Regularization is a way of avoiding overfit by restricting the magnitude of model coefficients (or in deep learning, node weights). A simple example of regularization is the use of ridge or lasso regression to fit linear models in the presence of collinear variables or (quasi-)separation. The intuition is that smaller coefficients are less sensitive to idiosyncracies in the training data, and hence, less likely to overfit.

Cross-validation is a way to safely reuse training data in nested model situations. This includes both the case of setting hyperparameters before fitting a model, and the case of fitting models (let’s call them *base learners*) that are then used as variables in downstream models, as shown in Figure 1. In either situation, using the same data twice can lead to models that are overtuned to idiosyncracies in the training data, and more likely to overfit.

**Figure 1** Properly nesting models with cross-validation

In general, if any stage of your modeling pipeline involves looking at the outcome (we’ll call that a *y-aware* stage), you cannot directly use the same data in the following stage of the pipeline. If you have enough data, you can use separate data in each stage of the modeling process (for example, one set of data to learn hyperparameters, another set of data to train the model that uses those hyperparameters). Otherwise, you should use cross-validation to reduce the nested model bias.

Cross-validation is relatively computationally expensive; regularization is relatively cheap. Can you mitigate nested model bias by using regularization techniques instead of cross-validation?

The short answer: no, you shouldn’t. But as, we’ve written before, demonstrating this is more memorable than simply saying “Don’t do that.”

**A simple example**

Suppose you have a system with two categorical variables. The variable x\_s has 10 levels, and the variable x\_n has 100 levels. The outcome y is a function of x\_s, but not of x\_n (but you, the analyst building the model, don’t know this). Here’s the head of the data.

## x\_s x\_n y

## 2 s\_10 n\_72 0.34228110

## 3 s\_01 n\_09 -0.03805102

## 4 s\_03 n\_18 -0.92145960

## 9 s\_08 n\_43 1.77069352

## 10 s\_08 n\_17 0.51992928

## 11 s\_01 n\_78 1.04714355

With most modeling techniques, a categorical variable with K levels is equivalent to K or K-1 numerical (indicator or dummy) variables, so this system actually has around 110 variables. In real life situations where a data scientist is working with high-cardinality categorical variables, or with a lot of categorical variables, the number of actual variables can begin to swamp the size of training data, and/or bog down the machine learning algorithm.

One way to deal with these issues is to represent each categorical variable by a single variable model (or base learner), and then use the predictions of those base learners as the inputs to a bigger model. So instead of fitting a model with 110 indicator variables, you can fit a model with two numerical variables. This is a simple example of nested models.

**Figure 2** Impact coding as an example of nested model

We refer to this procedure as “impact coding,” and it is one of the data treatments available in the vtreat package, specifically for dealing with high-cardinality categorical variables. But for now, let’s go back to the original problem.

**The naive way**

For this simple example, you might try representing each variable as the expected value of y - mean(y) in the training data, conditioned on the variable’s level. So the ith “coefficient” of the one-variable model would be given by:

*vi* = *E*[*y*|*x* = *si*] − *E*[*y*]

Where *si* is the *i*th level. Let’s show this with the variable x\_s (the code for all the examples in this article is below):

Convenience functions.

# function to calculate the rmse

rmse = function(ypred, y) {

resid = y - ypred

sqrt(mean(resid^2))

}

# function to calculate R-squared

rsquared = function(ypred, y) {

null\_variance = sum((y - mean(y))^2)

resid\_variance = sum((y - ypred)^2)

1 - (resid\_variance/null\_variance)

}

compare\_models = function(predframe) {

predictions = setdiff(colnames(predframe), "y")

data.frame(# pred\_type = predictions,

rmse = vapply(predframe[,predictions, drop=FALSE],

FUN = function(p) rmse(p, predframe$y),

numeric(1)),

rsquared = vapply(predframe[,predictions,drop=FALSE],

FUN = function(p) rsquared(p, predframe$y),

numeric(1))

)

}

**A simple example**

Set up the data and split into training and test sets.

set.seed(3453421)

