A Beginner's Introduction to Pydata: How to Build a Minimal Recommendation System

Welcome!

- About us
- Environment + data files check: https://us.pycon.org/2013/community/tutorials/28
- Ipydra check

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?

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

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 $U(i)$.

Goal of a recommendation system

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

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.

Content-based filtering

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

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

Content-based: simple ratings-based recommendations
Purely based on ratings information.

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

Collaborative filtering

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

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

Collaborative filtering: simple ratings-based recommendations
Also based solely on ratings information.

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

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.

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.

About this tutorial

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

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:

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

What this tutorial is not

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

Roadmap

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

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

Dataset

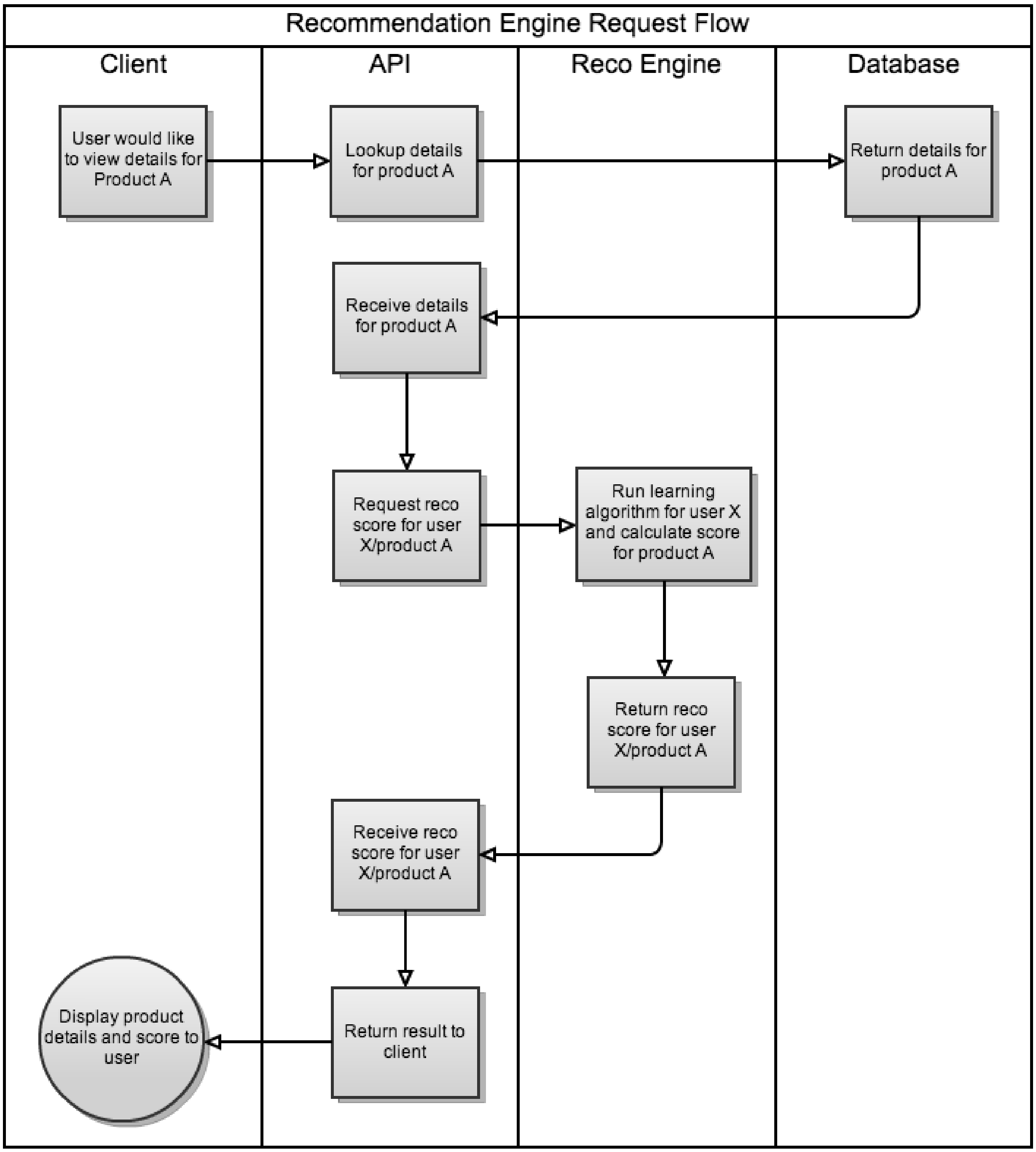
MovieLens from GroupLens Research: grouplens.org

