Q1. Google's search engine started off being entirely keyword-based. Over the years, 'semantic' features have been added on a regular basis. What are three examples of such ?semantics-based features

Google has added several semantic-based features beyond simple keyword matching. First, **RankBrain** interprets queries by mapping them into **entities**, allowing Google to understand user intent even when the exact keywords do not appear in the documents (e.g., identifying that "Apple founder" refers to **Steve Jobs**). Second, Google's **Knowledge Graph and Named Entity Recognition (NER)** extract structured entities and their relationships, enabling the engine to reason about meaning rather than just match strings—for example, distinguishing semantic structures like "red stoplight" vs. "stoplight red". Third, Google performs **semantic query expansion**, such as expanding *[child bicycle]* to include "child," "children," "children's," and "bicycling," ensuring results capture conceptual similarity rather than exact word forms

Q3. Information can often be organized hierarchically. What are three examples we studied? Explain each in a few sentences.

Knowledge Graph A taxonomy arranges concepts in a parent–child hierarchy, where broad categories sit above more specific ones. Because of inheritance, child nodes automatically carry the properties of their parents, allowing search systems to reason semantically about categories.2. Hierarchical Clustering Hierarchical clustering builds a tree of document groups, either by merging small clusters upward or splitting large ones downward. This structure lets IR systems explore data at multiple granularities and understand how documents group together.3. Website / URL Structure

Websites often follow hierarchical paths (e.g., /sports/basketball/NBA), reflecting section/subsection relationships. This organization helps crawling, PageRank flow, and allows search engines to infer which pages are central versus subordinate.

Q6. What are three emerging standards, in the world of agentic AI? Explain each in a few lines.

MCP (Model Context Protocol) is a standard for how agents communicate with external tools and back-end systems. It provides a consistent, structured way for agents to call tools and receive results.A2A (Agent-to-Agent communication) defines how multiple agents exchange messages and coordinate tasks. It enables agents to collaborate without custom communication logic.ADK (Agent Development Kit) is Google's modular framework for building and deploying AI agents. It standardizes core components—like memory, tools, and execution—so agents can be developed in a uniform, extensible way.

Use of a vector DB is one way to do 'RAG'. What is stored? How is it indexed (for quick retrieval)? How does retrieval work (on what basis)? Explain each, using (three) simple diagrams.

A vector DB stores **embeddings**—high-dimensional numeric vectors produced by an embedding model—from documents, chunks, tables, or other knowledge sources. Each stored item typically includes:documents,metadata,embedding vectors.Vector DBs index embeddings using ANN (Approximate Nearest Neighbor) algorithms such as HNSW graphs, IVF (Inverted File Indexing), or LSH. These structures cluster vectors in geometric space so that similarity search is much faster than scanning all items.When a query comes in, the system embeds the query into a **vector**, then performs a **nearest-neighbor search** to find stored vectors with the highest **cosine similarity** (or Euclidean similarity) to the query vector. The top-k most similar embeddings are returned, along with their associated text chunks.

Before an LLM (or more generally, an FM, ie foundation model) is put to use (eg by making it be available on Hugging Face, or via a server (eg. at OpenAI or Google or Perplexity etc), it needs to undergo three preparatory stages. Explain the stages, using a few lines for each

Pretraining.The model is first trained on massive unlabeled text corpora using self-supervised learning to acquire general language understanding and world knowledge. This produces a broad "base model" with no task-specific behavior yet.Instruction-tuning / Fine-tuning Next, the model is trained on curated human-written examples so it learns to follow instructions, format answers, and perform useful tasks. This step shapes the model's behavior beyond raw prediction.Safety alignment + Deployment preparation .Finally, the model undergoes safety tuning (e.g., RLHF, policy rules), evaluation, and optimization to ensure safe, reliable responses. Only after these checks is it packaged, hosted, and released for real-world use.

NER to 'graph RAG' - what is the connection (ie pipeline) that involves these two items? In other words, how would we start from plain text, and end up with a tool that helps us obtain high quality (non-hallucinatory) responses from an LLM? Explain the pipeline (steps).

