonization of Parameter Names

If you are new to these models, `lambda` and `alpha` are pretty arcane and don't tell you anything about what they do.

Other packages use different names for these parameters (`reg_param`, `penalty`, `lambda1`, `lambda2`, etc.) so it isn't very friendly.

The `parsnip` package tries to standardize on less jargony and more self-documenting. We use `penalty` (instead of `lambda`) and `mixture` instead of `alpha`. These will always be the same for models within an engine and between-models too.

For this problem, we have two tuning parameters:

- `mixture` must be between zero and one. A small grid is used for this parameter.
- `penalty` is not as clear-cut. We consider values on the $\log_{10}$ scale. Usually values less than one are sufficient but this is not always true.

Tuning the Model

Let's once again use grid search with a regular grid to find good values of `penalty` and `mixture`.

It turns out that evaluating values of `penalty` are *cheaper* than values of `mixture`. We'll tune a grid of 20 penalty values and 5 mixtures between ridge regression and the lasso.

```
glmn_grid <- expand.grid(penalty = 10^seq(-3, -1, length = 20), mixture = (0:5)/5)
```

The reason that penalties are cheap is that this model simultaneously computes parameter estimates for *all possible penalty values* (for a fixed mixture). This is the *sub-model trick*.

Using the grid above, we evaluate 120 models but only fit five.

Tuning the Model

```r
# We need to normalize the predictors:
glmn_rec <- chi_rec %>% step_normalize(all_predictors())

glmn_mod <-
  linear_reg(penalty = tune(), mixture = tune()) %>% set_engine("glmnet")

# Save the assessment set predictions
ctrl <- control_grid(save_pred = TRUE)

glmn_res <-
  tune_grid(
    glmn_rec,
    model = glmn_mod,
    resamples = data_folds,
    grid = glmn_grid,
    control = ctrl
  )
```

While We Wait, Can I Interest You in Parallelism?

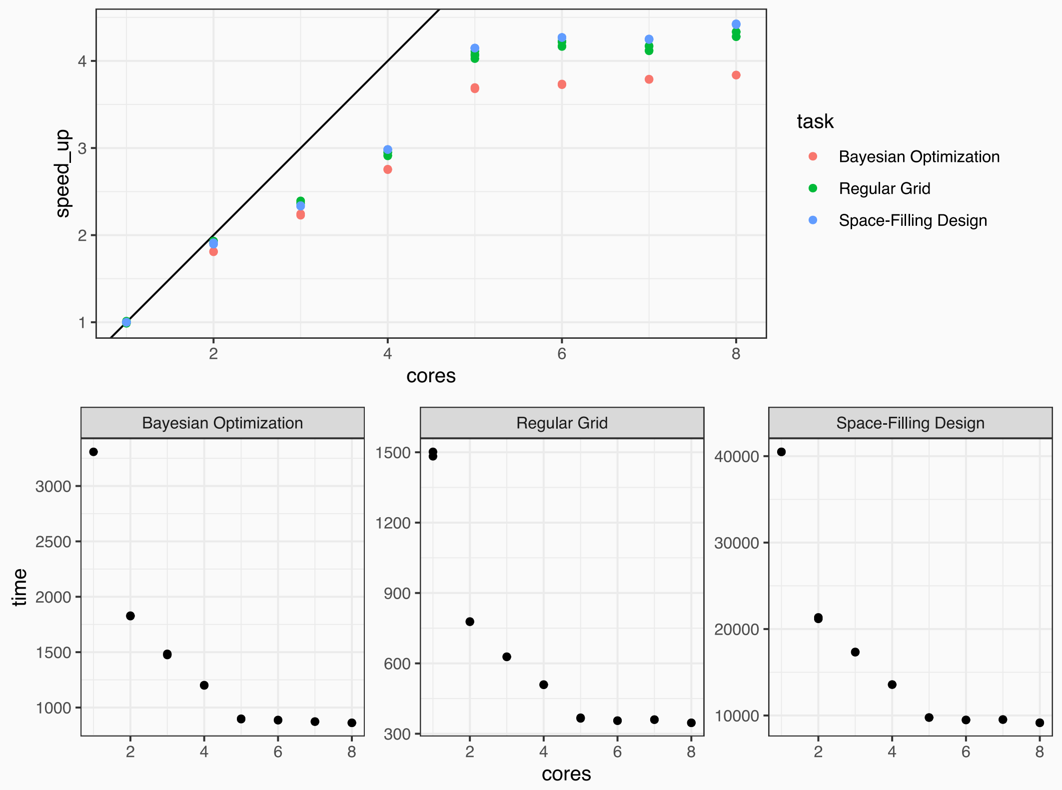
There is no real barrier to running these in parallel.

Can we benefit from splitting the fits up to run on multiple cores?

These speed-ups can be very model- and data-dependent but this pattern generally holds.

Note that there is little incremental benefit to using more workers than physical cores on the computer. Use `parallel::detectCores(logical = FALSE)`.

(A lot more details can be found in this blog post)



In these simulations, we estimated the speed-up by using the sub-model trick to be about *25-fold*.

Running in Parallel with {tune}

To loop through the models and data sets, `tune` uses the `foreach` package, which can parallelize `for` loops.

`foreach` has a number of *parallel backends* which allow various technologies to be used in conjunction with the package.

On CRAN, these are the "`do{X}`" packages, such as `doAzureParallel`, `doFuture`, `doMC`, `doMPI`, `doParallel`, `doRedis`, and `doSNOW`.

For example, `doMC` uses the `multicore` package, which forks processes to split computations (for unix and OS X). `doParallel` can be used for all operating systems.

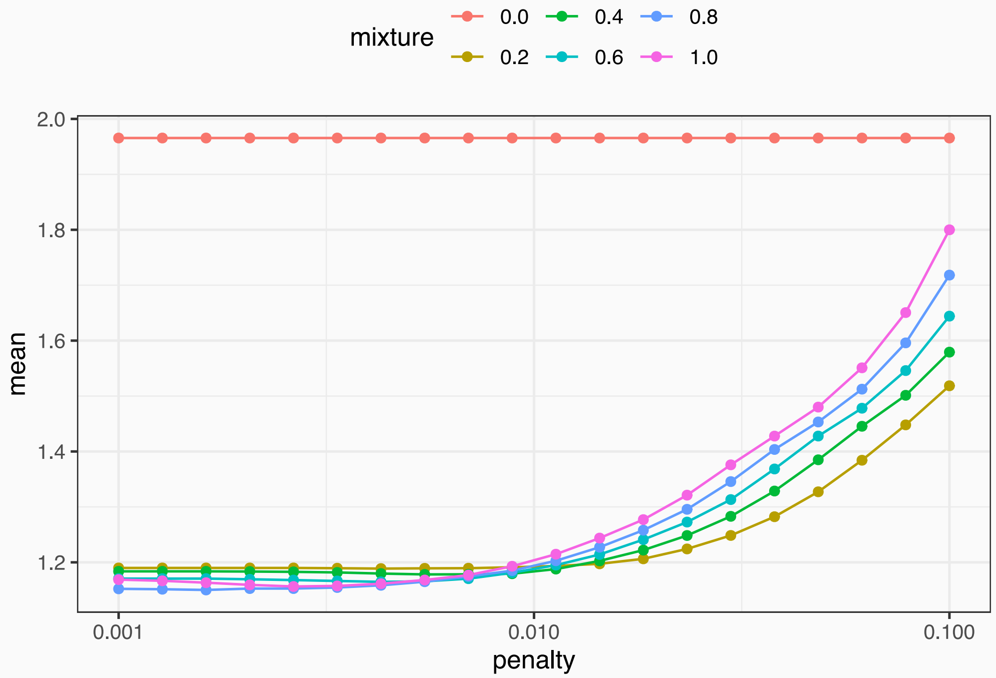
To use parallel processing in `tune`, no changes are needed when calling `tune_*()`.

The parallel technology must be *registered* with `foreach` prior to calling `tune_*()`:

```
library(doParallel)
cl <- makeCluster(6)
registerDoParallel(cl)

# run `tune_grid()`...

stopCluster(cl)
```

Plotting the Resampling Profile

```
rmse_vals <- collect_metrics(glmn_res) %>% filter(.metric == "r

rmse_vals %>%
  mutate(mixture = format(mixture)) %>%
  ggplot(aes(x = penalty, y = mean, col = mixture)) +
  geom_line() +
  geom_point() +
  scale_x_log10()
```



A pure ridge regression solution (`mixture = 0`) does poorly and the model seems to like a small amount of regularization overall.

The numerically best results

The numerically best results were:

```
show_best(glmn_res, metric = "rmse", maximize = FALSE)
```

```
## # A tibble: 5 x 7
##   penalty mixture .metric .estimator  mean     n std_err
##     <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl>
## 1 0.00162     0.8 rmse    standard    1.15     8  0.0688
## 2 0.00127     0.8 rmse    standard    1.15     8  0.0680
## 3 0.001       0.8 rmse    standard    1.15     8  0.0682
## 4 0.00207     0.8 rmse    standard    1.15     8  0.0699
## 5 0.00264     0.8 rmse    standard    1.15     8  0.0712
```

```
best_glmn <- select_best(glmn_res, metric = "rmse", maximize = FALSE)
best_glmn
```

```
## # A tibble: 1 x 2
##   penalty mixture
##     <dbl>   <dbl>
## 1 0.00162     0.8
```

Residual Analysis

Recall that the `save_pred = TRUE` option was used. That retains the held-out predictions for each resample and sub-model. Those are in a list column called `.predictions`.

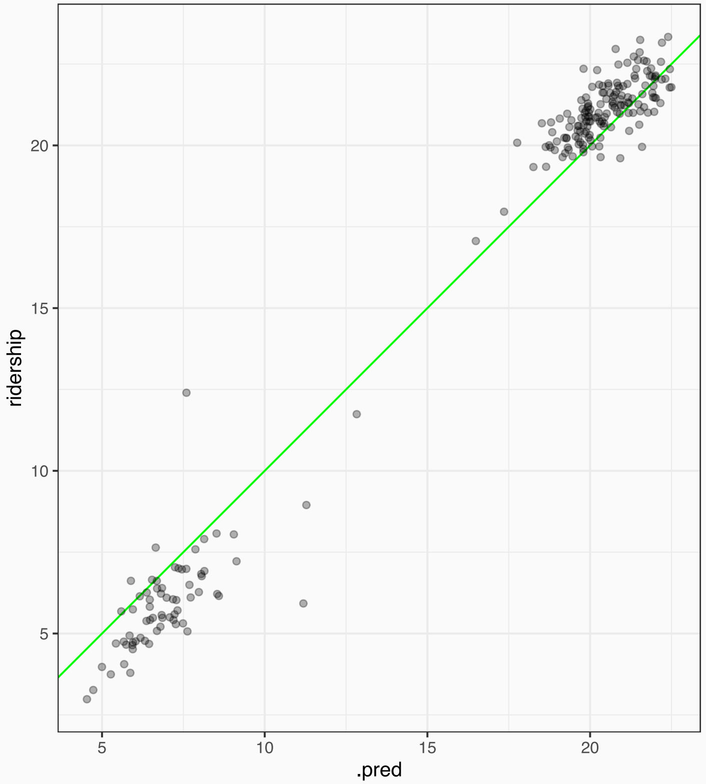
We can use `tidyr::unnest()` to get the results back or use this convenience function:

```
lr_pred <- collect_predictions(glmn_res)
lr_pred %>% slice(1:10)
```

```
## # A tibble: 10 x 6
##    id     .pred  .row penalty mixture ridership
##    <chr>  <dbl> <int>   <dbl>   <dbl>     <dbl>
##  1 Slice1  18.8  5461 0.001         0      19.6
##  2 Slice1  18.8  5461 0.00127       0      19.6
##  3 Slice1  18.8  5461 0.00162       0      19.6
##  4 Slice1  18.8  5461 0.00207       0      19.6
##  5 Slice1  18.8  5461 0.00264       0      19.6
##  6 Slice1  18.8  5461 0.00336       0      19.6
##  7 Slice1  18.8  5461 0.00428       0      19.6
##  8 Slice1  18.8  5461 0.00546       0      19.6
##  9 Slice1  18.8  5461 0.00695       0      19.6
## 10 Slice1  18.8  5461 0.00886       0      19.6
```

Observed Versus Predicted Plot

```r
# Keep the best model
lr_pred <-
  lr_pred %>%
  inner_join(best_glmn, by = c("penalty", "mixture"))

ggplot(lr_pred, aes(x = .pred, y = ridership)) +
  geom_abline(col = "green") +
  geom_point(alpha = .3) +
  coord_equal()
```



Which training set points had the worst results?

```
large_resid <-
  lr_pred %>%
  mutate(resid = ridership - .pred) %>%
  arrange(desc(abs(resid))) %>%
  slice(1:4)
```

```
library(lubridate)
Chicago %>%
  slice(large_resid$.row) %>%
  mutate(day = wday(date, label = TRUE)) %>%
  bind_cols(large_resid) %>%
  select(date, day, ridership, .pred, resid)
```

```
## # A tibble: 4 x 5
##   date       day   ridership .pred resid
##   <date>     <ord>     <dbl> <dbl> <dbl>
## 1 2016-07-04 Mon        5.92 11.2  -5.26
## 2 2016-03-12 Sat       12.4   7.59  4.80
## 3 2016-06-26 Sun        5.07  7.63 -2.56
## 4 2016-04-01 Fri       22.4  19.8   2.56
```

We have a July 4th holiday indicator yet still over-predicted.

