Smallest font

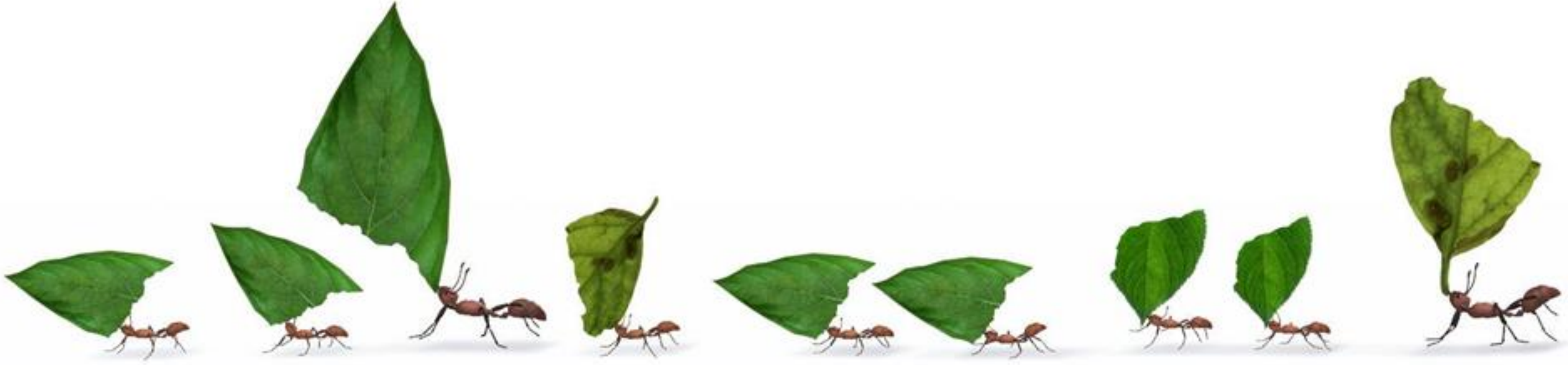
Please turn off and put
away your cell phone

Calibration slide

These slides are meant
to help with note-taking
They are no substitute
for lecture attendance

Smallest font

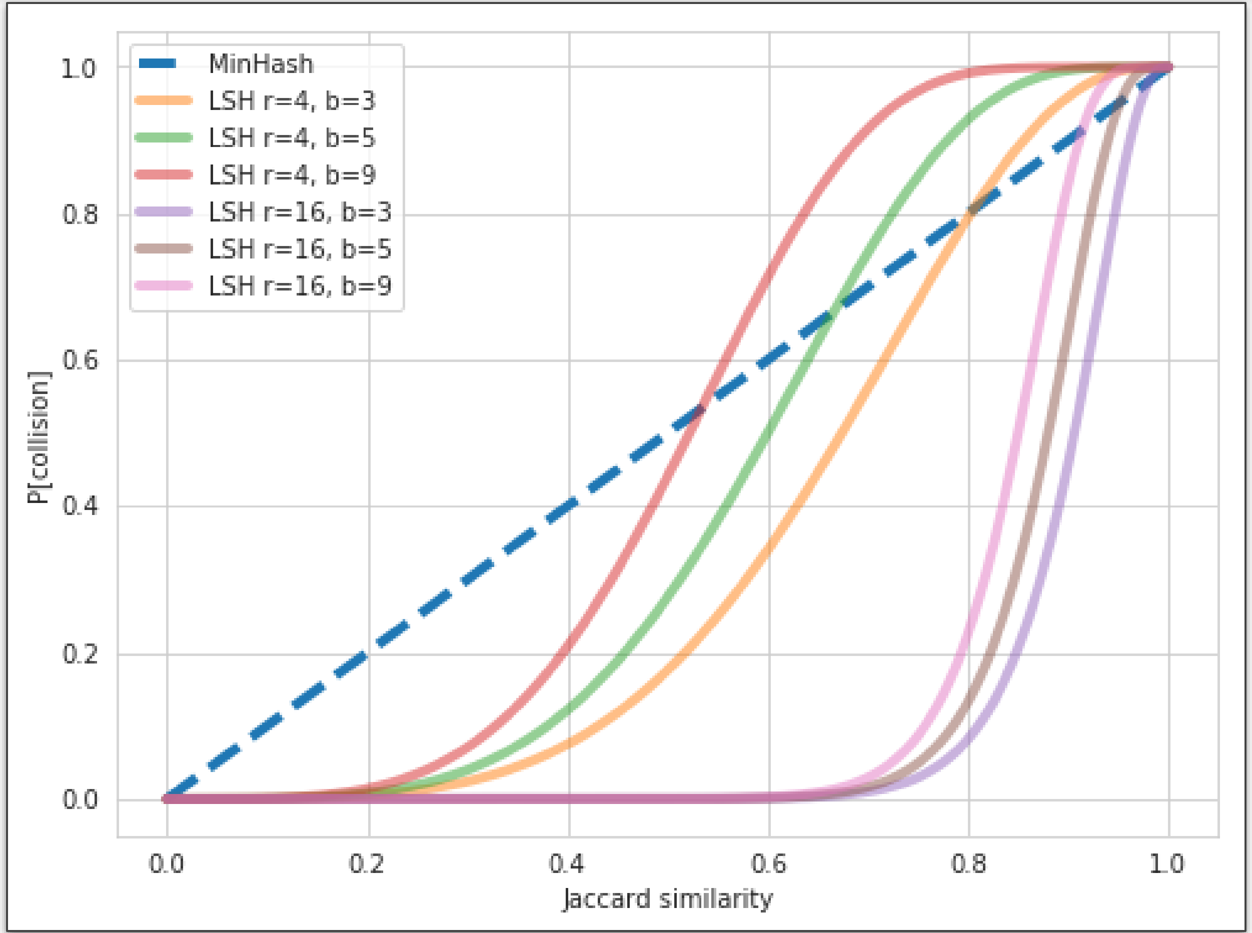
Big Data



Beyond set similarity: Spatial similarity search Graph algorithms

DS-GA 1004: Big Data

CDS: Locality-sensitive hashing (LSH)



Orientation

Search

Relevance from **content** similarity between query and document

Personalization of relevance from explicit or implicit feedback

Relevance from network **structure**

Recommendation

Graph algorithms

Beyond simple sets:
1) Bags (multi-sets)