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

Flow chart: the big picture

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

Out[1]:



NumPy: Numerical Python

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)
```

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])
```

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([-0., -0., -0.,  0.])
```

In [7]:
```
np.arange(4)
```

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

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, -2147483648,   613849754,  1073743870], 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)
```

Indexing and slicing

Just what you would expect from Python

In [14]:
```
arr = np.array([0, 9, 1.02, 4, 64])
arr[3]
```

Out[14]:
```
4.0
```

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

Out[15]:
```
array([ 9.  ,  1.02])
```

In [16]:
```
# set the last two elements to 555
arr[-2:] = 555
arr
```

Out[16]:
```
array([   0.  ,    9.  ,    1.02,  555.  ,  555.  ])
```

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 [17]:
```
arr_2d = np.array([[5,3,4],[0,1,2],[1,1,10],[0,0,0.1]])
arr_2d
```

Out[17]:
```
array([[  5. ,   3. ,   4. ],
       [  0. ,   1. ,   2. ],
       [  1. ,   1. ,  10. ],
       [  0. ,   0. ,   0.1]])
```

In [18]:
```
# get the first row
arr_2d[0]
```

Out[18]:
```
array([ 5.,  3.,  4.])
```

In [19]:
```
# get the first column
arr_2d[:,0]
```

Out[19]:
```
array([ 5.,  0.,  1.,  0.])
```

In [20]:
```
# get the first two rows
arr_2d[:2]
```

Out[20]:
```
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 [21]:
```
arr = np.array([0, 3, 1, 4, 64])
arr
```

Out[21]:
```
array([ 0,  3,  1,  4, 64])
```

In [22]:
```
slice = arr[2:4]
slice[1] = 99
arr
```

Out[22]:
```
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 [23]:
```
arr = np.array([10, 20])
idx = np.array([True, False])
arr[idx]
```

Out[23]:
```
array([10])
```

In [24]:
```
arr_2d = np.random.randn(4,8)
arr_2d
```

Out[24]:
```
array([[-0.8417,  0.5029, -1.2453, ...,  0.5515,  2.2922,  0.0415],
       [-1.1179,  0.5391, -0.5962, ..., -0.7479,  0.009 , -0.8781],
```

```
        [-0.1564,  0.2566, -0.9888, ..., -0.6377, -1.1876, -1.4212],
        [-0.1535, -0.2691,  2.2314, ...,  0.3704,  1.3596,  0.5019]])
```

In [25]:
```
arr_2d < 0
```

Out[25]:
```
array([[ True, False,  True, ..., False, False, False],
       [ True, False,  True, ...,  True, False,  True],
       [ True, False,  True, ...,  True,  True,  True],
       [ True,  True, False, ..., False, False, False]], dtype=bool)
```

In [26]:
```
arr_2d[arr_2d < 0]
```

Out[26]:
```
array([-0.8417, -1.2453, -1.058 , ..., -0.1535, -0.2691, -2.4348])
```