For this data set, I end up googling to see why my predictions fail.

Which training set points had the worst results?

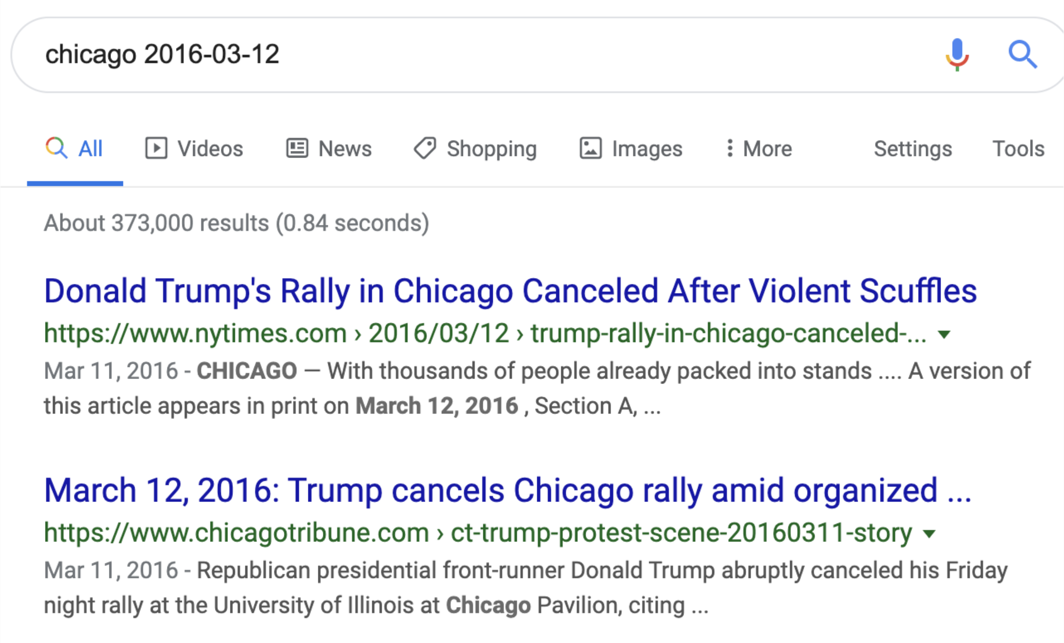
```r
large_resid <-
  lr_pred %>%
  mutate(resid = ridership - .pred) %>%
  arrange(desc(abs(resid))) %>%
  slice(1:4)
```

```r
library(lubridate)
Chicago %>%
  slice(large_resid$.row) %>%
  mutate(day = wday(date, label = TRUE)) %>%
  bind_cols(large_resid) %>%
  select(date, day, ridership, .pred, resid)
```

```
## # A tibble: 4 x 5
##   date       day   ridership .pred resid
##   <date>     <ord>     <dbl> <dbl> <dbl>
## 1 2016-07-04 Mon        5.92 11.2  -5.26
## 2 2016-03-12 Sat       12.4   7.59  4.80
## 3 2016-06-26 Sun        5.07  7.63 -2.56
## 4 2016-04-01 Fri       22.4  19.8   2.56
```

We have a July 4th holiday indicator yet still over-predicted.

For this data set, I end up googling to see why my predictions fail.



Creating a Final Model

Let's prep the recipe then fit the final glmnet model with the best parameters:

```r
glmn_rec_final <- prep(glmn_rec)
```

```r
glmn_mod_final <- finalize_model(glmn_mod, best_glmn)
glmn_mod_final
```

```
## Linear Regression Model Specification (regression)
##
## Main Arguments:
##   penalty = 0.00162377673918872
##   mixture = 0.8
##
## Computational engine: glmnet
```

```r
glmn_fit <-
  glmn_mod_final %>%
  fit(ridership ~ ., data = juice(glmn_rec_final))
```

```r
glmn_fit
```

```
## parsnip model object
##
## Fit in:  39ms
## Call:  glmnet::glmnet(x = as.matrix(x), y = y, fami
##
##    Df   %Dev Lambda
## 1   0 0.0000 7.2490
## 2   2 0.1117 6.6050
## 3   5 0.2175 6.0180
## 4   5 0.3095 5.4830
## 5   8 0.3869 4.9960
## 6   8 0.4523 4.5520
## 7   9 0.5068 4.1480
## 8   9 0.5524 3.7800
## 9   9 0.5904 3.4440
## 10  9 0.6221 3.1380
## 11 10 0.6488 2.8590
```

Using the glmnet Object

The `parsnip` object saves the optimized model that was fit to the entire training set in the slot `fit`.

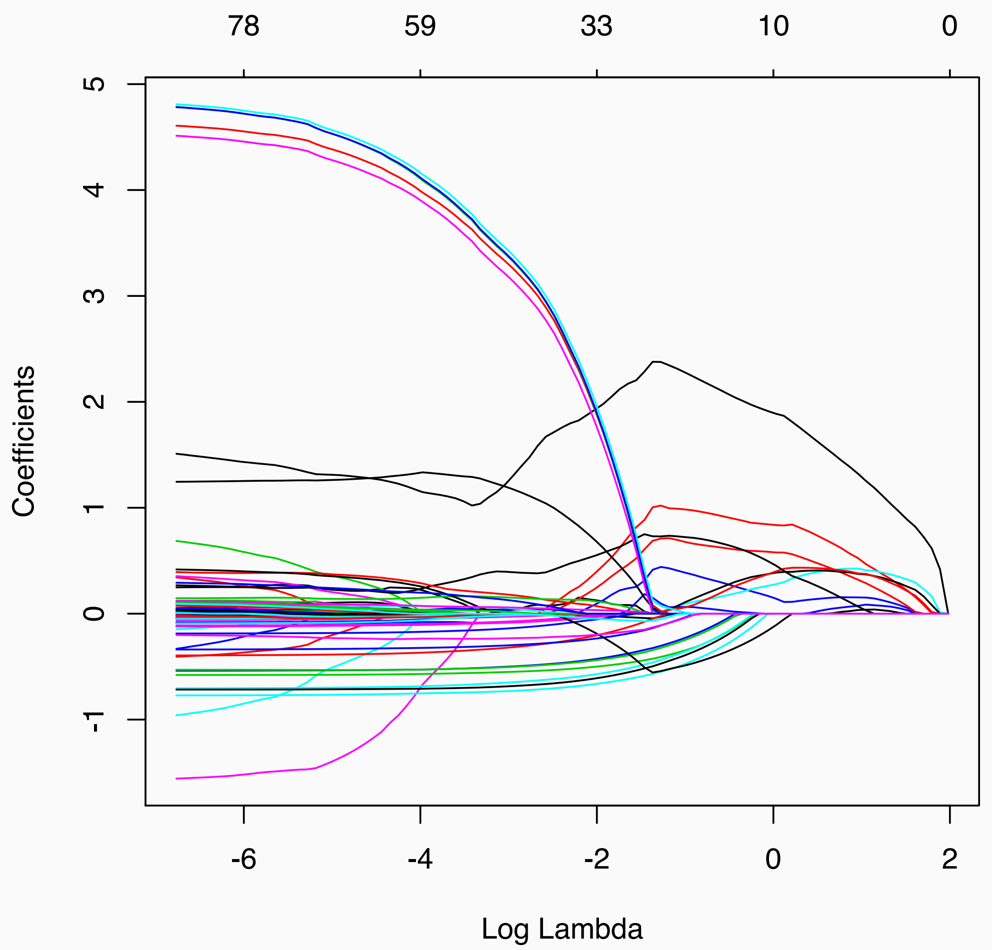
This can be used as it normally would.

The plot on the right is creating using

```
library(glmnet)
plot(glmn_fit$fit, xvar = "lambda")
```

However, **please don't use `predict(object$fit)`** !

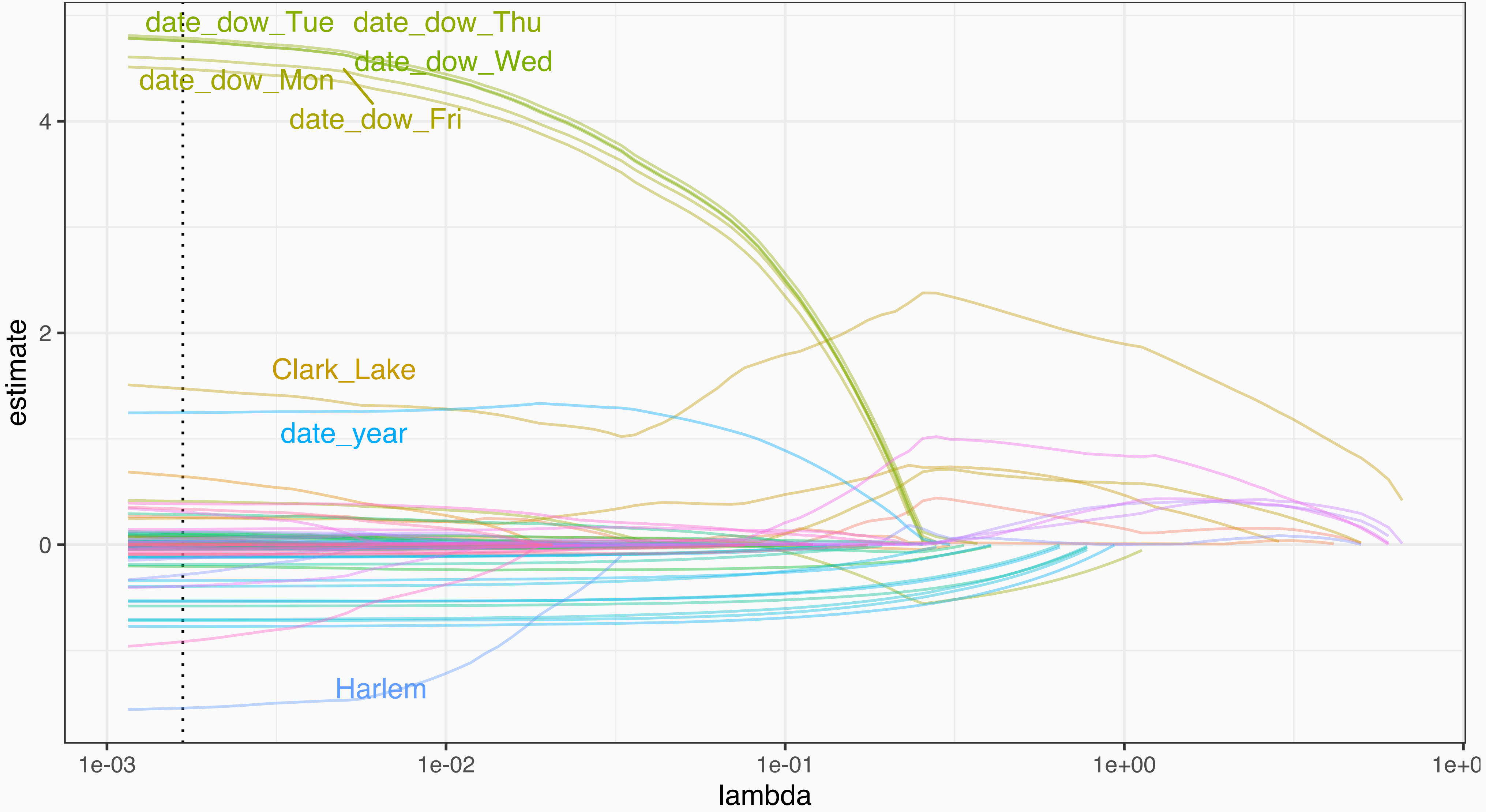
Use the `predict()` method on the object that is produced by `fit`.



