Modelling

Predictive Data Analysis in R

Sport Prediction

Predicting *who will win* is the holy grail of sports analytics and is a major research area of interest for sports statisticians.

Descriptive vs Predictive Models

Descriptive vs Predictive Models

- *Predictive modeling* usually refers to machine learning

Descriptive vs Predictive Models

- *Predictive modeling* usually refers to machine learning
- The goal of these techniques is to improve predictive performance

Descriptive vs Predictive Models

- *Predictive modeling* usually refers to machine learning
- The goal of these techniques is to improve predictive performance
- This is a very different goal from statistical models, like regression, where inference and interpretation are of primary importance

Descriptive vs Predictive Models

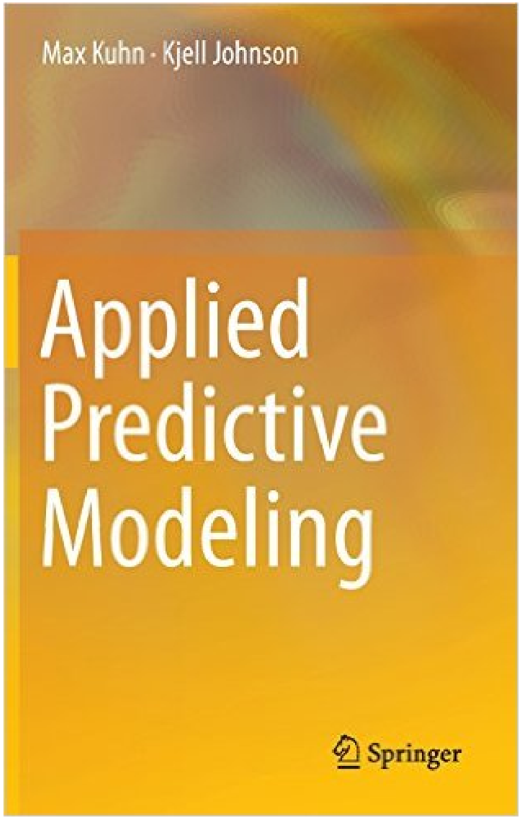
- *Predictive modeling* usually refers to machine learning
- The goal of these techniques is to improve predictive performance
- This is a very different goal from statistical models, like regression, where inference and interpretation are of primary importance
- Predictive models will often sacrifice interpretability for improved performance

Describe Before You Predict

- Because it is often a challenge to interpret the "how" of machine learning methods, it is good practice to first do some statistical modelling (e.g. `glm(y ~ x + ...)`)
- The reasons for this include:
  - Getting familiar with the interrelationships in your data
  - Identifying any issues not flagged during EDA
  - Developing some expectations for the predictive modelling results

Predictive Modelling with `caret`

- Once you are ready to develop your prediction model, the `caret` package, written by Max Kuhn, is a great resource for machine learning in `R`
- `caret` stand for Classification And REgression Training
- It includes 100+ predictive modelling methods
- It provides a unified & streamlines interface for building and evaluating models
- It allows parallel processing for faster computation
- You can install with `install.packages('caret')`



Modelling Steps

Modelling Steps

Step 1. Decide how to spend your data

Modelling Steps

Step 1. Decide how to spend your data

Step 2. Split the data into training and test

Modelling Steps

Step 1. Decide how to spend your data

Step 2. Split the data into training and test

Step 3. Conduct pre-processing (as needed)

Modelling Steps

Step 1. Decide how to spend your data

Step 2. Split the data into training and test

Step 3. Conduct pre-processing (as needed)

Step 4. Train the model with resampling

Modelling Steps

Step 1. Decide how to spend your data

Step 2. Split the data into training and test

Step 3. Conduct pre-processing (as needed)

Step 4. Train the model with resampling

Step 5. Evaluate the model with test data

Why Split Up the Data at All?

