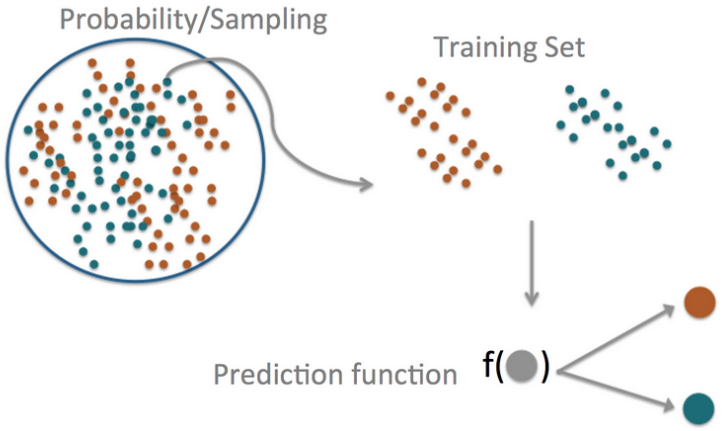
Practical Machine Learning Course Notes

Xing Su

Contents

Prediction

- **process for prediction** = population → probability and sampling to pick set of data → split into training and test set → build prediction function → predict for new data → evaluate
  - ***Note**: choosing the right dataset and knowing what the specific question is are paramount to the success of the prediction algorithm (GoogleFlu failed to predict accurately when people's search habits changed)*



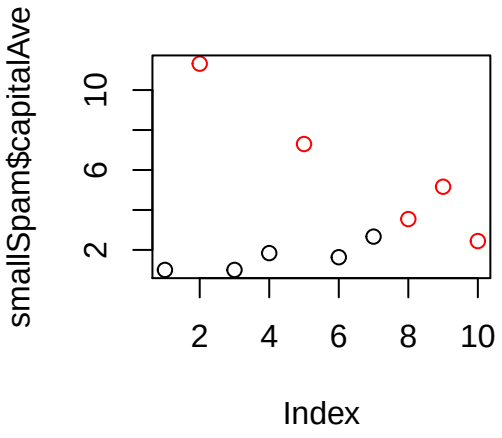
- **components of predictor** = question → input data → features *(extracting variables/characteristics)* → algorithm → parameters *(estimate)* → evaluation
- **relative order of importance** = question (concrete/specific) > data (relevant) > features (properly extract) > algorithms
- **data selection**
  - ***Note**: "garbage in = garbage out" → having the correct/relevant data will decide whether the model is successful*
  - data for what you are trying to predict is most helpful
  - more data → better models (usually)
- **feature selection**
  - good features → lead to data compression, retain relevant information, created based on expert domain knowledge
  - common mistakes → automated feature selection (can yield good results but likely to behave inconsistently with slightly different data), not understanding/dealing with skewed data/outliers, throwing away information unnecessarily
- **algorithm selection**
  - matter less than one would expect
  - getting a sensible approach/algorithm will be the basis for a successful prediction
  - more complex algorithms can yield incremental improvements
  - ideally *interpretable* (simple to explain), accurate, scalable/fast (may leverage parallel computation)
- prediction is effectively about ***trade-offs***
  - find the correct balance between interpretability vs accuracy vs speed vs simplicity vs scalability

  - *interpretability* is especially important in conveying how features are used to predict outcome
  - scalability is important because for an algorithm to be of practical use, it needs to be implementable on large datasets without incurring large costs (computational complexity/time)

In Sample vs Out of Sample Errors

- **in sample error** = error resulted from applying your prediction algorithm to the dataset you built it with
  - also known as *resubstitution error*
  - often optimistic (less than on a new sample) as the model may be tuned to error of the sample
- **out of sample error** = error resulted from applying your prediction algorithm to a new data set
  - also known as *generalization error*
  - out of sample error most important as it better evaluates how the model should perform
- in sample error $<$ out of sample error
  - reason is ***over-fitting***: model too adapted/optimized for the initial dataset
    - data have two parts: *signal* vs *noise*
    - goal of predictor (should be simple/robust) = find signal
    - it is possible to design an accurate in-sample predictor, but it captures both signal and noise
    - predictor won't perform as well on new sample
  - often times it is better to give up a little accuracy for more robustness when predicting on new data
- ***example***

```
# load data
p_load("kernlab")
data(spam); set.seed(333)
# picking a small subset (10 values) from spam data set
smallSpam <- spam[sample(dim(spam)[1],size=10),]
# label spam = 2 and ham = 1
spamLabel <- (smallSpam$type=="spam")*1 + 1
# plot the capitalAve values for the dataset with colors differentiated by spam/ham (2 vs 1)
plot(smallSpam$capitalAve,col=spamLabel)
```



```r
# first rule (over-fitting to capture all variation)
rule1 <- function(x){
    prediction <- rep(NA,length(x))
    prediction[x > 2.7] <- "spam"
    prediction[x < 2.40] <- "nonspam"
    prediction[(x >= 2.40 & x <= 2.45)] <- "spam"
    prediction[(x > 2.45 & x <= 2.70)] <- "nonspam"
    return(prediction)
}
# tabulate results of prediction algorithm 1 (in sample error -> no error in this case)
table(rule1(smallSpam$capitalAve),smallSpam$type)
```

```
##
##           nonspam spam
##   nonspam       5    0
##   spam          0    5
```

```r
# second rule (simple, setting a threshold)
rule2 <- function(x){
    prediction <- rep(NA,length(x))
    prediction[x > 2.8] <- "spam"
    prediction[x <= 2.8] <- "nonspam"
    return(prediction)
}
# tabulate results of prediction algorithm 2(in sample error -> 10% in this case)
table(rule2(smallSpam$capitalAve),smallSpam$type)
```

```
##
##           nonspam spam
##   nonspam       5    1
##   spam          0    4
```

```r
# tabulate out of sample error for algorithm 1
table(rule1(spam$capitalAve),spam$type)
```

```
##
##           nonspam spam
##   nonspam    2141  588
##   spam        647 1225
```

```r
# tabulate out of sample error for algorithm 2
table(rule2(spam$capitalAve),spam$type)
```

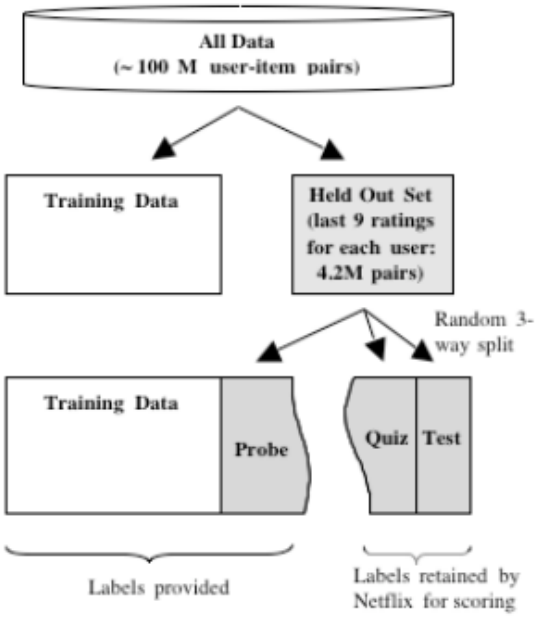
```
##
##           nonspam spam
##   nonspam    2224  642
##   spam        564 1171
```

```r
# accuracy and total correct for algorithm 1 and 2
rbind("Rule 1" = c(Accuracy = mean(rule1(spam$capitalAve)==spam$type),
    "Total Correct" = sum(rule1(spam$capitalAve)==spam$type)),
    "Rule 2" = c(Accuracy = mean(rule2(spam$capitalAve)==spam$type),
    "Total Correct" = sum(rule2(spam$capitalAve)==spam$type)))
```

```
##         Accuracy Total Correct
## Rule 1 0.7315801          3366
## Rule 2 0.7378831          3395
```

Prediction Study Design

- **procedures**
  1. define error rate (type I/type II)
  2. split data into:
     - training, testing, validation (optional)
  3. pick features from the training set
     - use cross-validation
  4. pick prediction function (model) on the training set
     - use cross-validation
  5. if no validation set
     - apply **1 time** to test set
  6. if there is a validation set
     - apply to test set and refine
     - apply **1 time** to validation
  - ***Note**: it's important to hold out an untouched sample to accurately estimate the out of sample error rate*
- benchmarks (i.e. set all variables = 0) can help pinpoint/test the model to see what is wrong with the model
- avoid small sample sizes
  - consider binary outcomes (i.e. coin flip)
  - for n = 1, the probability of perfect classification (100% accuracy) is 50%
  - for n = 10, the probability of perfect classification (100% accuracy) is 0.1%
  - so it's important to have bigger samples so that when you do get a high accuracy, it may actually be a significant result and not just by chance
- ***example: Netflix rating prediction competition***
  - split data between training and "held-out"
    - held-out included probe, quiz and test sets
    - probe is used to test the predictor built from the training dataset
    - quiz is used to realistically evaluate out of sample error rates
    - test is used to finally evaluate the validity of algorithm
  - important to not tune model to quiz set specifically



Sample Division Guidelines for Prediction Study Design

- for large sample sizes
  - 60% training
  - 20% test
  - 20% validation
- for medium sample sizes
  - 60% training
  - 40% test
  - no validation set to refine model (to ensure test set is of sufficient size)
- for small sample sizes
  - carefully consider if there are enough sample to build a prediction algorithm
  - no test/validation sets
  - perform cross validation
  - report caveat of small sample size and highlight the fact that the prediction algorithm has never been tested for out of sample error
- there should always be a test/validation set that is held away and should ***NOT*** be looked at when building model
  - when complete, apply the model to the held-out set only one time
- ***randomly sample*** training and test sets
  - for data collected over time, build training set in chunks of times
- datasets must reflect structure of problem
  - if prediction evolves with time, split train/test sets in time chunks (known as *backtesting* in finance)
- subsets of data should reflect as much diversity as possible

Picking the Right Data

- use like data to predict like
- to predict a variable/process X, use the data that's as closely related to X as possible
- weighting the data/variables by understanding and intuition can help to improve accuracy of prediction
- data properties matter → knowing how the data connects to what you are trying to measure
- predicting on unrelated data is the most common mistake
  - if unrelated data must be used, be careful about interpreting the model as to why it works/doesn't work

Types of Errors

- when discussing the outcome decided on by the algorithm, **Positive** = identified and **negative** = rejected
  - **True positive** = correctly identified (predicted true when true)
  - **False positive** = incorrectly identified (predicted true when false)
  - **True negative** = correctly rejected (predicted false when false)
  - **False negative** = incorrectly rejected (predicted false when true)
- ***example: medical testing***
  - *True positive* = Sick people correctly diagnosed as sick
  - *False positive* = Healthy people incorrectly identified as sick
  - *True negative* = Healthy people correctly identified as healthy
  - *False negative* = Sick people incorrectly identified as healthy

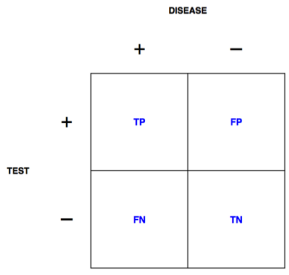
Notable Measurements for Error – Binary Variables

- **accuracy** = weights false positives/negatives equally
- **concordance** = for multi-class cases,

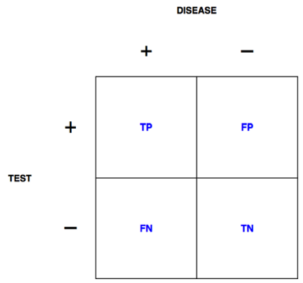
$$\kappa = \frac{accuracy - P(e)}{1 - P(e)}$$

where

$$P(e) = \frac{TP + FP}{total} \times \frac{TP + FN}{total} + \frac{TN + FN}{total} \times \frac{FP + TN}{total}$$

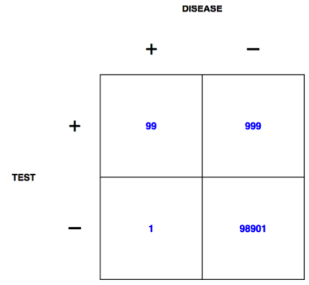


Sensitivity	→ Pr (positive test | disease)
Specificity	→ Pr (negative test | no disease)
Positive Predictive Value	→ Pr (disease | positive test)
Negative Predictive Value	→ Pr (no disease | negative test)
Accuracy	→ Pr (correct outcome)

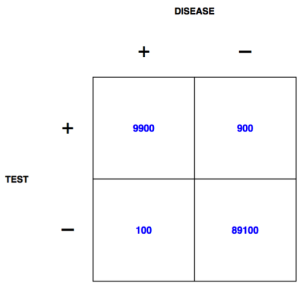


Sensitivity	→ TP / (TP+FN)
Specificity	→ TN / (FP+TN)
Positive Predictive Value	→ TP / (TP+FP)
Negative Predictive Value	→ TN / (FN+TN)
Accuracy	→ (TP+TN) / (TP+FP+FN+TN)

- ***example***
  - given that a disease has 0.1% prevalence in the population, we want to know what's probability of a person having the disease given the test result is positive? the test kit for the disease is 99% sensitive (most positives = disease) and 99% specific (most negatives = no disease)
  - what about 10% prevalence?



Sensitivity	$\rightarrow$ 99 / (99+1) = 99%
Specificity	$\rightarrow$ 98901 / (999+98901) = 99%
Positive Predictive Value	$\rightarrow$ 99 / (99+999) $\approx$ 9%
Negative Predictive Value	$\rightarrow$ 98901 / (1+98901) $>$ 99.9%
Accuracy	$\rightarrow$ (99+98901) / 100000 = 99%



Sensitivity	$\rightarrow$ 9900 / (9900+100) = 99%
Specificity	$\rightarrow$ 89100 / (900+89100) = 99%
Positive Predictive Value	$\rightarrow$ 9900 / (9900+900) $\approx$ 92%
Negative Predictive Value	$\rightarrow$ 89100 / (100+89100) $\approx$ 99.9%
Accuracy	$\rightarrow$ (9900+89100) / 100000 = 99%

Notable Measurements for Error – Continuous Variables

- **Mean squared error (MSE) =**

$$\frac{1}{n}\sum_{i=1}^{n}(Prediction_i - Truth_i)^2$$

- **Root mean squared error (RMSE) -**

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(Prediction_i - Truth_i)^2}$$

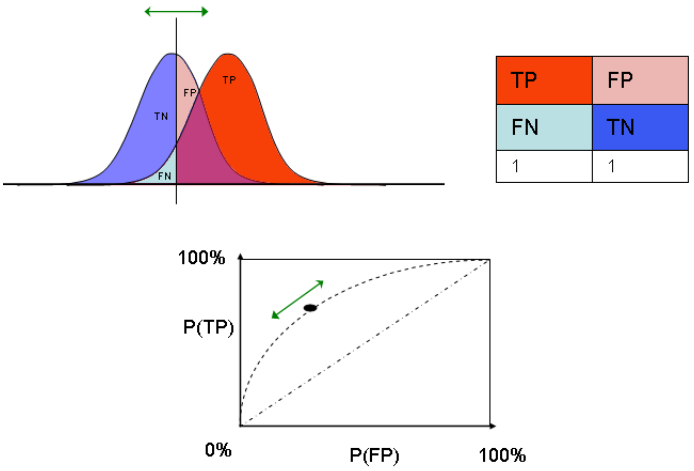
  - in the same units as variable
  - most commonly used error measure for continuous data
  - is not an effective measures when there are outliers
    - one large value may significantly raise the RMSE

- **median absolute deviation =**

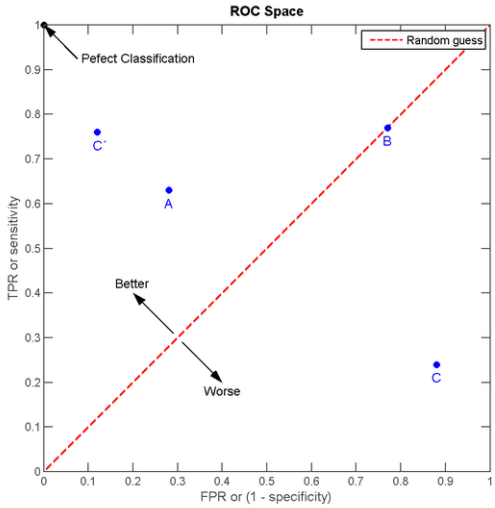
$$median(|Prediction_i - Truth_i|)$$

Receiver Operating Characteristic Curves

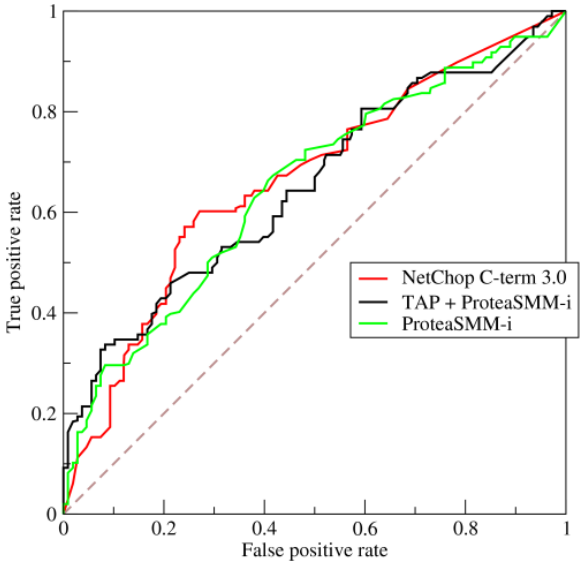
- are commonly used techniques to measure the quality of a prediction algorithm.
- predictions for binary classification often are quantitative (i.e. probability, scale of 1 to 10)
  - different cutoffs/threshold of classification ($> 0.8 \rightarrow$ one outcome) yield different results/predictions
  - **Receiver Operating Characteristic** curves are generated to compare the different outcomes



- ROC Curves
  - ***x-axis*** = 1 - specificity (or, probability of false positive)
  - ***y-axis*** = sensitivity (or, probability of true positive)
  - ***points plotted*** = cutoff/combination
  - ***areas under curve*** = quantifies whether the prediction model is viable or not
    - higher area $\rightarrow$ better predictor
    - area = 0.5 $\rightarrow$ effectively random guessing (diagonal line in the ROC curve)
    - area = 1 $\rightarrow$ perfect classifier
    - area = 0.8 $\rightarrow$ considered good for a prediction algorithm



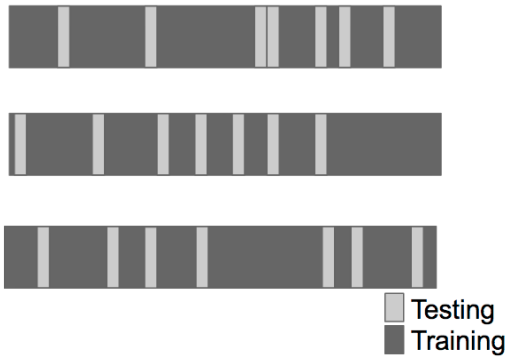
- ***example***
  - each point on the graph corresponds with a specificity and sensitivity



Cross Validation

- **procedures**
  1. split training set into sub-training/test sets
  2. build model on sub-training set
  3. evaluate on sub-test set
  4. repeat and average estimated errors
- **result**
  - we are able to fit/test various different models with different variables included to the find the best one on the cross-validated test sets
  - we are able to test out different types of prediction algorithms to use and pick the best performing one
  - we are able to choose the parameters in prediction function and estimate their values
  - ***Note**: original test set completely untouched, so when final prediction algorithm is applied, the result will be an unbiased measurement of the **out of sample accuracy** of the model*
- **approaches**
  - random subsampling
  - K-fold
  - leave one out
- **considerations**
  - for time series data data must be used in "chunks"
    - one time period might depending all time periods previously (should not take random samples)
  - if you cross-validate to pick predictors, the out of sample error rate may not be the most accurate and thus the errors should still be measured on independent data

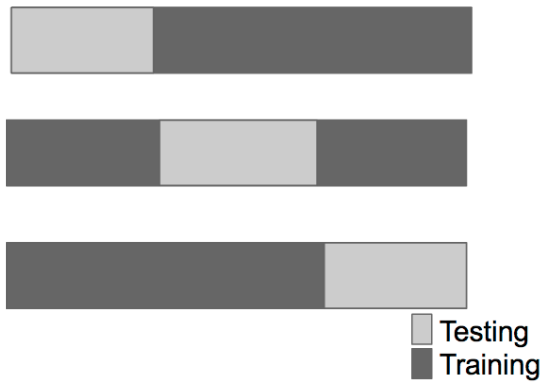
Random Subsampling



- a randomly sampled test set is subsetted out from the original training set
- the predictor is built on the remaining training data and applied to the test set

