**Data Camp: Machine Learning Courses**

**Data Camp: Machine learning with tree-based models in R**

Tree-based models

Interpretability + ease-of-use + accuracy

Make decisions + numeric predictions

Decision tree terminology: nodes

Root node – internal nodes – terminal nodes

Training decisions trees in R using the rpart package

Library(“rpart”)

# fit a classification model using a data frame to train the model

str(creditsub)

'data.frame': 522 obs. of 5 variables:

$ months\_loan\_duration: int 48 42 24 36 30 12 48 12 24 15 ...

$ percent\_of\_income : int 2 2 3 2 4 3 3 1 4 2 ...

$ years\_at\_residence : int 2 4 4 2 2 1 4 1 4 4 ...

$ age : int 22 45 53 35 28 25 24 22 60 28 ...

$ default : Factor w/ 2 levels "no","yes": 2 1 2 1 2 2 2 1 2 1 ...

Credit\_model <- rpart(formula = default ~ ., data = creditsub, method = class)

#display the results:

rpart.plot(x=credit\_model, yesno=2, type = 0, extra = 0)

**Introduction to classification trees:**

*Advantages:*

One of the biggest advantages - simple to understand, interpret, and visualize

Require no normalization for numerical features

Can handle both numerical and categorical features (inputs) natively

Can handle missing data elegantly – do you go down the train right or left (randomly)?

Requires little data preparation

Can model non-linearity in the data

Can be trained quickly on data sets

*Disadvantages:*

Large trees can be hard to interpret

Trees have high variance, which causes model performance to be poor

Trees overfit easily

**Overview of the modeling process:**

*Train/test split*

Example: Training on 80% or testing on 20%

*How to do it in R:*

# Figure out number of rows in the restaurant data frame

N <- nrow(restaurant)

# Number of rows for the training set (80% of the dataset)

N\_train<- round (0.8\*N)

# set a random seed for reproducibility

Set.seed(123)

# Create a vector of indices which is an 80% random sample

Train\_indices <- sample(1:N, N\_train)

# Subset the data frame to training indices only

restaurant\_train <- restaurant[train\_indices, ]

# Exclude the training inde

restaurant\_test <- restaurant[-train\_indices,]

# train the model to predict the binary response, “will\_wait”

restaurant\_model <- rpart(formula = will\_wait ~., data = restaurant\_train, method=”class”)

# formula: response variable ~ predictor variables

# so in our example the response “will\_wait” is a function of “.” Which are all of the predictor variables.

# Method = “class” since it’s a binary response.

**Evaluate model performance**

Predict(model, test\_dataset, type= \_ \_ \_)

*Evaluation metrics for binary classification*

Accuracy

Confusion matrix

Log-loss

AUC

*Accuracy* = (n of correct predictions)/(n of total data points)

*Confusion matrix* shows a more detailed break down (e.g. type I error (false positive) vs. type II error (false negative))

# package for generating confusion matrix

Library(caret)

# Generate predicted classes using the model object

class\_prediction <- predict(object = credit\_model,

newdata = credit\_test,

type = "class")

# Calculate the confusion matrix for the test set

confusionMatrix(data = class\_prediction,

reference = credit\_test$default)

**Use of splitting criterion in trees**

Comparing models with different splitting criterion.

****

How do you measure the purity of splits, so you know what method you use?

We want to minimize the impurity measure – Gini Index

The lower the Gini index, the higher the purity index.

**Comparing two models:**

# Train a gini-based model

credit\_model1 <- rpart(formula = default ~ .,

data = credit\_train,

method = "class",

parms = list(split = "gini"))

# Train an information-based model

credit\_model2 <- rpart(formula = default ~ .,

data = credit\_train,

method = "class",

parms = list(split = "information"))

# Generate predictions on the validation set using the gini model

pred1 <- predict(object = credit\_model1,

newdata = credit\_test,

type = "class")

# Generate predictions on the validation set using the information model

pred2 <- predict(object = credit\_model2,

newdata = credit\_test,

type = "class")

# Compare classification error

ce(actual = credit\_test$default,

predicted = pred1)

ce(actual = credit\_test$default,

predicted = pred2)

**Introduction to regression trees**

A regression model – predict a numeric outcome

The data can be continuous or integer valued

Train a regression tree in R

rpart(formula = ---, data = \_\_\_, method = “anova”)

we’ll use a training set, validation set (to tweak the parameters of the model), and test set

**Train a regression model**

# Train the model

grade\_model <- rpart(formula = final\_grade ~ .,

data = grade\_train,

method = "anova")

# Look at the model output

print(grade\_model)

# Plot the tree model

rpart.plot(x = grade\_model, yesno = 2, type = 0, extra = 0)

**Performance metrics for regression**

Mean absolute error (MAE)

Root mean square error (RMSE)

Lower values better!

RMSE punishes large errors more harshly, and therefore more useful when large errors are undesirable

Which one you use is largely chosen based on each test case

# Metrics package

library(metrics)

# Generate predictions on a test set

pred <- predict(object = grade\_model, # model object

newdata = grade\_test) # test dataset

# Compute the RMSE

rmse(actual = grade\_test$final\_grade,

predicted = pred)

[1] 2.278249

**What are the hyperparameters for a decision tree?**

?rpart.control()

# this allows you to tweak many parameters

Minsplit: minimum number of data points required to attempt a split

cp: complexity parameter

Maxdepth: depth of a decision tree

Cost-complexity parameter: penalty

plotcp(grade\_model)

# return the optimized model using the prune function:

# prune the model to the optimized cp value

model\_opt <- prune(tree=model, cp=cp\_opt)

# Plot the "CP Table"

plotcp(grade\_model)

# Print the "CP Table"

print(grade\_model$cptable)

# Retrieve optimal cp value based on cross-validated error

opt\_index <- which.min(grade\_model$cptable[, "xerror"])

cp\_opt <- grade\_model$cptable[opt\_index, "CP"]

# Prune the model (to optimized cp value)

grade\_model\_opt <- prune(tree = grade\_model,

cp = cp\_opt)

# Plot the optimized model

rpart.plot(x = grade\_model\_opt, yesno = 2, type = 0, extra = 0)

Random Forest

**Grid search for model selection**

What is a model hyperparameter?

