Model-evaluation

To evaluate machine learning algorithms, you really only need 3 things:

1. The dataset (obviously)
2. Test options
3. Evaluation metrics

I will explain these three things in detail below.

Required packages

```
install.packages(c("mlbench", "caret", "plotROC", "klaR"), dependencies=TRUE)
```

Test options

- **Split data** into training, test, and validation sets. Use your training set to train the model, use your validation set to fine tune the hyper-parameters, and then measure your model's predictive performance on the test set.
- **K-fold cross validation**: partition data into k subsets, alliteratively leave one subset out as test set, use the rest for training
- **Bootstrap resampling**: randomly sample from the dataset (with re-selection) against which to evaluate the model. In aggregate, the results provide an indication of the variance of the model's performance. Typically, a large number of re-sampling iterations are performed (thousands or tens of thousands).

If you are unsure which test options to use, I recommend start by using a 10-fold cross validation with multiple runs.

How to choose a test option

Understand randomness

The key concept to understand is randomness, a.k.a **stochasticity**. Most ML algorithms use some randomness in one way or another. Either the algorithm itself is random, such as stochastic gradient descent, or there is randomness within the training dataset. This does not mean that the predictions are going to be random, it implies that is **variance** in the predictions. Choosing the right test option is all about accounting for this variance.

Why not just train and test on the same dataset?

Lets say you trained a model on a dataset, then measured its accuracy on the same dataset. You get an accuracy of 95%, that's the true accuracy of the algorithm on the dataset, right? Not exactly...

What you did is over-fit on the dataset for training, and have no idea how well your model can generalize on **new** data.

Since we actually want to predict on new data, this is not a viable approach.

Split data

Advantages:

- easy to implement

Disadvantages:

- you lose a good amount of data from training the model, so there's high bias to the training set

You take the dataset, and split it into a training and a test set (and maybe a third validation set for hyper-parameter search, but we won't cover that here).

For example, you randomly shuffle the entire dataset, then use 2/3 as the training set and 1/3 as the test set. The algorithm runs on the training set, then the trained model is evaluated on the test set, and you get something like 85% accuracy.

```
# load the libraries
library(caret)
library(klaR)
# load the iris dataset
data(iris)
# define an 80%/20% train/test split of the dataset
split=0.80
trainIndex <- createDataPartition(iris$Species, p=split, list=FALSE)
data_train <- iris[ trainIndex,]
data_test <- iris[-trainIndex,]
# train a naive bayes model
model <- NaiveBayes(Species~., data=data_train)
# make predictions
x_test <- data_test[,1:4]
y_test <- data_test[,5]
predictions <- predict(model, x_test)
# summarize results
confusionMatrix(predictions$class, y_test)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         10          0         0
##   versicolor      0          8         1
##   virginica       0          2         9
##
## Overall Statistics
##
##                Accuracy : 0.9
##                  95% CI : (0.7347, 0.9789)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : 1.665e-10
##
##                   Kappa : 0.85
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: setosa Class: versicolor Class: virginica
```

```
## Sensitivity                 1.0000          0.8000          0.9000
## Specificity                 1.0000          0.9500          0.9000
## Pos Pred Value              1.0000          0.8889          0.8182
## Neg Pred Value              1.0000          0.9048          0.9474
## Prevalence                  0.3333          0.3333          0.3333
## Detection Rate              0.3333          0.2667          0.3000
## Detection Prevalence        0.3333          0.3000          0.3667
## Balanced Accuracy           1.0000          0.8750          0.9000
```

This is a fast and easy approach when you have a lot of data. When the dataset is very big, splitting the data can actually yield a fairly accurate measure of the actual performance of the algorithm.

But how good is the algorithm really? How **confident** are we that its true positive rate is 87%?

Lets say we use a different random seed this time, and split the data the same way – into 2/3 train and 1/3 test – we would in fact get a different result.

How do we account for this variance?

We can try to reduce the variance of the random process by doing it multiple times and getting the average. For example, we do the split data above 10 times, each time we set a new seed and get a new accuracy, then we compute the average and standard deviations of those accuracies.

But there is still the possibility that some samples are never used as either training or testing, while some samples are used multiple times. This might cause the algorithm to over-fit on those samples scene multiple times.

Solution: *k-fold cross validation.*

Cross validation (CV)

Advantages:

- CV uses more test data
- CV uses the entire training dataset for both training and evaluation, instead of some portion. In contrast, if you validate a model by using data generated from a random split, typically you evaluate the model only on 30% or less of the available data.
- CV evaluates the dataset as well as the model.
- CV does not simply measure the accuracy of a model, but also gives you some idea of how representative the dataset is and how sensitive the model might be to variations in the data.