We start with plain text and run **NER** to extract entities such as people, places, and organizations. Next, we identify relationships between these entities and convert them into **RDF-style triples**, which are then inserted into a **Knowledge Graph**. This graph is indexed so that queries return the most relevant entities and their connected facts, rather than just similar text chunks. In **Graph RAG**, these retrieved subgraphs are passed to the LLM as grounded factual context, ensuring that the model bases its answers on verified relationships and therefore produces far fewer hallucinations.

user-user-user... item-item-item... content-content-content... Explain the above, in terms of what we studied [as it relates to info' retrieval]

These three patterns represent **similarity chains** that arise in different retrieval methods. The **user–user–user** chain reflects *user-based collaborative filtering*, where we find users similar to a target user and use their preferences to make recommendations. The **item–item–item** chain corresponds to *item-based collaborative filtering*, where similarity between items (e.g., movies, products) is used to recommend items related to those a user already likes. The **content–content–content** chain represents *content-based retrieval*, where items are linked through their intrinsic features (text, embeddings, metadata) rather than user behavior, and recommendations follow the chain of most similar content.

When we upload pics/clips/songs... to social media, what specific mechanism to we use, to help others find our content when they search? Why do we need this specific mechanism?

When we upload pictures, videos, or songs to social media, we add **tags/metadata** such as captions, hashtags, titles, or category labels. These tags act as **indexing signals**, allowing the platform to place our content into its **inverted index** or embedding-based retrieval system so it can be found through search. We need this mechanism because raw multimedia files (images, audio, video) **do not contain searchable text**, so the platform cannot know what the content is about unless we provide descriptive labels. Tags therefore supply the semantic information required for search engines to retrieve our content accurately and match it to user queries.

What algorithms causes/leads to/results in/is implicated in... 'filter bubbles'? How does it lead to this?

The main algorithms that lead to **filter bubbles** are **collaborative filtering recommenders**—both *user-user* and *item-item* methods. These algorithms reinforce a user's past behavior by recommending items similar to what they (or similar users) have already consumed. Over time, the system repeatedly surfaces content from the same taste cluster, narrowing exposure and reducing diversity. Because the model optimizes for relevance and engagement—not variety—it unintentionally creates a self-reinforcing loop where a user sees only the viewpoints, products, or media that fit their existing profile.

What TWO other items can a search engine serve us (eg. via 'snippets'), in addition to what gets served already? Name each item, and briefly state why it would be of use to us.

Rating Summaries.**Rich Snippets** can show extra structured data such as **reviews, ratings, recipes, events, and prices** to help users decide which link is most relevant before clicking. This is useful because it saves time—users can compare quality, popularity, or key attributes directly in the results page without opening the site.In addition to snippets, search engines can serve **expanded or related queries** that are semantically connected to the original query. This is useful because it helps users refine or broaden their search—especially when the initial query is unclear or incomplete—and leverages statistical data from past searches to guide users toward more relevant results.

As you know, genAI (generative AI) is socalled because it can generate content (text, images, video, audio, more). HOW will this adversely affect search in the (near, even) future? Explain carefully

GenAI makes it extremely easy to generate thousands of low-quality, near-duplicate pages, which overwhelm crawling, deduplication, and ranking systems. This flood of synthetic content also allows groups to create highly targeted, personalized pages that reinforce a user's existing interests or beliefs, making **filter bubbles** tighter and harder for search engines to break. As search repeatedly surfaces content similar to what users already click, these AI-generated pages deepen the narrowing of viewpoints and reduce exposure to diverse or corrective information.

ChatGPT (for example) is said to "hallucinate" sometimes (or a lot of times, depending on the type of questions) - an unfortunate term (because only minds can hallucinate!) used by companies who serve this kind of AI products. This means that the bot provides an incorrect answer (which we can verify using our own knowledge or experience or by doing a good old search!). WHAT mechanism (in the algorithm) causes this to happen? Please be specific.

Hallucination occurs because an LLM's core training objective is **next-token prediction**, meaning it is optimized only to generate the most statistically likely continuation of text rather than to verify factual correctness. The model must always produce an answer, even when it has no reliable information, so it fills gaps by generating plausible-sounding text based on patterns in its training data. Since the loss function rewards fluency—not truth—the model can output sentences that look coherent but are factually wrong. And because LLMs lack grounding in external reality, when they encounter prompts outside what they have seen during training, they interpolate or invent details, resulting in "hallucinations."

