Homework 13: Some Practice with Recommender Systems

36-462/662, Data Mining

Due Saturday, 25 April 2020 at 10 pm

Agenda: Getting our hands dirty with actual implementing a matrix factorization algorithm, and trying it out for making recommendations.

Remember[1] that in matrix factorization, we have an $n \times p$ matrix $\mathbf{x}$, and we want to approximate it as the product of two matrices,

$$\mathbf{x} \approx \mathbf{f}\mathbf{g}^T$$

where $\mathbf{f}$ is $n \times q$ and $\mathbf{g}$ is $p \times q$, and q is much smaller than either n or p. This is called "matrix factorization" because we have (approximately) "factored" $\mathbf{x}$ into $\mathbf{f}$ and $\mathbf{g}$. The number q is the number of factors.

In recommendation systems, $\mathbf{x}$ is the matrix of ratings, with n users and p items. If every user had rated every item, we could do matrix factorization using PCA. Since most users have not rated most items, straightforward PCA doesn't work so well. One approach is to start with an initial guess about $\mathbf{f}$ and $\mathbf{g}$, and then iterate: holding $\mathbf{f}$ fixed, adjust $\mathbf{g}$ to best fit the ratings we do have, and then, holding $\mathbf{g}$ fixed, optimize $\mathbf{f}$.

In a little more detail, say that the rating of user i for item j is x_{ij}. Then we want

$$x_{ij} \approx \sum_{k=1}^{q} f_{ik} g_{jk}$$

We can assess how well the matrix of user scores $\mathbf{f}$ and of item scores $\mathbf{g}$ comes to matching the actual ratings $\mathbf{x}$ by the mean squared error on the rated items:

$$MSE = \frac{1}{|C|} \sum_{(i,j) \in C} \left(x_{ij} - \sum_{k=1}^{q} f_{ik} g_{jk} \right)^2$$

where C is the set of all user-item pairs where user i has rated item j, and $|C|$ is the number of such ratings. There are two alternative ways to write this,

$$MSE = \frac{1}{|C|} \sum_{i=1}^{n} \sum_{j \in I(i)} \left(x_{ij} - \sum_{k=1}^{q} f_{ik} g_{jk} \right)^2 \quad (1)$$

$$= \frac{1}{|C|} \sum_{j=1}^{p} \sum_{i \in U(j)} \left(x_{ij} - \sum_{k=1}^{q} f_{ik} g_{jk} \right)^2 \quad (2)$$

where $I(i)$ are the items which user i has rated, and $U(j)$ are the users who have rated item j. In the alternating least squares procedure, we first treat $\mathbf{g}$ as fixed, and then minimize equation (1) to find $\mathbf{f}$. Then we hold that new $\mathbf{f}$ fixed, and minimize equation (2) to find $\mathbf{g}$.

There is one last bit of math to notice before we proceed. This is to observe that if $\mathbf{d}$ is a diagonal $q \times q$ matrix, then

$$\mathbf{f}\mathbf{g}^T = \mathbf{f}\mathbf{d}\mathbf{d}^{-1}\mathbf{g}^T = \mathbf{f}\mathbf{d}(\mathbf{g}\mathbf{d}^{-1})^T$$

[1] In the notes, I wrote this as $\mathbf{x} \approx \mathbf{f}\mathbf{g}$ and said $\mathbf{g}$ should be $q \times p$. This is just a notational difference, but the way I'm writing it here is a bit more common when people use matrix factorization for recommendation systems, whereas the way I wrote it in the notes is more common in factor analysis in the social sciences.

This means matrix factorizations aren't unique. Once we've found one, we can scale each column of the user scores by any factors we want, provided we scale each column of the item scores by the reciprocal factors, and the predictions won't change. One way to fix this is to insist that every column of $\mathbf{g}$ have norm 1.

Notes

1. There are no online reading questions this week.
2. All of the code printed here will also be on the website.
3. The calculations for (4c) take about 1.5 minutes on my computer, and (4d) is proportionately longer. Use caching!

1. Components of Matrix Factorization

a. (8)

Explain the following statement, and how it relates to the following piece of code:

> Holding the item scores constant, we can find each user's scores on all factors by linearly regressing that user's ratings on the item scores.

```
estimate.user.scores <- function(ratings, item.scores) {
    ratings.on.items <- function(x) {
        df <- data.frame(rating.vector=x, item.scores)
        fit <- lm(rating.vector ~ 0+ ., data=df)
        return(coefficients(fit))
    }
    user.scores <- t(apply(ratings, 1, ratings.on.items))
    if (length(dim(user.scores))==2) {
        user.scores <- matrix(user.scores, ncol=ncol(item.scores))
    }
    return(user.scores)
}
```

For this to work, what's the minimum number of items must each user have rated, in terms of the number of factors (q in the math above)?

