Code

Introduction to Data Analysis in R - Machine Learning Workflow

Machine Learning Example

Machine learning has been a very popular way of building predictive models. It's popularity is largely due to the simplicity of the modeling process.

Oral toxicity dataset

We will be using a publicly available data set of oral toxicity of a set of ~10k compounds. Each compound is described by presence or absence of 1024 structural features.

Data download

Hide

```
temp=tempfile()
download.file("https://archive.ics.uci.edu/ml/machine-learning-databases/00508/qsar_oral_toxicity.zip", temp)
```

```
trying URL 'https://archive.ics.uci.edu/ml/machine-learning-databases/00508/qsar_oral_toxicity.zip'
Content type 'application/x-httpd-php' length 974163 bytes (951 KB)
downloaded 951 KB
```

Hide

```
tox = read.csv(unz(temp, "qsar_oral_toxicity.csv"), sep=";")
unlink(temp)
```

Let's take a brief look at the data

Hide

```
list(
  dim=dim(tox),
  outcome=table(tox$negative)
)
```

```
$dim
[1] 8991 1025

$outcome

negative positive
    8250      741
```

	negative <chr>	X0.970 <int>	X0.969 <int>	X0.968 <int>	X0.967 <int>	X0.966 <int>	X0.965 <int>	X0.964 <int>	X0.963 <int>
1	negative	0	0	0	0	0	0	0	0
2	negative	0	0	0	0	0	0	1	0

2 rows | 1-10 of 1025 columns

We confirm that we have 1024 binary features and one target (positive/negative). The dataset is heavily imbalanced, only 8% of compounds are toxic.

ML modeling pipeline

There are literally hundrends of libraries that we can use. A good starting set of tools is in `caret` and I recommend using it. The documentation (https://topepo.github.io/caret/) is excellent, and reading it on its own is like a Machine Learning course.

In fact `caret` is so good, that Python folk are now trying to replicate it with py-caret.

Hide

```
require(caret)
```

```
Loading required package: caret
Warning: package ‘caret’ was built under R version 4.1.2
Loading required package: ggplot2
Loading required package: lattice
Registered S3 method overwritten by 'data.table':
  method           from
  print.data.table
```

Hide

```
# parallel registration
require(doParallel)
```

```
Loading required package: doParallel
Warning: package ‘doParallel’ was built under R version 4.1.2
Loading required package: foreach
Loading required package: iterators
Loading required package: parallel
```

Hide

```
cl = makePSOCKcluster(detectCores() - 1)
registerDoParallel(cl)
```

Let's check for missing values and deduplicate the data. It's often the first step of any analysis.

Hide

```
dupes = duplicated(tox)
sum(dupes)
```

```
[1] 477
```

Splitting data

The data is always split into train/validation/test sets. Let's first split into train/test. We will then split the train when we discuss why we need the validation set.

Hide

```
head(tIdx, 10)
```

```
      Resample1
 [1,]         2
 [2,]         3
 [3,]         4
 [4,]         5
 [5,]         6
 [6,]         7
 [7,]         9
 [8,]        10
 [9,]        11
[10,]        12
```

Let's take a look at the positive/negative rates in our training/testing samples (we'd like them to be the same).

Hide

```
table(tox[tIdx, "negative"])
```

```
negative positive
    5847      540
```

Hide

```
table(tox[-tIdx, "negative"])
```

```
negative positive
    1948      179
```

We now ***completely forget*** about the testing data. As if it *didn't exist*. We will get back to it after all models are trained and cross-checked.

Hide

```
train.data = tox[tIdx,]
train.X = train.data[, 1:(ncol(train.data) - 1)]
train.y = train.data["negative"]


test.data = tox[-tIdx,]
test.X = test.data[, 1:(ncol(test.data) - 1)]
test.y = test.data["negative"]
```

Pre-processing

Pre-processing aims to reduce the complexity of the features so that the ML models train easier. Most often ***domain knowledge*** allows one to perform at least some pre-processing.

