AI Powered Search MEAP

➔ The book's goal is to guide through examples of the most applicable machine learning algorithms and techniques commonly used to build intelligent search systems.

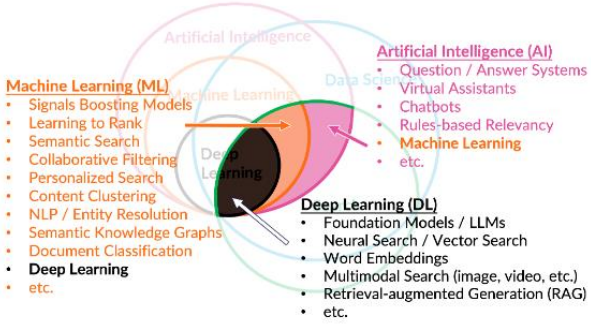
1. Introducing AI – Powered Search / Information Retrieval

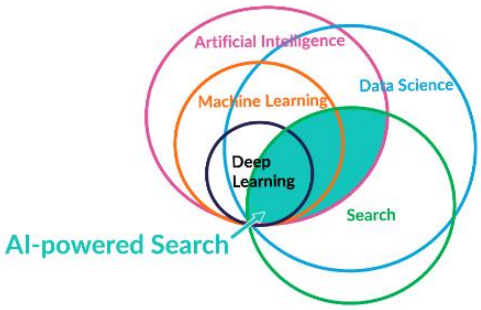
    a. Traditional Search:

➔ Search engine = matching (finding docs that match a certain query) + ranking (ordering those docs in order of relevance to the query)
➔ Before & after the search, we should include pre-processing steps to understand the query (before) and to extract answers or summarize from the matched docs (after)

    b. AI-powered Search (AIPS):

➔ Considering 3 different techniques to build AIPS: ML Techniques (without DP), Deep Learning Techniques and AI Techniques (without ML & DL)

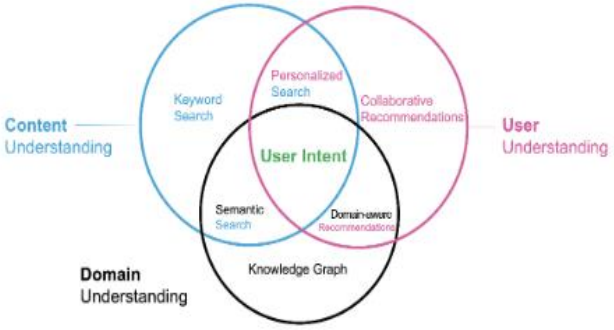




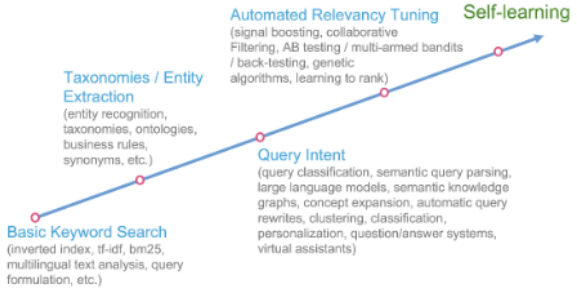
➔ ML Techniques: Signal boosting models (top docs per query or category) + Ranking classifiers (learning to rank) + Collaborative filtering models (to generate recommendations) + personalized search + Knowledge graphs + Semantic search (meaning not just keywords -> enabled by knowledge graphs) + NLP + Doc's classification.
➔ DL Techniques: Neural Networks (NN) (to understand queries & docs & rank & summarize) + LLMs (trained by text to understand meaning of words & phrases but LLMs can also be trained on other types of content -> images & audio & video) + Multimodal Search (text-to-image search, text-to-audio, image-to-

video, etc) + Words Embeddings (LLMs also used to generate embeddings -> vector representation of content – easier to compare to find similarities)

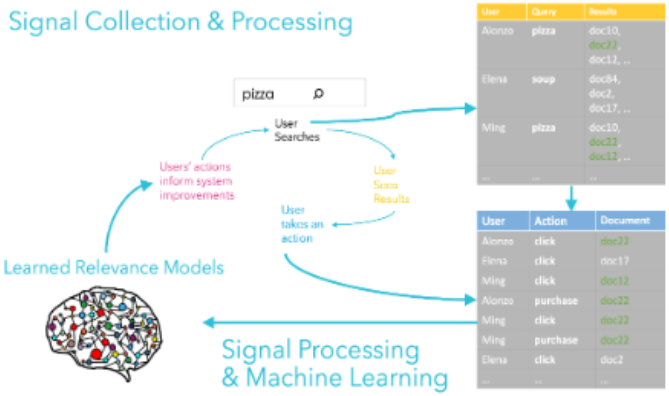
- ➔ AI Techniques: Chatbots (rule-based entirely) + questions-answering systems predefined + virtual assistants
- ➔ LLMs are trained on large portion of the internet (leading to good general understanding but also to hallucination which is not reliable for answering factual questions)
- ➔ RAG: To improve the accuracy of text generation in LLMs, usually it's included an external knowledge source in which it's applied 2 key components: Retrieval (from original dataset + extended knowledge source (+ accurate) and Generation (along with the retrieved information it generates more accurate and contextually rich output)
- ➔ Recommendation algorithms: Divided into 3 categories: Content-based recommenders, behavior-based recommenders, and multi-modal recommenders.
- ➔ Using the personalization profile information to find content can be addressed in 4 possible ways:
    - Ignore the profile and only use explicit inputs (traditional keyword search)
    - Use profile implicitly (listening to jazz music will recommend jazz music even though not explicitly expressed in your profile) + explicit user input (personalized search)
    - Use the profile explicitly (you set that you are interested in technology papers for instance) and provide the user an ability to adjust it (user-guided recommendations)
    - Use the profile explicitly with no ability for a user to adjust (traditional recommendations)
- ➔ Recommendation and search tasks shouldn't be separated and less alone managed by different teams -> data scientists usually in recommendation team with no background in search engines & engineering teams in search teams with no data scientist background.
- ➔ AIPS aims to unify both approaches (search + recommendations) by continuous learning from users + content.
- ➔ AIPS needs to include content understanding (search engines, semantic search, personalized search) + user understanding (recommendation engines, domain-aware recommendations) + domain understanding (knowledge graphs) to fully understand the user's intent.



