Data Mining Final Project

Robert Ward

December 22, 2017

Note

This was my final project for "Data Mining for the Social Sciences." It is not mean to be a traditional social science paper; it is more of a record of a data analysis process, in which I download and format a dataset, split it into training and test sets, and iterate through many different models in search of the best possible predictions. My other writing sample is a more traditional paper.

Introduction

In this project, I predict the scores of board games, using a dataset scraped from boardgamegeek.com's game database. BoardGameGeek is the main online hub of the board game community, with an enormous database of games, including various information about each game, some of it from the game itself (playing time, recommended age, etc.) and much of it from users (ratings, rules complexity ("weight") scores, recommended number of players, and more). Being able to predict which kinds of games will get good scores is, admittedly, not an issue of utmost importance to our society, but it could help guide aspiring game designers or inform publishers' decisions about which games to release.

The dataset comes from https://www.kaggle.com/gabrio/board-games-dataset. Here, it is imported in its R package form, which is available at https://github.com/9thcirclegames/bgg-analysis. Most of the data processing code chunks below also come from this github page.

Data Import and Processing

Loading packages and installing the package containing the dataset.

```
# Requirements and import
set.seed(12345)

options(java.parameters = "-Xmx5g")

if (!require("pacman")) install.packages("pacman")
```

```
## Loading required package: pacman
```

```
pacman::p_load("tidyverse",
               "arules",
               "dendextend",
               "dummies",
               "splitstackshape",
               "DT",
               "topicmodels",
               "bggAnalysis",
               "randomForest",
               "caret",
               "lars",
               "Matrix",
               "flam",
```

```
                "gam",
                "bartMachine",
                "stargazer",
                "ggplot2",
                "robustbase",
                "gbm")

devtools::install_github("9thcirclegames/bgg-analysis")
```

```
## Skipping install of 'bggAnalysis' from a github remote, the SHA1 (cef8a5b8) has not changed since las
##   Use `force = TRUE` to force installation
```

```
rmse.bgg <- function(model){
  pred <- predict(model, newdata = test)
  return(sqrt(mean((test$stats.average - pred) ^ 2, na.rm = T)))
}
```

Filtering the Data

This code, taken entirely from the package author, filters out a significant part of the dataset that is of less interest for predicting the scores of modern board games. It removes:

- Games with less than five ratings, which are often unpublished or homebrew projects, or which simply don't have enough ratings to have a meaningful score;
- Expansions to existing games, which share virtually all qualities with their base game (and thus do not add meaningful variance to the sample) and are likely judged differently than standalone games;
- Very old games, which are often "classic" games that are, again, likely judged differently from modern games in their ratings;
- Games from the last year, whose ratings may not have settled to their long-term average yet;
- Video games, which are... not board games.

```
data("BoardGames")

bgg.useful <- BoardGames %>%
  bgg.prepare.data() %>%
  filter(!is.na(details.yearpublished)) %>%
  filter(details.yearpublished <= 2016) %>%
  filter(details.yearpublished >= 1960) %>%
  filter(stats.usersrated >= 5, game.type == "boardgame") %>%
  mutate(stats.average.factor = discretize(stats.average,
                                           method="frequency",
                                           categories = 5,
                                           ordered = TRUE))

rownames(bgg.useful) <- make.names(bgg.useful$details.name, unique=TRUE)
```

Creating dummy variables

This is another chunk of code taken from the package author. The dataset has a number of string variables with a series of comma-separated tags describing each game's mechanics and categories; this separates them into individual dummy variables. These very sparse dummies are used, essentially, as a document term matrix for latent Dirichlet analysis below.

```
bgg.dummy <- cSplit_e(bgg.useful, "attributes.boardgamecategory", type="character", fill=0, drop=TRUE)
bgg.dummy <- cSplit_e(bgg.dummy, "attributes.boardgamemechanic", type="character", fill=0, drop=TRUE)
bgg.dummy <- cSplit_e(bgg.dummy, "attributes.boardgamefamily", type="character", fill=0, drop=TRUE)
bgg.dummy <- cSplit_e(bgg.dummy, "attributes.boardgameimplementation", type="character", fill=0, drop=TR

colnames(bgg.dummy) <- gsub(" ", "", colnames(bgg.dummy))
colnames(bgg.dummy) <- gsub("/", "-", colnames(bgg.dummy))
colnames(bgg.dummy) <- gsub("_", ".", colnames(bgg.dummy))

bgg.dummy <- cbind(
  bgg.dummy
  ,dummies::dummy("details.minplayers.factor", bgg.dummy, sep="=")
  ,dummies::dummy("details.maxplayers.factor", bgg.dummy, sep="=")
  ,dummies::dummy("details.playingtime.factor", bgg.dummy, sep="=")
  ,dummies::dummy("details.minage.factor", bgg.dummy, sep="=")
  ,dummies::dummy("stats.weight.factor", bgg.dummy, sep="=")
  ,dummies::dummy("stats.average.factor", bgg.dummy, sep="=")
  ,dummies::dummy("polls.language.dependence", bgg.dummy, sep="=")
)

colnames(bgg.dummy) <- make.names(colnames(bgg.dummy))

#

bgg.dummy.cat <- bgg.dummy %>% select(matches("attributes.boardgame(category|mechanic|family|implementa

# cutting out anything that has no tags

bgg.dummy.cat <- bgg.dummy.cat[which(rowSums(bgg.dummy.cat != 0)>=1),]

colnames(bgg.dummy.cat) <- gsub("attributes\\.boardgame(category|mechanic|family|implementation)\\.(.*)
```

Latent Dirichlet Allocation

Latent Dirichlet allocation is used to reduce the thousands of tag dummy variables down to a more manageable number of - essentially - categories or genres of board games. This code, again, was taken mostly from the package author; I have made some modifications to reduce execution time (reducing the number of iterations) and added a second LDA output that uses 15 instead of 25 categories. While the package author manually inspected his 25 categories and found them to be largely coherent and meaningful, they also appeared to split genres unecessarily, producing (for instance) three separate genres for wargames representing different historical periods.

