Week 1

Capstone project and Recommender Systems

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Introduction

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Introduction to Recommender Systems

Learning objectives

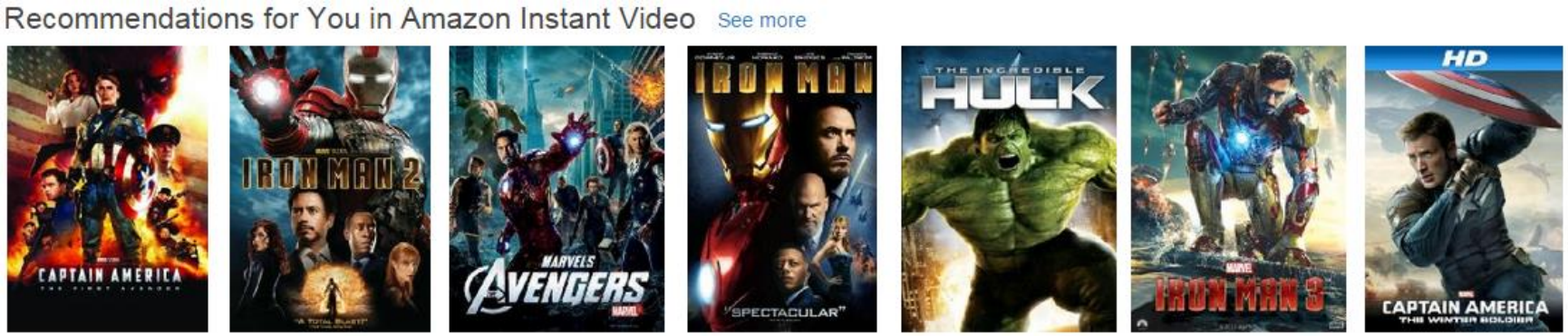
In this lecture we will...

- Motivate the problem of recommender systems as an important Data Product
- Describe several examples of recommender systems on the Web

Why recommendation?

The goal of recommender systems is…

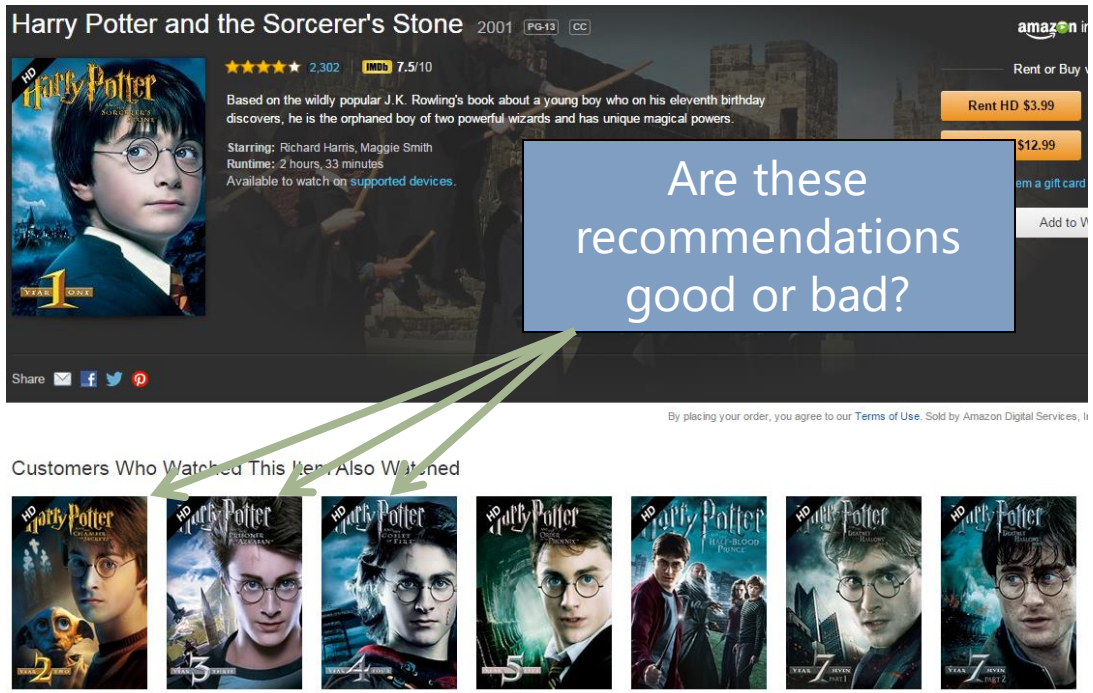
- To help people discover new content



Why recommendation?

The goal of recommender systems is…

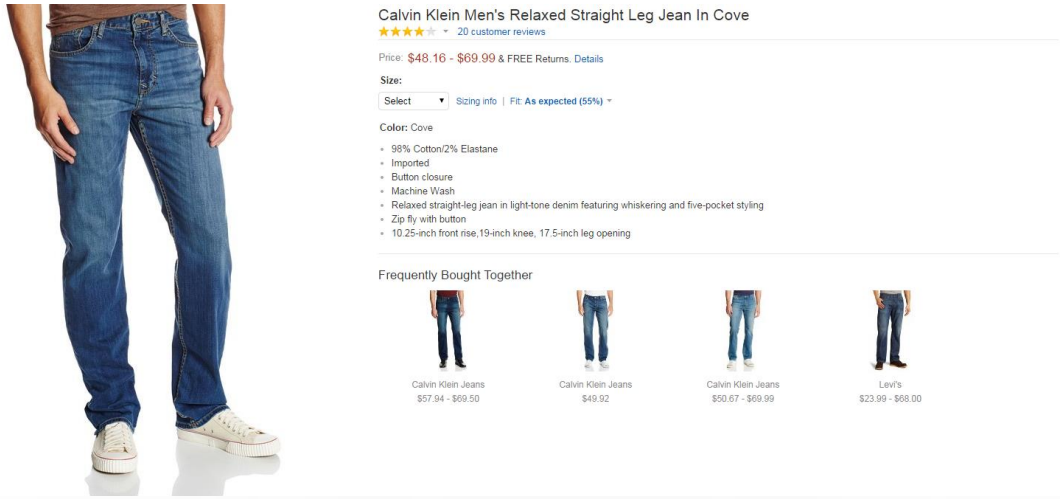
- To help us find the content we were already looking for



Why recommendation?

The goal of recommender systems is…

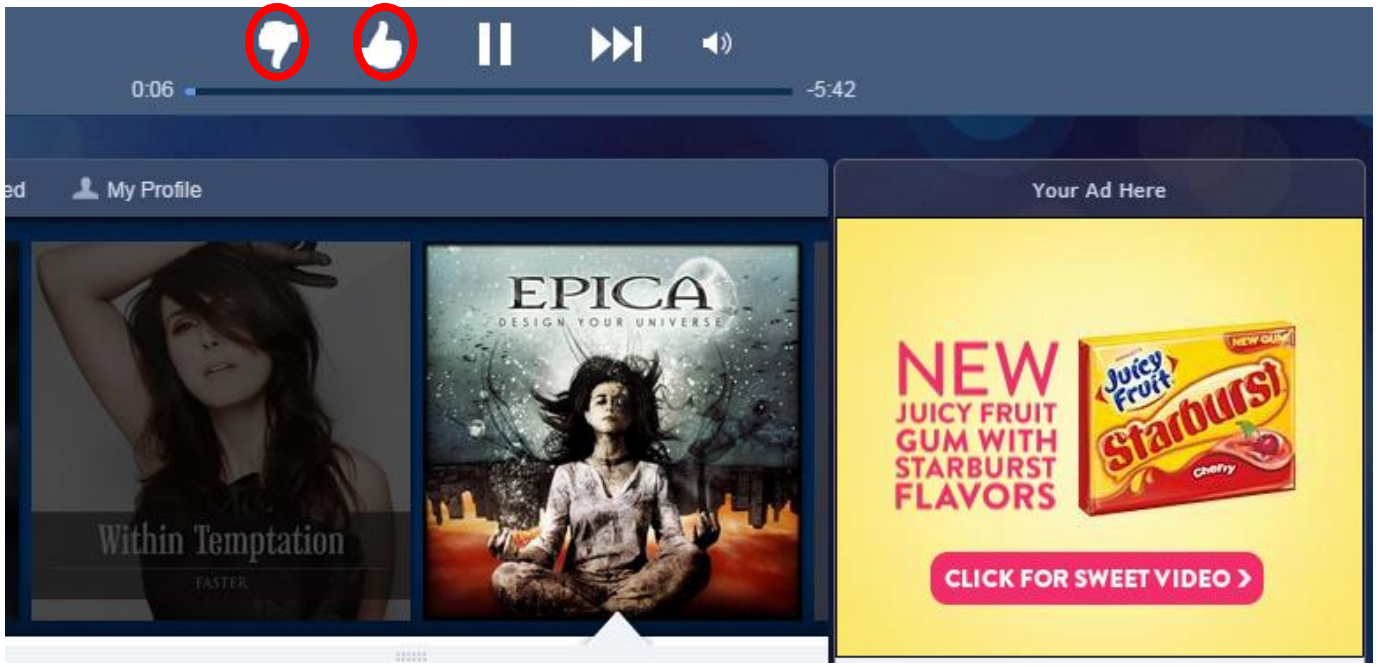
- To discover which things go together



Why recommendation?

The goal of recommender systems is…

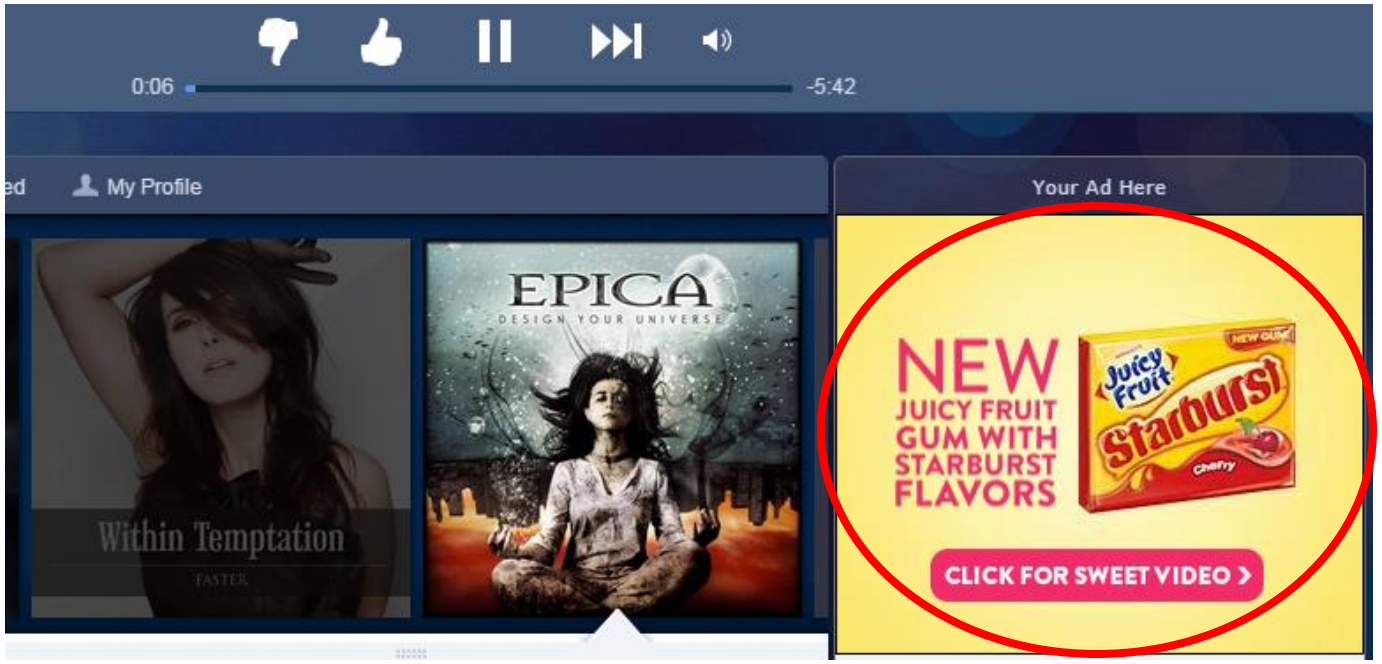
- To personalize user experiences in response to user feedback



Why recommendation?

The goal of recommender systems is…

- To recommend incredible products that are relevant to our interests



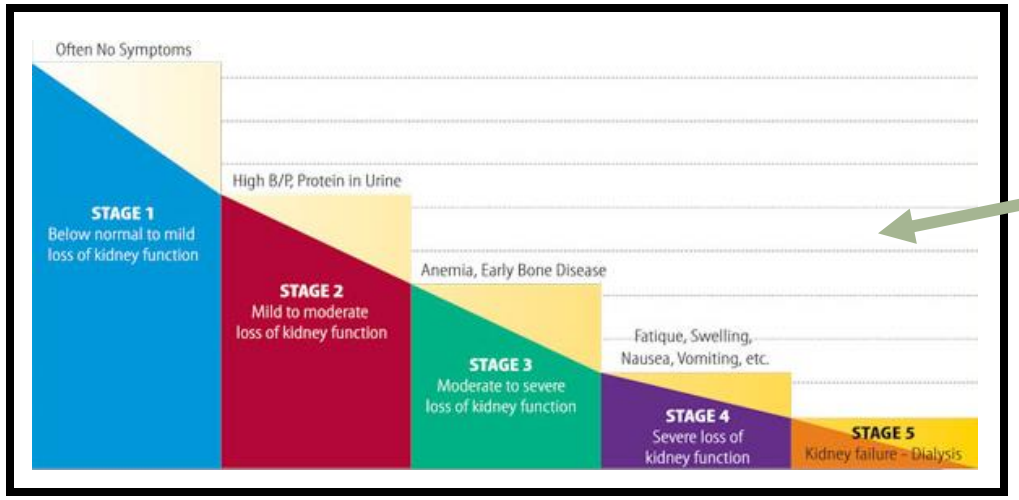
Why recommendation?