Disadvantages:

- Because CV trains and validates the model multiple times over a larger dataset, it is much more computationally intensive and takes much longer than validating on a random split.
- If the dataset is very big, you may have to resort to either sampling the data or stick to a split test.

Here, CV specifically refers to k-fold cross validation, where k is the number of splits we make in the data.

For example, suppose k=10. This will split the dataset into 10 subsets (folds), and the algorithm will iterate 10 times, each time it trains on 9 folds and tests on 1 without replacement. This way, each data sample will be used as training 9 times, and as test 1 time. The combined accuracy is not an average, but instead an exact accuracy measure on how many correct predictions were made.

Although CV does provide an unbiased estimate on the model's accuracy, it does not know a stochastic algorithm's internal randomness. This means it cannot account for the variance in the model's predictions. Furthermore, the splitting of data into folds is itself a random process, which means a single run does not estimate how the algorithm performs on different splits.

How do we account for the variance in the algorithm itself?

Run CV multiple times and take the average and standard deviation of the accuracies. The average gives us an estimate of the algorithm's *performance*, and the standard deviation helps us understand the *robustness* of the algorithm on the dataset.

For example,

```
# load the library
library(caret)
# load the iris dataset
data(iris)
# define training control
train_control <- trainControl(method="repeatedcv", number=10, repeats=3)
# train the model
model <- train(Species~., data=iris, trControl=train_control, method="nb")
# summarize results
print(model)
```

```
## Naive Bayes
##
## 150 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 135, 135, 135, 135, 135, 135, ...
## Resampling results across tuning parameters:
##
##   usekernel  Accuracy   Kappa
##   FALSE      0.9533333  0.9300000
##    TRUE      0.9577778  0.9366667
##
## Tuning parameter 'fL' was held constant at a value of 0
## Tuning
##  parameter 'adjust' was held constant at a value of 1
## Accuracy was used to select the optimal model using  the largest value.
## The final values used for the model were fL = 0, usekernel = TRUE
##  and adjust = 1.
```

Bootstrap resampling

This approach involves randomly sampling from the dataset (with replacement) against which to evaluate the model.

For example, randomly sample the dataset 10 times

```
# load the library
library(caret)
# load the iris dataset
data(iris)
# define training control
train_control <- trainControl(method="boot", number=100)
# train the model
model <- train(Species~., data=iris, trControl=train_control, method="nb")
```

```
## Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
## observation 16
```

```
# summarize results
print(model)
```

```
## Naive Bayes
##
## 150 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Bootstrapped (100 reps)
## Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
## Resampling results across tuning parameters:
##
##   usekernel  Accuracy   Kappa
##   FALSE      0.9553365  0.9321271
##    TRUE      0.9544445  0.9308040
##
## Tuning parameter 'fL' was held constant at a value of 0
## Tuning
##  parameter 'adjust' was held constant at a value of 1
## Accuracy was used to select the optimal model using  the largest value.
## The final values used for the model were fL = 0, usekernel = FALSE
##  and adjust = 1.
```

How do we compare different algorithms?

Lets say after doing k-fold CV with multiple runs, algorithm A yields one set of mean and standard deviation and algorithm B yields a different set of mean and standard deviation, and A has a higher accuracy than B, how do you know if the difference is meaningful?

Statistical significance

We can use a statistical significance test (like the U test) to compare different algorithms.

If we run k-fold CV with multiple runs using different ML algorithms, then each algorithm produces a list of numbers. We summarize these numbers with a statistic (such as mean and standard deviation). A statistical significance test quantifies how likely these different sets of numbers are drawn from the same population. If this quantity (p-value) is above some threshold, then the difference in the statistics between each algorithm are insignificant.

Error metrics

Alright! Suppose you trained a model using k-fold cross validation with multiple runs, and you believe it can make good predictions because you got an accuracy of 90%. How do you decide whether it is a **good enough** model to **solve your problem**. Classification accuracy alone is typically not enough information to make this decision. We need better metrics that can explain the performance of a model, more importantly, discriminate among different models.

Error metrics for classfication and regression models are different because they produce nominal or binary outputs and continuous outputs respectively.

Classification metrics

Accuracy and Kappa

Accuracy = # correct predictions / total # of predictions

Kappa (Cohen's Kappa) is essentially a measure of how well the classifier performed as compared to how well it would have performed simply by chance. This is a helpful measure for datasets that have imbalanced classes.

