Chapter 6

Neighborhood-Based Methods

One of the simplest and yet often most effective recommender system methods is based on this natural principle:

> Say we have a user U, for whom we want to predict the rating of an item I. We find the users in our existing data D who are most similar to U and who have seen rating I, and take our predicted value to be the average of those users' ratings of I.

Note that here the user U might be in D or might be new. As long as I is in D, we are in business.

Of course, we must define "similar." There are two common ways to do this. Recall our notation p denoting our number of predctors/features. This would include our user and item IDs, and possible covariates. Then consider these approaches:

- Define some distance function, and then find the k closest people in D to U.
- Develop a system of rectangles — hyperrectangles in p-dimensional space — and determine which one U falls in.

The first is basically *k-Nearest Neighbor regression* (kNN), a classic statistics/machine learning technique, though note that a major difference here is that we only consider users in D who have rated the same products as N.

The second can be viewed as something called *kernel regression*, but we will add a tree-based structure, with the result termed *Classification and Regression Analysis* (CART) for a single tree and *random forests* for a collection of trees. In the latter, we create a collection of systems of rectangles and average over them.[1]

[1]The terms "CART" and "random forests" have many variations, but we take the terms as generic.

6.1 kNN

Let's see how kNN works.

6.1.1 Notation

As before, let A denote the ratings matrix. The element a_{ij} in row i, column j, is the rating that user i has given/would give to item j. In the latter case, a_{ij} is unknown, and its predicted value will be denoted by $\widehat{a}_{ij}$. Following R notation, we will refer to the unknown values as NAs.

Note that for large applications, the matrix A is far too large to store in memory. One could resort to storage schemes for *sparse* matrices, e.g. *Compresed Row Storage*, but here we will simply use A to help explain concepts. In the **rectools** package,[2] the input data is run through **formUserData()** and algorithms use that instead of A. This function organizes the data into an R list, one element per user. Each such element records the ratings made by that user.

Let's refer to a new case to be predicted as NC, i.e. from above, predicting how a user U would rate an item I.

6.1.2 User-Based Filtering

In predicting how a given user would rate a given item, we first find all users that have rated the given item, then determine which of those users are most similar to the given user. Our prediction is then the average of the ratings of the given item among such "similar" users. A corresponding approach based on similar items, *item-based filtering*, is used as well. We focus on such methods in this chapter.

6.1.3 (One) Implementation

Below is code from **rectools** (somewhat simplified).[3] The arguments are:

- **origData:** The original dataset, after having been run through **formUserData()**.
- **newData:** The element of **origData** for NC.[4]
- **newItem:** ID number of the item to be predicted for NC.

[2] https://github.com/matloff/rectools

[3] This function was written largely by Vishal Chakraborti.

[4] If NC is new, not in the database (called *cold start*), we synthesize a list element for it, assuming NC has rated at least one item.

- k: The number(s) of nearest neighbors. Can be a vector.

Here is an example of using **formUserData()** on the MovieLens data:[5]

```
> head(ml)
V1  V2 V3
1 196 242  3
2 186 302  3
3  22 377  1
4 244  51  2
5 166 346  1
6 298 474  4
> mlud <- formUserData(ml)
> mlud[[3]]
$userID
[1] "3"

$itms
[1] 335 245 337 343 323 331 294 332 328 334 350 341 318 300
[15] 345 299 324 348 351 330 327 307 272 354 264 349 321 260
[29] 268 288 355 320 258 339 342 303 329 317 181 338 302 322
[43] 352 271 333 344 326 319 325 347 336 353 340 346

$ratings
335 245 337 343 323 331 294 332 328 334 350 341 318 300 345
1   1   1   3   2   4   2   1   5   3   3   1   4   2   3
299 324 348 351 330 327 307 272 354 264 349 321 260 268 288
3   2   4   3   2   4   3   2   3   2   3   5   4   3   2
355 320 258 339 342 303 329 317 181 338 302 322 352 271 333
3   5   2   3   4   3   4   2   4   2   2   3   2   3   2
344 326 319 325 347 336 353 340 346
4   2   2   1   5   1   1   5   5

attr(,"class")
[1] "usrDatum"
```

So, for any given user, **mlud** will show the items rating by this user and the ratings the user has given to those items. Here we see that user 3 has rated items 335, 245,, 337, 343,..., with ratings 1,1,1,3,...

