



CS 839 Next Generation Data Systems

Lecture 4: ANN & Vector Indexes

Tianyu Li
litianyu@cs.wisc.edu

Slide inspired by materials from
Diyi Yang, Tatsunori Hashimoto,
Mike Cafarella, Moshe Gabel,
Andy Pavlo

Administrivia

- Team Formation due today
 - Submit on canvas
 - Go to People/Groups/+Group to form groups
- Start thinking about what you want to work on as a project!
 - project proposals due 10/8

Recap: Lakehouses

- A unifying architecture for enterprise data management and processing in the cloud at scale
- But what can we ask about its contents?
 - SQL, Spark etc. great for processing structured, cleaned data
 - What about the meaning behind data?

Example

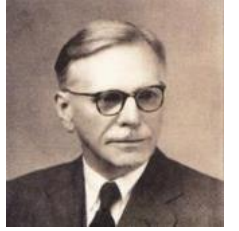
- An analyst may want to find customers who are having trouble signing in the ticketing system last week
- Customer tickets are recorded as free text + metadata
 - E.g. “I keep getting sent back to the login screen”, “I cannot access the password reset link”, “It won’t let me into my account”
- How to represent “meaning” in bytes?
 - ...and, how do we efficiently query meaning?

How to Represent Meaning?

- Idea: use a high-dimensional representation (i.e. an embedding)
 - E.g., [0.002, 0.244, 0.546, ..., 0.345]
- You can think of each dimension as having some meaning:
 - E.g.

[Word	Animal-related	Pat-able	Big ...]
[Cat	0.9	1.0	0.0 ...]
[Car	0.1	0.0	0.5 ...]
[Tiger	0.95	0.1	0.4 ...]
- In reality, we will try to learn the representation from data
 - Each dimension may not have a human readable meaning

How do we learn embeddings?



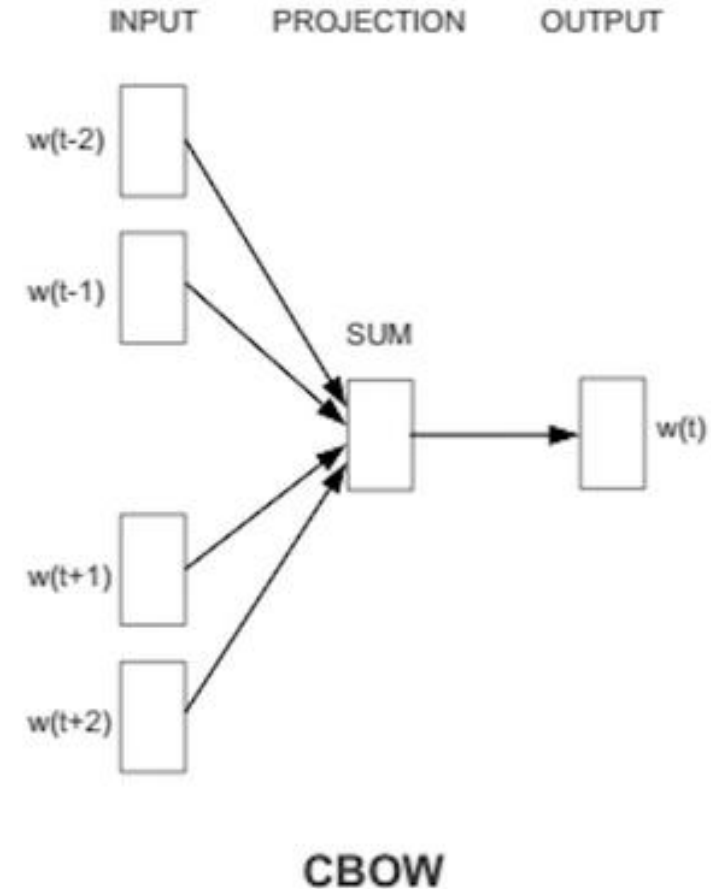
- As with any machine learning problem, must find right training paradigm that efficiently utilizes signal from training data at scale
 - Training data: all the text you can find
- Key Insight: A word's meaning is given by the words that frequently appear close-by (J.R. Firth 1957)
 - E.g., The cat sleeps on the sofa ← common
The dog sleeps on the sofa ← common
The tiger sleeps on the sofa ← not so common