In [27]:
```
arr_2d[(arr_2d > -0.1) & (arr_2d < 0)]
```

Out[27]:
```
array([-0.0191])
```

In [28]:
```
arr_2d[arr_2d < 0] = 0
arr_2d
```

Out[28]:
```
array([[ 0.    ,  0.5029,  0.    , ...,  0.5515,  2.2922,  0.0415],
       [ 0.    ,  0.5391,  0.    , ...,  0.    ,  0.009 ,  0.    ],
       [ 0.    ,  0.2566,  0.    , ...,  0.    ,  0.    ,  0.    ],
       [ 0.    ,  0.    ,  2.2314, ...,  0.3704,  1.3596,  0.5019]])
```

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

In [29]:
```
arr = np.arange(18).reshape(6,3)
arr
```

Out[29]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [30]:
```
# fancy selection of rows in a particular order
arr[[0,4,4]]
```

Out[30]:
```
array([[ 0,  1,  2],
       [12, 13, 14],
       [12, 13, 14]])
```

In [31]:
```
# index into individual elements and flatten
arr[[5,3,1],[2,1,0]]
```

Out[31]:
```
array([17, 10,  3])
```

In [32]:
```
# select a submatrix
arr[np.ix_([5,3,1],[2,1])]
```

Out[32]:
```
array([[17, 16],
       [11, 10],
       [ 5,  4]])
```

--> Go to question set

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 [33]:
```
arr = np.array([0, 9, 1.02, 4, 32])
arr - arr
```

Out[33]:
```
array([ 0.,  0.,  0.,  0.,  0.])
```

In [34]:
```
arr * arr
```

Out[34]:
```
array([    0.    ,    81.    ,     1.0404,    16.    ,  1024.    ])
```

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 [35]:
```
arr = np.array([0, 9, 1.02, 4, 64])
5 * arr
```

Out[35]:
```
array([   0. ,   45. ,    5.1,   20. ,  320. ])
```

In [36]:
```
10 + arr
```

Out[36]:
```
array([ 10.  ,  19.  ,  11.02,  14.  ,  74.  ])
```

In [37]:
```
arr ** .5
```

Out[37]:
```
array([ 0.  ,  3.  ,  1.01,  2.  ,  8.  ])
```

In [38]:
```
arr = np.random.randn(4,2)
arr
```

Out[38]:
```
array([[-0.8442,  0.    ],
       [ 0.5424, -0.3135],
       [ 0.771 , -1.8681],
       [ 1.7312,  1.4677]])
```

In [39]:
```
mean_row = np.mean(arr, axis=0)
mean_row
```

Out[39]:
```
array([ 0.5501, -0.1785])
```

In [40]:
```
centered_rows = arr - mean_row
np.mean(centered_rows, axis=0)
```

Out[40]:
```
array([-0.,  0.])
```

In [41]:
```
mean_col = np.mean(arr, axis=1)
mean_col
```

Out[41]:
```
array([-0.4221,  0.1144, -0.5485,  1.5994])
```

In [42]:
```
centered_cols = arr - mean_col
```

```
---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-42-bd5236897883> in <module>()
----> 1 centered_cols = arr - mean_col

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

In [106]:
```
# make the 1-D array a column vector
mean_col.reshape((4,1))
```

Out[106]:
```
array([[-0.4221],
       [ 0.1144],
       [-0.5485],
       [ 1.5994]])
```

In [107]:
```
centered_cols = arr - mean_col.reshape((4,1))
centered_cols.mean(axis=1)
```

Out[107]:
```
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 [108]:
```
np.nan != np.nan
```

Out[108]:
```
True
```

In [109]:
```
np.array([10,6,5,4,np.nan,1,np.nan]) == np.nan
```

Out[109]:
```
array([False, False, False, ..., False, False, False], dtype=bool)
```

In [110]:
```
np.isnan(np.array([10,6,5,4,np.nan,1,np.nan]))
```

Out[110]:
```
array([False, False, False, ...,  True, False,  True], dtype=bool)
```

--> Go to question set

pandas: Python Data Analysis Library

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 [43]:
```
import pandas as pd

pd.set_printoptions(precision=3, notebook_repr_html=True)
```

```
/usr/local/Cellar/python/2.7.3/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-
```

```
packages/pandas/core/format.py:1286: FutureWarning: set_printoptions is deprecated, use
set_option instead
  FutureWarning)
```

Series: labelled arrays

The pandas Series is the simplest datastructure to start with. It is a subclass of ndarray that supports more meaninful indices.