What is a “grid”?

What is the goal of a grid search?

how is the best model chosen? -> choose a good metric(s) (RMSE)

Set up the grid

# Establish a list of possible values for minsplit and maxdepth

splits <- seq(1,30,5)

depths <- seq(5,40,10)

# Create a data frame containing all combinations

hyper\_grid <- expand.grid(minsplit = splits, maxdepth = depths)

# Create an empty list to store models

models <- list()

# Execute the grid search

for (i in 1:nrow(hyper\_grid)){

# get minsplit, maxdepth values at row i

minsplit <- hyper\_grid$minsplit[i]

maxdepth <- hyper\_grid$maxdepth[i]

# train a model and store in the list

models[[i]] <- rpart(formula = response ~., data = train, methods = “anova”, minsplit=minsplit, maxdepth = maxdepth

## Create an empty vector to store RMSE values

rmse\_values <- c()

# Compute validation RMSE

for (I in 1:length(models)){

# Retrieve the i^th model from the list

model <- model[[i]]

# generate predictions on grade\_valid

pred <- predict(object = model, newdata = valid)

# Compute validation RMSE and add to the

rmse\_values[i] <- rmse(actual = valid$response, predicted = pred)

**Evaluate the grid**

Earlier in the chapter we split the dataset into three parts: training, validation and test.

A dataset that is not used in training is sometimes referred to as a "holdout" set. A holdout set is used to estimate model performance and although both validation and test sets are considered to be holdout data, there is a key difference:

* Just like a test set, a validation set is used to evaluate the performance of a model. The difference is that a validation set is specifically used to compare the performance of a group of models with the goal of choosing a "best model" from the group. All the models in a group are evaluated on the same validation set and the model with the best performance is considered to be the winner.
* Once you have the best model, a final estimate of performance is computed on the test set.
* A test set should only ever be used to estimate model performance and should not be used in model selection. Typically if you use a test set more than once, you are probably doing something wrong.

# Number of potential models in the grid

num\_models <- length(grade\_models)

# Create an empty vector to store RMSE values

rmse\_values <- c()

# Write a loop over the models to compute validation RMSE

for (i in 1:num\_models) {

# Retrieve the i^th model from the list

model <- grade\_models[[i]]

# Generate predictions on grade\_valid

pred <- predict(object = model,

newdata = grade\_valid)

# Compute validation RMSE and add to the

rmse\_values[i] <- rmse(actual = grade\_valid$final\_grade,

predicted = pred)

}

# Identify the model with smallest validation set RMSE

best\_model <- grade\_models[[which.min(rmse\_values)]]

# Print the model paramters of the best model

best\_model$control

# Compute test set RMSE on best\_model

pred <- predict(object = best\_model,

newdata = grade\_test)

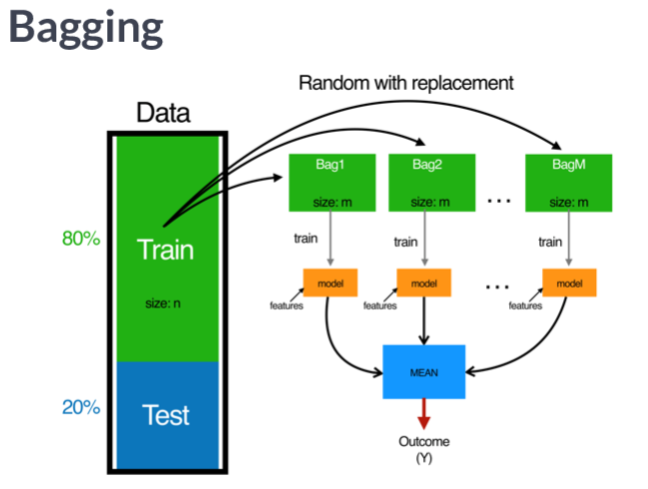
rmse(actual = grade\_test$final\_grade,

predicted = pred)

**Chapter 3: Introduction to bagged trees**

Bagging ~ averaging multiple models to reduce overfitting and reduce variance

Bootstrap AGGregatING



Bagging can drastically reduce the variance.

Can lead to a better fitting model

*Bagging in R*

library(ipred)

bagging(formula=response ~., data = dat)

The number of bagged trees can be specified using the nbagg parameter (default = 25)

predict works the same way

class\_predictions <- predict(object = rest\_model\_bag, newdata = restaurant\_test, type = “class”)

print(class\_predictions)

confusionMatrix(data=class\_pred, reference = restaurant\_test$will\_wait)

# *Predict on a test set and compute AUC*

In binary classification problems, we can predict numeric values instead of class labels. In fact, class labels are created only after you use the model to predict a raw, numeric, predicted value for a test point.

The predicted label is generated by applying a threshold to the predicted value, such that all tests points with predicted value greater than that threshold get a predicted label of "1" and, points below that threshold get a predicted label of "0".

In this exercise, generate predicted values (rather than class labels) on the test set and evaluate performance based on [**AUC (Area Under the ROC Curve)**](https://en.wikipedia.org/wiki/Receiver_operating_characteristic#Area_under_the_curve). The AUC is a common metric for evaluating the discriminatory ability of a binary classification model.

# Generate predictions on the test set

pred <- predict(object = credit\_model,

newdata = credit\_test,

type = "prob")

# `pred` is a matrix

class(pred)

# Look at the pred format

head(pred)

# Compute the AUC (`actual` must be a binary (or 1/0 numeric) vector)

auc(actual = ifelse(credit\_test$default == "yes", 1, 0),

predicted = pred[,"yes"])

[1] 0.7809724

*Cross-validation*

K-fold cross-validation

dataset size = 200 rows

k = 10(number of cross validation folds)

Will have 10 estimates of test set AUC

Then you will have the average of the cross-validated estimate of AUC

Using caret for cross validating models

library(caret)

train()

trainControl()

# specify the training configuration

ctrl <- traincontrol(method=”CV”, number = 5, classProbs = TRUE, summaryFunction = twoClassSummary)

set.seed(123)

credit\_model <- train(default ~., data = credit\_train, method=”treebag”,metric = “ROC” trControl = ctrl)

*Compare test set performance to CV performance*

In this excercise, you will print test set AUC estimates that you computed in previous exercises. These two methods use the same code underneath, so the estimates should be very similar.

