Hands-on with PyData:
How to Build a Minimal Recommendation Engine

April 8, 2014

1 Introduction

1.1 Welcome!

- About Unata
  - What we do
  - What we use Python for (everything!)
- Environment + data files check
  - Local Setup
    * This is the *prefered setup*.
    * Instructions to set up a local environment and links to download handout and data files: http://unatainc.github.io/pycon2014/
  - Hosted Setup with Ipydra:
    * This is only a *fallback option* for those without a local environment.
    * Hosted environment: http://pycon.unata.com

1.2 The Recommendation Problem

Recommenders have been around since at least 1992. Today we see different flavours of recommenders, deployed across different verticals:

- Amazon
- Netflix
- Facebook
- Last.fm.

What exactly do they do?

1.2.1 Definitions from the literature

- *In a typical recommender system people provide recommendations as inputs, which the system then aggregates and directs to appropriate recipients.* – Resnick and Varian, 1997
- *Collaborative filtering simply means that people collaborate to help one another perform filtering by recording their reactions to documents they read.* – Goldberg et al, 1992
- *In its most common formulation, the recommendation problem is reduced to the problem of estimating ratings for the items that have not been seen by a user. Intuitively, this estimation is usually based on the ratings given by this user to other items and on some other information [...] Once we can estimate ratings for the yet unrated items, we can recommend to the user the item(s) with the highest estimated rating(s).* – Adomavicius and Tuzhilin, 2005

- *Driven by computer algorithms, recommenders help consumers by selecting products they will probably like and might buy based on their browsing, searches, purchases, and preferences.* – Konstan and Riedl, 2012

1.2.2 Notation

- U is the set of users in our domain. Its size is $|U|$.
- I is the set of items in our domain. Its size is $|I|$.
- $I(u)$ is the set of items that user u has rated.
- $-I(u)$ is the complement of $I(u)$ i.e., the set of items not yet seen by user u.
- $U(i)$ is the set of users that have rated item i.
- $-U(i)$ is the complement of $U(i)$.
- $S(u, i)$ is a function that measures the utility of item i for user u.

1.2.3 Goal of a recommendation system

$$i^* = \operatorname{argmax}_{i \in -I(u)} S(u, i), \forall u \in U$$

1.2.4 Problem statement

The recommendation problem in its most basic form is quite simple to define:

user_id, movie_id	m_1	m_2	m_3	m_4	m_5
u_1	?	?	4	?	1
u_2	3	?	?	2	2
u_3	3	?	?	?	?
u_4	?	1	2	1	1
u_5	?	?	?	?	?
u_6	2	?	2	?	?
u_7	?	?	?	?	?
u_8	3	1	5	?	?
u_9	?	?	?	?	2

Given a partially filled matrix of ratings ($|U|x|I|$), estimate the missing values.

1.3 Well-known Solutions to the Recommendation Problem

1.3.1 Content-based filtering

Recommend based on the user's rating history.

Generic expression (notice how this is kind of a 'row-based' approach):

$$r_{u,i} = \operatorname{aggr}_{i' \in I(u)} [r_{u,i'}]$$

A simple example using the mean as an aggregation function:

$$r_{u,i} = \bar{r}_u = \frac{\sum_{i' \in I(u)} r_{u,i'}}{|I(u)|}$$

1.3.2 Collaborative filtering

Recommend based on other user's rating histories.

Generic expression (notice how this is kind of a 'col-based' approach):

$$r_{u,i} = \text{aggr}_{u' \in U(i)}[r_{u',i}]$$

A simple example using the mean as an aggregation function:

$$r_{u,i} = \bar{r}_i = \frac{\sum_{u' \in U(i)} r_{u',i}}{|U(i)|}$$

1.3.3 Hybrid solutions

The literature has lots of examples of systems that try to combine the strengths of the two main approaches. This can be done in a number of ways:

- Combine the predictions of a content-based system and a collaborative system.
- Incorporate content-based techniques into a collaborative approach.
- Incorporarte collaborative techniques into a content-based approach.
- Unifying model.

1.3.4 Challenges

Availability of item metadata Content-based techniques are limited by the amount of metadata that is available to describe an item. There are domains in which feature extraction methods are expensive or time consuming, e.g., processing multimedia data such as graphics, audio/video streams. In the context of grocery items for example, it's often the case that item information is only partial or completely missing. Examples include:

- Ingredients
- Nutrition facts
- Brand
- Description
- County of origin

New user problem A user has to have rated a sufficient number of items before a recommender system can have a good idea of what their preferences are. In a content-based system, the aggregation function needs ratings to aggregate.

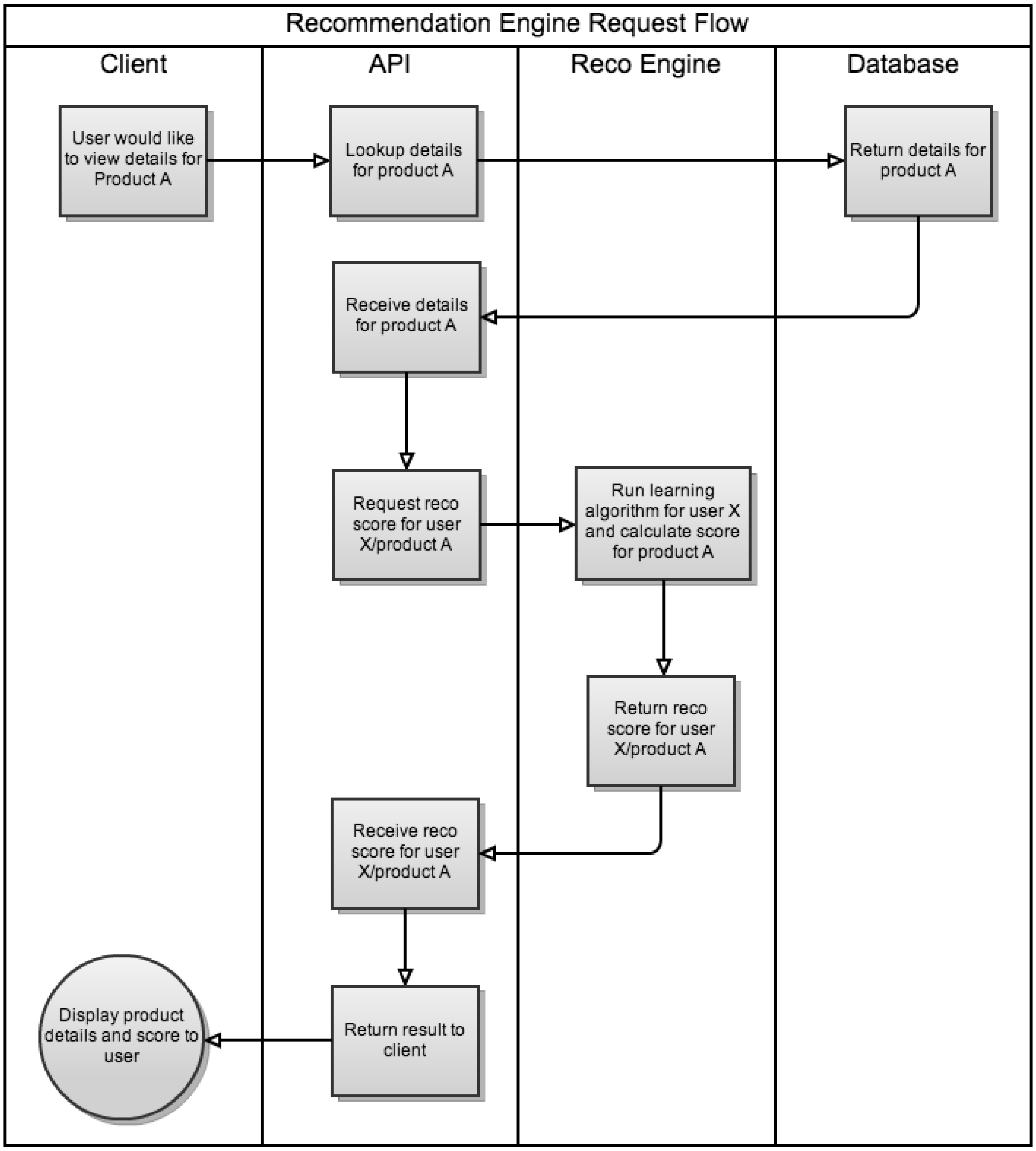
New item problem Collaborative filters rely on an item being rated by many users to compute aggregates of those ratings. Think of this as the exact counterpart of the new user problem for content-based systems.

