Computer Science 189 HW7

Shrihan Agarwal

May 5, 2022

1 Question 1: Honor Code

I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted.

Signed, Shrihan Agarwal 

People I worked with: Suhrid Saha (SID : 3034893174)

2 Question 2: Movie Recommender System

(a) From singular value decomposition, we have that,

$$\begin{aligned} R = UDV^T &= \sum_{k=1}^{m} \sigma_k u_k (v_k)^T \\ \implies R_{ij} &= \sum_{k=1}^{m} \sigma_k (u_k (v_k)^T)_{ij} \\ &= \sum_{k=1}^{m} \sigma_k (u_k)_i (v_k)_j \\ &= \sum_{k=1}^{m} \sigma_k U_{ik} V_{jk} \\ &= U_i D (V_j)^T \end{aligned}$$

where u_i, v_i are the ith **columns** of U and V respectively, and U_i, V_i are defined as the ith **rows** of U and V. Here, U is $n \times m$, D is $m \times m$ and V is $m \times m$.

(b) Based on the above answer, to get a training accuracy of 100%, we should use the ith row of U $x_i = U_i$, and D times the jth column of V, $y_i = DV_j^T$, as the feature vectors. This gives us $x_i \cdot y_j \approx R_{ij}$.

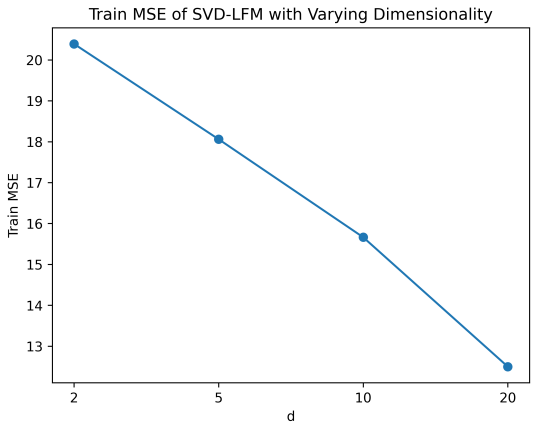
(c) Code for this part:

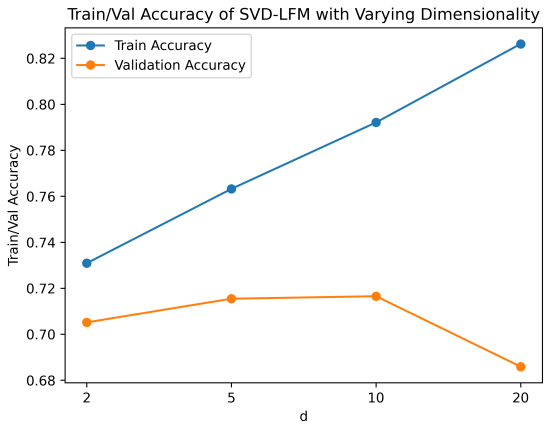
```python
# Part (c): SVD to learn low-dimensional vector representations
def svd_lfm(R):

    r = np.copy(R)

    # Fill in the missing values in R
    r[np.isnan(r)] = 0

    # Compute the SVD of R
    U, s, Vh = scipy.linalg.svd(r, full_matrices = False)

    # Construct user and movie representations
    user = U
    movie = np.diag(s) @ Vh


    user_vecs = user
    movie_vecs = movie.T
    return user_vecs, movie_vecs
```

(d) Code for this part:

```python
#Part (d): Compute the training MSE loss of a given vectorization
def get_train_mse(R, user_vecs, movie_vecs):

    UDVT = user_vecs @ movie_vecs.T

    r = np.copy(R)
    k = np.isnan(r)
    r[k] = 0
    UDVT[k] = 0
    mse_loss = np.linalg.norm(r - UDVT) ** 2

    return mse_loss
```

(e) Plots for this part:





The final values are:
MSE = 14165432.75
Training Accuracy: 0.7921
Validation Accuracy: 0.7165

for a value of d = 10. This is the point at which overfitting is minimized while simultaneously maintaining sufficient information for a good validation accuracy.

