Department of Computer Science
University of California, Los Angeles

# Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

Lecture 17: June 2, 2025

# Outline

1. Accessibility
2. Information Retrieval

1 Accessibility

2 Information Retrieval

# Accessibility


Large companies take accessibility very seriously. We all should!

# Accessibility (contd.)

# Accessibility

Accessibility is the practice of making a web app usable by as many people as possible. This includes those with:

1. No obvious barriers.
2. Individual physical needs
3. Individual mental needs including mental illness and neurodiversity
4. Mobile devices
5. Slow network connections

# Accessibility (contd.)

In some countries, laws exist requiring web sites and applications to be accessible. These countries/administrative areas include:

1. United States (ADA)
2. Canada (Accessible Canada Act)
3. European Union (EU Web Accessibility Directive)
4. Australia (Disability Discrimination Act)
5. United Kingdom (Equality Act)

Aside from being the **morally right thing to do**, there is also a capitalist argument to be made. Developing around various needs allows us to increase our market which means more $\$\$\$$.

# Accessibility (contd.)

We will focus a few broad areas:

1. Visual impairment
2. Hearing impairment
3. Mobile impairment
4. Cognitive impairment
5. Differences in education
6. Slow network connections

# Visual Impairment

Visual impairment includes:

- Total blindness
- Low-level vision
- Color blindness

# Visual Impairment (contd.)

Most visually impaired users, excluding those that are totally blind, simply use the zoom feature within their browser.

Web content isn't rendered in Braille directly, but screen readers often support refreshable Braille displays that translate semantic HTML into tactile output.

# Visual Impairment (contd.)

Some popular screen readers include JAWS, Dolphin Screen Reader, NVDA, Narrator (Windows), Orca (Linux), Talkback (Android), ChromeVox (ChromeOS) and Voiceover (iOS).

Dallas College and Indiana University have good information about developing for the Web for the vision impaired.

# Visual Impairment (contd.)

A good YouTube video with a demo of how a screen reader works:

UC San Francisco IT Web Services Department

# Visual Impairment (contd.)

Areas requiring special attention developing for screen readers:

- **Forms:** all form elements and their types should be announced.
  - Form labels should be clear and be announced.
  - The states, properties, values and changes to them must be announced.
- **Tables:** they must have names and header cells and should be simple with no empty rows, columns or cells.
- **Accordions/Expandos:** each panel should be able to be opened and its content accessed by the screen reader.
- **Carousels and Slide Shows:** auto-advancing slideshows can be problematic. Provide a Previous/Next button etc.
- **Images:** always include alt text.
- **Alerts, Errors and Pop-Ups**: they must fire in a way that the screen reader can alert the user.

# Visual Impairment (contd.)

Other tips:

- For vision impaired users that are not totally blind, use high contrast colors, or provide some kind of style that provides this high contrast.
- For the color blind, use proper color palettes whose contrast they can understand. For example, use blue and brown instead of green and red.
- Try to avoid conveying tone only with style and color.

# Visual Impairment (contd.)

Some of these technologies should also be adopted to address users that wear glasses or contact lenses.

*My Girl*: Macaulay Culkin as Thomas J., Anna Chlumsky as Vada Sultenfuss at Thomas J's funeral. He died after being stung by a hive of bees while trying to fetch Vada's mood ring.

*"Where are his glasses? He can't see without his glasses!" -Vada Sultenfuss*

(This scene was pretty traumatizing for millennials...)

# Hearing Impairment

Deaf and hard of hearing can range from mild to fully deaf.

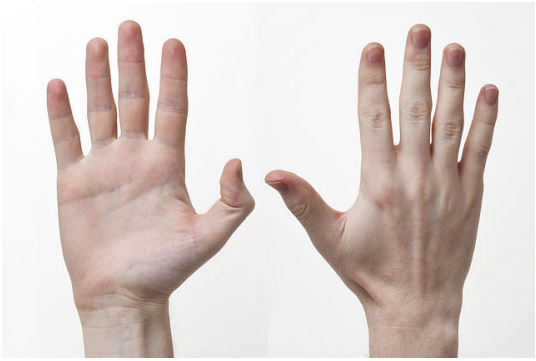
Assistive technologies for these groups are not as widespread because most of the web is visual.