How does the code handle the possibility that some users didn't rate some items, so that there are `NA` entries in the ratings matrix?

b. (7)

Explain the following statement, and how it relates to the following piece of code:

> Hold the user scores constant, we can find each item's scores on all factors by linearly regressing that item's ratings on the user scores.

```
estimate.item.scores <- function(ratings, user.scores) {
    ratings.on.users <- function(x) {
        df <- data.frame(rating.vector=x, user.scores)
        fit <- lm(rating.vector ~ 0+ ., data=df)
        return(coefficients(fit))
    }
    item.scores <- t(apply(ratings, 2, ratings.on.users))
    if (length(dim(item.scores))==2) {
```

```
        item.scores <- matrix(item.scores, ncol=ncol(user.scores))
    }
    norms <- apply(item.scores, 2, function(x) { sqrt(sum(x^2)) })
    item.scores <- sweep(item.scores, 2, norms, "/")
    return(item.scores)
}
```

For this to work, what's the minimum number times each item must have been rated by users, in terms of the number of factors (q in the math above)?

What is the point of the code between the call to `apply` and the return value?

c. (5)

Explain what the following code does:

```
rating.MSE <- function(ratings, user.scores, item.scores) {
    SSE <- mean((ratings - user.scores %*% t(item.scores))^2,
                na.rm=TRUE)
    return(SSE)
}
```

Why might this be an inefficient way to calculate whatever it's calculating if a lot of ratings are missing?

d. (5)

Explain what the following code does:

```
initialize.scores <- function(ratings, n.factors=1, stochastic=TRUE) {
    ratings <- as.matrix(ratings)
    mean.rating <- mean(ratings, na.rm=TRUE)
    if (stochastic) {
        sd.rating <- sd(ratings, na.rm=TRUE)
        ratings[is.na(ratings)] <- rnorm(n=sum(is.na(ratings)),
                                         mean=mean.rating,
                                         sd=sd.rating)
    } else {
        ratings[is.na(ratings)] <- mean.rating
    }
    ratings.pca <- prcomp(ratings, center=FALSE, scale.=FALSE, rank.=n.factors)
    users <- matrix(ratings.pca$x, ncol=n.factors)
    items <- matrix(ratings.pca$rot, ncol=n.factors)
    colnames(users) <- c()
    colnames(items) <- c()
    return(list(user.scores=users,
                item.scores=items))
}
```

e. (5)

Explain what the following code does:

```
estimate.scores <- function(ratings, n.factors=1, tol=0.01, maxit=100) {
    stopifnot(!all(is.na(ratings)))
```

```
    stopifnot(tol>0)
    stopifnot(maxit>1)
    ratings <- as.matrix(ratings)
    rating.sd <- sd(as.vector(ratings), na.rm=TRUE)
    n.users <- nrow(ratings)
    n.items <- ncol(ratings)
    old.MSE <- Inf
    initial.guesses <- initialize.scores(ratings, n.factors)
    user.scores <- initial.guesses$user.scores
    item.scores <- initial.guesses$item.scores
    new.MSE <- rating.MSE(ratings, user.scores, item.scores)
    iteration <- 0
    while ((abs(old.MSE - new.MSE) > tol*rating.sd)
           & (iteration < maxit)) {
        iteration <- iteration+1
        old.MSE <- new.MSE
        item.scores <- estimate.item.scores(ratings, user.scores)
        user.scores <- estimate.user.scores(ratings, item.scores)
        new.MSE <- rating.MSE(ratings, user.scores, item.scores)
    }
    return(list(user.scores=user.scores,
                item.scores=item.scores,
                diagnostics=list(converged=(iteration < maxit),
                                 iterations=iteration,
                                 MSE=new.MSE)))
}
```

Why does the code set `old.MSE` to infinity before it's calculated anything?

2. Try It Out Where We Know the Answers

a. (5)

Explain what the following code does:

```
rtestcase <- function(n.users, n.items, n.factors) {
    user.scores <- matrix(rnorm(n.users*n.factors),
                          ncol=n.factors)
    item.scores <- matrix(rnorm(n.items*n.factors),
                          ncol=n.factors)
    item.scores <- scale(item.scores, center=FALSE,
                         scale=apply(item.scores, 2,
                                     function(x) { sqrt(sum(x^2)) }))
    ratings <- user.scores %*% t(item.scores)
    return(list(ratings=ratings,
                user.scores=user.scores,
                item.scores=item.scores))
}
```

b. (5)

Run `rtestcase()` to produce a simulated data set with 50 users, 10 items, and two factors. Run `initialize.scores()` on the rating matrix, again with 2 factors. Plot (i) the estimated user scores against the true user scores, (ii) the estimated item scores against the true item scores, and (iii) the true ratings against the predicted ratings generated using the estimated scores. (You may find it helpful to convert these objects to vectors before plotting.) You should find that (iii) gives a much better match than either (i) or (ii).

c. (5)

Explain what the following code does.

```
obscure <- function(ratings, n.hide) {
    obscure.row <- function(x) {
        visible <- which(!is.na(x))
        to.obscure <- sample(visible, size=n.hide, replace=FALSE)
        x[to.obscure] <- NA
        return(x)
    }
    obscured.ratings <- t(apply(ratings, 1, obscure.row))
    return(obscured.ratings)
}
```

