Data 188: Introduction to Deep Learning

Intro to Recommendation Systems

Speaker: Eric Kim
Lecture 21 (Week 13)
2026-04-16, Spring 2026. UC Berkeley.

Announcements

- HW4 released: "Transformers for NLP (machine-translation)"
  - Groups of 4!
    - Ed post: "(HW4) Group finder thread"
  - Start early!

- Final exam scheduling google form sent out
  - See Ed post: "(Action Needed) Final Exam Scheduling"
  - DSP students with exam accommodations: we sent you an email with a separate form.
  - **Due date: must submit form by Wed April 22nd 11:59 PM PST!**

Today's lecture

Recommendation systems

Metric learning

Today's lecture

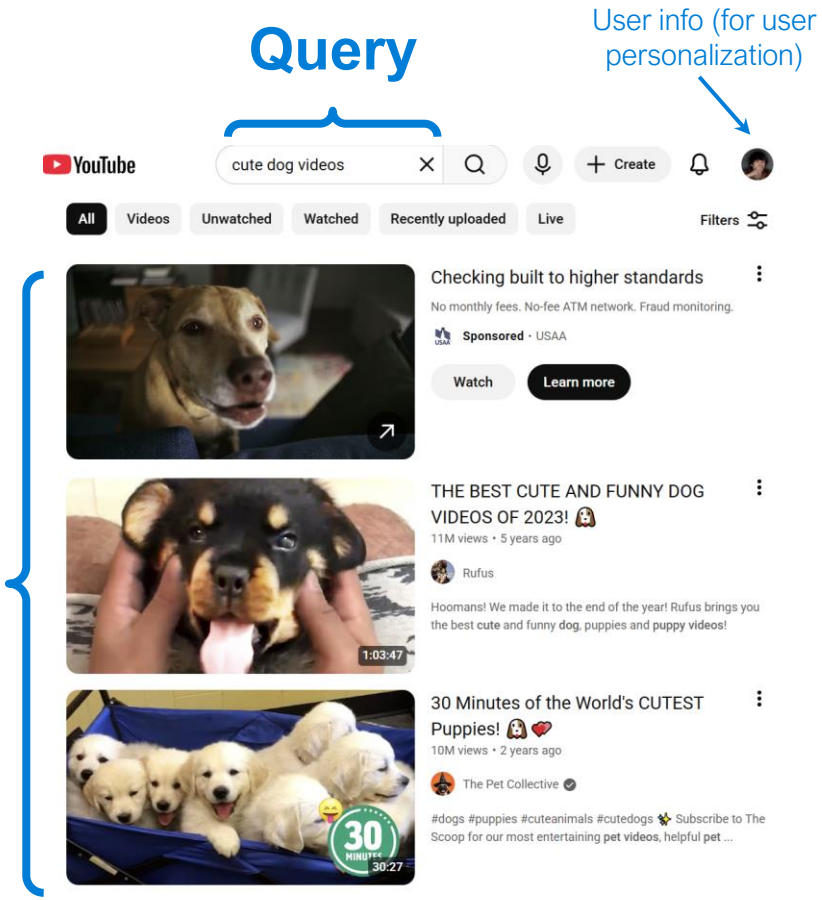
Recommendation systems

What is a recommendation system?

...and how do deep learning models fit in?

What is a recommendation system? ("recsys")

- Given a **corpus** of **items** (videos, websites, songs, products, etc): recommend the **User** something that they will like
- Examples:
  - Google: items are websites
  - YouTube/TikTok: items are videos
  - Amazon: items are products (shopping)
  - Pinterest: items are Pins



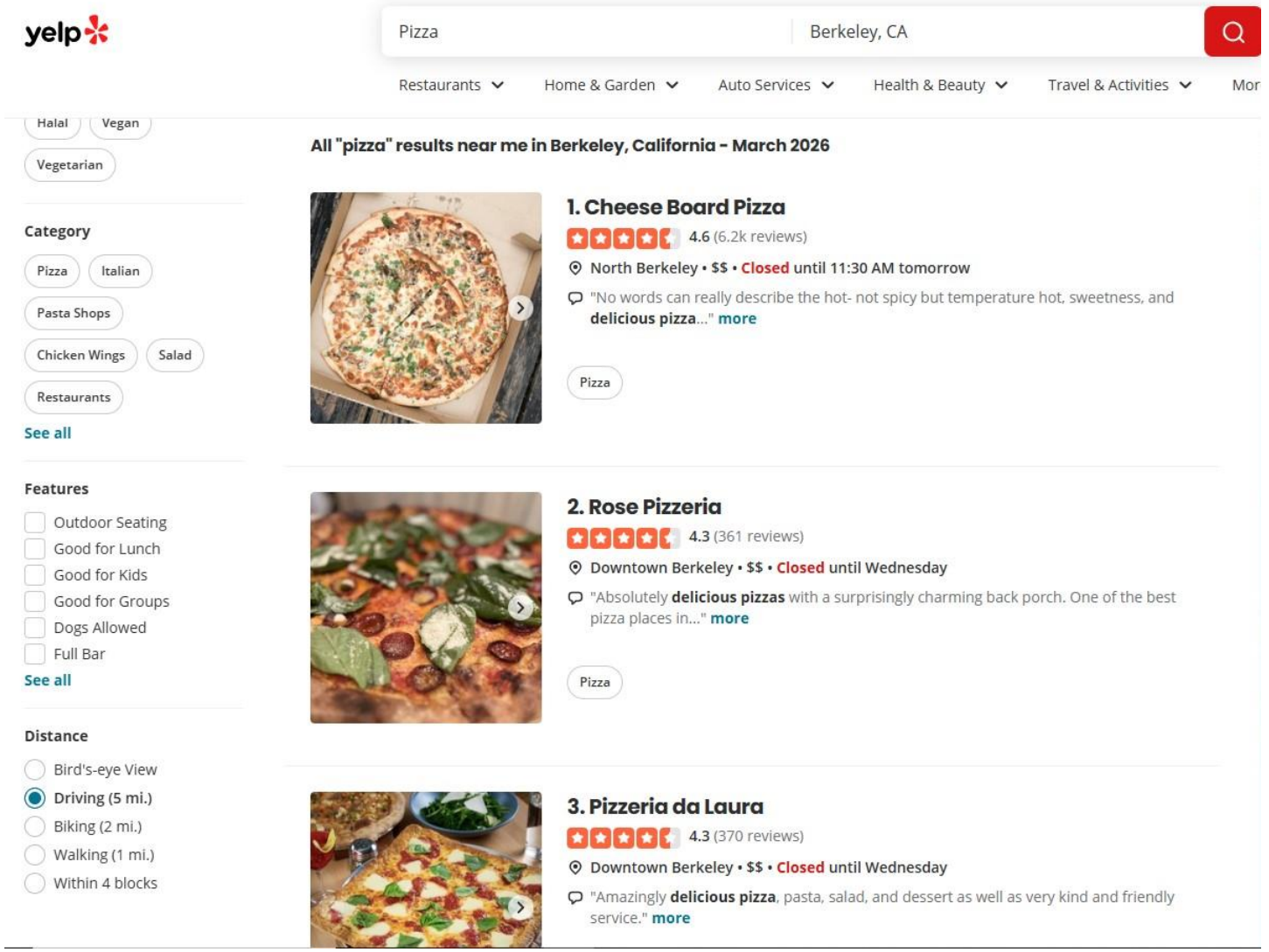
Retrieved results

More recsys examples

Query: text search query.
"Pizza"

Results: Top restaurants relevant to query.

Corpus: all restaurants indexed by Yelp.

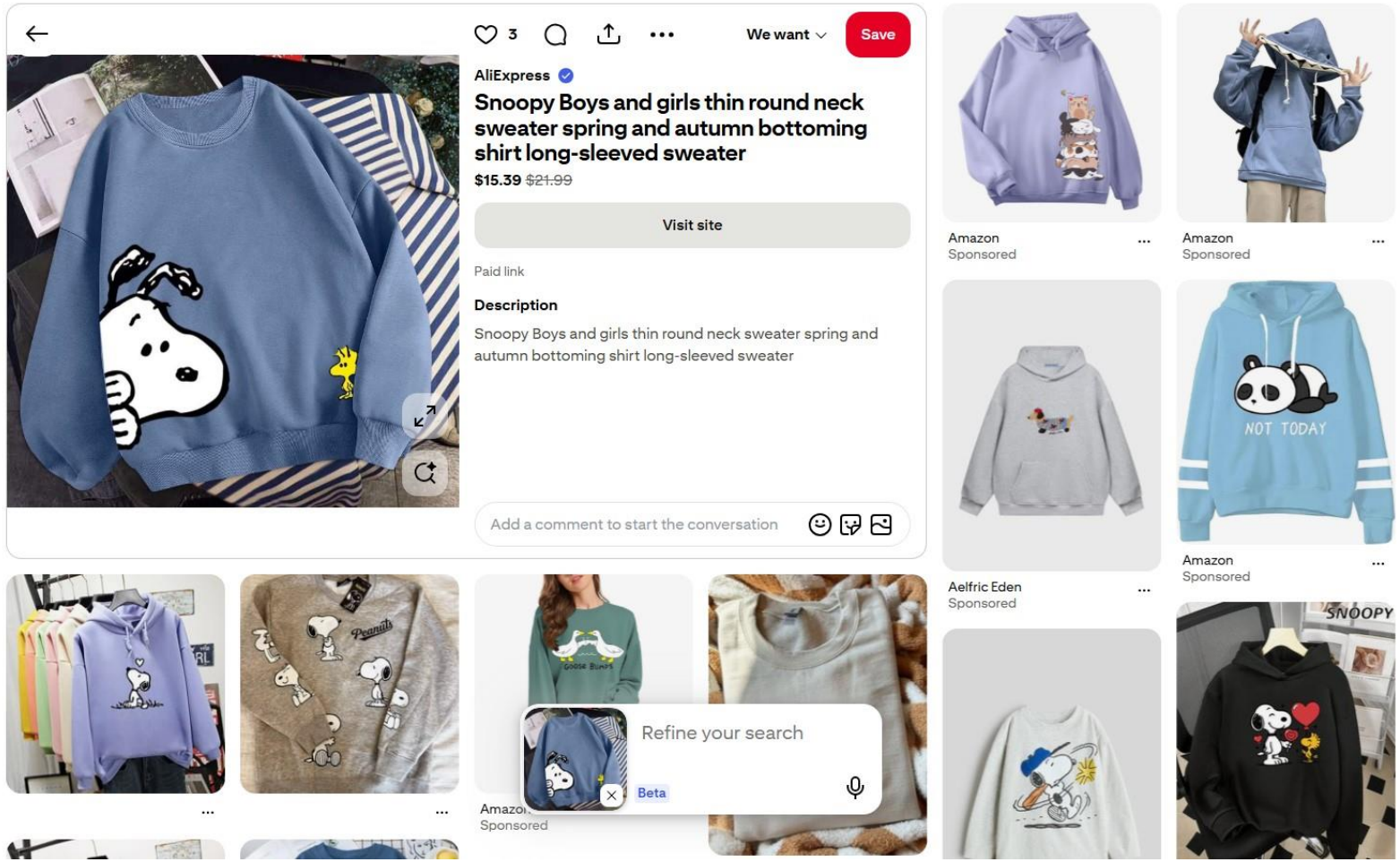


More recsys examples

Query: Image. **"Pin of a Snoopy sweater"**.

Results: Top images relevant to query image. **"Other sweaters with Snoopy (or cartoon characters)."**

Corpus: all images (Pins) created on Pinterest.

