Recommendation Systems

Luana Ruiz and Alejandro Ribeiro

October 7, 2020

In a recommendation system, we want to predict the ratings that customers would give to a certain product using the product's rating history and the ratings that these customers have given to similar products. Collaborative filtering solutions build a graph of product similarities using past ratings and consider the ratings of individual customers as graph signals supported on the nodes of the product graph. The underlying assumption is that there exist an underlying set of true ratings or scores as shown in Figure 1-(a), but that we only observe a subset of those scores as in Figure 1-(b). The set of scores that are not observed can be estimated from the set of scores that have been observed. This problem can thus be seen as an ERM problem, and our goal will be compare the ability of several learning parametrizations to solve it.

1 Data generation

To illustrate the problem of recommendation systems with a specific numerical example, we use the MovieLens-100k dataset. This dataset's zip folder can be downloaded from the class website and a description of the files is provided here. The MovieLens-100k dataset consists of 100,000 ratings given by $U = 943$ users to $M = 1682$ movies. The existing movie ratings are integer values between 1 and 5.

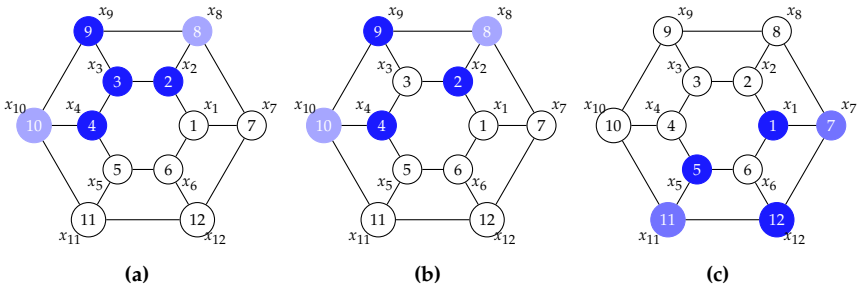


Figure 1. The graph represents product similarity in a recommendation system. If we are given samples (a) for training, any reasonable parametrization learns to complete the rating of node 3 when observing the signal in (b). Notably, the graph filter parametrization also generalizes to predicting the rating of node 6 in (c) when observing the signal in (b). This is true because of permutation equivariance. Graph neural networks inherit this generalization property from graph filters.

1.1 Loading the data. Download the MovieLens-100k dataset and store the movie ratings in a $U \times M$ numpy matrix $\mathbf{X}$ where $[\mathbf{X}]_{um} = x_{um}$ is the rating given by user u to movie m. If user u has not rated movie m, set $x_{um} = 0$.

1.2 Data clean-up. To make the data more meaningful, we will only consider movies rated by 150 or more users. Remove the columns of $\mathbf{X}$ with less than 150 nonzero entries. Record the new column index of the movie "Contact", which has index 257 in the original matrix (note that indices start at 0). *Obs.*: There should be no rows of all zeros, i.e., no users with empty rating entries in the "cleaned-up" matrix. If there were empty rows, we'd make sure to remove them as well.

The first step to building our recommendation system is to use the rating history of all movies to compute a graph of movie similarities, in which edges represent a similarity score between different movies. Recall that x_{um} is the rating that user u gives to movie m. Typically, movie m has been rated by a subset of users which we denote $\mathcal{U}_m$. We consider the sets of users $\mathcal{U}_{\ell m} = \mathcal{U}_\ell \cap \mathcal{U}_m$ that have rated movies ℓ and m and compute

correlations

$$\sigma_{\ell m} = \frac{1}{|\mathcal{U}_{\ell m}|} \sum_{u \in \mathcal{U}_{\ell m}} (x_{u\ell} - \mu_{\ell m})(x_{um} - \mu_{m\ell}), \tag{1}$$

where we use the average ratings $\mu_{\ell m} = (1/|\mathcal{U}_{\ell m}|) \sum_{u \in \mathcal{U}_{\ell m}} x_{u\ell}$ and $\mu_{m\ell} = (1/|\mathcal{U}_{\ell m}|) \sum_{u \in \mathcal{U}_{\ell m}} x_{um}$. The movie graph used in collaborative filtering is the one with normalized weights

$$w_{\ell m} \;=\; \sigma_{\ell m} \Big/ \sqrt{\sigma_{\ell\ell}\sigma_{mm}}\,. \tag{2}$$

An important point to note is that the movie similarity graph should only be built from training data. In order to do this, we split the users between a training and a test set and only use the ratings of the users in the training set to calculate the correlations $\sigma_{\ell m}$ following (1).