# Hearing Impairment (contd.)

**Closed captioning** and **transcripts** should be provided for video content. The language should be simplified so the user can keep up with the transcript.

Note that it is *best* to manually create the transcript or CC. AI is great, but it can be ineffective especially with accents or certain speech patterns.

# Mobility Impaired



Mobility impaired individuals have physical issues that prevents them from using standard input devices.

- Loss of limbs (particularly fingers, hands or arms)
- Neurological impairments that control the limbs, like Parkinson's
- **In both cases, users may not be able to use a mouse.**

For people that can't use a mouse, develop your app to use **keyboard input** more: shortcuts, tab button etc.

# Mobility Impaired (contd.)

In some cases, users use a head pointer:

# Mobility Impaired (contd.)

Some users are sensitive to animation (e.g. Parkinson's, epilepsy).

```
@media (prefers-reduced-motion: reduce) {
  * { animation: none !important; transition: none !importa
}
```

- Avoid parallax effects, looping animations
- Offer reduced motion settings if your app is animation-heavy

# Cognitive or Neurological Diversity

The term *cognitive impairment* is used broadly in the accessibility literature, I would prefer to call it "different ways of thinking and processing information."

- Elderly age
- Depression, schizophrenia, PTSD, epilepsy
- Neurodiversity: ADHD, ASD, Dyslexia

But how on earth would we design for depression and schizophrenia? It's not so much the conditions themselves, but the some of the limitations that they cause during a flare up, and things that our content may *trigger*.

# Cognitive or Neurological Diversity (contd.)

Some general rules of thumb:

1. **Deliver content in multiple ways**, e.g. text-to-speech for those that cannot read or that cannot process writing
2. **Easily understood content** – text that is easily understood
3. **Focus attention** on important content to maintain focus and prevent distraction or overstimulation
4. **Minimize distractions** such as unnecessary content or advertisements
5. Consistent web page layout and navigation
6. Consistent styles on things like visited and unvisited links
7. **Divide** long processes into logical, essential steps with progress indicators
8. Easy website authentication
9. Simple, clear error messages and simple error recovery

# Cognitive or Neurological Diversity (contd.)

Some additional tips that cover both cognitive/neurological challenges but also education level:

- Define unusual words and word usage. This is useful for non-native English and those with ASD (e.g. sarcasm).
- Define abbreviations using the `<abbr>` element or ruby text.
- Use short, simple words and sentences. Use active voice in the present indicative tense. Use proper spelling and grammar.
- Provide a TL;DR at a lower reading level (Flesch-Kincaid, Lexile scale)
- Teach the pronunciation of an unusual word or use a `<ruby>` element to do it.

# Slow Internet Connections


Only 80% of Americans subscribe to home broadband internet.

The others may rely on lossy mobile (3G/4G/5G), dial up, or do not have Internet at all. For those with slow connections:

1. Use caching liberally. HTTP should issue liberal cache controls to prevent re-fetching data.
2. Create a low bandwidth version of your app. Be aware that images and videos will use up a lot more data than text.
3. Provide multiple versions of images and videos that are compressed or of lower quality.
4. Reduce animations and the size of images or replace with a text-only version.

# Test, Test, Test

You should consider accessibility early on.

It gets very difficult to consider later on, or to retrofit your app for it.

# Test, Test, Test (contd.)

We should consider adding automated accessibility testing. In higher stake apps, it can be useful to speak with a focus group:

1. Does the data picker widget provide use for those with screen readers?
2. If content updates dynamically, do the visually impaired have some way of knowing about it (a bright visual clue or a sound)?
3. Are my UI controls accessible to both the mouse, touch and keyboard?