* The credit\_ipred\_model\_test\_auc object stores the test set AUC from the model trained using the ipred::bagging() function.
* The credit\_caret\_model\_test\_auc object stores the test set AUC from the model trained using the caret::train() function with method = "treebag".

Lastly, we will print the 5-fold cross-validated estimate of AUC that is stored within the credit\_caret\_model object. This number will be a more accurate estimate of the true model performance since we have averaged the performance over five models instead of just one.

On small datasets like this one, the difference between test set model performance estimates and cross-validated model performance estimates will tend to be more pronounced. When using small data, it's recommended to use cross-validated estimates of performance because they are more stable.

*Chapter 4: Introduction to Random Forest*

Random Forest algorithm is an improvement upon bagged trees.

Has better performance

Very popular machine learning method

Identical idea to bagging trees

The key difference is that we add some randomness to the model. Sample a subset of features at each split in a decision tree. In bagging all features are used. Adding the randomness reduces correlation between the sampled trees

Random forest in R:

library(randomForest)

model <- randomForest(formula response ~., data=train)

Understanding Random Forest model output

# Print the credit\_model output

print(credit\_model)

# Grab OOB error matrix & take a look

err <- credit\_model$err.rate

head(err)

Plotting the error rate as a function of the number of trees trained is useful to see how many trees to train

# *Evaluate out-of-bag error*

Here you will plot the OOB error as a function of the number of trees trained, and extract the final OOB error of the Random Forest model from the trained model object.

# Grab OOB error matrix & take a look

err <- credit\_model$err.rate

head(err)

# Look at final OOB error rate (last row in err matrix)

oob\_err <- err[500, "OOB"]

print(oob\_err)

# Plot the model trained in the previous exercise

plot(credit\_model)

# Add a legend since it doesn't have one by default

legend(x = "right",

legend = colnames(err),

fill = 1:ncol(err))

# *Evaluate model performance on a test set*

Use the caret::confusionMatrix() function to compute test set accuracy and generate a confusion matrix. Compare the test set accuracy to the OOB accuracy.

# Generate predicted classes using the model object

class\_prediction <- predict(object = credit\_model, # model object

newdata = credit\_test, # test dataset

type = "class") # return classification labels

# Calculate the confusion matrix for the test set

cm <- confusionMatrix(data = class\_prediction, # predicted classes

reference = credit\_test$default) # actual classes

print(cm)

# Compare test set accuracy to OOB accuracy

paste0("Test Accuracy: ", cm$overall[1])

paste0("OOB Accuracy: ", 1 - oob\_err)

*Advantages & Disadvantages of OOB estimates*

Can evaluate your model without a separate test set

Computer automatically by the randomForest() function

OOB Error only estimates error (not AUC, log-loss, etc)

Can’t compare Random Forest performance to other types of models

The main benefit of OOB is you can evaluate your model without needing to create a separate test set.

*Tuning a random forest model*

One of the easier algorithms to ‘tune’, compared to SVM or neural networks

ntree: number of trees in the ofrest (default = 500)

mtry: number of variables randomly samples as candidates at each split

sampsize: number of samples to train on

nodesize: minimum size (number of samples) of the terminal nodes

maxnodes: max number of terminal nodes

# the names may be different depending on the platform youre using

Tuning mtry with tuneRF() (by OOB error)

set.seed(1)

res<-tuneRF(x=train\_predictor\_df, y=train\_response\_vector, ntreeTry=500)

# Execute the tuning process

set.seed(1)

res <- tuneRF(x = subset(credit\_train, select = -default),

y = credit\_train$default,

ntreeTry = 500)

# Look at results

print(res)

# Find the mtry value that minimizes OOB Error

mtry\_opt <- res[,"mtry"][which.min(res[,"OOBError"])]

print(mtry\_opt)

# If you just want to return the best RF model (rather than results)

# you can set `doBest = TRUE` in `tuneRF()` to return the best RF model

# instead of a set performance matrix.

*Tuning a random forest via tree depth*

# Establish a list of possible values for mtry, nodesize and sampsize

mtry <- seq(4, ncol(credit\_train) \* 0.8, 2)

nodesize <- seq(3, 8, 2)

sampsize <- nrow(credit\_train) \* c(0.7, 0.8)

# Create a data frame containing all combinations

hyper\_grid <- expand.grid(mtry = mtry, nodesize = nodesize, sampsize = sampsize)

# Create an empty vector to store OOB error values

oob\_err <- c()

# Write a loop over the rows of hyper\_grid to train the grid of models

for (i in 1:nrow(hyper\_grid)) {

# Train a Random Forest model

model <- randomForest(formula = default ~ .,

data = credit\_train,

mtry = hyper\_grid$mtry[i],

nodesize = hyper\_grid$nodesize[i],

sampsize = hyper\_grid$sampsize[i])

# Store OOB error for the model

oob\_err[i] <- model$err.rate[nrow(model$err.rate), "OOB"]

}

# Identify optimal set of hyperparmeters based on OOB error

opt\_i <- which.min(oob\_err)

print(hyper\_grid[opt\_i,])

**Chapter 5: Introduction to boosting**

Boosting algorithms

Adaboost

Gradient boosting machine (“GBM”)

Adaboost algorithm: train decision tree with equally weighted observations

increase/lower the weights of the observations

second tree is grown on weighted data

New model: tree 1 + tree 2

classification error from this new 2-tree ensemble model

grow 3rd tree to predict the revised residuals

repeat this process for a specified number of iterations

*Gradient boosting machine*

Gradient boosting = gradient descent + boosting

fit an additive model (ensemble) in a forward. stage-wise manner

In each stage, introduce a “weak learner” e.g. decision tree to compensate the shortcomings of existing weak learners.