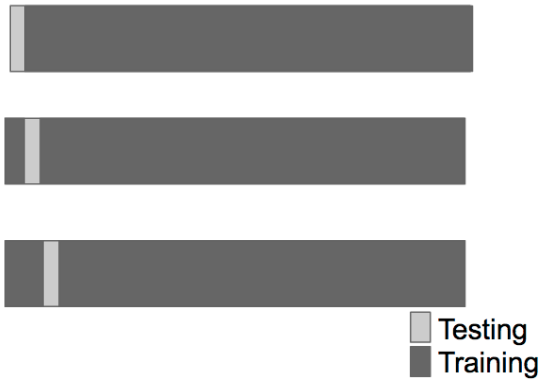
- the above are ***three*** random subsamplings from the same training set
- ***considerations***
  - must be done *without replacement*
  - random sampling with replacement = *bootstrap*
    - underestimates of the error, since the if we get one right and the sample appears more than once we'll get the other right
    - can be corrected with the (0.632 Bootstrap), but it is complicated

K-Fold



- break training set into K subsets (above is a 3-fold cross validation)
- build the model/predictor on the remaining training data in each subset and applied to the test subset
- rebuild the data K times with the training and test subsets and average the findings
- ***considerations***
  - larger k = less bias, more variance
  - smaller k = more bias, less variance

Leave One Out



- leave out exactly one sample and build predictor on the rest of training data
- predict value for the left out sample
- repeat for each sample

`caret` Package (tutorial)

- **core functionality**
  - preprocessing/cleaning data → `preProcess()`
  - cross validation/data splitting → `createDataPartition()`, `createResample()`, `createTimeSlices()`
  - train algorithms on training data and apply to test sets → `train()`, `predict()`
  - model comparison (evaluate the accuracy of model on new data) → `confusionMatrix()`
- machine learning algorithms in `caret` package
  - linear discriminant analysis
  - regression
  - naive Bayes
  - support vector machines
  - classification and regression trees
  - random forests
  - boosting
  - many others
- `caret` provides uniform framework to build/predict using different models
  - create objects of different classes for different algorithms, and `caret` package allows algorithms to be run the same way through `predict()` function

`obj` Class	Package	`predict` Function Syntax
`lda`	`MASS`	`predict(obj)` (no options needed)
`glm`	`stats`	`predict(obj, type = "response")`
`gbm`	`gbm`	`predict(obj, type = "response", n.trees)`
`mda`	`mda`	`predict(obj, type = "posterior")`
`rpart`	`rpart`	`predict(obj, type = "prob")`
`Weka`	`RWeka`	`predict(obj, type = "probability")`
`LogitBoost`	`caTools`	`predict(obj, type = "raw", nIter)`

Data Slicing

- `createDataPartition(y=data$var, times=1, p=0.75, list=FALSE)` → creates data partitions using given variable
  - `y=data$var` = specifies what outcome/variable to split the data on
  - `times=1` = specifies number of partitions to create (number of data splitting performed)
  - `p=0.75` = percent of data that will be for training the model
  - `list=FALSE` = returns a matrix of indices corresponding to **p%** of the data (training set)
    - ***Note**: matrix is easier to subset the data with, so `list = FALSE` is generally what is used*
    - `list=TRUE` = returns a list of indices corresponding to **p%** of the data (training set)

- the function effectively returns a list of indexes of the training set which can then be leveraged to subset the data
  - `training<-data[inTrain, ]` = subsets the data to training set only
  - `testing<-data[-inTrain, ]` = the rest of the data set can then be stored as the test set
- ***example***

```
# load packages and data
p_load("caret")
# create training set indexes with 75% of data
inTrain <- createDataPartition(y=spam$type,p=0.75, list=FALSE)
# subset spam data to training
training <- spam[inTrain,]
# subset spam data (the rest) to test
testing <- spam[-inTrain,]
# dimension of original and training dataset
rbind("original dataset" = dim(spam),"training set" = dim(training))
```

```
##                  [,1] [,2]
## original dataset 4601   58
## training set     3451   58
```

- `createFolds(y=data$var, k=10, list=TRUE, returnTrain=TRUE)` = slices the data in to k folds for cross validation and returns k lists of indices
  - `y=data$var` = specifies what outcome/variable to split the data on
  - `k=10` = specifies number of folds to create (See ***K Fold Cross Validation***)
    - each training set has approximately has $\frac{k-1}{k}\%$ of the data (in this case 90%)
    - each training set has approximately has $\frac{1}{k}\%$ of the data (in this case 10%)
  - `list=TRUE` = returns `k` list of indices that corresponds to the cross-validated sets
    - ***Note****: the returned list conveniently splits the data into* `k` *datasets/vectors of indices, so* `list=TRUE` *is generally what is used*
    - when the returned object is a list (called `folds` in the case), you can use `folds[[1]][1:10]` to access different elements from that list
    - `list=FALSE` = returns a vector indicating which of the `k` folds each data point belongs to (i.e. 1 - 10 is assigned for each of the data points in this case)
      - ***Note****: these group values corresponds to test sets for each cross validation, which means everything else besides the marked points should be used for training*
  - [only works when `list=T`] `returnTrain=TRUE` = returns the indices of the training sets
    - [default value when unspecified]`returnTrain=FALSE` = returns indices of the test sets
  - ***example***

```
# create 10 folds for cross validation and return the training set indices
folds <- createFolds(y=spam$type,k=10,list=TRUE,returnTrain=TRUE)
# structure of the training set indices
str(folds)
```

```
## List of 10
##  $ Fold01: int [1:4141] 1 2 3 4 6 7 8 9 10 12 ...
##  $ Fold02: int [1:4140] 1 2 3 4 5 7 8 9 10 11 ...
##  $ Fold03: int [1:4141] 1 3 4 5 6 7 8 9 10 11 ...
```

```
##  $ Fold04: int [1:4141] 2 3 4 5 6 7 8 9 10 11 ...
##  $ Fold05: int [1:4141] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold06: int [1:4141] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold07: int [1:4141] 1 2 3 4 5 6 8 9 11 12 ...
##  $ Fold08: int [1:4141] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold09: int [1:4141] 1 2 4 5 6 7 10 11 12 13 ...
##  $ Fold10: int [1:4141] 1 2 3 5 6 7 8 9 10 11 ...
```

```r
# return the test set indices instead
# note: returnTrain = FALSE is unnecessary as it is the default behavior
folds.test <- createFolds(y=spam$type,k=10,list=TRUE,returnTrain=FALSE)
str(folds.test)
```

```
## List of 10
##  $ Fold01: int [1:460] 15 16 18 40 45 62 68 81 82 102 ...
##  $ Fold02: int [1:459] 1 41 55 58 67 75 117 123 151 175 ...
##  $ Fold03: int [1:461] 3 14 66 69 70 80 90 112 115 135 ...
##  $ Fold04: int [1:460] 5 19 25 65 71 83 85 88 91 93 ...
##  $ Fold05: int [1:460] 6 10 17 21 26 56 57 104 107 116 ...
##  $ Fold06: int [1:459] 7 8 13 39 52 54 76 89 99 106 ...
##  $ Fold07: int [1:461] 4 23 27 29 32 33 34 38 49 51 ...
##  $ Fold08: int [1:460] 2 9 30 31 36 37 43 46 47 48 ...
##  $ Fold09: int [1:461] 12 20 24 44 53 59 60 64 84 98 ...
##  $ Fold10: int [1:460] 11 22 28 35 42 61 72 86 92 118 ...
```

```r
# return first 10 elements of the first training set
folds[[1]][1:10]
```

```
##  [1]  1  2  3  4  6  7  8  9 10 12
```

- `createResample(y=data$var, times=10, list=TRUE)` = create 10 resamplings from the given data with replacement
  - `list=TRUE` = returns list of n vectors that contain indices of the sample
    - ***Note**: each of the vectors is of length of the data, and contains indices*
  - `times=10` = number of samples to create

```r
# create 10 resamples
resamples <- createResample(y=spam$type,times=10,list=TRUE)
# structure of the resamples (note some samples are repeated)
str(resamples)
```

```
## List of 10
##  $ Resample01: int [1:4601] 1 4 4 4 7 8 12 13 13 14 ...
##  $ Resample02: int [1:4601] 3 3 5 7 10 12 12 13 13 14 ...
##  $ Resample03: int [1:4601] 1 2 2 3 4 5 8 10 11 12 ...
##  $ Resample04: int [1:4601] 1 3 3 4 7 8 8 9 10 14 ...
##  $ Resample05: int [1:4601] 2 4 5 6 7 7 8 8 9 12 ...
##  $ Resample06: int [1:4601] 3 6 6 7 8 9 12 13 13 14 ...
##  $ Resample07: int [1:4601] 1 2 2 5 5 6 7 8 9 10 ...
##  $ Resample08: int [1:4601] 2 2 3 4 4 7 7 8 8 9 ...
##  $ Resample09: int [1:4601] 1 4 7 8 8 9 12 13 15 15 ...
##  $ Resample10: int [1:4601] 1 3 4 4 7 7 9 9 10 11 ...
```

- `createTimeSlices(y=data, initialWindow=20, horizon=10)` = creates training sets with specified window length and the corresponding test sets
  - `initialWindow=20` = number of consecutive values in each time slice/training set (i.e. values 1 - 20)
  - `horizon=10` = number of consecutive values in each predict/test set (i.e. values 21 - 30)
  - `fixedWindow=FALSE` = training sets always start at the first observation
    * this means that the first training set would be 1 - 20, the second will be 1 - 21, third 1 - 22, etc.
    * but the test sets are still like before (21 - 30, 22 - 31, etc.)

```
# create time series data
tme <- 1:1000
# create time slices
folds <- createTimeSlices(y=tme,initialWindow=20,horizon=10)
# name of lists
names(folds)
```

```
## [1] "train" "test"
```

```
# first training set
folds$train[[1]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
# first test set
folds$test[[1]]
```

```
##  [1] 21 22 23 24 25 26 27 28 29 30
```

Training Options (tutorial)

- `train(y ~ x, data=df, method="glm")` = function to apply the machine learning algorithm to construct model from training data

```
# returns the arguments of the default train function
args(train.default)
```

```
## function (x, y, method = "rf", preProcess = NULL, ..., weights = NULL,
##     metric = ifelse(is.factor(y), "Accuracy", "RMSE"), maximize = ifelse(metric %in%
##         c("RMSE", "logLoss"), FALSE, TRUE), trControl = trainControl(),
##     tuneGrid = NULL, tuneLength = 3)
## NULL
```

- `train` function has a large set of parameters, below are the default options
  - `method="rf"` = default algorithm is random forest for training a given data set; `caret` contains a large number of algorithms
    * `names(getModelInfo())` = returns all the options for `method` argument
    * list of models and their information can be found here
  - `preProcess=NULL` = set preprocess options (see ***Preprocessing***)

- `weights=NULL` = can be used to add weights to observations, useful for unbalanced distribution (a lot more of one type than another)
- `metric=ifelse(is.factor(y), "Accuracy", "RMSE")` = default metric for algorithm is *Accuracy* for factor variables, and *RMSE*, or root mean squared error, for continuous variables
  - *Kappa* = measure of concordance (see ***Notable Measurements for Error – Binary Variables***)
  - *RSquared* can also be used here as a metric, which represents R^2 from regression models (only useful for linear models)
- `maximize=ifelse(metric=="RMSE", FALSE, TRUE)` = the algorithm should maximize *accuracy* and minimize *RMSE*
- `trControl=trainControl()` = training controls for the model, more details below
- `tuneGrid=NULL`
- `tuneLength=3`

```
# returns the default arguments for the trainControl object
args(trainControl)
```

```
## function (method = "boot", number = ifelse(grepl("cv", method),
##     10, 25), repeats = ifelse(grepl("cv", method), 1, number),
##     p = 0.75, search = "grid", initialWindow = NULL, horizon = 1,
##     fixedWindow = TRUE, verboseIter = FALSE, returnData = TRUE,
##     returnResamp = "final", savePredictions = FALSE, classProbs = FALSE,
##     summaryFunction = defaultSummary, selectionFunction = "best",
##     preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5),
##     sampling = NULL, index = NULL, indexOut = NULL, timingSamps = 0,
##     predictionBounds = rep(FALSE, 2), seeds = NA, adaptive = list(min = 5,
##         alpha = 0.05, method = "gls", complete = TRUE), trim = FALSE,
##     allowParallel = TRUE)
## NULL
```

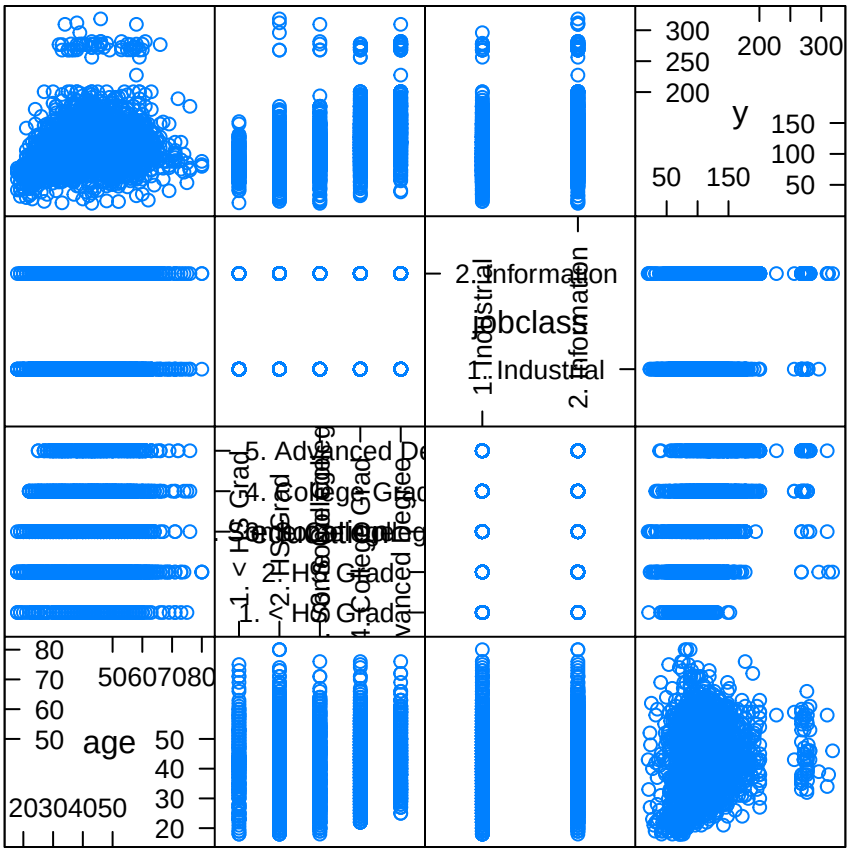
- `trainControl` creates an object that sets many options for how the model will be applied to the training data
  - ***Note**: the default values are listed below but you can use them to set the parameters to your discretion*
  - `method="boot"` =
    - `"boot"` = bootstrapping (drawing with replacement)
    - `"boot632"` = bootstrapping with adjustment
    - `"cv"` = cross validation
    - `"repeatedcv"` = repeated cross validation
    - `"LOOCV"` = leave one out cross validation
  - `number=ifelse(grepl("cv", method),10, 25)` = number of subsamples to take
    - `number=10` = default for any kind of cross validation
    - `number=25` = default for bootstrapping
    - ***Note**: `number` should be increased when fine-tuning model with large number of parameter*
  - `repeats=ifelse(grepl("cv", method), 1, number)` = numbers of times to repeat the subsampling
    - `repeats=1` = default for any cross validation method
    - `repeats=25` = default for bootstrapping
  - `p=0.75` = default percentage of data to create training sets
  - `initialWindow=NULL, horizon=1, fixedWindow=TRUE` = parameters for time series data

  - `verboseIter=FALSE` = print the training logs
  - `returnData=TRUE`, returnResamp = "final",
  - `savePredictions=FALSE` = save the predictions for each resample
  - `classProbs=FALSE` = return classification probabilities along with the predictions
  - `summaryFunction=defaultSummary` = default summary of the model,
  - `preProcOptions=list(thresh = 0.95, ICAcomp = 3, k = 5)` = specifies preprocessing options for the model
  - `predictionBounds=rep(FALSE, 2)` = specify the range of the predicted value
    - for numeric predictions, `predictionBounds=c(10, NA)` would mean that any value lower than 10 would be treated as 10 and no upper bounds
  - `seeds=NA` = set the seed for the operation
    - ***Note**: setting this is important when you want to reproduce the same results when the `train` function is run*
  - `allowParallel=TRUE` = sets for parallel processing/computations

Plotting Predictors (tutorial)

- it is important to only plot the data in the training set
  - using the test data may lead to over-fitting (model should not be adjusted to test set)
- goal of producing these exploratory plots = look for potential outliers, skewness, imbalances in outcome/predictors, and explainable groups of points/patterns
- `featurePlot(x=predictors, y=outcome, plot="pairs")` = short cut to plot the relationships between the predictors and outcomes

```
# load relevant libraries
p_load("ISLR", "ggplot2")
# load wage data
data(Wage)
# create training and test sets
inTrain <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
training <- Wage[inTrain,]
testing <- Wage[-inTrain,]
# plot relationships between the predictors and outcome
featurePlot(x=training[,c("age","education","jobclass")], y = training$wage,plot="pairs")
```



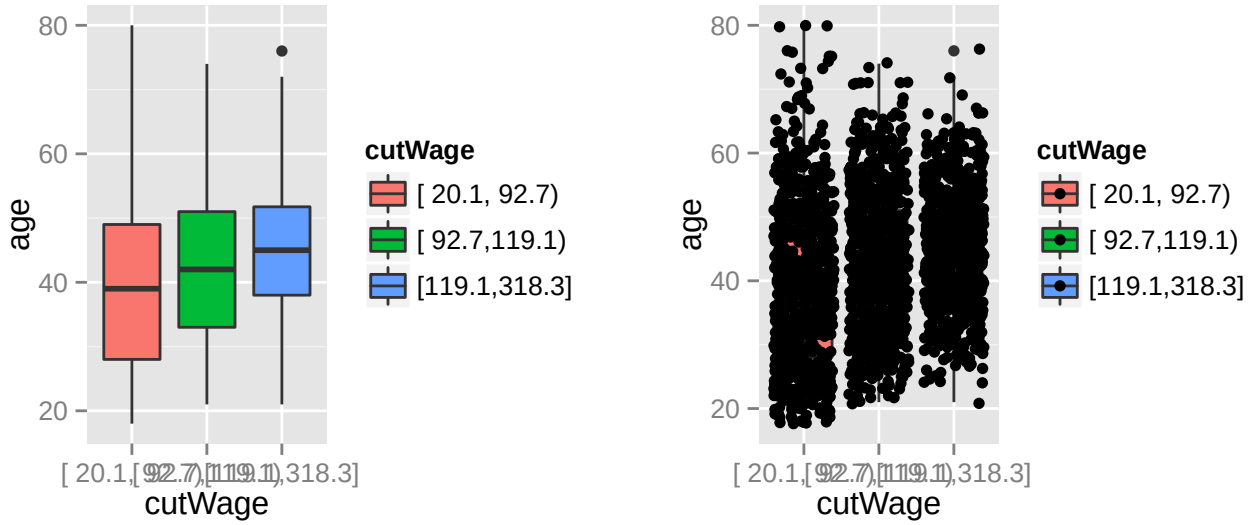
Scatter Plot Matrix

- `qplot(age, wage, color=eduction, data=training)` = can be used to separate data by a factor variable (by coloring the points differently)
  - `geom_smooth(method = "lm")` = adds a regression line to the plots
  - `geom=c("boxplot", "jitter")` = specifies what kind of plot to produce, in this case both the boxplot and the point cloud

```
# qplot plus linear regression lines
qplot(age,wage,colour=education,data=training)+geom_smooth(method='lm',formula=y~x)
```

- `cut2(variable, g=3)` = creates a new factor variable by cutting the specified variable into n groups (3 in this case) based on percentiles
  - ***Note**: `cut2` function is part of the `Hmisc` package, so `p_load("Hmisc")` must be run first*
  - this variable can then be used to tabulate/plot the data
- `grid.arrange(p1, p2, ncol=2)` = `ggplot2` function the print multiple graphs on the same plot
  - ***Note**: `grid.arrange` function is part of the `gridExtra` package, so `p_load("gridExtra")` must be run first*

```
# load Hmisc and gridExtra packages
p_load("Hmisc", "gridExtra")
# cut the wage variable
cutWage <- cut2(training$wage,g=3)
# plot the boxplot
p1 <- qplot(cutWage,age, data=training,fill=cutWage,
      geom=c("boxplot"))
# plot boxplot and point clusters
p2 <- qplot(cutWage,age, data=training,fill=cutWage,
      geom=c("boxplot","jitter"))
# plot the two graphs side by side
grid.arrange(p1,p2,ncol=2)
```



- `table(cutVariable, data$var2)` = tabulates the cut factor variable vs another variable in the dataset (ie; builds a contingency table using cross-classifying factors)
- `prop.table(table, margin=1)` = converts a table to a proportion table
  - `margin=1` = calculate the proportions based on the rows
  - `margin=2` = calculate the proportions based on the columns

```
# tabulate the cutWage and jobclass variables
t <- table(cutWage,training$jobclass)
# print table
t
```

