Introduction to model evaluation/validation

Stephanie J. Spielman

Data Science for Biologists, Spring 2020

Model evaluation and prediction

- R^2 tells how how much variation is explained **in the data the model was FIT ON**
  - fit on <===> "trained on"
- How good is the model at explaining variation **in data it does NOT know about?**
  - Should we even bother using our model to predict future outcomes?

More measures of model evaluation

- RMSE ("Root mean squared error") and MAE ("Mean absolute error")
  - Easily interpreted in units of "Y"
  - RMSE is very common! "Average" error we can expect when using this model
- Less easily interpreted, but also commonly-used evaluation measurements
  - MSE = mean squared error

`modelr` makes life easy!!

```
# use trace = F to suppress excessive output which hurts my eyeballs
fit <- step( lm(Sepal.Length ~ ., data = iris), trace = F  )
broom::tidy(fit)
## # A tibble: 6 x 5
##   term              estimate std.error statistic  p.value
##   <chr>                <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)          2.17     0.280       7.76 1.43e-12
## 2 Sepal.Width          0.496    0.0861      5.76 4.87e- 8
## 3 Petal.Length         0.829    0.0685     12.1  1.07e-23
## 4 Petal.Width         -0.315    0.151      -2.08 3.89e- 2
## 5 Speciesversicolor   -0.724    0.240      -3.01 3.06e- 3
## 6 Speciesvirginica    -1.02     0.334      -3.07 2.58e- 3
broom::glance(fit)
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <int>  <dbl> <dbl> <dbl>
## 1     0.867         0.863 0.307      188. 2.67e-61     6  -32.6  79.1  100.
## # … with 2 more variables: deviance <dbl>, df.residual <int>
```

`modelr` makes life easy!!

```
# use trace = F to suppress excessive output which hurts my eyeballs
fit <- step( lm(Sepal.Length ~ ., data = iris), trace = F  )
broom::tidy(fit)
## # A tibble: 6 x 5
##   term              estimate std.error statistic  p.value
##   <chr>                <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)          2.17     0.280       7.76 1.43e-12
## 2 Sepal.Width          0.496    0.0861      5.76 4.87e- 8
## 3 Petal.Length         0.829    0.0685     12.1  1.07e-23
## 4 Petal.Width         -0.315    0.151      -2.08 3.89e- 2
## 5 Speciesversicolor   -0.724    0.240      -3.01 3.06e- 3
## 6 Speciesvirginica    -1.02     0.334      -3.07 2.58e- 3
broom::glance(fit)
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <int>  <dbl> <dbl> <dbl>
## 1     0.867         0.863 0.307      188. 2.67e-61     6  -32.6  79.1  100.
## # … with 2 more variables: deviance <dbl>, df.residual <int>
```

```
## Obtain R^2 quickly with modelr
modelr::rsquare(fit, iris)
## [1] 0.8673123

## Obtain RMSE quickly with modelr
modelr::rmse(fit, iris)
## [1] 0.300627
```

Use that model going forward easily

```
## reminding you of the fit variable:
fit <- step( lm(Sepal.Length ~ ., data = iris), trace = F  )


## Extract the FORMULA as fit$formula
lm(fit$formula, data = iris)
##
## Call:
## lm(formula = fit$formula, data = iris)
##
## Coefficients:
##       (Intercept)        Sepal.Width       Petal.Length
##            2.1713             0.4959             0.8292
##       Petal.Width  Speciesversicolor   Speciesvirginica
##           -0.3152            -0.7236            -1.0235
```

Validation with testing/training

- Randomly **split** your dataset into two parts:
  - The "training" part (usually 60-80% of the data) **builds** aka **trains** the model
  - The "testing" part (the remaining 20-40%) evaluates aka **tests** the performance of the model
  - If model performs terribly on testing data, suggests model was *overfit*
  - Either way, performance is usually better on training data. **Why?**

Cross validation with a training and testing split

```
# Use dplyr::sample_frac() to randomly sample a fraction of rows
training_iris <- sample_frac(iris, 0.7) ## 70% into training
nrow(training_iris)
## [1] 105
```