- ➔ AIPS has per goal to develop the ability to find the content by matching & searching, the ability to understand the user by its specific preferences to return personalized results and the ability to interpret and understand the relation between words/phrases within a specific domain.
- ➔ AIPS must be an expert understanding of the domain the user is searching, an expert understanding of the user and their preferences, and an expert ability to match and rank arbitrary queries against any content.
- ➔ Progression to AIPS system:
- ➔ To develop the background of Relevance Engineers (Basic Keyword Search and techniques) read Relevant Search by Doug Turnbull & John Berryman (Manning 2016)



- ➔ AIPS system's key is Reflected Intelligence based of feedback loops to continually learn and improve the quality of search apps.
- ➔ In feedback loops, the searches, clicks, likes, add to carts, purchases, comments that users perform are called signals, and that is the data used train the ML models.



- ➔ Transformers: Introduced in 2017 by Google, it's a Neural Network Architecture highly influential in handling sequential data, particularly text. To make it simple, first it's encoded and then decoded. Encoder: Tokenization of phrases (subdivision into words of original sentences), Embeddings (numerical vectorial assignation to words capturing semantic), Self-Attention contextually adjusted embedding (relations between words + context influence and manipulate further the embedding of tokens). Decoder: Decode of the specific-context embeddings depending on the target value (translation, classification, answering a question, summarization, generation of new content and more). Key: Transformers are context sensitive.
- ➔ To build a AIPS system, you will need a core search engine such as Solr, Elasticsearch/OpenSearch or Vespa. Then, transform your data to make it useful for the search engine. (1.5.4) You will need as well as a processing framework like Spark and a continuous workflow scheduling mechanism to keep jobs running in sequence. Incoming data in a constant stream needs to be captured + stored.

2. Working with natural language

- ➔ Unstructured data: text (mixing dates, numbers, facts), images, audio, video.
- ➔ Semi-structured data: unstructured data + structured data
- ➔ For unstructured data like text, instead of using foreign key attributes like in SQL, we use fuzzy foreign keys. Instead of relating 2 different (or more) tables, it relates 2 different (or more) documents. Fuzzy refers to the fact that it's not exactly the same foreign key in both, but rather very similar (One doc talks about Trey Grainger and the other one about just Trey, in one video it's clearly written Trey Grainger and in one voicemail (audio) it's said the same name) -> This is called Entity resolution (refers to the same entity) and proves the hidden structure of text.
- ➔ Big problem: Polysemy poses a problem when it comes to multiple meanings for a single word, that's why it's important to leverage context to differentiate.
- ➔ Text Fields or Fields form a document, multiple documents form a corpus.
- ➔ Characters < character sequences < terms < term sequences < phrases < fields < documents < document sets < corpus.
- ➔ Distributional Hypothesis: Hypothesis stating that words that occur in similar contexts tend to share similar meanings.
- ➔ Words embeddings without context:

query | exact term lookup in inverted index

	apple	caffeine	cheese	coffee	drink	donut	food	juice	pizza	tea	water
latte	0	0	0	0	0	0	0	0	0	0	0
cappuccino	0	0	0	0	0	0	0	0	0	0	0
apple juice	1	0	0	0	0	0	0	1	0	0	0
cheese pizza	0	0	1	0	0	0	0	0	1	0	0
donut	0	0	0	0	0	1	0	0	0	0	0
soda	0	0	0	0	0	0	0	0	0	0	0
green tea	0	0	0	0	0	0	0	0	0	1	0
water	0	0	0	0	0	0	0	0	0	0	1
cheese bread sticks	0	0	1	0	0	0	0	0	0	0	0
cinnamon sticks	0	0	0	0	0	0	0	0	0	0	0

- ➔ Words embeddings with context:

	food	drink	dairy	bread	caffeine	sweet	calories	healthy
apple juice	0	5	0	0	0	4	4	3
cappuccino	0	5	3	0	4	1	2	3
cheese bread sticks	5	0	4	5	0	1	4	2
cheese pizza	5	0	4	4	0	1	5	2
cinnamon bread sticks	5	0	1	5	0	3	4	2
donut	5	0	1	5	0	4	5	1
green tea	0	5	0	0	2	1	1	5
latte	0	5	4	0	4	1	3	3
soda	0	5	0	0	3	5	5	0
water	0	5	0	0	0	0	0	5

- ➔ Vector similarity (one example of function, there are multiple possibilities)

Term Sequence:	Vector:
apple juice:	[1, 5, 0, 0, 0, 4, 4, 3]
cappuccino:	[0, 5, 3, 0, 4, 1, 2, 3]
cheese bread sticks:	[5, 0, 4, 5, 0, 1, 4, 2]
cheese pizza:	[5, 0, 4, 4, 0, 1, 5, 2]
cinnamon bread sticks:	[5, 0, 4, 5, 0, 1, 4, 2]
donut:	[5, 0, 1, 5, 0, 4, 5, 1]
green tea:	[0, 5, 0, 0, 2, 1, 1, 5]
latte:	[0, 5, 4, 0, 4, 1, 3, 3]
soda:	[0, 5, 0, 0, 3, 5, 5, 0]
water:	[0, 5, 0, 0, 0, 0, 0, 5]

