Cross-Validation

A 'what is it' and 'how to' for those out of the loop

In this post you will learn what Cross-Validation is, and how to do it in R and Python.

What is Cross-Validation

Cross-Validation is an analytical method for assessing how a modeling technique might generalize to new data gathered separately from the data set used to build the model. The technique can be referred to as CV, rotation estimation, or out-of-sample testing. CV draws upon techniques of random sampling and applies them to model validation within supervised learning task. There are several different methods for implementing Cross-Validation.

- Leave One Out Cross-Validation: R and Python
- Leave **p** Out Cross-Validation: Explanation
- k-Fold Cross Validation: R and Python
  - i-Nested k-Fold Cross Validation: R and Python
- Hold Out Cross Validation: R and Python
- Repeated Random Sub-Sampling: R and Python

Which is the best method for my task? Well, that depends. Leave p Out Cross-Validation can be computationally expensive, and maybe you can glean the same info from a 80/20 hold out cross validation. Perhaps the modeling task needs to meet strict prediction criteria before being deployed into production, so maybe you'll consider i-Nested k-Fold CV. It will be up to the Data Scientist to determine how they want to validate the performance of their models. Below I show case how each method is implemented in R and Python. Click on the programming language next to any method to skip ahead to that method's implementation.

How to Cross Validate

In true to tutorial fashion, I will be using the iris data set since most data scientist have access to this popular compilation of data by Ronald Fisher. I will be fiting a linear model. The residual will be quantitated.

Some Set Up

I will be using the iris data set:

```r
head(iris)
```

```
## # A tibble: 6 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
```

It is appropriate to create a test and training partition, even when using CV procedures.

```r
dim(iris)
```

```
## [1] 150   5
```

```r
iris_split <- initial_split(iris, prob = 0.8)
iris_train <- training(iris_split)
iris_test  <- testing(iris_split)
```

Leave One Out CV

Whats happening?

In this exhaustive method, we leave one record out of the model fitting process. The fitted model is then fed that left out observation. The prediction is compared to the actual result in a residual calculation. The residual calculation for each record is then averaged.

When to use?

Use on data sets expected to be "small". Here, I would judge a data set as small based on how computationally expensive it would be to run a Leave One Out CV process. It takes my machine just under a second (~750 milliseconds), which is alright, but it would not scale well. If one were to deploy this method on a cloud computing machine for several thousands of users' data, then the incurred cost in selecting a method would likely be high.

```r


err <- c()

# for each observation in our data set
for (i in 1:nrow(iris_train)) {

  # fit a model leaving that observation out
  lm <- lm(Petal.Length ~ Sepal.Length, data = iris_train[-i,])

  # then store the squared residual
  err[i] <- (iris_train$Petal.Length[i] - predict(lm, newdata = iris_train[i,]))**2
}

# report the average squared residual
print(round(mean(err), 4))
```

Leave One Out CV in R

```
## [1] 0.812
```

```python
#from sklearn import linear_model
import numpy as np
import pandas as pd
from sklearn import linear_model
from sklearn import metrics

# Point arr to the object iris dataframe
arr = r.iris_train

# Place holder for the mean square term
err = []

# Iterate through each observation in the dataframe
for i in range(0, arr.shape[0]):

  # Call the linear regression method from sklearn
  reg = linear_model.LinearRegression()

  # Point easily callable names to the training vectors
  # This code also drops the one observation out
  obs = arr.loc[arr.index.difference([i]), 'Sepal.Length']
  resp = arr.loc[arr.index.difference([i]), 'Petal.Length']

  # Fit the linear model
  reg.fit(obs.values.reshape(-1, 1), resp.values.reshape(-1, 1))

  # Point the true value and predicted value to easily callable objects
  y = arr.loc[i, 'Petal.Length']
  yhat = reg.predict(np.array(arr.loc[i, 'Sepal.Length']).reshape(-1, 1))

  # Quantitate the mean square error for each model fit
  err.append(metrics.mean_squared_error([y], yhat))
```

```python
print(round(sum(err)/len(err), 4))
```

Leave One Out CV in Python

```
## 0.812
```

Leave p Out CV

Whats happening?

In this exhaustive method, we leave some **p** observations out of the model fitting process. The fitted model is then fed the left out observations. The prediction is compared to the actual result via a mean squared error calculation. Additionally, we fit **every** combination of **p** variables, where order does not matter. So this is even more computationally expensive than leave one out CV.

