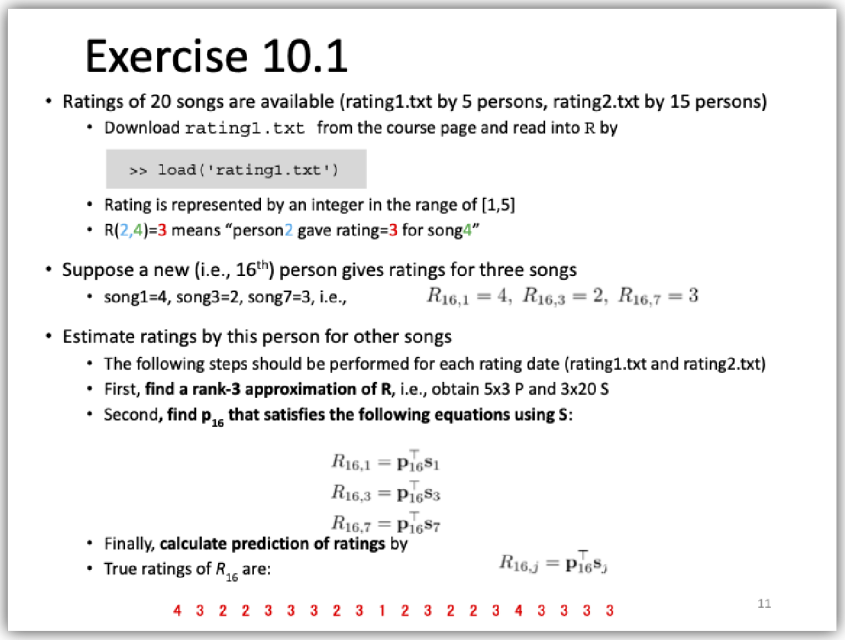
Report 10

Maximilian Fernaldy - C2TB1702



Exercise 10.1

- Ratings of 20 songs are available (rating1.txt by 5 persons, rating2.txt by 15 persons)
  - Download `rating1.txt` from the course page and read into R by

```
>> load('rating1.txt')
```

  - Rating is represented by an integer in the range of [1,5]
  - R(2,4)=3 means "person2 gave rating=3 for song4"
- Suppose a new (i.e., 16th) person gives ratings for three songs
  - song1=4, song3=2, song7=3, i.e., $R_{16,1} = 4,\ R_{16,3} = 2,\ R_{16,7} = 3$
- Estimate ratings by this person for other songs
  - The following steps should be performed for each rating date (rating1.txt and rating2.txt)
  - First, **find a rank-3 approximation of R**, i.e., obtain 5x3 P and 3x20 S
  - Second, **find $\mathbf{p}_{16}$ that satisfies the following equations using S:**

$$R_{16,1} = \mathbf{p}_{16}^{\top}\mathbf{s}_1$$
$$R_{16,3} = \mathbf{p}_{16}^{\top}\mathbf{s}_3$$
$$R_{16,7} = \mathbf{p}_{16}^{\top}\mathbf{s}_7$$

  - Finally, **calculate prediction of ratings** by $R_{16,j} = \mathbf{p}_{16}^{\top}\mathbf{s}_j$
  - True ratings of R_{16} are:

4 3 2 2 3 3 3 2 3 1 2 3 2 2 3 4 3 3 3 3

11

Introduction to Low-Rank Approximation

To completely understand low-rank approximation, a basic understanding of linear algebra is required, especially knowledge about the concepts of rank, matrix decomposition and Gaussian elimination. However, a part of it can be understood by visualization without the need of linear algebra prowess.

Imagine a matrix X. This matrix can encode data of any kind, such as pictures, exam scores, survey data, or anything. If matrix X has n columns, it doesn't necessarily mean that it has n unique columns. Some columns might be able to be expressed as other columns multiplied by some number. The number of unique columns in a matrix is what we call *rank*. Rank is useful in conveying the complexity of a data set and approximating a data set. Imagine if a video file had to contain independent images that don't correlate at all with each other. The file size would be very large and inefficient. Conversely, an image file that doesn't make use of the patterns in the image, blindly storing pixel by pixel of data will bloat in size very quickly. This is one of the most useful applications of approximation: data compression.