A glmnet Coefficient Plot

```
# Get the set of coefficients across penalty values
tidy_coefs <-
  broom::tidy(glmn_fit) %>%
  dplyr::filter(term != "(Intercept)") %>%
  dplyr::select(-step, -dev.ratio)

# Get the lambda closest to tune's optimal choice
delta <- abs(tidy_coefs$lambda - best_glmn$penalty)
lambda_opt <- tidy_coefs$lambda[which.min(delta)]

# Keep the large values
label_coefs <-
  tidy_coefs %>%
  mutate(abs_estimate = abs(estimate)) %>%
  dplyr::filter(abs_estimate >= 1.1) %>%
  distinct(term) %>%
  inner_join(tidy_coefs, by = "term") %>%
  dplyr::filter(lambda == lambda_opt)

# plot the paths and highlight the large values
tidy_coefs %>%
  ggplot(aes(x = lambda, y = estimate, group = term, col = term, label = term)) +
  geom_vline(xintercept = lambda_opt, lty = 3) +
  geom_line(alpha = .4) +
  theme(legend.position = "none") +
  scale_x_log10() +
  geom_text_repel(data = label_coefs, aes(x = .005))
```

A glmnet Coefficient Plot



glmnet Variable Importance

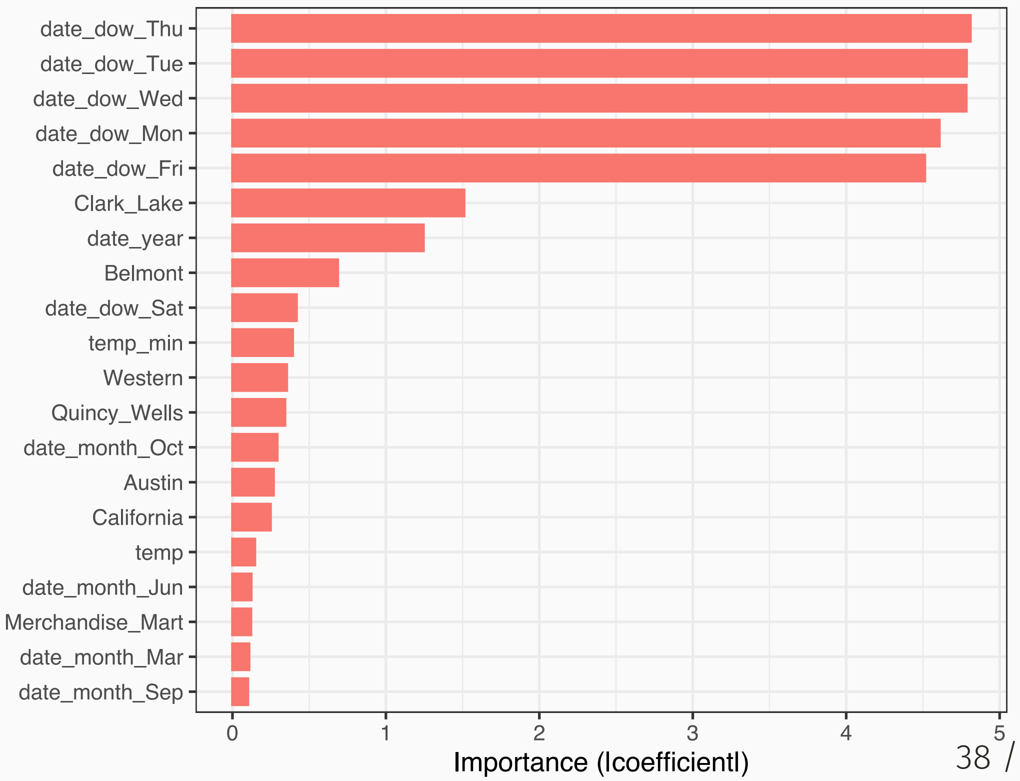
Variable importance scores are aggregate metrics that try to measure how much each predicted affected the model results.

These methods are specific to each model and not all models have ways to measure importance.

For (generalized) linear models, the simplest approach is to look at the absolute value of the regression coefficients (recall that we normalized the predictors).

The `caret` and `vip` packages have general interfaces to compute these measures and plot the results.

```r
library(vip)

vip(glmn_fit, num_features = 20L,
    # Needs to know which coefficients to use
    lambda = best_glmn$penalty)
```



What's Next?

The model is pretty simple right now.

If this level of performance is acceptable, we'd be done.

If not, more thorough residual analysis would be used to determine if more complex features should be used in the analysis.

There is the choice of making a simple model more complex or trying out a complex model.

We'll stop here with linear regression and try something else.

Multivariate Adaptive Regression Splines

Multivariate Adaptive Regression Splines (MARS)

MARS is a nonlinear machine learning model that develops sequential sets of artificial features that are used in linear models (similar to the previous spline discussion).

The features are "hinge functions" or single knot splines that use the function:

```r
h(x) <- function(x) ifelse(x > 0, x, 0)
```

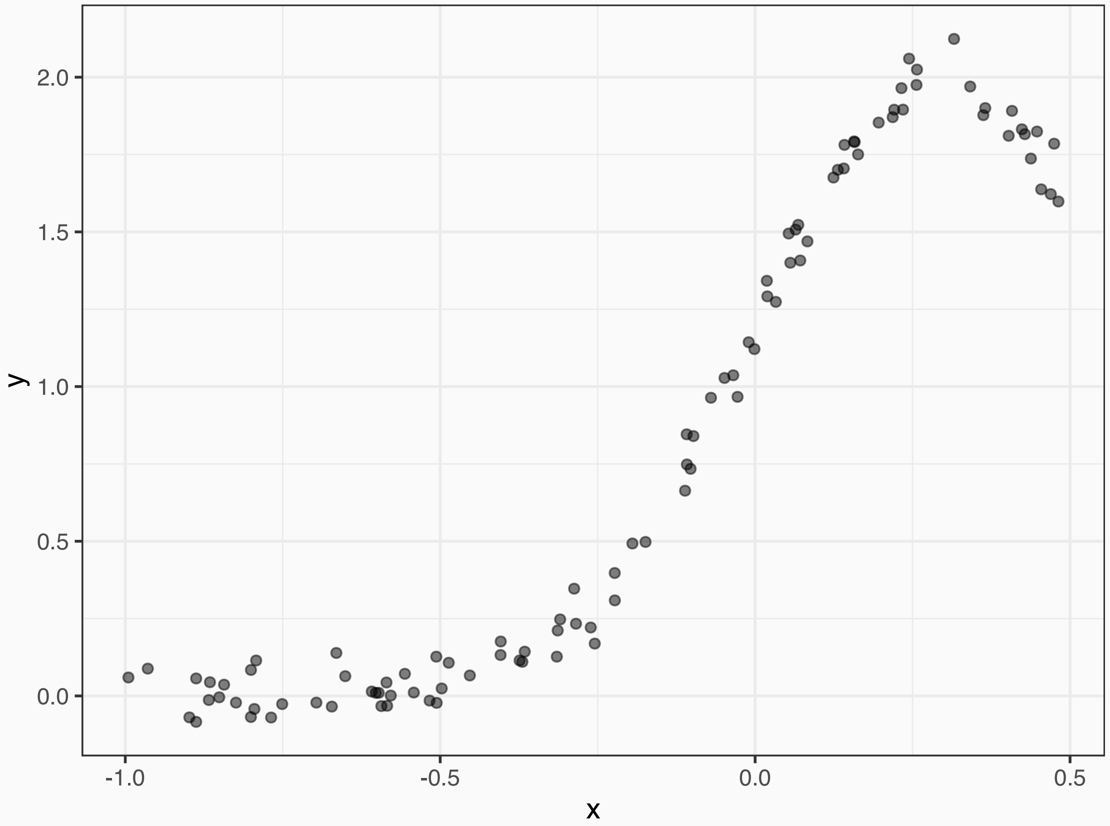
The MARS model does a fast search through every predictor and every value of each predictor to find a suitable "split" point for the predictor that results in the best features.

Suppose a value `x0` is found. The MARS model creates two model terms `h(x - x0)` and `h(x0 - x)` that are added to the intercept column. This creates a type of *segmented regression*.

These terms are the same as deep learning rectified linear units (ReLU).

Let's look at some example data...

Simulated Data: `y = 2 * exp(-6 * (x - 0.3)^2) + e`



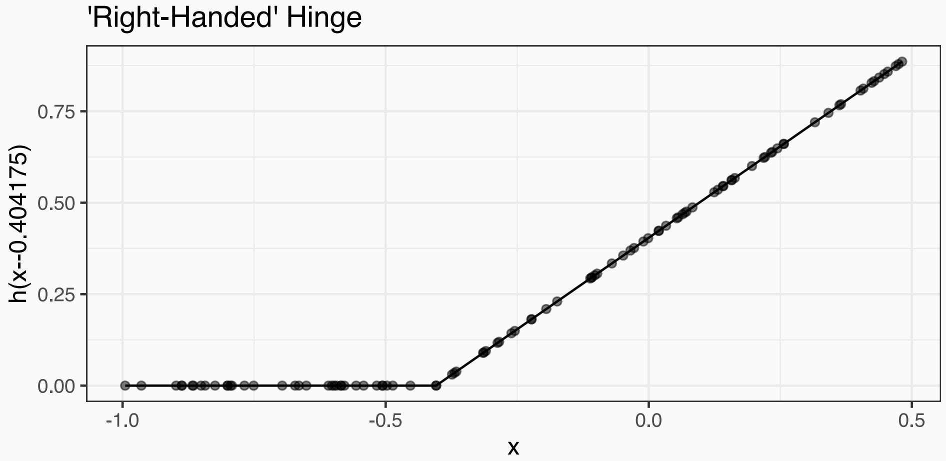
MARS Feature Creation -- Iteration #1

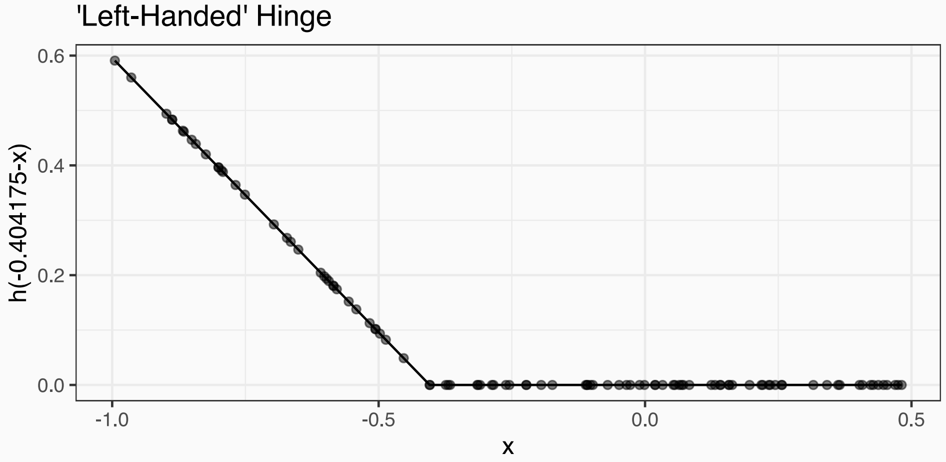
After searching through these data, the model evaluates all possible values of `x0` to find the best "cut" of the data. It finally chooses a value of -0.4041751.