Data sparsity When looking at the more general versions of content-based and collaborative systems, the success of the recommender system depends on the availability of a critical mass of user/item iteractions. We get a first glance at the data sparsity problem by quantifying the ratio of existing ratings vs $|U|x|I|$. A highly sparse matrix of interactions makes it difficult to compute similarities between users and items. As an example, for a user whose tastes are unusual compared to the rest of the population, there will not be any other users who are particularly similar, leading to poor recommendations.

1.4 Flow Chart: a Recommendation Engine in Context

```
In [1]: from IPython.core.display import Image
        Image(filename='./pycon_reco_flow.png')
```

```
Out[1]:
```



1.5 About this tutorial

1.5.1 References

We've put this together from our experience and a number of sources, please check the references at the bottom of this document.

1.5.2 Dataset

MovieLens from GroupLens Research: grouplens.org

The MovieLens 1M data set contains 1 million ratings collected from 6000 users on 4000 movies.

1.5.3 What this tutorial is

The goal of this tutorial is to provide you with a hands-on overview of two of the main libraries from the scientific and data analysis communities. We're going to use:

- numpy – numpy.org
- pandas – pandas.pydata.org
- (bonus) pytables – pytables.org

1.5.4 What this tutorial is not

- An exhaustive overview of the recommendation literature
- A set of recipes that will win you the next Netflix/Kaggle/etc challenge.

1.6 Roadmap

What exactly are we going to do? Here's a high-level overview:

- learn about NumPy arrays
- learn about Series and DataFrames
- iterate over a few implementations of a minimal reco engine
- challenge

1.7 NumPy: Numerical Python

1.7.1 What is it?

It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

```
In [2]: import numpy as np

        # set some print options
        np.set_printoptions(precision=4)
        np.set_printoptions(threshold=5)
        np.set_printoptions(suppress=True)

        # init random gen
        np.random.seed(2)
```

1.7.2 NumPy's basic data structure: the ndarray

Think of ndarrays as the building blocks for pydata. A multidimensional array object that acts as a container for data to be passed between algorithms. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying any data.

```
In [3]: import numpy as np

        # build an array using the array function
        arr = np.array([0, 9, 5, 4, 3])
        arr
```

```
Out[3]: array([0, 9, 5, 4, 3])
```

1.7.3 Array creation examples

There are several functions that are used to create new arrays:

- `np.array`
- `np.asarray`
- `np.arange`
- `np.ones`
- `np.ones_like`
- `np.zeros`
- `np.zeros_like`
- `np.empty`
- `np.random.randn` and other funcs from the random module

```
In [4]: np.zeros(4)

Out[4]: array([ 0.,  0.,  0.,  0.])

In [5]: np.ones(4)

Out[5]: array([ 1.,  1.,  1.,  1.])

In [6]: np.empty(4)

Out[6]: array([ -2.3158e+077,  -2.3158e+077,   6.9467e-310,   2.8248e-309])

In [7]: np.arange(4)

Out[7]: array([0, 1, 2, 3])
```

1.7.4 dtype and shape

NumPy's arrays are containers of homogeneous data, which means all elements are of the same type. The 'dtype' propery is an object that specifies the data type of each element. The 'shape' property is a tuple that indicates the size of each dimension.

```
In [8]: arr = np.random.randn(5)
        arr

Out[8]: array([-0.4168, -0.0563, -2.1362,  1.6403, -1.7934])

In [9]: arr.dtype

Out[9]: dtype('float64')

In [10]: arr.shape

Out[10]: (5,)

In [11]: # you can be explicit about the data type that you want
         np.empty(4, dtype=np.int32)

Out[11]: array([         0, -805306368,          0, -805306368], dtype=int32)

In [12]: np.array(['numpy','pandas','pytables'], dtype=np.string_)

Out[12]: array(['numpy', 'pandas', 'pytables'],
               dtype='|S8')

In [13]: float_arr = np.array([4.4, 5.52425, -0.1234, 98.1], dtype=np.float64)
         # truncate the decimal part
         float_arr.astype(np.int32)

Out[13]: array([ 4,  5,  0, 98], dtype=int32)
```

1.7.5 Indexing and slicing

Just what you would expect from Python

```
In [14]: arr = np.array([0, 9, 1, 4, 64])
         arr[3]
```

```
Out[14]: 4
```

```
In [15]: arr[1:3]
```

```
Out[15]: array([9, 1])
```

```
In [16]: arr[1:4:2]
```

```
Out[16]: array([9, 4])
```

```
In [17]: arr[::2]
```

```
Out[17]: array([ 0,  1, 64])
```

```
In [18]: arr[:2]
```

```
Out[18]: array([0, 9])
```

```
In [19]: arr[-2:]
```

```
Out[19]: array([ 4, 64])
```

```
In [20]: # set the last two elements to 555
         arr[-2:] = 555
         arr
```

```
Out[20]: array([  0,   9,   1, 555, 555])
```

(BONUS) Indexing behaviour for multidimensional arrays A good way to think about indexing in multidimensional arrays is that you are moving along the values of the shape property. So, a 4d array `arr_4d`, with a shape of `(w,x,y,z)` will result in indexed views such that:

- `arr_4d[i].shape == (x,y,z)`
- `arr_4d[i,j].shape == (y,z)`
- `arr_4d[i,j,k].shape == (z,)`

For the case of slices, what you are doing is selecting a range of elements along a particular axis:

```
In [21]: arr_2d = np.array([[5,3,4],[0,1,2],[1,1,10],[0,0,0.1]])
         arr_2d
```

```
Out[21]: array([[  5. ,   3. ,   4. ],
                [  0. ,   1. ,   2. ],
                [  1. ,   1. ,  10. ],
                [  0. ,   0. ,   0.1]])
```

```
In [22]: # get the first row
         arr_2d[0]
```

```
Out[22]: array([ 5.,  3.,  4.])
```

```
In [23]: # get the first column
         arr_2d[:,0]
```

```
Out[23]: array([ 5.,  0.,  1.,  0.])
```

```
In [24]: # get the first two rows
         arr_2d[:2]
```

```
Out[24]: array([[ 5.,  3.,  4.],
                [ 0.,  1.,  2.]])
```