Hide

```
# Let's see if some of the input sampels (both X and y) are duplicates
any(duplicated(train.data))
```

```
[1] FALSE
```

Question:

Why is pre-processing done after the data was split into train/test?

Cross-validation.

Machine learning models require some level of control while training to limit the effect of overfitting. Overfitting arises because in ML one typically uses many many features and a flexible model formulation, so without *control* the models would simply replicate the training data, while not learn anything. The degrees of freedom of the fit are often larger than the data set itself so *control* is a must.

Hide

```
samples = createDataPartition(train.data$negative, p=0.75, times=5, list=TRUE)

ctrl = trainControl(method='LGOCV',
                    p=0.75,
                    number=5,
                    index=samples,
                    sampling='down', # this is critical when handling class imbalance problems
                    classProbs=TRUE,
                    summaryFunction=twoClassSummary,
                    savePredictions=TRUE
                    )
```

There is a lot happening in this *control*. So let's talk over it.

A brief note on the evaluation metrics

$$Specificity = 1 - \frac{FalsePositives}{Positives}$$
$$Sensitivity = 1 - \frac{FalseNegatives}{Negatives}$$

It's always possible to make Specificity or Sensitivity 1, but not both. ROC curve (and it's area under curve [AUC]) determines how well we can balance good sensitivity with good specificity.

Model training with overfitting control

We will train a handful of models. The control is used to choose an appropriate set of hyper-parameters. For each modeling method the list of hyper-parameters can be checked with `caret::modelLookup(model_name)`.

First `random forest`.

Hide

```
require('randomForest')
modelLookup('rf')
```

mo…	parameter	label	forReg	forClass	probModel
<chr>	<chr>	<chr>	<lgl>	<lgl>	<lgl>
1 rf	mtry	#Randomly Selected Predictors	TRUE	TRUE	TRUE

1 row

Hide

```
rf.model = train(negative ~ ., data=train.data, method='rf', trControl=ctrl,
                 metric='ROC', ntree=10, tuneGrid=expand.grid(mtry=c(2,5,10,15,20)))
rf.model
```

```
Random Forest

6387 samples
1024 predictors
   2 classes: 'negative', 'positive'

No pre-processing
Resampling: Repeated Train/Test Splits Estimated (5 reps, 75%)
Summary of sample sizes: 4791, 4791, 4791, 4791, 4791
Addtional sampling using down-sampling

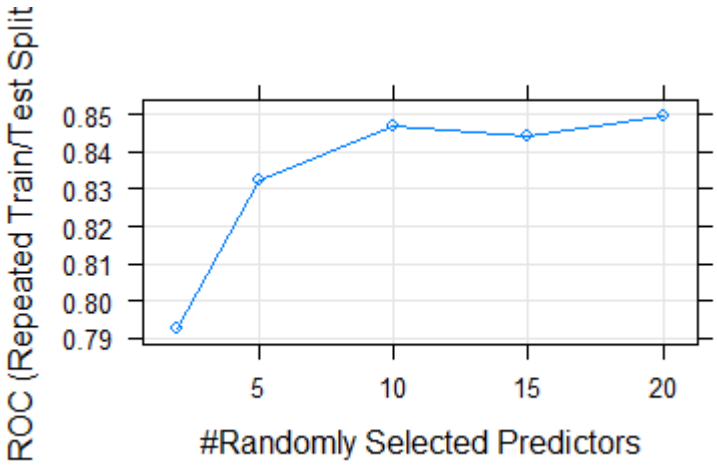
Resampling results across tuning parameters:

  mtry  ROC        Sens       Spec
   2    0.7924907  0.7809719  0.6548148
   5    0.8321119  0.7987680  0.7081481
  10    0.8469820  0.8056126  0.7496296
  15    0.8438832  0.7984942  0.7155556
  20    0.8496981  0.7965777  0.7422222

ROC was used to select the optimal model using the largest value.
The final value used for the model was mtry = 20.
```

Hide

```
plot(rf.model)
```



Then `support vector machines`. All it takes in `caret` is to change the `method` and adapt the hyperparameters.

model	parameter	label	forReg	forClass	probModel
<chr>	<chr>	<chr>	<lgl>	<lgl>	<lgl>
1 svmLinear	C	Cost	TRUE	TRUE	TRUE

1 row

Now the simplest linear model `glmnet` which stands for Elastic-Net Regression.

