Data Science - Scientific Computing in Python

Free eBook: https://pythonnumericalmethods.berkeley.edu/notebooks/Index.html (https://pythonnumericalmethods.berkeley.edu/notebooks/Index.html)

Citation:

(1) School of Computing - Statistical Data Analytics, University of California at Berkeley, Berkeley, CA 94720-1786

(2) Brad Nelson, Computational Topology, University of Chicago, Chicago

Topics:

```
1. Matrix Fundamentals
2. Distance Metrics
3. Sparse Matrix Operations
4. Week 3 Assignment Hints and Tricks
```

1. Matrix Fundamentals

All about Matrices:

Matrix Operations:

### Identity Matrix:
A special case of Scalar Square Matrices where all elements are zero except diagonals which are ones. Identity (I) is algebraic equivalent to '1' in arithmetics. Plays a central role in Linear Alg

```
I = [1 0 0]
    [0 1 0]                    - All diagonal elements are ones and others are zeros.
    [0 0 1]
```

### Inner Product (dot product):
It is the multiplication of each row vs column one-by one, in a vector sense, its cosine (dot) product.

A . B = |A||B|cos(A,B)

Example:
Vector A = (1, 10, -2) and Vector B = (2, 1, -4)
A.B = 1*2 + 10*1 + -2*-4 = 20

### Transpose:
Rotation of a matrix by clockwise 90.

```
A = [1 2 3]        => A' = [1 4]
    [4 1 2]                [2 1]
                           [3 2]
```

- Orthogonal Vectors: When two vectors are perpendicular to each other, i.e. angle between two vectors is 90 degrees, it's an Orthogonal Vector. E.g. Dot Product of two vectors, A . B = 0

- Orthogonal Matrices: $If\ A.A^T = A^T.A = I \quad => A^T = A^{-1}\ then\ A\ is\ an\ Orthogonal\ Matrix$

### Vectorization:

It is possible to vectorize a matrix, by stacking each element on top of each other column wise.

```
A = [a11 a12]           =>  vec(A)  = [a11]
    [a21 a22]                         [a21]
                                      [a12]
                                      [a22]
```

- *vec(AB) = vec(A) + vec(B)*

### Trace of a Matrix:
Sum of all diagonal elements

```
A = [1 2 3]            Trace(A) = 1 + 5 + 6 = 12
    [4 5 6]
```

### Inverse of a Matrix:

Addition, Subtraction and Multiplication is simple in matrices, but division is not. To use division, we utilize the concept of Inverse. Like in scalar, a x (b^-1) is another way to say a/b. In matrix algebra if a matrix division is not possible and the inverse is not defined we say that a matrix is `singular`

Inverse of a matrix only exists if A x (A^-1) = I i.e. equals to an Identity Matrix. If not, then we say the Inverse of that matrix does not exist ! *(TRIVIA: As per quantum physics, we really don't know if something exists or not for sure, it could exist and NOT exist at the same moment, and it is only revealed when we actually look at it, kind of saying moon does not exist if we don't look at it, but applied to sub-atomic particles. So in Statistics and Physics from the 21st Century onwards, we NEVER SAY it "does not" exist, but indeed it "MIGHT" not exist. Read more - John Bell Quantum Proof)*

Inverse of A = $\frac{1}{|A|}$ adj(A) |A| is the determinant of the matrix, adj(B) is the so-called adjoint matrix.

Determinant of the Matrix (|A|) - Is just the cofactor expansion, along any row or column you choose!

A = [a11 a12 a13]
[a21 a22 a23]
[a31 a32 a33]

Cofactor $C_{ij} = (-1)^{i+j} . Minor\ Matrix$ (i.e. with ith row and jth col dropped)

|A| = a11.C11 + a12.C12 + a13.C13
= a11.C11 + a21.C21 + a31.C31 = OR along any row or column.

$$= a_{11} . (-1)^{1+1} \begin{matrix}[a22\ a23]\\ [a32\ a33]\end{matrix} + a_{12} . (-1)^{1+2} \begin{matrix}[a21\ a23]\\ [a31\ a33]\end{matrix} + a_{13} . (-1)^{1+3} \begin{matrix}[a21\ a22]\\ [a31\ a32]\end{matrix}$$

$$= a11. \begin{matrix}[a22\ a23]\\ [a32\ a33]\end{matrix} - a12. \begin{matrix}[a21\ a23]\\ [a31\ a33]\end{matrix} + a13. \begin{matrix}[a21\ a22]\\ [a31\ a32]\end{matrix}$$

Example: Cofactor of 2x2 Matrix:- $\begin{matrix}[1\ \ 3]\\ [4\ -1]\end{matrix}$ $=> C = 1 * -1 - 3 * 4 = -13$

Adjoint of the Matrix - Is the ***Transpose*** of the Cofactor Matrix, each element is substituted by its Cofactor equivalent and then the final matrix is transposed.