Careful, it's a view! A slice does not return a copy, which means that any modifications will be reflected in the source array. This is a design feature of NumPy to avoid memory problems.

```
In [25]: arr = np.array([0, 3, 1, 4, 64])
         arr

Out[25]: array([ 0,  3,  1,  4, 64])

In [26]: subarr = arr[2:4]
         subarr[1] = 99
         arr

Out[26]: array([ 0,  3,  1, 99, 64])
```

(Fancy) Boolean indexing Boolean indexing allows you to select data subsets of an array that satisfy a given condition.

```
In [27]: arr = np.array([10, 20])
         idx = np.array([True, False])
         arr[idx]

Out[27]: array([10])

In [28]: arr_2d = np.random.randn(5)
         arr_2d

Out[28]: array([-0.8417,  0.5029, -1.2453, -1.058 , -0.909 ])

In [29]: arr_2d < 0

Out[29]: array([ True, False,  True,  True,  True], dtype=bool)

In [30]: arr_2d[arr_2d < 0]

Out[30]: array([-0.8417, -1.2453, -1.058 , -0.909 ])

In [31]: arr_2d[(arr_2d > -0.5) & (arr_2d < 0)]

Out[31]: array([], dtype=float64)

In [32]: arr_2d[arr_2d < 0] = 0
         arr_2d

Out[32]: array([ 0.    ,  0.5029,  0.    ,  0.    ,  0.    ])
```

(Fancy) list-of-locations indexing Fancy indexing is indexing with integer arrays.

```
In [33]: arr = np.array([100, 101, 130, 131, 321, 123])
         arr[[1, 3, 4]]

Out[33]: array([101, 131, 321])

In [34]: arr = np.arange(18).reshape(6,3)
         arr

Out[34]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14],
                [15, 16, 17]])
```

```
In [35]: # fancy selection of rows in a particular order
         arr[[0,4,4]]
```

```
Out[35]: array([[ 0,  1,  2],
                [12, 13, 14],
                [12, 13, 14]])
```

```
In [36]: # index into individual elements and flatten
         arr[[5,3,1],[2,1,0]]
```

```
Out[36]: array([17, 10,  3])
```

```
In [37]: # select a submatrix
         arr[np.ix_([5,3,1],[2,1])]
```

```
Out[37]: array([[17, 16],
                [11, 10],
                [ 5,  4]])
```

–> Go to "Numpy Questions: Indexing"

1.7.6 Vectorization

Vectorization is at the heart of NumPy and it enables us to express operations without writing any for loops. Operations between arrays with equal shapes are performed element-wise.

```
In [38]: arr1 = np.array([0, 9, 1.02, 4, 32])
         arr2 = np.array([1, 45, 7.8, 5, 90])
         arr1 - arr2
```

```
Out[38]: array([ -1.  , -36.  ,  -6.78,  -1.  , -58.  ])
```

```
In [39]: arr1 * arr1
```

```
Out[39]: array([    0.    ,    81.    ,     1.0404,    16.    ,  1024.    ])
```

1.7.7 Broadcasting Rules

Vectorized operations between arrays of different sizes and between arrays and scalars are subject to the rules of broadcasting. The idea is quite simple in many cases:

```
In [40]: arr = np.array([0, 9, 1.02, 4, 64])
         5 * arr
```

```
Out[40]: array([   0. ,   45. ,    5.1,   20. ,  320. ])
```

```
In [41]: 10 + arr
```

```
Out[41]: array([ 10.  ,  19.  ,  11.02,  14.  ,  74.  ])
```

```
In [42]: arr ** .5
```

```
Out[42]: array([ 0.  ,  3.  ,  1.01,  2.  ,  8.  ])
```

The case of arrays of different shapes is slightly more complicated. The gist of it is that the shape of the operands need to conform to a certain specification. Don't worry if this does not make sense right away.

```
In [43]: mtx = np.random.randn(4,2)
         mtx
```

```
Out[43]: array([[ 0.5515,  2.2922],
                [ 0.0415, -1.1179],
                [ 0.5391, -0.5962],
                [-0.0191,  1.175 ]])
```

```
In [44]: vec = np.array([100, 100])
         vec
```

```
Out[44]: array([100, 100])
```

```
In [45]: mtx + vec
```

```
Out[45]: array([[ 100.5515,  102.2922],
                [ 100.0415,   98.8821],
                [ 100.5391,   99.4038],
                [  99.9809,  101.175 ]])
```

```
In [46]: mean_row = np.mean(mtx, axis=0)
         mean_row
```

```
Out[46]: array([ 0.2782,  0.4383])
```

```
In [47]: centered_rows = mtx - mean_row
         centered_rows
```

```
Out[47]: array([[ 0.2732,  1.8539],
                [-0.2367, -1.5562],
                [ 0.2608, -1.0344],
                [-0.2974,  0.7367]])
```

```
In [48]: np.mean(centered_rows, axis=0)
```

```
Out[48]: array([-0.,  0.])
```

```
In [49]: mean_col = np.mean(mtx, axis=1)
         mean_col
```

```
Out[49]: array([ 1.4218, -0.5382, -0.0286,  0.5779])
```

```
In [50]: centered_cols = mtx - mean_col
```

```
    ---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

    <ipython-input-50-26322f66ff99> in <module>()
----> 1 centered_cols = mtx - mean_col


    ValueError: operands could not be broadcast together with shapes (4,2) (4,)
```

```
In [51]: # make the 1-D array a column vector
         mean_col.reshape((4,1))
```

```
Out[51]: array([[ 1.4218],
                [-0.5382],
                [-0.0286],
                [ 0.5779]])
```

```
In [52]: centered_cols = mtx - mean_col.reshape((4,1))
         centered_rows

Out[52]: array([[ 0.2732,  1.8539],
                [-0.2367, -1.5562],
                [ 0.2608, -1.0344],
                [-0.2974,  0.7367]])

In [53]: centered_cols.mean(axis=1)

Out[53]: array([-0.,  0.,  0., -0.])
```

A note about NANs: Per the floating point standard IEEE 754, NaN is a floating point value that, by definition, is not equal to any other floating point value.

```
In [54]: np.nan != np.nan

Out[54]: True

In [55]: np.array([10, 5, 4, np.nan, 1, np.nan]) == np.nan

Out[55]: array([False, False, False, False, False, False], dtype=bool)

In [56]: np.isnan(np.array([10, 5, 4, np.nan, 1, np.nan]))

Out[56]: array([False, False, False,  True, False,  True], dtype=bool)
```

–> Go to “Numpy Questions: Operations”

1.8 pandas: Python Data Analysis Library

1.8.1 What is it?

Python has long been great for data munging and preparation, but less so for data analysis and modeling. pandas helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R.

The heart of pandas is the DataFrame object for data manipulation. It features:

- a powerful index object
- data alignment
- handling of missing data
- aggregation with groupby
- data manipuation via reshape, pivot, slice, merge, join

```
In [57]: import pandas as pd

         pd.set_option('precision', 3, 'notebook_repr_html', True, )
```

1.8.2 Series: labelled arrays

The pandas Series is kind of like an ndarray (used to actually be a subclass of it) that supports more meaninful indices.

