Why do we use double CV?

Abra Brisbin

11/4/2022

library(ggformula)

## Warning: package 'ggformula' was built under R version 4.0.4

## Warning: package 'ggplot2' was built under R version 4.0.5

## Warning: package 'ggstance' was built under R version 4.0.5

## Warning: package 'ggridges' was built under R version 4.0.5

library(dplyr)

## Warning: package 'dplyr' was built under R version 4.0.5

library(caret)

## Warning: package 'caret' was built under R version 4.0.5

## Double Cross-Validation

When we do double cross-validation, we predict each test set (from the outer layer of CV) using the best model from the inner CV that was performed on the corresponding (outer layer) training set. At the end of the for loop of the outer CV, we have a vector containing 1 prediction for each data point. But those predictions may come from different models–this vector contains predictions from a “Frankenstein” model. **Why do we do this?**

The accuracy of these “Frankenstein” predictions are the best estimate of how well our model-selection process would do on similar data in the future. In other words, if we pick the “best” model by single cross-validation, how accurate will that model be on other data?

Suppose we have 10 models that (unbeknownst to us) are all equally good: For a random new data set, the accuracy of each model will be normally distributed with mean 0.75 and standard deviation 0.05. We compare them using single cross-validation and get a set of accuracies:

set.seed(987)  
accuracies = rnorm(10, mean = 0.75, sd = 0.05)  
accuracies

## [1] 0.7471493 0.7634997 0.7937236 0.7404827 0.8050349 0.7053121 0.7022516  
## [8] 0.7072071 0.6943490 0.6855637

which.max(accuracies)

## [1] 5

max(accuracies)

## [1] 0.8050349

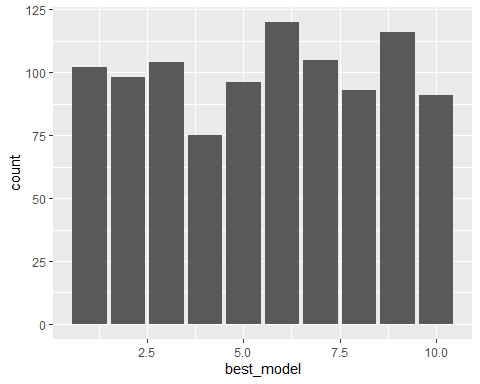
In this case, the “best” model was model 5, with an accuracy of 0.805. But the “true” long-term accuracy of model 5 on new data would still be 0.75, the mean of the normal distribution.

It isn’t just a coincidence that our “best” model appeared (in single CV) to have better accuracy than it actually had. Suppose we had 1000 data sets, and did single CV on all of them:

set.seed(987)  
n = 1000  
best\_model = numeric(length = n)  
best\_accuracy = numeric(length = n)  
  
for(ii in 1:n){  
 accuracies = rnorm(10, mean = 0.75, sd = 0.05)  
 best\_model[ii] = which.max(accuracies)  
 best\_accuracy[ii] = max(accuracies)  
}

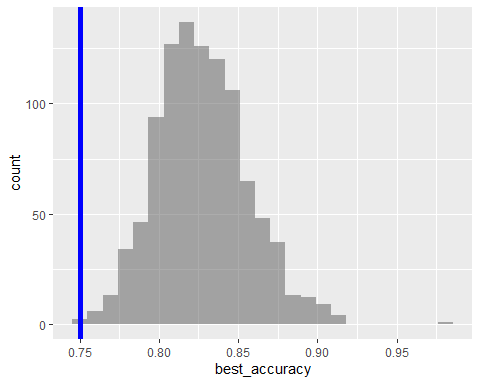
Each model comes out as “best” about equally often:

gf\_bar(~best\_model)



But the apparent accuracy of that model is consistently above 0.75:

gf\_histogram(~best\_accuracy) %>%  
 gf\_vline(xintercept = 0.75, col = "blue", lwd = 2)



This happens because we’re assessing the model using the same accuracy value that we used to select that it was the best. So, we’re more likely to choose model 5 as the “best” when it happened to do better than its long-run average. We’re more likely to choose model 8 as the “best” when *it* happened to do better than its long-run average. The only way we would get an estimate of 0.75 as the long-run accuracy of the “best” model is if one of the ten models had an accuracy of 0.75, and *all nine* other models had lower-than-average accuracy, by random chance. This is an unlikely event.

