

DS 740

Data Mining

Revisiting Cross-validation for Model Selection

Consistent application via caret package

Important note: Transcripts are **not** substitutes for textbook assignments.



Learning Objectives

By the end of this lesson, you will be able to:

- Understand syntax and purpose of functions from the **caret** package.
- Apply cross-validation for model selection using the **caret** package.
- Compare different modeling methods and hyperparameters as part of model selection.



Cross-Validation Measure for Quantitative *Response*

Review: Measure for k -fold CV with quantitative data ("honest" MSE):

$$CV_{(k)} = \sum_{j=1}^k \frac{n_j}{n} \left(\frac{1}{n_j} \sum_{\{i \text{ in fold } j\}} (y_i - \hat{y}_i)^2 \right) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_{(i)})^2$$

Select model which minimizes $CV_{(k)}$ as "best."

Minor revision: transformed measure is RMSE ("root mean-squared error"):

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} = \sqrt{MSE}$$

Comment: another alternative produced is MAE, "mean absolute error" – reference for comparison between RMSE and MAE: <https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d>

When working with a quantitative response variable, the cross-validation measure, CV measure, is simply the Mean Square Error, or MSE, computed using the predicted value, or \hat{y} , from a model that was fit without the observation. Because the model was fit not using the observation that we are predicting, we sometimes refer to this as the honest MSE.

If we are picking among several models, we select the best model as the one with smallest CV measure. This is review from an earlier lesson, and we're also going to mention a minor adjustment to this measure. Specifically, we're going to talk about a transformation, the square root of the MSE from cross-validation. This will be referenced as RMSE standing for Root Mean Square Error. It is monotonically related to MSE, which means that a model with smaller MSE will also have a smaller RMSE when compared to a second model.

The reason we use this method or this measure is that it tends to have magnitudes that are a bit more sensible in terms of a magnitude comparison but does not change which model is selected. Looking ahead to our use of Caret package, another measure that will be produced as part of the output is called MAE, standing for Mean Absolute Error. As we will not focus on its application, you may optionally read a compare-and-contrast article between our RMSE and MAE.

Notes:

Comment: another alternative produced is MAE, "mean absolute error" – reference for [comparison between RMSE and MAE](https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d).

Cross-Validation Measure for Qualitative *Response*

Review: Measure for k -fold CV for classification (percent inaccuracy):

$$CV_{(k)} = \frac{1}{n} \sum_{i=1}^n I(y_i \neq \hat{y}_i)$$

Select model which minimizes $CV_{(k)}$ as "best."

Comment: another alternative produced is "Cohen's Kappa":

- Briefly, equals 1 for a perfect model, 0 for a model that's randomly guessing;
- can be negative;
- better than % accuracy when the response variable is highly imbalanced.

Further description here: <https://towardsdatascience.com/cohens-kappa-9786ceceab58>

When working with a qualitative response variable, the cross-validation measure is the percent inaccuracy in the predicted classification-- in other words, the percent misclassification. This is, again, based on a model that was fit without using the observation to be predicted.

If we are picking among several models, we select the best model is the one with the smallest CV measure. That is, the smallest cross-validated measure of misclassification percentage. We will typically use the misclassification from cross-validation, but it is worthwhile to briefly mention another measure that will be produced as part of the output summaries called Cohen's kappa. This second option is preferred when the response classes are highly imbalanced. An optional reading is provided at the end of the slide.

Notes:

[Understanding Cohen's Kappa coefficient.](#)

Applying Cross-Validation: Review

- Set up **folds** for cross-validation
- For some methods (e.g. **linear models** and **kNN**), must write a **for** loop
- For other methods (e.g. **penalized regression**), can use **cv** function

```
nfolds = 10
groups = rep(1:nfolds,length=n) #produces list of group labels
set.seed(8.1)
cvgroups = sample(groups,n) #orders randomly
```

```
Model1 = (BodyFatSiri ~ Weight+BMI+Height) #dimensions
Model2 = (BodyFatSiri ~ . - Case - BodyFatBrozek - Density) # all
allLinModels = list(Model1,Model2)
for (m in 1:mLin) {
  allpredictedCV = rep(NA,n)
  for (i in 1:nfolds) {
    groupi = (cvgroups == i)

    #prediction via cross-validation
    lmfittCV = lm(formula = allLinModels[[m]],data=bodyfat,subset=!groupi)
    allpredictedCV[groupi] = predict.lm(lmfittCV,bodyfat[groupi,])
    allmodelCV[m] = mean((allpredictedCV-bodyfat$BodyFatSiri)^2)
  }
}
```

```
library(glmnet)
x = model.matrix(BodyFatSiri ~ . - Case - BodyFatBrozek - Density,data=bodyfat)[-1]
y = bodyfat$BodyFatSiri
cvLASSO = cv.glmnet(x, y, lambda=alllambda, alpha = 1, nfolds=nfolds, foldid=cvgroups)
allmodelCV[(mLin+mKNN+1):mmodels] = cvLASSO$cvm[order(cvLASSO$lambda)]
```