Let's look at some creation examples for Series

```
In [58]: values = np.array([2.0, 1.0, 5.0, 0.97, 3.0, 10.0, 0.0599, 8.0])
         ser = pd.Series(values)
         ser
```

```
Out[58]: 0     2.00
         1     1.00
         2     5.00
         3     0.97
         4     3.00
         5    10.00
         6     0.06
         7     8.00
         dtype: float64
```

```
In [59]: values = np.array([2.0, 1.0, 5.0, 0.97, 3.0, 10.0, 0.0599, 8.0])
         labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
         ser = pd.Series(data=values, index=labels)
         ser
```

```
Out[59]: A     2.00
         B     1.00
         C     5.00
         D     0.97
         E     3.00
         F    10.00
         G     0.06
         H     8.00
         dtype: float64
```

```
In [60]: movie_rating = {
             'age': 1,
             'gender': 'F',
             'genres': 'Drama',
             'movie_id': 1193,
             'occupation': 10,
             'rating': 5,
             'timestamp': 978300760,
             'title': "One Flew Over the Cuckoo's Nest (1975)",
             'user_id': 1,
             'zip': '48067'
             }
         ser = pd.Series(movie_rating)
         print ser
```

```
age                                              1
gender                                           F
genres                                       Drama
movie_id                                      1193
occupation                                      10
rating                                           5
timestamp                                978300760
title         One Flew Over the Cuckoo's Nest (1975)
user_id                                          1
zip                                          48067
dtype: object
```

```
In [61]: ser.index

Out[61]: Index([u'age', u'gender', u'genres', u'movie_id', u'occupation',
                u'rating', u'timestamp', u'title', u'user_id', u'zip'], dtype='object')

In [62]: ser.values

Out[62]: array([1, 'F', 'Drama', ..., "One Flew Over the Cuckoo's Nest (1975)", 1,
                '48067'], dtype=object)
```

Series indexing

```
In [63]: ser.loc['gender']

Out[63]: 'F'

In [64]: ser.loc[['gender', 'zip']]

Out[64]: gender        F
         zip       48067
         dtype: object

In [65]: bool_arr = np.array([False, False, False, False, False, True, False, False, False, False])
         bool_arr

Out[65]: array([False, False, False, ..., False, False, False], dtype=bool)

In [66]: ser.loc[bool_arr]

Out[66]: rating    5
         dtype: object

In [67]: ser.iloc[1]

Out[67]: 'F'

In [68]: ser.iloc[[1,2]]

Out[68]: gender        F
         genres    Drama
         dtype: object

In [69]: ser.ix['gender']

Out[69]: 'F'

In [70]: ser.ix[1]

Out[70]: 'F'

In [71]: ser['gender']

Out[71]: 'F'

In [72]: ser[[1,2]]

Out[72]: gender        F
         genres    Drama
         dtype: object
```

Operations between Series with different index objects

```
In [73]: ser_1 = pd.Series(data=[1,3,4], index=['A', 'B', 'C'])
         ser_2 = pd.Series(data=[5,5,5], index=['A', 'G', 'C'])
         print ser_1 + ser_2
```

```
A     6
B   NaN
C     9
G   NaN
dtype: float64
```

Automatic upcasting when performing operations between Series with different dtypes:

```
In [74]: ser_1 = pd.Series(data=[1,3,4], index=['A', 'B', 'C'], dtype=np.int32)
         ser_2 = pd.Series(data=[5,5,5], index=['A', 'G', 'C'], dtype=np.float64)
         ser_1 + ser_2
```

```
Out[74]: A     6
         B   NaN
         C     9
         G   NaN
         dtype: float64
```

1.8.3 DataFrame

The DataFrame is the 2-dimensional version of a Series, you can think of it as a spreadsheet whose columns are Series objects.

```
In [75]: # build from a dict of equal-length lists or ndarrays
         pd.DataFrame({'col_1': [0.12, 7, 45, 10], 'col_2': [0.9, 9, 34, 11]})
```

```
Out[75]:    col_1  col_2
         0   0.12    0.9
         1   7.00    9.0
         2  45.00   34.0
         3  10.00   11.0

         [4 rows x 2 columns]
```

You can explicitly set the column names and index values as well.

```
In [76]: pd.DataFrame(data={'col_1': [0.12, 7, 45, 10], 'col_2': [0.9, 9, 34, 11]},
                      columns=['col_1', 'col_2', 'col_3'])
```

```
Out[76]:    col_1  col_2 col_3
         0   0.12    0.9   NaN
         1   7.00    9.0   NaN
         2  45.00   34.0   NaN
         3  10.00   11.0   NaN

         [4 rows x 3 columns]
```

```
In [77]: pd.DataFrame(data={'col_1': [0.12, 7, 45, 10], 'col_2': [0.9, 9, 34, 11]},
                      columns=['col_1', 'col_2', 'col_3'],
                      index=['obs1', 'obs2', 'obs3', 'obs4'])
```

```
Out[77]:       col_1  col_2 col_3
        obs1    0.12    0.9   NaN
        obs2    7.00    9.0   NaN
        obs3   45.00   34.0   NaN
        obs4   10.00   11.0   NaN

        [4 rows x 3 columns]
```

You can also think of it as a dictionary of Series objects.

```
In [78]: movie_rating = {
             'gender': 'F',
             'genres': 'Drama',
             'movie_id': 1193,
             'rating': 5,
             'timestamp': 978300760,
             'user_id': 1,
             }
         ser_1 = pd.Series(movie_rating)
         ser_2 = pd.Series(movie_rating)
         df = pd.DataFrame({'r_1': ser_1, 'r_2': ser_2})
         df.columns.name = 'rating_events'
         df.index.name = 'rating_data'
         df
```

```
Out[78]: rating_events        r_1        r_2
         rating_data
         gender                 F          F
         genres             Drama      Drama
         movie_id            1193       1193
         rating                 5          5
         timestamp      978300760  978300760
         user_id                1          1

         [6 rows x 2 columns]
```

```
In [79]: df = df.T
         df
```

```
Out[79]: rating_data   gender genres movie_id rating  timestamp user_id
         rating_events
         r_1                F  Drama     1193      5  978300760       1
         r_2                F  Drama     1193      5  978300760       1

         [2 rows x 6 columns]
```

```
In [80]: df.columns
```

```
Out[80]: Index([u'gender', u'genres', u'movie_id',
                          u'rating', u'timestamp', u'user_id'], dtype='object')
```

```
In [81]: df.index
```

```
Out[81]: Index([u'r_1', u'r_2'], dtype='object')
```

```
In [82]: df.values
```

```
Out[82]: array([['F', 'Drama', 1193, 5, 978300760, 1],
                ['F', 'Drama', 1193, 5, 978300760, 1]], dtype=object)
```

Adding/Deleting entries

```
In [83]: df = pd.DataFrame({'r_1': ser_1, 'r_2': ser_2})
         df.drop('genres', axis=0)

Out[83]:                      r_1        r_2
         rating_data
         gender                 F          F
         movie_id            1193       1193
         rating                 5          5
         timestamp      978300760  978300760
         user_id                1          1

         [5 rows x 2 columns]

In [84]: df.drop('r_1', axis=1)

Out[84]:                      r_2
         rating_data
         gender                 F
         genres             Drama
         movie_id            1193
         rating                 5
         timestamp      978300760
         user_id                1

         [6 rows x 1 columns]
```