I use both the 15-topic and 25-topic models below and find minimal differences between their predictive performance, although both are useful.

As is shown below, the predicted probabilities for nearly all categories for most games are very low, perhaps thanks to the sparsity of the DTM and the fairly high number of topics. However, the topic model does produce what seem to be coherent categories, including groups of war games, card games, party games, etc.

```
#Set parameters for Gibbs sampling
gibbs.control.small <- list(burnin = 1000,
                      iter = 250,
                      thin = 125,
                      seed = list(2003,5,63,100001,765,287,899,101,49,3),
```

```
                      nstart = 10,
                      best = TRUE)


#Run LDA using Gibbs sampling
bgg.ldaOut <-LDA(bgg.dummy.cat,
                 k=25,
                 method="Gibbs",
                 control=gibbs.control.small)

bgg.ldaOut.15 <-LDA(bgg.dummy.cat,
                 k=15,
                 method="Gibbs",
                 control=gibbs.control.small)


round(posterior(bgg.ldaOut, bgg.dummy.cat)$topics[1:30, ], digits = 3)
```

```
##                                1     2     3     4     5     6     7
## Die.Macher                 0.033 0.033 0.033 0.050 0.033 0.033 0.033
## Dragonmaster               0.036 0.036 0.036 0.036 0.036 0.054 0.036
## Samurai                    0.033 0.033 0.033 0.033 0.033 0.033 0.033
## Tal.der.Könige             0.035 0.035 0.035 0.035 0.053 0.035 0.053
## Acquire                    0.036 0.036 0.036 0.036 0.036 0.036 0.036
## Mare.Mediterraneum         0.037 0.037 0.037 0.037 0.037 0.037 0.037
## Cathedral                  0.035 0.035 0.035 0.035 0.035 0.053 0.035
## Lords.of.Creation          0.038 0.038 0.038 0.038 0.038 0.038 0.038
## El.Caballero               0.037 0.037 0.037 0.037 0.037 0.037 0.037
## Elfenland                  0.050 0.033 0.033 0.050 0.033 0.033 0.033
## Bohnanza                   0.047 0.031 0.031 0.031 0.031 0.031 0.031
## Ra                         0.032 0.032 0.032 0.048 0.032 0.032 0.032
## Catan                      0.027 0.041 0.027 0.054 0.054 0.027 0.054
## Basari                     0.069 0.052 0.034 0.034 0.052 0.034 0.034
## Cosmic.Encounter           0.034 0.052 0.034 0.034 0.034 0.034 0.052
## MarraCash                  0.036 0.036 0.036 0.036 0.036 0.055 0.036
## Button.Men                 0.050 0.033 0.033 0.067 0.033 0.033 0.033
## RoboRally                  0.034 0.034 0.034 0.034 0.051 0.034 0.068
## Wacky.Wacky.West           0.036 0.036 0.036 0.036 0.036 0.055 0.036
## Full.Metal.Planète         0.056 0.037 0.037 0.056 0.037 0.037 0.056
## Gateway.to.the.Stars       0.037 0.037 0.037 0.037 0.037 0.037 0.056
## Magic.Realm                0.033 0.033 0.033 0.049 0.033 0.033 0.033
## Divine.Right               0.036 0.036 0.054 0.054 0.036 0.036 0.036
## Twilight.Imperium          0.046 0.031 0.046 0.031 0.031 0.031 0.108
## Battlemist                 0.036 0.036 0.054 0.036 0.036 0.036 0.036
## Age.of.Renaissance         0.034 0.034 0.034 0.034 0.034 0.034 0.034
## Supremacy                  0.036 0.036 0.054 0.036 0.036 0.036 0.036
## Illuminati...Deluxe.Edition 0.033 0.033 0.033 0.033 0.066 0.033 0.033
## Terrain.Vague              0.055 0.036 0.036 0.036 0.036 0.036 0.036
## Dark.Tower                 0.034 0.034 0.034 0.034 0.034 0.034 0.052
##                                8     9    10    11    12    13    14
## Die.Macher                 0.033 0.033 0.033 0.067 0.067 0.033 0.050
## Dragonmaster               0.036 0.071 0.071 0.036 0.036 0.036 0.036
## Samurai                    0.033 0.033 0.033 0.033 0.117 0.033 0.033
## Tal.der.Könige             0.035 0.035 0.035 0.070 0.053 0.035 0.035
```

```
## Acquire                     0.036 0.036 0.036 0.073 0.036 0.036 0.036
## Mare.Mediterraneum          0.037 0.037 0.037 0.037 0.074 0.037 0.056
## Cathedral                   0.035 0.035 0.035 0.035 0.035 0.035 0.035
## Lords.of.Creation           0.057 0.038 0.057 0.038 0.057 0.038 0.038
## El.Caballero                0.056 0.037 0.056 0.037 0.056 0.037 0.037
## Elfenland                   0.067 0.033 0.033 0.033 0.033 0.033 0.033
## Bohnanza                    0.031 0.047 0.031 0.078 0.047 0.047 0.031
## Ra                          0.032 0.032 0.032 0.081 0.032 0.032 0.032
## Catan                       0.027 0.041 0.041 0.095 0.041 0.027 0.027
## Basari                      0.034 0.034 0.034 0.052 0.052 0.034 0.034
## Cosmic.Encounter            0.052 0.052 0.034 0.052 0.034 0.052 0.034
## MarraCash                   0.036 0.055 0.036 0.055 0.036 0.036 0.036
## Button.Men                  0.083 0.050 0.033 0.033 0.050 0.033 0.033
## RoboRally                   0.034 0.034 0.051 0.034 0.034 0.034 0.068
## Wacky.Wacky.West            0.036 0.036 0.036 0.036 0.055 0.073 0.036
## Full.Metal.Planète          