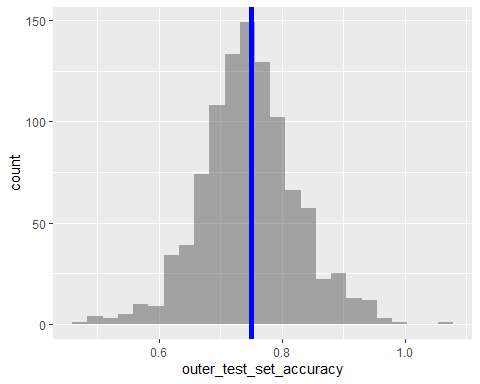
When we do double cross-validation, we still choose our best model using the single CV on the inside of the for loop. But then we use that model to predict a new set of data (the outer layer test set). If model 5 was the best model on the inner CV, it may or may not be the best model for the outer CV test set. That means its performance on the test set is more likely to be centered around its true long-run average performance for new data.

To illustrate this, let’s suppose that all 10 models still have mean accuracy of 0.75, but now Model 1 has sd of 0.01, Model 2 has sd of 0.02, etc.

set.seed(987)  
n = 1000  
best\_model = numeric(length = n)  
best\_accuracy = numeric(length = n)  
outer\_test\_set\_accuracy = numeric(length = n)  
  
for(ii in 1:n){  
 accuracies = rnorm(10, mean = 0.75, sd = seq(.01,.1, by = .01))  
 current\_best\_model = which.max(accuracies)  
 best\_model[ii] = current\_best\_model  
 best\_accuracy[ii] = max(accuracies)  
   
 outer\_test\_set\_accuracy[ii] = rnorm(1, mean = 0.75, sd = current\_best\_model/100)  
}

The accuracy as measured on the outer layer test set is approximately symmetrical around 0.75:

gf\_histogram(~outer\_test\_set\_accuracy) %>%  
 gf\_vline(xintercept = 0.75, color = "blue", lwd = 2)

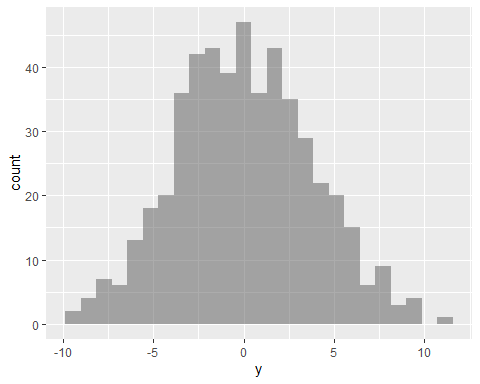


## An example with actual models

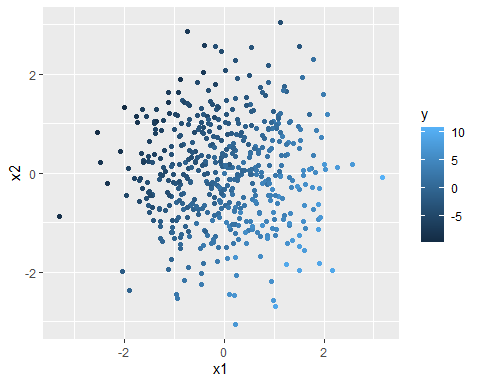
Let’s suppose that the actual relationship between , and is linear:

set.seed(987)  
x1 = rnorm(500, 0, 1)  
x2 = rnorm(500, 0, 1)  
y = 3\*x1 - 2\*x2 + rnorm(500, 0, 1)  
my\_data = data.frame(x1, x2, y)

gf\_histogram(~y)



gf\_point(x2~x1, color =~ y)



Let’s use single CV to compare KNN with k = 1 to 50. So, we’re comparing 50 different models.

ctrl = trainControl(method = "cv", number = 5)  
fit\_knn = train(y ~ x1+x2,  
 data = my\_data,  
 method = "knn",  
 tuneGrid = expand.grid(k = 1:50),   
 preProcess = c("center","scale"),  
 trControl = ctrl)  
fit\_knn

