AI Powered Search

Trey Grainger

MEAP

MANNING

MEAP Edition
Manning Early Access Program
AI-Powered Search
Version 8

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *AI-Powered Search*!

This book teaches you the knowledge and skills you need to deliver a highly-relevant search application that is able to automatically become more intelligent through continuous feedback loops that learn from every content update and user interaction.

As you can imagine given that goal, this is not an "introduction to search" book. In order to get the most out of this book, you should already be familiar with the core capabilities of modern search engines (inverted indices, relevance ranking, faceting, query parsing, text analysis, and so on) through experience with a technology like Apache Solr, Elasticsearch, or Apache Lucene. If you need to come up to speed quickly, *Solr in Action* (which I also wrote) provides you with all the search background necessary to dive head-first into *AI-Powered Search*.

Additionally, most of the examples in this book are in Python or JavaScript. You don't need to be an expert in either of these languages, but you should have some programming experience and be able to read and understand the examples.

Over my career, I've had the opportunity to dive deep into semantic search, recommendations, behavioral signals processing, learning to rank, and many other AI-powered search capabilities, publishing cutting-edge research in top journals and conferences and, more importantly, delivering working software at massive scale. As Founder of Searchkernel and as Lucidworks' former Chief Algorithms Officer and SVP of Engineering, I've also helped deliver many of these capabilities to hundreds of the most innovative companies in the world to help them power search experiences you probably use every single day.

In this book, I distill that experience into a practical guide to help you take your search applications to the next level, enabling those applications to continually learn (content-understanding, user-understanding, and domain-understanding) and improve in order to deliver optimally-relevant experiences with each and every user interaction. I'm working steadily on the book, and readers should expect a new chapter to arrive about every 1 to 2 months.

By purchasing the MEAP of *AI-Powered Search*, you gain early access to written chapters, and well as the ability to provide input into what goes into the book as it is being written. If you have any comments or questions along the way, please direct them to Manning's liveBook Discussion Forum for the book.

I would greatly appreciate your feedback and suggestions, as they will be invaluable toward making this book all it can be. Thanks again for purchasing the MEAP, thank you in advance for your input, and best wishes as you begin putting *AI-Powered Search* into practice!

—Trey Grainger

brief contents

1 Introducing AI-powered search

This chapter covers:

- The need for AI-powered search
- The dimensions of user intent
- Foundational technologies for building AI-powered Search
- How AI-powered search works

The search box has rapidly become the default user interface for interacting with data in most modern applications. If you think of every major app or website you visit on a daily basis, one of the first things you likely do on each visit is type or speak a query in order to find the content or actions most relevant to you in that moment.

Even in scenarios where you are not explicitly searching, you may instead be consuming streams of content customized for your particular tastes and interests. Whether these be video recommendations, items for purchase, e-mails sorted by priority or recency, news articles, or other content, you are likely still looking at filtered or ranked results and given the option to either page through or explictly filter the content with your own query.

Whereas the phrase "search engine" to most people brings up thoughts of a website like Google, Baidu, or Bing, that enables queries based upon a crawl of the entire public internet, the reality is that search is now ubiquitous - it is a tool present and available in nearly all of our digital interactions every day across the numerous websites and applications we use.

Furthermore, while not too long ago the expected response from a search box may have been simply returning "ten blue links" - a list of ranked documents for a user to investigate to find further information in response to their query - expectations for the intelligence-level of search technologies have sky-rocketed in recent years.

Today's search capabilities are expected to be:

- *Domain-aware*: understanding the entities, terminology, categories, and attributes of each specific use case and corpus of documents, not just leveraging generic statitistics on strings of text.
- *Contextual & Personalized*: able to take user context (location, last search, profile, previous interactions, user recommendations, and user classification), query context (other keywords, similar searches), and domain context (inventory, business rules, domain-specific terminology) in order to better understand user intent.
- *Conversational*: able to interact in natural language and guide users through a multi-step discovery process while learning and remembering relevant new information along the way.
- *Multi-modal*: able to resolve text queries, voice queries, search using images or video, or even monitor for events and send event-based pushed notifications.
- *Intelligent*: able to deliver predictive type-ahead, to understand what users mean (spelling correction, phrase and attribute detection, intent classification, conceptual searching, and so on) to deliver the right answers at the right time and to be constantly getting smarter.
- *Assistive*: moving beyond just delivery of links and information to delivering answers and action.

The goal of AI-powered search is to leverage automated learning techniques to deliver on these desired capabilities. While many organizations start with basic text search and spend many years trying to manually optimize synonyms lists, business rules, ontologies, field weights, and countless other aspects of their search configuration, some are beginning to realize that most of this process can actually be automated.

This book is an example-driven guide through the cutting-edge in machine learning algorithms applicable for building intelligent search systems. We'll not only walk through key concepts, but will also provide reusable code examples to cover data collection and processing techniques, as well as the self-learning query interpretation and relevance strategies employed to deliver AI-powered search capabilities across today's leading organizations - hopefully soon to include your own!

1.1 Searching for User Intent

In order to deliver AI-powered search, it's important that we establish up-front a cohesive understanding of the many dimensions involved in interpreting user intent and subsequently returning content matching that intent. Within the field of information retrieval, search engines and recommendation engines are the two most popular technologies employed in an attempt to deliver users the content needed to satisfy their information need. Many organizations think of search engines and recommendation engines as separate technologies solving different use cases, and indeed it is often the case that different teams within the same organization - often with different skillsets - work independently on separate search engines and recommendation engines. In this section, we'll dive into why separating search and recommendations into separate silos (independent functions and teams) can often lead to less than ideal outcomes.

1.1.1 Search Engines

A search engine is typically thought of as a technology for explictly entering queries and receiving a response. It is usually exposed to end users through some sort of text box into which a user can enter keywords or full natural-language questions, and the results are usually returned in a list alongside additional filtering options that enable further refinement of the initial query. Using this mechanism, search is leveraged as a tool for directed discovery of relevant content. Whenever a user is finished with their search session, however, they can usually issue a new query and start with a blank slate, wiping away any context from the previous search.

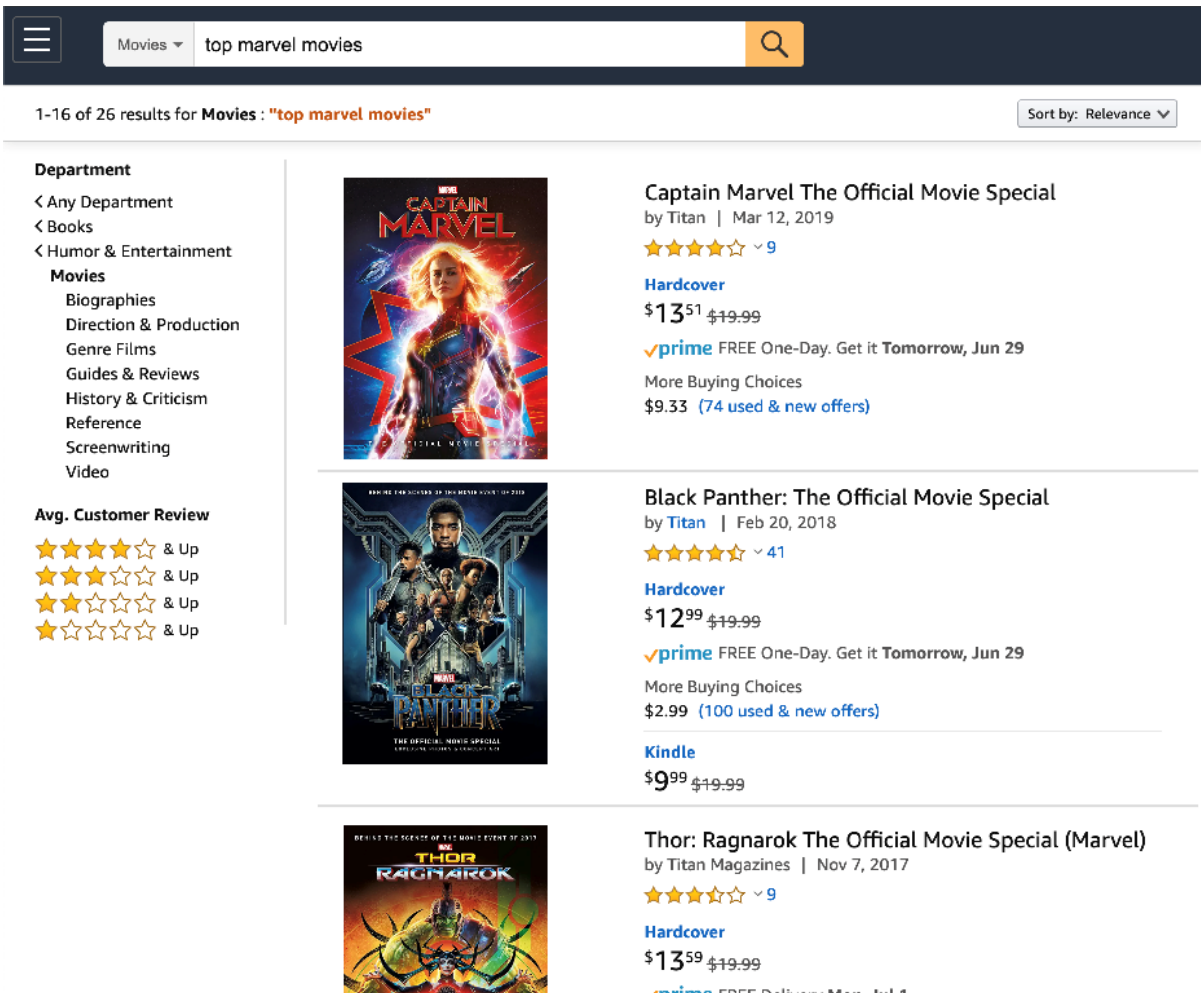


Figure 1.1 A typical search experience, with a user entering a query and seeing search results with filtering options to support further refinement of search results

Within the software engineering world, a search engine is one of the most cross-functional kinds of systems in existence. Most underlying search engine technology is designed to operate in a massively scalable way - scaling to millions, billions, or in a select cases trillions of documents, handling enormous volumes of concurrent queries, and delivering search results in hundreds of milliseconds or often less. In many cases, real-time data ingestion and near-real-searching on newly ingested data is also required, and all of this must be massively parallelizable across numerous servers in order to scale out to meet such high performance requirements.

Needless to say, engineering the above kind of system requires strong back-end developers with a solid understanding of distributed systems, concurrency, data structures, operating systems, and high-performance computing.

Search engines also require substantial work building search specific data structures like an inverted index, an understanding of linear algebra and vector similarity scoring, experience with text analysis and natural language processing, and knowledge of numerous search-specific kinds of data models and capabilities (spell checking, autosuggest, faceting, text highlighting, and so on).

Further, as relevancy becomes more important, search teams often employ relevance engineers who live and breath the domain of the their company, and who think in terms of ranking models, A/B testing, click-models, feature engineering, and query interpretation and intent discovery.

Of course, there are product managers driving requirements, data analysts assessing search quality and improvements, DevOps engineers running and scaling the search platform, and numerous other supporting roles surrounding a production search engine implementation.

It is rare to find someone with multiple of these skillsets, let alone all of them, making search a highly cross-functional discipline. In order for a search engine to fully interpret user intent, however, being able to combine a thorough understanding of your content, your users, and your domain is critical. We'll revisit why this is important after briefly discussing the related topic of recommendation engines.

1.1.2 Recommendations Engines

Whereas search engines are traditionally thought of as technology that requires explicit user-driven queries, most people in contrast think of recommendation engines as systems which accept no user input and only push out content to users based upon what it knows about them and believes best matches their computed interests. Both of these are oversimplifications, as we'll see in section 1.1.3, but they are nevertheless the prevailing ways in which search engines and recommendations are commonly distinguished. If you routinely visit Amazon.com or any other major ecommerce website, you are no doubt familiar with recommendation engines sections stating that "based on your interest in this item, you may also like …" or otherwise just recommending a list of items based upon your collective browsing and purchase history, similar to Figure 1.2. These recommendations often drive significant revenue for companies, and they help customers discover relevant, personalized, and related content that often complements what they were looking for explicitly.

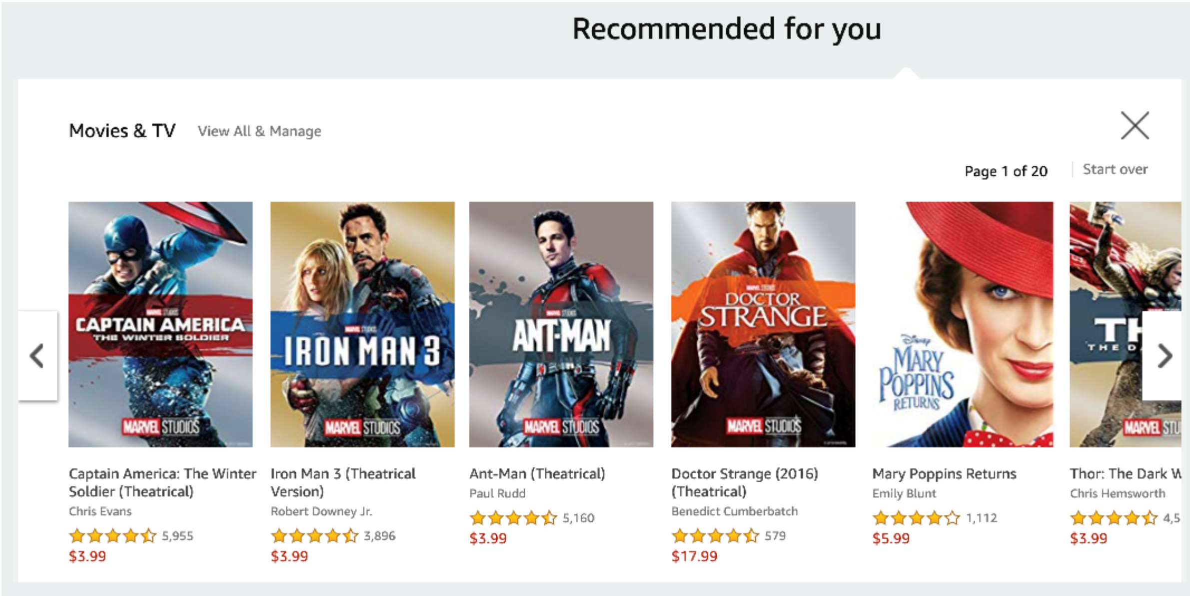


Figure 1.2 Recommendations based upon purchase patterns of user expressing interest in similar items

Recommendations engines, also commonly referred to as recommendation systems, come in many shapes and sizes. They employ algorithms which can take in inputs (user preferences, user behavior, content, and so on) and leverage those inputs to automatically match the most relevant content. Recommendation algorithms can roughly be divided into three categories: Content-based Recommenders, Behavior-based Recommenders, and Multi-modal Recommenders.

CONTENT-BASED RECOMMENDERS

These algorithms recommend new content based upon attributes shared between different entities (often between users and items, between items and items, or between users and users). For example, imagine a job search website. Jobs may have properties on them like "job title", "industry", "salary range", "years of experience", and "skills", and users will also have similar attributes on their profile or resume/CV. Based upon this, a content-based recommendation algorithm can figure out which of these features matter the most, and can then rank the best matching jobs for any given user based upon how well that job matches the user’s desired attributes. This is what’s known as an user-item (or "user to item") recommender.

Similarly, if a user likes a particular job, it is possible to leverage this same process to recommend similar other jobs based upon how well those jobs match the attributes for the first job that is known to be good. This type of recommendation is popular on product details pages, where a user is already looking at an item and it may be desirable to help them explore additional related items. This kind of a recommendation algorithm is known as an item-item (or "item to item") recommender.

Figure 1.3 demonstrates how a content-based recommender might leverage attributes about items with which a user has previously interacted in order to match similar items for that user. In this case, our user viewed the "detergent" product, and then the "fabric softener" and "dryer sheets" products are recommended based upon them matching within the same category field (the "laundry" category) and containing similar text to the "detergent" product within their product descriptions.

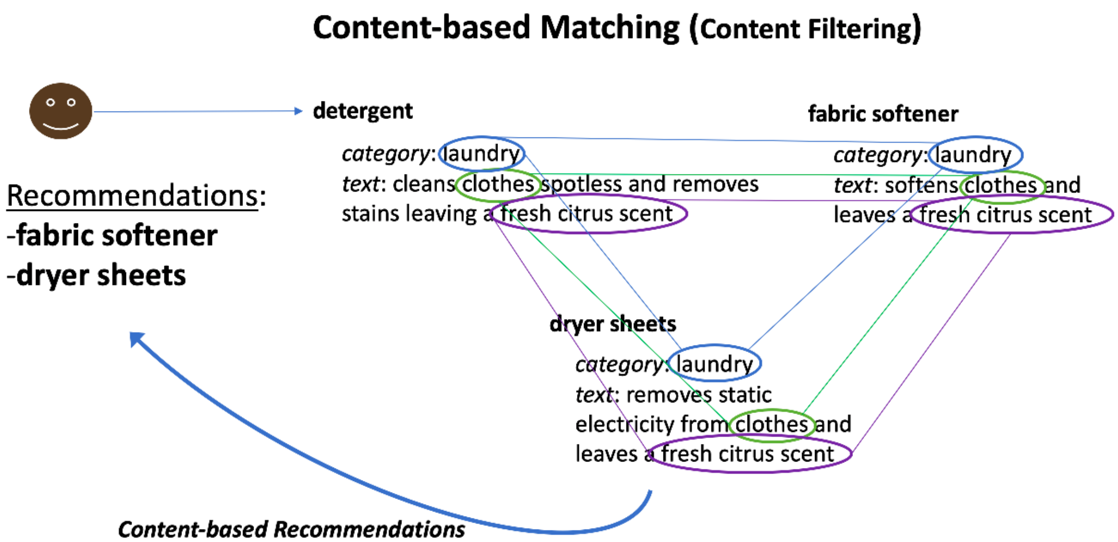


Figure 1.3 Content-based recommendations based upon matching attributes of an item of interest to a user, such as categories and text keywords.

It is also possible to match users to other users or really any entity to any other entity. In the context of content-based recommenders, all recommendations can be seen as "item-item" recommendations, where each item is just an arbitrary entity that shares attributes with the other entities being recommended.

BEHAVIOR-BASED RECOMMENDERS

These algorithms leverage user interactions with items (documents) to discover similar patterns of interest among groups of items. This process is called *collaborative filtering*, referring to the use of a multiple-person (collaborative) voting process to filter matches to those demonstrating the highest similarity, as measured by how many overlapping users interacted with the same items. The intuition here is that similar users (i.e. those with the similar preferences) tend to interact with the same items, and that when users interact with multiple items they are more likely to be looking for and interacting with similar items as opposed to unrelated items.

One amazing characteristic of collaborative filter algorithms is that they fully crowd-source the relevance scoring process to your end users. In fact, features of the items themselves (name, brand, colors, text, and so on) are not needed - all that is required is a unique ID for each item, and knowledge of which users interacted with which items. Further, the more users and interactions you have, the smarter these algorithms tend to get, because you have more people continually voting and informing your scoring algorithm. This often leads to collaborative filtering algorithms significantly outperforming content-based algorithms.

Figure 1.4 demonstrates how overlapping behavioral signals between mutliple users can be used to drive collaborative recommendations

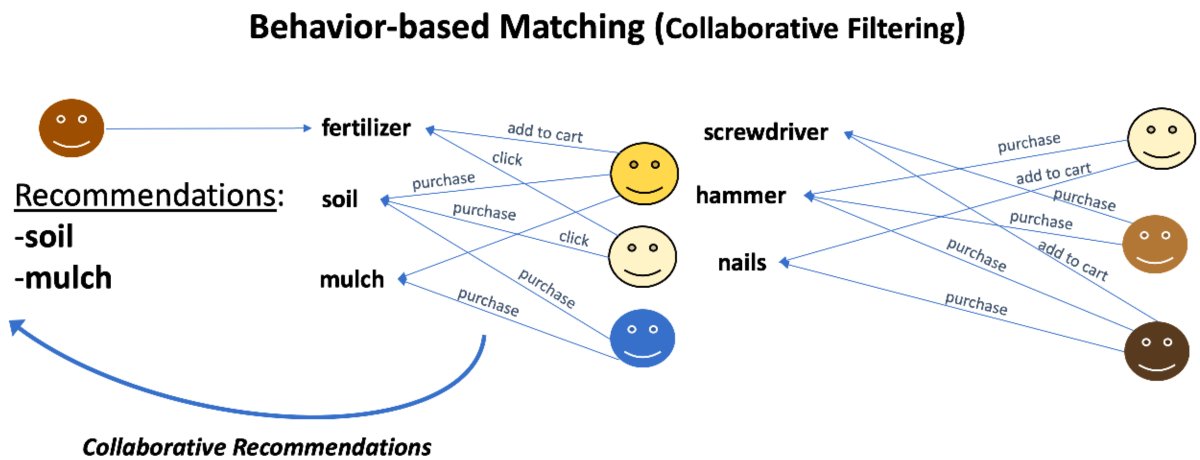


Figure 1.4 Recommendations based upon collaborative filtering, a technique leveraging the overlap between behavioral signals across multiple users.

Unfortunately, the same dependence upon user behavioral signals that makes collaborative filtering so powerful also turns out to be their Achilles' heel. What happens when there are only a few interactions with a particular item — or possibly none at all? The answer is that the item either never gets recommended (when there are no signals), or otherwise will likely to generate poor recommendations or show up as a bad match for other items (when there are few signals). This situation is known as the *Cold Start Problem*, and is a major challenge for behavior-based recommenders.

MULTI-MODAL RECOMMENDERS

Sometimes, you can have your cake and eat it, too. Such is the case with multi-modal recommenders, which attempt to combine the best of both the content-based and behavior-based recommender approaches. Since collaborative filtering tends to work best for items with many signals, but poorly when few or no signals are present, it is most effective to use both content-based features as a baseline and then to layer a model based on behavioral signals on top. This way, if few signals are present, the content-based matcher will still return results, while if there are many signals, then the collaborative filtering matches will take greater prominence. We'll cover methods for combining approaches in chapter 6, but it is easy to infer already that including some results from both approaches can give you the best of both worlds: high quality crowd-sourced matching, while avoiding the cold start problem for newer and less-well-discovered content. Figure 1.5 demonstrates how this can work in practice.

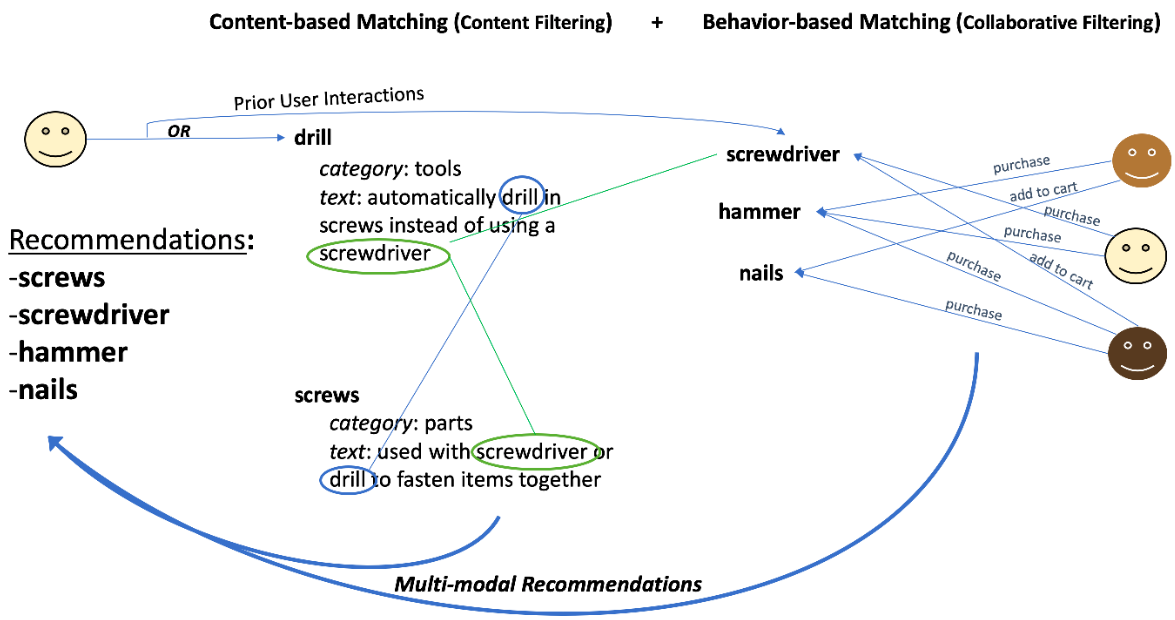


Figure 1.5 Multi-modal recommendations, which combine both content-based matching and collaborative filtering into a hybrid matching algorithm.

You can see in Figure 1.5 that the user could interact with either the drill (which has no signals and can thus only be used in recommendations through a content-based matcher) or the screwdriver (which has previous signals from other users, as well as content), and the user would receive recommendations in both cases. This provides the benefit that signals-based collaborative filtering can be used, while also enabling content-based matching for items where insufficient signals are currently available, such as is the case with the drill in this example. This kind of multi-modal recommendation algorithm provides significant flexibility and advantage over only leveraging content-based or behavior-based recommendations alone.

1.1.3 The Information Retrieval Continuum

We just finished discussing two different kinds of information retrieval systems: Search Engines and Recommendation Engines. The key differentiating factor between the two was that search engines are typically guided by users and match their explicitly-entered queries, whereas recommendation engines typically accept no direct user input and instead recommend — based upon already known or inferred knowledge — what a user may want to see next.

The reality, however, is that these two kinds of systems are really just two sides of the same coin. The goal, in both cases, is to understand what a user is looking for, and to deliver relevant results to meet that user's information need. In this section, we'll discuss the broad spectrum of capabilities that lie within the information retrieval continnum between search and recommendation systems

PERSONALIZED SEARCH

Let’s imagine a restaurants search engine. Our user, Michelle is on her phone in New York at lunch time and she types in a keyword search for "steamed bagels". She sees top rated steamed bagel shops in Greenville, South Carolina (USA), Columbus, Ohio (USA), and London (UK).

What’s wrong with these search results? Well, in this case, the answer is fairly obvious — Michelle is looking for lunch in New York, but the search engine is showing her results hundreds to thousands of miles away. But Michelle never *told* the search engine she only wanted to see results in New York, nor did she tell the search engine that she was looking for a lunch place close by because she wants to eat now…

Consider another scenario - Michelle is at the airport after a long flight and she searches her phone for "driver". The top results that come back are for a golf club for hitting the ball off a tee, followed by a link to printer drivers, followed by a screw driver. If the search engine knows Michelle’s location, shouln’t it be able to infer her intended meaning?

Using our job search example from earlier, let’s assume Michelle goes to her favorite job search engine and types in "nursing jobs". Similar to our restaurant example earlier, wouldn’t it be ideal if nursing jobs in New York showed up at the top of the list? What if she later types in "jobs in Seattle"? Wouldn’t it be ideal if — instead of seeing random jobs in Seattle (doctor, engineer, chef, etc.) — nursing jobs now showed up at the top of the list, since the engine previously learned that she is a nurse?

Each of the above are examples of personalized search - the process of combining both explicit user input *and* an implicit understanding of each user’s specific intent and preferences, into a query that can return a set of results specifically catering to that user. Doing this well is a tricky subject, as you have to carefully balance leveraging your understanding of the user without overriding anything for which they explicitly want to query. We’ll dive into how to gracefully balance these concerns later in chapter 6.

USER-GUIDED RECOMMENDATIONS

Just as it is possible to sprinkle in implicit and user-specific attributes into a search to generate personalized search, it is also possible to enable user-guided recommendations.

It is becoming increasingly more common for recommendation engines to allow users to see and edit their recommendation preferences. These preferences usually include a list of items the user has interacted with before, such as watched movies and ratings for a movie recommendation engine, or a list of previously viewed or purchased items for an ecommerce website. Across a wide array of use cases, these preferences could also include things like clothing sizes, brand affinities, favorite colors, prefered local store, favorite menu items, desired job titles and skills, prefered salary ranges, etc. In essence, these recommendation preference make up a user profile -

they define what is known about a customer, and the more control you can give a user to see, adjust, and improve this profile, the better you'll be able to understand your users and the happier they'll likely be.

SEARCH VS. RECOMMENDATIONS: THE FALSE DICHOTOMY

We've seen that when trying to find content for your end users, that personalization profile information can either be ignored (traditional keyword search), used implicity along with other explicit user input (personalized search), used explicitly with the ability for a user to adjust (user-guided recommendations), or used explicitly with no ability for a user to adjust (traditional recommendations). Figure 1.6 shows this personalization spectrum.

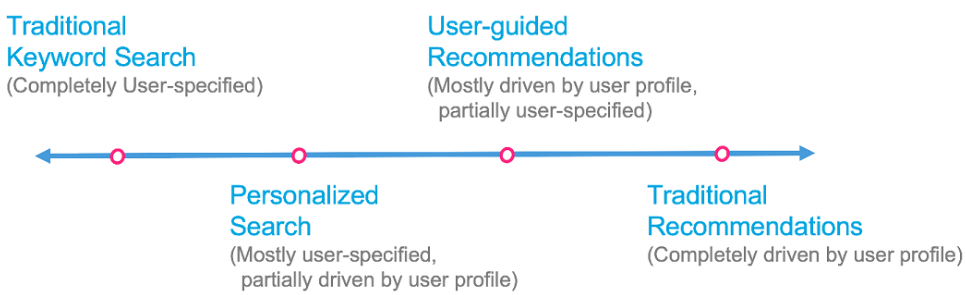


Figure 1.6 The personalization spectrum, showing traditional keyword search and traditional recommendations as simply two ends of a larger continuum

While the two ends of this personalization spectrum (traditional keyword search, and traditional recommendations) represent the extremes, they are also the two most common. Unfortunately, one of the biggest mistakes that I see in many organizations is the building of teams and organizations around the belief that search and recommendations are separate problems. This often leads to data science teams building complicated personalization and segmentation models that can only recommend content and can't do search at all, and for engineering teams to build large-scale keyword matching engines that can't take easy advantage of the robust personalization models built by the recommendations teams. More often than not, the recommendation teams are staffed by Data Scientists with minimal information retrieval background, and the search teams are often staffed by engineers with minimal data science background. Due to Conway's Law ("organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations."), this ultimately results in challenges solving problems along this personalization spectrum that need the best from both teams. In this scenario, more often than not the search and recommendation engines will naturally evolve into separate systems that can't as easily maximize for relevance needs across the full personalization spectrum. In this book, we'll focus on the shared techniques that enable search to become smarter and for recommendations to become more flexible through a unified approach, as an AI-powered search platform needs to be able to continuously learn from both your users and your content and enable your users to do the same.

1.1.4 Semantic Search and Knowledge Graphs

While we've presented search and recommendations as a personalization spectrum in Figure 1.6, with personalized search and user-guided recommendations in-between, there's still one more dimension that is critical for building a good AI-powered Search system — a deep understanding of the given domain. It's not enough to match on keywords and to recommend content based upon how users collectively interact with different documents. To build a truly smart search system, it is important for the engine to learn as much as it can about the domain. This includes learning all of the important domain-specific phrases, synonyms, and related terms, as well as identifying entities in documents and queries, generating a knowledge graphs that relate those entities, disambiguating the many nuanced meanings represented by domain-specific terminology, and ultimately being able to effectively parse, interpret, and conceptually match the nuanced intent of users within your domain. Figure 1.7 shows an example of a semantic parsing of a query, with the goal being to search for "things" (known entities) instead of "strings" (just text matching).

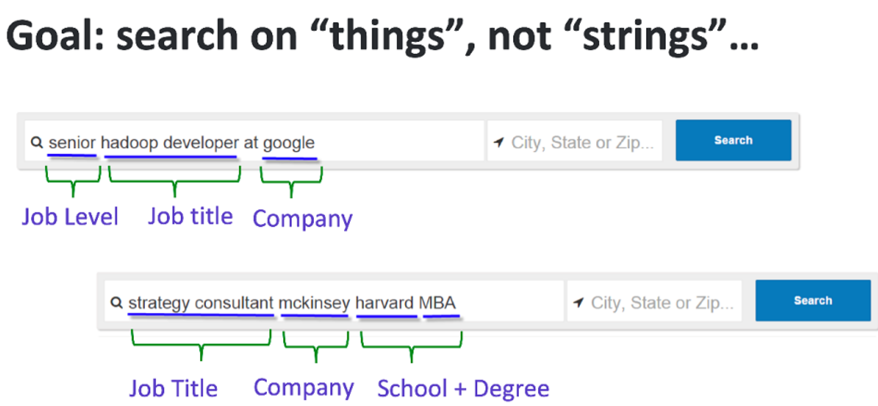


Figure 1.7 Semantic parsing of a query, demonstrating an understanding of the entities ("things") represented by query terms

Many companies have spent considerable money building out large teams to manually create dictionaries and knowledge graphs in an attempt to understand the relationships between entities in their users' queries in order to make their search smarter. This book will focus on a more scalable approach, however - building an AI-powered search engine that can automatically learn all of the above on a continuous basis. We'll dive deep into the topic of automatically generating this domain-specific intelligence when we cover Semantic Search further in chapter 5. We'll also dive into additional emerging techniques for conceptual search, such as dense vector search and neural search in chapter 9.

1.1.5 Understanding the Dimensions of User Intent

We've discussed the important role of traditional keyword search, of recommendations, and of the personalization spectrum in-between. We also discussed the need for Semantic Search (leveraging a Knowledge Graph) to be able to provide domain-specific understanding of your content and your user's queries. All of these are key pillars of a singular, larger goal, however: to fully understand user intent. Figure 1.8 provides a comprehensive model demonstrating the interplay between each of these key pillars of user intent.

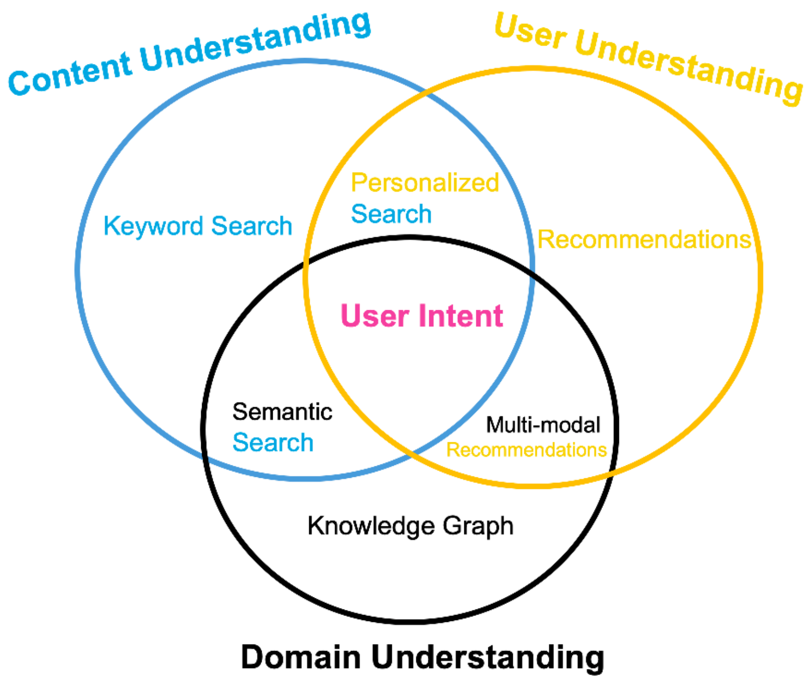


Figure 1.8 The dimensions of user intent, a combination of content understanding, user understanding, and domain understanding.

The top-left circle in Figure 1.8 represents "content understanding" - the ability to find any arbitrary content leveraging keywords and matching on attributes. The top-right circle represents "user understanding" - the ability to understand each user's specific preferences and leverage those to return more personalized results. Finally, the lower circle represents "domain understanding" - the ability to interpret words, phrases, concepts, entities, and nuanced interpretations and relationships between each of these within your own domain-specific context.

A query only in the "content understanding" circle represents traditional *keyword search*, enabling matching on keywords but without leveraging any domain- or user-specific context. A query only in the "user understanding" circle would be generated recommendations from collaborative filtering with no ability to override the inputs and no understanding of the domain or content underlying documents. A query only in the "domain understanding" bucket might be a structured query only on known tags, categories, or entities, or even a browse-like interface that enabled explorations of a *knowledge graph* of these domain-specific entities and their relationships, but without any user specific personalization or ability to find arbitrary terms, phrases, and content.

When traditional keyword search and recommendations overlap, we get *personalized search* or guided recommendations (discussed earlier). When traditional keyword search and knowledge graphs overlap, we get *semantic search* (also discussed earlier): a smart, domain-specific search experience. Finally, when recommendations and knowledge graphs overlap we get smarter *multi-modal recommendations* that not only match on the crowd-sourced interactions of users across similar documents, but also on a domain-specific understanding of the important attributes of those documents.

The holy grail for AI-powered search, however, is to harness the intersection of all three categories: personalized search, semantic search, and multi-modal recommendations. That is to say, in order to truly understand user intent, we need 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. AI-powered search starts with the three pillars of user intent (content, domain, and user), and then leverages intelligent algorithms to constantly learn and improve in each of these areas automatically. This includes techniques like automatically learning ranking criteria (see chapter 10), automatically learning user preferences (see chapter 8), and automatically learning knowledge graphs of the represented domain (see chapter 5). At the end of the day, the combination of each of these approaches - and balancing them appropriately against each other - provides the key to optimal understanding of users and their query intent, which is the end goals of our AI-powered search system.

1.2 Key Technologies for AI-powered search

This book is a guide for readers to take their search applications to the next level and build highly-relevant, self-learning search systems. In order to get to the "next level", however, this assumes readers already have experience with or are already familiar with key concepts behind traditional keyword search engines and modern data processing platforms.

While you could leverage numerous other open source or commercial search technologies to power your search engine, none of them carry the same industry mind-share, internal transparency, or flexibility as those we’ve included. While the techniques in this book are broadly applicable, we have chosen to focus on the top Lucene-based search technologies and data processing frameworks, with which we expect most readers will be familiar.

In this section, we’ll provide an overview of these key technologies that we will leverage or refer to throughout this book.

1.2.1 Apache Lucene: the core search library powering Apache Solr and Elasticsearch

Apache Lucene is an open-source Java framework for building rich search applications. Lucene does not provide a search "engine", but instead provides extensive java libraries that you can integrate into your own software to build inverted indices, perform text analysis and tokenization, convert data into efficient data types for search, and manage both the low-level operations and the functions necessary to perform most search operations (query parsing, faceting, index updates and segment merges, indexing and querying, and so on).

Lucene is internally used in more search applications than any other search technology today, but is typically not used directly. That is to say, whereas Lucene is a search library, other software projects have evolved on top of Lucene to expose out fully-featured search servers that can be run as scalable distributed systems across large computing clusters (Lucene, in contrast, runs locally on each node only). These search servers expose APIs (REST and other protocols), as well as client libraries that are ready to deploy and use out of the box. Though some people still build applications directly on top of Lucene, most organizations have switched to platforms leveraging one of the two most popular "open source" search engine projects (both powered internally by Lucene): Apache Solr or Elasticsearch.

1.2.2 Apache Solr: the open sourced, community-driven, relevance-focused search engine

Solr is also open source and is actually part of the Apache Lucene project, so it is likewise owned and maintained by the Apache Software Foundation. The source code for the combined "Lucene-Solr" project is found within the same public Git repository, and the same developers are officially committers on both projects. Whereas Lucene provides the core libraries and data structures to build a search engine, Solr provides an out of the box search server brimming with production-ready capabilities.

While we won’t go deep into all the capabilities of Solr in this book (*Solr in Action* took 638 pages, and that was back in 2014!), suffice it to say that 90% of the Fortune 500 use Solr to power some of the most advanced search applications in the world, and it is very feature-rich and well-suited for building an AI-powered search application.

1.2.3 Elasticsearch: the most used, anaytics-focused, full text search engine

Elasticsearch is the most downloaded and most well-known search engine technology today by developers, apart from Apache Lucene, of course, which internally powers both Solr and Elasticsearch. Elasticsearch has overtaken Solr over time in terms of number of deployments and overall developer mind share, largely due to the popularity of the ELK stack (Elasticsearch + Logstash + Kibana) being the industry-standard for ingesting, searching, and analyzing application logs. While log and event search is the main use case for Elasticsearch, it is also a very capable text search engine, providing close to full feature parity with Solr (plus some additional features unavailable in Solr) for traditional search use cases.

Elasticsearch is partially open sourced. It is owned and developed exclusively by a publicly-traded company called Elastic, with the code base including a hybrid of open source (ASL 2.0 licensed) code and commercially-licensable code (Elastic License) that requires a paid license to modify and in many cases requires a paid license to use.

Due to Elasticsearch's popularity and mindshare and relative ease of use, it is often the go-to search application for new developers getting started with search.

1.2.4 Lucidworks Fusion: the out-of-the-box AI-powered search Engine

Whereas Apache Solr and Elasticsearch provide solid core search engine capabilities, neither of them provide many of the AI-powered search capabilities and pipelines found in this book. Commercial vendors, like Lucidworks, can deliver end-to-end AI-powered search capabilities out of the box, whereas Solr and Elasticsearch are core matching engines but lack much of the extended infrastructure needed for AI-powered search. Lucidworks delivers Fusion using an "open core" model, where the central technologies Fusion relies upon (Apache Solr, Apache Spark) are completely open source, and customers thus have the full ability to see, debug, and modify their systems without being stuck with a black box system that restricts access to the core technologies, algorithms, and code.

NOTE Disclaimer: please note that while my comments and recommendations around Lucidwork's Fusion are sincere, I am also an employee of Lucidworks who helped build it and therefore you should assume some bias and do your own research. Leading third-party analysts (Gartner, Forrester, etc.) are a good source of information.

Lucidworks Fusion ships with most of the capabilities described in this book, giving you the opportunity to buy vs. build them all yourself. While this book is primarily focused on explaining how AI-powered search works under the covers and showing you how to implement it yourself, there's no doubt that leveraging a leading commercial vendor's product will get you a better results faster. Regardless of which path you choose, this book will cover many of the techniques used by products like Fusion, and will help Fusion customers better understand, tune, and improve their Fusion-powered search applications.

1.2.5 Apache Spark: the standard for large-scale data processing

Apache Spark is not a search engine, but it is today's go-to technology for large-scale data processing. In the late 2000's, Apache Hadoop was the technology of choice for cheap big data storage (HDFS) and distributed processing (Map/Reduce). Writing applications leveraging Hadoop's Map/Reduce framework, however, required many lines of fairly-involved code to perform even simple operations. Then, Apache Spark came along and faster, simpler, more user-friendly model for performing distributed operations across an HDFS cluster, and it rapidly displaced Hadoop's map/reduce implementation as the platform of choice for doing large-scale distributed data processing.

Spark also introduced the generic idea of data frames, which can pull from any arbitrary data source (not just HDFS). This means that even HDFS can be swapped out, as needed, with faster data platforms. Apache Solr and Elasticsearch, for example, can both be used as data sources and data targets of Spark jobs, enabling Spark to be run directly on top of your search platform to process and enhance your search data (content augmentation, user behavior analysis, machine learning model generation, and so on) and to take advantage of the speed of leveraging your search index when appropriate.

1.2.6 Strategy for this book

Though this book is a practical guide to delivering AI-powered search regardless of the underlying search engine, it is unfortunately not practical to create duplicate examples to target every engine readers may want to use, so this book will instead focus primarily on the technologies that best align with our learning goals.

Spark is the natural choice for our data processing engine for offline jobs. As discussed, it has largely replaced Hadoop's Map/Reduce in industry, and it can also directly pull and push to all of our discussed core search engines. We'll make heavy use of Spark for data processing and machine learning jobs to deliver our AI-powered search applications.

For our core search engine, however, we'll need to make a choice among the four search engine technologies under consideration: Apache Lucene, Apache Solr, Elastic's Elasticsearch, and Lucidworks Fusion. While the vast majority of this book will apply equally-well to all of these engines (and others not listed), we have decided to base most examples in this book on Apache Solr, with specific callouts to and examples for Elasticsearch and Fusion when and where appropriate.

The reasoning behind this is relatively straight-forward here. Leveraging Apache Lucene directly requires the most work in almost all situations, as you will have to build many "core" search server capabilities yourself, therefore it would bloat the book with unnecessary boilerplate code and low-level implementation discussions. Choosing Solr gives us all the needed Lucene capabilities in a ready-to-deploy software package.

On the opposite end of the spectrum, while Lucidworks Fusion is the most advanced and already contains many of the AI capabilities taught in this book (AI jobs, signals capture and processing, Machine Learning models, query pipelines, semantic search, content and user understanding, and so on), it also requires a commercial license and may not be an option for all readers. Since Fusion uses Solr as it's core matching engine, it means that by targeting Solr this book will still be directly applicable to Fusion users and will be able to provide them with a deeper understanding of what's going on under the covers.

Since Elasticsearch is currently the most popular, most downloaded traditional search engine, it would seem a natural target for this book. Unfortunately, since Elasticsearch's primary use case has historically been log and event search, it has trailed Solr on the development of key intelligent search features. Specifically, Solr provides a Text Tagger handler used for high-performance naive entity extraction, a Semantic Knowledge Graph used for dynamic concept expansion and relationship discovery between text phrases and entities, a statistical phrase identifier used for dynamic semantic parsing of queries, graph traversal capabilities for multi-level discovery of relationships within queries, and a Learning to Rank framework for training and deploying machine-learned relevance ranking models. All of these are building blocks for building AI-powered search, and since our goal in this book is to teach you the techniques to build an AI-powered search engine, it doesn't make sense to spend time reimplementing each of these core building blocks for Elasticsearch when they are already available out-of-the-box in Solr.

By choosing Solr, we also have the added benefit of leveraging a completely open source solution maintained by a software foundation with a diverse community of contributors (the Apache Software Foundation), as opposed to software controlled, developed, and licensed from commercial entities as a mix of open source and commercial code (both Fusion and Elasticsearch contain a hybrid of commercially-licensed and open source code). This guarantees that anything you use in this book is completely open source and reusable in your applications and continues to be into the future.

Regarding data processing, Solr maintains a streaming expression framework that rivals Sparks distributed processing capabilities in many ways. While we could leverage this framework in our examples to reduce dependencies, we'll instead choose to leverage Spark for most of our offline data crunching and machine learning tasks in order to keep our examples more generally-applicable across different search platforms. This will make all of the examples easily reusable for Elasticsearch, for example, since Spark can also talk to Elasticsearch.

Ultimately, our goal in this book is to teach you reusable techniques that you can apply regardless of your underlying search technology, and it will be much more expedient to accomplish this leveraging Solr's more robust set of freely available features here. I therefore expect Lucene, Fusion, and Elasticsearch users to all be able to easily follow along with the examples in this book and apply the concepts to their engine of choice.

1.3 Target Audience for AI-Powered Search

This book assumed certain prerequisite knowledge of the mechanics of a search engine. If you do not have prior experience working with Solr or Elasticsearch, you can come up to speed quickly by reading through the books *Solr in Action* or *Elasticsearch in Action*.

1.3.1 Targeted Skillsets and Occupations

This book is primarily targeted at search engineers, software engineers, and data scientists. The book will still provide relevant conceptual understanding of AI-powered search for product managers and business leaders who do not possess these skills, but for an engineer or data scientist to get the most out of this book and its code examples, they will need a basic understanding how search and relevancy work leveraging a technology like Solr or Elasticsearch.

1.3.2 System Requirements for Running Code Examples

In order to comfortably run all the examples in this book, you will need a Mac, Linux, or Windows computer, and we recommend a minimum of 8GB of RAM to be able to run through some of the more heavy-duty Spark jobs. You will need to ensure you have Java 8+ installed on your computer, but we will walk you through configuration of all other dependencies on an as-needed basis.

1.4 When to consider AI-powered search

AI-powered search, as can be seen from Figure 1.9, provides a much more sophisticated alternative to manual relevance tuning. For organizations that benefit greatly from better search relevancy and either invest or are thinking about investing significantly in their search platform, deploying AI-powered search should lead to significantly higher quality with significantly less effort in the long run.

That being said, installing Elasticsearch or Solr and ingesting documents is relatively easy, and if you only need basic keyword matching or are unable to invest the resources to build out and maintain an AI-powered search platform, then you might proceed cautiously with implementing too many of the techniques in this book, as it will definitely add more components and complexity to your search engine overall.

For organizations desparately needing to optimize their search relevance investment, however, the techniques in this book are an excellent guide in that endeavor.

1.5 How does AI-powered search work?

Now that we've laid out our end goal of matching user intent through content understanding, user understanding, and domain understanding, and we've laid out the key technology platforms we'll leverage and prerequisite knowledge you'll need to do pull this off, let's wrap up this chapter with an overview of what components are needed to actually deliver an AI-powered search platform. Search intelligence is not black and white with search either being basic or AI-powered. Instead, search intelligence typically matures along a predictable progression, as shown in Figure 1.9.