Adjoint of matrix A $= \begin{matrix}[a11\ a12\ a13]\\ [a21\ a22\ a23]\\ [a31\ a32\ a33]\end{matrix}$ $= \begin{matrix}[C_{11}\ C_{21}\ C_{31}]\\ [C_{12}\ C_{22}\ C_{32}]\\ [C_{13}\ C_{23}\ C_{33}]\end{matrix}$

PROPERTIES OF MATRICES

- $|A^T| = |A|$
- |AB| = |A| |B|

- $|AB| = |A|\,|B|$
- vec(AB) = vec(A) + vec(B)
- If A is a *m x m* matrix, and *k* is a scalar, then, $|kB| = k^m |B|$
- Determinant of the identity matrix is one: | I | = 1
- Inverse of Identity is Identity itself.
- $(A^T)^{-1} = (A^{-1})^T$
- $(AB)^{-1} = (A^{-1})(B^{-1})$

### Reduced Row Echelon Form of a Matrix (RREF):

We have seen that every linear system of equations can be written in matrix form. For example, the system

$$\begin{aligned} x + 2y + 3z &= 4 \\ 3x + 4y + z &= 5 \\ 2x + y + 3z &= 6 \end{aligned}$$

can be written as

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 1 \\ 2 & 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}.$$

The matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 1 \\ 2 & 1 & 3 \end{bmatrix}$$

is called the **matrix of coefficients** of the system. It is also useful to form the **augmented matrix**

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 5 \\ 2 & 1 & 3 & 6 \end{bmatrix}.$$

E.g. 3 linear equations with 3 unknowns, will yield a 3 x 4 Augmented matrix.

Definition 1. *A matrix is in* **row echelon form** *if*

1. Nonzero rows appear above the zero rows.

2. In any nonzero row, the first nonzero entry is a one (called the **leading one***).*

3. The leading one in a nonzero row appears to the left of the leading one in any lower row.

Definition 2. *A matrix is in* **reduced row echelon form (RREF)** *if the three conditions in Definition 1 hold and in addition, we have*

4. If a column contains a leading one, then all the other entries in that column are zero.

Any matrix can be transformed into its **RREF** by performing a series of operations on the rows of the matrix. The allowable operations are called elementary row operations. They are:

1. Divide a row by a nonzero number.
2. Subtract a multiple of a row from another row.
3. Interchange two rows. Performing any of these operations on an augmented matrix leads to a new system of equations which has the same set of solutions as the original system.

We could use Gaussian elimination for reducing a matrix into it's REF format and use Gauss-Jordan method to compute its Inverse.

Rank of a Matrix
For a *m x m* square matrix, if the Determinant of the matrix is *not* zero, then the rank is the highest order of rows or columns in the matrix, i.e. *m*. If the Determinant was in fact zero, then we consider any Minor Matrix with *m-1 x m-1* shape and compute its Determinant. If it is non-zero, the rank would be *m-1*, but if it's again zero, then we compute another Minor Matrix with *m-2 x m-2* shape, and so on.

Rank(Matrix) = Non zero rows in RREF format.

Nullity
To determine the null-space (or nullity) of a matrix can be determined using Rank and RREF format. For a m x n matrix,

Null(Matrix) = Number of columns - Rank(Matrix)

2. Distance Metrics

Vector Norm:

The **norm** of a vector is a measure of its length. There are many ways of defining the length of a vector depending on the metric used (i.e., the distance formula chosen). The most common is called the L_2 norm, which is computed according to the distance formula you are probably familiar with from grade school. The L_2 **norm** of a vector v is denoted by $\|v\|_2$ and $\|v\|_2 = \sqrt{\Sigma_i v_i^2}$. This is sometimes also called Euclidian length and refers to the "physical" length of a vector in one-, two-, or three-dimensional space. The L_1 norm, or "Manhattan Distance," is computed as $\|v\|_1 = \Sigma_i |v_i|$, and is named after the grid-like road structure in New York City. In general, the **p-norm,** L_p, of a vector is $\|v\|_p = \sqrt[p]{(\Sigma_i v_i^p)}$. The L_∞ **norm** is the p-norm, where $p = \infty$. The L_∞ norm is written as $||v||_\infty$ and it is equal to the maximum absolute value in v.