You can also delete in place with del:

```
In [85]: del df['r_2']
         df

Out[85]:                      r_1
         rating_data
         gender                 F
         genres             Drama
         movie_id            1193
         rating                 5
         timestamp      978300760
         user_id                1

         [6 rows x 1 columns]

In [86]: # careful with the order here
         df['r_3'] = ['F', 'Drama', 1193, 5, 978300760, 1]
         df

Out[86]:                      r_1        r_3
         rating_data
         gender                 F          F
         genres             Drama      Drama
         movie_id            1193       1193
         rating                 5          5
         timestamp      978300760  978300760
         user_id                1          1

         [6 rows x 2 columns]
```

```
In [87]: df['r_3'] = pd.Series({'gender': 'F'})
         df
```

```
Out[87]:                      r_1  r_3
         rating_data
         gender                 F    F
         genres             Drama  NaN
         movie_id            1193  NaN
         rating                 5  NaN
         timestamp      978300760  NaN
         user_id                1  NaN

         [6 rows x 2 columns]
```

–> Go to "Pandas questions: Series and DataFrames"

DataFrame indexing You can index into a column using it's label, or with dot notation

```
In [88]: df = pd.DataFrame(data={'col_1': [0.12, 7, 45, 10], 'col_2': [0.9, 9, 34, 11]},
                           columns=['col_1', 'col_2', 'col_3'],
                           index=['obs1', 'obs2', 'obs3', 'obs4'])
         df
```

```
Out[88]:        col_1  col_2 col_3
         obs1    0.12    0.9   NaN
         obs2    7.00    9.0   NaN
         obs3   45.00   34.0   NaN
         obs4   10.00   11.0   NaN

         [4 rows x 3 columns]
```

```
In [89]: df['col_1']
```

```
Out[89]: obs1     0.12
         obs2     7.00
         obs3    45.00
         obs4    10.00
         Name: col_1, dtype: float64
```

```
In [90]: df.col_1
```

```
Out[90]: obs1     0.12
         obs2     7.00
         obs3    45.00
         obs4    10.00
         Name: col_1, dtype: float64
```

You can also use multiple columns to select a subset of them:

```
In [91]: df[['col_2', 'col_1']]
```

```
Out[91]:        col_2  col_1
         obs1     0.9   0.12
         obs2     9.0   7.00
         obs3    34.0  45.00
         obs4    11.0  10.00

         [4 rows x 2 columns]
```

DataFrame has similar .loc and .iloc methods:

```
In [92]: df.loc['obs1', 'col_1']

Out[92]: 0.12

In [93]: df.iloc[0, 0]

Out[93]: 0.12
```

The .ix method gives you the most flexibility to index into certain rows, or even rows and columns:

```
In [94]: df.ix['obs3']

Out[94]: col_1     45
         col_2     34
         col_3    NaN
         Name: obs3, dtype: object

In [95]: df.ix[0]

Out[95]: col_1    0.12
         col_2     0.9
         col_3     NaN
         Name: obs1, dtype: object

In [96]: df.ix[:2]

Out[96]:       col_1  col_2 col_3
         obs1   0.12    0.9   NaN
         obs2   7.00    9.0   NaN

         [2 rows x 3 columns]

In [97]: df.ix[:2, 'col_2']

Out[97]: obs1    0.9
         obs2    9.0
         Name: col_2, dtype: float64

In [98]: df.ix[:2, ['col_1', 'col_2']]

Out[98]:       col_1  col_2
         obs1   0.12    0.9
         obs2   7.00    9.0

         [2 rows x 2 columns]
```

–> Go to "Pandas questions: Indexing"
–> 20 min break!!

1.9 The MovieLens dataset: loading and first look

Loading of the MovieLens dataset is based on the intro chapter of "Python for Data Analysis". The MovieLens data is spread across three files. We'll load each file using the `pd.read_table` function:

```
In [99]: users = pd.read_table('data/ml-1m/users.dat',
                               sep='::', header=None,
                               names=['user_id', 'gender', 'age', 'occupation', 'zip'])

         ratings = pd.read_table('data/ml-1m/ratings.dat',
                                 sep='::', header=None,
                                 names=['user_id', 'movie_id', 'rating', 'timestamp'])

         movies = pd.read_table('data/ml-1m/movies.dat',
                                sep='::', header=None,
                                names=['movie_id', 'title', 'genres'])

         # show how one of them looks
         ratings.head(5)
```

```
Out[99]:    user_id  movie_id  rating  timestamp
         0        1      1193       5  978300760
         1        1       661       3  978302109
         2        1       914       3  978301968
         3        1      3408       4  978300275
         4        1      2355       5  978824291

         [5 rows x 4 columns]
```

Using `pd.merge` we get it all into one big DataFrame.

```
In [100]: movielens = pd.merge(pd.merge(ratings, users), movies)
          movielens.head()
```

```
Out[100]:    user_id  movie_id  rating  timestamp gender  age  occupation    zip  \
          0        1      1193       5  978300760      F    1          10  48067
          1        2      1193       5  978298413      M   56          16  70072
          2       12      1193       4  978220179      M   25          12  32793
          3       15      1193       4  978199279      M   25           7  22903
          4       17      1193       5  978158471      M   50           1  95350

                                             title genres
          0  One Flew Over the Cuckoo's Nest (1975)  Drama
          1  One Flew Over the Cuckoo's Nest (1975)  Drama
          2  One Flew Over the Cuckoo's Nest (1975)  Drama
          3  One Flew Over the Cuckoo's Nest (1975)  Drama
          4  One Flew Over the Cuckoo's Nest (1975)  Drama

          [5 rows x 10 columns]
```

1.10 Evaluation

Before we start building our minimal reco engine we need a basic mechanism to evaluate the performance of our engine. For that we will:

- split the data into train and test sets
- introduce a performance criterion
- write an `evaluate` function.

1.10.1 Evaluation: split ratings into train and test sets

This subsection will generate training and testing sets for evaluation. You do not need to understand every single line of code, just the general gist:

- take a smaller sample from the full 1M dataset for speed reasons;
- make sure that we have at least 2 ratings per user in that subset;
- split the result into training and testing sets.

```
In [101]: # let's work with a smaller subset for speed reasons
          movielens = movielens.ix[np.random.choice(movielens.index, size=10000, replace=False)]
          print movielens.shape
          print movielens.user_id.nunique()
          print movielens.movie_id.nunique()
```

```
(10000, 10)
3677
2279
```

```
In [102]: user_ids_larger_1 = pd.value_counts(movielens.user_id, sort=False) > 1
          user_ids_larger_1 = user_ids_larger_1[user_ids_larger_1].index

          movielens = movielens.select(lambda l: movielens.loc[l, 'user_id'] in user_ids_larger_1)
          print movielens.shape
          assert np.all(movielens.user_id.value_counts() > 1)
```

```
(8512, 10)
```