(f) The Loss function is, on converting to matrix form:

$$L = \|XY^T - R\|_F^2 + \|X\|_F^2 + \|Y\|_F^2$$

For batched operations, we calculated the derivative with respect to matrix X, holding Y constant, and then Y holding X constant, as is suggested.

$$\begin{aligned}
\frac{\partial L}{\partial X} &= 2(XY^T - R)Y + 2X = 0 \\
&\implies (XY^T - R)Y + X = 0 \\
&\implies XY^TY - RY + X = 0 \\
&\implies X(Y^TY + I) - RY = 0 \\
&\implies X(Y^TY + I) = RY \\
&\implies X = RY(Y^TY + I)^{-1}
\end{aligned}$$

Here, the inverse always exists. This is because it is a PSD matrix, where Y^TY is PSD (PSD is satisfied by a matrix that can be composed into UU^T)and I is PSD, giving us a new PSD matrix. For $\frac{\partial L}{\partial V}$, we instead take the derivative with respect to V^T and take the transpose.

$$\begin{aligned}
\frac{\partial L}{\partial Y^T} &= 2X^T(XY^T - R) + 2Y^T = 0 \\
&\implies X^TXY^T - X^TR + Y^T = 0 \\
&\implies (X^TX + I)Y^T - X^TR = 0 \\
&\implies Y^T = (X^TX + I)^{-1}X^TR \\
&\implies Y^T = (I + I)^{-1}X^TR \\
&\implies Y = \frac{R^TX}{2}
\end{aligned}$$

Without batches, the loss is,

$$L = \sum_{i=1}\sum_{j\in S_i}(x_i \cdot y_j - R_{ij})^2 + \sum_{i=1}\|x_i\|^2 + \sum_{i=1}\|y_i\|^2$$

and so the derivative is,

$$\begin{aligned}
\frac{\partial L}{\partial x_i} &= \sum_{j\in S_i} 2(x_i \cdot y_j - R_{ij})y_j + 2x_i = 0 \\
&\implies \sum_{j\in S_i} 2(x_i \cdot y_j - R_{ij})y_j + 2x_i = 0 \\
&\implies \sum_{j\in S_i}(x_i \cdot y_j)y_j - \sum_{j\in S_i} R_{ij}y_j + x_i = 0 \\
&\implies \sum_{j\in S_i} y_j(x_i^Ty_j) - \sum_{j\in S_i} R_{ij}y_j + x_i = 0 \\
&\implies \sum_{j\in S_i} y_j(y_j^Tx_i) - \sum_{j\in S_i} R_{ij}y_j + x_i = 0 \\
&\implies \left(\sum_{j\in S_i} y_jy_j^T\right)(x_i) - \sum_{j\in S_i} R_{ij}y_j + x_i = 0 \\
&\implies \left(I + \sum_{j\in S_i} y_jy_j^T\right)x_i - \sum_{j\in S_i} R_{ij}y_j = 0
\end{aligned}$$

$$\implies \left(I + \sum_{j \in S_i} y_j y_j^T\right) x_i = \sum_{j \in S_i} R_{ij} y_j$$

$$\implies x_i = \left(I + \sum_{j \in S_i} y_j y_j^T\right)^{-1} \left(\sum_{j \in S_i} R_{ij} y_j\right)$$