Vector Similarity (a, b):

$$\cos(\theta) = \frac{a \cdot b}{|a| \times |b|}$$

Vector Similarity Scores:

Green Tea	
0.94	water
0.85	cappuccino
0.80	latte
0.78	apple juice
0.60	soda
...	...
0.19	donut

Vector Similarity Scores:

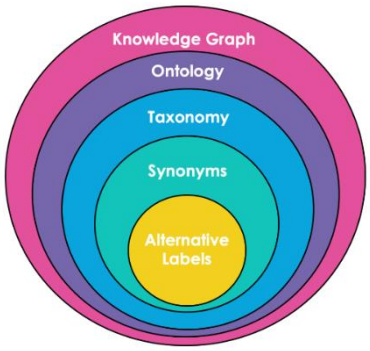
Cheese Pizza	
0.99	cheese bread sticks
0.91	cinnamon bread sticks
0.89	donut
0.47	latte
0.46	apple juice
...	...
0.19	water

Vector Similarity Scores:

Donut	
0.99	cinnamon bread sticks
0.89	cheese bread sticks
0.89	cheese pizza
0.57	apple juice
0.51	soda
...	...
0.07	water

➔ It is commonplace to encode also sentence embeddings, paragraph embeddings and even document embeddings.

➔ Domain-specific model knowledge techniques:



- Alternative Labels (Substitute term sequences with identical meanings)
  - CTO => Chief Technology Officer
  - specialise => specialize
- Synonyms (Substitute term sequences that can be used to represent the same or very similar things)
  - human => homo sapiens, mankind.
  - food => sustenance, meal.
- Taxonomy (a classification of things into categories)
  - human is mammal.
  - mammal is animal.
- Ontology (a mapping of relationships between types of things)
  - animal eats food.
  - human is animal.
- Knowledge Graph (An instantiation of an Ontology that also contains the things that are related)
  - John is human.
  - John eats food.

3. Ranking and content-based relevance

➔ Search engines fundamentally do 3 things: indexing (ingest content) + matching (return matching incoming queries) + ranking (sort return content based on some measure)

➔ Cosine Similarity: The closer to 1, the more similar 2 vectors are, the closer to 0, the less.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

➔ Term Frequency Calculation:
The t represents a term and d represents a document. TF equals the square root of the number of times the term appears in the current document, divided by the number of terms in the document to normalize.

$$TF(t \in d) = \frac{\sqrt{\text{d.count}(t)}}{\text{d.totalTerms}}$$

➔ Document Frequency (DF): D is the set of all docs, t the input term. DF is simply the number of documents containing the input term, the lower the number, the more specific and important the term is when seen in queries.

$$DF(t) = \sum_{d=1}^{|D|} \text{if} \quad t \in D_i : 1; \quad \text{if} \quad t \notin D_i : 0$$

➔ Inverse Document Frequency: |D| is the total count of all documents, t is the term, and DF(t) is the count of all documents containing the term. The lower the fraction, the more insignificant a term, and the higher, the more a term in a query should count toward the relevance score.

$$IDF(t) = 1 + \log\left(\frac{|D| + 1}{DF(t) + 1}\right)$$

➔ TF-IDF Score: A balanced feature-weighing & text-ranking score

$$\text{TF-IDF} = TF \cdot IDF^2$$

➔ BM25 Scoring Function: Default similarity algorithm in Apache Solr Lucene, it's better than the TF-IDF cosine similarity ranking. You need to download the software to execute the Python code to create the collection and add docs to it to later apply the BM25 function with a query. Bear in mind that this is only by looking at term matches alone. (Just like all the previous functions)

$$Score(q,d) = \sum (t \in q) \frac{idf(t) \cdot (tf(t \in d) \cdot (k+1))}{tf(t \in d) + k \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

Where:

t = term; **d** = document; **q** = query; **i** = index

tf(t in d) = numTermOccurrencesInDocument ½

idf(t) = 1 + log (numDocs / (docFreq + 1))

$|d| = \sum_{t \text{ in } d} 1$

$\text{avgdl} = \left(\sum_{d \text{ in } i} |d|\right) / \left(\sum_{d \text{ in } i} 1\right)$

k = Free parameter. Usually ~1.2 to 2.0. Increases term frequency saturation point.

b = Free parameter. Usually ~0.75. Increases impact of document normalization.

➔ Here is a partial list of common relevance techniques:

- Geospatial Boosting: Documents near the user running the query should rank higher.
- Date Boosting: Newer documents should get a higher relevancy boost.
- Popularity Boosting: Documents which are more popular should get a higher relevancy boost.
- Field Boosting: Terms matching in certain fields should get a higher weight than in other fields.
- Category Boosting: Documents in categories related to query terms should get a higher relevancy boost.
- Phrase Boosting: Documents matching multi-term phrases in the query should rank higher than those only matching the words separately.
- Semantic Expansion: Documents containing other words or concepts that are highly related to the query keywords and context should be boosted.

➔ Additive vs Multiplicative Boosting:

- Additive:
  - Concept: Involves adding extra features to the query's feature vector.
  - Application: Useful when you want to enhance relevance by considering additional distinct features.