Boosted trees improve the model fit by considering past fits (bagged trees don’t do this!)

In adaboost, “shortcomings” are identified by high-weight data points

in gradient boosting the “shortcomings” are identified by gradients

Advantages:

GBM is popular because it generally performs better than any other algorithm

directly optimizes cost function

Disadvantages:

problem is it may overfit (you’ll need to find a proper stopping point)

It is also sensitive to extreme values and noise

# Train a GBM:

model <- gbm(formula = response ~., distribution = ‘bernoulli’, data=train, n.trees=5000)

distribution of the response variable (Bernoulli is for binary variables)

n.trees default is 100.

*Understanding GBM model output:*

They will tell you which variables don’t have an influence. You also figure out the relative influence of each of the variables.

predict(model, type=”response”,n.trees=10000)

# note that you need to specify the n.trees for predict.gbm()

# Since we converted the training response col, let's also convert the test response col

credit\_test$default <- ifelse(credit\_test$default == "yes", 1, 0)

# Generate predictions on the test set

preds1 <- predict(object = credit\_model,

newdata = credit\_test,

n.trees = 10000)

# Generate predictions on the test set (scale to response)

preds2 <- predict(object = credit\_model,

newdata = credit\_test,

n.trees = 10000,

type = "response")

# Compare the range of the two sets of predictions

range(preds1)

range(preds2)

*Tuning a GBM model*

Important GBM parameters:

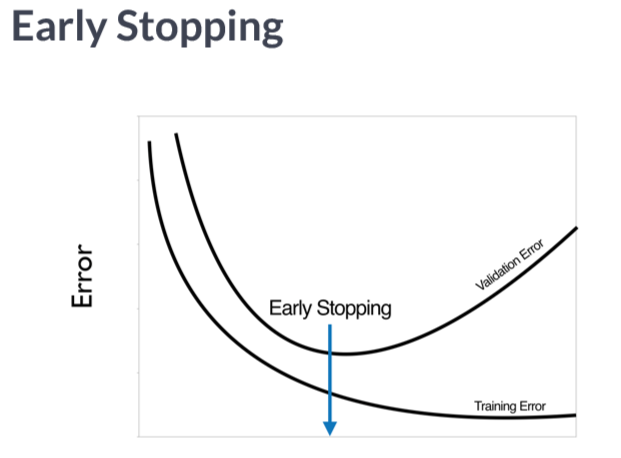
Some overlap with random forest

n.trees = number of trees

bag.fraction: proportion of observations to be sampled in each tree

n.minobsinnode: minimum number of observations in the trees terminal nodes

interaction.depth: maximum nodes per tree

shrinkage: learning rate (better to make more small steps than a few large steps)

# train a GBM model

model <- gbm(formulae = response ~., distribution = “bernoulli”, data=train, n.trees = 5000, cv.folds =3)

# get optimal ntree based on OOB error

ntree\_opt\_oob <- gbm.perf(model, method = “OOB”)

# get optimal ntree based on CV error

ntree\_opt\_cv <- gbm.perf(model, method =”cv”)

*Early stopping GBMs*

# Optimal ntree estimate based on OOB

ntree\_opt\_oob <- gbm.perf(object = credit\_model,

method = "OOB",

oobag.curve = TRUE)

# Train a CV GBM model

set.seed(1)

credit\_model\_cv <- gbm(formula = default ~ .,

distribution = "bernoulli",

data = credit\_train,

n.trees = 10000,

cv.folds = 2)

# Optimal ntree estimate based on CV

ntree\_opt\_cv <- gbm.perf(object = credit\_model\_cv,

method = "cv")

# Compare the estimates

print(paste0("Optimal n.trees (OOB Estimate): ", ntree\_opt\_oob))

[1] "Optimal n.trees (OOB Estimate): 3233"

print(paste0("Optimal n.trees (CV Estimate): ", ntree\_opt\_cv))

[1] "Optimal n.trees (CV Estimate): 7889"

# *OOB vs CV-based early stopping*

# Generate predictions on the test set using ntree\_opt\_oob number of trees

preds1 <- predict(object = credit\_model,

newdata = credit\_test,

n.trees = ntree\_opt\_oob)

# Generate predictions on the test set using ntree\_opt\_cv number of trees

preds2 <- predict(object = credit\_model,

newdata = credit\_test,

n.trees = ntree\_opt\_cv)

# Generate the test set AUCs using the two sets of preditions & compare

auc1 <- auc(actual = credit\_test$default, predicted = preds1) #OOB

auc2 <- auc(actual = credit\_test$default, predicted = preds2) #CV

# Compare AUC

print(paste0("Test set AUC (OOB): ", auc1))

[1] "Test set AUC (OOB): 0.777816736792894"

print(paste0("Test set AUC (CV): ", auc2))

[1] "Test set AUC (CV): 0.785530621785881"

*Model comparison via ROC Curve & AUC*

For the final exercise, we’ll compare the different types of models: Decision Trees, Bagged trees, Random Forest, and Gradient Boosting Machine (GBM)

The model with the highest AUC is considered the best model

# Generate the test set AUCs using the two sets of predictions & compare

actual <- credit\_test$default

dt\_auc <- auc(actual = actual, predicted = dt\_preds)

bag\_auc <- auc(actual = actual, predicted = bag\_preds)

rf\_auc <- auc(actual = actual, predicted = rf\_preds)

gbm\_auc <- auc(actual = actual, predicted = gbm\_preds)

# Print results

> sprintf("Decision Tree Test AUC: %.3f", dt\_auc)

[1] "Decision Tree Test AUC: 0.627"

> sprintf("Bagged Trees Test AUC: %.3f", bag\_auc)

[1] "Bagged Trees Test AUC: 0.781"

> sprintf("Random Forest Test AUC: %.3f", rf\_auc)

[1] "Random Forest Test AUC: 0.804"

> sprintf("GBM Test AUC: %.3f", gbm\_auc)

[1] "GBM Test AUC: 0.786"

\*\*\*In this case, we see that the Random Forest performed the best on the test set. With a bit more tuning of the GBM, the performance might be closer to that of the Random Forest. To save time, we only used 2-fold cross-validation to choose the optimal number of trees, but we encourage you to use more folds (at least 5 or 10) in practice. With more folds, you will have a better estimate of the optimal number of trees.

