Recommender Systems

Signal Data Science

In this assignment, we'll explore one way to make a recommender system, something which predicts the rating a user would give to some item. Specifically, we'll be using collaborative filtering on the MovieLens 1M Dataset, a set of one million different movie ratings. Collaborative filtering operates on the assumption that if one person A has the same opinion as another person B on item X, A is *also* more likely to have the same opinion as B on a *different* item Y than to have the opinion of a randomly chosen person on Y.

Collaborative filtering is a type of unsupervised learning and can serve as a *prelude* to dimensionality reduction (*e.g.*, with PCA or factor analysis) because filling in missing values is typically required for such methods. Specifically, we will be working with an imputation-based method of collaborative filtering, which infers *all* of the missing values from the given data.

In the following, write up your work in an R Markdown file with elaboration about *what* you're doing at each step and *why* you're doing it. Include interpretation of results as well whenever appropriate. Your goal should be to produce, at the end, an HTML (or PDF) file from the R Markdown writeup that gives a coherent and reasonably accessible description of the process you followed, the reasoning behind each step, and the results attained at the end.

Getting started

We'll first need to spend some time preparing the data before we can use any collaborative filtering methods.

- Download the MovieLens 1M Dataset. Read the associated `README.txt`, which describes the contents of the dataset.
- The first 5 lines of `ratings.dat` are:

```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
```

Use `read.csv()` with the appropriate options to load the file into R. The resulting data frame should have **1000209 rows** and **7 columns**.

- Restrict to the columns containing user IDs, movie IDs, and movie ratings. Name the columns appropriately.
- Compute the sets of `unique()` user IDs and movie IDs as well as the mean rating given. Compare the numbers of different user IDs and movie IDs with the *maximum* user ID and movie ID.
- Set the seed to **3** for consistency. Generate a training set using 80% of the data and a test set with the remaining 20%.

Because there are some movies which are rated by very few people and some people who rated very few movies, we have two corresponding problems: (1) there will be movies in the test set which were not rated by any people in the training set and (2) there will be people in the test set who do not show up in the training set. As such, we need to add to the training set a fake movie rated by every user and a fake user who rated every movie.

- Create two data frames corresponding to the above fake data using the previously calculated mean rating. (For the fake movie and user respectively, use a movie ID and user ID which are both 1 greater than their respective maximum values in the entire dataset.) When creating the fake user who has rated every movie, allow the movie IDs to range from 1 to the maximum movie ID in the dataset (which will include movie IDs not present in the dataset). The fake user should not have a rating for the fake movie.
- Perturb the ratings of the fake data slightly by adding a normally distributed noise term with mean 0 and standard deviation 0.01. Add your fake data to the training data frame, which should increase in size by **9994 rows**.

Next, we need to create a matrix containing rating data for (user, movie) pairs. We can store this as a *sparse* matrix, which is a special data structure designed for handling matrices where only a minority of the entries are filled in (because each user has only rated a small number of movies).

- Use `Incomplete()` to generate a sparse ratings matrix with one row per user ID and one column per movie ID. The resulting matrix should have **6041 rows** and **3953 columns**.

Alternating least squares

We will proceed to use the method of alternating least squares (ALS) to impute the missing entries in the sparse ratings matrix.[1]

Theoretical explanation

The operation of matrix multiplication allows us to *multiply* two matrices and form a new matrix. It is illustrated below:

$$\overset{4\times 2 \text{ matrix}}{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}} \overset{2\times 3 \text{ matrix}}{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}} = \overset{4\times 3 \text{ matrix}}{\begin{bmatrix} \cdot & x_{12} & x_{13} \\ \cdot & \cdot & \cdot \\ \cdot & x_{32} & x_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}$$

Figure 1: An illustration of matrix multiplication, where $x_{12} = a_{11}b_{12} + a_{12}b_{22}$ and $x_{33} = a_{31}b_{13} + a_{32}b_{23}$.

Note specifically the dimensions of the resulting matrix. If $\mathbf{A}$ is an $n \times p$ matrix and $\mathbf{B}$ is a $p \times m$ matrix, the product $\mathbf{AB}$ will have dimensions $n \times m$.[2] What if p is very small compared to n and m? We will be able to obtain quite a large matrix just from multiplying together two very narrow matrices (one tall and one wide).

In general, we find that it is possible to decompose large matrices into the *product* of multiple *smaller* matrices. This is the key behind the method of alternating least squares.

