Song Recommender System with SVD

(i)

To show that matrices AA^T and A^TA are symmetric we need to prove that they are equal to their own transposes. To do this, we used Sage to check that AA^T and A^TA are equal to $(AA^T)^T$ and $(A^TA)^T$ respectively.

```
In [1]:  A = matrix([[2,0,1],[3,1,-6]])

In [10]: A.transpose()*A, (A.transpose()*A).transpose()
Out[10]: (
         [ 13   3 -16]  [ 13   3 -16]
         [  3   1  -6]  [  3   1  -6]
         [-16  -6  37], [-16  -6  37]
         )

In [11]: assert (A.transpose()*A).transpose() == A.transpose()*A

In [12]: A*A.transpose(), (A*A.transpose()).transpose()
Out[12]: (
         [ 5  0]  [ 5  0]
         [ 0 46], [ 0 46]
         )

In [13]: assert A*A.transpose()==(A*A.transpose()).transpose()
```

Fig. 1 shows Sage code checking for symmetry.

(ii)

Let M be an $m \times n$ matrix. To show that MM^T and M^TM are both symmetric matrices we can use the the rule stating that the transpose of a product is the product of the transposes in reverse order such that

$$(AB)^T = B^TA^T$$

First we need to show that $M^T M$ equals its own transpose

$$(M^T M)^T = M^T \times (M^T)^T = M^T M$$

Now we need to show that MM^T equals its own transpose

$$(MM^T)^T = (M^T)^T \times M^T = MM^T$$

(iii)

To find orthogonal matrices U and V, such that $AA^T = UD_1U^T$ and $A^TA = VD_2V^T$ we can find the eigenvectors of AA^T and A^TA and create the matrices by using them as columns. Next, to find D_1 and D_2, we can take the eigenvalues of AA^T and A^TA and place them on the main diagonal.

```
In [44]:
1 AAT = A*A.transpose()
2 print(AAT.eigenvectors_right())
3
4 U = matrix([[0,1],[1,0]])
5 D_1 = matrix([[46,0],[0,5]])
6
7 assert AAT == U*D_1*U.transpose()

[(46, [
(0, 1)
], 1), (5, [
(1, 0)
], 1)]
```

Fig. 2 shows code finding matrix U.

We can then include an assert statement to check that the equality is satisfied

In [43]:

```
ATA = A.transpose()*A
print(ATA.eigenvectors_right())

col1 = vector([1,1/3,-2])
col2 = vector([1,0,1/2])
col3 = vector([1,-15,-2])

V = matrix([col1/col1.norm(),col2/col2.norm(),col3/col3.norm()]).
D_2 = matrix([[46,0,0],[0,5,0],[0,0,0]])

assert ATA == V*D_2*V.transpose()
```

```
[(46, [
(1, 1/3, -2)
], 1), (5, [
(1, 0, 1/2)
], 1), (0, [
(1, -15, -2)
], 1)]
```

Fig. 3 shows code finding matrix V.

As we can see, matrices D_1 and D_2 have the same non-zero eigenvalues. The only difference is that D_1 has another zero eigenvalue as the last main diagonal entry.

(iv)

Let M be an $m \times n$ matrix, $M^T M$ a symmetric matrix with eigenvalue λ, and x the corresponding eigenvector.

$$M^T M x = \lambda x$$

We can multiply by M both sides:

$$M M^T M x = \lambda M x$$

We can rewrite then as

$$(M M^T)(M x) = \lambda (M x)$$

Let vector y equal Mx

y is a non-zero vector, since λ is a non-zero eigenvalue $M^T M$

The expression becomes

$$(MM^T)y = \lambda y$$

We can now see that λ is also an eigenvalue for MM^T

To prove that the eigenvalues are also positive we can use Euclidian norms:

$$||x|| = \sqrt{x^T x}$$

Thus

$$\lambda||x||^2 = \lambda x^T x$$

or $\lambda||x||^2 = x^T \lambda x$

Since x is an eigenvector of $M^T M$

$$M^T Mx = \lambda x$$

We can multiply both sides by x^T

$$x^T(M^T Mx) = x^T \lambda x$$

We have, thus, found that $\lambda||x||^2 = x^T(M^T Mx)$

Using the associative property of matrix multiplication we can rewrite the expression as