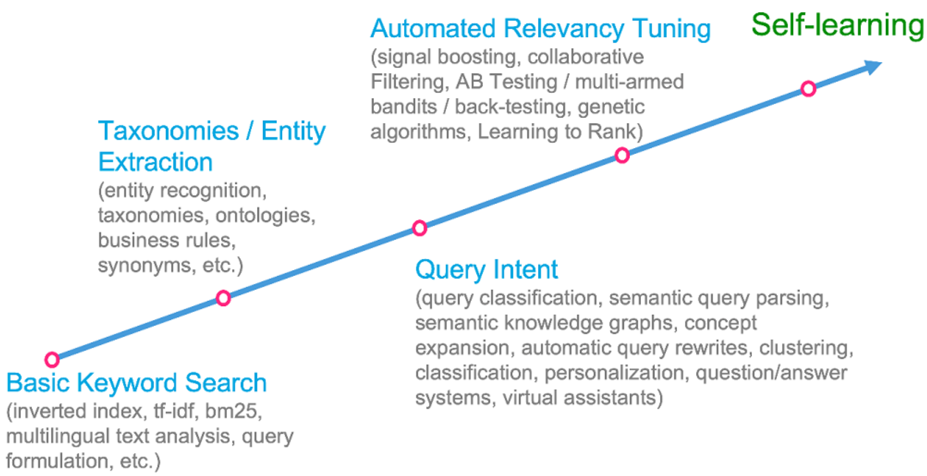


Figure 1.9 The Typical Search Intelligence Progression, from basic keyword search to a full self-learning search platform.

In this progression, organizations almost always start with basic keyword search in order to get up and running. Once in production, they realize their search relevancy needs to be improved, as which point they start manually tuning field weights, boosts, text and language analysis, and introducing additional features and functions to improve the aggregate relevancy of search results.

Eventually, they realize they need to inject domain understanding into their search capabilities, at which point organizations begin to invest in synonyms lists, taxonomies, lists of known entities, and domain-specific business rules. While these all help, they eventually also discover that relevant search is very much dependent upon successfully interpreting user queries and understanding user intent, so they begin investing in techniques for query classification, semantic query parsing, knowledge graphs, personalization, and other attempts to correctly interpret user queries.

Because these tasks successfully yield improvement, this success often results in the creation of large teams investing significant time manually tuning lists and parameters, and eventually organizations may realize that it is possible (and more expedient) to automate as much of that process as possible through learning from signals, user testing (A/B, multi-armed-bandit, and offline relevancy simulations), and building of machine-learned relevancy models. The end goal is to completely automate each of these steps along the search intelligence progression and enable the engine to be self-learning. It takes a while for most to get there, however, so it's important to start with a solid foundation.

1.5.1 The Core Search Foundation

The first step in building a search platform is almost always to get traditional keyword search working (the "content understanding" part back in Figure 1.8). Many people and teams have spent years or even a decade or more tuning and improving this step, and a whole discipline called *Relevance Engineering* has arisen that has historically focused significant efforts in understanding content, improving content for search, adjusting boosts, query parameters, and query functions, and otherwise trying to maximize the relevance of the traditional search experience. As relevance engineers become more sophisticated, their work often bleeds over into the realms of user understanding and recommendations, as well as into the domain-understanding and semantic search realm.

Our focus in *AI Powered Search* will be on automating the process of learning and optimizing search relevance to be continual and automatic, but a deep understanding of how to think like a relevance engineer and tune a search engine yourself would be quite helpful to you as background on that journey. For that background into the world of Relevance Engineering and tuning traditional keyword search relevance, I highly recommend the book *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016).

So what characteristics actually differentiate a well-tuned search engine and an AI-powered search engine? A well-tuned search engine clearly serves as the foundation upon which AI-powered search is built, but AI-powered search goes far beyond just being well-tuned and delivers on the ability to continuously learn and improve through Reflected Intelligence. *Reflected Intelligence* is the idea of leveraging continual feedback loops of user inputs, content updates, and user interactions with content in order to continually learn and improve the quality of your search application.

1.5.2 Reflected Intelligence through Feedback Loops

Feedback loops are critical to building an AI-powered search solution. Imagine if your entire middle school, secondary school, and possible post-secondary school education had consisted of nothing more that you reading text books: no teachers to ask questions, no exams to test your knowledge and provide feedback, and no classmates or others with which to interact, study, or collaborate. You would have probably hit endless walls where you were unable to fully grasp certain concepts or even understand what was being read, and you would probably have understood many ideas incorrectly and never even had the opportunity to realize or adjust your assumptions.

Search engines often operate this same way. Smart engineers push data to the search engine and tune certain features and feature weights, but then the engine just reads those configurations and that content and statically acts upon it the same way every time for repeated user queries. Search engines are the perfect kind of system for interactive learning, however, through the use of feedback loops. Figure 1.10 shows the idea of search engine feedback loops.

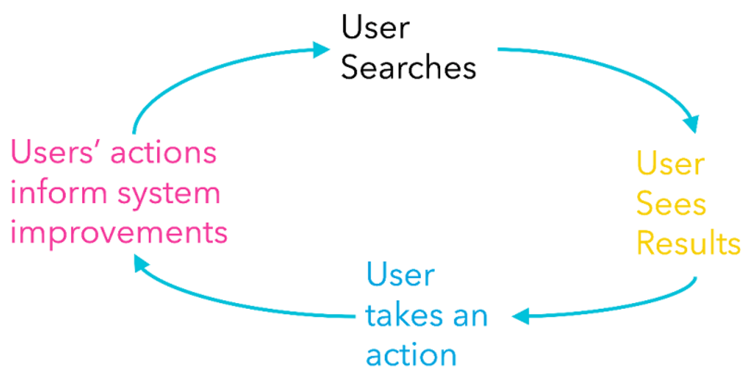


Figure 1.10 Reflected Intelligence through Feedback Loops

In Figure 1.10, you see the typical flow of information through a search feedback loop. First, a user issues a query. This query executes a search, which returns results to an end user, such as a specific answer or list of answers or a list of links to pages. Once presented with the list, the user then takes one or more actions. These actions usually start with clicks on documents, but those clicks can ultimately lead to adding an item to a shopping cart and purchasing it (ecommerce), giving the item a thumbs up or thumbs down (media consumption website), liking or commenting on the result (social media website), or any number of other context-specific actions.

Each of those actions taken by a user, however, is likely to provide clues as to the perceived relevance of the search results to that user. A thumbs up or thumbs down provides explicit positive and negative relevance judgements, as do add to cart, purchases, likes, and bookmarks. Clicks provide less clear signals, but usually indicate some perceived suitability as a search result.

These actions can then be leveraged to generate an improved relevance ranking model for future searches. For example, if the majority of users for a given query always click on result number three more often than on result one or two, this is a strong indicator that result number three is perceived as a better, more relevant result. Leveraging feedback loops, your search application can then automatically adjust the results ordering for that query in future searches, delivering an improved search experience for the next user's search. This feedback loop then continues again, ad infinitum, as the engine is constantly self-tuning.

BEHAVIORAL INTELLIGENCE FROM USER SIGNALS

The searches, clicks, likes, add to carts, purchases, comments, and any number of other interactions your users may have with your search application, are incredibly valuable data that your search application needs to capture. We collectively refer to these data points as *signals*. Signals provide a constant stream of feedback to your search application recording every meaningful interaction with your end users. These digital moments - whether a search, a list of returned search results, a click, a purchase, or anything in-between, can then be leveraged by machine learning algorithms to generate all kinds of models to power user understanding, content understanding, and domain understanding. Figure 1.11 shows the data flow for the collection and processing of signals in a typical AI-powered search application.

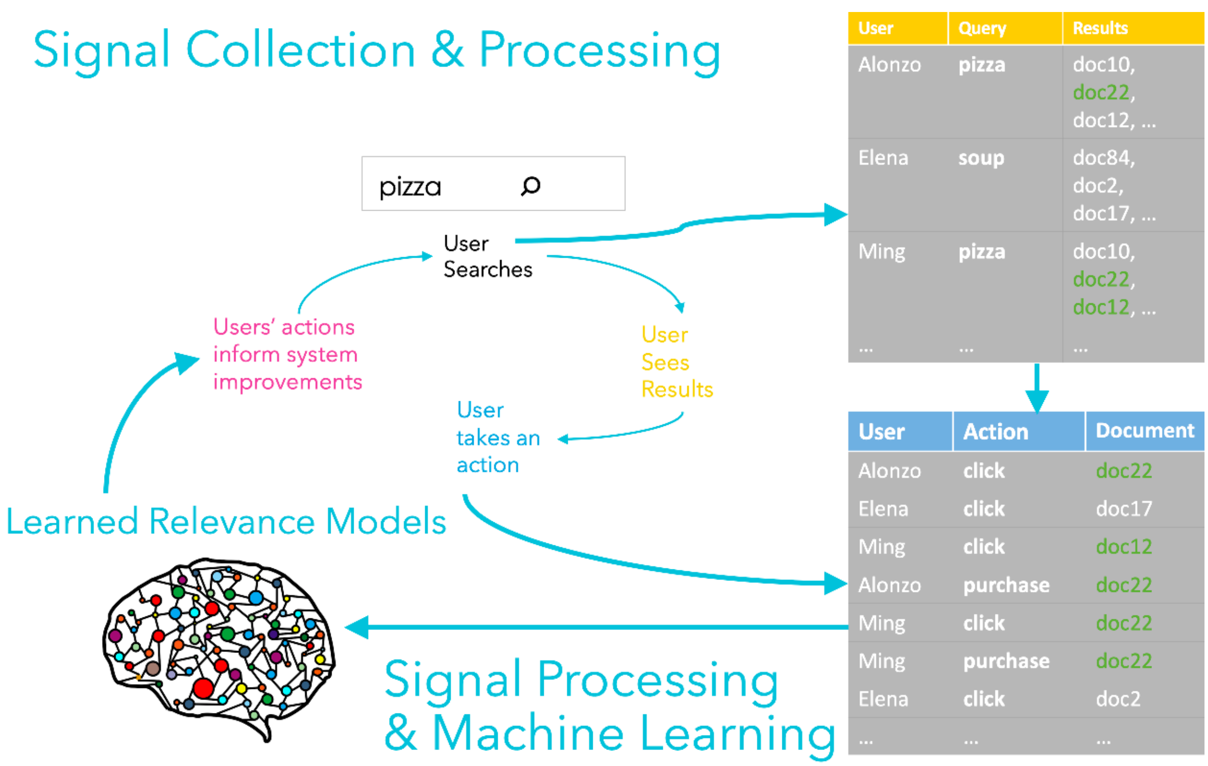


Figure 1.11 Signal collection & processing data flow

In Figure 1.11, you'll see signals being collected for each search, as well as resulting clicks and purchases, though unique signals can also be recorded for any other kind of user interaction (add to cart, facet click, bookmark, hover, or even page dwell time).

Signals are one of the two kinds of fuel that power the intelligence engine of your AI-powered search application, with the other type being your content.

CONTENT INTELLIGENCE FROM DOCUMENTS

While signals provide the constant stream of usage and feedback data to your search application, your content is also a rich source of information that can be leveraged in your feedback loops. For example, if someone searches for a particular keyword, the other keywords and top categories in the documents returned serve as a valuable datapoint that can be leveraged to tag or categorize the query. That data might be shown to end users (as facets, for example), and then users might interact with that data, which subsequently generates a signal from which the engine can then learn.

Further, the content of your documents contains a representative textual model of your domain. Entities, domain-specific terminology, and the sentences contained within your documents serve as a rich, semantic graph representing the domain of your data. That graph can be leveraged to drive powerful conceptual and semantic search that better understands your domain. We'll dive more deeply into understanding your content in chapter 3, and into these semantic search capabilities leveraging this rich semantic knowledge graph in chapter 5.

1.5.3 Curated vs. Black-box AI

Many modern AI techniques rely heavily on deep learning based on artificial neural networks. Unfortunately, it is very challenging in most cases for a human to understand specific factors that go into any particular prediction or output from the deep learning model due to the internal complexity of the learned model.

This results in a "black box AI" system, where the results may be correct or impressive, but they are not easy for someone to debug — or more importantly, correct — when the model makes an incorrect judgement. An entire field of *Explainable AI* (sometimes called *Interpretable AI* or *Transparent AI*) has arisen out of a need to be able to understand, curate, and trust these models.

In this book, while we will cover several deep learning approaches to search, we will largely focus our efforts on creating intelligence which can be expressed in human terms and then corrected and augmented by human intelligence. You can think of this as "AI-assisted human curation", or alternatively as "human-assisted AI", but either way the overriding philosophy of this book is to use AI to automate the process of search intelligence, while keeping the human in the loop and able to take control and inject their own subject matter expertise as needed.

1.5.4 Architecture for an AI-powered search engine

The architecture for an AI-powered search engine requires numerous building blocks to be assembled together to form a smart end-to-end system. You obviously need a core search engine like Solr or Elasticsearch. You need to feed your searchable content into the engine, transforming it to make it more useful. These transformation might include changes like:

- Classifying the document, adding the classification as a field
- Normalizing field values
- Entity Extraction from text, adding entities in separate fields
- Sentiment Analysis
- Clustering content, adding clusters as a field
- Phrase detection and annotation
- Pulling in additional data from a knowledge graph, external API, or other data source
- Part of Speech Detection and other Natural Language Processing steps
- Fact extraction (such as RDF triples)
- Spam or NSFW (Not Safe For Work) content detection
- Application of other Machine Learning models or ETL rules to enrich the document

Once the data is in the engine, your goal is to make it available for searching. This requires query pipelines, which can parse incoming queries, idenfity phrases and entities, expand to related terms/synonyms and concepts, correct misspellings, and then rewrite the query so your core engine can find the most relevant results.

Much of this query intelligence requires a robust understanding of your domain, however. This requires running batch jobs on your content and user signals in order to learn patterns and derive domain-specific intelligence. What are the most common misspellings from your users, and what do they choose as the correct spelling among multiple candidates? When a user searches for specific queries, which documents should be boosted as the most popular? For unknown queries, what is the ideal ranking among all the attributes/features available for matching?

We need access to most of these answers in a pre-computed (or quickly computable) fashion at query time, because we expect queries to be real-time and often to return within milliseconds. This requires a job processing framework (we'll use Spark in this book) and a workflow scheduling mechanism to keep the jobs running in sequence.

You'll also have a constant stream of new data incoming in the form of user signals, so you'll also need a mechanism for capturing and storing those (capturing on the front-end application, storing in your core engine or other back-end system).

The signals will then be used to generate all kinds of models — from signal boosting of specific items for popular queries, to generation of generalized learning to rank models which apply to all queries, to generation of user-specific recommendations and personalization preferences for each user or segment of users.

Ultimately, you'll end up with a system that receives constant streams of document changes and user signals, is constantly processing those updates to learn and improve the model, and is then constantly adjusting future search results and measuring the impact of changes to continually deliver more intelligent results. That is the key behind AI-powered search: the processes of continual learning and improvement based upon real user interactions and evolving content patterns, in order to fully understand user intent and deliver an ever improving search experience.

1.6 Summary

- Expectations for search sophistication are evolving, with end users expecting search to now be domain-aware, contextual and personalized, conversational, multi-modal (through text, voice, images, or even push-based events), intelligent, and assistive.
- Search and recommendations are the two extreme ends of a continuous personalization spectrum within information retrieval.
- Correctly interpreting user intent requires simultaneous understanding of your content, your user and their preferences, and the knowledge domain in which your platform operates.
- Optimal search relevancy lies at the intersection of Personalized Search (traditional keyword search + collaborative recommendations), Semantic Search (traditional keyword search + knowledge graphs), and Multi-modal Recommendations (collaborative recommendations + knowledge graphs).
- The techniques in this book should apply to most search platforms, but we'll primarily leverage Apache Solr and Apache Spark for most of our examples.
- AI-powered search operates on two kinds of fuel: content and user signals
- Reflected Intelligence - the use of feedback loops to continually collect signals, tune results, and measure improvements, is the engine the enables AI-powered search to learn and constantly improve.

2 Working with natural language

This chapter covers:

- Uncovering the hidden structure in unstructured data
- A search-centric philosophy of language and natural language understanding
- Exploring distributional semantics and word embeddings
- Modeling domain-specific knowledge
- Tackling challenges in natural language understanding and query interpretation
- Applying natural language learning techniques to both content and signals

In the first chapter, we provided a high-level overview of what it means to build an AI-powered search system. Throughout the rest of the book, we'll explore and demonstrate the numerous ways your search application can continuously learn from your content and your user behavioral signals in order to better understand your content, your users, and your domain, and to ultimately deliver users the answers they need. We will get much more hands on in chapter three, firing up a search server (Apache Solr), a data processing layer (Apache Spark), and starting with the first of our Jupyter notebooks, which we'll use throughout the book to walk through many step-by-step examples.

Before we dive into those hands-on examples and specific implementations (the "what"), however, it is important in this chapter that we first establish a shared mental model for the higher level problems we're trying to solve. Specifically, when it comes to intelligent search, we have to deal with many complexities and nuances in natural language - both in the content we're searching and in our users' search queries. We have to deal with keywords, entities, concepts, misspellings, synonyms, acronyms, ambiguous terms, explicit and implied relationships between concepts, hierarchical relationships usually found in taxonomies, higher-level relationships usually found in ontologies, and specific instances of entity relationships usually found in higher-level knowledge graphs.

While it might be tempting to dive immediately into some specific problems like how to automatically learn misspellings from content or how to discover synonyms from mining user search sessions, it's going to be more prudent to first establish a conceptual foundation that explains what *kinds* of problems we have to deal with in search and natural language understanding. Establishing that philosophical foundation will enable us to build better end-to-end solutions in our AI-powered search system, where all the parts work together in a cohesive and integrated way. This chapter will thus provide the philosophical underpinnings for how we tackle the problems of natural language understanding throughout this book and apply those solutions to make our search applications more intelligent. We'll begin by discussing some common misconceptions about the nature of free text and other unstructured data sources.

2.1 The myth of unstructured data

The term "unstructured data" has been used for years to describe textual data, because it does not appear to be formatted in a way that can be readily interpreted and queried. The widely held idea that text, or any other data that doesn't fit a pre-defined schema ("structure"), is actually "unstructured", however, is a myth that we'll spend time reconsidering throughout this section.

The term *unstructured data* arose out of the database world. The most common kind of database is the relational database, commonly called a "SQL database" after the Structured Query Language (SQL) most commonly used to query and retrieve results. In a world that requires your data to be broken into records (rows) with pre-defined attributes (columns) in order to query across those records, it makes sense that anything not fitting this model for structured querying would therefore be assigned the label of "unstructured data".

If you look up "unstructured data" in Wikipedia, it is defined as "information that either does not have a pre-defined data model or is not organized in a pre-defined manner". The entry goes on to say that "unstructured information is typically text-heavy, but may contain data such as dates, numbers, and facts, as well".

The phrase "unstructured data" really is a poor term to describe textual content, however. In reality, the terms and phrases present in text encode an enormous amount of meaning, and the linguistic rules applied to the text to give it meaning serve as their own structure. Calling text unstructured is a bit like calling a song playing on the radio "arbitrary audio waves". Even though every song has unique characteristics, most typically exhibit common attributes (tempo, melodies, harmonies, lyrics, and so on). Though these attributes may differ or be absent from song to song, they nevertheless fit common expectations that then enable meaning to be conveyed by and extracted from each song. Textual information typically follows similar rules - sentence structure, grammar, punctuation, interaction between parts of speech, and so on. Figure 2.1 shows an example of text which we'll explore a bit more in the upcoming sections as we investigate this structure further.

Trey Grainger works at Lucidworks.
He spoke at the Activate 2019 conference.
#Activate19 (Activate) was held in Washington,
DC September 9-12, 2019. Trey got his masters
from Georgia Tech.

Figure 2.1 Unstructured Data. This text represents typical unstructured data you may find in a search engine.

While text is the most commonly recognized kind of unstructured data, there are also several other kinds of unstructured data that share similar characteristics with textual data, as we'll see in the next section.

2.1.1 Types of unstructured data

Free text content is considered the primary type of unstructured data, but search engines are also commonly used to index many other kinds of data that similarly don't fit neatly into a structured database. Common examples include images, audio, video, and event logs. Figure 2.2 expands on our text example from Figure 2.1 and includes several other types of unstructured data, such as audio, images, audio, and video.

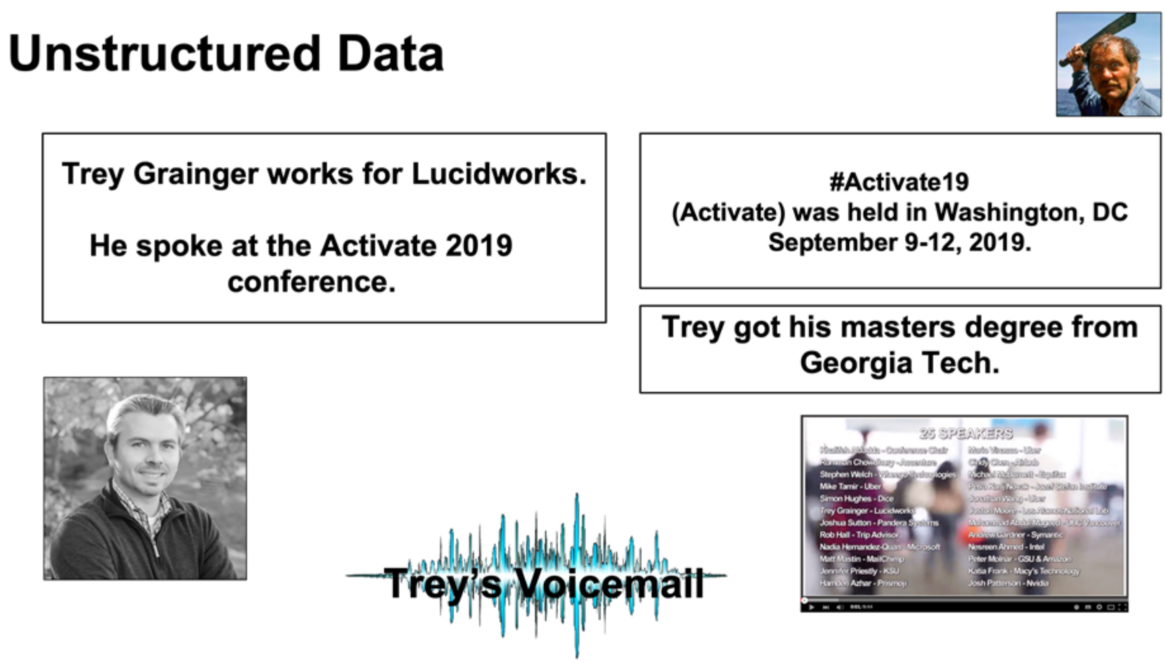


Figure 2.2 Multiple types of unstructured data. In addition to the text from the last example, we now see images, audio, and video, which Are other forms of unstructured data.

Audio is the most similar to text content, since it is often just another way to encode words and sentences. Of course, audio can include much more than just spoken words — it can include music and non-language sounds, and it can more effectively encode nuances such as emotion, tone of voice, and simultaneously overlapping communication.

Images are another kind of unstructured data. Just as words form sentences and paragraphs to express ideas, images form grids of colors that taken together form pictures.

Video, then, serves as yet another kind of unstructured data, as it is a combination of multiple images over time, as well as optional audio that coincides with the image progression.

Often times unstructured data may be found mixed with structured data, which we typically refer to as "semi-structured" data. Log data is a great example of such semi-structured data. Often logs end up being semi-structured, for example, having an event date, event type (such as warning vs. error or search vs. click), and some kind of log message or description in free text.

Technically speaking virtually any kind of file could be considered unstructured data, but we'll primarily deal with the aforementioned types throughout this book.

Search engines are often tasked with handling each of these kinds of unstructured data, so we'll dive into all of these at times throughout this book.

2.1.2 Data types in traditional structured databases

In order to better deal with our unstructured data, it may be useful to first contrast it with structured data. This will allow us to later draw parallels between how we can query unstructured data representations versus structured ones.

A record (row) in a SQL database is segmented into columns, which can each be of a particular data type. Some of these data types represent discrete values - values that come from an enumerated list. IDs, names, and textual attributes are a few examples of discrete values.

Continuous values are another type of data that may appear in a column. Dates and time ranges, numbers, and other column types which represent ranges without a finite number of possible values would all be considered continuous values.

Other content, sometimes informally referred to as "blobs", may not have values which can be easily queried or used as part of a query without slow, per-row processing of the text during each query. These blob-type fields are typically used as a dumping ground for unstructured data, and many contain things like free text, binary content, images, and so on.

Generally speaking, when one wants to relate different rows together or to relate them to rows in other database tables, "joins" will be performed on the discrete values. Joins leverage a shared value (often an ID field) to link two or more records together in order to form a composite record that relates each of those records together.

For example, if someone had two tables of data, one representing "employees" and another representing "companies", then there would likely be an "ID" column on the "companies" table, and a corresponding "company" column on the employees table. The company field on the employees table is known as a *foreign key*, which is a value that is shared across the two tables and which is used to link the records together based upon a shared identifier. Figure 2.3 demonstrates this example, showing examples of discrete values, continuous values, and a join across tables using a foreign key.

Structured Data

Employees Table

id	name	company	start_date
lw100	Trey Grainger	1234	2016-02-01
dis2	Walt Disney	9123	1928-11-28
tsla1	Elon Musk	5678	2003-07-01

Companies Table

id	name	start_date
1234	Lucidworks	2007-08-02
5678	Tesla	1928-11-28
9123	Disney	2003-07-01

Figure 2.3 Structured data in a typical database. Discrete values represent identifiers and enumerated values, continuous values represent data that falls within ranges, and foreign keys exist when the same value exists across two tables and can thus be used as a shared attribute which creates a relationship between corresponding rows from each table.

2.1.3 Joins, fuzzy joins, and entity resolution in unstructured data

Whereas structured data in a database is already in an easily queryable form, the reality is that unstructured data suffers less from a lack of structure, and more just from having a large amount of information packed into a very flexible structure. In this section, we'll walk through a concrete example that uncovers this hidden structure in unstructured data and demonstrates the ways it similarly be leveraged to find and join relationships between documents.

FOREIGN KEYS IN UNSTRUCTURED DATA

In the last section, we discussed how foreign keys can be used to join two rows together in a database based upon a shared identifier between the two records. In this section, we'll show how the same objective can actually also be achieved with unstructured data.

For example, we can easily map the idea of "foreign keys" used in a SQL table into the same unstructured information we previously explored in Figure 2.2. Notice in Figure 2.4 that two different sections of text both contain the word "Activate", which refers to a technology conference.

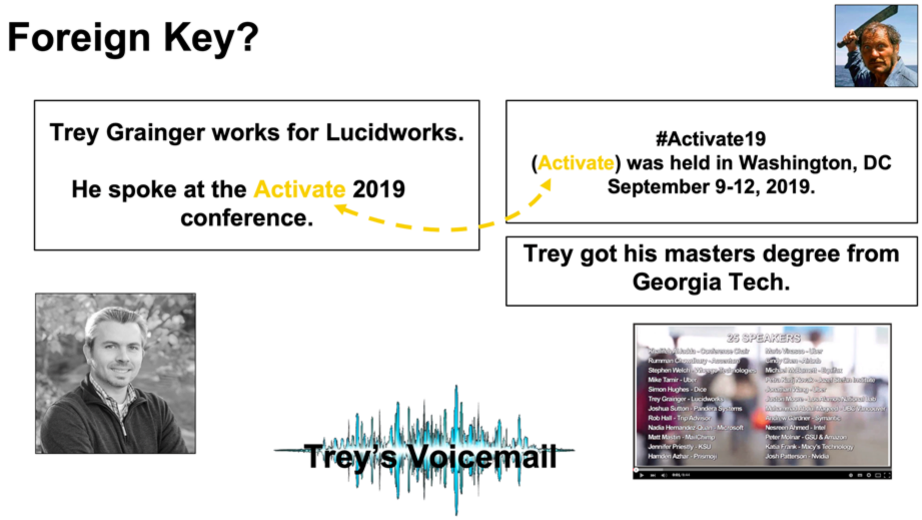


Figure 2.4 Foreign keys in unstructured data. In this example, the same term is being used to join across two related text documents.

The first instance indicates that I'm speaking at this conference, while the second block of text contains general information about the event. For purposes of our example, let's assume that every piece of information (block of text, image, video, and audio clip) is represented as a separate document in our search engine. As such, there is functionally very little difference between having two rows in a database table that each contain a column with the value of "Activate", and having separate documents in our search engine that each contain the value of "Activate". In both cases, we can think of these documents as related by a foreign key.

FUZZY FOREIGN KEYS IN UNSTRUCTURED DATA

With unstructured data, however, we have much more power than with traditional structured data modeling. In Figure 2.5, for example, notice that now two documents are linked that both refer to me - one using my full name of "Trey Grainger", and one simply using my first name of "Trey".

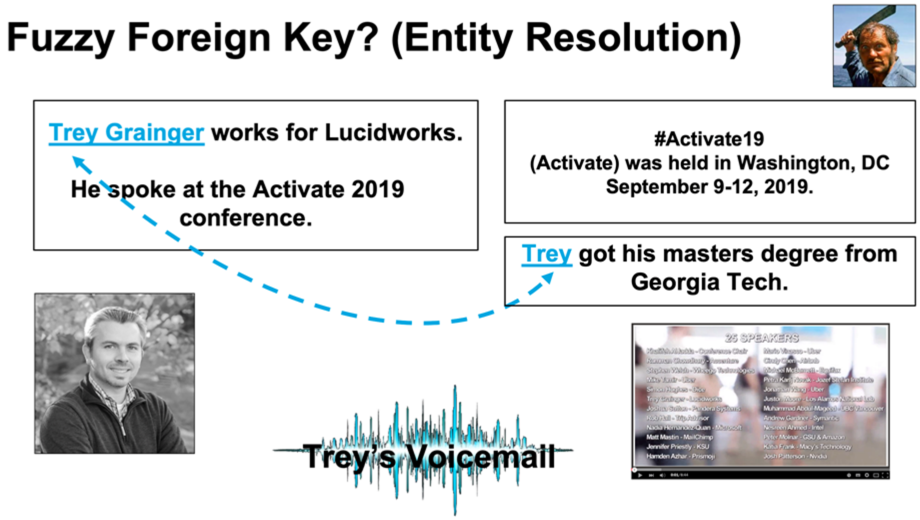


Figure 2.5 Fuzzy foreign keys. In this example, the same entity is being referenced using different term sequences, and a join is occurring based upon both phrases resolving to the same entity.

This is an example of entity resolution, where there are two different representations of the entity, but they can still be resolved to the same meaning, and therefore can still be used to join information between two documents. I like to think of this as a "Fuzzy Foreign Key", since it's still a foreign key, but not in a strict token matching sense, as it requires additional Natural Language Processing and entity resolution techniques to resolve.

Once we've opened that door to advanced text processing for entity resolution, of course, now we can learn even more from our unstructured information.

For example, not only do the names "Trey" and "Trey Grainger" in these documents refer to the same entity, but so do the words "he" and "his", as Figure 2.6 demonstrates.

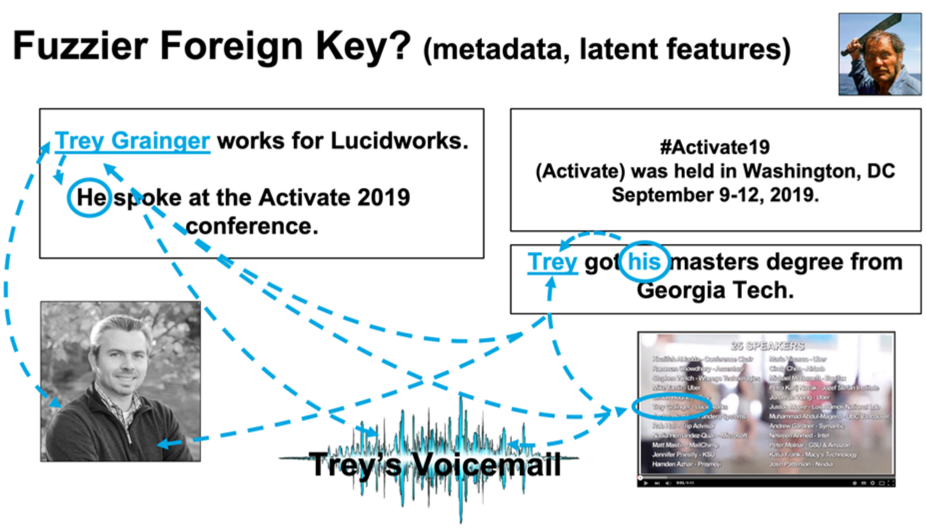


Figure 2.6 Fuzzier foreign keys. In this example, proper nouns, pronouns, images, and references within video are all resolved to the same entity, which can then be used to join across documents.

You'll also notice in Figure 2.6 that both an image of me (in the bottom-left corner, in case you have no idea what I look like) and a video containing a reference to my name are identified as related and joined back to the textual references. We're essentially relying on the hidden structure present in all of this unstructured information in order to understand the meaning, related the documents together, and learn even more about each of the referenced entities in those documents.

DEALING WITH AMBIGUOUS TERMS

So far, so good, but in real-world content it is not always appropriate to assume that the same term in multiple places carries the same meaning, or even that our entity resolution always resolves entities correctly. This problem of the same spelling of words and phrases having multiple potential meanings is called *polysemy*, and dealing with these ambiguous terms can be a huge problem in search applications.

You may have noticed an odd image in the upper-right-hand corner of the previous figures that seemed a bit out of place in our examples. This image is of a fairly terrifying man holding a machete. Apparently, if you go to Google and search for "Trey Grainger", this image comes back. If you dig in further, you'll see in Figure 2.7 that there's a Twitter user also named "Trey Grainger", and this image is his profile picture.

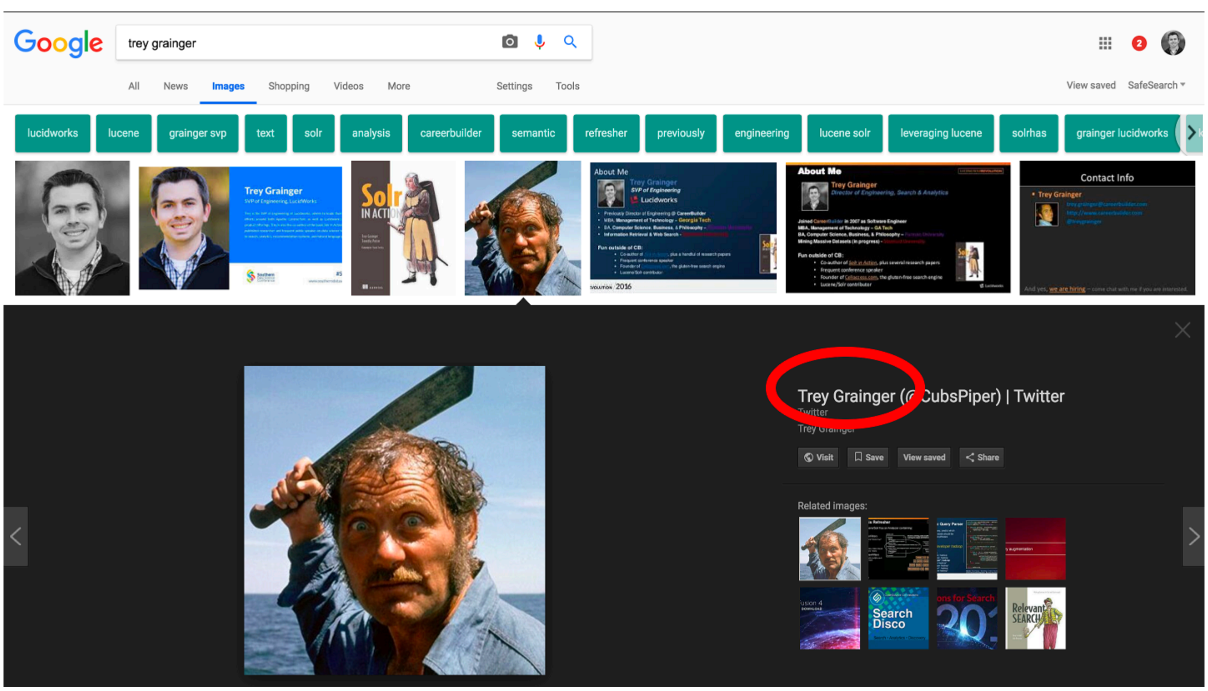


Figure 2.7 Polysemy. This image shows a Google search for the phrase "Trey Grainger". Pictures of multiple different people are returned because those people's names share the same spelling, making the term sequence "Trey Grainger" ambiguous.

The picture is apparently of Robert Shaw (who plays Quint in the 1975 movie Jaws), but it's definitely not the kind of thing you want people to first come across when they search for you online!

There are two key lessons to take away here. First, never Google yourself - you might be terrified at what you find. Second, and on a more serious note, polysemy is a major problem in search and natural language understanding. It's not safe to assume a term has a single meaning, or even a consistent meaning in different contexts, and it's important that our AI-powered search engine is able to leverage context to differentiate these different meanings.

UNSTRUCTURED DATA AS A GIANT GRAPH OF RELATIONSHIPS

In the previous sections we've seen that unstructured data not only contains rich information (entities and their relationships), but also that it is possible to relate together different documents by joining them on shared entities, similarly to how foreign keys work in traditional databases. Typical unstructured data contains so many of these relationships, however, that instead of thinking in terms of rows and colums, it may be more useful to think of them as a giant graph of relationships, as we'll explore in this section.

At this point, it should be clear that there is much more structure hidden in unstructured data than most people appreciate. Unstructured information is really more like "hyper-structured" information - it is a graph that contains much more structure than typical "structured data".

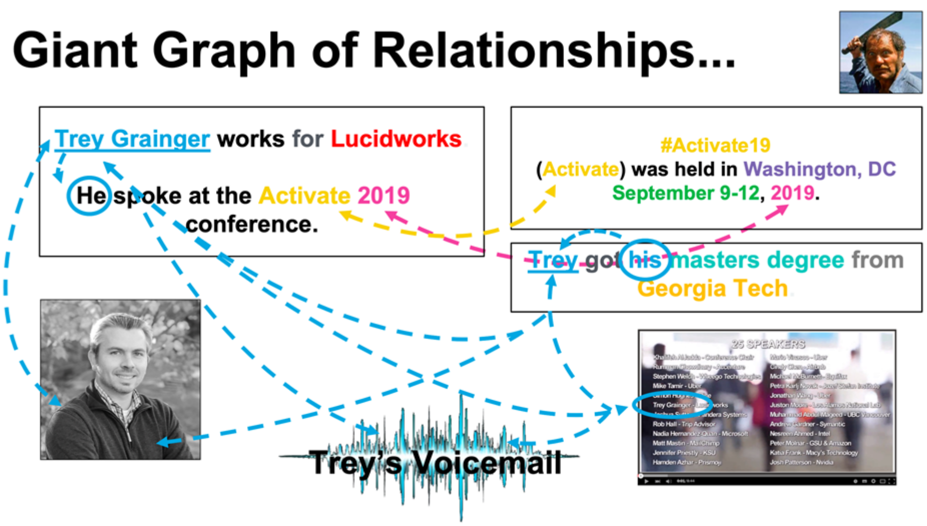


Figure 2.8 Giant graph of relationships. A rich graph of relationships emerges from even just a few related documents.

Figure 2.8 demonstrates this giant graph of relationships that is present in even the small handful of documents from our example. You can see names, dates, events, locations, people, companies, and other entities, and you can infer relationships between them leveraging joins between the entities across documents. You'll also notice that the images have been correctly disambiguated so that the machete guy is now disconnected from the graph.

If all of this can be learned from just a few documents, imagine what can be learned from the thousands, or millions, or billions of documents you have within your own search engine.

Part of an AI-powered search platform is being able to learn insights like this from your data. The question is, how do you leverage this enormous graph of semantic knowledge in order to drive this intelligence?

Fortunately, the inherent structure of the inverted index in your search engine makes it very easy to traverse this graph without any additional explicit data modeling required. We will dive deep into how to harness this semantic knowledge graph hidden in your data in chapter 5.

2.2 The structure of natural language

In the last section we discussed how text and unstructured data typically contain a giant graph of relationships which can be derived by looking at shared terms between different records. If you've been building search engines for a while, you are used to thinking about your content in terms of "documents", "fields", and "terms" within those fields. When interpreting the semantic meaning of your content, however, there are a few more levels to consider.

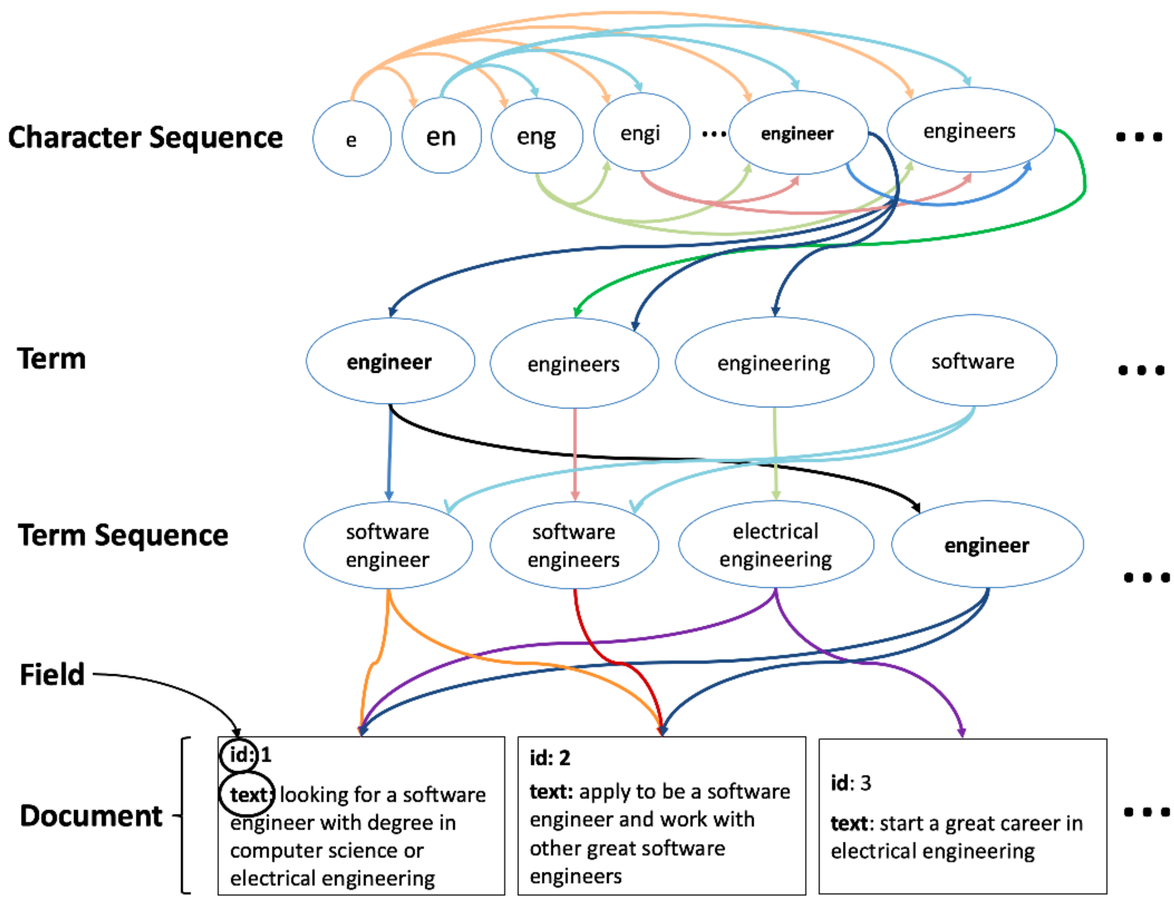


Figure 2.9 Semantic data encoded into free text content. Characters form character sequences, which form terms, which form term sequences, which form fields, which form documents, which form a corpus.

Figure 2.9 walks through these additional levels of semantic meaning. At the most basic level, you have *characters*, which are single letters, numbers, or symbols, such as the letter "e" in the figure. One or more characters are then combined together to form *character sequences* such as "e", "en," eng", … "engineer", "engineers". Some character sequences form terms, which are completed words or tokens that carry an identifiable meaning, such as "engineer", "engineers", "engineering", or "software". One or more terms can then be combined together into *term sequences* - usually called "phrases" when the terms are all sequential . These include things like "software engineer", "software engineers", and "senior software engineer". For simplicity in this book, we also consider single terms to be "term sequences", and thus any time we refer to "phrases", this is also inclusive of single-term phrases.

SIDEBAR **Term Sequences vs. Phrases**

You may look at these definitions and be wondering what the difference is between a "term sequence" and a "phrase". Quite simply, a phrase is a term sequence where all of the terms appear sequentially. For example "chief executive officer" is both a phrase and a term sequence, whereas "chief officer"~2 is only a term sequence, since it specifies a sequence of terms that is not necessarily sequential. In the vast majority of cases, you will only be dealing with sequential term sequences, hence we'll mostly use the word "phrase" for simplicity throughout the book when referring inclusively to both single and multi-term sequential term sequences.

Of course, we know that multiple term sequences together can form sentences, multiple sentences can form paragraphs, and that paragraphs can then be rolled up into even larger groups of text. For the purposes of a search engine, though, the next level of grouping we'll typically focus on above term sequences is simply a *field*. Text fields can be analyzed in any number of ways using a text analyzer, which typically includes techniques like splitting on white space and punctuation, lowercasing all terms so they are case insensitive, stripping out noise (stopwords and certain characters), stemming or lemmatization to reduce terms down to a base form, and removal of accents. If you would like a refresher on the text analysis process, I'd recommend checking out chapter 6 of *Solr in Action*.

One or more fields are then composed together into a *document*, and multiple documents form a *corpus* or collection of data. Whenever a query is executed against the search index, it filters the corpus into a *document set*, which is a subset of the corpus that specifically relates the query in question.

Each of these linguistic levels - character sequences, terms, term sequences, fields, documents, document sets, and the corpus, all provide unique insights into understanding your content and it's unique meaning within your specific domain.

2.3 Distributional semantics and word embeddings

Distributional semantics is a research area within the field Natural Language Processing that focuses upon the semantic relationships between terms and phrases based upon the distributional hypothesis. The distributional hypothesis is that words that occur in similar contexts tend to share similar meanings. It is summarized well by the popular quote: "You shall know a word by the company it keeps".[1]

> **TIP** *"You shall know a word by the company it keeps."*
> -John Rupert Firth

When applying machine learning approaches to your text, these distributional semantics become increasingly important, and the search engine makes it incredibly easy to derive the context for any linguistic representation present in your corpus. For example, if one wanted to find all documents about C-level executives, you could issue a query like:

```
c?o
```

This query would match "CEO", "CMO", "CFO, or any other CXO-style title, as it is asking for any character sequence starting with "c" and ending with "o" with a single character in-between.

The same kind of freedom exists to query for arbitrarily-complex term sequences, as well:

```
"VP Engineering"~2
```

This query would match "VP Engineering", "VP of Engineering", "Engineering VP", or even "VP of Software Engineering", as it is asking to find "VP" and "Engineering" within two positions (edit distances) of each other.

Of course, the nature of the inverted index also makes it trivial to support arbitary Boolean queries. For example, if someone searches for the term "Word", but we want to ensure any matched documents also contain either the term "Microsoft" or "MS" somewhere in the document, we could issue the following query:

```
(Microsoft OR MS) AND Word
```

Search engines support arbitrarily complex combinations of queries for character sequences, terms, and term sequences throughout your corpus, returning document sets that supply a unique context of content matching that query. For example, if I query for the term "pizza", the documents returned are more likely going to be restaurants than car rental companies, and if I query for the term "machine learning", I’m more likely to see jobs for data scientists or software engineers than for accountants, food service workers, or pharmacists. This means that you can infer a strong relationship between "machine learning" and "software engineering", and a weak relationship between "machine learning" and "food service worker". If you dig deeper, you'll also be able to see what other terms and phrases most commonly co-occur within the machine learning document set relative to in the rest of your corpus, and thereby better understand the meaning and usage of the phrase "machine learning". We'll dive into hands-on examples of leveraging these relationships in chapter 5.

In recent years, the distributional hypothesis has been applied to create semantic understandings of terms and term sequences through what are known as *word embeddings*. A word embedding is a numerical vector that is intended to represent the semantic meaning of a given term sequence (typically a word or phrase). The term sequence is encoded into a reduced-dimension vector which can be compared with the vectors for all of the other word embeddings within the corpus, in order to find the most semantically-related documents.

In order to understand this process, it may be useful to think of how a search engine works out of the box. Let's imagine a vector exists for each term that contains a value (dimension) for every word in your corpus. It might look something like Figure 2.10.

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

Figure 2.10 Vectors with one dimension per term in the inverted index. Every query on the left maps to a vector on the right, with a value of "1" for any term in the index that is also in the query, and a "0" for any term in the index that is not in the query.

Figure 2.10 demonstrates how document matching and similarity scoring typically works in most search engines by default. For every query, a vector exists which contains a dimension for every term that is in the inverted index. If that term exists in the query, the value in the vector is "1" for that dimension, and if that value does not exist in the query, then the value is "0" for that dimension. A similar vector exists for every document in the inverted index, with a "1" value for any term from the index that appears in the document, and a zero for all other terms.

When a query is executed, an exact lookup occurs in the index for any matched terms (post-text-analysis), and then a similarity score is calculated based on a comparison of the vector for the query and the vector for the document that is being scored relative to the query. We'll talk through the specific scoring calculation further in chapter 3, but that high-level understanding is sufficient for now.

There are obvious downsides to this approach. While it is great for finding documents with exact keywords matches, what happens when you want to find "related" things instead? For example, you'll notice in Figure 2.10 that the term "soda" appears in a query, but never in the index. Even though there are other kinds of drinks (apple juice, water, cappuccino, and latte), the search engine will always return zero results because it doesn't understand that the user is searching for a drink. Similarly, you'll notice that even though the term caffeine exists in the index, that queries for "latte", "cappuccino", and "green tea" will never match the term caffeine, even though they are related.