With respect to y_j,

$$\frac{\partial L}{\partial y_j} = \sum_{i \in S_j} 2(x_i \cdot y_j - R_{ij}) x_i + 2y_j = 0$$

$$\implies \sum_{i \in S_j} (x_i \cdot y_j - R_{ij}) x_i + y_j = 0$$

$$\implies \sum_{i \in S_j} (x_i \cdot y_j) x_i - \sum_{i \in S_j} R_{ij} x_i + y_j = 0$$

$$\implies \sum_{i \in S_j} (x_i \cdot y_j) x_i - \sum_{i \in S_j} R_{ij} x_i + y_j = 0$$

$$\implies \left(\sum_{i \in S_j} x_i x_i^T\right) (y_j) - \sum_{i \in S_j} R_{ij} x_i + y_j = 0$$

$$\implies y_j = \left(I + \sum_{i \in S_j} x_i x_i^T\right)^{-1} \sum_{i \in S_j} R_{ij} x_i$$

On trying batching, we found that the results weren't providing as stable a solution, with a varying MSE. Instead, we opted to loop over the training points in the end, as shown in the calculations above. With this method, we iteralively performed alternative minimization. The results were:

```
(astroconda) shri@Shrihans-MacBook-Air hw7 % python movie_recommender.py
movie_recommender.py:43: VisibleDeprecationWarning: Creating an ndarray from ragged nest
ys with different lengths or shapes) is deprecated. If you meant to do this, you must sp
  return np.array(user_rated_idxs), np.array(movie_rated_idxs)
Start optim, train MSE: 27574866.30, train accuracy: 0.5950, val accuracy: 0.5799
Iteration 1, train MSE: 13421216.24, train accuracy: 0.7611, val accuracy: 0.6431
Iteration 2, train MSE: 11474959.41, train accuracy: 0.7876, val accuracy: 0.6789
Iteration 3, train MSE: 10493324.86, train accuracy: 0.8007, val accuracy: 0.6989
Iteration 4, train MSE: 10040997.98, train accuracy: 0.8069, val accuracy: 0.7084
Iteration 5, train MSE: 9792296.83, train accuracy: 0.8098, val accuracy: 0.7100
Iteration 6, train MSE: 9649312.88, train accuracy: 0.8117, val accuracy: 0.7100
Iteration 7, train MSE: 9561491.69, train accuracy: 0.8130, val accuracy: 0.7060
Iteration 8, train MSE: 9503837.41, train accuracy: 0.8138, val accuracy: 0.7117
Iteration 9, train MSE: 9463660.97, train accuracy: 0.8144, val accuracy: 0.7111
Iteration 10, train MSE: 9434168.95, train accuracy: 0.8147, val accuracy: 0.7087
Iteration 11, train MSE: 9411512.64, train accuracy: 0.8150, val accuracy: 0.7119
Iteration 12, train MSE: 9393397.49, train accuracy: 0.8152, val accuracy: 0.7103
Iteration 13, train MSE: 9378404.19, train accuracy: 0.8155, val accuracy: 0.7125
Iteration 14, train MSE: 9365635.88, train accuracy: 0.8156, val accuracy: 0.7122
Iteration 15, train MSE: 9354518.75, train accuracy: 0.8157, val accuracy: 0.7125
Iteration 16, train MSE: 9344681.51, train accuracy: 0.8158, val accuracy: 0.7136
Iteration 17, train MSE: 9335879.18, train accuracy: 0.8159, val accuracy: 0.7144
Iteration 18, train MSE: 9327944.20, train accuracy: 0.8160, val accuracy: 0.7146
Iteration 19, train MSE: 9320755.69, train accuracy: 0.8161, val accuracy: 0.7149
Iteration 20, train MSE: 9314221.76, train accuracy: 0.8163, val accuracy: 0.7160
```

For (e), the best values are:
MSE = 14165432.75
Training Accuracy: 0.7921
Validation Accuracy: 0.7165

For (f), the final values are:
MSE = 9314221.76
Training Accuracy: 0.8163
Validation Accuracy: 0.7160

Comparing to (e), we find the validation accuracy is about the same. In (f), the training accuracy is slightly better, however.

3 Question 3: Regularized and Kernel k-Means

(a) When the number of clusters is equal to the number of sample points, each sample point forms its own cluster. This means that the mean of the sample point is in the same position as the point itself, and hence the objective L is simply 0.

