K-fold cross validation in the Tidyverse

Stephanie J. Spielman

11/7/2017

Requirements

This demo requires several packages:

- `tidyverse` (`dplyr`, `tidyr`, `tibble`, `ggplot2`)
- `modelr`
- `broom`
- `pROC`

Background

K-fold cross validation is a common approach to assess the performance of a given model. The method works by randomly dividing a dataset into K equal "folds" (generally K=10 is a good choice). First, folds 2-10 are used to *train* the model, and folds 1 is used to *test* the model. A quantity (such as RMSE for linear models, or AUC etc. for logistic regression) is calculated from the test predictions to help us determine the performance of the trained model on the test data. Next, folds 1 and 3-10 are used to train, and folds 2 is used for testing, etc. In the end, the model will have been trained on K training dataset folds and evaluated with K test dataset folds. The final distribution of quantities can be assessed to validate the model.

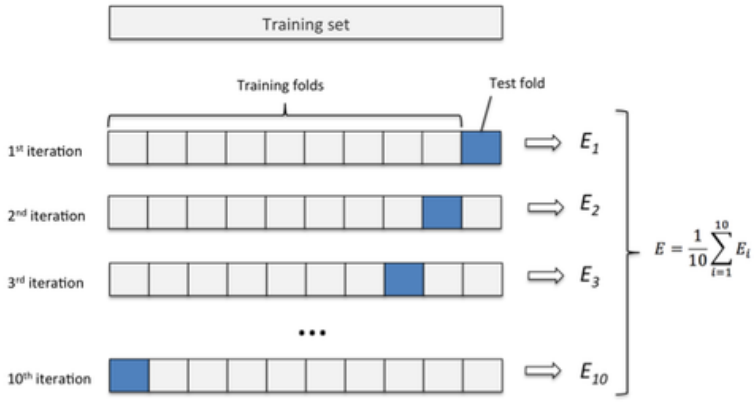


Figure 1: K-fold cross validation schematic

Necessary functions

Before beginning, there are a few new functions to be familiar with.

`nest()` and `unnest()`

The `tiydr::nest()` function converts rows of a dataframe into a list, and stores this list in a column called `data`:

```
## To ensure "pretty printing", we must force our input dataframe to be a tibble:
iris2 <- as.tibble(iris)

iris2 %>% nest()
```

```
## # A tibble: 1 x 1
##                 data
##               <list>
## 1 <tibble [150 x 5]>
```

You can further specify that certain columns *not* be nested.

```
### Subtract Sepal.Width from nest() to preserve it
iris2 %>% nest(-Sepal.Width)
```

```
## # A tibble: 23 x 2
##    Sepal.Width              data
##          <dbl>            <list>
##  1         3.5  <tibble [6 x 4]>
##  2         3.0 <tibble [26 x 4]>
##  3         3.2 <tibble [13 x 4]>
##  4         3.1 <tibble [11 x 4]>
##  5         3.6  <tibble [4 x 4]>
##  6         3.9  <tibble [2 x 4]>
##  7         3.4 <tibble [12 x 4]>
##  8         2.9 <tibble [10 x 4]>
##  9         3.7  <tibble [3 x 4]>
## 10         4.0  <tibble [1 x 4]>
## # ... with 13 more rows
```

In the above output, you can see that all rows corresponding to each `Sepal.Length` value have been packaged into a list in a new column called `data`, for later use.

The `tiydr::unnest()` undoes nesting:

```
iris2 %>%
    nest() %>%  ### make it all one column
    unnest() %>%   ### unnest to restore to original data frame state
    head()
```

```
## # A tibble: 6 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

`modelr::crossv_kfold()`

This function uses random sampling to create K folds for you from a given dataset and produces a data frame with three columns:

- `train`, the training fold
- `test`, the associated testing fold
- `.id`, the fold index for each train-test set. This column ranges from 1-K.

```
set.seed(101)
iris %>% crossv_kfold(5)
```

```
## # A tibble: 5 x 3
##            train           test   .id
##           <list>         <list> <chr>
## 1 <S3: resample> <S3: resample>     1
## 2 <S3: resample> <S3: resample>     2
## 3 <S3: resample> <S3: resample>     3
## 4 <S3: resample> <S3: resample>     4
## 5 <S3: resample> <S3: resample>     5
```

`broom::augment()`

We have seen this function before along with `tidy()` and `glance()`, but not really used it in depth. Now its utility is revealed: This function merges, into a single dataframe, the input data with the **fitted values** for a model, aka predictions! These values are stored in a column called `.fitted`. Below, I show you how you can directly see the true values of sepal length as well as what the model predicts the sepal lengths would be, for the relevant petal length.