1.3 Movie similarity graph. Randomly permute the rows of the rating matrix $\mathbf{X}$ (i.e., the users) and split them between 90% for training and 10% for testing. *Make sure to save the indices of the users in the training and test sets.* Using only the rows of $\mathbf{X}$ corresponding to the users in the training set, generate the adjacency matrix $\mathbf{W}$ following (1) and (2). Set the diagonal elements of $\mathbf{W}$ to zero. What is the number of nodes N of the movie similarity graph?

1.4 Sparsifying and normalizing the graph. Sparsify the matrix $\mathbf{W}$ by only keeping the 40 largest entries of each row and zero-ing out all other entries. This makes it so that each node is only connected to its 40 nearest-neighbors. Calculate the eigendecomposition of the matrix $\mathbf{W}$ and normalize it by the eigenvalue with largest absolute value.

The next step is to build the input-output samples of the training and test sets. Define the vector $\mathbf{x}_u = [x_{u1}; \dots x_{uN}]$ where x_{um} is the rating that user u gave to movie m, if available, or $x_{um} = 0$ otherwise. Further denote as $\mathcal{M}_u$ the set of movies rated by user u. Let $m \in \mathcal{M}_u$ be a movie rated by user u and define the sparse vector $\mathbf{y}_{um}$ whose unique nonzero entry is

$[\mathbf{y}_{um}]_m = x_{um}$. With these definitions we construct the training set

$$\mathcal{T} = \bigcup_{u, m \in \mathcal{M}_u} \big\{ (\mathbf{x}_{um}, \mathbf{y}_{um}) \,:\, \mathbf{x}_{um} = \mathbf{x}_u - \mathbf{y}_{um} \big\}. \tag{3}$$

This process is repeated for all the movies in the set $\mathcal{M}_u$ and for all users u.

In this lab, we will focus on predicting the ratings given to the movie Contact. Therefore, we only consider graph signals $\mathbf{x}_u$ corresponding to users u who have rated Contact.

1.5 Training data. Remove the users who have not rated Contact from the training set defined in 1.3. Then, generate the training data according to (3), i.e., construct input-output pairs $(\mathbf{x}_{um}, \mathbf{y}_{um})$ for users u in the training set. How many samples are there in the training set?

1.6 Test data. Remove the users who have not rated Contact from the test set defined in 1.3. Then, generate the test data according to (3), i.e., construct input-output pairs $(\mathbf{x}_{um}, \mathbf{y}_{um})$ for users u in the test set. How many samples are there in the test set?

2 Learning to predict ratings

Our goal is to learn a map that will produce outputs $\mathbf{y}_{um}$ when presented with inputs $\mathbf{x}_{um}$. E.g., in the case of Figure 1 we want to present Figure 1-(b) as an input and fill in a rating of movie $m = 3$ equal to the rating of movie $m = 3$ in Figure 1-(a). To do that we define the loss function

$$\ell\big(\Phi(\mathbf{x}_{um}; \mathcal{H}), \mathbf{y}_{um}\big) \;=\; \frac{1}{2}\Big(\mathbf{e}_m^T \Phi(\mathbf{x}_{um}; \mathcal{H}) - \mathbf{e}_m^T \mathbf{y}_{um}\Big)^2, \tag{4}$$

where the vector $\mathbf{e}_m$ is the mth entry of the canonical basis of $\mathbb{R}^n$. Since multiplying with $\mathbf{e}_m^T$ extracts the mth component of a vector, the loss in (4) compares the predicted rating $\mathbf{e}_m^T \Phi(\mathbf{x}_{um}; \mathcal{H}) = [\Phi(\mathbf{x}_{um}; \mathcal{H})]_m$ with the observed rating $\mathbf{e}_m^T \mathbf{y}_{um} = [\mathbf{y}_{um}]_m = x_{um}$. At execution time, this map can

be used to predict ratings of unrated movies from the ratings of rated movies. If we encounter the signal in Figure 1-(b) during execution time, we know the prediction will be accurate because we encountered this signal during training. If we are given the signal in Figure 1-(c), successful rating predictions depend on the choice of parametrization. Similarly to what we did in Lab 2, we will implement several parametrizations of $\Phi(\mathbf{x}_{um}; \mathcal{H})$ and compare their ability to accurately predict ratings for the movie Contact.