Now imagine adding incomplete data to matrix X, if we know that the new data has the same patterns as the other data, we can get an approximation of the actual missing data by finding those patterns in the old dataset and applying it to the new data to find the best fit.

This is where low-rank approximation comes in. For the matrix X with rank r, we can create an approximation of that matrix, with a lower rank. This approximation will allow us to interpolate missing properties of a new data point.

An example

Suppose that in a group of 5 friends, we have a movie-rating system from 1 to 5. The 5 friends have similar taste, but not exactly the same. They tend to like the same kinds of movies, which is why they established this system.

They watch the same movies each week and rate them after watching so the others can know how good a movie was.

Let's say that one week, one of the friends in the group, called John, was busy with homework from his engineering classes. He only had time to watch 2 out of the 3 movies that they agreed to watch. He still wants to watch the remaining one, but the friends need an approximation of how he would rate the movie so they can fill in a temporary data while they wait for him to watch the movie. Well, this is a perfect opportunity to use low-rank approximation.

	Movie 1	Movie 2	Movie 3
Jill	1	3	5
Jack	2	4	4
Jane	2	3	5
Jim	1	3	4
John	2	4	?

Figure 1 - The friends' movie rating system in a certain week

Jack is a mathematics major and he is a wizard at Octave. He starts up Octave and quickly began loading data and writing code:

```
M = [1,3,5; 2,4,4; 2,3,5; 1,3,4]

population = rows(M); % Number of people in the friend group
```

Loading in the movies as columns and people as rows, he made sure that his code was dynamic and that adding friends into the group would be easy. This is done by making the size of the `population` self-adjusting to the number of rows in the matrix, which means he can add new data to the matrix without breaking the code.

After this, he uses Octave's built-in SVD function to decompose the matrix.

```
[U,W,V] = svd(M); % SVD Decomposition to get L and R

% Getting the low-rank approximation of M
L = U*W;
R = V;
```

He also immediately converts the UWV matrices into the matrices L and R needed for low-rank approximation. He found the formula for converting SVD matrices into low-rank approximation here.

Next, he noticed that the dimensions need adjustment for the approximation:

```
% Use the first few columns of the decomposition matrices
decomposeRank = 2;
R = R(:,1:decomposeRank)';
L = L(:,1:decomposeRank);
```

Now the dimensions are right. To visualize, this is what the matrices look like now:

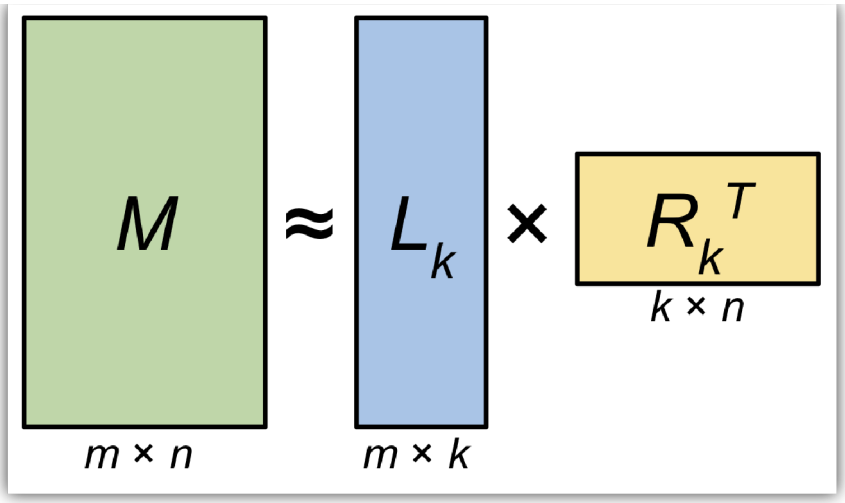


Figure 2 - Visualization of a low-rank approximation

As he only has a matrix of rank 3, he decides to use rank 2 approximation. He then adds the incomplete data he got from John:

```
% Load data of new person
M(population+1,1) = 2; M(population+1,2) = 4;
newSongRatings = [M(population+1,1), M(population+1,2)];
selectedR = [R(:,1), R(:,2)];
```