The **ROCR** package provides the prediction() and performance() functions which generate the data required for plotting the ROC curve, given a set of predictions and actual (true) values.

The more "up and to the left" the ROC curve of a model is, the better the model. The AUC performance metric is literally the "Area Under the ROC Curve", so the greater the area under this curve, the higher the AUC, and the better-performing the model is.

# List of predictions

preds\_list <- list(dt\_preds, bag\_preds, rf\_preds, gbm\_preds)

# List of actual values (same for all)

m <- length(preds\_list)

actuals\_list <- rep(list(credit\_test$default), m)

# Plot the ROC curves

pred <- prediction(preds\_list,actuals\_list)

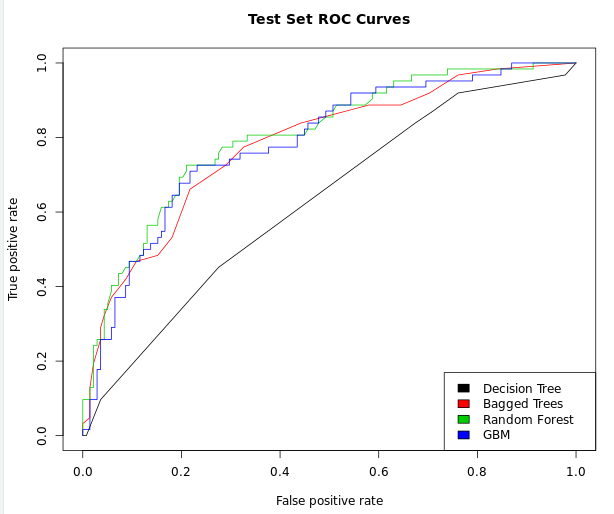
rocs <- performance(pred, "tpr", "fpr")

plot(rocs, col = as.list(1:m), main = "Test Set ROC Curves")

legend(x = "bottomright",

legend = c("Decision Tree", "Bagged Trees", "Random Forest", "GBM"),

fill = 1:m)

**

**Supervised learning in R: Regression**

**Chapter 1:**

Regression: Predict a numerical outcome (‘dependent variable’) from a set of inputs (‘independent variables’)

Different from classification (binary)

Linear Regression:

The most basic regression

Assumes that y is linearly related to x

cmodel <- lm(temperature ~ chirps\_per\_sec, data=cricket)

#formula:

temperature ~ chirps\_per\_sec (temperature is the output/dependent variable, right of ~ is your input variables, which could consist of multiple variables, concatenated by +)

data frame: cricket

# get all of the summary statistics from the model:

summary(cmodel)

*Predicting from the training data*

predict() by default returns the training data predictions

# unemployment is in your workspace

summary(unemployment)

# newrates is in your workspace

newrates

# Predict female unemployment in the unemployment data set

unemployment$prediction <- predict(unemployment\_model)

# load the ggplot2 package

library(ggplot2)

# Make a plot to compare predictions to actual (prediction on x axis)

ggplot(unemployment, aes(x = prediction, y = female\_unemployment)) +

geom\_point() +

geom\_abline(color = "blue")

# Predict female unemployment rate when male unemployment is 5%

pred <- predict(unemployment\_model, newdata = newrates)

# Print it

pred

Pros of linear regression:

easy to fit and to apply

concise

less prone to overfitting

Easy to interpret

Cons:

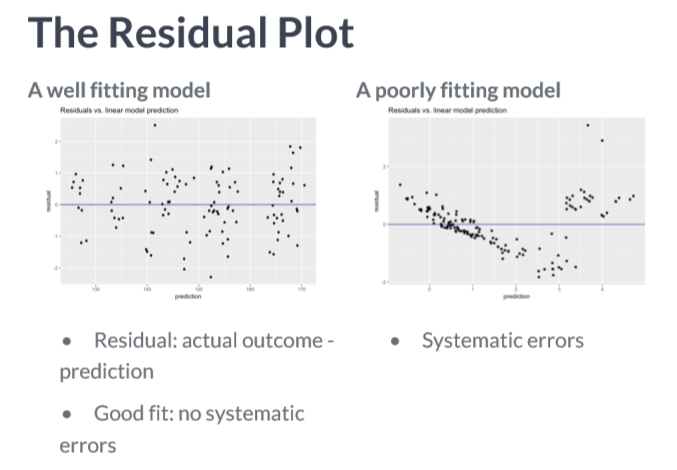
Can only express linear and additive relationships

Collinearity: when input variables are partially correlated. When highly correlated, the coefficients might change sign.

**Chapter 2: Evaluating a model graphically**

Prediction should be on X, response on Y

A residual plot. Want the same distribution of data points above and below the line.



Gain curve: useful for models that plot probability

GainCurvePlot()

Graphically evaluate the unemployment model

# unemployment, unemployment\_model are in the workspace

summary(unemployment)

summary(unemployment\_model)

# Make predictions from the model

unemployment$predictions <- predict(unemployment\_model, newdata=unemployment)

# Fill in the blanks to plot predictions (on x-axis) versus the female\_unemployment rates

ggplot(unemployment, aes(x = predictions, y = female\_unemployment)) +

geom\_point() +

geom\_abline()

# From previous step

unemployment$predictions <- predict(unemployment\_model)

# Calculate residuals

unemployment$residuals <- unemployment$female\_unemployment - unemployment$predictions

# Fill in the blanks to plot predictions (on x-axis) versus the residuals

ggplot(unemployment, aes(x = predictions, y = residuals)) +

geom\_pointrange(aes(ymin = 0, ymax = residuals)) +

geom\_hline(yintercept = 0, linetype = 3) +

ggtitle("residuals vs. linear model prediction")