The goal of recommender systems is…

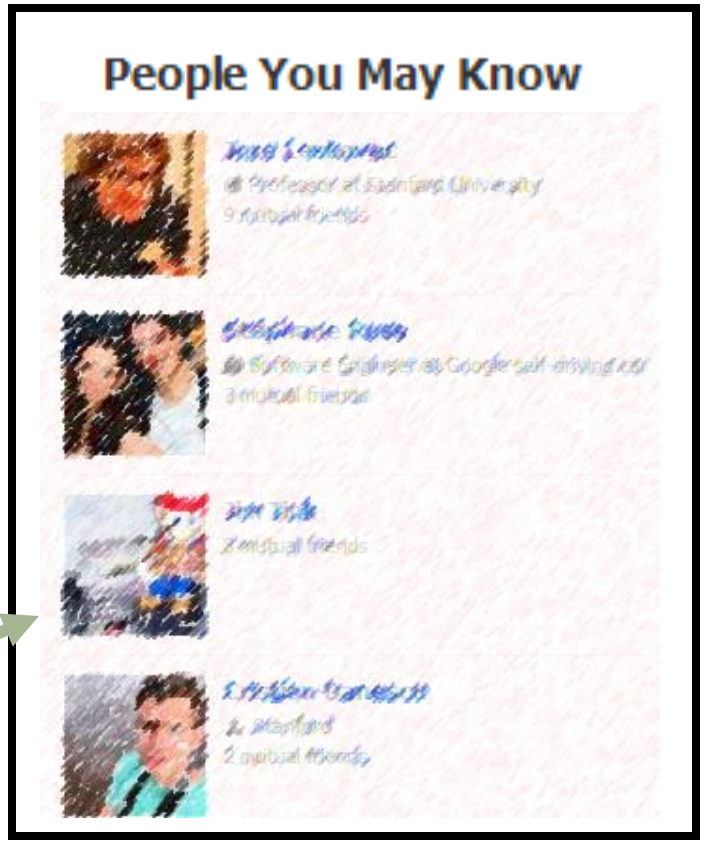
- To identify things that we **like**

Recommendation in non-e-commerce

The goal of recommender systems is…



- Priority Inbox
- Personalized Health
- Friend Recommendation



Why recommendation?

The goal of recommender systems is…

- To help people discover new content
- To help us find the content we were already looking for
- To discover which things go together
- To personalize user experiences in response to user feedback
- To identify things that we **like**
- **To model peoples preferences, opinions, and behavior**

Summary of concepts

- Described some of the common use-cases of recommender systems on the web

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Recommender Systems versus other forms of supervised learning

Learning objectives

In this lecture we will...

- Explain the difference between recommender systems and other types of machines learning
- Introduce the two recommender systems we will develop in this course

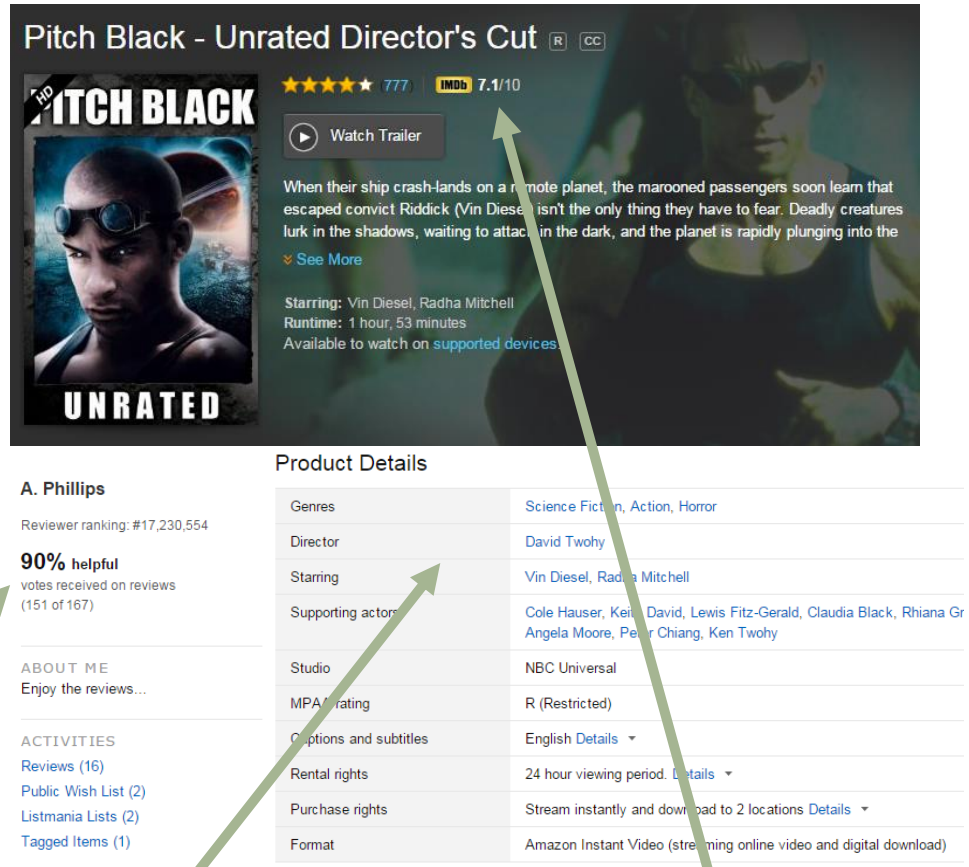
Recommending things to people

Suppose we want to build a movie recommender

e.g. which of these films will I rate highest?

Recommending things to people

We already have a few tools in our "supervised learning" toolbox that may help us

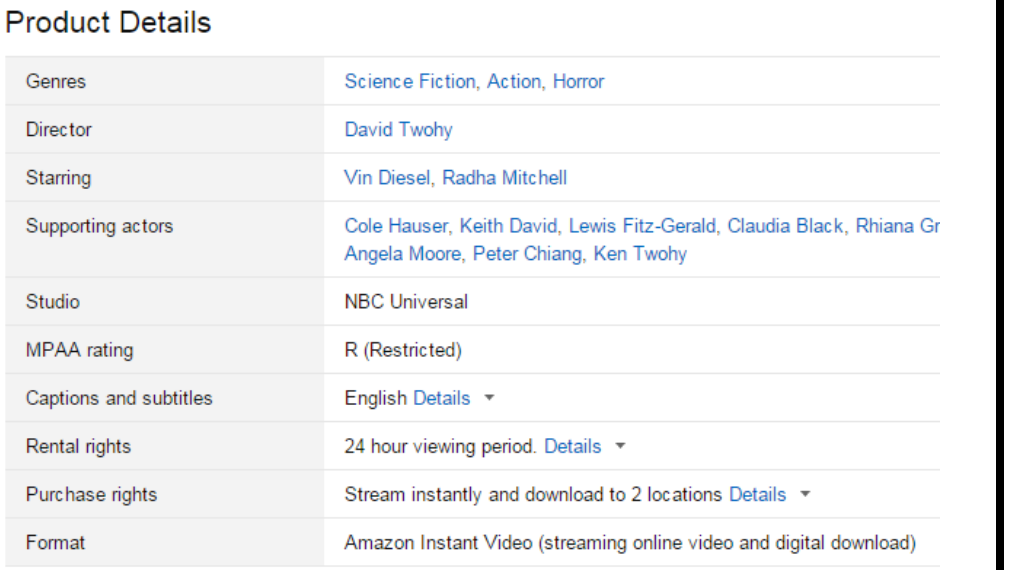


$$f(\text{user features}, \text{movie features}) \xrightarrow{?} \text{star rating}$$

Recommending things to people

$$f(\text{user features}, \text{movie features}) \xrightarrow{?} \text{star rating}$$

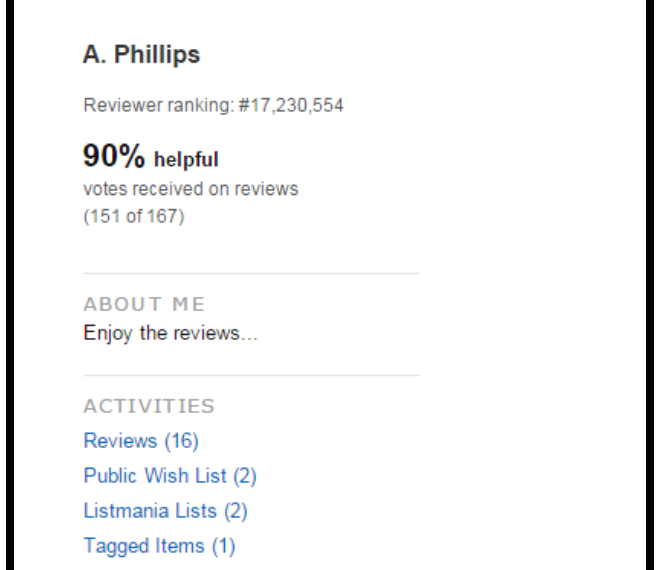
Movie features: genre, actors, rating, length, etc.



Product Details

Genres	Science Fiction, Action, Horror
Director	David Twohy
Starring	Vin Diesel, Radha Mitchell
Supporting actors	Cole Hauser, Keith David, Lewis Fitz-Gerald, Claudia Black, Rhiana Gr Angela Moore, Peter Chiang, Ken Twohy
Studio	NBC Universal
MPAA rating	R (Restricted)
Captions and subtitles	English Details
Rental rights	24 hour viewing period. Details
Purchase rights	Stream instantly and download to 2 locations Details
Format	Amazon Instant Video (streaming online video and digital download)

User features: age, gender, location, etc.



A. Phillips

Reviewer ranking: #17,230,554

90% helpful
votes received on reviews
(151 of 167)

ABOUT ME
Enjoy the reviews...

ACTIVITIES
Reviews (16)
Public Wish List (2)
Listmania Lists (2)
Tagged Items (1)

Recommending things to people

$$f(\text{user features}, \text{movie features}) \xrightarrow{?} \text{star rating}$$

With the models we've seen so far, we can build predictors that account for...

- Do women give higher ratings than men?
- Do Americans give higher ratings than Australians?
- Do people give higher ratings to action movies?
- Are ratings higher in the summer or winter?
- Do people give high ratings to movies with Vin Diesel?

So what **can't** we do yet?

Recommending things to people

$$f(\text{user features}, \text{movie features}) \stackrel{?}{\rightarrow} \text{star rating}$$

Consider the following linear predictor (e.g. from Course 1):

$$f(\text{user features}, \text{movie features}) = \langle \phi(\text{user features}); \phi(\text{movie features}), \theta \rangle$$

$$= \langle \phi(\text{user features}), \theta_{\text{user}} \rangle + \langle \phi(\text{movie features}), \theta_{\text{movie}} \rangle$$

Recommending things to people

But this is essentially just two separate predictors!

$$f(\text{user features}, \text{movie features}) =$$

$$= \underbrace{\langle \phi(\text{user features}), \theta_{\text{user}} \rangle}_{\text{user predictor}} + \underbrace{\langle \phi(\text{movie features}), \theta_{\text{movie}} \rangle}_{\text{movie predictor}}$$

That is, we're treating user and movie features as though they're **independent!**

Recommending things to people

But these predictors should (obviously?) **not** be independent

$$f(\text{user features}, \text{movie features}) = f(\text{user}) + f(\text{movie})$$

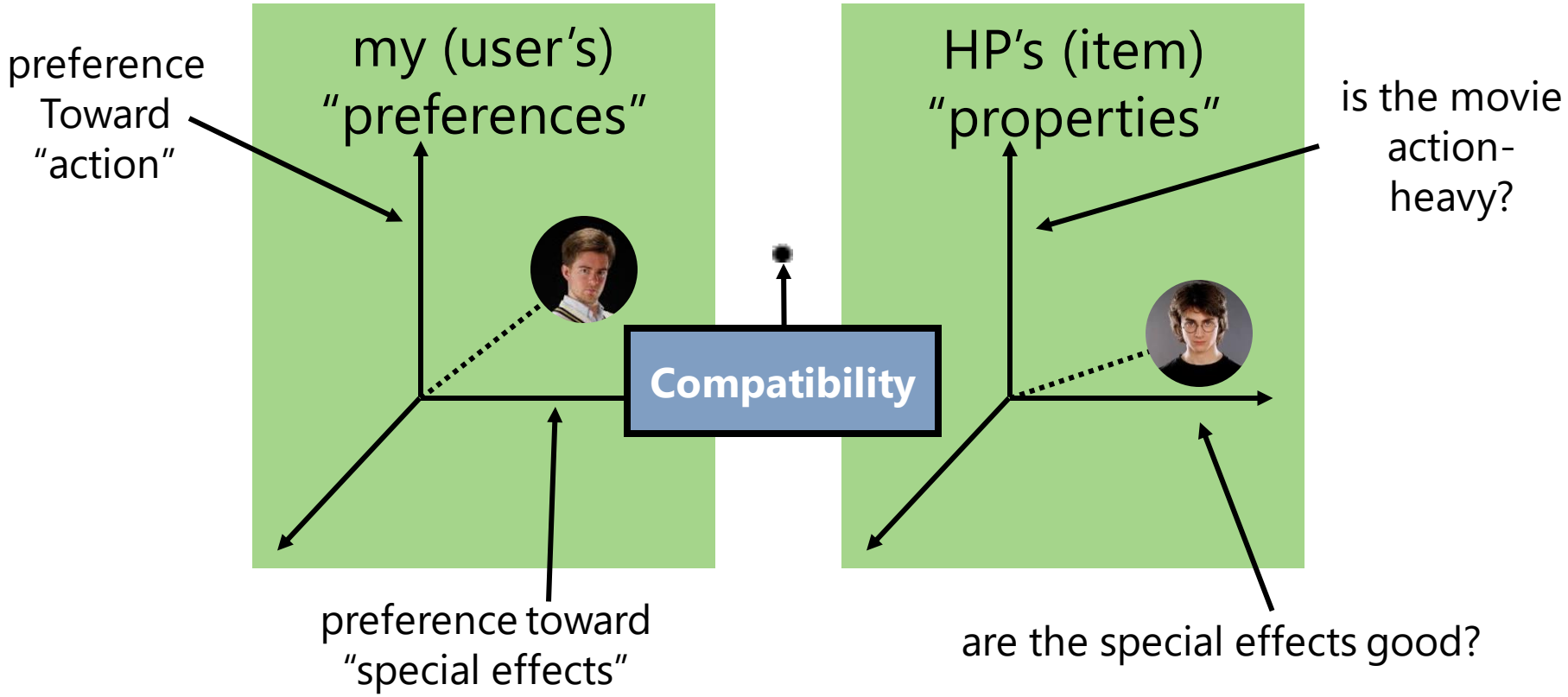
do I tend to give high ratings?

does the population tend to give high ratings to this genre of movie?