He selects the first two movies for R so that the approximation will take into account what John thinks about the first two movies as well. Satisfied with his set-up, he then finds the L for the new row by setting up this system of linear equations:

$$M_{5,1} = L_5 R_1$$
$$M_{5,2} = L_5 R_2$$

He realizes he can solve for L by inverting R:

$$M_{5,12} = L_5 R_{12}$$
$$L_5 = M_{5,12} R_{12}^{-1}$$

Note that the notation 5,12 means row 5, columns 1 and 2.

which he then converts into code:

```
L(population+1,:) = newSongRatings*inv(selectedR);
```

Finally, he can approximate John's rating for the remaining movie:

```
% Extrapolate data from the pattern
M(population+1,:) = L(population+1,:) * R;

M(population+1,:)
```

This outputs the following:

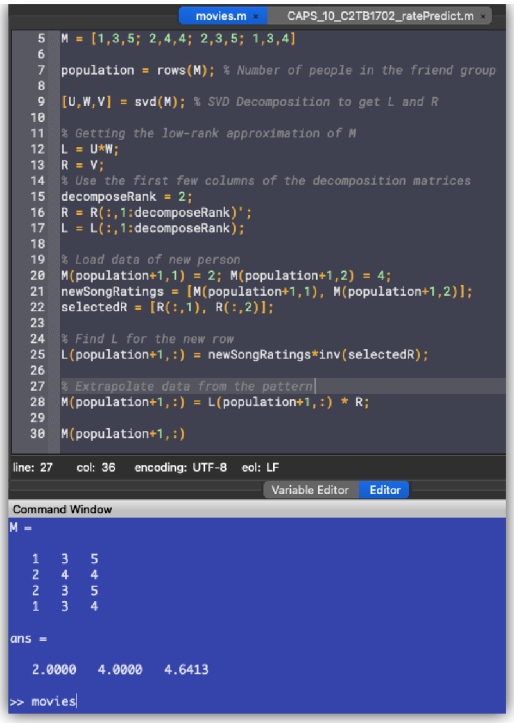


Figure 3 - Output of movies.m

John is projected to rate the movie a 4.6 out of 5. The friends can now fill in the missing data and carry on with their week.

	Movie 1	Movie 2	Movie 3
Jill	1	3	5
Jack	2	4	4
Jane	2	3	5
Jim	1	3	4
John	2	4	4.6

Figure 4 - The approximated rating predicts John will like the movie

Problem 1

In problem 1, we are asked to find a rank-3 approximation of R, which is a matrix containing rating data of 20 songs by a population of people. This is very similar to the example problem, except for some differences in the data size and naming.

For the explanation, we will use the first data set as it is smaller and easier to manage for explanation purposes. The script we will be using can work with both data sets, provided the correct data set is loaded.

First, we load the data set we want, and get the number of people who have already rated the songs.

```
% Load the rating data into R. Change datasets here.
load('rating1.txt')

population = rows(R); % Number of people rating songs
```

Next, we use the built-in SVD function to get values of P and S, which we will use for low-rank decomposition.

```
[U,W,V] = svd(R); % SVD Decomposition to get P and S
```

Since we want to get a rank 3 approximation of the matrix, we set `decomposeRank` to 3 to make sure we only use the first 3 columns of the matrices and adjust the dimensions of P and S.

```
% Getting the rank 3 approximation of R
P = U*W;
S = V;
% Use the first three columns of the decomposition matrices
decomposeRank = 3;
S = S(:,1:decomposeRank)';
P = P(:,1:decomposeRank);
```