Ndata = 500

nnoise = 100

nsig = 10

noise\_levels = paste0("n\_", sprintf('%02d', 1:nnoise))

signal\_levels = paste0("s\_", sprintf('%02d', 1:nsig))

sig\_amps = 2\*runif(1:nsig, min=-1, max=1)

names(sig\_amps) = signal\_levels

sig\_amps = sig\_amps - mean(sig\_amps) # mean zero

x\_s = sample(signal\_levels, Ndata, replace=TRUE)

x\_n = sample(noise\_levels, Ndata, replace=TRUE)

y = sig\_amps[x\_s] + rnorm(Ndata) # a function of x\_s but not x\_n

df = data.frame(x\_s=x\_s, x\_n=x\_n, y=y, stringsAsFactors=FALSE)

library(zeallot)

c(dtest, dtrain) %<-% split(df, runif(Ndata) < 0.5) # false comes first

head(dtrain)

## x\_s x\_n y

## 2 s\_10 n\_72 0.34228110

## 3 s\_01 n\_09 -0.03805102

## 4 s\_03 n\_18 -0.92145960

## 9 s\_08 n\_43 1.77069352

## 10 s\_08 n\_17 0.51992928

## 11 s\_01 n\_78 1.04714355

# for later - a frame to hold the test set predictions

pframe = data.frame(y = dtest$y)

**The naive way**

For this simple example, you might try representing each variable as the expected value of y - mean(y) in the training data, conditioned on the variable's level. So the ith "coefficient" of the one-variable model would be given by:

*vi* = *E*[*y*|*x* = *si*]−*E*[*y*]

Where *si* is the *i*th level.

"Fit" the one-variable model for x\_s.

# build the maps of expected values

library(rqdatatable) # yes, you can use dplyr or base instead...

library(wrapr)

xs\_means = dtrain %.>%

extend(., delta := y - mean(y)) %.>%

project(.,

meany := mean(y),

coeff := mean(delta),

groupby = 'x\_s') %.>%

order\_rows(.,

'x\_s') %.>%

as.data.frame(.)

xs\_means

## x\_s meany coeff

## 1 s\_01 0.7998263 0.8503282

## 2 s\_02 -1.3815640 -1.3310621

## 3 s\_03 -0.7928449 -0.7423430

## 4 s\_04 -0.8245088 -0.7740069

## 5 s\_05 0.7547054 0.8052073

## 6 s\_06 0.1564710 0.2069728

## 7 s\_07 -1.1747557 -1.1242539

## 8 s\_08 1.3520153 1.4025171

## 9 s\_09 1.5789785 1.6294804

## 10 s\_10 -0.7313895 -0.6808876

"Fit" the one-variable model for x\_n and treat or "prepare" the data (we are using terminology that is consistent with vtreat).

xn\_means = dtrain %.>%

extend(., delta := y - mean(y)) %.>%

project(.,

meany := mean(delta),

groupby = 'x\_n') %.>%

order\_rows(.,

'x\_n') %.>%

as.data.frame(.)

# the maps that convert categorical levels to numerical values

xs\_map = with(xs\_means, x\_s := coeff)

xn\_map = with(xn\_means, x\_n := meany)

prepare\_manually = function(coefmap, xcol) {

treated = coefmap[xcol]

ifelse(is.na(treated), 0, treated)

}

# "prepare" the data

dtrain\_treated = dtrain

dtrain\_treated$vs = prepare\_manually(xs\_map, dtrain$x\_s)

dtrain\_treated$vn = prepare\_manually(xn\_map, dtrain$x\_n)

head(dtrain\_treated)

## x\_s x\_n y vs vn

## 2 s\_10 n\_72 0.34228110 -0.6808876 0.64754957

## 3 s\_01 n\_09 -0.03805102 0.8503282 0.54991135

## 4 s\_03 n\_18 -0.92145960 -0.7423430 0.01923877

## 9 s\_08 n\_43 1.77069352 1.4025171 1.90394159

## 10 s\_08 n\_17 0.51992928 1.4025171 0.26448341

## 11 s\_01 n\_78 1.04714355 0.8503282 0.70342961

Now fit a linear model for y as a function of vs and vn.

model\_raw = lm(y ~ vs + vn,

data=dtrain\_treated)

summary(model\_raw)

##

## Call:

## lm(formula = y ~ vs + vn, data = dtrain\_treated)

##

## Residuals:

## Min 1Q Median 3Q Max

## -2.33068 -0.57106 0.00342 0.52488 2.25472

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -0.05050 0.05597 -0.902 0.368

## vs 0.77259 0.05940 13.006 <2e-16 \*\*\*

## vn 0.61201 0.06906 8.862 <2e-16 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 0.8761 on 242 degrees of freedom

## Multiple R-squared: 0.6382, Adjusted R-squared: 0.6352

## F-statistic: 213.5 on 2 and 242 DF, p-value: < 2.2e-16

Apply the naive model to the test data.