(b) The function to be minimized is,

$$\mu_i^* = \min_{\mu_i \in \mathbb{R}^d} \lambda \|\mu_i\|_2^2 + \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2$$

Taking the derivative with respect to μ_i to be 0, we get,

$$\begin{aligned}
&\frac{\partial}{\partial \mu_i}\left(\lambda\|\mu_i\|_2^2 + \sum_{X_j \in C_i} \|X_j - \mu_i\|_2^2\right) = 0 \\
&\implies 2\lambda\mu_i - \sum_{X_j \in C_i} 2(X_j - \mu_i) = 0 \\
&\implies \lambda\mu_i - \sum_{X_j \in C_i} X_j + |C_i|\mu_i = 0 \\
&\implies \mu_i = \frac{\sum_{X_j \in C_i} X_j}{\lambda + |C_i|}
\end{aligned}$$

as expected.

(c) We have to assign the sample point to the "closest" class. In order to do this, we need to compute the best class $\arg\min_k \|\phi(X_j) - \mu_k\|^2$ and assign each sample point to it. This is the missing step in the algorithm. We can find this class in terms of the kernel function by,

$$\begin{aligned}
\arg\min_k \|\phi(X_j) - \mu_k\|^2 &= \arg\min_k (\phi(X_j) - \mu_k)^T(\phi(X_j) - \mu_k) \\
&= \arg\min_k \left(\phi(X_j) - \frac{1}{|S_k|}\sum_{i \in S_k} \Phi(X_i)\right)^T \left(\phi(X_j) - \frac{1}{|S_k|}\sum_{i \in S_k} \Phi(X_i)\right) \\
&= \arg\min_k \left(\phi(X_j)^T - \frac{1}{|S_k|}\sum_{i \in S_k} \Phi(X_p)^T\right) \left(\phi(X_q) - \frac{1}{|S_k|}\sum_{i \in S_k} \Phi(X_i)\right) \\
&= \arg\min_k \left(\phi(X_j)^T\phi(X_j) - \frac{2}{|S_k|}\sum_{i \in S_k} \Phi(X_i)^T\Phi(X_j) - \frac{1}{|S_k|^2}\sum_{j \in S_k}\sum_{i \in S_k} \Phi(X_j)^T\Phi(X_i)\right) \\
&= \arg\min_k \left(\kappa(X_j, X_j) - \frac{2}{|S_k|}\sum_{i \in S_k} \kappa(X_j, X_i) - \frac{1}{|S_k|^2}\sum_{p \in S_k}\sum_{q \in S_k} \kappa(X_p, X_q)\right)
\end{aligned}$$

This is calculable by using the kernel function and hence evading the calculation of Φ.

(d) Since the algorithm initially computes for each point n, for each class k, n^2 computations, the overall runtime of the algorithm is $O(kn^3)$. This is very slow. I will try to use memoization here. We can notice that multiple calculations of $\kappa(X_i, X_j)$ are repeated, and so instead store the results of each computation in a n $\times$ n matrix for each combination of X_i and X_j. Now, we only need to compute the matrix once in time $O(n^2)$, and the rest is simply involving constant lookup times throughout. This significantly speeds up our computations.

Additionally, the third term:

$$\frac{1}{|S_k|^2} \sum_{p \in S_k} \sum_{q \in S_k} \kappa(X_p, X_q)$$

is constant for all datapoints n. We can compute and save this for every class k and then simply use the same value for all n until the next iteration of the algorithm, when we'll have to re-compute the value do to the changing set S_k. This adds another speed boost.