Compare CV measures, select model with *lowest* CV / metric

Up to this point, we have computed CV measures in multiple ways. Often, we had to write a for loop that splits the data and then repeatedly fit the model to the training set and predict the test set. Other methods have applicable cross-validation functions included, such as `cv.glmnet` for penalized regression.

Regardless of how we obtain the CV measures, we select the best model according to our criterion which looks for the lowest CV value. We are selecting the model that makes the honest error as small as possible.

Applying Cross-Validation via Caret Package

- Classification and Regression Training `library(caret)`
- Unified package for doing cross-validation of many machine learning methods, using consistent syntax
- Use the trainControl function to **define training method**

Caret uses random numbers → differently than base R

```
set.seed(8.1)
training = trainControl(method = "cv", number = 10)
```

- Cross-validation using caret (e.g. **linear models**):

1. form
2. data
3. method
4. trControl
5. tuneGrid (hyperparameters)
6. (optional arguments)

```
fit_caret_lm1 = train(BodyFatSiri ~ Weight+BMI+Height,
                      data = bodyfat,
                      method = "lm",
                      trControl = training)
fit_caret_lm1
```

We can perform that same process but now using a streamlined application via the Caret package. Caret stands for Classification and Regression Training. And this package provides wrapper functions for various model-fitting methods. We use a function first called traincontrol to set up a kind of training process we want to do. In this lecture, we are specifying cross-validation with 10 folds and storing that in the output called training.

We next use the train function to apply this process to the appropriate model-fitting method. Here, we are using lm for linear regression modeling, along with specification of the model and data frame. We must also reference the training process in the trControl argument. There are optional arguments. If we have thicker parameters, we can use tuneGrid and other arguments that will allow us to, for example, do scaling.

The output of the cross-validation training process applied to, in this case, a linear model will then be stored for further use, and here we store this as fit_caret_lm1. Note that this cross-validation training process can be applied now to other model-fitting methods as well, such as a different linear model, or we'll also try penalized regression models. And various other methods can be used as well.

Cross-Validation Measures From Caret

- Cross-validation using caret (e.g. **penalized regression**):

1. model
2. data
3. method
4. trControl
5. tuneGrid (hyperparameters)
6. (optional arguments)

```
alllambda = exp(-100:20/10)
fit_caret_LASSO = train(BodyFatSiri ~ . - Case - BodyFatBrozek - Density,
  data = bodyfat,
  method = "glmnet",
  trControl = training,
  tuneGrid = expand.grid(alpha=c(1), lambda=alllambda))
fit_caret_LASSO
```

Hyperparameters

RMSE (and other measures)
may be **slightly different**, as if
using a different random seed

```
> fit_caret_LASSO
glmnet

252 samples
17 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 226, 227, 228, 226, 227, 228, ...
Resampling results across tuning parameters:
```

lambda	RMSE	Rsquared	MAE
4.539993e-05	4.499710	0.7233437	3.658772
5.017468e-05	4.499710	0.7233437	3.658772

We next apply this very similarly with the model-fitting method of penalized regression. Here, we specify our hyperparameters, alpha and lambda values, via the tuneGrid argument. This will compute cross-validation measures for all different specified combinations of our hyperparameters.

As part of the output, we see that RMSE is automatically computed, as are a couple other measures. If we are selecting among multiple models, we can use this RMSE measure to choose the model with the smallest value. Note that Caret uses random numbers differently than base R, and thus these measures, RMSE and MAE and R squared may be slightly different and work as if you're using a different random seed.

Viewing the Best Measure and Fit (Values)

After applying the train function, we can view

- Values of all measures: `fit_caret$results`
- Just the measure of interest: `fit_caret$results$RMSE`
- The best parameters (as chosen by measure): `fit_caret$bestTune`
- The final model: `fit_caret$finalModel`

```
> min(fit_caret_LASSO$results$RMSE)
[1] 4.484314
> fit_caret_LASSO$bestTune
  alpha  lambda
82     1 0.1495686
> fit_caret_LASSO$finalModel
```

Once we have done the training, we can take a look at various values on the output. For example, if we wanted to take a look at all the different measures for various hyperparameters, we can take a look at results as a value peeled off from the output. And we can further peel off just the measure of interest, such as RMSE, and the best parameters can be viewed by bestTune.