# Managing Expectations

100% accessibility is not possible to achieve.

For example, a 3D pie chart cannot be interpreted by a screen reader. Instead, create a table and speak the content of the table.

Or just get rid of the pie chart entirely. Table is more accessible for everyone, quicker to code and less CPU intensive..

# Managing Expectations (contd.)

But, for a web app displaying a gallery of 3D art, it would be unreasonable to expect every piece of art to be accessible.

In these cases where it is not possible to address accessibility in a way that satisfies you or your users, it's wise to publish an **accessibility statement** that explains your attempts at making your site accessible. It really helps users understand your intent an engage with people with different needs.

# W3C Accessibility Standards

W3C hosts accessibility standards know as WGAC.

It lays out four **needs** for content:

1. Perceivable
2. Operable
3. Understandable
4. Robust

# What's New in WCAG 2.2 (2023)

Released October 2023, WCAG 2.2 adds 9 new success criteria to improve accessibility for users with cognitive and motor disabilities. Some highlights:

- Focus Not Obscured (2.4.11)
- Dragging Movements (2.5.7)
- Accessible Authentication (3.3.7)

# HTML is Accessible

**Caveat:** As long as text isn't too small, or hidden.

HTML is semantic. It is thus readable by sighted viewers and also be screen readers.

Screen readers use headers and other semantic structures to find information in a document.

**Tip:** In the developer tools, turn off CSS and see how the page looks. Does the raw **and its order** text make sense?

Native keyboard accessibility is already provided by the browser using the tab key.

# CSS Can be Dangerous

Some tips:

1. Try disabling CSS and see how it affects the main content of the page.
2. Use the correct semantic elements to mark up different content in HTML. Do not do this in HTML.
3. The source order of the content should make sense without CSS.
4. Interactive elements like buttons and links should have appropriate focus, hover and active states.

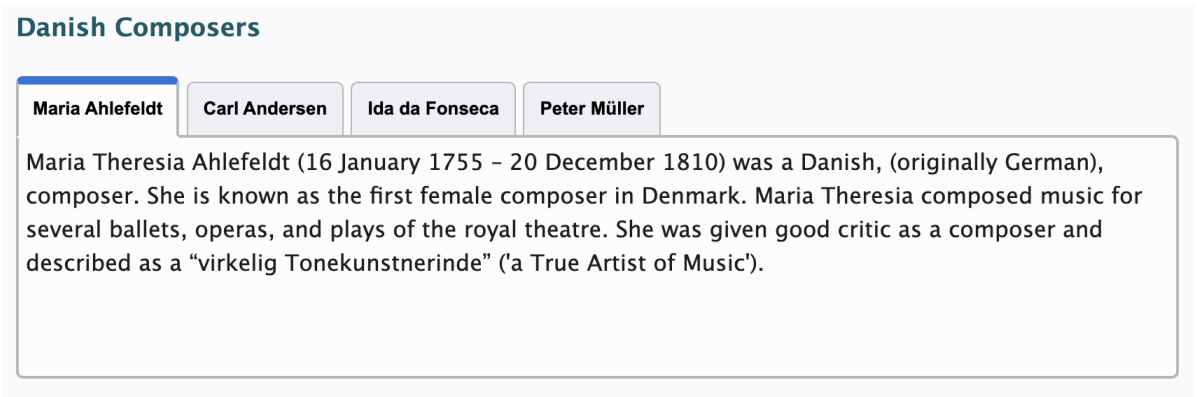
# CSS Can be Dangerous (contd.)

In your color schema, you should make sure that the foreground and the background color contrast well. You can use a tool like this one to do this.

# Be Careful Hiding Content

Be careful to not use `visibility: hidden` or `display: none` as the content is also hidden from the screen reader. More information here.

Tabs are ok because the content is not physically being hidden. The screen reader can still read the text aloud.