# unemployment is in the workspace (with predictions)

summary(unemployment)

# unemployment\_model is in the workspace

summary(unemployment\_model)

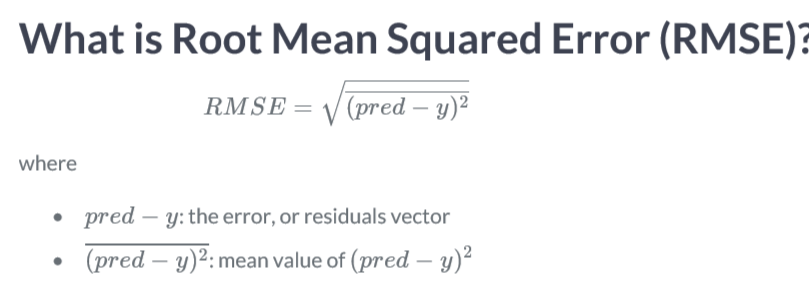
# Load the package WVPlots

library(WVPlots)

# Plot the Gain Curve

GainCurvePlot(unemployment, "predictions", "female\_unemployment", "Unemployment model")

Root mean squared error (RMSE)



How to evaluate if your RMSE is high?

If your RMSE is smaller than the SD, then your model is fairly good.

*Calculate RMSE*

# unemployment is in the workspace

summary(unemployment)

# For convenience put the residuals in the variable res

res <- unemployment$residuals

# Calculate RMSE, assign it to the variable rmse and print it

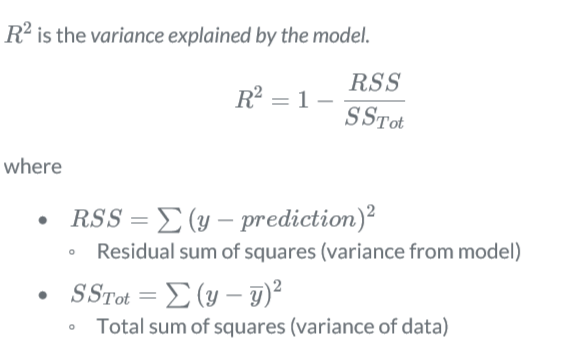
(rmse <- sqrt(mean(res^2)))

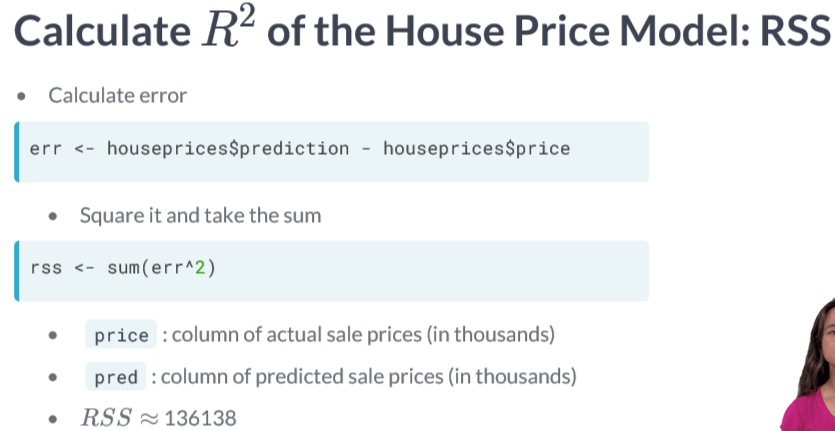
# Calculate the standard deviation of female\_unemployment and print it

(sd\_unemployment <- sd(unemployment$female\_unemployment))

What is R2

A value between 0-1. Near means model fits well.



****

Calculate R-Squared

# unemployment is in your workspace

summary(unemployment)

# unemployment\_model is in the workspace

summary(unemployment\_model)

# Calculate mean female\_unemployment: fe\_mean. Print it

(fe\_mean <- mean(unemployment$female\_unemployment))

# Calculate total sum of squares: tss. Print it

(tss <- sum((unemployment$female\_unemployment - fe\_mean)^2))

# Calculate residual sum of squares: rss. Print it

(rss <- sum(unemployment$residuals^2))

# Calculate R-squared: rsq. Print it. Is it a good fit?

(rsq <- 1-rss/tss)

# Get R-squared from glance. Print it

(rsq\_glance <- glance(unemployment\_model)$r.squared)

# Get the correlation between prediction and true outcome (rho)

(rho <- cor(unemployment$predictions,unemployment$female\_unemployment))

*Test/train split*

Create a cross-validation plan.

library(vtreat)

splitPlan <- kWayCrossValidation(nRows, nSplits, NULL, NULL)

# nRows: number of rows in the training data

# nSplits: number of folds (partitions) in the cross-validation

# remaining 2 arguments not needed.

*Generating a random test/train split*

If you have a data set dframe of size *N*N\*.75, and you want a random subset of approximately size 100∗X% of *N*(where *X* is between 0 and 1), then:

1. Generate a vector of uniform random numbers: gp = runif(N).
2. dframe[gp < X,] will be about the right size.
3. dframe[gp >= X,] will be the complement.