What if you don't just want to consider whether an item appears (in a set), but also how *often* it does so (in a multi-set/bag): **Ruzicka similarity**

[Chen, Philbin, Zisserman 2008]

- Idea: reduce bags to sets by uniquely identifying each repetition

$\{dog\} \rightarrow \{dog_1\}$

$\{dog, dog\} \rightarrow \{dog_1, dog_2\}$

$\{dog, dog, dog\} \rightarrow \{dog_1, dog_2, dog_3\}$

$$\begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ X = & [1, 3, 5, 7] \\ Y = & [2, 3, 1, 6] \end{array}$$

- Jaccard on expanded sets = **Ruzicka similarity** on original bags

$$R(A, B) = \frac{\sum_i \min(A[i], B[i])}{\sum_j \max(A[j], B[j])}$$

Σ(min(x_i, y_i)) = 1 + 3 + 1 + 6 = 11

Σ(max(x_i, y_i)) = 2 + 3 + 5 + 7 = 17

R(X, Y) = 11 / 17 ≈ 0.65

min(x_1, y_1) = min(1, 2) = 1

max(x_1, y_1) = max(1, 2) = 2

min(x_2, y_2) = min(3, 3) = 3

max(x_2, y_2) = max(3, 3) = 3

min(x_3, y_3) = min(5, 1) = 1

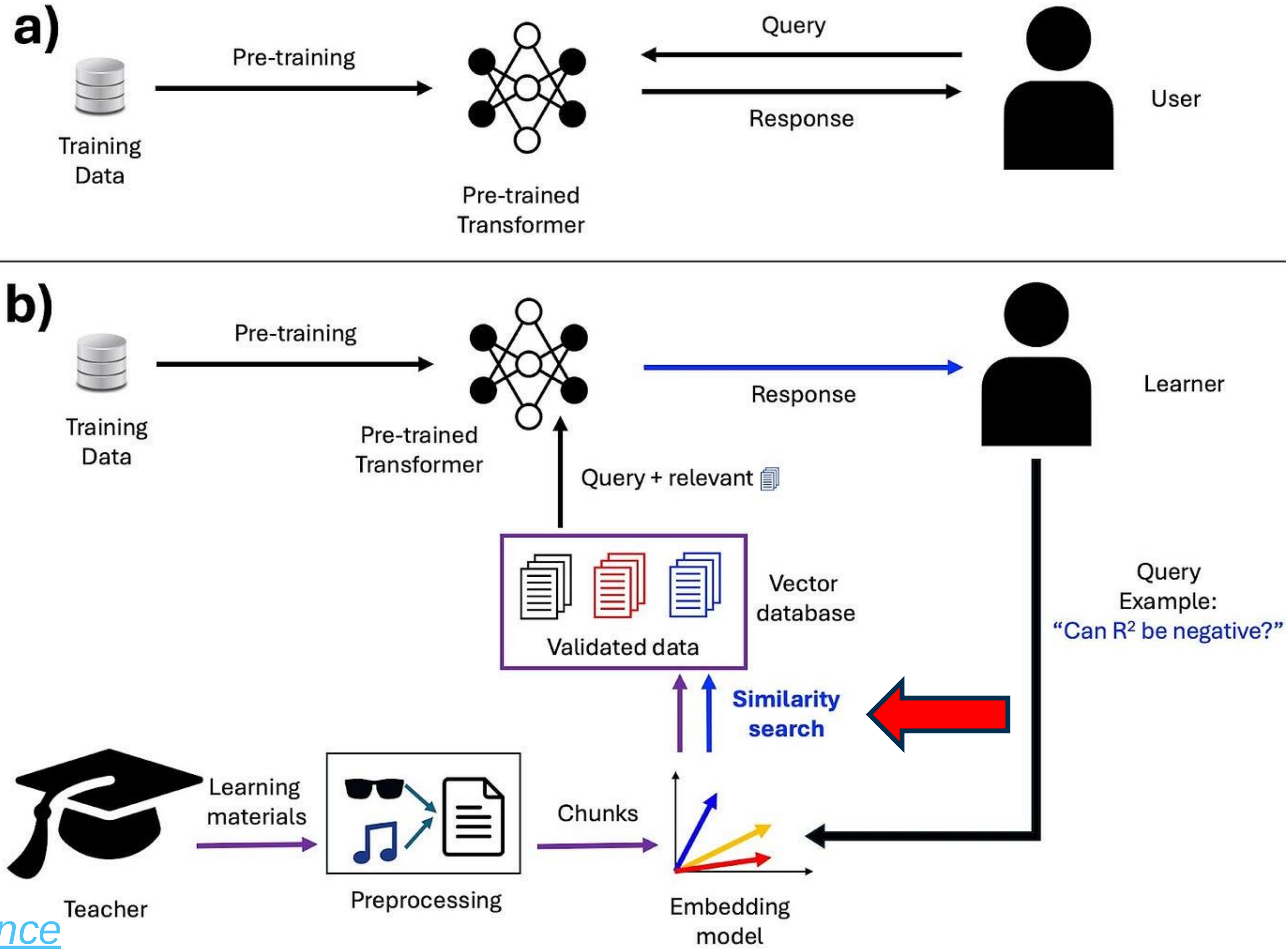
max(x_3, y_3) = max(5, 1) = 5

min(x_4, y_4) = min(7, 6) = 6

max(x_4, y_4) = max(7, 6) = 7

Beyond simple sets:
2) Spatial similarity search (as in a vector database)

A typical use case



[Wallisch & Sheikh 2024](#)
[Nature Machine Intelligence](#)

Cosine similarity is often a suitable metric to compare documents (and other high-dimensional objects)