$$\lambda||x||^2 = (x^T M^T)(Mx)$$

Using the transpose of a product rule we have that

$$\lambda||x||^2 = (Mx)^T(Mx)$$

and since $Mx = y$, and using the definition of norm we have

$$\lambda||x||^2 = ||y||^2$$

Since λ is a non-zero eigenvalue and y a non-zero vector, $||y||^2$ is positive.

We are then left with λ multiplied by $||x||^2$ equaling a positive value. Because x is itself a non-zero eigenvector, $||x||^2$ is positive as well, which means λ cannot be negative or zero and is, therefore, positive.

(v)

We can find Σ by simply multiplying our previously found U^T, A, and V^T

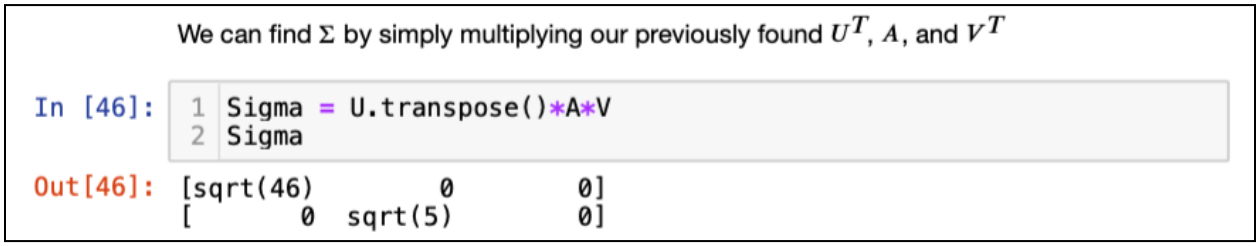

Fig. 4 shows code calculating matrix Σ.

As we can see above, Σ is a matrix with same dimensions as A with A's singular values on its main diagonal. These singular values, as observed, are square roots of the eigenvalues of AA^T and A^TA (previously found to be the same for the two matrices) arranged in descending order.

Σ therefore can be obtained by arranging in descending order on a diagonal the square roots of the diagonal entries of either D_1 or D_2

(b)

Following the process above we can find the SVDs of B, C, and D

```
In [124]:
B = matrix([[1,-4,2]])
BBT = B*B.transpose()
print(BBT.eigenvectors_right())
U = matrix([[1]])

[(21, [
(1)
], 1)]

In [125]:
BTB = B.transpose()*B
print(BTB.eigenvectors_right())
col1 = vector([1,-4,2])
col2 = vector([1,0,-1/2])
col3 = vector([0,1,2])
V = matrix([col1/col1.norm(),col2/col2.norm(),col3/col3.norm()]).t

[(21, [
(1, -4, 2)
], 1), (0, [
(1, 0, -1/2),
(0, 1, 2)
], 2)]

In [126]:
Sigma = U.transpose()*B*V
print(Sigma)

assert B == U*Sigma*V.transpose()

[sqrt(21)        0        0]
```

Fig. 5 shows code performing SVD to matrix B.

```
In [127]:
C = matrix([[0,2],[-1,0]])
CCT = C*C.transpose()
print(CCT.eigenvectors_right())
U = matrix([[1,0],[0,1]])

[(4, [
(1, 0)
], 1), (1, [
(0, 1)
], 1)]

In [128]:
CTC = C.transpose()*C
print(CTC.eigenvectors_right())
V = matrix([[0,1],[1,0]])

[(4, [
(0, 1)
], 1), (1, [
(1, 0)
], 1)]

In [129]:
Sigma = U.transpose()*C*V
print(Sigma)

assert C == U*Sigma*V.transpose()

[ 2  0]
[ 0 -1]
```