1+1=2 points: How do recommendation engines (REs) work? 1+1=2 points: What are two different uses for them when we search?

Point 1: Recommendation engines compute **similarity**—either between **users** (user–user collaborative filtering), **items** (item–item collaborative filtering), or **content** (content-based, using features such as embeddings or keywords).**Point 2:** They then recommend items that similar users liked, or items that are similar to what the current user has already interacted with, producing ranked suggestions tailored to the user.

1+1 = 2 points: What are two uses for them when we search?

Use 1: REs suggest **related queries** or expansions (e.g., "people also search for"), helping users refine or broaden their search based on similarity patterns among past users.

Use 2: REs recommend **related documents/pages**, surfacing content that is semantically or behaviorally connected to what the user just clicked, improving search discovery beyond simple keyword matching.

What is 'vector similarity search'?For what two different IR tasks are vector DBs useful? Name, and explain briefly why we couldn't do them without vector DBs.

Vector similarity search means representing documents or items as **vectors in embedding space** and retrieving those whose vectors are closest using measures like **cosine similarity**, as described in your notes' sections on vector space and similarity search.Vector databases are essential for **semantic retrieval**, because inverted indexes only match exact terms, while vector DBs let us find semantically similar items even when no words overlap. They are also required for **fast approximate nearest-neighbor (ANN) search**, since algorithms like FAISS or Annoy (from the notes) cluster vectors and enable rapid lookup without scanning millions of items.Without these vector DB structures, both tasks would be computationally infeasible because similarity would have to be computed exhaustively across all vectors.

What does 'OPL' stand for, in OPL stack? What is its main use?

OPL stands for **OpenAI + Pinecone + LangChain**, as described in your notes under "LLM + tasks + memory → OPL stack." Its main use is to build applications where an LLM can **store, retrieve, and use external knowledge**:**OpenAI** provides the LLM and embeddings,**Pinecone** stores vectors for fast semantic search, and**LangChain** orchestrates the workflow so the model can call the vector DB, tools, or other components.

How do 'RDF triples' make search better? Explain in a few lines.

RDF triples make search better by converting text into **structured facts** of the form <subject, predicate, object>, Because the search engine now knows **who is connected to what and how**, it can retrieve results based on **meaning and relationships**, not just matching keywords. This reduces ambiguity, improves entity understanding, and allows engines to answer factual questions directly by pulling the correct triples—even when the exact query wording does not appear in the documents

Name four algorithms we looked at, for IR tasks, that rely on iteration or recursion. For each, explain briefly, how the iteration or recursion helps (ie. what changes during each run).

PageRank repeatedly updates each page's rank based on the ranks of its in-linking pages, with the scores changing each iteration until they converge. **K-means clustering** iterates between reassigning documents to the nearest centroid and recomputing those centroids, changing both cluster membership and center positions each round. **Hierarchical agglomerative clustering** recursively merges the closest clusters, with the cluster set and distance relationships changing after each merge. Finally, **Rocchio relevance feedback** iteratively adjusts the query vector by moving it toward relevant documents and away from non-relevant ones, causing the query representation to change each iteration.

Name, and very briefly discuss 4 MLbased algorithms we looked (towards the end of the course!), for IR tasks.

Four ML-based algorithms we covered include **LDA**, which discovers hidden topics in a document set and helps IR by grouping documents thematically; **ANN** methods such as Annoy, which learn structures for fast similarity search in high-dimensional embedding space; **FAISS**, which uses K-means clustering and vector quantization to speed large-scale nearest-neighbor retrieval; and **t-SNE**, which learns a low-dimensional representation of document vectors to visualize clusters and semantic relationships.

TikTok's recommendation engine uses a specific data structure, to optimize how it works. What is the name of the data structure? In your own words, how does it work (you can explain a high level, no need for specifics)?

At a high level, Cuckoo Hashing uses **two different hash functions** and **two possible locations** for every key. When inserting an item, if its first location is full, the algorithm "kicks out" the existing item to its *other* hash location—just like a cuckoo bird pushing another egg out of a nest. This process continues until every item finds a place. The benefit is that **lookup becomes extremely fast (O(1))**, which is ideal for TikTok because its recommendation engine needs to retrieve user–video interaction signals *instantly* to personalize the feed as you scroll.