Word2vec

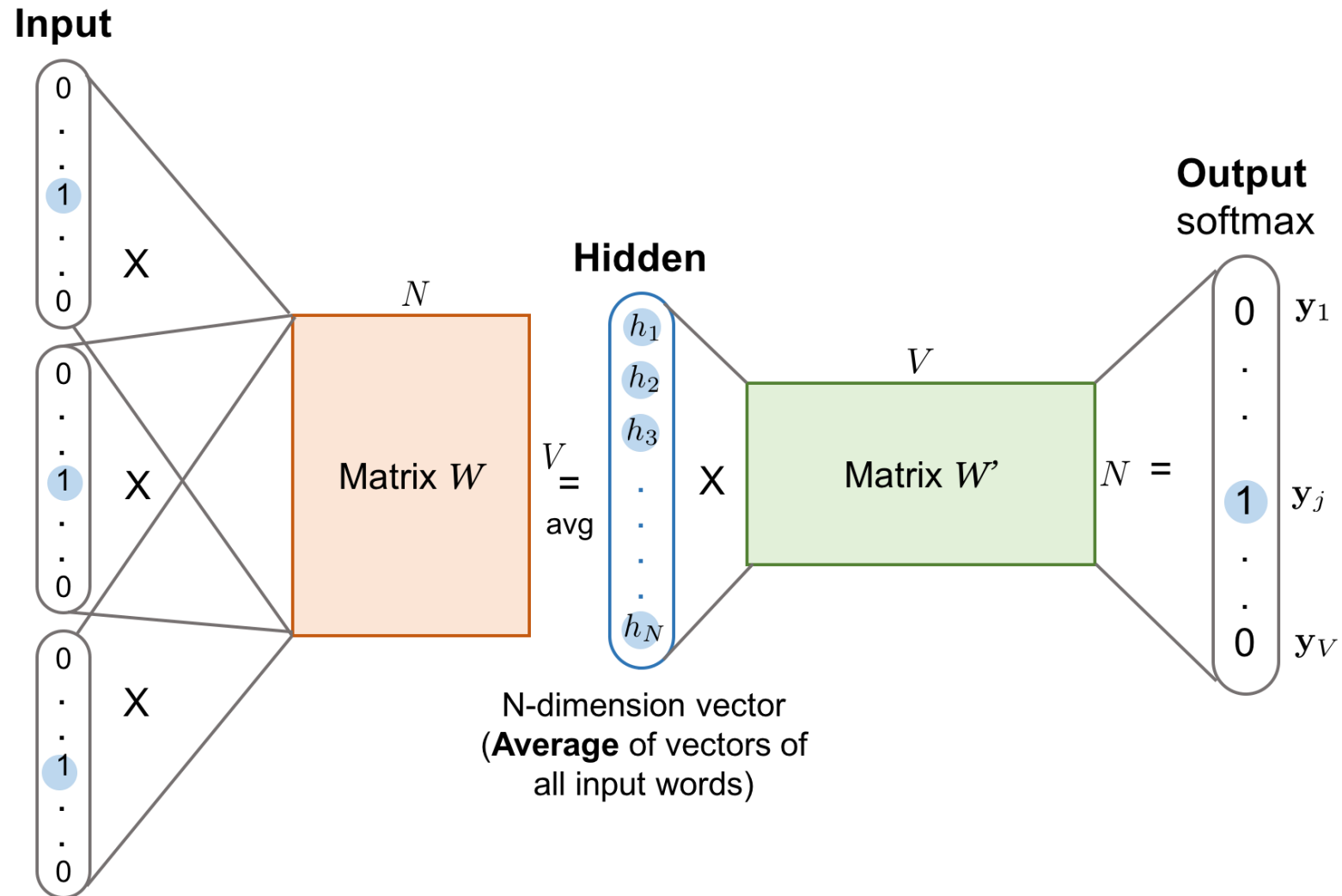
- Seminal work by Mikolov et. al. in 2013 from Google
- High level idea:
 - Use a simple network to learn $P(\text{word} \mid \text{context})$
 - The final embeddings distills meaning given enough data, as our previous insight predicts
 - E.g. $E(\text{"King"}) - E(\text{"Man"}) + E(\text{"Woman"}) \cong E(\text{"Queen"})$