# Use nrow to get the number of rows in mpg (N) and print it

(N <- nrow(mpg))

# Calculate how many rows 75% of N should be and print it

# Hint: use round() to get an integer

(target <- round(N\*.75))

# Create the vector of N uniform random variables: gp

gp <- runif(N)

# Use gp to create the training set: mpg\_train (75% of data) and mpg\_test (25% of data)

mpg\_train <- mpg[gp<.75,]

mpg\_test <- mpg[gp>=.75,]

# Use nrow() to examine mpg\_train and mpg\_test

nrow(mpg\_train)

nrow(mpg\_test)

*Evaluate a modeling procedure using n-fold cross-validation*

# mpg is in the workspace

summary(mpg)

# splitPlan is in the workspace

str(splitPlan)

# Run the 3-fold cross validation plan from splitPlan

k <- 3 # Number of folds

mpg$pred.cv <- 0

for(i in 1:k) {

split <- splitPlan[[i]]

model <- lm(cty ~ hwy, data = mpg[split$train, ])

mpg$pred.cv[split$app] <- predict(model, newdata = mpg[split$app, ])

}

# Predict from a full model

mpg$pred <- predict(lm(cty ~ hwy, data = mpg))

# Get the rmse of the full model's predictions

mpg$pred, mpg$cty)

# Get the rmse of the cross-validation predictions

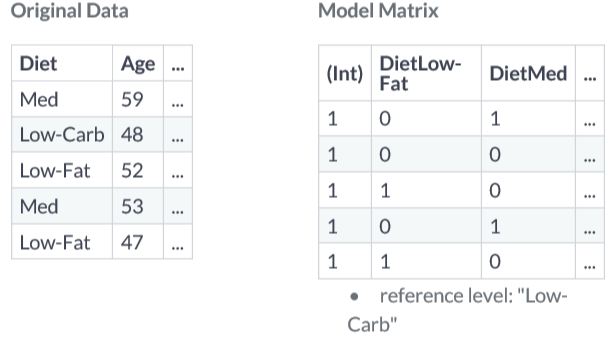
rmse(mpg$pred.cv, mpg$cty)

**Chapter 3:**

*Categorical inputs*

Categorical variables are converted to numerical values; a catergorical variable with N levels is converted into N-1 indicator variables.

One-hot encoding:



Issues with one-hot-encoding:

Too many levels can be a problem (e.g. zip codes)

Don’t hash with geometric methods! e.g. don’t call the diets variables as 1,2,3

*Examining the structure of categorical inputs:*

# Call str on flowers to see the types of each column

str(flowers)

# Use unique() to see how many possible values Time takes

unique(flowers$Time)

# Build a formula to express Flowers as a function of Intensity and Time: fmla. Print it

(fmla <- as.formula("Flowers ~ Intensity + Time"))

# Use fmla and model.matrix to see how the data is represented for modeling

mmat <- model.matrix(fmla,data=flowers)

# Examine the first 20 lines of flowers

head(flowers,n=20)

# Examine the first 20 lines of mmat

head(mmat,n=20)

# *Modeling with categorical inputs*

# flowers in is the workspace

str(flowers)

# fmla is in the workspace

fmla

# Fit a model to predict Flowers from Intensity and Time : flower\_model

flower\_model <- lm(fmla,flowers)

# Use summary on mmat to remind yourself of its structure

summary(mmat)

# Use summary to examine flower\_model

summary(flower\_model)

# Predict the number of flowers on each plant

flowers$predictions <- predict(flower\_model)

# Plot predictions vs actual flowers (predictions on x-axis)

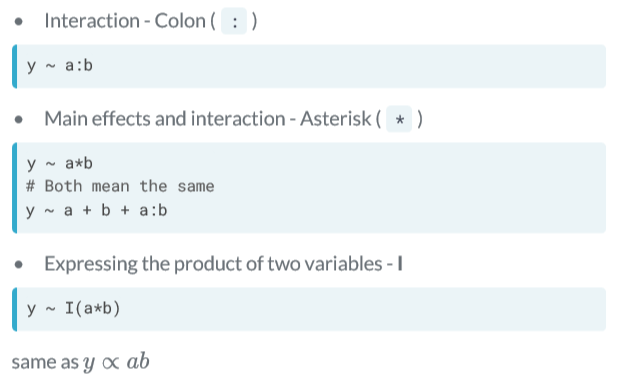
ggplot(flowers, aes(x = predictions, y = Flowers)) +

geom\_point() +

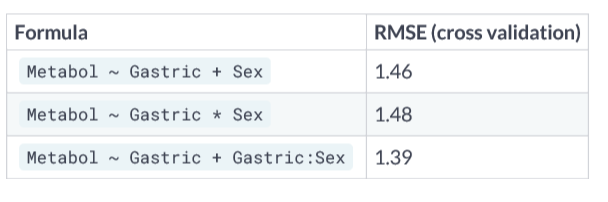
geom\_abline(color = "blue")

*Variable Interactions*

What if there is an interaction between 2 or more variables?



Finding the correct interaction pattern:



There are some regression methods that can learn the interactions, so that we don’t have to encode them ourselves.

*Modeling an interaction:*

# alcohol is in the workspace

summary(alcohol)

# Both the formulae are in the workspace

fmla\_add

fmla\_interaction

# Create the splitting plan for 3-fold cross validation

set.seed(34245) # set the seed for reproducibility

splitPlan <- kWayCrossValidation(nrow(alcohol), 3, NULL, NULL)

# Sample code: Get cross-val predictions for main-effects only model

alcohol$pred\_add <- 0 # initialize the prediction vector

for(i in 1:3) {

split <- splitPlan[[i]]

model\_add <- lm(fmla\_add, data = alcohol[split$train, ])

alcohol$pred\_add[split$app] <- predict(model\_add, newdata = alcohol[split$app, ])

}

# Get the cross-val predictions for the model with interactions

alcohol$pred\_interaction <- 0 # initialize the prediction vector

for(i in 1:3) {

split <- splitPlan[[i]]

model\_interaction <- lm(fmla\_interaction, data = alcohol[split$train, ])

alcohol$pred\_interaction[split$app] <- predict(model\_interaction, newdata = alcohol[split$app, ])

}

# Get RMSE

alcohol %>%

gather(key = modeltype, value = pred, pred\_add, pred\_interaction) %>%

mutate(residuals = Metabol - pred) %>%

group\_by(modeltype) %>%

summarize(rmse = sqrt(mean(residuals^2)))