Hide

```
require('glmnet')
```

```
Loading required package: glmnet
Warning: package ‘glmnet’ was built under R version 4.1.2
Loading required package: Matrix
Loaded glmnet 4.1-3
```

Hide

```
modelLookup('glmnet')
```

model <chr>	parameter <chr>	label <chr>	forReg <lgl>	forClass <lgl>	probModel <lgl>
1 glmnet	alpha	Mixing Percentage	TRUE	TRUE	TRUE
2 glmnet	lambda	Regularization Parameter	TRUE	TRUE	TRUE

2 rows

Hide

```
glmnet.model = train(negative ~ ., data=train.data, method='glmnet', trControl=ctrl,
                metric='ROC', tuneGrid=expand.grid(alpha=1,
                                                   lambda=c(0.00001, 0.0001, 0.001,
                                                            0.005, 0.01, 0.03, 0.05)))
glmnet.model
```

```
glmnet

6387 samples
1024 predictors
   2 classes: 'negative', 'positive'

No pre-processing
Resampling: Repeated Train/Test Splits Estimated (5 reps, 75%)
Summary of sample sizes: 4791, 4791, 4791, 4791, 4791
Addtional sampling using down-sampling

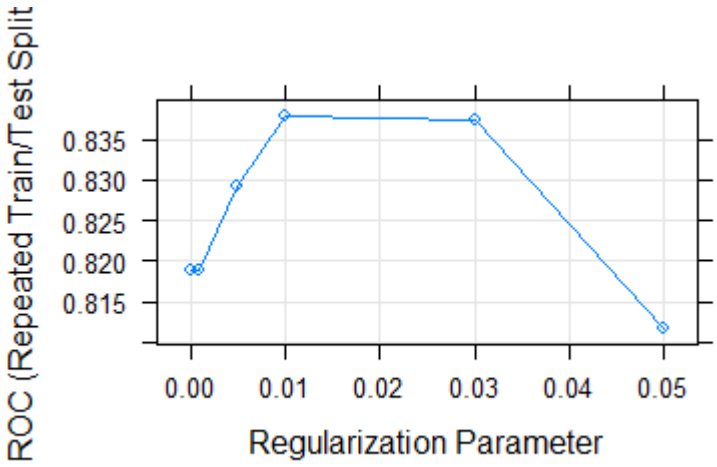
Resampling results across tuning parameters:

  lambda  ROC        Sens       Spec
  1e-05   0.8189044  0.7333333  0.7955556
  1e-04   0.8189044  0.7333333  0.7955556
  1e-03   0.8189044  0.7333333  0.7955556
  5e-03   0.8291211  0.7504449  0.7940741
  1e-02   0.8379654  0.7786448  0.7807407
  3e-02   0.8373803  0.8269678  0.7155556
  5e-02   0.8115781  0.8261465  0.6562963

Tuning parameter 'alpha' was held constant at a value of 1
ROC was used to select the optimal model using the largest value.
The final values used for the model were alpha = 1 and lambda = 0.01.
```

Hide

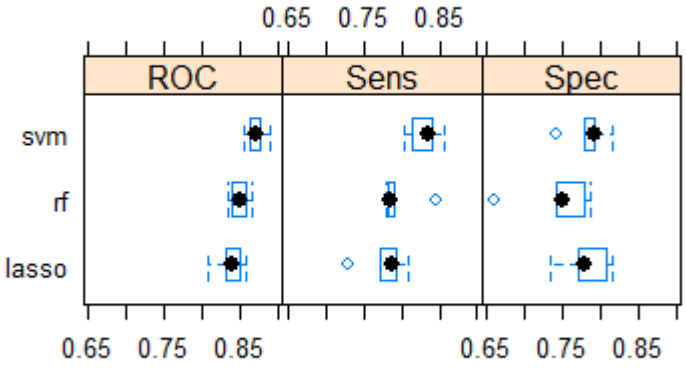
```
plot(glmnet.model)
```



`Caret` provides plenty of libraries to compare models based on their performance in cross-validation.