As a quick example of how this pairing works, consider this data set of 5 entries:

```
##   predictor response
## 1         1        A
## 2         2        B
## 3         3        C
## 4         4        D
## 5         5        E
```

A leave $\mathbf{p = 2}$ CV method would mean that we fit a model which leaves the following observations out as it iterates though:

```
##   predictor response
## 1         1        A
## 2         2        B
##   predictor response
## 1         1        A
## 3         3        C
##   predictor response
## 1         1        A
## 4         4        D
##   predictor response
## 1         1        A
## 5         5        E
##   predictor response
## 2         2        B
## 3         3        C
##   predictor response
## 2         2        B
## 4         4        D
```

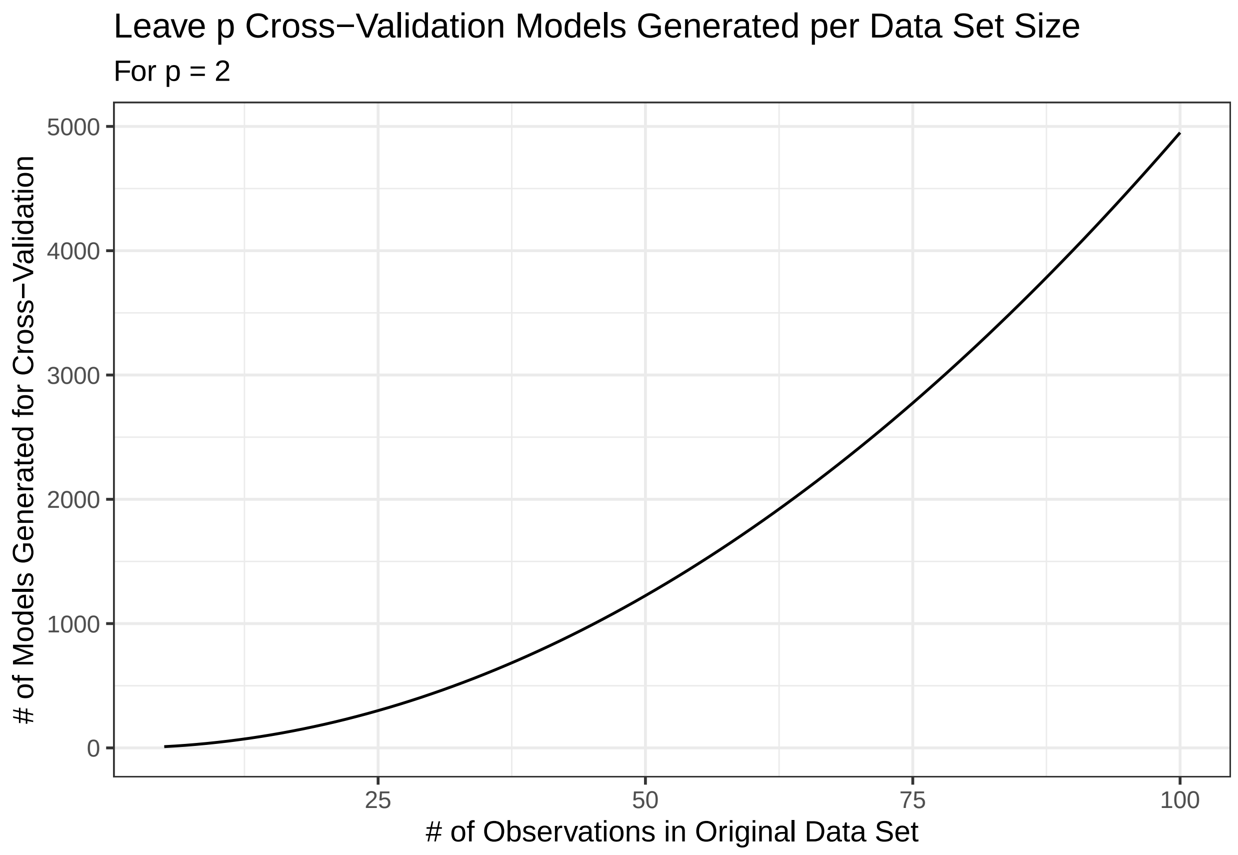
Using combinatorics we can figure out how many total models will be fitted. Order does not matter, so it's 5! / (2! * (5 - 2)!).