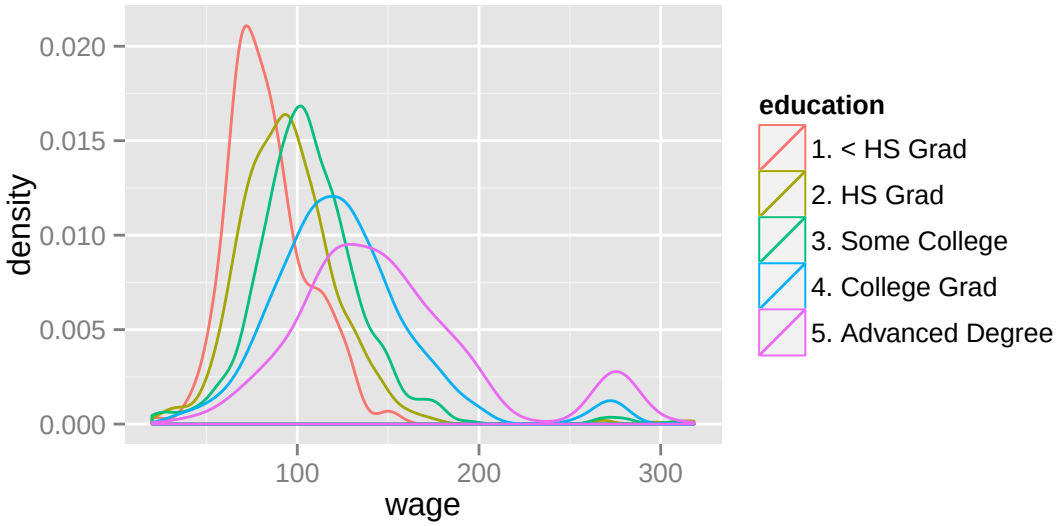
```
##
## cutWage          1. Industrial 2. Information
##   [ 20.1, 92.7)           445            256
##   [ 92.7,119.1)           376            347
##   [119.1,318.3]           269            409
```

```
# convert to proportion table based on the rows
prop.table(t,1)
```

```
##
## cutWage          1. Industrial 2. Information
##   [ 20.1, 92.7)       0.6348074      0.3651926
##   [ 92.7,119.1)       0.5200553      0.4799447
##   [119.1,318.3]       0.3967552      0.6032448
```

- `qplot(var1, color=var2, data=training, geom="density")` = produces density plot for the given numeric and factor variables
  - effectively smoothed out histograms
  - provides for easy overlaying of groups of data
    - break different variables up by group and see how outcomes change between groups

```
# produce density plot
qplot(wage,colour=education,data=training,geom="density")
```



Preprocessing (tutorial)

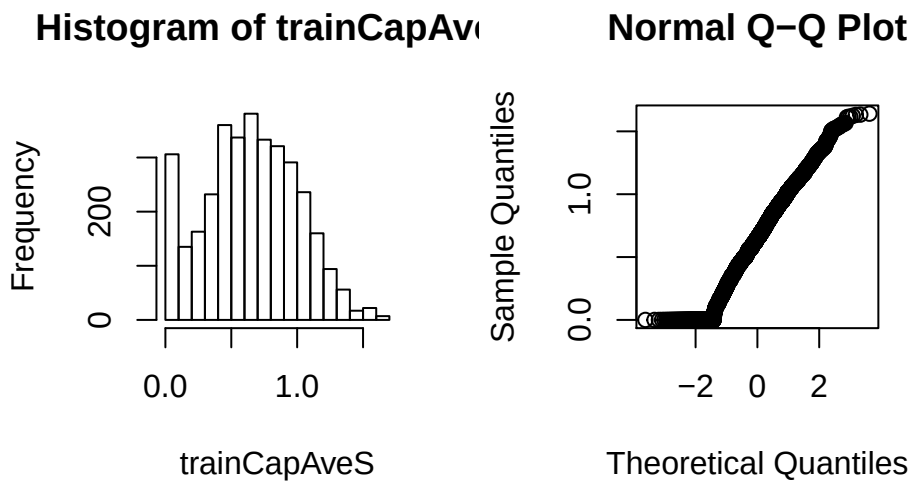
- some predictors may have strange distributions (i.e. skewed) and may need to be transformed to be more useful for prediction algorithm
  - particularly true for model based algorithms → naive Bayes, linear discriminate analysis, linear regression
- **centering** = subtracting the observations of a particular variable by its mean
- **scaling** = dividing the observations of a particular variable by its standard deviation
- **normalizing** = centering and scaling the variable → effectively converting each observation to the number of standard deviations away from the mean
  - the distribution of the normalized variable will have a mean of 0 and standard deviation of 1
  - ***Note****: normalizing data can help remove bias and high variability, but may not be applicable in all cases*
- ***Note****: if a predictor/variable is standardized when training the model, the same transformations must be performed on the* `test` *set with the mean and standard deviation of the* `train` *variables*
  - this means that the mean and standard deviation of the normalized test variable will ***NOT*** be 0 and 1, respectively, but will be close
  - transformations must likely be imperfect but test/train sets must be processed the same way
- `train(y~x, data=training, preProcess=c("center", "scale"))` = preprocessing can be directly specified in the `train` function
  - `preProcess=c("center", "scale")` = normalize all predictors before constructing model
- `preProcess(trainingData, method=c("center", "scale")` = function in the `caret` to standardize data
  - you can store the result of the `preProcess` function as an object and apply it to the `train` and `test` sets using the `predict` function

```r
# load spam data
data(spam)
# create train and test sets
inTrain <- createDataPartition(y=spam$type,p=0.75, list=FALSE)
training <- spam[inTrain,]
testing <- spam[-inTrain,]
# create preProcess object for all predictors ("-58" because 58th = outcome)
preObj <- preProcess(training[,-58],method=c("center","scale"))
# normalize training set
trainCapAveS <- predict(preObj,training[,-58])$capitalAve
# normalize test set using training parameters
testCapAveS <- predict(preObj,testing[,-58])$capitalAve
# compare results for capitalAve variable
rbind(train = c(mean = mean(trainCapAveS), std = sd(trainCapAveS)),
    test = c(mean(testCapAveS), sd(testCapAveS)))
```

```
##               mean      std
## train 6.097035e-18 1.000000
## test  7.548133e-02 1.633866
```

- `preprocess(data, method="BoxCox")` = applies BoxCox transformations to continuous data to help normalize the variables through maximum likelihood
  - ***Note**: note it assumes continuous values and DOES NOT deal with repeated values*
  - `qqnorm(processedVar)` = can be used to produce the Q-Q plot which compares the theoretical quantiles with the sample quantiles to see the normality of the data

```r
# set up BoxCox transforms
p_load("e1071")
preObj <- preProcess(training[,-58],method=c("BoxCox"))
# perform preprocessing on training data
trainCapAveS <- predict(preObj,training[,-58])$capitalAve
# plot histogram and QQ Plot
# Note: the transformation definitely helped to
# normalize the data but it does not produce perfect result
par(mfrow=c(1,2)); hist(trainCapAveS); qqnorm(trainCapAveS)
```



- `preProcess(data, method="knnImpute")` = impute/estimate the missing data using **k nearest neighbors (knn)** imputation
  - `knnImpute` = takes the k nearest neighbors from the missing value and averages the value to impute the missing observations
  - ***Note**: most prediction algorithms are not build to handle missing data*

```
# Make some values NA
p_load("RANN")
training$capAve <- training$capitalAve
selectNA <- rbinom(dim(training)[1],size=1,prob=0.05)==1
training$capAve[selectNA] <- NA
# Impute and standardize
preObj <- preProcess(training[,-58],method="knnImpute")
capAve <- predict(preObj,training[,-58])$capAve
# Standardize true values
capAveTruth <- training$capitalAve
capAveTruth <- (capAveTruth-mean(capAveTruth))/sd(capAveTruth)
# compute differences between imputed values and true values
quantile(capAve - capAveTruth)
```

```
##            0%           25%           50%           75%          100%
## -1.656344e+00  2.377772e-05  1.286900e-03  1.880821e-03  3.174413e-01
```

Covariate Creation/Feature Extraction

- **[level 1]**: construct covariate (usable metric, feature) from raw data depends heavily on application
  - ideally we want to summarize data without too much information loss
  - examples
    - *text files*: frequency of words, frequency of phrases (Google ngrams), frequency of capital letters
    - *images*: edges, corners, blobs, ridges (computer vision feature detection)
    - *webpages*: number and type of images, position of elements, colors, videos (A/B Testing)
    - *people*: height, weight, hair color, sex, country of origin
  - generally, more knowledge and understanding you have of the system/data, the easier it will be to extract the summarizing features
    - when in doubt, more features is always safer → lose less information and the features can be filtered during model construction
  - this process can be automated (i.e. PCA) but generally have to be very careful, as one very useful feature in the training data set may not have as much effect on the test data set
  - ***Note**: science is the key here, Google "feature extraction for [data type]" for more guidance*
    - the goal is always to find the salient characteristics that are likely to be different from observation to observation
- **[level 2]**: construct new covariates from extracted covariate
  - generally transformations of features you extract from raw data
  - used more for methods like regression and support vector machines (SVM), whose accuracy depend more on the distribution of input variables
  - models like classification trees don't require as many complex covariates
  - best approach is through exploratory analysis (tables/plots)
  - should only be performed on the train dataset
  - new covariates should be added to data frames under recognizable names so they can be used later
  - `preProcess()` can be leveraged to handle creating new covariates
  - ***Note**: always be careful about over-fitting*

Creating Dummy Variables

- convert factor variables to indicator/dummy variable → qualitative become quantitative
- `dummyVars(outcome~var, data=training)` = creates a dummy variable object that can be used through `predict` function to create dummy variables
  - `predict(dummyObj, newdata=training)` = creates appropriate columns to represent the factor variable with appropriate 0s and 1s
    - 2 factor variable → two columns which have 0 or 1 depending on the outcome
    - 3 factor variable → three columns which have 0, 0, and 1 representing the outcome
    - ***Note**: only one of the columns can have values of 1 for each observation*

```
# setting up data
inTrain <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
# create a dummy variable object
dummies <- dummyVars(wage ~ jobclass,data=training)
# create the dummy variable columns
head(predict(dummies,newdata=training))
```

```
##        jobclass.1. Industrial jobclass.2. Information
## 231655                       1                       0
## 86582                        0                       1
## 161300                       1                       0
## 155159                       0                       1
## 11443                        0                       1
## 376662                       0                       1
```

Removing Zero Covariates

- some variables have no variability at all (i.e. variable indicating if an email contained letters)
- these variables are not useful when we want to construct a prediction model
- `nearZeroVar(training, saveMetrics=TRUE)` = returns list of variables in training data set with information on frequency ratios, percent uniques, whether or not it has zero variance
  - `freqRatio` = ratio of frequencies for the most common value over second most common value
  - `percentUnique` = percentage of unique data points out of total number of data points
  - `zeroVar` = TRUE/FALSE indicating whether the predictor has only one distinct value
  - `nzv` = TRUE/FALSE indicating whether the predictor is a near zero variance predictor
  - ***Note**: when `nzv` = TRUE, those variables should be thrown out*

```
# print nearZeroVar table
nearZeroVar(training,saveMetrics=TRUE)
```

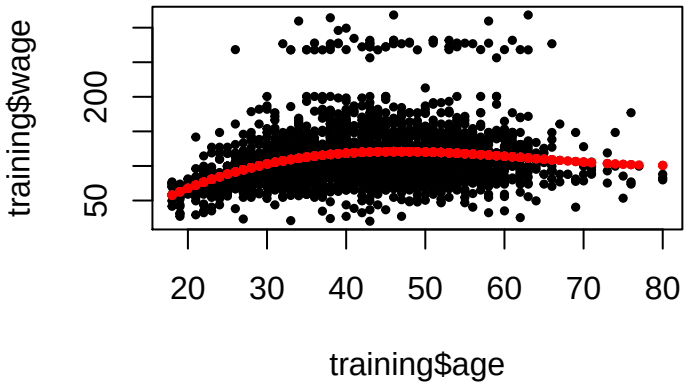
```
##            freqRatio percentUnique zeroVar   nzv
## year        1.017647    0.33301618   FALSE FALSE
## age         1.231884    2.85442436   FALSE FALSE
## sex         0.000000    0.04757374    TRUE  TRUE
## maritl      3.329571    0.23786870   FALSE FALSE
## race        8.480583    0.19029496   FALSE FALSE
## education   1.393750    0.23786870   FALSE FALSE
## region      0.000000    0.04757374    TRUE  TRUE
## jobclass    1.070936    0.09514748   FALSE FALSE
## health      2.526846    0.09514748   FALSE FALSE
## health_ins  2.209160    0.09514748   FALSE FALSE
## logwage     1.011765   18.83920076   FALSE FALSE
## wage        1.011765   18.83920076   FALSE FALSE
```

Creating Splines (Polynomial Functions)

- when you want to fit curves through the data, basis functions can be leveraged
- [`splines` package] `bs(data$var, df=3)` = creates 3 new columns corresponding to the var, var^2, and var^3 terms
- `ns()` and `poly()` can also be used to generate polynomials
- `gam()` function can also be used and it allows for smoothing of multiple variables with different values for each variable
- ***Note**: the same polynomial operations must be performed on the test sets using the `predict` function*

```
# load splines package
p_load("splines")
# create polynomial function
```

```
bsBasis <- bs(training$age,df=3)
# fit the outcome on the three polynomial terms
lm1 <- lm(wage ~ bsBasis,data=training)
# plot all age vs wage data
plot(training$age,training$wage,pch=19,cex=0.5)
# plot the fitted polynomial function
points(training$age,predict(lm1,newdata=training),col="red",pch=19,cex=0.5)
```




```
# predict on test values
head(predict(bsBasis,age=testing$age))
```

```
##              1          2           3
## [1,] 0.0000000 0.00000000 0.000000000
## [2,] 0.2368501 0.02537679 0.000906314
## [3,] 0.4163380 0.32117502 0.082587862
## [4,] 0.4308138 0.29109043 0.065560908
## [5,] 0.3625256 0.38669397 0.137491189
## [6,] 0.3063341 0.42415495 0.195763821
```

Multicore Parallel Processing

- many of the algorithms in the `caret` package are computationally intensive
- since most of the modern machines have multiple cores on their CPUs, it is often wise to enable ***multicore parallel processing*** to expedite the computations
- `doMC` package is recommended to be used for `caret` computations (reference)
  - `doMC::registerDoMC(cores=4)` = registers 4 cores for R to utilize
  - the number of cores you should specify depends on the CPU on your computer (system information usually contains the number of cores)
    - it's also possible to find the number of cores by directly searching for your CPU model number on the Internet
  - ***Note****: once registered, you should see in your task manager/activity monitor that 4 "R Session" appear when you run your code*

Preprocessing with Principal Component Analysis (PCA)

- constructing a prediction model may not require every predictor
- ideally we want to capture the ***most variation*** with the ***least*** amount of variables
  - weighted combination of predictors may improve fit
  - combination needs to capture the *most information*
- PCA is suited to do this and will help reduce number of predictors as well as reduce noise (due to averaging)
  - statistical goal = find new set of multivariate variables that are *uncorrelated* and explain as much variance as possible
  - data compression goal = find the best matrix created with fewer variables that explains the original data
  - PCA is most useful for linear-type models (GLM, LDA)
  - generally more difficult to interpret the predictors (complex weighted sums of variables)
  - ***Note****: outliers are can be detrimental to PCA as they may represent a lot of variation in data*
    - exploratory analysis (plots/tables) should be used to identify problems with the predictors
    - transformations with log/BoxCox may be helpful

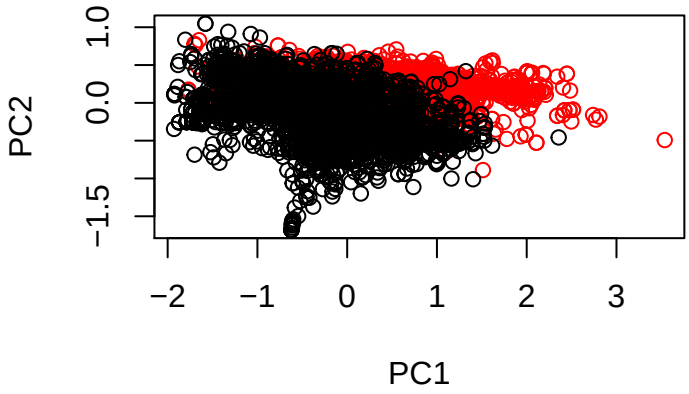
`prcomp` Function

- `pr<-prcomp(data)` = performs PCA on all variables and returns a `prcomp` object that contains information about standard deviations and rotations
  - `pr$rotations` = returns eigenvectors for the linear combinations of all variables (coefficients that variables are multiplied by to come up with the principal components) → how the principal components are created
  - often times, it is useful to take the `log` transformation of the variables and adding 1 before performing PCA
    - helps to reduce skewness or strange distribution in data
    - log(0) = - infinity, so we add 1 to account for zero values
    - makes data more Gaussian
  - `plot(pr)` = plots the percent variation explained by the first 10 principal components (PC)
    - can be used to find the PCs that represent the most variation

```
# load  spam data
data(spam)
# perform PCA on dataset
prComp <- prcomp(log10(spam[,-58]+1))
# print out the eigenvector/rotations first 5 rows and PCs
head(prComp$rotation[, 1:5], 5)
```

```
##                 PC1           PC2         PC3         PC4          PC5
## make    0.019370409  0.0427855959 -0.01631961  0.02798232 -0.014903314
## address 0.010827343  0.0408943785  0.07074906 -0.01407049  0.037237531
## all     0.040923168  0.0825569578 -0.03603222  0.04563653  0.001222215
## num3d   0.006486834 -0.0001333549  0.01234374 -0.01005991 -0.001282330
## our     0.036963221  0.0941456085 -0.01871090  0.05098463 -0.010582039
```

```r
# create new variable that marks spam as 2 and nospam as 1
typeColor <- ((spam$type=="spam")*1 + 1)
# plot the first two principal components
plot(prComp$x[,1],prComp$x[,2],col=typeColor,xlab="PC1",ylab="PC2")
```



caret Package

- `pp<-preProcess(log10(training[,-58]+1),method="pca",pcaComp=2,thresh=0.8))` = perform PCA with `preProcess` function and returns the number of principal components that can capture the majority of the variation
  - creates a `preProcess` object that can be applied using `predict` function
  - `pcaComp=2` = specifies the number of principal components to compute (2 in this case)
  - `thresh=0.8` = threshold for variation captured by principal components
    - `thresh=0.95` = default value, which returns the number of principal components that are needed to capture 95% of the variation in data
- `predict(pp, training)` = computes new variables for the PCs (2 in this case) for the training data set
  - the results from `predict` can then be used as data for the prediction model
  - ***Note**: the same PCA must be performed on the test set*

```r
# create train and test sets
p_load("caret")
inTrain <- createDataPartition(y=spam$type,p=0.75, list=FALSE)
training <- spam[inTrain,]
testing <- spam[-inTrain,]
# create preprocess object
preProc <- preProcess(log10(training[,-58]+1),method="pca",pcaComp=2)
# calculate PCs for training data
trainPC <- predict(preProc,log10(training[,-58]+1))
# run model on outcome and principle components
modelFit <- train(training$type ~.,data=trainPC,method="glm")
```

```r
# calculate PCs for test data
testPC <- predict(preProc,log10(testing[,-58]+1))
# compare results
confusionMatrix(testing$type,predict(modelFit,testPC))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     656   41
##    spam         82  371
##
##                Accuracy : 0.893
##                  95% CI : (0.8737, 0.9103)
##     No Information Rate : 0.6417
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7724
##  Mcnemar's Test P-Value : 0.0003101
##
##             Sensitivity : 0.8889
##             Specificity : 0.9005
##          Pos Pred Value : 0.9412
##          Neg Pred Value : 0.8190
##              Prevalence : 0.6417
##          Detection Rate : 0.5704
##    Detection Prevalence : 0.6061
##       Balanced Accuracy : 0.8947
##
##        'Positive' Class : nonspam
##
```

- alternatively, PCA can be directly performed with the `train` method
  - `train(outcome ~ ., method="glm", preProcess="pca", data=training)` = performs PCA first on the training set and then runs the specified model
    - effectively the same procedures as above (`preProcess` → `predict`)
- ***Note**: in both cases, the PCs were able to achieve 90+% accuracy*

```r
# construct model
modelFit <- train(training$type ~ .,method="glm",preProcess="pca",data=training)
# print results of model
confusionMatrix(testing$type,predict(modelFit,testing))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction nonspam spam
##    nonspam     668   29
##    spam         59  394
##
##                Accuracy : 0.9235
```

```
##                  95% CI : (0.9066, 0.9382)
##     No Information Rate : 0.6322
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8379
##  Mcnemar's Test P-Value : 0.001992
##
##             Sensitivity : 0.9188
##             Specificity : 0.9314
##          Pos Pred Value : 0.9584
##          Neg Pred Value : 0.8698
##              Prevalence : 0.6322
##          Detection Rate : 0.5809
##    Detection Prevalence : 0.6061
##       Balanced Accuracy : 0.9251
##
##        'Positive' Class : nonspam
##
```