        - Limitation: Requires carefully managing the weights and normalization of all features to ensure each contributes appropriately to the overall relevance. It can become complex and hard to control when multiple boosts are involved.
        - Control: Offers precise control over the contribution of each feature to the relevance score, but at the cost of complexity in managing and scaling these features.
    - Multiplicative:
        - Concept: Enhances relevance by multiplying the document's overall score by a calculated value.
        - Flexibility: Allows boosts to accumulate, leading to a fuzzier match. It doesn't require tight constraints like additive boosting.
        - Implementation: Can be implemented using specific query parsers or parameters.
        - Example: Multiplicative Boosting Example:
            - Scenario: Imagine you have a search engine for a library of books, and you want to enhance the search functionality to better serve user queries. Let's consider a user searching for "The Cat in the Hat."
            - Objective:
            - You want to boost the relevance of search results not only based on how well they match the query but also on additional factors like the popularity of the books and their publication dates.
            - Popularity Boost: Let's say each book in your library has a 'popularity' score in its metadata. You decide to use this score to boost the relevance of popular books. For instance, a book with a popularity score of 8 would get a stronger boost compared to a score of 3.
            - Formula: Relevance Score = Original Score * (1 + Popularity * Boost Factor)
                - If you want to consider further boosts, like recency, you will consider: Relevance Score = Original Score * (1 + Popularity * Boost Factor) * (1 + recency * Boost Factor)
                - Basically, you multiply again and again.

- Example: If you set the Boost Factor to 0.1 (or 10%):
- A book with a popularity score of 8 would have its relevance score multiplied by 1 + (8 * 0.1) = 1.8.
- A book with a popularity score of 3 would have its relevance score multiplied by 1 + (3 * 0.1) = 1.3.
- Query Implementation:
- If you're using a specific query parser, you can directly include this boost in your query. For instance, in Solr or similar systems:
- q=the cat in the hat&defType=edismax&boost=mul(popularity, 0.1)
- This query essentially tells the system to multiply the relevance score of each document (book) by 1 + 0.1 * popularity.

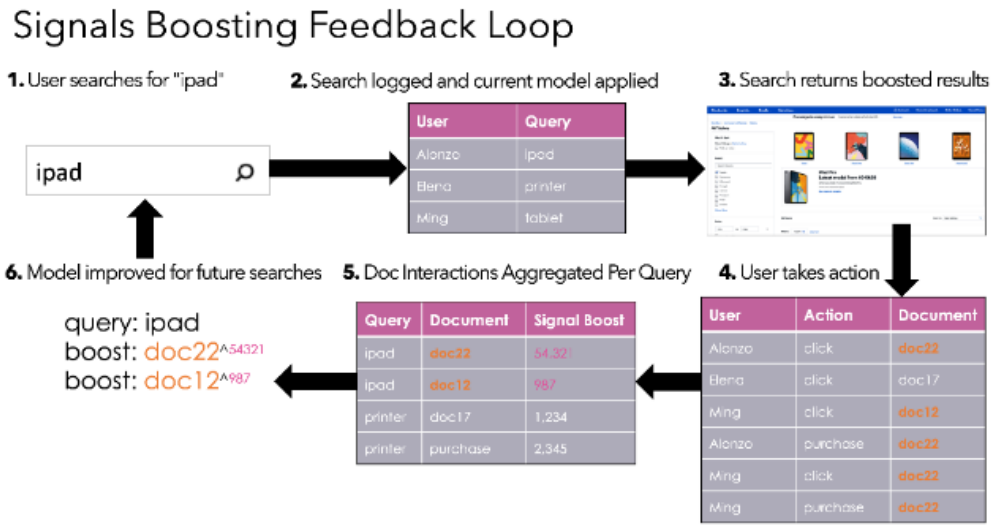
  - Caveat: While it offers greater flexibility and simplicity in combining different relevance features, there's a risk of certain features becoming too dominant, overshadowing others.

➔ Filtering (by matching) results before computing the similarity scoring function is primarily a performance optimization and the first few pages of search results would likely look the same regardless of whether you filter the results or just do relevance ranking.

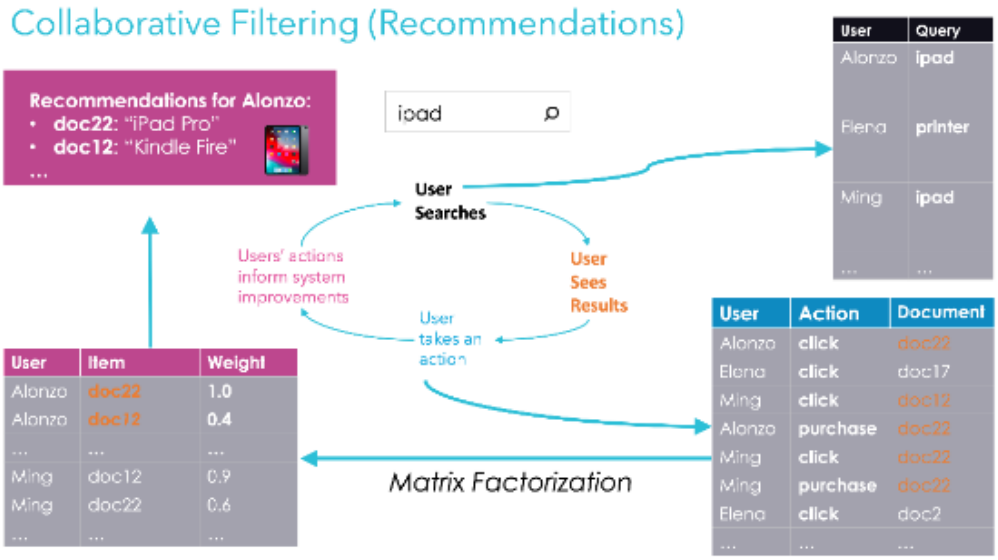
- Filtering in 4 different ways:
  - "Statue of liberty" -> matching docs containing this exact phrase
  - statue AND of AND liberty -> docs containing all the terms statue, of, and liberty, but not necessarily as a phrase.
  - statue OR of OR liberty -> docs containing any of the three terms, which means docs only containing “of” will match, but that documents containing statue and liberty should rank much higher due to TF-IDF and the BM25 scoring calculation.
  - statue of liberty -> combinations of at least 2 of the three terms.