5! / (2! * 3!) = (5 * 4 * 3 * 2 * 1) / (2 * 1 * 3 * 2 * 1) = 20 / 2 = 10 models.

In a Leave One Out approach we would only produce 5 models.

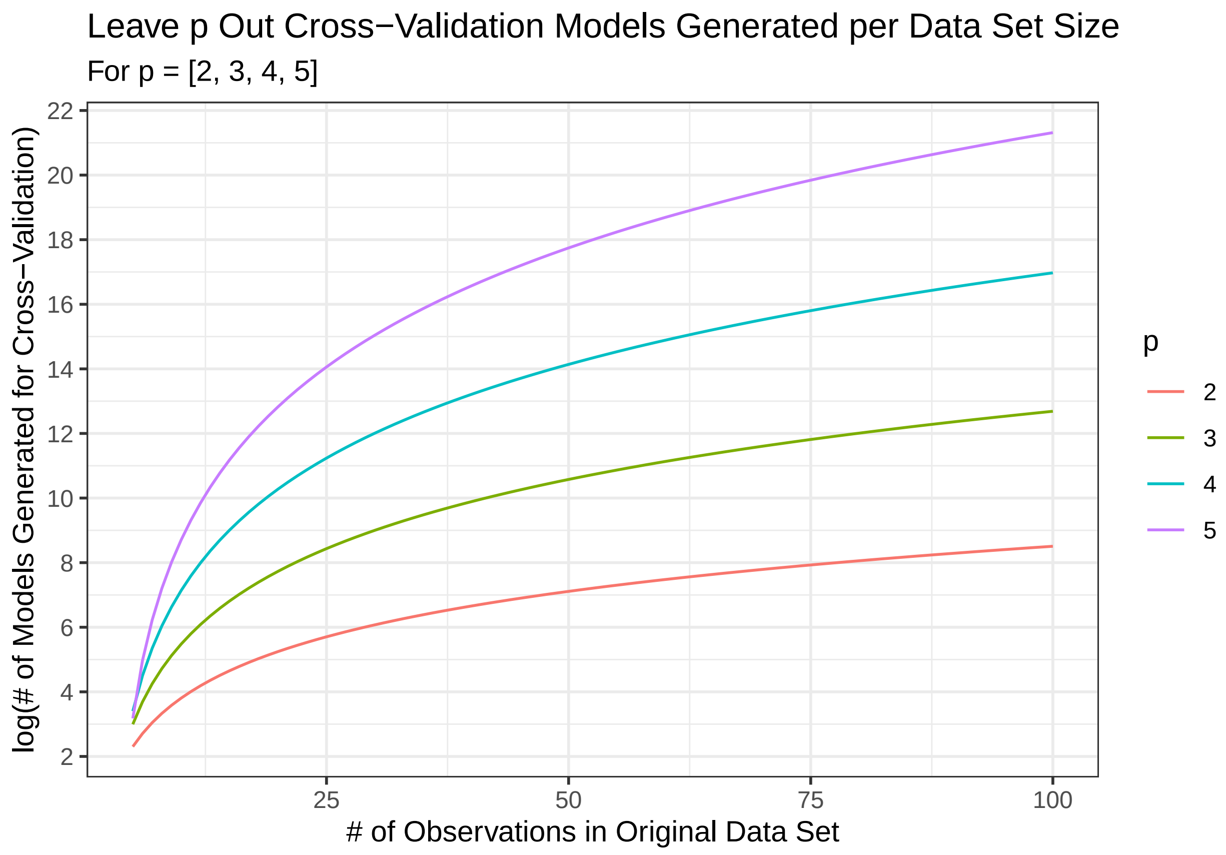
So that was only for a data set of 5 variables. How many models do we produce as our data set increases from 5, to 20, 50, or 100?

```
exhaustive_cv <- data.frame(records = 5:100) %>%
  mutate(models = factorial(records) / (2 * factorial(records - 2)))

exhaustive_cv %>%
  ggplot() +
  geom_line(aes(x = records, y = models)) +
  labs(x = '# of Observations in Original Data Set',
       y = '# of Models Generated for Cross-Validation',
       title = 'Leave p Cross-Validation Models Generated per Data Set Size',
       subtitle = 'For p = 2')
```