To do this, it creates these two new predictors that isolate different regions of `x`.

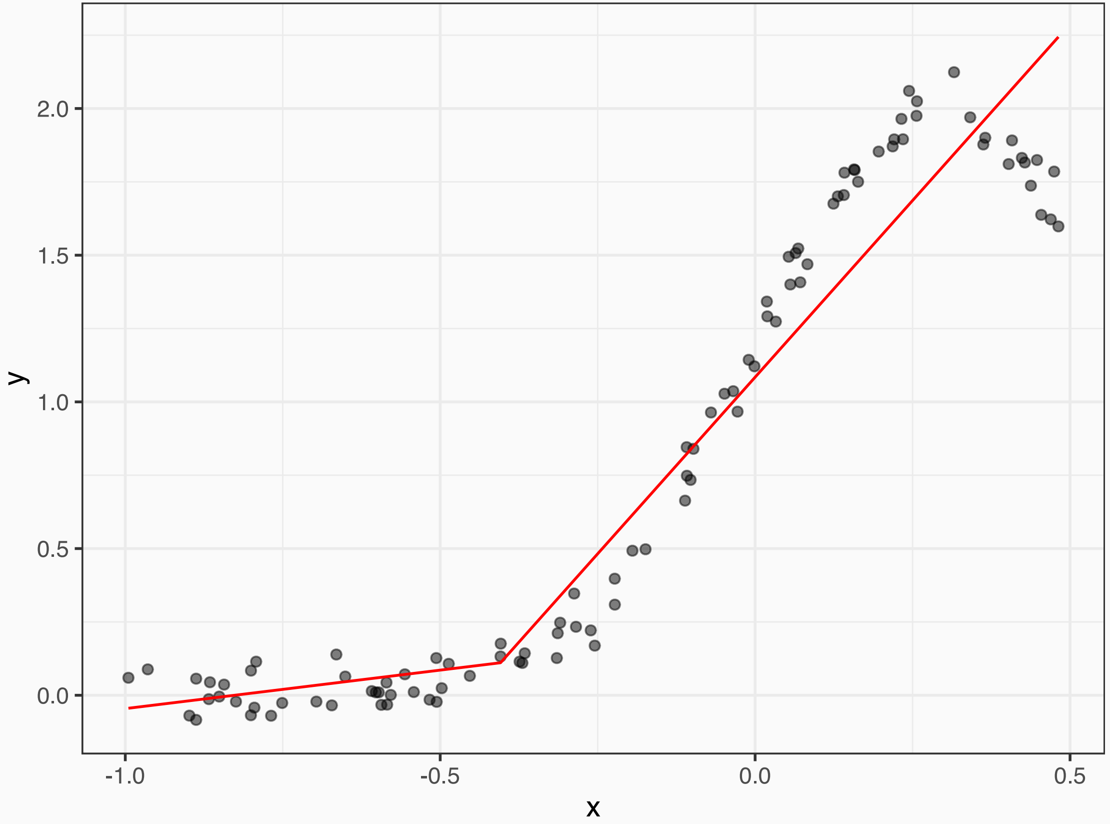
If we stop there, these two terms would be added into a linear regression model, yielding:

```
## y =
##   0.111
##   - 0.262 * h(-0.404175 - x)
##   +  2.41 * h(x - -0.404175)
```





Fitted Model with Two Features



Growing and Pruning

Similar to tree-based models, MARS starts off with a "growing" stage where it keeps adding new features until it reaches a pre-defined limit.

After the first pair is created, the next cut-point is found using another exhaustive search to see which split of a predictor is best *conditional on the existing features*.

Once all the features are created, a *pruning phase* starts where model selection tools are used to eliminate terms that do not contribute meaningfully to the model.

Generalized cross-validation (GCV) is used to efficiently remove model terms while still providing some protection from overfitting.

Model Size

There are two approaches:

1. Use the internal CGV to prune the model to the best subset size. This is fast but you don't learn much and it may under-select terms.

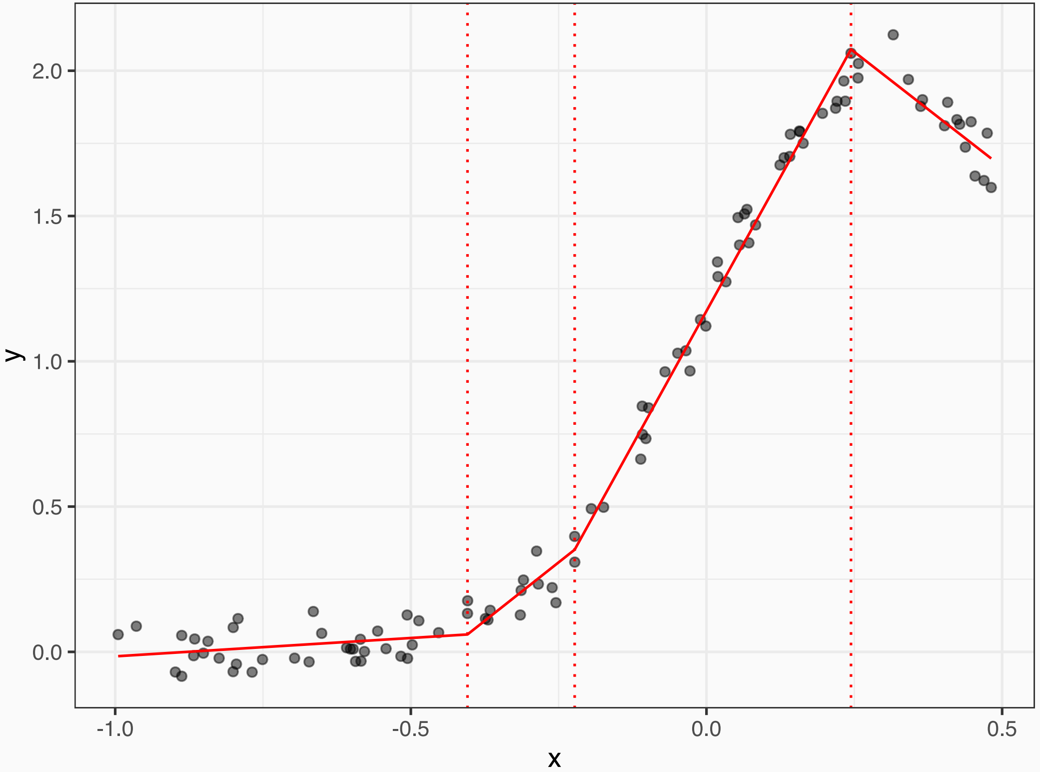
2. Use the external resampling (10-fold CV here) to tune the model as you would any other.

I usually don't start with GCV. Instead use method #2 above to understand the trends.

The Final Model

For the simulated data, the mars model only requires 4 features to model the data (via GCV).

```
## y =
##   0.0599
##   - 0.126 * h(-0.404175 - x)
##   +  1.61 * h(x - -0.404175)
##   +  2.08 * h(x - -0.222918)
##   -  5.27 * h(x - 0.244406)
```

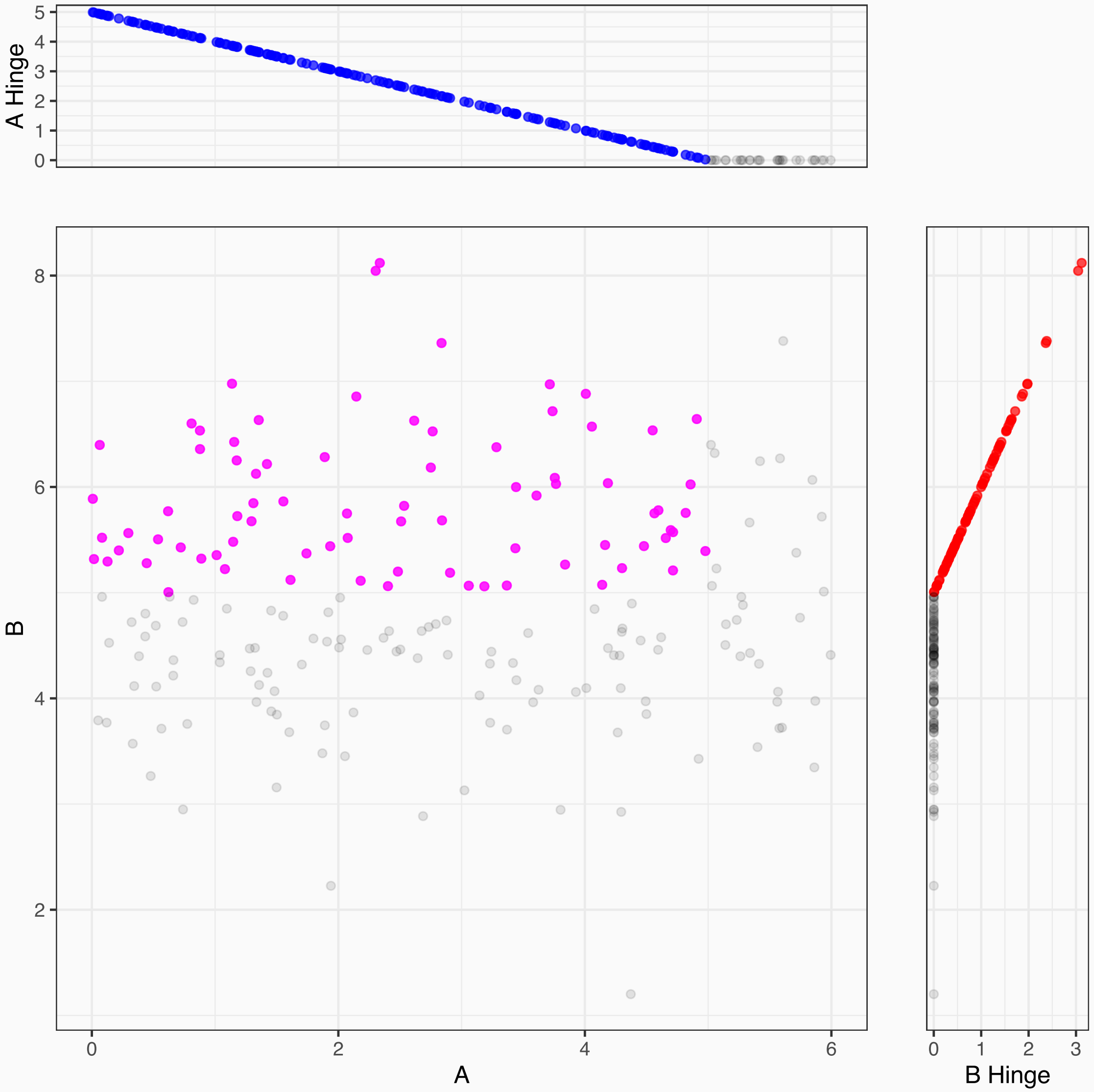


The parameters are estimated by added the MARS features into ordinary linear regression models using least squares.

