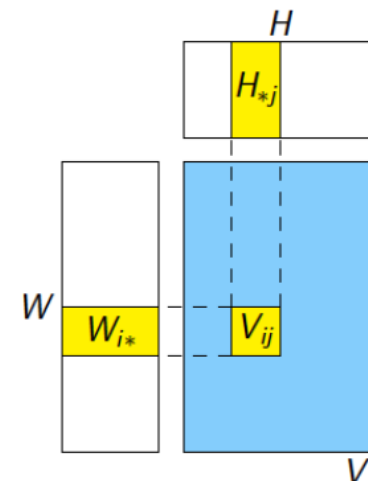
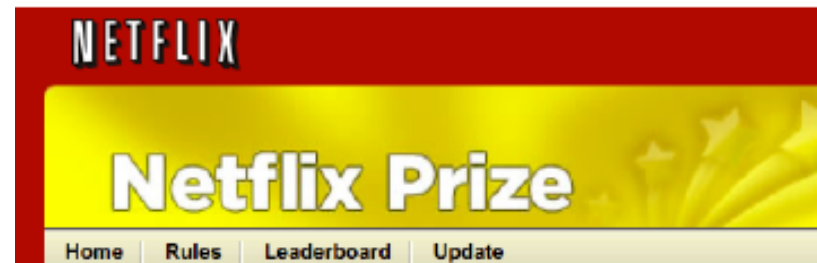
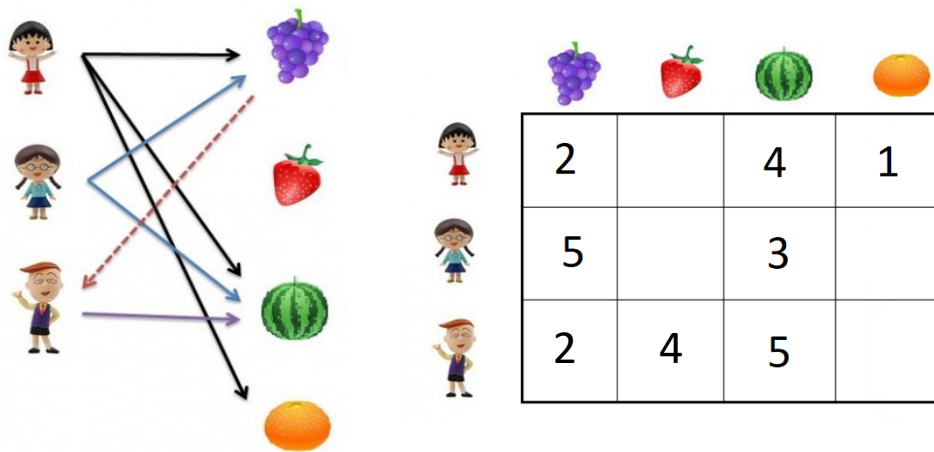


Matrix Factorization for Recommendation Systems



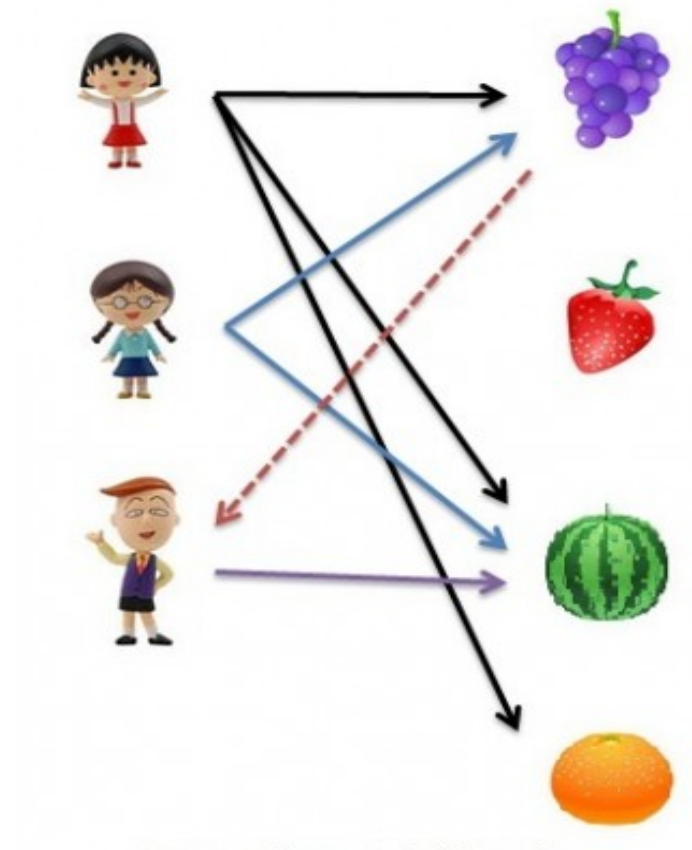
Many ideas/slides attributable to:
Liping Liu (Tufts), Emily Fox (UW)
Matt Gormley (CMU)

Prof. Mike Hughes

Matrix Factorization Objectives (day 23)

- New Task: Recommendation
 - Many “users”, many “items”
 - Predict which users will like each item
- Collaborative filtering
 - Unsupervised learning problem
 - Latent Factor model (Reading: Koren et al)
 - Training algorithm: Stochastic gradient descent (SGD)
- How to do gradients: Automatic differentiation
 - Python package: autograd

Recommendation Task: Which users will like which items?



