Yashar Deldjoo
Massimo Quadrana

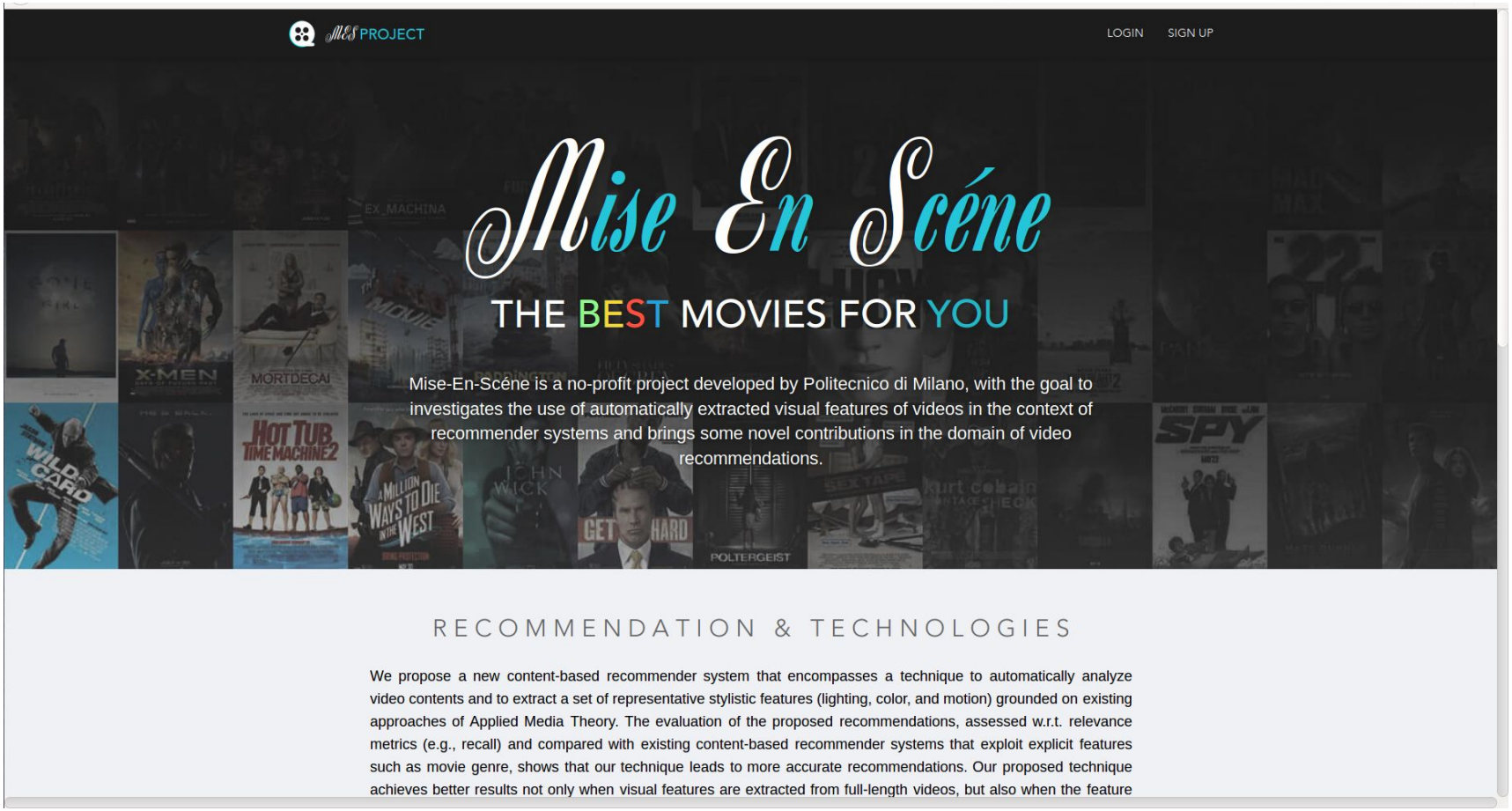
POLITECNICO DI MILANO

Lab 2: “Building the First Recommender System in Python”