For these reasons, it is now a common practice to use something called word embeddings to model a semantic meaning for term sequences in your index and queries. A *word embedding* for a term is a vector of features which represents the term's conceptual meaning in a semantic space. Figure 2.11 demonstrates the terms now mapped to a dimensionally-reduced vector that can serve as a word embedding.

	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

Figure 2.11 Word embeddings with reduced dimensions. In this case, instead of one dimension per term (exists or missing), now higher-level dimensions exist that score shared attributes across items such as "healthy", contains "caffeine" or "bread" or "dairy", or whether the item is "food" or a "drink".

With a new word embedding vector now available for each term sequence in the left-most column of Figure 2.11, we can now score the relationship between each pair of term sequences leveraging the similarity between their vectors. In Linear Algebra, we will use a cosine similarity function to score the relationship between two vectors, which is simply computed by performing a dot product between the two vectors and scaling it by the magnitudes (lengths) of each of the vectors. We'll visit the math in more detail in future chapters, but for now, Figure 2.12 shows the results of scoring the similarity between several of these vectors.

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):

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

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

Figure 2.12 Similarity between Word Embeddings. The dot product between vectors shows the items list sorted by similarity with "green tea", with "cheese pizza", and with "donut".

As you can see in Figure 2.12, since each term sequence is now encoded into a vector that represents its meaning in terms of higher-level features, this vector (or word embedding) can now be used to score the similarity of that term sequence with any other similar vector. You'll see three vector similarity lists at the bottom of the figure: one for "green tea", one for "cheese pizza", and one for "donut".

By comparing the vector similarity of "green tea" with all the other term sequences, we find that the top most related items are "water", "cappuccino", "latte", "apple juice", and "soda", with the least related being "donut". This makes intuitive sense, as "green tea" shares more attributes with the items higher in the list. For the "cheese pizza" vector, we see that the most similar other word embeddings are for "cheese bread sticks", "cinnamon breads sticks", and "donut", with "water" being at the bottom of the list. Finally, for the query "donut", we find the top items to be "cinnamon bread sticks", "cheese bread sticks", and "cheese pizza", with "water" once again being at the bottom of the list. These results do a great job of finding the most similar items to our original query.

It's worth noting that this vector scoring is only used in the calculation of similarity between items. In your search engine, there's usually a two-phase process whereby you first execute a keyword search and then score the resulting documents. Unless you're going to skip the first step and score all of your documents relative to your query vectors (which can be time and processing intensive), you’ll still need some combination of initial keyword or document set filtering. We’ll dive more into these mechanics for successfully implementing word embeddings and vector search in chapter 11.

These higher-level attribute vectors we’ve discussed might represent other term sequences in queries, or they could be term sequences within documents, or they could could even be entire documents. It is commonplace to encode terms and term sequences into word embeddings, but *sentence embeddings* (encoding a vector for an entire sentence), *paragraph embeddings* (encoding a vector for an entire paragraph), and *document embeddings* (encoding a vector for an entire document) are also common techniques. It’s also very common that dimensions themselves are more abstract than our examples here. For example, deep learning models may be applied that pull out seemingly unitelligible features from character sequences and the way that documents cluster together within the corpus. We wouldn’t be able to easily label such dimensions in the embedding vector, but as long as it improves the predictive power of the model to increase relevance, this is often not a huge concern for most search teams.

Ultimately, combining multiple models for harnessing the power of distributional semantics and word embeddings tends to create the best outcomes, and we’ll dive further into numerous approaches to leveraging these techniques throughout the rest of this book.

2.4 Modeling domain-specific knowledge

In chapter 1, we discussed the Search Intelligence Progression (refer to Figure 1.9), whereby organizations start with basic keyword search, and progress through several additional stages of improvement before they ultimately achieve a full self-learning system. The second stage in that search intelligence progression was that of building taxonomies and ontologies, and the third stage ("query intent") included the building of knowledge graphs. Unfortunately, there can sometimes be significant confusion among practitioners in the industry on proper definitions and use of key terminology like "ontology", "taxonomy", "synonym lists", "knowledge graphs", "alternative labels", and so on, it will benefit us to provide some definitions for use in this book so as to prevent any ambiguity. Specifically, we’ll lay out definitions for the key terms of "knowledge graph", "ontology", "taxonomy", "synonyms", and "alternative labels". Figure 2.13 shows a high level diagram for how they relate.

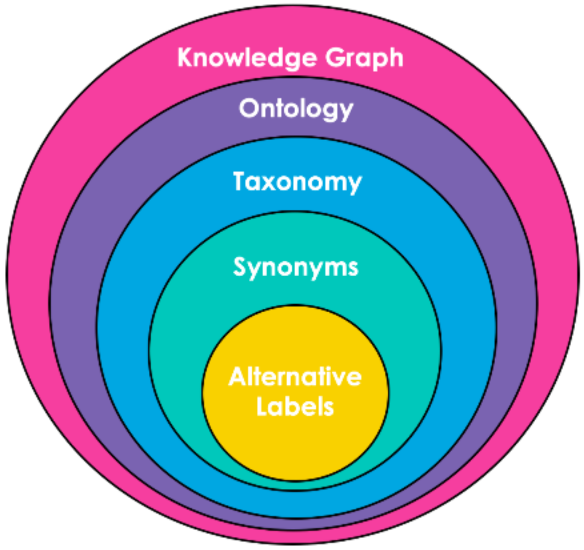


Figure 2.13 Levels of domain-specific knowledge modeling. Knowledge graphs extend ontologies, which extend taxonomies. Synonyms extend alternative labels and map to entries in taxonomies.

We'll define each of knowledge modeling techniques as follows:

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

Creation of Alternative Labels is the most straight-forward of these techniques to understand.

Acronyms (RN Registered Nurse) virtually always serve as alternative labels, as do misspellings and alternative spellings. Sometimes it is useful to keep these mappings stored in separate lists, particularly if you're using algorithms to determine them and you expect to allow for human modification of them and/or plan to re-run the algorithms later.

Synonyms are the next most common of the techniques, as virtually every search engine will have some implementation of a synonyms list. Alternative labels are a subset of a synonyms list and are the most obvious kind of synonym. Most people consider "highly related" term sequences to be synonyms, as well. For example, "software engineer" and "software developer" are often considered synonyms since they are usually used interchangably, even though there are some slight nuances in meaning between the two. Sometimes, you'll even see translations of words between languages showing up in synonyms for bilingual search use cases.

One key difference between Alternative Labels and more general Synonyms is that alternative labels can be seen as "replacement" terms for the original, whereas synonyms are more often used as "expansion" terms to add alongside the original. Implementations can vary widely, but this ultimately boils down to whether you are confident two term sequences carry exactly the same meaning (and want to normalize it), or whether you're just trying to include additional related term sequences so you don't miss other relevant results.

Taxonomies are the next step up from synonyms. Taxonomies focus less on substitute or expansion words, and instead focus on categorizing your content into a hierarchy. Taxonomical information will often be used to drive website navigation, to change behavior with a subset of search results (for example, show different faceting or filtering options based upon a parent product category), or to apply dynamic filtering based upon a category to which a query maps. For example, if someone searches for "range" on a home improvement website, the site might automatically filter down to "appliances" to remove noise of other products which contain phrases like "fall within the range" in their product description. Synonyms then map into a taxonomy as pointers to particular items within the taxonomy.

Whereas taxonomies tend to specify parent-child relationships between categories and then map things into those categories, ontologies provide the ability to define much richer relationships between things (term sequences, entities) within a domain. Ontologies typically define more abstract relationships, attempting to model the relationships between kinds of things in a domain - for example, "employee reports to boss", "CMO's boss is CEO", "CMO is employee". This makes ontologies really useful for deriving new information from known facts by mapping the facts into the ontology and then drawing logical conclusions based upon relationships in the ontology that can be applied to those facts.

Knowledge Graphs are the relative newcomer to the knowledge management space. Whereas ontologies define high-level relationships which apply to types of things, knowledge graphs tend to be full instantiations of ontologies that also include each of the specific entities that fall within those types. Using our previous ontology example, a knowledge graph would additionally have "Michael is CMO", "Michael reports to Marcia", and "Marcia is CEO" as relationships in the graph. Before knowledge graphs came into the forefront, it was common for these more detailed relationships to be modeled into ontologies, and many people still do this today. As a result, you'll often see the terms knowledge graph and ontology used interchangably, though this is becoming less common over time.

Throughout this book, we will mostly focus our discussions on alternative labels, synonyms, and knowledge graphs, since taxonomies and ontologies are mostly subsumed into knowledge graphs.

2.5 Challenges in natural language understanding for search

In the last few sections, we've discussed the rich graph of meaning embedded within unstructured data and text, as well as how distributional semantics and word embeddings can be leveraged to derive and score semantic relationships between term sequences in queries and documents. We also introduced key techniques for knowledge modeling. In this section, we'll discuss a few key challenges associated with Natural Language Understanding that we'll seek to overcome throughout this book.

2.5.1 The challenge of ambiguity (polysemy)

In section 2.1.3, we introduced the idea of polysemy, or ambiguous terms. In that section, we were dealing with an image tagged with the name "Trey Grainger", but which was referring to a different person than the author of this book. In textual data, however, we have the same problem, and it can get very messy.

Consider a word like "driver". Driver can refer broadly to a "vehicle driver", a kind of golf club for hitting the ball off a tee, software that enables a hardware device to work, a kind of tool (screwdriver), or the impetus for pushing something forward ("a key driver of success"). Clearly there are many potential meaning for this word, but in reality you could dive in and explore many even more granular meanings. For example, within the "vehicle driver" category, it could mean taxi driver, or Uber driver, or Lyft driver, or it could mean professional trucker like a CDL driver (someone with a Commercial Drivers License), or it could mean bus driver. Within the subset of bus drivers, it could mean a school bus driver, a driver of a public city bus, a driver for a tour bus, and so on. This list could continue being broken down into dozens of additional categories at a minimum.

Often times when building search applications, engineers will naively create static synonyms lists and assume terms have a singular meaning that can be applied universally. The reality, however, is that every term (word or phrase) takes on a unique meaning that is based upon the specific context in which it is being used.

TIP	Every term takes on a unique meaning that is based upon the specific context in which it is being used.

It’s not often practical to support an infinite number of potential meanings, though we will discuss techniques to do this with a semantic knowledge graph in chapter 5. Nevertheless, regardless of whether you support many meanings per phrase or just a few, it’s important to recognize the clear need to support multiple meanings for any given phrase your users may encounter.

2.5.2 The challenge of understanding context

I like to say that every term (word or phrase) you ever encounter is a "context-dependent cluster of meaning with an ambiguous label".

TIP	Every word or phrase is "a context-dependent cluster of meaning with an ambiguous label"

That is to say, there is a label (the textual representation of the term) that is being applied to some concept (a cluster of meaning) that is dependent upon the context in which it is found. By this definition, it is impossible to ever interpret a term without an understanding of the context in which it is found, and as such, creating fixed synonyms lists that aren’t able to take context into account is likely to create suboptimal search experiences for your users.

As we discussed in chapter 1, the context for a query includes more than just the search keywords and the content within your documents, however. It also includes an understanding of your domain, as well as an understanding of your user. Queries can take on entirely different meaning based upon what you know about your user and any domain-specific understanding you may have. This context is necessary to both detect and to resolve the kinds of ambiguity we discussed in the last section, as well as to ensure your users are receiving the most intelligent search experience possible.

Throughout this book, our focus will be on techniques to automatically learn contextual interpretations of each query based upon the unique context in which it is being used.

2.5.3 The challenge of personalization

Just because context is important doesn't mean it is always easy to apply correctly. It is always necessary to have the ability to perform basic keywords search as a fallback for when your system doesn't understand a query, and it is almost always useful to have pre-built domain understanding that can similarly be relied upon to help interpret queries. This pre-built domain understanding then ends up overriding some of the default keyword-based matching behavior (such as joining individual keywords into phrases, injecting synonyms, and correcting misspellings).

Once you begin better understanding your users, however, it is not always obvious how to apply user-specific personalization on top of the pre-existing content and domain-specific scoring. For example, say you learn that a particular user really likes Apple as a brand because they keep searching for iPhones. Does this mean that Apple should also be boosted when they are searching for watches and computers and television streaming boxes and keyboards and headphones and music players? It could be that the user only likes Apple-branded phones and that by boosting the brand in other categories you may actually frustrate the user. For example, even if the user did search for iPhone previously, how do you know they weren't just trying to compare the iPhone with other phones they were considering?

Out of all of the dimensions of relevancy, personalization is the easiest one to trip up on, and subsequently it is the one that is least-commonly seen in modern AI-powered search applications (outside of recommendation engines, of course). We'll work through these problems further in chapter 8 in order to highlight how to strike the right balance when rolling out a personalized search experience.

2.5.4 Challenges interpreting queries vs. documents

One common problem I see when engineers and data scientists first get started with search is a propensity to apply standard Natural Language Processing techniques like language detection, part of speech detection, phrase detection, and sentiment analysis to queries. All of those techniques were designed to operate on longer blocks of text - usually at the document, paragraph, or at least sentence level.

Documents tend to be longer and to supply significantly more context to the surrounding text, whereas queries tend to be short (a few keywords only) in most use cases, and even when they are longer they tend to combine multiple ideas together as opposed to supplying more linguistic context.

As such, when trying to interpret queries, you need to leverage external context as much as possible to interpret the query. Instead of using a Natural Language Processing library that typically relies upon sentence structure to interpret the query, for example, you can try to lookup the phrases from your query in your corpus of documents to find the most common domain-specific interpretations of them. Likewise, you can leverage the co-occurrence of terms within your query across previous user search sessions by mining your user behavioral signals. This enables you to learn real intention from similar users, which would be very challenging to accurately derive from a standard Natural Language Processing library on a consistent basis.

In short, queries need special handling and interpretation due to their tendency to be short and to often imply more than they state explicitly, so fully leveraging search-centric data science approaches to queries is going to generate much better results than traditional Natural Language Processing approaches.

2.5.5 Challenges interpreting query intent

While the process of parsing a query to understand the terms and phrases it contains is important, there is often a higher-level intent behind the query — a query type, if you will. For example, lets consider the inherent differences between following queries:

```
who is the CEO?
support
iphone screen blacked out
iphone
verizon silver iphone 8 plus 64GB
sale
refrigerators
pay my bill
```

The intent of the first query for "who is the CEO?" is clearly to find a factual answer and not a list of documents. The second query for "support" is trying to navigate to the support section of a website, or to otherwise contact the support team. The third query for "iphone screen blacked out" is also looking for support, but it is for a specific problem, and the person is likely to want to find troubleshooting pages that may exist to help with that specific problem before reaching out to the actual support team.

The next two queries for "iphone" and for "verizon silver iphone 8 plus 64GB" are quite interesting. While they are both for iphones, the first search is a general search, indicating a likely browsing or product research intent, whereas the second query is a much more specific variant of the first search, indicating the user knows exactly what they are looking for, and may be closer to making a purchasing decision. As such, the general query for "iphone" may do better to return a landing page that provides an overview of iphones and the available options, while the more specific query may do better to go straight to the product page with a purchase button immediately available. As a general rule of thumb, the more general a query, the more likely the user is just browsing, whereas more specific queries — especially when they refer to specific items by name — often indicate a purchase intent or desire to find a particular known item.

The query for "sale" indicates that the user is looking for items which are available for purchase at a discounted rate, which will invoke some specially-implemented filter or redirect to a particular landing page for an ongoing sale event. The query for "refrigerators" indicates that the use wants to browse a particular category of product documents. Finally, the query for "pay my bill" indicates that the user wants to take an action — the response to this query isn't a set or search results or even an answer, but instead a redirect to a bill review and payment section of the application.

Each of these queries contains an intent beyond just a set of keywords to be matched. Whether the intent is to redirect to a particular page, to apply particular filters, to browse or to purchase items, or even to take domain-specific actions, the point is that there is domain-specific nuance to how users may express their goals to your search engine. Often times, it can be difficult to automatically derive these domain-specific user intents automatically. We'll cover a few ways to do this in chapters 6 and 7, but it is also fairly common for businesses to implement specific business rules to handle these as one-off requests. Query Intent classifiers can certainly be built to handle subsets of this problem, but successfully interpreting every possible query intent still remains challenging when building out natural language query interpretation capabilities.

2.6 The fuel powering AI-powered search

In the first chapter, we introduced the idea of reflected intelligence - leveraging feedback loops to continually learn from both content and user interactions. This chapter has focused entirely on understanding the meaning and intelligence embedded within your content, and we'll begin diving into hands-on examples for how to automatically extract intelligence out of your content in the next chapter.

It's important to recognize, however, that many of the techniques we're applying to the "unstructured data" in your documents can also be just as readily applied to your user behavioral signals. For example, we discussed earlier in this chapter how the meaning of phrases can be derived from the other phrases that they appear with the most often within your corpus, using the example that "machine learning" appears more commonly with "data scientist" and "software engineer" than it does with "accountants", "food service workers", or "pharmacists".

If you abstract this idea beyond your documents and to your user behavior, you might also expect that the users querying your search engine are likely to exhibit similar query behavior that also falls inline with the distributional hypothesis. Specifically, people who are data scientists or who are searching for data scientists are far more likely to also search for or interact with documents about "machine learning", and the likelihood of a food service worker or accountant searching for machine learning content is much lower than the likelihood for a software engineering doing the same. We can thus apply these same techniques to learn related terms and term sequences from query logs, where instead of thinking of terms and term sequences mapping to fields in documents, we think of terms in queries and clicks on results mapping to user sessions, which then map to users.

Some search applications will be content-rich, but have very few user signals. Other search applications will have an enormous number of signals, but will have very little content or have content which poses challenges from an automated learning perspective. In an ideal scenario, you have great content and an enormous quantity of user signals to learn from, which allows combining the best of both worlds into an even smarter AI-powered search application. Regardless of which scenario you're in, keep in mind that your content and your user signals can both serve as fuel for your application, and you should do your best to maximize the collection and quality of collection of each.

Now that we've covered all the background needed to begin extracting meaning from your natural language content, it's time to roll up your sleeves and get hands-on. In the next chapter, we'll dive into lots of examples as we begin to explore content-based relevancy in an AI-powered search application.

2.7 Summary

- Unstructured data is a misnomer - it is really more like hyper-structured data, as it represents a giant graph of domain-specific knowledge.
- Search engines can leverage distributional semantics - interpreting the semantic relationships between terms and phrases based upon the distributional hypothesis - to harness rich semantic meaning at the level of character sequences, terms, term sequences (typically phrases), fields, documents, document sets, and an entire corpus.
- Distributional semantics approaches enable us to learn the nuanced meaning of our queries and content from their larger surrounding context.
- Word embeddings are a powerful technique for modeling and scoring based upon the semantic meaning of phrases instead of just pure text matching statistics.
- Domain specific knowledge is commonly modeled through a combination of alternative labels, synonyms lists, taxonomies, ontologies, and knowledge graphs. Knowledge graphs typically model the output from each of the other approaches into a unified knowledge representation of a particular domain.
- Polysemy (ambiguous terms), context, personalization, and query-specific Natural Language Processing approaches represent some of the more interesting challenges in natural language search.
- Content and user signals are both important fuel for our AI-powered search applications to leverage when solving natural language challenges.

3 Ranking and Content-based Relevance

This chapter covers:

- Executing queries and returning matching search results
- Ranking search results based upon how relevant they are to an incoming query
- Controlling and specifying your own ranking functions with function queries
- Catering ranking functions to a specific domain

Search engines fundamentally do three things: ingest content, return content matching incoming queries, and sort the returned content based upon some measure of how well it matches the query. *Relevance* is the term used to describe this notion of "how well the content matches the query". Most of the time the matched content is documents, and the returned and ranked content is those matched documents along with some corresponding metadata describing the documents.

In most search engines, the default relevance sorting is based upon a score indicating how well each keyword in a query matches the same keyword in each document, with the best matches yielding the highest relevance score and returned at the top of the search results. The relevance calculation is highly configurable, however, and can be easily adjusted on a per-query-basis in order to enable very sophisticated ranking behavior.

In this chapter, we will provide an overview of how relevance is calculated, how the relevance function can be easily controlled and adjusted through function queries, and how to implement popular domain-specific and user-specific relevance ranking features. We'll start by looking at how ranking actually works.

3.1 Scoring query and document vectors with cosine similarity

In Chapter 2, we demonstrated the idea of measuring the similarity of two vectors by calculating the cosine between them. We created vectors (lists of numbers, where each number represents the strength of some feature) representing different food items, and we then calculated the cosine (the size of the angle between the vectors) in order to determine their similarity. We'll expand upon that technique in this section, discussing how text queries and documents can map into vectors for ranking purposes. We'll further get into some popular text-based feature weighting techniques and how they can be integrated to create an improved relevance ranking formula.

3.1.1 Mapping text to vectors

In a typical search problem, we start with a collection of documents and we then try to rank documents based upon how well they match some user's query. In this section, we'll walk through the process of mapping the text of queries and documents into vectors.

In the last chapter, we used the example of a search for food and beverage items, like `apple juice`, so let's reuse that example here.

Query: `apple juice`

Let's assume we have two different documents that we would like to sort based upon how well they match this query.

Document 1:

```
Lynn: ham and cheese sandwich, chocolate cookie, ice water.
Brian: turkey avocado sandwich, plain potato chips, apple juice
Mohammed: grilled chicken salad, fruit cup, lemonade
```

Document 2:

```
Orchard Farms apple juice is premium, organic apple juice made from the freshest apples,
    never from concentrate. Its juice has received the regional award for best apple juice
    three years in a row.
```

If we mapped both of these documents (containing a combined 48 words) to vectors, they would map to a 48-word vector-space with the following dimensions:

```
[a, and, apple, apples, avocado, award, best, brian, cheese, chicken, chips, chocolate,
 concentrate, cookie, cup, farms, for, freshest, from, fruit, grilled, ham, has, ice, in, is, its,
 juice, lemonade, lynn, made, mohammed, never, orchard, organic, plain, potato, premium, received,
 regional, row, salad, sandwich, the, three, turkey, water, years]
```

If you recall in section 2.3, we proposed thinking of a query for the phrase `apple juice` as a vector containing a feature for every word in any of our documents, with a value of `1` for the terms `apple` and `juice`, and a value of `0` all other terms.

Since the word `apple` is in the third position and `juice` is in the 28th position of our 48-word vector-space, a query vector for the phrase `apple juice` would look as shown in Figure 3.1.

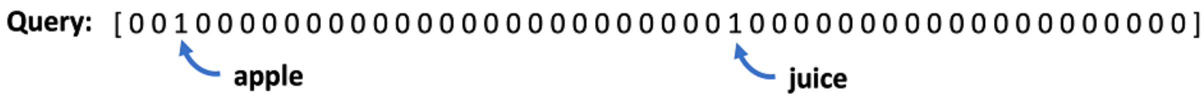


Figure 3.1 Query Vector. The query for apple juice is mapped to a vector containing one dimension for every known term, with a value of 1 for the terms apple and juice and a value of 0 for all other terms.

Note that even though the query vector only contains a non-zero value for two dimensions (representing the position of `apple` and `juice`), that it still contains values of `0` for all other possible dimensions. Representing a vector like this including every possible value is known as a *dense vector representation*.

Each of the documents also maps to the same vector space based upon each of the terms it contains:

```
Document 1:
[0 1 1 0 1 0 0 1 1 1 1 1 0 1 1 0 0 0 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 0 1 1 1 0 0 0 0 1 1 0 0 1 1 0]
```

```
Document 2:
[1 0 1 1 0 1 1 0 0 0 0 0 1 0 0 1 1 1 1 0 0 0 1 0 1 1 1 1 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 1 1 0 0 1]
```

With these these dense vector representations of our query and documents, we can now use linear algebra to measure the similarity between our query vector and each of the document vectors.

3.1.2 Calculating similarity between dense vector representations

To rank our documents, we just need to follow the same process we used in chapter 2 to calculate the cosine between each document and the query. This cosine value will then become the relevance score for each document, and we'll be able to sort documents based upon that relevance score.

Listing 3.1 shows how we would represent the query and document vectors in code, and how we would calculate the cosine similarity between the query and each document.

Listing 3.1 Cosine Similarity calculation between query and document vectors

```
query_vector = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                         0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

doc1_vector = np.array([0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1,
                        0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0])

doc2_vector = np.array([1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0,
                        1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1])

def cos_sim(vector1,vector2):
  return dot(vector1, vector2)/(norm(vector1)*norm(vector2))


doc1_score = cos_sim(query_vector, doc1_vector)
doc2_score = cos_sim(query_vector, doc2_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " + num2str(doc2_score))
```

Results:

```
Relevance Scores:
 doc1: 0.2828
 doc2: 0.2828
```

Interesting… Both documents received exactly the same relevance score, even though the documents contain lengthy vectors with very different content. It might not be immediately obvious what’s going on, so simplify the calculation by focusing only on the features that matter.

3.1.3 Calculating similarity between sparse vector representations

The key to understanding the calculation in the last section is understanding that the only features that matter are the ones shared between both the query and a document. All other features (words appearing in documents that don’t match the query) have zero impact on whether one document is ranked higher than another. As a result, we can remove all of the other insignificant terms from our vector to simplify the example, converting from a dense vector representation to what is known as a sparse vector representation.

In the last section, we worked with dense vector representations, which are vector representations including a value for every possible dimension, even if many of those values are `0` (see Figure 3.1). It is also possible to create a much smaller vector representation that only contains useful values, however. A vector representation only containing the non-zero values for a calculation is known as a *sparse vector representation*, as shown in Figure 3.2.

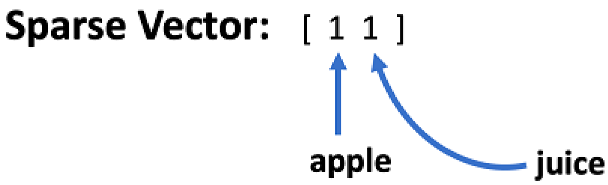


Figure 3.2 Sparse vector representation, which only contains the "present" features, unlike dense vector representations which also contain the 0 valued entries for every feature.

In most search engine scoring operations, we tend to deal with sparse vector representations because they are more efficient to work with when we are only dealing with scoring based upon the small number of features.

In addition, we can further simplify our calculations by creating sparse vectors that only include "meaningful entries" - the terms that are actually present in the query - as shown in Listing 3.2.

Listing 3.2 Cosine Similarity calculation between sparse query and document vectors

```
sparse_query_vector = [1, 1] #[apple, juice]
sparse_doc1_vector = [1, 1]
sparse_doc2_vector = [1, 1]

doc1_score = cos_sim(sparse_query_vector, sparse_doc1_vector)
doc2_score = cos_sim(sparse_query_vector, sparse_doc2_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " + num2str(doc2_score))
```

Results:

```
Relevance Scores:
 doc1: 1.0
 doc2: 1.0
```

Notice that `doc1` and `doc2` still yield the same relative score, but that now the actual score is `1.0`. If you remember, a `1.0` score from a cosine calculation means the vectors are perfect matches, and in fact, it should be obvious that since each of the sparse vectors contain the exact same values (`[1 1]`) that they are all equal and thus get a perfect score.

In fact, you'll notice several very interesting things:

- This simplified sparse vector calculation still shows both `doc1` and `doc2` returning equivalent relevance scores, since they both match all the words in the query.
- Even though the absolute score between the dense vector representation similarity (0.2828) and the sparse vector representation similarity (1.0) are different due to our new sparse vector only containing terms actually in the query, the scores are still the same relative to each other within each vector type.
- The feature weights for the two query terms (`apple`, `juice`) are exactly the same between the query and each of the documents, resulting in a cosine score of 1.0.

SIDEBAR **Vectors vs. Vector Representations**

We've been careful to refer to "dense vector representations" and "sparse vector representations" instead of "dense vectors" and "sparse "vectors". This is because there is an important difference between the idea of a vector and its representation.

Specifically, a dense vector is any vector that contains mostly non-zero values, whereas a sparse vector is any vector that contains mostly zero-values, regardless of how they are stored or represented. The vector representations, on the other hand, deal with the data structures we actually use to work with the vectors, and so since our query and document vectors are all sparse vectors, it makes sense to use a sparse vector representation to work with only the non-zero values.

In addition to these academic terminology distinctions, however, when we are working with our sparse vector representations, we are not actually preserving the original vector, but instead we are creating a brand new vector containing *only* the values that are present in the query. Since we've determined that values not in the query do not affect the relative difference query/document in score between any two documents, we simplify our calculations by creating a new sparse vector that only contains the values present in the query and, of course, we use a sparse vector representation to work with this sparse new vector.

Search engines adjust for these issues by not just considering each feature in the vector as a `1` (exists) or a `0` (does not exist), but instead providing a score for each feature based upon how *well* the feature matches. We'll discuss a few ways to do this in the following sections.

3.1.4 Term Frequency (TF): measuring how well documents match a term

The problem we encountered in the last section is that the features in our term vectors only signify *if* the word `apple` or `juice` exists, not how well each document actually represents either of the terms. We can see that the oddity of representing each term from the query as a feature with a value of `1` indicating it exists, is that both `doc1` and `doc2` will always have the same cosine similarity score for the query, even though qualitatively `doc2` is a much better match since it talks about apple juice much more.

Instead of using a value of `1` for each existing term, we can try to correct for this notion of "how well" a document matches by using the *term frequency*, which is a measure of the number of times a term occurs within each document. The idea here is that the more frequently a term occurs within a specific document, the greater the liklihood that the document is more related to the query.

If we replace the feature weights in our vector with a count of the number of times each term occurs within the document or query, then we get the vectors in Listing 3.3

Listing 3.3 Cosine similarity of term frequency vectors based upon raw term counts

```
doc1_tf_vector = [1, 1] #[apple:1, juice:1]
doc2_tf_vector = [3, 4] #[apple:3, juice:4]

query_vector = [1, 1] #[apple:1, juice:1]

doc1_score = num2str(cos_sim(query_vector, doc1_tf_vector))
doc2_score = num2str(cos_sim(query_vector, doc2_tf_vector))

print("Relevance Scores:\n doc1: " + str(doc1_score) + "\n doc2: " + str(doc2_score))
```

Results:

```
Relevance Scores:
 doc1: 1.0
 doc2: 0.9899
```

As you can see, `doc1` is considered a better cosine similarity match than `doc2`. This is because the terms `apple` and `juice` both occur "the same proportion of times" (one occurrence of each term for every occurrence of the other term) in both the query and in `doc1`, making them the most textually similar. Even though `doc2` is intuitively more "about" the query, mentioning the terms in the query significantly more. Since our goal is for documents like `doc2` with higher term frequency to score higher, we can overcome these by either:

1. Continuing to use cosine similarity, but modifying the query features to actually represent the "best" possible score for each query term, or
2. Switching from cosine similarity to another scoring function that increases as feature weights continue to increase.

Let’s try option 1 for now (we’ll visit option 2 in section 3.2).

SIDEBAR **Phrase matching and other relevance tricks**

By now, you may be wondering why we keep treating `apple` and `juice` as independent terms and why we don’t just treat `apple juice` as a phrase to boost documents higher the exact phrase. If so, your intuition is great, and this is one of many easy relevance tuning trick we’ll learn in chapter 3. For now, though, we’ll keep our query processing simple and just deal with individual keywords in order to stay focused on our main goal - explaining vector-based relevance scoring and text-based keyword scoring features.

In Listing 3.4, we adjust the feature weights in the query vector to be based upon the "best" possible match for each term.

Listing 3.4 Cosine similarity of term frequency vectors, with query weights per term normalized for best match.

```
doc1_tf_vector = [1, 1] #[apple:1, juice:1]
doc2_tf_vector = [3, 4] #[apple:3, juice:4]

*query_vector = np.maximum.reduce([doc1_tf_vector, doc2_tf_vector])* #[apple:3, juice:4]   ❶

doc1_score = cos_sim(query_vector, doc1_tf_vector)
doc2_score = cos_sim(query_vector, doc2_tf_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " + num2str(doc2_score))
```

❶ Query vector should represent the "best possible" match, so we include the top possible score for each term in the query vector.

Results:

```
Relevance Scores:
 doc1: 0.9899
 doc2: 1.0
```

As you can see, `doc2` now yields a higher cosine similarity with the query than `doc1`, an improvement that aligns better with our intuition.

While using the term frequency as the feature weight in our vectors certainly helps, textual queries exhibit additional challenges that also need to be considered. Thusfar, our documents have all contained every term from our queries, which does not match with most real-world scenarios. The following example will better demonstrate some of the limitations still present when using only term-frequency-based weighting for our text-based sparse vector similarity scoring. Let’s start with the following three text documents:

Document 1:

```
In light of the big reveal in the interview, the interesting thing is that the person in wrong
    probably made the right decision in the end.
```

Document 2:

```
My favorite book is the cat in the hat, which is about a crazy cat in a hat who breaks into a
    house and creates a crazy afternoon for two kids.
```

Document 3:

```
My careless neighbors apparently let a stray cat stay in their garage unsupervised, which
    resulted in my favorite hat that I let them borrow being ruined.
```

Let’s now map these documents into their corresponding (sparse) vector representations and calculate a similarity score. Listing 3.5 demonstrates a code example for ranking text similarity based upon term frequencies.

Listing 3.5 Ranking text similarity based upon Term Frequency.

```
doc1 = "In light of the big reveal in the interview, the interesting thing is that the person in
        wrong probably made the right decision in the end."
doc2 = "My favorite book is the cat in the hat, which is about a crazy cat in a hat who breaks
        into a house and creates a crazy afternoon for two kids."
doc3 = "My careless neighbors apparently let a stray cat stay in their garage unsupervised,
        which resulted in my favorite hat that I let them borrow being ruined."

def tf(content, term):
    tokenized_content = tokenize(content)
    term_count = tokenized_content.count(term.lower())
    return float(term_count)

doc1_tf_vector = [ tf(doc1,"the"), tf(doc1,"cat"), tf(doc1,"in"), tf(doc1,"the"), tf(doc1,"hat") ]
doc2_tf_vector = [ tf(doc2,"the"), tf(doc2,"cat"), tf(doc2,"in"), tf(doc2,"the"), tf(doc2,"hat") ]
doc3_tf_vector = [ tf(doc3,"the"), tf(doc3,"cat"), tf(doc3,"in"), tf(doc3,"the"), tf(doc3,"hat") ]

print ("labels: [the, cat, in, the, hat]")
print ("doc1_vector: [" + ", ".join(map(num2str,doc1_tf_vector)) + "]")
print ("doc2_vector: [" + ", ".join(map(num2str,doc2_tf_vector)) + "]")
print ("doc3_vector: [" + ", ".join(map(num2str,doc3_tf_vector)) + "]\n")

query = "the cat in the hat"
query_vector = np.maximum.reduce([doc1_tf_vector, doc2_tf_vector, doc3_tf_vector])
print ("query_vector: [" + ", ".join(map(num2str,query_vector)) + "]\n")

doc1_score = cos_sim(query_vector, doc1_tf_vector)
doc2_score = cos_sim(query_vector, doc2_tf_vector)
doc3_score = cos_sim(query_vector, doc3_tf_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: "
                                   + num2str(doc2_score)+ "\n doc3: " + num2str(doc3_score))
```

Results:

```
labels: [the, cat, in, the, hat]
doc1_vector: [6.0, 0.0, 4.0, 6.0, 0.0]
doc2_vector: [2.0, 2.0, 2.0, 2.0, 2.0]
doc3_vector: [0.0, 1.0, 2.0, 0.0, 1.0]

query_vector: [6.0, 2.0, 4.0, 6.0, 2.0]

Relevance Scores:
 doc1: 0.9574
 doc2: 0.9129
 doc3: 0.5
```

While we at least receive different relevance scores now for each document based upon the number of times each term matches, the ordering of the results doesn’t necessarily match our intuition about which documents are the best matches.

Intuitively, we would instead expect the following ordering:

1. **doc2**: because it is actually about the book *The Cat in the Hat*
2. **doc3**: because it matches all of the words `the`, `cat`, `in`, and `hat`
3. **doc1**: because it only matches the words `the` and `in`, even though it contains them many times

The problem here, of course, is that since every occurrence of any word is considered just as important, the more times ANY term appears, the more relevant that document becomes. In this case, `doc1` is getting the highest score, because it contains 16 total term matches (the first `the` six times, `in` four times, and the second `the` six times), yielding more total term matches than any other document.

It doesn't really make sense that a document containing a word 16 times should actually be considered 16-times as relevant, though. Usually real-world TF calculations dampen the effect of each additional occurrence of a word by calculating TF as the square root of the number of occurrences of each term. Additionally, term frequency is often also normalized relative to document length by dividing that dampened TF by the total number of terms in each document. Since longer documents are naturally more likely to contain any given term and to contain terms more often, this helps ensure that the score is normalized to the document length so that shorter and longer documents are treated equally. This final, normalized TF calculation can be seen in Figure 3.3.

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

Figure 3.3 Term Frequency Calculation. 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. The numerator dampens the additional relevance contribution of each additional occurrence of a term, while the denominator normalized that dampened frequency to the document length so that longer documents with more terms are comparable to shorter documents with less terms.

Going forward, we'll use this dampened TF calculation to ensure additional occurrences of the same term continue to improve relevance without having an outsized impact on the overall score, as it is generally better to match multiple different terms from a query than simply the same terms over and over.

With this improved TF calculation now in place, let's re-calculate our relevance ranking to see if there is any improvement in Listing 3.6.

Listing 3.6 Ranking text similarity based upon Term Frequency.

```
def tf(content, term):
    tokenized_content = tokenize(content)
    term_count = tokenized_content.count(term.lower())
    vector_length = len(tokenized_content)
    return float(np.sqrt(term_count)) / float(vector_length)

doc1_tf_vector = [ tf(doc1,"the"), tf(doc1,"cat"), tf(doc1,"in"), tf(doc1,"the"), tf(doc1,"hat") ]
doc2_tf_vector = [ tf(doc2,"the"), tf(doc2,"cat"), tf(doc2,"in"), tf(doc2,"the"), tf(doc2,"hat") ]
doc3_tf_vector = [ tf(doc3,"the"), tf(doc3,"cat"), tf(doc3,"in"), tf(doc3,"the"), tf(doc3,"hat") ]

print ("labels: [the, cat, in, the, hat]")
print ("doc1_vector: [" + ", ".join(map(num2str,doc1_tf_vector)) + "]")
print ("doc2_vector: [" + ", ".join(map(num2str,doc2_tf_vector)) + "]")
print ("doc3_vector: [" + ", ".join(map(num2str,doc3_tf_vector)) + "]\n")

query = "the cat in the hat"
query_vector = np.maximum.reduce([doc1_tf_vector, doc2_tf_vector, doc3_tf_vector])
print ("query_vector: [" + ", ".join(map(num2str,query_vector)) + "]\n")

doc1_score = cos_sim(query_vector, doc1_tf_vector)
doc2_score = cos_sim(query_vector, doc2_tf_vector)
doc3_score = cos_sim(query_vector, doc3_tf_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: "
                                   + num2str(doc2_score)+ "\n doc3: " + num2str(doc3_score))
```

Results:

```
labels: [the, cat, in, the, hat]
doc1_vector: [0.0942, 0.0, 0.0769, 0.0942, 0.0]
doc2_vector: [0.0456, 0.0456, 0.0456, 0.0456, 0.0456]
doc3_vector: [0.0, 0.0385, 0.0544, 0.0, 0.0385]

query_vector: [0.0942, 0.0456, 0.0769, 0.0942, 0.0456]

Relevance Scores:
 doc1: 0.9222
 doc2: 0.9559
 doc3: 0.5995
```

The normalized TF clearly helped, as `doc2` is now ranked the highest, as we would expect. This is mostly because of the dampening effect on number of term occurrences in `doc1` (which matched `the` and `in` so many times) so that each additional occurrrence contributes less to the feature weight than prior occurrences. Unfortunately, `doc1` is still ranked second highest, so even that wasn't enough to get the better matching `doc3` to the top.

Your intuition is probably also screaming right now, "Yeah, but nobody really cares about the words `the` and `in`. It's obvious that the words `cat` and `hat` should be given the most weight here instead!" And you would be right. Let's modify our scoring calculation to fix this oversight by introducing a new variable that takes the importance of each term into consideration.

3.1.5 Inverse Document Frequency (IDF): measuring the importance of a term in the query

While Term Frequency has proven useful at measuring how well a document matches a terms in a query, it unfortunately does little to differentiate between the importance of the terms in the query. In this section, we'll introduce a technique leveraging the significance of specific keywords based upon their frequency of occurrence across documents.

Document Frequency (DF) for a term is defined as the total number of documents in the search engine that contain the term, and it serves as a good measure for how important a term is. The intuition here is that more specific or rare words (like `cat` and `hat`) tend to be more important than common words (like `the` and `in`). The function used to calculate document frequency is shown in Figure 3.4.

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

Figure 3.4 Document Frequency Calculation. D is the set of all documents, and t is the input term. DF is simply the number of documents containing the input term, and the lower the number, the more specific and important the term is when seen in queries.

Since we would like words which are more important to get a higher score, we take an inverse of the document frequency (IDF), typically defined through function in Figure 3.5.

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

Figure 3.5 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 number, the more insignificant a term, and the higher, the more a term in a query should count toward the relevance score.

Carrrying forward our `the cat in the hat` example from the last section, a vector of IDFs would thus look as shown in Listing 3.7.

Listing 3.7 Calculating Inverse Document Frequency (IDF)

```
df_map = {"the": 9500, "cat": 100, "in":9000, "hat":50}
totalDocs = 10000                                                                    ❶

def idf(term):
    return 1 + np.log(totalDocs / (df_map[term] + 1) )                               ❷

idf_vector = np.array([idf("the"), idf("cat"), idf("in"), idf("the"), idf("hat")])   ❸

print ("labels: [the, cat, in, the, hat]\nidf_vector: " + vec2str(idf_vector))
```

1. Simulating that we have a representative sample of docs with meaningful real-world statistics
2. The IDF function, which dictates the importance of a term in the query
3. IDF is term-dependent, not document dependent, so it is the same for both queries and documents

Results:

```
labels: [the, cat, in, the, hat]
idf_vector: [1.0512, 5.5952, 1.1052, 1.0512, 6.2785]
```

These results look encouraging. The terms are all now ranked based upon their relative descriptiveness or significance/importance to the query:

1. `hat: 6.2785`,
2. `cat: 5.5952`,
3. `in: 1.1052`,
4. `the: 1.0512`

With a way to differentiate which documents are better matches for specific terms (TF) and a way to determine which specific terms should matter the most in any given query (IDF), we can now combine these two features together to generate a much more balanced relevance-ranking feature called TF-IDF.

3.1.6 TF-IDF: a balanced weighting metric for text-based relevance

We now have the two principle components of text-based relevance ranking:

- TF (measures how well a term describes a document)
- IDF (measures how important each term is)

Most search engines, and many other data science applications, leverage a combination of each of these factors as the basis for textual similarity scoring, using a variation of the function in Figure 3.6.

$$TF - IDF = TF \times IDF^2$$

Figure 3.6 TF-IDF score. Combines both the term frequency and inverse document frequency calculations together into a balanced text-ranking similarity score.

With this improved feature-weighting function in place, we can finally calculate a balanced relevance score (that weights both number of occurrences and usefulness of terms) for how well each of our documents match our query, as shown in Listing 3.8.

Listing 3.8 TF-IDF Ranking Code for the query `the cat in the hat`

```
def tf_idf(tf,idf):
    return tf * idf**2

query = "the cat in the hat"

print ("labels: [the, cat, in, the, hat]")
doc1_tfidf = [
               tf_idf(tf(doc1, "the"), idf("the")),
               tf_idf(tf(doc1, "cat"), idf("cat")),
               tf_idf(tf(doc1, "in"), idf("in")),
               tf_idf(tf(doc1, "the"), idf("the")),
               tf_idf(tf(doc1, "hat"), idf("hat"))
             ]
print("doc2_tfidf: " + vec2str(doc1_tfidf))

doc2_tfidf = [
               tf_idf(tf(doc2, "the"), idf("the")),
               tf_idf(tf(doc2, "cat"), idf("cat")),
               tf_idf(tf(doc2, "in"), idf("in")),
               tf_idf(tf(doc2, "the"), idf("the")),
               tf_idf(tf(doc2, "hat"), idf("hat"))
             ]
print("doc2_tfidf: " + vec2str(doc2_tfidf))

doc3_tfidf = [
               tf_idf(tf(doc3, "the"), idf("the")),
               tf_idf(tf(doc3, "cat"), idf("cat")),
               tf_idf(tf(doc3, "in"), idf("in")),
               tf_idf(tf(doc3, "the"), idf("the")),
               tf_idf(tf(doc3, "hat"), idf("hat"))
             ]
print("doc3_tfidf: " + vec2str(doc3_tfidf))

query_tfidf = np.maximum.reduce([doc1_tfidf, doc2_tfidf, doc3_tfidf])

doc1_relevance = cos_sim(query_tfidf,doc1_tfidf)
doc2_relevance = cos_sim(query_tfidf,doc2_tfidf)
doc3_relevance = cos_sim(query_tfidf,doc3_tfidf)

print("\nRelevance Scores:\n doc1: " + num2str(doc1_relevance) + "\n doc2: "
                                     + num2str(doc2_relevance) + "\n doc3: "
                                     + num2str(doc3_relevance))
```

Results:

```
labels: [the, cat, in, the, hat]
doc2_tfidf: [0.1041, 0.0, 0.094, 0.1041, 0.0]
doc2_tfidf: [0.0504, 1.4282, 0.0557, 0.0504, 1.7983]
doc3_tfidf: [0.0, 1.2041, 0.0664, 0.0, 1.5161]

Relevance Scores:
 doc1: 0.0758
 doc2: 0.9993
 doc3: 0.9979
```

Finally our search results make intuitive sense! `doc2` gets the highest score, since it matches the most important words the most, followed by `doc3`, which contains all the words, but not as many times, followed by `doc1`, which only contains an abundance of insignificant words.

SIDEBAR **Cosine similarity vs. TF-IDF matching score**

You may have noticed that for our query vectors, we've been using the "maximum" possible score for each feature, instead of calculating a TF-IDF value based upon the query. The reason for this is that we want documents which are better matches (i.e. more matches of each term) for the query to get higher cosine similarity scores, not documents which "contain the query words roughly the same number of times", which would occur since every word in the query occurs only once and cosine similarity only cares about the angle of vectors and not the magnitude. In practice, instead of normalizing the query to the maximum values from all documents like we have (which is challenging at scale), search engines will typically just sum the calculated values for all the feature weights to arrive at a final relevance score. This closely approximates a cosine similarity that is normalized for "best match" on each term, and it is a much easier calculation than trying to perform an actual cosine similarity calculation. The BM25 relevance calculation, which we'll introduce in the next section, will flip to this optimized method of calculating similarity.

This TF-IDF calculation is at the heart of many search engine relevance calculations, including the default similarity algorithm - called BM25 - used by both Apache Solr and Elasticsearch, which we will introduce in the next section.

3.2 Controlling the relevance calculation

In section 3.1, we showed how queries and documents can be represented as vectors, how cosine similarity can be used as a relevance function to compare queries and documents, and how TF-IDF ranking can be used to create a feature weight that balances both strength of occurrence (TF) and significance of a term (IDF) for each term in a term-based vector.

In this section, we'll show how a full relevance function can be specified and controlled in a search engine (Apache Solr), including common query capabilities, modeling queries as functions, ranking vs. filtering, and applying different kinds of boosting techniques.

In section 3.1, we used a cosine similarity calculation to determine the relevance ranking of documents relative to queries, ultimately arriving at TF-IDF as our balanced keyword-weighting metric. We discovered from Listing 3.3, however, that we can't simply use a value of `1` as the term frequency of each of our terms in the query vector, because then documents are largely ranked based upon how well they maintain a similar proportion of keywords. This is a side effect of using the cosine similarity, which only looks at the difference in *angle* between two vectors and ignores the magnitude of the distance. Specifically, for the query of `apple juice`, if the query vector was `[1 1]` then documents containing `[2 2]`, `[3 3]` … `[N N]` would all have an a cosine of `1.0` because they all have the keywords in the same relative proportion as the query, even though a document containing more occurrences like `[3 4]` intuitively seems more relevant than a document only containing `apple` and `juice` once (`[1 1]`) or twice (`[2 2]`).

To fix this, search engines typically utilize different scoring functions than pure cosine similarity for text-based ranking. In our previous examples from listing 3.4 to 3.8, we modified the weights for each term in our query vector to represent the "best possible" TF ranking for each term in any document, such that an "ideal" document would be one containing the top possible score for each keyword. This approach works well for our simple examples, but doesn't scale well in a larger production environment, so search engines instead typically take approaches which can iteratively calculate relevance one document at a time. One option is to use something like a *dot product*, which is a cosine calculation that is multiplied by the magnitude of each dimension in the vector. We won't cover that technique here, but will instead dive straight into the more common technique that is used by default in most search engines - to calculate a relevance score per keyword and then simply sum up the weights for each keyword.

Let's start with showing off the default similarity calculation leveraged by all Lucene-based search engines: BM25.