To show the approximated matrix, we can multiply P and S according to the general low-rank approximation formula:

$$R_k = P_k S_k$$

```
rank3Approx = P*S % Rank 3 approximation of the matrix
```

Which outputs:

```
rank3Approx =

Columns 1 through 14:

   1.1739   2.7993   2.8712   4.0787   3.8064   1.7344   1.2300   4.1347   3.0296   4.0945   4.0945   2.0226   3.0455   1.7661
   1.7199   2.2751   2.4816   2.8836   2.9679   2.3123   2.0536   3.2172   2.7936   2.7564   2.7564   2.1008   2.6664   2.0580
   4.1849   2.7776   1.9149   2.0855   2.7292   4.6959   4.3245   2.2252   3.0015   1.0832   1.0832   3.0499   1.9992   2.6913
   3.8519   3.1881   2.0116   2.9293   3.2879   4.2675   3.6525   2.7299   3.0315   1.9522   1.9522   2.9317   2.0544   2.3131
   2.0365   2.9846   3.8205   4.0108   4.1505   3.0084   2.8127   4.7870   4.0944   4.0706   4.0706   2.9285   4.1542   3.1279

Columns 15 through 20:

   2.2370   1.9647   1.9665   3.0716   2.0226   3.8624
   2.7464   2.8308   1.7671   2.1907   2.1008   3.3016
   4.2762   4.9199   2.9103   2.1339   3.0499   2.8689
   3.7523   4.1109   3.1311   2.8295   2.9317   3.0885
   3.9786   4.1182   2.1524   2.8450   2.9285   4.9266
```

Figure 5 - The rank 3 approximation of the ratings matrix

Problem 2

For the next step, we want to extrapolate missing data from a new person.

We can load the incomplete data of the new person into the data set, adding a new row.

```
% Load data of new person
R(population+1,1) = 4; R(population+1,3) = 2; R(population+1,7) = 3;
newSongRatings = [R(population+1,1), R(population+1,3), R(population+1,7)];
selectedS = [S(:,1), S(:,3), S(:,7)];
```

This time, instead of 2 known movie ratings, we have 3 song ratings. However, there are 20 songs in total, which means we have to approximate a staggering 17 ratings from the new person. Since the songs that the person rated were not the first 3, we also need a more specific indexing method than the one used in the example problem. We select all the elements in column 1, 3 and 7 of S, because those are the columns that we have the definite data from the new person.

Finally, we can find the new row of P, which is analogous to L in the example problem, by multiplying the array containing the known song ratings with the invert of the array containing the corresponding R entries. Similar to the example problem, for the first data set we have:

$$R_{6,1} = p_6 s_1$$
$$R_{6,3} = p_6 s_3$$

$$R_{6,7} = p_6 s_7$$

$$R_6 = P_6 S_{137}$$
$$P_6 = R_6 S_{137}^{-1}$$

However, we must realize that with bigger data sets, inverse matrices take increasingly longer to compute, which will slow down the extrapolation process by a lot when working with a large enough data set. Instead, we should use the Gaussian elimination. However, we want to solve for P, which does not fit into the general Gaussian Elimination formula:

$$AX = B, \text{ solve for } X$$

Luckily, we can use transposition in this manner:

$$R_{6,137} = P_6 S_{137} \Rightarrow R_{6,137}^T = S_{137}^T P_6^T \Rightarrow P_6^T = S_{137}^T \backslash R_{6,137}^T$$

Note: the notation 6,137 means the sixth row, columns 1, 3 and 7. The backslash indicates the gaussian elimination operator in Octave. Such notation doesn't actually exist in maths, but this suffices for the purpose of the report.