*Transforming the response before modeling*

The log transform for monetary data

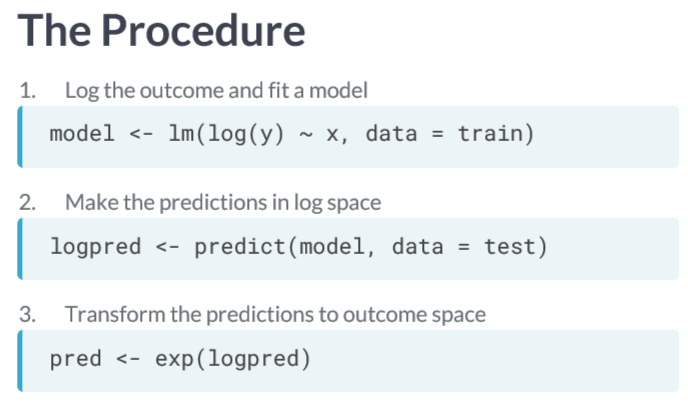
Monetary values: lognormally distributed

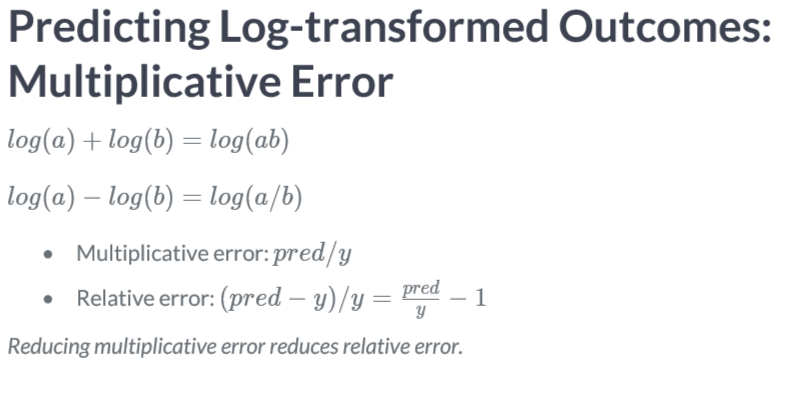
Long tail, wide dynamic range (e.g. 60-700k)

Mean is typically greater than the median

Predicting the mean will overpredict typical values

If you take log of lognormally distrubited data, than the mean tends to follow the median more closely





*Comparing RMSE and root-mean-squared relative error:*

# fmla.abs is in the workspace

fmla.abs

# model.abs is in the workspace

summary(model.abs)

# Add predictions to the test set

income\_test <- income\_test %>%

mutate(pred.absmodel = predict(model.abs, income\_test), # predictions from model.abs

pred.logmodel = exp(predict(model.log, income\_test))) # predictions from model.log

# Gather the predictions and calculate residuals and relative error

income\_long <- income\_test %>%

gather(key = modeltype, value = pred, pred.absmodel, pred.logmodel) %>%

mutate(residual = pred - Income2005, # residuals

relerr = residual / Income2005) # relative error

# Calculate RMSE and relative RMSE and compare

income\_long %>%

group\_by(modeltype) %>% # group by modeltype

summarize(rmse = sqrt(mean(residual^2)), # RMSE

rmse.rel = sqrt(mean(relerr^2))) # Root mean squared relative error

Why transform input variables

1. Domain knowledge/synthetic variables
   1. Intelligence ~ mass.brain/mass.body2/3
2. Pragmatic reasons
   1. reduce dynamic range
   2. meaningful changes in variable are multiplicative
   3. y approximately linear in f(x) rather than in x

Different possible fits:

Which is best?

*Input transformations: the hockey stick*

# houseprice is in the workspace

summary(houseprice)

# Create the formula for price as a function of squared size

(fmla\_sqr <- price ~ I(size^2))

# Fit a model of price as a function of squared size (use fmla\_sqr)

model\_sqr <- lm(fmla\_sqr,data=houseprice)

# Fit a model of price as a linear function of size

model\_lin <- lm(price~size,data=houseprice)

# Make predictions and compare

houseprice %>%

mutate(pred\_lin = predict(model\_lin,data=houseprice), # predictions from linear model

pred\_sqr = predict(model\_sqr,data=houseprice)) %>% # predictions from quadratic model

gather(key = modeltype, value = pred, pred\_lin, pred\_sqr) %>% # gather the predictions

ggplot(aes(x = size)) +

geom\_point(aes(y = price)) + # actual prices

geom\_line(aes(y = pred, color = modeltype)) + # the predictions

scale\_color\_brewer(palette = "Dark2")

# houseprice is in the workspace

summary(houseprice)

# fmla\_sqr is in the workspace

fmla\_sqr

# Create a splitting plan for 3-fold cross validation

set.seed(34245) # set the seed for reproducibility

splitPlan <- kWayCrossValidation(nrow(houseprice),3,NULL,NULL)

# Sample code: get cross-val predictions for price ~ size

houseprice$pred\_lin <- 0 # initialize the prediction vector

for(i in 1:3) {

split <- splitPlan[[i]]

model\_lin <- lm(price ~ size, data = houseprice[split$train,])

houseprice$pred\_lin[split$app] <- predict(model\_lin, newdata = houseprice[split$app,])

}

# Get cross-val predictions for price as a function of size^2 (use fmla\_sqr)

houseprice$pred\_sqr <- 0 # initialize the prediction vector

for(i in 1:3) {

split <- splitPlan[[i]]

model\_sqr <- lm(fmla\_sqr, data = houseprice[split$train, ])

houseprice$pred\_sqr[split$app] <- predict(model\_sqr, newdata = houseprice[split$app, ])

}

# Gather the predictions and calculate the residuals

houseprice\_long <- houseprice %>%

gather(key = modeltype, value = pred, pred\_lin, pred\_sqr) %>%

mutate(residuals =pred-price)

# Compare the cross-validated RMSE for the two models

houseprice\_long %>%

group\_by(modeltype) %>% # group by modeltype

summarize(rmse = sqrt(mean(residuals^2)))

**Chapter 4: Logistic Regression to Predict Probabilities**

**Predicting probabilities:**

* Predicting whether an event occurs (yes/no): classification
* Predicting the probability that an event occurs: regression
* Linear regression: predict values in [-infinity, infinity]
* Probabilities: limited to [0,1], interval
  + So we’ll call it non-linear

**Chapter 5:**