- For instance, one could express documents in terms of the magnitude of their embedding dimensions.
- Cosine similarity conceptualizes the similarity of two vectors **a** and **b** in terms of the angle θ between them, regardless of their length.
- This makes intuitive sense:

$$cosine\ similarity = \cos(\theta) = \frac{\boldsymbol{a}^T\boldsymbol{b}}{||\boldsymbol{a}||\,||\boldsymbol{b}||} = \frac{\boldsymbol{a}^T\boldsymbol{b}}{\sqrt{\boldsymbol{a}^T\boldsymbol{a}}\sqrt{\boldsymbol{b}^T\boldsymbol{b}}}$$

cos(0) = 1 cos(90) = 0 cos(180) = -1

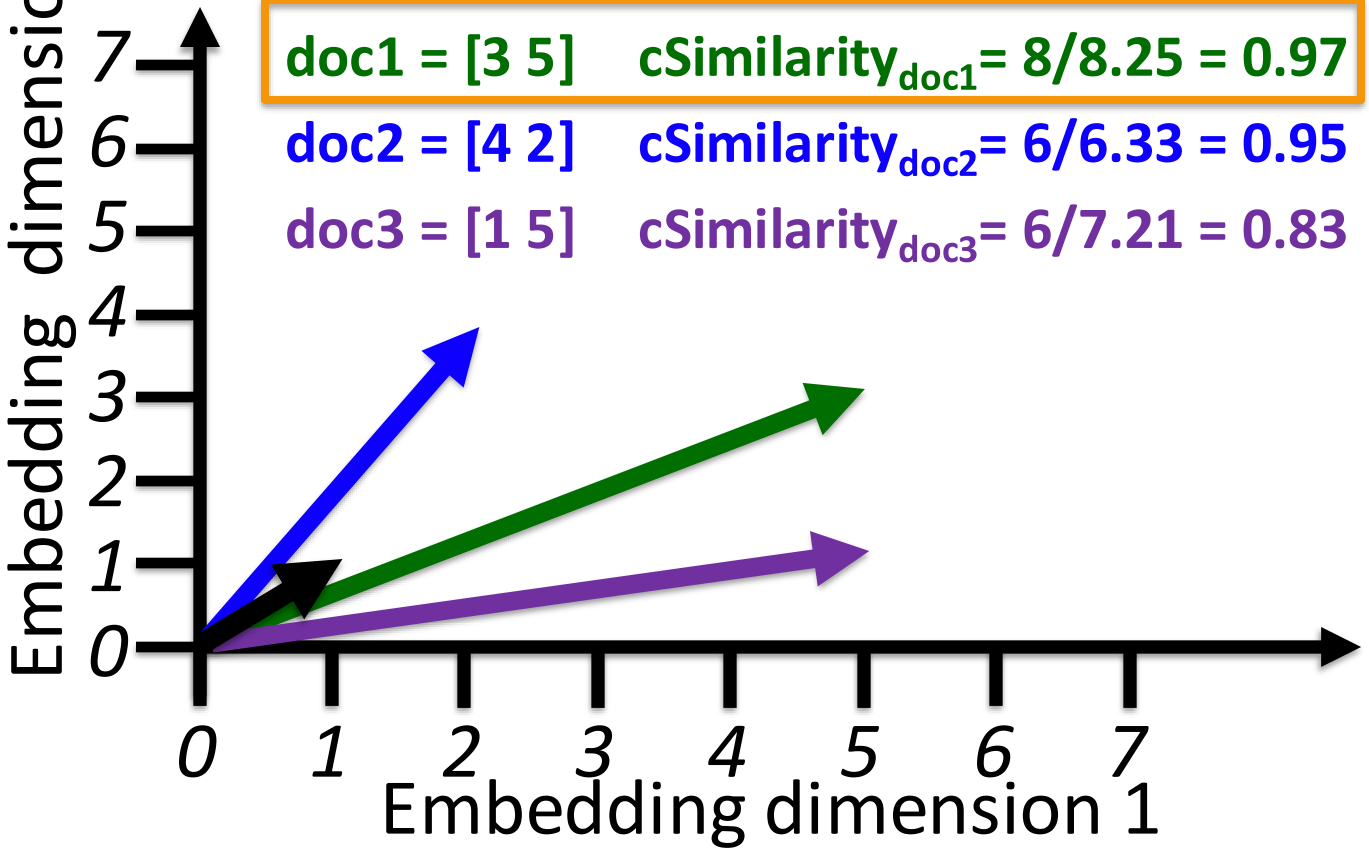
Determining relevance with cosine similarity

doc_0 = [1 1]

doc1 = [3 5] $\text{cSimilarity}_{doc1}$ = 8/8.25 = 0.97

doc2 = [4 2] $\text{cSimilarity}_{doc2}$ = 6/6.33 = 0.95

doc3 = [1 5] $\text{cSimilarity}_{doc3}$ = 6/7.21 = 0.83



Locality Sensitive Hashing (LSH) for cosine similarity

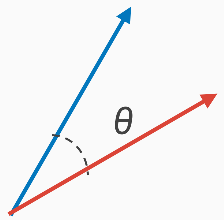
- Computing cosine similarity is straightforward between any two vectors, but can be computationally expensive if the dimensionality **d** of the vectors is large.
- **N** will be very large (many billions of vectors or more) in a modern vector database, so time complexity is O (Nd).
- The idea is then to use LSH to whittle down all vectors to a candidate set.
- Full cosine similarity is only computed for the vectors in the candidate set.
- LSH is valid but a bit abstract for sets. It is more geometrically intuitive here.
- Charikar (2002) introduced LSH for cosine similarity by grouping similar items into the same "bucket", by dividing the vector space in half.
- This data partitioning works by using random hyperplanes to slice the vector space.
- The hyperplane divides the space into two halves. The hash depends on which side it falls on. If they fall on the same side, the vectors "collide".
- The probability of a collision is related to the (cosine) similarity of the vectors.

LSH for cosine similarity
[Charikar 2002]

- What if we want to compare vectors $u, v \in \mathbf{R}^d$ by cosine similarity?

$$\mathrm{sim}(u, v) = \cos(\theta)$$

Straightforward, but can we approximate
the cosine similarity between
these vectors without computing the cosine
similarity between them?



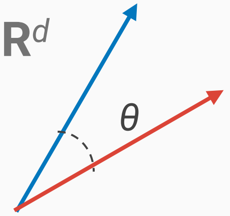
Idea: Pick a random direction (w) in the space. Technically, we do this by choosing w uniformly at random from a sphere in d-dimensional space.