Cross validation with a training and testing split

```
# Use dplyr::sample_frac() to randomly sample a fraction of rows
training_iris <- sample_frac(iris, 0.7) ## 70% into training
nrow(training_iris)
## [1] 105
```

```
# Get the "anti training" for testing.. with anti_join()!
# In anti_join(), the FULL data goes FIRST!!
testing_iris <- anti_join(iris, training_iris) ## remaining 30% into training
## Joining, by = c("Sepal.Length", "Sepal.Width", "Petal.Length",
## "Petal.Width", "Species")
nrow(testing_iris)
## [1] 45
```

Cross validation with a training and testing split

```
# Use dplyr::sample_frac() to randomly sample a fraction of rows
training_iris <- sample_frac(iris, 0.7) ## 70% into training
nrow(training_iris)
## [1] 105
```

```
# Get the "anti training" for testing.. with anti_join()!
# In anti_join(), the FULL data goes FIRST!!
testing_iris <- anti_join(iris, training_iris) ## remaining 30% into training
## Joining, by = c("Sepal.Length", "Sepal.Width", "Petal.Length",
## "Petal.Width", "Species")
nrow(testing_iris)
## [1] 45
```

```
## TRAIN the model on training data: data = training_iris !!
trained_model <- lm(fit$formula, data = training_iris)
```

- We should **NOT** use `step()` here. WHY?????

Compare training metrics to those on TESTING data

```
## How does the model do on data it was TRAINED ON?
modelr::rsquare(trained_model, training_iris)
## [1] 0.8797924
modelr::rmse(trained_model, training_iris)
## [1] 0.2977816
```

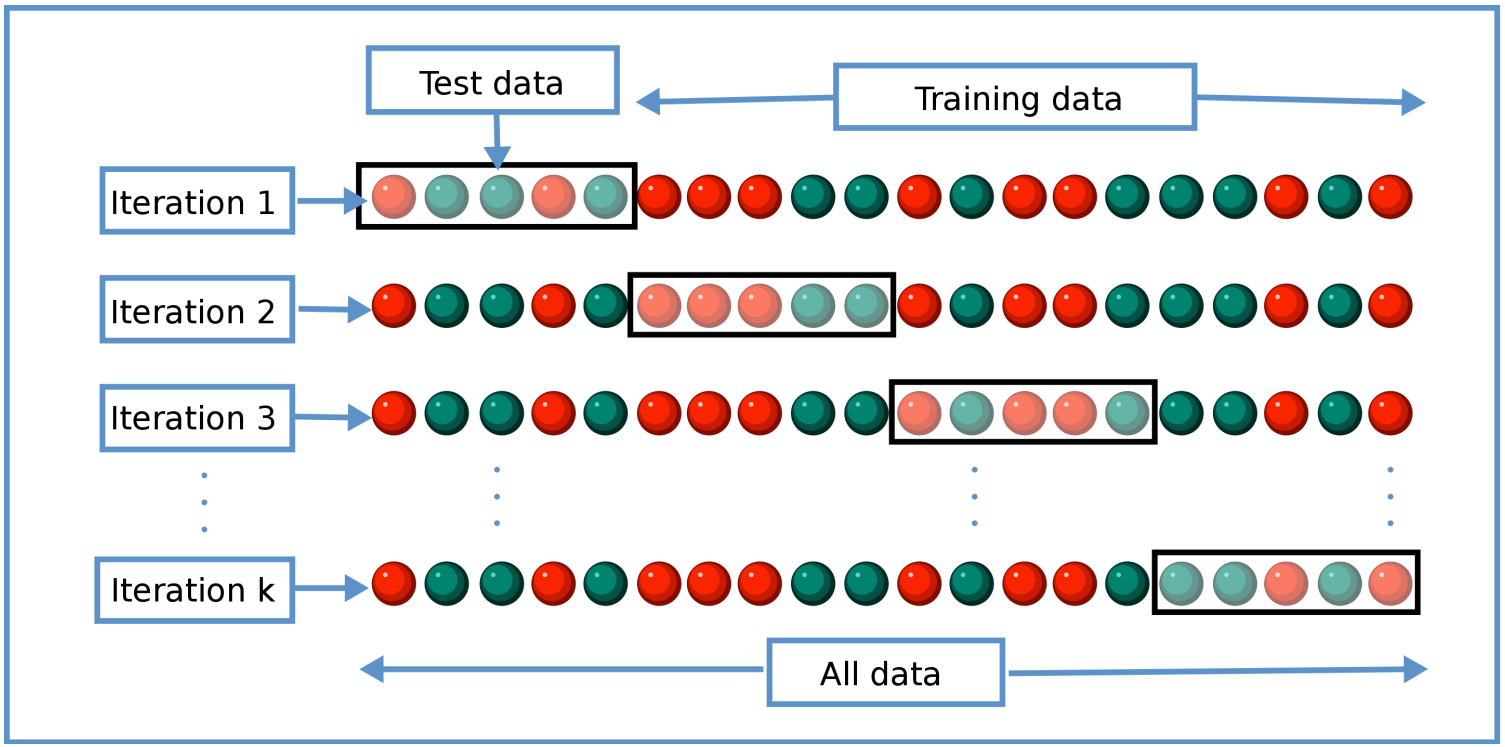
Compare training metrics to those on TESTING data

```
## How does the model do on data it was TRAINED ON?
modelr::rsquare(trained_model, training_iris)
## [1] 0.8797924
modelr::rmse(trained_model, training_iris)
## [1] 0.2977816
```

```
## How does the model do on data it has NEVER SEEN? The testing data!!
modelr::rsquare(trained_model, testing_iris)
## [1] 0.8119333
modelr::rmse(trained_model, testing_iris)
## [1] 0.3214049
```