Hide

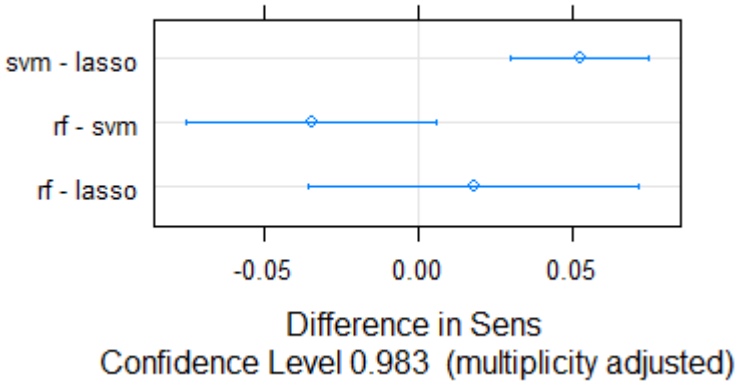
```
# first we combine the models into a list
models=list(rf=rf.model, svm=svm.model, lasso=glmnet.model)
# extract the prediction information from the cross-validation resampling
resamps = resamples(models)
# plot
bwplot(resamps)
```



Overall SVM tends to narrowly win.

This is very typical to machine learning problems: many algorithms will perform similarly. What limits the performance is the relevant information stored in the features, and not the algorithm that's used to build the model.

Question: how do we choose the final model (let's say for publication?)



Final evaluation of the model

This is always done on a test set that hasn't been used in any modeling decision, to remain unbiased.

	negative <dbl>	positive <dbl>	obs <fctr>	pred <fctr>	model <chr>	dataType <chr>	object <chr>
2	0.7	0.3	negative	negative	rf	Training	rf
3	0.8	0.2	negative	negative	rf	Training	rf

	negative <dbl>	positive <dbl>	obs <fctr>	pred <fctr>	model <chr>	dataType <chr>	object <chr>
4	1.0	0.0	negative	negative	rf	Training	rf
5	0.9	0.1	negative	negative	rf	Training	rf
6	0.8	0.2	negative	negative	rf	Training	rf
7	0.3	0.7	positive	positive	rf	Training	rf
9	0.6	0.4	negative	negative	rf	Training	rf
10	0.6	0.4	negative	negative	rf	Training	rf
11	0.3	0.7	negative	positive	rf	Training	rf
12	0.3	0.7	negative	positive	rf	Training	rf

1-10 of 15 rows Previous **1** 2 Next

Let's plot some ROC curves for our models.

Hide

```
require(pROC)
```

```
Loading required package: pROC
Warning: package 'pROC' was built under R version 4.1.2
Type 'citation("pROC")' for a citation.

Attaching package: 'pROC'

The following objects are masked from 'package:stats':

    cov, smooth, var
```

Hide

```
probs.test.svm = probs[probs$dataType=='Test' & probs$model=='svmRadialCost', ]
probs.train.svm = probs[probs$dataType=='Training' & probs$model=='svmRadialCost', ]

roc.svm.test = pROC::roc(probs.test.svm$obs, probs.test.svm$negative)
```

```
Setting levels: control = negative, case = positive
Setting direction: controls > cases
```