4 Question 4: The Training Error of AdaBoost

(a) The update rule is,

$$w_i^{(T+1)} = \frac{w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i))}{Z_T}$$

Summing over both sides, the equality holds.

$$\begin{aligned}
&\sum_{i=1}^{n} w_i^{(T+1)} = \sum_{i=1}^{n} \frac{w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i))}{Z_T} \\
&\implies \sum_{i=1}^{n} \frac{w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i))}{Z_T} = 1 \\
&\implies \frac{1}{Z_T} \sum_{i=1}^{n} w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i)) = 1 \\
&\implies \sum_{i=1}^{n} w_i^{(T)} \exp(-\beta_T \, y_i \, G_T(X_i)) = Z_T \\
&\implies \exp(-\beta_T) \sum_{y_i = G_T}^{n} w_i^{(T)} + \exp(\beta_T) \sum_{y_i \neq G_T}^{n} w_i^{(T)} = Z_T \\
&\implies \exp(-\beta_T) \sum_{i=1}^{n} w_i^{(T)} - \exp(-\beta_T) \sum_{y_i \neq G_T}^{n} w_i^{(T)} + \exp(\beta_T) \sum_{y_i \neq G_T}^{n} w_i^{(T)} = Z_T \\
&\implies \exp(-\beta_T) + (\exp(\beta_T) - \exp(-\beta_T)) \sum_{y_i \neq G_T}^{n} w_i^{(T)} = Z_T \\
&\implies \exp(-\beta_T) + (\exp(\beta_T) - \exp(-\beta_T)) \mathrm{err}_T = Z_T \\
&\implies \sqrt{\frac{\mathrm{err}_T}{1 - \mathrm{err}_T}} + \left(\sqrt{\frac{1 - \mathrm{err}_T}{\mathrm{err}_T}} - \sqrt{\frac{\mathrm{err}_T}{1 - \mathrm{err}_T}} \right) \mathrm{err}_T = Z_T \\
&\implies (1 - \mathrm{err}_T) \sqrt{\frac{\mathrm{err}_T}{1 - \mathrm{err}_T}} + \sqrt{\mathrm{err}_T} \sqrt{1 - \mathrm{err}_T} = Z_T \\
&\implies Z_T = 2\sqrt{(\mathrm{err}_T)(1 - \mathrm{err}_T)}
\end{aligned}$$

as expected. In the substitutions, I used the fact that,

$$\exp(-\beta_T) = \sqrt{\mathrm{err}_T} \sqrt{1 - \mathrm{err}_T}$$

(b) The recursive relation is,

$$w_i^{T+1} = \frac{1}{Z_T} w_i^T \exp(-\beta_T G_T(X_i) y_i)$$

We know that the base case for T = 2 is:

$$\begin{aligned} w_i^2 &= \frac{1}{Z_T} w_i^1 \exp(-\beta_1 G_1(X_i) y_i) \\ &= \frac{1}{Z_T} \frac{1}{n} \exp(-\beta_1 G_1(X_i) y_i) \end{aligned}$$

Let us assume the proposition is true for T = m. In this case,

$$w_i^{(m)} = \frac{1}{n \prod_{t=1}^{m-1} Z_t} e^{-y_i M(X_i)}$$

For the case T = m + 1, we can prove that the recursion simplifies once again to the equation above, proving the relation for all T by mathematical induction.

$$\begin{aligned} w_i^{(m+1)} &= \frac{1}{Z_m} w_i^m \exp(-\beta_m G_m(X_i) y_i) \\ &= \frac{1}{Z_m} \left(\frac{1}{n \prod_{t=1}^{m-1} Z_t} e^{-y_i M(X_i)} \right) \exp(-\beta_m G_m(X_i) y_i) \\ &= \left(\frac{1}{n \prod_{t=1}^{m} Z_t} e^{-y_i \sum_{t=1}^{m-1} \beta_t G_t(X_i)} \right) \exp(-\beta_m G_m(X_i) y_i) \\ &= \left(\frac{1}{n \prod_{t=1}^{m} Z_t} e^{-y_i \sum_{t=1}^{m-1} \beta_t G_t(X_i) - \beta_m G_m(X_i) y_i} \right) \\ &= \frac{1}{n \prod_{t=1}^{m} Z_t} e^{-y_i \sum_{t=1}^{m} \beta_t G_t(X_i)} \end{aligned}$$

Since the equation is true for n = m, this is true by mathematical induction.