Aspects of MARS Models

- The model also tests to see if a simple linear term is best (i.e. not split). This is also how dummy variables are evaluated.
- The model automatically conducts *feature selection*; if a predictor is never used in a split, it is functionally independent of the model. This is really good!
- If an additive model is used (as in the previous example), the functional form of each predictor can be determined (and visualized) independently for each predictor.
- A *second degree* MARS model also evaluates interactions of two hinge features (e.g. `h(x0 - x) * h(z - z0)`). This can be useful in isolating regions of bivariate predictor space since it divides two-dimensional space into four quadrants. (see next slide)

Second Degree MARS Term Example



MARS in R

The `mda` package has a `mars` function but the `earth` package is far superior.

The `earth()` function has both formula and non-formula interfaces. It can also be used with generalized linear models and flexible discriminant analysis.

To use the nominal growing and GCV pruning process, the syntax is

```
earth(y ~ ., data)

# or

earth(x = x, y = y)
```

The feature creation process can be controlled using the `nk`, `nprune`, and `pmethod` parameters although this can be somewhat complex.

There is a variable importance method that tracks the changes in the GCV results as features are added to the model.

MARS in via {parsnip} and {tune}

As with other models, a specification is made for the model. For our data:

```
# Let MARS decide the number of terms but tune the term dimensions
mars_mod <-  mars(prod_degree = tune())

# We'll decide via search:
mars_mod <-
  mars(num_terms = tune("mars terms"), prod_degree = tune(), prune_method = "none") %>%
  set_engine("earth") %>%
  set_mode("regression")
```

One issue with MARS is that it is based on linear regression. We know that linear regression doesn't do so well when the predictors are highly correlated.

We'll add to our recipe to de-correlate the data using principal component analysis:

```
mars_rec <-
  chi_rec %>%
  step_normalize(one_of(!!stations)) %>%
  step_pca(one_of(!!stations), num_comp = tune("pca comps"))
```

Segue --- Iterative Search Methods

Grid Versus Iterative Search

Grid search:

- The candidate values need to be pre-defined and don't learn from previous results.
    - You don't know the best values until all the computations are finished.
- With many parameters, it is difficult to efficiently cover the parameter space.
- Easily optimized via parallel processing and other tricks

Iterative Search:

- *Usually* builds a probability model to predict better parameters to test based on previous results.
- More flexibility in how the parameter space is searched.
- Less opportunities for efficiency optimizations are possible.

Any search procedure could be used.

- This repo shows examples using genetic algorithm, Nelder-Mead simplex search, and other approaches.

The most popular method is *Bayesian optimization*. We'll focus on this today.

Bayesian Optimization

Takes an initial set of results and uses these to build a model to predict new tuning parameters.

What makes it Bayesian?

- A Gaussian Process model, with some distributional assumptions, is usually the "meta-model".

This enables mean and variance predictions to be used (more later)

How Are the Tuning Data Used

The Gaussian Process uses the previous parameters as *predictors* and our performance measure as the *outcome*:

Tuning parameters are now predictors			Now the outcome
mars terms	**prod_degree**	**pca comps**	**mean**
71	1	11	1.203
70	2	20	1.825
58	2	7	1.825
15	2	16	2.623
88	1	1	1.134

Gaussian Process Model

This approach is often used in the analysis of spatial data and, as parameterized here, has some connections to kernel methods (such as support vector machines).

- I recommend Rasmussen and Williams (2006) (pdf) as a good place to start.

The model assumes that the model residuals follow a Gaussian distribution and that the model *parameters* have a joint Gaussian (prior) distribution.

Based on these assumptions, the predictive (posterior) distribution is also Gaussian

The trick that makes this model so useful here is how the covariance function of the *model inputs* is defined.

- A *kernel* function is often used that is small when the predictors are close and increases as they diverge.
- The radial basis function is a good example: $K(x_a, x_b) = exp(-0.5(x_a - x_b)^2)$

Gaussian Process Model Predictions

Using a nonlinear kernel function enables the GP to create nonlinear regression functions.

Since this is a Bayesian model, both the mean *and variance* of model performance can be predicted.

The predicted variance is usually dominated by the spatial relationships between predictors.

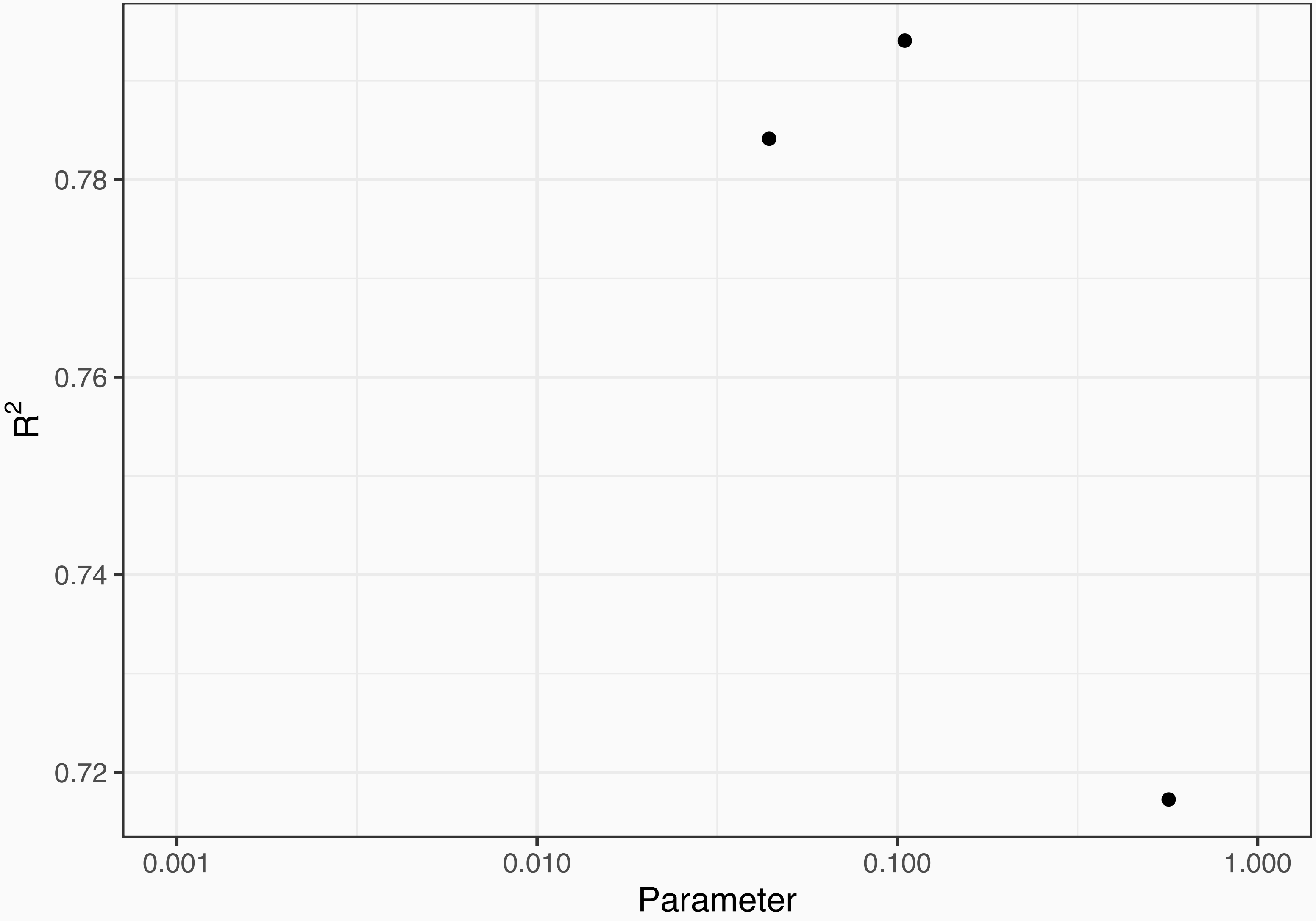
- Predictions made far away from the training data used for the GP tend to have very large variances.

We'll see examples of this in a minute.

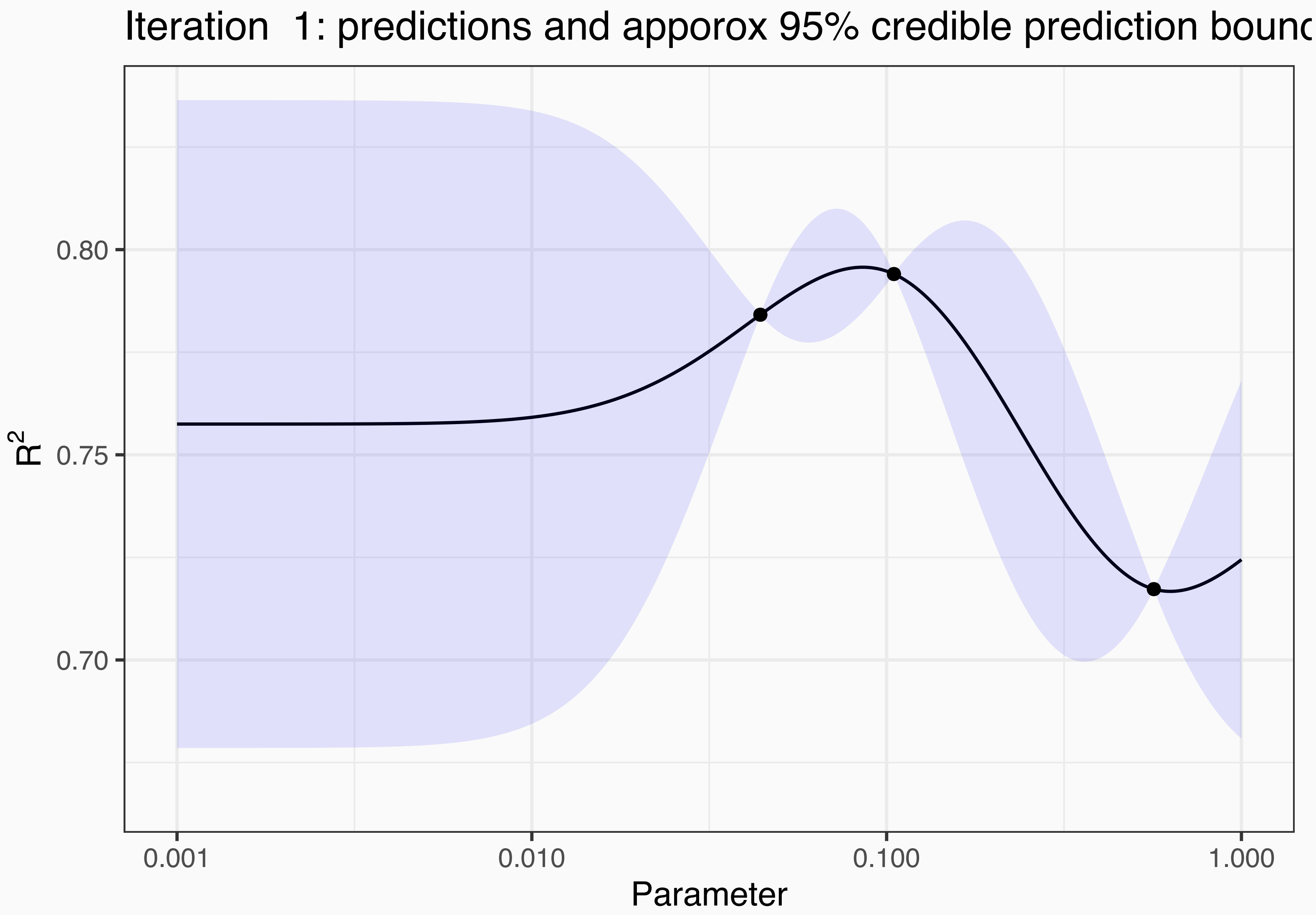
To start, suppose we have a single numeric tuning parameter and we are trying to maximize R^2

Three parameter values were sampled in the middle of the parameter's range.

Initial Grid Results



Initial GP Predictions



Selecting New Tuning Parameter Candidates

Using this model, the Bayesian optimization process would search the grid to find the "best" new parameters to evaluate using resampling.