LSH for cosine similarity
[Charikar 2002]

- What if we want to compare vectors u, $v \in \mathbf{R}^d$ by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector w randomly but uniformly from the unit sphere in $\mathbf{R}^d$
  - $h_w(x)$ $= 1$ if $w^T x >= 0$
    $= -1$ if $w^T x < 0$

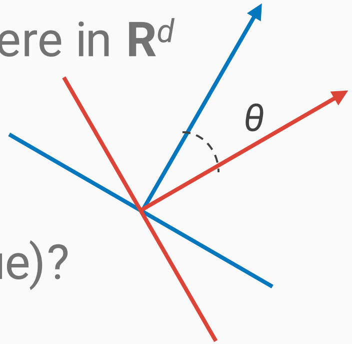


LSH for cosine similarity
[Charikar 2002]

- What if we want to compare vectors $u, v \in \mathbf{R}^d$ by cosine similarity?

$$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector w randomly but uniformly from the unit sphere in $\mathbf{R}^d$
    - $h_w(x) \quad = 1 \text{ if } w^T x >= 0$
      $\quad = -1 \text{ if } w^T x < 0$



- What's the probability of a collision (both same hash value)?

    - $\mathbf{P}[h_w(u) = h_w(v)] \quad = 1 - \mathbf{P}[h_w(u) \neq h_w(v)]$

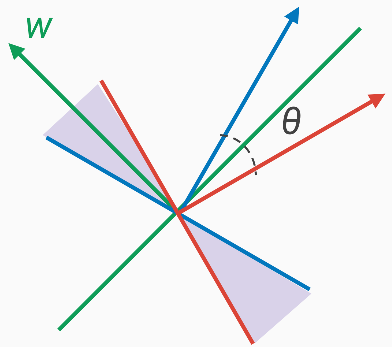
Two vectors (u, v) separated by an angle θ define a geometric region between them. The randomly chosen hyperplane (w) partitions space. The probability that the hyperplane falls between the two vectors (shaded region) directly relates to the angle θ:

LSH for cosine similarity
[Charikar 2002]

- What if we want to compare vectors u, $v \in \mathbf{R}^d$ by cosine similarity?

$$\mathrm{sim}(u, v) = \cos(\theta)$$

- Pick a vector w uniformly from the sphere in $\mathbf{R}^d$
  - $h_w(x) \quad = 1$ if $w^T x >= 0$
    $\quad = -1$ if $w^T x < 0$



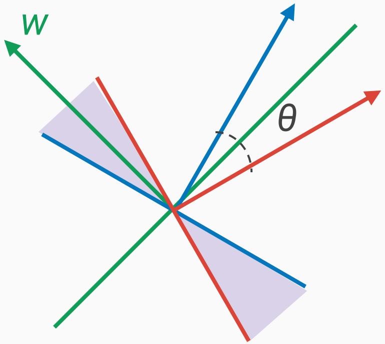
- What's the probability of collision?
  - $\mathbf{P}[h_w(u) = h_w(v)] \quad = 1 - \mathbf{P}[h_w(u) \neq h_w(v)]$
    $= 1 - \mathbf{P}[w$ in **shaded region**$]$

LSH for cosine similarity
[Charikar 2002]

- What if we want to compare vectors u, $v \in \mathbf{R}^d$ by cosine similarity?

  $$\text{sim}(u, v) = \cos(\theta)$$

- Pick a vector w uniformly from the sphere in $\mathbf{R}^d$
  - $h_w(x) = 1$ if $w^T x >= 0$
    $= -1$ if $w^T x < 0$



- What's the probability of collision?
  - $\mathbf{P}[h_w(u) = h_w(v)] = 1 - \mathbf{P}[h_w(u) \neq h_w(v)]$
    $= 1 - \mathbf{P}[w$ in **shaded region**$]$
    $= 1 - 2 \cdot |\theta| / 2\pi$
    $= 1 - |\theta|/\pi$

Not exactly cos(θ), but monotonically decreasing with |θ| ⇒ same rank-ordering

In Charikar's method, the signal amplification for LSH does not come from rows and blocks, but from **multiple projections:**

- P[**No collision** | single projection] = θ/π

  A single hyperplane hash is very coarse (the probability of two vectors colliding by getting the same hash value is high), which leads to many false positives (akin to having many stop words in sets)

Projections m: How many (m) random hyperplanes are used to partition the vector space

- P[**All projections do not collide** | *m* projections] = $(\theta/\pi)^m$
- P[At least one **Collision** | *m* projections] = $1 - (\theta/\pi)^m$

So much for similarity search
Both with sets and beyond
Now: Relevance from graphs

PageRank

Early web search (pre Google)

- Early search engines relied on matching text in **query** to text in **web page**
- Pages were crawled (by "spiders") at regular intervals and added to an index
- We've already discussed some tools for indexing and searching
- ***What could/did go wrong by relying on this approach?***

Yahoo.com in April, 1997 [archive.org]

Spam attacks

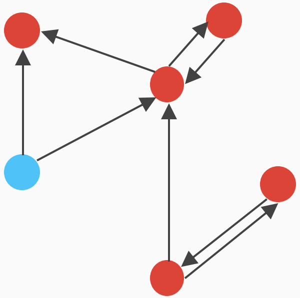
- Imagine that you want to get lots of traffic to your website
- You know that it is indexed regularly by search engines
- **Idea**: Discreetly fill your page with *all* of the **most popular search terms**
- End result: **$$$** selling items irrelevant to the user query

PageRank: use the structure of the network itself!

- Key insight: the **structure** of the network contains information!
- Publishers are more likely to link to pages that they trust
- It’s **easy** to make a spammy page
- It’s **hard** to get other people to link to it

The random surfer model

- Imagine the web as a directed graph
  - Nodes = pages
  - Directed Edges = links (have a direction)



- Model a user's activity as a random walk
  - **P**[Going to page *v* | Currently at page *u*] = [*u* → *v*] / out-degree(*u*)

[0 or 1]   total number of edges

- Users are more likely to land at pages with **high in-degree**
  - What is the steady-state distribution **P[v]**?