[5]The data have been read from disk without converting to R factors.

```
predict.usrData <- function(origData,newData,newItem,k)
{
# we first need to narrow origData down to the users who
# have rated newItem

# here oneUsr is one user record in origData; the function will look for a
# j such that element j in the items list for this user matches the item
# of interest, newItem; (j,rating) will be returned

checkNewItem <- function(oneUsr) {
   whichOne <- which(oneUsr$itms == newItem)
   if (length(whichOne) == 0) {
      return(c(NA,NA))
   } else return(c(whichOne,oneUsr$ratings[whichOne]))
}

found <- as.matrix(sapply(origData,checkNewItem))
# description of 'found':
# found is of dimensions 2 by number of users in training set
# found[1,i] = j means origData[[i]]$itms[j] = newItem;
# found[1,i] = NA means newItem wasn't rated by user i
# found[2,i] = rating in the non-NA case

# we need to get rid of the users who didn't rate newItem
whoHasIt <- which(!is.na(found[1,]))
origDataRatedNI <- origData[whoHasIt]
# now origDataRatedNI only has the relevant users, the ones who
# have rated newItem, so select only those columns of the found matrix
found <- found[,whoHasIt,drop=FALSE]

# find the distance from newData to one user y of origData; defined for
# use in sapply() below
onecos <- function(y) cosDist(newData,y,wtcovs,wtcats)
cosines <- sapply(origDataRatedNI,onecos)
# the vector cosines contains the distances from newData to all the
# original data points who rated newItem

# action of findKnghbourRtng(): find the mean rating of newItem in
# origDataRatedNI, for ki (= k[i]) neighbors
#
```

```
41 # if ki > neighbours present in the dataset, then the
42 # number of neighbours is used
43 findKnghbourRtng <- function(ki){
44   ki <- min(ki, length(cosines))
45   # nearby is a vector containing the indices of the ki closest neighbours
46   nearby <- order(cosines,decreasing=FALSE)[1:ki]
47   mean(as.numeric(found[2, nearby]))
48 }
49 sapply(k, findKnghbourRtng)
50 }
```

6.1.4 Not Really a Distance

Note that the distances were computed by the function **cosDist()**, which computes a "cosine" similarity:

```
find cosine distance between x and y, objects
# of 'usrData' class
#
# only items rated in both x and y are used; if none
# exist, then return NaN
#
#  wtcovs: weight to put on covariates; NULL if no covs
#  wtcats: weight to put on item categories; NULL if no cats

cosDist <- function(x,y,wtcovs=NULL,wtcats=NULL)
{
# rated items in common
commItms <- intersect(x$itms,y$itms)
if (length(commItms)==0) return(NaN)
# where are those common items in x and y?
xwhere <- which(!is.na(match(x$itms,commItms)))
ywhere <- which(!is.na(match(y$itms,commItms)))
xvec <- x$ratings[xwhere]
yvec <- y$ratings[ywhere]
if (!is.null(wtcovs)) {
   xvec <- c(xvec,wtcovs*x$cvrs)
   yvec <- c(yvec,wtcovs*y$cvrs)
}
if (!is.null(wtcats)) {
```

```
   xvec <- c(xvec,wtcats*x$cats)
   yvec <- c(yvec,wtcats*y$cats)
}

xvec %*% yvec / (l2a(xvec) * l2a(yvec))
}

l2a <- function(x) sqrt(x %*% x)
```

Basically, the "distance" between two rows u and v of A is defined by

$$\frac{u'v}{||u||_2 \; ||v||_2} \tag{6.1}$$

This not really a distance,[6] but it is a common measure of similarity between two vectors in machine learning. In two or three dimensions, it really is the cosine of the angle between u and v.

Note that larger cosines mean the vectors are more similar. We find the k most similar rows in D to U, and average their ratings of the given item.

6.1.5 Regression Analog

Recall the method of k-nearest neighbor (kNN) regression estimation from Chapter 3, involving prediction of weight from height and age:

> To estimate $E(W \mid H = 70, A = 28)$, we could find, say, the 25 people in our sample for whom (H, A) is closest to (70,28), and average their weights to produce our estimate of $E(W \mid H = 70, A = 28)$.

So kNN RS is really the same as kNN regression

6.1.6 Choosing k

As we have already seen with RS, regression and machine learning methods, the typical way to choose a model is to use cross-validation. This is true for kNN RS as well; we can choose the value of k via cross-validation.