Predicting with Regression

- **prediction with regression** = fitting regression model (line) to data → multiplying each variable by coefficients to predict outcome
- useful when the relationship between the variables can be modeled as linear
- the model is easy to implement and the coefficients are easy to interpret
- if the relationships are non-linear, the regression model may produce poor results/accuracy
  - ***Note**: linear regressions are generally used in combination with other models*
- **model**

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \ldots + \beta_p X_{pi} + e_i$$

  - where β_0 is the intercept (when all variables are 0)
  - $\beta_1, \beta_2, \ldots, \beta_p$ are the coefficients
  - $X_{1i}, X_{2i}, \ldots, X_{pi}$ are the variables/covariates
  - e_i is the error
  - Y_i is the outcome
- **prediction**

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_{1i} + \hat{\beta}_2 X_{2i} + \ldots + \hat{\beta}_p X_{pi}$$

  - where $\hat{\beta}_0$ is the estimated intercept (when all variables are 0)
  - $\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_p$ are the estimated coefficients
  - $X_{1i}, X_{2i}, \ldots, X_{pi}$ are the variables/covariates
  - $\hat{Y}_i$ is the ***predicted outcome***

R Commands and Examples

- `lm<-lm(y ~ x, data=train)` = runs a linear model of outcome y on predictor x → univariate regression
  - `summary(lm)` = returns summary of the linear regression model, which will include coefficients, standard errors, t statistics, and p values
  - `lm(y ~ x1+x2+x3, data=train)` = run linear model of outcome y on predictors x1, x2, and x3
  - `lm(y ~ ., data=train` = run linear model of outcome y on all predictors
- `predict(lm, newdata=df)` = use the constructed linear model to predict outcomes ($\hat{Y}_i$) for the new values
  - `newdata` data frame must have the same variables (factors must have the same levels) as the training data
  - `newdata=test` = predict outcomes for the test set based on linear regression model from the training
  - ***Note**: the regression line will not be a perfect fit on the test set since it was constructed on the training set*
- RSME can be calculated to measure the accuracy of the linear model
  - ***Note**: $RSME_{test}$, which estimates the out-of-sample error, is almost always **GREATER** than $RSME_{train}$*

```
# load data
data(faithful)
# create train and test sets
inTrain <- createDataPartition(y=faithful$waiting, p=0.5, list=FALSE)
trainFaith <- faithful[inTrain,]; testFaith <- faithful[-inTrain,]
# build linear model
```

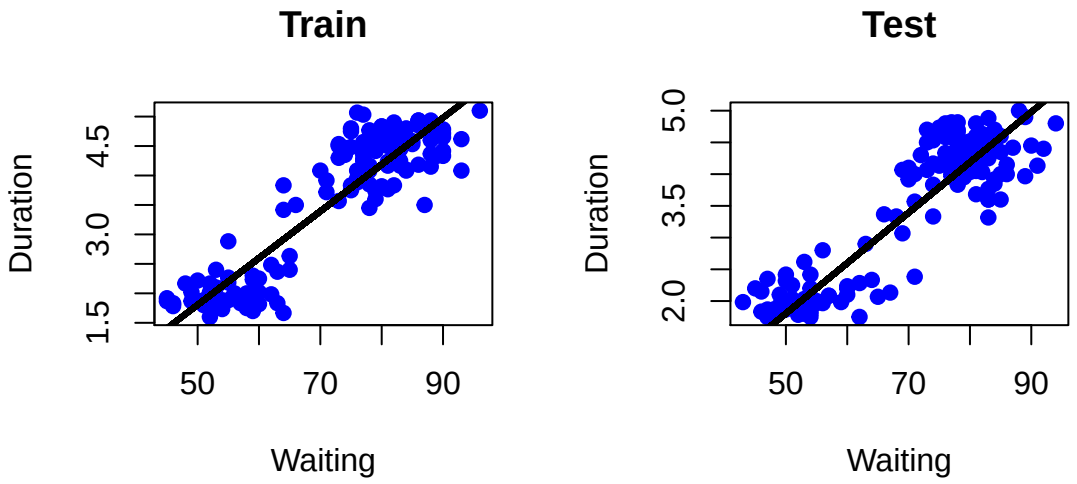
```r
lm1 <- lm(eruptions ~ waiting,data=trainFaith)
# print summary of linear model
summary(lm1)
```

```
##
## Call:
## lm(formula = eruptions ~ waiting, data = trainFaith)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.24867 -0.36292  0.00002  0.35768  1.19858
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2.165648   0.227486   -9.52   <2e-16 ***
## waiting      0.079396   0.003146   25.24   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5013 on 135 degrees of freedom
## Multiple R-squared:  0.8251, Adjusted R-squared:  0.8238
## F-statistic: 636.9 on 1 and 135 DF,  p-value: < 2.2e-16
```

```r
# predict eruptions for new waiting time
newdata <- data.frame(waiting=80)
predict(lm1,newdata)
```

```
##     1
## 4.186
```

```r
# create 1 x 2 panel plot
par(mfrow=c(1,2))
# plot train data with the regression line
plot(trainFaith$waiting,trainFaith$eruptions,pch=19,col="blue",xlab="Waiting",
    ylab="Duration", main = "Train")
lines(trainFaith$waiting,predict(lm1),lwd=3)
# plot test data with the regression line
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue",xlab="Waiting",
    ylab="Duration", main = "Test")
lines(testFaith$waiting,predict(lm1,newdata=testFaith),lwd=3)
```

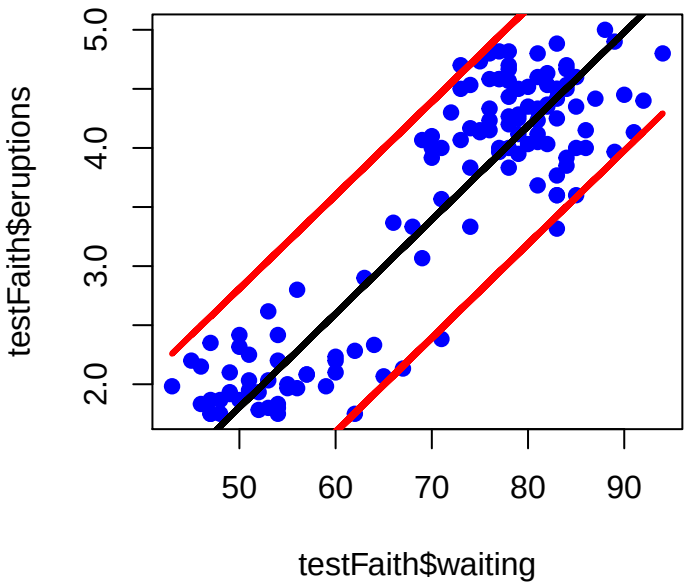



```
# Calculate RMSE on training and test sets
c(trainRMSE = sqrt(sum((lm1$fitted-trainFaith$eruptions)^2)),
    testRMSE = sqrt(sum((predict(lm1,newdata=testFaith)-testFaith$eruptions)^2)))
```

```
## trainRMSE  testRMSE
##  5.824859  5.788547
```

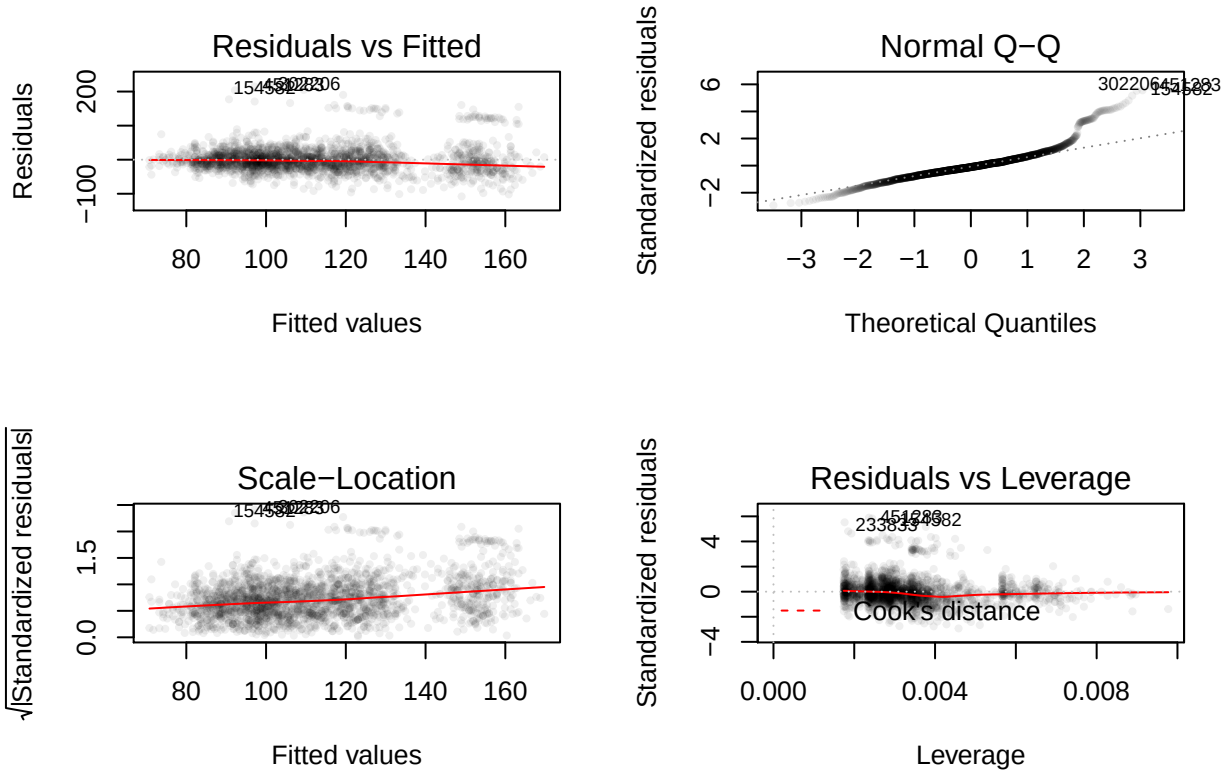
- `pi<-predict(lm, newdata=test, interval="prediction")` = returns 3 columns for `fit` (predicted value, same as before), `lwr` (lower bound of prediction interval), and `upr` (upper bound of prediction interval)
  - `matlines(x, pi, type="l")` = plots three lines, one for the linear fit and two for upper/lower prediction interval bounds

```
# calculate prediction interval
pred1 <- predict(lm1,newdata=testFaith,interval="prediction")
# plot data points (eruptions, waiting)
plot(testFaith$waiting,testFaith$eruptions,pch=19,col="blue")
# plot fit line and prediction interval
matlines(testFaith$waiting,pred1,type="l",,col=c(1,2,2),lty = c(1,1,1), lwd=3)
```



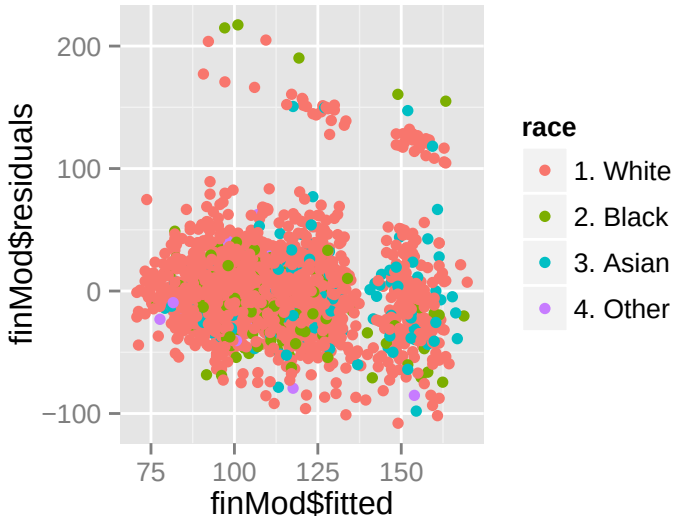
- `lm <- train(y ~ x, method="lm", data=train)` = run linear model on the training data → identical to `lm` function
  - `summary(lm$finalModel)` = returns summary of the linear regression model, which will include coefficients, standard errors, t statistics, and p values → identical to `summary(lm)` for a `lm` object
  - `train(y ~ ., method="lm", data=train)` = run linear model on all predictors in training data
    - multiple predictors (dummy/indicator variables) are created for factor variables
  - `plot(lm$finalModel)` = construct 4 diagnostic plots for evaluating the model
    - ***Note**: more information on these plots can be found at `?plot.lm`*
    - ***Residuals vs Fitted***
    - ***Normal Q-Q***
    - ***Scale-Location***
    - ***Residuals vs Leverage***

```
# create train and test sets
inTrain <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
# fit linear model for age jobclass and education
modFit<- train(wage ~ age + jobclass + education,method = "lm",data=training)
# store final model
finMod <- modFit$finalModel
# set up 2 x 2 panel plot
par(mfrow = c(2, 2))
# construct diagnostic plots for model
plot(finMod,pch=19,cex=0.5,col="#00000010")
```



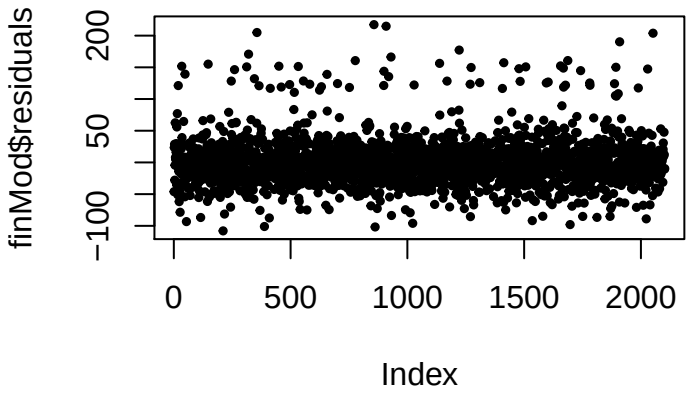
- plotting residuals by fitted values and coloring with a variable not used in the model helps spot a trend in that variable.

```
# plot fitted values by residuals
qplot(finMod$fitted, finMod$residuals, color=race, data=training)
```



- plotting residuals by index (ie; row numbers) can be helpful in showing missing variables
  - `plot(finMod$residuals)` = plot the residuals against index (row number)
  - if there's a trend/pattern in the residuals, it is highly likely that another variable (such as age/time) should be included.
    * residuals should not have relationship to index

```
# plot residual by index
plot(finMod$residuals,pch=19,cex=0.5)
```



- here the residuals increase linearly with the index, and the highest residuals are concentrated in the higher indexes, so there must be a missing variable

Prediction with Trees

- **prediction with trees** = iteratively split variables into groups (effectively constructing decision trees) → produces nonlinear model
  - the classification tree uses interactions between variables → the ultimate groups/leafs may depend on many variables
- the result (tree) is easy to interpret, and generally performs better predictions than regression models when the relationships are ***non-linear***
- transformations less important → monotone transformations (order unchanged, such as log) will produce same splits
- trees can be used for regression problems as well and use RMSE as *measure of impurity*
- however, without proper cross-validation, the model can be ***over-fitted*** (especially with large number of variables) and results may be variable from one run to the next
  - it is also harder to estimate the uncertainty of the model
- `party`, `rpart`, `tree` packages can all build trees

Process

1. start with all variables in one group
2. find the variable that best splits the outcomes into two groups
3. divide data into two groups (*leaves*) based on the split performed (*node*)
4. within each split, find variables to split the groups again
5. continue this process until all groups are sufficiently small/homogeneous/"pure"

Measures of Impurity (Reference)

$$\hat{p}_{mk} = \frac{\sum_i^m \mathbb{1}(y_i = k)}{N_m}$$

- $\hat{p}_m k$ is the probability of the objects in group m to take on the classification k
- N_m is the size of the group
- **Misclassification Error**

  $$1 - \hat{p}_{m\ k(m)}$$

  where $k(m)$ is the most common classification/group
  - 0 = perfect purity
  - 0.5 = no purity
    - ***Note**: it is not 1 here because when $\hat{p}_{m\ k(m)} < 0.5$ or there's not predominant classification for the objects, it means the group should be further subdivided until there's a majority*
- **Gini Index**

  $$\sum_{k \neq k'} \hat{p}_{mk} \times \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} p_{mk}^2$$

  - 0 = perfect purity
  - 0.5 = no purity

- **Deviance**

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log_e \hat{p}_{mk}$$

  - 0 = perfect purity
  - 1 = no purity

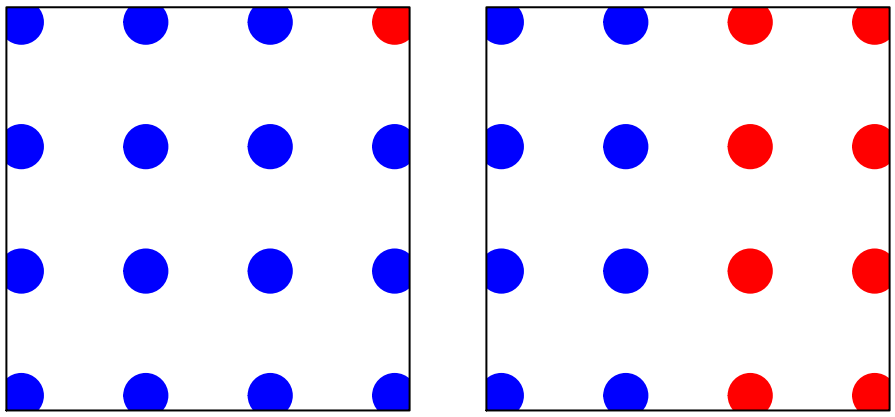
- **Information Gai**

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log_2 \hat{p}_{mk}$$

  - 0 = perfect purity
  - 1 = no purity

- ***example***

```
# set margin and seed
par(mar=c(1,1,1,1), mfrow = c(1, 2)); set.seed(1234);
# simulate data
x = rep(1:4,each=4); y = rep(1:4,4)
# plot first scenario
plot(x,y,xaxt="n",yaxt="n",cex=3,col=c(rep("blue",15),rep("red",1)),pch=19)
# plot second scenario
plot(x,y,xaxt="n",yaxt="n",cex=3,col=c(rep("blue",8),rep("red",8)),pch=19)
```



- left graph
  - **Misclassification:** $\frac{1}{16} = 0.06$
  - **Gini:** $1 - [(\frac{1}{16})^2 + (\frac{15}{16})^2] = 0.12$
  - **Information:**$-[\frac{1}{16} \times \log_2(\frac{1}{16}) + \frac{15}{16} \times \log_2(\frac{1}{16})] = 0.34$
- right graph
  - **Misclassification:** $\frac{8}{16} = 0.5$
  - **Gini:** $1 - [(\frac{8}{16})^2 + (\frac{8}{16})^2] = 0.5$
  - **Information:**$-[\frac{8}{16} \times \log_2(\frac{8}{16}) + \frac{8}{16} \times \log_2(\frac{8}{16})] = 1$

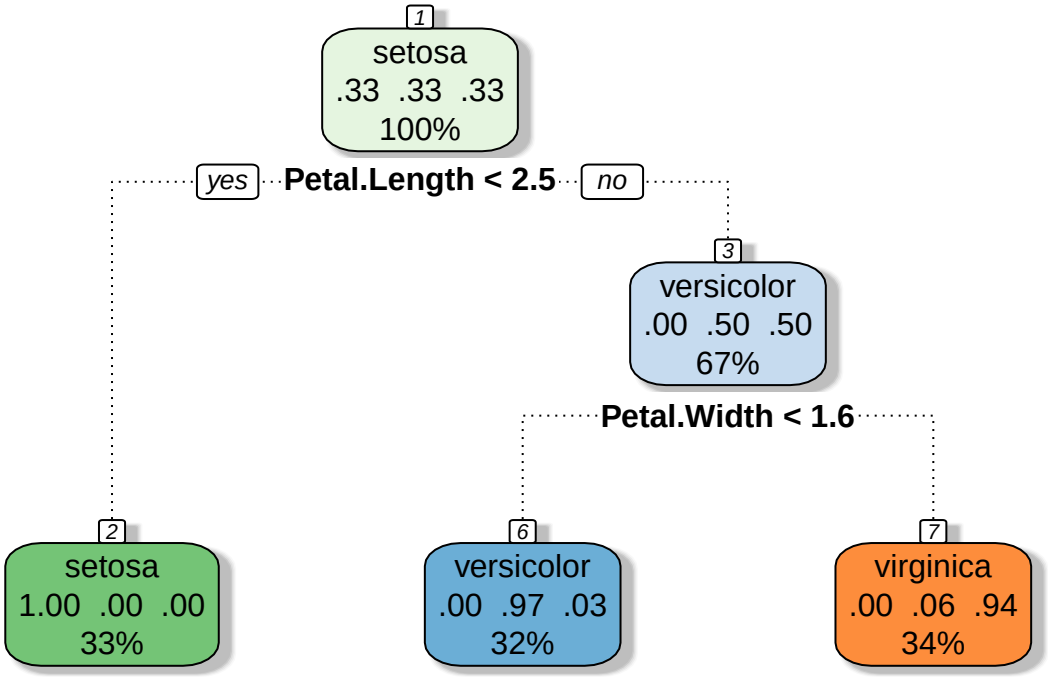
Constructing Trees with caret Package

- `tree<-train(y ~ ., data=train, method="rpart")` = constructs trees based on the outcome and predictors
  - produces an `rpart` object, which can be used to `predict` new/test values
  - `print(tree$finalModel)` = returns text summary of all nodes/splits in the tree constructed
- `plot(tree$finalModel, uniform=TRUE)` = plots the classification tree with all nodes/splits
  - [`rattle` package] `fancyRpartPlot(tree$finalModel)` = produces more readable, better formatted classification tree diagrams
  - each split will have the condition/node in bold and the splits/leafs on the left and right sides following the "yes" or "no" indicators
    - "yes" → go left
    - "no" → go right

```
# load iris data set
data(iris)
# create test/train data sets
inTrain <- createDataPartition(y=iris$Species,p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
# fit classification tree as a model
modFit <- train(Species ~ .,method="rpart",data=training)
# print the classification tree
print(modFit$finalModel)
```

```
## n= 105
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 105 70 setosa (0.33333333 0.33333333 0.33333333)
##   2) Petal.Length< 2.45 35  0 setosa (1.00000000 0.00000000 0.00000000) *
##   3) Petal.Length>=2.45 70 35 versicolor (0.00000000 0.50000000 0.50000000)
##     6) Petal.Width< 1.65 34  1 versicolor (0.00000000 0.97058824 0.02941176) *
##     7) Petal.Width>=1.65 36  2 virginica (0.00000000 0.05555556 0.94444444) *
```

```
# plot the classification tree
# Note: Need to install Gtk+ 2.x before using rattle. On OS/X:
# brew install gtk+
# I should just need to run the following:
#p_load("rattle")
# But I actually need to run this, because apparently the dependencies aren't loading:
p_load("RGtk2")
p_load("rpart")
p_load("rpart.plot")
p_load("rattle")
rattle::fancyRpartPlot(modFit$finalModel)
```




```
# predict on test values
predict(modFit,newdata=testing)
```