Let's look at some creation examples for Series

In [44]:

```
import pandas as pd

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

```
0     2.00
1     1.00
2     5.00
3     0.97
4     3.00
5    10.00
6     0.06
7     8.00
```

In [45]:

```
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)
print ser
```

```
A     2.00
B     1.00
C     5.00
D     0.97
E     3.00
F    10.00
G     0.06
H     8.00
```

In [46]:

```
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
```

In [47]:
```
ser.index
```

Out[47]:
```
Index([age, gender, genres, ..., title, user_id, zip], dtype=object)
```

In [48]:
```
ser.values
```

Out[48]:
```
array([1, 'F', 'Drama', ..., "One Flew Over the Cuckoo's Nest (1975)", 1,
       '48067'], dtype=object)
```

Series indexing

In [49]:
```
ser[0]
```

Out[49]:
```
1
```

In [50]:
```
ser['gender']
```

Out[50]:
```
'F'
```

In [51]:
```
ser.get_value('gender')
```

Out[51]:
```
'F'
```

Operations between Series with different index objects

In [52]:
```
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
```

DataFrame

The DataFrame is the 2-dimensional version of a Series.
Let's look at some creation examples for DataFrame
You can think of it as a spreadsheet whose columns are Series objects.

In [53]:
```
# 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[53]:

	col_1	col_2
0	0.12	0.9
1	7.00	9.0
2	45.00	34.0
3	10.00	11.0

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

In [54]:
```
pd.DataFrame(data={'col_1': [0.12, 7, 45, 10], 'col_2': [0.9, 9, 34, 11]},
```

```
In [54]: 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[54]:

	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

```
In [55]: 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[55]:

	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

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

```
In [56]: 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[56]:

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

```
In [57]: df = df.T
         df
```

Out[57]:

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

In [58]:
```
df.columns
```

Out[58]:
```
Index([gender, genres, movie_id, rating, timestamp, user_id], dtype=object)
```

In [59]:
```
df.index
```

Out[59]:
```
Index([r_1, r_2], dtype=object)
```

Adding/Deleting entries

In [60]:
```
df = pd.DataFrame({'r_1': ser_1, 'r_2': ser_2})
df.drop('genres', axis=0)
```

Out[60]:

	r_1	r_2
gender	F	F
movie_id	1193	1193
rating	5	5
timestamp	978300760	978300760
user_id	1	1

In [61]:
```
df.drop('r_1', axis=1)
```

Out[61]:

	r_2
rating_data	
gender	F
genres	Drama
movie_id	1193
rating	5
timestamp	978300760
user_id	1

In [62]:
```
# careful with the order here
df['r_3'] = ['F', 'Drama', 1193, 5, 978300760, 1]
df
```

Out[62]:

	r_1	r_2	r_3
rating_data			
gender	F	F	F
genres	Drama	Drama	Drama
movie_id	1193	1193	1193
rating	5	5	5

timestamp	978300760	978300760	978300760
user_id	1	1	1

--> Go to question set

DataFrame indexing

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

In [63]:
```
df['r_1']
```

Out[63]:
```
rating_data
gender                F
genres            Drama
movie_id           1193
rating                5
timestamp     978300760
user_id               1
Name: r_1
```

In [64]:
```
df.r_1
```

Out[64]:
```
rating_data
gender                F
genres            Drama
movie_id           1193
rating                5
timestamp     978300760
user_id               1
Name: r_1
```

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

In [65]:
```
df[['r_2', 'r_1']]
```

Out[65]:

	r_2	r_1
rating_data		
gender	F	F
genres	Drama	Drama
movie_id	1193	1193
rating	5	5
timestamp	978300760	978300760
user_id	1	1

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

In [66]:
```
df.ix['gender']
```

Out[66]:
```
r_1    F
r_2    F
r_3    F
Name: gender
```

In [67]:
```
df.ix[0]
```

Out[67]:
```
r_1    F
r_2    F
r_3    F
Name: gender
```

In [68]:
```
df.ix[:2]
```

Out[68]:

	r_1	r_2	r_3
rating_data			
gender	F	F	F
genres	Drama	Drama	Drama

In [69]:
```
df.ix[:2, 'r_1']
```

Out[69]:
```
rating_data
gender              F
genres          Drama
Name: r_1
```

In [70]:
```
df.ix[:2, ['r_1', 'r_2']]
```

Out[70]:

	r_1	r_2
rating_data		
gender	F	F
genres	Drama	Drama

--> Go to question set

The MovieLens dataset: loading and first look

Loading of the MovieLens dataset here is based on the intro chapter of 'Python for Data Analysis".

The MovieLens data is spread across three files. Using the `pd.read_table` method we load each file:

In [71]:
```
import pandas as pd

