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

Welcome!

- About Unata
  - What we do and what we use Python for (everything!)
- Environment + data files check
  - Instructions to set up a local environment and links to download handout and data files: http://unatainc.github.io/pycon2015/ (http://unatainc.github.io/pycon2015/)

About this tutorial

References

Credit where credit is due. Please check the references at the bottom of this document.

Dataset

MovieLens from GroupLens Research: grouplens.org (http://www.grouplens.org/)

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

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 (http://www.numpy.org)
- pandas -- pandas.pydata.org (http://pandas.pydata.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/etc challenge.

Roadmap

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

- quick introduction to the recommendation problem (theory)
- learn about Pandas Series and DataFrames (practice)
- known solutions & challenges to the recommendation problem (theory)
- load data, setup evaluation functions, test dummy solution (practice)

- minimal reco engine v1.0 (practice)
- more formulas! (theory)
- pandas aggregation & minimal reco engine v1.1 (practice)
- challenge (practice)

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 $U(i)$.
- $S(u,i)$ is a function that measures the utility of item i for user u.

Goal of a recommendation system

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

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.

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 [144]:

```
import numpy as np
import pandas as pd

# set some print options
np.set_printoptions(precision=4)
np.set_printoptions(threshold=5)
np.set_printoptions(suppress=True)
pd.set_option('precision', 3, 'notebook_repr_html', True, )

# init random gen
np.random.seed(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 [165]:

```
ser = pd.Series([2.0, 1.0, 5.0, 0.97, 3.0, 10.0, 0.06, 8.0])
ser
```

Out[165]:

```
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 [166]:

```
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[166]:

```
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 [167]:

```
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 [168]:

```
ser.index
```

Out[168]:

```
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 [169]:

```
ser.values
```

Out[169]:

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

Series indexing

In [170]:

```
ser.loc['gender']
```

Out[170]:

```
'F'
```

In [171]:

```
ser.loc[['gender', 'zip']]
```

Out[171]:

```
gender        F
zip       48067
dtype: object
```

In [173]:

```
booleans = np.array([False, False, False, False, False, True, False, False, Fa
booleans
```

Out[173]:

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

In [174]:

```
ser.loc[booleans]
```

Out[174]:

```
rating    5
dtype: object
```

In [175]:

```
ser.iloc[1]
```

Out[175]:

```
'F'
```

In [176]:

```
ser.iloc[[1,2]]
```

Out[176]:

```
gender         F
genres     Drama
dtype: object
```

In [177]:

```
ser.ix['gender']
```

Out[177]:

```
'F'
```

In [178]:

```
ser.ix[1]
```

Out[178]:

```
'F'
```

In [179]:

```
ser['gender']
```

Out[179]:

```
'F'
```

In [180]:

```
ser[[1,2]]
```

Out[180]:

```
gender         F
genres     Drama
dtype: object
```

Operations between Series with different index objects

```
In [181]:
```

```
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 [183]:
```

```
ser_1 = pd.Series(data=[1,3,4], index=['A', 'B', 'C'], dtype='int')
ser_2 = pd.Series(data=[5,5,5], index=['A', 'G', 'C'], dtype='float')
ser_1 + ser_2
```

```
Out[183]:
```

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

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 [184]:
```

```
# 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[184]:
```

	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 [185]:

```
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[185]:

	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 [186]:

```
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[186]:

	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 [187]:

```
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[187]:

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 [188]:

```
df = df.T
df
```

Out[188]:

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 [189]:

```
df.columns
```

Out[189]:

```
Index([u'gender', u'genres', u'movie_id', u'rating', u'timestamp', u'user_i
d'], dtype='object')
```

```
In [190]:
```

```
df.index
```

```
Out[190]:

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

```
In [191]:
```

```
df.values
```

```
Out[191]:

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

Adding/Deleting entries

```
In [192]:
```

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

```
Out[192]:
```

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

```
In [193]:
```

```
df.drop('r_1', axis=1)
```

```
Out[193]:
```