But what about a feature like "do **I** give high ratings to **this genre** of movie"?

Recommending things to people

Recommender Systems go beyond the methods we've seen so far by trying to model the **relationships** between people and the items they're evaluating



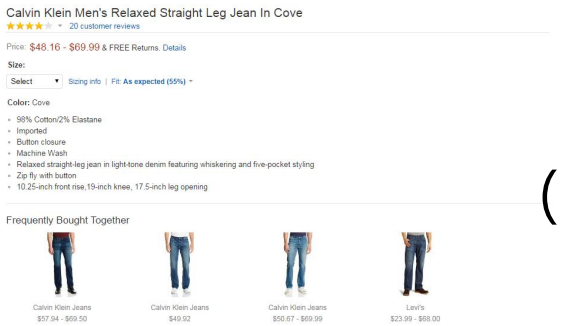
Recommending things to people

We'll look at two common types of recommender systems in this course:

1. **"Similarity-based"** recommender systems and collaborative filtering
2. **"Model-based"** recommender systems

Recommending things to people

- **"Similarity-based"** recommender systems measure similarity between items in terms of users who have purchased them



(E.g. "People who bought *X* also bought *Y*")

Recommending things to people

- **"Model-based"** recommender systems use machine learning to estimate specific outcomes

(E.g. rating prediction on Netflix)

Summary of concepts

- Introduced two classes of recommender systems for future study

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Collaborative filtering-based recommendation

Learning objectives

In this lecture we will...

- Introduce collaborative-filtering (or “similarity-based”) recommender systems
- Demonstrate three instances of this type of recommender system

Defining similarity between users & items

Q: How can we measure the **similarity** between two **users?**
A: In terms of the **items** they purchased!

Q: How can we measure the similarity between two **items?**
A: In terms of the users who purchased them!

Defining similarity between users & items

e.g.:
Amazon

Calvin Klein Men's Relaxed Straight Leg Jean In Cove

★★★★☆ 20 customer reviews

Price: $48.16 - $69.99 & FREE Returns. Details

Size:

Select | Sizing info | Fit: As expected (55%)

Color: Cove

- 98% Cotton/2% Elastane
- Imported
- Button closure
- Machine Wash
- Relaxed straight-leg jean in light-tone denim featuring whiskering and five-pocket styling
- Zip fly with button
- 10.25-inch front rise,19-inch knee, 17.5-inch leg opening

Frequently Bought Together


Calvin Klein Jeans
$57.94 - $69.50


Calvin Klein Jeans
$49.92


Calvin Klein Jeans
$50.67 - $69.99


Levi's
$23.99 - $68.00

Customers Who Viewed This Item Also Viewed

Customers Who Bought This Item Also Bought

Page 3

Definitions

I_u = set of items purchased by user *u*

U_i = set of users who purchased item *i*

Definitions

Or equivalently...

$$R = \overbrace{\begin{pmatrix} 1 & 0 & \cdots & 1 \\ 0 & 0 & & 1 \\ \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & 1 \end{pmatrix}}^{\text{items}} \Bigg\} \text{users}$$

R_u = binary representation of items purchased by *u*

$R_{.,i}$ = binary representation of users who purchased *i*

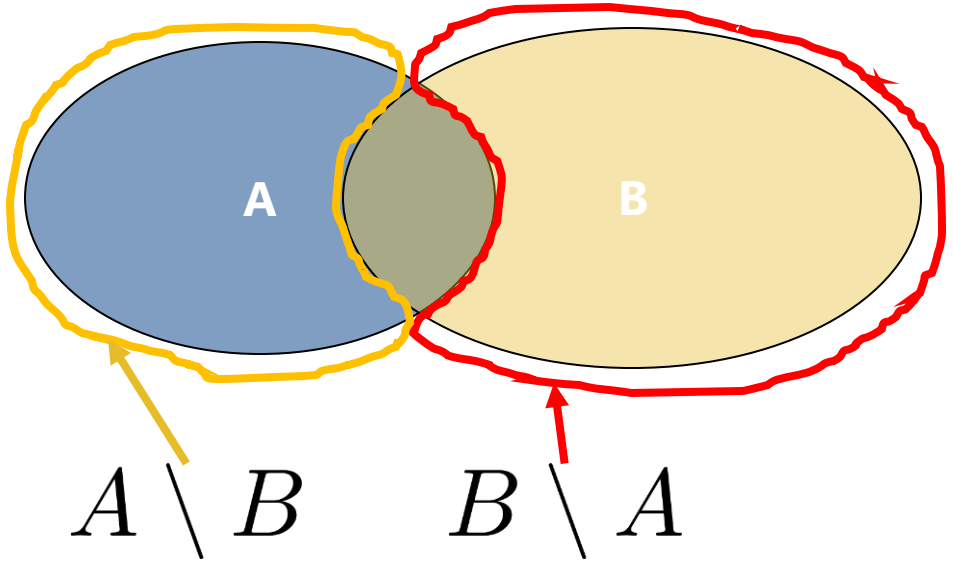
$$I_u = \{i | R_{u,i} = 1\} \qquad U_i = \{u | R_{u,i} = 1\}$$

0. Euclidean distance

Euclidean distance:

e.g. between two items i,j (similarly defined between two users)

$$|U_i \setminus U_j| + |U_j \setminus U_i| = \|R_i - R_j\|$$



0. Euclidean distance

Euclidean distance:

e.g.: U_1 = {1,4,8,9,11,23,25,34}
U_2 = {1,4,6,8,9,11,23,25,34,35,38}
U_3 = {4}
U_4 = {5}

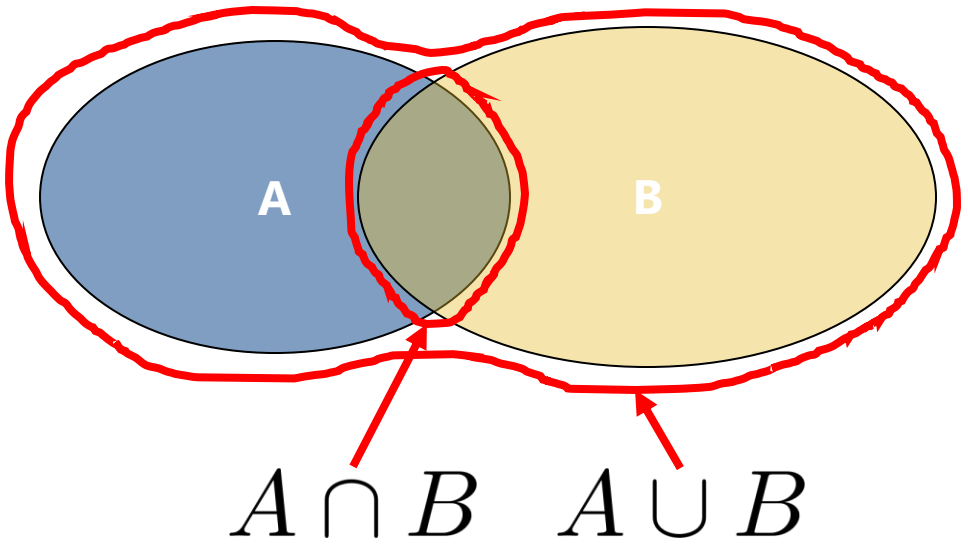
$$|U_1 \setminus U_2| + |U_2 \setminus U_1| = 3$$
$$|U_3 \setminus U_4| + |U_3 \setminus U_4| = 2$$

Problem: favors small sets, even if they have few elements in common

1. Jaccard similarity

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

$$\text{Jaccard}(U_i, U_j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$$



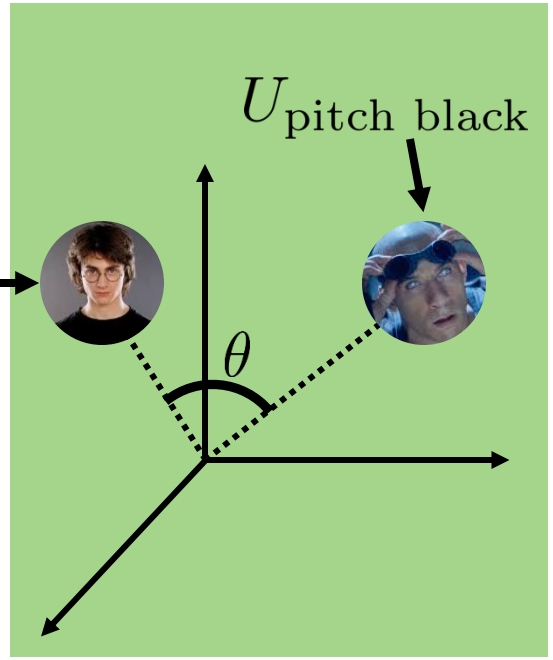
→ Maximum of 1 if the two users purchased **exactly the same** set of items
(or if two items were purchased by the same set of users)

→ Minimum of 0 if the two users purchased **completely disjoint** sets of items
(or if the two items were purchased by completely disjoint sets of users)

2. Cosine similarity

$$\text{Cosine}(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$$

$$\theta = \cos^{-1}\left(\frac{A \cdot B}{\|A\| \|B\|}\right)$$



(vector representation of users who purchased harry potter)

$$\cos(\theta) = 1$$

(theta = 0) → *A* and *B* point in exactly the same direction

$$\cos(\theta) = -1$$

(theta = 180) → *A* and *B* point in opposite directions (won't actually happen for 0/1 vectors)

$$\cos(\theta) = 0$$

(theta = 90) → *A* and *B* are orthogonal

2. Cosine similarity

Why cosine?

- Unlike Jaccard, works for arbitrary vectors
- E.g. what if we have **opinions** in addition to purchases?

$$R = \begin{pmatrix} 1 & 0 & \cdots & 1 \\ 0 & 0 & & 1 \\ \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & 1 \end{pmatrix} \rightarrow \begin{pmatrix} -1 & 0 & \cdots & 1 \\ 0 & 0 & & -1 \\ \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & -1 \end{pmatrix}$$

bought and **liked**

didn't buy

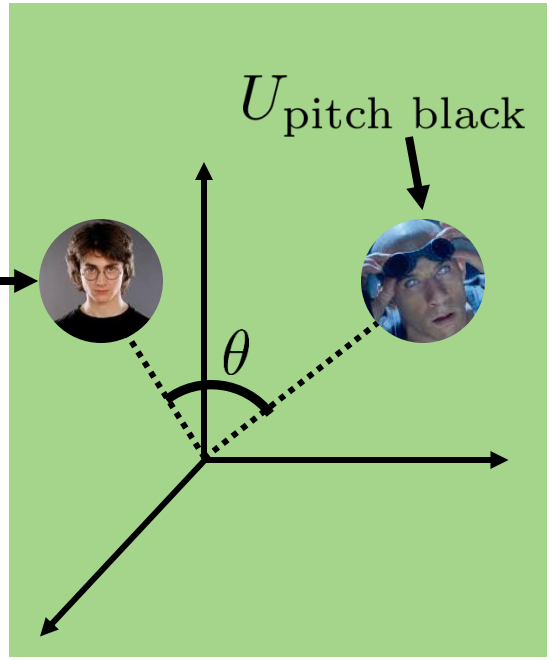
bought and **disliked**

2. Cosine similarity

E.g. our previous example, now with "thumbs-up/thumbs-down" ratings

$U_{\text{harry potter}}$

(vector representation of users' **ratings of** Harry Potter)



$$\cos(\theta) = 1$$

(theta = 0) → Rated by the same users, and they all agree

$$\cos(\theta) = -1$$

(theta = 180) → Rated by the same users, but they **completely disagree** about it

$$\cos(\theta) = 0$$

(theta = 90) → Rated by different sets of users

3. Pearson correlation

What if we have numerical ratings (rather than just thumbs-up/down)?

$$R = \begin{pmatrix} -1 & 0 & \cdots & 1 \\ 0 & 0 & & -1 \\ \vdots & & \ddots & \vdots \\ 1 & 0 & \cdots & -1 \end{pmatrix} \rightarrow \begin{pmatrix} 4 & 0 & \cdots & 2 \\ 0 & 0 & & 3 \\ \vdots & & \ddots & \vdots \\ 5 & 0 & \cdots & 1 \end{pmatrix}$$

bought and **liked**

didn't buy

bought and **disliked**

3. Pearson correlation

What if we have numerical ratings (rather than just thumbs-up/down)?

3. Pearson correlation

What if we have numerical ratings (rather than just thumbs-up/down)?