3.2.1 BM25: Lucene's default text-similarity algorithm

BM25 is the name of the default similarity algorithm in Apache Lucene, Apache Solr, Elasticsearch, Lucidworks Fusion, and other Lucene-based search engines. BM25 (short for Okapi "Best Matching" version 25) was first published in 1994, and it demonstrates improvements over standard TF-IDF cosine similarity ranking in many real-world, text-based ranking evaluations.

BM25 still uses TF-IDF at its core, but it also includes several other parameters which make it easier to control things like term frequency saturation point and document length normalization, and it sums up the weights for each matched keyword instead of calculating a cosine. The full BM25 calculation is shown in Figure 3.7.

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

Where:

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

tf(t in d) = numTermOccurrencesInDocument ½

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

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

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

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.

Figure 3.7 BM25 Scoring Function. It still leverages TF and IDF prominently, but provides more control over how much each additional occurrence of a term contributes to the score (the k parameter), and how much scores are normalized based upon document length the b parameter).

Instead of reimplementing all of this math in Python to explain it, let's now switch over to using our search engine and see how it performs the calculation. Let's start by creating a collection in Solr (Listing 3.9) and adding some documents (using our previous `the cat in the hat` example), as shown in Listing 3.10.

Listing 3.9 Creating a collection. A collection contains a specific schema and configuration for holding a group of documents, and is the unit upon which we will add documents, search, rank, and retrieve search results.

```
import sys
sys.path.append('..')
from aips import *

collection = "cat_in_the_hat"
create_collection(collection)

#Ensure the fields we need are available
upsert_text_field(collection, "title")
upsert_text_field(collection, "description")
```

Response:

```
Wiping 'cat_in_the_hat' collection
Status: Success

Creating cat_in_the_hat' collection
Status: Success

Adding 'title' field to collection
Status: Success

Adding 'description' field to collection
Status: Success
```

Listing 3.10 Adding documents to a collection

```
docs = [
    {
        "id": "doc1",
        "title": "Worst",
        "description": "The interesting thing is that the person in the wrong made the right
                        decision in the end."
    },
    {
        "id": "doc2",
        "title": "Best",
        "description": "My favorite book is the cat in the hat, which is about a crazy cat who
                        breaks into a house and creates a crazy afternoon for two kids."

    },
    {
        "id": "doc3",
        "title": "Okay",
        "description": "My neighbors let the stray cat stay in their garage, which resulted in
                        my favorite hat that I let them borrow being ruined."
    }
]
print("\nAdding Documents to '" + collection + "' collection")
response = requests.post(solr_url + collection + "/update?commit=true", json=docs).json()
print("Status: " "Success" if response["responseHeader"]["status"] == 0 else "Failure" )
```

Response:

```
Adding Documents to 'cat_in_the_hat' collection
Status: Success
```

With our documents added to the search engine, we can now issue our query and see the full BM25 scores. Listing 3.11 demonstrates how to run our search for the query `the cat in the hat` and to request the detailed relevance calculation back with each document.

Listing 3.11 Ranking by and inspecting the BM25 similarity score

```
query = "the cat in the hat"
request = {
    "query": query,
    "fields": ["id", "title", "description", "score", "[explain style=html]"],
    "params": {
      "qf": "description",
      "defType": "edismax",
      "indent": "true"
    }
}
from IPython.core.display import display,HTML
display(HTML("<br/><strong>Query: </strong><i>" + query + "</i><br/><br/><strong>Ranked Docs:</strong>")
response = str(requests.post(solr_url + collection
               + "/select", json=request).json()["response"]["docs"]).replace('\\n', '')
                 .replace(", '", ",<br/>'")
display(HTML(response))
#print(str(response))
```

Response:

```
Query: the cat in the hat
```

```
Ranked Docs:

[{'id': 'doc2',
'title': ['Best'],
'description': ['My favorite book is the cat in the hat, which is about a crazy cat who breaks
                 into a house and creates a crazy afternoon for two kids.'],
'score': 0.6823196,
'[explain]': '
    0.6823196 = sum of:
        0.15655403 = weight(description:the in 1) [SchemaSimilarity], result of:
            0.15655403 = score(freq=2.0), product of:
                2.0 = boost
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.58620685 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.19487953 = weight(description:hat in 1) [SchemaSimilarity], result of:
            0.19487953 = score(freq=1.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.4146341 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.27551934 = weight(description:cat in 1) [SchemaSimilarity], result of:
            0.27551934 = score(freq=2.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.58620685 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.05536667 = weight(description:in in 1) [SchemaSimilarity], result of:
            0.05536667 = score(freq=1.0), product of:
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.4146341 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
'}, {'id': 'doc3',
'title': ['Okay'],
'description': ['My neighbors let the stray cat stay in their garage, which resulted in my
                 favorite hat that I let them borrow being ruined.'],
'score': 0.62850046,
'[explain]': '
    0.62850046 = sum of:
        0.120666236 = weight(description:the in 2) [SchemaSimilarity], result of:
            0.120666236 = score(freq=1.0), product of:
                2.0 = boost
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.45182723 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
```

```
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.21236044 = weight(description:hat in 2) [SchemaSimilarity], result of:
            0.21236044 = score(freq=1.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.45182723 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.21236044 = weight(description:cat in 2) [SchemaSimilarity], result of:
            0.21236044 = score(freq=1.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.45182723 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.08311336 = weight(description:in in 2) [SchemaSimilarity], result of:
            0.08311336 = score(freq=2.0), product of:
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.6224256 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
'}, {'id': 'doc1',
'title': ['Worst'],
'description': ['The interesting thing is that the person in the wrong made the
                right decision in the end.'],
'score': 0.3132525,
'[explain]': '
    0.3132525 = sum of:
        0.2234835 = weight(description:the in 0) [SchemaSimilarity], result of:
            0.2234835 = score(freq=5.0), product of:
                2.0 = boost
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.83682007 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    5.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    17.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.089769006 = weight(description:in in 0) [SchemaSimilarity], result of:
            0.089769006 = score(freq=2.0), product of:
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.6722689 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    17.0 = dl, length of field
                    22.666666 = avgdl, average length of field
'}]
```

While the BM25 calculation is more complex than the TF-IDF feature weight calculations we saw in the last section, it is fundamentally still leverages TF-IDF at its core. Therefore it should be no suprise that the ranked search results actually return in the same relative order as our TF-IDF calculations from the Listing 3.8:

```
Ranked Results (Listing 3.8: TF-IDF Cosine Similarity)
 doc2: 0.998
 doc3: 0.9907
 doc1: 0.0809

Ranked Results (Listing 3.9: BM25 Similarity)
 doc2: 0.6878265
 doc3: 0.62850046
 doc1: 0.3132525
```

Our query for `the cat in the hat` can still very much be thought of as a vector of the BM25 scores for each of the terms: ["the", "cat", "in", "the", "hat"].

What may not be obvious, however, is that the feature weights for each of these terms are actually just overridable functions. Instead of thinking of our query as simply a bunch of keywords, we can think of our query as a mathematical function composed of other functions, where some of those functions take keywords as inputs and return numerical values (scores) back to be used in the relevance calculation. For example, our query could alternatively be expressed as the vector:

```
[ query("the"), query("cat"), query("in"), query("the"), query("hat") ]
```

In Solr query syntax, this would be:

```
q={!func}query("the") {!func}query("cat") {!func}query("in")
  {!func}query("the") {!func}query("hat")
```

If we execute this "functionized" version of the query, we will get the exact same relevance score as if we had just executed the query directly. Listing 3.12 shows the code to perform this version of the query.

Listing 3.12 Text Similarity using the Query Function

```
query = '{!func}query("the") {!func}query("cat") {!func}query("in")
         {!func}query("the") {!func}query("hat")'
request = {
    "query": query,
    "fields": ["id", "title", "score"],
    "params": {
      "qf": "description",
      "defType": "edismax",
      "indent": "true"
    }
}
display(HTML("<strong>Query</strong>: <i>" + query + "</i><br/><br/><strong>Results:</strong>"))
response = str(requests.post(solr_url + collection + "/select",
    json=request).json()["response"]["docs"]).replace('\\n', '').replace(", ", ",<br/>'")
display(HTML(response))
```

Response:

```
Query: {!func}query("the") {!func}query("cat") {!func}query("in")
       {!func}query("the") {!func}query("hat")

Results:
[{'id': 'doc2',
''title': ['Best'],
''score': 0.6823196},
'{'id': 'doc3',
''title': ['Okay'],
''score': 0.62850046},
'{'id': 'doc1',
''title': ['Worst'],
''score': 0.3132525}]
```

As expected, the scores are exactly the same as before - we've simply substituted in explicit functions where implicit functions were previously assumed. Once we realize that every term in a query to the search engine is actually just a configurable scoring function, it opens up tremendous possibilities for manipulating that scoring function!

3.2.2 Functions, functions, everywhere!

Now that we've seen that the relevance score for each term in our queries is simply a function operating on that term to generate a feature weight, the next logical question is "what *other* kinds of functions can I use in my queries?".

We've already encountered the `query` function (at the end of section 3.2.1), which is effectively the default calculation that executes whenever no explicit function is specified, and which uses the BM25 similarity algorithm by default.

But what if we want to consider some other features in our scoring calculation, perhaps some that are not text-based?

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.

Many of these techniques are built into specific query parsers in Solr, either through query syntax or through query parser options. For example, field boosting can be accomplished through the `qf` parameter on the `edismax` query parser. The following query, for example, provides a 10X relevancy boost for matches in the title field, and a 2.5X relevancy boost for matches in the `description` field.

```
q={!type=edismax qf="title^10 description^2.5"}the cat in the hat
```

Boosting on full phrase matching, on two-word phrases, and on three-word phrases is also a native feature of the edismax query parser:

- Boost docs containing the exact phrase `"the cat in the hat"` in the `title` field:
  ```
  q={!type=edismax qf="title description" pf=title}the cat in the hat
  ```
- Boost docs containing the two-word phrases `"the cat"`, `"cat in"`, `"in the"`, or `"the hat"` in the `title` or `description` field:
  ```
  q={!type=edismax qf="title description" pf2="title description"}the cat in the hat
  ```
- Boost docs containing the three-word phrases `"the cat in"` or `"in the hat"` in the `description` field:
  ```
  q={!type=edismax qf="title description" pf3=description}the cat in the hat
  ```

Many of the relevancy boosting techniques will require constructing your own features leveraging function queries, however. For example, if we wanted to create a query that did nothing more than boost the relevance ranking of documents geographically closest to the user running the search (relevance based on distance away), we could issue the following query:

```
q=*:*&
 sort=geodist(location, $user_latitude, $user_longitude) asc&
 user_latitude=33.748&
 user_longitude=-84.39
```

That last query is using the `sort` parameter to strictly order documents by the calculated value from the `geodist` function. This works great if we want to order results by a single feature, but what if we want to construct a more nuanced sort based upon multiple features? To accomplish this, we will update our query to include each of these function in each document’s relevance calculation, and then sort by the relevance score:

```
q={!func}scale(query($keywords),0,25)
  {!func}recip(geodist($lat_long_field,$user_latitude,$user_longitude),1,25,1)
  {!func}recip(ms(NOW/HOUR,modify_date),3.16e-11,25,1)
  {!func}scale(popularity,0,25)
  &keywords="basketball"&
  lat_long_field=location&
  user_latitude=33.748&
  user_longitude=-84.391
```

That query does a few interesting things:

- It constructs a query vector containing four features: BM25 Keywords relevance score (higher is better), geo distance (lower is better), publication date (newer is better), and popularity (higher is better).

- Each of the feature values is scaled between `0` and `25` so that they are all comparable, with the best keyword/geo/publication date/popularity score getting a score of `25`, and the worst getting a score close to `0`.
- Thus a "perfect score" would add up to `100` (25 + 25 + 25 + 25), and the worst score would be approximately `0`.
- Since the relative contribution of `25` is specified as part of the query for each function, we can easily change the weights of any feature on the fly to give preference to certain features in the final relevance calculation.

With the last query, we have now fully taken the relevance calculation into our own hand by modeling our relevance features and giving them weights. While this is very powerful, it still requires significant manual effort and testing to figure out which features matter for a given domain, and what their relative weights should be. In chapter 10 we will walk through building Machine-learned Ranking models to automatically make those decisions for us (a process known as "Learning to Rank"). For now, however, our goal was to ensure you understood the mechanics of modeling features in query vectors, and controlling their weights.

> **SIDEBAR** **Deeper dives on function queries**
>
> If you'd like to learn more about how to utilize function queries, I recommend reviewing chapter 15 of *Solr in Action* (Trey Grainger and Timothy Potter, Manning, 2014) on "Complex Query Operations"", for a much fuller exposition. For a full list of available function queries in Solr, you can also check out the documentation in the function query section of the Solr Reference Guide.

While we've seen the power of utilizing functions as features in our queries, thusfar all of our examples have been what are called "additive" boosts, where the sum of the values of each function calculation comprise the final relevance score. It is also frequently useful to combine functions in a fuzzier, more flexible way through "multiplicative" boosts, which we'll cover in the next section.

3.2.3 Choosing multiplicative vs. additive boosting for relevance functions

One last topic to address concerning how we control our relevance functions is the idea of using multiplicative vs. additive boosting of relevance features.

In all of our examples to this point, we have added multiple features together into our query vector to contribute to the score. For example, the following queries will all yield equivalent relevance calculations assuming they are all filtered down to the same result set (i.e. `fq=the cat in the hat`):

```
Text query (score + filter)
  q=the cat in the hat

Function Query (score only, no filter)
```

```
  q={!func}query("the cat in the hat")

Multiple Function Queries (score only, no filter)
  q={!func}query("the")
    {!func}query("cat")
    {!func}query("in")
    {!func}query("the")
    {!func}query("hat")

Boost Query (score only, no filter)
  q=*:*&bq=the cat in the hat
```

The kind of relevance boostin in each of these examples is known as *additive boosting*, and maps well to our concept of a query as nothing more than a vector of features that needs to have its similarity compared across documents.

In many cases, however, we are likely to want to specify what are known as *multiplicative boosts* as part of our relevance calculations. Instead of inserting additional features into our vector, multiplicative boosts increase the relevance of an entire document by multiplying the document's full score by some calculated value.

For example, if we wanted to query for `the cat in the hat`, but wanted the popularity of documents (those with a higher number in the popularity field) to have a less constrained effect, we can't easily do this by just adding another feature into our query vector - at least not without modifying the weights of all the other features, plus any additional normalization that may be applied by the BM25 ranking function. If we wanted to apply multiple boosts like this (for example, boosting both on popularity AND on publication date), then the option of modeling this as an additive boost becomes unreasonably complex and harder to control.

In section 1.2.2, we were able to successfully utilize additive boosting by explicitly constraining the minimum and maximum values for the features in our query vector so that each feature provides a known contribution to the overall relevance function.

Multiplicative boosting enables boosts to "pile up" on each other, however, because each of the boosts is multiplied against the overall relevance score for the document, resulting in a much fuzzier match and preventing the need for the kind of tight constraints we had to supply for our additive boost example.

To supply a multiplicative boost, you can either use the `boost` query parser (`{!boost}`) in your query vector or, if you are using the `edismax` query parser, the simplified `boost` query param. For example, to multiply a document's relevance score by ten times the value in the `popularity` field, you would do either:

```
q=the cat in the hat&
  defType=edismax&
  boost=mul(popularity,10)
```

OR

```
`q={!boost b=mul(popularity,10)}the cat in the hat
```

In general, multiplicative boosts enable you greater flexibility to combine different relevance features without having to explicitly pre-define and exact relevance formula accounting for every potential contributing factor. On the other hand, this flexibility can lead to unexpected consequences if the multiplicative boost values for particular features get too high and overshadow other features. Additive boosts can be a pain to manage, because you have to explicitly scale them so that they can be combined together and maintain a predictible contribution to the overall score, but once you've done this you maintain tight control over the relevance scoring calculation and range of scores.

Both additive and multiplicative boosts can be useful in different scenarios, so it's best to consider the problem at hand and experiment with what gets you the best results. We've now covered the major ways to control relevance ranking in the search engine, but matching and filtering of documents can often be just as important, so we'll cover them in the next section.

3.2.4 Differentiating matching (filtering) vs. ranking (scoring) of documents

We opened up this chapter by stating that search engines fundamentally do three things: ingest content, return content matching incoming queries, and sort the returned content based upon some measure of how well it matches the query. Thusfar, we've only talked about the third capability (relevance ranking), however.

We've only really spoken of queries as feature vectors, and we've only discussed relevance ranking as a process of either calculating a cosine similarity or of adding up document scores for each each feature (keyword or function) in the query. This may seem a bit strange, since most search books start with coverage of matching keywords in the search engine's inverted index to filter result sets well before discussing relevance.

We've delayed the discussion of filtering results until this point on purpose, however, in order to focus on relevance and the idea of queries and documents as vectors of features to be compared and ranked based upon similarity.

A pre-requisite for comparing queries with documents, of course, is that the search engine has already ingested some content from which those features are derived. Once content is ingested, there are then two steps involved in executing a query:

- *Matching*: Filtering results to a known set of possible answers
- *Ranking*: Ordering all of the possible answers by relevance

In reality, we can often completely skip step 1 (matching/filtering) and still see the exact same results on page one (and for many pages), since the most relevant results should generally show up first. If you think back to chapter 2, we even saw some vector scoring calculations (comparing feature vectors for food items - i.e. "apple juice" vs. "donut") where we would have been unable to filter results at all, and we instead had to first score every document to determine which ones to return based upon relevance alone. In this scenario, we didn't even have any keywords or other attributes which could be leveraged as a filter.

So if the initial matching phase is effectively optional, then why do it at all? One obvious answer is that it is significant performance optimization. Instead of iterating through every single document and calculating a relevance score, we can greatly speed up both our relevance calculations and the overall response time of our search engine by first filtering the initial result set to a smaller set of documents which are logical matches.

Of course, there are also additional benefits to filtering our results sets, in that the total document count is reduced and we can provide analytics (facets) on the set of logically-matching documents in order to help the user further explore and refine their results set. Finally, there are plenty of scenarios where "having logical matches" should actually be considered among the most important features in the ranking function, and thus simply filtering on logical matches up-front can greatly simplify the relevance calculation. We'll discuss these tradeoffs in the next section.

3.2.5 Logical matching: weighting the relationships between terms in a query

We just mentioned that filtering results before scoring them is primarily a performance optimization and that the first few pages of search results would likely look the same regardless of whether you filter the results or just do relevance ranking.

This only holds true, however, if your relevance function successfully contains features which already appropriately boost better logical matches. For example, consider the difference between expectations for the following queries:

1. `"statue of liberty"`
2. `statue AND of AND liberty`
3. `statue OR of OR liberty`
4. `statue of liberty`

From a logical matching standpoint, the first query will be very precise, only matching documents contain the *exact* phrase `statue of liberty`. The second query will only match documents containing all of the terms `statue`, `of`, and `liberty`, but not necessarily as a phrase. The third query will match any document containing any of the three terms, which means documents *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.

In theory, if phrase boosting is turned on as a feature then documents containing the full phrase should rank highest, followed by documents containing all terms, followed by documents containing any of the words. Assuming that happens, then you should see the same ordering of results regardless of whether you filter them to logical, Boolean matches, or whether you only sort based on a relevance function.

In practice, though, users often consider the logical structure of their queries to be highly relevant to the documents they expect to see, so respecting this logical structure and filtering *before* ranking allows you to remove results which users' queries indicate are safe to remove.

Sometimes the logical structure of user queries is ambiguous, however, such as with our fourth example of the query `statue of liberty`. Does this logically mean `statue AND of AND liberty`, `statue OR of OR liberty`, or something more nuanced like `(statue AND OF) OR (statue AND liberty) OR (of AND liberty)`, which essentially means "match at least two of three terms". Using the "minimum match" (`mm`) parameter in Solr enables you to control these kinds of matching thresholds easily, even on a per-query basis:

- 100% of query terms must match (equivalent to `statue AND of AND liberty`):

```
q=statue of liberty&
 mm=100%
```

- At least one query term + 0% of additional query terms must match (equivalent to `statue OR of OR liberty`):

```
q=statue of liberty&
 mm=0%
```

- At least two query terms must match (equivalent to `(statue AND of) OR (statue AND liberty) OR (of AND liberty)`):

```
q=statue of liberty&
 mm=2
```

The `mm` parameter in Solr allows you to specify a minimum match threshold as either a percentage (0% to 100%) of terms, a number of terms (1 to N terms), or as a step function like `mm=2<-30% 5<3`, which means "All terms are required if there are less than 2 terms, up to 30% of terms can be missing if there are less than 5 terms, and at least 3 terms must exist if there are 5 or more terms. The `mm` parameter works with the `edismax` query parser, which is the primary query parser we will use for text-matching queries in this book. You can consult the edismax section of the Solr Reference Guide for more details on how to fine-tune your logical matching rules with these minimum match capabilities.

When thinking about constructing relevance functions, the idea of filtering and scoring can often get mixed up, particularly since Solr itself mixes concerns in the query parameter. We'll attempt to separate these concerns in the next section.

3.2.6 Separating concerns: filtering vs. scoring

In section X.X.X we differentiated between the ideas of "matching" and "ranking". Matching of results is logical and is implemented by filtering search results down to a subset of documents, whereas ranking of results is qualitative, and is implemented by scoring all documents relative to the query and then sorting them by that calculated score. In this section, we'll some cover techniques to provide maximum flexibility in controlling matching and ranking by cleanly separating out the concerns of filtering and scoring.

Solr has two primary ways to control filtering and scoring, the "query" (`q` parameter) and the "filters" (zero or more `fq` parameters). Consider the following request:

```
q=the cat in the hat&
 fq=category:books&
 fq=audience:kid&
 defType=edismax&
 mm=100%&
 qf=description
```

In this query, Solr is being instructed to filter the possible result set down to only documents with a value of `books` in the `category` field and also a value of `kid` in the `audience` field. In addition to those filters, however, the query itself also acts as a filter, so the result set gets further filtered down to only documents also containing (100%) of the values `the`, `cat`, `in`, and `hat` in the `description` field.

The logical difference between the `q` and `fq` parameters is that the `fq` only acts as a filter, whereas the `q` acts as *both* a filter and feature vector for relevance ranking. This dual use of the `q` parameter is helpful default behavior for queries, but mixing the concerns of filtering and scoring in the same parameter can be suboptimal for more advanced queries, especially if we're simply trying to manipulate the relevance calculation and not arbitrarily removing results from our document set.

There are a few ways to address this:

1. Model the `q` parameter as a function (functions only count toward relevance and do not filter):

   ```
   q={!func}query("{!type=edismax qf=description mm=100% v=$query}")&
     fq={!cache=false v=$query}&
     query=the cat in the hat
   ```

2. Make your query match all documents (no filtering or scoring) and apply a Boost Query (`bq`) parameter to incluence relevance without scoring:

   ```
   q=*:*
     &bq={!type=edismax qf=description mm=100% v=$query}&
     fq={!cache=false v=$query}&
     query=the cat in the hat
   ```

Between these three parameters, `q` both filters and then boosts based upon relevance, `fq` only filters, and `bq` only boosts. As such, both of these approaches are logically equivalent, but we'll go with option 2 throughout this book since it is a bit cleaner to use the dedicated `bq` parameter which was designed to contribute toward the relevance calculation without filtering.

You may have noticed that both versions of the query also contain a filter query which filters on the query:

```
q=*:*
  &bq={!type=edismax qf=description mm=100% v=$query}&
  fq={!cache=false v=$query}&
  query=the cat in the hat
```

Since the `q` parameter intentionally no longer filters our search results, this `fq` parameter is now required if we still want to filter to the user-entered query. By constructing our queries this way, we allow our relevance function to be entirely separated from the filtering logic, which often makes it much easier to construct complex ranking functions. The special `cache=false` parameter there is used to turn off caching of the filter. Caching of filters is turned on by default in Solr since filters tend to be reused often across requests. Since the `$query` parameter is user-entered and wildly variable in this case (not frequently reused across requests), it doesn't make sense to pollute the search engine's caches with these values. If you try to filter on user-entered queries without turning the cache off, it will waste system resources and likely slow down your search engine.

The overarching theme here is that it is possible to cleanly separate logical filtering from ranking features in order to maintain full control and flexibility over your search results. While going through this effort may be overkill for simple text-based ranking, separating out these concerns becomes critical when attempting to build out more sophisticated ranking functions, as we'll see in the coming chapters.

Now that we understand the mechanics of how construct these kinds of purpose-built ranking functions, let's wrap up this chapter with a brief discussion of how to apply these techniques to implement user and domain-specific relevance ranking.

3.3 Implementing user and domain-specific relevance ranking

In section 3.2, we walked through how we can easily and dynamically modify the parameters of our query-to-document similarity algorithm, including passing in our own functions as features which contribute to the score, in addition to just text-based relevance ranking.

While text-based relevance ranking using BM25, TF-IDF vector cosine similarity, or some other kind of statistics-based approach on word occurrences can provide decent "general" search relevance out of the box, it can't hold its own against good domain-specific relevance factors. For example, if you travel to Boston, Massachusetts in the United States and you open up a restaurant finding app on your phone and search for `hamburger`, you probably won't be very happy with the search engine if the top answers are for `Five Guys Burgers and Fries in Austin, Texas`, `McDonald's in Anchorage, Alaska`, and `Hungry Jack in Sydney, Australia`. Even if these were the best keyword matches and even if they were the most popular results for the query, the expectation from users is that you will consider "distance from my location" as one of the most (if not *the* most) important of factors in the relevance determination.

In fact, most people would probably intuitively be able to tell you that the following attributes matter the most to them within these various domains:

- Restaurant Search: `geographical proximity`, `user-specific dietary restrictions`, `user-specific taste preferences`, and `price range`
- News Search: `freshness` (date), `popularity`, and `geographical area`
- Ecommerce: `liklihood of conversion` (click-through, add-to-cart, and/or purchase)
- Movie Search: `name match` (title, actor, etc.), `popularity` of document, `release date`, `critic review score`
- Job Search: `job title`, `job level`, `compensation range`, `geographical proximity`, `job industry`
- Web Search: `keyword match on page`, `popularity of page`, `popularity of website`, `location of match on page` (in title, header, body, etc.), `quality of page` (duplicate content, spammy content, etc.)

Obviously these are just examples, and you can probably think of many more factors that would matter and impact relevance for each use case. The point, however, is that every search engine and domain has unique features which need to be considered to deliver an optimal search experience.

Instead of walking through how to manually construct queries for each of the scenarios above (hopefully the last section gave you the tools you need to figure that out), this book will instead focus on showing you how to use machine learning to build an AI-powered search engine that can automatically learn how to both generate and weight these kinds of features to determine the optimal relevance algorithm.

This chapter has barely scratched the surface on the myriad ways that you can control and manipulate the matching and ranking functions in order to return the best content and domain-specific relevance-ranked search results. An entire profession exists - called a "relevance engineer" - that is dedicated to tuning search relevancy using techniques like this within many organizations. If you'd like to dive deeper, I highly recommend the book *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016), which provides an expert guide on this kind of relevance engineering.

The purpose of this chapter was to give you the base knowledge and tools you'll need in the coming chapters to impact relevance ranking as we begin integrating more automated machine learning techniques into our search applications. We'll begin applying all of this in our next chapter on crowdsourced relevance and signal boosting.

3.4 Summary

- We can map queries and documents to dense or sparse numerical vectors and then assign a relevance rank to documents based upon either a cosine similarity between the vectors or some other similarity scoring calculation
- Leveraging TF-IDF or the BM25 similarity calculations (based upon TF-IDF) for our text similarity scores provides us with a more meaningful measure of feature (keyword) importance in our queries and documents, enabling improved text ranking over just looking at term matches alone.
- Text similarity scoring is just one of many kinds of functions we can invoke as a feature within our queries for relevance ranking. We can inject functions within our queries along with keyword matching and scoring, as each keyword phrase is effectively just a ranking function anyway.
- It is useful to separate "filtering" and "scoring" as separate concerns in order to have better control when specifying our own ranking functions.
- In order to optimize relevance, we need to create domain-specific relevance functions and also leverage user-specific features instead of relying just on keyword matching and ranking. We'll focus on doing this through automated learning approaches throughout the rest of this book.

4 Crowdsourced Relevance

This chapter covers:

- How to harness your users' collective insights to improve the relevance of your search platform
- Collecting and working with user behavioral signals
- Leveraging Reflected Intelligence to create self-tuning models like signals boosting, collaborative recommendations and personalization, and machine-learned ranking
- Building an end-to-end signals boosting model
- Crowdsourced learning from content-based signals

4.1 Intro to Crowdsourced relevance

In chapter one, we introduced the dimensions of user intent as "content understanding", "user understanding", and "domain understanding". In order to create an optimal AI-powered search platform, we need to be able to combine each of these contexts to understand our users' query intent. The question, though, is how do we derive these understandings?

Many different sources of information may exist from which we can learn: documents, databases, internal knowledge graphs, user behavior, domain experts, and so on. Some organizations have teams that manually tag documents with topics or categories, and some even outsource these tasks using tools like Amazon Mechanical Turk, which allows them to crowdsource answers from people all around the world. For identifying malicious behavior or errors on websites, companies will often allow their users to report problems and even suggest corrections. All of these are examples of crowdsourcing - relying upon the inputs from many people to learn new information.

When it comes to search relevance, crowdsourcing can play a vital role, though it is usually important not to annoy your valued customers by constantly asking them for help. Fortunately, it is often possible to learn from your users implicitly based upon their behaviors without having to bother them by explicitly asking for input. For example, if you are trying to find the best documents to return for a given search, why not examine your logs to determine which documents other users most often clicked on in response to the same query?

4.2 Working with User Signals

Every time a customer takes an action - whether it be issuing a query, clicking on a result, purchasing a product, or otherwise taking some action on the search results - this provides a signal of that user's intent. We can log each of these signals and then process them in order to learn insights about each user, about groups of users, and about an entire user base. This section will introduce you to the power of leveraging user signals, introduce a sample ecommerce dataset we will use throughout the book, and walk you through the mechanics of collecting, storing, and processing the user signals.

4.2.1 Signals vs. Content

When building search engines, we have two high-level types of data - "content" and "signals". Most of the content we deal with is in the form of documents. Documents can represent web pages, product listings, computer files, images, facts, or any other type of information that we may want to search through. Content documents usually contain fields with some kind of text through which we can search and find relationships, along with additional fields representing other attributes related to the content (author, size, color, dates, and so on). The defining characteristic of these content documents is that they contain the information through which people are searching (and hopefully also the answers for which they are searching). In addition to traditional documents, however, many other kinds of content can be incorporated into a search experience. An externally-built knowledge graph, a list of entities (people, places, things), customer or employee-created comments, tags, or attributes that are added to the documents, and so on, are all forms of content.

"Signals", on the other hand, reflect how users engage with content. When someone issues a query, they receive a set of documents with content back. Perhaps they click on a result, add it to their shopping cart, bookmark the document, or take other similar actions. We refer to these interactions as signals, and their defining characteristic is that they provide external insights that can be used to understand how users want to interact with content. Of course, those signals can also later be added to documents along with the pre-existing content, and if you are building an application that allows for searching through the signals, those signals actually then become new content for that application. Notwithstanding that in some cases signals can also be treated as content depending on the use case, the point here is that there are two key sources of information we can use to improve search: the attributes of the items being searched upon (content), and the observed user interactions with the items (signals).

For many of the important tasks we undertake when building AI-powered search, we can derive similar outcomes by using either the content or the signals, but they give us two different perspectives of relevance. In ideal cases, we can actually bring in both of those perspectives to build an even smarter system, but it is useful to understand the strengths and weaknesses of each approach to understand how to best leverage them.

As an example, if we are trying to find a synonym for the word "driver", we can look through all the text content for words that commonly appear in the same documents. We may, in this case, find words in priority order (by percentage of documents they appear within) like "taxi" (40%), "car" (35%), "golf" (15%), "club" (12%), "printer"(3%), "linux" (3%), and "windows" (1%). Similarly, we could look at our collected signals for all users who searched for *driver* and aggregate the most common other keywords for which they searched, and find similar words in priority order like "screwdriver" (50%), "printer" (30%), "windows" (25%), "mac" (15%), "golf" (2%), "club" (2%). The lists derived from signals versus content might be similar, or they could look very different. The content-based approach tells us the most represented meanings within our documents, whereas the signals-based approach tells us the most represented meanings being looked for by our *users*.

Since our end goal is to present users what they are looking for, we would tend to favor the signals-derived meanings over the content-derived meanings in most cases. However, what if we don't actually have good content that maps to the signals-derived meaning? Do we use the content-derived meaning, or do we try to suggest other related searches based upon the signals data? What if we don't have enough signals or if the signals data is not very clean? Can we somehow clean up the signals-derived data using the content-derived data?

We run into similar questions with recommendations: content-based recommendations leverage attributes in documents but don't understand users, whereas signals-based recommendations don't understand content attributes and don't work at all on items which don't have sufficient interactions. Content-based recommendations may recommend on features that are unimportant to users, whereas signals-based recommendations may create self-reinforcing loops where people only interact with items they are recommended, and then only those items get recommended because they were the only ones with which users interacted.

Ideally, we want to create a balanced system that can leverage the best of both content-derived and signals-derived intelligence. While this chapter will focus primarily on signals-derived, crowdsourced intelligence, a major goal of this book is to show how to balance and combine both content-based and signals-based approaches to yield an optimal AI-powered search experience.

4.2.2 Setting up our product and signals datasets (RetroTech)

We will leverage various datasets throughout this book as we explore different use cases, but it is also valuable to have a consistent example that we can build on as we progress. Web search and Ecommerce are two of the most universally-recognizable use cases for AI-powered search today, and we felt that of the two, Ecommerce presented the best opportunity for exploring the widest variety of AI-powered search techniques. Ecommerce examples will most likely also map more easily to the real-world use cases being delivered by the largest portion of the likely readers of this book. It it worth noting that most techniques in this book apply across use cases - web search, enterprise search, site search, desktop search, mail search, ediscovery, job search, product search, support portal search - you name it. The deciding factor on when to use any given technique typically relates more to the volume and variety of content and signals than it does to the particular use case.

With that said, let’s introduce our example use case and dataset: RetroTech!

THE RETROTECH USE CASE

With aggressive competition among retailers selling the latest and greatest electronics, multimedia, and tech products, it is hard for a small online business to compete. However, there is a niche, but emerging segment of the population that chooses to avoid the latest and greatest products and to instead fall back to the familiar technology of decades past. The RetroTech company was launched to meet the needs of this unique group of consumers, offering vintage products that may be hard to find on today’s shelves.

LOADING THE PRODUCT CATALOG

The RetroTech website currently has around 50,000 products available for sale, so we’ll need to load those into our search engine to get started. If you built the *AI-Powered Search* code base to run the chapter 3 examples, then your search engine is already up and running. If you haven’t yet done that, the instructions for building and running all of the book’s examples are found in Appendix A, which you can run through now to get setup.

With your search engine up and running, the next thing we need to do is download the Retrotech dataset that accompanies this book. The dataset includes two CSV files, one containing all of Retrotech’s products, and another containing one year of signals data from Retrotech’s users. Listing 4.1 shows a few rows of the product catalog dataset to get you familiar with the format.

Listing 4.1 Exploring the RetroTech product catalog

```
"upc","name","manufacturer","shortDescription","longDescription"
"096009010836","Fists of Bruce Lee - Dolby - DVD",\N,\N,\N
"043396061965","The Professional - Widescreen Uncut - DVD",\N,\N,\N
"085391862024","Pokemon the Movie: 2000 - DVD",\N,\N,\N
"067003016025","Summerbreeze - CD","Nettwerk",\N,\N
"731454813822","Back for the First Time [PA] - CD","Def Jam South",\N,\N
"024543008200","Big Momma's House - Widescreen - DVD",\N,\N,\N
"031398751823","Kids - DVD",\N,\N,\N
"037628413929","20 Grandes Exitos - CD","Sony Discos Inc.",\N,\N
"060768972223","Power Of Trinity (Box) - CD","Sanctuary Records",\N,\N
```

You can see that products are identified by a UPC (Universal Product Code) number and then have a name, a manufacturer, and both a short description (used as a teaser in search results) and a long description (the full description used on product details pages).

Since our goal is to search for products, our next step will be to send them to the search engine to be indexed. To enable search on our RetroTech product catalog, let's run the document indexing code in Listing 4.2 to send the product documents to the search engine.

Listing 4.2 Send product documents to the search engine

```
#Create Products Collection
products_collection="products"
create_collection(products_collection)

#Modify Schema to make some fields explicitly searchable by keyword
upsert_text_field(products_collection, "upc")
upsert_text_field(products_collection, "name")
upsert_text_field(products_collection, "longDescription")
upsert_text_field(products_collection, "manufacturer")

print("Loading Products...")
csvFile = "../data/retrotech/products.csv"
product_update_opts={"zkhost": "aips-zk", "collection": products_collection,
                     "gen_uniq_key": "true", "commit_within": "5000"}
csvDF = spark.read.format("com.databricks.spark.csv").option("header",
                     "true").option("inferSchema", "true").load(csvFile)
csvDF.write.format("solr").options(**product_update_opts).mode("overwrite").save()
print("Products Schema: ")
csvDF.printSchema()
print("Status: Success")
```

Results:

```
Wiping 'products' collection
Creating products' collection
Status: Success
Adding 'upc' field to collection
Status: Success
Adding 'name' field to collection
Status: Success
Adding 'longDescription' field to collection
Status: Success
Adding 'manufacturer' field to collection
Status: Success
Loading Products...
Products Schema:
root
```

```
|-- upc: long (nullable = true)
|-- name: string (nullable = true)
|-- manufacturer: string (nullable = true)
|-- shortDescription: string (nullable = true)
|-- longDescription: string (nullable = true)

Status: Success
```

Finally, to verify that the documents are now indexed and searchable, let’s run an example keyword search. Listing 4.3 shows an example search for "ipod", a true classic device!

Listing 4.3 Running a search on the product catalog

```
query = "ipod"

collection = "products"
request = {
    "query": query,
    "fields": ["upc", "name", "manufacturer", "score"],
    "limit": 5,
    "params": {
      "qf": "name manufacturer longDescription",
      "defType": "edismax",
      "sort": "score desc, upc asc"
    }
}

search_results = requests.post(solr_url + collection + "/select",
                               json=request).json()["response"]["docs"]
display(HTML(render_search_results(query, search_results)))
```

The results of the *ipod* search from Listing 4.3 are shown in figure 4.1, demonstrating that our products are now indexed and searchable.

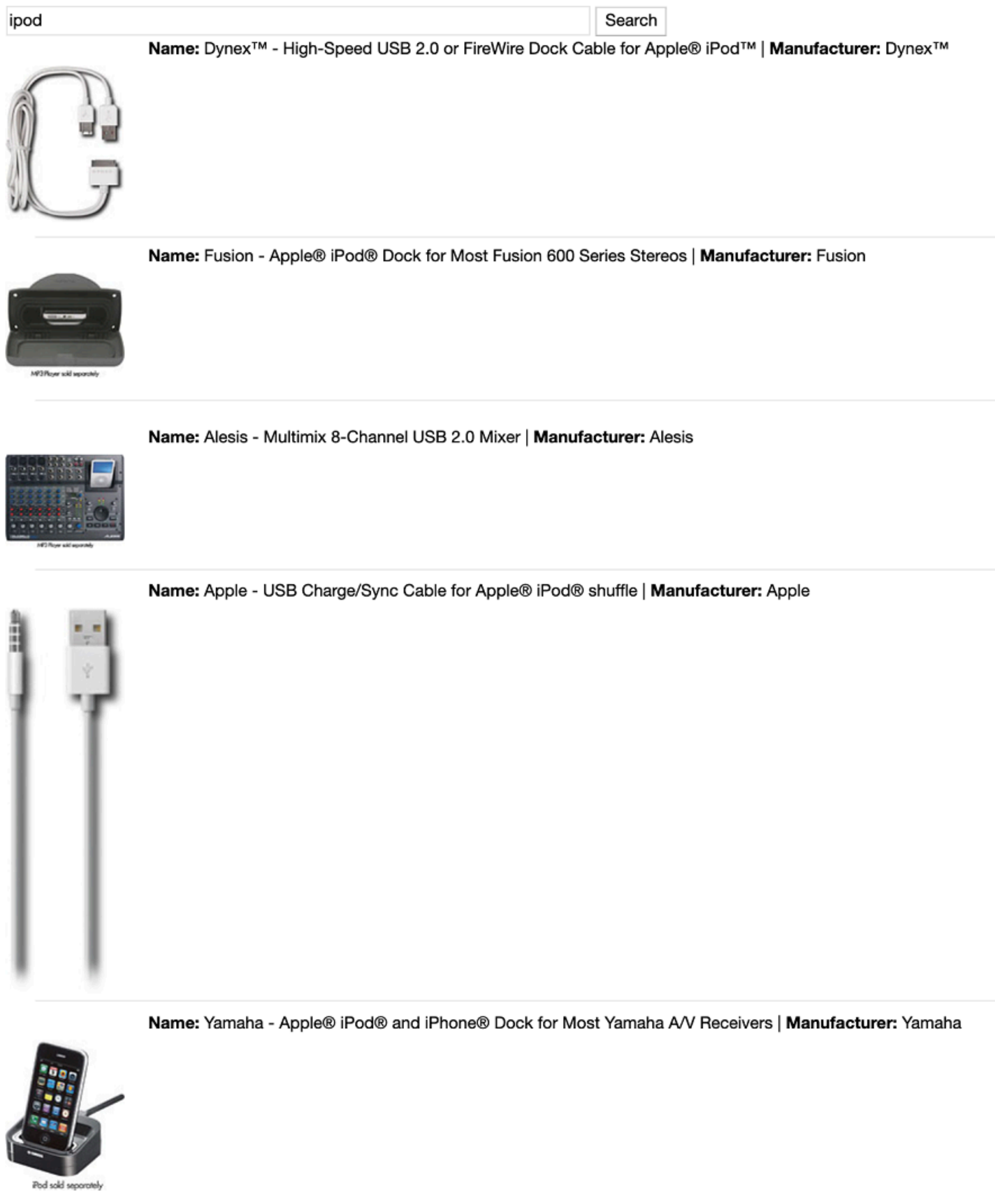


Figure 4.1 Product Search Results. We can see that the product catalog has been indexed and a query for ipod now returns search results.

With our products indexed, we now have an out of the box "keyword matching" search engine. We'll use this as our base and start introducing more intelligent AI-powered search features throughout the rest of the book. Our next step will be to introduce our signals data.

LOADING THE SIGNALS DATA

Since RetroTech is running on your computer, you're not going to have real users visiting, running searches, clicking and purchasing, and otherwise generating signals. As such, we've generated a dataset for you to use that approximates the kind of signal activity you'd expect to see in similar real-world datasets.

For simplicity, we will store our signals in the search engine to enable them to be leveraged both in real-time search scenarios and for external processing. Running Listing 4.4 will simulate and index some sample signals that we can leverage throughout the rest of the chapter.

Listing 4.4 Indexing the User Signals Dataset

```
#Create Signals Collection
signals_collection="signals"
create_collection(signals_collection)

print("Loading Signals...")
csvFile = "../data/retrotech/signals.csv"
signals_update_opts={"zkhost": "aips-zk", "collection": signals_collection, "gen_uniq_key":
                     "true", "commit_within": "5000"}
csvDF = spark.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema",
                          "true").load(csvFile)
csvDF.write.format("solr").options(**signals_update_opts).mode("overwrite").save()
print("Signals Schema: ")
csvDF.printSchema()
print("Status: Success")
```

Results:

```
Wiping 'signals' collection
Creating signals' collection
Status: Success
Loading Signals...
Signals Schema:
root
 |-- query_id: string (nullable = true)
 |-- user: string (nullable = true)
 |-- type: string (nullable = true)
 |-- target: string (nullable = true)
 |-- signal_time: timestamp (nullable = true)

Status: Success
```

With our Retrotech product and signals data all loaded, we soon begin exploring ways in which we can use the signals data to enhance our search relevance. Before we dive into these crowdsourced relevance techniques, though, let's first explore the signals data a bit so we can understand how signals are structured, used, and collected in real-world systems.

4.2.3 Exploring the signals data

Different types of signals have different attributes which need to be recorded. For example, for a "query" signal, we would want to record the user's keywords. For a "click" signal we would want to record which document was clicked upon, as well as which query resulted in the click. For later analysis, we'd also want to record which documents were returned to and possibly viewed by a user after a query.

In order to make things more extensible and avoid custom code for every new signal type, we've adopted a generic format for representing signals in this book. This will likely differ from how you log your signals, but as long as you can subsequently map your signals into this format then all of the code in this book should work without requiring use-case specific modifications.

The signals format we will use in this book is as follows:

- **query_id**: a unique id for the query signal that originated this signal.
- **user**: an identifier representing a specific user of the search engine
- **type**: what kind of signal (query, click, purchase, etc.)
- **target**: the content to which the signal at this *signal_time* applies.
- **signal_time**: the date and time the signal occurred

As an example, assume a user performed the following sequence of actions:

1. issued a query for `ipad` and had three documents (`doc1`, `doc2`, and `doc3`) returned.
2. clicked on `doc1`.
3. went back and clicked on `doc3`.
4. added `doc3` to the shopping cart.
5. went back and searched for `ipad cover` and had two documents returned (`doc4`, `doc5`).
6. clicked on `doc4`.
7. added `doc4` to the shopping cart.
8. purchased the items in the shopping cart (`doc3`, `doc4`).

These interactions would result in the following signals:

Table 4.1 Example signals format

query_id	user	type	target	signal_time
1	u123	query	ipad	2020-05-01-09:00:00
1	u123	results	doc1,doc2,doc3	2020-05-01-09:00:00
1	u123	click	doc1	2020-05-01-09:00:10
1	u123	click	doc3	2020-05-01-09:00:29
1	u123	add-to-cart	doc1	2020-05-01-09:03:40
2	u123	query	ipad cover	2020-05-01-09:04:00
2	u123	results	doc4,doc5	2020-05-01-09:04:00
2	u123	click	doc4	2020-05-01-09:04:40
2	u123	add-to-cart	doc4	2020-05-01-09:05:50
1	u123	purchase	doc3	2020-05-01-09:07:15
2	u123	purchase	doc4	2020-05-01-09:07:15

A few things to note about the signals format:

1. **The `query` type and the `results` type signal are broken up into separate signals.** This isn't necessary, as they occur at the same time, but this allows us to keep the table structure consistent and not have to add an extra `results` column that only applies to the `query` signal. Also, if the user hits the "next page" link or scrolls down the page and sees additional results, this structure will allow us to append a new signal at the new time for those results being returned without having to go back and modify the original signal.
2. **Every signal ties back to the `query_id` of the original `query` signal that started the series of content interactions.** The `query_id` is not just a reference to the keywords entered by the user, but is instead a reference to the specific `query` signal identifying a time-stamped instance of the user's query. Because results for the same query keywords can change over time, this enables us to do more sophisticated processing of how users reacted to the specific set of results they were shown for a query.
3. **Most signal types contain one item in the `target`, but the `results` type contains an ordered list of documents.** The reason is that it's important to preserve the exact ordering of the search results, so the "target" is really an ordered list of documents in this case instead of independent list of documents. The order of results will matter for some algorithms we introduce later in the book to measure relevance.
4. **The checkout resulted in a separate `purchase` signal for each item instead of just one `checkout` signal.** This is because we need to track that the two items that were purchased originated from separate queries. You could, of couse, also add a `checkout` signal type if you wanted to track the transaction, and possibly list the two purchases as the `target`, but this is superfluous for our needs in this book, so we'll avoid this added complexity.

With these raw signals available as our building blocks, we can now start thinking about how to link the signals together to begin learning about our users and their interests. In the next section, we'll discuss ways to model users, sessions, and requests within our search platform.

4.2.4 Modeling users, sessions, and requests

In the last section, we took a look at the structure of user signals as a list of independent interactions tied back to an original query. We assumed that a "user" was present with a unique id, but how does one identify and track a unique user? Furthermore, once you have identified how to track unique users, what is the best way to break their interactions up into sessions to understand when their context may have changed?

The concept of a user, in web search, can be very fluid. If your search engine has authenticated (logged-in) users, then you already have some kind of an internal user id to track them. If your search engine supports unauthenticated access or is publicly available, however, then you will have many running searches with no formal user id. That does not mean you can’t track them, however, just that it requires a more fluid interpretation of what a "user" means. The reason we want to track a user is so that we can related different signals together to learn interaction patterns, and without some shared identifier representing that the same user is issuing multiple requests, it is not easy to tie those interactions together.

If we think of the known information as a hierarchy from "best" representation of a user to "worst" it will look something like:

- *User ID*: a unique user id that persists across devices (authenticated)
- *Device ID*: a unique id that persists across sessions on the same device (such as a device id or an IP address + device fingerprint)
- *Browser ID*: a unique id that persists across sessions in the same application/browser only (a persistent cookie id)
- *Session ID*: a unique id that persists across a single session (such as cookie in a browser’s incognito mode)
- *Request ID*: a unique id that only persists for a single request (a browser with cookies turned off)

In most modern search applications, and certainly in most ecommerce applications, we typically have to deal with all of these. As a rule of thumb, you want to tie a user to the most durable identifier - the one high up the list as possible. Both the links between Request IDs and Session IDs, as well as the links between Session IDs and Browser IDs are through the user’s cookie, so ultimately the Browser ID (persistent unique ID stored in the cookie) is the common-denominator for each of these. Specifically:

- If a user has persistent cookies enabled, one Browser ID can have many Session IDs, which can have many Request IDs.
- If a user clears cookies after each session (such as by using incognito mode), one Browser has one Session ID, which can have many Request IDs.
- If a user turns off cookies then each Request ID has a new Session ID and a new Browser ID.