Markov chains

- Let **M**[v, u] = P[v | u]
  - Columns (u) = current states (N pages)   Rows (v) = next states (N pages)

- **M** is a **stochastic matrix**: non-negative, each column sums to 1

Markov chains

- Let **M**[v, u] = P[v | u]
  - Columns (u) = current states (N pages) Rows (v) = next states (N pages)
- **M** is a **stochastic matrix**: non-negative, each column sums to 1
- Let **p** be a non-negative vector in $\mathbf{R}^N$ that sums to 1
  - (**M**p)[v] marginalizes over *u* to compute probability of state *v*

Markov chains

- Let $\mathbf{M}[v, u] = P[v \mid u]$
  - Columns (u) = current states (N pages) Rows (v) = next states (N pages)

- **M** is a **stochastic matrix**: non-negative, each column sums to 1

- Let **p** be a non-negative vector in $\mathbf{R}^N$ that sums to 1
  - $(\mathbf{M}p)[v]$ marginalizes over *u* to compute probability of state *v*

- So: One step maps

$$p[v] \rightarrow (\mathbf{M}p)[v] = \sum_u P[v \mid u] * p[u] = p'[v]$$

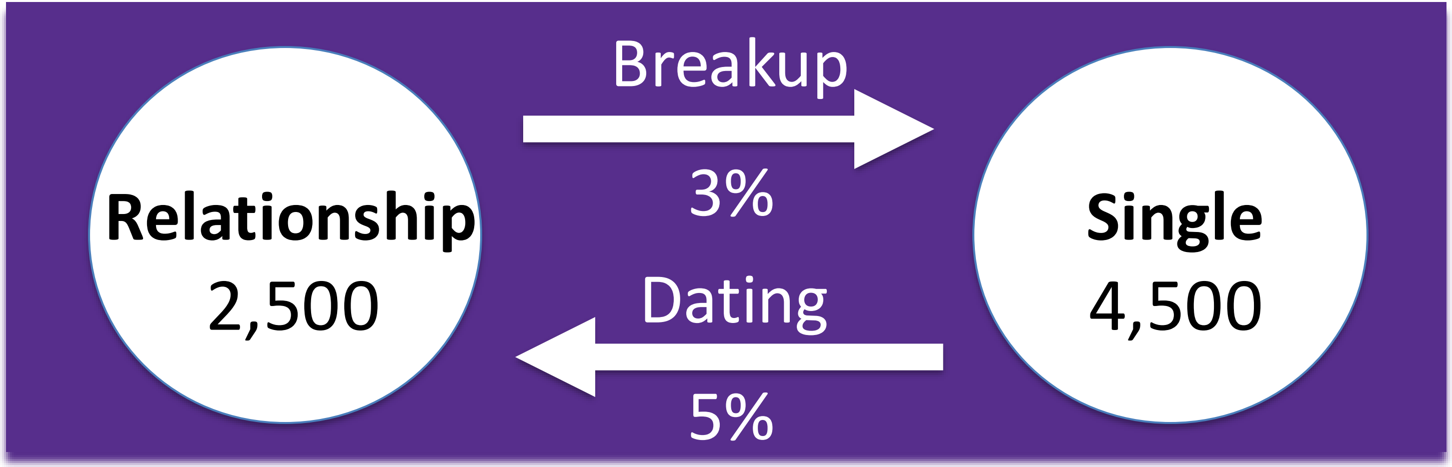
Steady-state distributions

- A steady-state distribution satisfies the identity $\mathbf{p} = \mathbf{M}\mathbf{p}$
- Such a $\mathbf{p}$ exists if there is a *path* connecting every pair $u \rightarrow v$ in a finite number of steps
    - If that is the case, we say that **M** is ***irreducible*** or ***strongly connected*** *[more on this soon]*

Steady-state distributions

- A steady-state distribution satisfies the identity **p** = **Mp**
- Such a **p** exists if there is a *path* connecting every pair u → v in a finite number of steps
  - If that is the case, we say that **M** is ***irreducible*** or ***strongly connected*** *[more on this soon]*
- **p** = **eigenvector** of **M** with **eigenvalue** 1
  - The largest possible for a stochastic matrix!
- PageRank(*u*) = p[*u*] = probability of random surfer being at node *u*
- *This is all a bit abstract, let's give a vivid, real life example of such modeling.*

Spoiler: The Eigenvector of a Markov matrix represent the long term equilibrium of a dynamic system:



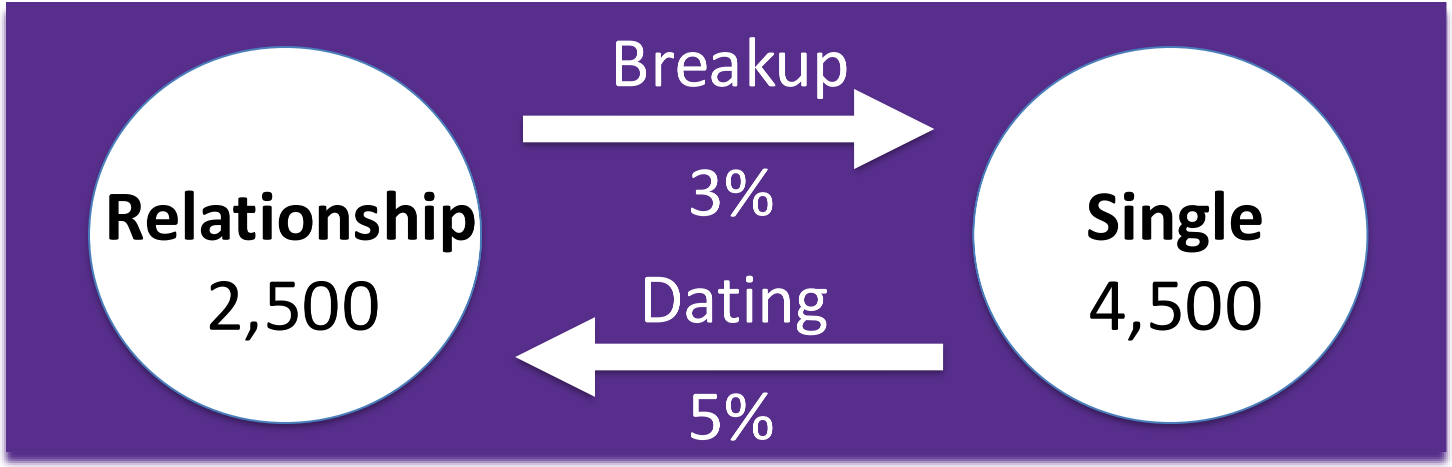
After the 1st month:

In relationships: 0.97(2,500) + 0.05(4,500) = 2,650

Single: 0.03(2,500) + 0.95(4,500) = 4,350

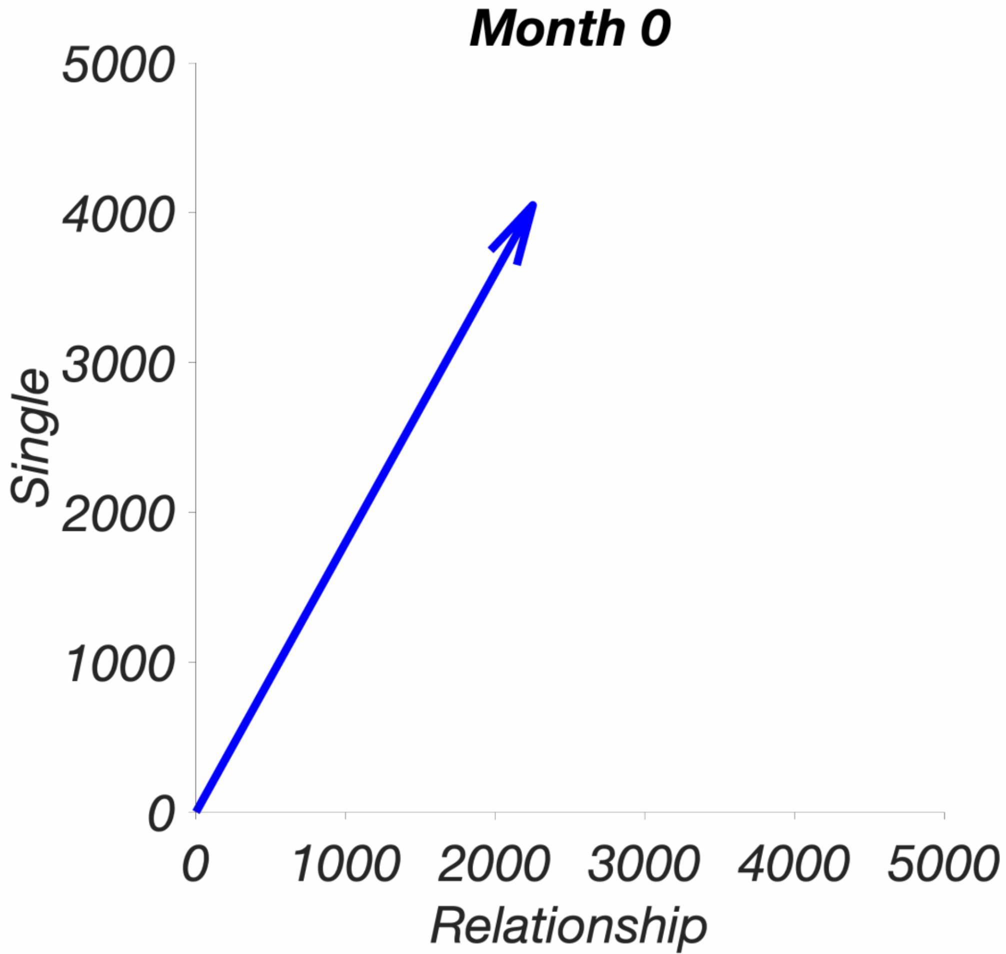
How about the long term?

Modeling this with a Markov Matrix

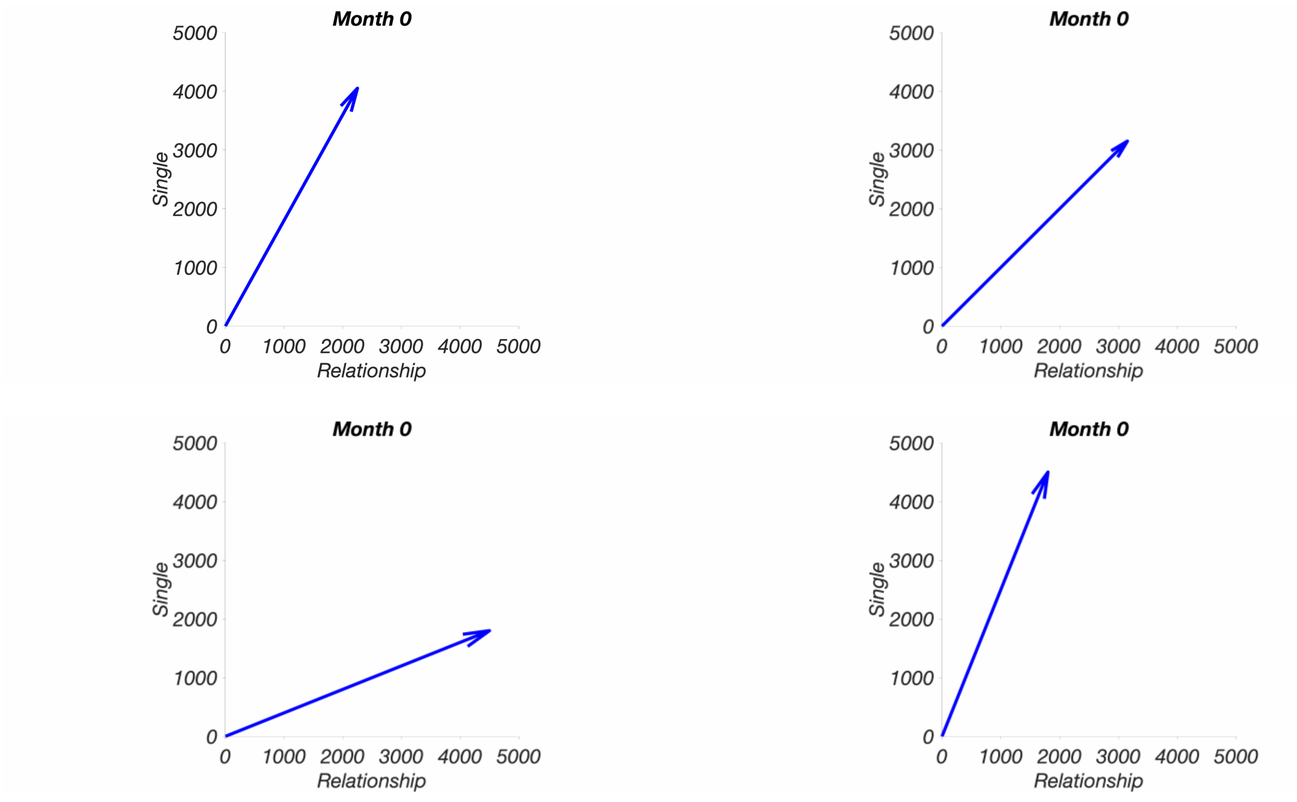


$$\begin{pmatrix} 0.97 & 0.05 \\ 0.03 & 0.95 \end{pmatrix} \begin{pmatrix} R_0 \\ S_0 \end{pmatrix} = \begin{pmatrix} R_{+1} \\ S_{+1} \end{pmatrix}$$

(Where) does this system stabilize?