Given two vectors (eg like shown below), what two 'similarity measures' can we calculate ? What do vectors have to do, with IR in the first place?!

Cosine Similarity Measures the angle between two vectors. If the angle is small (cosine close to 1), the vectors are very similar.

(From notes: cosine formula and examples in Section 7.5.)Euclidean Distance Measures how far apart two vectors are in space; smaller distance = more similar.

(From notes: KNN distance formulas and clustering sections use Euclidean distance.)

What do vectors have to do with IR in the first place?!

*In IR, every **document** and every **query** is converted into a **vector** (using TF-IDF weights or embeddings). Once everything is represented as vectors, the search engine can compute **similarity** between the query vector and document vectors to decide which documents are most relevant. This is the core idea of the **Vector Space Model**, where relevance = closeness between vectors in high-dimensional term space.*

(From notes: "Document & Query representation as vectors" and "Cosine similarity for ranking.")

what Does a Search Engine Do?
Core architecture
Two largely independent pipelines:
Online query pipeline (serving users)
User query → Query processing (parsing, rewriting)
→ Inverted index lookup → Ranking → Snippets → UI (results page).
Offline indexing pipeline (building/refreshing the index)
Crawlers fetch web pages → Content cleaning & deduplication
→ Indexer builds and updates inverted index (and other structures)
→ Stored in distributed systems (GFS / BigTable).
Why separate?
Queries must be served in milliseconds, while crawling and indexing can be continuous and much slower.

Q: Why do search engines need an index instead of scanning raw pages at query time?

A: Scanning raw pages requires reading every document for each query, which is O(N × page_length) and far too slow. An inverted index precomputes a mapping from terms to the posting lists of documents containing them, so a query only needs to access the postings for its terms, making retrieval efficient enough for real-time responses.

MapReduce, GFS, BigTable
13.1 MapReduce
High-level:
Map: process input records and emit intermediate (key, value) pairs.
Shuffle: group intermediate values by key and distribute to reducers.
Reduce: aggregate values for each key to produce final output.
Fault tolerance:
If a worker fails, its tasks are reassigned and recomputed.
Master tracks task status; straggler tasks can be duplicated.
Uses in IR:
Inverted index construction.
Document statistics computation.
Log processing.
13.2 GFS (Google File System)
Master node + chunk servers.
Files split into large chunks (e.g., 64 MB), replicated across servers.
Master manages metadata and placement; clients talk directly to chunk servers.
13.3 BigTable
Sparse, distributed, persistent multidimensional sorted map.
Indexed by (row key, column family, timestamp).
Used to store large-scale structured data such as web indexes, logs, and user data.

Q1: Why is MapReduce well-suited for building a web-scale inverted index?

A: Each document can be processed independently in the Map phase to emit (term, docID) pairs. The Shuffle groups all occurrences of each term together, and the Reduce phase can then aggregate postings lists for each term, naturally aligning with the inverted index structure and allowing parallel processing across many machines.

Q2: What roles do GFS and BigTable play in large-scale IR systems?

A: GFS provides a distributed, fault-tolerant file system for storing large raw and intermediate data (e.g., crawled pages, MapReduce inputs/outputs). BigTable provides a scalable, low-latency storage for structured data like indexes or metadata, enabling efficient random access to rows and columns needed during query processing

Query Processing Heuristics & Champion Lists
Search engines apply many optimizations to speed up scoring:
Ignore low-IDF terms (stopwords, frequent terms).
Only score documents that contain **multiple query terms**.
Maintain **champion lists** (aka fancy lists): for each term, a list of top-r documents by TF–IDF for that term.
High/low lists: split postings into high-impact and low-impact tiers; score from high tier first.
These heuristics reduce the number of documents that need detailed scoring.

Q: What is a champion list and why is it useful?

A: A champion list is a precomputed list of top-r documents for each term, ranked by term importance (e.g., TF–IDF). At query time, instead of scoring all documents containing the term, the engine first focuses on documents in the champion lists, which greatly reduces scoring cost while still likely capturing the most relevant results.