```
fit <- lm(Sepal.Length ~ Petal.Length, data=iris)
augment(fit) %>% select(Sepal.Length, Petal.Length, .fitted) %>% head()
```

```
##   Sepal.Length Petal.Length  .fitted
## 1          5.1          1.4 4.879095
## 2          4.9          1.4 4.879095
## 3          4.7          1.3 4.838202
## 4          4.6          1.5 4.919987
## 5          5.0          1.4 4.879095
## 6          5.4          1.7 5.001771
```

Performing cross validation on a linear model

This section demonstrates how to run a k-fold cross validation for a linear model. Specifically, we will validate the model: `lm(Sepal.Length ~ Petal.Length + Species)` with K=10 folds.

To begin, we will set the random seed to a random number of our choosing:

```
set.seed(53629)
```

Next, we create the K=10 folds and create 10 models on the `train` column. Note that we have seen similar use of `purrr:map()` when running permutation tests.

```
iris %>%
  crossv_kfold(10) %>%
  mutate(model = purrr::map(train, ~lm(Sepal.Length ~ Petal.Length, data=.))) -> trained.models

trained.models
```

```
## # A tibble: 10 x 4
##              train           test   .id    model
##             <list>         <list> <chr>   <list>
##  1 <S3: resample> <S3: resample>    01 <S3: lm>
##  2 <S3: resample> <S3: resample>    02 <S3: lm>
##  3 <S3: resample> <S3: resample>    03 <S3: lm>
##  4 <S3: resample> <S3: resample>    04 <S3: lm>
##  5 <S3: resample> <S3: resample>    05 <S3: lm>
##  6 <S3: resample> <S3: resample>    06 <S3: lm>
##  7 <S3: resample> <S3: resample>    07 <S3: lm>
##  8 <S3: resample> <S3: resample>    08 <S3: lm>
##  9 <S3: resample> <S3: resample>    09 <S3: lm>
## 10 <S3: resample> <S3: resample>    10 <S3: lm>
```

In the output, we see four columns:

- `train`, the training fold, produced by `crossv_kfold()`
- `test`, the associated testing fold, produced by `crossv_kfold()`
- `.id`, the fold index for each train-test set, produced by `crossv_kfold()`
- `model`, the fitted linear model, produced in the call to `purrr:map()`

Now, we need to evaluate each trained model on its respective test data. Happily, `modelr` makes this easy with convenience functions `rmse()`. This function takes two arguments as follows: `rmse(fitted model, dataset)`. Therefore, in one call to `rmse()`, we can calculate the RMSE from the trained model on the test data.

To accomplish this, we need to use the function `purrr::map2()`, which is like `map()` but when there are two inputs to the function of interest. We use this function as `purrr::map2_dbl()` to specify that we want numeric (double) output:

```
map2_dbl(trained.models$model, trained.models$test, rmse) -> test.rmse

test.rmse
```

```
##         1         2         3         4         5         6         7
## 0.3835638 0.3390923 0.3912646 0.3774837 0.3694881 0.3975375 0.4263055
##         8         9        10
## 0.4251174 0.5014940 0.4835578
```

These values correspond to the root mean square error of our model, which can be interpretted as the error in model inferences. Recall, we are predicting Sepal.Length in our model, which is distributed as:

```
summary(iris$Sepal.Length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   5.100   5.800   5.843   6.400   7.900
```

Compared to the range of values for Sepal.Lengths, our errors are rather low (in the ~10% range), so our model is pretty good. We can further run some summary statistics on the RMSE:

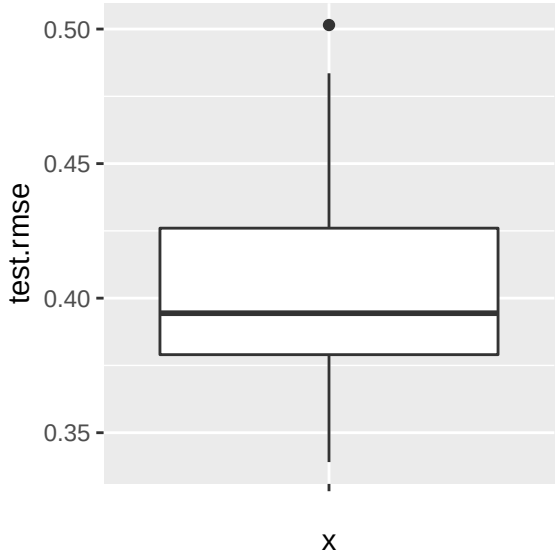
```
summary(test.rmse)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.3391  0.3790  0.3944  0.4095  0.4260  0.5015
```