4. Crowdsourced Relevance

➔ When building search engines, we have two high-level types of data: "content" and "signals".
➔ Content : Our documents (web pages, product listings, computer files, images, videos, etc)
➔ Signals: It reflects how users engage with content.
➔ Reflected Intelligence models: Is all about creating feedback loops that constantly learn and improve (by applying machine learning techniques) based on evolving user interactions (which produce signals that we collect), this creates "learned relevance models" which live in a constant feedback loop of user's interactions, improving search results and getting smarter and showing more relevant products over time, and of course, automatically adjusting as user interests and content evolve.
➔ We'll consider 3 categories (there are more): Signals boosting models (popularized relevance -> improve relevance of the most popular + highest traffic queries), Collaborative filtering models (personalized relevance) and Learning to Rank models (generalized relevance).
    - Signal Boosting Models: By searching "ipad", the result list could be plenty of accessories instead of directly the tablet, therefore, by applying the signal boosting (counts how many people clicked on a certain product after having searched the same query) we can boost the topmost relevant products according to the query and show them first in the list of results.
      Drawback: Self-reinforcing cycle -> documents that are already popular tend to receive a higher boost, resulting in them getting even more signals, while documents that have never been seen get no signals boosting.



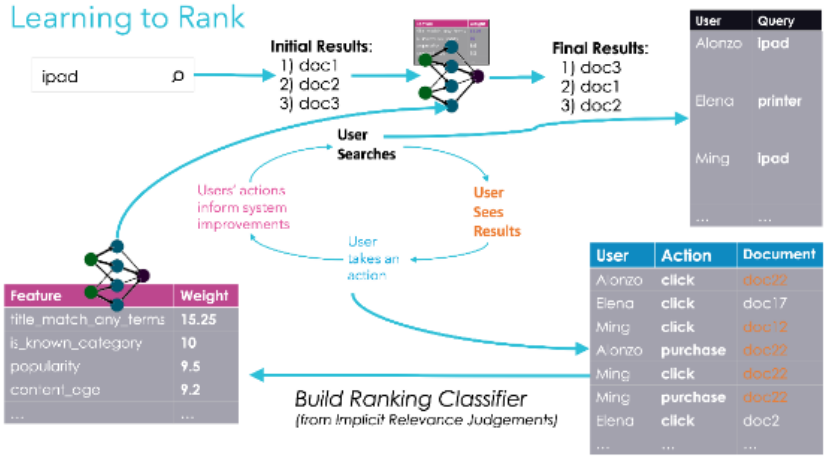
- Collaborative Filtering Models: Whereas signal boosting models determine which results are usually the most popular across many users, personalized relevance focuses on determining which items are most likely to be relevant for a specific user. Collaborative filtering is the process of using observations about the preferences of some users to predict the preferences of other users ("users who liked this item also liked these items"). It uses a matrix that relates user & item with a weight representing the strength of the positive interactions (clicks, purchases, ratings, and so on).
  Drawback: Cold start problem -> Scenario where returning results is dependent upon the existence of signals, but where new documents that have never generated signals are not getting displayed.



ALERT: Signal boosting & Collab filtering only work on docs that already have signals -> a substantial portion of queries and docs won't benefit from these approaches until they receive traffic and generate signals.

- Learning to Rank Models (LTR):
  - Learning to Rank involves training a model to rank search results based on their relevance to a query.
  - This system uses relevance judgments (lists mapping queries to their ideally ranked set of documents) to train a relevance model.
  - The model, once trained, can be used to rank search results for all queries.
  - Example: Imagine an online bookstore with a search feature.

- A user searches for "iPad."
- Initial Ranking: The search engine initially ranks the results based on keyword matching, perhaps using a simple algorithm like BM25.
- Top-N Results: The search engine selects the top few hundred results for further processing.
- Learning to Rank Process is applied with feature identification: In the LTR model, features like title_match_any_terms, is_known_category, popularity, content_age are identified as important for determining the relevance of a book to the search query.
- Model Training: A ranking classifier is trained using these features. This training is based on historical data, which might include expert judgments on what constitutes a relevant book for a given search term or user behavior data indicating which books users found most relevant.
- Model Deployment: This trained model is then deployed in the production search engine.
- Re-ranking: When a user searches for "iPad," the initial set of search results is re-ranked by the deployed LTR classifier. This classifier uses the learned features to determine a more nuanced relevance ordering of the books.