Types of Distance Measures:

Given: Point A (100, 200); Point B (140, 50)

1. Minkowski Distance ($p\ norm$)

p-norm for $1 <= p <= \infty$: $||x||_p = (|x_1|^p + |x_2|^p + |x_3|^p + ...)^{1/p}$

Minkowski distance calculates the distance between two real-valued vectors. Controlled by p-norm it generalizes various distance metrics. Minkowski(p=1) is the same as Manhattan.

2. Manhattan Distance ($l_1\ norm$ / Taxicab/Rectilinear/Snake):

$$||x||_1 = |x_1| + |x_2| + ... + |x_n|$$

manhattan(A,B) = |A| - |B| = |100 - 140| + |200 - 50| = 190

3. Euclidean Norm ($l_2\ norm$):

$$||x||_1 = \sqrt{|x_1| + |x_2| + ... + |x_n|} = \sqrt{\hat{x} . \hat{x}}$$

euclidean(A,B) = $\sqrt{(100 - 140)^2 + (200 - 50)^2}$

4. Chebyshev Distance ($l_\infty\ norm$ / Chessboard Distance):

$$chebyshev\ distance = max(|x_1|, |x_2|, ..., |x_n|)$$

chebyshev(A,B) = max(A, B) = max(|100 - 140|, |200 - 50|) = max(40, 150) = 150

5. Hamming Distance:

Hamming distance calculates the distance between two binary vectors, also referred to as binary strings or bit strings. You are most likely going to encounter bit strings when you one-hot encode categorical of bit strings. You are most likely going to encounter bit strings when you one-hot encode categorical columns of data.

Labels: Red, Blue, Green, Yellow, White, Black

$\widehat{y_1} = [0, 0, 0, 0, 0, 1]$ Hamming_dist (y1, y2) = 1 difference = ⅙

$\widehat{y_2} = [0, 0, 0, 0, 1, 1]$

$\widehat{y_3} = [0, 0, 0, 0, 1, 0]$ Hamming_dist (y1, y3) = 2 difference = 2/6

$\widehat{y_4} = [0, 1, 0, 0, 0, 0]$ Hamming_dist (y2, y3) = 1 difference = 1/6

6. Frobenius Distance:

The Frobenius norm, sometimes also called the Euclidean norm (a term unfortunately also used for the vector L2-norm), is the matrix norm of an m×n matrix A defined as the square root of the sum of the absolute squares of its elements.

A = [1, 2] => $||A||_F = \sqrt{1^2 + 2^2 + -1^2 + 2^2} = \sqrt{10}$

[-1 2]

3. Numpy Basics

Importing packages

In [1]:
```
import numpy as np   # can be anything, 'import numpy as numpaaa'
- you get it 'np' is just an industry acronym.
```

Array 1

In [2]:
```
a = np.array(
    [
        [10,4,2],
        [1,6,3]
    ])

a
```

Out[2]:
```
array([[10,  4,  2],
       [ 1,  6,  3]])
```

Array 2

```
In [3]: b = np.array([[1,2,3], [4,5,1]])
        b
```

```
Out[3]: array([[1, 2, 3],
               [4, 5, 1]])
```

Dimensions of a: 2 Rows x 3 Columns = 2 x 3

Dimensions of b: 2 x 3

1. min()

```
In [4]: a.min()
```

```
Out[4]: 1
```

```
In [5]: store = a[0][0]

        for i in a:
            for j in i:
                if j < store:
                    store = j
        store
```

```
Out[5]: 1
```

2. max()

```
In [6]: a.max()
```

```
Out[6]: 10
```

```
In [7]: store = a[0][0]

        for i in a:
            for j in i:
                if j>store:
                    store =j
        store
```

```
Out[7]: 10
```

3. mean()

In [8]:
```
a.mean()
```
Out[8]:
```
4.333333333333333
```

In [9]:
```
sum_, count = 0.0, 0

for row in a:
    for col_value in row:
        sum_ += col_value
        count+=1

sum_/count
```
Out[9]:
```
4.333333333333333
```

4. argmin()

- axis = 0 : Along the columns
- axis = 1 : Along the rows

In [10]:
```
print(a)
np.argmin(a, axis = 1)
```
```
[[10  4  2]
 [ 1  6  3]]
```
Out[10]:
```
array([2, 0])
```

In [11]:
```
# Calculating for: argmin(a, axis = 1)
min_list = []

for row in a:
    store = row[0]
    min_index = index = 0

    for j in row:
        if j < store:
            store = j
            min_index = index
        index +=1
    min_list.append(min_index)

min_list
```
Out[11]:
```
[2, 0]
```

In [12]:
```
print(a)
np.argmin(a, axis = 0)
```

```
[[10  4  2]
 [ 1  6  3]]
```