For example lets train a GML using 5-fold CV on the PimaIndiansDiabetes dataset, which has a class breakdown of 65% POS and 35% NEG.

```
# load libraries
library(caret)
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# prepare resampling method
control <- trainControl(method="cv", number=5)
set.seed(7)
fit <- train(diabetes~., data=PimaIndiansDiabetes, method="glm", metric="Accuracy", trControl=control)
# display results
print(fit)
```

```
## Generalized Linear Model
##
## 768 samples
##   8 predictor
##   2 classes: 'neg', 'pos'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 615, 615, 614, 614, 614
## Resampling results:
##
##   Accuracy   Kappa
##   0.7774043  0.4851161
```

In this example, the accuracy is only 76%, whereas Kappa is 48%, which is interesting given the baseline is 65%.

Confusion Matrix (not a metric)

A confusion matrix is an N X N matrix, where N is the number of classes being predicted. For a binary classifier, the matrix would be 2 x 2. Although this matrix is not a metric, it helps us organize and see the important results from our predictions.

Definition of key terms:

- True positive (TP): predicted `a` and actual `a`
- True negative (TN): predicted `b` and actual `b`
- False positive (FP): predicted `a` but actual `b`
- False negative (FN): predicted `b` but actual `a`

```
numLlvs <- 2
confusionMatrix(
```

```
  factor(sample(rep(letters[1:numLlvs], 200), 50)),
  factor(sample(rep(letters[1:numLlvs], 200), 50)))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  a  b
##          a 12  9
##          b 13 16
##
##                Accuracy : 0.56
##                  95% CI : (0.4125, 0.7001)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 0.2399
##
##                   Kappa : 0.12
##  Mcnemar's Test P-Value : 0.5224
##
##             Sensitivity : 0.4800
##             Specificity : 0.6400
##          Pos Pred Value : 0.5714
##          Neg Pred Value : 0.5517
##              Prevalence : 0.5000
##          Detection Rate : 0.2400
##    Detection Prevalence : 0.4200
##       Balanced Accuracy : 0.5600
##
##        'Positive' Class : a
##
```

- *POS*: positive class
- *NEG*: negative class
- *Accuracy*: the proportion of the total number of predictions that were correct.
- *Kappa*: normalized accuracy.
- *Sensitivity (aka Recall)*: the proportion of actual positive cases which are correctly identified.
  - `recall = TP / (TP + FN)`
- *Specificity*: the proportion of actual NEG's which are correctly identified.
- *Pos Pred Value (aka Precision)*: the proportion of POS's that were correctly identified.
  - `precision = TP / (TP + FP)`
- *Neg Pred Value*: the proportion of NEG's that were correctly identified.
- *Prevalence*: how often does the POS condition actually occur in our sample?
- *Detection Rate*: How likely is it to detect a true positive?
- *Detection Prevalence*: Overall, how often is it wrong?

Precision/Recall and F Score

F Score is a weighted average of the true positive rate (recall) and precision.

AUC (Area Under ROC Curve)

The AUC is an error metric for binary classifiers, while the ROC is a plot that visualizes the performance of a classifier over all possible thresholds. The true positive rate (y-axis) is plotted against the false positive rate (x-axis) as you vary the threshold for assigning observations to a given class.

The AUC is a measure of the model's ability to discriminate between two classes. An area of 1.0 represents a model predicted every test sample correctly. An area of 0.5 means the model is as a good as a fair coin.

Example:

```
# load libraries
library(caret)
library(plotROC)
# Load the dataset
data("PimaIndiansDiabetes")
control <- trainControl(method="cv",
                        summaryFunction=twoClassSummary,
                        classProbs=T,
                        savePredictions = T)
control <- trainControl(method="cv", number=5,
                        classProbs=TRUE, savePredictions =TRUE,
                        summaryFunction=twoClassSummary)
fit <- train(diabetes~., data=PimaIndiansDiabetes,
             method="rf", preProc=c("center", "scale"),
             metric="ROC", trControl=control)
```

```
## randomForest 4.6-12
```