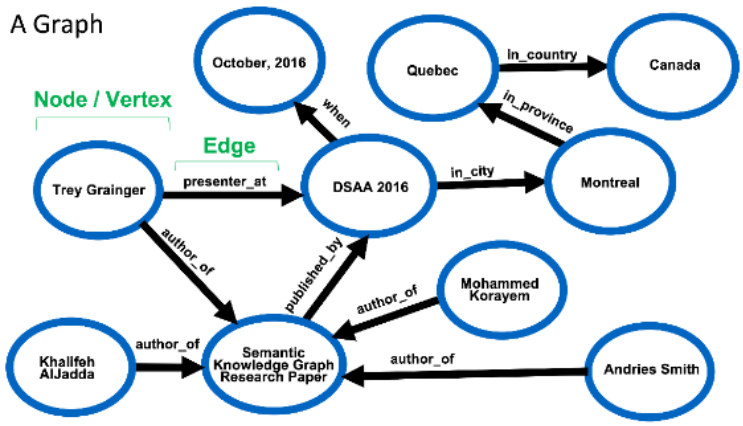


- Other Reflected Intelligence Models: 4.2.5

- Sentiment Analysis -> Negative, Neutral or positive according to customer reviews and then you can boost or penalize the source docs accordingly.
- PageRank -> Famous algorithm that played a pivotal role in Google's rise to become the leading search engine. It was a novel approach at the time of its creation, significantly enhancing the relevance of search results.
    - Concept: PageRank goes beyond just analyzing the text of a web page. Instead, it evaluates the web page's importance based on its relationships with other pages on the internet.
    - Link Analysis: The core of PageRank is analyzing how different web pages link to each other. It measures incoming and outgoing links to and from a web page.
    - Quality Measure: The algorithm assumes that high-quality, authoritative pages are more likely to receive links from other reputable sources. Conversely, these authoritative sources seldom link to low-quality pages.

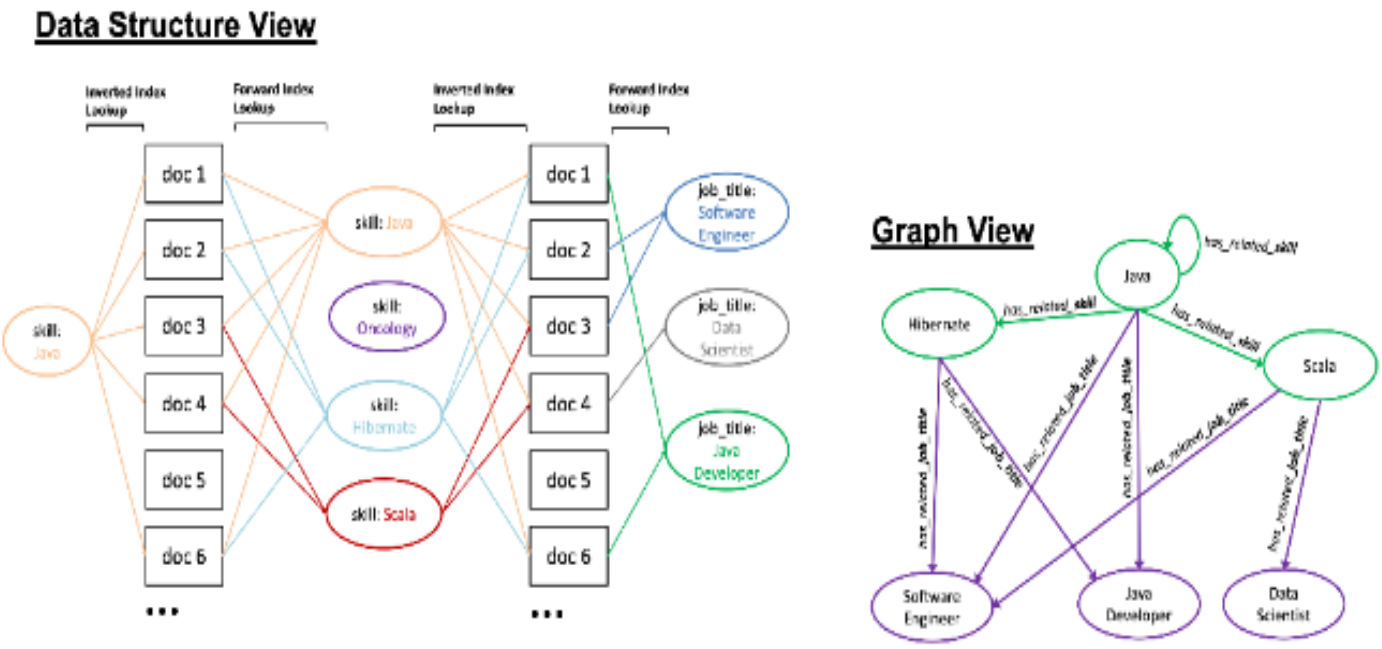
5. Knowledge Graphs Learning

➔ We'll cover how to build an explicit knowledge graph by hand (using graph database like Neo4j, ArangoDB), auto-generate an explicit knowledge graph (ConceptNet, DBpedia, LLM), and leverage a semantic knowledge graph that is already present within your search index.



- ➔ Apache Solr allows you to create a knowledge graph and integrate it into your search engine (2x1)
- ➔ Dependency graph -> breakdown of the parts of each word and phrase in a sentence along with an indication of which words refer to which other words, composed of 2 types of relationships: arbitrary & hyponym (hyponym is apple, hypernym is fruit and apple would be a hypernym for seed and hyponym for fruit) relationships.
- ➔ To extract arbitrary relationships we use RDF Triple, for hyponyms/hypernyms regular expressions (hyponym (=node) is a (edge) hypernym (=node)) -> problem: disconnection from rest of content -> we need a semantic knowledge graph
- ➔ Semantic knowledge graph: compact, auto-generated model for real-time traversal and ranking of any relationship within a domain -> Whereas a search engine typically finds and ranks documents relative to a query (a query to documents match), we can think of a semantic knowledge graph as a search engine that instead finds and ranks terms that best match a query.