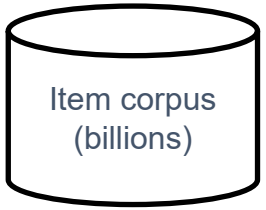


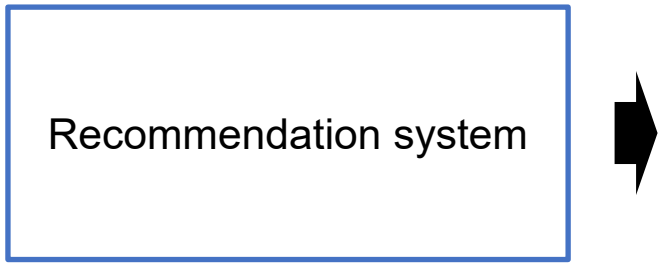
Recsys high-level system overview

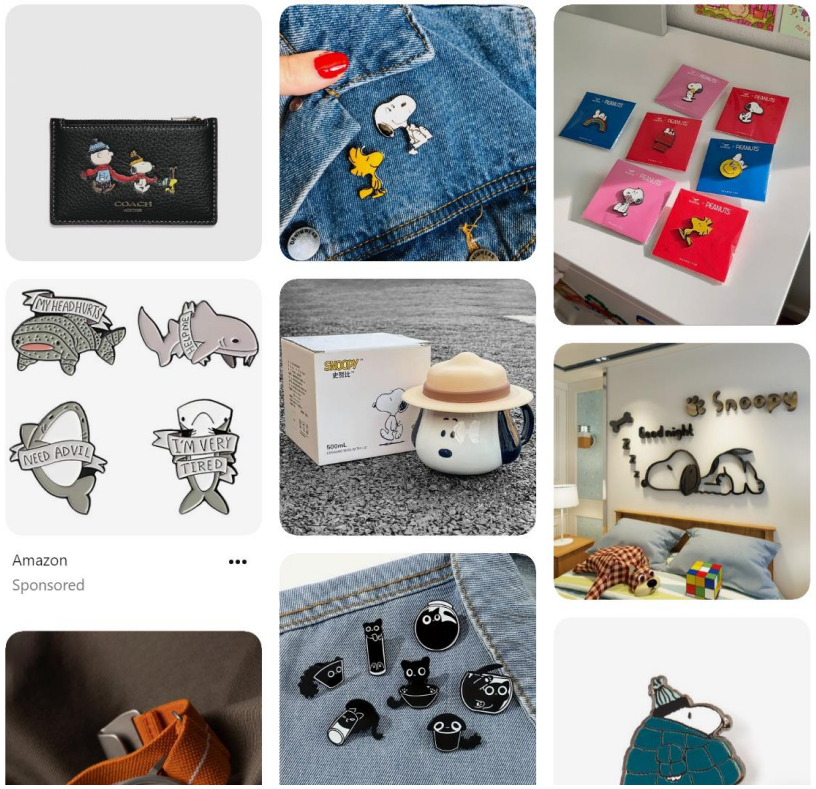
At a high level, a recsys performs:

Given a **query** and a **corpus**, return the results from the **corpus** that is most relevant to the **query**.

"item to item" recommendation







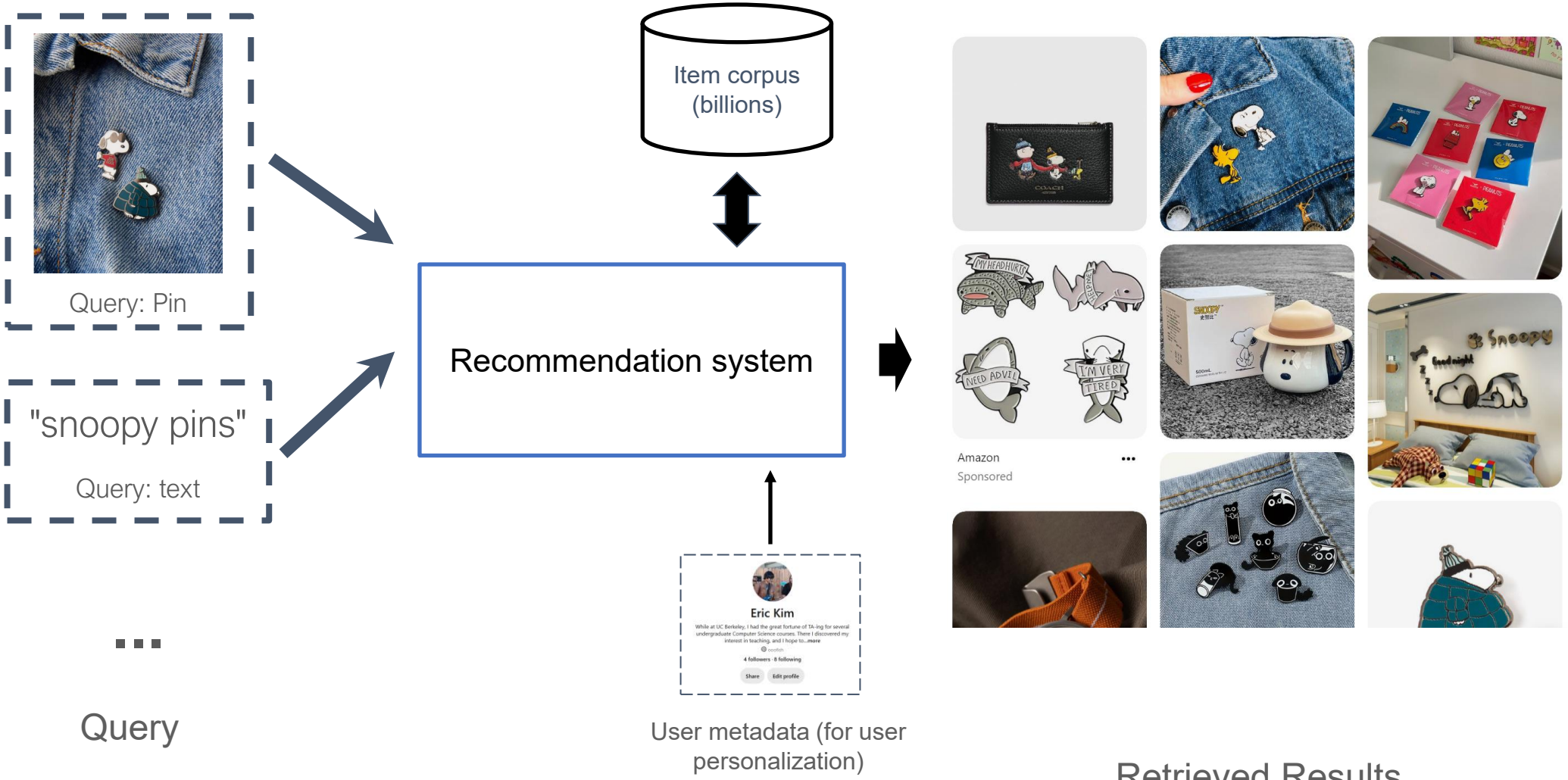
Query Item (websites, videos, products, Pins, etc)

User metadata (for user personalization)

Retrieved Results

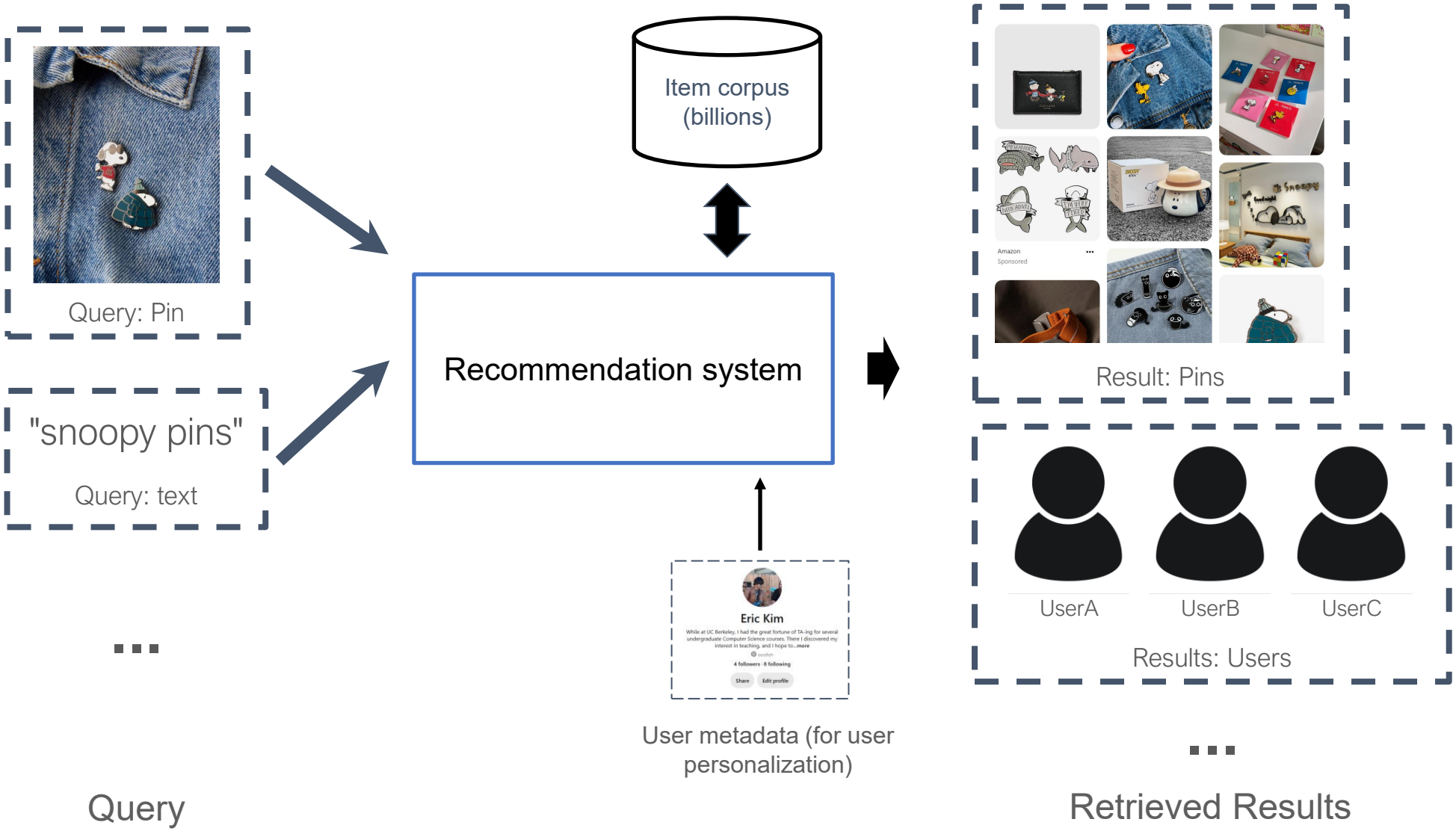
https://www.pinterest.com/pin/553802704115026l2/

Query can take many forms



https://www.pinterest.com/pin/553802704115026l2/

Results can also take various forms!



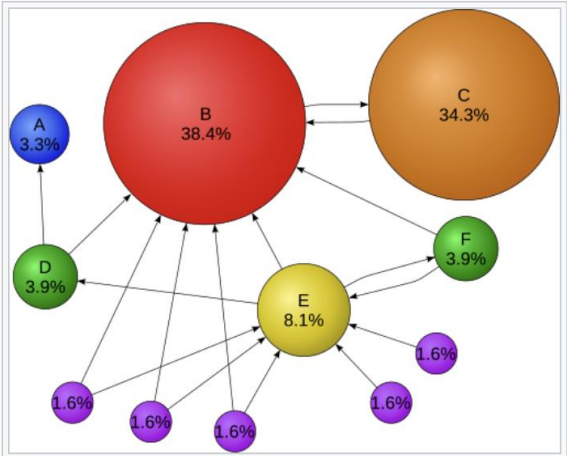
https://www.pinterest.com/pin/553802704115026l2/

Task: Related Content

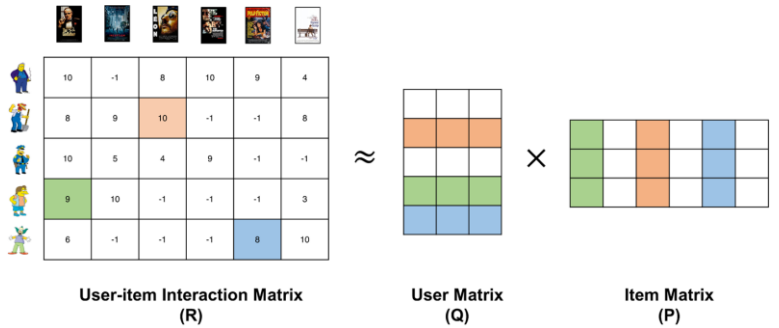
- Related content ("item to item"). Given a query item, recommend other related items
  - Goal: recommend content that the User is likely to engage with
  - Ex: if I'm shopping for white shoes, recommend me: other shoes from other brands, outfits that may go well with that shoe, etc.
  - (Optional) User personalization. How to recommend content that is tailored to a User's specific interest?