We now generate train and test subsets by marking 20% of each users's ratings, using groupby and apply.

```
In [103]: def assign_to_set(df):
              sampled_ids = np.random.choice(df.index,
                                             size=np.int64(np.ceil(df.index.size * 0.2)),
                                             replace=False)
              df.ix[sampled_ids, 'for_testing'] = True
              return df

          movielens['for_testing'] = False
          grouped = movielens.groupby('user_id', group_keys=False).apply(assign_to_set)
          movielens_train = movielens[grouped.for_testing == False]
          movielens_test = movielens[grouped.for_testing == True]
          print movielens.shape
          print movielens_train.shape
          print movielens_test.shape
          assert len(movielens_train.index & movielens_test.index) == 0
```

```
(8512, 11)
(5845, 11)
(2667, 11)
```

Store these two sets in text files:

```
In [104]: movielens_train.to_csv('data/my_generated_movielens_train.csv')
          movielens_test.to_csv('data/my_generated_movielens_test.csv')
```

1.10.2 Evaluation: performance criterion

Performance evaluation of recommendation systems is an entire topic all in itself. Some of the options include:

- RMSE: $\sqrt{\frac{\sum(\hat{y}-y)^2}{n}}$
- Precision / Recall / F-scores
- ROC curves
- Cost curves

```
In [105]: def compute_rmse(y_pred, y_true):
              """ Compute Root Mean Squared Error. """

              return np.sqrt(np.mean(np.power(y_pred - y_true, 2)))
```

1.10.3 Evaluation: the 'evaluate' method

```
In [106]: def evaluate(estimate_f):
              """ RMSE-based predictive performance evaluation with pandas. """

              ids_to_estimate = zip(movielens_test.user_id, movielens_test.movie_id)
              estimated = np.array([estimate_f(u,i) for (u,i) in ids_to_estimate])
              real = movielens_test.rating.values
              return compute_rmse(estimated, real)
```

1.11 Minimal reco engine v1.0: simple mean ratings

1.11.1 Content-based filtering using mean ratings

With this table-like representation of the ratings data, a basic content-based filter becomes a one-liner function.

```
In [107]: def estimate1(user_id, movie_id):
              """ Simple content-filtering based on mean ratings. """

              user_condition = movielens_train.user_id == user_id
              return movielens_train.loc[user_condition, 'rating'].mean()

          print 'RMSE for estimate1: %s' % evaluate(estimate1)
```

```
RMSE for estimate1: 1.2459605121
```

1.11.2 Collaborative-based filtering using mean ratings

```
In [108]: def estimate2(user_id, movie_id):
              """ Simple collaborative filter based on mean ratings. """

              user_condition = movielens_train.user_id != user_id
              movie_condition = movielens_train.movie_id == movie_id
              ratings_by_others = movielens_train.loc[user_condition & movie_condition]
              if ratings_by_others.empty:
                  return 3.0
              else:
                  return ratings_by_others.rating.mean()

          print 'RMSE for estimate2: %s' % evaluate(estimate2)
```

```
RMSE for estimate2: 1.13464798007
```

–> Go to “Reco systems questions: Data Loading + Estimation Functions”

1.12 More formulas!

Here are some basic ways in which we can generalize the simple mean-based algorithms we discussed before.

1.12.1 Generalizations of the aggregation function for content-based filtering: incorporating similarities

Possibly incorporating metadata about items, which makes the term ‘content’ make more sense now.

$$r_{u,i} = k \sum_{i' \in I(u)} sim(i, i')\; r_{u,i'}$$

$$r_{u,i} = \bar{r}_u + k \sum_{i' \in I(u)} sim(i, i')\; (r_{u,i'} - \bar{r}_u)$$

Here k is a normalizing factor,

$$k = \frac{1}{\sum_{i' \in I(u)} |sim(i, i')|}$$

and $\bar{r}_u$ is the average rating of user u:

$$\bar{r}_u = \frac{\sum_{i \in I(u)} r_{u,i}}{|I(u)|}$$

1.12.2 Generalizations of the aggregation function for collaborative filtering: incorporating similarities

Possibly incorporating metadata about users.

$$r_{u,i} = k \sum_{u' \in U(i)} sim(u, u')\; r_{u',i}$$

$$r_{u,i} = \bar{r}_u + k \sum_{u' \in U(i)} sim(u, u')\; (r_{u',i} - \bar{r}_u)$$

Here k is a normalizing factor,

$$k = \frac{1}{\sum_{u' \in U(i)} |sim(u, u')|}$$

and $\bar{r}_u$ is the average rating of user u:

$$\bar{r}_u = \frac{\sum_{i \in I(u)} r_{u,i}}{|I(u)|}$$

1.13 Aggregation in pandas

1.13.1 Groupby

The idea of groupby is that of *split-apply-combine*:

- split data in an object according to a given key;
- apply a function to each subset;
- combine results into a new object.

```
In [109]: movielens_train.groupby('gender')['rating'].mean()
```

```
Out[109]: gender
          F         3.58
          M         3.53
          Name: rating, dtype: float64
```

```
In [110]: movielens_train.groupby(['gender', 'age'])['rating'].mean()
```

```
Out[110]: gender  age
          F       1      3.54
                  18     3.46
                  25     3.59
                  35     3.63
                  45     3.47
                  50     3.75
                  56     3.74
          M       1      3.33
                  18     3.50
                  25     3.46
                  35     3.60
                  45     3.56
                  50     3.71
                  56     3.88
          Name: rating, dtype: float64
```

1.13.2 Pivoting

Let's start with a simple pivoting example that does not involve any aggregation. We can extract a ratings matrix as follows:

```
In [111]: # transform the ratings frame into a ratings matrix
          ratings_mtx_df = movielens_train.pivot_table(values='rating',
                                                       rows='user_id',
                                                       cols='movie_id')
          ratings_mtx_df.head(3)
```

```
Out[111]: movie_id  1    2    3    4    5    6    8    10   11   16   17   18   19   20   21   22   23  \
          user_id
          3         NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
          5         NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
          6         NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN

          movie_id  24   25   26
          user_id
          3         NaN  NaN  NaN ...
          5         NaN  NaN  NaN ...
          6         NaN  NaN  NaN ...

          [3 rows x 1967 columns]
```

```
In [193]: # grab another subsquare of the ratings matrix to actually diplay some real entries!
          ratings_mtx_df.loc[3:11, 1196:1200]
```

```
Out[193]: movie_id  1196  1197  1198  1199  1200
          user_id
```

```
3           NaN     5   NaN   NaN   NaN
5           NaN   NaN   NaN   NaN   NaN
6           NaN   NaN   NaN   NaN   NaN
10          NaN   NaN   NaN   NaN   NaN
11          NaN   NaN   NaN   NaN   NaN

[5 rows x 5 columns]
```

The more interesting case with `pivot_table` is as an interface to `groupby`:

```
In [194]: movielens_train.pivot_table(values='rating', rows='age', cols='gender', aggfunc='mean')

Out[194]: gender     F     M
          age
          1       3.54  3.33
          18      3.46  3.50
          25      3.59  3.46
          35      3.63  3.60
          45      3.47  3.56
          50      3.75  3.71
          56      3.74  3.88

          [7 rows x 2 columns]
```