When building search platforms, most organizations do not properly plan for and design their signals tracking mechanisms. By not being able to correlate visitors with their different queries, result sets with users and queries, item interactions with visitors, queries, and results viewed, many organizations make it difficult to maximize the relevance of their search platform through enhanced analytics and through automated search relevance approaches. In some cases it is possible to derie missing signals tracking information after the fact (such as modeling signals into likely sessions using timestamps, which we'll demonstrate in chapter 6), but it is usually best to design the system to better handle user tracking upfront as part of the system design to prevent potential information loss. The topics of implementing a productionized user tracking system, safely obfuscating user data to protect user privacy, and so on are beyond the scope of this book, but we felt it important for readers to understand these high level mechanisms for how to related user interactions such that they can add maximum value to your AI-powered search efforts. In the next section, we'll start to introduce how these signals actually get used to improve relevance through a process known as "Reflected Intelligence"

4.3 Introduction to Reflected Intelligence

In the last section, we covered how to capture signals from users as they interaction with your search engine. While the signals themselves are useful to help us understand how our search engine is being used, they also serve as the primary inputs for building models that can continually learn from user interactions and enable your search engine to self-tune its relevance model. In this section we'll introduce how these self-tuning model work through the concept of Reflected Intelligence.

4.3.1 What is Reflected Intelligence?

Imagine you are an employee at a hardware store and someone asks you where they can find a hammer and you tell them "isle two". A few minutes later you see the same person walk from isle two to isle five without a hammer, and then walk out of isle five holding a hammer. The next day someone else asks for a hammer, you also tell them "aisle two", and you observe a nearly identical pattern of behavior. You would be a pretty lousy employee if you didn't spot this pattern and adjust your advice going forward to provide a better experience for your customers. Now imagine if you continued to give the same, poor answer hundreds or even thosands of times to every new customer who came through the door looking for a hammer.

Unfortunately, this is exactly how most search engines tend to work - they have a fairly static set of documents that are returned for each query, regardless of who each user is or how prior users have reacted to the list of documents shown. Thankfully, we can improve upon this substantially by applying machine learning techniques to the collected signals from user interactions. This enables us to learn about users' intent from their signals and then reflect that knowledge back to improve future search results. This process is called *Reflected Intelligence*.

Reflected Intelligence is all about creating feedback loops that constantly learn and improve based on evolving user interactions. Figure 4.2 demonstrates a high-level overview of implementing a Reflected Intelligence process.

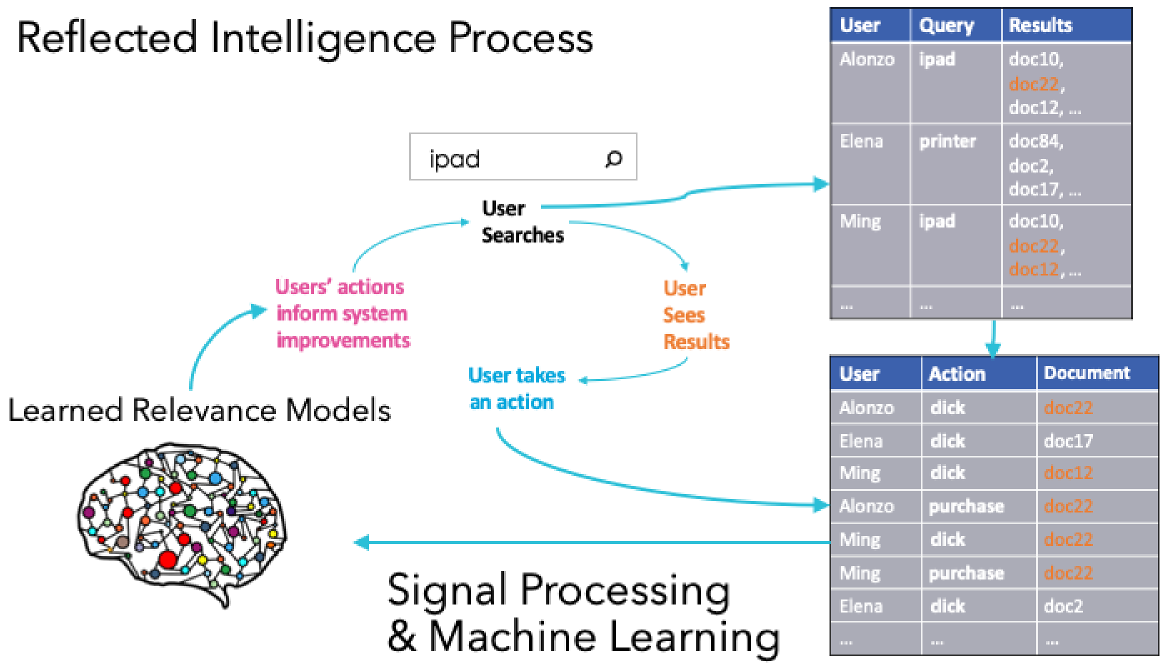


Figure 4.2 Reflected Intelligence Process. A user issues a query, sees results, and takes a set of actions. Those actions (signals) are then processed to create learned relevance models that improve future searches.

Let's walk through the Reflected Intelligence process shown in Figure 4.2. A user named Alonzo runs a search, entering the term *ipad* in the search box. A query signal is logged, containing the list of all search results displayed to Alonzo. Alonzo then sees the list of search results, and takes some actions. In the figure, Alonzo clicks on a document (`doc22`) and then purchases the product that document represents, resulting in two additional corresponding signals. All of Alonzo's signals, along with the signals from every other user, can then be aggregated and otherwise processed by various machine learning algorithms to create learned relevance models.

These learned relevance models may boost the most popular results for specific queries, personalize results for each specific user and their interests, or even learn which general attributes of the documents being searched tend to matter the most and tune the ranking algorithms to factor those attributes. The models could also learn how to better interpret user queries, such as identifying common misspellings, phrases, synonyms, or other linguistic patterns and domain-specific terminology.

Once these learned relevance models are generated, they can then be deployed back into the production search engine and immediately be applied to enhance the outcomes of future queries. The process then begins again, with the next user running a search, seeing (now hopefully improved) search results, and interacting with them. This creates a self-learning system which improves with every additional user interaction, getting continually smarter and more relevant over time, and also automatically adjusting as user interests and content evolve.

In the following sections, we will explore a few different categories of reflected intelligence models, including signals boosting (popularized relevance), collaborative filtering (personalized relevance), and learning to rank (generalized relevance). We'll start with one of the simplest and also most most impactful: signals boosting models.

4.3.2 Popularized Relevance through Signals Boosting

The most popular queries sent to your search engine tend to also be the most important ones to optimize from a relevance standpoint. Thankfully, since more popular queries generate more signals, this means that they are generally much easier to crowdsource reflected intelligence models for in order to improve relevance. Signals boosting models operate well on the most common and highest volume queries, making them a way to learn a "popularized relevance" model.

Listing 4.5 demonstrates an out of the box search for the query *ipad* in our RetroTech search engine.

Listing 4.5 Sending a keyword search for the query *ipad* to the search engine.

```
query = "ipad"

collection = "products"
request = {
    "query": query,
    "fields": ["upc", "name", "manufacturer", "score"],
    "limit": 5,
    "params": {
      "qf": "name manufacturer longDescription",
      "defType": "edismax"
    }
}

search_results = requests.post(solr_url + collection + "/select",
                               json=request).json()["response"]["docs"]
display(HTML(render_search_results(query, search_results)))
```

As expected the results of this query will return many documents containing the keyword *ipad* in them, and based upon what we learned in chapter 3 about how keyword relevance is scored leveraging TF-IDF and the BM25 ranking algorithm, the documents containing the term *ipad* the most times will typically rank the highest. Figure 4.3 shows the results of the query from Listing 4.5.

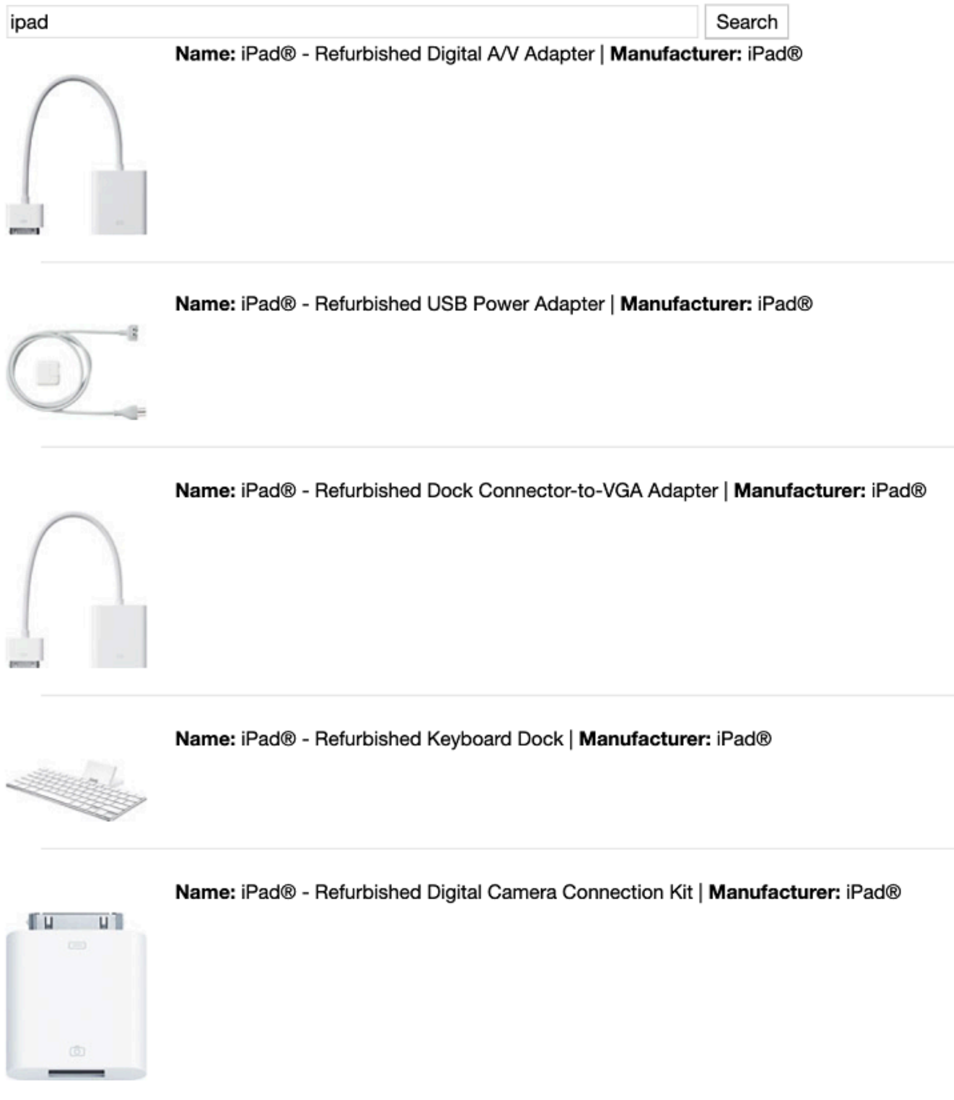


Figure 4.3 Results of a keyword search for the query ipad. Results are returned primarily based upon number of occurrences of the keyword, so accessories mentioning the keyword multiple times rank higher than the actual product the user intended to see.

While these results all contain the word "ipad" in their content multiple times, most users would be disappointed with these results since they are secondary accessories as opposed to the main product type that was the focus of the search. As you can probably guess, it is very hard to figure out the main product versus the secondary accessories just from the text in the product documents. For very popular queries, however, it is likely that many customers will run the same queries over and over again and fight through the frustrating search results to ultimately find the real products they are seeking. Signals Boosting is a technique for leveraging this aggregate user behavior from popular queries in order to automatically learn and return the best products.

Figure 4.4 demonstrates how how signals boosting works as a continuous feedback loop.

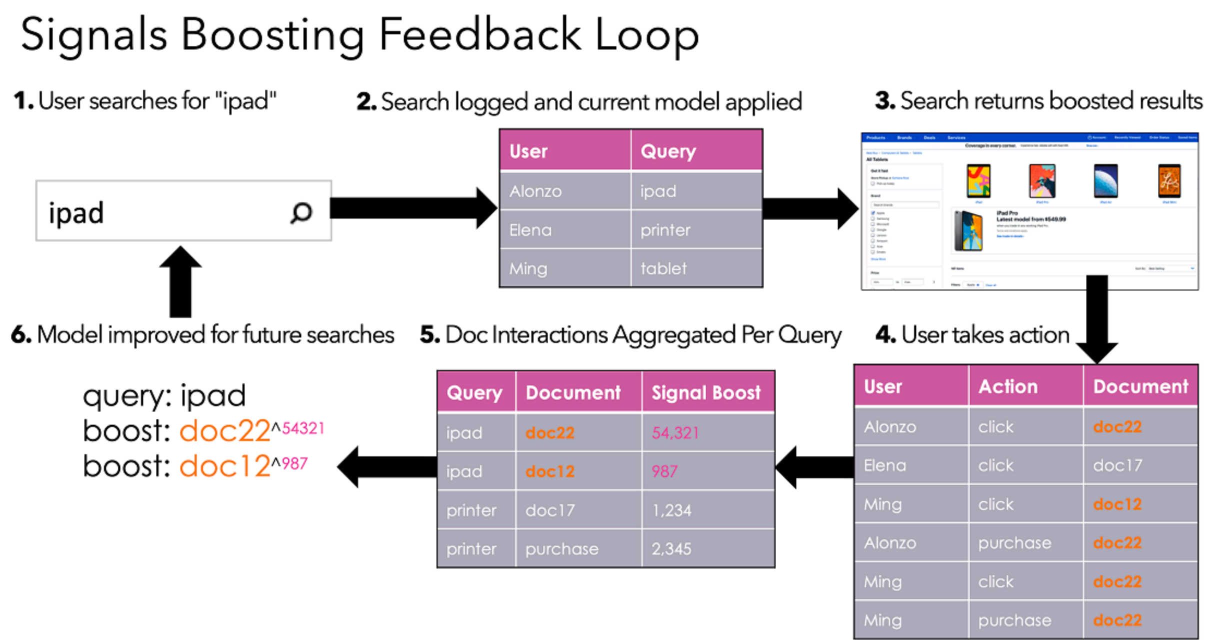


Figure 4.4 Signals Boosting Feedback Loop. A user’s search is logged, and the current signals boosting model is applied to return boosted results. After users take action on those results, the signals from all user interactions with documents are aggregated by originating query to genrate an updated signals boosting model to further improve future searches.

Signals boosting is one of the simplest forms of Reflected Intelligence, but also one of the most impactful for improving relevance of your most popular, highest-traffic queries. Once your products are indexed and you’ve started collecting signals for your users' queries and document interactions, the only additional steps necessary for implementing signals boosting are to aggregate your signals, and then to add your aggregated signals as boosts to either your queries or your documents. Listing 4.6 demonstrates a simple model for aggregating signals into a side-car collection.

SIDEBAR **Sidecar Collections**

Sidecar collections are additional collections that sit in your search engine alongside a primary collection and which contain other useful data to improve your search application. In our ecommerce example, our primary collection is the *products* collection, and we’ve already introduced our *signals* collection, which can be considered a sidecar collection. In this section we will also introduce a *signals_boosting* sidecar collection, which we will leverage at query time to enhance our queries. Throughout the book, we’ll introduce several other sidecar collections to store the inputs for and outputs of our self-learning models.

Listing 4.6 Generating a signals boosting model through aggregating signals.

```
products_collection="products"
signals_collection="signals"
signals_boosting_collection="signals_boosting"

create_collection(signals_boosting_collection)

signals_opts={"zkhost": "aips-zk", "collection": signals_collection}
signals_boosting_opts={"zkhost": "aips-zk", "collection": signals_boosting_collection,
                       "gen_uniq_key": "true", "commit_within": "5000"}

df = spark.read.format("solr").options(**signals_opts).load()
df.registerTempTable("signals")

print("Aggregating Signals to Create Signals Boosts...")

signals_aggregation_query = """
select q.target as query, c.target as doc, count(c.target) as boost
  from signals c left join signals q on c.query_id = q.query_id
  where c.type = 'click' AND q.type = 'query'
  group by query, doc
  order by boost desc
"""

spark.sql(signals_aggregation_query).write.format("solr").
    options(**signals_boosting_opts).mode("overwrite").save()
print("Signals Aggregation Completed!")
```

Results:

```
Wiping 'signals_aggregation' collection
Creating signals_aggregation' collection
Status: Success
Aggregating Signals to Create Signals Boosts...
Signals Aggregation Completed!
```

The important part of Listing 4.6 is the *signals_aggregation_query*, which we've actually just defined as a SQL query to keep the example more readable. For every query, we are getting the list of documents that users have clicked on in the search results for that query, along with a count of how many times the document has been clicked on. By ordering the documents by the count of times clicked for each query, we now have an ordered list of the documents that user's tended to choose to interact with for each query.

The intuition here is that users will tend to choose the products they believe are the most relevant, so if we were to boost these documents then we would expect our top search results to become more relevant. We'll test this theory out in Listing 4.7 by using these aggregated counts as Signals Boosts on our next query. Let's revisit our previous query for *ipad*.

Listing 4.7 Generating a Signals Boosting Query to improve search relevance ranking for top queries and documents.

```
query = "ipad"

signals_boosts_query = {
    "query": query,
    "fields": ["doc", "boost"],
    "limit": 10,
    "params": {
      "defType": "edismax",
      "qf": "query",
      "sort": "boost desc"
    }
}

signals_boosts = requests.post(solr_url + signals_boosting_collection + "/select",
    json=signals_boosts_query).json()["response"]["docs"]
print("Boost Documents: \n")
print(signals_boosts)

product_boosts = ""
for entry in signals_boosts:
    if len(product_boosts) > 0:  product_boosts += " "
    product_boosts += '"' + entry['doc'] + '"^' + str(entry['boost'])

print("\nBoost Query: \n" + product_boosts)


collection = "products"
request = {
    "query": query,
    "fields": ["upc", "name", "manufacturer", "score"],
    "limit": 5,
    "params": {
      "qf": "name manufacturer longDescription",
      "defType": "edismax",
      "indent": "true",
      "sort": "score desc, upc asc",
      "qf": "name manufacturer longDescription",
      "boost": "sum(1,query({! df=upc v=$signals_boosting}))",
      "signals_boosting": product_boosts
    }
}

search_results = requests.post(solr_url + collection + "/select",
                               json=request).json()["response"]["docs"]
display(HTML(render_search_results(query, search_results)))
```

Boost Documents:

```
[{'doc': '885909457588', 'boost': 966}, {'doc': '885909457595', 'boost': 205},
 {'doc': '885909471812', 'boost': 202}, {'doc': '886111287055', 'boost': 109},
 {'doc': '843404073153', 'boost': 73}, {'doc': '635753493559', 'boost': 62},
 {'doc': '885909457601', 'boost': 62}, {'doc': '885909472376', 'boost': 61},
 {'doc': '610839379408', 'boost': 29}, {'doc': '884962753071', 'boost': 28}]
```

Boost Query:

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109 "843404073153"^73
"635753493559"^62 "885909457601"^62 "885909472376"^61 "610839379408"^29 "884962753071"^28
```

The query in Listing 4.7 does two noteworthy things:

1. It queries the *signals_boosting* sidecar collection to look up the ordered (by boost) list of documents that received the highest signals boosts for the query, and it transforms those signals boosts into a query to the search engine
2. It then passes that boosting query to the search engine with the following relevance boosting parameters:
   - *"boost": "sum(1,query({! df=upc v=$signals_boosting}))"*
   - *"signals_boosting": product_boosts* where product boosts is the above listed "Boost Query".

Once the query is execued, we can take a look at how it improved our search results. If you remember from Figure 4.3, our original keyword search for *ipad* returned mostly iPad accessories, as opposed to actual iPad devices. Figure 4.5 demonstrates the new results based upon signals boosting being applied on top of the keyword query.

Figure 4.5 Search results with Signals Boosting Enabled. Instead of iPad accessories showing up as before, we now see actual iPads, because we have crowdsourced the answers based upon the documents with which users actually choose to interact.

The new results after applying signals boosting are significantly better than the keyword-only results. We now see actual iPads, the product the user typed in and almost certainly intended to find. You can expect to see similarly good results from most of the other popular queries in your search engine, since the more people that interact with them, the greater the reliability of this crowdsourcing approach for figuring out relevance. Of course, as we move further down the list of popular products, the relevance improvements from signals boosting will start to decline, and with insufficient signals we may even reduce relevance in many cases. Thankfully, we'll introduce many other techniques that can better handle the queries without adequate signals volume.

The goal of this section was to walk you through an initial, concrete example of impelmenting an end-to-end reflected intelligence model. The signals aggregation used in this impelementation was very simple, though the results speak for themselves. There are many considerations and nuances to consider when implementing a signals boosting model - whether to boost at query time or on documents, how to increase the weight of newer signals versus older signals, how to avoid malicious users trying to boost particular products in the search results by generating bogus signals for the system to interpret, how to introduce and blend signals from different sources, and many more considerations. If you'd like to learn more about these topics, we'll be diving in much deeper in chapter 8, Signals Boosting Models.

Let's move on from Signals Boosting for now, however, and discuss a few other types of Reflected Intelligence models.

4.3.3 Personalized Relevance through Collaborative Filtering

In the last section, we covered Signals Boosting, which we referred to as "popularized relevance", since it determines the most popular answers to common queries across all users. In this section, we'll introduce a Reflected Intelligence approach called collaborative filtering, which would be better described as "personalized relevance".

Collaborative filtering is the process of using observations about the preferences of some users to predict the preferences of other users. You've no doubt seen collaborative filtering in action many times before. It is the most popular type of algorithm used by recommendation engines, and it is the source of the common "users who liked this item also liked these items" recommendations lists that appear on many websites. Figure 4.6 demonstrates how collaborative filtering follows this same Reflected Intelligence feedback loop that we saw for signals boosting models.

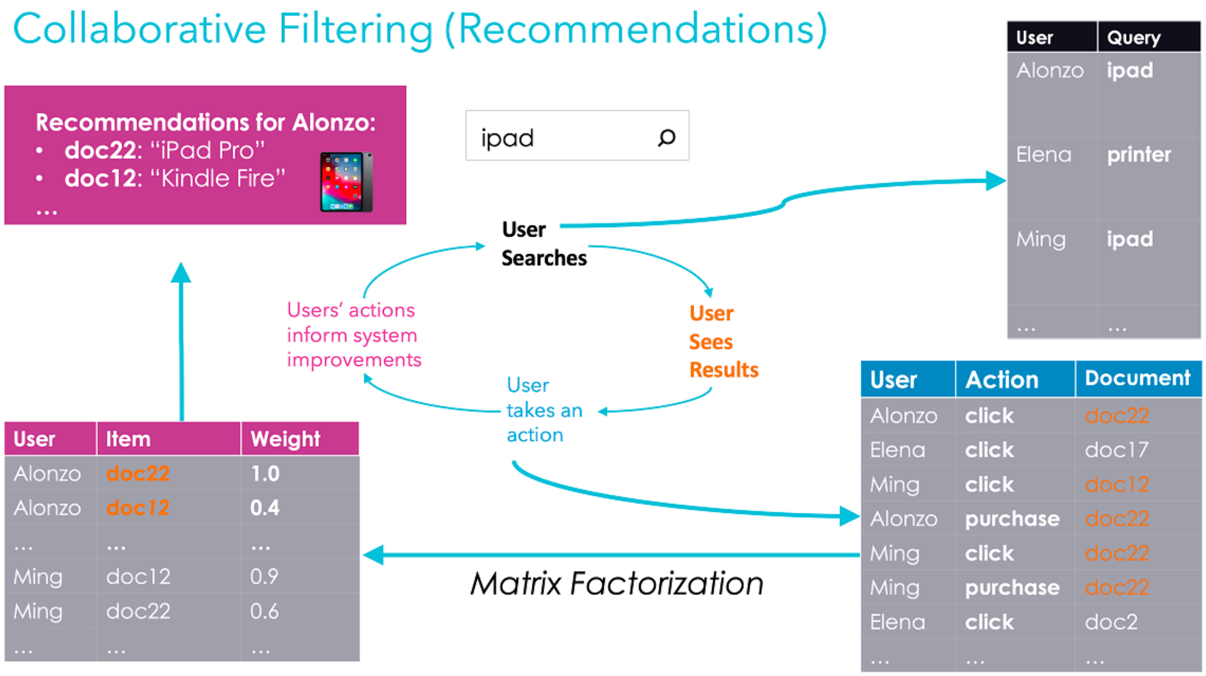


Figure 4.6 Collaborative filtering for user-to-item recommendations. Based upon his past behavior, our user (Alonzo) receives recommendations based upon items other users liked, where those users also interacted with the same items as Alonzo.

Like signals boosting, collaborative filtering is built leveraging a continuous feedback loop, where signals are collected, models are built based upon those signals, those models are then used to generate relevant matches, and the results of those interactions are then logged again as additional signals. Just as in other Reflected Intelligence models, the interactions between users and items are logged as signals. Collaborative filtering approaches will typically generate a user-item interaction matrix mapping each user to each item (document), with the strength of relationship between each user and item being based upon the strength of the positive interactions (clicks, purchases, ratings, etc.).

If the interaction matrix is sufficiently populated, it is possible to infer recommendations from it for any particular user or item by directly looking up the other users who interacted with the same item and then boosting other items (similar to signals boosting) with which those users also interacted. If the user-item interaction matrix is too sparsely populated, however then a *matrix factorization* approach will typically be applied. Matrix factorization is the process of breaking the user-item interaction matrix into two matrices: one mapping users to latent "factors", and another mapping those latent factors to items. This is similar to the dimensionality reduction approach we mentioned in chapter 3, where we switched from mapping phrases associated with food items from exact keywords (a vector including an element for every word in the inverted index), to a much smaller number of dimensions that described the food items and allowed us to match the meaning without having to map to every specific item (keyword). This makes it possible to derive preferences of users for items, as well as similarity between items, based upon very sparse data. In the context of matrix factorization for collaborative filtering, the latent factors represent attributes of our documents which are learned to be important indicators of shared interests across users. By matching other documents based upon these factors, we can are essentially using crowdsourcing find other similar documents matching the same shared interests.

There are other ways to generate recommendations which we'll explore later in the book, such as through content-based recommendations. Collaborative filtering is unique, however, in that it can learn preferences and tastes of users for other documents without having to know anything about the content of the document - all decisions are made entirely by observing the interactions of users with content and determining the strength of the similarity based upon those observations. We will dive much deeper into collaborative filtering, including code examples, when we discuss implementing Personalized Search in chapter 9.

As powerful as collaborative filtering can be for learning user interests and tastes based entirely on crowdsourced relevance, it unfortunately suffers from a major flaw known as the *cold start problem*.

The cold start problem describes a scenario where returning results is dependent upon the existance of signals, but where new documents that have never generated signals are not getting displayed. This creates a catch-22 situation where new content is unlikely to be shown and generate signals because it has never generated any signals that would lead to it being shown. To some degree, signals boosting models demonstrate a similar problem, where documents that are already popular tend to receive higher boost, resulting in them geting even more signals, while documents that have never been seen get no signals boosting. This creates a self-reinforcing cycle that can lead to a lack of diversity in search results.

Instead of only having popularized relevance or personalized relevance, which are dependent upon user interactions with specific items, it is generally also necessary to leverage a more generalized relevance model that can apply to all searches and documents and not just the most popular ones. This goes a long way toward solving the cold-start problem. We'll explore how crowdsourced relevance can be generalized in the next section through the concept of Learning to Rank.

4.3.4 Generalized Relevance through Learning to Rank

If signals boosting ("popularized relevance") and collaborative filtering ("peronalized relevance") both only work on documents which already have signals, this means that a substantial portion of queries and documents will not benefit from these approaches until they start receiving traffic and generating signals. This is where Learning to Rank proves valuable as a form of "generalized relevance".

Learning to Rank, also known as Machine-learned Ranking, is the process of building and using a ranking classifier that can score how well any documents (even those never seen before) match any arbitrary query. You can think of the ranking classifier as a trained relevance model. Instead of manually tuning search boosts and other parameters, the Learning to Rank process trains a machine learning model that can learn the important features of your documents and then score search results appropriately. Figure 4.7 shows the general flow for rolling out Learning to Rank

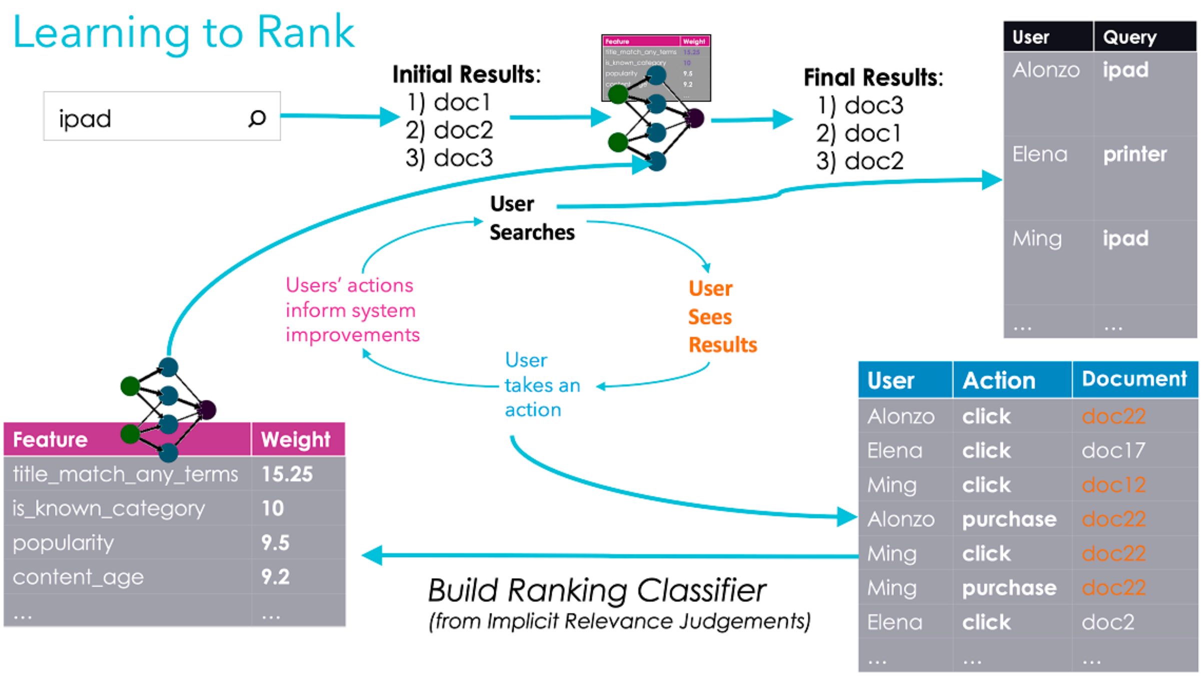


Figure 4.7 Learning to Rank (Generalized Relevance). A ranking classifier is built from user judgements about the known relevance of documents for each query (training set). That more advanced classifier model is then used to re-rank search results so that the top ranked documents are more relevant.

In a learning to rank system, the same high-level Reflected Intelligence process applies (refer to Figure 4.2) that we saw in signals boosting and collaborative filtering. The difference is that Learning to Rank can use relevance judgements lists (maps of queries to their ideal ranked set of document) to automatically train a relevance model that can be applied generally to all queries. You'll see that the output of the "Build Ranking Classifier" step is a model of relevance features (*title_match_any_terms*, *is_known_caegory*, *popularity*, *content_age*), and that model gets deployed into the production search engine periodically to enhance search results. The features in a very simple machine-learned ranking model might be readable like this, but there is no requirement that a ranking classifier actually be interpretable and explainable like this, and many advanced, deep-learning-based ranking classifiers are not.

Additionally, notice in Figure 4.7 that the live user flow goes from searching on the word *ipad* to and initial set of search results, to running through the deployed learning to rank classifier, to returning a final set of re-sorted search results. This final set of results is re-ranked based upon the learned ranking function in the ranking classifier. Since the ranking classifier is typically much more intelligent and leverages more complicated ranking parameters than a traditional keyword-ranking relevance model, it is usually way too slow to use the ranking classifier to score all of the matching documents in the search engine. Instead, Learning to Rank will often still use an initial, faster ranking function (such as BM25) function to find a the top-N documents (usually hundreds or thousands of documents) and then only run that subset of documents through the ranking classifier to get a better relevance ordering for the top results. It is possible to leverage the ranking classifier as the main relevance function instead of applying this re-ranking technique, but it is more common to see a re-ranking approach, since it is typically much faster while still yielding approximately the same resuls.

Learning to Rank can leverage either explicit relevance judgements (created manually be experts) or implicit judgements (derived from user signals), or some combination of the two. We'll dive deep into examples of implementing Learning to Rank from both explicit and implicit judgements lists in chapter 10 and 11.

In the next section, we'll discuss a few additional examples of useful signals-based Reflected Intelligence models.

4.3.5 Other reflected intelligence models

In addition to diving deeper into Signals Boosting (chapter 8), Collaborative Filtering (chapter 9), and Learning to Rank (chapter 10), there are many other kinds of reflected intelligence that we'll explore throughout this book. In chapter 6, we'll explore mining user queries to automatically learn domain-specific phrases, common misspellings, synonyms, and related terms, and in chapter 11 we'll explore automated ways of learning relevance judgements from users from their interactions such that we can automatically generate training data for interesting machine learning approaches.

In general, every interaction between a user and some content creates a connection - an edge in a graph - that we can use to understand emerging relationships and derive deeper insights. Figure 4.8 demonstrates some of the various relationships we can learn by exploring this interaction graph.

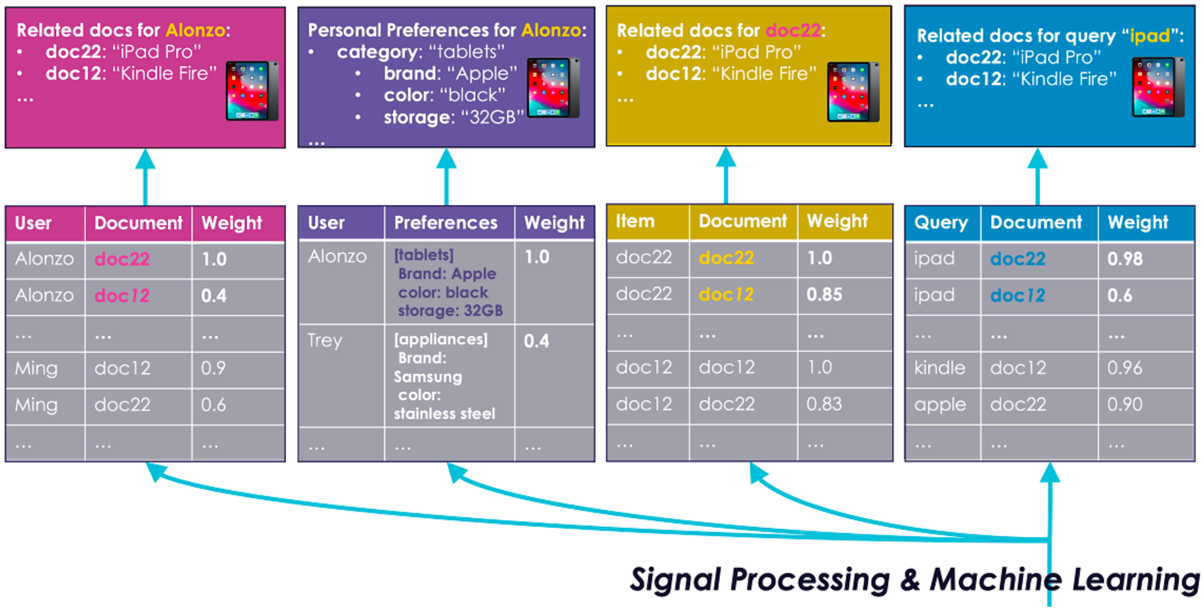


Figure 4.8 Many Reflected Intelligence models. The first box represents user-to-item similarity for recommendations, the second shows learning of specific attribute-based preferences for a user's profile, the third shows learning item-to-item based similarity for recommendations, and the last shows learning query-to-item recommendations.

Figure 4.8 shows how the same incoming signals data can be processed differently through various signal aggregation and machine learning approaches to learn similarity between users and items, to learn specific attribute-based preferences to generate a profile of a user's tastes, to learn similarity between items, and to generate query to item recommendations. We'll continue to explore these techniques in the chapters to come, but it is good to keep in mind that the signals data contains a treasure trove of potential insights that often provide just as much benefit as the documents from which their interactions are derived. Reflected Intelligence is the concept of learning from your users and reflecting that intelligence back, so it is not constrained to only the signals boosting, collaborative filtering, and the Learning to Rank techniques we've described. In the next section, we'll even even discuss a few ways to derive Reflected Intelligence from content instead of signals.

4.3.6 Crowdsourcing from content

While we typically think of crowdsourcing as asking people to provide input, we've seen thusfar in this chapter that implicit feedback can often provide as much or even more value in aggregate across many user signals. While this chapter has been entirely focused on leveraging user signals to do this crowdsourcing, it's also important to point out that it is often possible to use content itself to crowdsource intelligence for your AI-powered search platform.

For example, if you are trying to figure out the general quality of your documents, you may be able to look at customer reviews to either get product rating or to see if the product has been reported as abusive or spam. If the customer has left comments, you may be able to run a sentiment analysis algorithm on the text to determine if the comments are positive, neutral, or negative, and then boost or penalize the source documents accordingly.

We mentioned that in chapter 6 we'll walking through how to mine user signals to automatically learn domain specific terminology (phrases, misspellings, synonyms, etc.). Just as we can take user queries and interactions to learn this terminology, we should also realize that our documents were themselves typically writen by people, and that very similar kinds of relationships between terminology are therefore reflected in the written content, as well. We'll explore these content-based relationships further in the next chapter.

One of the most well-known search algorithms in existance is the Page Rank aglorithm - the breakthrough algorithm which enabled Google to rise to prominence as the most relevant search engine and stay there for many years. Page rank essentially goes beyond the text in any given document and looks at the behavior of all other web page creators to see how they have linked to other documents within their own documents. By measuring the incoming and outgoing links, it is then possible to create a measure of "quality" of a website based upon the assumption that people are more likely to link to higher-quality, more authoratative sources, and that those higher-quality sources are less likely to link to lower quality sources. This idea of going beyond the content that exists within a single document and instead relating it to external content - whether it be direct user interactions (signals), comments and feedback posted on a forum, links between websites, or even just the usage of terminology in different, nuanced ways across other documents is incredibly powerful. The art and science of leveraging all the available information about your content and from your users, is key to building a highly-relevant AI-powered search engine. In the next chapter, we'll take a look at the concept of knowledge graphs and how we can leverage some of these relationships embedded in the links between documents to automatically learn domain understanding from our content.

4.4 Summary

- Content and signals are the two sources of "fuel" to power an AI-powered search engine, with signals being the primary source for crowdsourced relevance.
- Reflected Intelligence is the process of creating continuously-learning feedback loops that improve from each user interaction and reflect that learned intelligence back to automatically increase the relevance of future results.
- Signals boosting is a form of "popularized" relevance, which usually has the biggest impact on your highest-volume, most popular queries.
- Collaborative filtering is a form of "personalized" relevance, which is able to use patterns of user interaction with items to learn preferences of users or the strength of relationship between items, and to then recommend similar items based upon those learned relationships.
- Learning to Rank is a form of "generalized" relevance, and is the process of training a ranking classifier based upon relevance judgements lists (queries mapped to correctly-ranked lists of documents) that can be applied to rank all documents and avoid the cold start problem.
- Other kinds of Reflected Intelligence exist, including some techniques for leveraging content (instead of just signals) for crowdsourced relevance.

5 Knowledge graph learning

This chapter covers:

- Building and working with knowledge graphs
- Implementing open information extraction to generate knowledge graphs from text
- Using semantic knowledge graphs for query expansion and rewriting, relationship discovery, query classification, and query-sense disambiguation
- Discovering the nuanced context for interpreting each query
- Interpreting documents with semantic knowledge graphs to power content-based document recommendations

In the last chapter we focused on learning relationships between documents based upon crowdsourced interactions linking those documents. While these interactions were primary user behavioral signals, we closed out the chapter also discussing links between documents that appear within the documents themselves - for example, leveraging hyperlinks between documents. In chapter 2, we also discussed how the textual content in the documents, rather than being "unstructured data", is really more like a giant graph of "hyperstructured data" containing a rich graph of semantic relationships connecting the many character sequences, terms, and phrases that exist across the fields within our collections of documents.

In this chapter, we will demonstrate how to leverage this giant graph of rich relationships within our content in order to better interpret your domain-specific terminology. We'll accomplish this through the use of both traditional knowledge graphs - which enable explicit modeling of relationships within a domain, and through semantic knowledge graphs, which enable real-time inference of nuanced semantic relationships within a domain.

We'll also play with several fun datasets in this chapter in order to show some variety in how knowledge graphs can be built and applied to improve query undertanding across different domains.

5.1 Working with knowledge graphs

In section 1.4, we introduced the idea of *knowledge graphs* and discussed how they relate to other types of knowledge models such ontologies, taxonomies, synonyms, and alternative labels. Knowledge graphs, if you recall, integrate each of the other types of knowledge models together, so we are discussing them all collectively as we build out and refer to "knowledge graphs" in this chapter.

A knowledge graph (or any graph, for that matter), is represented through the concept of nodes and edges. Nodes are often also referred to as vertices, but we'll use the terminology "node" in this book. A *node* is an entity represented in the knowledge graph (such as a term, a person, place, thing, or concept), whereas an *edge* represents a relationship between two nodes. Figure 5.1 shows an example of a graph displaying nodes and edges.

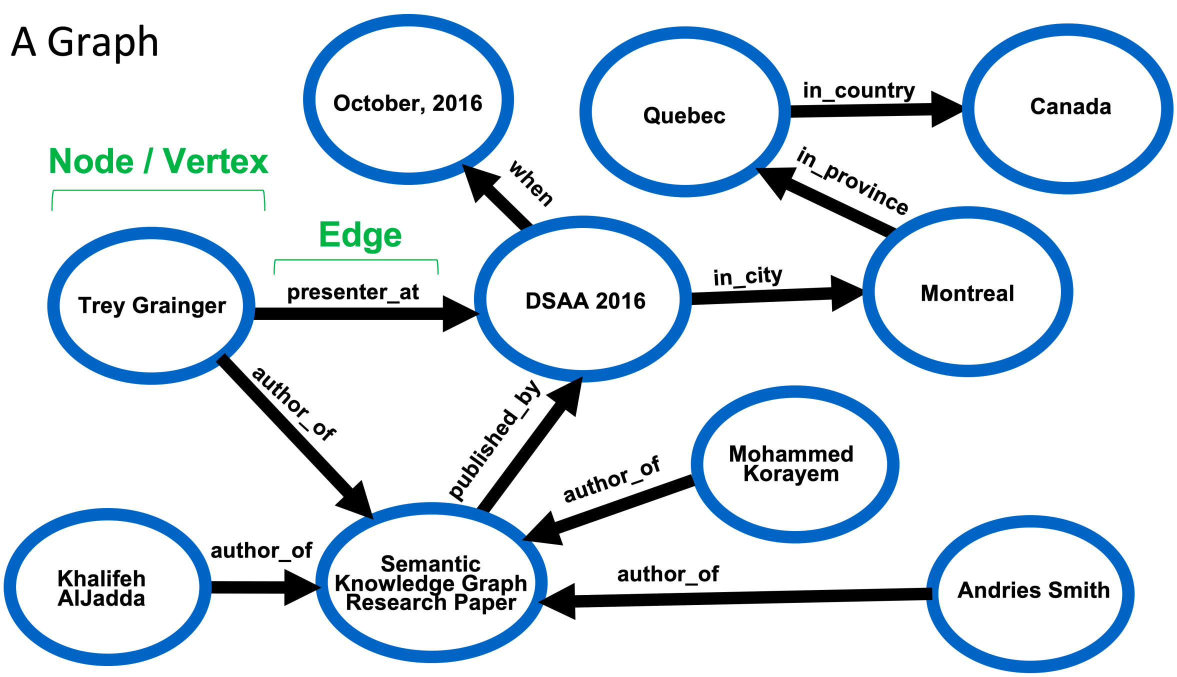


Figure 5.1 A Graph Structure. Graphs are composed of "nodes" (also known as "vertices") that represent entities, and edges which represent a relationship with another node. Graphs provide a way to model knowledge and infer new insights by traversing (or "following") the edges between nodes.

In this figure, you can see four nodes representing authors, one node representing a research paper they wrote together, one node representing the academic conference at which the paper was presented and published, and then nodes representing the city, province, country, and dates during which the conference was held. By traversing (or "following") the independent edges between nodes, you could for example, infer that one of the authors was in Canada in Montreal in October 2016. While any structure with nodes and edges like this is considered a graph, this particular graph represents factual knowledge and is therefore also considered a knowledge graph.

There are numerous ways to build and represent knowledge graphs, both through explicitly modeling data into a graph containing nodes and edges, as well as through dynamically materializing (discovering) nodes and edges from your data on the fly though what's known as a *semantic knowledge graph*. In this chapter, we'll walk through examples of building an explicit knowledge graph by hand, of auto-generating an explicit knowledge graph, and of leveraging a semantic knowledge graph that is already present within your search index.

To get started with knowledge graphs, you have essentially three options:

1. Build your own knowledge graph from scratch using a graph database (Neo4j, ArangoDB, etc.)
2. Plug in a pre-existing knowledge graph (ConceptNet, DBPedia, etc.)
3. Auto-generate a knowledge graph from your data, leveraging your content directly to extract knowledge.

Each approach has its strengths and weaknesses, and they are not necessarily mutually exclusive. If you are building a general knowledge search engine (such as a web search engine), then leveraging the second option of leveraging a pre-existing knowledge graph is a great place to start. If your search engine is more domain-specific, however, there is a good chance that your domain-specific entities and terminology will not be present, and that you will really need to create your own bespoke knowledge graph.

In this chapter we will focus primarily on the third option - auto-generating a knowledge graph from your content. The other techniques (manually generating a knowledge graph and leveraging a pre-existing knowledge graph) are already covered well in external materials - just do a web search for technologies like SPARQL, RDF Triples, and Apache Jena or for pre-existing knowledge graphs like DBPedia and Yago. That being said, you will ultimately need to be able to override your knowledge graph and add your own content, so we will include examples of how you can integrate both explicitly-defined (built with a specific list of pre-defined relationships) and implicitly-defined knowledge graphs (auto-generated relationships discovered dynamically from the data) into your search platform. We'll start off with an overview of how to integrate an explicitly-defined knowledge graph.

5.2 Building a knowledge graph explicitly into your search engine

Many organizations spend considerable resources building out knowledge graphs for their organizations, but then end up having trouble figuring out the best way to integrate those knowledge graphs within their search engines. We have fortunately chosen a search engine (Apache Solr) for our examples that has explicit graph traversal capabilities directly built in, so there is no need to pull in a new, external system to implement or traverse our knowledge graphs in this book.

While there may be some advantages to using a different tool such as Neo4J or ArangoDB, such as more sophisticated graph traversal semantics, using an external graph database like this makes coordinating requests, keeping data in sync, and infrastucture management more complex. Additionally, because some kinds of graph operations can only be done effectively in the search engine, like the semantic knowledge graph traversals we'll encounter shortly, leveraging the search engine as a unified platform search and knowledge graph capabilities enables us to combine the best of both worlds.

We will focus extensively on implementing a semantic search search system in chapter 7, including semantic query parsing, phrase extraction, misspelling detection, synonym expansion, and query rewriting, all of which will be modeled into an explicitly-built knowledge graph. Since the purpose of this chapter is to focus on knowledge graph *learning*, we'll save the discussion of explicit knowledge graph building and traversal until chapter 7 when we can tie everything from this chapter and chapter 6 (Learning Domain-specific Language) together into the appropriate knowledge graph structure.

5.3 Automatic extraction of knowledge graphs from content

While it is necessary to be able to manually add nodes and edges to your knowledge graphs, as we mentioned in the previous section, maintaining a large scale knowledge graph manually is very challenging. It requires substantial subject matter expertise, must be actively kept up to date with changing information, and is subject to the biases and errors of those maintaining it.

Open Information Extraction is an evolving area of Natural Language Processing (NLP) research. Open Information Extraction aims to extract facts directly out of your text content. This is often done using NLP libraries and language models to parse sentences and assess the dependency graph between them. A *dependency graph* is a break down of the parts of speech for each word and phrase in a sentence, along with an indication of which words refer to which other words. In this section, we'll use a language model and dependency graphs to extract two different types of relationships: arbitrary relationships and hyponym relationships.

5.3.1 Extracting arbitrary relationships from text

Given the "hyperstructured" nature of text and the rich relationships expressed within typical sentences and paragraphs, it stands to reason that we should be able to identify both the subjects of sentences and the ways in which they are related. In this section, we'll focus on this on extracting arbitrary relationships between the entities descibed within the sentences of our text content.

By analyzing the nouns and verbs within a sentence, it is often possible to infer a fact that is present in the sentence by mapping the subject, verb, and object of the sentence into an RDF triple representing a subject (starting node), relationship (edge), and object (ending node). For example, in the sentence "Colin attends Riverside High School", the verb "attends" can be extracted as relationship type connecting the subject ("Colin") with the object ("Riverside High School").

Listing 5.1 walks through an example of using the SpaCy library to extract facts from text content. Spacy is leading Natural Language Processing library that ships with state of the art statistical neural network models for part of speech tagging, dependency parsing, text categorization, and named entity recognition.

Listing 5.1 Extracting arbitrary relationships using SpaCy NLP to relate nouns and verbs