- We wouldn't want 1-star ratings to be parallel to 5-star ratings
- So we can subtract the average – values are then **negative** for below-average ratings and **positive** for above-average ratings

items rated by both users average rating by user *v*

$$\mathrm{Sim}(u,v) = \frac{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R_u})(R_{v,i} - \bar{R_v})}{\sqrt{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R_u})^2 \sum_{i \in I_u \cap I_v} (R_{v,i} - \bar{R_v})^2}}$$

3. Pearson correlation

Compare to the cosine similarity:

Pearson similarity (between users):

items rated by both users average rating by user *v*

$$\mathrm{Sim}(u,v) = \frac{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R}_u)(R_{v,i} - \bar{R}_v)}{\sqrt{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R}_u)^2 \sum_{i \in I_u \cap I_v} (R_{v,i} - \bar{R}_v)^2}}$$

Cosine similarity (between users):

$$\mathrm{Sim}(u,v) = \frac{\sum_{i \in I_u \cap I_v} R_{u,i} R_{v,i}}{\sqrt{\sum_{i \in I_u \cap I_v} R_{u,i}^2 \sum_{i \in I_u \cap I_v} R_{v,i}^2}}$$

Collaborative filtering in practice

How does amazon generate their recommendations?

Given a product: 

Let U_i be the set of users who viewed it

Rank products according to: $\frac{|U_i \cap U_j|}{|U_i \cup U_j|}$ (or cosine/pearson)










.86 .84 .82 .79 ...

Linden, Smith, & York (2003)

Collaborative filtering in practice

Note: (surprisingly) that we built something pretty useful out of **nothing but rating data** – we didn't look at any features of the products whatsoever

Collaborative filtering in practice

But: we still have
a few problems left to address…

1. This is actually kind of slow given a huge enough dataset – if one user purchases one item, this will change the rankings of **every other item that was purchased by at least one user in common**
2. Of no use for **new users** and **new items** ("cold-start" problems
3. Won't necessarily encourage diverse results

Summary of concepts

- Introduced three similarity-based recommender systems
- Described situations where each of these systems would be preferable over the others

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Latent factor models, part 1

Learning objectives

In this lecture we will...

- Introduce machine-learning based recommender systems ("latent-factor models")
- Introduce some history behind this type of model

Latent factor models

In the previous lecture we looked at approaches that try to define some definition of user/user and item/item **similarity**

Recommendation then consists of

- Finding an item *i* that a user likes (gives a high rating)
- Recommending items that are similar to it (i.e., items *j* with a similar rating profile to *i*)

Latent factor models

What we've seen so far are **unsupervised** approaches and whether the work depends highly on whether we chose a "good" notion of similarity

So, can we perform recommendations via **supervised** learning?

Latent factor models

e.g. if we can model

$$f(\text{user features}, \text{movie features}) \rightarrow \text{star rating}$$

Then recommendation
will consist of identifying

$$recommendation(u) = \arg\max_{i \in \text{unseen items}} f(u, i)$$

Background: The Netflix prize

In 2006, Netflix created a dataset of **100,000,000** movie ratings
Data looked like:

$$(\text{userID}, \text{itemID}, \text{time}, \text{rating})$$

The goal was to reduce the (R)MSE at predicting ratings:

$$\text{RMSE}(f) = \sqrt{\frac{1}{N}\sum_{u,i,t \in \text{test set}} (f(u,i,t) - r_{u,i,t})^2}$$

model's prediction — ground-truth

Whoever first manages to reduce the RMSE by **10%** versus
Netflix's solution wins **$1,000,000**

The Netflix prize

This led to **a lot** of research on rating prediction by minimizing the Mean-Squared Error

(it also led to a lawsuit against Netflix, once somebody managed to de-anonymize their data)

We'll look at a few of the main approaches

Rating prediction

Let's start with the simplest possible model:

$$f(u, i) = \alpha$$

user item

$$\alpha = \frac{1}{N} \sum_{u,i \in \text{training data}} r_{u,i}$$

Here the RMSE is just equal to the **standard deviation** of the data

(and we cannot do any better with a 0th order predictor)

Rating prediction

What about the **2nd** simplest model?

$$f(u, i) = \alpha + \beta_u + \beta_i$$

user item

how much does this user tend to rate things above the mean?

does this item tend to receive higher ratings than others

e.g.

$\alpha = 4.2$

$\beta_{\text{pitch black}} = -0.1$

$\beta_{\text{julian}} = -0.2$

Rating prediction

The optimization problem becomes:

$$\arg\min_{\alpha,\beta} \underbrace{\sum_{u,i}(\alpha + \beta_u + \beta_i - R_{u,i})^2}_{\text{error}} + \underbrace{\lambda\left[\sum_u \beta_u^2 + \sum_i \beta_i^2\right]}_{\text{regularizer}}$$

Although various solution techniques exist, we can solve this problem via gradient descent

Rating prediction

Differentiate:

$$\arg\min_{\alpha,\beta} \frac{1}{N} \sum_{u,i} (\alpha + \beta_u + \beta_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2 \right]$$

Rating prediction

Differentiate:

$$\frac{\partial \text{obj}}{\partial \beta_u} = \frac{1}{N} \sum_{u,i} 2(\alpha + \beta_u + \beta_i - R_{u,i}) + 2\lambda\beta_u$$

Rating prediction

Looks good (and actually works surprisingly well), but doesn't solve the basic issue that we started with

$$f(\text{user features}, \text{movie features}) =$$

$$= \underbrace{\langle \phi(\text{user features}), \theta_{\text{user}} \rangle}_{\text{user predictor}} + \underbrace{\langle \phi(\text{movie features}), \theta_{\text{movie}} \rangle}_{\text{movie predictor}}$$

That is, we're **still** fitting a function that treats users and items independently

Summary of concepts

- Introduced latent factor models
- Described some history behind this type of model
- Showed how simple versions of this model can be optimized using gradient descent

On your own...

- Compute the gradient update equations for the remaining terms

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Latent factor models, part 2

Learning objectives

In this lecture we will...

- Further develop the latent factor model introduced in the previous lecture

Rating prediction

So far we developed an approach that incorporates user and item biases

$$f(u, i) = \alpha + \beta_u + \beta_i$$

user item user bias item bias

e.g.

$\alpha = 4.2$

$\beta_{\text{pitch black}} = -0.1$

$\beta_{\text{julian}} = -0.2$

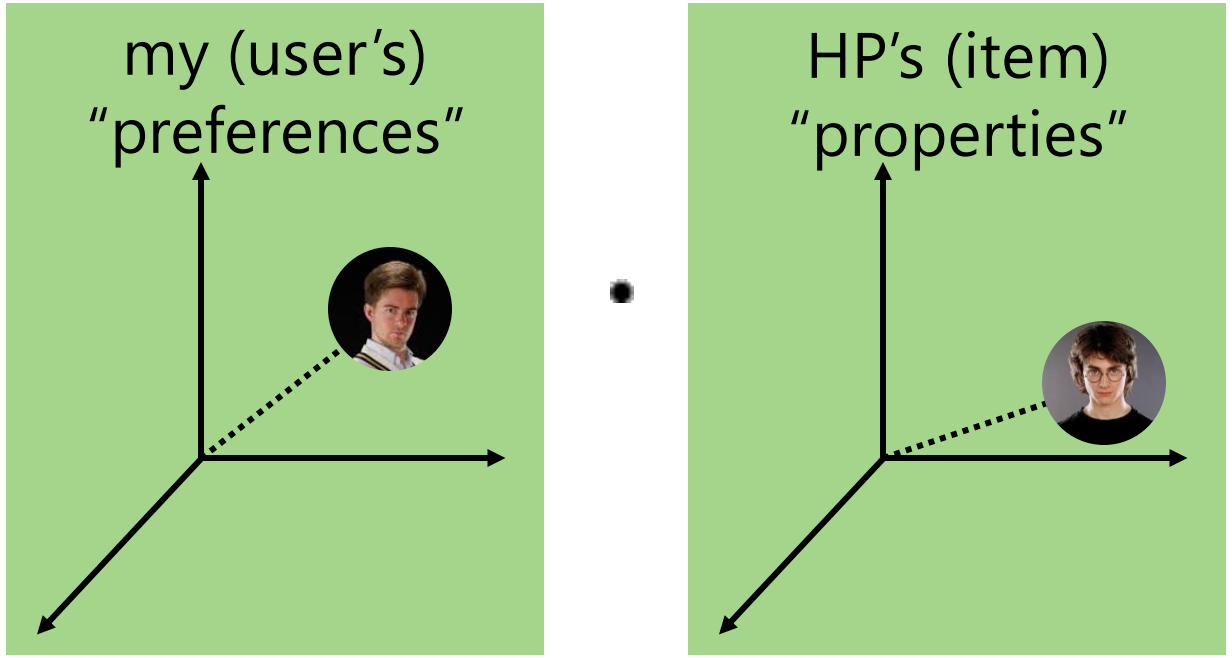
Rating prediction

So far we developed an approach that incorporates user and item biases

We saw that this type of approach, while effective, cannot actually **personalize** recommendations to individual users

Recommending things to people

How about an approach based on **dimensionality reduction?**



i.e., let's come up with low-dimensional representations of the users and the items so as to best explain the data

Dimensionality reduction

Methodologically, our idea is based on the idea of **matrix factorization:**

$$R = \begin{pmatrix} 5 & 3 & \cdots & 1 \\ 4 & 2 & & 1 \\ 3 & 1 & & 3 \\ 2 & 2 & & 4 \\ 1 & 5 & & 2 \\ \vdots & & \ddots & \vdots \\ 1 & 2 & \cdots & 1 \end{pmatrix}$$

What is the best low-rank approximation of *R* in terms of the mean-squared error?

Dimensionality reduction

Methodologically, our idea is based on the idea of **matrix factorization:**

$$R = \begin{pmatrix} 5 & 3 & \cdots & 1 \\ 4 & 2 & & 1 \\ 3 & 1 & & 3 \\ \vdots & & \ddots & \vdots \\ 1 & 2 & \cdots & 1 \end{pmatrix}$$

Singular Value Decomposition

(square roots of) eigenvalues of RR^T

$$R = U\Sigma V^T$$

eigenvectors of RR^T

eigenvectors of $R^T R$

This gives the "best" rank-K approximation (in terms of the MSE)

Dimensionality reduction

But! Our matrix of ratings is only partially observed; and it's **really big!**

$$R = \begin{pmatrix} 5 & 3 & \cdots & \cdot \\ 4 & 2 & & 1 \\ 3 & \cdot & & 3 \\ \cdot & 2 & & 4 \\ 1 & 5 & & \cdot \\ \vdots & & \ddots & \vdots \\ 1 & 2 & \cdot\cdot & \cdot \end{pmatrix}$$

Missing ratings

SVD is **not defined** for partially observed matrices, and it is **not practical** for matrices with 1Mx1M+ dimensions

Latent-factor models

So instead, let's solve approximately using gradient descent

$$R = \begin{pmatrix} 5 & 3 & \cdots & \cdot \\ 4 & 2 & & 1 \\ 3 & \cdot & & 3 \\ \cdot & 2 & & 4 \\ 1 & 5 & & \cdot \\ \vdots & & \ddots & \vdots \\ 1 & 2 & \cdots & \cdot \end{pmatrix}$$

users

items

K-dimensional representation of each item

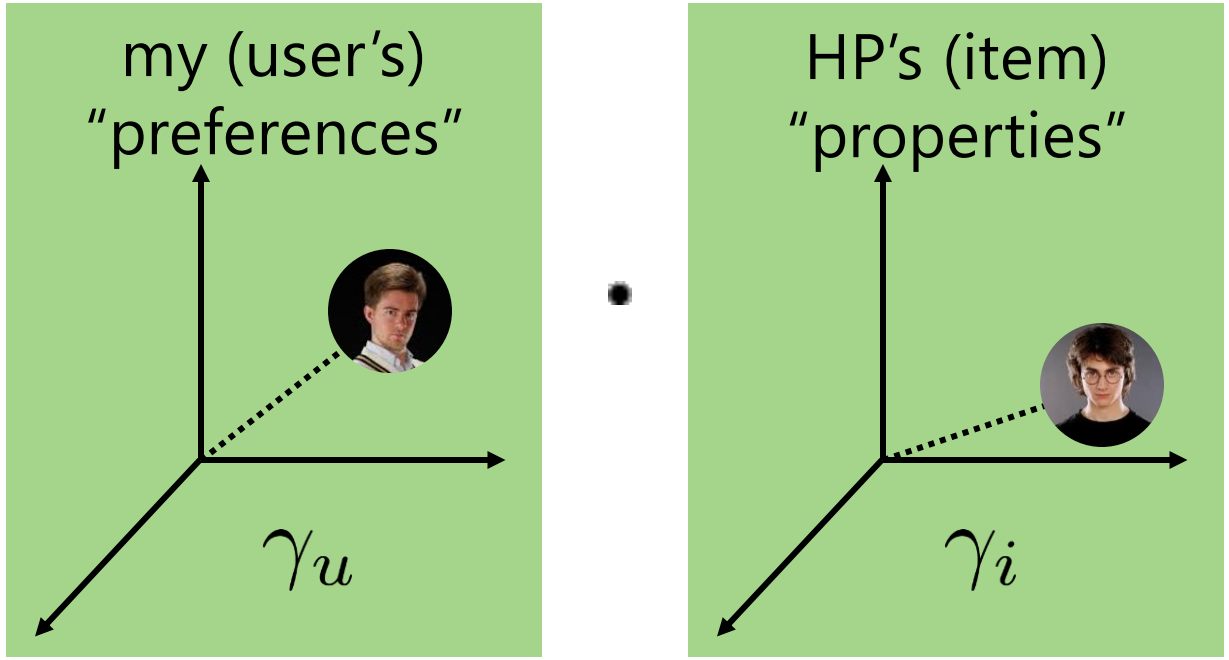
$$R \simeq UV^T$$

K-dimensional representation of each user

Latent-factor models

Let's write this as:

$$f(u,i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$$



Latent-factor models

Let's write this as:

$$f(u,i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$$

Our optimization problem is then

$$\arg\min_{\alpha,\beta,\gamma} \underbrace{\sum_{u,i}(\alpha+\beta_u+\beta_i+\gamma_u\cdot\gamma_i-R_{u,i})^2}_{\text{error}} + \underbrace{\lambda\left[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|_2^2 + \sum_u \|\gamma_u\|_2^2\right]}_{\text{regularizer}}$$

Latent-factor models

$$\arg\min_{\alpha,\beta,\gamma} \frac{1}{N} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|_2^2 + \sum_u \|\gamma_u\|_2^2 \right]$$

Latent-factor models

$$\frac{\partial \text{obj}}{\partial \gamma_{u,k}} = \frac{1}{N} \sum_{i \in I_u} 2\gamma_{i,k}(\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i}) + 2\lambda\gamma_{u,k}$$

Latent-factor models

$$\frac{\partial \text{obj}}{\partial \gamma_{u,k}} = \frac{1}{N} \sum_{i \in I_u} 2\gamma_{i,k}(\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i}) + 2\lambda\gamma_{u,k}$$

- **Note:** applying gradient descent with these terms will **not necessarily** lead to a global optimum
- Need to be careful about initialization (or run with several random restarts) in order to obtain a good solution

Summary of concepts

- Finished our description of latent factor models
- Showed how to optimize the model using gradient descent

On your own...