You can pass in a list of functions, such as `[np.mean, np.std]`, to compute mean ratings and a measure of disagreement.

```
In [195]: movielens_train.pivot_table(values='rating', rows='age', cols='gender', aggfunc=[np.mean, np.s

Out[195]:         mean        std
          gender     F     M     F     M
          age
          1       3.54  3.33  1.30  1.24
          18      3.46  3.50  1.23  1.16
          25      3.59  3.46  1.11  1.13
          35      3.63  3.60  1.04  1.04
          45      3.47  3.56  1.11  1.04
          50      3.75  3.71  1.05  1.07
          56      3.74  3.88  1.06  1.06

          [7 rows x 4 columns]
```

1.14 Minimal reco engine v1.1: implicit sim functions

We're going to need a user index from the users portion of the dataset. This will allow us to retrieve information given a specific user_id in a more convenient way:

```
In [196]: user_info = users.set_index('user_id')
          user_info.head(5)

Out[196]:        gender  age  occupation    zip
          user_id
          1           F    1          10  48067
          2           M   56          16  70072
          3           M   25          15  55117
          4           M   45           7  02460
```

```
5            M   25         20  55455

[5 rows x 4 columns]
```

With this in hand, we can now ask what the gender of a particular user_id is like so:

```
In [197]: user_id = 3
          user_info.loc[user_id, 'gender']

Out[197]: 'M'
```

1.14.1 Collaborative-based filtering using implicit sim functions

Using the pandas aggregation framework we will build a collaborative filter that estimates ratings using an implicit `sim(u,u')` function to compare different users.

```
In [198]: def estimate3(user_id, movie_id):
              """ Collaborative filtering using an implicit sim(u,u'). """

              user_condition = movielens_train.user_id != user_id
              movie_condition = movielens_train.movie_id == movie_id
              ratings_by_others = movielens_train.loc[user_condition & movie_condition]
              if ratings_by_others.empty:
                  return 3.0

              means_by_gender = ratings_by_others.pivot_table('rating', rows='movie_id', cols='gender')
              user_gender = user_info.ix[user_id, 'gender']
              if user_gender in means_by_gender.columns:
                  return means_by_gender.ix[movie_id, user_gender]
              else:
                  return means_by_gender.ix[movie_id].mean()

          print 'RMSE for reco3: %s' % evaluate(estimate3)

RMSE for reco3: 1.20272358698
```

At this point it seems worthwhile to write a `learn` function to pre-compute whatever datastructures we need at estimation time.

```
In [199]: class Reco3:
              """ Collaborative filtering using an implicit sim(u,u'). """

              def learn(self):
                  """ Prepare datastructures for estimation. """

                  self.means_by_gender = movielens_train.pivot_table('rating', rows='movie_id', cols='ge

              def estimate(self, user_id, movie_id):
                  """ Mean ratings by other users of the same gender. """

                  if movie_id not in self.means_by_gender.index:
                      return 3.0

                  user_gender = user_info.ix[user_id, 'gender']
                  if ~np.isnan(self.means_by_gender.ix[movie_id, user_gender]):
                      return self.means_by_gender.ix[movie_id, user_gender]
```

```
            else:
                return self.means_by_gender.ix[movie_id].mean()

    reco = Reco3()
    reco.learn()
    print 'RMSE for reco3: %s' % evaluate(reco.estimate)
```

```
RMSE for reco3: 1.20272358698
```

```
In [200]: class Reco4:
              """ Collaborative filtering using an implicit sim(u,u'). """

              def learn(self):
                  """ Prepare datastructures for estimation. """

                  self.means_by_age = movielens_train.pivot_table('rating', rows='movie_id', cols='age')

              def estimate(self, user_id, movie_id):
                  """ Mean ratings by other users of the same age. """

                  if movie_id not in self.means_by_age.index:
                      return 3.0

                  user_age = user_info.ix[user_id, 'age']
                  if ~np.isnan(self.means_by_age.ix[movie_id, user_age]):
                      return self.means_by_age.ix[movie_id, user_age]
                  else:
                      return self.means_by_age.ix[movie_id].mean()

          reco = Reco4()
          reco.learn()
          print 'RMSE for reco4: %s' % evaluate(reco.estimate)
```

```
RMSE for reco4: 1.21311405537
```

1.15 Mini-Challenge!

- Not a real challenge
- Focus on understanding the different versions of our minimal reco
- Try to mix and match some of the ideas presented to come up with a minimal reco of your own
- Evaluate it!

1.15.1 Mini-Challenge: first round

Implement an `estimate` function of your own using other similarity notions, eg.:

- zip code
- movie genre
- occupation

1.16 Minimal reco engine v1.2: custom similarity functions

1.16.1 A few similarity functions

These were all written to operate on two pandas Series, each one representing the rating history of two different users. You can also apply them to any two feature vectors that describe users or items. In all cases, the higher the return value, the more similar two Series are. You might need to add checks for edge cases, such as divisions by zero, etc.

- Euclidean 'similarity'

$$sim(x, y) = \frac{1}{1 + \sqrt{\sum (x - y)^2}}$$

```
In [201]: def euclidean(s1, s2):
              """Take two pd.Series objects and return their euclidean 'similarity'."""
              diff = s1 - s2
              return 1 / (1 + np.sqrt(np.sum(diff ** 2)))
```

- Cosine similarity

$$sim(x, y) = \frac{(x.y)}{\sqrt{(x.x)(y.y)}}$$

```
In [202]: def cosine(s1, s2):
              """Take two pd.Series objects and return their cosine similarity."""
              return np.sum(s1 * s2) / np.sqrt(np.sum(s1 ** 2) * np.sum(s2 ** 2))
```

- Pearson correlation

$$sim(x, y) = \frac{(x - \bar{x}).(y - \bar{y})}{\sqrt{(x - \bar{x}).(x - \bar{x}) * (y - \bar{y})(y - \bar{y})}}$$

```
In [203]: def pearson(s1, s2):
              """Take two pd.Series objects and return a pearson correlation."""
              s1_c = s1 - s1.mean()
              s2_c = s2 - s2.mean()
              return np.sum(s1_c * s2_c) / np.sqrt(np.sum(s1_c ** 2) * np.sum(s2_c ** 2))
```

- Jaccard similarity

$$sim(x, y) = \frac{(x.y)}{(x.x) + (y.y) - (x.y)}$$

```
In [204]: def jaccard(s1, s2):
              dotp = np.sum(s1 * s2)
              return dotp / (np.sum(s1 ** 2) + np.sum(s2 ** 2) - dotp)

          def binjaccard(s1, s2):
              dotp = (s1.index & s2.index).size
              return dotp / (s1.sum() + s2.sum() - dotp)
```

1.16.2 Collaborative-based filtering using custom sim functions

```
In [205]: class Reco5:
              """ Collaborative filtering using a custom sim(u,u'). """

              def learn(self):
                  """ Prepare datastructures for estimation. """
```

```
            self.all_user_profiles = movielens.pivot_table('rating', rows='movie_id', cols='user_

        def estimate(self, user_id, movie_id):
            """ Ratings weighted by correlation similarity. """

            user_condition = movielens_train.user_id != user_id
            movie_condition = movielens_train.movie_id == movie_id
            ratings_by_others = movielens_train.loc[user_condition & movie_condition]
            if ratings_by_others.empty:
                return 3.0

            ratings_by_others.set_index('user_id', inplace=True)
            their_ids = ratings_by_others.index
            their_ratings = ratings_by_others.rating
            their_profiles = self.all_user_profiles[their_ids]
            user_profile = self.all_user_profiles[user_id]
            sims = their_profiles.apply(lambda profile: pearson(profile, user_profile), axis=0)
            ratings_sims = pd.DataFrame({'sim': sims, 'rating': their_ratings})
            ratings_sims = ratings_sims[ratings_sims.sim > 0]
            if ratings_sims.empty:
                return their_ratings.mean()
            else:
                return np.average(ratings_sims.rating, weights=ratings_sims.sim)

    reco = Reco5()
    reco.learn()
    print 'RMSE for reco5: %s' % evaluate(reco.estimate)
```

```
RMSE for reco5: 1.08702256721
```

1.16.3 Mini-Challenge: second round

Implement an `estimate` function of your own using other custom similarity notions, eg.:

- euclidean
- cosine

1.17 [BONUS] PyTables

1.17.1 What is it?

PyTables is a package for managing hierarchical datasets and designed to efficiently and easily cope with extremely large amounts of data.