- Testing our data on independent samples is the strongest form of validation and evaluation
- If we trained and tested on the same data, we risk *overfitting* which is the machine-learning equivalent to a "Monday morning quarterback"
- We also resample among training for additional protection against overfitting

Using `createDataPartition`

We can use `createDataPartition` to split our data:

```
createDataPartition(y, times, p, list, ...)
```

Argument	Description
y	Outcome vector to balance sampling on
times	Number of partitions
p	Proportion of each partition allocated to training
list	Logical whether list is returned

Note: Using 70% of our data for training is typical

Example: Partitioning Data

In this example, we create one partition with 70% of our dataset allocated to training.

```r
library(caret) # Load caret

# Returns matrix of indices for obs in training
train <- createDataPartition(
  y = data$outcome,
  times = 1,
  p = 0.7,
  list = F
)
```

Practice: Using CreateDataPartition

Modify the previous code to obtain 5 partitions for your training samples with 60% of observeations in each devoted to training.

Practice: Using CreateDataPartition

Modify the previous code to obtain 5 partitions for your training samples with 60% of observeations in each devoted to training.

Answer:

```
train <- createDataPartition(
  y = data$outcome,
  times = 5,
  p = 0.6,
  list = F
)
```

Pre-Processing

Before we split our data, we need to pre-process our data. The pre-processing can protect against some loss in model accuracy due to scale, skew, or high correlation.

Common pre-processing steps are:

1. Centering - Give all variables a common mean of 0
2. Standardizing - Give all variables a common scale
3. Remove highly correlated variables
4. Reduce dimension (when $n \sim p$)

The `train` Function

The main workhouse function for model training in `caret` is `train`. Here are the main arguments you need to know to get started.

Argument	Description
form	Formula (y ~ x)
data	Data frame of training data
method	Character of the ML method to be used
metric	Performance metric for summarizing
tuneLength	Sets granularity for tuning parameter if tuneGrid not specified
trControl	Control parameters for training
tuneGrid	Data frame that gives explicit range for tuning parameters

Practice: Pre-Processing

What function would you use to standardize your model features?

Practice: Pre-Processing

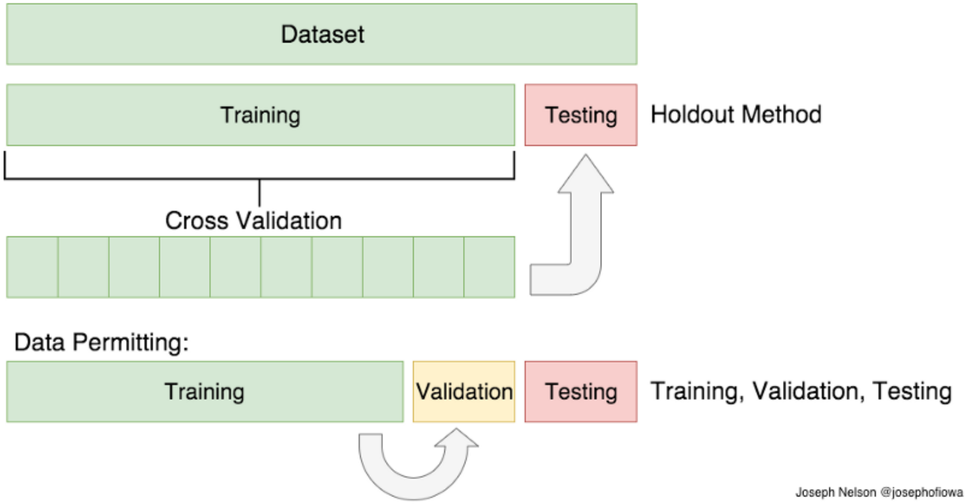
What function would you use to standardize your model features?

Answer: The `scale` function

Training Control

- The `trControl` argument is a list that controls a number of aspects of training including resampling and how we summarise performance with each resample.

- The resampling is an important additional measure to protect against overfitting when training



Example: Using `trControl`

In our example, we will use `trControl` to use 5-fold cross validation and a two-class summary for our performance measures.

```
ctrlSpecs <- trainControl(
    method = "repeatedCV",
    repeats = 5,
    summaryFunction = twoClassSummary, # function from caret
    classProbs = TRUE # Needed to use twoClassSummary
)
```

Grid Tuning

- The `tuneGrid` is a way to give a specific grid for the tuning parameters of the method
- You can use `expand.grid` to make a range of parameters
- To determine the parameters to tune and their variable names you can use `getModelInfo`