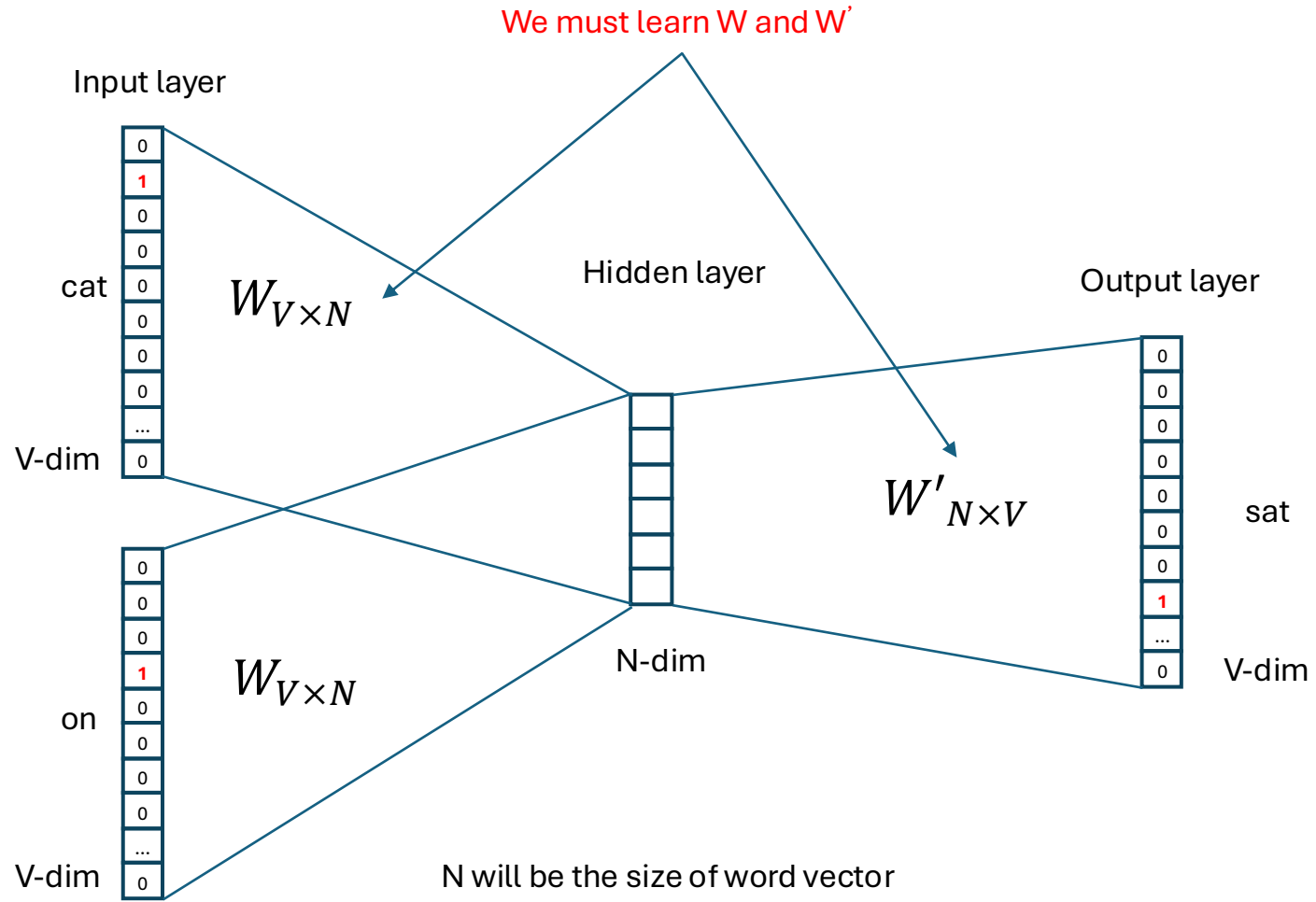
Continuous Bag of Words (CBOW)

- Takes vector embeddings of n words before target and n words after and adds them (as vectors).
- It is a “fill-in-the-blank” task
- It works well even though we remove word order. The vector sum is meaningful enough to deduce missing word.