0.037 0.056 0.037 0.037 0.037 0.037 0.037
## Gateway.to.the.Stars        0.037 0.037 0.056 0.037 0.056 0.056 0.037
## Magic.Realm                 0.066 0.033 0.115 0.033 0.033 0.033 0.066
## Divine.Right                0.054 0.036 0.036 0.036 0.071 0.036 0.036
## Twilight.Imperium           0.031 0.031 0.046 0.046 0.062 0.046 0.031
## Battlemist                  0.036 0.036 0.071 0.036 0.054 0.036 0.036
## Age.of.Renaissance          0.034 0.034 0.034 0.068 0.153 0.034 0.034
## Supremacy                   0.036 0.036 0.036 0.036 0.054 0.036 0.036
## Illuminati...Deluxe.Edition 0.049 0.049 0.033 0.049 0.033 0.033 0.033
## Terrain.Vague               0.055 0.055 0.055 0.036 0.036 0.036 0.036
## Dark.Tower                  0.052 0.034 0.086 0.034 0.034 0.034 0.034
##                                15    16    17    18    19    20    21
## Die.Macher                  0.033 0.033 0.067 0.033 0.033 0.033 0.033
## Dragonmaster                0.036 0.036 0.036 0.054 0.036 0.036 0.036
## Samurai                     0.050 0.067 0.033 0.033 0.033 0.033 0.033
## Tal.der.Könige              0.035 0.035 0.035 0.035 0.070 0.035 0.035
## Acquire                     0.036 0.055 0.036 0.036 0.036 0.036 0.073
## Mare.Mediterraneum          0.037 0.037 0.037 0.037 0.037 0.037 0.037
## Cathedral                   0.035 0.140 0.035 0.035 0.035 0.035 0.035
## Lords.of.Creation           0.038 0.038 0.038 0.038 0.038 0.038 0.038
## El.Caballero                0.037 0.056 0.037 0.037 0.037 0.037 0.037
## Elfenland                   0.050 0.033 0.067 0.033 0.033 0.033 0.067
## Bohnanza                    0.078 0.047 0.031 0.031 0.031 0.047 0.031
## Ra                          0.032 0.048 0.065 0.032 0.065 0.032 0.048
## Catan                       0.095 0.027 0.041 0.027 0.041 0.041 0.041
## Basari                      0.034 0.034 0.034 0.034 0.034 0.034 0.034
## Cosmic.Encounter            0.034 0.034 0.034 0.034 0.034 0.034 0.034
## MarraCash                   0.055 0.036 0.055 0.036 0.036 0.036 0.036
## Button.Men                  0.033 0.033 0.033 0.033 0.033 0.033 0.033
## RoboRally                   0.034 0.051 0.034 0.034 0.051 0.034 0.034
## Wacky.Wacky.West            0.036 0.036 0.036 0.036 0.036 0.036 0.036
## Full.Metal.Planète          0.037 0.037 0.037 0.037 0.037 0.037 0.037
## Gateway.to.the.Stars        0.037 0.037 0.037 0.037 0.037 0.037 0.037
## Magic.Realm                 0.033 0.033 0.033 0.033 0.033 0.049 0.033
## Divine.Right                0.036 0.036 0.036 0.036 0.036 0.036 0.036
## Twilight.Imperium           0.031 0.046 0.031 0.046 0.031 0.031 0.031
## Battlemist                  0.036 0.071 0.036 0.036 0.036 0.036 0.036
## Age.of.Renaissance          0.034 0.034 0.034 0.034 0.034 0.034 0.034
## Supremacy                   0.036 0.036 0.071 0.036 0.036 0.036 0.036
```

```
## Illuminati...Deluxe.Edition 0.049 0.049 0.049 0.033 0.049 0.033 0.049
## Terrain.Vague               0.036 0.036 0.055 0.036 0.036 0.036 0.036
## Dark.Tower                  0.034 0.034 0.034 0.034 0.052 0.052 0.034
##                                22    23    24    25
## Die.Macher                  0.033 0.067 0.033 0.033
## Dragonmaster                0.036 0.036 0.036 0.036
## Samurai                     0.033 0.067 0.033 0.033
## Tal.der.Könige              0.035 0.035 0.035 0.035
## Acquire                     0.036 0.036 0.036 0.036
## Mare.Mediterraneum          0.037 0.037 0.037 0.056
## Cathedral                   0.035 0.035 0.035 0.035
## Lords.of.Creation           0.038 0.038 0.038 0.038
## El.Caballero                0.037 0.037 0.037 0.037
## Elfenland                   0.033 0.033 0.050 0.033
## Bohnanza                    0.047 0.047 0.031 0.031
## Ra                          0.032 0.032 0.048 0.048
## Catan                       0.027 0.027 0.027 0.027
## Basari                      0.034 0.052 0.052 0.034
## Cosmic.Encounter            0.034 0.052 0.034 0.052
## MarraCash                   0.036 0.036 0.036 0.036
## Button.Men                  0.033 0.033 0.033 0.067
## RoboRally                   0.034 0.034 0.051 0.034
## Wacky.Wacky.West            0.036 0.055 0.036 0.036
## Full.Metal.Planète          0.037 0.037 0.037 0.037
## Gateway.to.the.Stars        0.037 0.037 0.037 0.037
## Magic.Realm                 0.033 0.033 0.033 0.033
## Divine.Right                0.054 0.036 0.036 0.036
## Twilight.Imperium           0.046 0.031 0.031 0.031
## Battlemist                  0.036 0.036 0.036 0.036
## Age.of.Renaissance          0.034 0.034 0.034 0.034
## Supremacy                   0.054 0.036 0.036 0.054
## Illuminati...Deluxe.Edition 0.033 0.033 0.033 0.049
## Terrain.Vague               0.036 0.036 0.036 0.036
## Dark.Tower                  0.034 0.034 0.034 0.052
```

```
#very low probs... but they make some sense.
  #probably because of sparsity of tag matrix
```