```
sd(test.rmse)
```

```
## [1] 0.05077768
```

```
## Convert to data frame and plot:
as.data.frame(test.rmse) %>% ggplot(aes(x="", y=test.rmse)) + geom_boxplot()
```



In the boxplot, there is one curious point (literally): the outlier > 0.5. This point probably corresponds to a slightly "pathological" test set, i.e. with different features from the rest of the data.

Finally, we can compare our RMSE from the test data to RMSE from the training data. If our model is robust, then these will have similar distributions:

```
map2_dbl(trained.models$model, trained.models$train, rmse) -> train.rmse


## Run a test on train/test rmse, remembering that these are PAIRED by k-fold!
wilcox.test(train.rmse, test.rmse, paired=T)
```

```
##
##  Wilcoxon signed rank test
##
```

```
## data:  train.rmse and test.rmse
## V = 28, p-value = 1
## alternative hypothesis: true location shift is not equal to 0
```

A quick hypothesis test to compare distributions (Wilcoxon signed-rank) gives P=0.695, suggesting that these RMSE values are from the same underlying distribution and do not significantly differ. In total, therefore, the K-fold cross validation showed us that we have a fairly robust model.

Performing cross validation on a logistic regression model

The procedure for cross-validation is the same for any type of model, but `modelr` does not have any convenience functions to summarize validations (as in, `rmse()` above). Therefore, in this and other such circumstances, we'll need to go through the full procedure of predicting from our test data and evaluating the predictions directly. Here, we will fit the logistic regression: `glm(outcome ~ ., data = biopsy, family=binomial)`, as in class.

We begin in much the same way:

```
biopsy <- read.csv("biopsy.csv")

## Make the folds and train the models
biopsy %>%
  crossv_kfold(10) %>%
  mutate(model = purrr::map(train, ~glm(outcome ~ ., data=., family=binomial))) -> trained.models

trained.models
```

```
## # A tibble: 10 x 4
##              train           test   .id     model
##             <list>         <list> <chr>    <list>
##  1 <S3: resample> <S3: resample>    01 <S3: glm>
##  2 <S3: resample> <S3: resample>    02 <S3: glm>
##  3 <S3: resample> <S3: resample>    03 <S3: glm>
##  4 <S3: resample> <S3: resample>    04 <S3: glm>
##  5 <S3: resample> <S3: resample>    05 <S3: glm>
##  6 <S3: resample> <S3: resample>    06 <S3: glm>
##  7 <S3: resample> <S3: resample>    07 <S3: glm>
##  8 <S3: resample> <S3: resample>    08 <S3: glm>
##  9 <S3: resample> <S3: resample>    09 <S3: glm>
## 10 <S3: resample> <S3: resample>    10 <S3: glm>
```

Again, we have our training and test data, their associated IDs, and the models fit to the training data. We now must actually grab predictions out of test, using some `tidyverse` magic with the `unnest()` and `map2()` functions. We want to run `predict()`, as usual, with the testing data in the `test` column with the trained models in `model` column. We use `map2()` to send both these inputs into `predict()` (with `type = "response"` to actually get predicted probabilities), and we shove the whole thing into `unnest()` to reveal the output.

```
trained.models %>%
  unnest( pred = map2( model, test, ~predict( .x, .y, type = "response")) ) -> test.predictions

test.predictions
```

```
## # A tibble: 683 x 2
##      .id        pred
##    <chr>       <dbl>
##  1    01 0.006050276
##  2    01 0.976505808
##  3    01 0.889464889
##  4    01 0.613395178
##  5    01 0.803507983
##  6    01 0.012506410
##  7    01 0.014735220
##  8    01 0.009638777
```

```
##  9    01 0.999022854
## 10    01 0.004247523
## # ... with 673 more rows
```

Our output here is a data frame where, for each row in each dataset (see the `.id` column!), we have predicted the probability of malignancy. In order to evaluate the model, we must compare these results to the **true** outcome in the test data. To do this, we need to modify our last line of code a little bit. We will include the function `broom::augment()` to get everything in one go. Recall: `augment()` not only does predictions, it also shows them in relation to the actual data (true outcome!). Therefore, we mainly use `augment()` to grab the `outcome` column from the test data. The function will do predictions as well (like the `predict()` line), but it will give us the *linear.predictors* when we really want the *fitted.values*, so we keep `predict()`.