Recommender Systems Course

Table of Contents

Mise En Scene Project



http://www.cybersoft.io/

Table of Contents

Dense Matrix

- Row-Major Order (e.g. C)

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Address	Value
0	a_{00}
1	a_{01}
2	a_{02}
3	a_{10}
4	a_{11}
5	a_{12}
6	a_{20}
7	a_{21}
8	a_{22}

Dense Matrix

- Column-Major Order (e.g. Fortran)

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Address	Value
0	a_{00}
1	a_{10}
2	a_{20}
3	a_{01}
4	a_{11}
5	a_{21}
6	a_{02}
7	a_{12}
8	a_{22}

Table of Contents

The difference between row-major and column-major order is simply that the order of the dimensions is reversed.

Dense Matrix

- **Row-major Order**
  - Entries in the same **row** are contiguous in memory.
  - Indexing: Fast (left-right), Slow (Top-down)
  - Examples: C/C++/C#, Pascal, Mathematica, SAS, python-numpy
- **Column-major Order**
  - Entries in the same **column** are contiguous in memory.
  - Indexing: Slow (left-right), Fast (Top-down)
  - Examples: Fortan, Open-GL, MATLAB, R, Octave, Julia

Dense Matrix

- In Numpy:
  - numpy.array([1, 2, 3], order='C')
    # row-major order; the default (C = The C Language)
  - numpy.array([1, 2, 3], order='F')
    # column-major order (F = Fortran)

Row v.s. Column-Major Order

```
import numpy as np
import time
A = np.ones((20000,20000), order = 'F')

t0 = time.time()
for i in range(20000):
   A[:,i] *= 10

t1 =  time.time()
print("Column-Order (F): " + str(t1-t0))
-------------------------------------------------

t0 = time.time()
for i in range(20000):
   A[i,:] *= 10

t1 = time.time()
print("Row-Order (C): " + str(t1-t0))
```

Column-Order (F):
0.38 (seconds)

Row-Order (C):
2.95 (seconds)

Row v.s. Column-Major Order

```
import numpy as np
import time
A = np.ones((20000,20000), order = 'F')

t0 = time.time()
for i in range(20000):
   A[:,i] *= 10

t1 =  time.time()
print("Column-Order (F): " + str(t1-t0))
-------------------------------------------------

t0 = time.time()
for i in range(20000):
   A[i,:] *= 10

t1 = time.time()
print("Row-Order (C): " + str(t1-t0))
```

Column-Order (F):
0.38 (seconds)

7.5 times !!!

Row-Order (C):
2.95 (seconds)

Row v.s. Column-Major Order

```
import numpy as np
import time
A = np.ones((20000,20000), order = 'F')

t0 = time.time()
for i in range(20000):
   A[:,i] *= 10

t1 =  time.time()
print("Column-Order (F): " + str(t1-t0))
-------------------------------------------------

t0 = time.time()
for i in range(20000):
   A[i,:] *= 10

t1 = time.time()
print("Row-Order (C): " + str(t1-t0))
```

Column-Order (F):
0.38 (seconds)

7.5 times !!!

Row-Order (C):
2.95 (seconds)

Row v.s. Column-Major Order

```
import numpy as np
import time
A = np.ones((20000,20000), order = 'C')

t0 = time.time()
for i in range(20000):
   A[:,i] *= 10

t1 =  time.time()
print("Column-Order (F): " + str(t1-t0))
-------------------------------------------------

t0 = time.time()
for i in range(20000):
   A[i,:] *= 10

t1 = time.time()
print("Row-Order (C): " + str(t1-t0))
```

Column-Order (F):
2.98 (seconds)

Row-Order (C):
0.37 (seconds)

Row v.s. Column-Major Order

```
import numpy as np
import time
A = np.ones((20000,20000), order = 'C')

t0 = time.time()
for i in range(20000):
   A[:,i] *= 10

t1 =  time.time()
print("Column-Order (F): " + str(t1-t0))
-------------------------------------------------

t0 = time.time()
for i in range(20000):
   A[i,:] *= 10

t1 = time.time()
print("Row-Order (C): " + str(t1-t0))
```

Column-Order (F):
2.98 (seconds)

7.5 times !!!