Out[12]:
```
array([1, 0, 0])
```

In [13]:
```
min_list = []

for col in range(len(a[0])):

    store = a[0][col]
    min_index = index = 0

    for row in range(len(a)):
        val = a[row][col]
        if val < store:
            store = val
            min_index = index

        index+=1
    min_list.append(min_index)


min_list
```

Out[13]:
```
[1, 0, 0]
```

5. argmax()

In [14]:
```
print(a)
np.argmax(a, axis=1)
```

```
[[10  4  2]
 [ 1  6  3]]
```

Out[14]:
```
array([0, 1])
```

In [15]:
```
max_list = []

for i in a:
    store = i[0]
    index = 0

    for j in i:
        if j >= store:
            store = j
            max_index = index
        index +=1
    max_list.append(max_index)
max_list
```

Out[15]:
```
[0, 1]
```

In [16]:
```
a
```

Out[16]:
```
array([[10,  4,  2],
       [ 1,  6,  3]])
```

In [17]:
```
"""
Intuiton:

Some neural network classifier's probabilistic output values for th
ree labels:

                Label1     Label2     Label3
Output_Unit_1     0.98        0.02      0
Output_Unit_2     0.33        0.33      0.33
Output_Unit_3     0.25        0.50      0.25
Output_Unit_4     0.05        0.45      0.499

How would you extract the 'softmax' output for most probable predic
ted label? - Use argmax.
"""

arr = np.array([
    [0.98, 0.02, 0],
    [0.33, 0.33, 0.33],
    [0.25, 0.50, 0.25],
    [0.05, 0.45, 0.499]
])
```

In [18]:
```
class_labels = {0: "Label1", 1: "Label2", 2: "Label3"}
list(map(lambda x: class_labels[x], np.argmax(arr, axis=1)))
```

Out[18]:
```
['Label1', 'Label1', 'Label2', 'Label3']
```

6. linalg.solve

```
In [19]: A = np.array([[1,2],[3,4]])
         B = np.array([[10],[20]])

         np.linalg.solve(A,B)

Out[19]: array([[0.],
                [5.]])
```

7. flatten

```
In [20]: a.flatten()

Out[20]: array([10,  4,  2,  1,  6,  3])
```

8. ravel()

```
In [21]: a.ravel()

Out[21]: array([10,  4,  2,  1,  6,  3])
```

Difference between the Numpy flatten and ravel functions?

- The main functional distinction is that flatten is a function of an ndarray object and hence only works with genuine numpy arrays. ravel(), on the other hand, is a library-level function that may be invoked on any object that can be correctly parsed.

9. dot Product

```
In [22]: print(a)
         print(a.shape)
         print("-")
         print(b.T)
         print(b.T.shape)

         [[10  4  2]
          [ 1  6  3]]
         (2, 3)
         -
         [[1 4]
          [2 5]
          [3 1]]
         (3, 2)
```

```
In [23]: a.dot(b.T)

Out[23]: array([[24, 62],
                [22, 37]])

In [24]: np.dot(a, b.T)

Out[24]: array([[24, 62],
                [22, 37]])
```

Item to Item Similarity: The very first step is to build the model by finding similarity between all the item pairs. The similarity between item pairs can be found in different ways. One of the most common methods is to use cosine similarity.

Formula for Cosine Similarity:

$$Similarity(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| * ||\vec{B}||}$$

Prediction Computation: The second stage involves executing a recommendation system. It uses the items (already rated by the user) that are most similar to the missing item to generate rating. We hence try to generate predictions based on the ratings of similar products. We compute this using a formula which computes rating for a particular item using weighted sum of the ratings of the other similar products.

$$rating(U, I_i) = \frac{\sum_j rating(U,I_j) * s_{ij}}{\sum_j s_{ij}}$$

EXTRA READ: https://www.geeksforgeeks.org/item-to-item-based-collaborative-filtering/ (https://www.geeksforgeeks.org/item-to-item-based-collaborative-filtering/)

10. * for

- multiplies each element with corresponding positioned element in another same shape matrix

In [25]:
```
print(a)
print(a.shape)
print("-")
print(b.T)
print(b.T.shape)
```

```
[[10  4  2]
 [ 1  6  3]]
(2, 3)
-
[[1 4]
 [2 5]
 [3 1]]
(3, 2)
```