5000 models are generated for data set with 100 observations. For modern computation, 5000 iterations is easily churned through. Though, a data set of 100 observations is very small for the type of Data Science tasks commonly done. This is just for $\mathbf{p = 2}$ as well…

```
large_p <- expand.grid(records = 5:100, p = 2:5) %>%
  mutate(models = factorial(records) / (p * factorial(records - p)),
         p = factor(p))

large_p %>%
  ggplot() +
  geom_line(aes(x = records, y = log(models), color = p)) +
  labs(x = '# of Observations in Original Data Set',
       y = 'log(# of Models Generated for Cross-Validation)',
       title = 'Leave p Out Cross-Validation Models Generated per Data Set Size',
       subtitle = 'For p = [2, 3, 4, 5]') +
  scale_y_continuous(breaks = scales::pretty_breaks(8))
```



I logarthmically transformed the y-axis to ensure that the data is viewable. It's clear that the number of models explode as **p** grows, and this is just for a data set of 100 observations.

For 100 observations and **p = 5** the number of models generated is 21ˆe, or 1.8 **billion**. Exhaustive methods are computationally infeasible to justify scaling up when there are other methods that approximate well enough.

k-Fold Cross-Validation

Whats happening?

For most modeling validation procedures, k-Fold CV is a sufficient standard to validate by. When implemented correctly. For k-Fold CV our data set is randomly sampled into **k** different groups. Typically **k** is set to 10.

The model is fitted on all data but the **k**th group. The **k**th left out group is then used as testing data, which means that it is fed into the fitted model and a goodness of fit metric is calculated. This procedure is iterated until each of the **k** groups has been left out of the fitting step once. The resulting **k** goodness of fit metrics are then averaged to produce a final single metric to compare to other modeling procedures.

Max Kuhn and Julia Silge has done a fantastic write up on this method, with great visuals. If k-Fold CV is new to you, then check them out as a great starting resource: Tidy Modeling in R

When to use?

k-Fold CV is a reliable method for understanding how a model might generalize to independent new data. It also goes by V-Fold Cross Validation. This method computationally scales up well. The biggest concern with k-Fold CV is implementing it correctly. Recall that cross-validation is a procedure to validate a model and understand how it might generalize. *It is not* a procedure for feature selection nor is it for naive model selection.

I will continue using the Iris data set. Additionally, I will follow the play book from the Tidy Modeling in R book by Max and Julia.

```r
iris_folds <- vfold_cv(iris_train, v = 10)

lm_fit <- lm_model %>%
  fit_resamples(Petal.Length ~ Sepal.Length, resamples = iris_folds)

lm_fit
```

k Fold CV in R

```
## # Resampling results
## # 10-fold cross-validation
## # A tibble: 10 x 4
##    splits           id     .metrics         .notes
##    <list>           <chr>  <list>           <list>
##  1 <split [101/12]> Fold01 <tibble [2 x 3]> <tibble [0 x 1]>
##  2 <split [101/12]> Fold02 <tibble [2 x 3]> <tibble [0 x 1]>
##  3 <split [101/12]> Fold03 <tibble [2 x 3]> <tibble [0 x 1]>
##  4 <split [102/11]> Fold04 <tibble [2 x 3]> <tibble [0 x 1]>
##  5 <split [102/11]> Fold05 <tibble [2 x 3]> <tibble [0 x 1]>
##  6 <split [102/11]> Fold06 <tibble [2 x 3]> <tibble [0 x 1]>
##  7 <split [102/11]> Fold07 <tibble [2 x 3]> <tibble [0 x 1]>
##  8 <split [102/11]> Fold08 <tibble [2 x 3]> <tibble [0 x 1]>
##  9 <split [102/11]> Fold09 <tibble [2 x 3]> <tibble [0 x 1]>
## 10 <split [102/11]> Fold10 <tibble [2 x 3]> <tibble [0 x 1]>
```

The Tidymodels package has made k-Fold CV in R rather succint. Our RMSE calculation is automatically calculated by the method. Our metric is embedded in a tibble.

```r
lm_fit %>%
  mutate(RMSE =
           unlist(
             map(.metrics,
                 function(x) {
                   x %>%
                     filter(.metric == 'rmse') %>%
                     pull(.estimate)
                   }
                 )
             )
         ) %>%

  summarise(avg_RMSE = mean(RMSE))
```

```
## # A tibble: 1 x 1
##   avg_RMSE
##       <dbl>
## 1     0.891
```