Row-Order (C):
0.37 (seconds)

Coordinate List (COO) Format

The triplets can be stored as: [row, cols, values]

	0	1	2	3	4	5
0	1			1		1
1		1				
2						1
3			1		1	1
4	1				1	

rows = [0, 0, 0, 1, 2, 3, 3, 3, 4, 4]
cols = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

→ **The COO format needs 3nnz elements to store the matrix.**

Compressed Sparse Row (CSR) Format

The triplets can be stored as: [rowptr, cols, values]

	0	1	2	3	4	5
0	1			1		1
1		1				
2						1
3			1		1	1
4	1				1	

rows = [0, 0, 0, 1, 2, 3, 3, 3, 4, 4]

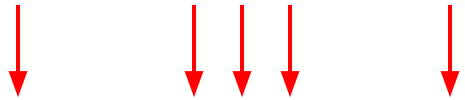
cols = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]

values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Compressed Sparse Row (CSR) Format

The triplets can be stored as: [rowptr, cols, values]

	0	1	2	3	4	5
0	1			1		1
1		1				
2						1
3			1		1	1
4	1				1	



rows = [0, 0, 0, 1, 2, 3, 3, 3, 4, 4]

cols = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]

values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Compressed Sparse Row (CSR) Format

The triplets can be stored as: [row, cols, values]

	0	1	2	3	4	5
0	1			1		1
1		1				
2						1
3			1		1	1
4	1				1	

rowptr = [0, 3, 4, 5, 8]
colind = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

→ **The CSR format needs 2nnz+n elements to store the matrix.**

Compressed Sparse Row (CSR) Format

The triplets can be stored as: [row, cols, values]

	0	1	2	3	4	5
0	1			1		1
1		1				
2						1
3			1		1	1
4	1				1	

rowptr = [0, 3, 4, 5, 8]
colind = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

→ **row pointer**
column index
values

Compressed Sparse Row (CSC) Format

Similarly, we have compressed sparse column (CSC) format.

Sparse Formats in SciPy

```
> import numpy as np
> import scipy.sparse as sp
> rowind = [0, 0, 0, 1, 2, 3, 3, 3, 4, 4]
>  colind = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
> values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> A = sp.coo_matrix( (values,(colind,rowind)),dtype=np.float64 )
```

Sparse Formats in SciPy

```
> import numpy as np
> import scipy.sparse as sp
> rowind = [0, 0, 0, 1, 2, 3, 3, 3, 4, 4]
>  colind = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
> values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> A = sp.coo_matrix( (values,(colind,rowind)),dtype=np.float64 )
```

```
> print(A)
```



```
(0, 0)    1
(0, 3)    1
(0, 5)    1
(1, 1)    1
(2, 5)    1
(3, 2)    1
(3, 4)    1
(3, 5)    1
(4, 1)    1
(4, 4)    1
```

Sparse Formats in SciPy

```
> import numpy as np
> import scipy.sparse as sp
> rowind = [0, 0, 0, 1, 2, 3, 3, 3, 4, 4]
>  colind = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
> values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> A = sp.coo_matrix( (values,(colind,rowind)),dtype=np.float64 )
```

```
> B = A.tocsr()
> print(B)
```

→

```
(0, 0)  1.0
(1, 1)  1.0
(1, 4)  1.0
(2, 3)  1.0
(3, 0)  1.0
(4, 3)  1.0
(4, 4)  1.0
(5, 0)  1.0
(5, 2)  1.0
(5, 3)  1.0
```

Sparse Formats in SciPy

```
> import numpy as np
> import scipy.sparse as sp
> rowptr = [0, 3, 4, 5, 8]
>  colind = [0, 3, 5, 1, 5, 2, 4, 5, 1, 4]
> values = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> B= sp.csr_matrix( (values,(colind,rowptr)),dtype=np.float64 )
```

```
> print(B)
```



```
(0, 0)   1.0
(1, 1)   1.0
(1, 4)   1.0
(2, 3)   1.0
(3, 0)   1.0
(4, 3)   1.0
(4, 4)   1.0
(5, 0)   1.0
(5, 2)   1.0
(5, 3)   1.0
```