By using Gaussian elimination, we can get the transpose of P_6, which means if we just transpose it again, we will get the sixth, new row of P. In code form:

```
% Gaussian elimination is faster than inverse matrices in large data sets.
P(population+1,:) = (selectedS'\newSongRatings')';
```

Finally, we can approximate the missing 17 data properties using the newly acquired row of P.

```
R(population+1,:) = P(population+1,:) * S;
```

Which is just a row-specific form of the general approximation formula: $R = PS$. Displaying only the new row, we get these values:

```
ans =

Columns 1 through 14:

  4.0000   4.2335   2.0000   4.4395   4.3833   4.1772   3.0000   3.4395   3.1498   3.2061   3.2061   3.0000   1.9163   1.7103

Columns 15 through 20:

  3.1498   3.0935   4.0000   4.2897   3.0000   3.3833

>>
```

Figure 6 - Output of ratePredict.m

Comparing this to the actual ratings, we see that excluding the known data in songs 1, 3 and 7, there are serious discrepancies between the approximated rating and the actual rating:

4 3 2 2 3 3 3 2 3 1 2 3 2 2 3 4 3 3 3 3

Figure 7 - The real ratings of the new person

To quantify this, let's take the mean differences of the datasets:

$$M.D. = \frac{\sum_{i=1}^{n} |x_i - y_i|}{n}$$

This formula compares each corresponding values of the two datasets, finds their difference, adds them up and divides them by the number of data points. Applying this to Octave:

```
% Average difference
differences = abs(R(population+1,:) - actualratings)
meanDifference = sum(differences)/songs
```

Running this with the first dataset outputs:

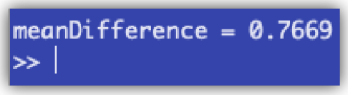


Figure 8 - The mean difference is high

Which means each rating is off by 0.77 points on average. This high mean difference is due to many reasons, some of which include the very low data sample size and the unknown relationship between the listeners if there is even a relationship at all. It might be that the six listeners has nothing in common with each other, which means we cannot use the patterns in the data to extrapolate ratings. This is why we need to understand algorithms to use them. We need to understand what an algorithm takes into account, what it does exactly with the data, what its strengths and limitations are. Because no algorithm is perfect and all-knowing. Data can be manipulative and sneaky a lot of times. A complete and thorough understanding of the algorithm we use will help in getting better predictions and extrapolated data.

Output of the second data set

Does having more data points to start with make the prediction more accurate? We find out by replacing the dataset to a larger one:

```
% Load the rating data into R. Change datasets here.
load('rating2.txt')
```

which outputs:

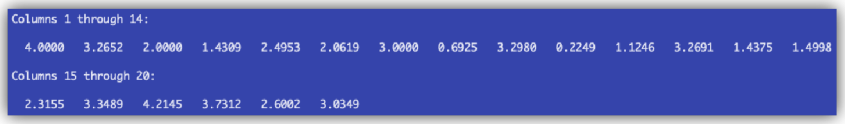


Figure 9 - Approximated ratings of the sixteenth person

We find out that the mean deviation is considerably less than the smaller dataset's:

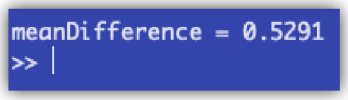


Figure 10 - The mean difference has gone down a considerable amount

This is still a fairly inaccurate approximation of the listener's taste. To get better approximations, a data scientist should find out the relationship between the different listeners. Do they tend to gravitate toward the same genre of music? Are they from the same region of the world? What sort of preferences do they have in common? Knowing these factors can help algorithms cluster data points into better groups, which will drive the accuracy of prediction models upwards.

Footnotes

From this exercise, we have learned that matrix decomposition and approximations are very useful in digital file handling, big data processing and pattern predictions. However, the complete understanding of algorithms is needed to utilize them to their full potential. Otherwise, predictions might just as well be random.

Additionally, it can be learned that when writing code for data, it's best to generalize when possible. This allows for dynamic data handling, which means the code does not need to be changed to accommodate changes in data size. Not only does this save time and energy, it's cleaner and more efficient.