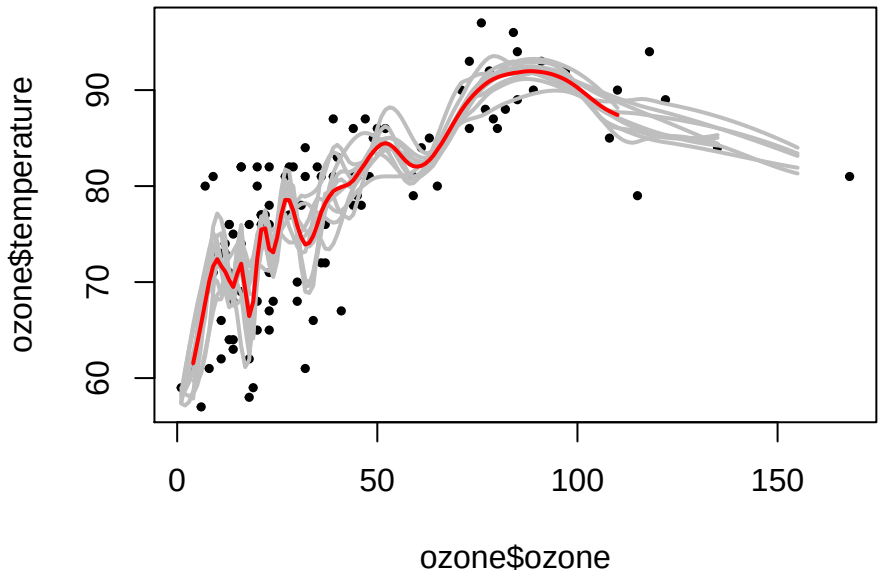
```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     versicolor versicolor versicolor
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] virginica  virginica  virginica  virginica  virginica  virginica
## [37] versicolor virginica  virginica  versicolor versicolor virginica
## [43] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

Bagging

- **bagging** = bootstrap aggregating
  - resample training data set (with replacement) and recalculate predictions
  - average the predictions together or majority vote
  - more information can be found here
- averaging multiple complex models have ***similar bias*** as each of the models on its own, and ***reduced variance*** because of the average
- most useful for non-linear models
- ***example***
  - `loess(y ~ x, data=train, span=0.2)` = fits a smooth curve to data

* `span=0.2` = controls how smooth the curve should be

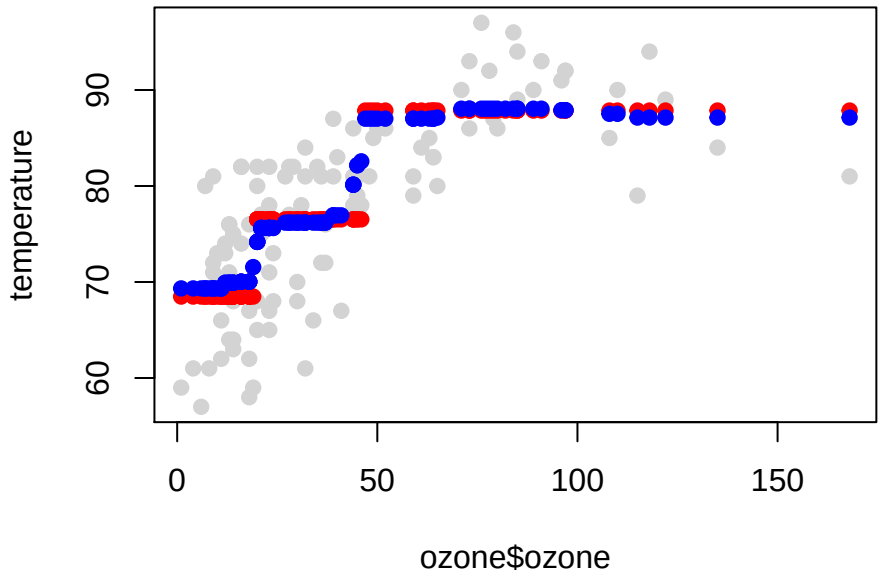
```
# load data
p_load("ElemStatLearn")
data(ozone,package="ElemStatLearn")
# reorder rows based on ozone variable
ozone <- ozone[order(ozone$ozone),]
# create empty matrix
ll <- matrix(NA,nrow=10,ncol=155)
# iterate 10 times
for(i in 1:10){
    # create sample from data with replacement
    ss <- sample(1:dim(ozone)[1],replace=T)
    # draw sample from the dataa and reorder rows based on ozone
    ozone0 <- ozone[ss,]; ozone0 <- ozone0[order(ozone0$ozone),]
    # fit loess function through data (similar to spline)
    loess0 <- loess(temperature ~ ozone,data=ozone0,span=0.2)
    # prediction from loess curve for the same values each time
    ll[i,] <- predict(loess0,newdata=data.frame(ozone=1:155))
}
# plot the data points
plot(ozone$ozone,ozone$temperature,pch=19,cex=0.5)
# plot each prediction model
for(i in 1:10){lines(1:155,ll[i,],col="grey",lwd=2)}
# plot the average in red
lines(1:155,apply(ll,2,mean),col="red",lwd=2)
```



Bagging Algorithms

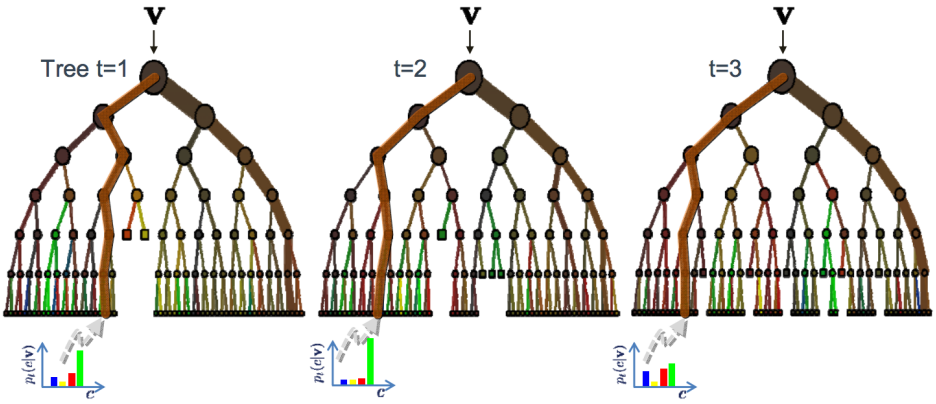
- in the `caret` package, there are three options for the `train` function to perform bagging
  - `bagEarth` - Bagged MARS (documentation)
  - `treebag` - Bagged CART (documentation)
  - `bagFDA` - Bagged Flexible Discriminant Analysis (documentation)
- alternatively, custom `bag` functions can be constructed (documentation)
  - `bag(predictors, outcome, B=10, bagControl(fit, predict, aggregate))` = define and execute custom bagging algorithm
    - `B=10` = iterations/resampling to perform
    - `bagControl()` = controls for how the bagging should be executed
      - `fit=ctreeBag$fit` = the model ran on each resampling of data
      - `predict=ctreeBag$predict` = how predictions should be calculated from each model
      - `aggregate=ctreeBag$aggregate` = how the prediction models should be combined/averaged
- ***example***

```
# load relevant package and data
p_load("party")
data(ozone,package="ElemStatLearn")
# reorder rows based on ozone variable
ozone <- ozone[order(ozone$ozone),]
# extract predictors
predictors <- data.frame(ozone=ozone$ozone)
# extract outcome
temperature <- ozone$temperature
# run bagging algorithm
treebag <- bag(predictors, temperature, B = 10,
               # custom bagging function
               bagControl = bagControl(fit = ctreeBag$fit,
                                       predict = ctreeBag$pred,
                                       aggregate = ctreeBag$aggregate))
# plot data points
plot(ozone$ozone,temperature,col='lightgrey',pch=19)
# plot the first fit
points(ozone$ozone,predict(treebag$fits[[1]]$fit,predictors),pch=19,col="red")
# plot the aggregated predictions
points(ozone$ozone,predict(treebag,predictors),pch=19,col="blue")
```


temperature
60
70
80
90
0
50
100
150
ozone$ozone

Random Forest

- **random forest** = extension of bagging on classification/regression trees
  - one of the most used/accurate algorithms along with boosting



- **process**
  - bootstrap samples from training data (with replacement)
  - split and bootstrap variables
  - grow trees (repeat split/bootstrap) and vote/average final trees
- **drawbacks**
  - algorithm can be slow (process large number of trees)
  - hard to interpret (large numbers of splits and nodes)
  - over-fitting (difficult to know which tree is causing over-fitting)
  - ***Note**: it is extremely important to use cross validation when running random forest algorithms*

R Commands and Examples

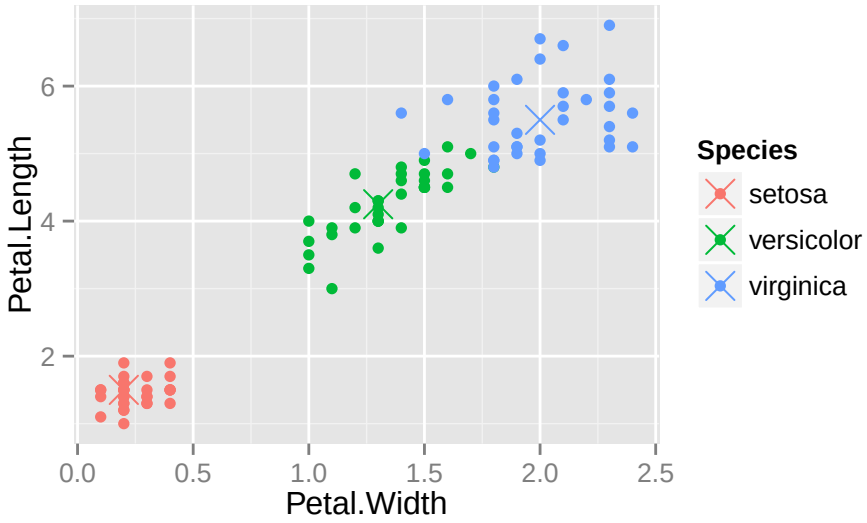
- `rf<-train(outcome ~ ., data=train, method="rf", prox=TRUE, ntree=500)` = runs random forest algorithm on the training data against all predictors
  - ***Note**: random forest algorithm automatically bootstrap by default, but it is still important to have train/test/validation split to verify the accuracy of the model*
  - `prox=TRUE` = the proximity measures between observations should be calculated (used in functions such as `classCenter()` to find center of groups)
    - `rf$finalModel$prox` = returns matrix of proximities
  - `ntree=500` = specify number of trees that should be constructed
  - `do.trace=TRUE` = prints logs as the trees are being built → useful by indicating progress to user
  - ***Note**: `randomForest()` function can be used to perform random forest algorithm (syntax is the same as `train`) and is much faster*
- `getTree(rf$finalModel, k=2)` = return specific tree from random forest model

- `classCenters(predictors, outcome, proximity, nNbr)` = return computes the cluster centers using the `nNbr` nearest neighbors of the observations
  - `prox = rf$finalModel$prox` = proximity matrix from the random forest model
  - `nNbr` = number of nearest neighbors that should be used to compute cluster centers
- `predict(rf, test)` = apply the random forest model to test data set
  - `confusionMatrix(predictions, actualOutcome)` = tabulates the predictions of the model against the truths
    - ***Note**: this is generally done for the validation data set using the model built from training*
- ***example***

```
p_load("randomForest")
# load data
data(iris)
# create train/test data sets
inTrain <- createDataPartition(y=iris$Species,p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
# apply random forest
modFit <- train(Species~ .,data=training,method="rf",prox=TRUE)
# return the second tree (first 6 rows)
head(getTree(modFit$finalModel,k=2))
```

```
##   left daughter right daughter split var split point status prediction
## 1             2              3         4        0.70      1          0
## 2             0              0         0        0.00     -1          1
## 3             4              5         4        1.75      1          0
## 4             6              7         3        5.30      1          0
## 5             0              0         0        0.00     -1          3
## 6             8              9         3        4.95      1          0
```

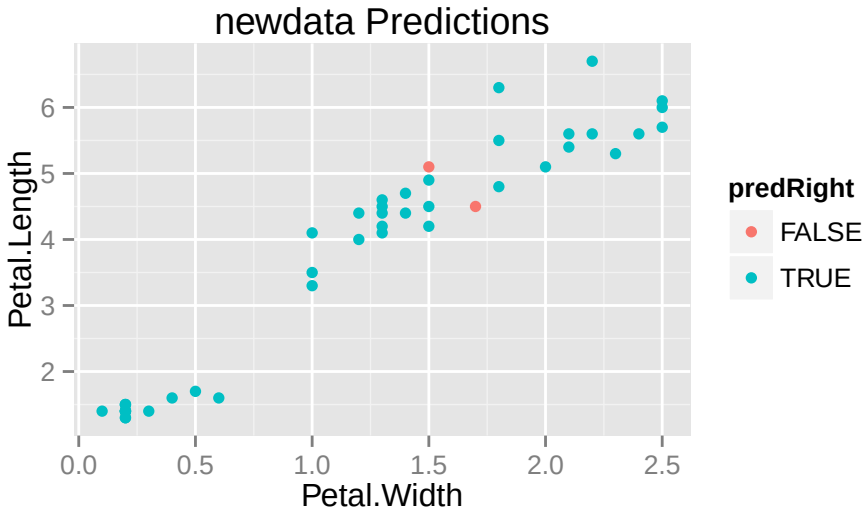
```
# compute cluster centers
irisP <- classCenter(training[,c(3,4)], training$Species, modFit$finalModel$prox)
# convert irisP to data frame and add Species column
irisP <- as.data.frame(irisP); irisP$Species <- rownames(irisP)
# plot data points
p <- qplot(Petal.Width, Petal.Length, col=Species,data=training)
# add the cluster centers
p + geom_point(aes(x=Petal.Width,y=Petal.Length,col=Species),size=5,shape=4,data=irisP)
```




```
# predict outcome for test data set using the random forest model
pred <- predict(modFit,testing)
# logic value for whether or not the rf algorithm predicted correctly
testing$predRight <- pred==testing$Species
# tabulate results
table(pred,testing$Species)
```

```
##
## pred         setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         15         2
##   virginica       0          0        13
```

```
# plot data points with the incorrect classification highlighted
qplot(Petal.Width,Petal.Length,colour=predRight,data=testing,main="newdata Predictions")
```



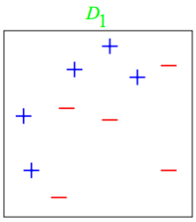
Boosting

- **boosting** = one of the most widely used and accurate prediction models, along with random forest
- boosting can be done with any set of classifiers, and a well-known approach is gradient boosting
- more detail tutorial can be found here
- **process**: take a group of weak predictors → weight them and add them up → result in a stronger predictor
  - start with a set of classifiers $h_1, \ldots, h_k$
    - examples: all possible trees, all possible regression models, all possible cutoffs (divide data into different parts)
  - calculate a weighted sum of classifiers as the prediction value

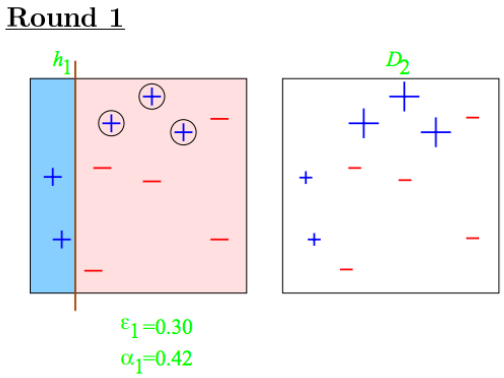
$$f(x) = \sum_i \alpha_i h_i(x)$$

    where α_i = coefficient/weight and $h_i(x)$ = value of classifier
    - goal = minimize error (on training set)
    - select one h at each step (iterative)
    - calculate weights based on errors
    - up-weight missed classifications and select next h

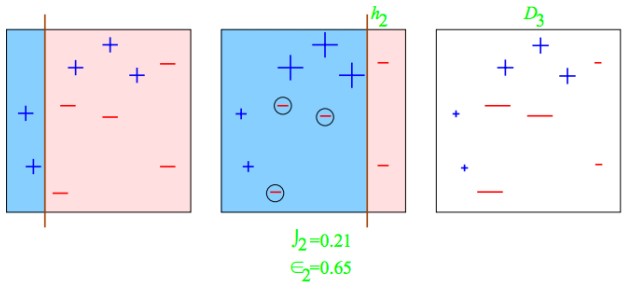
- ***example***



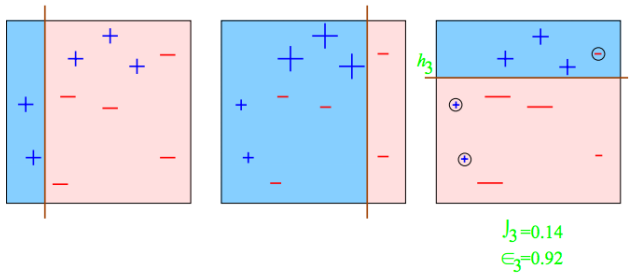
- we start with space with **blue** + and **red -** and the goal is to classify all the object correctly
- only straight lines will be used for classification



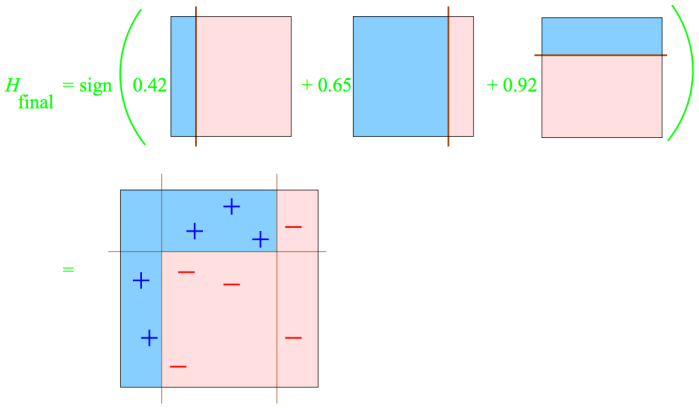
Round 2



Round 3



Final Hypothesis



- from the above, we can see that a group of weak predictors (lines in this case), can be combined and weighed to become a much stronger predictor

R Commands and Examples

- `gbm <- train(outcome ~ variables, method="gbm", data=train, verbose=F)` = run boosting model on the given data
  - options for `method` for boosting
    - `gbm` - boosting with trees
    - `mboost` - model based boosting
    - `ada` - statistical boosting based on additive logistic regression
    - `gamBoost` for boosting generalized additive models
    - ***Note**: differences between packages include the choice of basic classification functions and combination rules*
- `predict` function can be used to apply the model to test data, similar to the rest of the algorithms in `caret` package
- ***example***

```
p_load("gbm")
# load data
data(Wage)
# remove log wage variable (we are trying to predict wage)
Wage <- subset(Wage,select=-c(logwage))
# create train/test data sets
inTrain <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
training <- Wage[inTrain,]; testing <- Wage[-inTrain,]
# run the gbm model
modFit <- train(wage ~ ., method="gbm",data=training,verbose=FALSE)
# print model summary
print(modFit)
```