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

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

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

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

Out[71]:

	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

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

In [72]:
```
movielens = pd.merge(pd.merge(ratings, users), movies)
movielens
```

Out[72]:
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id       1000209  non-null values
movie_id      1000209  non-null values
rating        1000209  non-null values
timestamp     1000209  non-null values
gender        1000209  non-null values
age           1000209  non-null values
occupation    1000209  non-null values
zip           1000209  non-null values
title         1000209  non-null values
genres        1000209  non-null values
dtypes: int64(6), object(4)
```

Evaluation

Before we attempt to express the basic equations for content-based or collaborative filtering we need a basic mechanism to evaluate the performance of our engine.

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 [73]:
```
# 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)
3664
2236
```

In [74]:
```
user_ids_larger_1 = pd.value_counts(movielens.user_id, sort=False) > 1
movielens = movielens[user_ids_larger_1[movielens.user_id]]
print movielens.shape
np.all(movielens.user_id.value_counts() > 1)
```

```
(8506, 10)
```

(8506, 10)

Out[74]: True

We now generate train and test subsets using groupby and apply.

In [75]:
```
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_train.shape
print movielens_test.shape
print movielens_train.index & movielens_test.index
```

```
(5838, 11)
(2668, 11)
Int64Index([], dtype=int64)
```

Store these two sets in text files:

In [76]:
```
movielens_train.to_csv('data/movielens_train.csv')
movielens_test.to_csv('data/movielens_test.csv')
```

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 [77]:
```
def compute_rmse(y_pred, y_true):
    """ Compute Root Mean Squared Error. """
    return np.sqrt(np.mean(np.power(y_pred - y_true, 2)))
```

Evaluation: the 'evaluate' method

In [78]:
```
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)
```

Minimal reco engine v1.0: simple mean ratings

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 [79]: def estimate1(user_id, item_id):
             """ Simple content-filtering based on mean ratings. """
             return movielens_train.ix[movielens_train.user_id == user_id, 'rating'].mean()

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

```
RMSE for estimate1: 1.23078247597
```

Collaborative-based filtering using mean ratings

```
In [80]: def estimate2(user_id, movie_id):
             """ Simple collaborative filter based on mean ratings. """
             ratings_by_others = movielens_train[movielens_train.movie_id == movie_id]
             if ratings_by_others.empty: return 3.0
             #return  movielens_train.ix[movielens_train.movie_id == movie_id, 'rating'].mean()
             return ratings_by_others.rating.mean()

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

```
RMSE for estimate2: 1.1234279896
```

--> Go to question set

More formulas!