```
text = """
Data Scientists build machine learning models. They also write code.
Companies employ Data Scientists. Software Engineers also write code.
Companies employ Software Engineers.
"""

def generate_graph(text):
    parsed_text = lang_model(text)
    parsed_text = resolve_coreferences(parsed_text) # "they" => "Data Scientists"
    sentences = get_sentences(parsed_text) # Data Scientists also write code.
    => ['nsubj, 'advmod', ROOT', 'dobj', 'punct']

    facts=list()
    for sentence in sentences:
        facts.extend(resolve_facts(sentence)) # subj:(Companies), rel:(employ), obj:(basketball)
    return facts

graph = generate_graph(text)
for i in graph: print(i)
```

Results:

```
sentence: Data Scientists build machine learning models.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Data Scientists also write code.
dependence_parse: ['nsubj', 'advmod', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Companies employ Data Scientists.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Software Engineers also write code.
dependence_parse: ['nsubj', 'advmod', 'ROOT', 'dobj', 'punct']
```

```
---------------------
sentence: Companies employ Software Engineers.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
---------------------
['Data Scientists', 'build', 'machine learning models']
['Data Scientists', 'write', 'code']
['Companies', 'employ', 'Data Scientists']
['Software Engineers', 'write', 'code']
['Companies', 'employ', 'Software Engineers']
```

As you can see, the example code has taken the text content, parsed it into sentences, and then determined the subjects, relationships, and objects within those sentences. Those subject/relationship/object tuples can then be saved off into a explicitly-built knowledge graph and traversed.

Figure 5.2 provides a visualization of this extracted graph.

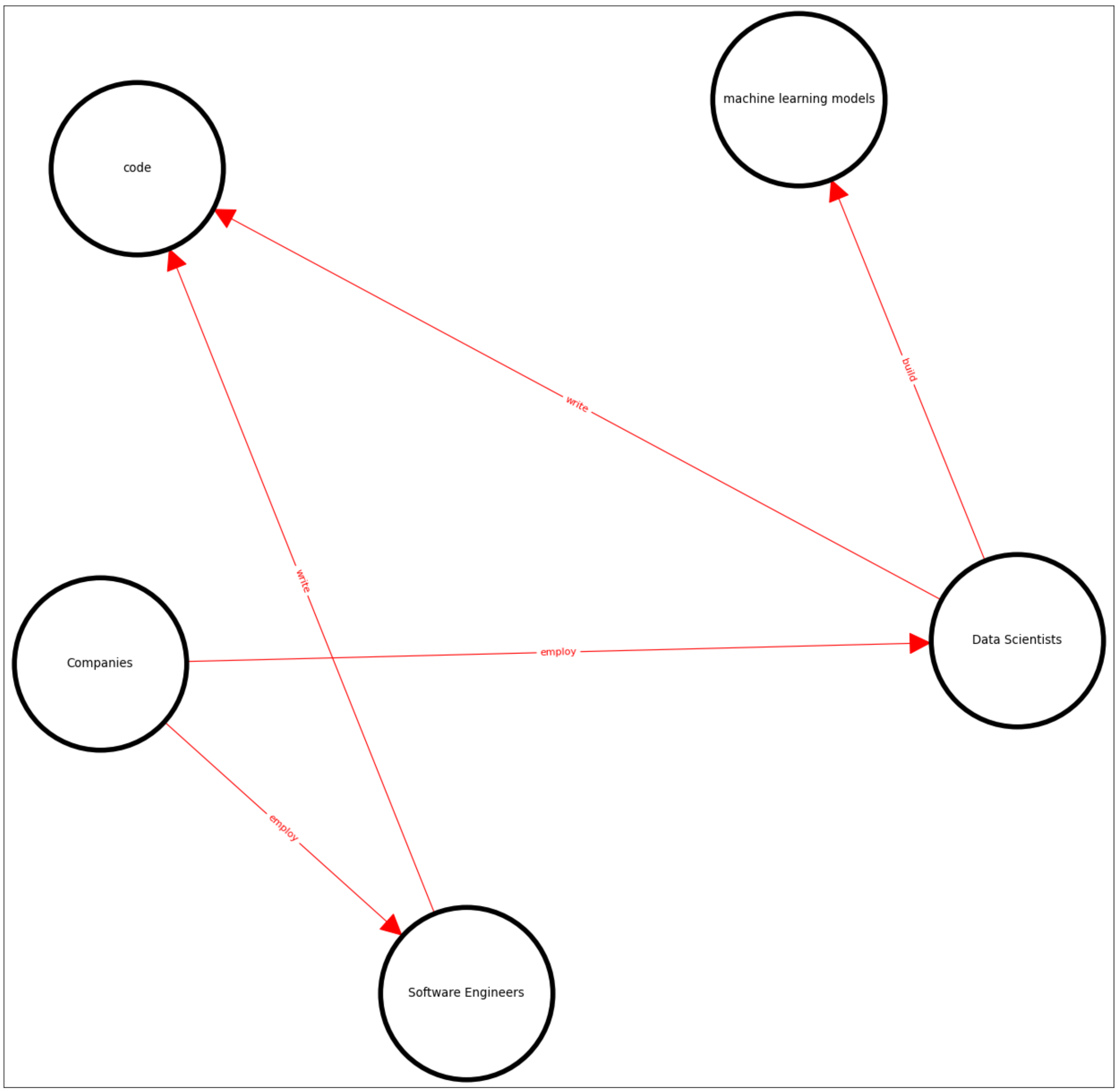


Figure 5.2 Extracted Knowledge Graph. The nodes and edges in this graph were automatically extracted from textual content based upon part of speech patterns.

The example in Figure 5.2 is very simple, and much more involved algorithms exist to extract facts from more sophisticated linguistic patterns. While we are using the SpaCy library in the code example, which leverages a deep-learning-based neural language model to detect parts of speech, phrases, and dependencies and co-references with the input text, the mechanism we leverged for then parsing those linguistic ouputs was more rule-based, following known semantic patterns within the English language.

Unfortunately, when parsing arbitrary verbs into relationships this way, the extracted relationships can become quite noisy. Since verbs conjugate differently, have synonyms, and have overlapping meanings, it is often necessary to prune, merge, and otherwise cleanup any list of arbitrary extracted relationships.

In contrast, some relationship types are much simpler, such as statistical relationships ("is related to") and hyponyms ("is a"). We'll spend the rest of the chapter covering these two special types, starting with hyponyms.

5.3.2 Extracting hyponyms from text

While it can be challenging mapping arbitrary verbs to clean lists of relationships within a knowledge graph, extracting hyponyms and hypernyms can be much easier. *Hyponyms* are entities that maintain an "is a" or "is instance of" relationship with a more general form of the entites, with the more general form being called a *hypernym*. For example, for the relationships between the terms *phillips head*, *screw driver*, and *tool*, we would say that *phillips head* is a hyponym of *screw driver*, that *tool* in a hypernym of *screw driver*, and that *screw driver* is both a hypernym of *phillips head* and a hyponym of *tool*.

One common and fairly accurate way to extract hyponym / hypernym relationships from text is through the use of Hearst patterns.[1]. These patterns describe common linguistic templates that reliably indicate the presence of hyponyms within sentences. Listing 5.2 demonstrates a few examples of such patterns.

Listing 5.2 Example Hearst Patterns which identify hyponym relationships to they hypernyms

```
simple_hearst_patterns = [
    (  '(NP_\\w+ (, )?such as (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
    (
        '(such NP_\\w+ (, )?as (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
    (
        '((NP_\\w+ ?(, )?)+(and |or )?other NP_\\w+)',
        'last'
    ),
    (
        '(NP_\\w+ (, )?include (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
    (
        '(NP_\\w+ (, )?especially (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
]
```

Each of these five simple patterns are represented as a Python tuple, with the first entry being a *regular expression*, and the second being a position within the pattern match (i.e. *first* or *last*). If you are unfamiliar with regular expressions, they provide a common and powerful syntax for pattern matching within strings. Anywhere you see the *NP_* characters, this stands for the existence of a *noun phrase* within a sentence, and the position specified in the second element of the tuple (*first* or *last*) indicates which noun phrase in the sentence represents the hypernym, with all other noun phrases matching the pattern considered the hyponyms.

In our example code in Listing 5.3, we run through almost 50 of these Hearst patterns to match many combinations of "is a" relationships within our content.

Listing 5.3 Extracting hyponym relationships using Hearst Patterns

```
text_content = """
Many data scientists have skills such as machine learning, python, deep learning, apache spark,
or collaborative filtering, among others Job candidates most prefer job benefits such as
commute time, company culture, and compensation.
Google, Microsoft, or other tech companies might sponsor the conference.
Big cities, such as San Francisco, New York, and Miami appeal to new graduates.
Many job roles, especially software engineering, registered nurse, and DevOps Engineer are
in high demand.
There are such job benefits as 401(k) matching, work from home, and flexible spending accounts.
Programming languages, i.e. python, java, ruby and scala.
"""

extracted_relationships = h.find_hyponyms(text_content)

facts = list()
for pair in extracted_relationships:
    facts.append([pair[0], "is_a", pair[1]])

print(*facts, sep="\n")
```

Response:

```
['machine learning', 'is_a', 'skill']
['python', 'is_a', 'skill']
['deep learning', 'is_a', 'skill']
['apache spark', 'is_a', 'skill']
['collaborative filtering', 'is_a', 'skill']
['commute time', 'is_a', 'job benefit']
['company culture', 'is_a', 'job benefit']
['compensation', 'is_a', 'job benefit']
['Google', 'is_a', 'tech company']
['Microsoft', 'is_a', 'tech company']
['San Francisco', 'is_a', 'big city']
['New York', 'is_a', 'big city']
['Miami', 'is_a', 'big city']
['software engineering', 'is_a', 'job role']
['registered nurse', 'is_a', 'job role']
['DevOps Engineer', 'is_a', 'job role']
['python', 'is_a', 'programming language']
['java', 'is_a', 'programming language']
['ruby', 'is_a', 'programming language']
['scala', 'is_a', 'programming language']
```

As you can see from Listing 5.3, by focusing on extracting a fixed type (and the most prevalent type of relationship - the "is_a" relationship), we are able to generate a nice, clean list of relationships with the more specific term (the hyponym) pointing to the more general term (the hypernym) with an "is_a" edge. Figure 5.3 demonstrates this generated graph visually.

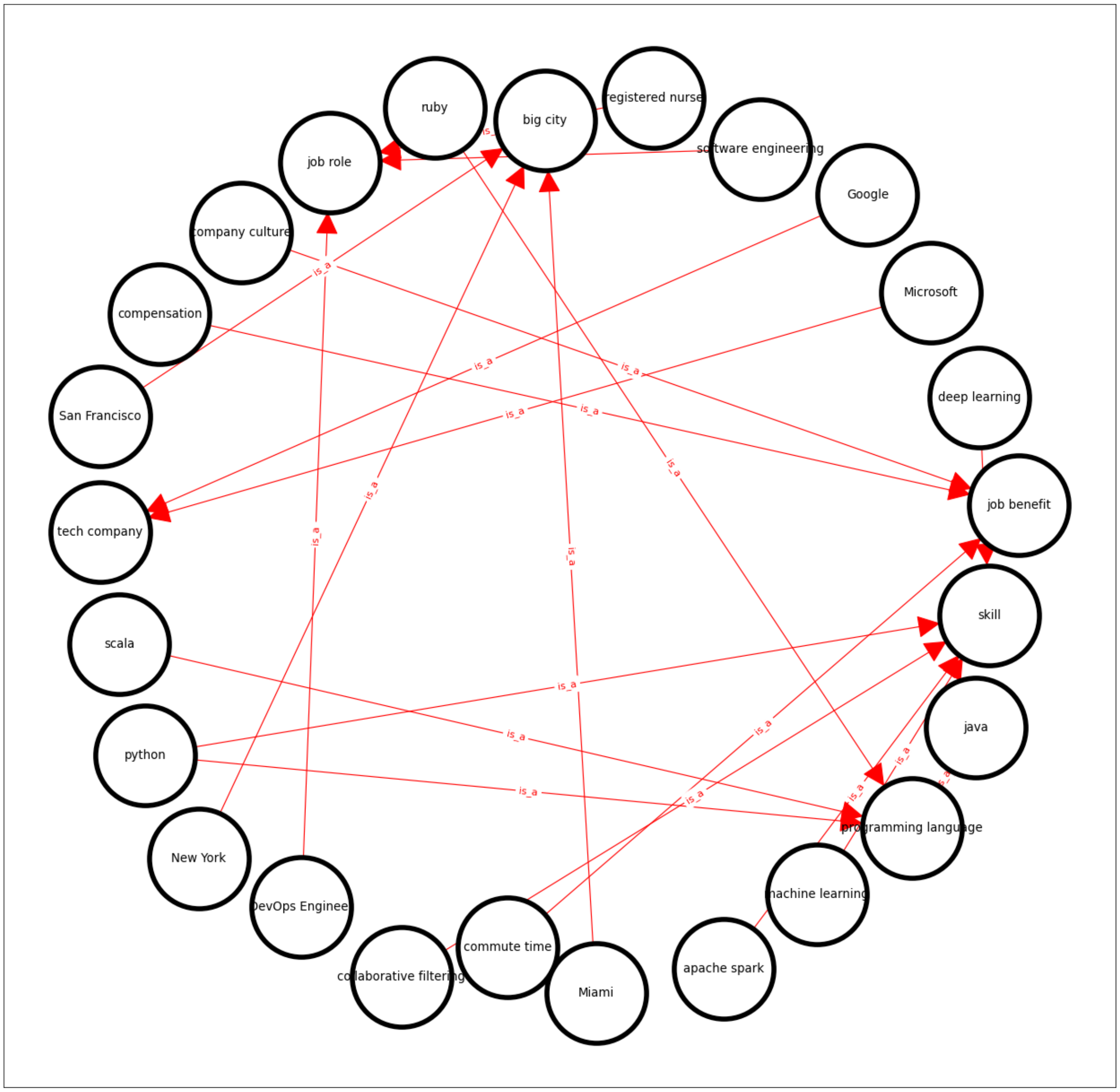


Figure 5.3 Knowledge graph derived from Hearst Patterns. We can see that all nodes are connected to other nodes through an "is_a" edge.

The inconsistency and noise that exists with arbitrary relationship extraction in the last section is now gone. We could still have ambiguity about the relationship between similar terms (for example, misspellings, alternative spellings, known phrases, or synonyms), but those are much easier to resolve. In fact, we'll spend the entire next chapter discussing how to learn this kind of domain-specific language from your signals and content in order to leverage it when interpreting incoming user queries.

While it can be useful to extract information from our text into an explicit knowledge graph for later traversal, the reality is that this kind of extraction is a lossy process, as the representation of the items gets disconnected from the originating context of those items within our content (the surrounding text and documents containing the text). In the next section, we'll introduce an entirely different kind of knowledge graph - a semantic knowledge graph - that is optimized to enable real-time traversal and ranking of the relationships between terms and phrases within our content without having to be explititly built and without having to even separate terms from their original textual context.

5.4 Learning intent by traversing semantic knowledge graphs

In chapter 2, sections 2.1-2.2, we discussed the myth of text content being "unstructured data", and how in reality text documents represent hyper-structured data. We discussed the distributional hypothesis ("a word shall be known by the company it keeps"), and walked through how character sequences, terms, and phrases (or other arbitrary term sequences) can be thought of as fuzzy foreign keys relating similar concepts between documents. We also discussed how these links between documents can be thought of as edges in a giant graph of relationships, enabling us to learn the contextual meaning of the terms and entities present within our corpus of documents.

In this section, we introduce a semantic knowledge graph, a tool and technique which will enable us to traverse a graph of those semantic relationships present within our documents.

5.4.1 What is a semantic knowledge graph?

A semantic knowledge graph is a "compact, auto-generated model for real-time traversal and ranking of any relationship within a domain".[2] 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.

For example, if we indexed a collection of documents about health topics and we searched for the pain reliever `advil`, instead of returning documents that contain the term `advil`, a semantic knowledge graph would automatically (with no manual list creation or data modeling required) return values like:

```
advil  0.71
motrin  0.60
aleve  0.47
ibuprofen  0.38
alleve  0.37
```

Results like these could be though of as "dynamic synonyms", but instead of the terms having the same meaning, they are more like conceptually-related terms. You could expand a traditional search engine query for "advil" to include these other terms, for example, in order to improve the recall of your search results or boost documents that conceptually match the meaning of `advil`, instead of just those containing the 5-letter string of `a-d-v-i-l`.

In addition to finding related terms, a semantic knowledge graph is able to traverse between fields in your inverted index ("find most related skills to this job title"), to traverse multiple levels deep ("find most related job titles to this query, and then find the most related skills for this query and each of those job titles"), and to use any arbitrary query you can send to the search engine as a node in the graph traversal to find semantically-related terms in any field.

The use cases for a semantic knowledge graph are diverse. It can be used for query expansion, for generating content-based recommendations, for query classification, for query disambiguation, for anomaly detection, for data cleansing, and for predictive analytics. We'll explore several of these throughout the remainder of this chapter, as soon as we get our datasets for testing our semantic knowledge graphs ready.

5.4.2 Indexing the datasets

A semantic knowledge graph works best on datasets where there is a decent overlap of terms being used together across documents. The more often two words tend to appear within documents the better we are able to determine whether those terms appear statistically more often we would expect.

While Wikipedia is often a good starting dataset for many usecases, since Wikipedia usually has a single page about a major topic that is supposed to be authoritative, there is actually not a ton of overlap across douments, making Wikipedia a less than optimal dataset for this use case.

That being said, any kind of website where users submit the content (questions, forum posts, job postings, reviews) will tend to make for an excellent dataset for a semantic knowledge graph use case.

For this chapter, we have selected two primary datasets, a *jobs* dataset (job board postings) and a series of StackExchange data dumps including posts from the following forums:

- health
- scifi
- devops
- outdoors
- travel
- cooking

In order to index the datasets, please run through the Index Datasets Jupyter notebook prior to running any of the semantic knowledge graph examples

5.4.3 Structure of a semantic knowledge graph

In order to make best use of a semantic knowledge graph, it is useful to understand how the graph works based upon its underlying structure.

Unlike a traditional knowledge graph, which must be explicitly modeled into nodes and edges, a semantic knowledge graph is *materialized* from the underlying inverted index of your search engine. This means that all you have to do to use a semantic knowledge graph is to index documents into your search engine. No extra data modeling is required.

The inverted index and a corresponding forward index then serve as the underlying data structure which enables real-time traversal and ranking of any arbitrary semantic relationships present within your collection of documents. Figure 5.4 demonstrates how documents get mapped into both the forward index and the inverted index.

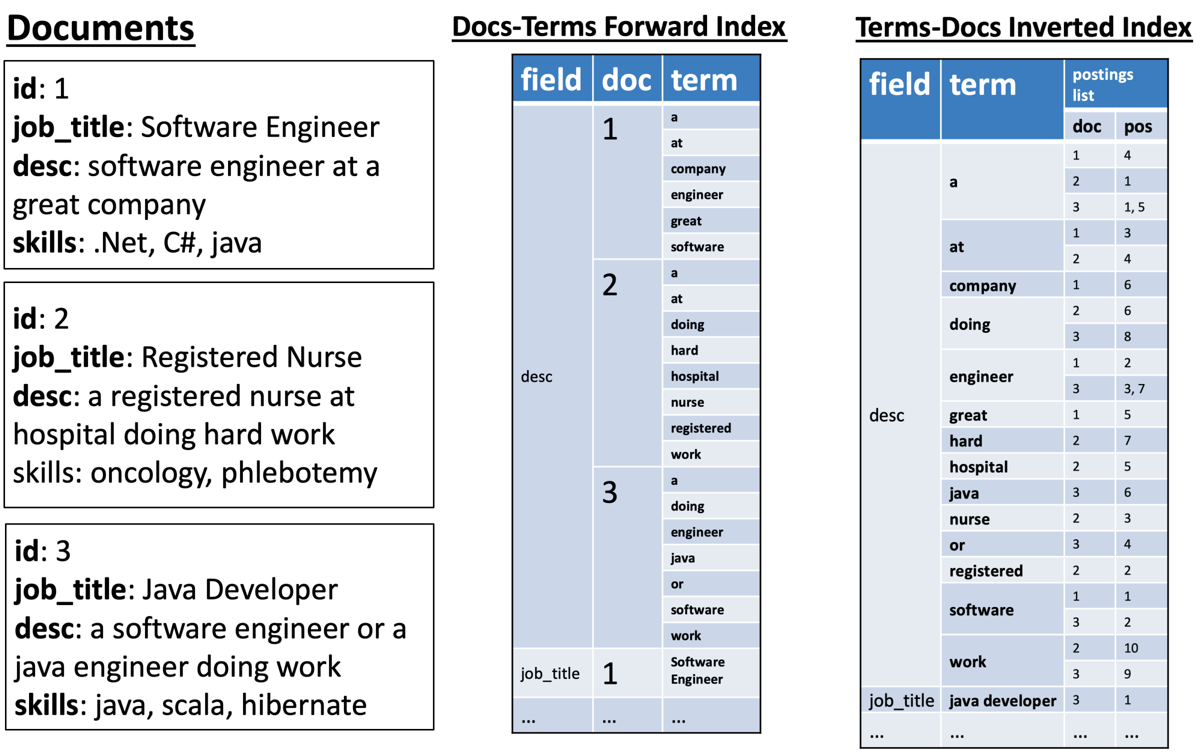


Figure 5.4 Inverted Index and Forward Index. Documents get mapped into an inverted index, which maps documents lists of terms, and a forward index, which maps terms back to lists of documents. Having the ability to map both directions will prove important for graph traversal and relationship discovery.

On the left of the figure, you can see three documents, each of which have a `job_title` field, a `desc` field, and a `skills` field. The right side of the figure shows how these documents are mapped into your search engine's inverted index. We see that the inverted index maps each field to a list of terms, and then maps each term to a postings list containing a list of documents (along with positions in the documents, as well as some other data not included in the figure). This make it quick and efficient to look up any term in any field and find the set of all documents containing that term.

In addition to the well-known inverted index, however, you will also see the less-well known *forward index* in the center of Figure 5.4. A forward index can be thought of as an *uninverted index*: for each field, it maps each document to a list of terms contained within that document. A forward index is what search engines use to generate facets on search results. In Lucene-based search engines like Solr and Elasticsearch, a forward index is usually generated at index time for a field by enabling a feature called `docValues` on the field. Alternatively, Apache Solr also allows you to generate the same forward index by "uninverting" the inverted index in memory on the fly at query time, enabling faceting even on fields for which `docValues` weren't added to the index.

If you have the ability to search for any arbitrary queries and find sets of documents through an inverted index, and then you also have the ability to take arbitrary sets of documents and look up terms in those documents, this means that by doing two traversals (terms to documents to terms) that you can find all of the related terms that also appear together in documents containing the original query. Figure 5.5 demonstrates how such a traversal would occur, including a data structure view, a set theory view, and a graph view.

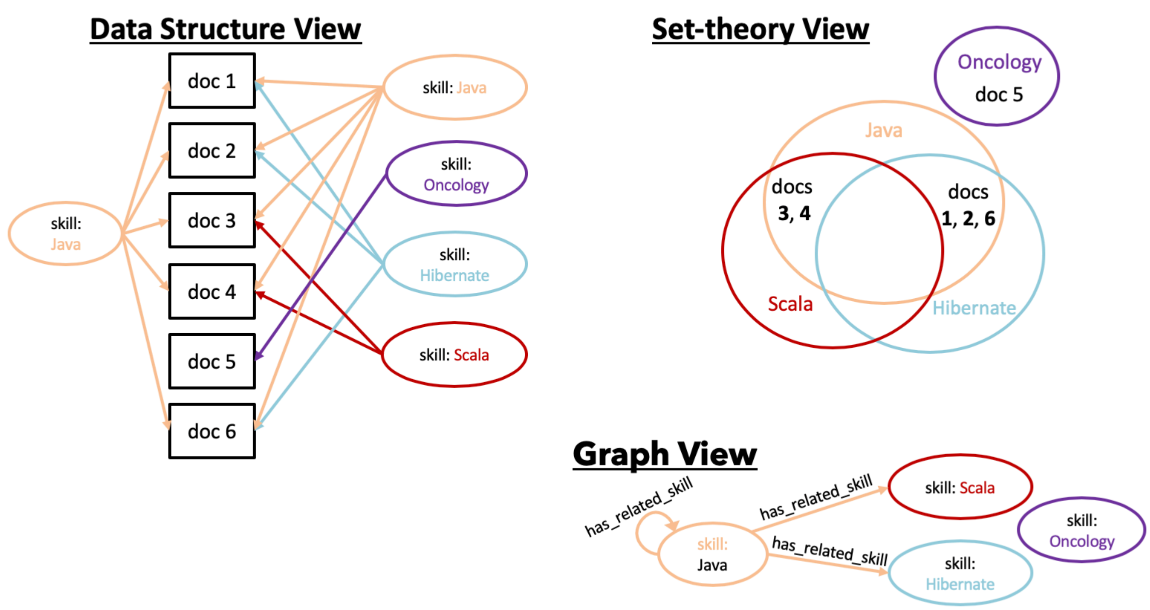


Figure 5.5 Three Representations of a Semantic Knowledge Graph. The Data Structure View shows terms mapped to sets of documents, the Set-theory View shows how the intersection of sets of documents actually forms the relationship between them, and the Graph View, showing the nodes and edges.

In the data structure view, which represents our inverted and forward indices, we see how terms are related to documents based upon whether they appear within them. Those "links" are ultimately only present if there is an intersection between the docs that any two nodes (terms) appear within in the set-theory view. The graph view, finally, demonstrates a third view into the same underlying data structures, but in this case we see nodes (instead of document sets) and edges (instead of intersecting document sets). Essentially, our semantic knowledge graph exists as an abstraction on top of the inverted index that is already built and updated anytime our search engine indexes content.

We typically think of the primary function of search engines being to accept a query, to find matching documents, and to return those documents in a relevance-ranked order. We devoted all of chapter 3 to discussing this process, walking through matching (sections 3.2.4 - 3.2.6), TF*IDF ranking (section 3.1), and the commonly-used BM25 ranking function (section 3.2).

Any arbitrary query (anything you can express in Solr syntax) can be a node in your graph, and you can traverse from that node to any other term (or arbitrary query) in any field. Additionally, since each traversal of an edge between two nodes leverages both an inverted index (terms to docs) and a forward index (docs to terms), it is trivial to chain these traversals together into a multi-level graph traversal, as shown in Figue 5.6.

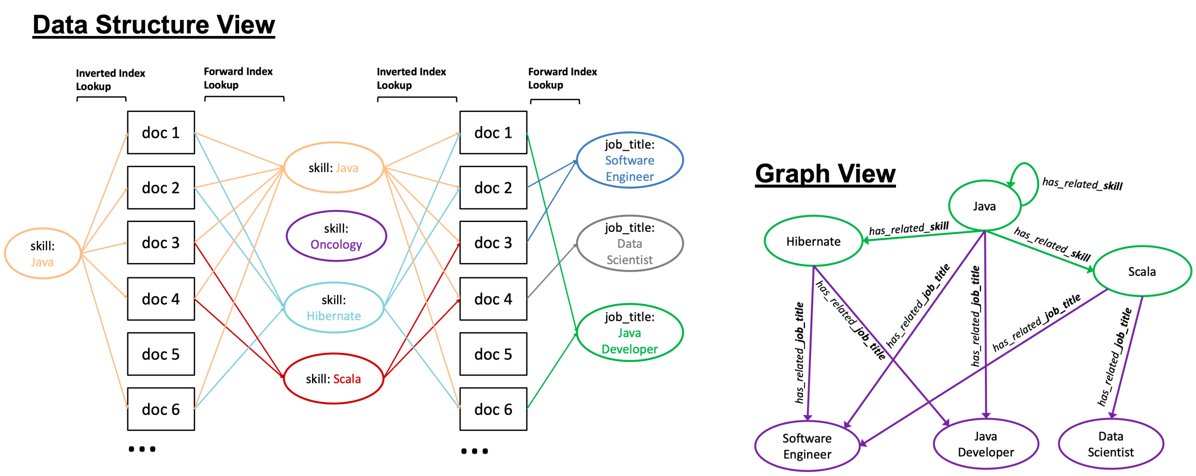


Figure 5.6 Multi-level Graph Traversal. In the data structure view, we see two traversals, through the inverted index and then the forward index each time. In the graph structure view, we see the corresponding two-level traversal from skills, to skills, to job titles.

In the figure, you see a traversal from a skill (*java*) to a layer of other skills (*java*, *oncology*, *hibernate*, and *scala*), to a layer of job titles (*software engineer*, *data scientist*, *java developer*). You an see that not all nodes are connected - the node for *oncology*, for example, does not appear in the graph traversal view because none of the original nodes can connect to it through any edges since there are no overlapping documents.

Given that not all possible nodes are going to be relevant for any given traversal, it is also important for semantic knowledge graphs to be able score and assign a weight to the relationships between nodes so that those edges can be prioritized during any graph traversal. We will cover the scoring and assignment of weights to edged in the next section.

5.4.4 Calculating edge weights to score relatedness of nodes

Given that the primary function of a semantic knowledge graph is to discover and score the strength of the semantic relationship between nodes (where the nodes can represent any word, phrase, or arbitrary query), the ability to generate a semantic similarity score is critical. But what does "semantic similarity" actually mean?

If you recall the Distributional Hypothesis, which was introduced in chapter 2, it says that words that appear together in the same contexts and with similar distributions tend to share similar meanings. Intuitively this makes sense - documents about "pain" or "swelling" will be more likely to occur in documents that also mention "advil", "ibuprofen", or "ice pack". Interestingly, though, "ice pack" may also occur in documents containing terms like "cooler", "road trip", or "cold", whereas "advil" and "ibuprofen" likely would not.

These examples show words with similar meanings with their contexts, but let's also consider words like "a", "the", "of", "and", "if", "they", and countless other very common/general words. Indeed, these words will also appear heavily within the same contexts of "pain", "swelling", "advil", "ibuprofen", or any of the other words we examined. This points to the second part of the distributional hypothesis - that the words must also occur with similar distributions. In essence, this means that given some number of documents containing a first term, any second term tends to be semantically similar to the first term if it co-occurs in the same documents as the first term more often that it co-occurs in documents with other random terms.

Practically, since "the" or "a" tend to co-occur commonly with almost all other terms, they are not considered semantically similar to those terms even though their level of co-occurrence is high. For terms like "pain" and "ibuprofen", however, they occur together statistically way more often than either term appears with random other terms, and therefore they are considered semantically similar.

One easy way to measure this semantic relatedness is through the relatedness calculation (`z`) that follows.

```
z = (countFg(x) - t\otalDocsFG * probBG(x)) / sqrt(t\otalDocsFG * probBG(x) * (1 - probBG(x))
```

This relatedness calculation (conceptually similar to a "z-score" in a normal distribution) relies on the concept of a "foreground" set of documents and a "background" set of documents, and enables the distribution of the term "*x*" to be statistically compared between the two sets. For example, if the foreground set was all documents matching the query "pain", and the background set was all documents, then the relatedness of the word "advil" would be a measure of how much more often "advil" occurs in documents also containing the word "pain" (foreground set) versus in any random document (background set).

If two terms are highly related, their relatedness will be a positive number approaching `1.0`. If the terms are highly unrelated (they tend to occur in divergent domains only) then the score would be closer to `-1.0`. Finally, term that just aren't semantically related at all - like stopwords, will tend to have a relatedness score close to zero.

Apache Solr has semantic knowledge graph capabilities built directly into its faceting API. Faceting provides the ability to traverse from terms to sets of documents to terms, and a *relatedness* aggregation function implements the semantic similarity calculation we just described. Listing 5.4 demonstrates a search for "advil" within a dataset of Stack Exchange Health forum discussions, and enables us to find the most semantically-related terms.

Listing 5.4 Discovering related nodes. We traverse from our original query of "advil" to the most semantically-related terms (nodes) in the "body" field of our documents.

```
collection="health"

request = {
    "params": {
        "qf": "title body",
        "fore": "{!type=$defType qf=$qf v=$q}",
        "back": "*:*",
        "defType": "edismax",
        "rows": 0,
        "echoParams": "none",
        "omitHeader": "true"
    },
    "query": "advil",
    "facet": {
        "body": {
            "type": "terms",
            "field": "body",
            "sort": { "relatedness": "desc"},
            "mincount": 2,
            "limit": 8,
            "facet": {
                "relatedness": {
                    "type": "func",
                    "func": "relatedness($fore,$back)"
                }
            }
        }
    }
}

search_results = requests.post(solr_url + collection + "/select", json=request).json()

for bucket in search_results["facets"]["body"]["buckets"]:
  print(str(bucket["val"]) + "  " + str(bucket["relatedness"]["relatedness"]))
```

Results:

```
advil  0.70986
motrin  0.59897
aleve  0.4662
ibuprofen  0.38264
alleve  0.36649
tylenol  0.33048
naproxen  0.31226
acetaminophen  0.17706
```

As you can see, out of all terms within any of the text of the forum posts in the Stackexchange Health dataset, the ranked order of the most semantically related terms to "advil" was a nice, clean list of other pain killers which are most similar to "advil". This is the magic of leveraging the distributional hypothesis to discover and rank terms by semantic similarity - it provides us with the ability to automatically discover relationships on the fly that can be used to further improve our understanding of incoming queries. In the next section, we'll discuss how we can apply this understanding to improve query relevance.

5.4.5 Using semantic knowledge graphs for query expansion

Matching and ranking on only the keywords entered during a search does not always provide sufficient context to find and rank the best results. In these cases, you can significantly improve the quality of search results by expanding or otherwise augmenting queries to include conceptually-related terms. In this section, we'll walk through how to generate these related terms, and we'll demonstrate several strategies for applying those term to enhance they quality of your search results.

Given the ability to start with any keyword or query and to find highly-related other terms in any field, one obvious use case for a semantic knowledge graph is for dynamically expanding queries to include related terms. This enables documents to match which do not necessarily contain the exact keywords entered by the user but which do contain other terms which carry a very similar meaning.

For example, instead of a user's query for "advil" (from our last example) only matching documents with the string "advil" in them, we could use the semantic knowledge graph to automatically discover related terms and then match additional documents and boost the relevance score based upon the relatedness score of each expanded term. The final query submitted to the search engine, instead of being `advil`, might instead be something like `advil OR motrin^0.59897 OR aleve^0.4662 OR ibuprofen^.3824 OR ….`

Let's walk through the steps needed to implementing this kind of query expansion, leveraging a dataset from a different domain this time (our scifi dataset). Listing 5.5 provides the first step in this process, running a search for the obscure search term "vibranium" on our scifi dataset.

Listing 5.5 Discovering context for unknown terms. In this case, an obscure query for "vibranium" brings back useful context for anyone unfamiliar with the fictional term.

```
query = "vibranium"

collection = "stackexchange"

request = {
    "query": query,
    "params": {
        "qf": "title body",
        "fore": "{!type=$defType qf=$qf v=$q}",
        "back": "*:*",
        "defType": "edismax",
        "rows": 0,
        "echoParams": "none",
        "omitHeader": "true"
    },
    "facet": {
        "body": {
            "type": "terms",
            "field": "body",
            "sort": { "relatedness": "desc"},
            "mincount": 2,
            "limit": 8,
            "facet": {
                "relatedness": {
                    "type": "func",
                    "func": "relatedness($fore,$back)"
                }
            }
        }
    }
}

search_results = requests.post(solr_url + collection + "/select", json=request).json()

for bucket in search_results["facets"]["body"]["buckets"]:
  print(str(bucket["val"]) + "  " + str(bucket["relatedness"]["relatedness"]))
```

Response:

```
vibranium  0.92227
wakandan  0.75429
wakanda  0.75295
adamantium  0.7447
panther's  0.69835
klaue  0.68083
klaw  0.65195
panther  0.65169
```

For anyone unfamiliar with the term "vibranium", it is a strong, fictional metal that exists in Marvel comic books and movies (best popularized through the 2018 Hollywood hit *Black Panther*). The most related terms that came back were related to "Wakanda", the fictional country from which vibranium originates, "adamantium", another strong (fictional) metal from Marvel comics, and the words "panther" (from the name Black Panther) and the name Klaue and alternative spelling Klaw, a character in the Black Panther comic books and movie that is heavily associated with the metal vibranium.

You may or may not have any familiarity with vibranium or any of these related pieces of information - and the fact that you don't need to is exactly why an auto-generated knowledge graph is so useful. By leveraging a semantic knowledge graph and expanding your query to include additional related context, you can drastically improve the recall of your search requests, and by boosting results that best match your query conceptually (as opposed to just the text), you may also be able to improve the precision of your top-ranked search results.

Listing 5.6 demonstrates an example of translating this original query along with the semantic knowledge graph output into an expanded query.

Listing 5.6 Generation of an expanded query from the related nodes returned from the semantic knowledge graph.

```
query_expansion = ""

terms = search_results["facets"]["body"]["buckets"]
for bucket in search_results["facets"]["body"]["buckets"]:
  term = bucket["val"]
  boost = bucket["relatedness"]["relatedness"]
  if len(query_expansion) > 0:
    query_expansion += " "
  query_expansion += " " + term + "^" + str(boost)

expanded_query = query + "^5" + query_expansion

print("Expanded Query:\n" + expanded_query)
```

Results:

```
Expanded Query:
vibranium^5 vibranium^0.92228  wakandan^0.75429  wakanda^0.75295  adamantium^0.7447
panther's^0.69835  klaue^0.68083  klaw^0.65195  panther^0.65169
```

In this case, we are doing a simple Boolean OR search for any of the keywords related to the original query (`vibranium`), boosting the original query term's weight by a factor of 5x and weighting each subsequent term's impact on the relevance score based upon it's semantic similarity score. The choice to boost the original term by 5x is arbitrary - you can choose any value here to assign a relative relevance boost versus the other (expanded) terms.

You might also notice that the term vibranium appears twice - first as the original term and then again as an expanded term (since the term is *also* the most semantically similar to itself). This will almost always be the case if you are searching for individual keywords, but since your query might have phrases or other constructs that make the original query different than the terms that actually come back (if any), it is usually a good idea to still include the original query as part of the expanded/rewritten query so users don't get frustrated by it being ignored.

While the prior expanded query should rank results pretty well (prioritizing documents matching multiple related terms), it is also heavily focused on recall (expanding to include anything relevant) as opposed to precision (ensuring everything included is relevant). There are many different ways to construct an augmented query, depending on yout primary goals.

Rewritten queries can perform a simple expansion, require a minimum percentage or number of terms to match, require specific terms like the original query to match, or even just change the ranking of the same initial results set. Listing 5.7 demonstrates several examples, leveraging minimum match thresholds and percentages, which can tilt the scale between precision and recall as needed.

Listing 5.7 Different Query Augmentation Strategies.

```
simple_expansion = 'q={!edismax qf="title body" mm="0%"}' + query + " " + query_expansion
increase_conceptual_precision = 'q={!edismax qf="title body" mm="30%"}' + query + " "
+ query_expansion
increase_precision_reduce_recall = 'q={!edismax qf="title body" mm="2"}' + query + " AND "
+ ( query_expansion + )
same_results_better_ranking = 'q={!edismax qf="title body" mm="2"}' + query \
  + "&boost=" + "query($expanded_query)&expanded_query=" + query_expansion


print("Simple Query Expansion:\n" + simple_expansion)
print("\nIncreased Precision, Reduced Recall Query:\n" + increase_conceptual_precision)
print("\nIncreased Precision, No Reduction in Recall:\n" + increase_precision_reduce_recall)
print("\nSlightly Increased Recall Query:\n" + slightly_increased_precision)
print("\nSame Results Better Conceptual Ranking:\n" + same_results_better_ranking)
```

The final queries for each of these different query expansions techniques are as follows:

Simple Query Expansion:

```
q={!edismax qf="title body" mm="0%"}vibranium vibranium^0.92227  wakandan^0.75429  wakanda^0.75295
        adamantium^0.7447  panther's^0.69835  klaue^0.68083  klaw^0.65195  panther^0.65169
```

This simple query expansion is the same as previously described, matching any documents containing either the original query or any of the semantically-related terms.

Increased Precision, Reduced Recall Query:

```
q={!edismax qf="title body" mm="30%"}vibranium vibranium^0.92227  wakandan^0.75429
        wakanda^0.75295  adamantium^0.7447  panther's^0.69835  klaue^0.68083  klaw^0.65195
        panther^0.65169
```

This example specifies a "minimum match" threshold of 30% (`mm=30%`), meaning that in order for a document to match it must contain at least 30% (rounded down) of the terms in the query.

Increased Precision, No Reduction in Recall:

```
q={!edismax qf="title body" mm="2"}vibranium AND (vibranium^0.92227  wakandan^0.75429
        wakanda^0.75295  adamantium^0.7447  panther's^0.69835  klaue^0.68083  klaw^0.65195
        panther^0.65169)
```

This query requires the original query term (`vibranium`) to match, and *also* requires that at least two terms match (`vibranium` plus one more term). This means that a document must contain an additional terms beyond just the original query, but must also still match the original query.

Slightly Increased Recall Query:

```
q={!edismax qf="title body" mm="2"}vibranium vibranium^0.92227  wakandan^0.75429
        wakanda^0.75295  adamantium^0.7447  panther's^0.69835  klaue^0.68083  klaw^0.65195
        panther^0.65169
```

This query requires two terms to match, but does not explicitly require the original query, so it can expand to other documents which are conceptually similar but don't necessarily have to contain the original query term. Since the term `vibranium` is repeated twice, any document containing just the word `vibranium` will also match.

Same Results Better Conceptual Ranking:

```
q={!edismax qf="title body" mm="2"}vibranium&boost=query($expanded_query)
        &expanded_query=vibranium^5 vibranium^0.92227  wakandan^0.75429  wakanda^0.75295
        adamantium^0.7447  panther's^0.69835  klaue^0.68083  klaw^0.65195  panther^0.65169
```

This final query returns the exact same documents as the original query for `vibranium`, but it ranks them differently according to how well they match the semantically-similar terms from the knowledge graph. This ensures the keyword exists in all matched documents and that all documents containing the user's query are returned, but it then enables the ranking to be greatly improved to better understand the domain-specific context for the term to boost more relevant documents.

Of course, there are an unlimited number of possible query permutations you can explore when rewriting your query to include enhanced semantic context, but the above examples should provide a good sense of the kinds of options available and tradeoffs you'll want to consider.

5.4.6 Using semantic knowledge graphs for content-based recommendations

In the last section, we explored how to augment queries by discovering and leveraging related nodes from the semantic knowledge graph, including multiple ways of structuring rewritten queries to optimize for precision, recall, or even improved conceptual ranking over the same results. In addition to expanding queries with semantically-related terms, it is also possible to use the semantic knowledge graph to generate content-based recommendations by translating document into queries based upon semantic similarity of the terms within the documents.

Since nodes in the semantic knowledge graph can represent any arbitrary query, we can take the content from documents (individual terms, phrases, or other values) and model them as arbitrary nodes to be scored relative to some known context about the document. This means we can take dozens or hundreds of terms from a document, score them all relative to the topic of the document, and then take the top most semantically-similar terms and use them to generate a query best representing the nuanced, contextual meaning of the document.

Listing 5.8 walks through an example of translating a document that is classified as a "star wars" document and ranking all the terms in the document relative to that topic.

Listing 5.8 Content-based Recommendations. We can pass the terms and phrases of a document to the semantic knowledge graph, score their semantic similarity to the topic of the document, and generate a query representing the most semantically-important elements of the document.

```
import collections
from mergedeep import merge

print(solr_url)
collection="stackexchange"
classification="star wars"

document="""this doc contains the words luke, magneto, cyclops, darth vader,
           princess leia, wolverine, apple, banana, galaxy, force, blaster,
           and chloe."""

#run an entity extractor to parse out keywords to score
parsed_document = ["this", "doc", "contains", "the", "words", "luke", \
                   "magneto", "cyclops", "darth vader", "princess leia", \
                   "wolverine", "apple", "banana", "galaxy", "force", \
                   "blaster", "and", "chloe"]

request = {"query": classification, "params": {}, "facet": {}}

i=0
for term in parsed_document:
    i+=1
    key = "t" + str(i)
    key2 = "${" + key + "}"
    request["params"][key] = term
    request["facet"][key2] = {
        "type": "query",
        "q": "{!edismax qf=${qf} v=" + key2 + "}",
        "facet": {"stats": "${relatedness_func}"}
    }


print(json.dumps(request,indent="  "))

full_request = merge(request_template, request)
search_results = requests.post(solr_url + collection + "/select", json=full_request).json()

def parse_scores(search_results):
    results = collections.OrderedDict()
    for key in search_results["facets"]:
        if key != "count" and key != "" and "stats" in search_results["facets"][key]:
            relatedness = search_results["facets"][key]["stats"]["relatedness"]
            results[key] = relatedness
    return list(reversed(sorted(results.items(), key=lambda kv: kv[1])))

scored_terms = parse_scores(search_results)

for scored_term in scored_terms:
    print (scored_term)
```

Generated Knowledge Graph Lookup:


```
{
  "query": "star wars",
  "params": {
    "t1": "this",
    "t2": "doc",
```

```
    "t3": "contains",
    "t4": "the",
    "t5": "words",
    "t6": "luke",
    "t7": "magneto",
    "t8": "cyclops",
    "t9": "darth vader",
    "t10": "princess leia",
    "t11": "wolverine",
    "t12": "apple",
    "t13": "banana",
    "t14": "galaxy",
    "t15": "force",
    "t16": "blaster",
    "t17": "and",
    "t18": "chloe"
  },
  "facet": {
    "${t1}": {
      "type": "query",
      "q": "{!edismax qf=${qf} v=${t1}}",
      "facet": {
        "stats": "${relatedness_func}"
      }
    },
    ...
    "${t18}": {
      "type": "query",
      "q": "{!edismax qf=${qf} v=${t18}}",
      "facet": {
        "stats": "${relatedness_func}"
      }
    }
  }
}
```

Scored Nodes:

```
('luke', 0.66366)
('darth vader', 0.6311)
('force', 0.59269)
('galaxy', 0.45858)
('blaster', 0.39121)
('princess leia', 0.25119)
('this', 0.13569)
('the', 0.12405)
('words', 0.08457)
('and', 0.07755)
('contains', 0.04734)
('banana', -0.00128)
('doc', -0.00185)
('cyclops', -0.00418)
('wolverine', -0.0103)
('magneto', -0.01112)
('apple', -0.01861)
```

From these results, you can see a list of terms from the document that is nicely ordered based upon semantic similarity to the topic of "star wars". Terms with lower scores will have no relatedness or a negative relatedness with the specified topic. If we filter to terms above a `0.25` positive relatedness, as performed in Listing 5.9, we get a very clean list of relavant terms from the document.

Listing 5.9 Mapping scored phrases from the document to a query, filtering out low relatedness scores.

```
rec_query = ""

for scored_term in scored_terms:
  term = scored_term[0]
  boost = scored_term[1]
  if len(rec_query) > 0:
    rec_query += " "
  if boost > 0.25:
    rec_query += term + "^" + str(boost)

print("Expanded Query:\n" + rec_query)
```

Expanded Query:

```
luke^0.66366 darth vader^0.6311 force^0.59269 galaxy^0.45858 blaster^0.39121 princess leia^0.25119
```

Listing 5.10 demonstrates the last step step in this process, actually running the search to return the top ranked documents most semantically-similar to the original document.

Listing 5.10 Content-based Recommendations Request passing in the scored terms as the query.