```
## Stochastic Gradient Boosting
##
## 2102 samples
##   10 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  RMSE      Rsquared   RMSE SD   Rsquared SD
##   1                   50      35.64604  0.3316490  1.501233  0.02402838
##   1                  100      34.93971  0.3436530  1.497245  0.02382398
##   1                  150      34.84303  0.3453109  1.511559  0.02428236
##   2                   50      34.95187  0.3453277  1.474202  0.02475740
##   2                  100      34.76109  0.3482782  1.456151  0.02414784
##   2                  150      34.80180  0.3466793  1.448168  0.02353360
##   3                   50      34.81083  0.3478153  1.489981  0.02434105
##   3                  100      34.82856  0.3458632  1.472464  0.02449900
##   3                  150      35.00002  0.3401647  1.465331  0.02446706
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
```

```
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were n.trees = 100,
##  interaction.depth = 2, shrinkage = 0.1 and n.minobsinnode = 10.
```

Model Based Prediction

- **model based prediction** = assumes the data follow a probabilistic model/distribution and use *Bayes' theorem* to identify optimal classifiers/variables
  - can potentially take advantage of structure of the data
  - could help reduce computational complexity (reduce variables)
  - can be reasonably accurate on real problems
- this approach does make ***additional assumptions*** about the data, which can lead to model failure/reduced accuracy if they are too far off
- ***goal*** = build parameter-based model (based on probabilities) for conditional distribution $P(Y = k \mid X = x)$, or the probability of the outcome Y is equal to a particular value k given a specific set of predictor variables x
  - ***Note**: X is the data for the model (observations for all predictor variables), which is also known as the **design matrix***
- **typical approach/process**
  1. start with the quantity $P(Y = k \mid X = x)$
  2. apply *Bayes' Theorem* such that

$$P(Y = k \mid X = x) = \frac{P(X = x \mid Y = k)P(Y = k)}{\sum_{\ell=1}^{K} P(X = x \mid Y = \ell)P(Y = \ell)}$$

     where the denominator is simply the sum of probabilities for the predictor variables are the set specified in x for all outcomes of Y
  3. assume the term $P(X = x \mid Y = k)$ in the numerator follows a parameter-based probability distribution, or $f_k(x)$
     - common choice = ***Gaussian distribution***

$$f_k(x) = \frac{1}{\sigma_k\sqrt{2\pi}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

  4. assume the probability for the outcome Y to take on value of k, or $P(Y = k)$, is determined from the data to be some known quantity π_k
     - ***Note**: $P(Y = k)$ is known as the **prior probability***
  5. so the quantity $P(Y = k \mid X = x)$ can be rewritten as

$$P(Y = k \mid X = x) = \frac{f_k(x)\pi_k}{\sum_{\ell=1}^{K} f_\ell(x)\pi_\ell}$$

  6. estimate the parameters (μ_k, σ_k^2) for the function $f_k(x)$ from the data
  7. calculate $P(Y = k \mid X = x)$ using the parameters
  8. the outcome Y is where the value of $P(Y = k \mid X = x)$ is the highest
- prediction models that leverage this approach
  - ***linear discriminant analysis*** = assumes $f_k(x)$ is multivariate Gaussian distribution with **same** covariance for each predictor variables
    - effectively drawing lines through "covariate space"
  - ***quadratic discriminant analysis*** = assumes $f_k(x)$ is multivariate Gaussian distribution with **different** covariance for predictor variables
    - effectively drawing curves through "covariate space"
  - ***normal mixture modeling*** = assumes more complicated covariance matrix for the predictor variables

- ***naive Bayes*** = assumes independence between predictor variables/features for model building (covariance = 0)
  - ***Note:*** *this may be an incorrect assumption but it helps to reduce computational complexity and may still produce a useful result*

Linear Discriminant Analysis

- to compare the probability for outcome $Y = k$ versus probability for outcome $Y = k$, we can look at the ratio of

$$\frac{P(Y = k \mid X = x)}{P(Y = j \mid X = x)}$$

- take the **log** of the ratio and apply Bayes' Theorem, we get

$$\log \frac{P(Y = k \mid X = x)}{P(Y = j \mid X = x)} = \log \frac{f_k(x)}{f_j(x)} + \log \frac{\pi_k}{\pi_j}$$

  which is effectively the log ratio of probability density functions plus the log ratio of prior probabilities

  - ***Note:*** log = ***monotone*** *transformation, which means taking the* log *of a quantity does not affect implication of the ratio since th* log(*ratio*) *is* directly correlated *with ratio*

- if we substitute $f_k(x)$ and $f_l(x)$ with Gaussian probability density functions

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

  so the ratio can be simplified to

$$\log \frac{P(Y = k \mid X = x)}{P(Y = j \mid X = x)} = \log \frac{\pi_k}{\pi_j} - \frac{1}{2}(\mu_k + \mu_j)^T \Sigma^{-1}(\mu_k + \mu_j) + x^T \Sigma^{-1}(\mu_k - \mu_j)$$

  where Σ^{-1} = covariance matrix for the predictor variables, x^T = set of predictor variables, and μ_k / μ_j = mean of k, j respectively

- as annotated above, the log-ratio is effectively an equation of a line for a set of predictor variables $x \rightarrow$ the first two terms are constants and the last term is in the form of $X\beta$

  - ***Note:*** *the lines are also known as* ***decision boundaries***

- therefore, we can classify values based on ***which side of the line*** the value is located (k vs j)

- **discriminant functions** are used to determine value of k, the functions are in the form of

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k \Sigma^{-1} \mu_k + log(\mu_k)$$

  - plugging in the set of predictor variables, x^T, into the discriminant function, we can find the value of k that ***maximizes*** the function $\delta_k(x)$
  - the terms of the discriminant function can be estimated using maximum likelihood

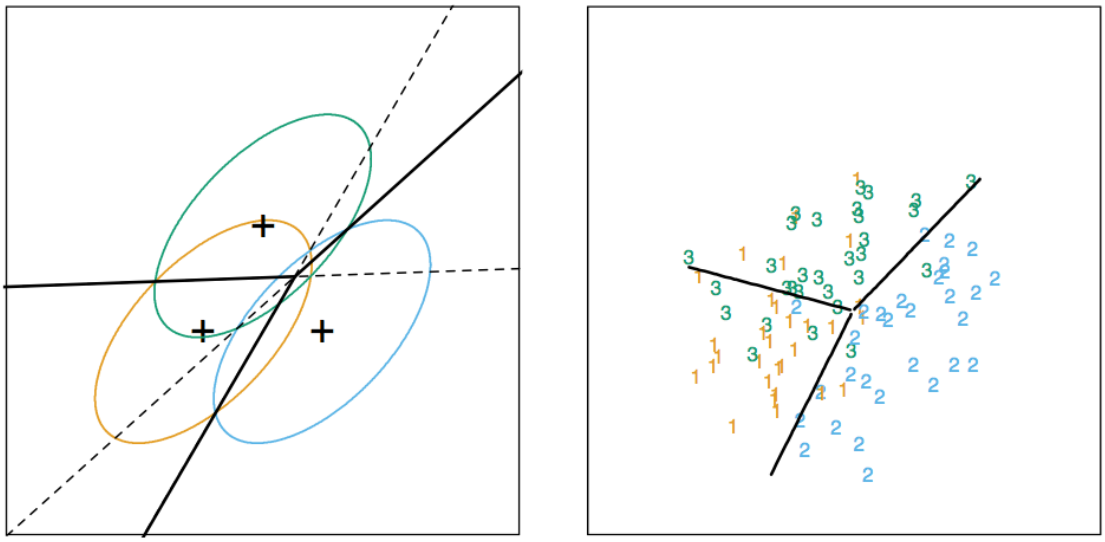
- the predicted value for the outcome is therefore $\hat{Y}(x) = argmax_k \delta_k(x)$

- ***example***

  - classify a group of values into 3 groups using 2 variables (x, y coordinates)
  - 3 lines are draw to split the data into 3 Gaussian distributions
    - each line splits the data into two groups $\rightarrow$ 1 vs 2, 2 vs 3, 1 vs 3

    - each side of the line represents a region where the probability of one group (1, 2, or 3) is the highest

  - the result is represented in the the following graph



- **R Commands**
  - `lda<-train(outcome ~ predictors, data=training, method="lda")` = constructs a linear discriminant analysis model on the predictors with the provided training data
  - `predict(lda, test)` = applies the LDA model to test data and return the prediction results in data frame
  - ***example: caret package***

```
# load data
data(iris)
# create training and test sets
inTrain <- createDataPartition(y=iris$Species,p=0.7, list=FALSE)
training <- iris[inTrain,]
testing <- iris[-inTrain,]
# run the linear discriminant analysis on training data
lda <- train(Species ~ .,data=training,method="lda")
# predict test outcomes using LDA model
pred.lda <- predict(lda,testing)
# print results
pred.lda
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     versicolor versicolor versicolor
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] virginica  virginica  virginica  virginica  virginica  virginica
## [37] virginica  virginica  virginica  virginica  virginica  virginica
## [43] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

Naive Bayes

- for predictors $X_1, \ldots, X_m$, we want to model $P(Y = k \mid X_1, \ldots, X_m)$
- by applying *Bayes' Theorem*, we get

$$P(Y = k \mid X_1, \ldots, X_m) = \frac{\pi_k P(X_1, \ldots, X_m \mid Y = k)}{\sum_{\ell=1}^{K} P(X_1, \ldots, X_m \mid Y = k)\pi_\ell}$$

- since the denominator is just a sum (constant), we can rewrite the quantity as

$$P(Y = k \mid X_1, \ldots, X_m) \propto \pi_k P(X_1, \ldots, X_m \mid Y = k)$$

  or the probability is **proportional to** the numerator
  - ***Note**: maximizing the numerator is the same as maximizing the ratio*

- $\pi_k P(X_1, \ldots, X_m \mid Y = k)$ can be rewritten as

$$\begin{aligned}\pi_k P(X_1, \ldots, X_m \mid Y = k) &= \pi_k P(X_1 \mid Y = k) P(X_2, \ldots, X_m \mid X_1, Y = k) \\ &= \pi_k P(X_1 \mid Y = k) P(X_2 \mid X_1, Y = k) P(X_3, \ldots, X_m \mid X_1, X_2, Y = k) \\ &= \pi_k P(X_1 \mid Y = k) P(X_2 \mid X_1, Y = k) \ldots P(X_m \mid X_1 \ldots, X_{m-1}, Y = k)\end{aligned}$$

  where each variable has its own probability term that depends on all the terms before it
  - this is effectively indicating that each of the predictors may be dependent on other predictors

- however, if we make the assumption that all predictor variables are **independent** to each other, the quantity can be simplified to

$$\pi_k P(X_1, \ldots, X_m \mid Y = k) \approx \pi_k P(X_1 \mid Y = k) P(X_2 \mid Y = k) \ldots P(X_m \mid , Y = k)$$

  which is effectively the product of the prior probability for k and the probability of variables $X_1, \ldots, X_m$ given that $Y = k$
  - ***Note**: the assumption is naive in that it is unlikely the predictors are completely independent of each other, but this model still produces **useful** results particularly with **large number of binary/categorical variables***
    - text and document classification usually require large quantities of binary and categorical features

- **R Commands**
  - `nb <- train(outcome ~ predictors, data=training, method="nb")` = constructs a naive Bayes model on the predictors with the provided training data
  - `predict(nb, test)` = applies the naive Bayes model to test data and return the prediction results in data frame
  - ***example: `caret` package***

```
p_load("klaR")
# using the same data from iris, run naive Bayes on training data
nb <- train(Species ~ ., data=training,method="nb")
# predict test outcomes using naive Bayes model
pred.nb <- predict(nb,testing)
# print results
pred.nb
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     versicolor versicolor versicolor
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] virginica  virginica  virginica  virginica  virginica  virginica
## [37] virginica  virginica  virginica  virginica  virginica  virginica
## [43] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

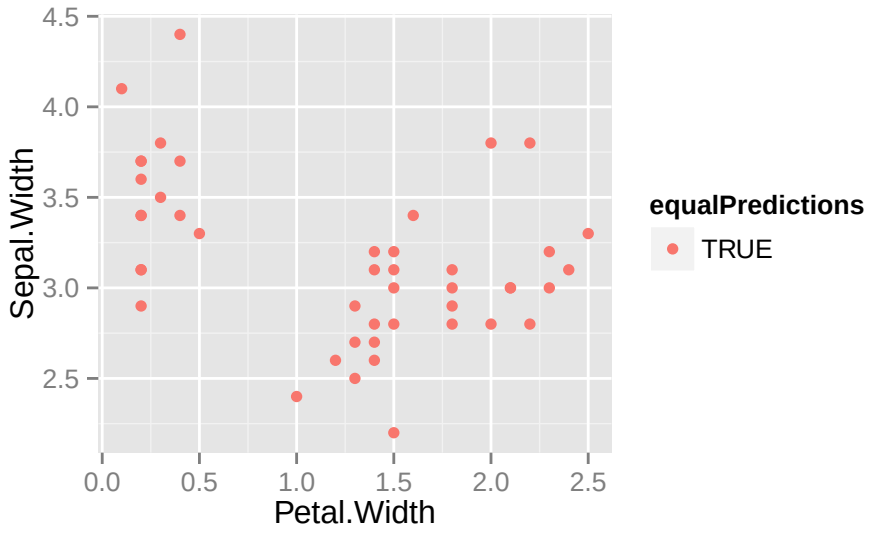
Compare Results for LDA and Naive Bayes

- linear discriminant analysis and naive Bayes generally produce similar results for small data sets
- for our example data from `iris` data set, we can compare the prediction the results from the two models

```
# tabulate the prediction results from LDA and naive Bayes
table(pred.lda,pred.nb)
```

```
##             pred.nb
## pred.lda     setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         15         0
##   virginica       0          0        15
```

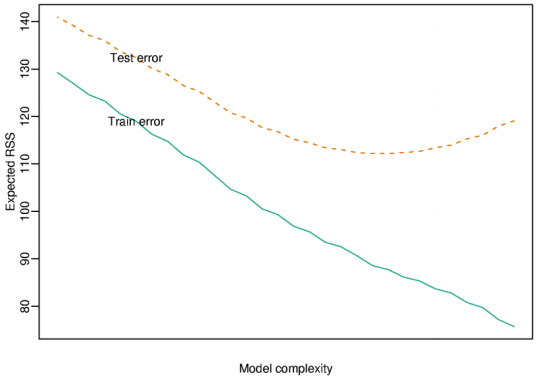
```
# create logical variable that returns TRUE for when predictions from the two models match
equalPredictions <- (pred.lda==pred.nb)
# plot the comparison
qplot(Petal.Width,Sepal.Width,colour=equalPredictions,data=testing)
```



- as we can see from above, only one data point, which is located inbetween the two classes is predicted differently by the two models

Model Selection

- the general behavior of the errors of the training and test sets are as follows
  - as the number of predictors used increases (or model complexity), the error for the prediction model on *training* set **always decreases**
  - the error for the prediction model on *test* set **decreases first and then increases** as number of predictors used approaches the total number of predictors available



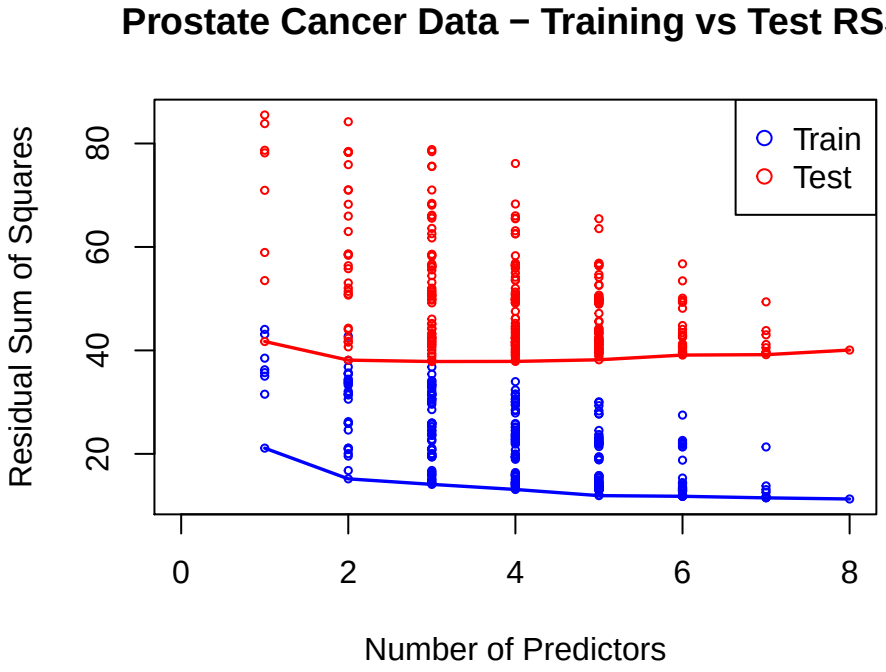
- this is expected since as more predictors used, the model is more likely to *overfit* the training data
- **goal** in selecting models = ***avoid overfitting*** on training data and ***minimize error*** on test data
- ***approaches***
  - split samples
  - decompose expected prediction error
  - hard thresholding for high-dimensional data
  - regularization for regression
    - ridge regression
    - lasso regression
- ***problems***
  - time/computational complexity limitations
  - high dimensional

Example: Training vs Test Error for Combination of Predictors

- ***Note****: the code for this example comes from Hector Corrada Bravo's Practical Machine Learning Course*
- to demonstrate the behavior of training and test errors, the `prostate` dataset from *Elements of Statistical Learning* is used
- all combinations of predictors are used to produce prediction models, and Residual Squared Error (RSS) is calculated for all models on both the training and test sets

```r
# load data and set seed
data(prostate); set.seed(1)
# define outcome y and predictors x
covnames <- names(prostate[-(9:10)])
y <- prostate$lpsa; x <- prostate[,covnames]
# create test set predictors and outcomes
train.ind <- sample(nrow(prostate), ceiling(nrow(prostate))/2)
y.test <- prostate$lpsa[-train.ind]; x.test <- x[-train.ind,]
# create training set predictors and outcomes
y <- prostate$lpsa[train.ind]; x <- x[train.ind,]
# p = number of predictors
p <- length(covnames)
# initialize the list of residual sum squares
rss <- list()
# loop through each combination of predictors and build models
for (i in 1:p) {
    # compute matrix for p choose i predictors for i = 1...p (creates i x p matrix)
    Index <- combn(p,i)
    # calculate residual sum squares of each combination of predictors
    rss[[i]] <- apply(Index, 2, function(is) {
        # take each combination (or column of Index matrix) and create formula for regression
        form <- as.formula(paste("y~", paste(covnames[is], collapse="+"), sep=""))
        # run linear regression with combination of predictors on training data
        isfit <- lm(form, data=x)
        # predict outcome for all training data points
        yhat <- predict(isfit)
        # calculate residual sum squares for predictions on training data
        train.rss <- sum((y - yhat)^2)
        # predict outcome for all test data points
        yhat <- predict(isfit, newdata=x.test)
        # calculate residual sum squares for predictions on test data
        test.rss <- sum((y.test - yhat)^2)
        # store each pair of training and test residual sum squares as a list
        c(train.rss, test.rss)
    })
}
# set up plot with labels, title, and proper x and y limits
plot(1:p, 1:p, type="n", ylim=range(unlist(rss)), xlim=c(0,p),
    xlab="Number of Predictors", ylab="Residual Sum of Squares",
    main="Prostate Cancer Data - Training vs Test RSS")
# add data points for training and test residual sum squares
for (i in 1:p) {
    # plot training residual sum squares in blue
    points(rep(i, ncol(rss[[i]])), rss[[i]][1, ], col="blue", cex = 0.5)
    # plot test residual sum squares in red
    points(rep(i, ncol(rss[[i]])), rss[[i]][2, ], col="red", cex = 0.5)
}
# find the minimum training RSS for each combination of predictors
minrss <- sapply(rss, function(x) min(x[1,]))
# plot line through the minimum training RSS data points in blue
lines((1:p), minrss, col="blue", lwd=1.7)
# find the minimum test RSS for each combination of predictors
minrss <- sapply(rss, function(x) min(x[2,]))
```