Using `getModelInfo`

- We can get a model of interest with `getModelInfo('model')` or get info from all models with `getModelInfo()`.
- This returns a list per model with information about the model type, parameters, grid function, etc.
- Here is an example with the `rpart` model.

```
getModelInfo('rpart')[['rpart']][1:4]
```

```
## $label
## [1] "CART"
##
## $library
## [1] "rpart"
##
## $type
## [1] "Regression"     "Classification"
##
## $parameters
##   parameter   class                label
## 1        cp numeric Complexity Parameter
```

Performance Metrics

- We can use the `metric` argument to choose our performance metric for training evaluation
- There are many metrics for evaluating classification. In general, it is best to choose one when choosing among model approaches

Metric	Description
Accuracy	Proportion of exactly correct classifications
AUC	Area under the ROC curve
Sensitivity	The true positive rate (also called 'recall')
Specificity	The true negative rate
LogLoss	Prediction-weighted loss function

Setting Performance Metric

There is no one correct performance measure. In fact, multiple should be evaluated when testing. For training, the "log loss" is good all around measure. Here is how we can set our control specs to use it.

```
ctrlSpecs <- trainControl(
    method = "repeatedCV",
    repeats = 5,
    summaryFunction = mnLogLoss,
    classProbs = TRUE
)
```

Practice: Alternative Resampling

Look at the `method` options using the documentation on `trainControl`. Find an alternative resampling approach and modify the `ctrlSpecs` object to use this method.

Practice: Alternative Resampling

Look at the `method` options using the documentation on `trainControl`. Find an alternative resampling approach and modify the `ctrlSpecs` object to use this method.

In this alternative we use 10 bootstrap resamples:

```
ctrlSpecs <- trainControl(
    method = "boot",
    repeats = 10,
    summaryFunction = mnLogLoss,
    classProbs = TRUE
)
```

Models

There are more than 100 models to try in `caret`! Below is just a sample of some popular kinds. The full list is available with the online caret book.

Category	Description	Examples
Forest	Ensemble of multiple decision trees with bagging (boostrap aggregation)	rf, rfRules, cforest
Boosted	Incremental building of multiple classifiers, which is a kind of correlated ensembling	gbm, adaboost, C5.0
Support Vector Machines	Collection of regression lines that try to maximally separate classes	svmLinear, svmRadial

Random Forest

Let's have a look at each category and how we could train each in `caret`. Below we use the `rf` method to fit a random forest. The `tuneLength` is set to 10 to have a randomly generated grid for the forest parameters.

```
rfFit <- train(
    outcome ~ .,
    data = trainingData,
    method = "rf",
    tuneLength = 10,
    trControl = ctrlSpecs,
    metric = "logLoss"
)
```

Practice: Grid Tuning

Use `getModelInfo` to find the tuning parameters for the `rf` method. Create a data frame with 10 different options and modify the `train` function to use these in place of the random tuning with `tuneLength`.

Practice: Grid Tuning

Use `getModelInfo` to find the tuning parameters for the `rf` method. Create a data frame with 10 different options and modify the `train` function to use these in place of the random tuning with `tuneLength`.

```
# Find tuning parameters
getModelInfo("rf")[["rf"]]$parameters
```

```
##   parameter   class                         label
## 1      mtry numeric #Randomly Selected Predictors
```

```
grid <- data.frame(.mtry = seq(5, 25, length = 10))

# Modify train function
rfFit <- train(
    outcome ~ .,
    data = trainingData,
    method = "rf",
    tuneGrid = grid,
    trControl = ctrlSpecs,
    metric = "logLoss"
)
```

Evaluate the Model

Evaluate the Model

- We can see preformance metrics across the different tuning parameters using: `print` or `plot`

Evaluate the Model

- We can see preformance metrics across the different tuning parameters using: `print` or `plot`
- Also, the selected model can be extracted with `finalModel` and it will have all the class properties of the source method

Evaluate the Model

- We can see preformance metrics across the different tuning parameters using: `print` or `plot`
- Also, the selected model can be extracted with `finalModel` and it will have all the class properties of the source method

For example, with a random forest, we can look at feature importance:

```
library(randomForest) # Class methods for RF

importance(rfFit$finalModel) # Variable importance
```

Resources

- Web book on caret
- Applied Predictive Modeling
- Introduction to Statistical Learning