2.1 Loss function. Write a function that computes the loss defined in (4) for a specific movie index (in our case, this index will correspond to "Contact"). *Hint:* you can instantiate the PyTorch class `torch.nn.MSELoss` to compute the mean squared error.

While you have the option to use the modules you implemented in Lab 2, in this lab we recommend that you use the learning architectures from the Alelab GNN library. This is an extensive PyTorch library with faster implementations of several graph neural network architectures. A particularly useful module for this lab is the class `LocalGNN`, which can be found under `Modules/architectures.py`. Check its documentation to understand how it works. Also check `Utils/graphML.py`, which contains all the graph machine learning modules (filters, pooling operators, and nonlinearities) that can be used to parametrize the `LocalGNN`.

An important point to note is that in the `LocalGNN` we distinguish between two types of layers: graph convolutional layers and readout layers. Each graph convolutional layer consists of an arbitrary multi-feature graph filter followed by a pointwise nonlinearity. In contrast, readout layers consists of a graph filter (or graph filter bank) in which K (the number of filter taps) is necessarily 1. Therefore, readout layers only mix the feature values of each individual node, i.e., they do not carry out local exchanges between neighboring nodes. Any number of graph convolutional and readout layers can be defined (including none), but the readout layers must always succeed the graph convolutional layers.

In the following experiments, we ask that you train all architectures using ADAM with step size $\epsilon = 0.005$, $E = 40$ epochs and batch size $Q_t = 5$.

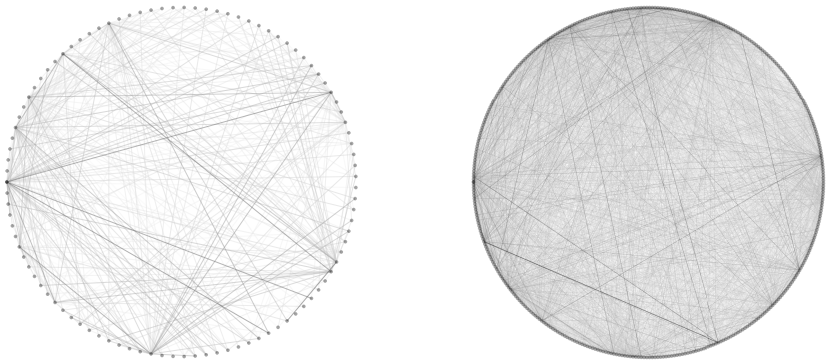
2.2 Linear parametrization vs. graph filter. Implement a generic linear parametrization (i.e., a matrix mapping N nodes to N nodes) to solve the recommendation problem. Then, use the `LocalGNN` class to instantiate a graph filter consisting of 1 convolutional layer with $F_1 = 64$ and $K_1 = 5$, and of 1 readout layer with $F_2 = 1$. Train and test both parametrizations on 5 or more random realizations of the training-test split. Report the number of parameters of both models and the mean RMSE achieved by each of them. What do you observe?

2.3 FCNN vs. GNN. Implement a FCNN with 2 layers and 64 hidden units in the first layer to solve the recommendation problem. Then, use the `LocalGNN` class to instantiate a GNN consisting of 1 convolutional layer with $F_1 = 64$, $K_1 = 5$ and ReLU nonlinearity, and of 1 readout layer with $F_2 = 1$. Train and test both parametrizations on 5 or more random realizations of the training-test split. Report the number of parameters of both models and the mean RMSE achieved by each of them. What do you observe?