Need recommendation everywhere

Google

amazon

LinkedIn

ebay

The New York Times

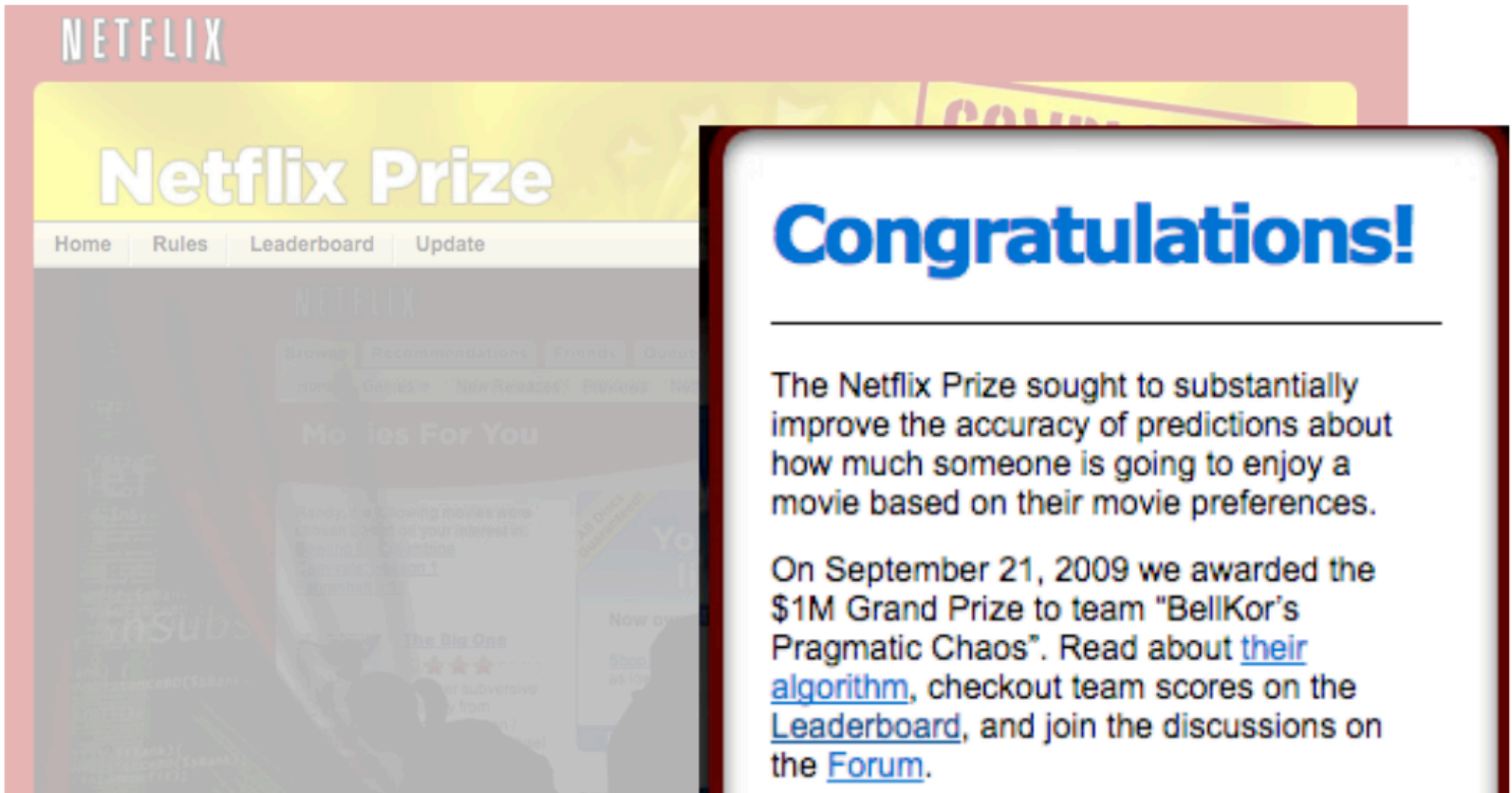
yelp

trivago



airbnb

Motivation: Netflix Prize



The image is a composite of two screenshots. The background is a screenshot of the Netflix Prize website, which features a yellow header with the 'Netflix Prize' title and navigation links like 'Home', 'Rules', 'Leaderboard', and 'Update'. Below this, there's a section for 'Movies For You' with various movie recommendations. Overlaid on the right side of the website screenshot is a white rectangular box with a dark red border. This box contains a large blue 'Congratulations!' heading, followed by a paragraph explaining the prize's goal to improve movie prediction accuracy, and another paragraph announcing the \$1M Grand Prize awarded to the 'BellKor's Pragmatic Chaos' team on September 21, 2009. The announcement includes links to 'their algorithm', the 'Leaderboard', and the 'Forum'.