- A nonlinear optimization routine or grid search can be used here.

Once the resampling results are obtained, another GP is fit and the process repeats.

How do we select the best parameter? Bayesian optimization introduced the idea of *acquisition functions*.

These functions are used to make trade-offs between exploitation and exploration:

- exploration: search new areas of the parameter space that seem promising
- exploitation: search in the vicinity of the existing best results.

This is usually a trade-off between mean and variance.

Acquisition Function Based on Credible Intervals

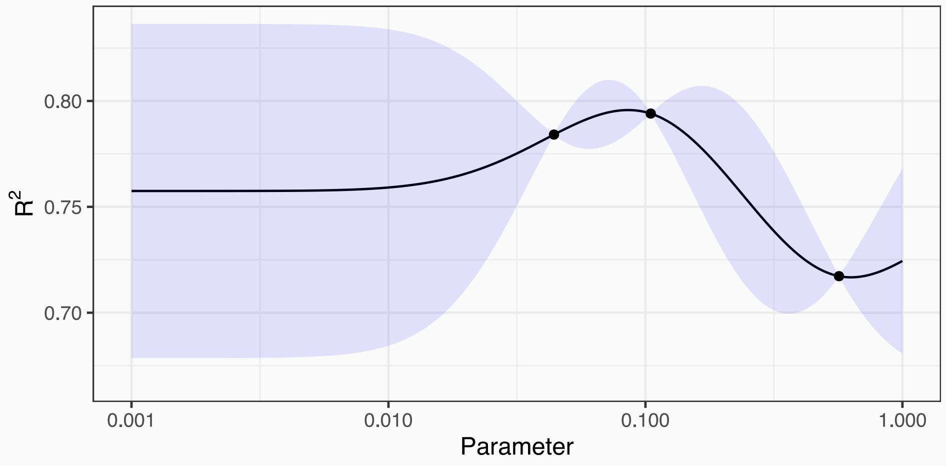
Since we want to maximize R^2, this function would seek to maximize the *lower credible bound*.

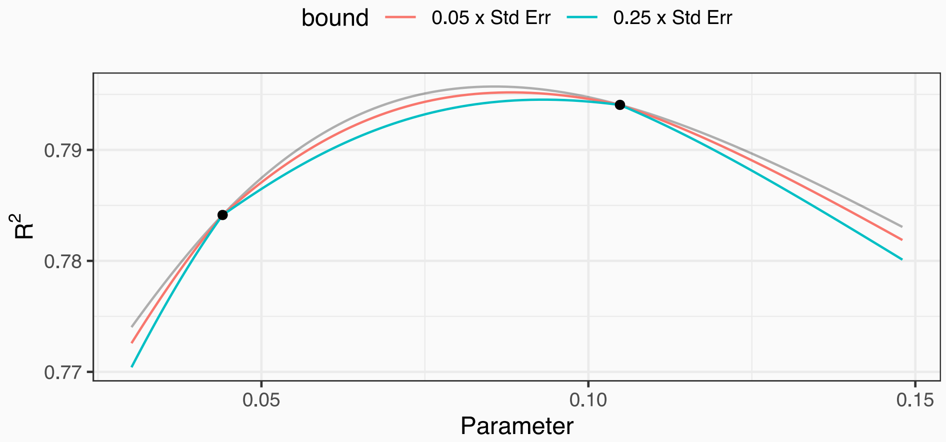
The size of the bound controls the trade-off:

$$bound = \mu(\theta) - C \times \sigma(\theta)$$

where θ is the vector of tuning parameters.

This isn't very helpful since it often mirrors the mean value.





Acquisition Function for Expected Improvement

One of the most popular methods is based on the expected improvement for new parameters, which is relative to the current best results.

Let's denote the best (mean) performance value was m_{opt} and assume that we are maximizing performance.

The expected improvement is calculated using:

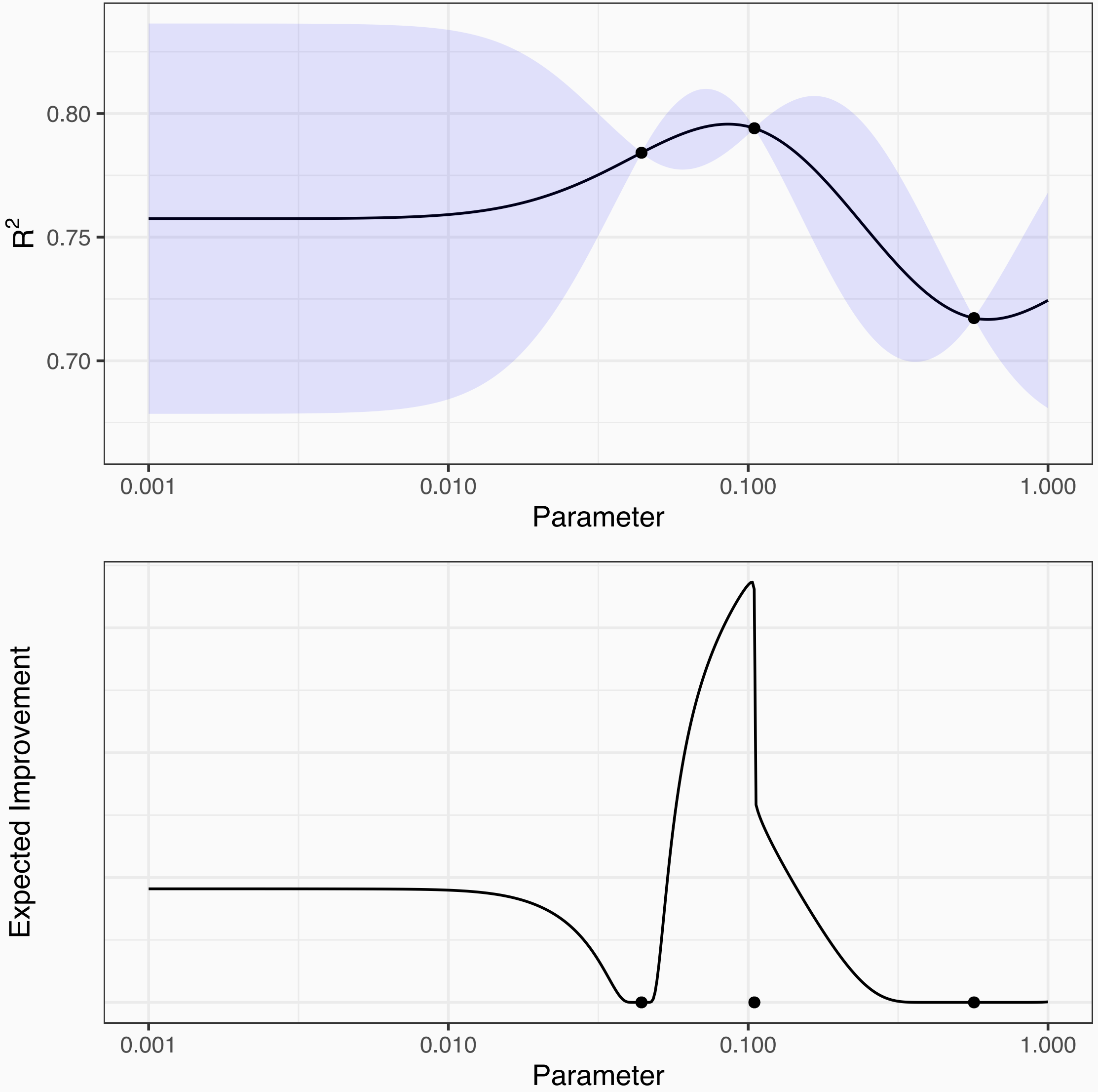
$$EI(\theta; m_{opt}) = \delta(\theta)\Phi\left(\frac{\delta(\theta)}{\sigma(\theta)}\right) + \sigma(\theta)\phi\left(\frac{\delta(\theta)}{\sigma(\theta)}\right)$$

$$\textbf{where}$$

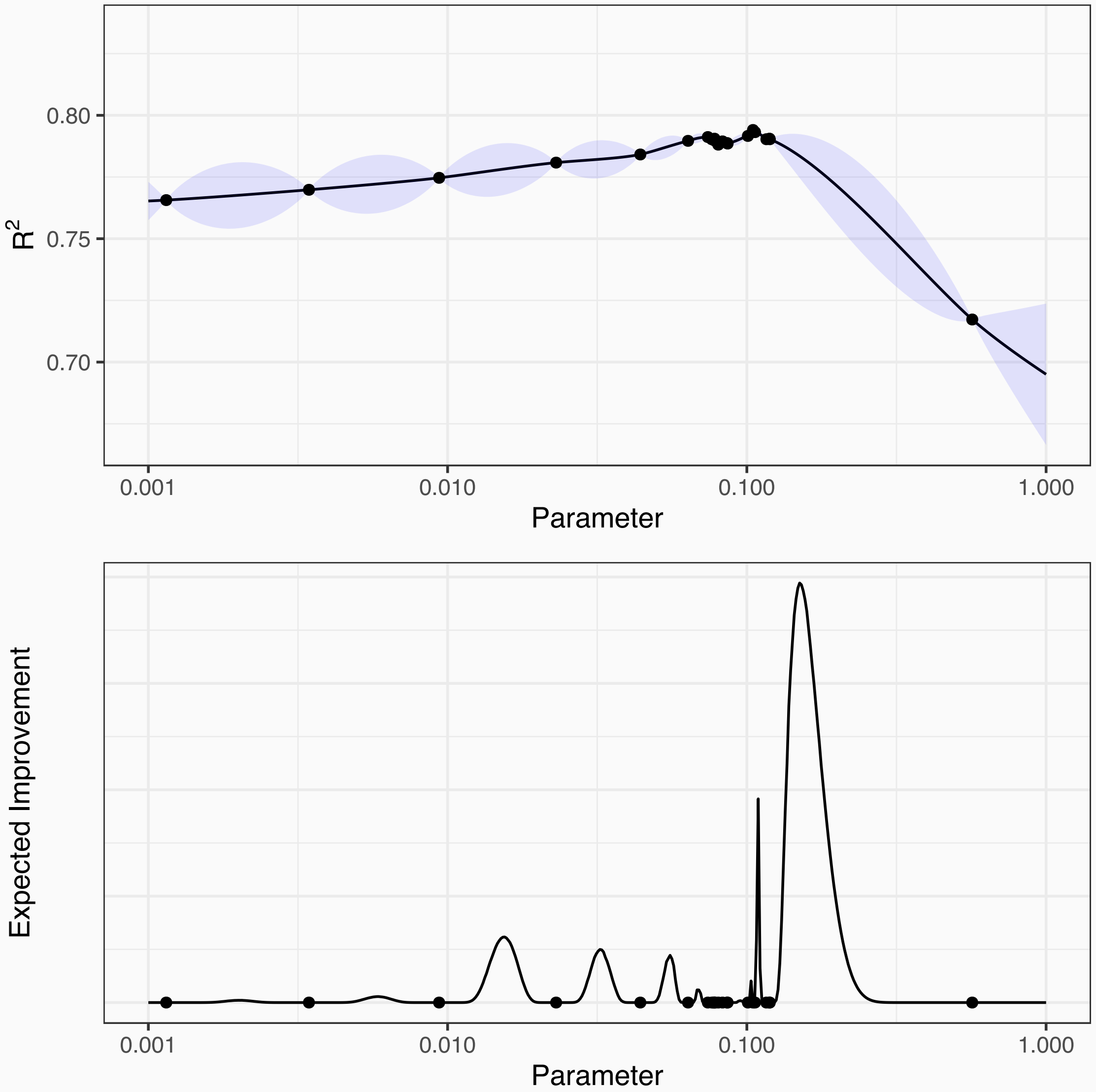
$$\delta(\theta) = \mu(\theta) - m_{opt}$$

The function $\Phi(\cdot)$ is the cumulative standard normal and $\phi(\cdot)$ is the standard normal density.