Fig. 6 shows the code performing SVD to matrix C.

```
In [130]:
D = matrix([[-4,1],[-2,-4],[0,-2],[2,-2]])
DDT = D*D.transpose()
print(DDT.eigenvectors_right())
col1 = vector([1,-4,-2,-2])/vector([1,-4,-2,-2]).norm()
col2 = vector([1,1/2,0,-1/2])/vector([1,1/2,0,-1/2]).norm()
col3 = vector([1,0,-3/2,2])/vector([1,0,-3/2,2]).norm()
col4 = vector([0,1,-3,1])/vector([0,1,-3,1]).norm()
U = matrix([col1,col2,col3,col4]).transpose()

[(25, [
(1, -4, -2, -2)
], 1), (24, [
(1, 1/2, 0, -1/2)
], 1), (0, [
(1, 0, -3/2, 2),
(0, 1, -3, 1)
], 2)]

In [131]:
DTD = D.transpose()*D
print(DTD.eigenvectors_right())
V = matrix([[0,1],[1,0]])

[(25, [
(0, 1)
], 1), (24, [
(1, 0)
], 1)]

In [132]:
Sigma = U.transpose()*D*V
print(Sigma)

assert D == U*Sigma*V.transpose()

[           5            0]
[           0 -4*sqrt(3/2)]
[           0            0]
[           0            0]
```

Fig. 7 shows the code performing SVD to matrix D.

(c)

To set up this part, we first chose a selection of six significantly different music genres. Having a broader range of genre options would hopefully allow us to map preference patterns more accurately without bias from related or similar styles of music. We then selected what were according to various sources some of the most notorious songs from each genre. The choices were then assigned an ID and put into a list as can be seen below.

Song 0 - Pop - Billie Jean by Michael Jackson
Song 1 - Hip Hop - Lose Yourself by Eminem
Song 2 - Rock - Stairway To Heaven by Led Zeppelin
Song 3 - Electronic/Dance - Around The World by Daft Punk
Song 4 - Folklore - The Times They Are A-Changin by Bob Dylan
Song 5 - Classical - Für Elise by Ludwig van Beethoven

We then asked five of our classmates who were willing to participate in our data collection to meet us, one by one, and let them listen to our songs one at a time. We then asked them to rate the song. For simplicity, a binary rating system was chosen where participants would score with 1 song they liked and 0 songs they did not.

The following results were produced, which provided the data for our matrix entries in order for us to do SVD

	Song 0	Song 1	Song 2	Song 3	Song 4	Song 5
Bruno	1	0	1	1	0	1
Alan	1	1	0	1	0	1
Marta	1	0	1	1	1	1
Pelle	1	1	1	1	1	1
Humberto	0	1	1	0	0	0

Fig. 8 shows a table with the results of our data collection. Columns represent our songs from the list with their respective id, rows represent the preferences of our classmates.

With the data collected, we could then proceed with our calculations.

(i)

In [302]:
```
# import packages that will ease calculations
# given our rather large sized matrix
import numpy as np
from scipy.linalg import svd

# define the collected data as an array
data = np.array([[1, 0, 1, 1, 0, 1],
       [1, 1, 0, 1, 0, 1],
       [1, 0, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1],
       [0, 1, 1, 0, 0, 0]])
# define a function that will perform SVD
def singular_value_decomposition(data, k=0, option=False):
    U, Sigma, VT = svd(data)
    U = U[:,:k]
    VT = VT[:k,:]
    return U, Sigma, VT

U, Sigma, VT = singular_value_decomposition(data, 3)
U, Sigma, VT
```

Out[302]:
```
(array([[ 0.44721333,  0.30827579, -0.05847405],
        [ 0.41509776, -0.10565717,  0.85168735],
        [ 0.51171556,  0.37427509, -0.42089303],
        [ 0.58053707, -0.24859972, -0.10912218],
        [ 0.16975859, -0.83181665, -0.2866197 ]]),
 array([4.11504122e+00, 1.37992482e+00, 1.20931294e+00, 8.36543739e-01,
        8.35608707e-17]),
 array([[ 0.47498035,  0.28320334,  0.41536025,  0.47498035,  0.26542933,
          0.47498035],
        [ 0.23790715, -0.85952041, -0.28832404,  0.23790715,  0.09107407,
          0.23790715],
        [ 0.21764267,  0.37702852, -0.72364144,  0.21764267, -0.43827795,
          0.21764267]]))
```

Fig. 9 shows code using NumPy performing SVD to our matrix.