“Classic” recommendation system approaches

- PageRank (Google, 1996/1997). Represent website inbound/outbound connections as a graph, and utilize graph theory to compute a quality score for each page.
- Text-based information retrieval techniques
  - TF-IDF score, Bag-of-words models
- Collaborative filtering (aka matrix factorization)
- While older methods are still valid (and in use today at many companies), today we’ll focus on DNN-based approaches



A simple illustration of the Pagerank algorithm. The percentage shows the perceived importance, and the arrows represent hyperlinks.



https://buomsoo-kim.github.io/recommender%20systems/2020/09/25/Recommender-systems-collab-filtering-12.md/

Scoring functions

- Assume we have a **scoring function** that, given two items, outputs a similarity score between [0.0, 1.0]. 0.0 means "low similarity", 1.0 means "high similarity".
  - `f(item_a, item_b) -> [0.0, 1.0]`
- **Simple recommendation system**: given a query item, **score** all items, **sort** by score, and show the top K=50 results to the User.

```
sorted([f(item_query, item_candidate)
for item_candidate in item_corpus])
```

Query Candidate

f( , )=0.91

High similarity

f( , )=0.1

Low similarity

Recsys: scale challenges

To scale up to billions of items (and hundreds of millions of users), there are several challenges:

Data size. Suppose we represented each item as an embedding vector of 256 float32's. Loading all embeddings for 1e9 items => **1024 GB of CPU memory**

(likely can't fit all items into a single machine, especially since we usually have many features/signals for each item!)

Latency. Doing a naive for-loop over 1e9 items (and calculating all query<->item scoring functions) is too slow, especially for real-time recommendations.

Scalable Recsys system diagram: two stage

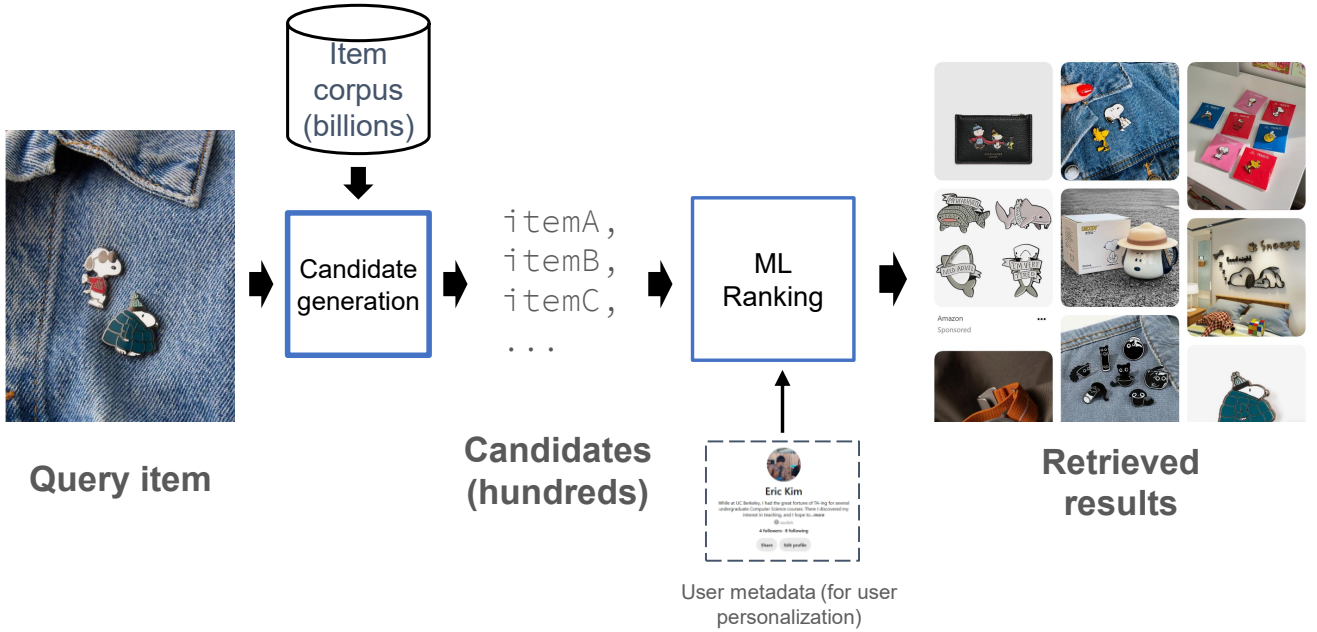
Idea: break up system into two stages:

Stage 1: "Candidate generation". Lightweight retrieval. Reduce corpus from billions to, say, hundreds.

Optimize for speed, scale, and recall. Tradeoff speed for precision.

Stage 2: "Ranking". Given Stage 1 results, determine best results (often via an ML ranking model).

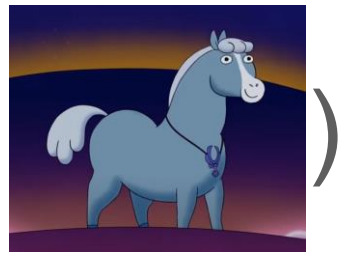
Optimize for precision (and scale). Since there are fewer candidates (100's vs billions), we can use heavier-duty ML models (and more intricate features).

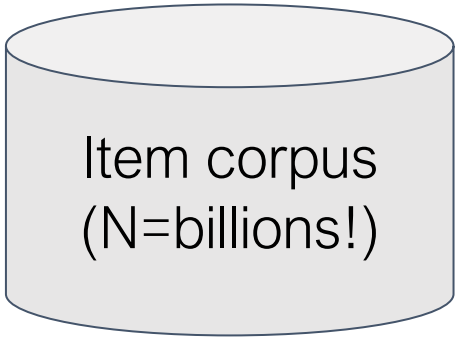


Embedding retrieval at scale

- Armed with a good embedding model and an embedding metric, we're nearly there to a retrieval system!
- Algorithm: given query item, compute similarity between each query and all items in the corpus. Sort by similarity
  - Aka "nearest neighbor search"
- **Problem**: corpus can be very large (Billions!). Linear search is too slow: we want results in real time (ie <200ms latency)
  - Also: corpus is too large to fit on a single machine!
- **Solution**: approximate distributed nearest neighbor!

f( , )=0.91

f( , )=0.1



Approximate nearest neighbor ("ANN")

- Idea: rather than compute “exact” nearest neighbor (too slow), compute approximate results (faster)
- Tradeoff: speed vs fidelity
- Popular algorithms:
  - Locality-sensitive hashing (LSH)
  - HNSW [link]
- FAISS (Facebook AI Similarity Search) is a popular open-source ANN library.
  - Supports GPU-accelerated ANN search!

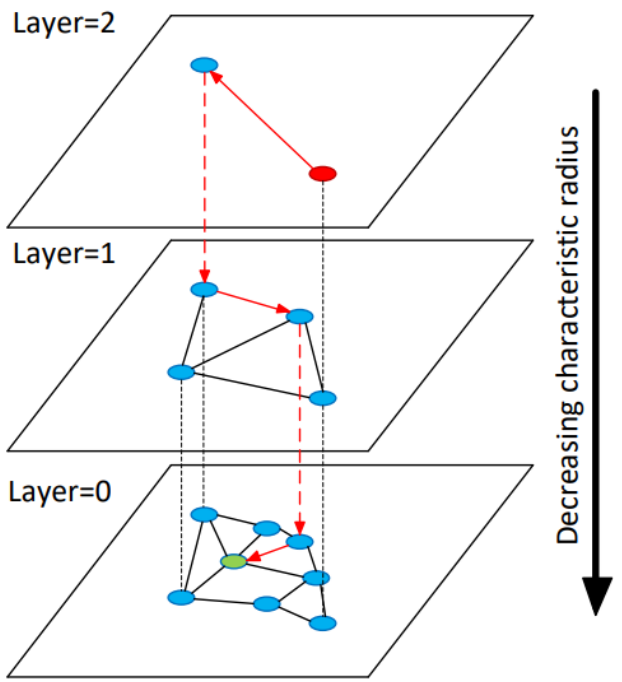
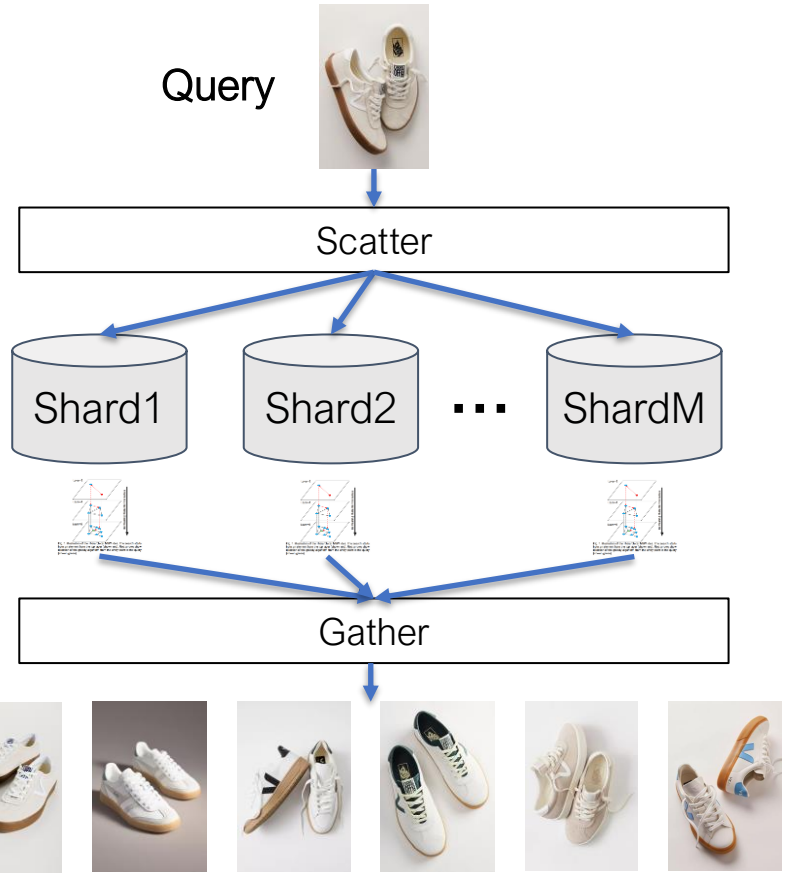


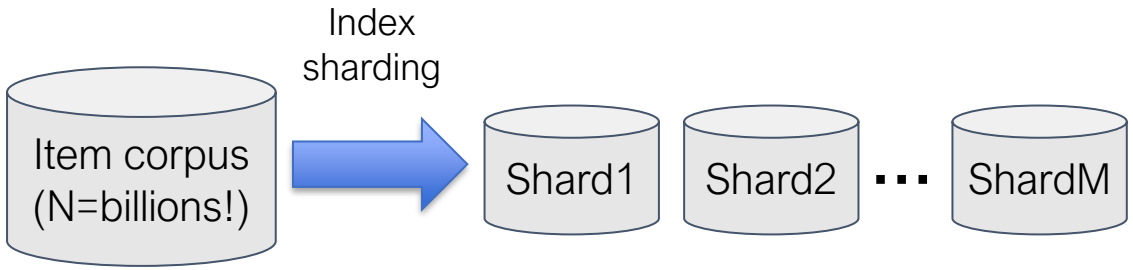
Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