Acquisition Function for Expected Improvement



Expected Improvement After 20 Iterations



Exploration using Expected Improvement

The previous equation has a fixed trade-off between the mean and variance. One method to make the search explore more is to define a trade-off value:

$$\delta(\theta) = \mu(\theta) - m_{opt} - \tau$$

where τ is the amount of performance that we are willing to sacrifice (in the original units).

Larger values of τ will result in more novel candidate values. The value of τ can change over time so that exploration is the focus initially and exploitation is the goal in later iterations.

Another approach is to add *uncertainty samples*. This is an idea from the field of Active Learning where we add a new point that is most likely to help the model get better.

In this context, we sample a candidate value with very large variance.

In Summary

Baysian optimization is an iterative method for searching for reasonable tuning parameters.

It requires:

- The range/list of possible parameters (in the transformed scale)
- An initial set of resampled parameter results.
- The resampling scheme.
- A performance metric to optimize.
- An acquisition function.
- The maximum number of iterations.

We have a function called `tune_bayes()` for this.

Parameter Ranges

`tune_bayes()` can access the default ranges defined by the `dials` package.

For illustration, we'll change those ranges by adding the recipe and model to a *workflow*:

```
chi_wflow <-
  workflow() %>%
  add_recipe(mars_rec) %>%
  add_model(mars_mod)

chi_set <-
  parameters(chi_wflow) %>%
  update(
    `pca comps`  =  num_comp(c(0, 20)), # 0 comps => PCA is not used
    `mars terms` = num_terms(c(2, 100))
  )
```

This is an *optional* step.

A workflow is a new container that can bundle models, recipes, and other objects. It's still in development.

Running the Optimization

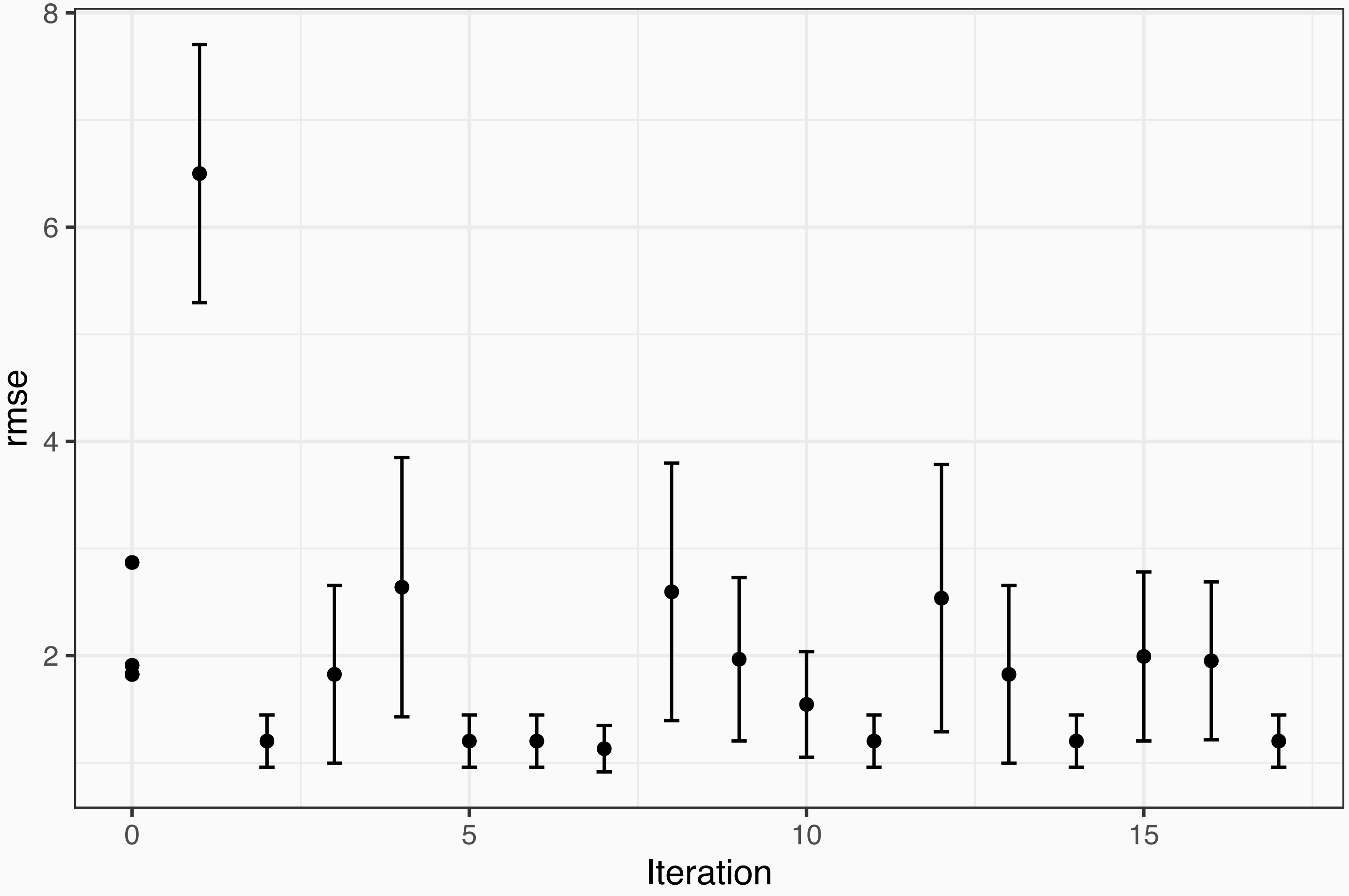
```r
library(doMC)
registerDoMC(cores = 8)

ctrl <- control_bayes(verbose = TRUE, save_pred = TRUE)

# Some defaults:
#   - Uses expected improvement with no trade-off. See ?exp_improve().
#   - RMSE is minimized
set.seed(7891)
mars_search <-
  tune_bayes(
    chi_wflow,
    resamples = data_folds,
    iter = 25,
    param_info = chi_set,
    metrics = metric_set(rmse),
    initial = 4,
    control = ctrl
  )
```

Example of Logging

```
❯  Generating a set of 4 initial parameter results
✓ Initialization complete

Optimizing rmse using the expected improvement

── Iteration 1 ──────────────────────────────────────────────────────────

i Current best:        rmse=1.29 (@iter 0)
i Gaussian process model
✓ Gaussian process model
i Generating 2877 candidates
i Predicted candidates
i mars terms=8, prod_degree=2, pca comps=0
i Estimating performance
✓ Estimating performance
ⓧ Newest results:    rmse=2.379 (+/-0.285)

── Iteration 2 ──────────────────────────────────────────────────────────

<snip>

── Iteration 4 ──────────────────────────────────────────────────────────

i Current best:        rmse=1.29 (@iter 0)
i Gaussian process model
✓ Gaussian process model
i Generating 2915 candidates
i Predicted candidates
i mars terms=100, prod_degree=1, pca comps=20
i Estimating performance
✓ Estimating performance
♥ Newest results:    rmse=1.074 (+/-0.0701)
```

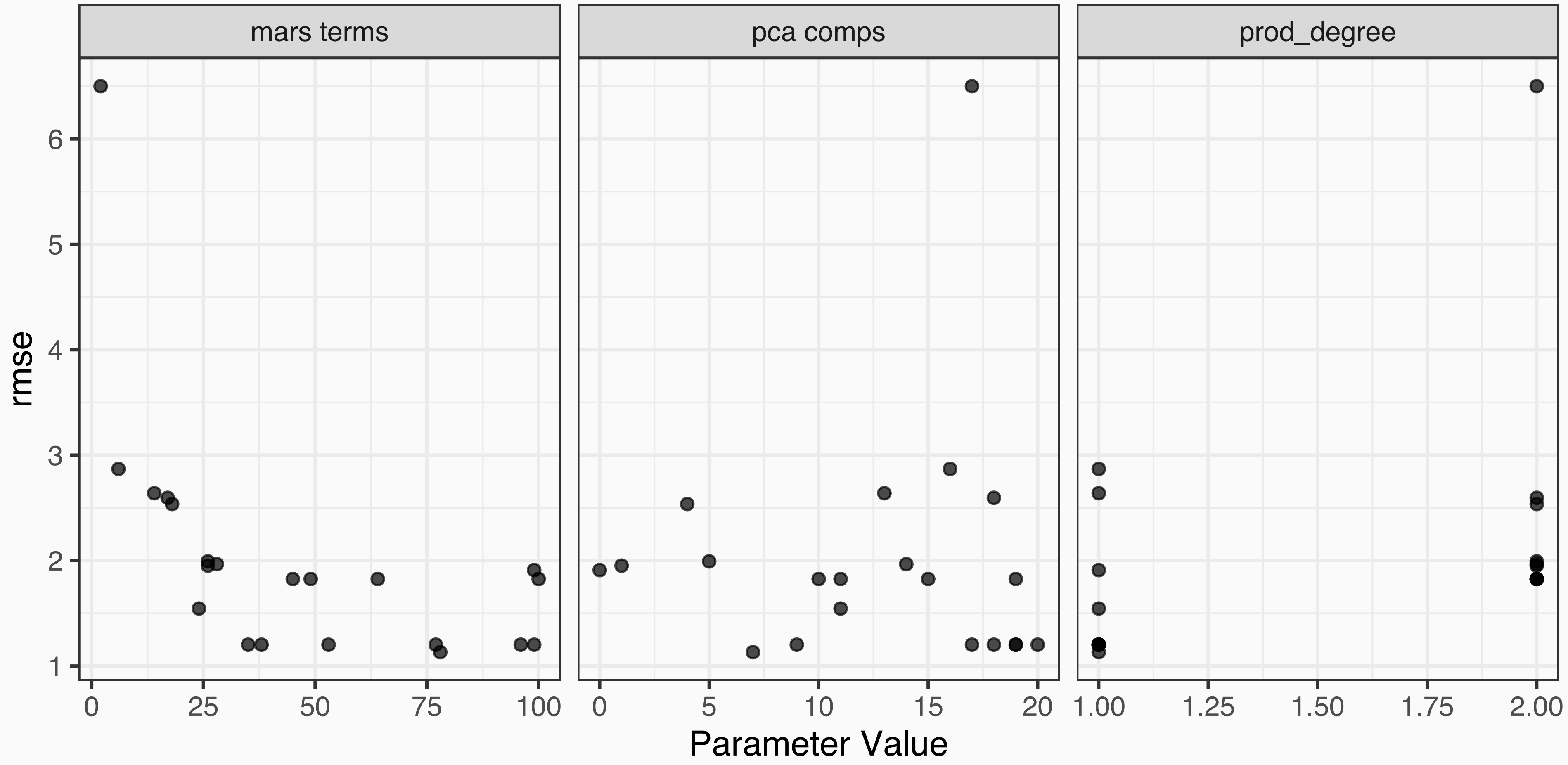
Performance over iterations

```r
autoplot(mars_search, type = "performance")
```