In the situation of a simpler model such as `lm`, we can take a look at the final model with that value `final model`. However, in the situation of penalized regression, the output version of that will not look very nice. We can, however, use that for further applications as we'll see in the next lesson.

Applicable Methods

- Functions from the caret package are widely applicable to numerous modeling methods – see <https://topepo.github.io/caret/available-models.html> for a full review
- Most methods (covered thus far) can be implemented via caret package
- Compare (via appropriate measure) numerous models from different methods – must have response on same scale

There is a large set of methods that are available to be used within the Caret package. The link here gives a review of all available methods. I will mention that it is a substantive list, noting that there are some methods that appear through different fitting functions, as well as some combinations of methods in there.

Most of the methods covered thus far in this class can be used inside the Caret package. Syntax will differ, of course, as depending on the hyperparameters and the other specifications we need to make. We'll see an example of that with k-nearest neighbors. An important understanding is that if we wish to compare models fit and assessed from different methods, we must be utilizing the response on the same scale so that the measures that we're using are comparable.

Notes:

Functions from the caret package are widely applicable to numerous [modeling methods](#).

Body Fat Example Revisited

Using `bodyfat.csv` data set, with author's description and adjustments.

Response: *BodyFatSiri* (using Siri's equation) - quantitative

Predictors: various body measurements – all quantitative

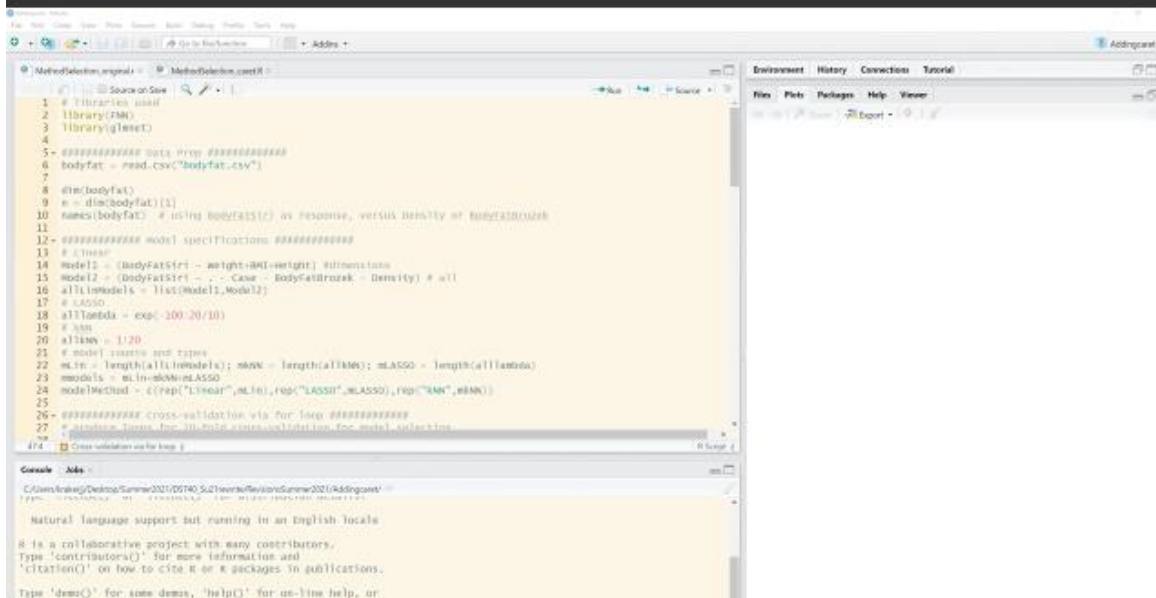
Models considered:

- Model 1 is a linear model with 3 predictors only.
- Model 2 is a linear model with all 14 predictors.
- Models k1, k2, ... k20 are kNN models (using all predictor variables) with 1, 2, ... 20 nearest-neighbors, respectively.
- Models p1, p2, ..., p121 are penalized regression (LASSO) models, with increasing values of penalty.

We revisit the body fat data example. We'll consider fitting a total of 143 models; two different linear models, one with all the predictors and one with just a subset of the predictors; 20 k-nearest neighbors models with varying numbers of nearest neighbors; and 121 LASSO models with varying values of the penalty hyperparameter λ .