Distributed ANN

Idea: split ("shard") the ANN index to M machines

"Scatter/gather" architecture





Full index is too big to fit on a single machine

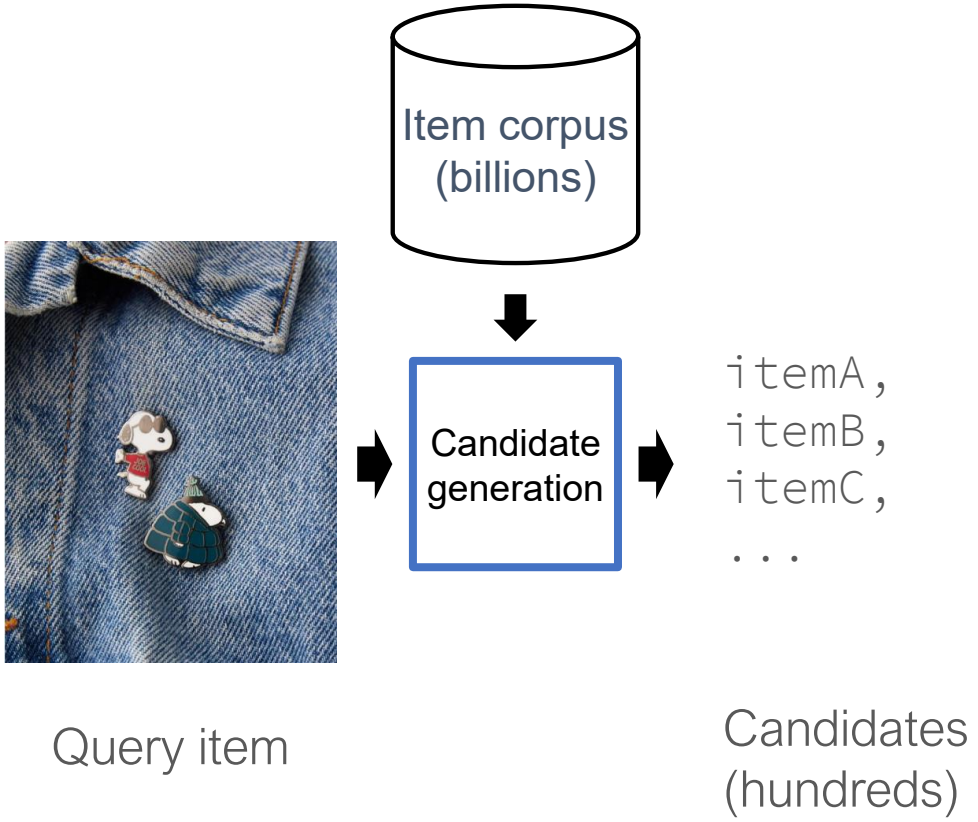
Each shard is small enough to fit on a single machine (ex: in CPU memory)

Scatter: each shard performs its own NN search

Gather: aggregate each shard's NN results to produce final NN results

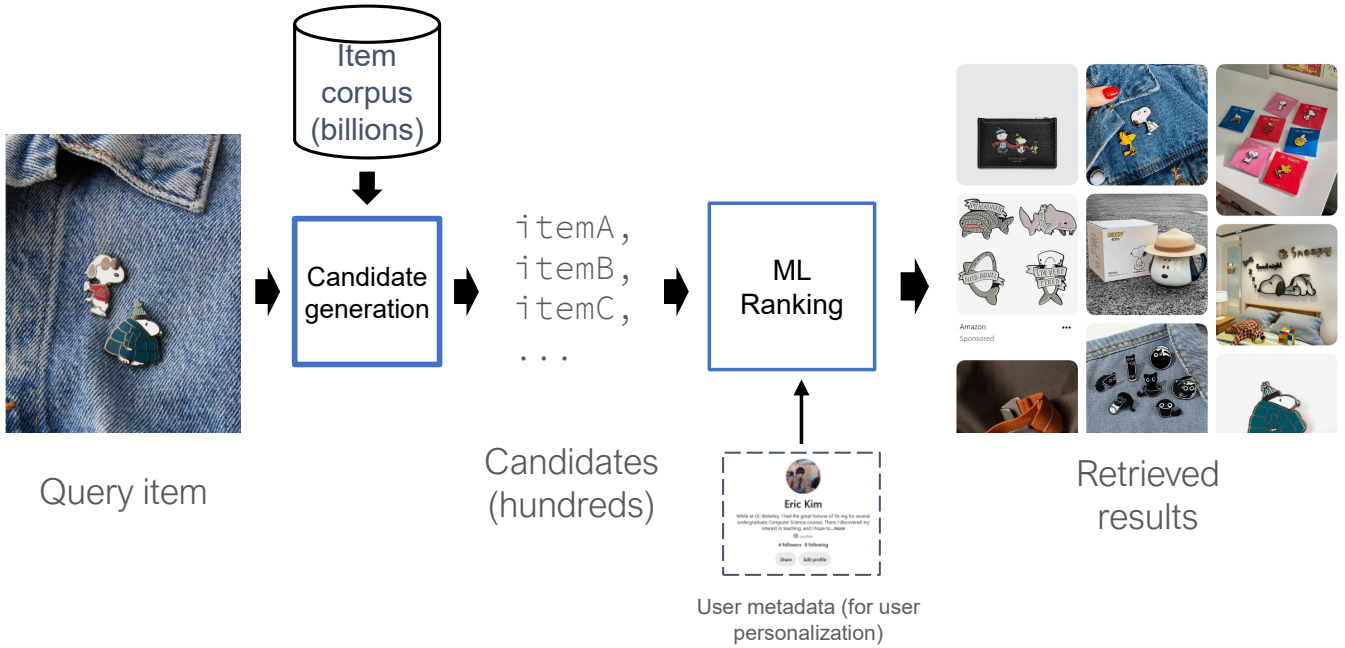
Candidate generation

- “initial lightweight retrieval”
- Goal: filter from Billions of corpus items down to hundreds.
- Design goals
  - Speed.
  - Prioritize recall over precision
- Popular choice: embedding model + ANN



Ranking stage

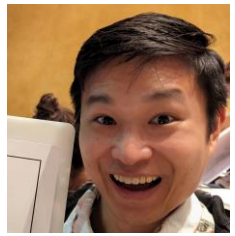
- Given candidates from candidate generator: rerank them via a ML model
- Since we have fewer candidates (hundreds, instead of billions), we can use heavier-duty ML models
- Optimize for business metrics (ex: user clickthrough rate, ad impressions, etc)
- Can inject User personalization here too!



ML ranking

- Task: Pointwise ranking. Given query_user and candidate_item, predict probability that query_user will click on ("engage with") the candidate_item.
  - **User features**: user demographics, user interests, recently engaged content, etc
  - **Item features**: image/text embedding, topic, etc

Candidate Item | User | Probability that User will click ("engage") with Item

f( , )=0

f( , )=1.0

Two-tower models

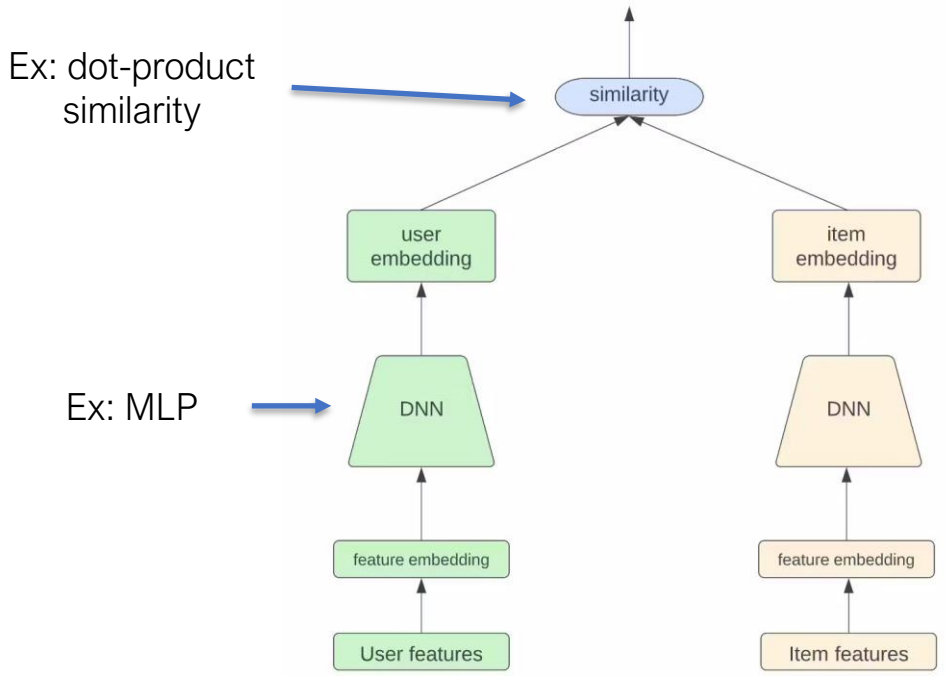
Training data: user engagement data.

Model architecture: "two tower model". Learn user/item embedding representations with a similarity function to predict engagement.

Loss function: binary classification. "Given User and candidate Item, did User click the Item?"

Positive examples: items that the User engaged with.

Negative examples: items that the User did NOT engage with.

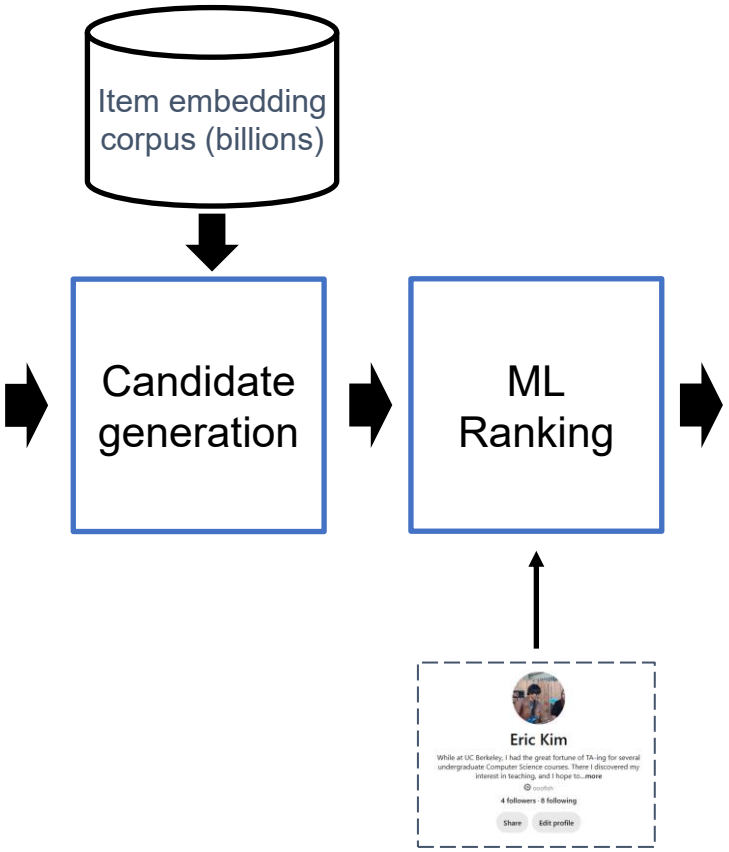


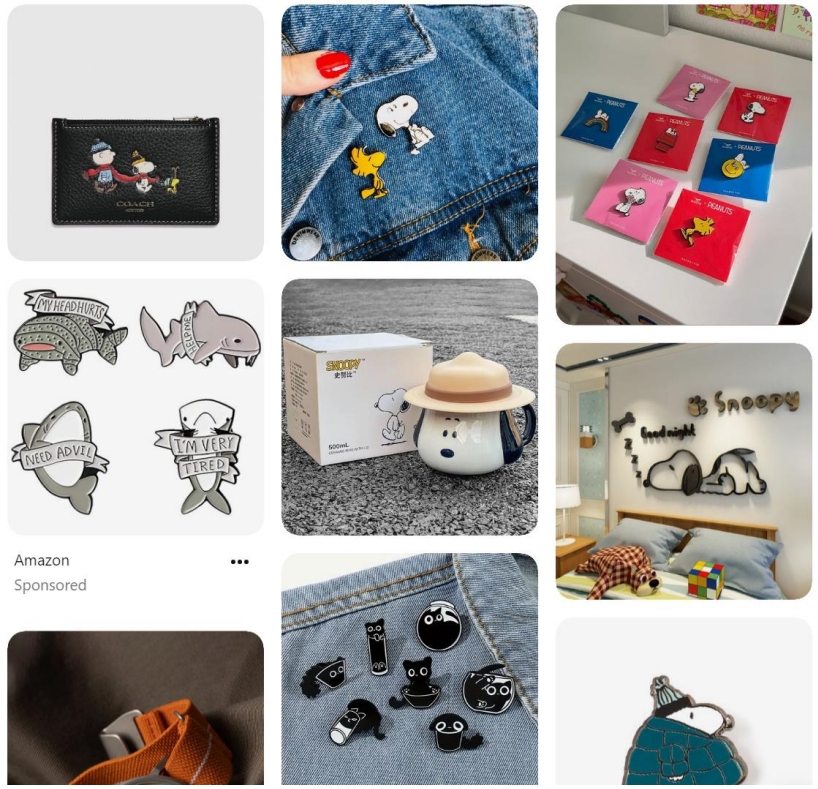
user_id	timestamp	user_action	item
erickim	2025-07-28 01:15:00	USER_WEBSITE_CLICK	https://c88c.org/su25/
erickim	2025-07-28 01:18:00	USER_VIDEO_WATCH	https://www.youtube.com/watch?v=dQw4w9WgXcQ&list=RDdQw4w9WgXcQ&start_radio=1
some_student	2025-07-29 15:12:00	USER_WEBSITE_CLICK	https://c88c.org/su25/
...	...	...	...