Netflix Prize

Home Rules Leaderboard Update

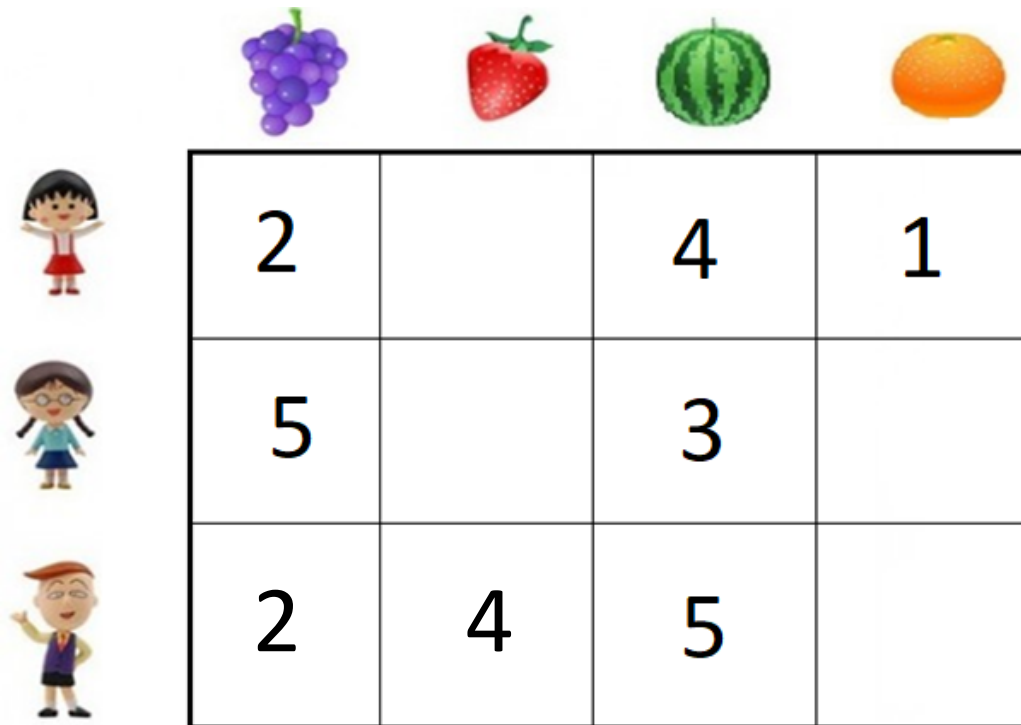
Congratulations!








The Netflix Prize sought to substantially improve the accuracy of predictions about how much someone is going to enjoy a movie based on their movie preferences.

On September 21, 2009 we awarded the \$1M Grand Prize to team "BellKor's Pragmatic Chaos". Read about [their algorithm](#), checkout team scores on the [Leaderboard](#), and join the discussions on the [Forum](#).

Observed data

- The “value”, “utility”, or “rating” of items to users
 - In practice, very sparse, many entries unknown