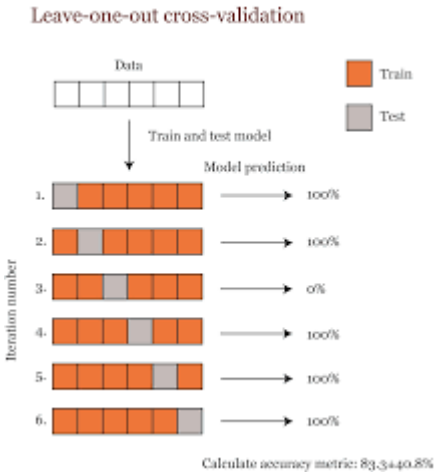
K-fold cross validation

- Randomly divide the whole dataset into "K" equal chunks aka folds
- Perform K iterations of model training and testing
  - "Hold back" data each time for testing!
- Get RMSE and R^2 for each iteration, and look at full distribution



More robust: Leave-one-out cross validation (LOOCV)

- K-folds on speed: each "test" size is N=1!!
- For small datasets, LOOCV probably "better"



Running a K-fold CV to evaluate a model that predict Sepal Lengths

```
# decide your K!
folds <- 10

# Use the amazzzzing function modelr::crossv_kfold()
crossv_kfold(iris, folds)
## # A tibble: 10 x 3
##    train        test         .id
##    <named list> <named list> <chr>
##  1 <resample>   <resample>   01
##  2 <resample>   <resample>   02
##  3 <resample>   <resample>   03
##  4 <resample>   <resample>   04
##  5 <resample>   <resample>   05
##  6 <resample>   <resample>   06
##  7 <resample>   <resample>   07
##  8 <resample>   <resample>   08
##  9 <resample>   <resample>   09
## 10 <resample>   <resample>   10
```

A necessary detour: functional programming with `purrr`

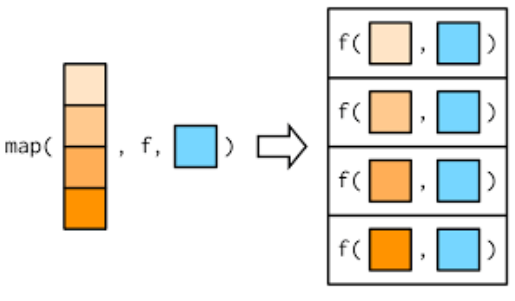
```
## log of a number
log(5)
## [1] 1.609438

## log of an array of numbers
log(1:4)
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
```