Sparse Formats in SciPy

```
> print(A.todense())
> print(B.todense())
```

```
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  1.]
 [ 0.  0.  0.  1.  0.]
 [ 1.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  1.]
 [ 1.  0.  1.  1.  0.]]
```

Sparse Formats in SciPy

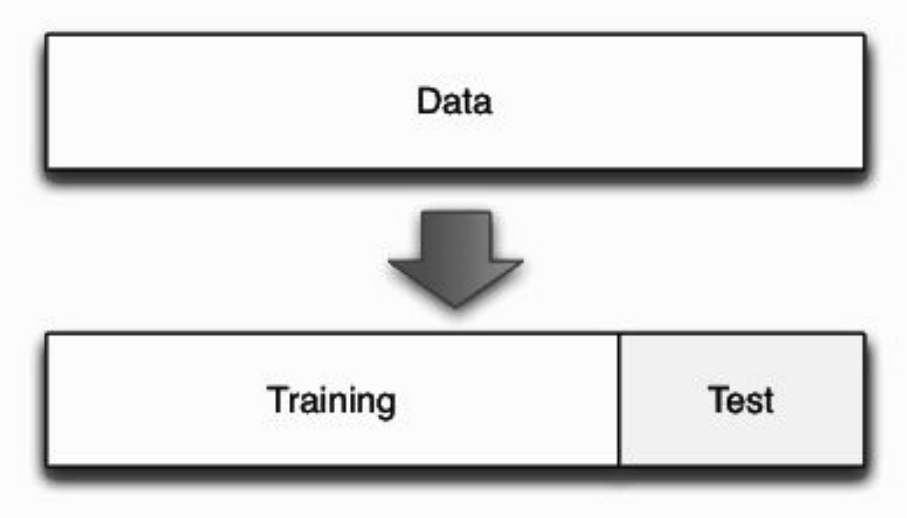
Comparisons:

COO	CSR	CSC
Facilitates fast conversion among sparse formats	Efficient arithmetic operations CSR + CSR, CSR * CSR, etc.	Efficient arithmetic operations CSC + CSC, CSC * CSC, etc.
Very fast conversion to and from CSR/CSC formats	**Efficient** **row** slicing. **Slow** *column* slicing operations	**Slow** *row* slicing. **Efficient** **column** slicing operations
Intended usage: COO is a fast format for constructing **sparse matrices**. Once a matrix has been created, convert to CSR or CSC format for fast operations	Changes to the sparsity structure (adding/deleting nonzeros) are expensive . e.g. good for **user-user CF**	Changes to the sparsity structure (adding/deleting nonzeros) are expensive . e.g. good for **item-item CF**