				
	2		4	1
	5		3	
	2	4	5	

Task: Recommendation








Supervised
Learning

Content filtering

Unsupervised
Learning

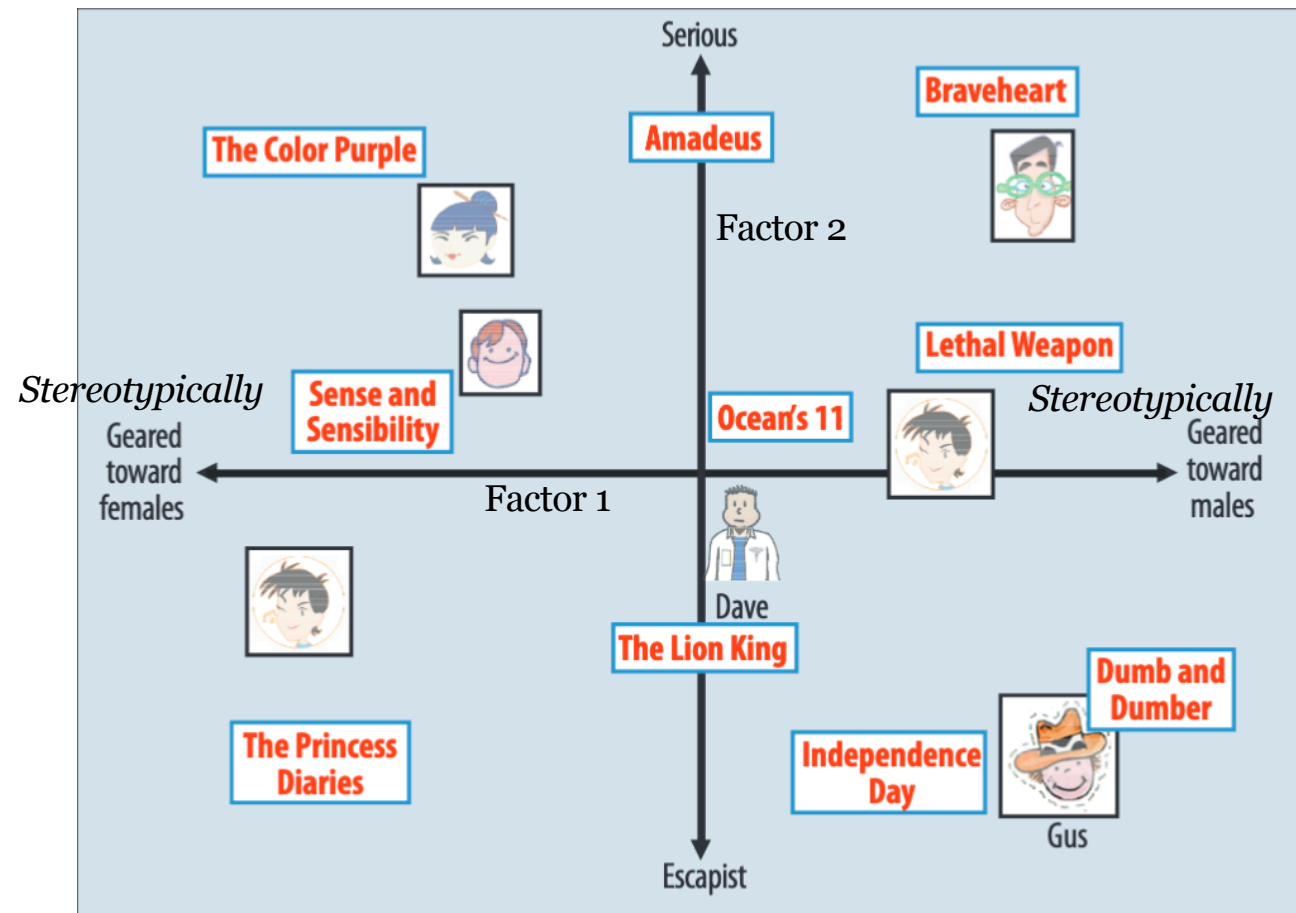
Collaborative filtering

Reinforcement
Learning

				
	2	?	4	1
	5		3	
	2	4	5	

Collaborative filtering via Latent Factor Models

Latent Factor Methods for Collaborative Filtering



Assumption:

- Both movies and users can be explained via a low-dimensional space

Latent Factor Recommendation

To find new movies to recommend to Joe

- 1) Find Joe's embedding vector in the learned "factor" space
- 2) Recommend movies with similar embedding vectors

Fig. Credit: Koren et al. '09

Cartoon View of Matrix Factorization with 2 latent factors

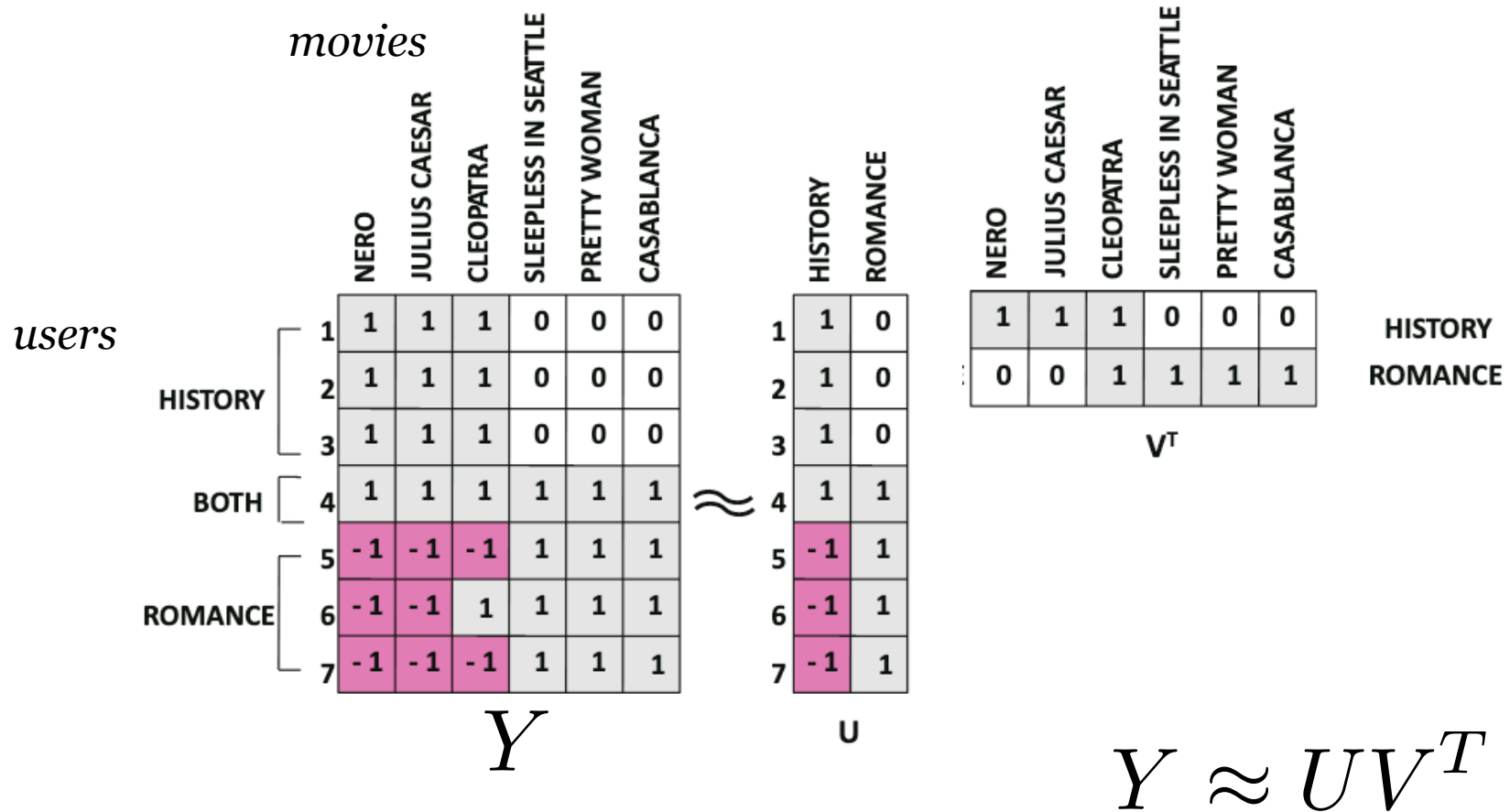
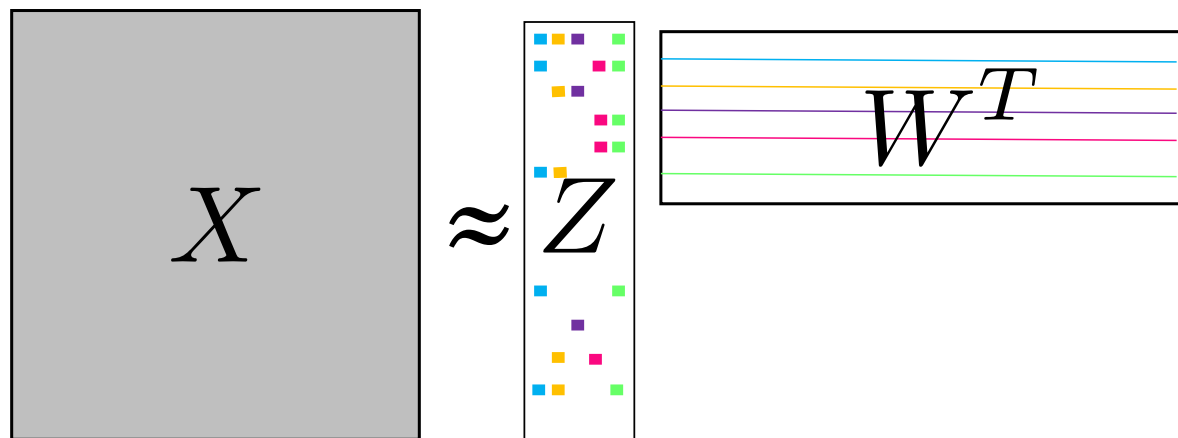