```r
# plot line through the minimum test RSS data points in blue
lines((1:p), minrss, col="red", lwd=1.7)
# add legend
legend("topright", c("Train", "Test"), col=c("blue", "red"), pch=1)
```



- from the above, we can clearly that test RSS error approaches the minimum at around 3 predictors and increases slightly as more predictors are used

Split Samples

- the best method to pick predictors/model is to split the given data into different test sets
- **process**
  1. divide data into training/test/validation sets (60 - 20 - 20 split)
  2. train all competing models on the training data
  3. apply the models on validation data and choose the best performing model
  4. re-split data into training/test/validation sets and repeat steps 1 to 3
  5. apply the overall best performing model on test set to appropriately assess performance on new data
- **common problems**
  - limited data = if not enough data is available, it may not be possible to produce a good model fit after splitting the data into 3 sets
  - computational complexity = modeling with all subsets of models can be extremely taxing in terms of computations, especially when a large number of predictors are available

Decompose Expected Prediction Error

- the outcome Y_i can be modeled by
$$Y_i = f(X_i) + \epsilon_i$$
where ϵ_i = error term
- the **expected prediction error** is defined as
$$EPE(\lambda) = E\left[\left(Y - \hat{f}_\lambda(X)\right)^2\right]$$
where λ = specific set of tuning parameters
- estimates from the model constructed with training data can be denoted as $\hat{f}_\lambda(x^*)$ where $X = x^*$ is the new data point that we would like to predict at
- the expected prediction error is as follows
$$E\left[\left(Y - \hat{f}_\lambda(x^*)\right)^2\right] = \sigma^2 + \left(E[\hat{f}_\lambda(x^*)] - f(x^*)\right)^2 + E\left[\hat{f}_\lambda(x^*) - E[\hat{f}_\lambda(x^*)]\right]^2$$
$$= \text{Irreducible Error} + \text{Bias}^2 + \text{Variance}$$
  - **goal of prediction model** = minimize overall expected prediction error
  - irreducible error = noise inherent to the data collection process → cannot be reduced
  - bias/variance = can be traded in order to find optimal model (least error)

Hard Thresholding

- if there are more predictors than observations (high-dimensional data), linear regressions will only return coefficients for some of the variables because there's not enough data to estimate the rest of the parameters
  - conceptually, this occurs because the design matrix that the model is based on cannot be inverted
  - ***Note**: ridge regression can help address this problem*
- **hard thresholding** can help estimate the coefficients/model by taking subsets of predictors and building models
- **process**
  - model the outcome as
$$Y_i = f(X_i) + \epsilon_i$$
where ϵ_i = error term
  - assume the prediction estimate has a linear form
$$\hat{f}_\lambda(x) = x'\beta$$
where only λ coefficients for the set of predictors x are ***nonzero***
  - after setting the value of λ, compute models using all combinations of λ variables to find which variables' coefficients should be set to be zero
- **problem**
  - computationally intensive
- ***example***
  - as we can see from the results below, some of the coefficients have values of `NA`

```r
# load prostate data
data(prostate)
# create subset of observations with 10 variables
small = prostate[1:5,]
# print linear regression
lm(lpsa ~ .,data =small)
```

```
##
## Call:
## lm(formula = lpsa ~ ., data = small)
##
## Coefficients:
## (Intercept)       lcavol      lweight          age         lbph
##     9.60615      0.13901     -0.79142      0.09516           NA
##         svi          lcp      gleason        pgg45    trainTRUE
##          NA           NA     -2.08710           NA           NA
```

Regularized Regression Concept (Resource)

- **regularized regression** = fit a regression model and adjust for the large coefficients in attempt to help with bias/variance trade-off or model selection
  - when running regressions unconstrained (without specifying any criteria for coefficients), the model may be susceptible to high variance (coefficients explode → very large values) if there are variables that are highly correlated
  - controlling/regularizing coefficients may slightly ***increase bias*** (lose a bit of prediction capability) but will ***reduce variance*** and improve the prediction error
  - however, this approach may be very demanding computationally and generally does not perform as well as random forest/boosting
- **Penalized Residual Sum of Squares (PRSS)** is calculated by adding a penalty term to the prediction squared error

$$PRSS(\beta) = \sum_{j=1}^{n}(Y_j - \sum_{i=1}^{m}\beta_{1i}X_{ij})^2 + P(\lambda;\beta)$$

  - penalty shrinks coefficients if their values become too large
  - penalty term is generally used to reduce complexity and variance for the model, while respecting the structure of the data/relationship
- ***example: co-linear variables***
  - given a linear model,

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

    where X_1 and X_2 are nearly perfectly correlated (co-linear)
  - the model can then be approximated by this model by omitting X_2, so the model becomes

$$Y = \beta_0 + (\beta_1 + \beta_2)X_1 + \epsilon$$

  - with the above model, we can get a good estimate of Y
    - the estimate of Y will be biased
    - but the variance of the prediction may be reduced

Regularized Regression - Ridge Regression

- the penalized residual sum of squares (PRSS) takes the form of

$$PRSS(\beta_j) = \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

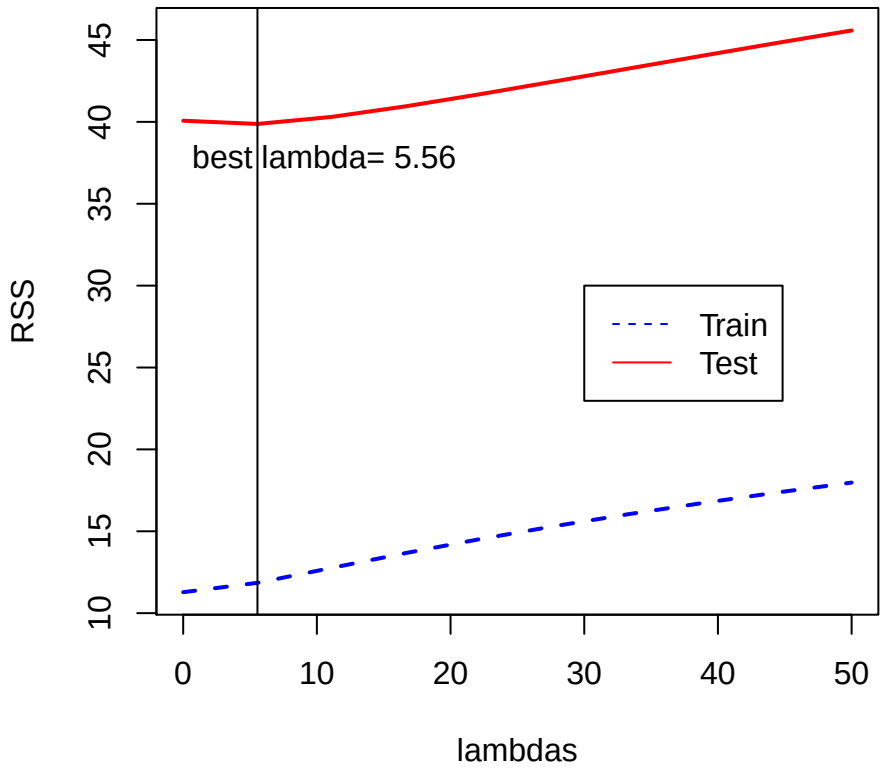
- this is equivalent to solving the equation

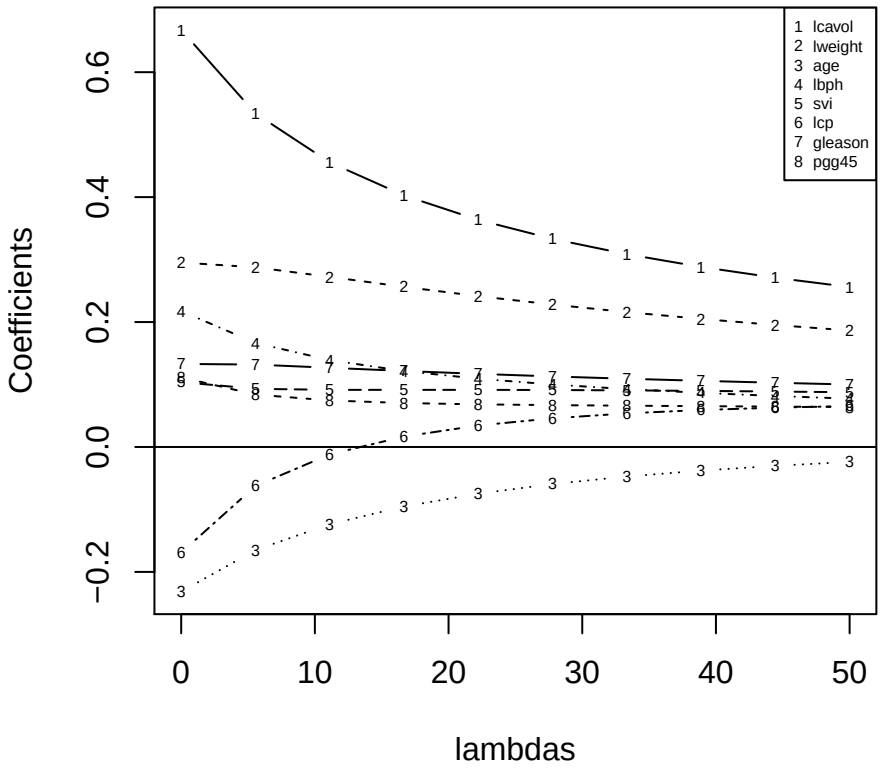
$$PRSS(\beta_j) = \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2$$

  subject to constraint $\sum_{j=1}^{p} \beta_j^2 \leq s$ where s is inversely proportional to λ
  - if the coefficients β_j are large in value, the term $\sum_{j=1}^{p} \beta_j^2$ will cause the overall PRSS value to increase, leading to worse models
  - the presence of the term thus requires some of the coefficients to be small
- inclusion of λ makes the problem *non-singular* even if $X^T X$ is not invertible
  - this means that even in cases where there are more predictors than observations, the coefficients of the predictors can still be estimated
- λ = tuning parameter
  - controls size of coefficients or the amount of regularization
  - as $\lambda \to 0$, the result approaches the least square solution
  - as $\lambda \to \infty$, all of the coefficients receive large penalties and the conditional coefficients $\hat{\beta}_{\lambda=\infty}^{ridge}$ approaches zero collectively
  - λ should be carefully chosen through cross-validation/other techniques to find the optimal tradeoff of bias for variance
  - ***Note**: it is important realize that all coefficients (though they may be shrunk to very small values) will **still be included** in the model when applying ridge regression*
- **R Commands**
  - [`MASS` package] `ridge<-lm.ridge(outcome ~ predictors, data=training, lambda=5)` = perform ridge regression with given outcome and predictors using the provided λ value
    - ***Note**: the predictors are **centered and scaled first** before the regression is run*
    - `lambda=5` = tuning parameter
    - `ridge$xm` = returns column/predictor mean from the data
    - `ridge$scale` = returns the scaling performed on the predictors for the ridge regression
      - ***Note**: all the variables are divided by the biased standard deviation* $\sum(X_i - \bar{X}_i)/n$
    - `ridge$coef` = returns the conditional coefficients, β_j from the ridge regression
    - `ridge$ym` = return mean of outcome
  - [`caret` package] `train(outcome ~ predictors, data=training, method="ridge", lambda=5)` = perform ridge regression with given outcome and predictors
    - `preProcess=c("center", "scale")` = centers and scales the predictors before the model is built
      - ***Note**: this is generally a good idea for building ridge regressions*
    - `lambda=5` = tuning parameter
  - [`caret` package] `train(outcome ~ predictors, data=training, method="foba", lambda=5, k=4)` = perform ridge regression with variable selection

    - `lambda=5` = tuning parameter
    - `k=4` = number of variables that should be retained
        - this means that `length(predictors)-k` variables will be eliminated
  - [`caret` package] `predict(model,test)` = use the model to predict on test set → similar to all other `caret` algorithms

- ***example: ridge coefficient paths vs*** λ
  - using the same `prostate` dataset, we will run ridge regressions with different values of λ and find the optimum λ value that minimizes test RSS





Regularized Regression - LASSO Regression

- LASSO (least absolute shrinkage and selection operator) was introduced by Tibshirani (Journal of the Royal Statistical Society 1996)
- similar to **ridge**, with slightly different penalty term
- the penalized residual sum of squares (PRSS) takes the form of

$$PRSS(\beta_j) = \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

- this is equivalent to solving the equation

$$PRSS(\beta_j) = \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2$$

  subject to constraint $\sum_{j=1}^{p} |\beta_j| \leq s$ where s is inversely proportional to λ
- λ = tuning parameter
  - controls size of coefficients or the amount of regularization
  - large values of λ will set some coefficient equal to zero

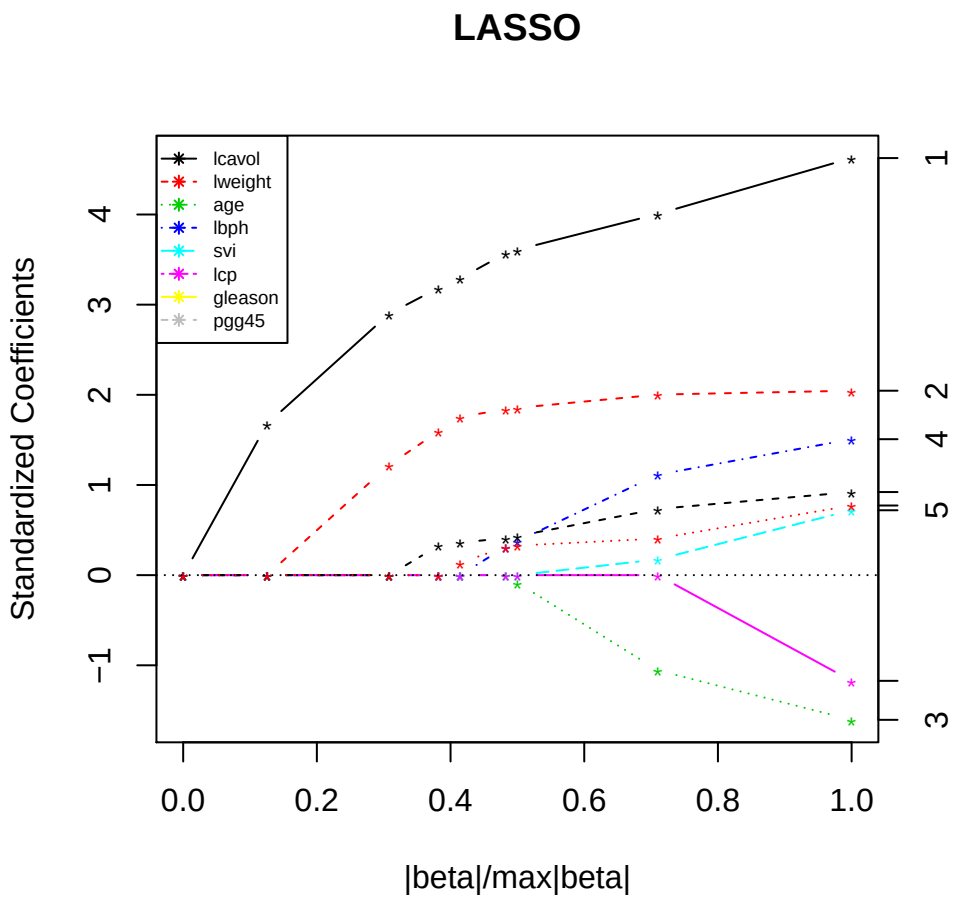
    - ***Note**: LASSO effectively performs model selection (choose subset of predictors) while shrinking other coefficients, where as Ridge only shrinks the coefficients*

- **R Commands**
  - [`lars` package] `lasso<-lars(as.matrix(x), y, type="lasso", trace=TRUE)` = perform lasso regression by adding predictors one at a time (or setting some variables to 0)
    - ***Note**: the predictors are **centered and scaled first** before the regression is run*
    - `as.matrix(x)` = the predictors must be in matrix/dataframe format
    - `trace=TRUE` = prints progress of the lasso regression
    - `lasso$lambda` = return the λs used for each step of the lasso regression
    - `plot(lasso)` = prints plot that shows the progression of the coefficients as they are set to zero one by one
    - `predit.lars(lasso, test)` = use the lasso model to predict on test data
      - ***Note**: more information/documentation can be found in `?predit.lars`*
  - [`lars` package] `cv.lars(as.matrix(x), y, K=10, type="lasso", trace=TRUE)` = computes K-fold cross-validated mean squared prediction error for lasso regression
    - effectively the `lars` function is run `K` times with each of the folds to estimate the
    - `K=10` = create 10-fold cross validation
    - `trace=TRUE` = prints progress of the lasso regression
  - [`enet` package] `lasso<-enet(predictors, outcome, lambda = 0)` = perform elastic net regression on given predictors and outcome
    - `lambda=0` = default value for λ
      - ***Note**: lasso regression is a special case of elastic net regression, and forcing `lambda=0` tells the function to fit a lasso regression*
    - `plot(lasso)` = prints plot that shows the progression of the coefficients as they are set to zero one by one
    - `predict.ent(lasso, test)`= use the lasso model to predict on test data
  - [`caret` package] `train(outcome ~ predictors, data=training, method="lasso")` = perform lasso regression with given outcome and predictors
    - ***Note**: outcome and predictors must be in the same dataframe*
    - `preProcess=c("center", "scale")` = centers and scales the predictors before the model is built
      - ***Note**: this is generally a good idea for building lasso regressions*
  - [`caret` package] `train(outcome~predictors,data=train,method="relaxo",lambda=5,phi=0.3)` = perform relaxed lasso regression on given predictors and outcome
    - `lambda=5` = tuning parameter
    - `phi=0.3` = relaxation parameter
      - `phi=1` corresponds to the regular Lasso solutions
      - `phi=0` computes the OLS estimates on the set of variables selected by the Lasso
  - [`caret` package] `predict(model,test)` = use the model to predict on test set → similar to all other `caret` algorithms

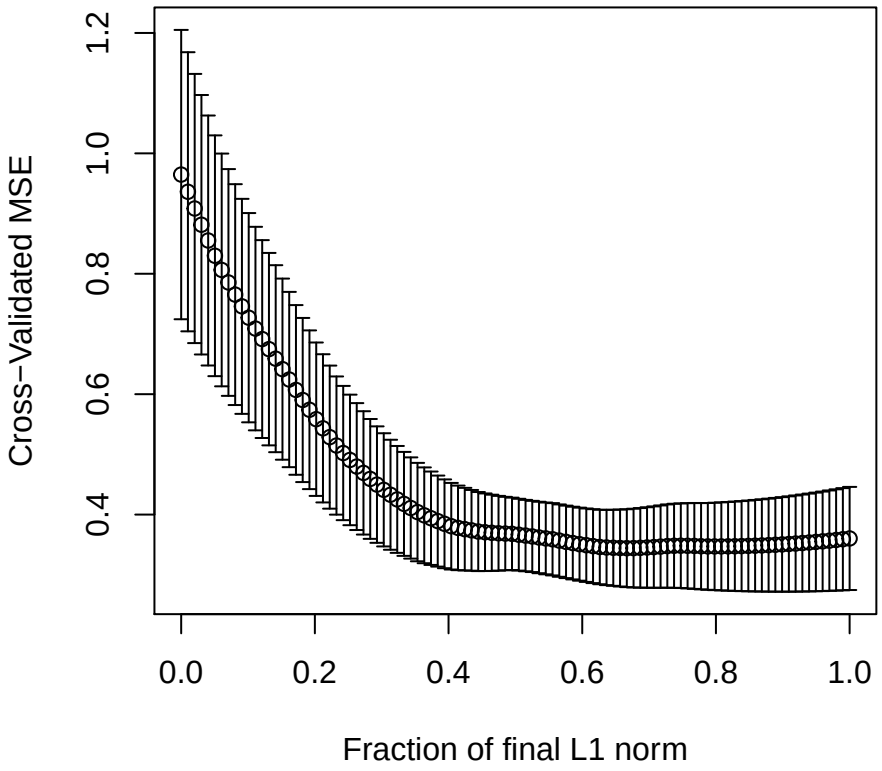
- ***example: `lars` package***

```
# load lars package
p_load("lars")
# perform lasso regression
lasso.fit <- lars(as.matrix(x), y, type="lasso", trace=TRUE)
# plot lasso regression model
plot(lasso.fit, breaks=FALSE, cex = 0.75)
```

```r
# add legend
legend("topleft", covnames, pch=8, lty=1:length(covnames),
    col=1:length(covnames), cex = 0.6)
```




