Problem 1: Predicting AirBnB review scores

a

```
reviews = read.csv("airbnb-small.csv", stringsAsFactors = F)

for (nStar in 1:5){
  nStar_reviews = reviews[reviews$review_scores_rating == nStar,]
  cat("Rating:", nStar,"\n",
      "Number of reviews:", nrow(nStar_reviews), "\n",
      "Average length (in chars):", mean(nchar(nStar_reviews$comments)),"\n")

}
```

```
## Rating: 1
##  Number of reviews: 62
##  Average length (in chars): 464.5968
## Rating: 2
##  Number of reviews: 56
##  Average length (in chars): 597.7321
## Rating: 3
##  Number of reviews: 156
##  Average length (in chars): 388.8013
## Rating: 4
##  Number of reviews: 708
##  Average length (in chars): 276.476
## Rating: 5
##  Number of reviews: 3191
##  Average length (in chars): 289.4184
```

The number of reviews are unbalanced. I would guess that the normal rating is 5 stars, and guests only rate lower if it something they are explicitly not happy about. Also the length of the reviews are longer for low ratings, i.e. people are more inclined to fully report bad experiences than good experiences.

b

```
corpus = Corpus(VectorSource(reviews$comments)) # An array of document
corpus = tm_map(corpus, tolower)
corpus = tm_map(corpus, removeWords, stopwords("english"))
corpus = tm_map(corpus, removeWords, c("airbnb"))
corpus = tm_map(corpus, stemDocument)
strwrap(corpus[[1]])
```

```
## [1] "good stay, issu direct instruct differ actual properti ."
```

```
strwrap(corpus[[2]])
```

```
## [1] "home quiet neighborhood flushing. room stay newli renovated. enjoy"
## [2] "stay. access manhattan via one bus 7 train. recommend host home."
```

```
strwrap(corpus[[3]])
```

```
## [1] "ernest welcom us generous beauti home plenti inform apart area."
## [2] "enjoy wonder stay comfort avail surroundings. can recommend ernest"
## [3] "host apart great manhattan get away."
```

c

```
frequencies = DocumentTermMatrix(corpus)
findFreqTerms(frequencies, lowfreq=800)
```

```
## [1] "host"   "room"   "stay"   "apart"  "great"  "locat"  "place"  "nice"
## [9] "realli"
```

```
sparse = removeSparseTerms(frequencies, 0.99)
sparse
```

```
## <<DocumentTermMatrix (documents: 4173, terms: 426)>>
## Non-/sparse entries: 66916/1710782
## Sparsity           : 96%
## Maximal term length: 14
## Weighting          : term frequency (tf)
```

So we have 426 terms left.

d

```
reviewTerms = as.data.frame(as.matrix(sparse))
reviewTerms$rlength = nchar(reviews$comments)
reviewTerms$feeling = cut(x=reviews$review_scores_rating, breaks = c(0, 3, 5))
levels(reviewTerms$feeling) = c("Negative", "Positive")

reviewTrain = reviewTerms[reviews$date < "2018-01-01",]
reviewTest = reviewTerms[reviews$date >= "2018-01-01",]
table(reviewTrain$feeling)

##
## Negative Positive
##      233     2955

table(reviewTest$feeling)

##
## Negative Positive
##       41      944
```

So the proportion of negative reviews in the training set is 0.0731, and 0.0416 in the testing set. This indicates reviews get more positive over time, and perhaps we have a shift in the underlying distribution of reviews between training and test, which would make the prediction of our model worse on the testing set.

e

```
review.tree1 = rpart(feeling ~., data=reviewTrain, cp=0.01)
```

The three trees can be seen in fig. 1, fig. 2 and fig. 3. They have a CP of 0.01, 0.005 and 0.001, respectively.

Tree one is the simplest tree, and the review will be negative if the word *bathroom* is mentioned along with *expect*, or the term *recommen* is absent , but the term *host* is not.