# ARIA

Simple HTML form controls are usually accessible, but if you are building more complex controls, consider using WAI-ARIA (Accessible Rich Internet Applications).

It provides additional new HTML attributes for complex form controls and updating panels.

# Auditing Tools

There are a few accessibility auditing tools you can use:

- **Lighthouse** in Chrome DevTools, audits tab
- axe DevTools (by Deque)
- WAVE for quick visual feedback
- NVDA + Firefox

# Accessibility Testing Checklist

1. Make sure your HTML is as semantically correct as possible. Validating it is a good start, as is using an Auditing tool.
2. Check that your content makes sense when the CSS is turned off.
3. Make sure your functionality is keyboard accessible. Test using Tab, Return/Enter, etc.
4. Make sure your non-text content has text alternatives. An Auditing tool is good for catching such problems.
5. Make sure your site's color contrast is acceptable, using a suitable checking tool.
6. Make sure hidden content is visible by screen readers.
7. Make sure that functionality is usable without JavaScript wherever possible.
8. Use ARIA to improve accessibility where appropriate.
9. Run your site through an Auditing tool.
10. Test it with a screen reader.
11. Include an accessibility policy/statement somewhere findable on your site to say what you did.
12. Provide a dark mode or respect user's color scheme preference
13. Avoid animation unless user explicitly opts in (via settings or reduced-motion detection)

1 Accessibility

2 Information Retrieval
- Search
- Recommendation Systems

# Information Retrieval

Information Retrieval (IR) is a somewhat old-fashioned term that refers to the modern concept of finding information from a large collection data.

Information Retrieval includes, but is not limited to, the following tasks:

1. Search including web Search
2. Recommendation systems
3. Large language models (LLMs)

IR involves issuing a **query** and receiving a series of relevant records. A **database**, **data store** or **index** powers the system.

# Search

Search is perhaps the most common form of information retrieval but it is not limited to web search.