Fig. Credit: Aggarwal 2016
By way of M. Gormley

Recall: PCA as Matrix Factorization



Compared to PCA, Latent Factor models (LF) for recommendation are

Similar

- Use a K -dimensional latent space
- Use linear inner products to do "prediction"
- Measure reconstruction cost with mean-squared error

Different

- PCA required orthogonality constraints on W , while LF is less strict
- LF interprets each column of data as an "item", not a "feature dimension"
- PCA requires fully observed data, the LF models we'll develop can handle realistic missingness patterns

Latent Factor Model: Prediction

Assume a known number of factors K

- User i represented by vector $u_i \in \mathbb{R}^K$
- Item j represented by vector $v_j \in \mathbb{R}^K$

We predict the rating y for user-item pair (i,j) as:

$$\hat{y}_{ij} = \underbrace{\sum_{k=1}^K u_{ik} v_{jk}}_{u_i^T v_j}$$

Intuition:

Two items with similar v vectors
get similar ratings from the same user;
Two users with similar u vectors
give similar ratings to the same item

Inner product of:

- User vector
- Item vector

Latent Factor Model: Training

- Find parameters that minimize squared error

$$\min_{u_i \in \mathbb{R}^K, v_j \in \mathbb{R}^K} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - u_i^T v_j)^2$$

Which pairs do we use?

Squared error between

- True rating
- Predicted rating

- How to optimize?
 - **Stochastic gradient descent**
 - Use random minibatch of user-item pairs