We will then compute RMSE for each model. We'll do this first using our prior applications for cross-validation and then show the streamlined version using the `Caret` function.

DS740 - Cross-Validation for Honest Prediction & Model Selection



```
1 # Libraries used
2 library(rsk)
3 library(glmnet)
4
5 ##### Data Prep #####
6 bodyfat <- read.csv("bodyfat.csv")
7
8 dim(bodyfat)
9 n <- dim(bodyfat)[1]
10 names(bodyfat) # using bodyfat[12] as response, versus density of bodyfat[1:11]
11
12 ##### Model Specifications #####
13 # Linear
14 Model1 <- (bodyfat[12] ~ weight + BMI + weight) # dimensions
15 Model2 <- (bodyfat[12] ~ 2 * Case + BodyFatTrozak + Density) # all
16 allModels <- list(Model1, Model2)
17 # LASSO
18 allLambda <- exp(-log(100:20/10))
19 # kNN
20 allKNN <- 1:20
21 # model names and types
22 mLn <- length(allModels); mKNN <- length(allKNN); mLASSO <- length(allLambda)
23 models <- mLn+mKNN+mLASSO
24 modelMethod <- c(rep("Linear", mLn), rep("LASSO", mLASSO), rep("kNN", mKNN))
25
26 ##### cross-validation via for loop #####
27 # random loop for 10-fold cross-validation for model selection
```

Console - Jobs

```
C:\Users\Aravind\Desktop\Summer2021\DS740\Summer2021\Revisions\Summer2021\Addingpart1
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
```

The first part of this recording will go over a summary of method selection using cross-validation as we've applied it in prior applications. To start with, I am going to open up the libraries that I will need for k-nearest neighbors and for lasso fitting. I'm going to make sure that I have access to the file, which is stored in the same location that this dot R file is stored, and I'm going to do my usual quick review of the variable names in the data frame.

A reminder that I'm going to be using BodyFatSiri as the response, and thus I want to only use one of those four first variables. The remaining 14 variables are going to operate as my predictors or potential predictors. I'm next going to go in and specify linear models, lambda parameters, and K nearest neighbors parameters, as we have done in the past. So this is all review stuff, and we're then going to count up how many of each of these items exist for a total number of models labeled as n models. And I'm also going to put together a vector that says model method.

So if I take a look at this, n models tells me I have a total of 143 models, and that sounds a lot, but I'm specifying a different model for each of my different potential lambdas. And since I used a lot of lambdas, because they're frankly pretty cheap to fit, that means I have a lot of potential lambda values. Model method lists off all those different types of models for my 143 models.

And then going to set up cross-validation for model selection via for loop, and this is what we have been doing throughout the last several lessons, where we've taken a look at the CV value as measured for a particular cross validation process and used that to select among a set of potential models. All right, so I've also set up storage for the CV values, and finally I am going to run through a cross-validation computation for each

type of model. But because with our CV process, this is actually rather complex, rather individualized for each different type of modeling method.

We have to do a separate computation for each type, so for m in 1 up to m_{Lin} I am going to do an actual for loop, and we've done that a handful of times now. For lasso models, I have this `cv.glmnet` function, which does cross validation for me, so I'm going to use the CV groups I set up here at the beginning.

Specifying my x , my y , fitting my cross-validation, and then pulling out the CV values in the correct order and storing them among my all CV values. Now, if I pause here and take a look at `allmodelCV`, you'll notice that I have filled in the two CV values for my linear model fits, and it looks like 121 values for my various lasso fits. And I still need to fill in my CV values for my kNN fits.

All right. In order to set up my kNN models, I do want to use standardization, as we talked about all the way back in lesson one. And so that involves not just a for loop but a for loop in which we have to do standardization scaling of the x data as computed from the training set. So we go ahead and do that, and we take a quick look at what is finally now stored in `allmodelCV`. And we see that it is complete. It has finished by pulling in the CV values from our kNN model fits for cross-validation.

Finally, we're going to take a look at which of these is best. So, quite literally, we're going to look for which is the min of all our CV values. And so if I take a look here, linear model is not best, kNN is not best, but lasso is best with the specific parameter selected here.

So what I did is a little if-else check that said is the method at the order location of the minimum of CV, and so it's worthwhile checking out what is `order.min`, the 66th value. The model method-- remember model method is this full string or vector of strings, and I'm going to take a look at the 66th value, which is, of course, going to be one of the lassos. And so this is just a way of pulling out the specifications of the best model, either the linear model itself, the parameter for k -nearest neighbors, which is the number of nearest neighbors, or the λ for the Lasso parameterization. Well, again, I find that the best model selected by this process is lasso I'm going to go ahead and fit that, again, as we have done in the past and pull off the coefficients.