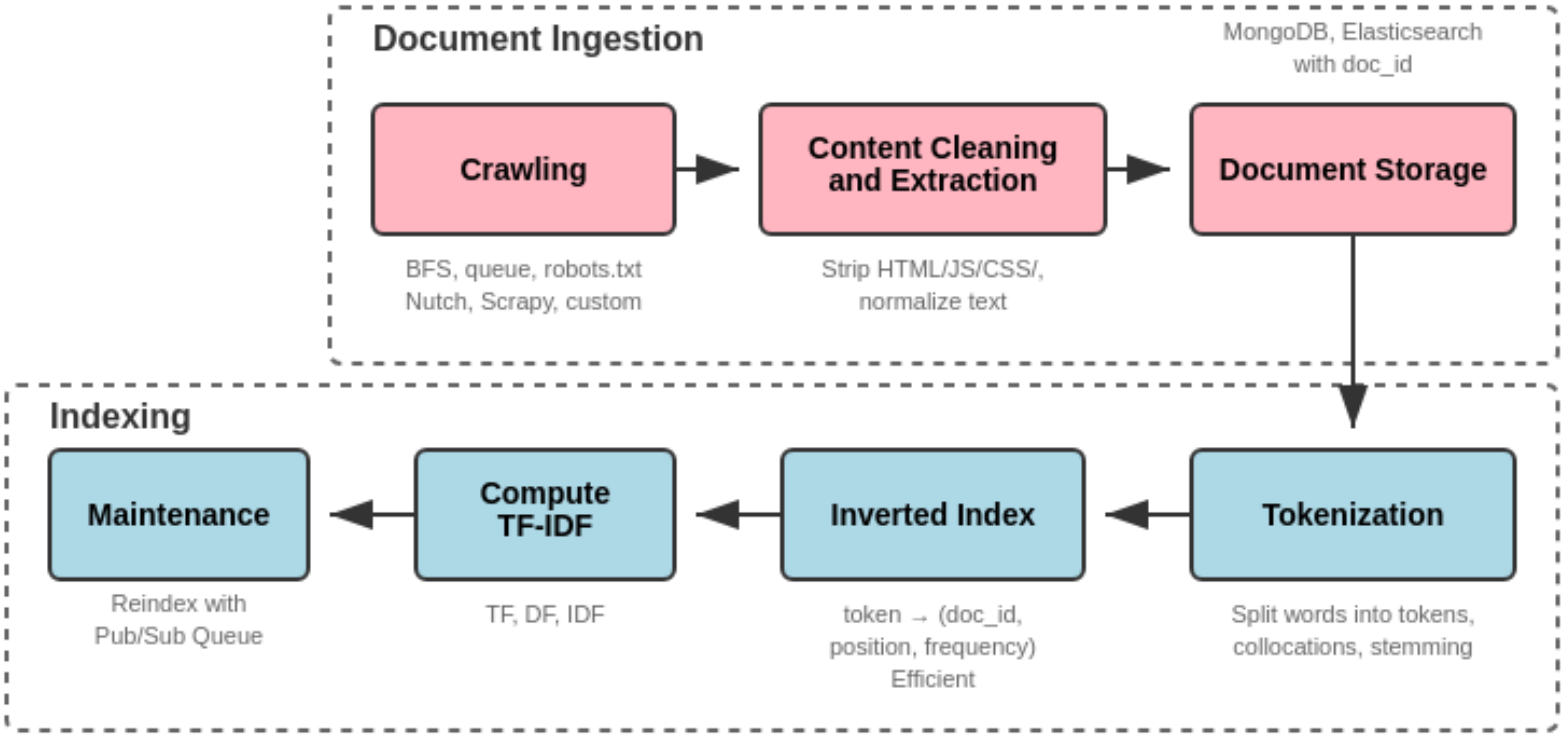


# Conventional Search

Search, including web search, generally follows this flow:

1. Document ingestion and indexing
2. Querying

# Conventional Search: Ingestion/Indexing



# Conventional Search: Ingestion/Indexing (contd.)

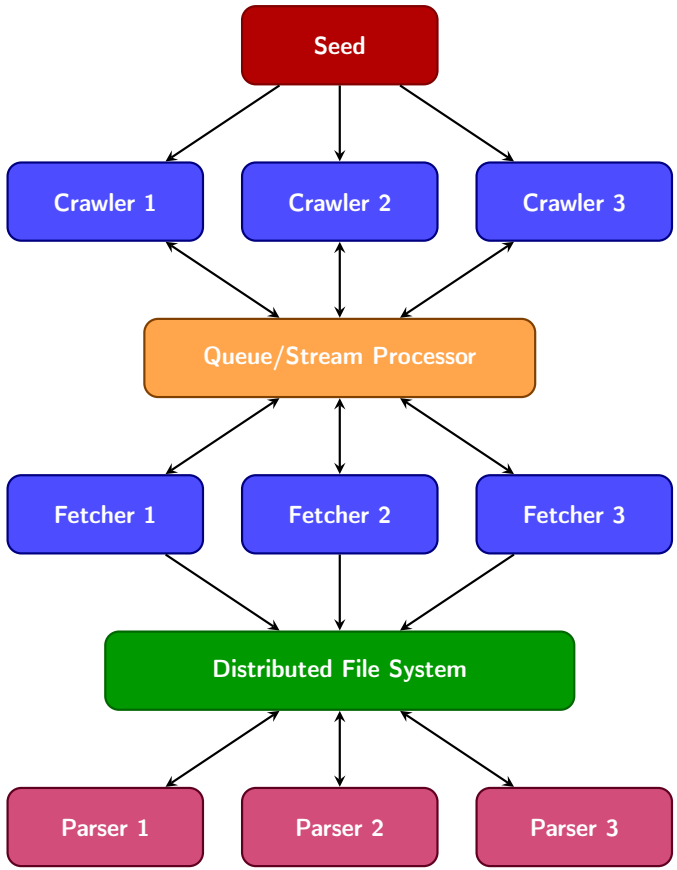
The crawling and ingestion phase includes the following:

1. Using breadth-first search or similar to discover content. We respect `robots.txt`.
2. In my implementation, I add these links to a queue.
3. In enterprise search, the page is processed:
   - Strip HTML, CSS, JS
   - Remove punctuation, remove stopwords, convert to lowercase
   - Apply basic NLP to detect entities and collocations
   - (In my implementation, this is a separate step)
4. Store the cleaned text in MongoDB, Elasticsearch etc. with a unique `doc_id`

# Conventional Search: Ingestion/Indexing (contd.)

To the right is one architecture I have used for distributed web crawling and scraping.

There are many others and I do not claim that mine is the best.



# Conventional Search: Ingestion/Indexing (contd.)

The indexing phase includes:

1. Split the cleaned text into tokens, collocations and entities
2. Stemming (running $\rightarrow$ run)
3. We currently have an index `doc_id` $\rightarrow$ `tokens`
4. Now create *inverted index* `token` $\rightarrow$ `doc_ids`
5. Compute and store statistics (namely TF-IDF) for each token in each document

# Conventional Search: Ingestion/Indexing (contd.)

TF-IDF measures the importance of a term to a document. Stopwords have very low TF-IDF. **Why?**

$$\text{TF-IDF}_{td} = \frac{f_{td}}{\sum_{t' \in d} f_{t',d}} \times \log\left(\frac{N}{1 + n_t}\right)$$

There are better measures of importance, such as BM25, word embeddings and BERT, cross-encoder models but TF-IDF is a good start.

# Conventional Search: Ingestion/Indexing (contd.)

A measure of influence of an individual web page can be found in **PageRank** which is a graph-based algorithm that ranks web pages based on the number and quality of links pointing to them.

$$PR(B) = (1 - d) + d \sum_{i \in \text{In(B)}} \frac{PR(i)}{L_i}$$

Where $PR(B)$ is the PageRank of page $B$, $d = 0.85$, In(B) is the set of pages linking to $B$, $PR(i)$ is the PageRank of page $i$, and $L_i$ is the number of links on page $i$. It is no longer the main signal used by Google (previous called BackRub).

# Conventional Search: Retrieval

Given a query $q$ we need to return documents that are relevant to the query.

1. Process the query (e.g. remove stopwords, stemming, punctuation, collocations, tokenize)
2. Look up each term in query in the inverted index
3. Perform the intersection or union of the document IDs
4. Compute a vector or TF-IDF for the terms in the query and in each document.
5. Compute a similarity score between both vectors
6. Rank according th the scores
7. Filter out certain documents

# Conventional Search: Retrieval (contd.)

Human evaluation and live traffic can be used to **learn to rank** using ML models. Popular queries and their ranked results should be cached.

A common distance metric still used today is **cosine similarity**.

$$\cos(\theta) = \frac{V_q \cdot V_d}{\|V_q\| \, \|V_d\|}$$

# Open Source Search Infrastructure

**Crawling/Scraping**



**Indexing**


**Retrieval/Search**


# Modern Search Infrastructure


Pinecone is a managed **vector database** that provides a simple API for storing and retrieving vectors. It is used for similarity search, recommendation systems, and other applications that require fast and scalable vector search capabilities.

# Vector Databases

Vector databases use vector embeddings to represent data points such as documents and terms in a high-dimensional space.

1. The user trains their own embeddings or uses pre-trained embeddings (e.g. BERT, Sentence Transformers)
2. Each vector embedding (document or term) is upserted into the vector database with an ID and some metadata
3. Each query is encoded using the same model.
4. The vector database performs a similarity search (cosine distance or nearest neighbors) and returns the most similar vectors and their IDs

Vector databases replace the indexing and retrieval phases of search.

# What is the Point?

There are two advantages to learning about search:

1. It may help you understand SEO for your web app
2. You may/should implement a search engine for your web app.

Elasticsearch, Apache Solr, Algolia, Redisearch are some options.

