HW 1: don't do Gini

(first graphical taste of $O(n^2)$)

Tubes = {reachable forward from IN}
∩ {reachable backward from OUT}
↓
not in SCC

Rest of vertices are tendrils
↳ in large weak component

How many? (tubes | tendrils) Use weak components

Gradeable up soon (Friday)

HW 2 release next week
↳ link prediction

Gettin' supervised
(for link prediction)

(We'll use some notion of

(for link prediction)

We'll use some notion of a ground truth

↳ existing links and topology

We can construct "features" using this topology

Use those features to train some algorithm

Issue: explicit features might be tough to define

Solution: use implicit or "latent" features

To do so: matrix factorization

Matrix factorization for LP

Consider: we have no metadata

we don't have any idea on what explicit topological features to use

we must rely data-driven feat

features to use

we want a data-independent approach

Matrix factorization in general:
aka matrix decompositions

$$X = UV$$

matrix ↑↗ factors

you might have seen:

* singular value decompositions

$$X = U\Sigma V^*$$

* solving linear systems

$$X = LV$$

* Cholesky, QR, Eigendecomps etc.

For link predictions:

we'll use $A = UVU^T$

A: adjacency matrix; U: feature matrix; V: feature interaction matrix

Our predictions: $a_{ij} = U_i V U_j^T$

a_{ij}: (edge (i, j))

We're trying minimize:

$$\min_{U,V} \sum_{\substack{\text{all} \\ \text{nonzeros} \\ \text{in } A}} |a_{ij} - U_i V U_j^T|$$

really, minimize

$$\min_{U,V} \sum_{\substack{\text{all} \\ nnz \in A}} (a_{ij} - U_i V U_j^T)^2$$

How? Gradient descent

For predictions:

Consider nonzeros in A

look at error relative to $P = UVU^T$

largest values in P relative to zeros in A serve as possible prediction for edges

---

Gradient Descent

$$a_{n+1} = a_n - \alpha \nabla f(a_n)$$

α = learning rate

a_n = features at current step

a_{n+1} = features at next step

a_{n+1} = features at next step

$\nabla f(a_n)$ = how our error changes based on current features

Really: consider how the error changes relative to input

→ move in the direction that decreases error most

$$\min_{U,V} (A - UVU^T)^2$$

↳ $$\min_{U,V} \forall_{i,j} (a_{ij} - U_i V U_j^T)^2$$

↑ our error e_{ij}

$$e_{ij}^2 = (a_{ij} - U_i V U_j^T)^2$$

$$\frac{de_{ij}^2}{dU_i} = (a_{ij} - U_i V U_j^T)(-V U_j^T)$$
$$= -e_{ij} V U_j^T$$

$$\frac{de_{ij}^2}{dU_j} = -e_{ij} U_i V$$

$$\frac{de_{ij}^2}{dV} = -e_{ij} U_i U_j^T$$

$\frac{dV}{dV} = -e_{ij} U_{i-j}$

↳ How our error changes with respect to U, V

Our update equations:

$$U_i = U_i + \alpha e_{ij} V U_j^T$$

$$U_j = U_j + \alpha e_{ij} U_i V$$

$$V = V + \alpha e_{ij} U_i U_j^T$$

$A \in \mathbb{R}^{n \times n}$

$U \in \mathbb{R}^{n \times k}$

$V \in \mathbb{R}^{k \times k}$

k = number of latent features

$k << n$ in general

Note: naively training on *all* values

$O(n^2)$ → of A will bias towards zero

if we train only on nonzeros,

$O(m)$ → we'll bias towards all ones

Solution: weight zeros and ones separately in training

Solution: weight zeros and ones separately in training

$$\min_{U,V} \sum_{\text{nonzeros in } A} (a_{ij} - U_i V U_j^T)^2 + w \sum_{\text{zeros in } A} (a_{ij} - U_i V U_j^T)^2$$

weighting to account for sparsity of A

Other possible solution:

select same subset of zero values in A for training

Issue: we aren't constraining the values of U, V

$\Rightarrow$ they'll blow up

Regularization

$\Rightarrow$ we'll include the magnitude of values in U, V as part of our optimization

$$\min_{U,V} \sum_{nnz} (a_{ij} - U_i V U_j^T)^2 + w \sum_{z} (a_{ij} - U_i V U_j^T) + \beta_1 \|U\|_{fro} + \beta_2 \|V\|_{fro}$$

↑ sum of squared values

values

also called "loss terms"

parameter to control impact on training

magic derivations

...

Our new update equations:

$$U_i = U_i + \alpha(e_{ij} V U_j^T - \beta_1 U_i)$$

$$U_j = U_j + \alpha(e_{ij} U_i V - \beta_1 U_j)$$

$$V = V + \alpha(e_{ij} U_i U_j^T - \beta_2 V)$$

Prediction: take $P = UVU^T$

and max values in P that are zero in A

Collaborative Filtering via recommender systems

→ making a selection from available options (filtering) using given user preferences (collaborative)

... preferences
(collaborative)

General approach
via matrix factorization

- we don't need explicit features
- we can train on weighted graphs

Netflix Challenge

~2007 or so

Netflix: we have a predictor
for user-movie rating

→ predict movie ratings (1-5)
for a given user

Challenge: improve our algo by 10%
→ you'll get $1,000,000

Took a few years
- winning teams used a variety of ML techniques
- Matrix factorization got to 7% by itself

7% by itself

Interesting Notes:

Temporal effects of rating

- some movies were rated higher immediately after viewing instead of being rated later (Patch Adams)
- some had opposite (Memento)

Black sheep

- some people are unpredictable

Naive algorithm:

$$a_{ij} = 0.5(\text{avg}(a_{ij}) + \text{avg}(a_{ij}))$$

j: average for user i

i: average for movie j

Adapt MF for CF

(MF4CF)

Consider a user-movie bipartite graph
an edge a_{ij} is from user i to
movie j - weight of a_{ij} is rating

We can use a bipartite adjacency

where $B_x = \{\text{users}\}$
$B_y = \{\text{movies}\}$

$|B_x| = n$
$|B_y| = m$ $\quad A \in \mathbb{R}^{n \times m}$

Let's setup our MF
- we want to solve $A = UV^T$

$A \in \mathbb{R}^{n \times m}$ $\quad U \in \mathbb{R}^{n \times k}$ $\quad V \in \mathbb{R}^{m \times k}$

U: user feature, V: movie features

- we have some k latent features

- our minimization problem

$$\min_{U,V} \Sigma (a_{ij} - u_i v_j)^2 + B_1 \|U\|_f + B_2 \|V\|_f$$

- our update equations

$$u_i = u_i + \alpha (e_{ij} v_j - B_1 u_i)$$

$$v_j = v_j + \alpha (e_{ij} u_i - B_2 v_j)$$

$$v_j = v_j + \alpha (e_{ij} u_i - \beta_2 v_j)$$

Considers and Challenges

Sparsity of data
- → fewer ratings than non-ratings

Scale of data
- → computation and opt. slow

Generalization:
- → does it work on novel data?
- → do not overfit

"cold-start"
- → how to predict for a new user or new movie?

black sheep
- → tough to predict for certain users
- → just add noise