# apply to test data

dtest\_treated = dtest

dtest\_treated$vs = prepare\_manually(xs\_map, dtest$x\_s)

dtest\_treated$vn = prepare\_manually(xn\_map, dtest$x\_n)

pframe$ypred\_naive = predict(model\_raw, newdata=dtest\_treated)

# look at the predictions on holdout data

compare\_models(pframe) %.>% knitr::kable(.)

|  | **rmse** | **rsquared** |
| --- | --- | --- |
| ypred\_naive | 1.303778 | 0.2311538 |

**The right way: cross-validation**

Let's fit the correct nested model, using vtreat.

library(vtreat)

library(wrapr)

xframeResults = mkCrossFrameNExperiment(dtrain, qc(x\_s, x\_n), "y",

codeRestriction = qc(catN),

verbose = FALSE)

# the plan uses the one-variable models to treat data

treatmentPlan = xframeResults$treatments

# the cross-frame

dtrain\_treated = xframeResults$crossFrame

head(dtrain\_treated)

## x\_s\_catN x\_n\_catN y

## 1 -0.6337889 0.91241547 0.34228110

## 2 0.8342227 0.82874089 -0.03805102

## 3 -0.7020597 0.18198634 -0.92145960

## 4 1.3983175 1.99197404 1.77069352

## 5 1.3983175 0.11679580 0.51992928

## 6 0.8342227 0.06421659 1.04714355

variables = setdiff(colnames(dtrain\_treated), "y")

model\_X = lm(mk\_formula("y", variables),

data=dtrain\_treated)

summary(model\_X)

##

## Call:

## lm(formula = mk\_formula("y", variables), data = dtrain\_treated)

##

## Residuals:

## Min 1Q Median 3Q Max

## -3.2157 -0.7343 0.0225 0.7483 2.9639

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -0.04169 0.06745 -0.618 0.537

## x\_s\_catN 0.92968 0.06344 14.656 <2e-16 \*\*\*

## x\_n\_catN 0.10204 0.06654 1.533 0.126

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 1.055 on 242 degrees of freedom

## Multiple R-squared: 0.4753, Adjusted R-squared: 0.471

## F-statistic: 109.6 on 2 and 242 DF, p-value: < 2.2e-16

We can compare the performance of this model to the naive model on holdout data.

dtest\_treated = prepare(treatmentPlan, dtest)

pframe$ypred\_crossval = predict(model\_X, newdata=dtest\_treated)

compare\_models(pframe) %.>% knitr::kable(.)

|  | **rmse** | **rsquared** |
| --- | --- | --- |
| ypred\_naive | 1.303778 | 0.2311538 |
| ypred\_crossval | 1.093955 | 0.4587089 |

The correct model has a much smaller root-mean-squared error and a much larger R-squared than the naive model when applied to new data.

**An attempted alternative: regularized models.**

For a one-variable model, L2-regularization is simply Laplace smoothing. Again, we'll represent each "coefficient" of the one-variable model as the Laplace smoothed value minus the grand mean.

*vi* = ∑*xj* = *siyi*/(count*i* + *λ*)−*E*[*yi*]

Where count*i* is the frequency of *si* in the training data, and *λ* is the smoothing parameter (usually 1). If *λ* = 1 then the first term on the right is just adding one to the frequency of the level and then taking the "adjusted conditional mean" of y.

"Fit" a regularized model to x\_s.

# build the coefficients

lambda = 1

xs\_regmap = dtrain %.>%

extend(., grandmean = mean(y)) %.>%

project(.,

sum\_y := sum(y),

count\_y := n(),

grandmean := mean(grandmean), # pseudo-aggregator

groupby = 'x\_s') %.>%

extend(.,

vs := (sum\_y/(count\_y + lambda)) - grandmean

) %.>%

order\_rows(.,

'x\_s') %.>%

as.data.frame(.)