Tree number two is a bit bigger, but we can see some of the trends from the first tree. A review tends to be positive if positive terms like *great* or *recommen* is present, or the review is short. Also the term *expect* is a bad sign, which makes sense as it would be used frequently by complaining guests who did not get what they expected.

In the last tree, we can for example also see that the absence of the term *help* and *stay* is positive. Surprisingly, the term *nois* is positive. I would expected this to be correlated with noise complaints and therefore negative. Inspecting the reviews with this term included reveals that most of the comments is about the low amount of noise, which explains the value in the tree.

f

The baseline could be to just guess the majority every time, so positive.

```
printMetrics = function(preds){
  CM = table(reviewTest$feeling, preds)
  print(CM)
  acc = sum(diag(CM)) / sum(CM)
  tpr = CM[2,2]/sum(CM[2,])
  fpr = CM[1,2]/sum(CM[1,])

  cat("Accuracy: ", acc, "\n")
  cat("TPR:", tpr, "\n")
  cat("FPR:", fpr, "\n")
}
printMetrics(predict(review.tree1, newdata=reviewTest, type = "class"))

##           preds
##            Negative Positive
##   Negative        3       38
##   Positive        7      937
## Accuracy:  0.9543147
## TPR: 0.9925847
## FPR: 0.9268293

printMetrics(predict(review.tree2, newdata=reviewTest, type = "class"))
```