The task at hand is that given a matrix $\mathbf{X}$ with many missing entries, we want to construct a filled-in matrix $\mathbf{Z}$ which *minimizes some loss function*. It turns out that we can write a *regularized* cost function which makes this task straightforward, and also that we can write the solution which minimizes the cost function as

$$\mathbf{Z} \approx \mathbf{AB}^\intercal$$

for an appropriate choice of a tall matrix $\mathbf{A}$ and a wide matrix $\mathbf{B}^\intercal$, where the operator $\intercal$ denotes the *transpose* of a matrix, (flipping a $n \times m$ matrix so that its dimensions become $m \times n$). Note that for the existing data in $\mathbf{X}$ we simply use that rating data directly in the filled-in matrix $\mathbf{Z}$ instead of the approximated values in $\mathbf{AB}^\intercal$ (hence the $\approx$ symbol).

[1] Hastie *et al.* (2014), Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares.
[2] If the matrices all have real values, we can write $\mathbf{A} \in \mathbb{R}^{n\times p}$, etc.

Our imputation method is an indirect one in the sense that instead of *directly* trying to calculate missing values from existing ones, we ask what the optimal filled-in matrix $\mathbf{Z}$ would look like and infer the missing values based on an analysis of $\mathbf{Z}$. Precisely, we are trying to minimize the differences between the filled-in entries of $\mathbf{X}$ and the corresponding entries of $\mathbf{AB}^\intercal$ along with a regularization term controlled by a parameter λ.[3] Our cost function only considers the matrix entries which correspond to existing data (the filled-in values of $\mathbf{X}$), but the fashion in which we estimate $\mathbf{A}$ and $\mathbf{B}$ operate on the *entirety* of each matrix. Consequently, the entries of $\mathbf{AB}^\intercal$ corresponding to *missing* data in $\mathbf{X}$ serve as rating estimates.

Our task is now simply to estimate the matrices $\mathbf{A}$ and $\mathbf{B}$. It turns out that the optimal estimates are related via the equation

$$\mathbf{B} = \left(\mathbf{A}^\intercal\mathbf{A} + \lambda\mathbf{I}\right)^{-1}\mathbf{A}^\intercal\mathbf{Z}$$

and vice versa with $\mathbf{A}$ and $\mathbf{B}$ switched, where λ is the regularization parameter and $\mathbf{I}$ is the identity matrix.[4] For mathematical reasons, this is actually equivalent to running a regularized[5] least squares regression for each column of $\mathbf{Z}$ with the columns of $\mathbf{A}$ as predictors, with the coefficient estimates corresponding to entries of $\mathbf{B}$![6]

As such, this suggests a strategy for estimating $\mathbf{A}$ and $\mathbf{B}$. First, we start by initializing $\mathbf{A}$. Next, we use the regression strategy described above both to generate predictions for $\mathbf{Z}$ and to generate an estimate for $\mathbf{B}$. Next, we can switch the places of $\mathbf{A}$ and $\mathbf{B}$ in the above equation and use the same process to update $\mathbf{Z}$ and $\mathbf{A}$. We repeat in this *alternating* fashion until we achieve convergence.

The output of `softImpute()`

After running the algorithm described above, `softImpute()` returns the imputed matrix $\mathbf{Z}$ in a *special form*. It turns out that the product $\mathbf{AB}^\intercal$ is related to $\mathbf{Z}$

[3] One can ask the question of why we don't simply estimate $\mathbf{A} = \mathbf{B} = \mathbf{0}$ in the degenerate case of $\mathbf{X}$ having *no* missing values. If that were the result, it would contradict the fact that $\mathbf{AB}^\intercal$ should be equal to the soft-thresholded SVD of $\mathbf{Z}$ (presented later in this exposition)! The reason is that although $\mathbf{AB}^\intercal$ contains information about the *differences* between $\mathbf{X}$ and $\mathbf{Z}$, we don't try to impute those differences directly (in which case we might stop immediately if there were no differences whatsoever) but rather *infer* them by trying to bring $\mathbf{X}$ and $\mathbf{AB}^\intercal$ closer together.

[4] The identity matrix is a matrix with 1 on the diagonal and 0 elsewhere. Multiplying it by a different matrix leaves that matrix unchanged.

[5] Specifically, this is equivalent to using Tihkonov regularized linear regression with Tikhonov matrix $\Gamma = (\lambda\mathbf{I})^{1/2}$. This is also called *ridge regression* and reduces to L^2 regularization in the case where Γ is the identity matrix. We are essentially running a linear regression of each column of $\mathbf{Z}$ with the columns of $\mathbf{A}$ as predictors and getting $\mathbf{B}$ back as the coefficient estimates.