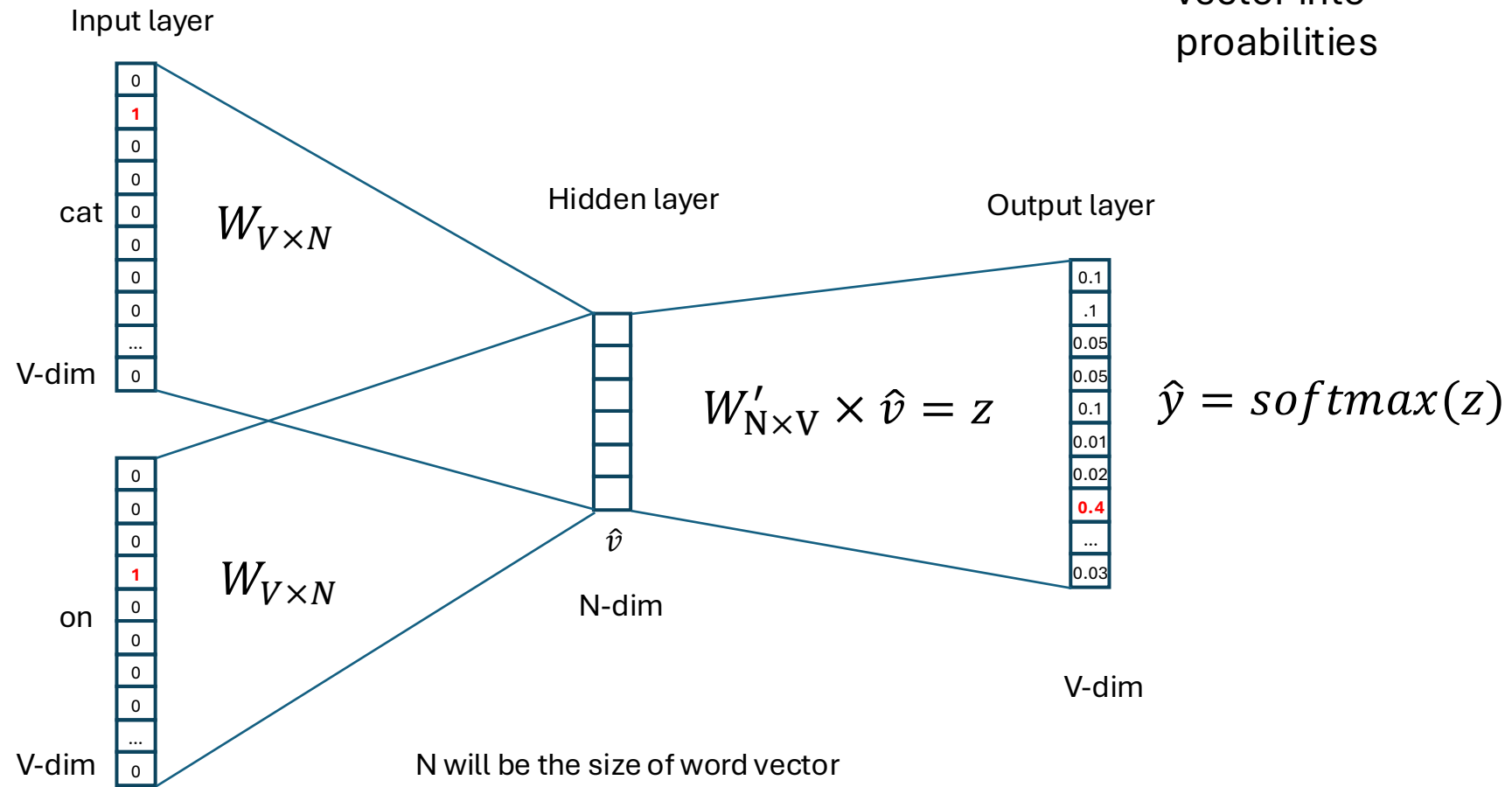
CBOW

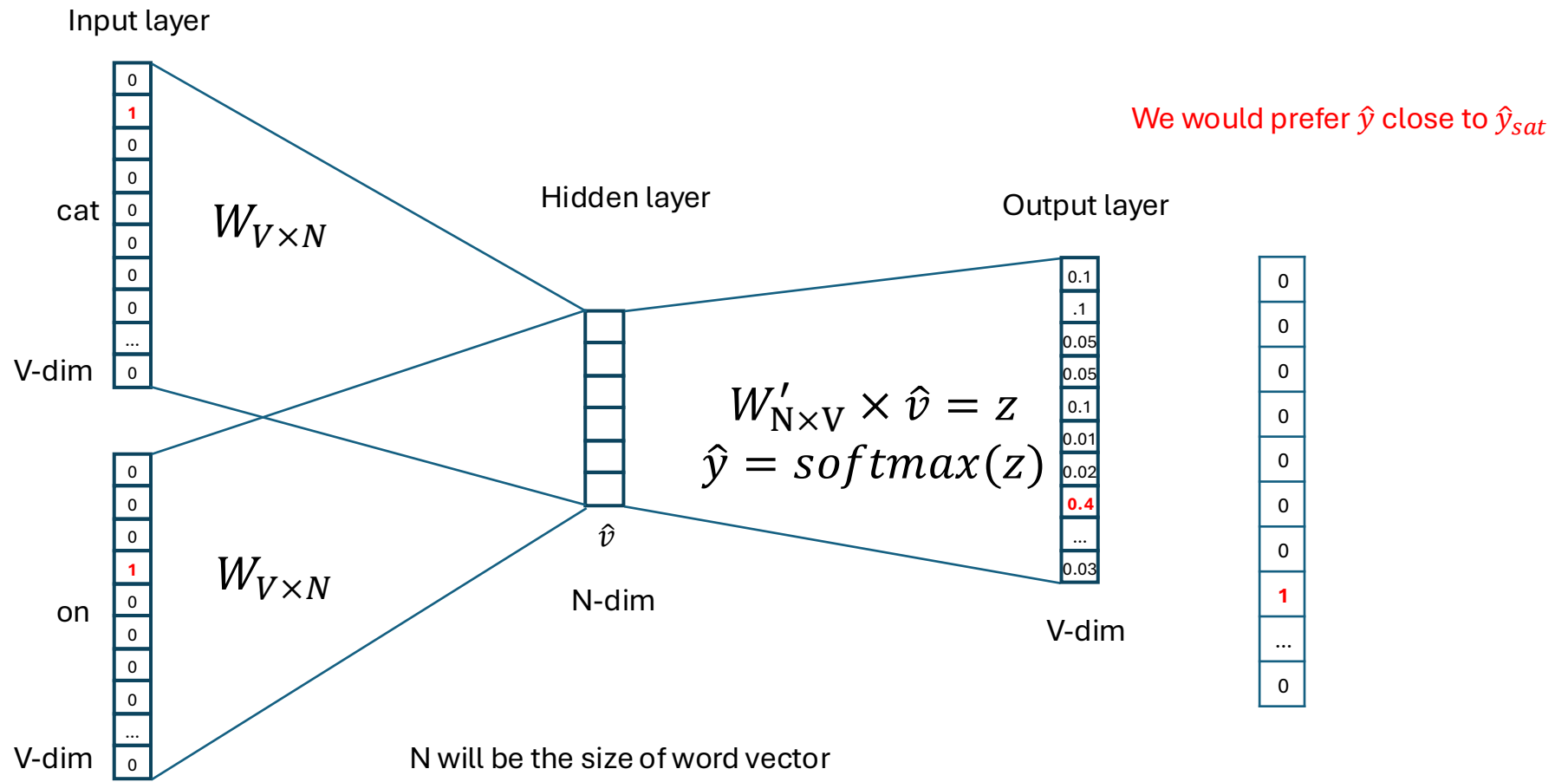




$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Softmax turns the vector into probabilities





Where to go from here?