```
trained.models %>%
  unnest( fitted = map2(model, test, ~augment(.x, newdata = .y)),
  pred = map2( model, test, ~predict( .x, .y, type = "response")) ) -> test.predictions

test.predictions %>% select(.id, outcome, pred )
```

```
## # A tibble: 683 x 3
##      .id   outcome        pred
##    <chr>    <fctr>       <dbl>
##  1    01    benign 0.006050276
##  2    01 malignant 0.976505808
##  3    01 malignant 0.889464889
##  4    01 malignant 0.613395178
##  5    01 malignant 0.803507983
##  6    01    benign 0.012506410
##  7    01    benign 0.014735220
##  8    01    benign 0.009638777
##  9    01 malignant 0.999022854
## 10    01    benign 0.004247523
## # ... with 673 more rows
```

Behold: A data frame with three columns:

- `.id`, the fold 1-10
- `outcome`, the TRUE outcome known from the test data
- `pred`, the PREDICTED probability of malignancy for the test data using the respective Kth training model

We can use this outcome to get an AUC for all testing folds, using the `pROC::roc()` function on columns in the dataframe:

```
test.predictions %>%
    group_by(.id) %>%
    summarize(auc = roc(outcome, .fitted)$auc) %>%
    select(auc)
```

```
## # A tibble: 10 x 1
##          auc
##        <dbl>
##  1 1.0000000
##  2 1.0000000
##  3 0.9956427
##  4 1.0000000
##  5 0.9909666
##  6 1.0000000
##  7 0.9920949
##  8 0.9963370
##  9 0.9845411
## 10 0.9957035
```

Now, these are some insanely high AUC's (all >= 0.98!), showing that we have a great model here. Does the training data show something similar? Let's find out using the same procedure we used to fit the test data - we simply replace the `test` column with the `train` column:

```
train.predictions <- trained.models %>% unnest(  fitted = map2(model, train, ~augment(.x, newdata = .y)),
                                                 pred = map2( model, train, ~predict( .x, .y, type = "response"
```

```
train.predictions %>%
  group_by(.id) %>%
  summarize(auc = roc(outcome, .fitted)$auc) %>%  ### outcome from the true data, .fitted from augment's outpu
  select(auc)
```

```
## # A tibble: 10 x 1
##          auc
##        <dbl>
##  1 0.9961449
##  2 0.9959055
##  3 0.9964423
##  4 0.9959880
##  5 0.9970153
##  6 0.9958824
##  7 0.9967117
##  8 0.9961694
##  9 0.9970644
## 10 0.9963982
```

Just as good! We are confirmed that the training and testing data give similar predictions (a nearly perfect model).

More advanced

Not satified with just AUC and wanting some confusion matrix measures across folds? Here some complex but fantastic `tidyverse` magic to get you there.

First, we need to convert the predicted test probabilities to predictions, assuming a cutoff of 0.5. We also need to summarize predictions across folds:

```
## How to change pred column from probability to real prediction
test.predictions %>%
  select(.id, outcome, pred ) %>%
  mutate(pred = ifelse(pred >= 0.5, "malignant", "benign"))
```

```
## # A tibble: 683 x 3
##      .id   outcome      pred
##    <chr>    <fctr>     <chr>
##  1    01    benign    benign
##  2    01 malignant malignant
##  3    01 malignant malignant
##  4    01 malignant malignant
##  5    01 malignant malignant
##  6    01    benign    benign
##  7    01    benign    benign
##  8    01    benign    benign
##  9    01 malignant malignant
## 10    01    benign    benign
## # ... with 673 more rows
```

```
## Tally it all up by fold
test.predictions %>%
  select(.id, outcome, pred ) %>%
  mutate(pred = ifelse(pred >= 0.5, "malignant", "benign")) %>%
  group_by(.id, outcome, pred) %>% tally()
```

```
## # A tibble: 34 x 4
## # Groups:   .id, outcome [?]
##      .id   outcome      pred     n
##    <chr>    <fctr>     <chr> <int>
```