2.4 Graph filter vs. GNNs. Use the `LocalGNN` class to instantiate a GNN consisting of 2 convolutional layers with $F_1 = 64$, $F_2 = 32$, $K_1 = K_2 = 5$ and ReLU nonlinearity, and of 1 readout layer with $F_2 = 1$. Train and test this GNN, the GNN from 2.3 and the graph filter from 2.2 on 5 or more random realizations of the training-test split. Report the number of parameters of all models and the RMSE achieved by each of them. What do you observe?

3 Transferring recommendation systems

In most practical scenarios in which recommendation systems are used, the product portfolio is constantly changing. An example is the Netflix movie catalog, which gets new movie and TV shows added to it every month. As discussed in Lab 2, when graphs are dynamic it is sometimes impractical to re-train the GNN every time the graph changes. If the product portfolio is too large, it may also be too costly to train the GNN on the full product graph. In general, we want to be able to train GNNs



(a) (b)

Figure 2. Movie networks built from the ratings of 100 (a) ad 400 (b) movies in the MovieLens 100k dataset. The signals on these graphs correspond to the ratings given by user 1. The darker the node, the higher the rating, and the darker the edge, the higher the rating difference between the endnodes.

on moderately sized graphs and apply them to similar graphs that are large and/or dynamic without much performance degradation. It turns out that this is possible in graphs such as the movie graphs in Figure 2 where, even if nodes are added or removed, a certain overall structure is preserved. This transference property of GNNs, also known as *transferability*, has already been illustrated in the synthetic source localization example of Lab 2. Here, we analyze how it holds up in the more realistic scenario of movie recommendation.

3.1 Training the GNN on a small graph. Train and test one and two-layer GNNs with same hyperparameters as in 2.4 using the graph and the data you generated in Section 1. For 5 or more realizations of the train-test split, report the test RMSE achieved by each model and save both models.

3.2 Large graph. Repeat item 1.2 to consider movies rated by at least 50 users and store the data in a matrix $\mathbf{X}_2$. Then, repeat items 1.3–1.4 using $\mathbf{X}_2$ and the training and test set user indices you saved in 1.3 to generate the new adjacency matrix $\mathbf{W}_2$. What is the new number of nodes N_2?

3.3 Transferability to large graph. Use the function `changeGSO` of `LocalGNN` to change the GSO of the GNNs trained in 3.1 to $\mathbf{W}_2$. Repeat item 1.6 to generate the new test data from $\mathbf{X}_2$ for each data split. Test the GNNs on this new test data and report the RMSEs for 5 or more realizations of the train-test split. What do you observe?

3.4 Larger graph. Repeat item 1.2 to consider movies rated by at least 10 users and store the data in a matrix $\mathbf{X}_3$. Then, repeat items 1.3–1.4 using $\mathbf{X}_3$ and the training and test set user indices you saved in 1.3 to generate the new adjacency matrix $\mathbf{W}_3$. What is the new number of nodes N_3?

3.5 Transferability to larger graph. Use the function `changeGSO` of `LocalGNN` to change the GSO of the GNNs trained in 3.1 to $\mathbf{W}_3$. Repeat item 1.6 to generate the new test data from $\mathbf{X}_3$ for each data split. Test the GNNs on this new test data and report the RMSEs for 5 or more realizations of the train-test split. What do you observe?

4 Report

Do not take much time to prepare a lab report. We do not want you to report your code and we don't want you to report your work. Just give us answers to items we ask. Specifically, give us the following:

Item	**Report deliverable**
Item 1.1	Do not report.
Item 1.2	Do not report.
Item 1.3	Number of nodes of the graph.
Item 1.4	Do not report.
Item 1.5	Number of training samples.
Item 1.6	Number of test samples.
Item 2.1	Do not report.
Item 2.2	Number of parameters. Average RMSEs. Comment.
Item 2.3	Number of parameters. Average RMSEs. Comment.
Item 2.4	Number of parameters. Average RMSEs. Comment.
Item 3.1	RMSEs.
Item 3.2	Number of nodes of the graph.
Item 3.3	RMSEs. Comment.
Item 3.4	Number of nodes of the graph.
Item 3.5	RMSEs. Comment.

We will check that your answers are correct. If they are not, we will get back to you and ask you to correct them. As long as you submit responses, you get an A for the assignment. It counts for 16% of your final grade.