In [26]:
```
np.multiply(a,10)
```

Out[26]:
```
array([[100,  40,  20],
       [ 10,  60,  30]])
```

In [27]:
```
np.multiply(a,b)
```

Out[27]:
```
array([[10,  8,  6],
       [ 4, 30,  3]])
```

In [28]:
```
result_array = ([[0,0,0],[0,0,0]])
a_rows,a_cols = a.shape

for i in range(a_rows):
    for j in range(a_cols):
        result_array[i][j] = a[i][j]*b[i][j]

result_array
```

Out[28]:
```
[[10, 8, 6], [4, 30, 3]]
```

11. Reshape

In [29]:
```
k = np.arange(4)
k
```

Out[29]:
```
array([0, 1, 2, 3])
```

In [30]:
```
k.reshape(2,2)
```

Out[30]:
```
array([[0, 1],
       [2, 3]])
```

12. identity / eye

```
In [31]: i_m = np.identity(3)
         i_m

Out[31]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

```
In [32]: b = np.empty(i_m.size)


         s = 0
         i_m_rows, i_m_cols = i_m.shape
         for i in range(i_m_rows):
             for j in range(i_m_cols):
                 if i == j:
                     b[s] = 1
                     s += 1
                 else:
                     b[s] = 0
                     s += 1
         print("without using inbuilt api diagonal array is\n",b.reshape(3,
         3))

         without using inbuilt api diagonal array is
          [[1. 0. 0.]
          [0. 1. 0.]
          [0. 0. 1.]]
```

13. sum()

```
In [33]: a.sum()

Out[33]: 26
```

```
In [34]: sum = 0

         for i in range(a.size):
             sum += a.item(i)
         sum

Out[34]: 26
```

14. diag()

```
In [35]: k = np.arange(9).reshape(3,3)
         k

Out[35]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

```
In [36]: np.diag(k)
```

```
Out[36]: array([0, 4, 8])
```

15. tril / triu

```
In [37]: np.tril(a)
```

```
Out[37]: array([[10,  0,  0],
                [ 1,  6,  0]])
```

```
In [38]: b = np.empty(9)
         s = 0
         a_rows, a_cols = a.shape
         for i in range(a_rows):
             for j in range(a_cols):
                 if i < j:
                     b[s] = 0
                     s += 1
                 else:
                     b[s] = a[i,j]
                     s += 1
         print("without using inbuilt api\n",b.reshape(3,3))
```

```
without using inbuilt api
 [[1.000000e+001 0.000000e+000 0.000000e+000]
 [1.000000e+000 6.000000e+000 0.000000e+000]
 [0.000000e+000 1.852187e-317 0.000000e+000]]
```

Sparse Matrix Operations

There exists various formats in which sparse matrices can be represented. Formats can be divided into two groups:

1. Those that support efficient modification, such as DOK (Dictionary of keys), LIL (List of lists), or COO (Coordinate list). These are typically used to construct the matrices.

1. Those that support efficient access and matrix operations, such as CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column).

`Coordinate list (COO)`

COO stores a list of (row, column, value) tuples. Ideally, the entries are sorted (by row index, then column index) to improve random access times. This is another format which is good for incremental matrix construction

'coo_matrix' is optimized to ***construct*** a sparse matrix. It is internally different than csr_matrix but when you simply print, it looks same.

`Compressed sparse Row (CSR)`

The compressed sparse row (CSR) or compressed row storage (CRS) format represents a matrix M by three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices. This format allows fast row access and matrix-vector multiplications.

The CSR format stores a sparse m × n matrix M in row form using three (one-dimensional) arrays (A, IA, JA). Let NNZ denote the number of nonzero entries in M. (Note that zero-based indices shall be used here.)

There are **two** types of CS matrices:-

`csr_matrix` considers row first and a `csc_matrix` considers column first.

```
E.g: Let's take a matrix,

mat = [[1, 0, 0],
       [5, 0, 2],
       [0, -1, 0],
       [0, 0, 3]]
```

csr_matrix gives the position of non-zero element in the row first then goes to second row, then to third and so on. for instance, csr_matrix(mat) returns:

```
(0, 0)  1.0 -- first row
(1, 0)  5.0 -- second row
(1, 2)  2.0 -- second row
(2, 1)  -1.0 --third row
(3, 2)  3.0 -- fourth row
```

Similarly **csc_matrix** gives the position of non-zero elements in the first column, then second column, and so on.

```
(0, 0)  1.0 -- first column
(1, 0)  5.0 -- first column
(2, 1)  -1.0 -- second column
(1, 2)  2.0 -- third column
```

Extra Read :

https://rushter.com/blog/scipy-sparse-matrices/ (https://rushter.com/blog/scipy-sparse-matrices/)

Week 3 Assignment

Building Recommender Systems for Movie Rating Prediction¶

Dataset:

```
- Notebook Editor > Module3_USL_v4 > Navigate > Files/home/jovyan/workrele
aseModule3data/*
```