A useful thing that could be done to complete this is, well, first to take a look at the root-mean-square error, and since the CV measure that we're using for a numeric response is just the honest mean-square error, we can take the square root of that and call that RMSE. And so I'm going to put that into a data frame side by side with the model methods we used and then pick out some different colors, and I'm going to assign those colors according to the modeling method.

So this effectively, if I wanted to run just the section, gives a numeric label to each of my modeling methods rather than the original text, and so I assign a color used according to that, assuming I actually run that line. Make sure that I ran the results as well. Similarly, I am going to take the `as.numeric` factor model method, and that's my numeric label for each of the methods, and I'm going to use some different characters. If I define this, you'll see that `charused` equals 15, 15, 10.

There's other ways of handling this. I could have picked out a different set of labels and did a subset, for example, but this turned out to be a way that gives me three different character-plotting labels that I like. So I'm going to plot the RMSE against the model label different colors and plotting characters according to the model method being used, and then I'm going to draw in a vertical line at the location where that minimum RMSE, or, equivalently, the minimum CV measure occurred. So that occurs, as we noted previously, at model labeled-- well, it's the 66th model in our list, and it's one of our lasso, which we can see as these blue sort of star-shaped characters. And the next thing we'll do is apply this process in a smoother fashion using `Caret`.

Selection via Cross-validation + Model Fit

Model selection **must** be considered part of the full model-fitting process.

1. Models $m = 1, 2, \dots, M$ from which to select.

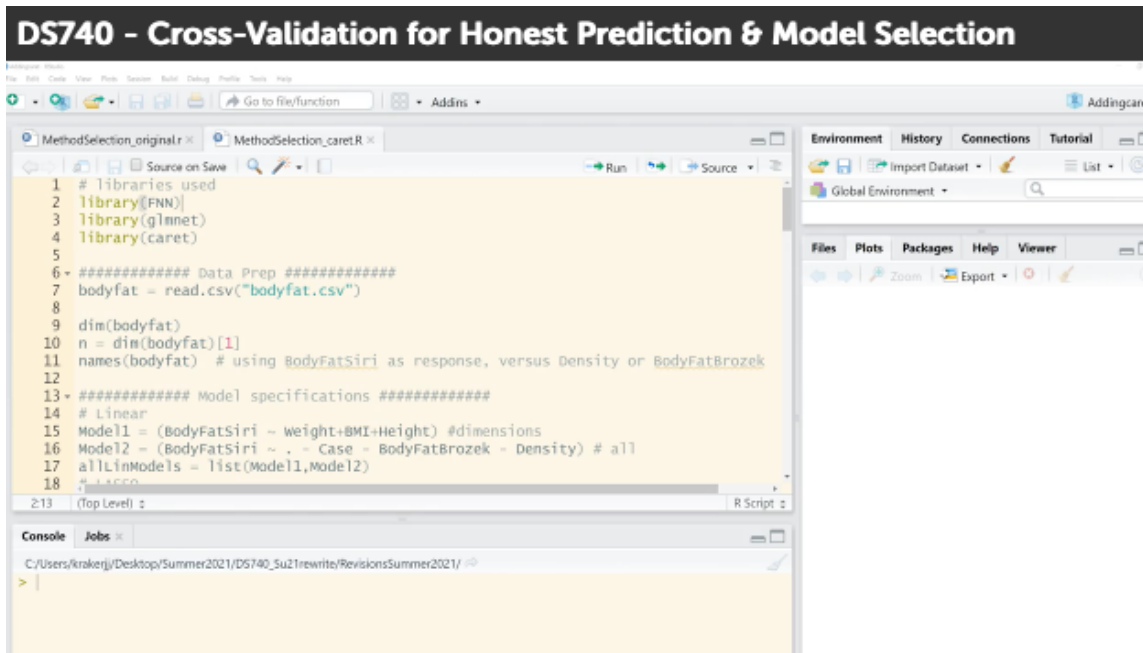
2. **Internal cross-validation (can be applied using caret package):**

Split data randomly into k sets. Fit each model on all k training sets, predicting data in corresponding testing set and compute $CV_{(k)}$.

3. Select model which minimizes $CV_{(k)}$ as "best" – fit this selected model on **all** the data for final fitted model.