- Compute the gradient update equations for the remaining terms

Week 2

Capstone project and Recommender Systems

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Implementing a similarity-based recommender

Learning objectives

In this lecture we will...

- Implement a simple recommender system that recommends products based on the Jaccard similarity

Code: Reading the data

First we read the data. Note we use a larger dataset for this exercise, though could use a smaller one if running time is an issue.

In [1]:
```
import gzip
from collections import defaultdict
import random
import numpy
import scipy.optimize
```

In [2]:
```
path = "/home/jmcauley/datasets/mooc/amazon/amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz"
```

In [3]:
```
f = gzip.open(path, 'rt', encoding="utf8")
```

In [4]:
```
header = f.readline()
header = header.strip().split('\t')
```

Code: Reading the data

Our goal is to make recommendations of products based on users' purchase histories. The only information needed to do so is **user and item IDs**

In [5]:
```
dataset = []
```

In [6]:
```
for line in f:
    fields = line.strip().split('\t')
    d = dict(zip(header, fields))
    d['star_rating'] = int(d['star_rating'])
    d['helpful_votes'] = int(d['helpful_votes'])
    d['total_votes'] = int(d['total_votes'])
    dataset.append(d)
```

In [7]:
```
dataset[0]
```

Out[7]:
```
{'marketplace': 'US',
 'customer_id': '45610553',
 'review_id': 'RMDCHWD0Y5OZ9',
 'product_id': 'B00HH62VB6',
 'product_parent': '618218723',
 'product_title': 'AGPtek® 10 Isolated Output 9V 12V 18V Guitar Pedal Board Power Supply Effect Pedals
with Isolated Short Cricuit / Overcurrent Protection',
```

Code: Useful data structures

To perform set intersections/unions efficiently, we first build data structures representing the set of items for each user and users for each item

In [8]:
```
# Useful data structures
```

In [9]:
```
usersPerItem = defaultdict(set)
itemsPerUser = defaultdict(set)
```

U_i

I_u

In [10]:
```
itemNames = {}
```

In [11]:
```
for d in dataset:
    user,item = d['customer_id'], d['product_id']
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    itemNames[item] = d['product_title']
```

Code: Jaccard similarity

The Jaccard similarity implementation follows the definition directly:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In [12]:
```
def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    return numer / denom
```

Recommendation

We want a recommendation function that return **items similar to a candidate item *i*.** Our strategy will be as follows:

- Find the set of users who purchased *i*
- Iterate over all other items other than *i*
- For all other items, compute their similarity with *i (and store it)*
- Sort all other items by (Jaccard) similarity
- Return the most similar

Code: Recommendation

Now we can implement the recommendation function itself:

In [13]:
```
def mostSimilar(i):
    similarities = []
    users = usersPerItem[i]
    for i2 in usersPerItem:
        if i2 == i: continue
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse=True)
    return similarities[:10]
```

$$\text{Jaccard}(U_i, U_j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$$

Code: Recommendation

Next, let's use the code to make a recommendation. The query to the system is just a product ID:

In [14]:
```
dataset[2]
```

Out[14]:
```
{'marketplace': 'US',
 'customer_id': '6111003',
 'review_id': 'RIZR67JKUDBI0',
 'product_id': 'B0006VMBHI',
 'product_parent': '603261968',
 'product_title': 'AudioQuest LP record clean brush',
 'product_category': 'Musical Instruments',
 'star_rating': 3,
 'helpful_votes': 0,
 'total_votes': 1,
 'vine': 'N',
 'verified_purchase': 'Y',
 'review_headline': 'Three Stars',
 'review_body': 'removes dust. does not clean',
 'review_date': '2015-08-31'}
```

In [15]:
```
query = dataset[2]['product_id']
```

Code: Recommendation

Next, let's use the code to make a recommendation. The query to the system is just a product ID:

```
In [16]: mostSimilar(query)

Out[16]: [(0.028446389496717725, 'B00006I5SD'),
          (0.01694915254237288, 'B00006I5SB'),
          (0.015065913370998116, 'B000AJR482'),
          (0.014204545454545454, 'B00E7MVP3S'),
          (0.008955223880597015, 'B001255YL2'),
          (0.008849557522123894, 'B003EIRVO8'),
          (0.008333333333333333, 'B0015VEZ22'),
          (0.008219178082191781, 'B00006I5UH'),
          (0.008021390374331552, 'B00008BWM7'),
          (0.007656967840735069, 'B000H2BC4E')]
```

Code: Recommendation

Finally, let's look at the items that were recommended:

```
In [17]: itemNames[query]

Out[17]: 'AudioQuest LP record clean brush'

In [18]: [itemNames[x[1]] for x in mostSimilar(query)]

Out[18]: ['Shure SFG-2 Stylus Tracking Force Gauge',
          'Shure M97xE High-Performance Magnetic Phono Cartridge',
          'ART Pro Audio DJPRE II Phono Turntable Preamplifier',
          'Signstek Blue LCD Backlight Digital Long-Playing LP Turntable Stylus Force Scale Gauge Tester',
          'Audio Technica AT120E/T Standard Mount Phono Cartridge',
          'Technics: 45 Adaptor for Technics 1200 (SFWE010)',
          'GruvGlide GRUVGLIDE DJ Package',
          'STANTON MAGNETICS Record Cleaner Kit',
          'Shure M97xE High-Performance Magnetic Phono Cartridge',
          'Behringer PP400 Ultra Compact Phono Preamplifier']
```

Recommending more efficiently

Our implementation was not very efficient. The slowest component is the iteration over all other items:

- Find the set of users who purchased *i*
- **Iterate over all other items other than *i***
- For all other items, compute their similarity with *i (and store it)*
- Sort all other items by (Jaccard) similarity
- Return the most similar

This can be done more efficiently as most items will have no overlap

Recommending more efficiently

In fact it is sufficient to iterate over **those items purchased by one of the users who purchased *i***

- Find the set of users who purchased *i*
- **Iterate over all users who purchased *i***
- Build a candidate set from all items those users consumed
- For items in this set, compute their similarity with *i (and store it)*
- Sort all other items by (Jaccard) similarity
- Return the most similar

Code: Faster implementation

Our more efficient implementation works as follows:

In [19]:
```
def mostSimilarFast(i):
    similarities = []
    users = usersPerItem[i]
    candidateItems = set()
    for u in users:
        candidateItems = candidateItems.union(itemsPerUser[u])
    for i2 in candidateItems:
        if i2 == i: continue
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse=True)
    return similarities[:10]
```

Code: Faster recommendation

Which ought to recommend the same set of items, but **much** more quickly:

```
In [20]: mostSimilarFast(query)

Out[20]: [(0.028446389496717725, 'B00006I5SD'),
          (0.01694915254237288, 'B00006I5SB'),
          (0.015065913370998116, 'B000AJR482'),
          (0.014204545454545454, 'B00E7MVP3S'),
          (0.008955223880597015, 'B001255YL2'),
          (0.008849557522123894, 'B003EIRVO8'),
          (0.008333333333333333, 'B0015VEZ22'),
          (0.00821917808219178, 'B00006I5UH'),
          (0.008021390374331552, 'B00008BWM7'),
          (0.007656967840735069, 'B000H2BC4E')]
```

Summary of concepts

- Implemented a similarity-based recommender based on the Jaccard similarity
- Showed how to make our implementation more efficient

On your own...

- Our code recommends *items* that are similar to a given item. Adapt it to recommend *users* who are similar to a given user
- Typcially we want to recommender items similar to one to which a user has already given a high rating (e.g. "you'll like X because you liked Y"). Adapt our code so that it takes as input a given *user*, and recommends items similar to those that user liked

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Using our similarity-based recommender for rating prediction

Learning objectives

In this lecture we will...

- Show how similarity-based recommenders can be used as a heuristic for rating prediction
- Provide code examples to do so

Collaborative filtering for rating prediction

In the previous lecture we provided code to make recommendations based on the **Jaccard similarity**

How can the same ideas be used for rating prediction?

Collaborative filtering for rating prediction

A simple heuristic for rating prediction works as follows:

- The user (*u*)'s rating for an item *i* is a weighted combination of all of their previous ratings for items *j*
- The weight for each rating is given by the Jaccard similarity between *i* and *j*

Collaborative filtering for rating prediction

This can be written as:

$$r(u,i) = \frac{1}{Z} \sum_{j \in I_u \setminus \{i\}} r_{u,j} \cdot \mathrm{sim}(i,j)$$

Normalization constant

All items the user has rated other than i

$$Z = \sum_{j \in I_u \setminus \{i\}} \mathrm{sim}(i,j)$$

Code: Collaborative filtering for rating prediction

Now we can adapt our previous recommendation code to predict ratings

In [22]:
```
# More utility data structures
```

In [23]:
```
reviewsPerUser = defaultdict(list)
reviewsPerItem = defaultdict(list)
```

List of reviews per user and per item

In [24]:
```
for d in dataset:
    user,item = d['customer_id'], d['product_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
```

In [25]:
```
ratingMean = sum([d['star_rating'] for d in dataset]) / len(dataset)
```

In [26]:
```
ratingMean
```

Out[26]: 4.251102772543146

We'll use the mean rating as a baseline for comparison

Code: Collaborative filtering for rating prediction

Our rating prediction code works as follows:

In [27]:
```
def predictRating(user,item):
    ratings = []
    similarities = []
    for d in reviewsPerUser[user]:
        i2 = d['product_id']
        if i2 == item: continue
        ratings.append(d['star_rating'])
        similarities.append(Jaccard(usersPerItem[item],usersPerItem[i2]))
    if (sum(similarities) > 0):
        weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]
        return sum(weightedRatings) / sum(similarities)
    else:
        # User hasn't rated any similar items
        return ratingMean
```

$$r(u,i) = \frac{1}{Z} \sum_{j \in I_u \setminus \{i\}} r_{u,j} \cdot \mathrm{sim}(i,j)$$

Code: Collaborative filtering for rating prediction

As an example, select a rating for prediction:

In [28]:
```
dataset[1]
```

Out[28]:
```
{'marketplace': 'US',
 'customer_id': '14640079',
 'review_id': 'RZSL0BALIYUNU',
 'product_id': 'B003LRN53I',
 'product_parent': '986692292',
 'product_title': 'Sennheiser HD203 Closed-Back DJ Headphones',
 'product_category': 'Musical Instruments',
 'star_rating': 5,
 'helpful_votes': 0,
 'total_votes': 0,
 'vine': 'N',
 'verified_purchase': 'Y',
 'review_headline': 'Five Stars',
 'review_body': 'Nice headphones at a reasonable price.',
 'review_date': '2015-08-31'}
```

In [29]:
```
u,i = dataset[1]['customer_id'], dataset[1]['product_id']
```

In [30]:
```
predictRating(u, i)
```

Out[30]:
```
5.0
```

Code: Collaborative filtering for rating prediction

Similarly, we can evaluate accuracy across the entire corpus:

In [31]:
```
def MSE(predictions, labels):
    differences = [(x-y)**2 for x,y in zip(predictions,labels)]
    return sum(differences) / len(differences)
```

In [32]:
```
alwaysPredictMean = [ratingMean for d in dataset]
```

In [33]:
```
cfPredictions = [predictRating(d['customer_id'], d['product_id']) for d in dataset]
```

In [34]:
```
labels = [d['star_rating'] for d in dataset]
```

In [35]:
```
MSE(alwaysPredictMean, labels)
```

Out[35]: 1.4796142779564334

In [36]:
```
MSE(cfPredictions, labels)
```

Out[36]: 1.6146130004291603

Collaborative filtering for rating prediction

Note that this is just a **heuristic** for rating prediction

- In fact in this case it did *worse* (in terms of the MSE) than always predicting the mean
- We could adapt this to use:

1. A different similarity function (e.g. cosine)
2. Similarity based on users rather than items
3. A different weighting scheme

Summary of concepts

- Showed how similarity-based recommenders can be used to predict ratings
- Provided code for an implementation of this idea

On your own...

- Adapt the code to implement one of the modifications on the previous slide (e.g. cosine, or similarity between users rather than items), and measure its effect on performance

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Implementing a simple (bias-only) latent factor model

Learning objectives

In this lecture we will...