(c)

$$\begin{aligned} \sum_{i=1}^{n} \exp(-y_i M(X_i)) &= \sum_{y_i M(X_i) < 0} \exp(-y_i M(X_i)) + \sum_{y_i M(X_i) \geq 0} \exp(-y_i M(X_i)) \\ &\geq \sum_{y_i M(X_i) < 0} \exp(-y_i M(X_i)) \\ &\geq \sum_{y_i M(X_i) < 0} 1 \\ &= B \end{aligned}$$

This uses the fact that $\exp(-y_i M(X_i)) \geq 0$ for all negative $y_i M(X_i)$ for the first inequality, and that $\exp(-y_i M(X_i)) \geq 1$ for all positive $y_i M(X_i)$ in the second inequality. Therefore, we have that,

$$\sum_{i=1}^{n} e^{-y_i M(X_i)} \geq B$$

(d) From (2), we can show that $Z_t \leq 0.9998$ for all $\text{err}_t \leq 0.49$. The base case is,

$$Z_T = 2\sqrt{(\text{err}_t)(1 - \text{err}_t)} \leq 2\sqrt{(0.49)(0.51)} \leq 0.9998 \, \forall \, t$$

This is true because $\sqrt{x(1-x)}$ is a strictly increasing function over the domain $x \in (0, 0.5)$. We can then apply a sum onto both sides of (3):

$$\begin{aligned}\sum_{i=1}^{n} w_i^{T+1} = 1 &= \sum_{i=1}^{n} \frac{e^{-M(X_i)y_i}}{n\prod_{i=1}^{T} Z_t} \\ &= n\prod_{i=1}^{T} Z_t \sum_{i=1}^{n} e^{-M(X_i)y_i} \\ &= \frac{1}{n\prod_{i=1}^{T} Z_t} \sum_{i=1}^{n} e^{-M(X_i)y_i} \\ &= \frac{1}{n\prod_{i=1}^{T} Z_t} \sum_{i=1}^{n} e^{-M(X_i)y_i}\end{aligned}$$

By (4),

$$\implies \sum_{i=1}^{n} e^{-M(X_i)y_i} = n\prod_{i=1}^{T} Z_t \geq n(\max_t Z_t)^T = n(0.9998)^T \geq B$$

As T tends to infinity, the sum above tends to zero. However,

$$\lim_{T\to\infty} \sum_{i=1}^{n} e^{-M(X_i)y_i} = \lim_{T\to\infty} n(0.9998)^T = 0 \geq B \implies B = 0.$$

Therefore, as T tends to infinity, the training accuracy will go to 100%.

(e) A decision tree compares features and chooses one of the best, if not the best feature for each split, in order to classify as many points as possible correctly. As the misclassified points have updated weights, the features chosen in the small trees are chosen to classify many different sets of weighted points, eventually classifying each point correctly (eventually, if misclassified repeatedly, one point will have such a high weight it will outweigh the main sample). In doing so, the decision trees optimize for the most important features for many different weight combinations.

Since each decision tree is combined in a linear combination, the best weights β_t effectively choose the best features and weight them higher. This shows that the decision trees are effectively performing subset selection.