In [39]:
```
"""
DATASET

# 1. "users.csv"
- MV_users == users == data[0]
- Shape: (6040 x 5)
- rows ==> 6040 users
- cols ==> 'User_ID', 'Gender', 'Age', 'Occupation', 'Zip-code'

# 2. "movies.csv"
- MV_movies == movies == data[1]
- Shape: (3883 x 21)
- rows ==> 3883 movies
- cols ==> MID, Title, Year, Genres (Documentry, Comedy, horror, Ad
venture, Animation, etc)

# 3. "train.csv"
- train == data[2]
- Shape: (700146 x 3)
- rows ==> Rating Value for a (UID, MID)
- cols ==> UID, MID, Rating [(1,2,3,4,5, 1 - lowest rating, 5 - hig
hest rating, 0 - missing value)]

# 4. "test".csv"
- test == data[3]
- Shape: (3883 x 21)
- rows ==> Rating Value for a (UID, MID)
- cols ==> UID, MID, Rating [(1,2,3,4,5, 1 - lowest rating, 5 - hig
hest rating)]
"""
print("Dataset Loaded.")
```

```
Dataset Loaded.
```

Building Recommender Systems for Movie Rating Prediction

In this assignment, we will build a recommender systems that predict movie ratings. MovieLense (https://grouplens.org/datasets/movielens/) has currently 25 million user-movie ratings. Since the entire data is too big, we use a 1 million ratings subset MovieLens 1M (https://www.kaggle.com/odedgolden/movielens-1m-dataset), and we reformatted the data to make it more convenient to use.

Starter codes

Now, we will be building a recommender system which has various techniques to predict ratings. The `class RecSys` has baseline prediction methods (such as predicting everything to 3 or to average rating of each user) and other utility functions. `class ContentBased` and `class Collaborative` inherit `class RecSys` and further add methods calculating item-item similarity matrix. You will be completing those functions using what we learned about content-based filtering and collaborative filtering.

`RecSys` 's `rating_matrix` method converts the (user id, movie id, rating) triplet from the train data (train data's ratings are known) into a utility matrix for 6040 users and 3883 movies.
Here, we create the utility matrix as a dense matrix (numpy.array) format for convenience. But in a real world data where hundreds of millions of users and items may exist, we won't be able to create the utility matrix in a dense matrix format (For those who are curious why, try measuring the dense matrix self.Mr using .nbytes()). In that case, we may use sparse matrix operations as much as possible and distributed file systems and distributed computing will be needed. Fortunately, our data is small enough to fit in a laptop/pc memory. Also, we will use numpy and scipy.sparse, which allow significantly faster calculations than calculating on pandas.DataFrame object.
In the `rating_matrix` method, pay attention to the index mapping as user IDs and movie IDs are not the same as array index.