- ➔ Multi-level graph traversal: Skill -> Similar skills -> job titles
- ➔ Semantic relatedness formula:



Concept:

- **Foreground Set**: A subset of documents where a specific term or query is prominent. For example, documents containing the word "pain."
- **Background Set**: The general set of documents, representing the overall distribution of terms.
- **Purpose**: The goal is to compare how often a specific term appears in the foreground set compared to its general occurrence rate in the background set.

Formula:

The relatedness calculation is given by:

$$z = \frac{\text{countFg}(x) - \text{totalDocsFG} \times \text{probBG}(x)}{\sqrt{\text{totalDocsFG} \times \text{probBG}(x) \times (1 - \text{probBG}(x))}}$$

Where:

- `countFg(x)`: The frequency of term `x` in the foreground set.
- `totalDocsFG`: The total number of documents in the foreground set.
- `probBG(x)`: The probability of term `x` occurring in the background set.

Interpretation of the Formula:

- The numerator measures the difference between the actual frequency of `x` in the foreground set and the expected frequency of `x` in the foreground set based on its background probability.
- The denominator is a normalization factor, essentially standardizing this difference.
- The formula resembles a z-score calculation, which measures how many standard deviations an element is from the mean.

Interpretation of Scores:

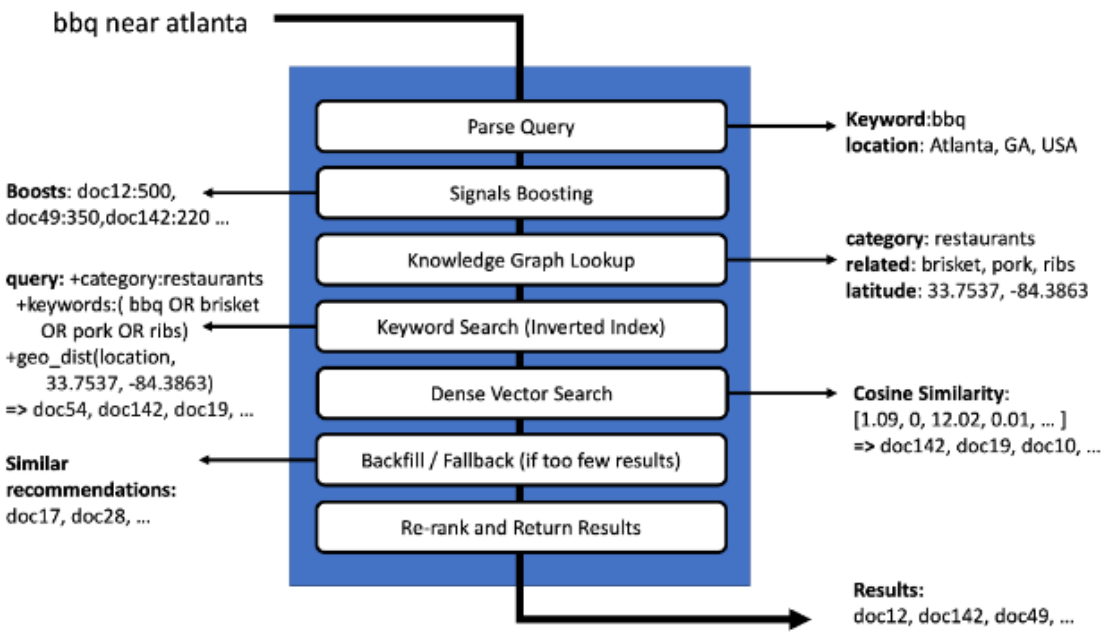
- **Score Close to +1.0**: Terms are highly related. For instance, "Advil" might have a high relatedness score in the context of "pain" if it appears in pain-related documents much more often than in random documents.
- **Score Close to -1.0**: Terms are highly unrelated, indicating they appear in divergent contexts.
- **Score Close to 0**: Terms are not semantically related or occur together at a frequency close to what is expected by chance (e.g., stopwords).

6. Using context to learn domain-specific language.
   Not sure if useful for the patent case

7. Interpreting query intent through semantic search

➔ Basic content to discuss when building a AIPS system:

- Pipelines -> index docs (parse + analyze + store), interpret and rank as stages to easily swap, and re-order them to experiment the best order of stages.
- Models -> LLM (chapters 13-14), learning rank model (chapters 10-12), signal boosting model (chapter 8) or knowledge graphs (synonyms, misspellings, ontologies and so on, chapter 5-6)
- Conditional Fallbacks -> a fallback model is a keyword-based model that can handle any query imperfectly, then you add complexity through layers



- ➔ 3 different approaches commonly used to interpret individual keywords and relate them: keyword search (BM-25) + knowledge graph search (semantic knowledge graphs) + dense vector search (cosine similarity on nearest neighborhood of vectors)
- ➔ (7.2) Indexing & searching on local reviews dataset: interesting but not necessary.
- ➔ Query interpretation pipelines usually follow -> parsing the query for semantic search + enriching it + transforming it+ searching with the semantically enhanced query.
- ➔ Chapter 7 not finished, I jump straight to Chapter 13

13. Semantic search with dense vectors + LLM

- ➔ Transformers we'll be explained in the chapter.
- ➔ We'll see techniques of how to get embeddings from text.
- ➔ How did you, as a baby, child, teen, and beyond, learn the meaning of words? You were told, and you consumed knowledge and its representation! People who taught you already had this knowledge and the power to express it. Aside from someone pointing out a cat and saying "kitty" to you, you also watched movies and videos, and then moved on to literature and instructional material. You read books, blogs, periodicals, and letters. Through all this and more, you incorporated the knowledge into yourself, creating in your brain a dense representation of

concepts and how they relate to one another and allow you to reason.