Regardless of starting position



It stabilizes on the dominant Eigenvector. Eigenvalue: 1

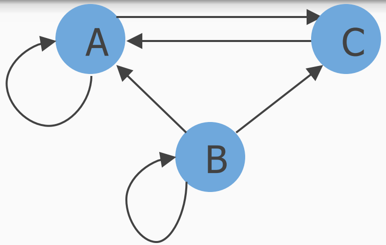
Markov chain Eigenvectors

- This works fine (if you have small data). But:
- Standard eigenvector solvers do not scale to the web (or big data in general)
  - $O(N^3)$ cost to solve in general, can be smaller if sparse, but N can still be huge!

- Instead, use **power iteration** (aka the "power method")
  - Initialize $p_0[u] \leftarrow 1/N$ (uniform distribution)
  - For $i = 1, 2, \ldots, T_{max}$
    - $p_i \leftarrow \mathbf{M}p_{i-1}$ ($p_i = \mathbf{M}^i\, p_0$) [This can be parallelized over rows of M]

- This will eventually converge to the stationary distribution
  - (If such a distribution exists...)

Small network example of power iteration: Initialization



p$_1$
?
?
?

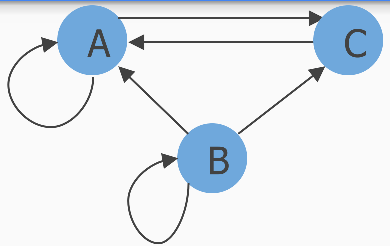
M

	A	B	C
A	½	⅓	1
B	0	⅓	0
C	½	⅓	0

p$_0$
⅓
⅓
⅓

- Initialize $p_0[u] \leftarrow 1/N$
- For $i = 1, 2, \ldots, T_{max}$
  - $p_i \leftarrow M p_{i-1}$

Small network example of power iteration: p_1



p_1
11/18
2/18
5/18

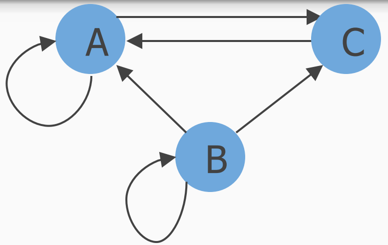
M

	A	B	C
A	½	⅓	1
B	0	⅓	0
C	½	⅓	0

p_0
⅓
⅓
⅓

- Initialize $p_0[u] \leftarrow 1/N$
- For $i = 1, 2, \ldots, T_{max}$
    - $p_i \leftarrow \mathbf{M} p_{i-1}$

Small network example of power iteration: p_2



M

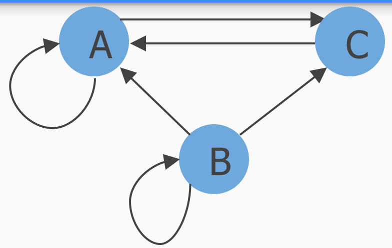
p_2
67/108
4/108
37/108

	A	B	C
A	½	⅓	1
B	0	⅓	0
C	½	⅓	0

p_1
11/18
2/18
5/18

- Initialize $p_0[u] \leftarrow 1/N$
- For $i = 1, 2, \ldots, T_{max}$
  - $p_i \leftarrow \mathbf{M} p_{i-1}$

Small network example of power iteration: p_3



M

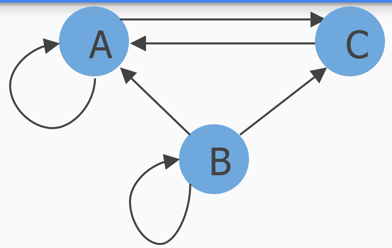
p_3
431/648
8/648
209/648

	A	B	C
A	½	⅓	1
B	0	⅓	0
C	½	⅓	0

p_2
67/108
4/108
37/108

- Initialize $p_0[u] \leftarrow 1/N$
- For $i = 1, 2, \ldots, T_{max}$
  - $p_i \leftarrow \mathbf{M}p_{i-1}$

Small network example of power iteration: Steady state



The pagerank vector

p_n
2/3
0
1/3

M

	A	B	C
A	½	⅓	1
B	0	⅓	0
C	½	⅓	0

p_n
2/3
0
1/3

- Initialize $p_0[u] \leftarrow 1/N$
- For $i = 1, 2, \ldots, T_{max}$
  - $p_i \leftarrow Mp_{i-1}$

If you want to check your work in Python

- **M**[v, u] = P[v | u] (probability of going to v from u)
  - Each column is a probability distribution
  - ⇒ **M.sum(axis=0)** should be all ones

- **evals, evecs = np.linalg.eig(M)** is almost, but not quite, what we want

If you want to check your work in Python

- M[v, u] = P[v | u] (probability of going to v from u)
  - Each column is a probability distribution
  - ⇒ **M.sum(axis=0)** should be all ones

- **evals, evecs = np.linalg.eig(M)** is almost, but not quite, what we want

- **Remember**:

  evecs have unit Euclidean (L_2) norm: $\boldsymbol{v} = \boldsymbol{v} \,/\, \|\boldsymbol{v}\|_2$

  Probability distributions have unit L_1 norm: $\boldsymbol{p} = \boldsymbol{v} \,/\, \|\boldsymbol{v}\|_1 \neq \boldsymbol{v}$

If you want to check your work in Python

- M[v, u] = P[v | u] (probability of going to v from u)
  - Each column is a probability distribution
  - ⇒ **M.sum(axis=0)** should be all ones

- **evals, evecs = np.linalg.eig(M)** is almost, but not quite, what we want

- **Remember**:

  evecs have unit Euclidean (L_2) norm: $\boldsymbol{v} = \boldsymbol{v} \,/\, \|\boldsymbol{v}\|_2$

  Probability distributions have unit L_1 norm: $\boldsymbol{p} = \boldsymbol{v} \,/\, \|\boldsymbol{v}\|_1 \neq \boldsymbol{v}$

- So you have to re-normalize each column:
  - **evecs /= np.abs(evecs).sum(axis=0, keepdims=True)**

You may also have to flip the sign of *v*.