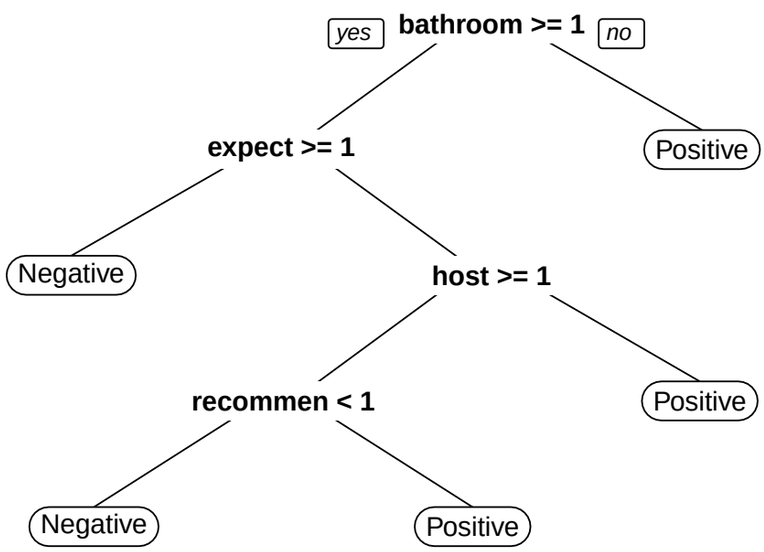

Figure 1: CART tree with CP = 0.01

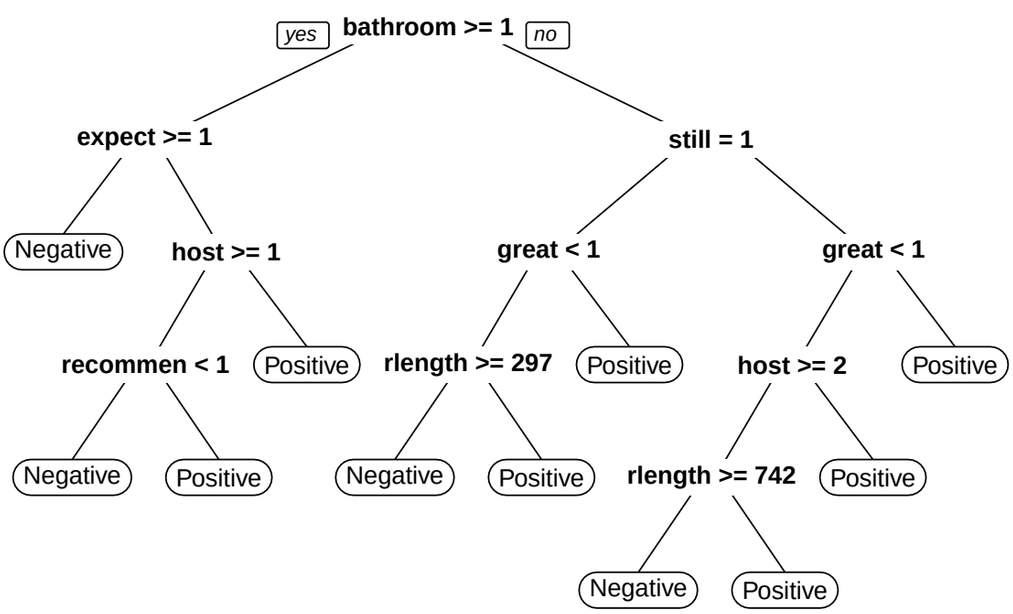

Figure 2: CART tree with CP = 0.005

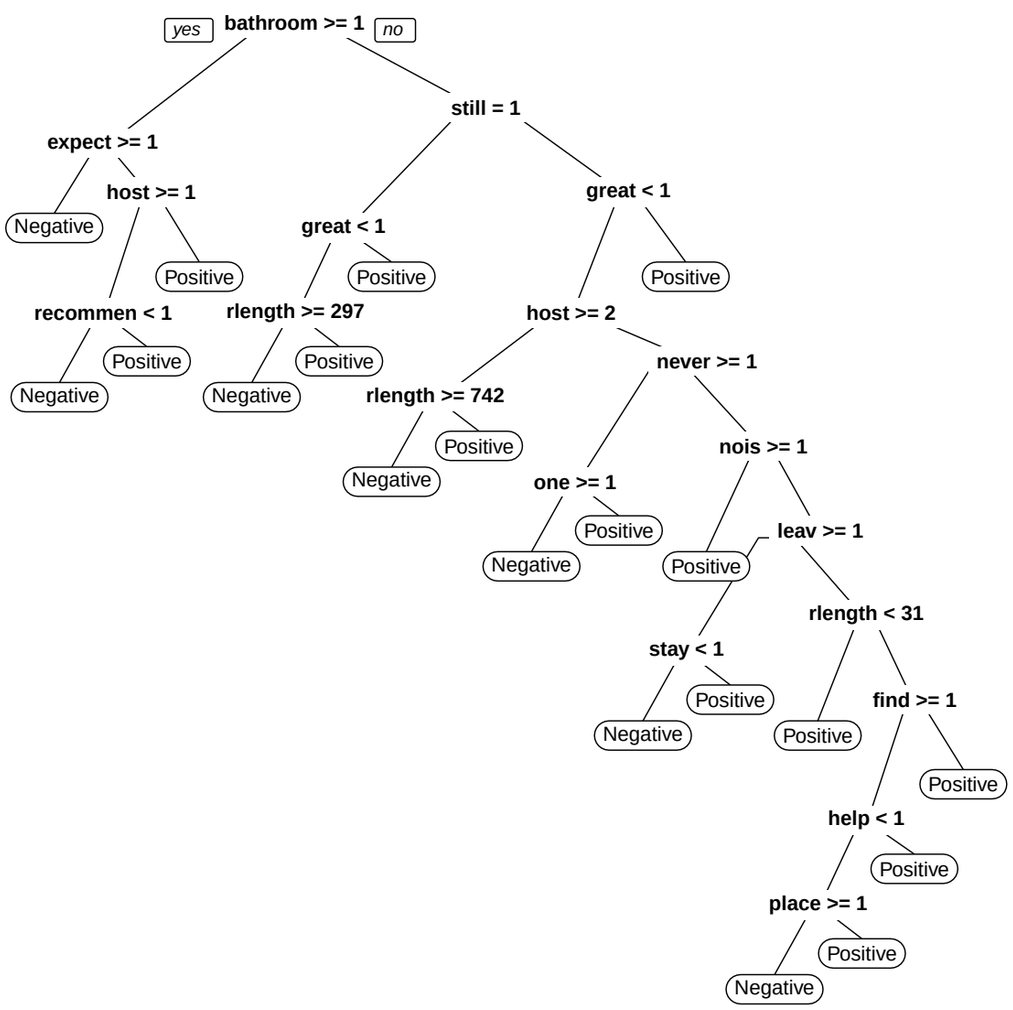

Figure 3: CART tree with CP = 0.001

```
##           preds
##            Negative Positive
##   Negative        4       37
##   Positive       13      931
## Accuracy:  0.9492386
## TPR: 0.9862288
## FPR: 0.902439
```

```
printMetrics(predict(review.tree3, newdata=reviewTest, type = "class"))
```

```
##           preds
##            Negative Positive
##   Negative        4       37
##   Positive       19      925
## Accuracy:  0.9431472
## TPR: 0.9798729
## FPR: 0.902439
```

While the baseline has an accuracy of 0.9584, true positive rate of 1, and a false positive rate of 1. So for all models, both the false and true positive rate is very high, which is simply because the ratio of positive to negative is very high, so guessing mostly positive will always be a good solution when considering accuracy as a metric.

Problem 2: Recommending Songs

a

```
ranks = seq(1,10)
prop.validate = 0.05
scores = cf.evaluate.ranks(music.train, ranks, prop.validate)
```

From the figure fig. 4 is seems a low number of users is the best, i.e. 2, 3 or 4, based on which metric you observe. I did not expect it to be so few, as I thought there would be more diverse taste in music. Lets continue with 3 users, as this is the middle value of our candidates.