User engagement data logs

https://hackernoon.com/understanding-the-two-tower-model-in-personalized-recommendation-systems

Putting it all together: a recommendation system





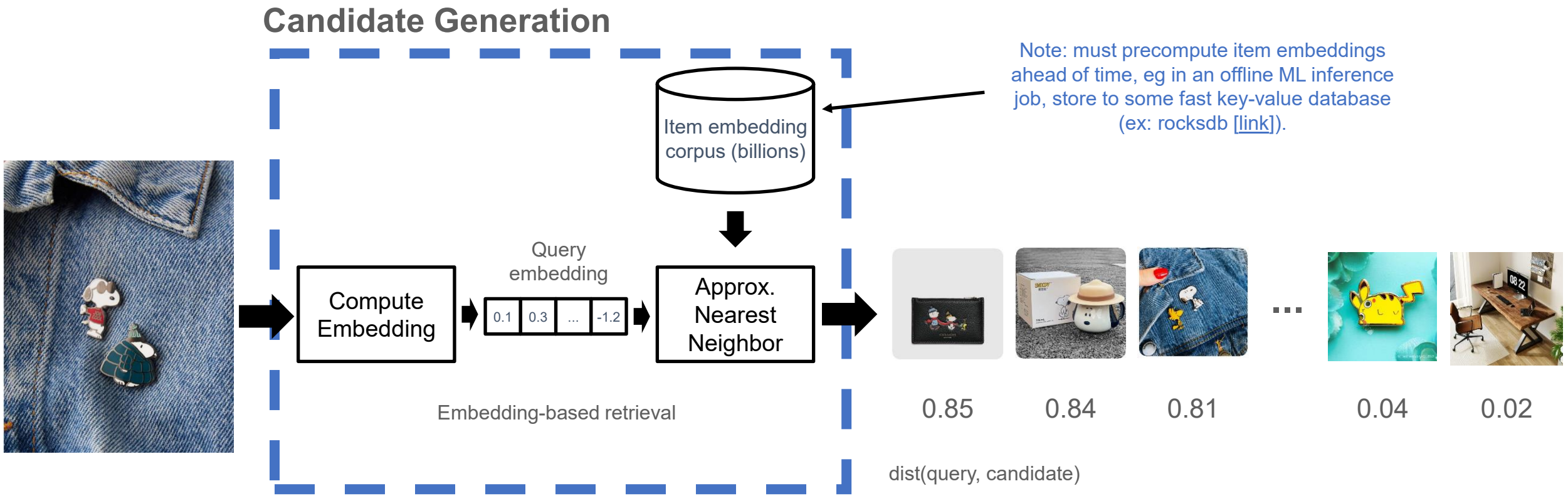
Query Item

https://www.pinterest.com/pin/55380270411502612/

User metadata (for user personalization)

Retrieved Results

Candidate generation: some details



Note: can also have multiple candidate generators, eg ones that focus on surfacing newly created content ("cold start problem")

Ranking: some details

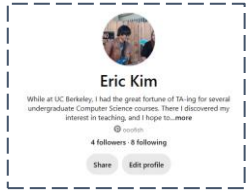
...

Retrieved candidates

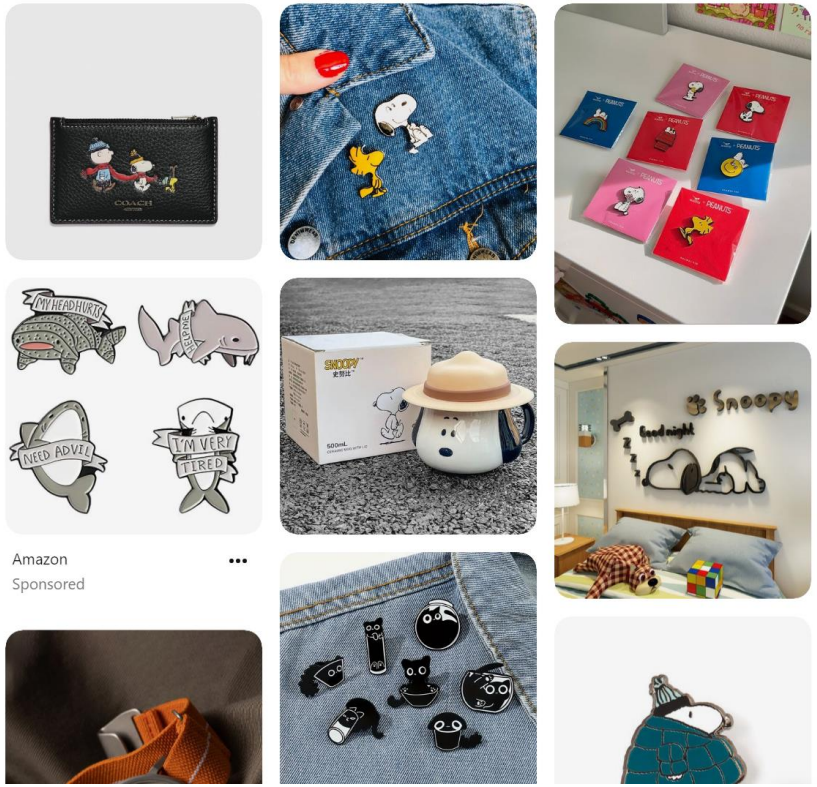
Ranking

Ranker (ML)

(optional) additional business logic



User metadata (for user personalization)



Final retrieved results (shown to User)

To learn more about a real-world retrieval system, see: "Related Pins at Pinterest: The Evolution of a Real-World Recommender System" [link]

Scoring functions: how to design?

- Recall: we want a **scoring function** that, given two items, outputs a similarity score between [0.0, 1.0]. 0.0 means "low similarity", 1.0 means "high similarity".
  - `f(item_a, item_b) -> [0.0, 1.0]`
- How to design a good scoring function `f(item_a, item_b)`?

f( , )=0.91

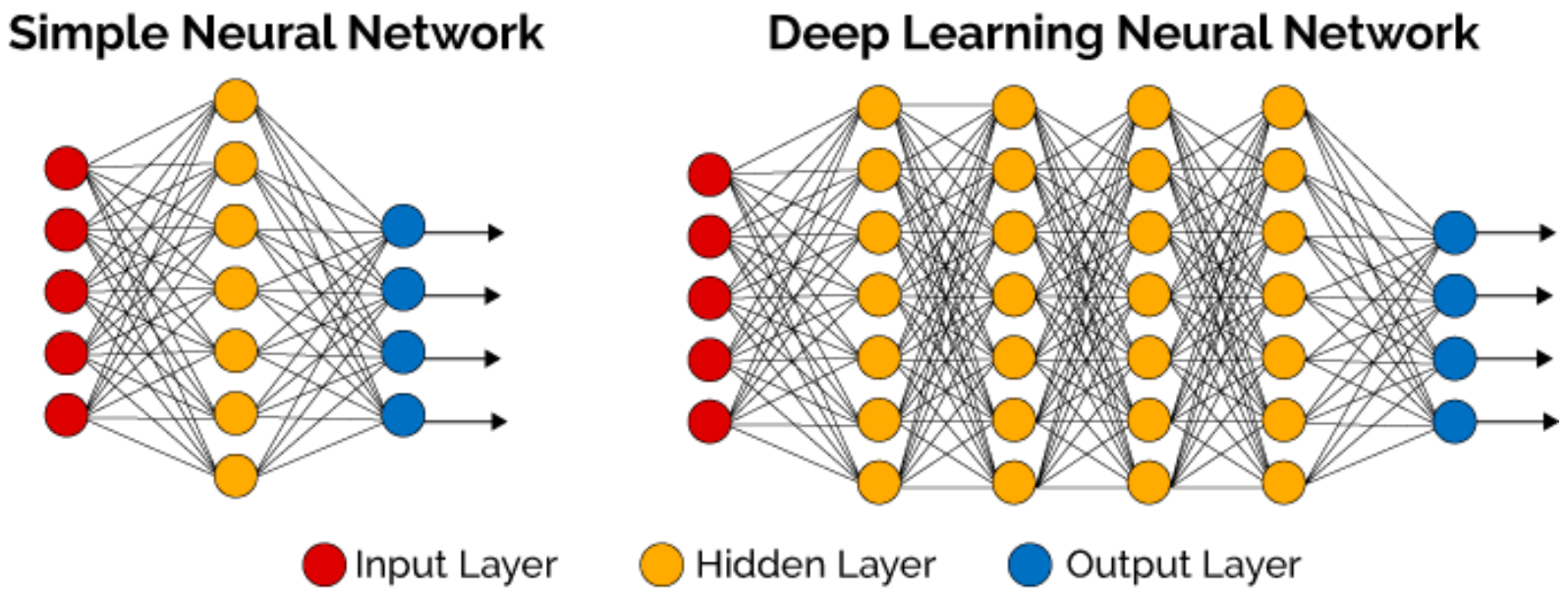
High similarity

f( , )=0.1

Low similarity

Deep learning: representation learning

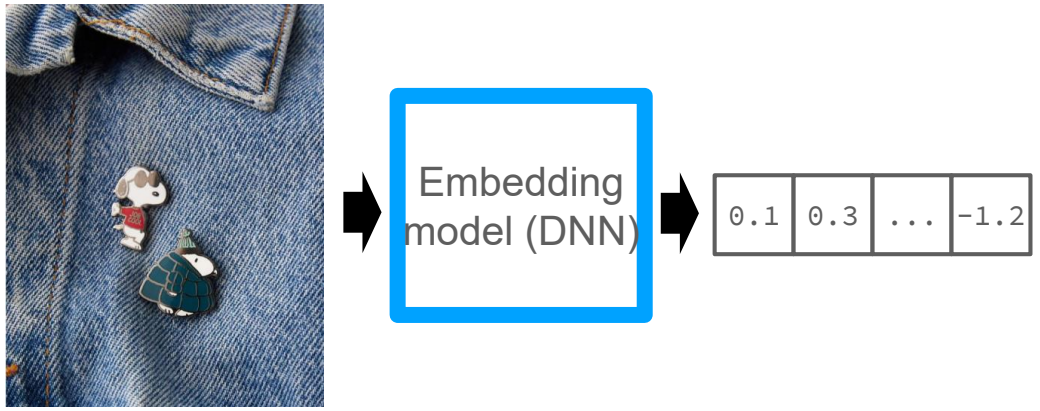
- Deep learning's strength is its ability to learn **strong (semantic) representations** for downstream tasks (classification, object detection, text generation, etc).
- Idea: let's leverage DNNs to learn item representations, aka "**embeddings**"!



https://medium.com/data-science/neural-network-embeddings-explained-4d028e6f0526

What is an embedding?

- An embedding is a high-dimensional vector representation, eg a 256 dimensional array of floats.
- Ideally, two items that are **similar** should have **similar embeddings**
- Learning: with enough training data (and GPU compute!), DNN's can learn extremely effective embedding spaces for downstream tasks

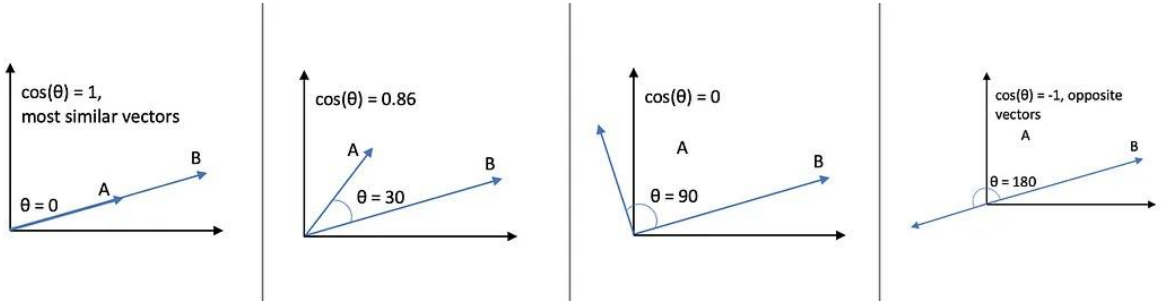


Item (ex: image, video, user, etc)

Item embedding ("representation")

How to compare embeddings?