- Begin implementing the latent factor model (starting with a simple bias-only version)
- Introduce another library for gradient descent

Latent factor models

We'll start by implementing a "bias only" model, since its gradient update equations are simpler:

$$\arg\min_{\alpha,\beta} \frac{1}{N} \sum_{u,i} (\alpha + \beta_u + \beta_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2 \right]$$

With this model, our tasks will be to:

1. Write down code for the gradient equations
2. Use a gradient descent library to optimize the model

Code: Latent factor models

We'll start by defining some basic terms. Note that code to determine some of these values (e.g. the rating mean) is given in previous lectures

```
In [38]: N = len(dataset)
         nUsers = len(reviewsPerUser)
         nItems = len(reviewsPerItem)
         users = list(reviewsPerUser.keys())
         items = list(reviewsPerItem.keys())

In [39]: alpha = ratingMean

In [40]: userBiases = defaultdict(float)
         itemBiases = defaultdict(float)
```

Alpha and beta (userbiases) are parameters we'll fit. This code sets their **initial values** (alpha to the mean rating, and beta_u/beta_i to zero)

Code: Latent factor models

Our prediction function in this case just implements the bias only model:

$$f(u, i) = \alpha + \beta_u + \beta_i$$

user item user bias item bias

In [41]:
```
def prediction(user, item):
    return alpha + userBiases[user] + itemBiases[item]
```

Code: Latent factor models

The first complex function to implement is this "unpack" function. The gradient descent library we'll use expects a single vector of parameters (θ), which we have to unpack to produce alpha and beta:

In [42]:
```
def unpack(theta):
    global alpha
    global userBiases
    global itemBiases
    alpha = theta[0]
    userBiases = dict(zip(users, theta[1:nUsers+1]))
    itemBiases = dict(zip(items, theta[1+nUsers:]))
```

Code: Latent factor models

The next function (also required by the gradient descent library) just implements the full cost function:

$$\frac{1}{N}\sum_{u,i}(\alpha+\beta_u+\beta_i-R_{u,i})^2+\lambda\left[\sum_u\beta_u^2+\sum_i\beta_i^2\right]$$

In [43]:
```
def cost(theta, labels, lamb):
    unpack(theta)
    predictions = [prediction(d['customer_id'], d['product_id']) for d in dataset]
    cost = MSE(predictions, labels)
    print("MSE = " + str(cost))
    for u in userBiases:
        cost += lamb*userBiases[u]**2
    for i in itemBiases:
        cost += lamb*itemBiases[i]**2
    return cost
```

Code: Latent factor models

Next we implement the derivative function, which has a corresponding derivative term for each parameter:

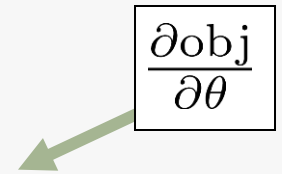
In [44]:
```
def derivative(theta, labels, lamb):
    unpack(theta)
    N = len(dataset)
    dalpha = 0
    dUserBiases = defaultdict(float)
    dItemBiases = defaultdict(float)
    for d in dataset:
        u,i = d['customer_id'], d['product_id']
        pred = prediction(u, i)
        diff = pred - d['star_rating']
        dalpha += 2/N*diff
        dUserBiases[u] += 2/N*diff
        dItemBiases[i] += 2/N*diff
    for u in userBiases:
        dUserBiases[u] += 2*lamb*userBiases[u]
    for i in itemBiases:
        dItemBiases[i] += 2*lamb*itemBiases[i]
    dtheta = [dalpha] + [dUserBiases[u] for u in users] + [dItemBiases[i] for i in items]
    return numpy.array(dtheta)
```

$$\frac{\partial \text{obj}}{\partial \beta_u} = \frac{1}{N}\sum_{u,i} 2(\alpha + \beta_u + \beta_i - R_{u,i}) + 2\lambda\beta_u$$

Code: Latent factor models

Note that the function must also **return** the derivatives as a single vector:

In [44]:
```
def derivative(theta, labels, lamb):
    unpack(theta)
    N = len(dataset)
    dalpha = 0
    dUserBiases = defaultdict(float)
    dItemBiases = defaultdict(float)
    for d in dataset:
        u,i = d['customer_id'], d['product_id']
        pred = prediction(u, i)
        diff = pred - d['star_rating']
        dalpha += 2/N*diff
        dUserBiases[u] += 2/N*diff
        dItemBiases[i] += 2/N*diff
    for u in userBiases:
        dUserBiases[u] += 2*lamb*userBiases[u]
    for i in itemBiases:
        dItemBiases[i] += 2*lamb*itemBiases[i]
    dtheta = [dalpha] + [dUserBiases[u] for u in users] + [dItemBiases[i] for i in items]
    return numpy.array(dtheta)
```



Code: Latent factor models

Before we test this model, let's see the accuracy of always predicting the mean for comparison:

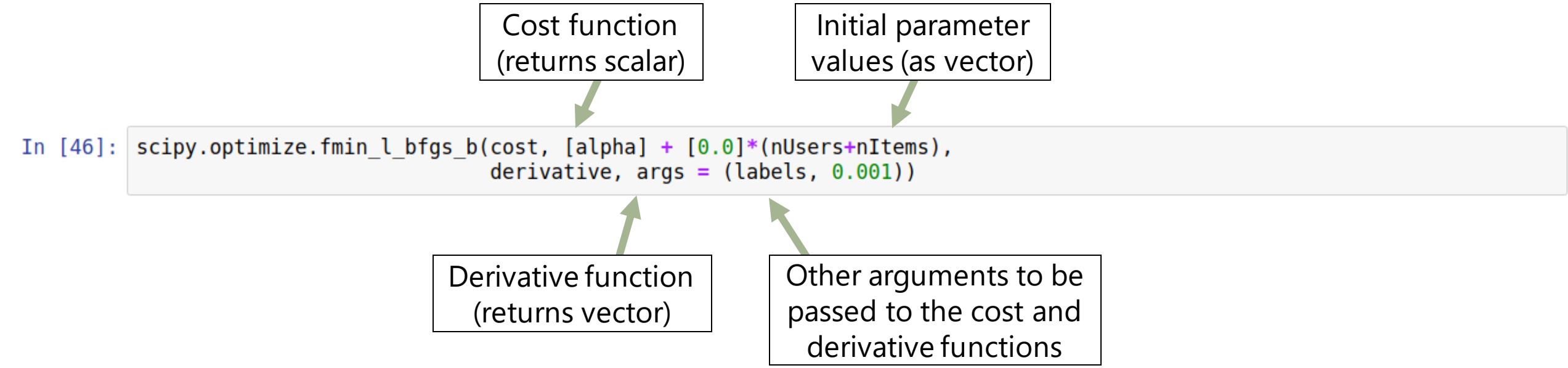
```
In [45]: MSE(alwaysPredictMean, labels)
Out[45]: 1.4796142779564334
```

Code: Gradient descent with LBFGS

The gradient descent library we'll use is called lbfgs

(see "Updating Quasi-Newton Matrices with Limited Storage", Nocedal, 1980)

It has a simple interface which works as follows:



Code: Gradient descent with LBFGS

The code runs for several iterations, and returns a value of theta (a vector of parameters) after convergence. Convergence criteria etc. can be set with additional arguments

```
In [46]: scipy.optimize.fmin_l_bfgs_b(cost, [alpha] + [0.0]*(nUsers+nItems),
                                      derivative, args = (labels, 0.001))
```

```
MSE = 1.4796142779564334
MSE = 1.468686355953835
MSE = 2.696168718200339
MSE = 1.4681419018494128
MSE = 1.4523523347391176
MSE = 1.4513575397272926
MSE = 1.4476987674764965
MSE = 1.4421925605950903
MSE = 1.4415262672088038
MSE = 1.4413460037417305
MSE = 1.4413976122440515
MSE = 1.441406601709925
```

```
Out[46]: (array([ 4.24278450e+00,  8.29337738e-04, -2.44445091e-03, ...,
                  8.31080528e-04,  1.66853365e-03,  1.10894814e-03]),
          1.4574364057349303,
          {'funcalls': 12,
           'grad': array([-4.52665780e-07, -4.81839660e-09,  1.79803085e-08, ...,
                  -5.05371183e-09, -1.09454104e-08, -4.95190851e-09]),
           'nit': 9,
           'task': b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL',
           'warnflag': 0})
```

Gradient descent with LBFGS

This is a general-purpose gradient descent algorithm, which simply requires that we provide a cost function ($f(x)$), and a derivative function ($f'(x)$). It can be (relatively) straightforwardly adapted to other gradient descent problems.

(we could also have used Tensorflow for the same task)

In the next lecture we'll extend this code to implement the full latent-factor model

Summary of concepts

- Implemented a bias-only version of a latent factor model

On your own...

- (HARD) Try implementing the same model using Tensorflow

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Implementing a latent factor model (Part 2)

Learning objectives

In this lecture we will...

- Complete our implementation of a full-fledged latent factor model

Latent factor models

In this lecture we'll extend our implementation from the previous lecture to implement the complete latent factor model:

$$\arg\min_{\alpha,\beta,\gamma} \frac{1}{N} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|_2^2 + \sum_u \|\gamma_u\|_2^2 \right]$$

Again our tasks will be to:

1. Write down code for the gradient equations
2. Use a gradient descent library to optimize the model

Code: Latent factor models

Again we start with a few definitions/utilities:

In [48]:
```
alpha = ratingMean
```

In [49]:
```
userBiases = defaultdict(float)
itemBiases = defaultdict(float)
```

In [50]:
```
userGamma = {}
itemGamma = {}
```

In [51]:
```
K = 2
```

Number of latent factors (i.e., dimensionality of gamma)

In [52]:
```
for u in reviewsPerUser:
    userGamma[u] = [random.random() * 0.1 - 0.05 for k in range(K)]
```

In [53]:
```
for i in reviewsPerItem:
    itemGamma[i] = [random.random() * 0.1 - 0.05 for k in range(K)]
```

Code: Latent factor models

Again we need a method to convert from a flat parameter vector (θ), to our parameters (α, β, and γ)

In [54]:
```
def unpack(theta):
    global alpha
    global userBiases
    global itemBiases
    global userGamma
    global itemGamma
    index = 0
    alpha = theta[index]
    index += 1
    userBiases = dict(zip(users, theta[index:index+nUsers]))
    index += nUsers
    itemBiases = dict(zip(items, theta[index:index+nItems]))
    index += nItems
    for u in users:
        userGamma[u] = theta[index:index+K]
        index += K
    for i in items:
        itemGamma[i] = theta[index:index+K]
        index += K
```

Code: Latent factor models

Other utility functions, and our updated prediction function:

In [55]:
```
def inner(x, y):
    return sum([a*b for a,b in zip(x,y)])
```

In [56]:
```
def prediction(user, item):
    return alpha + userBiases[user] + itemBiases[item] + inner(userGamma[user], itemGamma[item])
```

Code: Latent factor models

Our updated cost function:

In [57]:
```
def cost(theta, labels, lamb):
    unpack(theta)
    predictions = [prediction(d['customer_id'], d['product_id']) for d in dataset]
    cost = MSE(predictions, labels)
    print("MSE = " + str(cost))
    for u in users:
        cost += lamb*userBiases[u]**2
        for k in range(K):
            cost += lamb*userGamma[u][k]**2
    for i in items:
        cost += lamb*itemBiases[i]**2
        for k in range(K):
            cost += lamb*itemGamma[i][k]**2
    return cost
```

Code: Latent factor models

And our updated derivative function (which is now much more complicated):

In [58]:
```
def derivative(theta, labels, lamb):
    unpack(theta)
    N = len(dataset)
    dalpha = 0
    dUserBiases = defaultdict(float)
    dItemBiases = defaultdict(float)
    dUserGamma = {}
    dItemGamma = {}
    for u in reviewsPerUser:
        dUserGamma[u] = [0.0 for k in range(K)]
    for i in reviewsPerItem:
        dItemGamma[i] = [0.0 for k in range(K)]
    for d in dataset:
        u,i = d['customer_id'], d['product_id']
        pred = prediction(u, i)
        diff = pred - d['star_rating']
        dalpha += 2/N*diff
        dUserBiases[u] += 2/N*diff
        dItemBiases[i] += 2/N*diff
        for k in range(K):
            dUserGamma[u][k] += 2/N*itemGamma[i][k]*diff
            dItemGamma[i][k] += 2/N*userGamma[u][k]*diff
    for u in userBiases:
        dUserBiases[u] += 2*lamb*userBiases[u]
        for k in range(K):
            dUserGamma[u][k] += 2*lamb*userGamma[u][k]
    for i in itemBiases:
        dItemBiases[i] += 2*lamb*itemBiases[i]
        for k in range(K):
            dItemGamma[i][k] += 2*lamb*itemGamma[i][k]
    dtheta = [dalpha] + [dUserBiases[u] for u in users] + [dItemBiases[i] for i in items]
    for u in users:
        dtheta += dUserGamma[u]
    for i in items:
        dtheta += dItemGamma[i]
    return numpy.array(dtheta)
```

Code: Latent factor models

Finally we can run the model, and observe its performance:

```
In [60]: scipy.optimize.fmin_l_bfgs_b(cost, [alpha] + # Initialize alpha
                                       [0.0]*(nUsers+nItems) + # Initialize beta
                                       [random.random() * 0.1 - 0.05 for k in range(K*(nUsers+nItems))], # Gamma
                                derivative, args = (labels, 0.001))
```

```
MSE = 1.479608415450981
MSE = 1.4775862084545475
MSE = 1.4700816865700903
MSE = 222.53787811707255
MSE = 1.476000685456728
MSE = 1.442834314114482
MSE = 1.437020715186501
MSE = 1.43888307457425
MSE = 1.4401943730605336
MSE = 1.4413020656548463
MSE = 1.44139215463875
MSE = 1.4413770550547866
MSE = 1.4413880297947146
```

```
Out[60]: (array([ 4.24278456e+00, -2.47327688e-03,  8.54347926e-04, ...,
                  9.32933328e-07, -3.73487662e-08, -2.54208102e-07]),
          1.457436360990594,
          {'funcalls': 13,
           'grad': array([-4.99544482e-07, -3.70588802e-10, -9.13958353e-10, ...,
                   1.86010695e-09, -7.46646010e-11, -5.08283939e-10]),
           'nit': 10,
           'task': b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL',
           'warnflag': 0})
```

(though note that we could be overfitting!)