This code adds the topic assignments to the main dataset.

```
#create list of topics
bgg.ldaOut.main.topics.df <- as.data.frame(topics(bgg.ldaOut))
bgg.ldaOut.15.main.topics.df <- as.data.frame(topics(bgg.ldaOut.15))
colnames(bgg.ldaOut.main.topics.df) <- "topic"
colnames(bgg.ldaOut.15.main.topics.df) <- "topic.15"

#create variable to merge on from rownames
bgg.useful$joinkey <- rownames(bgg.useful)
bgg.ldaOut.main.topics.df$joinkey <- rownames(bgg.ldaOut.main.topics.df)
bgg.ldaOut.15.main.topics.df$joinkey <- rownames(bgg.ldaOut.15.main.topics.df)

bgg.topics <- left_join(bgg.useful, bgg.ldaOut.main.topics.df, by = 'joinkey')
bgg.topics <- left_join(bgg.topics, bgg.ldaOut.15.main.topics.df, by = 'joinkey')

bgg.topics$topic <- as.factor(bgg.topics$topic)
```

```
bgg.topics$topic.15 <- as.factor(bgg.topics$topic.15)
```

More new variables

This code creates an "age" variable indicating the number of years since a game was published.

It also makes simplified versions of the factor variables that contain information about user-voted suggested numbers of players for each game, combining the "best with" and "recommended with" responses and recoding missings as "not recommended." This is a somewhat questionable move, but it does greatly improve the predictive performance of models using these variables, although, ultimately, models without any of them perform better.

```
bgg.topics$details.age <- 2017 - bgg.topics$details.yearpublished

#Create new version of numplayers factors that just have "no" and "yes", with NAs set to "no"
bgg.topics.new <- bgg.topics %>% select(starts_with("polls.suggested_numplayers"), joinkey)
bgg.topics.new[is.na(bgg.topics.new)] <- "NotRecommended"
bgg.topics.new <- bgg.topics.new %>% mutate_all(funs(recode_factor(., NotRecommended = "No", Recommende
colnames(bgg.topics.new) <- gsub("numplayers", "numplayers_simple", colnames(bgg.topics.new))

bgg.topics <- left_join(bgg.topics, bgg.topics.new, by = "joinkey")
```

```
## Warning: Column `joinkey` joining character vector and factor, coercing
## into character vector
```

```
#remove join key and bgg.topics.new
bgg.topics <- bgg.topics %>% select(-joinkey)
rm(bgg.topics.new)
```

Training and Test Sets

The data is split into training and test sets.

```
set.seed(12345)

in_train <- createDataPartition(y = bgg.topics$stats.average, p = 3 / 4, list = FALSE)
train <- bgg.topics[ in_train, ]
test  <- bgg.topics[-in_train, ]
```

Prediction

OLS

After running a variety of OLS models, the best performance (RMSE of 0.7050208) appears to come from using all of the plausible predictors except for the "suggested number of players" polls. In their raw form (not shown), these variables have so many missing values that the model only has roughly 300 out of about 20,000 observations to work with, and the predictions suffer, as expected. In their simplified form, they produce much better predictions, but models that do not include them are still more accurate.

The best model here uses the 25-group LDA variable, but the 15-group variable performs only slightly worse.