➔ We will process documents and get the embeddings and store them in some kind of embedding index. Then, at search time, we will process the query to get embeddings, and use those query embeddings to search the indexed document embeddings. The embeddings for both (documents and queries) exist in the same vector space. This allows us to effectively compare the vector of a query, to the vectors of all the documents in the content we want to search. The document vectors that are the most similar to the query vector are referred to as nearest neighbors.

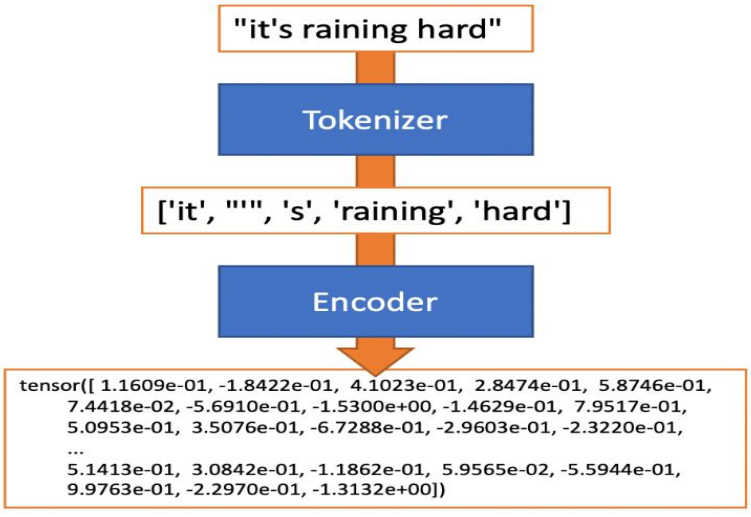
$$\text{Vector Similarity } (a, b): \quad \cos(\theta) = \frac{a \cdot b}{|a| \times |b|}$$

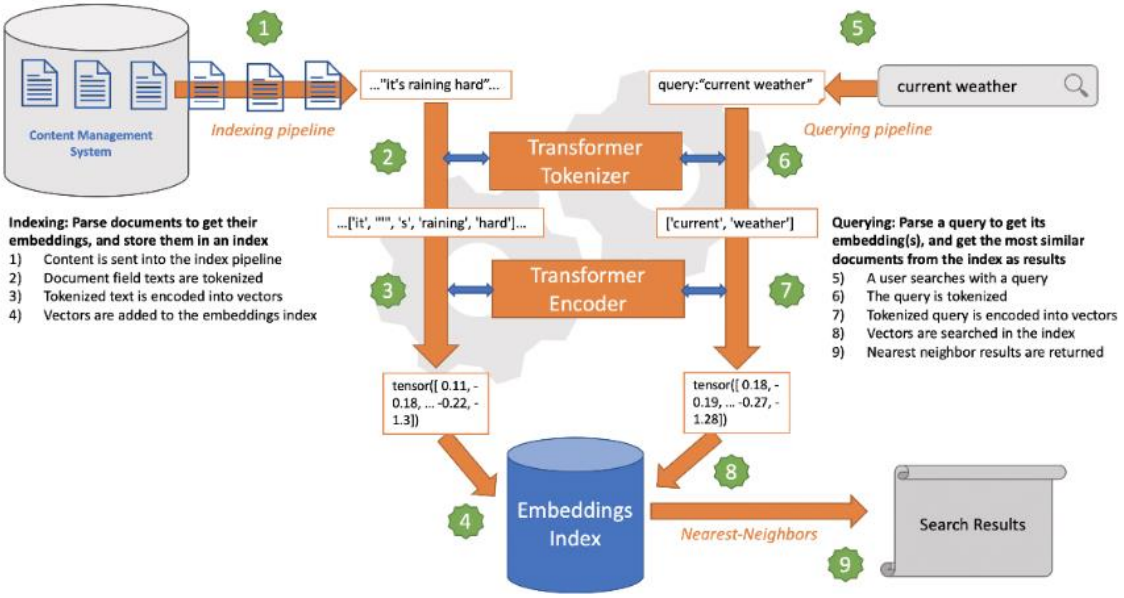
➔ Basically, we need to understand how to get the embeddings (the best possible one) from the content + queries from user, and then compare with vector similarity or any other similar distance measurement, the index of embeddings of the content with the query to find the nearest neighbors.

➔ We'll use Transformer Encoders to get the embeddings.

Transformers are a class of deep neural network architectures, that are optimized towards encoding meaning as vectors, and decoding the meaning into a transformed representation of that meaning.

➔ The following picture explains the idea.





- ➔ Even if we were willing to calculate cosine similarities for all our documents, this will just get slower and slower as we scale to millions of documents, so we ideally would only score documents which have a high chance of being similar.
- ➔ We can accomplish this goal by performing what is known as an approximate-nearest-neighbor (ANN) search. Using ANN search will efficiently give us the most closely related concepts when given a term, without the overhead of calculating embedding similarities across the entire dictionary.
- ➔ To implement our ANN search, we will be using an index-time strategy to store searchable content vectors ahead of time in a specialized data structure, we will use Hierarchical Navigable Small World (HNSW) graphs to index and query our dense vectors.
- ➔ HNSW will cluster similar vectors together as it builds the index. Navigable Small World graphs work by organizing data into neighborhoods and connecting the neighborhoods with one another with probable relationships. When a dense vector is being indexed, the most appropriate neighborhood and its potential connections are identified and stored in the graph data structure -> 13.5.4.
- ➔