Run it on the demo ratings matrix you made earlier, and verify that (i) it creates the right over-all number of NAs, (ii) it creates the right number of NAs per row, and (iii) it doesn't change any *other* entries in the matrix.

d. (5)

Use the `obscure()` function to create a matrix from your earlier demo ratings matrix where two ratings per user are replaced by NAs. Call the resulting matrix `obscured`. Run `initialize.scores()` on this new matrix with NAs. Check that the new initial scores are *not* the same as the initial scores that you got on the full matrix. Using these initial scores for items and users, calculate predicted ratings for all items and users. Make a plot of true ratings against predicted ratings, using different plotting symbols (or colors, etc.) to indicate whether the true rating was obscured or not. Do these initial estimates do a better job of matching the un-obscured ratings? How can you tell?

e. (5)

Run `estimate.scores()` on the matrix `obscured` that you made in the previous problem. Using those estimated scores, calculate predicted ratings for all combinations of users and items. Plot the true ratings against the estimated ratings, as in the previous question. Do these predictions better match the un-obscured ratings than the initial predictions? Do they better match the obscured ratings than the initial predictions?

3. Keep Asking It Questions Whose Answers We Know

a. (10)

Write a function, `test.rmse()`, which takes 3 arguments:

- A ratings matrix with some obscured entries (`train`)
- The ratings matrix it was derived from (`test`)
- And a model of the kind returned by `estimate.scores` (`mdl`)

Your function should calculate predictions for all ratings, using `mdl`, and then calculate the RMSE for the ratings which are shown in `test` but were `NA` in `train`. Your code should allow for the possibility that `test` has some `NA` entries as well, but you can assume that everything which is an `NA` in `test` will also be an `NA` in train.

Using your `test.rmse()` and your `obscure` matrix, calculate, and plot, the RMSE on the hidden ratings for using from 1 to 5 factors. Why is 2 factors the best? (If it is not the best, you almost certainly have a mistake somewhere in your code.)

b. (5)

Why is what you did in the previous problem an estimate of prediction error?

4. Try It on Something Real

The `MovieLens` project was an early movie recommendation system, running during the 1997–1998 academic year. It had 943 users who gave 1–5 ratings on about 1664 movies, with about 100,000 ratings in the data set. The data is part of the `recommenderlab` package in R, as `MovieLense` (sic), but in a slightly annoying format, so I have saved it as a matrix in `MovieLense.csv`. Load the data and make sure it has the right dimensions.

a. (5)

As you saw in (1a) and (1b) above, matrix factorization has trouble if there are users who've rated too few items, or items rated by too few users, compared to the number of factors. Create two histograms, one of the number of ratings per user, and the other of the number of ratings per item. Report the number of users who have rated < 6 items, and the number of items rated by < 10 users. Remove those rows and columns from the data set. (If necessary, iterate.) Report the dimensions of the data set you are left with. Use it in all further problems, unless otherwise instructed.

b. (5)

Using the `obscured()` function, create a version of the data set which hides 1 extra ratings per user. Check that this has worked properly.

c. (5)

Do matrix factorization with between 1 and 5 latent factors on the obscured data set. Use `test.rmse()` to calculate, and plot, the RMSE of these models. What number of factors works best?

d. (5)

Repeat what you did in (4c) 9 more times, and average the RMSEs you get from each run. (Each run will obscure a different set of ratings.) Plot the results. What number of factors works best here?

Rubric (10)

The text is laid out cleanly, with clear divisions between problems and sub-problems. The writing itself is well-organized, free of grammatical and other mechanical errors, and easy to follow. All plots and tables are generated using code embedded in the document and automatically re-calculated from the data. Plots are carefully labeled, with informative and legible titles, axis labels, and (if called for) sub-titles and legends; they are placed near the text of the corresponding problem. All quantitative and mathematical claims are supported by appropriate derivations, included in the text, or calculations in code. Numerical results are reported to appropriate precision. Code is properly integrated with a tool like R Markdown or knitr, and both the knitted file and the source file are submitted. The code is indented, commented, and uses meaningful names. All code is relevant; there are no dangling or useless commands. All parts of all problems are answered with actual coherent sentences, and raw computer code or output are only shown when explicitly asked for.

Extra Credit A (5)

Suppose that when a new user joins the system, they are required to rate at least q movies already in the system before getting recommendations. Explain how to use the `estimate.user.scores()` function to then generate predictions for the user. Turn this into a way of doing cross-validation across users. Does it change any of the results for MovieLens?

Extra Credit B (5)

The reason we need at least q rating per user or per item is that otherwise we are trying to estimate more coefficients than there are observations. Ordinary least squares can't do this, but methods for high-dimensional, "$p > n$" regression are perfectly happy to do so. Modify the code provided to use ridge regression; include a way to pick the ridge penalty. Does it change any of the results for MovieLens?