- Word embeddings are fun, but quite limited
 - A word's meaning will depend on its context, so a single representation per word cannot work for many cases (e.g., apple)
- Modern embeddings: models like BERT/GPT learn to compute representations from context
 - Transformers, etc. augment simple embedding table
 - Details beyond the scope of this class

Mental Model for Modern Embeddings

- Similar to word2vec: there's a learned embedding table over some pre-determined token vocabulary (computationally derived from the corpus)
- However, instead of simple average, each token interacts with other tokens in the passage using learned weights (i.e., self-attention)
- Details defer between models, but every token becomes a “contextualized” embedding after passing through the model

How to represent passages?

- After a forward pass, we have one contextualized embedding per token in an input passage
- This is awesome, but still does not allow us to directly “compare” passages.
 - E.g. “I can’t login” with “I am having trouble signing on to the system”

Passage Embeddings

- Goal: precompute a representation for each passage, compare them with a simple similarity score
 - Many ways to do so mathematically, e.g., cosine similarity. Beyond the scope of this class
- Simple Idea: average them?
 - Called Mean pooling
 - Other pooling available, again, beyond the scope of this class

Sentence-BERT

- Problem: BERT/GPT is trained for masked/next token prediction, not to make similarities meaningful
 - Out-of-the-box, this will get you bad retrieval results
- Idea: ***post-train*** BERT specifically for retrieval
 - Take advantage of language understanding already learned (i.e., pre-trained)
 - Learn from actual retrieval task data to fine tune the representation for average to become meaningful

Sentence-BERT: Training

- Start from labeled dataset of sentence relevance (e.g., humans score pairs of sentences for semantic similarity on a scale of 1 to 10)
- Predicted similarity: cosine similarity between two pooled vectors
- Backpropagate away on underlying BERT
 - Again, details beyond the scope of this class

The Shape of a Solution

- Sentence-BERT makes it practical to represent text as dense vectors and use some notion of distance between them to predict semantic similarity
 - People also figured out how to represent other things as embeddings (e.g., images, audio)
- This is great: we can compute embeddings offline and search for semantic relevance by comparing a query embedding with these pre-computed embeddings

Retrieval: Metrics

- Retrieval: Given a (NL) query and a (text) corpus, return the items relevant to the query
 - Results are usually ranked
- Two quality metrics:
 - Recall: Fraction of relevant items returned
 - Precision: Fraction of returned items relevant
- Other metrics also often used to highlight different aspects of quality (e.g., F1, nDCG, MRR)

Retrieval: Interface

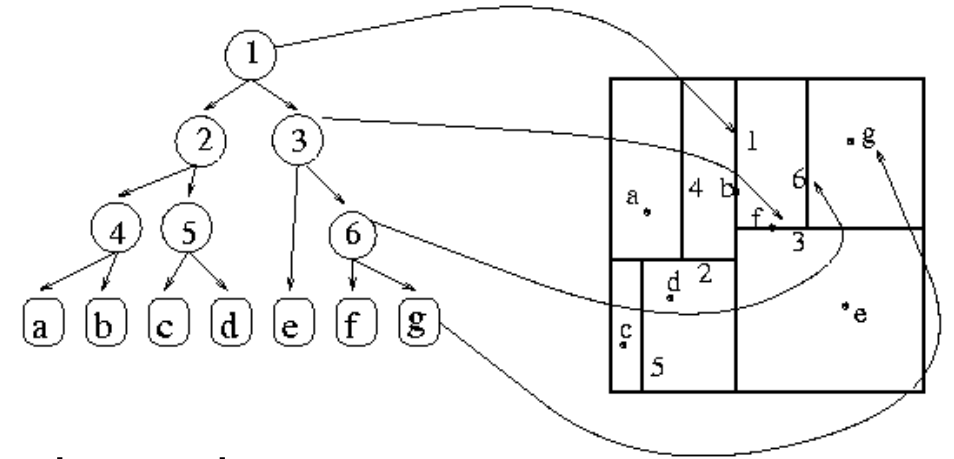
- Most standard retrieval operation: k-Nearest Neighbor Search
 - Represent query and corpus as a vectors
 - Find and return the k vectors that are closest to query according to desired distance function (or highest similarity)
- Other interfaces exist, but less common
 - E.g., return all vectors closer than x

Starting Point

- Simple solution to kNN: scan all the vectors!
 - Compute distance for every pair and take top k
 - $O(ND)$ where N is corpus size, D is dimension size
- Not a bad solution for small problem sizes
 - In addition to being dead simple, also highly parallelizable
- Inevitably though, scanning is too slow at scale
 - Want to rely on pre-computation to save compute at query time (i.e., indexing)