A necessary detour: functional programming with `purrr`

```
## log of a number
log(5)
## [1] 1.609438

## log of an array of numbers
log(1:4)
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
```




```
## using purrr::map returns a !!!LIST!!!
purrr::map( 1:4, log )
## [[1]]
## [1] 0
##
## [[2]]
## [1] 0.6931472
##
## [[3]]
## [1] 1.098612
##
## [[4]]
## [1] 1.386294
```

map_TYPE to specify a different type of output

```
## purrr, I'd really like an array of *doubles* to come out of this
purrr::map_dbl(1:4, log)
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
```

map_TYPE to specify a different type of output

```
## purrr, I'd really like an array of *doubles* to come out of this
purrr::map_dbl(1:4, log)
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
```

```
## map2 means there are TWO inputs
## recall: log has a second optional argument for BASE! The output is log base 2
purrr::map2_dbl(1:4, 2, log)
## [1] 0.000000 1.000000 1.584963 2.000000
```

A second necessary detour: writing our own functions

```
add_two_numbers <- function(a, b)
{
  a + b
}


add_two_numbers(10, 12)
## [1] 22
add_two_numbers(5, -5)
## [1] 0
```

Back to K-fold

```
crossv_kfold(iris, folds) -> iris_kfold
iris_kfold
## # A tibble: 10 x 3
##    train        test         .id
##    <named list> <named list> <chr>
##  1 <resample>   <resample>   01
##  2 <resample>   <resample>   02
##  3 <resample>   <resample>   03
##  4 <resample>   <resample>   04
##  5 <resample>   <resample>   05
##  6 <resample>   <resample>   06
##  7 <resample>   <resample>   07
##  8 <resample>   <resample>   08
##  9 <resample>   <resample>   09
## 10 <resample>   <resample>   10
```

Using `purrr::map` to run a model at each row

```
## DEFINE THE FUNCTION!! IT'S JUST THE MODEL!!!
## THIS IS IMPORTANT: As written ASSUMES (!!!!!!) fir$formula was defined in code
ABOVE this function
my_iris_model <- function(input_data){
  lm(fit$formula, data = input_data)
}

iris_kfold %>%
  mutate( model_fit = purrr::map( train, my_iris_model ) )
## # A tibble: 10 x 4
##    train        test         .id   model_fit
##    <named list> <named list> <chr> <named list>
##  1 <resample>   <resample>   01    <lm>
##  2 <resample>   <resample>   02    <lm>
##  3 <resample>   <resample>   03    <lm>
##  4 <resample>   <resample>   04    <lm>
##  5 <resample>   <resample>   05    <lm>
##  6 <resample>   <resample>   06    <lm>
##  7 <resample>   <resample>   07    <lm>
##  8 <resample>   <resample>   08    <lm>
##  9 <resample>   <resample>   09    <lm>
## 10 <resample>   <resample>   10    <lm>
```

Using `purrr:map2_dbl` to get our metrics

- Recall: `modelr::rmse(MODEL, DATA)`
- Recall: `modelr::rsquare(MODEL, DATA)`

```
iris_kfold %>%
  mutate( model_fit = purrr::map( train, my_iris_model ) ) %>%
  mutate( test_rmse = purrr::map2_dbl(model_fit, test, rmse),
          test_rsq  = purrr::map2_dbl(model_fit, test, rsquare))
## # A tibble: 10 x 6
##    train        test         .id   model_fit    test_rmse test_rsq
##    <named list> <named list> <chr> <named list>     <dbl>    <dbl>
##  1 <resample>   <resample>   01    <lm>             0.338    0.628
##  2 <resample>   <resample>   02    <lm>             0.297    0.909
##  3 <resample>   <resample>   03    <lm>             0.236    0.942
##  4 <resample>   <resample>   04    <lm>             0.343    0.853
##  5 <resample>   <resample>   05    <lm>             0.376    0.891
##  6 <resample>   <resample>   06    <lm>             0.310    0.800
##  7 <resample>   <resample>   07    <lm>             0.336    0.849
##  8 <resample>   <resample>   08    <lm>             0.255    0.926
##  9 <resample>   <resample>   09    <lm>             0.367    0.707
## 10 <resample>   <resample>   10    <lm>             0.257    0.740
```

Putting it all together: model selection and k-fold CV

```
# Initial model selection (with ENTIRE data set) to determine predictors
fit <- step(lm(Sepal.Length ~ ., data = iris), trace =  F)

# Function for use in cross-validation
my_iris_model <- function(input_data){
  lm(fit$formula, data = input_data)
}

# Validate your model
folds <- 10
crossv_kfold(iris, folds) %>%
  mutate( model_fit = purrr::map(train, my_iris_model),
          test_rmse = purrr::map2_dbl(model_fit, test, rmse),
          test_rsq  = purrr::map2_dbl(model_fit, test, rsquare)) -> result_kfold
```