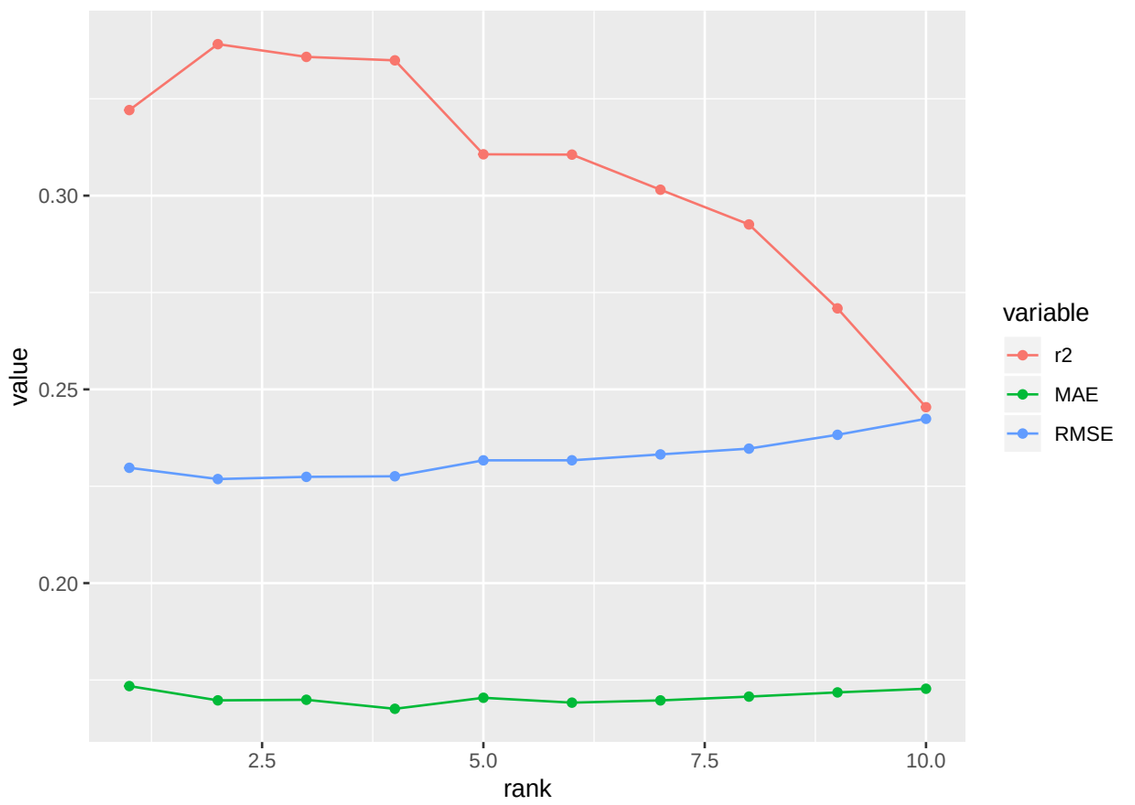

Figure 4: Model performance as a function of the rank

b

```
set.seed(15071)
fit = softImpute(mat.scaled, rank.max=3, maxit=1000)
preds <- impute(fit, music.test$userID, music.test$songID)

avg = mean(music.test$rating)
TSS = sum((music.test$rating - avg)**2)
RSS = sum((music.test$rating - preds)**2)
Rsq = 1 - RSS/TSS
Rsq

## [1] 0.3181344
```

So the out-of-sample R^2 of the model is 0.31813. We refit the model in order to take advantage of all our training data. The heuristic function "evaulate.ranks" uses some training data as a holdout set to quickly evaluate the ranks, and we should include this data when training the final model.

c

```
ratings = fit$v * fit$d
avgRatings = aggregate(ratings, by=list(songs$genre), FUN=mean)
avgRatings

##      Group.1           V1          V2         V3
## 1    Country -4.816633546 -0.51448623 -2.0631771
## 2 Electronic  0.009125378  0.54557832 -2.2738995
## 3       Folk  0.898628366 -0.37795619  1.2327005
## 4        Pop  0.011434655 -0.37685056  0.7640469
## 5        Rap -0.470765902 -0.58339534 -2.1622541
## 6        RnB -0.064429170 -0.15124626 -1.5798741
## 7       Rock -0.061498548 -0.07043769 -0.8551108
```

The first archetype ("V1") clearly does not like Country, but is a fan of Folk. It is pretty much indifferent to the rest, with an exception of a minor distaste for Rap. The second ("V2") only likes Electronic and slighly dislikes everything else. The last archetype ("V3") likes Folk and Pop and strongly dislikes everying else.

d

```
daisy.ID = 1584
daisy.weights = fit$u[daisy.ID,]
daisy.weights

## [1]  0.01416295 -0.01847687  0.02429789

df = data.frame(
  scores = fit$v %*% daisy.weights,
  songID = seq(1:nrow(fit$v))
)

daisy.ratings = music[music$userID == daisy.ID,]
daisy.unrated = subset(df, !(songID %in% daisy.ratings$songID))

# Top unrated
daisy.topscores.unrated = head(daisy.unrated[order(-daisy.unrated$scores),])
daisy.topsongs.unrated = subset(songs, (songID %in% daisy.topscores.unrated$songID))

# Top already rated
daisy.topscores.rated = head(daisy.ratings[order(-daisy.ratings$rating),])
daisy.topsongs.rated = subset(songs, (songID %in% daisy.topscores.rated$songID))

daisy.topsongs.unrated

##     songID              songName year          artist genre
## 59      59 They Might Follow You 2007     Tiny Vipers   Pop
## 110    110                   Mia 2007  Emmy The Great  Folk
## 124    124         Fast As I Can 2000    Erin McKeown  Folk
## 221    221    Livin' On A Prayer 1986        Bon Jovi  Rock
## 224    224      Vanilla Twilight 2009        Owl City   Pop
## 736    736       I Gotta Feeling 2009 Black Eyed Peas   Pop

daisy.topsongs.rated
```

```
##     songID             songName year                   artist genre
## 61      61           Monkey Man 2007            Amy Winehouse   Pop
## 92      92         One And Only 2005    Mariah Carey / Twista   RnB
## 146    146   LDN (Switch Remix) 2009               Lily Allen   Pop
## 244    244        In The Mirror 2005                Nada Surf  Rock
## 274    274           Drive Away 2003 The All-American Rejects  Rock
## 512    512 Shake A Tail Feather 2005        The Cheetah Girls   Pop
```

This seems reasonable, the algorithm mainly recommends songs in genres she already likes.