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

Content-based: generalizations of the aggregation function

Possibly incorporating metadata about items.

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

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

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:

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

Collaborative filtering: generalizations of the aggregation function

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:

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

Aggregation in pandas

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 [81]:
```
print movielens.groupby('gender')['rating'].mean()
```

```
gender
F         3.61
M         3.54
Name: rating
```

In [82]:
```
print movielens.groupby(['gender', 'age'])['rating'].mean()
```

```
gender  age
F       1      3.62
        18     3.53
        25     3.58
        35     3.67
        45     3.63
        50     3.64
        56     3.84
M       1      3.39
        18     3.52
        25     3.51
        35     3.58
        45     3.55
        50     3.69
        56     3.58
Name: rating
```

Pivoting

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

In [83]:
```
# transform the ratings frame into a ratings matrix
ratings_mtx_df = movielens.pivot_table(values='rating',
                                        rows='user_id',
                                        cols='movie_id')
# with an integer axis index only label-based indexing is possible
ratings_mtx_df.ix[ratings_mtx_df.index[-15:],ratings_mtx_df.columns[:15]]
```

Out[83]:

movie_id	1	2	4	5	6	7	10	11	12	13	15	16	17	18	19
user_id															
6006	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2	NaN	NaN
6007	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6010	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6011	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6014	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6016	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6018	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6019	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6021	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6022	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6025	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6030	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6031	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6036	NaN	NaN	NaN	NaN	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6037	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

In [84]:
```python
by_gender_title = movielens.groupby(['gender', 'title'])['rating'].mean()
print by_gender_title
```

```
gender  title
F       'Til There Was You (1997)             2.00
        10 Things I Hate About You (1999)     3.00
        101 Dalmatians (1961)                 3.67
        101 Dalmatians (1996)                 4.00
        187 (1997)                            1.00
        2 Days in the Valley (1996)           2.50
        20 Dates (1998)                       3.00
        200 Cigarettes (1999)                 1.00
        2001: A Space Odyssey (1968)          4.25
        28 Days (2000)                        4.50
        42 Up (1998)                          4.00
        8 Heads in a Duffel Bag (1997)        4.00
        About Last Night... (1986)            3.00
        Abyss, The (1989)                     4.00
        Ace Ventura: Pet Detective (1994)     4.00
...
M       X-Files: Fight the Future, The (1998)    3.67
        X-Men (2000)                             3.89
        Yards, The (1999)                        3.00
        Year of Living Dangerously (1982)        3.00
        Yellow Submarine (1968)                  3.00
        Yojimbo (1961)                           5.00
        You Can't Take It With You (1938)        4.00
        You've Got Mail (1998)                   2.75
        Young Doctors in Love (1982)             3.00
        Young Frankenstein (1974)                4.00
        Young Guns (1988)                        3.75
        Young Guns II (1990)                     1.50
        Young Poisoner's Handbook, The (1995)    2.67
```

```
        Zero Effect (1998)                              3.50
        eXistenZ (1999)                                 4.00
Name: rating, Length: 3108
```

In [85]:
```
by_gender_title = movielens.groupby(['gender', 'title'])['rating'].mean().unstack('gender')
by_gender_title.head(10)
```

Out[85]:

gender	F	M
title		
'Til There Was You (1997)	2.00	NaN
'burbs, The (1989)	NaN	3.00
...And Justice for All (1979)	NaN	3.00
10 Things I Hate About You (1999)	3.00	3.17
101 Dalmatians (1961)	3.67	3.50
101 Dalmatians (1996)	4.00	3.00
12 Angry Men (1957)	NaN	5.00
13th Warrior, The (1999)	NaN	3.00
187 (1997)	1.00	NaN
2 Days in the Valley (1996)	2.50	4.00

In [86]:
```
by_gender_title = movielens.pivot_table('rating', rows='title', cols='gender')
by_gender_title.head(10)
```