Supervised Learning vs Unsupervised Matrix Completion

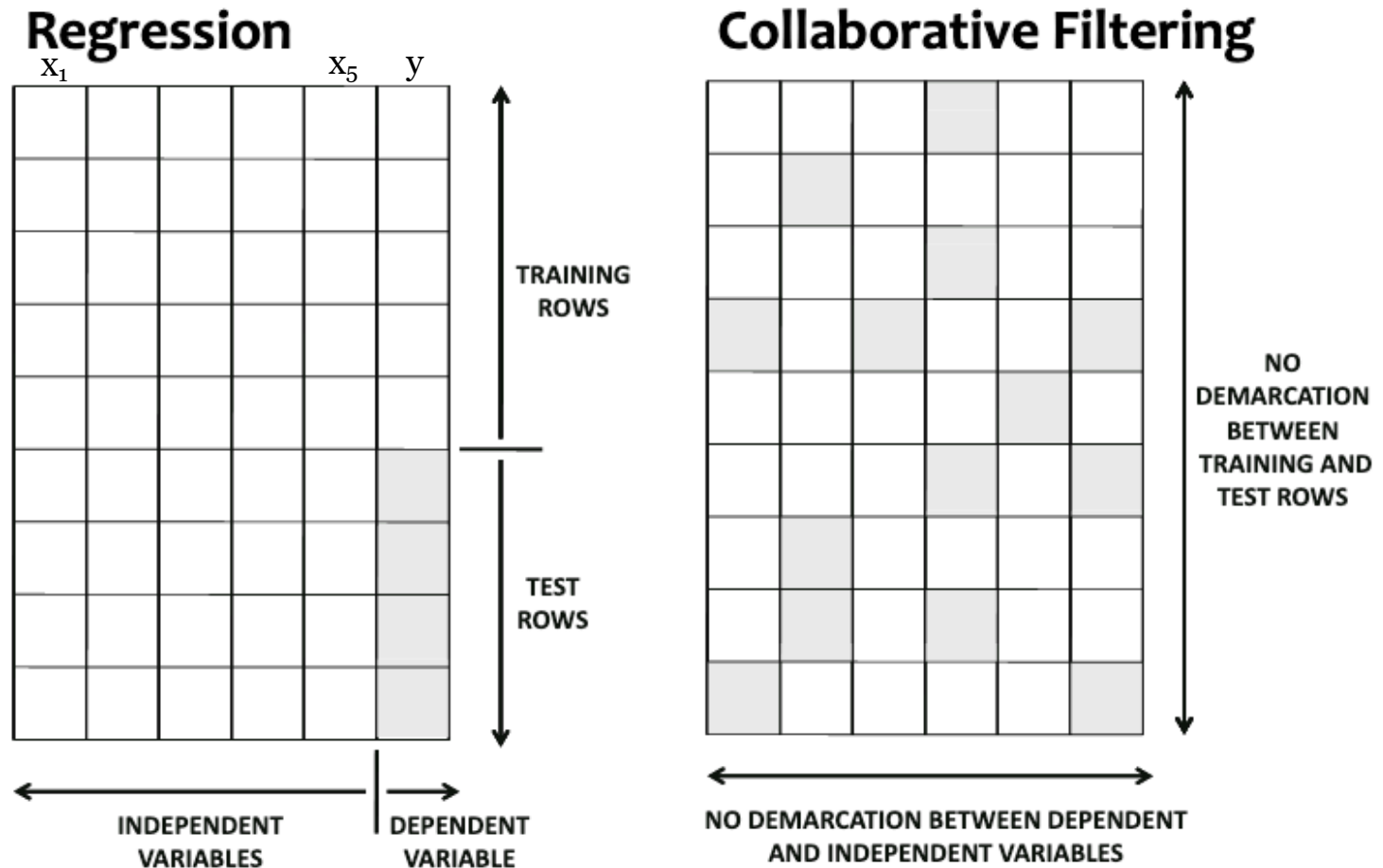
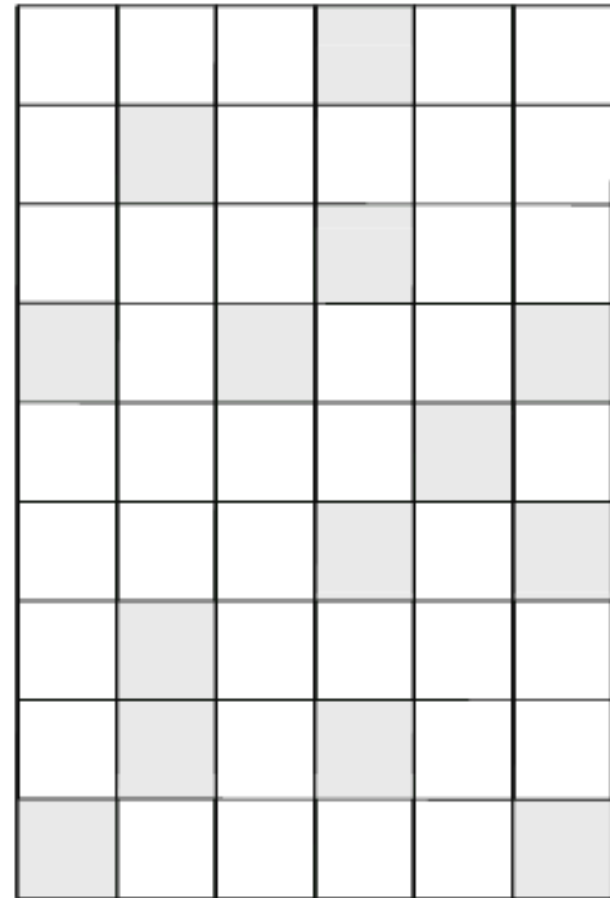