[6] We can check that the dimensions match up. Suppose that $\mathbf{Z} \in \mathbb{R}^{n\times p}$, $\mathbf{A} \in \mathbb{R}^{n\times f}$, and $\mathbf{B} \in \mathbb{R}^{p\times f}$. Then the columns of $\mathbf{Z}$ and $\mathbf{A}$ all have n entries each, and so we can run p different linear regressions (one for each column of $\mathbf{Z}$) and get out f coefficient estimates each time. We therefore estimate $p \times f$ different coefficient estimates in total, which matches up with the dimensions of $\mathbf{B}$.

in yet another fashion!

Taking a step back: in general, *all* matrices can be decomposed into a product of the form $\mathbf{UDV}^\intercal$ called the singular value decomposition (SVD) where $\mathbf{D}$ is a diagonal matrix (the only nonzero entries are on the diagonal). We can compute a *modified* version of the SVD for $\mathbf{Z}$ called the *soft-thresholded SVD* formed by taking $\mathbf{D}$ and shrinking the entries on its diagonal toward 0 by a value λ, setting an entry d_i equal to 0 if $|d_i| \leq \lambda$.[7] With the modified matrix $\mathbf{D}^\star$, we can compute the soft-thresholded SVD as $S_\lambda(\mathbf{Z}) = \mathbf{UD}^\star\mathbf{V}^\intercal$.

The connection between $\mathbf{AB}^\intercal$ and $\mathbf{Z}$ lies in the somewhat remarkable relation

$$\mathbf{AB}^\intercal = S_\lambda(\mathbf{Z})$$

for the optimal estimates of $\mathbf{A}$ and $\mathbf{B}$.

Indeed, `softImpute()` will return three matrices as `$u`, `$d`, and `$v`, corresponding to the matrices in $S_\lambda(\mathbf{Z}) = \mathbf{UD}^\star\mathbf{V}^\intercal$. From those, we also know $\mathbf{AB}^\intercal$, and so the imputed matrix $\mathbf{Z} \approx \mathbf{AB}^\intercal$ can be calculated.

Connection to dimensionality reduction

From the definition of the soft-thresholded SVD, we see that increasing λ sufficiently high will make every value in $\mathbf{D}^\star$ equal to 0. The immediate takeaway is that by calculating the maximum value in $\mathbf{D}$, we can establish an *upper bound* for the values of λ to test. However, there is a more important and subtler interpretation of the results of ALS in connection with the regularization parameter.

It is likely that the optimal value of λ is one which drives some *but not all* of the values in $\mathbf{D}$ to 0. An $n \times n$ diagonal matrix with a *rank* of k, *i.e.*, k nonzero values on the diagonal can simply be rewritten as a $k \times k$ diagonal matrix without any nonzero values on the diagonal. Our decomposition then becomes the product of (1) $\mathbf{U}$ (a tall $n \times f$ matrix), (2) $\mathbf{D}^\star$ (a small square $f \times f$ matrix), and (3) $\mathbf{V}^\intercal$ (a wide $f \times m$ matrix) for some small value of f. We can interpret this as being able to *summarize* both users and movies in terms of f factors, with the columns of $\mathbf{U}$ being factor scores for users and the rows of $\mathbf{V}^\intercal$ being factor scores for movies.

If a user has factor scores $\mathbf{u} = (u_1, u_2, \ldots, u_f)$, a movie has factor scores $\mathbf{m} = (m_1, m_2, \ldots, m_f)$, and the diagonal entries of $\mathbf{D}^\star$ are given by $\{d_1, d_2, \ldots, d_f\}$, then the predicted rating for that user–movie pair is simply given a weighted inner product of $\mathbf{u}$ and $\mathbf{m}$ equal to

[7] Soft-thresholding is basically solving a L^1 regularized cost function very rapidly by looking at the first derivative. Refer back to the theoretical discussion in *Linear Regression: Regularization* for some related details. We can therefore think of soft-thresholded SVD as a sort of L^1 regularized version of SVD which shrinks the singular values closer to 0.

$$\langle \mathbf{u}, \mathbf{m} \rangle = \mathbf{u}^\intercal \mathbf{D} \mathbf{m} = \sum_{i=1}^{f} u_i d_i m_i.$$

Using ALS on movie ratings

We're finally ready to actually apply ALS-based imputation to our ratings. First, we need to prepare our data and calculate what values of the regularization parameter λ we'll search over.

- Use `biScale()` to scale both the columns and the rows of the sparse ratings matrix with `maxit=5` and `trace=TRUE`. You can ignore the resulting warnings (increasing the number of maximum iterations doesn't improve the outcome, which you can verify for yourself).