- Popular approach: utilize simple embedding similarity measures to compare two items.
  - Ex: dot product, cosine similarity, L2 distance. Easy to implement!
- Great: now we know our scoring function `f(item_a, item_b)`!
- **Big question**: how do we learn the item embeddings?
  - In particular: how do we learn them such that embedding similarity metrics "works"?



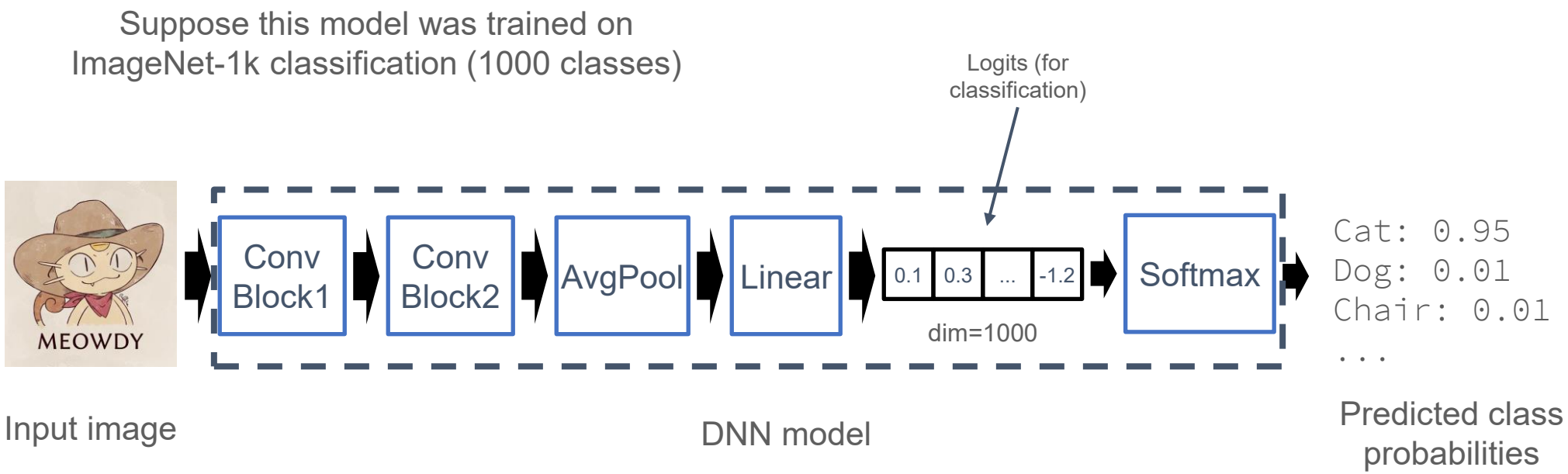
$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Pictured: cosine similarity between vectors A, B

https://towardsdatascience.com/cosine-similarity-how-does-it-measure-the-similarity-maths-behind-and-usage-in-python-50ad30aad7db

Designing embedding models

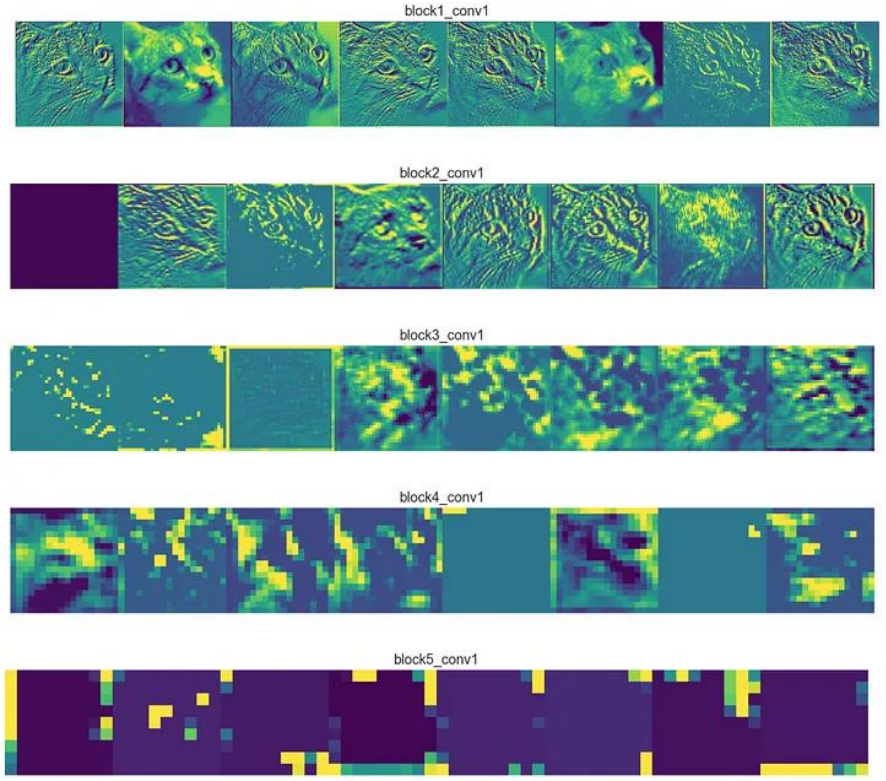
- Popular approach: take an existing DNN model trained for some task, and use some **intermediate feature** (aka "hidden state") as the **embedding representation**



Convnet: image embedding

In a Convnet, which intermediate features should we use as the image embedding representation?

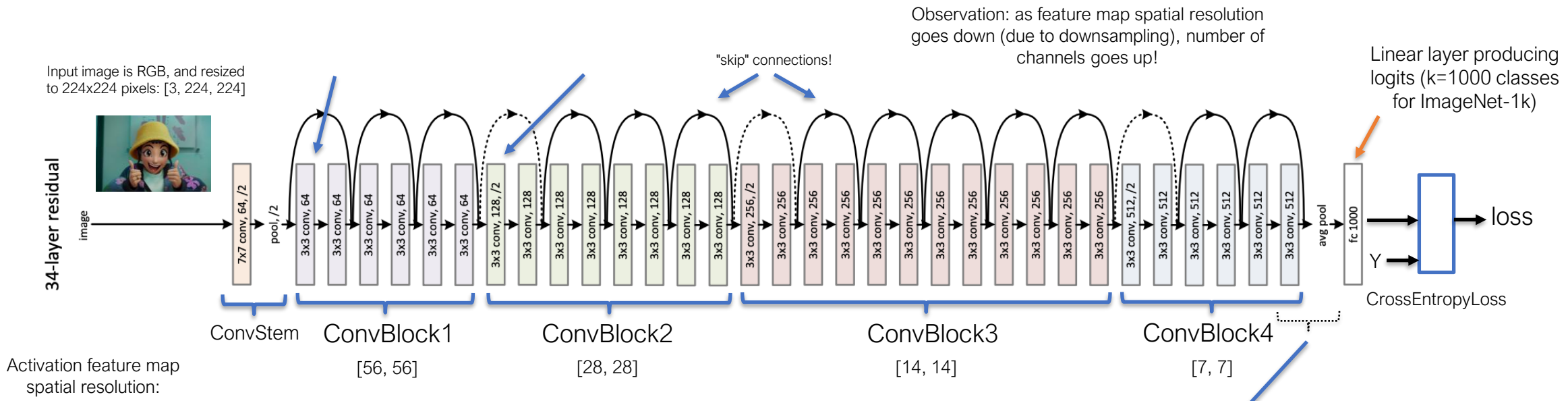
Low level: edge detectors



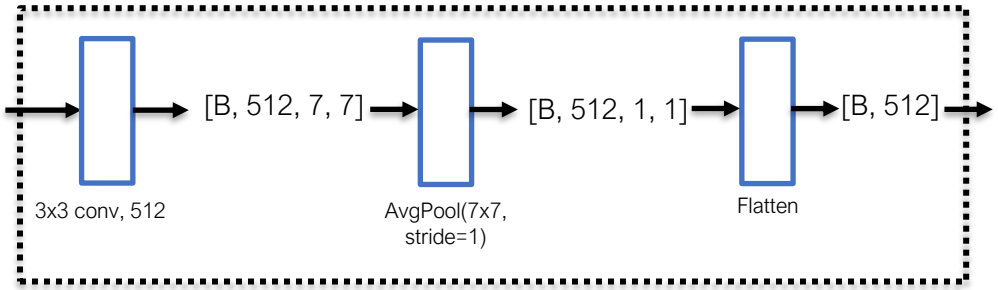
High level: object detectors. Intuition/hope: high activations ideally mean "there is a semantic object (cat, dog) in this image!"

Interpretation: by stacking many conv layers, CNNs learn **hierarchical features**.
Lower level layers: Low-level image features (edges).
Middle layers: Mid-level image features (shapes)
Final layers: "semantic" features (eg part detectors, face detectors)

ResNet: image embedding



Popular choice: use the output of `AvgPool` as the **image embedding representation**.
=> Each image embedding has size=512.



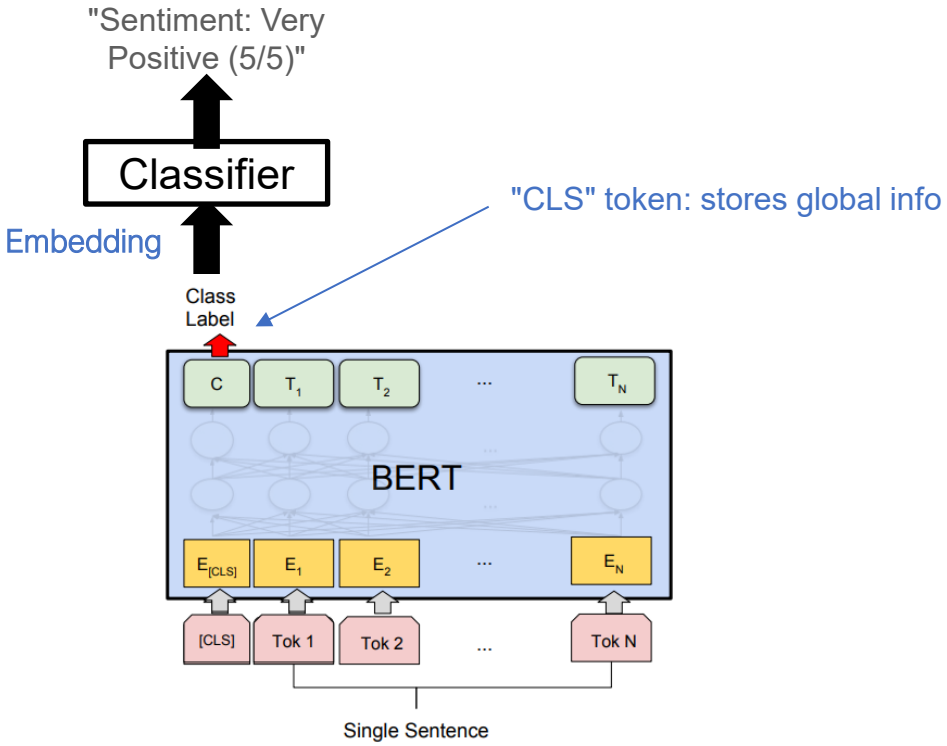
Intuition: final AvgPool summarizes the information in each spatial feature map prior to the linear classifier.
Idea: perhaps each of the 512 channels contains the "visual concepts" present in the image?

Resnet paper, "Deep Residual Learning for Image Recognition".