# References


- Introduction to Information Retrieval by *Manning, Raghavan, Schütze*
- CS 276: Information Retrieval and Web Search by *Manning and Nayak*
- PageRank Citation Ranking *Page, Brin et. al.*

# Recommendation Systems

Recommendation systems are a form of information retrieval that suggest items to users based on their preferences, behavior, and other factors.


# Recommendation Systems: Ingestion

These sites capture the following data:

1. General interaction data
2. Active interaction data
3. Passive interaction data
4. Item metadata
5. User metadata

All of this data is logged and stored in various data stores, or data warehouses.

# Recommendation Systems: Modeling

Typically, we construct a sparse matrix X where each row is a user and each column is an item.

Thus $X_{ij}$ is the interaction (e.g rating, click) of user $i$ with item $j$.

The value of $X_{ij}$ is often binary (0 or 1), a frequency, rating etc. and are usually scaled (e.g. mean centering, z-score).

# Collaborative Filtering

Then we decompose the matrix X into two matrices U and V using either:

1. Truncated Singular Value Decomposition (SVD) where $k$ is the number of latent factors: $X \approx U_k \Sigma_k V_k^T$
2. Alternating Least Squares (ALS) / Stochastic Gradient Descent (SGD): $X \approx UV^T$

Other methods include **Non-Negative Matrix Factorization**, **Neural Collaborative Filtering** and **Deep Learning**.

# Collaborative Filtering (contd.)

In the Truncated SVD case, we can use the singular values to compute the similarity between users and items.

- $U_k$ is the user-user similarity matrix (e.g. PYMK)
- $V_k$ is the item-item similarity matrix (e.g. similar items)
- $\Sigma_k$ is the importance of each latent factor (e.g. similar to eigenvalues)

# Collaborative Filtering (contd.)

Then, we can score each item vs. each user to see how much they may like it.

$$\hat{r}_{ui} = u_u^T \cdot v_i$$

To compute the top-$k$ for a user:

$$\hat{r}_{u\cdot} = U_u \cdot V^T$$

then sort by predicted score in decreasing rating after filtering out items the user has already interacted with.

# Collaborative Filtering (contd.)

Finally, to get recommendations for all users:

$$\hat{R} = U \cdot V^T$$

The dot product captures alignment in the latent space. A dot product of 0 suggests little/no interest.

# Collaborative Filtering (contd.)

People You May Know (PYMK), roommate matching, high school scheduling and dating apps can use collaborative filtering.

The quantity $U \cdot U^T$ computes similarity betwen users. The quantity $V \cdot V^T$ computes similarity between items.

# Collaborative Filtering (contd.)

$$U \cdot U^T$$

- Can be used to find similar users
- Can be used to find users that liked the same items
- Can be used to find users that liked similar items
- Can be used as an input to spectral clustering, community detection and graph-based segmentation

Whereas $V \cdot V^T$ can be used to find similar items without respect to the user. Point 4 above also applies.

# So What is a Quick Way to Do This?

Yep, with a vector database!

1. Store each user vector (rows of $U$)
2. Store each item vector (rows of $V$)
3. Find the most similar users to a given user $UU^T$
4. Find the top-$k$ most relevant items for a user $UV^T$

But instead of computing a dot product for all items, vector databases return top-$k$ vectors in milliseconds by using optimized approximate nearest neighbor (ANN) search algorithms using algorithms like HNSW, IVF, or PQ.

# So What is the Point?

Recommendation systems are useful to users and help build trust with you and your app.

They also increase engagement, retention and thus. . . monetization.

# References


- Recommender Systems: The Textbook by *Aggarwal*
- Google's Machine Learning for Recommendation Systems
- CS 246: Mining Massive Data Sets by *Leskovec*
- Deep Neural Networks for YouTube Recommendations*Covington*, *Adams*, *Sargin*