These models do not include three variables that increase predictive accuracy but could not be used for true out-of-sample prediction on new games: the numbers of users who rated the game, own the game, and commented on the game. All three are measures of popularity that, unsurprisingly, drive down the RMSE,

but could not be measured or estimated before a game was published. While some of the other data in these models is also crowdsourced, it would be possible to get a reasonable estimate of rules complexity and language dependence before a game was released.

```
#Big model, no BGG rating/ownership vars
big.bgg.simple.pre <- lm(stats.average ~ topic + details.maxplayers + details.maxplaytime + details.mina
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence
   + polls.suggested_numplayers_simple.1 + polls.suggested_numplayers_simple.2 + polls.suggested_numplay
   + polls.suggested_numplayers_simple.4 + polls.suggested_numplayers_simple.5 + polls.suggested_numplay
   + polls.suggested_numplayers_simple.7 + polls.suggested_numplayers_simple.8 + polls.suggested_numplay
   + polls.suggested_numplayers_simple.10 + polls.suggested_numplayers_simple.Over, data = train)

rmse.bgg(big.bgg.simple.pre) #0.7423041
```

```
## [1] 0.7423041
```

```
#A smaller model that removes the suggested number of players polls. Quite a bit better.
lm.pre.nonumplay <- lm(stats.average ~ topic + details.maxplayers + details.maxplaytime + details.minage
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train)

rmse.bgg(lm.pre.nonumplay) #0.7050208
```

```
## [1] 0.7050208
```

```
#Same as previous, but with 15 topics instead of 25.
  #Predictions are marginally worse
lm.pre.nonumplay.15 <- lm(stats.average ~ topic.15 + details.maxplayers + details.maxplaytime + details
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train)

# summary(lm.pre.nonumplay.15)
rmse.bgg(lm.pre.nonumplay.15) #0.7082523
```

```
## [1] 0.7082523
```

```
#What about a smaller model, without playerage or language dependence?
  #Much worse.
lm.smaller.15.pre <- lm(stats.average ~ topic.15 + details.maxplayers + details.maxplaytime + details.mi
   + details.minplaytime + details.age + details.playingtime + stats.averageweight, data = train)

rmse.bgg(lm.smaller.15.pre) #0.9017884
```

```
## [1] 0.9017884
```

```
#Far worse than the no-interaction models, rank-deficient fit, need to step this down

lm.ints.15.pre <- lm(stats.average ~ (topic.15 + details.maxplayers + details.maxplaytime + details.mina
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage)^2, data = train)

rmse.bgg(lm.ints.15.pre) #0.7898109
```

```
## Warning in predict.lm(model, newdata = test): prediction from a rank-
## deficient fit may be misleading
```

```
## [1] 0.7898109
```

Feature selection

A model with all the predictors used in the best model above, plus all of their interactions, leads to a rank-deficient fit and somewhat higher prediction error (although with other random seeds, it sometimes produces absurdly high RMSEs of 4 or more). Here, step() is used to find the best subset of that model. That model, which estimates 182 coefficients to the previous best model's 37, actually predicts worse than the all-direct-and-interaction-effects model.

Next, step() was used to find the best subset of the reigning best model. This produces a very small improvement in RMSE (down to 0.7049926), by droppping maxplayers and maxplaytime, producing the new best model.

```
#Better model than the kitchen-sink version it comes from, but not as good as hand-selected simple mode
lm_subset <- step(lm.ints.15.pre, trace = FALSE)

rmse.bgg(lm_subset) #0.8454722
```

```
## [1] 0.8454722
```

```
step_formula <- lm_subset$call[[2]]

length(lm_subset$coefficients)
```

```
## [1] 178
```

```
length(lm.pre.nonumplay.15$coefficients)
```

```
## [1] 37
```

```
#What if we give our best model so far to step?
lm_subset_smaller <- step(lm.pre.nonumplay, trace = FALSE)
rmse.bgg(lm_subset_smaller) #0.7049926
```

```
## [1] 0.7049926
```

```
#Drops maxplayers and maxplaytime. Looks like they're unnecessary.
setdiff(names(coef(lm.pre.nonumplay)), names(coef(lm_subset_smaller)))
```

```
## [1] "details.maxplayers"  "details.maxplaytime"
```

```
step_smaller_formula <- lm_subset_smaller$call[[2]]
```

Penalized Linear Models

Lasso regression improves slightly on the results of step(), nudging the minimum RMSE down to 0.7041202. These results use the much larger step() model as a starting point; the step with the lowest error had estimated 85 out of 177 coefficients as zero.