From the SVD results, we can infer that the most important latent factors reflect features or patterns of the sampled group's preferences in music. These are hidden, not quite straightforward variables that influence the ratings the songs get. In our case, we chose the first three factors as the most important, in decreasing order of importance. These factors indicate dominant patterns that can explain parts of the data's variability. Based on these, we can speculate what these factors are.

The first one, for instance, could reflect a preference for popular music as scores high for the items corresponding to pop, hip-hop/rap, and classical music.

The second could be related more to a preference for urban, rhythmic music associated with street culture, as it has high loadings on the items corresponding to hip-hop/rap and eletronic/dance music.

The third could reflect a preference for live, more traditional music based on the high scores on the items corresponding to folklore and classical music.

(ii)

All of the consumers in the dataset's pairwise similarities are represented by the product MM^T. This is due to the fact that each MM^T member (i,j) is the dot product of the preference vectors of consumers i and j. When two vectors are compared, the dot product calculates their cosine similarity because the dot product is defined as the product of the magnitudes of the two vectors and the cosine of the angle between them, which shows how similar the directions of the compared vectors are.

The more similar a pair of customers' tastes are, the higher the value of MM^T(i,j). For instance, the dot product of customers 1 and 2's preference vectors is 4, which is greater than the dot product of customers 1 and 5's preference

vectors, which is 2. This shows that consumers 1 and 2 have more similar music preferences than customers 1 and 5.
These measures of similarity between customers' preferences can then be used for clustering or recommendation purposes.

(iii)

In SVD, matrix V represents the latent features underlying the objects being assessed, while matrix U reflects the latent features underlying customers' preferences. V compares products based on user preferences for these features, whereas U compares users based on their preferences for these features. By analyzing U and V, we may gain insights that we can use to learn more about user preferences in various settings, including music reviews or product recommendations.

Using the above explanation, we can create a code that would use dot products of the vectors in U to find how similar the musical preferences of our consumers are or V to find how “similarly likable” a song is given another liked song. For our simulation, we chose the latter approach, as can be observed below.

```
In [303]:
# define function that will automatically recommend songs
def recommend_songs(liked_song, VT, number_of_recs=3):
    # initialize a list where we will store our recommendations
    recs = []
    # compare the liked song's dot products with all other elements
    for ele in range(len(VT[0])):
        # bypass case in which liked song is dotted with itself
        if ele != liked_song:
            recs.append([ele,np.dot(VT[:,ele],VT[:,liked_song])])
    # code will create tuples with dot product value and index
    # compare based on dot product value, id is key
    # add tuples to final recommendations list
    recommendations = [i[0] for i in sorted(recs, key=lambda x: x[1],reverse=True)]

    # return as many recommendations as specified in input sorted from likeliest
    # to least likely to be liked based on the initial song
    return recommendations[:number_of_recs]

In [304]:
recommend_songs(1,VT)

Out[304]:
[2, 0, 3]
```

Fig. 10 shows code giving recommendations of songs based on a song the user already likes. The code was adapted from Miller (2017) for our problem.

In the figure above, we can see that given that we liked song number 1, *Lose Yourself* by Eminem, the algorithm will recommend us 2, *Stairway To Heaven* by Led Zeppelin, 0, *Billie Jean* by Michael Jackson, and 3, *Around The World* by Daft Punk (which may not be surprising as these are more "modern" choices compared to the folklore and classical options, perhaps for a consumer with more modern music preferences).

AI Policy

For this assignment, we did not resort to the use of AI for uses other than minor, explanations, LaTeX translations, etc.

References

Miller, Z. W. (2017, November 17). *Recommendation Engines for Dummies*. ZWMiller. Retrieved April 19, 2023, from http://zwmiller.com/projects/simple_recommender.html