Assessment of this process: upcoming lesson

Since we use model selection to reach our final fitted model, that becomes intrinsically part of the model-fitting process. And so if we start with a set of m models from which to select, whatever process we go through, such as cross-validation, to select the best model, we will then fit the best model on the full data set once we have selected it. And we then further need to assess how well that entire process works. And that will be covered in our next lesson.



We now look at the application of cross validation for model selection using the functions from the caret package. So in opening up the libraries used, in addition to the previous two, we'll also use the caret package after we set our working directory. We read in our data as before.

Take a look at the dimensions and the data frame names. The model specifications are all the same, as what we talked about with our usual cross validation application. And it is in the specification of the training method, as well as the actual computation of the cross validation measures, where things change up a bit.

So in here, we're going to be using the full data set. So we're going to specify data used equals body fat. And while this might seem like a superfluous step in this section, it will become relevant in the next presentation in this lesson. We set the seed before specifying our train control method to be cross validation. So method equals CV. Number equals 10 specifies the number of folds.

And we then separately fit each model with its variety of hyperparameters. With our linear models, we had two of them. We do have to fit those separately. So we don't have a way of specifying the specific subsets that we're interested in fitting.

So in the storage fit_caret_lm1, we are going to train our linear model 1. And so we can either specify the full model, or we could have put in model 1 here. The reason for writing out the model in full is for continuity with later methods. Specify the data, method for fitting, and how we are doing the training. And that's it. We run that.

We can take a look at the output. And the important feature that we're going to be tracking throughout this is the RMSE. Remember, that's the Root Mean Square Error for the honest errors. And so that will be kept track of stored in among the output from this linear model cross validated fit, and we'll track that at the end.

Same thing with using all 14 predictors. That is, modeling the response on everything except the unwanted information, which is comprised of just the simple case number, as well as the other two potential responses. Again, we can take a look at the output. There is not a lot here with the linear regression. Again, the big thing to keep in mind is the summary measures of that fit.

Lasso is where we have to add a little bit in. So one of the potential arguments that we can use in this function after specifying the model, the data used, the method-- which we're going to be using glmnet-- the training process that we're going to be applying-- which was the CB with 10 folds-- we additionally want to specify a grid of 10 parameters. And if we were fitting an elastic net, we would be fitting all our lambda values for all possible alpha values.

In this case, we just have one alpha value. Alpha equals 1, which specifies the lasso. And we're going to be looking at all lambdas, along with an alpha of 1. The generic function for doing that crossing of the different parameters is called `expand.grid`.

So again, we run this. This one's going to take a little bit longer to run. And in addition to this, this particular application runs into difficulty because of, if we take a look at the output, there is an N-A-N here. And this has to do with basically overfitting or calling on a lambda that shrinks too much.

And so I'll go back to the beginning here and reset all lambda. Instead of going up to 20, let's just go up to 15. Again, just having recalled from our first run of this, we really didn't need to worry about any of the much higher lambdas. So trimming off a few of those will not affect our best selected model.

So with that in mind, I am going to run this command, run the set of commands again for our linear regression fits and then our Lasso fit. And now it did not produce or did not result in an error. And we can see when we take a look at the output that-- again, having to scroll up in our output-- this tells us that appropriate measures were able to be produced for all lambdas that we defined and that we included as potential hyperparameters.

Now, I might just-- there's a bit of output here. I might just want to take a look at the metrics. So I take a look at the results. And this spells out my alpha values. Well, all my alpha values are 1s. All my lambda values, all now 116 of those. My RMSE measures r^2 squared MAE and MAE as well, as well as the standard deviation associated with those.

So I may, since I'm focusing on my RMSE, just wish to select the RMSE from off these results. And so I can further pull off the RMSE values. I can find the minimum of those values, which is connected with the best tune that is the set of best hyperparameters.

And you might think when I take a look at my final model, well, it'll just give me some nice coefficients or some sort of summary. But this one's a little bit tricky because it actually concurrently fits all the different lambdas. And so the final model is actually quite horrendously messy. And so, while we can use the final model in terms of further predictions, actually taking a look at it and finding specific meaning from this output is not going to be something we'll do with this specification.

So our last method that we're going to use is the K nearest neighbors. So again, we want to apply the training function to our model fit on all potential predictors, specify the data, the method of knn, our training process. And preprocess equals center and scale allows us to standardize the data in this cross-validation.

And so when we had to do this early on in the first and second lessons, it was a little bit more complex to do that inside a for loop. All we have to do is specify preprocess center and scale to have the training function in the caret package automatically do that for us. And we're going to specify the hyperparameters as expanding the grid of only our one set of values. K is our list of potential K nearest neighbors.