Forward stagewise regression performs very similarly to lasso, but does not quite match its lowest RMSE.

```
### Lasso

# Medium-sized model that performed best in lm

X_pre_nonum <- model.matrix(lm.pre.nonumplay)[ , -1]

Y_pre_nonum <- bgg.topics[as.integer(rownames(X_pre_nonum)), ]$stats.average
```

```r
lasso_pre_nonum_15 <- lars(X_pre_nonum, Y_pre_nonum, type = "lasso", intercept = TRUE)

X_star_lasso_small <- model.matrix(lm.pre.nonumplay, data = test)[,-1]
Y_test_nonum_pre <- bgg.topics[as.integer(rownames(X_star_lasso_small)), ]$stats.average

pred_lasso_pre_nonum_15 <- predict(lasso_pre_nonum_15, newx = X_star_lasso_small, type = "fit")
min(sqrt(colMeans( (Y_test_nonum_pre - pred_lasso_pre_nonum_15$fit) ^ 2 ))) #0.7049247
```

```
## [1] 0.7049247
```

```r
# summary(lasso_pre_nonum_15)

##Starting with the larger step model

X_step_big <- model.matrix(lm_subset)[ , -1]

Y_step_big <- bgg.topics[as.integer(rownames(X_step_big)), ]$stats.average

lasso_step_big <- lars(X_step_big, Y_step_big, type = "lasso", intercept = TRUE)

which.min(lasso_step_big$Cp)
```

```
## 220
## 221
```

```r
X_star_lasso_big <- model.matrix(lm_subset, data = test)[,-1]
Y_test_lasso_big <- bgg.topics[as.integer(rownames(X_star_lasso_big)), ]$stats.average

pred_lasso_step_big <- predict(lasso_step_big, newx = X_star_lasso_big, type = "fit")
min(sqrt(colMeans( (Y_test_lasso_big - pred_lasso_step_big$fit) ^ 2 ))) #0.7041202
```

```
## [1] 0.7041202
```

```r
best_lasso_row <- which.min(sqrt(colMeans( (Y_test_lasso_big - pred_lasso_step_big$fit) ^ 2 )))

#How many variables and how many zeroes?
length(coef(lasso_step_big)[best_lasso_row , ])
```

```
## [1] 177
```

```r
sum(coef(lasso_step_big)[best_lasso_row , ] == 0)
```

```
## [1] 85
```

```r
#Forward Stagewise

fs_big <- lars(X_step_big, Y_step_big, type = "forward.stagewise", intercept = TRUE)
names(fs_big)
```

```
##  [1] "call"       "type"       "df"         "lambda"     "R2"
##  [6] "RSS"        "Cp"         "actions"    "entry"      "Gamrat"
## [11] "arc.length" "Gram"       "beta"       "mu"         "normx"
## [16] "meanx"
```

```r
which.min(lasso_step_big$Cp) #209-210
```

```
## 220
## 221
```

```
pred_fs_big <- predict(fs_big, newx = X_star_lasso_big, type = "fit")
min(sqrt(colMeans( (Y_test_lasso_big - pred_fs_big$fit) ^ 2 ))) #0.704442
```

```
## [1] 0.704442
```

```
#FS again with smaller X
fs_small <- lars(X_pre_nonum, Y_pre_nonum, type = "forward.stagewise", intercept = TRUE)

pred_fs_small <- predict(fs_small, newx = X_star_lasso_small, type = "fit")
min(sqrt(colMeans( (Y_test_nonum_pre - pred_fs_small$fit) ^ 2 ))) #0.704719
```

```
## [1] 0.704719
```

Robust regression

Robust regression performs poorly.

```
robust_small <- lmrob(stats.average ~ topic.15 + details.maxplayers + details.maxplaytime + details.mina
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train, fast.s.large.n = Inf)
```

```
## Warning in lmrob.S(x, y, control = control, mf = mf): S refinements did not
## converge (to refine.tol=1e-07) in 200 (= k.max) steps

## Warning in lmrob.S(x, y, control = control, mf = mf): S refinements did not
## converge (to refine.tol=1e-07) in 200 (= k.max) steps
```

```
rmse.bgg(robust_small) #0.7813466
```

```
## [1] 0.7858521
```

Nonlinear regression

GAM performs the best of any model used so far, with a RMSE of 0.6895108, although this is not very far below lasso.

The output of plot.gam() is not shown here due to conflicts between gam and mgcv, despite mgcv not being explicitly loaded, that proved intractable during knitting. Based on these plots, however, it appears that some predictors - most notably minimum age recommended for the game and the number of years since the game was released - have nonlinear relationships with the expected score, which GAM with splines fits better than the linear models used so far. The two nonlinear variables mentioned above appear so nonlinear in part because of a few very high outliers; however, after testing some linear models without these outliers, it appears that their presence does not hurt predictive accuracy.

Cross-validated FLAM, unfortunately, takes too long to run.

```
# GAM
m_gam <- gam::gam(stats.average ~ topic + s(details.minage) + s(details.minplaytime) +
    s(details.age) + s(details.playingtime) + s(stats.averageweight) +
    polls.language_dependence + polls.suggested_playerage, data = train)

rmse.bgg(m_gam) #0.6895108
```

```
## [1] 0.6895108
```

Tree methods

Random forests perform well, lowering the best RMSE from 0.6895108 to 0.6794091. The best model uses all of the valid predictor variables except for the "recommended number of players" predictors. Interestingly, the model that uses the formula produced by step() starting with all non-"number of players" predictors and their interactions produces the second-best RMSE so far, although it underperformed in other models. Finally, the 15-topic version of the LDA variable performs slightly better than the 25-topic version.

The two most important variables in the best random forest model are the "weight," or rules complexity, of the game, and the age of the game. This is hardly surprising, given that the top ranks of the BGG database are dominated by new and relatively complex games. (This may reflect something about the higher potential of more complex games and a trend toward increasing quality, but it may also reflect the fact that the board game enthusiast community gets unreasonably excited about new things and fetishizes complexity.) The topic, or genre/category variable, also had fairly high importance.

```
set.seed(12345)
```

```
#Random Forest
require(randomForest)

#best models so far.

rf <- randomForest(stats.average ~ topic + details.maxplayers + details.maxplaytime + details.minage
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train, importance = TRUE, na.action =

rmse.bgg(rf) #0.6797512
```