Out[86]:

gender	F	M
title		
'Til There Was You (1997)	2.00	NaN
'burbs, The (1989)	NaN	3.00
...And Justice for All (1979)	NaN	3.00
10 Things I Hate About You (1999)	3.00	3.17
101 Dalmatians (1961)	3.67	3.50
101 Dalmatians (1996)	4.00	3.00
12 Angry Men (1957)	NaN	5.00
13th Warrior, The (1999)	NaN	3.00
187 (1997)	1.00	NaN
2 Days in the Valley (1996)	2.50	4.00

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 [87]:
```
user_info = users.set_index('user_id')
user_info.head(5)
```

Out[87]:

Out[87]:

	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

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

In [88]:
```
user_id = 6
user_info.ix[user_id, 'gender']
```

Out[88]: 'F'

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 [89]:
```
def estimate3(user_id, movie_id):
    """ Collaborative filtering using an implicit sim(u,u'). """
    ratings_by_others = movielens_train[movielens_train.movie_id == movie_id]
    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: 1.17400824171
```

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

In [90]:
```
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: 1.17400824171
```

In [91]:
```
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: 1.20520133441
```

Minimal reco engine v1.2: custom similarity functions

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 [92]:
```
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 [93]:
```
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 [94]:
```
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 [95]:

```
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)
```

Collaborative-based filtering using custom sim functions

In [96]:

```
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_i

    def estimate(self, user_id, movie_id):
        """ Ratings weighted by correlation similarity. """
        ratings_by_others = movielens_train[movielens_train.movie_id == movie_id]
        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: 1.06037523514
/usr/local/Cellar/python/2.7.3/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-
packages/pandas/core/frame.py:2762: FutureWarning: set_index with inplace=True  will return
None from pandas 0.11 onward
  " from pandas 0.11 onward", FutureWarning)
```

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!

PyTables

What is it?

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

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.

Sample file structure

```
/ (RootGroup) ''
/historic (Group) ''
/historic/purchases (Table(79334510,)) 'Purchases table'
  description := {
  "product_id": StringCol(itemsize=16, shape=(), dflt='', pos=0),
  "purchase_date": Float32Col(shape=(), dflt=0.0, pos=1),
  "purchase_id": StringCol(itemsize=16, shape=(), dflt='', pos=2),
  "user_id": StringCol(itemsize=16, shape=(), dflt='', pos=3)}
  byteorder := 'little'
  chunkshape := (1260,)
  autoIndex := True
  colindexes := {
    "purchase_id": Index(9, full, shuffle, zlib(1)).is_CSI=False,
    "user_id": Index(9, full, shuffle, zlib(1)).is_CSI=False,
    "product_id": Index(9, full, shuffle, zlib(1)).is_CSI=False,
    "purchase_date": Index(9, full, shuffle, zlib(1)).is_CSI=False}
/itemops (Group) ''
/userops (Group) ''
/userops/recent_ui_idxs (CArray(2, 120388)) ''
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := (1, 8192)
/userops/ui_idxs (CArray(2, 4170272)) ''
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := (1, 16384)
/userops/inc_mtx (Group) ''
/userops/inc_mtx/data (Array(1918289,)) ''
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
/userops/inc_mtx/indices (Array(1918289,)) ''
  atom := Int32Atom(shape=(), dflt=0)
```