## k-Nearest Neighbors   
##   
## 500 samples  
## 2 predictor  
##   
## Pre-processing: centered (2), scaled (2)   
## Resampling: Cross-Validated (5 fold)   
## Summary of sample sizes: 400, 400, 400, 400, 400   
## Resampling results across tuning parameters:  
##   
## k RMSE Rsquared MAE   
## 1 1.410034 0.8603522 1.1377181  
## 2 1.221679 0.8951534 0.9875711  
## 3 1.204639 0.8979816 0.9771716  
## 4 1.182133 0.9019506 0.9573086  
## 5 1.161928 0.9055853 0.9396516  
## 6 1.137173 0.9106940 0.9163857  
## 7 1.127336 0.9126898 0.9078820  
## 8 1.128000 0.9131563 0.9023992  
## 9 1.130968 0.9138805 0.9076264  
## 10 1.128931 0.9147308 0.9029288  
## 11 1.125431 0.9168101 0.8947038  
## 12 1.131441 0.9163906 0.9013864  
## 13 1.127139 0.9178713 0.8984712  
## 14 1.135126 0.9174825 0.9007026  
## 15 1.137930 0.9177121 0.9027697  
## 16 1.145405 0.9174535 0.9068672  
## 17 1.152442 0.9174759 0.9103957  
## 18 1.159580 0.9175105 0.9163397  
## 19 1.160877 0.9180406 0.9168425  
## 20 1.165674 0.9178894 0.9181887  
## 21 1.174448 0.9173493 0.9222356  
## 22 1.177938 0.9173719 0.9215668  
## 23 1.187187 0.9166241 0.9262822  
## 24 1.192207 0.9165140 0.9290588  
## 25 1.194210 0.9172707 0.9299091  
## 26 1.201416 0.9169718 0.9343711  
## 27 1.210098 0.9161269 0.9401244  
## 28 1.224411 0.9142244 0.9509656  
## 29 1.228116 0.9143943 0.9518290  
## 30 1.238905 0.9133643 0.9578806  
## 31 1.245252 0.9132728 0.9634124  
## 32 1.250117 0.9131838 0.9652795  
## 33 1.256601 0.9129337 0.9701563  
## 34 1.260800 0.9134207 0.9756607  
## 35 1.266263 0.9135694 0.9771906  
## 36 1.276055 0.9127537 0.9838662  
## 37 1.285398 0.9118787 0.9875273  
## 38 1.293923 0.9112166 0.9939545  
## 39 1.300328 0.9113237 0.9967027  
## 40 1.307733 0.9110293 1.0004721  
## 41 1.314445 0.9107278 1.0053868  
## 42 1.321399 0.9101224 1.0104401  
## 43 1.331259 0.9097805 1.0160762  
## 44 1.337382 0.9093656 1.0220521  
## 45 1.343580 0.9091024 1.0248401  
## 46 1.347838 0.9092444 1.0274626  
## 47 1.354378 0.9090977 1.0309162  
## 48 1.360608 0.9089703 1.0340456  
## 49 1.367344 0.9082788 1.0393277  
## 50 1.376676 0.9076544 1.0447385  
##   
## RMSE was used to select the optimal model using the smallest value.  
## The final value used for the model was k = 11.

The best model was k = 11, with RMSE = 1.125431. What is the RMSE of this best model on new data? We can test this by generating many new data sets, using the same linear relationship between , and :

set.seed(123)  
n = 1000  
long\_run\_RMSE = numeric(length = n)  
  
for(ii in 1:n){  
 x1 = rnorm(500, 0, 1)  
 x2 = rnorm(500, 0, 1)  
 y = 3\*x1 - 2\*x2 + rnorm(500, 0, 1)  
 my\_new\_data = data.frame(x1, x2, y)  
   
 predictions = predict(fit\_knn, newdata = my\_new\_data)  
 long\_run\_RMSE[ii] = sqrt(mean((predictions - my\_new\_data$y)^2))  
}

mean(long\_run\_RMSE)

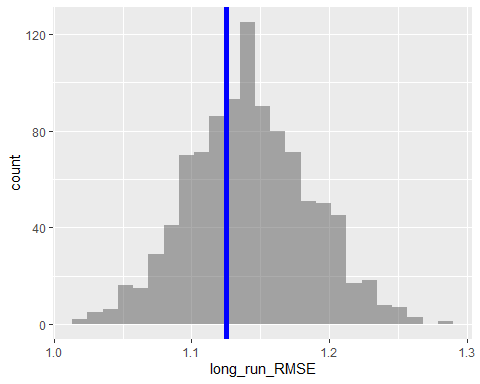
## [1] 1.142108

median(long\_run\_RMSE)

## [1] 1.14104

The long-run RMSE on new data averages 1.14, slightly higher (worse) than the RMSE from single CV. The histogram of the RMSEs on new data is shifted to the right, compared to the RMSE from single CV:

gf\_histogram(~long\_run\_RMSE) %>%  
 gf\_vline(xintercept = 1.125431, color = "blue", lwd = 2)



### Maybe we just got unlucky?

Maybe the single CV we did just happened to underestimate the RMSE (overestimate the accuracy) by chance, due to the random split of the folds. Let’s try estimating the RMSE using single CV 100 times.

ctrl = trainControl(method = "cv", number = 5)  
RMSE\_single\_CV = numeric(100)  
  
for(jj in 1:100){  
  
 fit\_knn = train(y ~ x1+x2,  
 data = my\_data,  
 method = "knn",  
 tuneGrid = expand.grid(k = 1:50),   
 preProcess = c("center","scale"),  
 trControl = ctrl)  
   