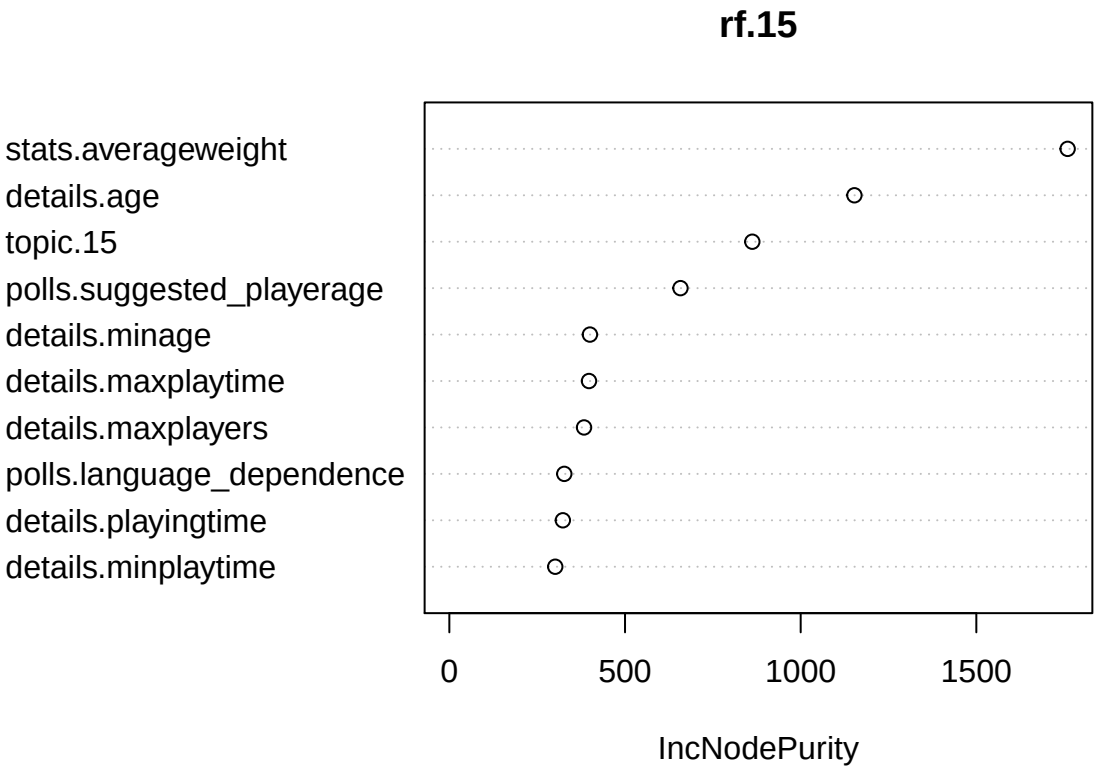
```
## [1] 0.6797512
```

```
rf.15 <- randomForest(stats.average ~ topic.15 + details.maxplayers + details.maxplaytime + details.mina
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train, importance = TRUE, na.action =

rmse.bgg(rf.15) #0.6794091
```

```
## [1] 0.6794091
```

```
varImpPlot(rf.15, type=2)
```




```
#Smaller model
rf_small <- randomForest(step_smaller_formula, data = train, importance = TRUE, na.action = "na.omit")

rmse.bgg(rf_small) #0.6865579
```

```
## [1] 0.6865579
```

```
#big model from step.
rf_big <- randomForest(step_formula, data = train, importance = TRUE, na.action = "na.omit")

rmse.bgg(rf_big) #0.679553
```

```
## [1] 0.679553
```

```
#As before, adding the "suggested number of players" variables makes the predictions worse.

rf_big_numplayers <- randomForest(stats.average ~ topic + details.maxplayers + details.maxplaytime + de
   + details.minplaytime + details.age + details.playingtime + stats.averageweight + polls.language_dep
   + polls.suggested_numplayers_simple.1 + polls.suggested_numplayers_simple.2 + polls.suggested_numpla
   + polls.suggested_numplayers_simple.4 + polls.suggested_numplayers_simple.5 + polls.suggested_numpla
   + polls.suggested_numplayers_simple.7 + polls.suggested_numplayers_simple.8 + polls.suggested_numpla
   + polls.suggested_numplayers_simple.10 + polls.suggested_numplayers_simple.Over, data = train, impor
   na.action = "na.omit")

rmse.bgg(rf_big_numplayers) #0.7084726
```

```
## [1] 0.7084726
```

Boosting with gbm() does not produce good results. Increasing the number of trees from 100 (not shown) to

1,000 improves the predictions signficantly, and going up to 10,000 improves them slightly more, but the error is still well above that of random forests.

```
#Boosting

boosted <- gbm(stats.average ~ topic + details.maxplayers + details.maxplaytime + details.minage
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train, interaction.depth = 4, shrinka
   n.trees = 1000, n.cores = parallel::detectCores())
```

```
## Distribution not specified, assuming gaussian ...
```

```
pred.boost <- predict(boosted, newdata = test, n.trees = 1000)

sqrt(mean((test$stats.average - pred.boost)^2, na.rm = T)) # 0.8273503
```

```
## [1] 0.8273503
```

```
boosted_less_deep <- gbm(stats.average ~ topic + details.maxplayers + details.maxplaytime + details.mina
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train, interaction.depth = 2, shrinka
   n.trees = 1000, n.cores = parallel::detectCores())
```

```
## Distribution not specified, assuming gaussian ...
```

```
pred.boost_less_deep <- predict(boosted_less_deep, newdata = test, n.trees = 1000)

sqrt(mean((test$stats.average - pred.boost_less_deep)^2, na.rm = T)) #0.8362381
```

```
## [1] 0.8362381
```

```
boosted_many <- gbm(stats.average ~ topic + details.maxplayers + details.maxplaytime + details.minage
   + details.minplaytime + details.age + details.playingtime + stats.averageweight
   + polls.language_dependence + polls.suggested_playerage, data = train, interaction.depth = 4, shrinka
   n.trees = 10000, n.cores = parallel::detectCores())
```

```
## Distribution not specified, assuming gaussian ...
```

```
pred.boost_many <- predict(boosted_many, newdata = test, n.trees = 10000)

sqrt(mean((test$stats.average - pred.boost_many)^2, na.rm = T)) #0.823153
```

```
## [1] 0.823153
```