```
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
/userops/inc_mtx/indptr (Array(446122,)) ''
  atom := Int32Atom(shape=(), dflt=0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
/userops/inc_mtx/shape (Array(2,)) ''
  atom := Int64Atom(shape=(), dflt=0)
  maindim := 0
  flavor := 'python'
  byteorder := 'little'
  chunkshape := None
```

File creation

In [97]:
```
import tables as tb

h5file = tb.openFile('data/tutorial.h5', mode='w', title='Test file')
h5file
```

Out[97]:
```
File(filename=data/tutorial.h5, title='Test file', mode='w', rootUEP='/',
filters=Filters(complevel=0, shuffle=False, fletcher32=False))
/ (RootGroup) 'Test file'
```

Group and dataset creation

In [98]:
```
group_1 = h5file.createGroup(h5file.root, 'group_1', 'Group One')
group_2 = h5file.createGroup('/', 'group_2', 'Group Two')
h5file
```

Out[98]:
```
File(filename=data/tutorial.h5, title='Test file', mode='w', rootUEP='/',
filters=Filters(complevel=0, shuffle=False, fletcher32=False))
/ (RootGroup) 'Test file'
/group_1 (Group) 'Group One'
/group_2 (Group) 'Group Two'
```

In [99]:
```
h5file.createArray(group_1, 'random_arr_1', np.random.randn(30),
                   "Just a bunch of random numbers")
```

Out[99]:
```
/group_1/random_arr_1 (Array(30,)) 'Just a bunch of random numbers'
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
```

In [100]:
```
h5file
```

Out[100]:
```
File(filename=data/tutorial.h5, title='Test file', mode='w', rootUEP='/',
filters=Filters(complevel=0, shuffle=False, fletcher32=False))
/ (RootGroup) 'Test file'
/group_1 (Group) 'Group One'
/group_1/random_arr_1 (Array(30,)) 'Just a bunch of random numbers'
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
```

```
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
/group_2 (Group) 'Group Two'
```

In [101]:
```
h5file.root.group_1.random_arr_1
```

Out[101]:
```
/group_1/random_arr_1 (Array(30,)) 'Just a bunch of random numbers'
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
```

In [102]:
```
h5file.root.group_1.random_arr_1[:5]
```

Out[102]:
```
array([-0.3357,  1.7229,  0.2558, -1.0513,  0.4501])
```

Group attributes

In [103]:
```
from datetime import datetime
h5file.setNodeAttr(group_1, 'last_modified', datetime.utcnow())
group_1._v_attrs
```

Out[103]:
```
/group_1._v_attrs (AttributeSet), 4 attributes:
   [CLASS := 'GROUP',
    TITLE := 'Group One',
    VERSION := '1.0',
    last_modified := datetime.datetime(2013, 3, 7, 22, 58, 39, 795635)]
```

In [104]:
```
h5file.getNodeAttr(group_1,'last_modified')
```

Out[104]:
```
datetime.datetime(2013, 3, 7, 22, 58, 39, 795635)
```

Handling things that don't fit in memory

In [105]:
```
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")
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-105-f89c8425e082> in <module>()
      2 ndim = 6000000
      3 h5file.createArray(group_3, 'random_group_3',
----> 4                    numpy.zeros((ndim,ndim)), "A very very large array")

MemoryError:
```

In [111]:
```
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 [112]:
```
earr
```

Out[112]:
```
/group_3/EArray (EArray(10, 10)) 'A very very large array, second try.'
  atom := Int8Atom(shape=(), dflt=0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'irrelevant'
  chunkshape := (6553, 10)
```

References and further reading

- 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.
- Resnick, Paul, and Hal R. Varian. “Recommender Systems.” Commun. ACM 40, no. 3 (March 1997): 56–58. doi:10.1145/245108.245121.
- 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.
- 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.
- Koren, Y., R. Bell, and C. Volinsky. “Matrix Factorization Techniques for Recommender Systems.” Computer 42, no. 8 (2009): 30–37.
- William Wesley McKinney. Python for Data Analysis. O’Reilly, 2012.
- Toby Segaran. Programming Collective Intelligence. O’Reilly, 2007.
- 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.
- Shani, G., D. Heckerman, and R. I Brafman. “An MDP-based Recommender System.” Journal of Machine Learning Research 6, no. 2 (2006): 1265.
- Joseph A. Konstan, John Riedl. "Deconstructing Recommender Systems." IEEE Spectrum, October 2012.