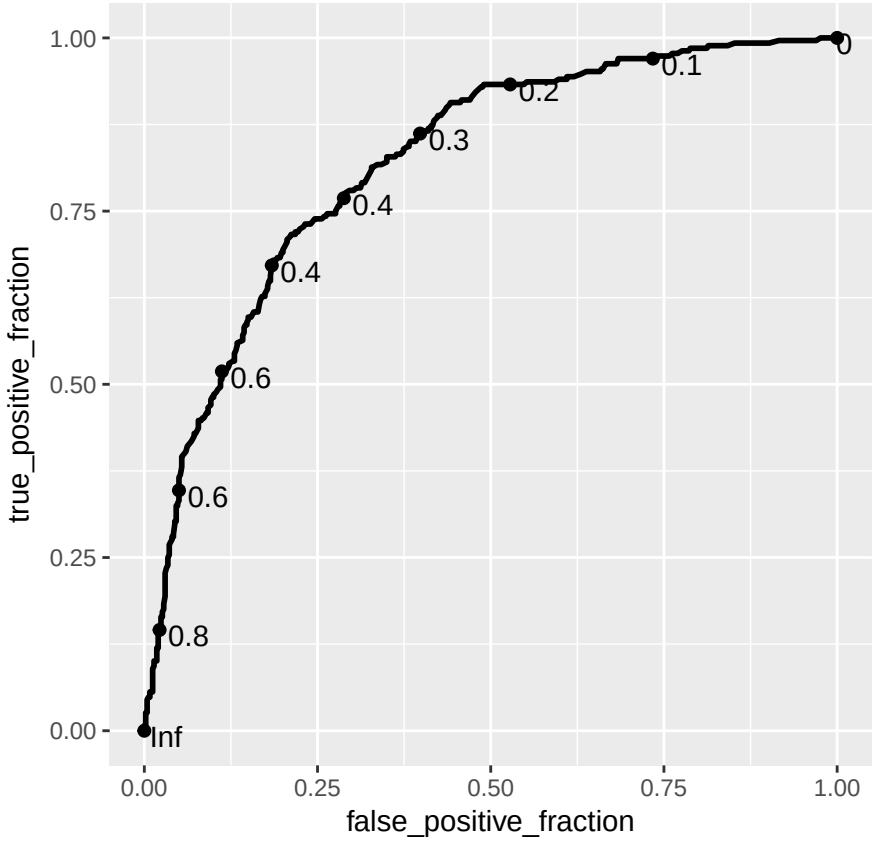
```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
# Select a parameter setting
selectedIndices <- fit$pred$mtry == 2
# Plot:
ggplot(fit$pred[selectedIndices, ],
       aes(m = pos, d = factor(obs, levels = c("neg", "pos")))) +
       geom_roc(hjust = -0.4, vjust = 1.5) + coord_equal()
```

```
## Warning in verify_d(data$d): D not labeled 0/1, assuming neg = 0 and pos =
## 1!
```



Here, the `ROC` should really be the `AUC`.

LogLoss

LogLoss is more prevalent for multiclass classifiers. LogLoss evaluates the probabilities produced by classifiers.

```
# load libraries
library(caret)
# load the dataset
data(iris)
# prepare resampling method
control <- trainControl(method="cv", number=5, classProbs=TRUE, summaryFunction=mnLogLoss)
set.seed(7)
fit <- train(Species~., data=iris, method="rpart", metric="logLoss", trControl=control)
# display results
print(fit)
```

```
## CART
##
## 150 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 120, 120, 120, 120, 120
## Resampling results across tuning parameters:
##
```

```
##   cp    logLoss
##   0.00  0.6124167
##   0.44  0.3904565
##   0.50  1.0986123
##
## logLoss was used to select the optimal model using  the smallest value.
## The final value used for the model was cp = 0.44.
```

Regression metrics

Since the output of regression models is continuous, the go to metrics are different than classifiers. The go to metrics are RMSE and R-squared. Let's go over both!

Root Mean Squared Error (RMSE)

RMSE assumes that the error are unbiased and follow a normal distribution. RMSE is the average deviation from of the predictions from the truths.

$$RMSE = \sqrt{\frac{\sum_{k=1}^{n}(Predicted_i - Actual_i)^2}{N}}$$

Key things to remember:

- The power of 'square root' empowers this metric to show large number deviations.
- The 'squared' nature of this metric helps to deliver more robust results which prevents cancelling the positive and negative error values. In other words, this metric aptly displays the plausible magnitude of error term.
- It avoids the use of absolute error values which is highly undesirable in mathematical calculations. When we have more samples, reconstructing the error distribution using RMSE is considered to be more reliable.
- RMSE is highly affected by outlier values. Hence, make sure you've removed outliers from your data set prior to using this metric.
- As compared to mean absolute error, RMSE gives higher weightage and punishes large errors.

R-squared

R-squared provides a "goodness of fit" measure for the predictions to the observations. This is a value between 0 and 1 for no-fit and perfect fit respectively.

Example: linear regression with 5-fold CV:

```
# load libraries
library(caret)
# load data
data(longley)
# prepare resampling method
control <- trainControl(method="cv", number=5)
set.seed(7)
fit <- train(Employed~., data=longley, method="lm", metric="RMSE", trControl=control)
# display results
print(fit)
```

```
## Linear Regression
##
## 16 samples
```

```
##  6 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 12, 13, 12, 14, 13
## Resampling results:
##
##   RMSE       Rsquared   MAE
##   0.4419474  0.9927128  0.4080994
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Notice that using `RMSE` metric gives us both RMSE and Rsquared results.