In [ ]:
```
class RecSys():

    def __init__(self,data):

        # stores everything as a tuple
        self.data=data

        # stores 6040 user IDs (1, 2, 3, ..., 6039, 6040)
        self.allusers = list(self.data.users['uID'])

        # stores 3883 movie IDs (1, 2, 3, ..., 3951, 3952)
        self.allmovies = list(self.data.movies['mID'])
        self.genres = list(self.data.movies.columns.drop(['mID', 't
itle', 'year']))

        # Stores actual MID: Index MID mapping
        """
        mid2idx --> {Movie_ID: Index}
        {
            MID: Index
            1:    0,
            2:    1,
            3:    2,
            ...
            3951: 3881
            3952: 3882
        }
        """
        self.mid2idx = dict(zip(self.data.movies.mID,list(range(len
(self.data.movies)))))

        # Stores actual UID: Index UID mapping
        """
        uid2idx --> {User_ID: Index}
            {
                UID: Index
              1:    0,
                2:    1,
                3:    2,
                ...
                6039: 6038
                6040: 6039
            }
        """
        self.uid2idx = dict(zip(self.data.users.uID,list(range(len
(self.data.users)))))

        # Stores User - Movie Rating, using "indexes" instead of ac
tual IDs.
        # 6040 x 3883
        self.Mr = self.rating_matrix()

        # Stores
        self.Mm = None
        # Stores sim scores between all movies, using "indexes" ins
tead of actual IDs.
        # 3883 x 3883
        self.sim = np.zeros((len(self.allmovies),len(self.allmovie
s)))
        return

    def rating_matrix(self):
        """
        Convert the rating matrix to numpy array of shape (#alluser
s, #allmovies)

        self.Mr --> Creates a Compressed Sparse Matrix containing m
appings of actual User IDs and Movie IDs
                    with usuable IDs. Actual IDs may be alphanumeri
c or something which may not be visible to
                    the developer, so we need to use self.Mr for ac
cessing IDs.

                    - 6040 users and 3883 movies.
                    - Rows represent users, cols represent movie ra
tings.
                    - Most of them are zero (could be missing valu
e!).
                    - uid2idx represets actual index of users.

                    E.g.
                    User #2233's ratings are: '[5 0 0 ... 0 0 0]'
                    User #2233 has rated `157` movies!
        """

        # Stores indexes of actual User_IDS, and not the original U
ID values
        ind_user = [self.uid2idx[x] for x in self.data.train.uID]

        # Stores indexes of actual Movie_IDs, and not the original
MID values
        ind_movie = [self.mid2idx[x] for x in self.data.train.mID]

        # Final Rating Matrix (self.Mr)
        # - containing indexes instead of original userIds and Movi
eIds.
        # - `coo_matrix`: is used to construct a CS matrix for opti
mized operations.
        rating_train = list(train.rating)
        return np.array(coo_matrix((rating_train, (ind_user, ind_mo
vie)), shape=(len(self.allusers), len(self.allmovies))).toarray())


    def predict_everything_to_3(self):
        """
        Predict everything to 3 for the test data.

        ===> Construct and return a numpy array containing each ele
ment as 3.
        HINT: One could use np.ones to construct a numpy array quic
```

```
kly.
        """
        # your code here
        return


    def predict_to_user_average(self):
        """
        Predict to average rating for the user. Returns numpy array
of shape (#users,)

        ==> Means we need to compute "average" movie rating for all
users provided in self.Mr. We can compute
            the average by taking sum of all ratings for all the us
ers divided by count of **valid** ratings.

            Once we get the average rating for all users, we need t
o return some of its ratings i.e.
            for Test data Actual UIDs.

        Hint: Remember we have stored "indexes" of actual UIDs and
MIDs in self.Mr, so for extracting
              a particular User id from self.Mr we need to provide
it's index, `self.uid2idx[Actual_User_ID]`

              Computing sum of a matrix could be easily done using
`matrix.sum(axis)`

              The algorithm is:
                  1. Average Rating Computation for all users
                  2. Retireve index of test users from `data.test.u
ID`
                  3. Compute rating for all test users using Step 1
and 2.
        """
        # your code here
        return


    def predict_from_sim(self,uid,mid):
        """
        Predict a user rating on a movie given userID and movieID

         ==> Means we need to compute average Movie Rating for a gi
ven index of UID and index of MovieID.

         Hint: Remember we have stored "indexes" of actual UIDs and
MIDs in self.Mr, so for extracting
               a particular User id from self.Mr we need to provide
it's index, self.uid2idx[Actual_User_ID].

               Use an example of uid = 2233; mid = 440; to test res
ults.

               Taking just the dot product will yield a Biased Rati
ng matrix since it contains movies which
```

```
               were not rated by the user (zeros). To minimise this
bias, we should divide this product by
               the count of valid ratings.

               One could utilize np.dot(Matrix, boolean) to count n
umber of valid ratings!

               The algorithm is:
                   1. Get index of the provided user id (index_user
ID)
                   2. Get all the user ratings for the user using i
ndex_userID (ratings_index_userID)
                   3. Get index of the provided movie id (index_mov
ieID)
                   4. Get all the similarity scores using index_mov
ieID (movie_sims)
                   5. Take the **averaged** dot product.

        More on:-
        https://www.geeksforgeeks.org/item-to-item-based-collaborat