1.17.2 HDF5

From hdfgroup.org: *HDF5 is a Hierarchical Data Format consisting of a data format specification and a supporting library implementation.*

HDF5 files are organized in a hierarchical structure, with two primary structures: groups and datasets.

- HDF5 group: a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
- HDF5 dataset: a multidimensional array of data elements, together with supporting metadata.

1.17.3 Pandas + PyTables Integration

In [206]:
```
import tables as tb

store = pd.HDFStore('data/store.h5')
store
```

Out[206]:
```
<class 'pandas.io.pytables.HDFStore'>
File path: data/store.h5
/pycon2014/movielens            frame_table  (typ->appendable,nrows->8512,ncols->11,indexers->
```

In [207]:
```
store.put('/pycon2014/movielens', movielens, format='table', data_columns=True)
store
```

Out[207]:
```
<class 'pandas.io.pytables.HDFStore'>
File path: data/store.h5
/pycon2014/movielens            frame_table  (typ->appendable,nrows->8512,ncols->11,indexers->
```

In [208]:
```
store.select('/pycon2014/movielens', "columns=['user_id', 'rating']", start=0, stop=5)
```

Out[208]:
```
        user_id  rating
242166     3560       5
327994     3196       3
768999     3574       4
14121      1744       4
344208     2105       2

[5 rows x 2 columns]
```

In [209]:
```
store.select('/pycon2014/movielens', where=u'rating>4', start=0, stop=20)
```

Out[209]:
```
        user_id  movie_id  rating  timestamp gender  age  occupation    zip  \
242166     3560       296       5  966796358      F   25           6  74105
624536     5283      2761       5  961166145      M   18           2  63138
512865     5767      1200       5  958176105      M   25           2  75287

                          title                    genres for_testing
242166      Pulp Fiction (1994)               Crime|Drama       False
624536  Iron Giant, The (1999)       Animation|Children's       False
512865           Aliens (1986)  Action|Sci-Fi|Thriller|War       False

[3 rows x 11 columns]
```

In [212]:
```
store.close()
```

1.17.4 Direct File Manipulation: Node Attributes Example

In [213]:
```
import tables as tb
from datetime import datetime

hdf_file = tb.open_file('data/store.h5', 'r+')
node = hdf_file.getNode('/pycon2014/movielens')
hdf_file.setNodeAttr(node, 'last_modified', datetime.utcnow())
node._v_attrs
```

Out[213]:
```
/pycon2014/movielens._v_attrs (AttributeSet), 15 attributes:
   [CLASS := 'GROUP',
```

```
          TITLE := '',
          VERSION := '1.0',
          data_columns := ['user_id', 'movie_id', 'rating', 'timestamp',
                           'gender', 'age', 'occupation', 'zip', 'title', 'genres', 'for_testing'],
          encoding := None,
          index_cols := [(0, 'index')],
          info := {1: {'type': 'Index', 'names': [None]}, 'index': {}},
          last_modified := datetime.datetime(2014, 4, 7, 20, 50, 43, 588596),
          levels := 1,
          nan_rep := 'nan',
          non_index_axes := [(1, ['user_id', 'movie_id', 'rating', 'timestamp',
                                  'gender', 'age', 'occupation', 'zip', 'title', 'genres', 'for_testing'])],
          pandas_type := 'frame_table',
          pandas_version := '0.10.1',
          table_type := 'appendable_frame',
          values_cols := ['user_id', 'movie_id', 'rating', 'timestamp',
                          'gender', 'age', 'occupation', 'zip', 'title', 'genres', 'for_testing']]
```

```
In [214]: hdf_file.getNodeAttr(node,'last_modified')
```

```
Out[214]: datetime.datetime(2014, 4, 7, 20, 50, 43, 588596)
```

1.17.5 Handling things that don't fit in memory

```
In [ ]: group_3 = h5file.createGroup(h5file.root, 'group_3', 'Group Three')
        ndim = 6000000
        h5file.createArray(group_3, 'random_group_3',
                           numpy.zeros((ndim,ndim)), "A very very large array")
```

```
In [ ]: rows = 10
        cols = 10
        earr = h5file.createEArray(group_3, 'EArray', tb.Int8Atom(),
                                   (0, cols), "A very very large array, second try.")

        for i in range(rows):
            earr.append(numpy.zeros((1, cols)))
```

```
In [ ]: earr
```

2 References and further reading

1. Goldberg, D., D. Nichols, B. M. Oki, and D. Terry. “Using Collaborative Filtering to Weave an Information Tapestry.” Communications of the ACM 35, no. 12 (1992): 61–70.
2. Resnick, Paul, and Hal R. Varian. “Recommender Systems.” Commun. ACM 40, no. 3 (March 1997): 56–58. doi:10.1145/245108.245121.
3. Adomavicius, Gediminas, and Alexander Tuzhilin. “Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions.” IEEE Transactions on Knowledge and Data Engineering 17, no. 6 (2005): 734–749. doi:http://doi.ieeecomputersociety.org/10.1109/TKDE.2005.99.
4. Adomavicius, Gediminas, Ramesh Sankaranarayanan, Shahana Sen, and Alexander Tuzhilin. “Incorporating Contextual Information in Recommender Systems Using a Multidimensional Approach.” ACM Trans. Inf. Syst. 23, no. 1 (2005): 103–145. doi:10.1145/1055709.1055714.
5. Koren, Y., R. Bell, and C. Volinsky. “Matrix Factorization Techniques for Recommender Systems.” Computer 42, no. 8 (2009): 30–37.
6. William Wesley McKinney. Python for Data Analysis. O’Reilly, 2012.
7. Toby Segaran. Programming Collective Intelligence. O’Reilly, 2007.
8. Zhou, Tao, Zoltan Kuscsik, Jian-Guo Liu, Matus Medo, Joseph R Wakeling, and Yi-Cheng Zhang. “Solving the Apparent Diversity-accuracy Dilemma of Recommender Systems.” arXiv:0808.2670 (August 19, 2008). doi:10.1073/pnas.1000488107.
9. Shani, G., D. Heckerman, and R. I Brafman. “An MDP-based Recommender System.” Journal of Machine Learning Research 6, no. 2 (2006): 1265.
10. Joseph A. Konstan, John Riedl. “Deconstructing Recommender Systems.” IEEE Spectrum, October 2012.