5 Code Appendix

If not below, please go to the next page for the code appendix.

```python
import os
import scipy.io
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

# Load training data from MAT file
R = scipy.io.loadmat('movie_data/movie_train.mat')['train']

# Load validation data from CSV
val_data = np.loadtxt('movie_data/movie_validate.txt', dtype=int, delimiter=',')

# Helper method to get training accuracy
def get_train_acc(R, user_vecs, movie_vecs):
    num_correct, total = 0, 0
    for i in range(R.shape[0]):
        for j in range(R.shape[1]):
            if not np.isnan(R[i, j]):
                total += 1
                if np.dot(user_vecs[i], movie_vecs[j])*R[i, j] > 0:
                    num_correct += 1
    return num_correct/total

# Helper method to get validation accuracy
def get_val_acc(val_data, user_vecs, movie_vecs):
    num_correct = 0
    for val_pt in val_data:
        user_vec = user_vecs[val_pt[0]-1]
        movie_vec = movie_vecs[val_pt[1]-1]
        est_rating = np.dot(user_vec, movie_vec)
        if est_rating*val_pt[2] > 0:
            num_correct += 1
    return num_correct/val_data.shape[0]

# Helper method to get indices of all rated movies for each user,
# and indices of all users who have rated that title for each movie
def get_rated_idxs(R):
    user_rated_idxs, movie_rated_idxs = [], []
    for i in range(R.shape[0]):
        user_rated_idxs.append(np.argwhere(~np.isnan(R[i, :])).reshape(-1))
    for j in range(R.shape[1]):
        movie_rated_idxs.append(np.argwhere(~np.isnan(R[:, j])).reshape(-1))
    return np.array(user_rated_idxs), np.array(movie_rated_idxs)

# Part (c): SVD to learn low-dimensional vector representations
def svd_lfm(R):

    r = np.copy(R)

    # Fill in the missing values in R
    r[np.isnan(r)] = 0

    # Compute the SVD of R
    U, s, Vh = scipy.linalg.svd(r, full_matrices = False)

    # Construct user and movie representations
    user = U
    movie = np.diag(s) @ Vh


    user_vecs = user
    movie_vecs = movie.T
    return user_vecs, movie_vecs

# Part (d): Compute the training MSE loss of a given vectorization
# def get_train_mse_old(R, user_vecs, movie_vecs):
```

```
#     # Compute the training MSE loss
#     mse_loss = 0
#     count = 0
#     for i in range(user_vecs.shape[0]):
#         for j in range(movie_vecs.shape[0]):
#             if not np.isnan(R[i, j]):
#                 dots = user_vecs[i].dot(movie_vecs[j])
#                 mse_loss += (dots - R[i, j]) ** 2
#                 count += 1

#     return mse_loss / count

#Part (d): Compute the training MSE loss of a given vectorization
def get_train_mse(R, user_vecs, movie_vecs):

    UDVT = user_vecs @ movie_vecs.T

    r = np.copy(R)
    k = np.isnan(r)
    r[k] = 0
    UDVT[k] = 0
    mse_loss = np.linalg.norm(r - UDVT) ** 2

    return mse_loss

# Part (e): Compute training MSE and val acc of SVD LFM for various d
d_values = [2, 5, 10, 20]
train_mses, train_accs, val_accs = [], [], []
user_vecs, movie_vecs = svd_lfm(np.copy(R))
for d in d_values:
    train_mses.append(get_train_mse(np.copy(R), user_vecs[:, :d], movie_vecs[:, :d]))
    train_accs.append(get_train_acc(np.copy(R), user_vecs[:, :d], movie_vecs[:, :d]))
    val_accs.append(get_val_acc(val_data, user_vecs[:, :d], movie_vecs[:, :d]))
plt.clf()
plt.plot([str(d) for d in d_values], train_mses, 'o-')
plt.title('Train MSE of SVD-LFM with Varying Dimensionality')
plt.xlabel('d')
plt.ylabel('Train MSE')