Fig. Credit: Aggarwal 2016
By way of M. Gormley

Setting up Collaborative Filtering task

Real data will have known and unknown entries

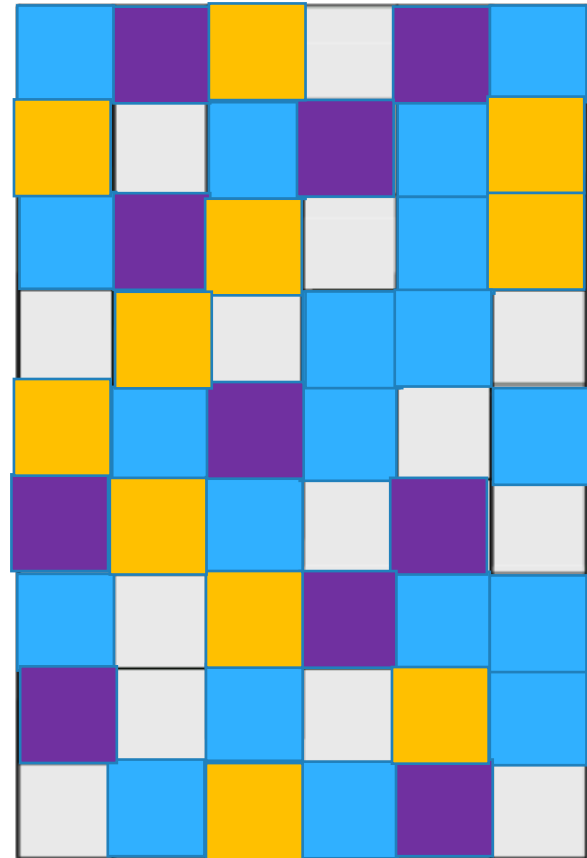


Setting up Collaborative Filtering task

Real data will have known and unknown entries

Divide known user-item pairs at random into:

- Training
- Validation
- Test



Assumption (for Project C): We only care about predictions among known sets of users and items.
Do not worry about any new users or new items. (Obviously, in real world need to handle new users/items)

Latent Factor Model: Training

- Find parameters that minimize squared error

$$\min_{u_i \in \mathbb{R}^K, v_j \in \mathbb{R}^K} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - u_i^T v_j)^2$$

Which pairs do
we use?

Only the blue squares
in the matrix Y

Squared error between

- True rating
- Predicted rating

- How to optimize?
 - **Stochastic gradient descent**
 - Use random minibatch of user-item pairs

Improvement: Include intercept parameters!

- Overall “average rating” μ
- Per-user scalar b_i
- Per-item scalar c_j

$$\hat{y}_{ij} = \mu + b_i + c_j + \sum_{k=1}^K u_{ik} v_{jk}$$

Why include these? Improve accuracy

Some items just more popular

Some users just more positive

Project C

Goal: Latent factor models for MovieLens100k dataset
943 users, 1683 movies

Problems 1, 2, and 3 develop increasingly interesting models:

$$M_1 \quad \min_{\mu \in \mathbb{R}} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - \mu)^2$$

$$M_2 \quad \min_{\mu \in \mathbb{R}, b \in \mathbb{R}^N, c \in \mathbb{R}^M} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - \mu - b_i - c_j)^2$$

$$M_3 \quad \min_{\mu, b, c, \{u_i\}_{i=1}^N, \{v_j\}_{j=1}^M} \sum_{i,j \in \mathcal{I}^{\text{train}}} (y_{ij} - \mu - b_i - c_j - u_i^T v_j)^2$$

We'll use SGD to fit them all! With gradients from autograd.

Matrix Factorization Objectives (day 23)

- Explain Recommendation Task
 - Predict which users will like each item
- Collaborative filtering
 - Unsupervised learning problem
 - Latent Factor model (Reading: Koren et al)
 - Training algorithm: Stochastic gradient descent (SGD)
- How to do gradients: Automatic differentiation
 - Python package: autograd

autograd : Univariate functions

```
## Import numpy  
import numpy as np
```

```
## Import autograd  
import autograd.numpy as ag_np  
import autograd
```

$$f(x) = x^2$$

$$g(x) \triangleq \frac{\partial}{\partial x} f(x)$$

```
def f(x):  
    return ag_np.square(x)
```

```
g = autograd.grad(f)
```

```
# 'g' is just a function.  
# You can call it as usual,  
  
g(0.0)
```

autograd : Multivariate functions

```
## Import numpy
import numpy as np
```

```
## Import autograd
import autograd.numpy as ag_np
import autograd
```

$$f(x) = (x_1 - 1)^2 + (x_2 - 1)^2 - x_1 x_2$$

```
def f(x_D):
```

TODO use ag_np.square and other ag_np calls

$$\begin{aligned} g(x) &\triangleq \nabla_x f(x) \\ &= \left[\frac{\partial}{\partial x_1} f(x) \quad \frac{\partial}{\partial x_2} f(x) \quad \dots \quad \frac{\partial}{\partial x_D} f(x) \right] \end{aligned}$$

```
g = autograd.grad(f)
```

```
# 'g' is just a function.
# You can call it as usual,
```

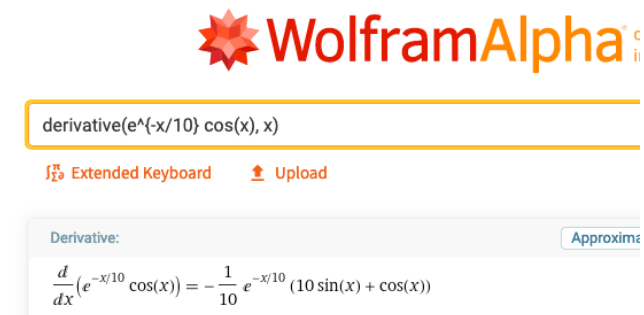
```
g(x_D)
```

Breakout: Lab on autograd

Automatic differentiation

What it is NOT

- Symbolic differentiation
 - Symbolic systems know how to do algebra, simplify, etc
 - Like asking Wolfram Alpha
- Numeric differentiation
 - Perturb input by 0.0001 and estimate the change in function value



Automatic differentiation

What it is: **back propagation**

For more demos and info:

<https://github.com/HIPS/autograd>

What's going on under the hood?