BART produces very similar predictions to random forest. Oddly, despite setting the seed immediately before running the model, bartMachine() produced slightly different RMSEs with each run, ranging from 0.675 (the best model) to 0.680 (marginally worse than the best random forests.) A BART model with the larger subset of predictors used above also performs reasonably well, but others do better.

```
set.seed(12345)

#bartMachine

set_bart_machine_num_cores(parallel::detectCores())
```

```
## bartMachine now using 4 cores.
```

```
#Bart with best hand-chosen model variables

bart <- bartMachine(X = as.data.frame(X_pre_nonum), y = Y_pre_nonum, mem_cache_for_speed = FALSE)
```

```
## bartMachine initializing with 50 trees...
## bartMachine vars checked...
## bartMachine java init...
## bartMachine factors created...
## bartMachine before preprocess...
## bartMachine after preprocess... 47 total features...
## bartMachine sigsq estimated...
## bartMachine training data finalized...
## Now building bartMachine for regression ...
## evaluating in sample data...done
```

```
pred_bm <- predict(bart, new_data = as.data.frame(X_star_lasso_small))

sqrt(mean((Y_test_nonum_pre - pred_bm) ^ 2, na.rm = T))
```

```
## [1] 0.6802953
```

```
bart_big <- bartMachine(X = as.data.frame(X_step_big), y = Y_step_big, mem_cache_for_speed = FALSE)
```

```
## bartMachine initializing with 50 trees...
## bartMachine vars checked...
## bartMachine java init...
## bartMachine factors created...
## bartMachine before preprocess...
## bartMachine after preprocess... 178 total features...
## bartMachine sigsq estimated...
## bartMachine training data finalized...
## Now building bartMachine for regression ...
## evaluating in sample data...done
```

```
pred_bm_big <- predict(bart_big, new_data = as.data.frame(X_star_lasso_big))

sqrt(mean((Y_test_lasso_big - pred_bm_big) ^ 2, na.rm = T))
```

```
## [1] 0.6810015
```

Neural Network

A neural network performs fairly poorly. A number of parameter tweaks (greater depth, more iterations), not shown here, did not improve the performance.

```
set.seed(12345)

library(nnet)

nn <- nnet(step_formula, data = train,
           size = 2, rang = 0.1, decay = 5e-4, maxit = 200, linout = TRUE)
```

```
## # weights:  359
## initial  value 339186.150966
## iter  10 value 6322.221967
## iter  20 value 5416.176356
## iter  30 value 5349.415781
## iter  40 value 5305.807083
## iter  50 value 5264.943584
## iter  60 value 5230.226615
```

```
## iter  70 value 5192.612670
## iter  80 value 5146.861520
## iter  90 value 5119.907143
## iter 100 value 5099.678978
## iter 110 value 5070.255883
## iter 120 value 5029.291233
## iter 130 value 5017.281823
## iter 140 value 5014.367179
## iter 150 value 5013.133594
## iter 160 value 5007.897527
## iter 170 value 5006.918217
## iter 180 value 5006.080115
## iter 190 value 5003.240082
## iter 200 value 4999.786532
## final  value 4999.786532
## stopped after 200 iterations
```

```
pred_nn <- predict(nn, newdata = test)
rmse.bgg(nn)
```

```
## [1] 0.7804006
```

Conclusion

After trying a variety of linear, nonlinear, and tree-based regression models, the best results come from tree-based models: random forests and BART. The rankings of other models varied slightly based on the random split of the data used (which I discovered thanks to some errors in setting the seed), but random forests and BART were always the best. This is hardly surprising, given the generally good performance of such models.

However, they only provide a very small improvement in predictions relative to the best hand-chosen OLS models, which were not terribly hard to arrive at: they include all of the reasonable predictors in the dataset minus one class of predictors that introduced a very large degree of missingness. This is particularly true for this random split of training and test data - the difference between OLS and BART was about twice as large in others - but it was never particularly big. (In addition, a GAM with the same predictors as the best OLS models removed most of the gap between linear regression and tree models.) For the purposes of pure prediction, of course, even small improvements are welcome, but it underscores the sometimes-limited value of complex and computationally costly data mining techniques.

Finally, it is worth noting that a different approach to incorporating qualitative board game data could potentially produce quite different results. The two versions of LDA used here produced very similar predictive accuracy, but they also had fairly high importance in the best models, which suggests that a totally different scheme for categorizing boardgames, or perhaps including another LDA variable built from game descriptions, could significantly affect the results. Given that much of what makes games popular and successful is not easily quantifiable, a deeper, more nuanced text analysis might be a powerful way to extend this project.