```r
# plots the cross validation curve
lasso.cv <- cv.lars(as.matrix(x), y, K=10, type="lasso", trace=TRUE)
```


Cross−Validated MSE
0.4
0.6
0.8
1.0
1.2
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of final L1 norm

Combining Predictors

- **combining predictors** = also known as *ensembling methods in learning*, combine classifiers by averaging/voting to improve accuracy (generally)
  - this reduces interpretability and increases computational complexity
  - boosting/bagging/random forest algorithms all use this idea, except all classifiers averaged are of the same type
- Netflix Competition was won by a team that blended together 107 machine learning algorithms, Heritage Health Prize was also won by a combination of algorithms
  - ***Note**: the winning algorithm for Netflix was not used because it was too computationally complex/intensive, so the trade-off between accuracy and scalability is very important*
- **approach**
  - combine similar classifiers using bagging/boosting/random forest
  - combine different classifiers using model stacking/ensembling
    - build an odd number of models (we need an odd number for the majority vote to avoid ties)
    - predict with each model
    - combine the predictions and predict the final outcome by majority vote
- **process: simple model ensembling**
  1. ***build multiple models*** on the training data set
  2. use the models to ***predict*** on the training/test set
     - *predict on training* if the data was divided to only training/test sets
     - *predict on test* if the data was divided into training/test/validation sets
  3. combine the prediction results from each model and the true results for the training/test set into a ***new data frame***
     - one column for each model
     - add the true outcome from the training/test set as a separate column
  4. train the new data frame with a ***new model*** the true outcome as the outcome, and the predictions from various models as predictors
  5. use the combined model fit to predict results on the training/test data
     - calculate the RMSE for all models, including combined fit, to evaluate the accuracy of the different models
     - ***Note**: the RMSE for the combined fit should generally be **lower** than the the rest of the models*
  6. to predict on the final test/validation set, use all the initial models to predict on the data set first to ***recreate a prediction data frame*** for the test/validation set like in ***step 3***
     - one column for each model, no truth column this time
  7. ***apply the combined fit*** on the combined prediction data frame to get the final resultant predictions

Example - Majority Vote

- suppose we have 5 independent classifiers/models
- each has 70% accuracy
- **majority vote accuracy (mva)** = probability of the majority of the models achieving 70% at the same time

$$
\begin{aligned}
\text{majority vote accuracy} &= p(3\ \textit{correct},\ 2\ \textit{wrong}) + p(4\ \textit{correct},\ 1\ \textit{wrong}) \\
&\quad + p(5\ \textit{correct}) \\
&= \binom{5}{3} \times (0.7)^3(0.3)^2 + \binom{5}{4} \times (0.7)^4(0.3)^1 - \binom{5}{5}(0.7)^5 \\
&= 10 \times (0.7)^3(0.3)^2 + 5 \times (0.7)^4(0.3)^2 - 1 \times (0.7)^5 \\
&= 83.7
\end{aligned}
$$

- with 101 classifiers, the majority vote accuracy becomes 99.9%

Example - Model Ensembling

```
# set up data
inBuild <- createDataPartition(y=Wage$wage,p=0.7, list=FALSE)
validation <- Wage[-inBuild,]; buildData <- Wage[inBuild,]
inTrain <- createDataPartition(y=buildData$wage,p=0.7, list=FALSE)
training <- buildData[inTrain,]; testing <- buildData[-inTrain,]
# train the data using both glm and random forest models
glm.fit <- train(wage ~.,method="glm",data=training)
rf.fit <- train(wage ~.,method="rf",data=training,
    trControl = trainControl(method="cv"),number=3)
# use the models to predict the results on the testing set
glm.pred.test <- predict(glm.fit,testing)
rf.pred.test <- predict(rf.fit,testing)
# combine the prediction results and the true results into new data frame
combinedTestData <- data.frame(glm.pred=glm.pred.test,
    rf.pred = rf.pred.test,wage=testing$wage)
# run a Generalized Additive Model (gam) model on the combined test data
comb.fit <- train(wage ~.,method="gam",data=combinedTestData)
# use the resultant model to predict on the test set
comb.pred.test <- predict(comb.fit, combinedTestData)
# use the glm and rf models to predict results on the validation data set
glm.pred.val <- predict(glm.fit,validation)
rf.pred.val <- predict(rf.fit,validation)
# combine the results into data frame for the comb.fit
combinedValData <- data.frame(glm.pred=glm.pred.val,rf.pred=glm.pred.val)
# run the comb.fit on the combined validation data
comb.pred.val <- predict(comb.fit,combinedValData)
# tabulate the results - test data set RMSE Errors
rbind(test = c(glm = sqrt(sum((glm.pred.test-testing$wage)^2)),
        rf = sqrt(sum((rf.pred.test-testing$wage)^2)),
        combined = sqrt(sum((comb.pred.test-testing$wage)^2))),
    # validation data set RMSE Errors
    validation = c(sqrt(sum((glm.pred.val-validation$wage)^2)),
        sqrt(sum((rf.pred.val-validation$wage)^2)),
        sqrt(sum((comb.pred.val-validation$wage)^2))))
```

```
##                  glm        rf  combined
## test        858.7074  888.0702  849.3771
## validation 1061.0891 1086.2027 1057.8264
```

Forecasting

- **forecasting** = typically used with time series, predict one or more observations into the future
  - data are dependent over time so subsampling/splitting data into training/test is more complicated and must be done very carefully
- specific patterns need to be considered for time series data (*time series decomposition*)
  - **trend** = long term increase/decrease in data
  - **seasonal** = patterns related to time of week/month/year/etc
  - **cyclic** = patterns that rise/fall periodically
- ***Note**: issues that arise from time series are similar to those from spatial data*
  - dependency between nearby observations
  - location specific effects
- all standard predictions models can be used but requires more consideration
- ***Note**: more detailed tutorial can be found in Rob Hyndman's Forecasting: principles and practice*
- **considerations for interpreting results**
  - unrelated time series can often seem to be correlated with each other (*spurious correlations*)
  - geographic analysis may exhibit similar patterns due to population distribution/concentrations
  - extrapolations too far into future can be dangerous as they can produce in insensible results
  - dependencies over time (seasonal effects) should be examined and isolated from the trends
- **process**
  - ensure the data is a time series data type
  - split data into training and test sets
    - both must have consecutive time points
  - choose forecast approach (SMA - `ma` vs EMA - `ets`, see below) and apply corresponding functions to training data
  - apply constructed forecast model to test data using `forecast` function
  - evaluate accuracy of forecast using `accuracy` function
- **approaches**
  - ***simple moving averages*** = prediction will be made for a time point by averaging together values from a number of prior periods

$$Y_t = \frac{1}{2 * k + 1} \sum_{j=-k}^{k} y_{t+j}$$

  - ***exponential smoothing/exponential moving average*** = weight time points that are closer to point of prediction than those that are further away

$$\hat{y}_{t+1} = \alpha y_t + (1 - \alpha)\hat{y}_{t-1}$$

    - ***Note**: many different methods of exponential smoothing are available, more information can be found here*

R Commands and Examples

- `quantmod` package can be used to pull trading/price information for publicly traded stocks
  - `getSymbols("TICKER", src="google", from=date, to=date)` = gets the ***daily*** high/low/open/close price and volume information for the specified stock ticker
    - returns the data in a data frame under the stock ticker's name
    - `"TICKER"` = ticker of the stock you are attempting to pull information for
    - `src="google"` = get price/volume information from Google finance
      - default source of information is Yahoo Finance
    - `from` and `to` = from and to dates for the price/volume information
      - both arguments must be specified with `date` objects
    - ***Note**: more information about how to use `getSymbols` can be found in the documentation `?getSymbols`*
  - `to.monthly(GOOG)` = converts stock data to monthly time series from daily data
    - the function aggregates the open/close/high/low/volume information for each day into monthly data
    - `GOOG` = data frame returned from `getSymbols` function
    - ***Note**: `?to.period` contains documentation for converting time series to OHLC (open high low close) series*
  - `googOpen<-Op(GOOG)` = returns the opening price from the stock data frame
    - `Cl(), Hi(), Lo()` = returns the close, high and low price from the stock data frame
  - `ts(googOpen, frequency=12)` = convert data to a time series with `frequency` observations per time unit
    - `frequency=12` = number of observations per unit time (12 in this case because there are 12 months in each year → converts data into ***yearly*** time series)
- `decompose(ts)` = decomposes time series into trend, seasonal, and irregular components by using moving averages
  - `ts` = time series object
- `window(ts, start=1, end=6)` = subsets the time series at the specified starting and ending points
  - `start` and `end` arguments must correspond to the **time unit** rather than the ***index***
    - for instance, if the `ts` is a yearly series (`frequency = 12`), `start`/`end` should correspond to the row numbers or year (each year has 12 observations corresponding to the months)
    - `c(1, 7)` can be used to specify the element of a particular year (in this case, July of the first year/row)
    - ***Note**: you can use 9.5 or any decimal as values for `start`/`end`, and the closest element (June of the 9th year in this case) will be used*
    - ***Note**: `end=9-0.01` can be used a short cut to specify "up to 9", since `end = 9` will include the first element of the 9th row*
- `forecast` package can be used for forecasting time series data
  - `ma(ts, order=3)` = calculates the simple moving average for the order specified
    - `order=3` = order of moving average smoother, effectively the number of values that should be used to calculate the moving average
  - `ets(train, model="MMM")` = runs exponential smoothing model on training data
    - `model = "MMM"` = method used to create exponential smoothing
      - ***Note**: more information can be found at `?ets` and the corresponding model chart is here*
  - `forecast(ts)` = performs forecast on specified time series and returns 5 columns: forecast values, high/low 80 confidence intervals bounds, high/low 95 percent interval bounds

        * `plot(forecast)` = plots the forecast object, which includes the training data, forecast values for test periods, as well as the 80 and 95 percent confidence interval regions
    - `accuracy(forecast, testData)` = returns the accuracy metrics (RMSE, etc.) for the forecast model

- `quandl` package is also used for finance-related predictions
- ***example: decomposed time series***

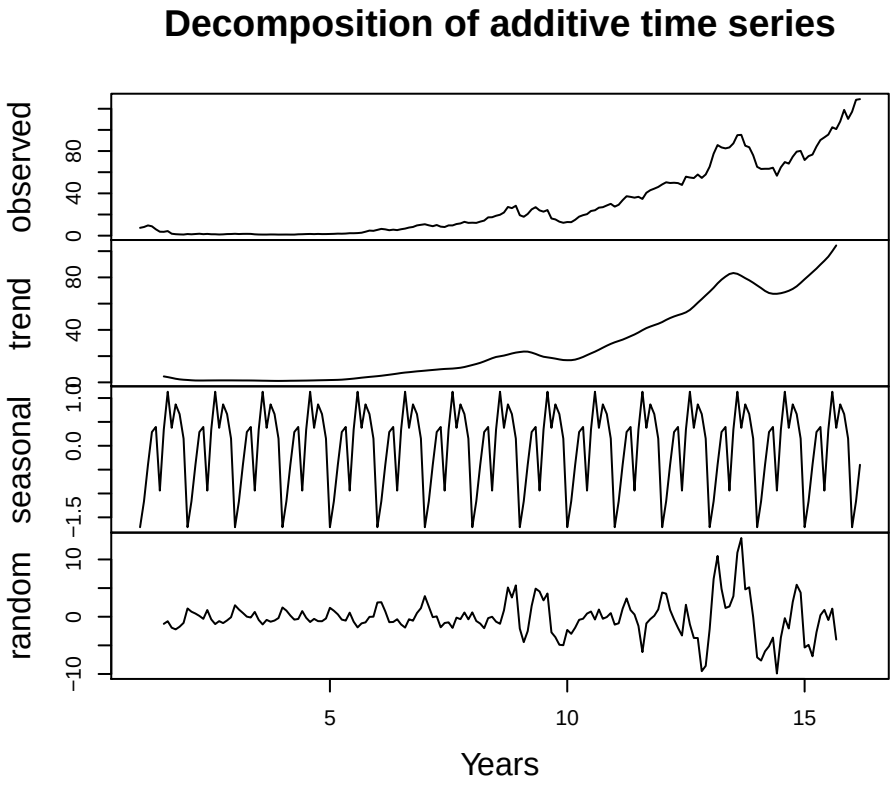
```
# load quantmod package
p_load("quantmod");
```

```
##
## The downloaded binary packages are in
##  /var/folders/1b/8brmpmcd0fqbhn4v_cl5qrz80000gn/T//Rtmp8XgJGH/downloaded_packages
```

```
# specify to and from dates
from.dat <- as.Date("01/01/00", format="%m/%d/%y")
to.dat <- as.Date("3/2/15", format="%m/%d/%y")
# get data for AAPL from Google Finance for the specified dates
getSymbols("AAPL", src="google", from = from.dat, to = to.dat)
```

```
## [1] "AAPL"
```

```
# convert the retrieved daily data to monthly data
mAAPL <- to.monthly(AAPL)
# extract the closing price and convert it to yearly time series (12 observations per year)
ts <- ts(Cl(mAAPL), frequency = 12)
# plot the decomposed parts of the time series
plot(decompose(ts),xlab="Years")
```

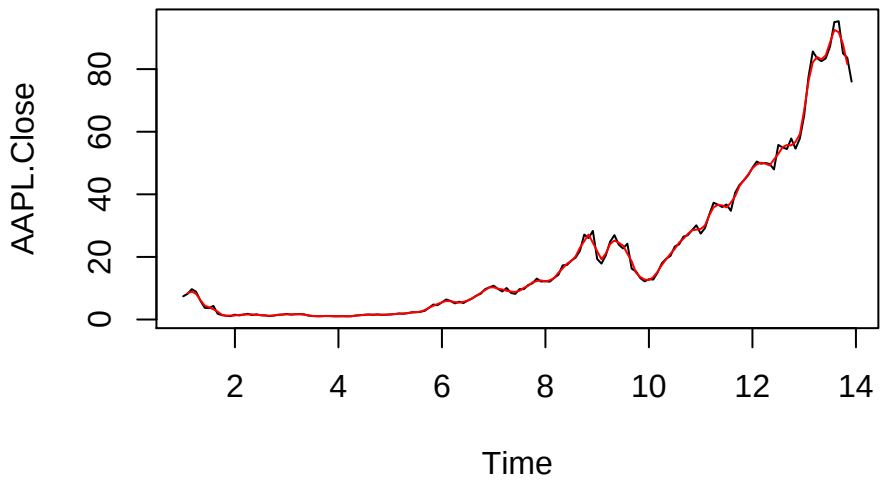


- *example: forecast*

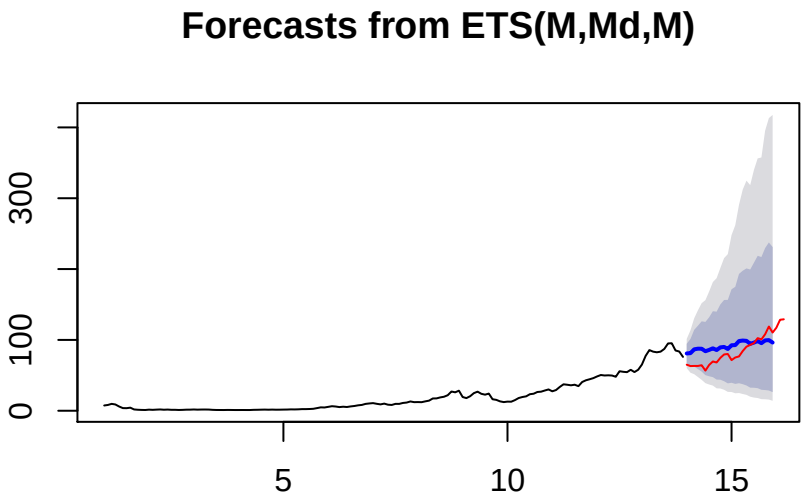
```
# load forecast library
p_load("forecast")
```

```
##
## The downloaded binary packages are in
##  /var/folders/1b/8brmpmcd0fqbhn4v_cl5qrz80000gn/T//Rtmp8XgJGH/downloaded_packages
```

```
# find the number of rows (years)
rows <- ceiling(length(ts)/12)
# use 90% of the data to create training set
ts.train <- window(ts, start = 1, end = floor(rows*.9)-0.01)
# use the rest of data to create test set
ts.test <- window(ts, start = floor(rows*.9))
# plot the training set
plot(ts.train)
# add the moving average in red
lines(ma(ts.train,order=3),col="red")
```




```
# compute the exponential smoothing average
ets <- ets(ts.train,model="MMM")
# construct a forecasting model using the exponential smoothing function
fcast <- forecast(ets)
# plot forecast and add actual data in red
plot(fcast); lines(ts.test,col="red")
```



```r
# print the accuracy of the forecast model
accuracy(fcast,ts.test)
```

```
##                       ME      RMSE       MAE         MPE     MAPE
## Training set   0.05089724  2.604205  1.427588  -0.8627284 10.32254
## Test set     -10.68215774 16.711650 14.965273 -16.8508577 20.70343
##                   MASE       ACF1 Theil's U
## Training set 0.1716568 0.06484368        NA
## Test set     1.7994616 0.83467782  3.480289
```

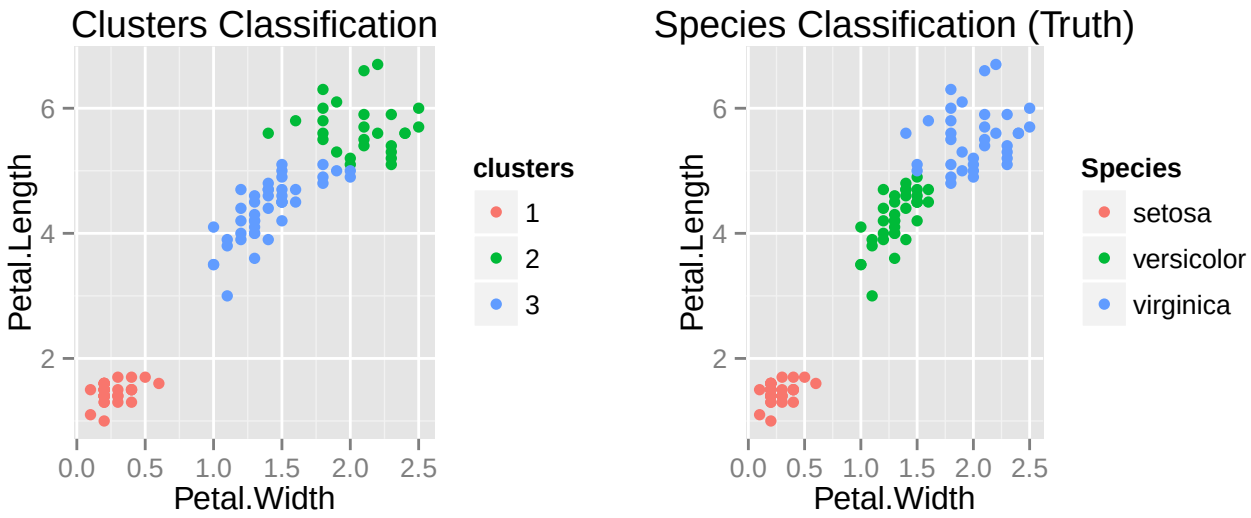
Unsupervised Prediction

- **supervised classification** = predicting outcome when we know what the different classifications are
  - *example*: predicting the type of flower (setosa, versicolor, or virginica) based on sepal width/length
- **unsupervised classification** = predicting outcome when we don't know what the different classifications are
  - *example*: splitting all data for sepal width/length into different groups (cluster similar data together)
- **process**
  - provided that the labels for prediction/outcome are unknown, we first build clusters from observed data
    - creating clusters are not noiseless process, and thus may introduce higher variance/error for data
    - ***K-means*** is an example of a clustering approach
  - label the clusters
    - interpreting the clusters well (sensible vs non-sensible clusters) is incredibly challenging
  - build prediction model with the clusters as the outcome
    - all algorithms can be applied here
  - in new data set, we will predict the clusters labels
- unsupervised prediction is effectively a ***exploratory technique***, so the resulting clusters should be carefully interpreted
  - clusters may be highly variable depending on the method through which the data is sample
- generally a good idea to create custom clustering algorithms for given data as it is **crucial** to define the process to identify clusters for interpretability and utility of the model
- unsupervised prediction = basic approach to recommendation engines, in which the tastes of the existing users are clustered and applied to new users

R Commands and Examples

- `kmeans(data, centers=3)` = can be used to perform clustering from the provided data
  - `centers=3` = controls the number of clusters the algorithm should aim to divide the data into
- `cl_predict` function in `clue` package provides similar functionality

```
# load iris data
data(iris)
# create training and test sets
inTrain <- createDataPartition(y=iris$Species,p=0.7, list=FALSE)
training <- iris[inTrain,]; testing <- iris[-inTrain,]
# perform k-means clustering for the data without the Species information
# Species = what the true clusters are
kMeans1 <- kmeans(subset(training,select=-c(Species)),centers=3)
# add clusters as new variable to training set
training$clusters <- as.factor(kMeans1$cluster)
# plot clusters vs Species classification
p1 <- qplot(Petal.Width,Petal.Length,colour=clusters,data=training) +
    ggtitle("Clusters Classification")
p2 <- qplot(Petal.Width,Petal.Length,colour=Species,data=training) +
    ggtitle("Species Classification (Truth)")
grid.arrange(p1, p2, ncol = 2)
```



- as we can see, there are three clear groups that emerge from the data
  - this is fairly close to the actual results from Species
  - we can compare the results from the clustering and Species classification by tabulating the values

```
# tabulate the results from clustering and actual species
table(kMeans1$cluster,training$Species)
```

```
##
##     setosa versicolor virginica
##   1     35          0         0
##   2      0          0        27
##   3      0         35         8
```

- with the clusters determined, the training data can be trained on all predictors with the clusters from k-means as outcome

```
# build classification trees using the k-means cluster
clustering <- train(clusters ~.,data=subset(training,select=-c(Species)),method="rpart")
```

- we can compare the prediction results on training set vs truth

```
# tabulate the prediction results on training set vs truth
table(predict(clustering,training),training$Species)
```

```
##
##     setosa versicolor virginica
##   1     35          0         0
##   2      0          0        29
##   3      0         35         6
```

- similarly, we can compare the prediction results on test set vs truth

```r
# tabulate the prediction results on test set vs truth
table(predict(clustering,testing),testing$Species)
```

```
##
##     setosa versicolor virginica
##   1     15          0         0
##   2      0          1        12
##   3      0         14         3
```