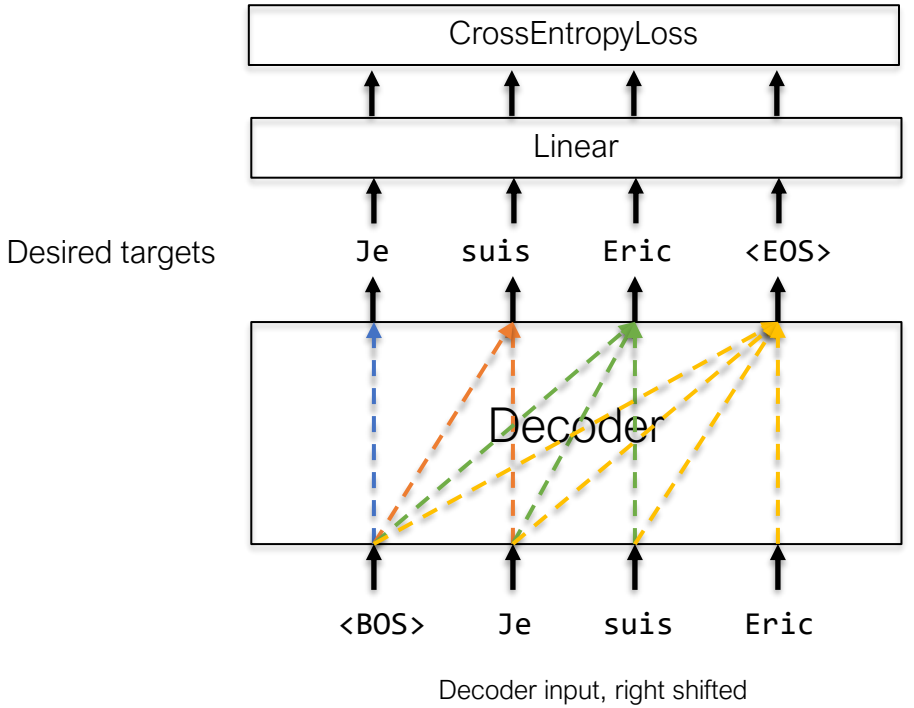
(optional) if you're curious, here is the resnet model code in pytorch:
https://github.com/huggingface/transformers/blob/main/src/transformers/models/resnet/modeling_resnet.py
Huggingface page: https://huggingface.co/microsoft/resnet-50
Config for resnet50: https://huggingface.co/microsoft/resnet-50/blob/main/config.json

Transformer encoder embedding

For transformer encoder models: use the [CLS] token embedding, or some aggregation (pooling) of all output tokens. (Similar idea for decoder output tokens)



(b) Single Sentence Classification Tasks: SST-2, CoLA



Per-token embeddings

Colored lines show allowed cross-token information flow, enforced by causal mask in masked self attention block (to avoid information leakage)

Ex: when predicting "suis", decoder can only use information from: [<BOS>, "Je"]

Embedding visualization

- Heuristic: to check if your model is indeed learning a “healthy” embedding metric space, try clustering the embeddings + visualize them!
- How to project a high-dimensional embedding (eg 256-dim) to 2D? Lots of ways to do this
- Approach 1: PCA dimensionality reduction
- Approach 2: t-SNE (pictured applied to image embeddings) [link]

https://arxiv.org/abs/1702.04680

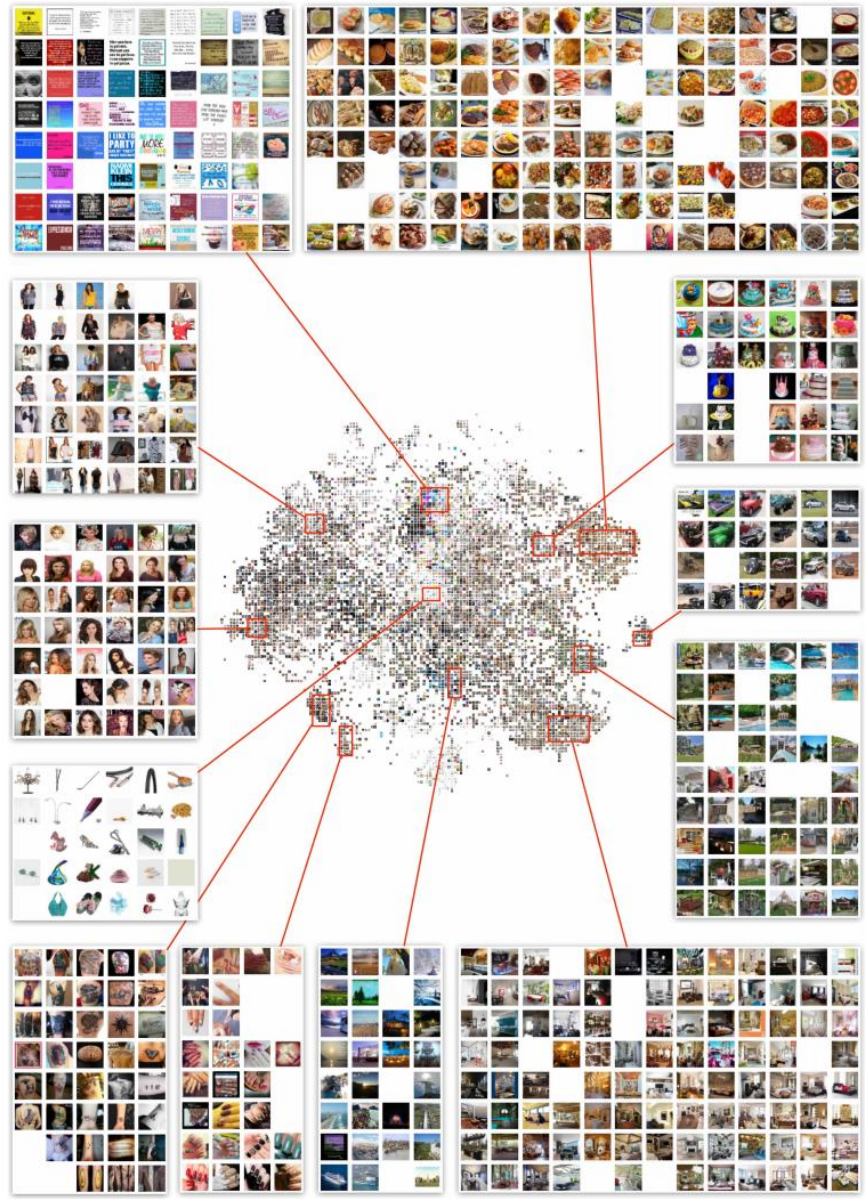
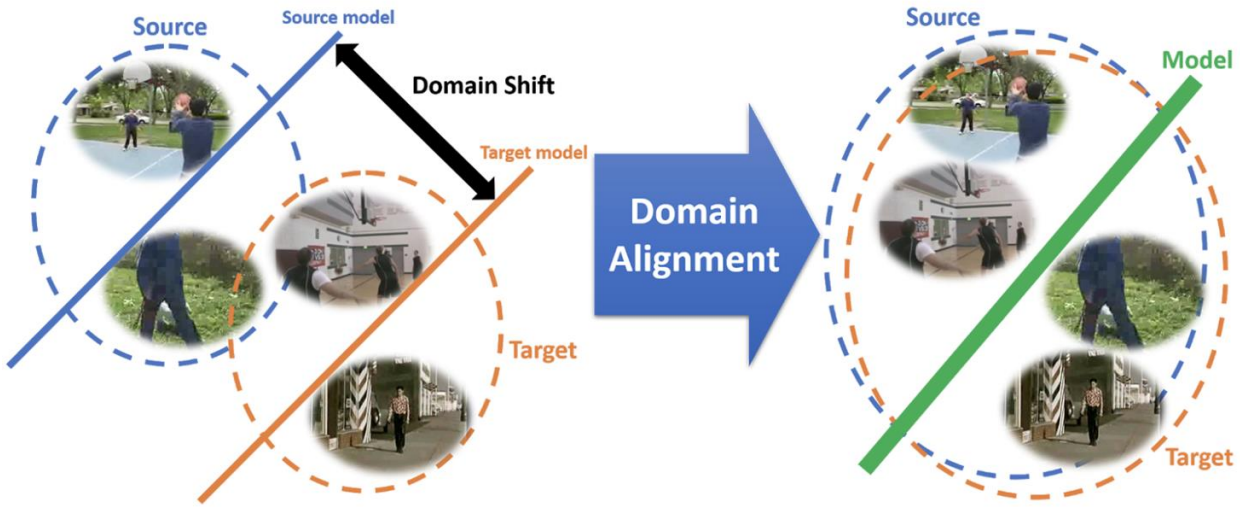


Figure 6: Visualization of binarized embeddings of Pinterest images extracted from fine-tuned VGG16 FC6 layer.

Domain shift

- In practice, we take a pre-trained model (eg image classifier trained on ImageNet-1k), and do another training run ("fine-tuning") on our internal dataset (eg Pinterest/Instagram images).
- Reason: target images (eg Pinterest/Instagram) often have different characteristics than what the pre-trained model has seen (eg ImageNet-1k)
  - Finetuned embeddings usually perform much better than pretrained embeddings!
- In ML jargon, called **"domain shift"**

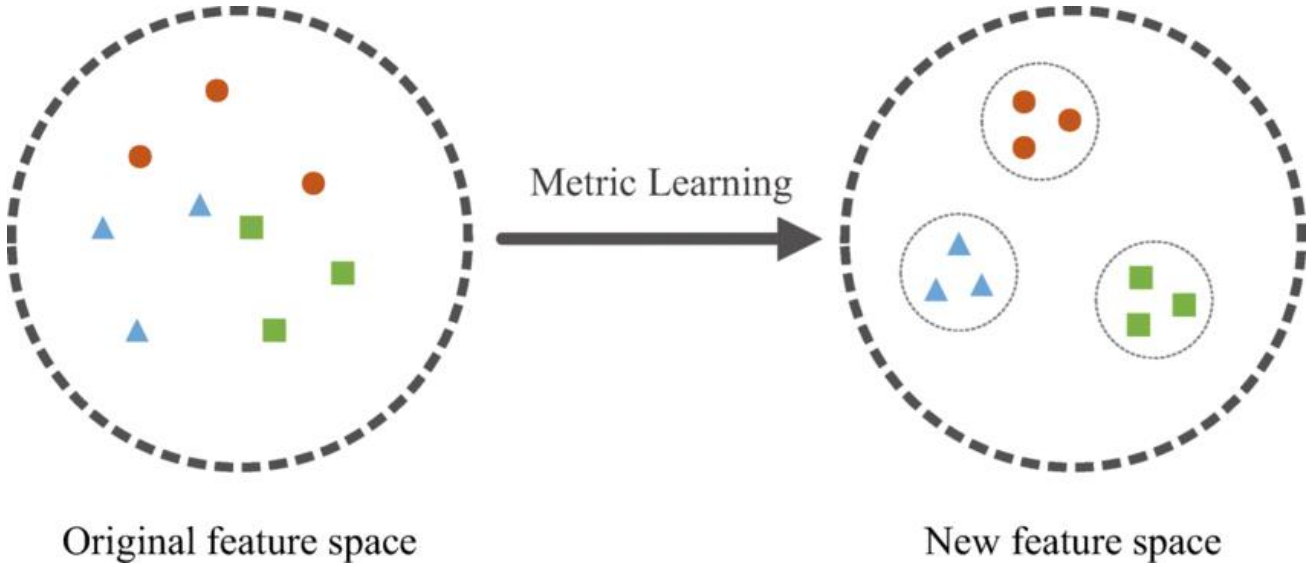


Example:
Source domain: ImageNet-1k
Target domain: Pinterest/Instagram images

https://mlatgt.blog/2019/09/10/overcoming-large-scale-annotation-requirements-for-understanding-videos-in-the-wild/