Existing Work: Spatial Indexes

- Interest in indexing multi-dimensional data (e.g., Geospatial) long before modern vector search
 - Often serves bounding box queries, etc.
- Key idea: Branch & Bound
 - When serving kNN queries, compute lower-bound of distance in a box and skip it if it cannot improve the current best



Problem

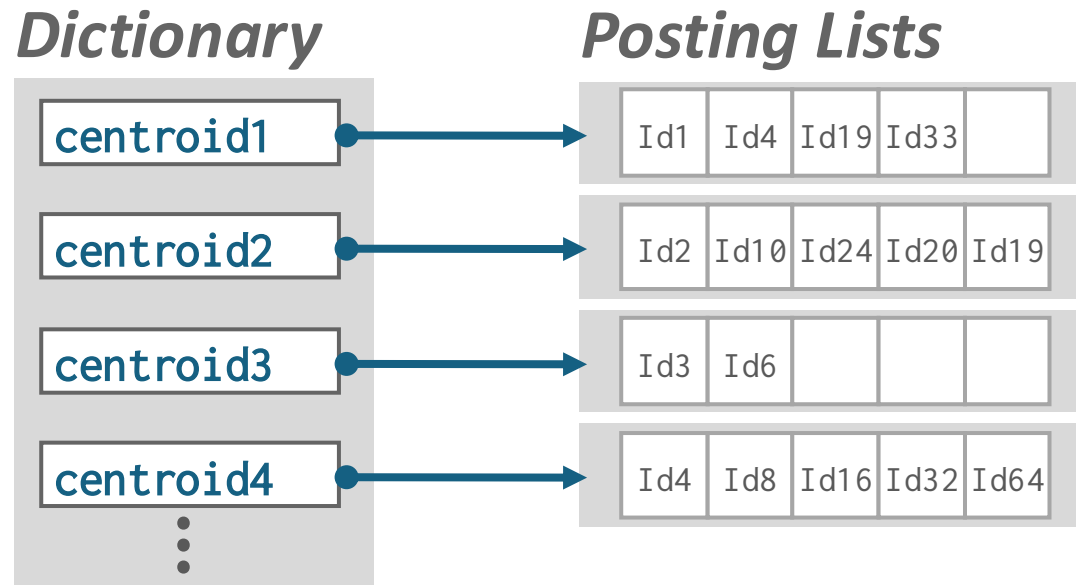
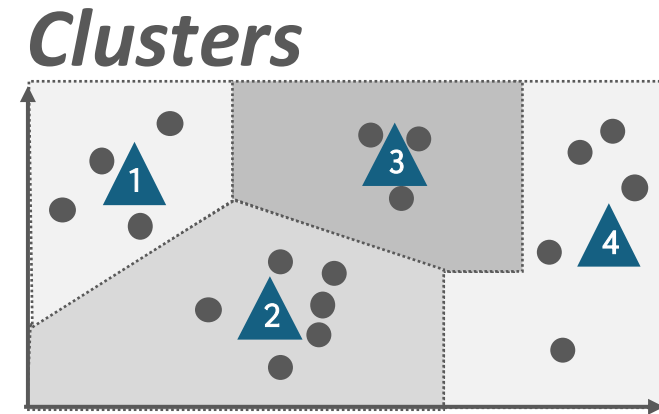
- K-D tree, etc. often cannot prune effectively for high dimensional space
 - Distance becomes less effective at pruning at high dimension
 - Empirically, often cannot perform much better than $O(ND)$
- Intuition: for each partition skipped, exact kNN must prove that skipped partitions cannot improve the result
 - What if simply skip unlike partitions and accept that we may miss true neighbors?

Approximate Nearest Neighbors

- Only require that the algorithm returns some near-neighbors but potentially missing a true neighbor
 - Early variants with Locality Sensitive Hashing (LSH) give guarantees on how near with high probability
 - Modern implementations often work better in practice but lack theoretical guarantees
- Two common variants:
 - Partition-based
 - Graph-based

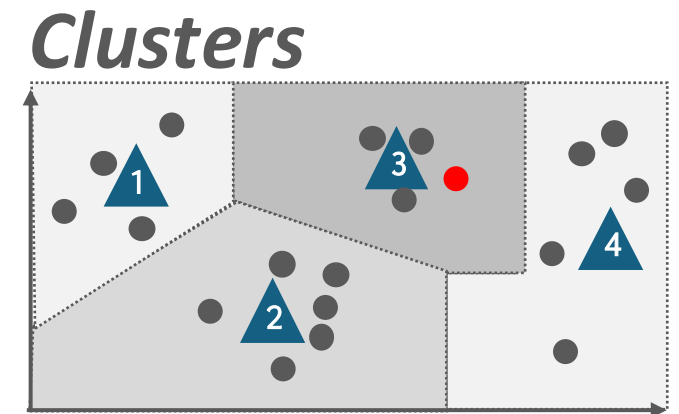
Inverted File Index (IVF)

- Simple Idea: Partition vectors into smaller groups using a clustering algorithm and then build an inverted index that maps cluster centroids to records.