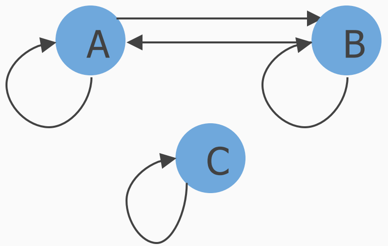
If *v* is an eigenvector, so is -*v*!

Requirement:
The graph must be connected!

- If the graph is not connected, there is not a **unique leading eigenvector (stationary distribution)**

- The **requirements** for PageRank **are not met**.

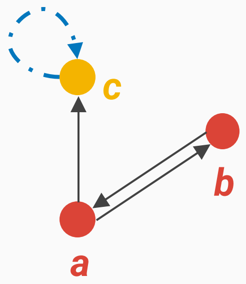
  Remember: we need every vertex to be reachable by every other vertex!



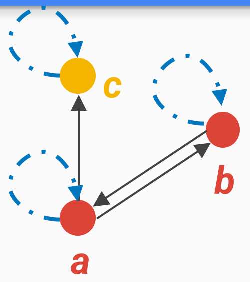
	A	B	C
A	½	½	0
B	½	½	0
C	0	0	1

What about “sinks”?

- A sink is a **vertex with no outgoing edges**, its column in M is all zeros.
  - This will not yield a valid probability distribution!
- Common fix: add a **self-loop** to any nodes having out-degree 0
  - **Nodes with outgoing edges** do not need to be modified
- Result is a well-formed transition matrix



“Spider traps”



- Node ***c*** is a **spider trap** in this graph – no outgoing links
- A random surfer landing at ***c*** can never leave!

	a	b	c
a	⅓	½	0
b	⅓	½	0
c	⅓	0	1

p_0
⅓
⅓
⅓

Power iteration

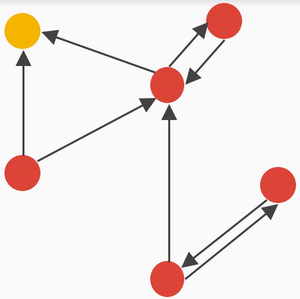
p
0
0
1

Traps are easy to create!

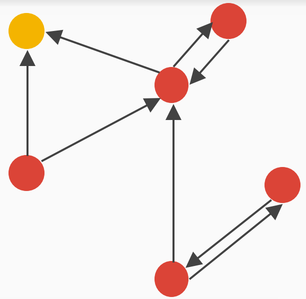
All it takes is one link from a well-connected vertex to cause serious damage!

Solution: Teleportation / random restart

- The web is decidedly **not strongly** connected
- Not all pages have outward links (column sum = 0)



Teleportation / random restart



- The web is decidedly **not strongly** connected
- Not all pages have outward links (column sum = 0)
- Solution: **teleportation**!
- With probability a, follow the links
  With probability 1-a, jump uniformly at random

$$M[v, u] \rightarrow a * M[v, u] + (1 - a) * 1/N$$

Using PageRank to improve search

- PageRank uses network topology to score relevance of each vertex (page)
- $p[v] > p[u] \Rightarrow v$ is “better connected” than u
  - It does not use content! But still a reasonable measure of importance

- Basic search implementation:
  - Use **text search** (LSH, etc.) to find candidate items
  - Use **pagerank** (and other cues) to order results

- Can we do better?

Improving further: Personalizing PageRank

- The uniform teleportation model isn't realistic
  - Jumping to ***any*** page? Really?

- If $e = [1, 1, 1, \dots 1]$, then PageRank + teleportation computes

$$\mathbf{p} = (a * \mathbf{M} + (1-a) * 1/N * ee^T)\mathbf{p}$$

- $1/N * e$ is the uniform distribution (taking the role of random teleportation)
  - what if we replaced it by something else? [that could take preferences into account]

Personalized PageRank

$\mathbf{p} = (a * \mathbf{M} + (1-a) * 1/N * ee^T)\mathbf{p}$

$= a * \mathbf{Mp} + (1-a) * 1/N * ee^T\mathbf{p}$ ($e^T\mathbf{p} = 1$ because $\mathbf{p}$ is a probability distribution)

$= a * \mathbf{Mp} + (1-a) * 1/N * e$ (replace $1/N * e$ by a preference distribution $\mathbf{q}$)

$= a * \mathbf{Mp} + (1-a) * \mathbf{q}$

- $\mathbf{q}$ is the *personalization vector* (distribution)
  - E.g., uniform over pages about data science
  - Idiosyncratic to a given user

We can still do power iteration here!

Not more computationally expensive

But more flexible to "bias" PageRank

Distributed PageRank with Spark

```python
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends()  # Get example graph

# Run PageRank until convergence to tolerance "tol".
results = g.pageRank(resetProbability=0.15, tol=0.01)
# Display resulting pageranks and final edge weights
# Note that the displayed pagerank may be truncated, e.g., missing the E notation.
# In Spark 1.5+, you can use show(truncate=False) to avoid truncation.
results.vertices.select("id", "pagerank").show()
```

$$\mathbf{p} \leftarrow a * \mathbf{Mp} + (1\text{-}a) * \mathbf{q}$$

- Core computation is **matrix multiplication**
  - We know how to do that with map-reduce
  - Complexity depends on network sparsity
- Also possible in Spark using the GraphX package
- High-level interface: GraphFrames
  - *"GraphX is to RDDs as GraphFrames are to DataFrames."*

Next week

- Socio-cultural impact of Big Data
  - Filter bubbles
  - Polarization
  - (Differential) privacy
- After that: Present and future of Big Data

Q&R