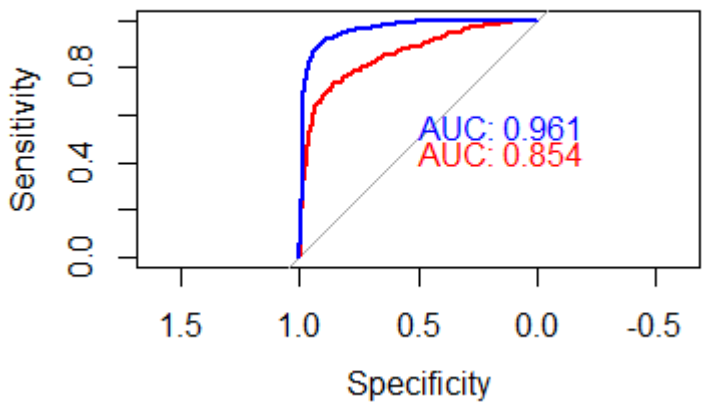
Hide

```
roc.svm.train = pROC::roc(probs.train.svm$obs, probs.train.svm$negative)
```

```
Setting levels: control = negative, case = positive
Setting direction: controls > cases
```

Hide

```
plot.roc(roc.svm.test, print.auc=TRUE, col='red')
plot.roc(roc.svm.train, print.auc=TRUE, add=TRUE, col='blue', print.auc.adj=c(0,0))
```



Hide

```
probs.test.svm = probs[probs$dataType=='Test' & probs$model=='svmRadialCost', ]
probs.test.rf = probs[probs$dataType=='Test' & probs$model=='rf', ]
probs.test.lasso = probs[probs$dataType=='Test' & probs$model=='glmnet', ]

roc.svm.test = pROC::roc(probs.test.svm$obs, probs.test.svm$negative)
```

```
Setting levels: control = negative, case = positive
Setting direction: controls > cases
```

Hide

```
roc.rf.test = pROC::roc(probs.test.rf$obs, probs.test.rf$negative)
```

```
Setting levels: control = negative, case = positive
Setting direction: controls > cases
```

Hide

```
roc.lasso.test = pROC::roc(probs.test.lasso$obs, probs.test.lasso$negative)
```

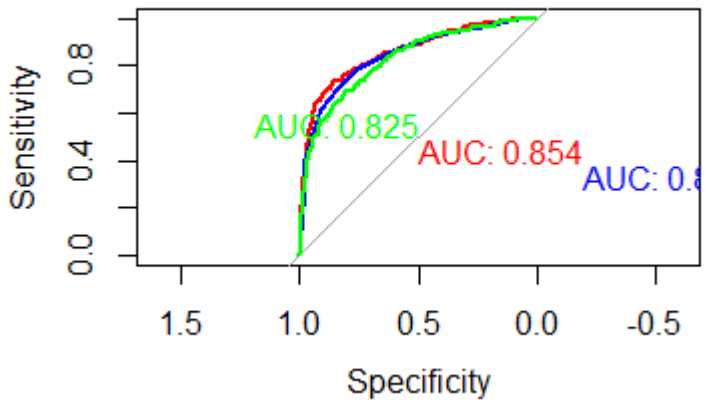
```
Setting levels: control = negative, case = positive
Setting direction: controls > cases
```

Hide

```
plot.roc(roc.svm.test, print.auc=TRUE, col='red')
plot.roc(roc.rf.test, print.auc=TRUE, add=TRUE, col='blue', print.auc.adj=c(-1,2))
```

Hide

```
plot.roc(roc.lasso.test, print.auc=TRUE, add=TRUE, col='green', print.auc.adj=c(1,0))
```



Finally, let's print the confusion matrix and other metric of our top models on the test set.