[6] IN math terms, it's not a *metric*.

6.1.7 Item-Based Filtering

Consider again our setting in which we wish to predict the rating user U would give to item I. We could switch the above procedure, trading rows for columns. We would find the columns corresponding to items U has rated, then find the closest k of those columns to column I. The ratings given by U in those closest column would then be averaged to yield our prediction.

6.1.8 Covariates

To accommodate covariates, we simply add covariate columns to the input matrix, say now with columns 'userId', 'itemId', 'rating' and age'. Note that they figure into the distance measure, just like the user and item I dummies.

6.2 CART and Random Forests

CART is based on forming a *recursive partitioning* of the data space. First the space is split in two, then each of the two parts is split in two, and so on, forming a binary tree. Splitting along a branch stops when some criterion is no longer met.

6.2.1 Motivating Example

Let's illustrate CART with the R package **partykit**.[7] We'll predict weight from height and age in the baseball player data.

```
> library(regtools)
> data(mlb)
> mlb <- mlb[,c(4:6)]
> head(mlb)
  Height Weight   Age
1     74    180 22.99
2     74    215 34.69
3     72    210 30.78
4     72    210 35.43
5     73    188 35.71
6     69    176 29.39
> ctout <- ctree(Weight ~ .,data=mlb)
```

[7] The name is an allusion to the recursive partioning nature of CART.

Here is the tree that is produced:

```
> node_party(ctout)
[1] root
|   [2] V2 <= 73
|   |   [3] V2 <= 70
|   |   |   [4] V3 <= 29.56 *
|   |   |   [5] V3 > 29.56 *
|   |   [6] V2 > 70
|   |   |   [7] V3 <= 32.51
|   |   |   |   [8] V2 <= 72 *
|   |   |   |   [9] V2 > 72 *
|   |   |   [10] V3 > 32.51 *
|   [11] V2 > 73
|   |   [12] V2 <= 75
|   |   |   [13] V3 <= 27.55
|   |   |   |   [14] V2 <= 74 *
|   |   |   |   [15] V2 > 74 *
|   |   |   [16] V3 > 27.55 *
|   |   [17] V2 > 75
|   |   |   [18] V2 <= 79
|   |   |   |   [19] V3 <= 26.03 *
|   |   |   |   [20] V3 > 26.03 *
|   |   |   [21] V2 > 79 *
```

We'll look more at the tree shortly, but first suppose we wish to predict the weight of a player who is 72 inches tall, age 31.

```
> predict(ctout,data.frame(Height=72,Age=31))
       1
189.9091
```

Where did that come from? Look at the tree. Height $\leq$ 73? Yes, so go to node 3. Height $\leq$ 70? No, go to node 6, etc., winding up at node 8. The asterisk indicates a leaf node, so we're done traversing the tree.‘ No, what is the mean weight in that node?

```
> ht <- mlb$Height
> age <- mlb$Age
> mean(mlb[70 < ht & ht <= 72 & age <= 32.51,]$Weight)
[1] 189.9091
```

6.2.2 Use in Recommender Systems

Clearly, it would be difficult to use CART directly on, say, the MovieLens data. Since the user and item IDs are not *ordinal*, i.e. do not have an inherent underlying ordering, we'd need to form dummy variables, and thus test for one of them at a time. The tree would be greatly unbalanced, to the right, and we'd have a computational problem, at the least.

But actually, it's a lot worse than that. Think of a node "User ID = 168?" In cases where we take the left branch, i.e. the data point is indeed for user 168, the next question might be, say, Item ID = 12?" The key point is that **there will be at most 1 data point satisfying both conditions.** That's not enough to make a good tree. And indeed, CART software typically sets a minimum node size as a hyperparameter; for **ctree(), it's minsplit**.

An alternative is to do an *embedding* of the user and item ID data, a form of dimension reduction. What we could do here is replace each user ID by the mean rating given by that user, and do the same for the movies:

```
> userMeans <- tapply(ml$V3,ml$V1,mean)
> itemMeans <- tapply(ml$V3,ml$V2,mean)
> mlemb <- mlb
> mlemb$V1 <- userMeans[ml$V2]
> mlemb$V2 <- itemMeans[ml$V2]
> mlemb$V1 <- as.vector(mlemb$V1)
> mlemb$V2 <- itemMeans[ml$V2]
> mlemb$V2 <- as.vector(mlemb$V2)
> ctout <- ctree(V3 ~ .,data=mlemb)
```

We could then use **predict()** as before.