xs\_regmap

## x\_s sum\_y count\_y grandmean vs

## 1 s\_01 20.795484 26 -0.05050187 0.8207050

## 2 s\_02 -37.302227 27 -0.05050187 -1.2817205

## 3 s\_03 -22.199656 28 -0.05050187 -0.7150035

## 4 s\_04 -14.016649 17 -0.05050187 -0.7282009

## 5 s\_05 19.622340 26 -0.05050187 0.7772552

## 6 s\_06 3.129419 20 -0.05050187 0.1995218

## 7 s\_07 -35.242672 30 -0.05050187 -1.0863585

## 8 s\_08 36.504412 27 -0.05050187 1.3542309

## 9 s\_09 33.158549 21 -0.05050187 1.5577086

## 10 s\_10 -16.821957 23 -0.05050187 -0.6504130

"Fit" a regularized model to x\_m. Apply the one variable models for x\_s and x\_n to the data.

xn\_regmap = dtrain %.>%

extend(., grandmean = mean(y)) %.>%

project(.,

sum\_y := sum(y),

count\_y := n(),

grandmean := mean(grandmean), # pseudo-aggregator

groupby = 'x\_n') %.>%

extend(.,

vn := (sum\_y/(count\_y + lambda)) - grandmean

) %.>%

order\_rows(.,

'x\_n') %.>%

as.data.frame(.)

# the maps that convert categorical levels to numerical values

vs\_map = xs\_regmap$x\_s := xs\_regmap$vs

vn\_map = xn\_regmap$x\_n := xn\_regmap$vn

# "prepare" the data

dtrain\_treated = dtrain

dtrain\_treated$vs = prepare\_manually(vs\_map, dtrain$x\_s)

dtrain\_treated$vn = prepare\_manually(vn\_map, dtrain$x\_n)

head(dtrain\_treated)

## x\_s x\_n y vs vn

## 2 s\_10 n\_72 0.34228110 -0.6504130 0.44853367

## 3 s\_01 n\_09 -0.03805102 0.8207050 0.42505898

## 4 s\_03 n\_18 -0.92145960 -0.7150035 0.02370493

## 9 s\_08 n\_43 1.77069352 1.3542309 1.28612835

## 10 s\_08 n\_17 0.51992928 1.3542309 0.21098803

## 11 s\_01 n\_78 1.04714355 0.8207050 0.61015422

Now fit the overall model:

model\_reg = lm(y ~ vs + vn, data=dtrain\_treated)

summary(model\_reg)

##

## Call:

## lm(formula = y ~ vs + vn, data = dtrain\_treated)

##

## Residuals:

## Min 1Q Median 3Q Max

## -2.30354 -0.57688 -0.02224 0.56799 2.25723

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -0.06665 0.05637 -1.182 0.238

## vs 0.81142 0.06203 13.082 < 2e-16 \*\*\*

## vn 0.85393 0.09905 8.621 8.8e-16 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 0.8819 on 242 degrees of freedom

## Multiple R-squared: 0.6334, Adjusted R-squared: 0.6304

## F-statistic: 209.1 on 2 and 242 DF, p-value: < 2.2e-16

Comparing the performance of the three models on holdout data.

# apply to test data

dtest\_treated = dtest

dtest\_treated$vs = prepare\_manually(vs\_map, dtest$x\_s)

dtest\_treated$vn = prepare\_manually(vn\_map, dtest$x\_n)

pframe$ypred\_reg = predict(model\_reg, newdata=dtest\_treated)

# compare the predictions of each model

compare\_models(pframe) %.>% knitr::kable(.)

|  | **rmse** | **rsquared** |
| --- | --- | --- |
| ypred\_naive | 1.303778 | 0.2311538 |
| ypred\_crossval | 1.093955 | 0.4587089 |
| ypred\_reg | 1.267648 | 0.2731756 |

## x\_s meany coeff

## 1 s\_01 0.7998263 0.8503282

## 2 s\_02 -1.3815640 -1.3310621

## 3 s\_03 -0.7928449 -0.7423430

## 4 s\_04 -0.8245088 -0.7740069

## 5 s\_05 0.7547054 0.8052073

## 6 s\_06 0.1564710 0.2069728

## 7 s\_07 -1.1747557 -1.1242539

## 8 s\_08 1.3520153 1.4025171

## 9 s\_09 1.5789785 1.6294804

## 10 s\_10 -0.7313895 -0.6808876

In other words, whenever the value of x\_s is s\_01, the one variable model vs returns the value 0.8503282, and so on. If you do this for both variables, you get a training set that looks like this:

## x\_s x\_n y vs vn

## 2 s\_10 n\_72 0.34228110 -0.6808876 0.64754957

## 3 s\_01 n\_09 -0.03805102 0.8503282 0.54991135

## 4 s\_03 n\_18 -0.92145960 -0.7423430 0.01923877

## 9 s\_08 n\_43 1.77069352 1.4025171 1.90394159

## 10 s\_08 n\_17 0.51992928 1.4025171 0.26448341

## 11 s\_01 n\_78 1.04714355 0.8503282 0.70342961

Now fit a linear model for y as a function of vs and vn.

model\_raw = lm(y ~ vs + vn,

data=dtrain\_treated)

summary(model\_raw)

##

## Call:

## lm(formula = y ~ vs + vn, data = dtrain\_treated)

##

## Residuals:

## Min 1Q Median 3Q Max

## -2.33068 -0.57106 0.00342 0.52488 2.25472

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -0.05050 0.05597 -0.902 0.368

## vs 0.77259 0.05940 13.006 <2e-16 \*\*\*

## vn 0.61201 0.06906 8.862 <2e-16 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 0.8761 on 242 degrees of freedom

## Multiple R-squared: 0.6382, Adjusted R-squared: 0.6352

## F-statistic: 213.5 on 2 and 242 DF, p-value: < 2.2e-16

Note that this model gives significant coefficients to both vs and vn, even though y is not a function of x\_n (or vn). Because you used the same data to fit the one variable base learners and to fit the larger model, you have overfit.

**The right way: cross-validation**

The correct way to impact code (or to nest models in general) is to use cross-validation techniques. Impact coding with cross-validation is already implemented in vtreat; note the similarity between this diagram and Figure 1 above.

**Figure 3** Cross-validated data preparation with vtreat

The training data is used both to fit the base learners (as we did above) and to also to create a data frame of cross-validated base learner predictions (called a *cross-frame* in vtreat). This cross-frame is used to train the overall model. Let’s fit the correct nested model, using vtreat.

library(vtreat)

library(wrapr)

xframeResults = mkCrossFrameNExperiment(dtrain,

qc(x\_s, x\_n), "y",

codeRestriction = qc(catN),

verbose = FALSE)

# the plan uses the one-variable models to treat data

treatmentPlan = xframeResults$treatments

# the cross-frame

dtrain\_treated = xframeResults$crossFrame

head(dtrain\_treated)

## x\_s\_catN x\_n\_catN y

## 1 -0.6337889 0.91241547 0.34228110

## 2 0.8342227 0.82874089 -0.03805102

## 3 -0.7020597 0.18198634 -0.92145960

## 4 1.3983175 1.99197404 1.77069352

## 5 1.3983175 0.11679580 0.51992928

## 6 0.8342227 0.06421659 1.04714355

variables = setdiff(colnames(dtrain\_treated), "y")

model\_X = lm(mk\_formula("y", variables),

data=dtrain\_treated)

summary(model\_X)

##

## Call:

## lm(formula = mk\_formula("y", variables), data = dtrain\_treated)

##

## Residuals:

## Min 1Q Median 3Q Max

## -3.2157 -0.7343 0.0225 0.7483 2.9639

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -0.04169 0.06745 -0.618 0.537

## x\_s\_catN 0.92968 0.06344 14.656 <2e-16 \*\*\*

## x\_n\_catN 0.10204 0.06654 1.533 0.126

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 1.055 on 242 degrees of freedom

## Multiple R-squared: 0.4753, Adjusted R-squared: 0.471

## F-statistic: 109.6 on 2 and 242 DF, p-value: < 2.2e-16

This model correctly determines that x\_n (and its one-variable model x\_n\_catN) do not affect the outcome. We can compare the performance of this model to the naive model on holdout data.

|  | **rmse** | **rsquared** |
| --- | --- | --- |
| ypred\_naive | 1.303778 | 0.2311538 |
| ypred\_crossval | 1.093955 | 0.4587089 |

The correct model has a much smaller root-mean-squared error and a much larger R-squared than the naive model when applied to new data.

**An attempted alternative: regularized models.**

But cross-validation is so complicated. Can’t we just regularize? As we’ll show in the appendix of this article, for a one-variable model, L2-regularization is simply Laplace smoothing. Again, we’ll represent each “coefficient” of the one-variable model as the Laplace smoothed value minus the grand mean.