16. Vector Databases, Semantic Search, ANN & FAISS

16.1 What is vector similarity search?

Represent items (documents, sentences, images, etc.) as vectors in a high-dimensional embedding space.
Given a query, compute its embedding and find nearest vectors using cosine similarity or Euclidean distance.
Similar vectors correspond to semantically similar items.

16.2 Why vector DBs are needed

Traditional inverted indexes:
Work on exact or fuzzy term matching.
Cannot easily capture semantic similarity when terms do not overlap.
Vector DBs:
Store embeddings and allow **Approximate Nearest Neighbor (ANN)** search.
Key IR tasks:
Semantic retrieval / RAG: retrieve semantically related documents even if they use different words.
Fast large-scale similarity search: handle millions/billions of vectors with sub-linear query time using ANN structures.
Without vector DBs:
We would need to compute similarity between the query vector and every document vector, which is O(N) per query and not feasible at scale.

16.3 ANN algorithms & FAISS

ANN (Approximate Nearest Neighbor): trade a small amount of accuracy for large speed gains.
Approaches: tree-based (Annoy), graph-based (HNSW), and **inverted file indexing** as in FAISS.
FAISS IVF (Inverted File Index):
Cluster database vectors into K clusters via K-means.
For each cluster, maintain a list (inverted list) of vectors assigned to that centroid.
For a query:
Find its nearest centroids.
Only search within associated inverted lists to find nearest neighbors.

Q1: What is vector similarity search and how does it differ from traditional keyword search?
A: Vector similarity search retrieves items based on the distance between dense vector embeddings, which capture semantic information, rather than based on exact term overlap. Unlike keyword search, which relies on matching tokens in an inverted index, vector search can find relevant documents that share meaning but not necessarily the same words.
Q2: Why are vector databases and ANN methods like FAISS necessary for large-scale semantic retrieval?
A: At large scale, computing the similarity between a query vector and all document vectors is too slow. Vector databases use ANN structures like FAISS, which cluster vectors and restrict search to a few promising regions, allowing semantic nearest-neighbor retrieval in sub-linear time and making real-time semantic search feasible.

9. Snippets, Rich Snippets, Knowledge Panels

9.1 Standard snippets

Goal: show a short text summary under each search result that helps user decide whether to click.
Typical algorithm:
Identify paragraphs containing query terms.
Score each paragraph by:
Positional features (early paragraphs may be favored).
Length (avoid overly long or short ones).
Term coverage.
Example: Score_k = β_k + max(APL, MPL) (as in your notes).
Select top-scoring paragraph and truncate around query terms.

9.2 Rich snippets / structured snippets

Extracted from structured data embedded in pages (e.g., schema.org markup).
May include:
Ratings, reviews, prices, events, recipe info, FAQ answers, etc.
Benefits:
Users: quick understanding of key attributes.
Sites: higher click-through rate.
Search engine: better signals on result usefulness.

9.3 Knowledge graph panels (entity cards)

Generated from Knowledge Graph / RDF triples.
Show facts about entities (person, organization, place) directly on the results page.

Q1: How do rich snippets differ from traditional text snippets?
A: Traditional snippets are extracted from page text to show a relevant paragraph with highlighted query terms. Rich snippets, in contrast, use structured data (such as ratings, reviews, prices, or event details) embedded in the page to display enhanced information like stars, prices, or dates, providing more actionable information at a glance.
Q2: How do RDF triples and knowledge graphs improve snippet quality?
A: RDF triples encode facts as <subject, predicate, object>, enabling the search engine to understand entity relationships. Using this structured knowledge, the engine can generate knowledge panels or fact-based snippets that directly answer user questions instead of just showing text fragments, resulting in more precise and informative snippets.

8. PageRank & Link Analysis

8.1 Basic idea

PageRank models a **random surfer**:
At each step, surfer follows a random outgoing link with probability β, or jumps to a random page with probability (1−β).
PageRank of a page is the steady-state probability that the surfer is on that page.

8.2 Formula

For page j:

$$PR(j) = \frac{1-\beta}{N} + \beta \sum_{i \to j} \frac{PR(i)}{outdeg(i)}$$

N = total number of pages.
β ≈ 0.85 (damping factor).