plt.savefig(fname='train_mses.png', dpi=600, bbox_inches='tight')
plt.clf()
plt.plot([str(d) for d in d_values], train_accs, 'o-')
plt.plot([str(d) for d in d_values], val_accs, 'o-')
plt.title('Train/Val Accuracy of SVD-LFM with Varying Dimensionality')
plt.xlabel('d')
plt.ylabel('Train/Val Accuracy')
plt.legend(['Train Accuracy', 'Validation Accuracy'])
plt.savefig(fname='trval_accs.png', dpi=600, bbox_inches='tight')

print(train_mses)
print(train_accs)
print(val_accs)


# Part (f): Learn better user/movie vector representations by minimizing loss
best_d = 10 #d_values[np.argmax(val_accs)] #TODO(f): Use best from part (e)
np.random.seed(20)
user_vecs = np.random.random((R.shape[0], best_d))
movie_vecs = np.random.random((R.shape[1], best_d))
user_rated_idxs, movie_rated_idxs = get_rated_idxs(np.copy(R))

def get_A(user_vecs, movie_vecs, R):

    UDVT = user_vecs @ movie_vecs.T

    r = np.copy(R)
    k = np.isnan(r)
    r[k] = 0
```

```
    UDVT[k] = 0

    A = (r - UDVT)
    print(A.shape)
    return A


# Part (f): Function to update user vectors
# def update_user_vecs(user_vecs, movie_vecs, R, user_rated_idxs):

#     U = user_vecs
#     V = movie_vecs

#     R[np.isnan(R)] = 0
#     user_vecs_new = R @ V @ np.linalg.inv((V.T @ V) + np.eye(V.shape[1]))
#     print(np.mean(user_vecs_new), np.std(user_vecs_new))
#     return user_vecs_new

# # Part (f): Function to update user vectors
# def update_movie_vecs(user_vecs, movie_vecs, R, movie_rated_idxs):

#     # Update movie_vecs to the loss-minimizing value
#     U = user_vecs
#     V = movie_vecs

#     R[np.isnan(R)] = 0
#     movie_vecs_new = R.T @ U / 2

#     return movie_vecs_new

def update_user_vecs(user_vecs, movie_vecs, R, user_rated_idxs):
    user_vecs_new = np.zeros(user_vecs.shape)
    y = movie_vecs
    x = user_vecs


    for i in range(x.shape[0]):
        l = y.shape[1]
        youter = np.eye(l)
        Ry = np.zeros(y.shape[1])


        for j in user_rated_idxs[i]:
            youter += np.outer(y[j], y[j])
            Ry += R[i, j] * y[j].T

        user_vecs_new[i] = np.linalg.inv(youter) @ (Ry)


    return user_vecs_new

def update_movie_vecs(user_vecs, movie_vecs, R, movie_rated_idxs):

    movie_vecs_new = np.zeros(movie_vecs.shape)
    y = movie_vecs
    x = user_vecs


    for j in range(y.shape[0]):
        l = x.shape[1]
        xouter = np.eye(l)
        Rx = np.zeros(x.shape[1])


        for i in movie_rated_idxs[j]:
            xouter += np.outer(x[i], x[i])
            Rx += R[i, j] * x[i].T
```

```
        movie_vecs_new[j] = np.linalg.inv(xouter) @ (Rx)



    return movie_vecs_new

# Part (f): Perform loss optimization using alternating updates
train_mse = get_train_mse(np.copy(R), user_vecs, movie_vecs)
train_acc = get_train_acc(np.copy(R), user_vecs, movie_vecs)
val_acc = get_val_acc(val_data, user_vecs, movie_vecs)
print(f'Start optim, train MSE: {train_mse:.2f}, train accuracy: {train_acc:.4f}, val accuracy: {val_acc:.4f}')
for opt_iter in range(20):
    user_vecs = update_user_vecs(user_vecs, movie_vecs, np.copy(R), user_rated_idxs)
    movie_vecs = update_movie_vecs(user_vecs, movie_vecs, np.copy(R), movie_rated_idxs)
    train_mse = get_train_mse(np.copy(R), user_vecs, movie_vecs)
    train_acc = get_train_acc(np.copy(R), user_vecs, movie_vecs)
    val_acc = get_val_acc(val_data, user_vecs, movie_vecs)
    print(f'Iteration {opt_iter+1}, train MSE: {train_mse:.2f}, train accuracy: {train_acc:.4f}, val accuracy: {val_acc:.4f}')
```