 RMSE\_single\_CV[jj] = min(fit\_knn$results$RMSE)  
}

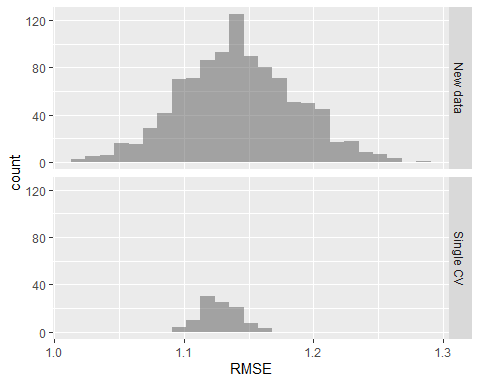
(Note, even though we have a for loop, we are *not* doing double CV. The “data =” is still the full data set, not train\_out. This is just single CV, done 100 times.)

On average, the RMSEs from single CV underestimate the RMSEs on the new data:

mean(RMSE\_single\_CV)

## [1] 1.127528

RMSE\_df = data.frame(RMSE = c(RMSE\_single\_CV, long\_run\_RMSE), method = c(rep("Single CV", 100), rep("New data", 1000)))  
  
RMSE\_df %>%  
 gf\_histogram(~RMSE) %>%  
 gf\_facet\_grid(method ~ .)



### Double CV to the rescue!

Now let’s create a “Frankenstein” model by performing double CV and saving the predictions of the best model from each fold.

ctrl = trainControl(method = "cv", number = 5)  
k = 5 # using 5-fold outer CV  
n = dim(my\_data)[1]  
groups = rep(1:k, length = n)  
cvgroups = sample(groups, n)  
allpredictedCV = numeric(length = n)  
  
for(ii in 1:5){  
 groupii = (cvgroups == ii)  
 train\_out = my\_data[!groupii, ]  
 test\_out = my\_data[groupii, ]  
   
 fit\_knn = train(y ~ x1+x2,  
 data = train\_out,  
 method = "knn",  
 tuneGrid = expand.grid(k = 1:50),   
 preProcess = c("center","scale"),  
 trControl = ctrl)  
   
 print(paste("Best model in fold", ii, "uses", fit\_knn$bestTune, "nearest neighbors"))  
  
 predictions = predict(fit\_knn, newdata = test\_out)  
 allpredictedCV[groupii] = predictions  
   
}

## [1] "Best model in fold 1 uses 8 nearest neighbors"  
## [1] "Best model in fold 2 uses 8 nearest neighbors"  
## [1] "Best model in fold 3 uses 7 nearest neighbors"  
## [1] "Best model in fold 4 uses 8 nearest neighbors"  
## [1] "Best model in fold 5 uses 5 nearest neighbors"

sqrt(mean((allpredictedCV - my\_data$y)^2))

## [1] 1.150478

The RMSE of the Frankenstein model is not an underestimate of the true long-run performance. In this case, it was actually an overestimate. Let’s repeat this process 100 times and see what happens:

ctrl = trainControl(method = "cv", number = 5)  
k = 5 # using 5-fold outer CV  
n = dim(my\_data)[1]  
num\_reps =100  
RMSE = numeric(length = num\_reps)  
  
for(jj in 1:num\_reps){  
  
 groups = rep(1:k, length = n)  
 cvgroups = sample(groups, n)  
 allpredictedCV = numeric(length = n)  
  
 for(ii in 1:5){  
 groupii = (cvgroups == ii)  
 train\_out = my\_data[!groupii, ]  
 test\_out = my\_data[groupii, ]  
   
 fit\_knn = train(y ~ x1+x2,  
 data = train\_out,  
 method = "knn",  
 tuneGrid = expand.grid(k = 1:50),   
 preProcess = c("center","scale"),  
 trControl = ctrl)  
   
 predictions = predict(fit\_knn, newdata = test\_out)  
 allpredictedCV[groupii] = predictions  
   
 }  
 RMSE[jj] = sqrt(mean((allpredictedCV - my\_data$y)^2))  
}

On average, the RMSE from the “Frankenstein” model of double CV is much closer to the RMSE on new data.

mean(RMSE)

## [1] 1.144005

RMSE\_df = data.frame(RMSE = c(long\_run\_RMSE, RMSE), method = c(rep("New data", 1000), rep("Double CV", 100)))  
  
RMSE\_df %>%  
 gf\_histogram(~RMSE) %>%  
 gf\_facet\_grid(method ~ .)