ive-filtering/#:~:text=Prediction%20Computation"
        """
        # your code here
        return


    def predict(self):
        """
        Predict ratings in the test data. Returns predicted rating
in a numpy array of size (# of rows in testdata,)

        ===> Construct and return a numpy array containing predicte
d ratings
        HINT: One could use `self.predict_from_sim()` passing every
uid and mid from the test data.
        """
        test_preds = []
        for i in range(len(self.data.test)):
            # your code here
        return


    def rmse(self,yp):
        yp[np.isnan(yp)]=3 #In case there is nan values in predicti
on, it will impute to 3.
        yt=np.array(self.data.test.rating)
        return np.sqrt(((yt-yp)**2).mean())


class ContentBased(RecSys):
    def __init__(self,data):
        super().__init__(data)
        self.data=data
        self.Mm = self.calc_movie_feature_matrix()

    def calc_movie_feature_matrix(self):
        """
        Create movie feature matrix in a numpy array of shape (#all
movies, #genres)

        HINT: Use `self.data.movies` and explore its genres.
        """
        # your code here
        return

    def calc_item_item_similarity(self):
        """
        Create item-item similarity using Jaccard similarity

        self.sim --> all movie similarity scores are stored in this
matrix.

                     - A Movie x Movie matrix
                     - 3883 rows and 3883 cols
                     - Each row-col pair represent a similarity sco
re, how similar the movie is.

        Does the super class update self.sim values in the other cl
ass?
        Best way to debug this? See it for yourself. Take a small e
xample and check.

        Jaccard similarity Function:
        https://www.statology.org/jaccard-similarity-python/#:~:tex
t=%23define%20Jaccard%20Similarity%20function

        OR,
        self.sim = 1 - pairwise_distances(self.Mm, metric="jaccar
d")

        """
        # your code here
        return


class Collaborative(RecSys):
    def __init__(self,data):
        super().__init__(data)

    def calc_item_item_similarity(self, simfunction, *X):
        """
        Create item-item similarity using similarity function.
        X is an optional transformed matrix of Mr
        """
        if len(X)==0:
            self.sim = simfunction()
        else:
            # *X passes in a tuple format of (X,), to X[0] will be
the actual transformed matrix
            self.sim = simfunction(X[0])

    def cossim(self):
        """
        Calculates item-item similarity for all pairs of items usin
g cosine similarity (values from 0 to 1)
        on utility matrix.

        Returns a cosine similarity matrix of size (#all movies, #a
ll movies)

        Both Boundary checks (Step 5, 7) are important.

        HINT:
        Scope = Movie Ratings Array ==> self.Mr

        userid   movie1  movie2  movie3  movie440 ... movie3883
        user1      0         0        0       0          0
        user2      0         0        0       0          0
        user3      0         0        0       0          0
         ...
        user6040   0         0        0       0          0

        Algorithm:
        1. Compute **averaged** movie ratings for all users (movie_
ratings_allUsers)

        2. Create a sparse matrix for operating cosine on its value
s:
        $:> movie_ratings_array = np.repeat(
                                    np.expand_dims(movie_ratings_al
lUsers, axis=1), self.Mr.shape[1], axis=1
                                  )

        3. Take care of all the zero ratings (missing value/itentio
nally we don't know):
        $:> movie_ratings_array_adjusted = self.Mr + (self.Mr==0)*m
ovie_ratings_array - movie_ratings_array

        4. Average all the ratings: divide by its magnitude!
        $:> MR_avg = movie_ratings_array_adjusted/np.sqrt((movie_ra
tings_array_adjusted**2).sum(axis=0))

        5. Put a Boundary check # 1: since dividing by magnitude ma
y produce inf, zeros, etc. Set nans to 0.

        6. Perform an item-item cosine similarity using: np.dot(mat
rix.T, matrix)

        7. Put a Boundary check # 2: Covariance/correlation values
for np.dot([M.T, M]) matrix should have
           diagonal set to 1. This may cause an autograder issue, s
o explicilty set diagonals to 1.

        $:> for i in range(len(self.allmovies)):
                item_item_cosine_sim[i, i] = 1

        8. Normalized Cosine Formula:
        -> Check the Resources/Slide Deck M3..
        """
        # your code here
        pass


    def jacsim(self,Xr):
        """
        Calculates item-item similarity for all pairs of items usin
g jaccard similarity (values from 0 to 1)
        Xr is the transformed rating matrix.

        Check Jaccard Distance and Similarity methodology.
        """

        # n = Xr.shape[1]
        # maxr = int(Xr.max())
        # if maxr>1:
        #     intersection = np.zeros((n,n)).astype(int)
        #     for i in range(1, maxr+1):
        #         csr = csr_matrix((Xr==i).astype(int))
        #         intersection = intersection + np.array(csr.T.dot
(csr).toarray()).astype(int)

        # Convert Xr into a CSR format
        # csr0 = csr_matrix((Xr>0).astype(int))

        # Take the dot product
        # nz_inter = np.array(csr0.T.dot(csr0).toarray()).astype(in
t)

        # Formula JS:
        # A = (Xr>0).astype(bool)
        # rowsum = A.sum(axis=0)
        # rsumtile = np.repeat(rowsum.reshape((n,1)),n,axis=1)
        # union = rsumtile.T + rsumtile - nz_inter

        # Perform the two boundary checks:-
        #  - since dividing by magnitude may produce inf, zeros, et
c. Set nans to 0.
        #  - Covariance/correlation values for np.dot([M.T, M]) mat
rix should have
        #    diagonal set to 1.

        # your code here...

        return
        pass
```