Summarizing our results

```
result_kfold %>%
  summarize(mean_rmse = mean(test_rmse),
            mean_rsq  = mean(test_rsq))
## # A tibble: 1 x 2
##   mean_rmse mean_rsq
##       <dbl>    <dbl>
## 1     0.313    0.835
```

- We expect, when used on data the model has never seen, it will predict R^2 of variation in sepal lengths
- We expect, when used on data the model has never seen, the model predictions will be roughly *RMSE* units off (the average residual is the RMSE value)

Summarizing our results

```
result_kfold %>%
  summarize(mean_rmse = mean(test_rmse),
            mean_rsq  = mean(test_rsq))
## # A tibble: 1 x 2
##   mean_rmse mean_rsq
##       <dbl>    <dbl>
## 1     0.313    0.835
```

- We expect, when used on data the model has never seen, it will predict R^2 of variation in sepal lengths
- We expect, when used on data the model has never seen, the model predictions will be roughly *RMSE* units off (the average residual is the RMSE value)

```
summary(iris$Sepal.Length)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   5.100   5.800   5.843   6.400   7.900
```

Visualizing our evaluation

- Any kind of standard continuous distribution plot will do here:

```
result_kfold %>%
  ggplot(aes(x = "", y = test_rmse)) + geom_boxplot() +
  xlab("") + ylab("Mean RMSE")-> rmse_box

result_kfold %>%
  ggplot(aes(x = "", y = test_rsq)) + geom_boxplot() +
  xlab("") + ylab("Mean R^2") -> rsq_box

## using patchwork:
rmse_box + rsq_box
```

How to do LOOCV?

- Same exact way except use **`modelr::crossv_loo()`**
  - BUT I think there's a bug in **`rsquare()`** when used on LOO output, so let's just do RMSE.

```
# Reminding you of this function, but you only need to define it once
my_iris_model <- function(input_data){
  lm(fit$formula, data = input_data)
}


crossv_loo(iris) %>%
  mutate( model_fit = purrr::map( train, my_iris_model ),
          test_rmse = purrr::map2_dbl(model_fit, test, rmse)) -> result_iris_loo

# Summary
result_iris_loo %>%
  summarize(mean_rmse = mean(test_rmse))
## # A tibble: 1 x 1
##   mean_rmse
##       <dbl>
## 1     0.253
```

Making predictions!

Once we have our model, we can predict future outcomes. **Remember: Model evaluation is NOT the same as model fitting.**

```
# reminder for what our model predictors are
broom::tidy(fit)
## # A tibble: 6 x 5
##   term              estimate std.error statistic  p.value
##   <chr>                <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)          2.17     0.280       7.76 1.43e-12
## 2 Sepal.Width          0.496    0.0861      5.76 4.87e- 8
## 3 Petal.Length         0.829    0.0685     12.1  1.07e-23
## 4 Petal.Width         -0.315    0.151      -2.08 3.89e- 2
## 5 Speciesversicolor   -0.724    0.240      -3.01 3.06e- 3
## 6 Speciesvirginica    -1.02     0.334      -3.07 2.58e- 3
```

Making predictions!

- Predict using a tibble with columns **EXACTLY NAMED** as model predictor variables in the model formula

```
## Oh look a new observation where we measured PREDICTORS
new_iris <- tibble(Sepal.Width  = 4.1,
                   Petal.Length = 3.7,
                   Petal.Width  = 1.1,
                   Species      = "virginica")

# arguments in order: NEWDATA, MODEL
modelr::add_predictions(new_iris, fit)
## # A tibble: 1 x 5
##   Sepal.Width Petal.Length Petal.Width Species    pred
##         <dbl>        <dbl>       <dbl> <chr>     <dbl>
## 1         4.1          3.7         1.1 virginica  5.90

# or, base R with OPPOSITE arguments. womp.
predict(fit, new_iris)
##        1
## 5.902445
```

BONUS Q: Why did the **`modelr`** authors flip these arguments?