Table of Contents

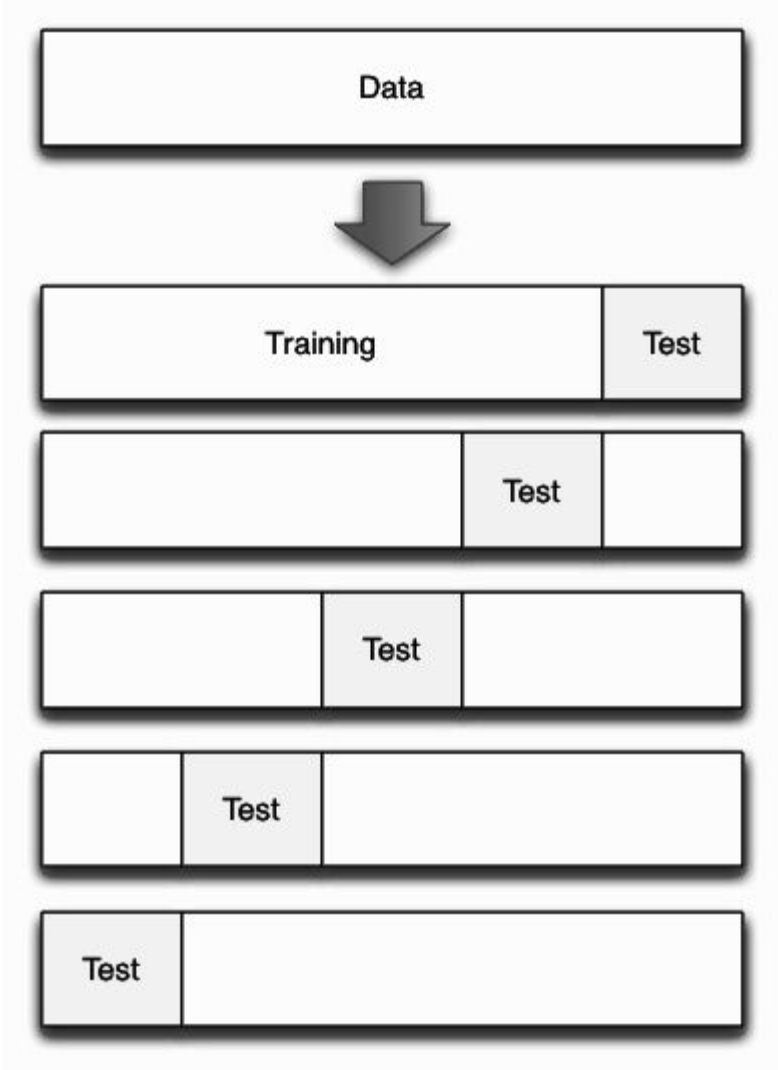
Evaluation - Design Issues

HOLD OUT



Evaluation - Design Issues

Cross Validation



Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗

Prec@k
1/1

Recall@k

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗

Prec@k
1/1
2/2

Recall@k

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗		

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	

Recall@k

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/5	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/6	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/7	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/8	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/9	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/10	

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/5	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/6	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/7	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/8	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/9	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/10	

Precision @10 for user

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	1/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/5	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/6	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/7	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/8	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/9	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/10	

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	1/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	2/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/5	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/6	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/7	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/8	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/9	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/10	

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	1/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	2/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	2/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/5	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/6	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/7	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/8	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/9	
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/10	

Offline Evaluation Metrics

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Prec@k	Recall@k
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	1/1	1/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/2	2/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	2/3	2/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/4	3/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/5	3/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/6	3/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	3/7	3/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/8	4/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/9	4/4
	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	4/10	4/4

Recall @10 for user

Offline Evaluation Metrics

Is precision@k affected by position of recommended items?

Offline Evaluation Metrics

An Example:

Precision @10 = ?

Offline Evaluation Metrics

An Example:

4/10

4/10

Offline Evaluation Metrics

An Example:

4/10

4/10

How to differentiate between the two above recommendation results?

AVERAGE PRECISION (AP)

Offline Evaluation Metrics

- **Average Precision (AP)**
  - Computes precision at every relevant position & averages it out

(1/1 + 2/2 + 3/4 +4/8)/4 = 0.80

(1/3 + 2/7 + 3/9 +4/10)/4 = 0.34

Offline Evaluation Metrics

- **Average Precision (AP)**
  - Computes precision at every relevant position & averages it out

$(1/1 + 2/2 + 3/4 + 4/8)/4 = 0.80$

Average Precision is sensitive to ranking;
Precision is NOT !

$(1/3 + 2/7 + 3/9 + 4/10)/4 = 0.34$

Offline Evaluation Metrics

- **Average Precision (AP)**
  - Computes precision at every relevant position & averages it out

(1/1 + 2/2 + 3/4 +4/8)/4 = 0.80

MAP = (0.80+0.34)/2 = 0.57

(1/3 + 2/7 + 3/9 +4/10)/4 = 0.34

Table of Contents

- An Example of an Online Video Recommender
- Dense matrix
  - Column-major vs. Row-major order
- Sparse matrix
  - Coordinate List (COO) Format
  - Compressed Sparse Row (CSR) Format
  - Compressed Sparse Column (CSC) Format
- Evaluation
  - Online v.s. offline
  - Metrics
  - Design issues (hold out/Cross Validation)
- **Build your first non-personalized recommender system**

Questions

Thanks for your Attention!