8.3 Problems: dead ends & spider traps

Dead ends: pages with no outgoing links; they absorb rank and reduce the total rank mass.
Spider traps: groups of pages that only link to each other, trapping rank.
Taxation (damping) plus adding random jumps solves these issues by continuously "re-injecting" rank uniformly.

8.4 Iterative computation

Initialize PR(j) = 1/N for all j.
Repeatedly update using the formula until convergence (change below a threshold or fixed number of iterations).

Q1: Explain how the damping factor in PageRank helps handle dead ends and spider traps.
A: The damping factor β models the probability of following a link, while (1−β) models the probability of jumping to a random page. Even if a page or group of pages has no outgoing links or forms a trap, the random jumps continuously redistribute rank across the entire web, preventing rank from being permanently absorbed by dead ends or spider traps.
Q2: Given a small graph with 3–4 pages, compute PageRank after one or two iterations.
A: On the exam, you would plug the initial PR values into the formula for each page, sum contributions from in-links, apply the (1−β)/N base term, and compute updated PR values, showing each step.

12. Clustering: K-means & Hierarchical

12.1 K-means clustering

Algorithm:
Choose k initial centroids (random or heuristic).
Assign each data point to the nearest centroid.
Recompute centroid of each cluster as the mean of assigned points.
Repeat 2–3 until assignments converge.
Complexity: O(i × k × n × m) (i iterations, n points, m dimensions).
Used for:
Document clustering, topic exploration, pre-clustering for indexing.

12.2 Hierarchical clustering

Agglomerative (bottom-up):
Start with each document as its own cluster.
Repeatedly merge the two closest clusters.
Continue until only one cluster remains (or desired number).
Linkage strategies:
Single-link (min distance)
Complete-link (max distance)
Average-link
Centroid
Output: dendrogram (tree of clusters).

Q1: How does K-means differ from hierarchical agglomerative clustering?
A: K-means is a partitioning method that requires specifying k in advance and iteratively refines cluster assignments and centroids; it is efficient and scalable. Hierarchical agglomerative clustering builds a nested tree of clusters by successively merging the closest clusters and does not require a predefined k but is more computationally expensive.
Q2: Why might we use clustering in IR?
A: Clustering can group similar documents together, enabling topic-based browsing, improving navigation, or providing diversified search results by selecting documents from different clusters, which can reduce redundancy and increase coverage of different aspects.

Other Topics: TikTok, OPL, LLM Hallucinations

17.1 TikTok's data structure: Cuckoo hashing

Uses two hash functions h_1 and h_2; each key has two possible positions.
Insert:
If the first position is occupied, eject the existing key and move it to its alternate position; repeat if needed.
If too many displacements occur, rebuild the table.
Benefit:
O(1) worst-case lookup (constant-time retrieval), important for real-time recommendation at massive scale.

17.2 OPL stack

O = OpenAI (LLM + embeddings).
P = Pinecone (vector DB).
L = LangChain (orchestration / tool-calling layer).
Main use:
Building applications where an LLM can **retrieve external knowledge from a vector DB** and **call tools** via an orchestration framework, i.e., RAG + agents with memory and tools.

17.3 Why LLMs hallucinate (mechanism)

Training objective: **maximize next-token predictive likelihood** on large text corpora (self-supervised).
The model is optimized to produce *fluent, likely text given the prompt*, not to check factual correctness.
When the model faces gaps in knowledge or under-specified prompts, it will still produce the most probable-looking continuation, which may be factually wrong.
No inherent grounding to external reality; by default, it doesn't query databases or verify facts unless integrated into a broader system (e.g., RAG, Graph RAG).