`lambda0()` will calculate the lowest value of the regularization parameter which gives a zero matrix for **D**, *i.e.*, drives all rating estimates to zero.

- Use `lambda0()` on the scaled matrix and store the returned value as `lam0`.
- Create a vector of λ values to test by (1) generating a vector of 20 *decreasing* and uniformly spaced numbers from `log(lam0)` to 1 and then (2) calculating e^x with each of the previously generated values as x. You should obtain a vector where entries 1 and 5 are respectively 103.21 and 38.89.

Finally, we need to initialize some data structures to store the results of our computations.

- Initialize a data frame `results` with three columns: `lambda`, `rank`, and `rmse`, where the `lambda` column is equal to the previously generated sequence of values of λ to test. Initialize a list `fits` as well to store the results of alternating least squares for each value of λ.

We are now ready to impute the training data with alternating least squares. For each value of λ, we will obtain as a result of `softImpute()` factor scores for every movie and every user. As described above, we can then use those to *impute* the ratings in the test set and calculate a corresponding RMSE to evaluate the quality of the imputation in order to determine the optimal amount of regularization.

- Iterate through the calculated values of λ. For each one, do the following:
  - Use `softImpute()` with the current value of λ on the scaled sparse ratings matrix. In order to reduce computation time and find a low-dimensionality solution, constrain the rank of **D** to a maximum of 30. `rank.max=30` to restrict solutions to a maximum rank of 30 and `maxit=1000` to control the number of iterations allowed. For all but the first call of `softImpute()`, pass into the `warm.start` parameter the *previous* result of calling `softImpute()` to reduce the required

computation time via a "warm start". Read the documentation for details on what these parameters mean.

  - Calculate the *rank* of the solution by (1) rounding the values of the diagonal matrix **D** (stored in `$d`) to 4 decimal places and (2) calculating the number of nonzero entries in the rounded matrix.
  - Use `impute()` to calculate ratings for the test set using the results of `softImpute()`. (Pass in to `impute()` the calculated matrix decomposition as well as the user and movie ID columns in the test set.) Calculate the corresponding RMSE between the predicted ratings and the actual ratings.
  - Store the output of `softImpute()` in the previously initialized list `fits` as well as the calculated rank and RMSE in the corresponding row of the `results` data frame. Print out the results of the current iteration as well.

You should find that the minimum RMSE is attained at approximately $\lambda \approx 20$ with an RMSE of approximately 0.858.

- Store the best-performing soft-thresholded SVD into a variable called `best_svd`.

Analyzing the results

Now that we have a way to fill in missing entries, we can do some further analysis of the MovieLens dataset.

- As with the ratings dataset, load the movies dataset (in `movies.dat`) and name the columns appropriately.
- How many different genres are listed in the dataset? (You may find `strsplit()` helpful.) There is a single genre which is obviously the result of a data entry error. Add an appropriately named column for all of the *other* genres to serve as an *indicator variable* for whether each movie belongs to a particular genre. Fill in the entries of those columns accordingly.
- Restrict to movies which were listed at least once in the ratings dataset.

Examine the dimensions of the calculated matrix **V** in `best_svd`. The ith row corresponds to the movie with ID i and the jth column represents the "scores" for the jth "movies factor" (loosely speaking). We're interested in analyzing these "factors". To that end:

- Subset `best_svd$v` with the movie IDs in the movie dataset which remain after removing rows corresponding to movies not present in the ratings dataset. (Pay attention to the data type of the movie ID column, which is

loaded in as a *factor*.) After doing so, add the factor columns to the data frame created from the movies dataset.[8]

Next, we'll illustrate one possible path of analysis by looking at the "Drama" genre.

- Examine the correlation between the indicator variable for movies tagged as dramas and the factor columns. Using `glm()`, run an unregularized logistic regression of the indicator variables against the factors.
- Use `CVbinary()` (from the `DAAG` package) on the resulting model to generate *cross-validated probability predictions* for the whole dataset (stored in `CVbinary(fit)$cvhat`). Plot the associated ROC curve and calculate the AUC.

We now have a *probability* for each movie corresponding to how likely it is to be a drama or not given the information stored in the factor variables.

- Create a new data frame with (1) movie titles, (2) the indicator variable for dramas, and (3) the predicted probability for each movie. Order the rows from largest to smallest probability. Which movies are the most likely to be dramas and which movies are the most unlikely to be dramas? How well does this correspond with the actual genre labeling in the dataset?
- Repeat the above analysis for 3 other genres of your choice.

[8] Something like `movies = cbind(movies, best_svd$v[as.numeric(as.character(movies$mid)),])`. (Be sure to understand what this code does!)