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

Add a new column using dictionary notation:

```
In [198]:
```

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

```
Out[198]:
```

	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

```
In [199]:
```

```
df['r_4'] = pd.Series({'gender': 'M'})
df
```

```
Out[199]:
```

	r_1	r_3	r_4
rating_data			
gender	F	F	M
genres	Drama	Drama	NaN
movie_id	1193	1193	NaN
rating	5	5	NaN
timestamp	978300760	978300760	NaN
user_id	1	1	NaN

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

DataFrame indexing

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

```
In [200]:
```

```
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[200]:
```

	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

In [201]:

```
df['col_1']
```

Out[201]:

```
obs1     0.12
obs2     7.00
obs3    45.00
obs4    10.00
Name: col_1, dtype: float64
```

In [202]:

```
df.col_1
```

Out[202]:

```
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 [203]:

```
df[['col_2', 'col_1']]
```

Out[203]:

	col_2	**col_1**
obs1	0.9	0.12
obs2	9.0	7.00
obs3	34.0	45.00
obs4	11.0	10.00

DataFrame has similar .loc and .iloc methods:

In [204]:

```
df.loc['obs1', 'col_1']
```

Out[204]:

```
0.12
```

```
In [205]:
```

```
df.iloc[0, 0]
```

```
Out[205]:

0.12
```

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

```
In [206]:
```

```
df.ix['obs3']
```

```
Out[206]:

col_1     45
col_2     34
col_3    NaN
Name: obs3, dtype: object
```

```
In [207]:
```

```
df.ix[0]
```

```
Out[207]:

col_1    0.12
col_2     0.9
col_3     NaN
Name: obs1, dtype: object
```

```
In [208]:
```

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

```
Out[208]:
```

	col_1	**col_2**	**col_3**
obs1	0.12	0.9	NaN
obs2	7.00	9.0	NaN

```
In [209]:
```

```
df.ix[:2, 'col_2']
```

```
Out[209]:

obs1    0.9
obs2    9.0
Name: col_2, dtype: float64
```

```
In [210]:
```

```
df.ix[:2, ['col_1', 'col_2']]
```

```
Out[210]:
```

	col_1	col_2
obs1	0.12	0.9
obs2	7.00	9.0

--> Go to "Pandas questions: Indexing"

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 [211]:
```

```
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[211]:
```

	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 [212]:

```
movielens = pd.merge(pd.merge(ratings, users), movies)
movielens.head()
```

Out[212]:

	user_id	movie_id	rating	timestamp	gender	age	occupation	zip	title	genres
0	1	1193	5	978300760	F	1	10	48067	One Flew Over the Cuckoo's Nest (1975)	Drama
1	2	1193	5	978298413	M	56	16	70072	One Flew Over the Cuckoo's Nest (1975)	Drama
2	12	1193	4	978220179	M	25	12	32793	One Flew Over the Cuckoo's Nest (1975)	Drama
3	15	1193	4	978199279	M	25	7	22903	One Flew Over the Cuckoo's Nest (1975)	Drama
4	17	1193	5	978158471	M	50	1	95350	One Flew Over the Cuckoo's Nest (1975)	Drama

Challenge prep: evaluation mechanism

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.

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 [213]:

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

```
(10000, 10)
3698
2275
```

In [214]:

```
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
print movielens.shape
assert np.all(movielens.user_id.value_counts() > 1)
```

```
(8442, 10)
```

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

In [215]:

```
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
```

```
(8442, 11)
(5801, 11)
(2641, 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')
```

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 [216]:

```
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 [217]:

```
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)
```

In [218]:

```
def my_estimate_function(user_id, movie_id):
    return 3
```

In [219]:

```
print 'RMSE for my estimate function: %s' % evaluate(my_estimate_function)
```

```
RMSE for my estimate function: 1.25716424791
```

--> Go to "Mini Challenge prep: data loading & evaluation functions"

Well-known Solutions to the Recommendation Problem

Content-based filtering

Recommend based on the user's rating history.

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

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

A simple example using the mean as an aggregation function:

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

Collaborative filtering

Recommend based on other user's rating histories.

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

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

A simple example using the mean as an aggregation function:

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

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.

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 [220]:

```
def content_mean(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(content_mean)
```

```
RMSE for estimate1: 1.26844670375
```

--> Go to "Reco systems questions: Minimal reco engine v1.0"

More formulas!

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

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:

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

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:

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

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 [222]:

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

Out[222]:

```
gender
F    3.59
M    3.51
Name: rating, dtype: float64
```

In [223]:

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

Out[223]:

```
gender  age
F       1      3.61
        18     3.43
        25     3.61
        35     3.65
        45     3.54
        50     3.67
        56     4.07
M       1      3.38
        18     3.44
        25     3.43
        35     3.61
        45     3.55
        50     3.76
        56     3.70
Name: rating, dtype: float64
```

Pivoting

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

In [224]:

```
# transform the ratings frame into a ratings matrix
ratings_mtx_df = movielens_train.pivot_table(values='rating',
                                             index='user_id',
                                             columns='movie_id')
ratings_mtx_df.head(3)
```

Out[224]:

movie_id	1	2	3	4	5	6	7	10	11	14	...	3930	3932	3935	3938	394
user_id																
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaI
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaI
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaI

3 rows × 1934 columns

In [225]:

```
# grab another subsquare of the ratings matrix to actually diplay some real en
ratings_mtx_df.loc[11:16, 1196:1200]
```

Out[225]:

movie_id	1196	1197	1198	1199	1200
user_id					
11	NaN	NaN	NaN	NaN	NaN
13	5	NaN	NaN	NaN	NaN
15	NaN	NaN	NaN	NaN	NaN

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

In [226]:

```
movielens_train.pivot_table(values='rating', index='age', columns='gender', ag
```

Out[226]:

gender	F	M
age		
1	3.61	3.38
18	3.43	3.44
25	3.61	3.43
35	3.65	3.61
45	3.54	3.55
50	3.67	3.76
56	4.07	3.70

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

In [227]:

```
movielens_train.pivot_table(values='rating', index='age', columns='gender', ag
```

Out[227]:

	mean		std	
gender	**F**	**M**	**F**	**M**
age				
1	3.61	3.38	1.28	1.30
18	3.43	3.44	1.20	1.15
25	3.61	3.43	1.08	1.14
35	3.65	3.61	1.02	1.04
45	3.54	3.55	1.00	1.06
50	3.67	3.76	1.14	1.04
56	4.07	3.70	0.96	1.08

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 [228]:

```
user_info = users.set_index('user_id')
user_info.head(5)
```

Out[228]:

	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 [229]:

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

Out[229]:

```
'M'
```

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 [230]:
```

```
def collab_gender(user_id, movie_id):
    """ Collaborative filtering using an implicit sim(u,u') based on gender. "

    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', index='movie_id'
    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 collab_gender: %s' % evaluate(collab_gender)
```

```
RMSE for collab_gender: 1.1953061054
```

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

```
In [231]:
```

```
class CollabGenderReco:
    """ Collaborative filtering using an implicit sim(u,u'). """

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

        self.means_by_gender = movielens_train.pivot_table('rating', index='mc

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

```
RMSE for CollabGenderReco: 1.1953061054
```

Break!!

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!

Mini-Challenge: first round

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

- collaborative filter based on age similarities
- collaborative filter based on zip code similarities
- collaborative filter based on occupation similarities
- content filter based on movie genre

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 [232]:

```
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 [233]:

```
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 [234]:

```
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 [235]:

```
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 [236]:
```

```
class CollabPearsonReco:
    """ Collaborative filtering using a custom sim(u,u'). """

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

        self.all_user_profiles = movielens.pivot_table('rating', index='movie_

    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_conditi
        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_prof
        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 = CollabPearsonReco()
reco.learn()
print 'RMSE for CollabPearsonReco: %s' % evaluate(reco.estimate)
```

```
RMSE for CollabPearsonReco: 1.08629365955
```

Mini-Challenge: second round

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

- euclidean
- cosine

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 (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.