Now, when we take a look at the output here, we get a similar set of measures. Like I did up here for the Lasso, I might want to take a look at best tune or final model. So we could add in best tune. Tells us that 13 is the best number of nearest neighbors.

Or, final model. Well, as it turns out, the model specification is a little bit difficult to summarize. And so it just gives us a very simple summary saying, we're going to take the 13 nearest neighbors from the training set that is for an observation in our validation set. We would take-- or in our testing set, we would take the 13 nearest neighbors from the training set to predict that observation in the testing set. But it can't produce a nice closed formed model for us to describe.

So, going through this, if we go back and count, we really had one `fit_caret_lm1`, `fit_caret_lm2`, `fit_caret_lasso`-- that's our third-- and `fit_caret_KNN`-- that's our fourth-- applications of this function. The first two were linear. The next was Lasso. The final was KNN. And each of them produced the best parameters or some specification of the model.

So all best parameters, number of predictors is one of the things that we could-- predictor variables is one of the things that we could specify for the linear. `Fit_caret_lasso`. Pulling off `bestTune` tells us the alpha and lambda that are best. And so I want to put this as my third item. But it contains two values.

So I'm actually going to make a list here because a list can contain differently sized items, and even different type of items. And that becomes important if we want to keep track of all our best models. So, pulling off the final model from each of these first three, for KNN, we're just going to include the full model fit. And I'll talk about that more in the next lesson.

The all best RMSE, remember that there's only one model that we cross validate for linear model 1, one for linear model 2. But with the Lasso, we're actually picking between eventually 116 different models. So we want to get the smallest. That is, the minimum value of the RMSE. And same thing for K nearest neighbors. Were picking among 20 of those.

So finally, after we've defined all best types, all best pars, all best models, all best RMSE and kept track of them and stored them, we want to pick out one of those four best models that is ultimately the best across all of the methods. And I can do that by picking the smallest criterion value-- the smallest RMSE. I had four values of RMSE stored here. We maybe want to check that out. Four values.

And just tracking through this, it looks like the third of those corresponding to Lasso is the smallest. So if I take which is the smallest from among all best RMSE and pick `all_best_Types` at that location, that will be my one best type. And similarly, `one_best_Pars` and `one_best_Model`. If I take a look at the one best type, it's Lasso, which we kind of already knew.

`One_best_Pars` is the alpha of 1 and a lambda of 0.0223. And as we said before, one best model is not going to be convenient to view. So we're not going to specifically look at that, but we will be able to use that in prediction.

I may very well wish to take a look at the best model. So I'm going to pull off from `one_best_Pars`. I'm going to take a look at the first element on what turns out to be a list type, and then pull off lambda from that. And I'm going to store that as my `LASSOlambda`. And as we did back in lesson 5, we can use that lambda to produce the coefficients using this one best model as fit to our data, or as selected from among our cross validation fit to the data.

And after I define `LASSOlambda`, I will also get `LASSOcoef`. And so, now I can take a look at those `LASSOcoef`, noting that BMI surprisingly is actually removed. But it's removed after including weight and height. Knee unsurprisingly is removed. But all the other predictors are retained in the final Lasso model. And so you will see the statement, at least a part of the final model in your lesson presentation.

All right. So this last part is simply a bit of a visual to try and explain how these four different applications of the train function really got produced results that were narrowed down to the best model. So, specifications of the number of models as

before. And then, keeping track of all RMSE is done in the first few lines. I then plot these values.

And I'll note, a very-- well, a different way that we used the caret function as compared to our prior process. With the caret function, we found, we specified the best model of each of the four types. So the best. We only had one linear model 1-- the best. Only one linear model 2. And then the best of all the last models was identified, and the best of all the KNN models was identified.

Out of those we finally narrowed it down to the one best model among all the different 1 plus 2 plus 115-- 116, excuse me-- plus 10, 118, 138 different models and we picked up the one that produces the minimum RMSE criteria. The process by which we did that in summary was very consistently applied by using the train function with slightly different specifications, but very similar for all our methods.

Thanks for your time. And I hope this helped give you an introduction to the train function in the caret package.

Selection via Cross-validation + Model Fit

Full model-fitting process: in example

1. Models $m = 1, 2, \dots, M$ from which to select.

Linear regression:

Model 1 with *Abs Weight*, and *Height*

Model 2 with fourteen predictors

Penalized regression (LASSO):

Start with $\alpha = 1$ and consider

116 different possible λ -values

k-Nearest Neighbors (kNN):

Consider

20 different possible values of k

2. Internal cross-validation (can be applied using caret package):

Split data randomly into k sets. Fit each model on all k training sets, predicting data in corresponding testing set and compute $CV_{(k)}$ – see [MethodSelection_caret.R](#).