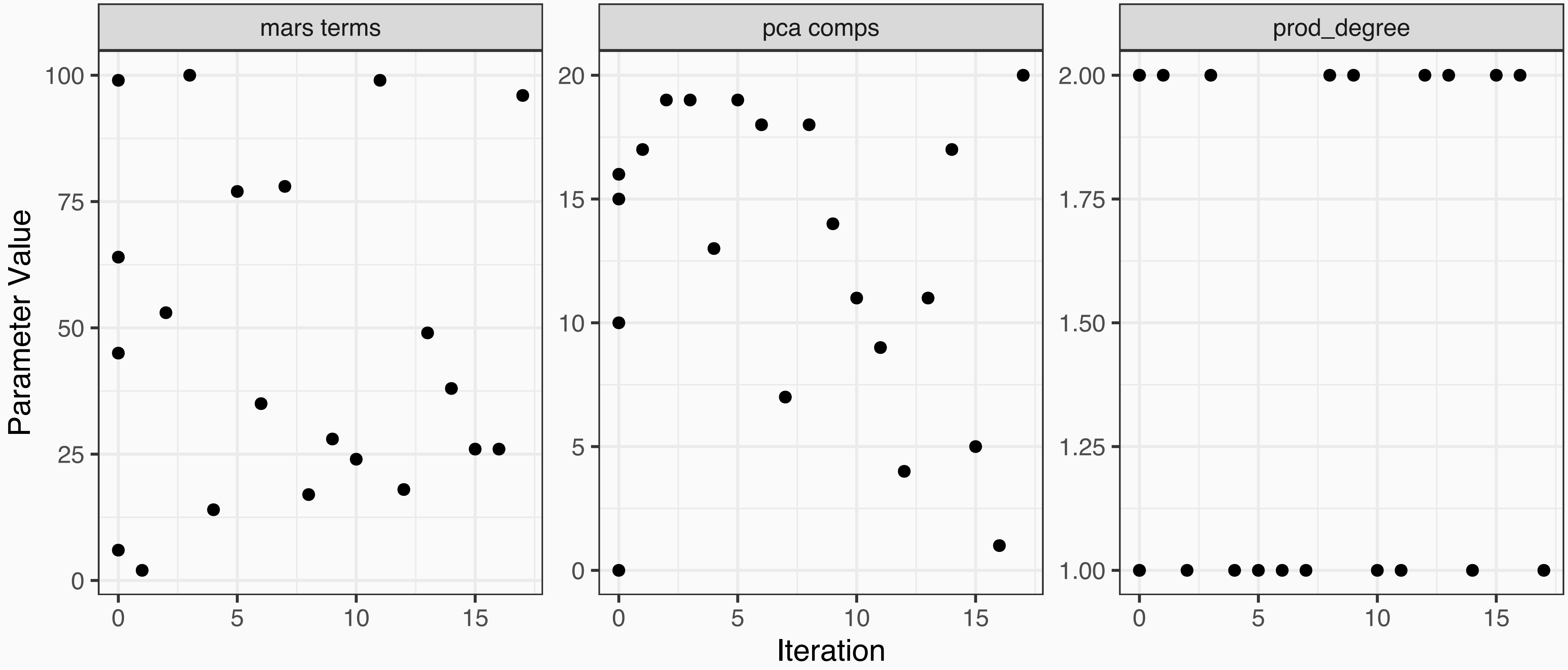
Performance versus parameters

```
autoplot(mars_search, type = "marginals")
```



Parameters over iterations

```r
autoplot(mars_search, type = "parameters")
```



Results

`collect_metrics()` and `show_best()` work the same here as with grid search:

```
show_best(mars_search, maximize = FALSE)
```

```
## # A tibble: 5 x 9
##   `mars terms` prod_degree `pca comps` .iter .metric .estimator  mean     n std_err
##          <int>       <int>       <int> <dbl> <chr>   <chr>      <dbl> <int>   <dbl>
## 1           78           1           7     7 rmse    standard    1.13     8  0.0940
## 2           35           1          18     6 rmse    standard    1.20     8  0.106
## 3           38           1          17    14 rmse    standard    1.20     8  0.106
## 4           53           1          19     2 rmse    standard    1.20     8  0.106
## 5           77           1          19     5 rmse    standard    1.20     8  0.106
```

For MARS, it has been my observation that decond-degree (i.e. non-additive) models can have better performance but also large variability.

These results about the same as the `glmnet` model (RMSE of 1.15 versus 1.131).

Performance Compared to Grid Search

For illustration, all 4158 possible sub-models for these three tuning parameters were assessed. We can use these results to see if the Bayesian optimization was effective.

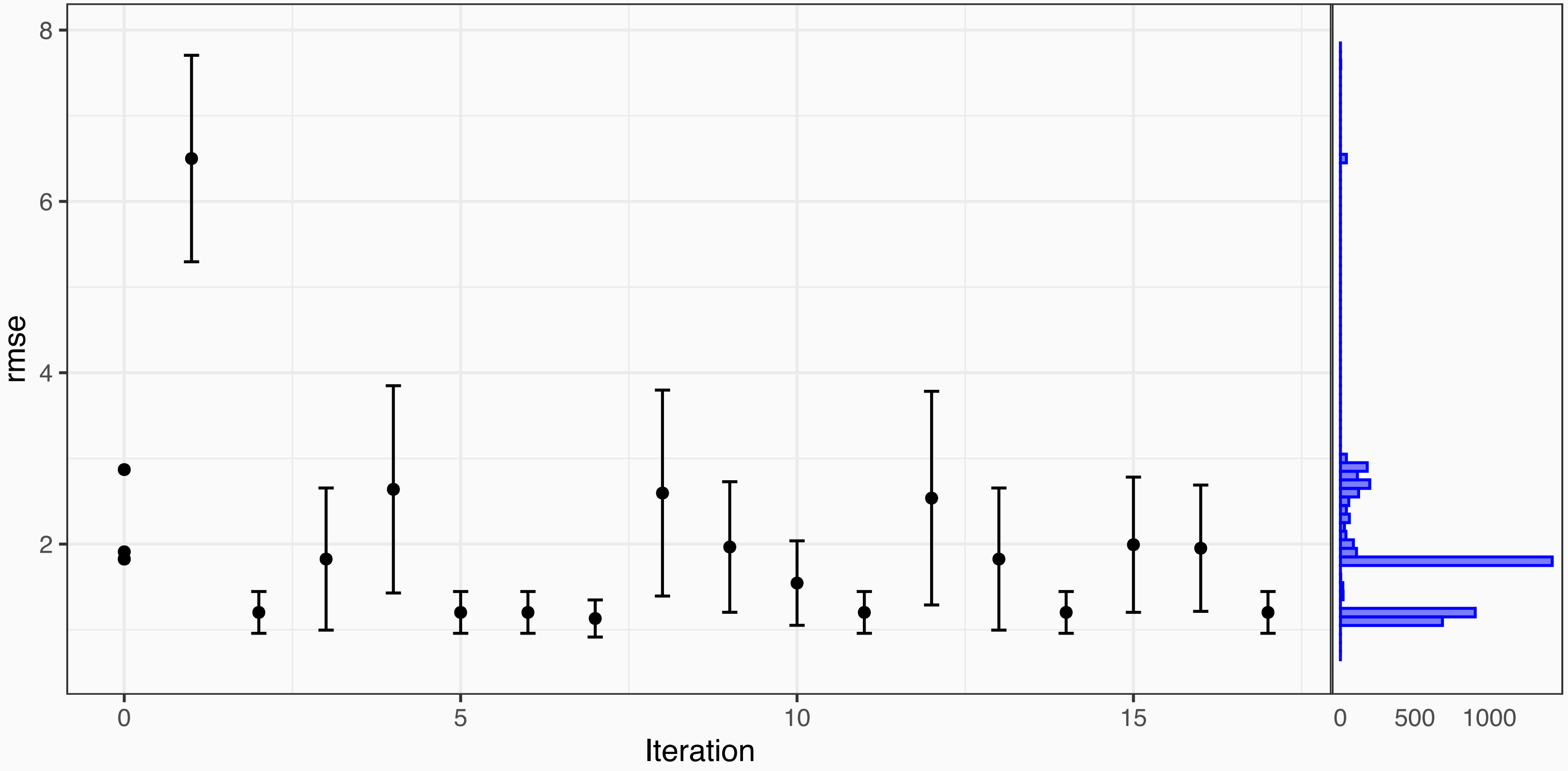
The initial set of samples (pre-GP model) had a minimum RMSE of 1.825.

The final search results yielded a model with RMSE = 1.131 which was better than 89.4% of the exhaustive grid search results.

What were the best results? They mostly used a lot of terms, less than 10 PCA components, and were additive (i.e. `prod_degree = 1`):

```
## # A tibble: 6 x 5
##   `mars terms` prod_degree `pca comps`  RMSE  rank
##          <int>       <int>       <int> <dbl> <int>
## 1           24           1           1  1.12     1
## 2           24           1           2  1.12     2
## 3           25           1           3  1.12     3
## 4           29           1           3  1.13    11
## 5           29           1           4  1.13    12
## 6           29           1           5  1.13    13
```

Performance Compared to Grid Search



My Thoughts on Bayesian Optimization

It is a reasonable approach to optimizing models.

It comes from the deep learning literature. While I believe the literature, I feel like the method is somewhat overfit to their problems.

For example, DL models

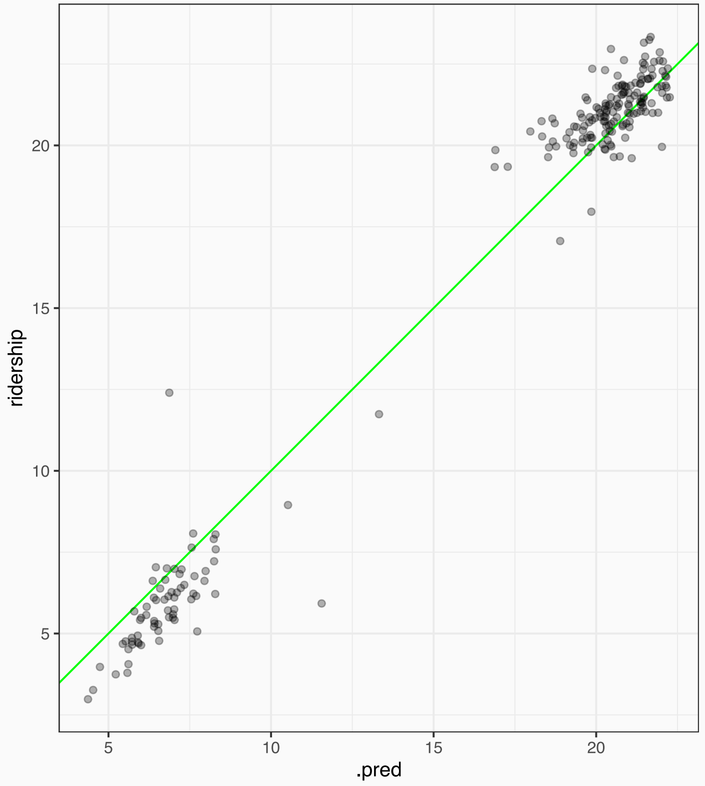
- tend to have far more critical parameters than many other models
- can't exploit sub-models and parallel process *single models*
- are created using massive data sets and a single validation set
- have well defined ranges of tuning parameters (and non-uniform priors on those)

Non DL models/problems tend to have performance profiles that plateaus and less *a priori* knowledge.

Also, space-filling designs do a much better job of finding good starting values than regular or random grids.

Assessment Set Results (Again)

```
mars_pred <-
  mars_search %>%
  collect_predictions() %>%
  inner_join(
    select_best(mars_search, maximize = FALSE),
    by = c("mars terms", "prod_degree", "pca comps")
  )

ggplot(mars_pred, aes(x = .pred, y = ridership)) +
  geom_abline(col = "green") +
  geom_point(alpha = .3) +
  coord_equal()
```



Finalizing the recipe and model

```
best_mars <- select_best(mars_search, maximize = FALSE)
best_mars
```

```
## # A tibble: 1 x 3
##   `mars terms` prod_degree `pca comps`
##          <int>       <int>       <int>
## 1           78           1           7
```

```
final_mars_rec <-
  mars_rec %>%
  finalize_recipe(best_mars) %>%
  prep()

final_mars_mod <-
  mars_mod %>%
  finalize_model(best_mars) %>%
  fit(ridership ~ ., data = juice(final_mars_rec))
```

Variable importance

MARS models can measure importance in a few different ways:

- The number of times that a column is used in a feature.
- The reduction in error when each term is added to the model (preferred).

The `earth` package has a generalized cross-validation (GCV) estimate for the latter metric.

```
vip(final_mars_mod, num_features = 20L, type = "gcv")
```