Inverted File Index (IVF)

- To search: first, scan all centroids to find p closest centroids, then scan their postings and return the best k candidates
 - p is tunable
- What about points close to the edge?
 - Almost always $p > 1$
 - Points close to edges are often cross-listed



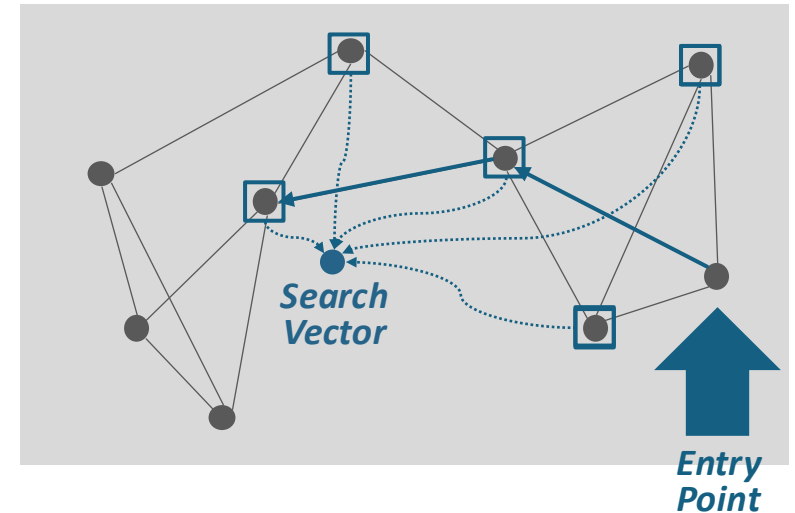
IVF over Time

- To add points, simply find closest centroid and add to posting
- Problem: clusters may no longer match data after a while
 - Must periodically rebuild clusters or otherwise rebalance them incrementally

Graph Indexes

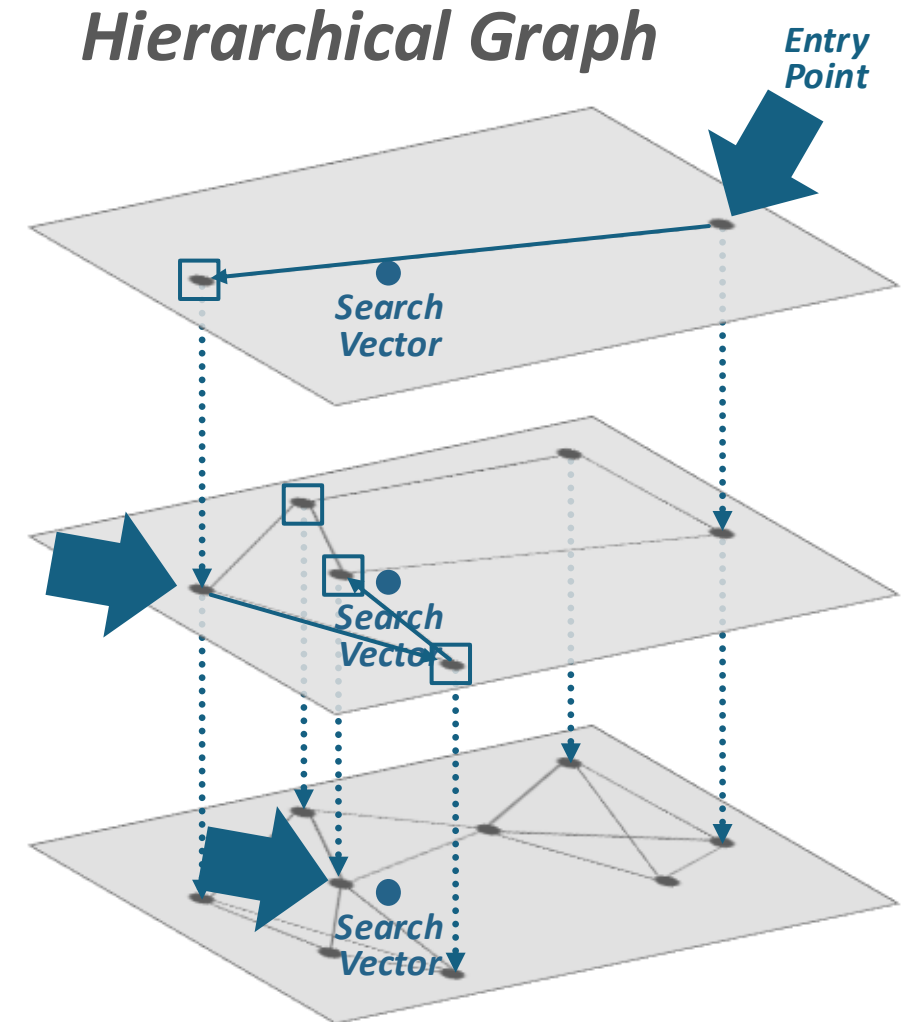
- Idea: Build a graph where each node represents a vector and it has edges to some limit of nearest neighbors
- To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