3. Select model (from Linear 1, Linear 2, best LASSO, best kNN) which minimizes RMSE as “best” – fit this selected model on all the data for final fitted model.

```
> options(scipen=999)
> round(LASSOcoef,4)
```

Lowest cross-validated RMSE value is 4.505, for best LASSO model with $\lambda = 0.02237$.

Model fit to all data results in:

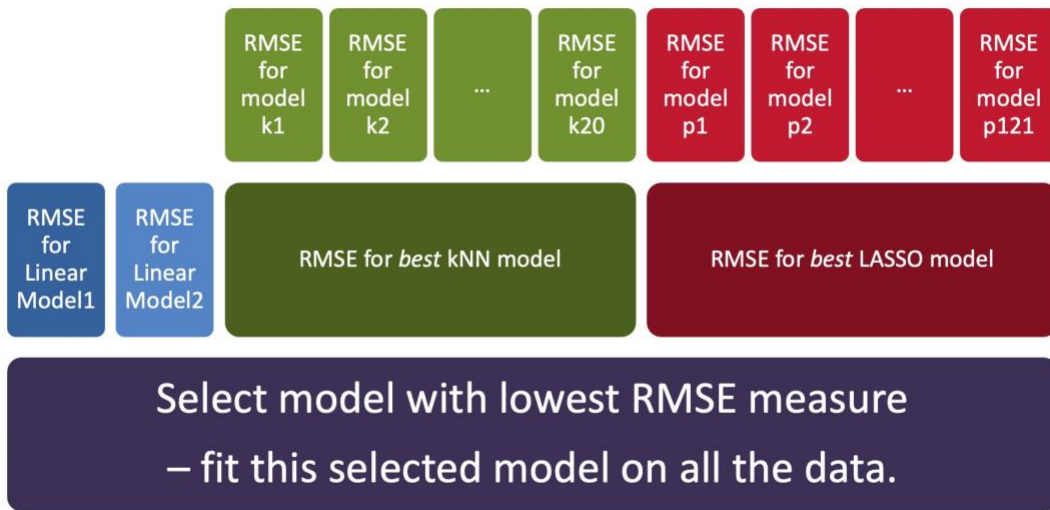
$\hat{y} = -16.6763 + 0.0636 \cdot \text{Age} - 0.0772 \cdot \text{Weight} - 0.0862 \cdot \text{Height} - 0.4731 \cdot \text{Neck} \dots$

We now summarize this model-fitting process as applied to our example using the functions from the Caret package. We began with a variety of models, including model 1 for linear regression with three predictors, model 2 for linear regression with all 14 predictors, a variety of penalized regression, specifically LASSO models indexed by the different hyperparameter values for lambda, and similarly, a variety of k-nearest neighbors models identified by the hyperparameter values of k.

We then used the train function from the Caret package applied to these different methods appropriately to run our internal cross-validation. As we did on the prior slide, you may find code for this running in MethodSelection_caret.R. Finally, after we got a best model from each of these four applications of the train function applied for each of these different methods, we selected the best of these four methods by seeing which one minimized RMSE and then fit this final selected model in all data to get our final fitted model.

This happened to be a LASSO model with lambda equals 0.02237. And to display the coefficients without scientific notation, you may use the commands off at the right. And we state this as a fitted linear regression equation, just not including those predictors whose coefficients were 0. And we can interpret coefficient similarly to how we would with any linear equation.

Flowchart of Model Selection for Caret Example



We here summarize this process of model selection applied for our example using Caret. We first compute a measure RMSE for the first linear model. We also compute or obtain an RMSE measure for the second linear model. And internally in the Caret function, the RMSE measures computed for all the different k-nearest neighbors models and outputs just the measure for the best kNN model.

Similarly, the measures are computed for all the different LASSO models, but we just get the output measure for the best LASSO model. So out of those four models, the linear model 1, linear model 2, best, kNN model, and best LASSO model, we pick out the best one according to the lowest measure and then fit the selected model on all the data.

Summary

- Reviewed measures for CV (with mentioned alternatives).
- Introduced the caret package to streamline the cross-validation process for model selection.
- May select best model across several methods, as long as measure is produced from same-scale response.
- Model selection is part of the model-fitting process

To use cross-validation for both model selection and assessment, there must be two levels of cross-validation (demonstrated next).