Summary of concepts

- Completed our implementation of the latent factor model

On your own...

- Try optimizing the code to make use of numpy to be more efficient
- Experiment with different initialization strategies for alpha, beta, and gamma
- Experiment with different values of the parameter K

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Description of Capstone Tasks (Part 1)

Learning objectives

In this lecture we will...

- Describe the various components and tasks of the **capstone project**

Capstone tasks

The Capstone project requires you to:

1. **Harness your knowledge** of machine learning, evaluation, and feature design
2. **Implement four practical tasks** on a real-world dataset

Capstone tasks

The dataset you are given is a relatively large set of Amazon **musical instrument** reviews

- The code stub already separates the data into “training” and “test portions”
- You have to implement techniques that lead to good performance **on the test set**, but which are based only on data from the **training set**
- In other words, you have to implement your own training/validation/test pipeline

Capstone tasks

The **four tasks consist of the following:**

1. A basic **data processing** task
2. A **Classification** task
3. A **Regression** task
4. A **Recommender Systems** task

Task 1: Data processing

For the **data processing task,** you must implement a variety of simple functions to compute basic statistics of the data. E.g.:

- How many unique users are there in the dataset?
- What is the average rating?
- What fraction of reviews are verified?

- For each of these tasks you must fill in a function stub

Task 2: Classification

For the **classification task,** you must **predict whether an Amazon review corresponds to a verified purchase.**

- This is a **binary classification task**
- The **main challenge** in this task is that the data are **imbalanced** – i.e., most reviews correspond to verified purchases
- Thus we will use a balanced evaluation metric (the BER)
- So, you must select classification techniques that lead to good performance on this evaluation metric
- Your method should beat a simple baseline that performs logistic regression based on the length and rating of the review

Task 3: Regression

For the **regression task,** you must use word features (and other features) to perform **sentiment analysis**

- This is a **regression task** (rating prediction)
- The **main challenge** in this task is to properly avoid **overfitting,** since you will be using high-dimensional features
- To do this, you will have to carefully implement a train/validation/test pipeline
- You will also have to carefully engineer your features to consider the different choices when pre-processing text
- You should beat a simple baseline that considers the 100 most popular words only

Task 4: Recommendation

For the **recommendation task,** you must predict ratings that users will give to items

- This is a **recommender systems** task (predict a rating given a user and an item)
- The **main challenge** in this task is to **correctly implement a complex model**
- Again you will have to be careful about overfitting, as well as initialization
- Your solution should outperform a simple bias-only model

Capstone project: evaluation

For all tasks, your goal is to beat the baselines given **on the test set**

- Beating these solutions on the **training set** should be easy – you can make small modifications to the existing techniques.
- However these performance gains may not translate well to the test set unless you are careful about **overfitting** and correctly implementing a validation pipeline
- To do so will require leveraging several ideas from throughout this Specialization

Summary of concepts

- Introduced the Capstone project for Course 4

Week 3

Web Server Frameworks for Python

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Introduction to Web Frameworks

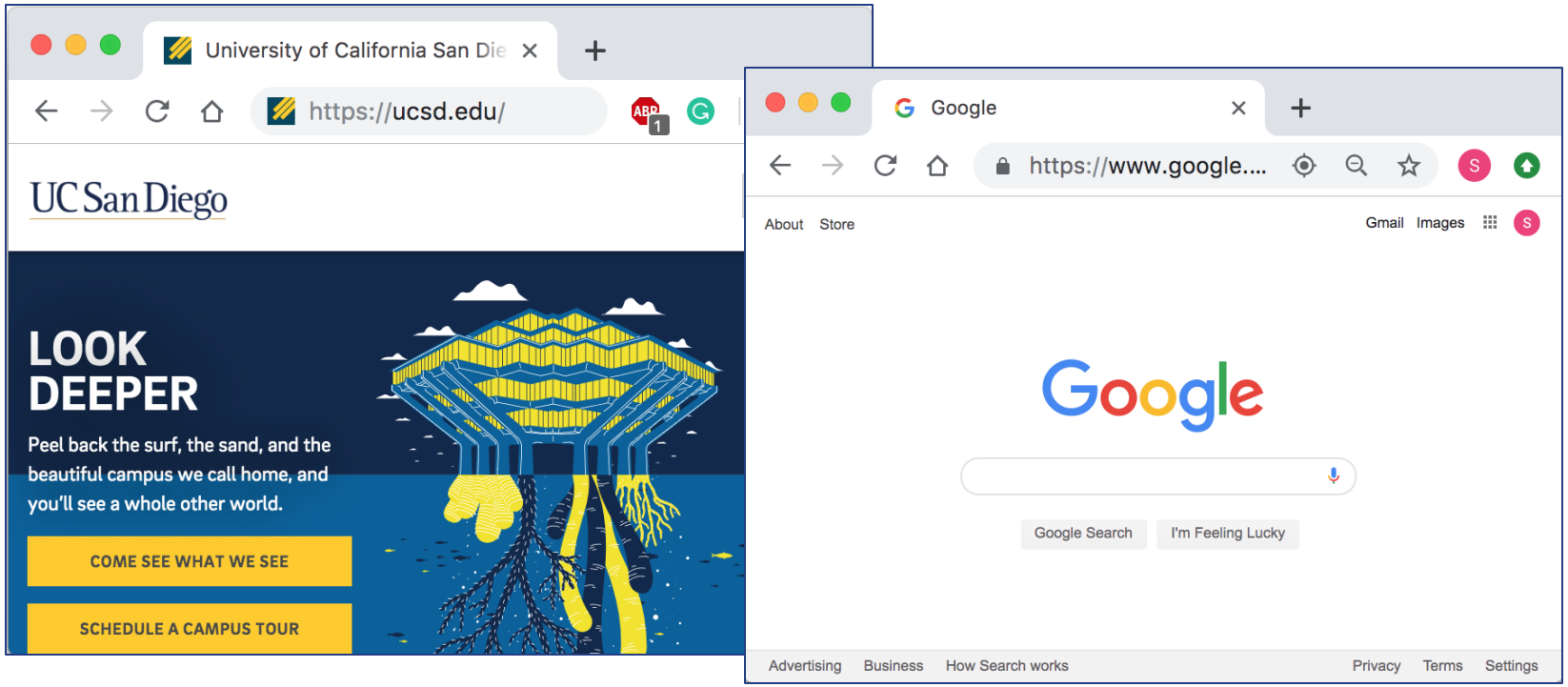
Learning objectives

In this lecture we will learn…

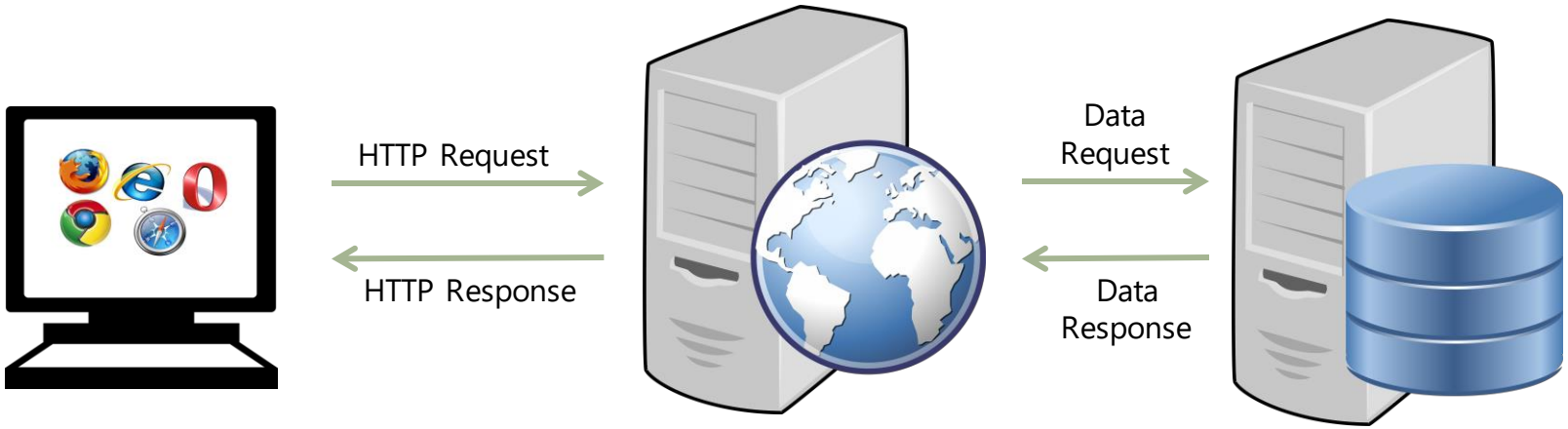
- Background on server-side web frameworks

Web Terminology

- Web page: a document
- Website: webpages + multimedia + domain name

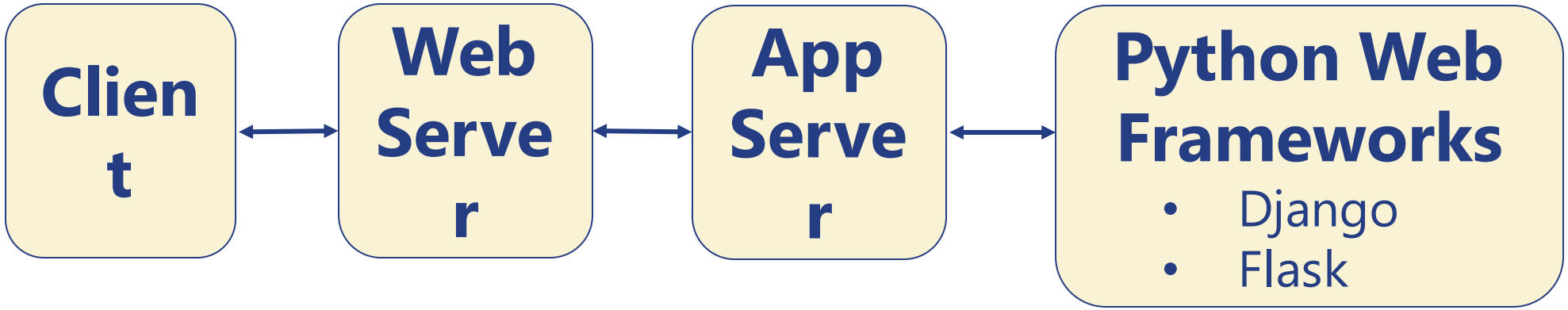


Web Server and Communication Protocol



Some examples….

Python Web Frameworks



Summary of concepts

- Basic terminology and basic architecture of web applications
  - Webpage
  - Website
  - Web Server
- What a web framework means

Django: A Python Web framework

Course: Learn to build a Web Application using Django

Lecture: A Short Introduction to Django

Learning objectives

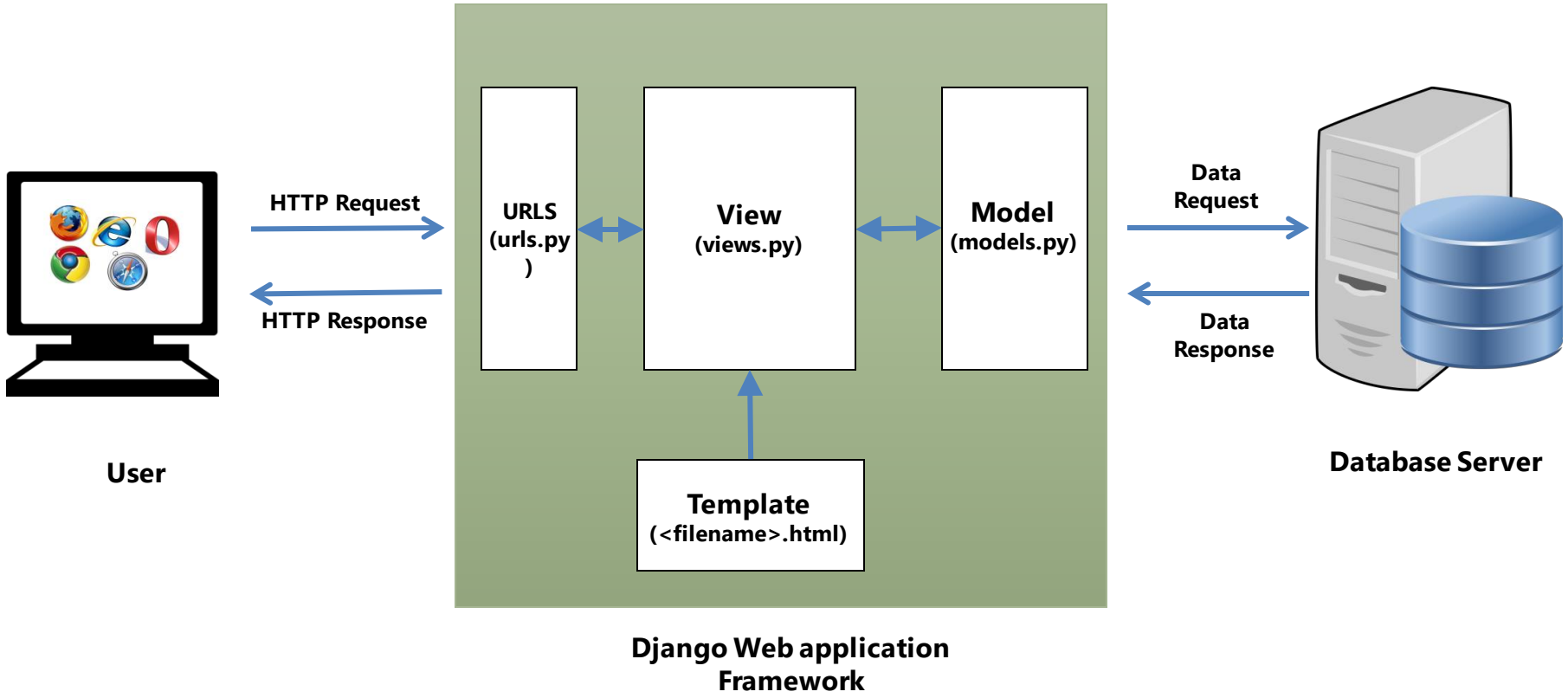
In this lecture we will learn…

- What is Django?
- Django Architecture
- Why Django
- Real world companies built with Django Framework

Django

- Python-based high level web framework
- Free and open-source
- Model-View-Template (MVT) Design Pattern

Django MVT Architecture



Advantages - Why Django?