Hide

```
confusionMatrix(probs.test.svm$pred, probs.test.svm$obs)
```

```
Confusion Matrix and Statistics

          Reference
Prediction negative positive
  negative     1647       48
  positive      301      131

               Accuracy : 0.8359
                 95% CI : (0.8195, 0.8514)
    No Information Rate : 0.9158
    P-Value [Acc > NIR] : 1

                  Kappa : 0.3516

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.8455
            Specificity : 0.7318
         Pos Pred Value : 0.9717
         Neg Pred Value : 0.3032
             Prevalence : 0.9158
         Detection Rate : 0.7743
   Detection Prevalence : 0.7969
      Balanced Accuracy : 0.7887

       'Positive' Class : negative
```

Hide

```
```r
confusionMatrix(probs.test.rf$pred, probs.test.rf$obs)
```


```
<!-- rnb-source-end -->

<!-- rnb-output-begin eyJkYXRhIjoiQ29uZnVzaW9uIE1hdHJpeCBhbmQgU3RhdGlzdGljc1xuXG4gICAgICAgICAgUm
VmZXJlbmNlXG5QcmVkaWN0aW9uIG5lZ2F0aXZlIHBvc2l0aXZlXG4gIG5lZ2F0aXZlICAgICAxNjY1ICAgICAgIDU1XG4gIH
Bvc2l0aXZlICAgICAgMjgzICAgICAgMTI0XG4gICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICBcbi
AgICAgICAgICAgICAgIEFjY3VyYWN5IDogMC44NDExICAgICAgICAgIFxuICAgICAgICAgICAgICAgICA5NSUgQ0kgOiAoMC
44MjQ5LCAwLjg1NjQpXG4gICAgTm8gSW5mb3JtYXRpb24gUmF0ZSA6IDAuOTE1OCAgICAgICAgICBcbiAgICBQLVZhbHVlIF
tBY2MgPiBOSVJdIDogMSAgICAgICAgICAgICAgIFxuICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIC
AgXG4gICAgICAgICAgICAgICAgICBLYXBwYSA6IDAuMzQ2OSAgICAgICAgICBcbiAgICAgICAgICAgICAgICAgICAgICAgIC
AgICAgICAgICAgICAgIFxuIE1jbmVtYXIncyBUZXN0IFAtVmFsdWUgOiA8MmUtMTYgICAgICAgICAgXG4gICAgICAgIC
AgICAgICAgICAgICAgICAgICAgICAgICAgICBcbiAgICAgICAgICAgIFNlbnNpdGl2aXR5IDogMC44NTQ3ICAgIC
AgICAgIFxuICAgICAgICAgICAgU3BlY2lmaWNpdHkgOiAwLjY5MjcgICAgICAgICAgXG4gICAgICAgICBQb3MgUHJlZCBWYW
x1ZSA6IDAuOTY4MCAgICAgICAgICBcbiAgICAgICAgIE5lZyBQcmVkIFZhbHVlIDogMC4zMDQ3ICAgICAgICAgIFxuICAgIC
AgICAgICAgIFByZXZhbGVuY2UgOiAwLjkxNTggICAgICAgICAgXG4gICAgICAgICBEZXRlY3Rpb24gUmF0ZSA6IDAuNzgyOC
AgICAgICBcbiAgIERldGVjdGlvbiBQcmV2YWxlbmNlIDogMC44MDg3ICAgICAgICAgIFxuICAgICAgQmFsYW5jZWQgQW
NjdXJhY3kgOiAwLjc3MzcgICAgICAgICAgXG4gICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICBcbi
AgICAgICAnUG9zaXRpdmUnIENsYXNzIDogbmVnYXRpdmUgICAgICAgICAgIFxuICAgICAgICAgICAgICAgICAgICAgICAgIC
AgICAgICAgICAgICAgXG4ifQ== -->
```


Confusion Matrix and Statistics

```
      Reference
```

Prediction negative positive negative 1665 55 positive 283 124

```
          Accuracy : 0.8411
            95% CI : (0.8249, 0.8564)
No Information Rate : 0.9158
P-Value [Acc > NIR] : 1

             Kappa : 0.3469
```

Mcnemar’s Test P-Value : <2e-16

```
       Sensitivity : 0.8547
       Specificity : 0.6927
    Pos Pred Value : 0.9680
    Neg Pred Value : 0.3047
        Prevalence : 0.9158
    Detection Rate : 0.7828
```

Detection Prevalence : 0.8087
Balanced Accuracy : 0.7737
```

```
   'Positive' Class : negative
```

```
```