Q1: What is Cuckoo hashing and why is it attractive for systems like TikTok's recommendation engine?
A: Cuckoo hashing assigns each key two potential locations via two hash functions; on insertion, a collision causes an existing key to be displaced to its alternative location. This scheme ensures O(1) worst-case lookup and high memory utilization, which is critical in recommendation systems that need to retrieve user–item interactions extremely quickly at large scale.
Q2: What does the OPL stack stand for, and what is its primary purpose?
A: OPL stands for OpenAI, Pinecone, and LangChain. It is a pattern for building LLM applications where OpenAI provides the model and embeddings, Pinecone provides vector storage for semantic retrieval, and LangChain orchestrates calls among the LLM, vector DB, and external tools, enabling retrieval-augmented and agent-style applications.
Q3: At the algorithmic level, why do LLMs generate hallucinations?
A: LLMs are trained to predict the next token that is most probable given previous tokens, optimizing for fluency and likelihood rather than factual accuracy. When the model lacks knowledge or the prompt is outside its training distribution, it still outputs plausible-sounding text by extrapolating patterns, which leads to fluent but incorrect statements referred to as hallucinations

10. Spelling Correction & Autocomplete

10.1 Spelling correction

Key components:
Edit distance (Levenshtein distance): minimum number of insertions, deletions, and substitutions to transform one string into another.
Confusion matrix: probabilities of different spelling errors (e.g., substituting one letter for another).
Noisy channel model:
Observed string = corrupted version of true word.
Choose candidate word that maximizes P(true word) × P(observed | true word).
Pipeline:
Generate candidate corrections within a small edit distance.
Score them using language model + confusion matrix.
Pick the highest-scoring candidate.

10.2 Autocomplete

Use a **Trie (prefix tree)** built over query logs.
For a given prefix, traverse the trie and output completions sorted by frequency/popularity.
Complexity ~O(length of prefix) for lookup.

Q1: Why do we need a confusion matrix in spelling correction?
A: A confusion matrix encodes how likely different types of spelling errors are (e.g., substituting 'e' for 'r'). Combining this with language priors in a noisy channel model allows the spelling corrector to choose not just words with small edit distance, but those corrections that are statistically most plausible given how people actually make mistakes.
Q2: Why is a trie an appropriate data structure for autocomplete?
A: A trie compactly organizes strings by shared prefixes, so we can quickly traverse from the root down to the node representing the query prefix and then enumerate possible completions. This makes prefix lookup efficient and naturally supports listing suggestions in order of popularity.

Knowledge Graphs, RDF, QA & Graph RAG

14.1 Taxonomy, ontology, knowledge base

Taxonomy: tree-like hierarchy of concepts (e.g., Animal → Mammal → Dog), with property inheritance.
Ontology: more general graph modeling entities, attributes, and multiple types of relations.
Knowledge base: collection of entities and relations with an inference engine.

14.2 RDF triples

Basic fact representation: **<subject, predicate, object>**
Example: <"Einstein", "bornIn", "Ulm">.
Triples are nodes and edges in a knowledge graph.
Benefits for search:
Allow semantic queries over entity relationships.
Enable direct answering of factual questions.
Improve disambiguation and snippet quality.

14.3 QA pipeline

Typical pipeline:
Question processing: POS tagging, NER, identify question type and target entities.
Passage retrieval: use inverted index to fetch candidate passages.
Answer extraction: use semantic cues, NER, and KG to locate actual answer spans.

14.4 NER → RDF → Knowledge Graph → Graph RAG

Pipeline to build a **Graph RAG** system:
Named Entity Recognition (NER): extract entities (people, organizations, locations) from text.
Relation extraction: identify relationships between entities (worksAt, bornIn, locatedIn, etc.).
RDF triples: encode facts as <subject, predicate, object>.
Build knowledge graph: nodes = entities, edges = typed relations.
Graph retrieval: for a user query, retrieve relevant entities and subgraphs (neighbors, paths).
LLM grounded answering: feed these graph facts into the LLM as context to constrain answers, reducing hallucinations.

Q1: How do RDF triples improve search compared to plain text indexing?
A: RDF triples store explicit semantic relationships between entities, enabling the search engine to reason over facts and relations rather than just matching keywords. This supports more precise answering of factual queries and reduces ambiguity, as the system can retrieve "Bob works at Google" through the appropriate predicate even if the exact query wording differs from the text.
Q2: Describe the pipeline from NER to a Graph RAG-style system.
A: Starting with raw text, we run NER to extract entities and relation extraction to detect relationships between them, encoding these as RDF triples. These triples populate a knowledge graph that can be queried for relevant subgraphs based on a user query; the retrieved graph facts are then provided as grounding context to an LLM, which uses them to produce more factually reliable, low-hallucination answers.