- Loose coupling between Model, View and Template
- Fast and easy web-development
- Library of pre-built packages
- Security
- Scalability
- Versatile

Using Django Framework

Summary of concepts

- **Described Django web framework architecture, main features, and Django based popular web application platforms.**

Flask: A Python Web framework

Course: Learn to build a Web Application using Django

Lecture: A Short Introduction to Flask

Learning objectives

In this lecture we will learn…

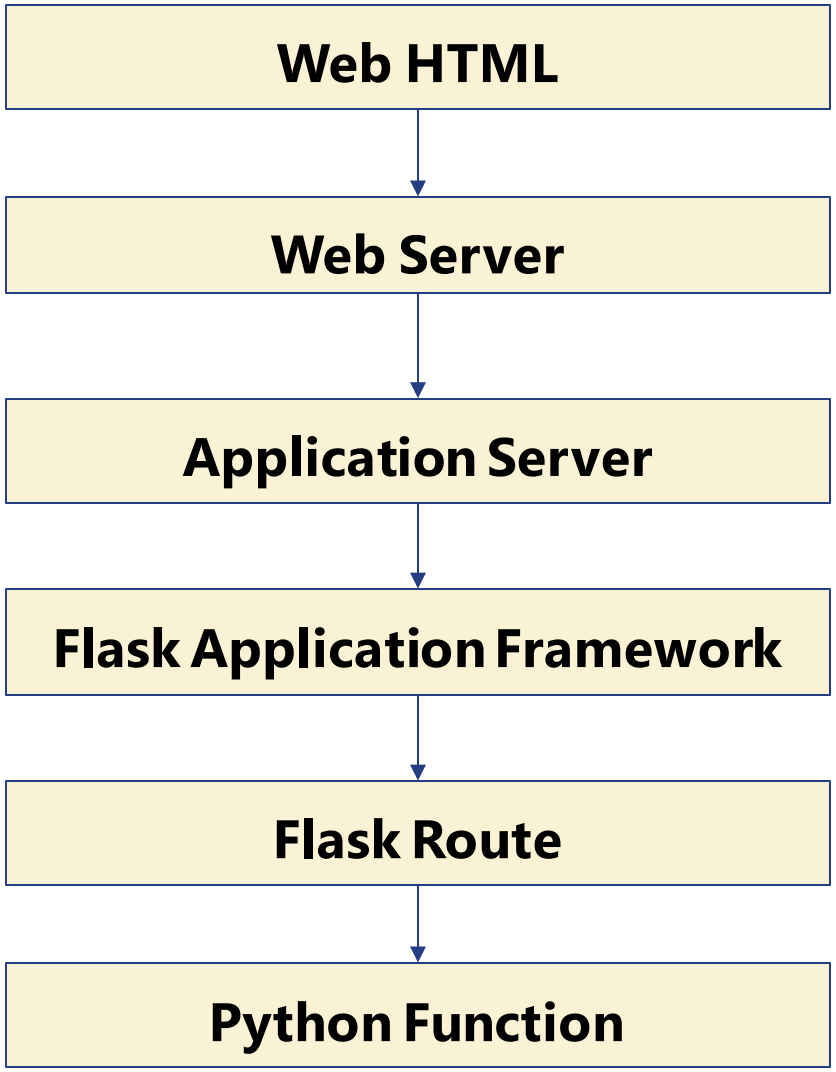
- What is Flask?
- Flask Architecture
- A data product in Flask using our models

Flask

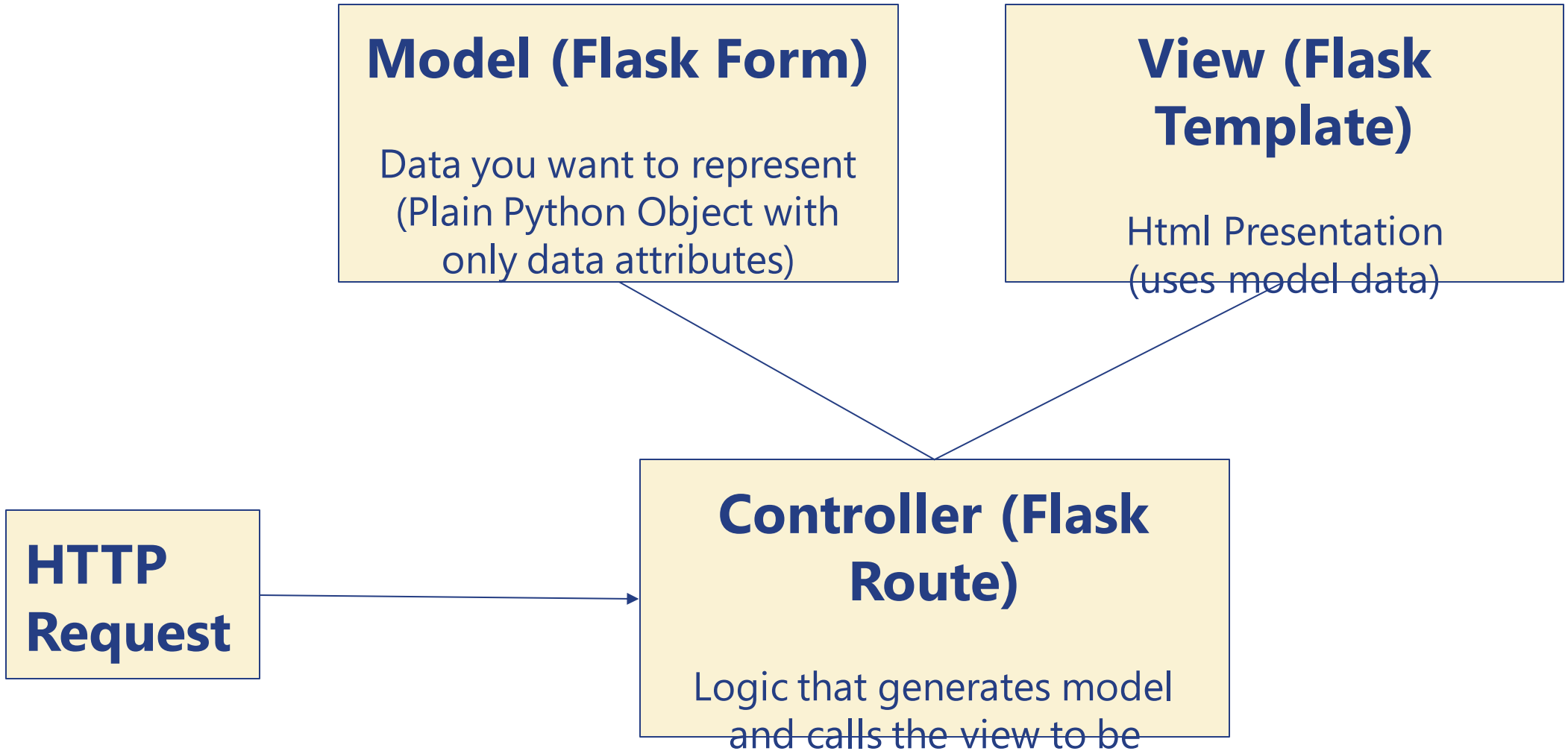
- Micro-framework
- Lightweight and secure

- Some Flask-powered websites
  - Netflix
  - Lyft
  - Uber
  - Airbnb
  - …

Flask Application Architecture



Model-View-Controller (MVC) Architecture



Start Flask

Main Flask Object

```
from flask import Flask

app = Flask(__name__)

if __name__ == '__main__':

    app.run(debug=True, host='0.0.0.0')
```

Flask Routes

HTTP Route, can be anything you want, this is for http://localhost/

HTTP Verb (GET, POST, PUT, PATCH, DELETE)

Python function to run when the / route is called

```
@app.route('/', methods=['GET'])

def index():

    return 'hello from flask!'
```

Object that hold model code

```
mlModel = SimilarItemsMlModel()

@app.route('/items/similar', methods=['GET', 'POST'])

def similar_item_form():

    form = SimilarItemsForm()

    if request.method == 'POST':

        form.similarItems =
mlModel.get_similar_items(form.item_id.data)

    return render_template('similar.html', form=form)
```

Object that holds form data

Get similar items if POST

Render html given form

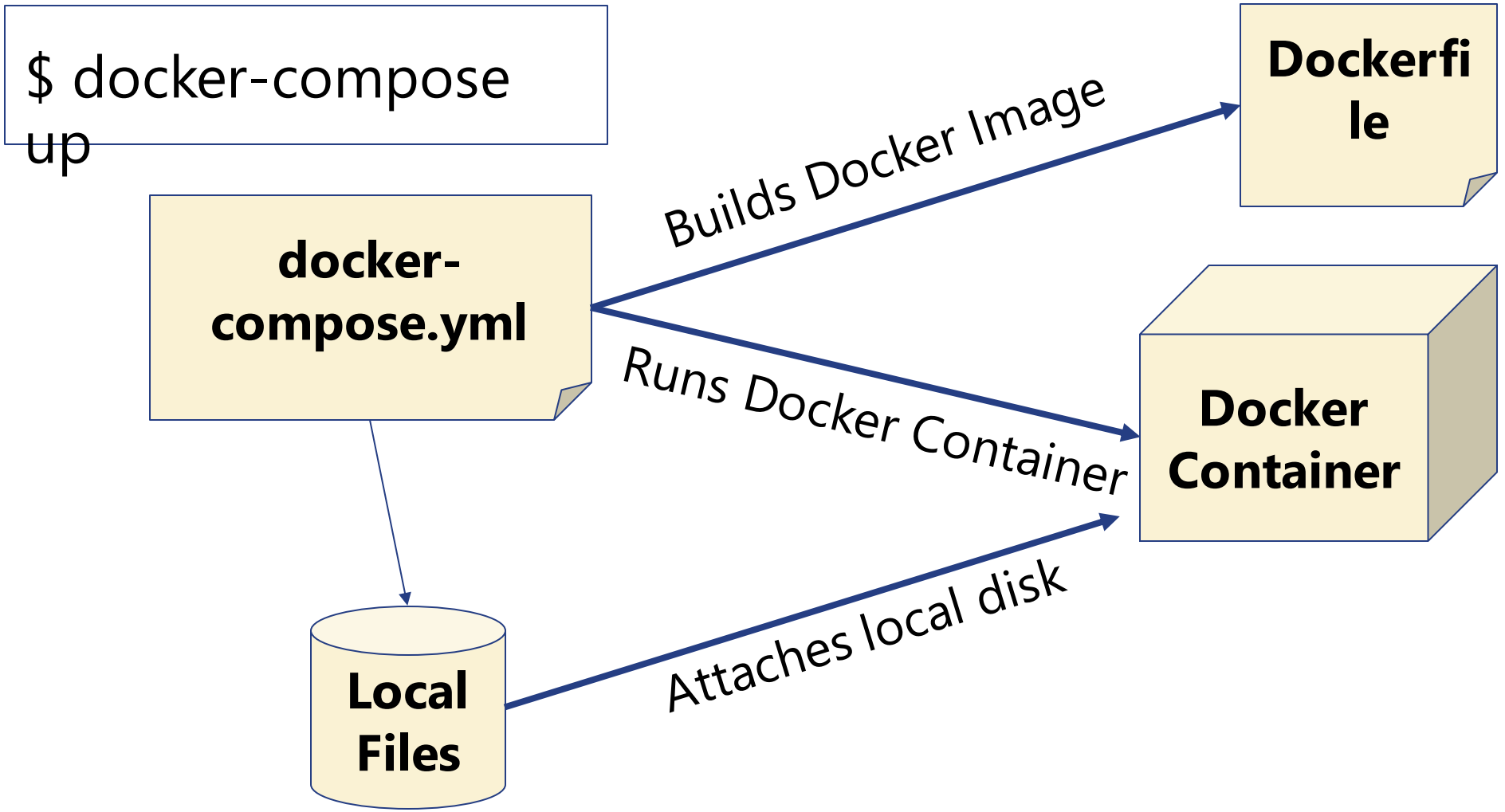
Flask Html Templates

Python loop logic

```
<ul>

{% for item in data['items'] %}

<li>{{ item['product_id'] }}: {{ item['product_title'] }}</li>

{% endfor %}

</ul>
```

Print Python data

Docker & Docker-Compose



Summary of concepts

- **Described Flask web framework architecture, main features, and a Flask example for recommender models we developed**

Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Description of Capstone Tasks (Part 2)