*vi* = ∑*xj*=*si* *yi*/(count*i* + *λ*) − *E*[*yi*]

Where count*i* is the frequency of *si* in the training data, and *λ* is the smoothing parameter (usually 1). If *λ* = 1 then the first term on the right is just adding one to the frequency of the level and then taking the “adjusted conditional mean” of y.

Again, let’s show this for the variable x\_s.

## x\_s sum\_y count\_y grandmean vs

## 1 s\_01 20.795484 26 -0.05050187 0.8207050

## 2 s\_02 -37.302227 27 -0.05050187 -1.2817205

## 3 s\_03 -22.199656 28 -0.05050187 -0.7150035

## 4 s\_04 -14.016649 17 -0.05050187 -0.7282009

## 5 s\_05 19.622340 26 -0.05050187 0.7772552

## 6 s\_06 3.129419 20 -0.05050187 0.1995218

## 7 s\_07 -35.242672 30 -0.05050187 -1.0863585

## 8 s\_08 36.504412 27 -0.05050187 1.3542309

## 9 s\_09 33.158549 21 -0.05050187 1.5577086

## 10 s\_10 -16.821957 23 -0.05050187 -0.6504130

After applying the one variable models for x\_s and x\_n to the data, the head of the resulting treated data looks like this:

## x\_s x\_n y vs vn

## 2 s\_10 n\_72 0.34228110 -0.6504130 0.44853367

## 3 s\_01 n\_09 -0.03805102 0.8207050 0.42505898

## 4 s\_03 n\_18 -0.92145960 -0.7150035 0.02370493

## 9 s\_08 n\_43 1.77069352 1.3542309 1.28612835

## 10 s\_08 n\_17 0.51992928 1.3542309 0.21098803

## 11 s\_01 n\_78 1.04714355 0.8207050 0.61015422

Now fit the overall model:

##

## Call:

## lm(formula = y ~ vs + vn, data = dtrain\_treated)

##

## Residuals:

## Min 1Q Median 3Q Max

## -2.30354 -0.57688 -0.02224 0.56799 2.25723

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -0.06665 0.05637 -1.182 0.238

## vs 0.81142 0.06203 13.082 < 2e-16 \*\*\*

## vn 0.85393 0.09905 8.621 8.8e-16 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 0.8819 on 242 degrees of freedom

## Multiple R-squared: 0.6334, Adjusted R-squared: 0.6304

## F-statistic: 209.1 on 2 and 242 DF, p-value: < 2.2e-16

Again, both variables look significant. Even with regularization, the model is still overfit. Comparing the performance of the models on holdout data, you see that the regularized model does a little better than the naive model, but not as well as the correctly cross-validated model.

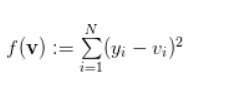
|  | **rmse** | **rsquared** |
| --- | --- | --- |
| ypred\_naive | 1.303778 | 0.2311538 |
| ypred\_crossval | 1.093955 | 0.4587089 |
| ypred\_reg | 1.267648 | 0.2731756 |

**The Moral of the Story**

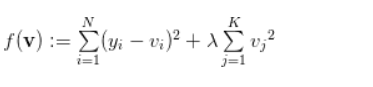
Unfortunately, regularization is not enough to overcome nested model bias. Whenever you apply a y-aware process to your data, you have to use cross-validation methods (or a separate data set) at the next stage of your modeling pipeline.

**Appendix: Derivation of Laplace Smoothing as L2-Regularization**

Without regularization, the optimal one-variable model for y in terms of a categorical variable with K levels {*sj*} is a set of K coefficients **v** such that

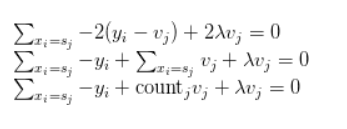


is minimized (N is the number of data points). L2-regularization adds a penalty to the magnitude of **v**, so that the goal is to minimize

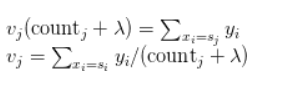


where *λ* is a known smoothing hyperparameter, usually set (in this case) to 1.

To minimize the above expression for a single coefficient *vj*, take the deriviative with respect to *vj* and set it to zero:



Where count*j* is the number of times the level *sj* appears in the training data. Now solve for *vj*:



This is Laplace smoothing. Note that it is also the one-variable equivalent of ridge regression.