Hierarchical Navigable Small Worlds (HNSW)

- Idea: higher level graphs are sparser and allow for fast traversal
 - Search greedily at each layer and move down afterwards
- To build/update index:
 - Randomly determine max levels for new entry
 - Add edges to some configurable number of neighbors (with some heuristics favoring diversity)



Quantization

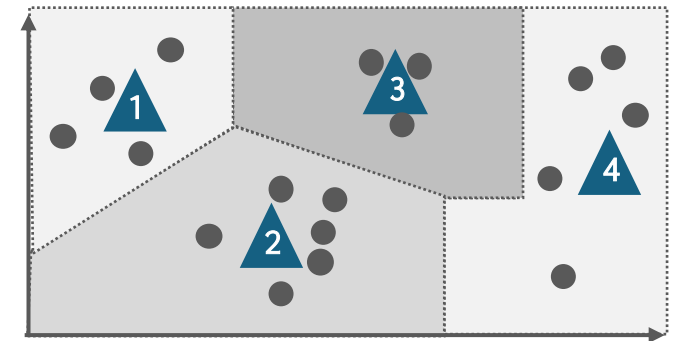
- So far, we have assumed that we store entire vectors
- New form of optimization unfamiliar in prior systems: quantization
 - Essentially, lossy compression of each vector
- Main approaches:
 - Scalar quantization
 - Vector quantization
 - Product quantization

Scalar Quantization

- For each dimension, reduce independently to n bits
 - i.e. 32 bit float to 8 bits
- Often used for models in general
- If done right, can often reduce memory and speed up operations at very minor loss in recall
 - Aggressive compression may incur greater recall loss

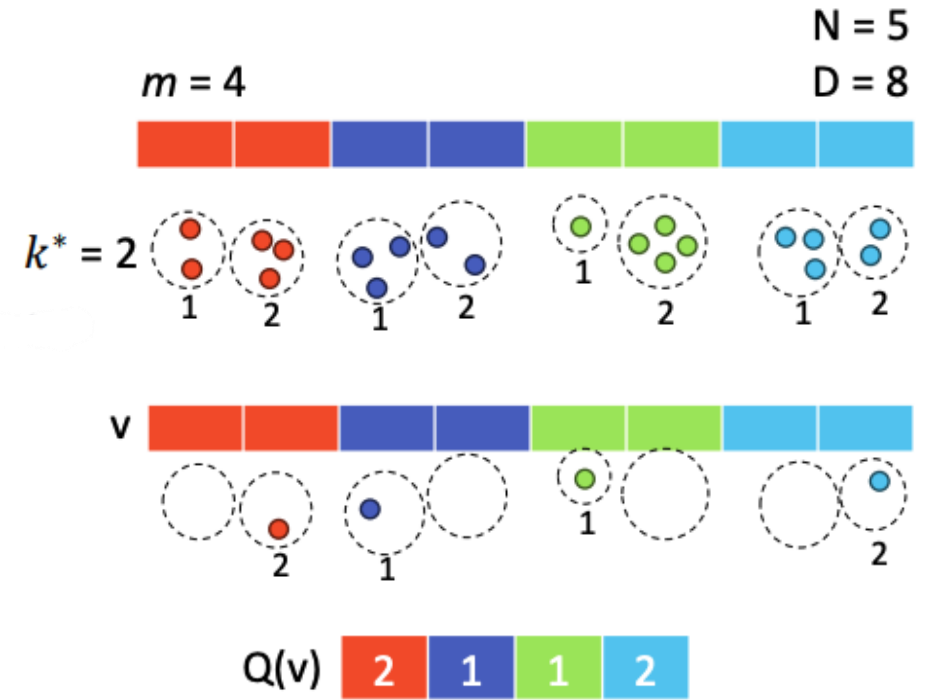
Vector Quantization

- Idea: cluster vectors and obtain centroids per cluster, and simply use the centroid to represent the entire cluster
 - Analogy: dictionary compression but with lossy assignment
 - Bonus: can compute query vector distance to centroids once and reuse as a lookup table during search
- Problem: Number of clusters needed for good resolution grows exponentially with dimension



Product Quantization

- Idea: split space to m chunks and run vector quantization for each subspace separately and concatenate the result
- Essentially – instead of storing size n^m centroids, store mn sub-centroids use cartesian product to reconstruct
- Catch: PQ cannot jointly explore correlation across subspaces and may increase quantization error. However, empirically this is often ok



Discussion

- Which one do you think is the primary bottleneck to vector DB accuracy, embedding models or the search algorithm/system? How would you distinguish the two?
- Are embeddings just another column with an index, or do they require a different database architecture?
- What kind of hardware/execution environment favor IVF vs. HNSW? Can you think of ways to mitigate?
- What are some novel system challenges for vector indexes under changing workloads (e.g., concurrent read-write, bulk insert/delete, skew)?

Next Week

- We will cover how to turn algorithms and building blocks from today into full-on systems
- Reminder to find teammates and think about what to work on!