```
##  1    01    benign    benign    44
##  2    01 malignant    benign     3
##  3    01 malignant malignant    22
##  4    02    benign    benign    46
##  5    02 malignant    benign     1
##  6    02 malignant malignant    22
##  7    03    benign    benign    50
##  8    03    benign malignant     1
##  9    03 malignant    benign     2
## 10    03 malignant malignant    16
## # ... with 24 more rows
```

Here's what we see, for fold 1:

- There are 38 benign predictions for benign patients, making **38 true negatives**
- There are 2 benign predictions for malignant patients, making **2 false negatives**
- There are 29 malignant predictions for malignant patients, making **29 true positive**
- There are 0 malignant predictions for benign patients, making **0 false positives**

Therefore, for fold 1, the True Positive rate is $29/(29+2) = 0.935$.

We can do this for all K's with a lot of (complicated but awesome) magic. I **strongly recommed** playing around with this code, line by line, until you understand what it does. Note in particular the `mutate()` line, which cycles through various conditions in order to assign either as True Positive (TP), FP, TN, or FN.

```
## Create a dataframe confusion matrix
test.predictions %>%
  select(.id, outcome, pred ) %>%
  mutate(pred = ifelse(pred >= 0.5, "malignant", "benign")) %>%
  group_by(.id, outcome, pred) %>%
  tally() %>%
  mutate(class =  ifelse(outcome == pred & pred == "malignant", "TP",
                  ifelse(outcome != pred & pred == "malignant", "FP",
                  ifelse(outcome == pred & pred == "benign", "TN", "FN")))) %>%
  ungroup() %>%  ### We want to ditch the `outcome` column, so remove it from grouping
  select(.id, n, class) %>%  ### Retain only columns of interest; use spread to get a column per classificatio
  spread(class, n) -> confusion

confusion
```

```
## # A tibble: 10 x 5
##      .id    FN    FP    TN    TP
##  * <chr> <int> <int> <int> <int>
##  1    01     3    NA    44    22
##  2    02     1    NA    46    22
##  3    03     2     1    50    16
##  4    04     1    NA    40    27
##  5    05     2     2    39    25
##  6    06    NA    NA    40    28
##  7    07     1     2    44    21
##  8    08     2     1    41    24
##  9    09    NA     2    43    23
## 10    10     1     2    47    18
```

```
## Use the tidyr::replace_na() function to replace all NA's in columns TP, TN, FP, FN with 0:
confusion <- replace_na(confusion, list(TP=0, TN=0, FP=0, FN=0))
confusion
```

```
## # A tibble: 10 x 5
##      .id    FN    FP    TN    TP
##  * <chr> <dbl> <dbl> <dbl> <dbl>
##  1    01     3     0    44    22
##  2    02     1     0    46    22
##  3    03     2     1    50    16
```

```
##  4    04     1     0    40    27
##  5    05     2     2    39    25
##  6    06     0     0    40    28
##  7    07     1     2    44    21
##  8    08     2     1    41    24
##  9    09     0     2    43    23
## 10    10     1     2    47    18
```

```
## Some classifer metric across all folds
confusion %>%
  group_by(.id) %>%
  summarize(TPR      = TP/(TP+FN),
            Accuracy = (TP+TN)/(TP+TN+FP+FN),
            PPV      = TP/(TP+FP)) -> fold.metrics

fold.metrics
```

```
## # A tibble: 10 x 4
##       .id       TPR  Accuracy       PPV
##    <chr>     <dbl>     <dbl>     <dbl>
##  1    01 0.8800000 0.9565217 1.0000000
##  2    02 0.9565217 0.9855072 1.0000000
##  3    03 0.8888889 0.9565217 0.9411765
##  4    04 0.9642857 0.9852941 1.0000000
##  5    05 0.9259259 0.9411765 0.9259259
##  6    06 1.0000000 1.0000000 1.0000000
##  7    07 0.9545455 0.9558824 0.9130435
##  8    08 0.9230769 0.9558824 0.9600000
##  9    09 1.0000000 0.9705882 0.9200000
## 10    10 0.9473684 0.9558824 0.9000000
```

```
## Finally we can summarize:
fold.metrics %>% summarize(meanTPR = mean(TPR), meanAcc = mean(Accuracy), meanPPV=mean(PPV))
```

```
## # A tibble: 1 x 3
##     meanTPR   meanAcc   meanPPV
##       <dbl>     <dbl>     <dbl>
## 1 0.9440613 0.9663257 0.9560146
```

At long last, our K-fold validation gave us a mean TPR of 0.95, mean accuracy of 0.966, and mean positive predictive of 0.95. Together, this all points to a very good model.