Pretrained models and metrics

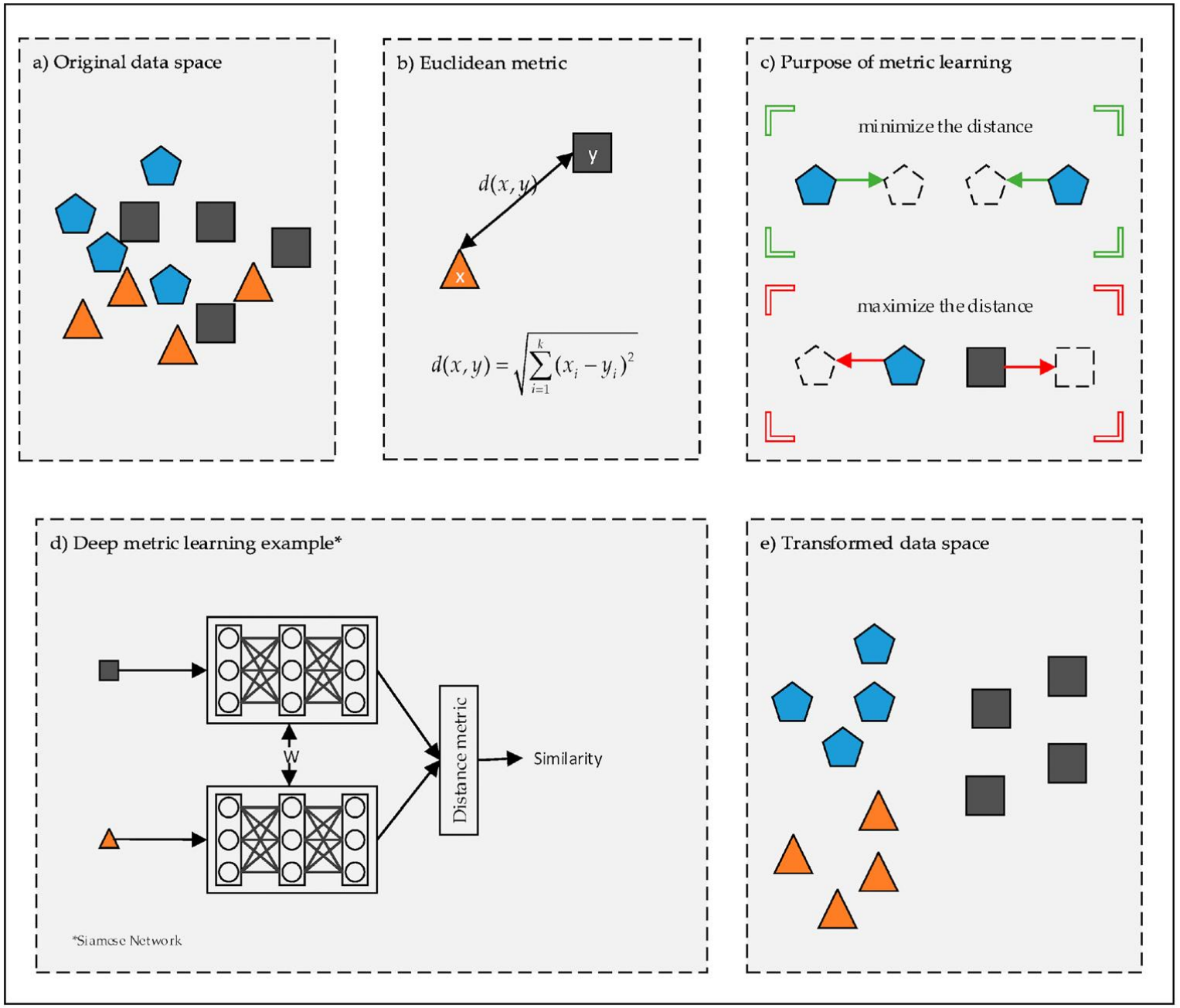
- Funny enough: in practice, using image embeddings from pretrained image classification models works quite well even though there's no "metric learning" going on
  - Training loss is image classification, not anything "metric-y/distance-y"
- Idea: can we directly optimize for learning a good embedding that "behaves well" for some metric (eg cosine similarity)?



https://www.researchgate.net/figure/Working-mechanism-of-metric-learning-The-metric-learning-method-aims-to-find-a_fig3_355201453

Metric learning

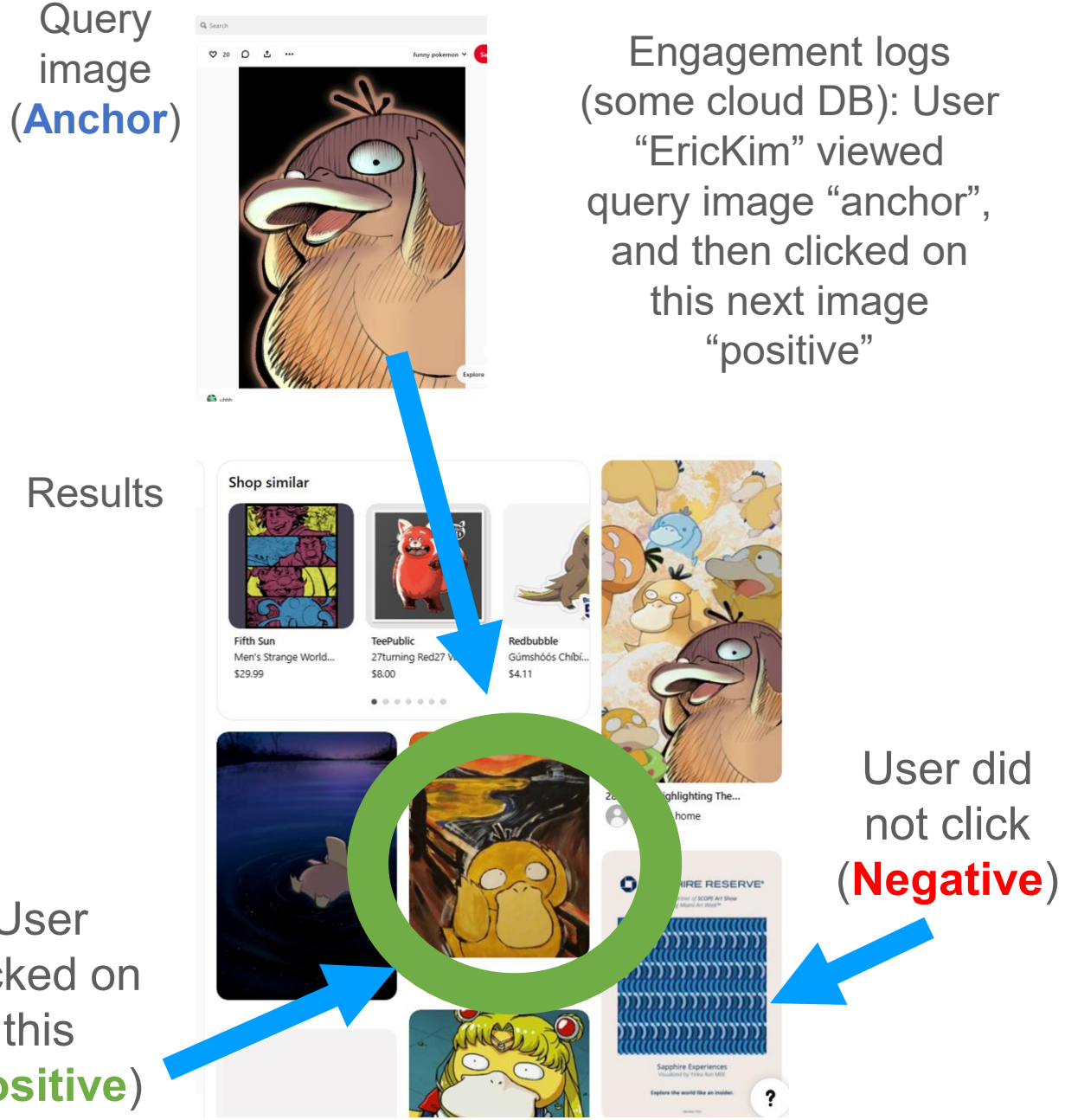
- Metric learning: a well-studied problem in ML to learn a good feature representation where distance metrics “work well”
- “Deep” metric learning: train a DNN that learns a good embedding representation that works well with your desired distance metric (L2, cosine dist, etc)



https://www.mdpi.com/2073-8994/11/9/1066

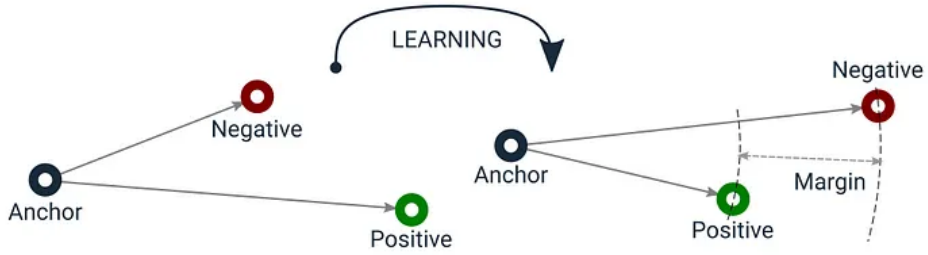
Dataset: triplets

- Suppose we have a labeled dataset of (anchor, positive, negative)
- Example: user engagement logs.
  - **Anchor**: Query image/post/video that a User viewed
  - **Positive**: Next image/post/video the User clicked on next
  - **Negative**: An image/post/video that the User didn't click on
    - Or: random negatives works well in practice too



Metric learning: triplet loss

- Idea: design a training loss that pulls (anchor, positive) embeddings close to each other, and (anchor, negative) embeddings far away
- New loss! “Triplet loss”
- Pytorch: `torch.nn.TripletMarginWithDistanceLoss`



The loss function for each sample in the mini-batch is:

$$L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}$$

where

$$d(x_i, y_i) = \|\mathbf{x}_i - \mathbf{y}_i\|_p$$

This equation uses the Lp norm (eg L1, L2, etc) as the distance metric, but in principle you can use any metric like: cosine similarity, dot product, etc

Takeaways

- Modern recommendation systems (as of 2026) heavily utilize machine learning
  - Representation learning, "learning to rank"
- The single most valuable edge that companies have: **user engagement data**
  - Companies (like Google/Meta/etc) constantly log all actions you perform on the app/website, and train models on this data.
  - Deep learning is notoriously data-hungry
- In addition to ML modeling, lots of interesting and engineering challenges to make things work at scale
  - Ex: distributed approximate nearest neighbor, ML feature stores, ML serving infrastructure. "big data". Fun stuff!

What's next?

- (As of 2026) One neat idea: User sequence modeling
- Idea: train sequence models (like transformer model) on sequences of user engagement history. Ex: a user's last 6 months of activity
  - Challenge: very high computation costs for both training and serving! But can be very effective.

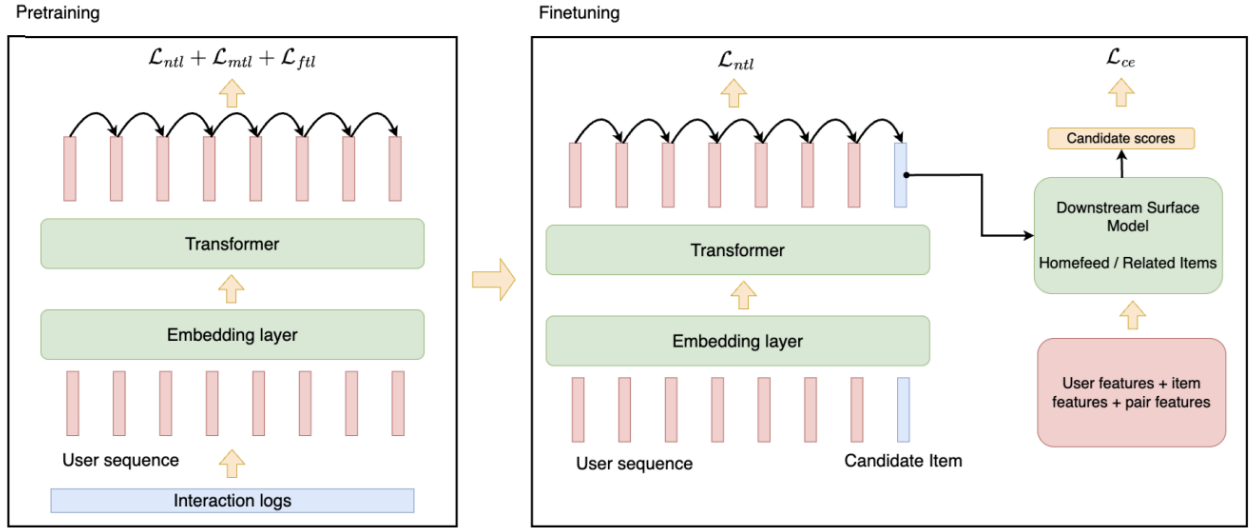


Figure 1: Training and fine-tuning of PinFM

https://arxiv.org/abs/2507.12704