6.2.3 Tuning Parameters

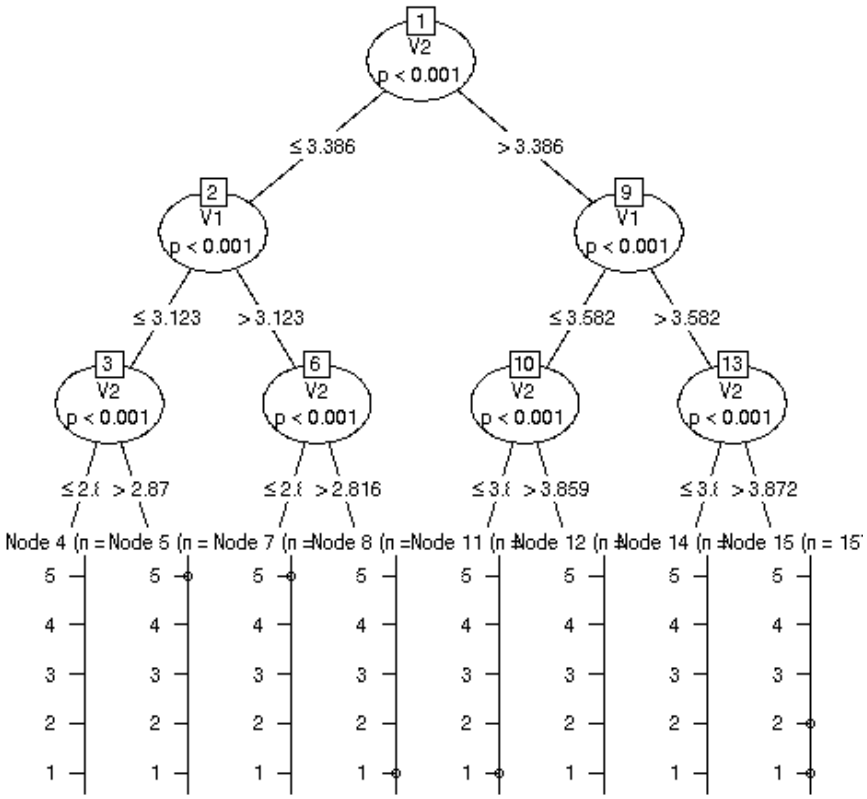
The **ctree()** function has various tuning parameters. We saw one of them above, **minsplit**. Another is **maxdepth**, the maximum number of levels we wish to allow in the tree. The default value, 0, means no limit. You can see in the output above the **ctree()** built us a 5-level tree. The tuning parameters must be set via the function **ctree_control()**.

The return value of **ctree()** has class, of course, **'party'**. It has various generic functions. We saw **predict()** above; **print()** gives output similar to, but a little more detailed than, **node_party()**.

Another is **plot()**, which due to screen space issues, is only usable for a few levels. Here we set **maxdepth** to 3,

```
> ctout <- ctree(V3 ~ .,data=mlemb,control = ctree_control(maxdepth=3))
> plot(ctout)
```

l



Actually, the plotting comes from the fact that **ctout** is also of class **'BinaryTree'**

6.2.4 Covariates

Covariates are easily handled, e.g.

```
ctout <- ctree(V3 ~ V1+V2+age+gender,data=mlemb)
```

6.2.5 Random Forests

Some years after usage of CART became widespread, there was concern that the procedure was too sensitive to small changes in the data. Consider the root node, for instance. If the data is changed slightly, that first split may change somewhat, with the change cascading down the entire tree. In other words, the tree is quite subject to sampling variation, meaning in turn that the predicted value of any particular future case will have a higher variance.

The solution was to generate many trees, and combine the results. New trees are generated by resampling from our original data: For a dataset of n data points, we randomly sample n times, *with replacement*, and run CART on that new sample. We then predict a new case by obtaining a prediction from each tree, then aggregating the predictions: In a regression setting, we average all the predicted values, while in a classification setting, our final prediction is whichever class was predicted most often aomong the various trees. The resampling is an example of the *bootstrap*, so the entire process is called *bootstrap aggregating*, or *bagging*.

The original idea for all this came from Tin Kam Ho. Leo Breiman, one of the developers of CART, refined it and coined the term *random forests*.

The **partykit** package includes the function **cforest()** for this technique.