To compute the gradient, Autograd first has to record every transformation that was applied to the input as it was turned into the output of your function. To do this, Autograd wraps functions (using the function `primitive`) so that when they're called, they add themselves to a list of operations performed. Autograd's core has a table mapping these wrapped primitives to their corresponding gradient functions (or, more precisely, their vector-Jacobian product functions). To flag the variables we're taking the gradient with respect to, we wrap them using the `Box` class. You should never have to think about the `Box` class, but you might notice it when printing out debugging info.

After the function is evaluated, Autograd has a graph specifying all operations that were performed on the inputs with respect to which we want to differentiate. This is the computational graph of the function evaluation. To compute the derivative, we simply apply the rules of differentiation to each node in the graph.

Reverse mode differentiation

Given a function made up of several nested function calls, there are several ways to compute its derivative.

For example, given $L(x) = F(G(H(x)))$, the chain rule says that its gradient is $dL/dx = dF/dG * dG/dH * dH/dx$. If we evaluate this product from right-to-left: $(dF/dG * (dG/dH * dH/dx))$, the same order as the computations themselves were performed, this is called forward-mode differentiation. If we evaluate this product from left-to-right: $(dF/dG * dG/dH) * dH/dx$, the reverse order as the computations themselves were performed, this is called reverse-mode differentiation.

Compared to finite differences or forward-mode, reverse-mode differentiation is by far the more practical method for differentiating functions that take in a large vector and output a single number. In the machine learning community, reverse-mode differentiation is known as 'backpropagation', since the gradients propagate backwards through the function. It's particularly nice since you don't need to instantiate the intermediate Jacobian matrices explicitly, and instead only rely on applying a sequence of matrix-free vector-Jacobian product functions (VJPs). Because Autograd supports higher derivatives as well, Hessian-vector products (a form of second-derivative) are also available and efficient to compute.

Recall: Backprop

(from Unit 3)

Assume: Computation graph with known ordering

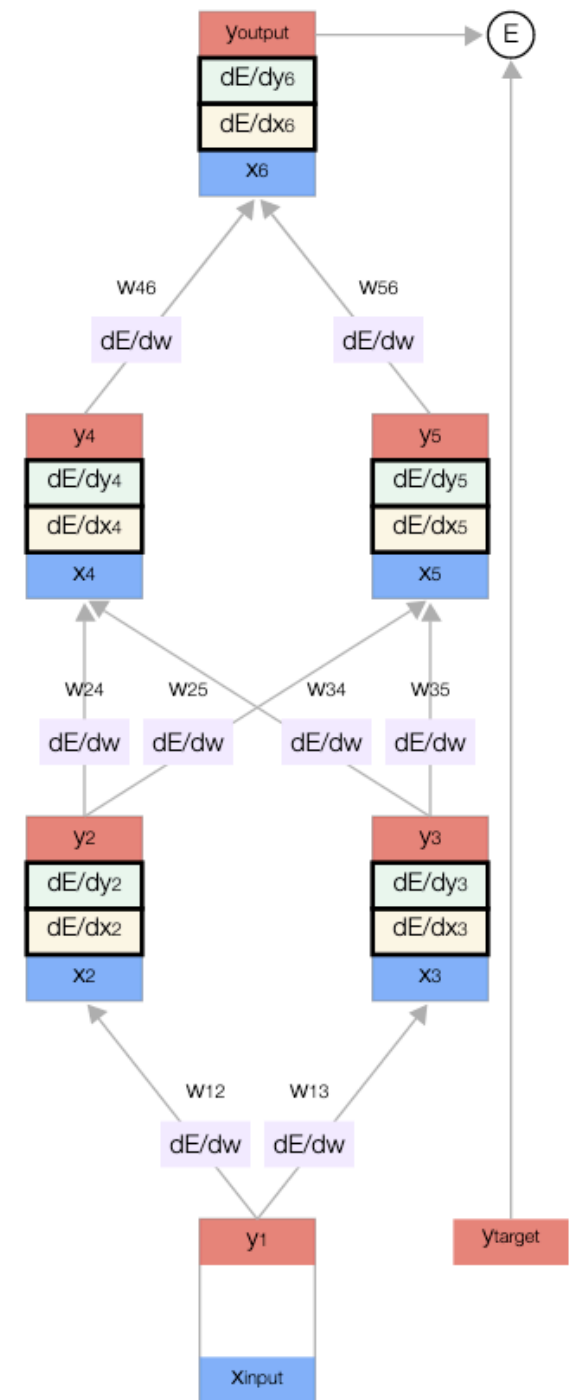
0. Do forward pass
1. Update each non-terminal node **in reverse order**
2. Update each edge's gradient wrt weights

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j}$$

Chain rule update

Each term either:

- simple derivative of known function
- looked up as computation of previous step



Backprop for autograd

Computation graph and ordering *discovered* by monitoring the Python interpreter

0. Do forward pass
1. Update each non-terminal node **in reverse order**
2. Update each input parameter's gradient

Chain rule update involving several terms

Each term either:

- simple derivative of known function
- looked up as result of previous step

As long as we can compute each elementary operation's derivative, we can do this!