```
import collections

collection="stackexchange"

request = {
    "params": {
        "qf": "title body",
        "defType": "edismax",
        "rows": 5,
        "echoParams": "none",
        "omitHeader": "true",
        "mm": "0",
        "fl": "title",
        "fq": "title:[* TO *]" #only show docs with titles to make the example readable
    },
    "query": rec_query
}

search_results = requests.post(solr_url + collection + "/select", json=request).json()
print(json.dumps(search_results, indent="  "))
```

Response:

```
{
  "response": {
    "numFound": 2864,
    "start": 0,
    "docs": [
      {
        "title": "\"Help me, Obi-Wan Kenobi\" -- how does Leia know who Obi-Wan is?"
      },
      {
        "title": "Why couldn't Snoke or Kylo Ren trace Luke using the Force?"
      },
      {
```

```
        "title": "Did Luke know the \"Chosen One\" prophecy?"
      },
      {
        "title": "Was Darth Vader at his strongest during Episode III?"
      },
      {
        "title": "Is there evidence that Lucas intentionally introduced similarities between
        Episodes I and IV?"
      }
    ]
  }
}
```

As you can see, we have just created a content-based recommendations algorithm. We discussed the idea of leveraging user behavioral signals (searches, clicks, etc.) to generate recommendations (collaborative filtering) in chapter 4, we will not always have sufficient signals to rely solely on signals-based recommendations approaches. Having the ability to take a document and find other similar documents based upon content of the first document provides us with an excellent additional tool with which we can provide context- and domain-aware recommendations without relying on user signals.

While the example in this section generated a content-based recommendations query based upon terms actually in the starting document, it is worth keeping in mind that the semantic knowledge graph is not restricted to using the terms passed in. You could add an extra level to the traversal to find additional terms that are semantically-related to the terms in the original document, but not actually contained within it. This can be particularly useful for niche topics where not enough documents match the recommendations query, as traversing further will open up new possibilities for exploration.

In the next section, we'll take a quick step beyond the "is_related_to" relationships and see if we can leverage the semantic knowledge graph to generate and traverse some more interesting edges.

5.4.7 Using semantic knowledge graphs to model arbitrary relationships

Thus far, all of our semantic knowledge graph traversals have leveraged an "is_related_to" relationship. That is to say, we've been finding the strength of the semantic relationship between two words or phrases using the relatedness function, but we have only measured that the nodes are "related", not how they are related. What if we could find other kinds of edges between nodes instead of just "is_related_to" type edges? In this section, we'll explore how to do exactly that.

If you recall, the nodes in a semantic knowledge graph are materialized on the fly by executing a query that matches a set of documents. If the node you start with is `engineer`, that node is internally represented as the set of all documents containing the word `engineer`. If the node is labeled as `software engineer`, that node is internally represented as the set of all documents containing the term `software` intersected with all documents containing the term `engineer`. If the search is for `"software engineer" OR java` then it is internally represented as the set of all documents containing the term `software` one position before the term `engineer` (a phrase) unioned with the set of all documents containing the term `java`.

You may also recall that an edge is formed by finding the set of documents which two nodes share in common. This means that *both* nodes and edges are internally represented using the same mechanism - a set of documents. Practically speaking, this means that if we can construct a node using a query that approximates an interesting relationship (as opposed to an entity), that we can relate two nodes together through the "relationship node" in a similar way to how an edge would be used to relate the nodes together in a traditional graph structure.

Let’s work through an example. Revisiting our scifi dataset let’s say we wanted to ask a question about "Jean Grey", one of the popular characters from Marvel Comics X-Men comic books, TV shows, and movies. Specifically, let’s say that we wanted to figure out who was in love with Jean Grey.

We can accomplish this by using a starting node of "Jean Grey", traversing to the node "in love with", and then requesting the top related terms associated with "in love with" within the context of "Jean Grey". Listing 5.11 demonstrates this query. By traversing through a node designed to capture an explicit linguistic relationship ("in love with" in this case), we can use the intermediate node to model an edge between the starting and terminating node.

Listing 5.11 Materializing an edge through a "relationship node".

```
collection = "scifi"

starting_node = '"jean grey"'
relationship = '"in love with"'

request = {
    "query": starting_node,
    "params": {
        "qf": "body",
        "fore": "{!type=$defType qf=$qf v=$q}",
        "back": "*:*",
        "defType": "edismax",
        "rows": 0,
        "echoParams": "none",
        "omitHeader": "true"
    },
    "facet": {
        "in_love_with":{
            "type": "query",
            "query": "{!edismax qf=body v=$relationship}",
            "facet": {
                "terminating_nodes": {
                    "type": "terms",
                    "field": "body",
                    "mincount": 25,
                    "limit": 9,
                    "sort": { "body_relatedness": "desc"},
                    "facet": {
                        "body_relatedness": {
                            "type": "func",
                            "func": "relatedness($fore,$back)"
                        }
                    }
                }
            }
        }
    }
}

search_results = requests.post(solr_url + collection + "/select", json=request).json()

for bucket in search_results["facets"]["in_love_with"]["terminating_nodes"]["buckets"]:
  print(str(bucket["val"]) + "  " + str(bucket["body_relatedness"]["relatedness"]))
```

Response:

```
jean  0.85044
grey  0.74965
cyclops  0.61313
summers  0.60624
xavier  0.54697
wolverine  0.49361
x  0.46596
mutant  0.46248
magneto  0.43692
```

If the X-Men comics or the names coming back in the response are unfamiliar to you, here's a quick crash course: Jean Grey has recurring relationships with two mutants, one named Cyclops (real name: Scott Summers) and one named Wolverine. Additionally, and unknown to most fans, two of Jean Grey's mentors, Professor Charles Xavier and Magneto, were known to have a love interest in Jean Grey at points throughtout the comic books.

If we examine the results from Listing 5.7, we see all of these expected names listed. The first two terms "jean" and "grey" are obviously the most related, since we are searching for "in love" relative to "jean grey", so her name is obviously going to be highly semantically related to itself. The next two terms, "cyclops" and "summers" both refer to the same person, Jean's most prominent love interest. Then we see "xavier" and "wolverine", and the final result in the list is for "magneto". Figure 5.7 demonstrates the underlying graph relationships for this traversal visually.

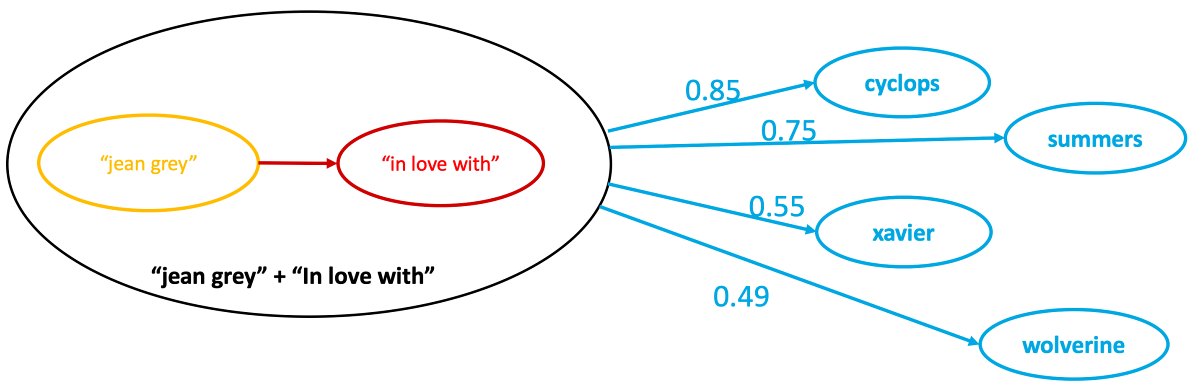


Figure 5.7 Traversing arbitrarily-defined edge types. By materializing a new node with the combined context of both the originating node ("jean grey") and a new node ("in love with"), we can traverse from that combined node ("jean grey" + "in love with") to other nodes. This is equivalent saying we are traversing from "jean grey" through an edge of "in love with" to the other nodes.

By using an intermediate node (i.e. "in love with") to model a relationship between other nodes, we can form any arbitrarily-typed edge between nodes, as long as we can express that edge as a search query.

While the results of our graph traversal in Listing 5.11 were pretty good overall, we do see, however, that the terms "x" (presumably from "x-men") and "mutant" also show up. Jean Grey and all of the other listed people are mutants in the X-Men comics, which is why these terms are so semantically related. They are clearly not great answers to the question "Who is in love with Jean Grey?", however.

This brings up an important point: the semantic knowledge graph is a statistical knowledge graph. The existence of the "in love with" relationship is purely based upon statistical correlations of terms within our collection, so just like with any ontology learning approach, there is going to be noise. That being said, for an auto-generated graph with no explicit modeling of entities (we just indexed inidividual keywords, with no sense that they represent people), these results are actually quite good.

If we wanted to improve the quality of these results, one of the easiest things to do would be to run pre-processing on the content to extract out entities (people, places, and things) and index those instead of just single-term keywords. This would cause actual people's names (i.e. "Scott Summers", "Charles Xavier", "Jean Grey") to be returned instead of just individual keywords ("summers", "xavier", "jean", "grey").

It is also worth pointing out that the traversal of relationships depends entirely on whether those relationships were discussed in the underlying corpus of documents. In this case, plenty of forum posts exist discussing each of these peoples' relationships with Jean Grey. Had insufficient documents existed, the results returned may have been poor or non-existent. In order to avoid noise in our results, we set a *mincount* threshold of *25*, indicating that at least 25 documents must exist discussing "jean grey", "in love with", and the other terms found and scored. We recommend setting a *mincount*, as well, to some number greater than *1* to avoid false positives.

While exploring arbitrary linguistic relationships like "in love with" can be useful for an exploratory standpoint, it is usually sufficient from a query understanding standpoint to stick with the default "is related to" relationship and just leverage the relatedness scores between terms for most semantic search use cases. While we have thusfar focused on finding similar keywords in a text field, our semantic knowledge graph traversals are not limited to just ranking the relatedness of keywords. In the next section, we'll explore how to use the semantic knowledge graph to classify queries.

5.4.8 Using semantic knowledge graphs for query classification

K-Nearest Neighbor classification is a type of classification which takes a datapoint (such as a query or term) and tries to find the top K other datapoints that are the most similar in a vector space. A semantic knowledge graph traversal essentially does a k-nearest neighbor search at each level of the graph traversal. This means that if we have a "category" or "classification" field present on our documents, that we can actually ask the semantic knowledge graph to "find the category with the highest relatedness to my starting node". Since the starting node is typically a user's query, this means we can use a semantic knowledge graph to classify the query.

Listing 5.12 demonstrates running a search for a few keywords and getting a category classification back. For simplicity, since we have indexed multiple different Stack Exchange categories (scifi, health, cooking, devops, etc.), we'll use those categories as our classifications. Let's find the most semantically-related categories for a few queries.

Listing 5.12 Query classification leveraging the semantic knowledge graph.

```
def run_query_classification(query,keywords_field="body",classification_field="category",
    classification_limit=5,min_occurrences=5):

    classification_query = {
        "params": {
            "qf": keywords_field,
            "fore": "{!type=$defType qf=$qf v=$q}",
            "back": "*:*",
            "defType": "edismax",
            "rows": 0,
            "echoParams": "none",
            "omitHeader": "true"
        },
        "query": query,
        "facet": {
            "classification":{
                "type": "terms",
                "field": classification_field,
                "sort": { "classification_relatedness": "desc"},
                "mincount": min_occurrences,
                "limit": classification_limit,
                "facet": {
                    "classification_relatedness": {
                        "type": "func",
                        "func": "relatedness($fore,$back)"
                    }
                }
            }
        }
    }

    search_results = requests.post(solr_url + collection + "/select",
    json=classification_query).json()

    print("Query: " + query)
    print("  Classifications: ")
    for classification_bucket in search_results["facets"]["classification"]["buckets"]:
        print("    " + str(classification_bucket["val"]) + "  "
        + str(classification_bucket["classification_relatedness"]["relatedness"]))
    print("\n")

run_query_classification( query="docker", classification_field="category",
classification_limit=3 )
run_query_classification( query="airplane", classification_field="category",
classification_limit=1 )
run_query_classification( query="airplane AND crash", classification_field="category",
classification_limit=2 )
run_query_classification( query="camping", classification_field="category",
classification_limit=2 )
run_query_classification( query="alien", classification_field="category",
classification_limit=1 )
run_query_classification( query="passport", classification_field="category",
classification_limit=1 )
run_query_classification( query="driver", classification_field="category",
classification_limit=2 )
run_query_classification( query="driver AND taxi", classification_field="category",
classification_limit=2 )
run_query_classification( query="driver AND install", classification_field="category",
classification_limit=2 )
```

Results:

```
Query: docker
  Classifications:
```

```
    devops  0.8376


Query: airplane
  Classifications:
    travel  0.20591


Query: airplane AND crash
  Classifications:
    scifi  0.01938
    travel  -0.01068


Query: camping
  Classifications:
    outdoors  0.40323
    travel  0.10778


Query: alien
  Classifications:
    scifi  0.51953


Query: passport
  Classifications:
    travel  0.73494


Query: driver
  Classifications:
    travel  0.23835
    devops  0.04461


Query: driver AND taxi
  Classifications:
    travel  0.1525
    scifi  -0.1301


Query: driver AND install
  Classifications:
    devops  0.1661
    travel  -0.03103
```

This request leverages the semantic knowledge graph to find the top K nearest neighbors based upon a comparison of the semantic similarity between the query and each available classification (within the category field). As you can see, each query was assigned one or more potential classifications based upon its semantic similarity with each classification.

We see classification scores for each potential category classification for each query, with `airplane` and `passport` classified to `travel`, `camping` classified to `outdoors`, and `alien` classified to `scifi`. When we refine the `airplane` query to a more specific query like `airplane AND crash`, however, we see that the category changes from `travel` to `scifi`, because documents about airplane crashes are more likely to occur within `scifi` documents than travel documents.

Similarly, we can see a word like `driver`, which can have multiple polysemous (ambiguous) meanings, returns two potential classifications (`travel` or `devops`), but with the `travel` category being the clear choice when no other context is provided. When additional context *is* provided, however, we can see that the query `driver AND taxi` gets appropriately classified to the `travel` category, while `driver AND install` gets appropriately classified to the `devops` category.

Because of the ability for the semantic knowledge graph to find semantic relationships using the context of any arbitrary query, this makes it an ideal tool for on-the-fly classification of arbitrarily-complex incoming queries.

Not only do queries need to be classified, but we just saw an example of an ambiguous term (`driver`) that further needs to have its multiple meanings differentiated so the search engine use the correct interpretation. This can be accomplished by adding just one more graph traversal to our query, which we'll walk through next.

5.4.9 Using semantic knowledge graphs for query-sense disambiguation

One of the hardest challenges in interpreting users' intent from their queries is understanding exactly what they mean by each word. The problem of polysemy, or ambiguous terms, can significantly affect your search results.

For example, if someone comes to your search engine and searches for the term "driver", this could have many different possible meanings. Some of these meanings include:

1. A vehicle operator (the taxi driver)
2. Software which makes part of a computer work (install a printer driver or other device driver)
3. A kind of golf club (swing the driver)
4. A kind of tool (i.e. screwdriver)
5. Something that moves an effort forward (key driver of success)

Likewise if someone searches for a "server", this could mean someone who takes orders and waits on tables at a restaurant, or it could mean a computer that runs some software as a service.

Ideally we want our search engine to be able to disambiguate each of these word senses and generate a unique list of related terms within each disambiguated context.

In section 5.4.9, we demonstrated how to use the semantic knowledge graph to automatically classify queries into a set of known categories. Given that we already know how to classify our queries, it is trivial to add an additional traversal after the query classification to contextualize the list of related terms to each specific query classification.

In other words, by traversing from query to classification and then to terms, we are able to generate a list of terms that describe a contextualized interpretation of the original query within each of the top classifications.

Listing 5.13 demonstrates a function which will execute this kind of disambiguating query against the semantic knowledge graph.

Listing 5.13 Disambiguating a query's intent across different contexts

```
def run_disambiguation_query(query,keywords_field="body",context_field="category",
    keywords_limit=10,context_limit=5,min_occurrences=5):

    disambiguation_query = {
        "params": {
            "qf": keywords_field,
            "fore": "{!type=$defType qf=$qf v=$q}",
            "back": "*:*",
            "defType": "edismax",
            "rows": 0,
            "echoParams": "none",
            "omitHeader": "true"
        },
        "query": query,
        "facet": {
            "context":{
                "type": "terms",
                "field": context_field,
                "sort": { "context_relatedness": "desc"},
                "mincount": min_occurrences,
                "limit": context_limit,
                "facet": {
                    "context_relatedness": {
                        "type": "func",
                        "func": "relatedness($fore,$back)"
                    },
                    "keywords": {
                        "type": "terms",
                        "field": keywords_field,
                        "mincount": min_occurrences,
                        "limit": keywords_limit,
                        "sort": { "keywords_relatedness": "desc"},
                        "facet": {
                            "keywords_relatedness": {
                                "type": "func",
                                "func": "relatedness($fore,$back)"
                            }
                        }
                    }
                }
            }
        }
    }

    search_results = requests.post(solr_url + collection + "/select",
    json=disambiguation_query).json()

    print("Query: " + query)
    for context_bucket in search_results["facets"]["context"]["buckets"]:
        print("  Context: " + str(context_bucket["val"]) + "  " +
        str(context_bucket["context_relatedness"]["relatedness"]))
        print("    Keywords: ")
        for keywords_bucket in context_bucket["keywords"]["buckets"]:
            print("      " + str(keywords_bucket["val"]) + "  " +
            str(keywords_bucket["keywords_relatedness"]["relatedness"]))
        print ("\n")
```

By traversing first to a specific context (category field) and then to keywords (body field), we can find the most related contexts and then the most related terms to the original query that are specific to that context. You can see from this listing that a `context` field (the `category` field by default) and a `keywords` field (the `body` field by default) are used as part of a two-level traversal. For any query that is passed in, we first find the most semantically-related category, and then within that category we find the most semantically related terms to the original query within that category.

Listing 5.14 demonstrates how to call this function, passing in three different queries containing ambiguous terms for which we want to find differentiated meanings, and Table 5.1 demonstrates the results of these three graph traversals.

Listing 5.14 Running Query Disambiguation for Several Queries. Each disambiguation context (`category` field) is scored relative the query, and each discovered keyword (`body` field) is scored relative to both the query and the disambiguation context.

```
run_disambiguation_query( query="server", context_field="category", keywords_field="body" )
run_disambiguation_query( query="driver", context_field="category", keywords_field="body",
context_limit=2 )
run_disambiguation_query( query="chef", context_field="category", keywords_field="body",
context_limit=2 )
```

The results of the queries in Listing 5.14 can be found in Table 5.1 - 5.3.

Query: server		
Context: devops 0.787 Keywords: server 0.91786 servers 0.69526 docker 0.66753 code 0.65852 configuration 0.60976 deploy 0.60332 nginx 0.5847 jenkins 0.57877 git 0.56514 ssh 0.55581	Context: scifi -0.27326 Keywords: server 0.56847 computer 0.16903 computers 0.16403 servers 0.14156 virtual 0.12126 communicate 0.09928 real 0.098 storage 0.09732 system 0.08375 inside 0.0771	Context: travel -0.28334 Keywords: server 0.74462 tipping 0.47834 tip 0.39491 servers 0.30689 vpn 0.27551 tips 0.19982 restaurant 0.19672 bill 0.16507 wage 0.1555 restaurants 0.15309

Table 5.1. Contextualized related terms lists by category for the query "server"

Table 5.1 shows the top most semantically-related categories for the query `server`, followed by the most semanticall-related keywords from the `body` field within each of those contexts. Based upon the data, we see that the category of `devops` is the most semantically related (positive score of 0.787), whereas the next two categories both contained substantially negative scores (-0.27326 for `scifi` and 0.28334 for `travel`). This indicates that when someone searches for the query `server`, the devops category is overwhelmingly the most likely category in which that term is going to be meaningful.

If we look at the different terms lists that come back for each of the categories, we also see several distinct meanings arise. In the devops category a very specific meaning of the term server is intended, specifically focused on tools related to managing, building, and deploying code to a computer server. In the scifi category, a more general understanding of a server is communicated, but still related to computing. In the travel category, on the other hand, the overwhelming sense of the word server is related to someone working in a restaurant, as we see terms like tipping, restaurant, and bill showing up. Interestingly, one particular kind of computer server, as vpn also shows up, because this is the one kind of server that is highly recommended for people to use when traveling to protect their internet communications.

When implementing an intelligent search application using this data, if you know the context of the user is related to travel, it would make sense to use the specific meaning within the travel category. Absent that kind of context, however, the best choice it to typically choose either the most semantically-related category or the most polular category among your users.

The results of the queries in Listing 5.14 can be found in Table 5.1 - 5.3.

Query: driver	
Context: travel 0.23835 Keywords: driver 0.91524 drivers 0.68676 taxi 0.6008 car 0.54811 license 0.51488 driving 0.50301 taxis 0.45885 vehicle 0.45044 drive 0.43806 traffic 0.43721	Context: devops 0.04461 Keywords: driver 0.71977 ipam 0.70462 aufs 0.63954 overlayfs 0.63954 container_name 0.63523 overlay2 0.56817 cgroup 0.55933 docker 0.54676 compose.yml 0.52032 compose 0.4626

Table 5.2. Contextualized related terms lists by category for the query "driver"

Table 5.2 demonstrates a query disambiguation for the query "driver". In this case, there are two related categories, with travel being the most semantically-related (score: 0.23835) and devops being much less semantically-related. We can see two very distinct meanings of driver appear within each of these contexts, with driver in the travel category being related to taxi, car, license, driving, and vehicle, whereas within the devops category driver is related to ipam, aufs, overlayfs, and so on, which are all different kinds of computer-related drivers.

If someone searches for the word driver in your search engine, they clearly do not intend to find documents about both of these meanings of the word driver, and they might be confused if you included both meanings in your search results by only searching for the string `driver` in your inverted index. There are several ways to deal with multiple potential meanings for queried keywords, such as grouping results by meaning to highlight the differences, choosing only the most likely meaning, carefully interspersing different meanings within the search results to provide diversity, or providing alternative query suggestions for different contexts, but usually an intentional choice here is much better than just lazily lumping multiple different meaning together. By leveraging a semantic interpretation of the query like this section demonstrates, you can much better understand your users' intent and deliver more relevant, contextualized search results.

As a final example, Table 5.3 demonstrates the query disambiguation for the query "chef".

Query: chef	
Context: devops 0.4461 Keywords: chef 0.90443 cookbooks 0.76403 puppet 0.75893 docs.chef.io 0.71064 cookbook 0.69893 ansible 0.64411 www.chef.io 0.614 learn.chef.io 0.61141 default.rb 0.58501 configuration 0.57775	Context: cooking 0.15151 Keywords: chef 0.82034 cooking 0.29139 recipe 0.2572 taste 0.21781 restaurant 0.2158 cook 0.20727 ingredients 0.20257 pan 0.18803 recipes 0.18285 fried 0.17033

Table 5.3. Contextualized related terms lists by category for the query "chef"

The top two contexts for the query `chef` both show reasonably positive relatedness scores, indicating that both meanings are likely interpretations. While the `devops` context has a higher score (0.4461) than the `cooking` context (0.15151), it would still be important to take the user's context into consideration as best as possible when choosing between these two meanings. The meaning of `chef` witin the `devops` context is related to the Chef configuration management software used to build and deploy servers (related terms include `puppet`, `ansible`, etc.) whereas within the cooking context it is referring to a person who prepares food (`cooking`, `taste`, `restaurant`, `ingredients`, etc.).

Interestingly, the Chef software makes used of the idea of "recipies" and "cookbooks" in its terminology, which was originally borrowed from the idea of a chef in a kitchen, so we may even see overlap between terms as we go further down the list, even thought those other terms are actually also ambiguous across the two broad classifications we are using (`devops` and `cooking`)

Of course, if you have a more fine-grained classification available on your documents, you may be able to derive even more nuanced, contextualized interpretations of your users' queries, making a semantic knowledge graph highly effective at nuanced query interpretation and expansion. By combining query classification, term disambiguation, and query expansion together, a semantic knowledge graph can power enhanced domain-specific and highly contextualized semantic search capabilities within your AI-powered search engine.

5.5 Using knowledge graphs for semantic search

By providing the ability to accept arbitrary queries and dynamically discover related terms on the fly in a very context sensitive way, semantic knowledge graphs become a key tool for query interpretation and relevance ranking. We’ve seen that not only can semantic knowledgege graphs help interpret and expand queries, but that they also provide the ability classify queries and keywords on the fly, as well as to disambiguate multiple senses of the terms in each query.

We also explored early in the chapter how to build expicit knowledge graphs, both manually and through open information extraction techniques, and how to traverse those graphs to pull in useful facts. What may not be obvious yet, however, is how to actually parse arbitrary incoming queries and look up the appropriate pieces in the knowledge graph. We’ll spend the majority of chapter 7 covering how to build out an end-to-end semantic search system which can parse queries and integrate each of these knowledge graph capabilities. Before we do that, however, there are some very specific kinds of relationships we need to add to our knowledge graph that are particularly important for search engines, such as misspellings, synonym, and domain-specific phrases. We’ll cover how to automatically learn each of these sources of domain-specific terminology from your user signals or your content as we move into the next chapter on learning domain-specific language.

5.6 Summary

- Knowledge graphs model the relationships between entities within your domain and can be built explicitly with known relationships or can be extracted dynamically from your content.
- Open information extraction, the process of extracting facts from your content (subject, relationship, object triples) can be used to learn arbitrary relationships (typically results in noisy data) or to extract hyponym/hypernym relationships (less noisy) from text into an explicit knowledge graph.
- Semantic knowledge graphs enable traversal and ranking of arbitrary semantic relationships between any content within your search index. This allows you to use your content directly as a knowledge graph without any data modeling required beyond just indexing your content.
- Content-based recommendations that don't rely on user signals can be generated by ranking the most semantically-interesting terms and phrases from documents and using them as a query to find and rank other related documents.
- Semantic knowledge graphs enable better understanding of user intent by powering domain- and context-sensitive query expansion and rewriting, relationship discovery, query classification, and query-sense disambiguation.

8 Signals Boosting Models

This chapter covers:

- Aggregating user signals to create a popularity-based ranking model
- Normalizing signals to best enhance relevance for noisy query input
- Fighting signal spam and user manipulation of crowdsourced signals
- Applying time decays to prioritize recent signals as more relevant
- Blending multiple signal types together into a unified signals boosting model
- Scaling signals boosting for flexibility and performance using query time vs. index-time signals boosting.

In chapter 4, we covered crowdsourced relevance and techniques for implementing it utilizing reflected intelligence and feedback loops. We covered three different categories of reflected intelligence: Signals Boosting (popularized relevance), Collaborative Filtering (personalized relevance), and Learning to Rank (generalized relevance). In this chapter, we'll dive deeper into the first of these - implementing Signals Boosting to enhance the relevance ranking of your most popular queries and documents.

For most search engines, you will find that a relatively small number of popular queries tend to make up a significant portion of your total query volume. These popular queries also tend to lead to more clicks and other positive signals (such as purchases in an ecommerce use case), which enable stronger inferences about the popularity of top search results for any given query.

While we will explore personalized relevance in chapter 9 and generalized relevance in chapters 10 and 11, this chapter will dive deeper into implementing "popularized relevance" through direct boosting of search results based upon signals boosting models. Signals boosting models are the key to ensuring our most important and highest-visibility queries are best tuned to return the most relevant documents.

8.1 Basic signals boosting

In section 4.3.2, we built our first signals boosting model on the Retrotech dataset, enabling a significant boost in relevance for the most frequently searched and clicked search results. In this section, we'll recap that process of creating a simple signals boosting model, which we will build upon in the upcoming sections the cater to some more advanced needs.

Figure 8.1 demonstrates the before and after searches for the query `ipad` previously demonstrated in section 4.3.2.

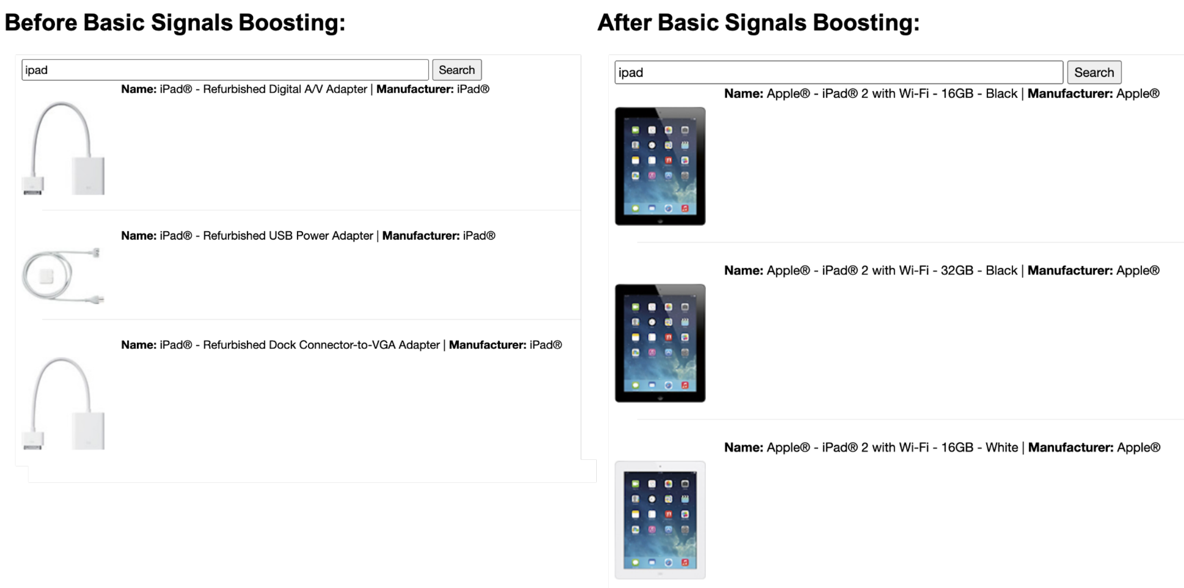


Figure 8.1 Search results before and after applying a signals boosting model.

The signals boosting model that led to the improved relevance in Figure 8.1 is a basic signals boosting model. It looks at all documents ever clicked for a given query, and then applies a boost equal to the total number of past clicks on that document for that query.

While the basic signal boosting model covered in section 4.3.2 provides greatly improved relevance, it is unfortunately succeptible some data biases and even manipulation. In the next section, we'll discuss some techniques for removing noise in the signals to maximize the quality your signals boosting models and reduce the opportunity for undesirable biases.

8.2 Normalizing Signals

In this section, we'll discuss the importance of normalizing incoming user queries prior to aggregation. The signals boosting model demonstrated in the last section generates aggregated boosts for each query and document pair, but given that end users can enter any arbitrary text as a query, this means that the aggregated signals are inherently noisy. Specifically, since the incoming queries have not been normalized into a common form, this means that variations of a query are likely to be treated as entirely separate queries. Listing 8.1 demonstrates a list of all queries with boosted documents for the most popular model of IPad from our Retrotech dataset.

Listing 8.1 Find the most popular queries associated with the most popular iPad model

```
query = "885909457588" #most popular iPad

def show_raw_boosted_queries(signals_boosting_collection):
    signals_boosts_query = {
        "query": "\"" + query + "\"",
        "fields": ["query", "boost"],
        "limit": 20,
        "params": {
          "defType": "edismax",
          "qf": "doc",
          "sort": "boost desc"
        }
    }

    signals_boosts = requests.post(solr_url + signals_boosting_collection
                     + "/select", json=signals_boosts_query).json()["response"]["docs"]

    boosted_queries = ""
    for entry in signals_boosts:
        boosted_queries += '"' + entry['query'] + '" : ' + str(entry['boost']) + "\n"

    print("Raw Boosted Queries")
    print(boosted_queries)

signals_boosting_collection = "basic_signals_boosts"
show_raw_boosted_queries(signals_boosting_collection)
```

Results:

```
Raw Boosted Queries
"iPad" : 1050
"ipad" : 966
"Ipad" : 829
"iPad 2" : 509
"ipad 2" : 347
"Ipad2" : 261
"ipad2" : 238
"Ipad 2" : 213
"I pad" : 203
"i pad" : 133
"IPad" : 77
"Apple" : 76
"I pad 2" : 60
"apple ipad" : 55
"Apple iPad" : 53
"ipads" : 43
"tablets" : 42
"apple" : 41
```

```
"iPads" : 38
"i pad 2" : 38
```

You can see from the output of Listing 8.1 that many variations of the same queries exist in the basic signals boosting model. The biggest culprit of the variations seems to be case-sensitivity, as we see "iPad", "ipad", "Ipad", and "IPad" as common variants. Spacing appears to be another issue, with "ipad 2" vs "i pad 2" vs. "ipad2". We even see singular vs. plural representations in "ipad" vs. "ipads".

Given that most keyword search fields are case insensitive, and that many also ignore plural representations of terms and split on case changes and letter to number transitions between words, keeping separate query terms and boosts for variations that are non-distinguishable by the search engine can be counter productive. Not only is it unnecessary, but is actually diffuses the value of your signals, since the signals are divided across different variations of the same keywords with lower boosts as opposed to being coalesced into more meaningful queries with stronger boosts.

It is up to you to figure out how sophisticated your query normalization should be prior to signals aggregation, but even just lowercasing incoming queries to make the signals aggregation case insensitive can go a long way. Listing 8.2 demonstrates the same basic signals aggregation as before, but this time with the queries lowercased first.

Listing 8.2 Basic case-insensitivity normalization of boosted queries

```
signals_collection = "signals"
signals_boosting_collection = "normalized_signals_boosts"

normalized_signals_aggregation_query = """
      select lower(q.target) as query,   ❶
          c.target as doc,
          count(c.target) as boost   ❷
        from signals c left join signals q on c.query_id = q.query_id
        where c.type = 'click' AND q.type = 'query'
        group by query, doc   ❷
        order by boost desc
        """

aggregate_signals(signals_collection, signals_boosting_collection,
    normalized_signals_aggregation_query)

show_raw_boosted_queries(signals_boosting_collection)
```

❶ Normalizing case by lowercasing each query

❷ Grouping by normalized query increases the count of signals for that query, increasing the signal boost

Results:

```
Raw Boosted Queries
"ipad" : 2939
"ipad 2" : 1104
"ipad2" : 540
```

```
"i pad" : 341
"apple ipad" : 152
"ipads" : 123
"apple" : 118
"i pad 2" : 99
"tablets" : 67
"tablet" : 61
"ipad 1" : 52
"apple ipad 2" : 27
"hp touchpad" : 26
"ipaq" : 20
"i pad2" : 19
"wi" : 19
"apple computers" : 18
"apple i pad" : 15
"ipad 2 16gb" : 15
"samsung galaxy" : 14
```

That list of raw boosted queries is already looking much cleaner! Not only is there less redundancy, but you'll notice that the strength of the signals boosts has increased, because more signals are being attributed to a canonical form of the query (the lowercased version).

Often just lowercasing the queries, and maybe removing whitespace or extraneous characters, is sufficient normalization of queries prior to signals aggregation. The important takeaway from this section, though, is that the signals boosting model becomes stronger the better you are able to ensure that identical queries are treated identically when they are aggregated together.

Variations in queries aren't the only kind of noise we need to worry about in our data, however. In the next section, we'll talk about how to overcome significant potential problems caused by spam in our user-generated click signals.

8.3 Fighting Signal Spam

Anytime we use crowdsourced data, such as click signals, to influence the behavior of the search engine, we need to ask ourselves "How might our users manipulate the data inputs to create an undesirable result?". In this section, we'll demonstrate how a user could spam the search engine with click signals to manipulate search results, and we'll show you how to stop it.

8.3.1 Using signal spam to manipulate search results

Let's imagine we have a user who, for whatever reason, really hates Star Wars and thinks that the most recent movies are complete garbage. They feel so strongly, in fact, they they want to ensure any searches for `star wars` always return a trash can as the top search result. This user knows a thing or two about search engines and has noticed that your killer relevance algorithms seem to be leveraging user signals and signals boosting. Figure 8.2 shows the default response for the query `star wars`, with signals boosting bringing the most popular products to the top of the search results.

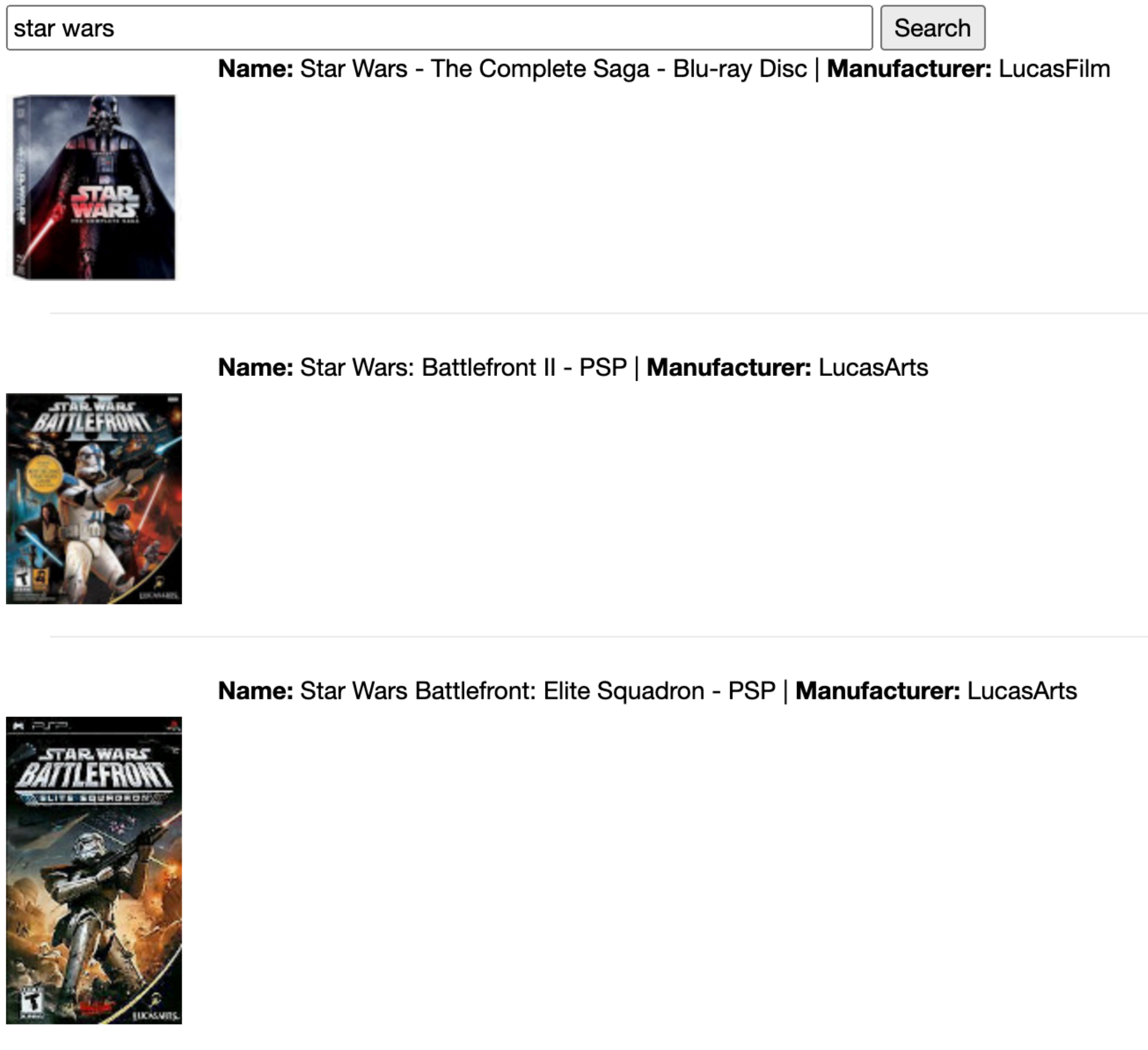


Figure 8.2 Most popular search results for the query "star wars", with signals boosting turned on.

The user decides that since your search engine ranking is based upon popular items, that they will spam the search engine with a bunch of searches for `star wars` and follow up with a bunch of fake clicks on the Star Wars themed trash can they found, in order to try to make the trash can show up at the top of the search results.

In order to simulate this scenario, we'll run a simple script in Listing 8.3 to generate 5000 queries for `star wars` and 5000 corresponding clicks on the trash can after running that query.

Listing 8.3 Generating spam queries and clicks to manipulate the ranking of a document due to signals boosting.

```
....
import datetime

spam_user = "u8675309"
spam_query = "star wars"

spam_signal_boost_doc_upc = "45626176"   ❶

num = 0
while (num < 5000):   ❷
    query_id = "u8675309_0_" + str(num)

    next_query_signal = {
        "query_id": query_id,
        "user": spam_user,
        "type":"query",
        "target": spam_query,
        "signal_time": datetime.datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ"),
        "id":"spam_signal_query_" + str(num)
    }

    next_click_signal = {
        "query_id": query_id,
        "user": spam_user,
        "type":"click",
        "target": spam_signal_boost_doc_upc,
        "signal_time": datetime.datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ"),
        "id":"spam_signal_click_" + str(num)
    }

    collection = "signals"
    requests.post(solr_url + collection + "/update/json/docs", json=next_query_signal)   ❷
    requests.post(solr_url + collection + "/update/json/docs", json=next_click_signal)   ❷
    num+=1

requests.post(solr_url + collection + "/update/json/docs?commit=true")   ❸

signals_collection = "signals"
signals_aggregation_collection = "signals_boosts_with_spam"
aggregate_signals(signals_collection, signals_aggregation_collection,
        normalized_signals_aggregation_query)   ❹
```

❶ Document for trash can the spammer wants to move to the top of the search results

❷ Send 5,000 query and click signals to the search engine

❸ Commit the signals to the index

❹ Run the signals aggregation to generate the signals boosting model including the spammy signals

Listing 8.3 sends thousands of spammy query and click signals to our search engine, modeling the same outcome we would see if a user searched and clicked on a particular search result thousands of times. The listing then re-runs the basic signals aggregation to see the impact those signals have on our signals boosting model.

To see the impact on our search results, Listing 8.4 runs a search for the query `star wars`, now incorporating the manipulated signals boosting model in order to see the effect of the malicious user's spammy click behavior.

Listing 8.4 Search results for query "star wars" using the manipulated signals boosting model

```
query = "star wars"
collection = "products"

signals_boosts = get_query_time_boosts(query, "signals_boosts_with_spam")   ❶
boosted_query = get_main_query(query, signals_boosts)

search_results = requests.post(solr_url + collection + "/select",
        json=boosted_query).json()["response"]["docs"]
print(search_results)
display(HTML(render_search_results(query, search_results)))   ❸
```

❶ Load signals boosts from the signals boosting model that included the spammy signals

❷ Boost the "star wars" using the signals boosting model

❸ Display the search results

Figure 8.3 shows the new manipulated search results generated from Listing 8.4, with the Star Wars trash can returned in the top spot.

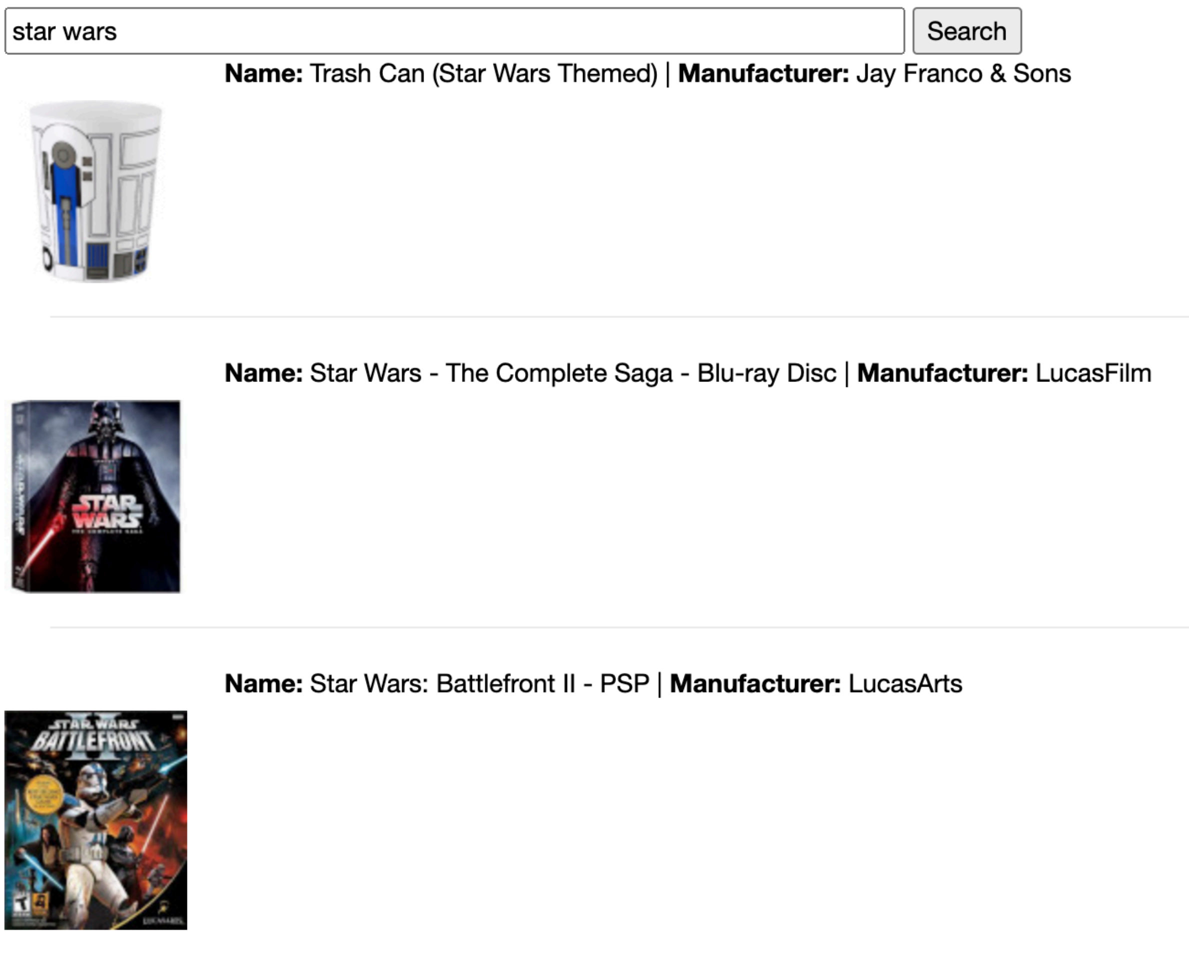


Figure 8.3 Search results manipulated by a user spamming the search engine with fake signals to affect the top result.

The spammer was successful, and these manipulated search results will now be seen by every subsequent visitor to the Retrotech website who searches for `star wars`! Looks like we're going to need to make our signals boosting model more robust to combat this kind of signal spam from malicious users.

8.3.2 Combatting signal spam through user-based filtering

If you are going to use crowdsourced data like user signals to influence your search engine ranking, then it is important to take steps to minimize the ability for your users to manipulate your signals-based ranking algorithm.

In order to combat the "Star Wars trash can" problem we just demonstrated, the simplest technique to start would be to ensure that duplicate clicks by the same user only get one "vote" in the signals boosting aggregation. That way, whether a malicious user clicks one time or a million times, their clicks only count as one signals and therefore have no material impact on the signals boosting model. Listing 8.5 reworks the signals aggregation query to only count unique click signals from each user.

Listing 8.5 Deduplicating signals per user to prevent undue influence by a single user

```
signals_collection = "signals"
signals_aggregation_collection = "signals_boosts_anti_spam"

anti_spam_aggregation_query = """
  select query, doc, count(doc) as boost from (
    select c.user, lower(q.target) as query, c.target as doc,
    max(c.signal_time) as boost    ❷
    from signals c left join signals q on c.query_id = q.query_id
    where c.type = 'click' AND q.type = 'query'
    group by c.user,    ❶
      q.target, c.target
  ) as x
  group by query, doc
  order by boost desc
"""

aggregate_signals(signals_collection, signals_aggregation_collection, anti_spam_aggregation_query)
```

❶ Group by user the limit each user to only one "vote" per query/doc pair in the signals boosting model

❷ Signal date is the most recent signal from the user only if there are duplicates

If we re-run the `star wars` query from Listing 8.4 with this new `signals_boosts_anti_spam` model, we'll now see that our normal search results have returned and look the same again as Figure 8.2. This is because the extra, spammy signals from our malicious user have now all been reduced to a single bad signal, which we show in Table 8.1.

You can see that the aggregated signal count in the "signals_anti_spam" model has a total much closer to the `normalized_signals_boosts` model that we built before the spam signals were generated. Since each user is limited to one signal per query/document pair in the `signals_boosts_anti_spam` model, the ability for users to manipulate the signal boosting model is now substantially reduced.

Table 8.1 The 5000 spammy signals have been deduplicated to one signal in the antispam signal boosting model model

model	query	doc	boost
before spam signals (`normalized_signals_boosts`)	star wars	400032015667	0 (no signals yet)
after spam signals (`normalized_signals_boosts`)	star wars	400032015667	5000
after spam signals (`signals_boosts_anti_spam`)	star wars	400032015667	1

You could, of course, identify any user accounts that appear to be spamming your search engine and remove their signals entirely from your signals boosting aggregation, but reducing the reach of the signals through deduplication is simpler and often accomplishes the same end goal of restoring a good crowdsourced relevance ranking.

In our example from Listing 8.6, we leveraged user ids as the key identifier to deduplicate spammy signals, but any identifier will work here: user id, session id, browser id, IP address, or even some kind of browser fingerprint. As long as you find some identifier to unique identify users or to otherwise identify low-quality traffic (like bots and web scrapers), then you can use that information to deduplicate signals. If none of those techniques work and you have too much noise in your click signals, you can also choose to only look at click signals from known (authenticated) users who you presumably have much more confidence in being legitimate traffic.

One final way to mitigate signal spam is to find a way to separate the important signal types from the noisy ones that can be easily-manipulated. For example, generating signals from running queries and clicking on search results is easy. Signals from purchasing a product are much harder to manipulate, however, as they typically require users to log in or enter payment information before a purchase will be recorded. The odds of someone maliciously purchasing 5,000 Star Wars trash cans are quite low, because there are multiple financial and logistical barriers to doing this.

Not only is it valuable to weight purchases as stronger signals than clicks from the standpoint of fighting spam, it is also valuable from a relevance standpoint to give purchases a higher weight, because they are more clear indicators of intent than just clicks. In the next section, we'll walk through exactly how to combine different signal types into a signals boosting model that considers the relative importance of each different signal type.

8.4 Combining multiple signal types

Thusfar we've only worked with two signals types - queries and clicks. For some search engines (such as web search engines), click signals may be the only good source of crowd-sourced data available to build a signals boosting model. Often times, however, many different signal types exist that can provide additional and often much better inputs for building a signals boosting model.

In our Retrotech dataset, we have several signal types that are common to ecommerce use cases:

- query
- click
- add-to-cart
- purchase

While clicks in response to queries are helpful, they don't necessarily imply a strong interest in the product, as someone could just be browsing to see what's available. If someone adds a product to their shopping cart, this typically represents a much stronger signal of interest than a click. A purchase is then an even stronger signal that a user is interested in a product, as the user is willing to pay money to receive the item for which they searched.

While some ecommerce websites may receive enough traffic to ignore click signals entirely and only focus on add-to-car and purchase signals, it is often more useful to make use of all signal types when calculating signals boosts. Thankfully, combining multiple signal types is as simple as just assigning relative weights as multipliers to each signal type when performing the signals aggregation:

```
signals_boost = (1 * sum(click_signals)) + (10 * sum(add_to_cart_signals))
            + (25 * sum(purchase_signals))
```

By counting each click as 1 signal, each add-to-cart as 10 signals, and each purchase as 25 signals, this makes each purchase carry 25 times as much weight in the signals boosting model than just a click. In other words, 25 different people would need to click on a product in response to a query to count as much as one person actually purchasing the product as a result of the same query.

This helps reduce noise from less reliable signals and boost more reliable signals, while still making use of the large volume of less reliable signals in cases (like obscure items) where better signals are less prevalent. Listing 8.7 demonstrates a signals aggregation designed to combine different signal types with different weights.

Listing 8.6 Combining multiple signal types with different weights

```
signals_collection="signals"
signals_aggregation_collection="signals_boosts_weighted_types"

mixed_signal_types_aggregation = """
select query, doc,
( (1 * click_boost) + (10 * add_to_cart_boost) + (25 * purchase_boost) ) as boost   ❶
from (
  select query, doc,
    sum(click) as click_boost,   ❷
    sum(add_to_cart) as add_to_cart_boost,   ❷
    sum(purchase) as purchase_boost   ❷
  from (
      select lower(q.target) as query, cap.target as doc,
        if(cap.type = 'click', 1, 0) as click,
        if(cap.type = 'add-to-cart', 1, 0) as  add_to_cart,
        if(cap.type = 'purchase', 1, 0) as purchase
      from signals cap left join signals q on cap.query_id = q.query_id
      where (cap.type != 'query' AND q.type = 'query')
    ) raw_signals
  group by query, doc
) as per_type_boosts
"""

aggregate_signals(signals_collection, signals_aggregation_collection,
        mixed_signal_types_aggregation)
```

❶ Multiple signals combined with different relative weights toward to total boost value

❷ Each signal type gets summed up independently before being combined

You can see from the SQL query that the overall boost for each query / document par is calculated by counting all clicks with a weight of 1, counting all add-to-cart signals and multiplying them by a weight of 10, and then counting all purchase signals and multiplying them by a weight of 25.

These suggested weights of 10x for add-to-cart signals and 25x for purchase signals should work well in practice in many ecommerce scenarios, but these relative weights are also fully configurable for each domain. Your website may be set up such that almost everyone who adds a product to their cart purchases the product (for example, a grocery store delivery app, where the only purpose of using the website is to fill a shopping cart and purchase). In these cases, you could find that adding an item to a shopping cart adds no additional value, but that *removing* an item from a shopping cart should actually carry a negative weight indicating the product is a bad match for the query.

In this case, you may want to introduce the idea of *negative signals boosts*. Just as we've discussed clicks, add-to-carts, and purchases as signals of user intent, your user experience may also have numerous ways to measure user dissatisfaction with your search results. For example, you may have a thumbs-down button, a remove from cart button, or you may be able to track product returns after a purchase. You may even want to count documents in the search results which were skipped over, and record a "skip" signal for those documents to indicate the user saw them but didn't show interest. We'll cover the topic of managing clicked versus skipped documents further in chapter 11 when we discuss click modeling.

Thankfully, handling negative feedback is just as easy as handling positive signals: instead of assigning increasingly positive weights to signals, you can assign increasingly negative weights to negative signals. For example:

```
type_based_signal_weight =
1 * sum(click_signals) + ( 10 * sum(add_to_cart_signals) ) + ( 25 * (purchase_signals) )
    + ( -0.25 * sum(skipped_doc_signals) ) + ( -20 * sum(remove_from_cart_signals) )
    + ( -100 * sum(returned_item_signals) )
    + ( -50 * sum(negative_post_about_item_in_review_signals) )
```

This simple, linear function provides a highly configurable signals-based ranking model, taking in multiple input parameters and returning a ranking score based upon the relative weights of those parameters. You can combine as many useful signals as you want into this weighted signals aggregation to improve the robustness of the model. Of course, tuning the weights of each of the signal types to achieve an optimal balance may take some effort. You can do this manually, or you can leverage a machine learning technique called Learning to Rank to do this. We'll explore Learning to Rank in-depth in chapters 10 and 11.

Not only is it important to weight different kinds of signals relative to each other, but it can also sometimes be necessary to weight the *same* kind of signals differently against each other. In the next section, we'll discuss one key example of doing this: assigning higher value to more recent interactions.

8.5 Time decays and short-lived signals

Signals don't always maintain their usefulness indefinitely. In the last section, we showed how signals boosting models can be adjusted to weight different kinds of signals as more important than others. In this section, we'll address a different challenge - factoring in the "temporal value" of signals as they age and become less useful.

Imagine three different search engine use cases: an ecommere search engine with stable products, a job search engine, and a news website. If we have an ecommerce search engine, like Retrotech, the documents (products) often stay around for years, and the best products are often those that have a long track record of interest.

If we have a job search engine, the documents (jobs) may only stick around for a few weeks or months until the job is filled, and then they disappear forever. While the documents are present, however, newer clicks or job applications aren't necessarily any more important as signals than older interactions.

In a news search engine, while the news articles stick around forever, newer articles are generally way more important than older articles, and newer signals definitely are more important than older signals, as people’s interests change on a daily, if not hourly basis.

Let’s dive into these usecases and demonstrate how to best handle signals boosting for time-sensitive documents vs. time-sensitive signals.

8.5.1 Handling time-sensitive documents

In our Retrotech use case, our documents are intentionally old, having been around for a decade or more, and interest in them likely only increases as the products become older and more "retro". As such, we don’t often have massive spikes in popularity for items, and newer signals don’t necessarily carry significantly more importance than older signals. This type of use case is a bit atypial, but plenty of search use cases do deal with more "static" document sets like this. The best solution in this case is the strategy we’ve already taken thusfar in this chapter: to process all signals within a reasonable time period of months or years and give them fairly equal weight. When all time periods carry the same weight, this also means that the signals boosting model likely doesn’t need to be rebuilt that often, since the model only changes slowly over time and the frequent processing of signals is unnecessary computational overhead.

In a job search use case, however, the scenario is very different. For the sake of argument, let’s say that on average it takes 30 days to fill a job opening. This means the document representing that job will only be present in the search engine for 30 days, and that any signals collected for that document are only useful for signals boosting during that 30 days window. When a job is posted, it will typically be very popular for the first few days since it is new and is likely to attract many existing job seekers, but all interactions with that job at any point during the 30 days are just as useful. In this case, all click signals should get an equal weight, and all job application signals should likewise receive an equal weight (at a weight higher than the click signals, of course). Given the very short lifetime of the documents, however, it is important that all signals are used as quickly as possible in order to make the best use of their value.

Use cases with short-lived documents, like in the job search use case, don’t usually actually make good candidates for signals boosting, as the documents often get deleted by the time the signals boosting model becomes any good. As a result, it can often make more sense to look at personalized models (like collaborative filtering, covered in chapter 9) and generalizable relevance models (like Learning to Rank, covered in chapters 10 and 11) for these use cases instead.

In both the Retrotech use case and the job search use case, the signals were just as useful for the entire duration of the document’s existence. In the news search use case, which we’ll see next, the time sensitivity is more related to the age of the document and the signals themselves.

8.5.2 Handling time-sensitive signals

In a news search engine use case, the most recently published news gets the most visibility and usually the most interaction, so most recent signals are considerably more valuable than older signals. Some news items may be very popular and relevant for days or longer, but generally the signals from the last ten minutes are more valuable than the signals from the last hour, which are more valuable than the signals from the last day, and so on. News search is an extreme use case where signals both need to be processed quickly and where more recent signals need to be weighted as substantially more important than older signals.

One easy way to model this is by using a decay function, such as a half-life function, which cuts the weight assigned to a signal by half (50%) over equally-spaced time spans. For example, a decay function with a half-life of 30 days would assign 100% weight to a signal that happens `now` , 75% weight to a signal from 15 days ago, 50% weight to a signal from 30 days ago, 25% weight to a signal from 60 days ago, 12.5% weight to a signal from 90 days ago, and so on. The math for implementing a decay function is:

```
time_based_signal_weight = starting_weight * 0.5^(signal_age/half_life)
```

When applying this calculation, the `starting_weight` will usually be the relative weight of a signal based upon it's type, for example a weight of 1 for clicks, 10 for add-to-cart signals, and 25 for purchase signals. Most of the time the `starting_weight` will just be `1` unless you are combining multiple signal types, as shown previously in section 8.4.

The `signal_age` is how old the signal is, and the `half_life` is how long it takes for the signal to lose half of it's value. Figure 8.4 demonstrates how this decay function impacts signals weights over time for different half-life values.

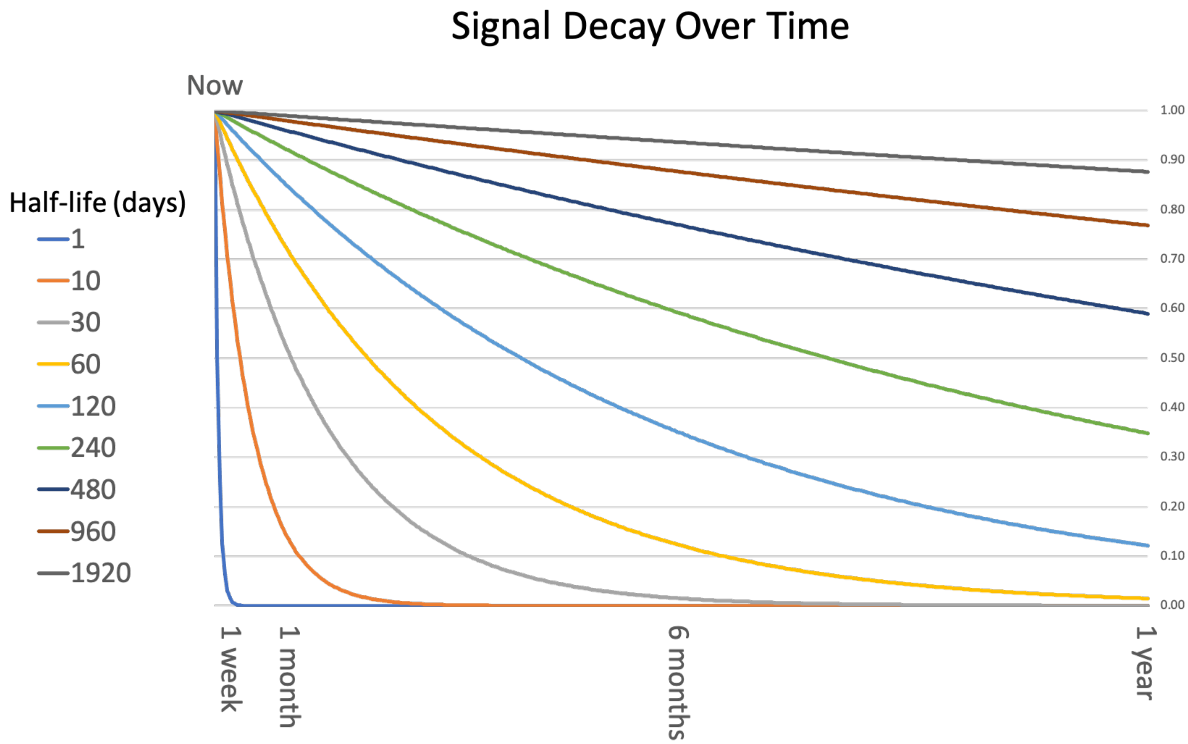


Figure 8.4 Signal decay over time based upon various half life values

The one day half-life is very aggressive and is pretty impractical in most usecases, as it is unlikely you would be able to collect enough signals in a day to power meaningful signals boosting, and the liklihood of your signals becoming irrelevant that quickly is low.

The 30 day, 60 day, and 120 day half-lifes do a good job of aggressively discounting older signals, but keeping their residual value contributing to the model over a six to twelve month period. If you have really long-lived documents, you could push out even longer, making use of signals over the course of many years. Listing 8.8 demonstrates an updated signal aggregation query that implements a half-life of 30 days for each signal:

Listing 8.7 Appling a time decay function to the signals boosting model

```
signals_collection="signals"
signals_boosting_collection="signals_boosts_time_weighted"

half_life_days = 30
target_date = '2020-06-01 00:00:00.0000' #Will usually be now(), but can be any past time
                                          # you want to emphasize signals from.
signal_weight = 1 #can make this a function to differentiate weights for different signal types

time_decay_aggregation = """
select query, doc, sum(time_weighted_boost) as boost from (
    select user, query, doc, """ + signal_weight + """ * pow(0.5, (datediff('""" + target_date
      + "', signal_time) / " + str(half_life_days) + """)) as time_weighted_boost from (
        select c.user as user, lower(q.target) as query,
               c.target as doc, max(c.signal_time) as signal_time
        from signals c left join signals q on c.query_id = q.query_id
        where c.type = 'click' AND q.type = 'query'
        AND c.signal_time <= '""" + target_date + """'
        group by c.user, q.target, c.target
    ) as raw_signals
) as time_weighted_signals
group by query, doc
order by boost desc
"""

aggregate_signals(signals_collection, signals_boosting_collection, time_decay_aggregation)
```

This decay function has a few unique configurable parameters:

- It contains a `half_life_days` parameter, which calculates a weighted average using a configurable half-life, which we've set as 30 days to start.
- It contains a `signal_weight` parameter, which can be replaced with a function returning a weight by signal type, as shown in the last section (click = 1, add to cart = 10, purchase = 25, etc.).
- It contains a `target_date` parameter, which is the date at which a signal gets the full value of `1`. Any signals before this date will be decayed based upon the half-life, and any signals after this date will be ignored (filtered out).

Your `target_date` will usually be the current date, so that you are making use of your most up-to-date signals and assigning them the highest weight. However, you could also apply it to past periods if your documents have seasonal patterns that repeat monthly or yearly.

While our product documents don't change very often, and the most recent signals aren't necessarily any more valuable than older signals, there are potentially annual patterns we could find in a normal ecommerce data set. For example, certain types of products may tend to be more popular around major holidays like Mother's day, Father's day, and Black Friday. Likewise, searches for something like a "shovel" may take on a different meaning in the summer (shovel for digging dirt) versus the winter (shovel for removing snow from the sidewalk). If you explore your signals, any number of trends may emerge for which time sensitivity should impact how your signals are weighted.

Ultimately, Signals are a lagging indicator. They are a reflection of what your users just did, but they are only useful as predictions of future behavior if the patterns learned are likely to repeat themselves.

Having now explored techniques for improving our signals models through query normalization, mitigating spam and relevance manipulation, combining multiple signal types with different relative weights, and applying time decays to signals, you should be able flexibly implement the signals boosting models most appropriate for your use case. When rolling out signals boosting at scale, however, there are two different approaches you can take to optimize for flexibility versus performance. We'll cover these two approaches in the next section.

8.6 Index-time vs. Query-time boosting: balancing scale vs. flexibility

All the the signals boosting models in the chapter have been demonstrated using *query-time boosting*, which loads signal boosts from a separate `signals_booss` collection for each user query at query time and modifies the user's query to add the boosts prior to sending it to the search engine. It is also possible to implement boosting models using *index-time boosting*, where boosts are added directly to documents for the queries to which those boosts apply. In this section, we'll discuss the benefits and tradeoffs of each of these approaches.

8.6.1 Benefits and drawbacks of query-time boosting

Query-time boosting, as we've seen, turns each query into a two step process, where each incoming user query is looked up in the `signals_boosting` collection, and then any found boosted documents are used to modify the user's query. Query-time boosting is the most common way to implement signals boosting, but it comes with both its benefits and drawbacks.

BENEFITS OF QUERY-TIME BOOSTING

Query-time boosting's primary architecural characteristic is that it keeps the main search collection (`products`) and the signals boosting collection (`*_signals_boosts`) separate. This separation provides a number of benefits, including:

1. Allowing the signals for each query to be updated incrementally by only modifying the one document representing that query
2. Allowing Boosting to be turned on or off easily by just not doing a lookup or modifying the user's query
3. Allowing different signals boosting algorithms to be swapped in at any time

Ultimately, by boosting specific documents for a given query at query time, the flexibility to change the boosts at any point in time based upon the current context is the major advantage of query-time signals boosting.

DRAWBACKS OF QUERY-TIME BOOSTING

While flexible, query time boosting also introduces some downsides with regard to query performance, scale, and relevance which may make it inappropriate for certain use cases. Specifically, query-time boosts:

1. Require an extra search to lookup boosts before the boosted search is executed, adding more processing (executing two searches) and latency (the final query has to wait on the results of the signals lookup query before being processed)
2. Doesn’t handle long-lists of documents to boost for a query in a scalable way, requiring tradeoffs between user experience and relevance versus query speed and scale
3. Doesn’t support search results pagination very well

The first downside is straight-forward, as each query essentially becomes two queries executed back-to-back, which increases the total search time. The second downside may not be as obvious, however, so it is worth exploring a bit further.

In query-time boosting, we look up a specific number of documents to boost higher in the search results for a query. For example, in our `ipad` search example from Figure 8.1 (see Listing 4.7 for the code), the boost for the query ultimately becomes:

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109
"843404073153"^73  "885909457601"^62  "635753493559"^62  "885909472376"^61
"610839379408"^29 "884962753071"^28
```

This boost contains 10 documents, but only because that is the number of boosts we requested. Assuming we only showed ten documents on the first page, then the whole first page will look good… but what if the user navigates to page 2? In this case there won’t be any boosted documents shown, because only the first 10 documents with signals for the query were boosted!

In order to boost documents for the second page, we would need to ensure we have at least enough document boosts to cover the full first two pages, which means increasing from 10 boosts to 20 boosts (modifying the "limit" parameter to 20 on the boost lookup query):

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109
"843404073153"^73  "635753493559"^62  "885909457601"^62  "885909472376"^61
"610839379408"^29  "884962753071"^28  "635753490879"^27  "885909457632"^26
"885909393404"^26  "716829772249"^23  "821793013776"^21  "027242798236"^15
"600603132827"^14 "886111271283"^14 "722868830062"^13 "092636260712"^13
```

You can thus mostly solve this problem by increasing the number of boosts looked up every time someone navigates to the "next" page, but this will very quickly slow down subsequent queries, as page 3 will require looking up and applying 30 boosts, page 10 will require 100 boosts, and so on. For a use case where only a small number of boosted documents exists for each query this is not a big problem, but for many use cases, there may be hundreds or thousands of documents that would benefit from being boosted. In our query example of `ipad`, for example, there are more than 200 documents which contain aggregated signals, so most of those documents will never be boosted at all unless someone pages very deep into the search results, and at that point the queries are likely to be slow, and at some point could even time out.

Only including a subset of the boosts presents another problem, as well: search results aren't always strictly ordered by the boost value! We've made the assumption that requesting the top 10 boosts will be enough for the first page of 10 results, but in reality the boost is only one of the factors that affects relevance. It could be that documents further down in the boost list have a higher base relevance score and that if their boosts were also loaded that they would jump up to the first page of search results.

As a result, as a user navigates from page one to two and the number of boosts loaded increases, some of the results might jump up to page one and never be seen or jump down to page two and be see again as a duplicate. When someone then moves on to page three, the results from all three pages could further get shuffled around.

Even if these results are much more relevant than search results without signals boosting applied, it doesn't make a very optimal user experience. Index-time signals boosting can help overcome these drawbacks, as we'll show in the next section.

8.6.2 Implementing Index-time signals boosting

Index-time signals boosting turns the signals boosting problem on its head - instead of boosting popular documents for queries at query time, we boost popular queries for documents at indexing time. This is accomplished by adding popular queries to a field in each document, along with their boost value. Then, at query time, we simply search on the new field, and if the field contains the term from our query then it it gets automatically boosted based upon the boost value indexed for the term.

When implementing index-time boosting, we leverage the exact same signals aggregations to generate pairs of documents and boost weights for each query. Once those signals boosts have been generated, we just have to add one additional step to our workflow: updating the products collection to add a field onto each document containing each term for which the document should be boosted, along with the associated numeric boost weight. Listing 8.9 demonstrates this additional step in our workflow.

Listing 8.8 Mapping signals boosts from a separate query-time collection to a field in the main collection

```
signals_boosts_collection="normalized_signals_boosts"
signals_boosts_opts={"zkhost": "aips-zk", "collection": signals_boosts_collection}
df = spark.read.format("solr").options(**signals_boosts_opts).load()
df.registerTempTable(signals_boosts_collection)   ❶


products_collection="products_with_signals_boosts"
products_read_boosts_opts={"zkhost": "aips-zk", "collection": products_collection}
df = spark.read.format("solr").options(**products_read_boosts_opts).load()
df.registerTempTable(products_collection)   ❷

boosts_query = """   ❸
SELECT p.*, b.signals_boosts from (
  SELECT doc, concat_ws(',',collect_set(concat(query, '|', boost))) as signals_boosts
  FROM """ + signals_boosts_collection + """ GROUP BY doc
) b inner join """ + products_collection + """ p on p.upc = b.doc
"""

products_write_boosts_opts={"zkhost": "aips-zk", "collection": products_collection,
                            "gen_uniq_key": "true", "commit_within": "5000"}   ❹
spark.sql(boosts_query).write.format("solr").options(**products_write_boosts_opts)
     .mode("overwrite").save()
```

❶ Load a previously-generated signals boosting model

❷ Register the product table so we can load from it and save back to it with boosts added

❸ Insert all keywords with signals boosts for each document into a new "signals_boosts" field on the document

❹ Save the products back to the products collection, with the updated "signals_boosts" added

The code in Listing 8.9 reads all previously-generated signals boosts for each document and then maps the queries and boosts into a new `signals_boosts` field on each product document as a comma-separated list to terms (user queries) with a corresponding signals boosting weight for each term.

This `signals_boosts` field is a specialized field in Solr containing a DelimitedPayloadBoostFilter, which allows for terms (queries) to be indexed with associated boosts that can be used to influence query-time scoring. For example, for the most popular iPad, the product document will now be modified to look as follows:

```
{...
   "id": "885909457588",
   "name": "Apple® - iPad® 2 with Wi-Fi - 16GB - Black"
   "signals_boosts": "iPad|1050, ipad|966, Ipad|829, iPad 2|509, ipad 2|347, Ipad2|261, ipad2|238,
                      Ipad 2|213, I pad|203, i pad|133, IPad|77, Apple|76, I pad 2|60,
                      apple ipad|55, Apple iPad|53, ipads|43, tablets|42, apple|41, iPads|38,
                      i pad 2|38, Ipads|35, IPad 2|35, Apple ipad|34, tablet|32 ,iPad2|31,
                      Tablet|29, Tablets|25, iPad 1|21, ipaq|20"
 ...
 }
```

At query time, this `signals_boosts` field will be searched upon, and if the query matches one or more of the values in the field, the score for that document will be boosted relative to boost value.

Listing 8.10 demonstrates how to perform a query utilizing index-time signals boosts, harnessing the `payload` function in the search engine to boost based upon the indexed payload (the boost value) associated with the user's query.

Listing 8.9 Performing a query that ranks based upon index-time signals boosts

```
query = "ipad"

def get_query(query, signals_boosts_field):
    request = {
        "query": query,
        "fields": ["upc", "name", "manufacturer", "score"],
        "limit": 3,
        "params": {
          "qf": "name manufacturer longDescription",
          "defType": "edismax",
          "indent": "true",
          "sort": "score desc, upc asc",
          "qf": "name manufacturer longDescription",
          "boost": "payload(" + signals_boosts_field + ", \"" + query + "\", 1, first)"
        }
    }

    return request

collection = "products_with_signals_boosts"
boosted_query = get_query(query, signals_boosts_field)
print("Main Query:")
print(boosted_query)

search_results = requests.post(solr_url + collection + "/select",
                        json=boosted_query).json()["response"]["docs"]
print("\nSearch Results (Basic Signals Boosting): ")
print(search_results)
display(HTML(render_search_results(query, search_results)))
```

Figure 8.5 shows the results of the index-time signals boosting. As you can see, the results now look identical to the query-time signals boosting output shown previously in Figure 4.1.

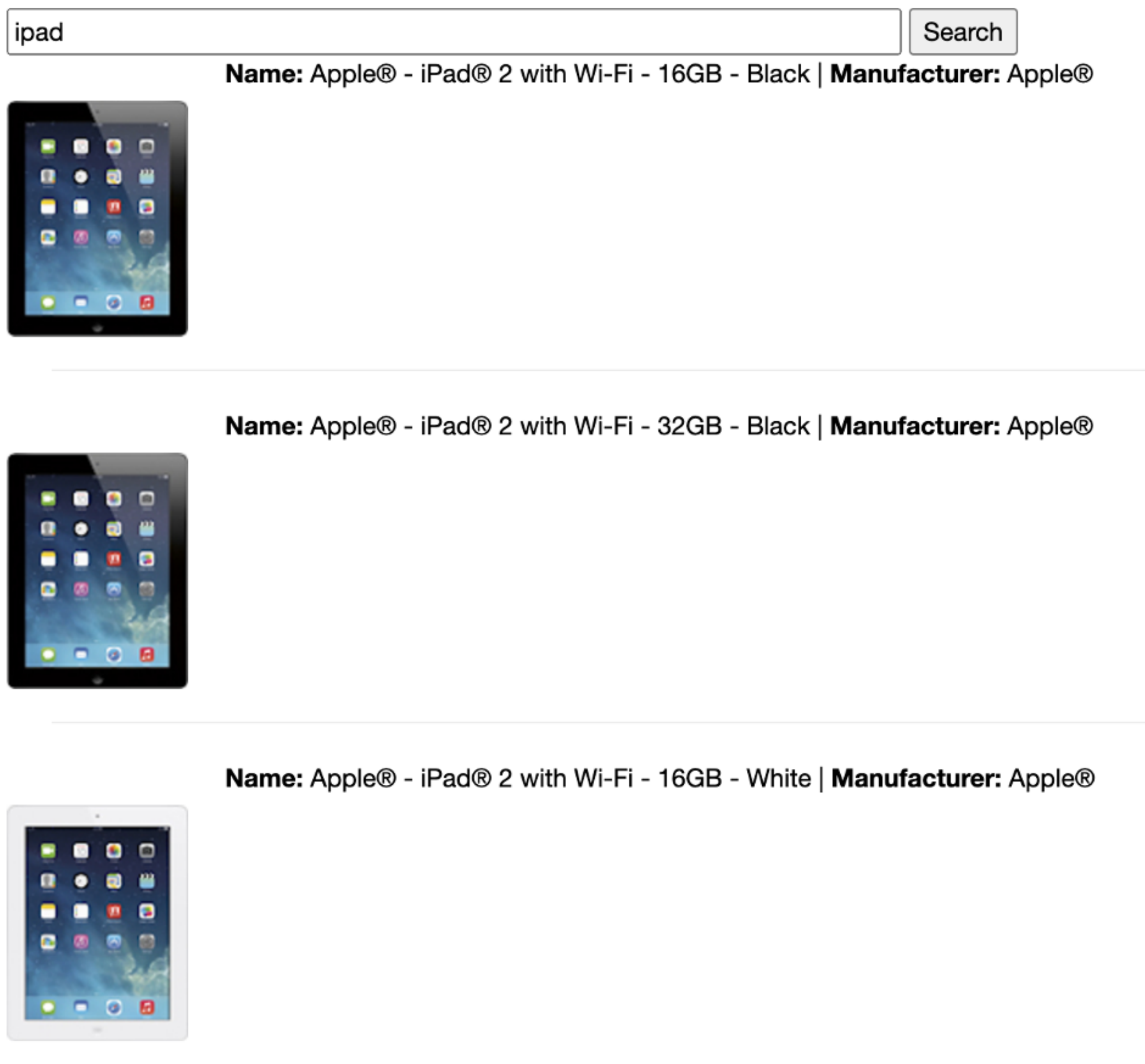


Figure 8.5 Index time signals boosting, demonstrating similar results as query-time index boosting

The relevance scores will not be identical when using the `payload` function to boost index-time boosts versus adding document boosts for query-time signals boosting, but the relative ordering of results should be similar. The index-time signals boosting will apply to all documents with a signals boost as opposed to only the top documents with a signals boost, making index-time boosting more comprehensive, among other benefits.

BENEFITS OF INDEX-TIME BOOSTING

Index-time boosting solves for most of the drawbacks of query-time boosting:

1. The query workflow is simpler and faster because it doesn't require doing two queries - one to look up the signals boosts and another to run a boosted query using those signals boosts.
2. Each query is more efficient and faster per boost as the number of boosts increases, because the boost query is a single keyword search against the boost field as opposed to a boost query for an increasing number of documents which need to be boosted.

3. Results paging is no longer a problem, because ALL documents matching the query are boosted, not just the top-N that can be efficiently loaded and added to the query.

Given these characteristics, index-time boosting can substantially improve relevance and consistency of results ordering by ensuring all queries receive consistent and complete boosting of all their matching documents, and it can substantially improve query speed by making queries more efficient and removing extra lookups prior to execution of the main query to the search engine.

DRAWBACKS OF INDEX-TIME BOOSTING

If index-time boost solves all of the problems of query-time boosting, they why wouldn't we always use index-time signals boosting over query-time signals boosting?

The main drawback of index-time boosting is that since the boost values for a query are indexed onto each document (each document contains the terms for which that document should be boosted), this means that adding or removing a keyword from the signals boosting model requires reindexing all documents associated with that keyword. If signals boosting aggregations are updated incrementally (on a per-keyword basis), then this means potentially reindexing all of the documents within your search engine on a continuous basis. If your signals boosting model is updated in batch for your entire index, then at a minimum this means reindexing potentially all of your documents every time your signals boosting model is regenerated.

This kind of indexing pressure adds operational complexity to your search engine. In order to keep query performance fast and consistent, you will likely want to separate indexing of documents onto separate servers from where the search indexes are hosted for serving queries.

SIDEBAR **Separation of Concerns: Indexing vs. Querying**

In Apache Solr, indexes are broken into one or more "shards", which are partitions containing a subset of the documents in a collection. Each shard can have one or more replicas, which are each exact copies of all the data belonging to their shard. When a search is run, Solr sends the query to one replica of each shard, the query is run in parallel on each of those replicas, and the results are aggregated and returned as a full set of results to the end users. The primary purpose of adding more shards is to allow for more documents to be searched in less time, and the primary purposes of adding more replicas are to add fault tolerance and to enable a larger number of searches to be run against the same number of shards.

Solr has three different types of replicas: NRT (Near-realtime), TLOG (transaction log), and PULL replicas. By default, all replicas are NRT, which means every replica indexes every document update when it comes in. This allows document updates to be available immediately on each replica, but it can also very negatively impact query time on those replicas if lots of documents are being indexed constantly. The other replica types (TLOG and PULL) are able to pull indexes from an NRT replica instead of doing the indexing work in duplicate, which can allow a separation of concerns within the cluster to allow indexing on the NRT replica that is isolated from the querying operations on the TLOG and PULL replicas.

If you plan to do index-time signals boosting and expect to be constantly reindexing signals, you should strongly consider isolating index and query time operations to ensure your query performance isn't negatively impacted by the significant additional indexing overhead from ongoing indexing of signals boosts.

The other drawback of index-time boosting, which is also related to the requirement that all documents affected by a signal be reindexed upon changes, is that making changes to your signals boosting function can require more planning. For example, if you would like to change your weight for click vs. purchase signals from 1:25 to 1:20, then you may want to create a `signals_boosts_2` field with the new weights, reindex all of your documents adding the new boosts, and then flip over your query to use the new field instead of the original `signals_boosts` field. Otherwise, your boost values and ranking scores will fluctuate inconsistently until all of your documents scores have been updated.

If those drawbacks can be worked around, however, then implementing index-time signals boosting can solve all of the drawback of query-time signals boosting, leading to better query performance, full support for results paging, and use of all signals from all documents as opposed to just a sampling from the most popular documents.

8.7 Summary

- Signals boosting is a type of ranking algorithm which aggregates user signals to create relevance boosts for the most popular documents matching each query.
- It is important to normalize user queries to better clean up noise in user signals and build a more robust signals boosting model.
- Crowdsourced data is subject to manipulation, so it is important to explicitly prevent spam and malicious signals from impacting the quality of your relevance models.
- You can combine different signal types into a single signals boosting model by assigning relative weights to each signal type and doing a weighted sum of values across signal types.
- Introducing a time-decay function enables recent signals to carry more weight than older signals, allowing older signals to phase out over time.
- Signal boosting models can be productionized using query-time signals boosting (more flexible) or index-time signals boosting (more scalable and more consistent relevance ranking).

10 Learning to Rank

This chapter covers

- Using machine learning to build generalized relevance ranking
- Ranking within the search engine using machine learning models
- How learning to rank is different from other machine learning methods
- Building a robust and generalizable ranking model

In this chapter, we'll explore *Learning to Rank* (LTR): using machine learning to create a generalized ranking function. We'll start by seeing where LTR compares to solutions in previous chapters. We'll then begin our explorations with simple models using Solr's LTR capabilities, walking through the steps of training and ranking search results with an LTR model. Finally, we'll close with discussion of the different choices and options along the path to performing LTR.

10.1 The Limits of Collaborative Filtering Ranking

We've seen from chapter 4 we can use collaborative filtering to predict which documents are likely to satisfy specific queries, based on similar queries. Consider the two `red shoe` and `scarlett shoes` queries in Table 10.1:

Table 10.1 Comparing the success of different products between the query red shoe and the query scarlett shoes

Product	q=Red Shoe	q=Scarlett Shoes
	CTR=0.9	CTR=0.9
	CTR=0.01	CTR=0.01
	CTR=0.5	(Not Returned)
	CTR=0.01	CTR=0.01

We see that there's a shoe, the 3rd row, that a normal search for `scarlett shoes` doesn't return. Given the high similarity in click-through-rate (CTR) for the shoe in the 1st row for the `red shoe` and `scarlett shoes` queries, and also given the high similarity in CTR between the shoe in the 1st row with the shoe in the 3rd row for the query `red shoe`, we might therefore recommend the red high heels in row 3 for the `scarlet shoes` query.

Search queries, however, don't neatly line up like Table 10.1. As shown in Figure 10.1, there are popular *head queries*, like `red shoes`, with a great deal of signals. Here collaborative filtering works great. Conversely, a *tail query* (containing a small number of signals) like `red rose colored shoes` may have too little data to benefit from such a solution. A *torso query* (containing a medium number of signals), may have enough data in some cases, but often will still not contain enough signals to make definitive decisions. This lack of data to deliver meaningful recommendations is what's known as the *cold start* problem.

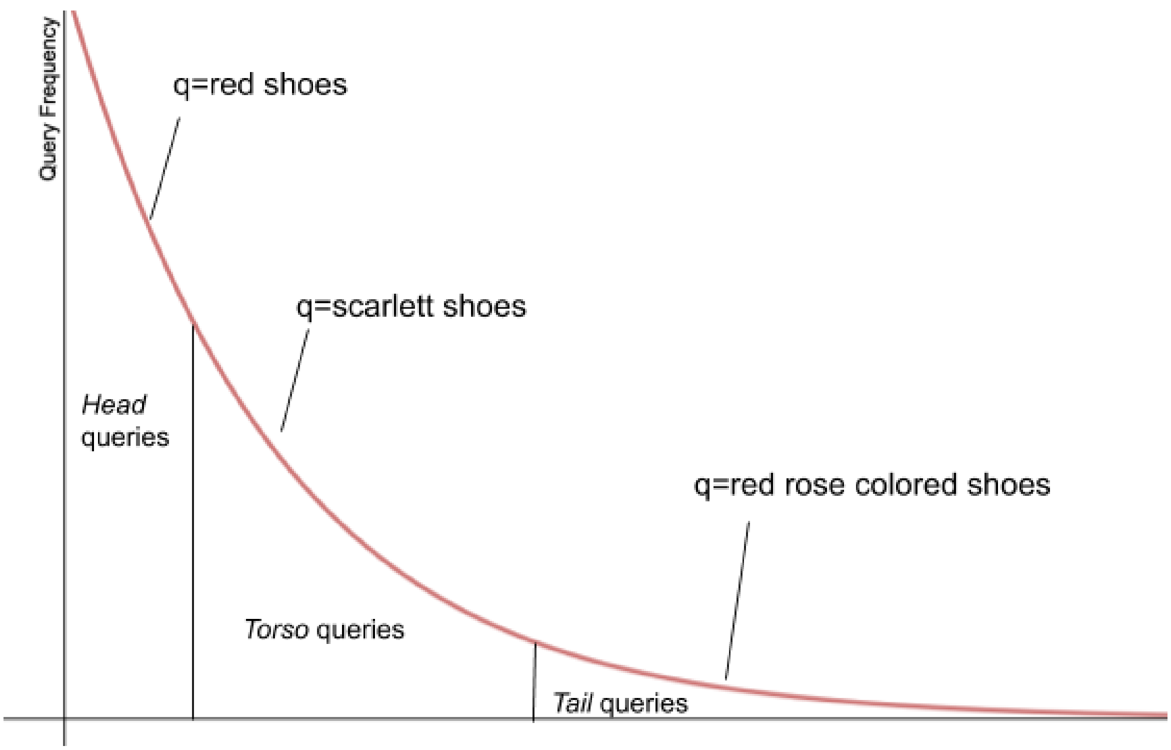


Figure 10.1 Head, Torso, and Tail Queries in a Search System. We can apply collaborative filtering to head queries more successfully, but optimizing relevance well means generalizing beyond to improve the torso and tail as well.

Table 10.2 shows an example query, `red rose colored shoes`, that would have too little data for collaborative filtering:

Table 10.2 We can't easily compare signals between head queries and tail queries

Product	`q=red shoe`	`q=red rose colored shoes`
	CTR=0.9	CTR=??
	CTR=0.01	CTR=??
	CTR=0.5	CTR=??
	CTR=0.01	CTR=??

Solving past the head matters! Many search queries are one-offs we have never seen before. Moreover, users running long tail queries often have more specific intent, such as to take a follow on action like making a purchase.

In short, for whatever techniques we apply on our head queries, we should also seek more generalizable solutions that can also optimize further down the tail.

10.2 Learning to Rank: Generalized Optimization of Relevance

How can we overcome the cold start problem with relevance ranking, in a statistically robust fashion, to cover all the likely queries we'll encounter (head or tail)?

Recall manual relevance tuning from Chapter 3. We observe factors that correspond with relevant results and we combine those factors mathematically into a *ranking function*. The ranking function returns a relevance score that orders results as closely as possible to our ideal ranking.

For example, consider a movie search engine, with documents like the one in Listing 10.1:

Listing 10.1 Movie with common fields out of TheMovieDB corpus. Here we show fields for The Social Network.

```
{'title': ['The Social Network'],
 'overview': ['On a fall night in 2003, Harvard undergrad and computer programming genius Mark
             Zuckerberg sits down at his computer and heatedly begins working on a new idea. In a
             fury of blogging and programming, what begins in his dorm room as a small site among
             friends soon becomes a global social network and a revolution in communication. A
             mere six years and 500 million friends later, Mark Zuckerberg is the youngest
             billionaire in history... but for this entrepreneur, success leads to both personal
             and legal complications.'],
 'tagline': ["You don't get to 500 million friends without making a few enemies."],
 'release_year': 2010}
```

These movies come from TheMovieDB(tmdb) corpus (themoviedb.org), which we'll use in this chapter. If you wish to follow along with the code for this chapter please use this chapter's first notebook to setup TheMovieDB corpus.

Through endless iterations and tweaks, we might arrive at a generalizable movie ranking function that looks something like Listing 10.2.

Listing 10.2 Solr query against movie index using The Movie DB data. The `title` field will be searched and scored with BM25, with a weight of 10. Same with overview, but with a weight of 20. Finally `release_year` is incorporated as an additive boost, biasing towards recent movies.

```
q=title:(${keywords})^10 overview:(${keywords})^20 {!func}release_year^0.01
```

Manually optimizing this function takes time and effort. Such an optimization is perfect for machine learning.

Learning to Rank (LTR) takes our proposed relevance factors, and learns an optimal ranking function. Learning to Rank takes several forms: from a simple set of linear weights (like the boosts here) to a complex deep learning model.

To learn the ropes, we'll build a simple LTR model in this chapter. We'll find the optimal weights for `title`, `overview`, and `release_year` in a scoring function like the one above. With this relatively simple task, however, we'll see the full lifecycle of developing a Learning to Rank solution prior to tackling more advanced techniques in the next chapter.

10.2.1 Practical Learning to Rank

We focus on building Learning to Rank for production search systems, which can be quite different from a research or Kaggle competition context. We not only need relevant results, but results returned suitably fast, with mainstream, well-understood infrastructure and techniques.

For this reason, we will focus on building Learning to Rank with Solr's Learning to Rank plugin. This plugin will rerank within the search engine, improving performance, and scaling learning to rank to true 'big data' problems. These lessons go beyond Solr, though - Elasticsearch also has a community provided plugin, with nearly identical concepts, and other search systems will tend to follow the same general steps outlined in this chapter.

Figure 10.2 outlines the workflow for developing a practical Learning to Rank solution.

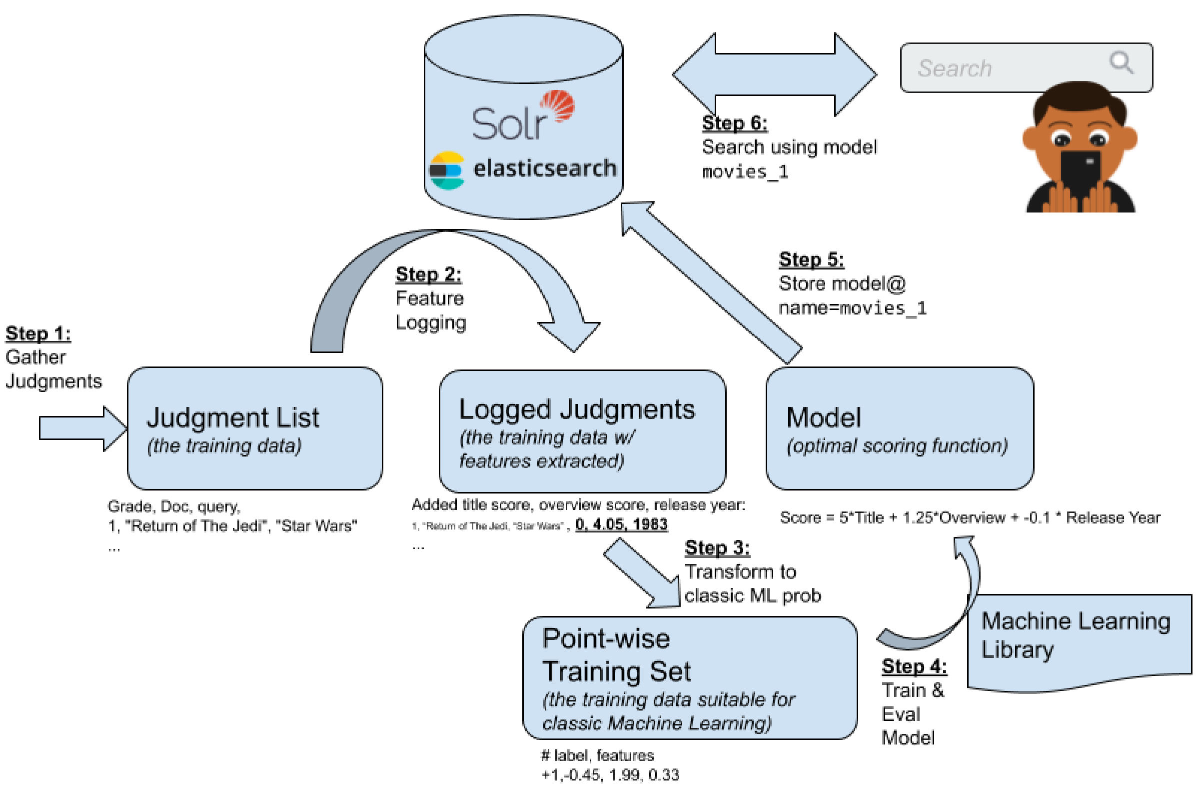


Figure 10.2 The workflow for building a Learning to Rank system.

In Figure 10.2, you may notice similarities between LTR and classic Machine Learning systems. But the exceptions are what make it interesting. Let's summarize Figure 10.2 at a high level, before we dive into each step:

1. *Training data* in Learning to Rank corresponds to a list of judgments or a *judgment list*. A *judgment* simply labels a document as relevant or irrelevant for a query. In Figure 10.2, "Return of the Jedi" is labeled relevant (grade of `1`), for the query `star wars` in step 1.
2. A *feature* in Learning to Rank corresponds to a value the search engine can compute easily and quickly to input into our ranking function. This means we make our features Solr queries like the `title:(${keywords})` in Listing 10.2. We extract or *log* these features from the search engine into our judgment list in step 2.
3. The training data goes through a number of preparations to turn the task of ranking - placing relevant documents above irrelevant ones - into a more normal-looking machine learning problem (step 3).
4. Model execution happens within the search engine. This makes it fast, but also places some real-world performance constraints on the complexity of features and how they can be evaluated efficiently in real-time search use cases.

The rest of the chapter will walk through each of these steps in detail to build our first LTR implementation. Let's get cracking!

10.3 Step 1: A Judgment List, Starting with Ground Truth

We start our Learning to Rank journey with a *judgment list*, which is a list of relevance labels or *grades*, indicating relevance of a document to a query. Grades can come in a variety of forms. For now we'll stick to simple *binary judgments* - a 0 to indicate an irrelevant document, and a 1 for a relevant one.

Using a `Judgment` class provided with this book's chapter, we label "The Social Network" as relevant for the query `social network` by creating a `Judgment` in Listing 10.3.

Listing 10.3 Label movie "The Social Network" (doc_id=37799) as relevant (grade=1) for keywords 'social network'

```
from ltr.judgments import Judgment
Judgment(grade=1, keywords='social network', doc_id=37799)
```

It's more interesting to look over multiple queries. In Listing 10.4, we have `social network` and `star wars` as two different queries, with movies graded as relevant or irrelevant.

Listing 10.4 Labeling movies relevant (grade=1) or irrelevant (grade=0) for queries `social network` and `star wars`

```
mini_judg_list=[
    # for 'social network' query
    Judgment(grade=1, keywords='social network', doc_id='37799'), #The Social Network
    Judgment(grade=0, keywords='social network', doc_id='267752'), # chicagoGirl
    Judgment(grade=0, keywords='social network', doc_id='38408'), # Life As We Know It
    Judgment(grade=0, keywords='social network', doc_id='28303'), # The Cheyenne Social Club

    # for 'star wars' query
    Judgment(grade=1, keywords='star wars', doc_id='11'), # Star Wars, A New Hope
    Judgment(grade=1, keywords='star wars', doc_id='1892'), # Return of the Jedi
    Judgment(grade=0, keywords='star wars', doc_id='54138'),# Star Trek Into Darkness
    Judgment(grade=0, keywords='star wars', doc_id='85783'), # The Star
    Judgment(grade=0, keywords='star wars', doc_id='325553'), # Battlestar Galactica
]
```

You can see that in Listing 10.4, we labeled "Star Trek" and "Battlestar Galactica" as irrelevant for `star wars`, but "Return Of The Jedi" as relevant.

You're hopefully asking yourself - where did these grades come from!? Hand labeled by movie experts? Based on user clicks? Good questions! Creating a good training set, based on user interactions with search results, is crucial for getting LTR to work well. It's such a critical topic that we'll punt on most of step one until chapter 11. For now, we will continue exploring LTR nuts and bolts with a preexisting judgment list we've provided.

Each judgment will also have a `features` vector, which can be used to train a model. The first feature in the `features` vector could be made to correspond to the `title` BM25 score, the second to the `overview` BM25 score, and so on. We haven't populated the `features` vectors yet, however, so they'll currently be empty if you try to inspect them:

```
In: mini_judg_list[0].features
Out: []
```

Let's use the search engine to gather some features!

10.4 Step 2 - Feature Logging

For the purposes of LTR, a *feature* is some numerical attribute of the document, the query, or the query-document relationship. Features are the mathematical lego building blocks we use to build a ranking function. You've already seen a manual ranking function with features from Listing 10.2: the keyword's score in the `title` field is one such feature. As is the `release_year` and `overview` keyword scores.

```
q=title:(${keywords})^10 overview:(${keywords})^20 {!func}release_year^0.01
```

Of course, the features you end up using could be more complex or domain-specific, such as commute distance in job search, or some knowledge graph relationship between query and document. Anything you can compute relatively quickly when a user searches might be a reasonable feature.

Feature logging takes a judgment list and computes features for each labeled query-document pair. If we computed the values for each feature from Listing 10.2 for each relevant/irrelevant document for the query `social network` we would arrive at something like Table 10.3.

Table 10.3 Features logged for the query social network for relevant (grade=1) / irrelevant (grade=0) documents

Grade	Movie	`title:(${keywords})`	`overview:(${keywords})`	`{!func}release_year`
1	Social Network	8.243603	3.8143613	2010.0
0	chicagoGirl	0.0	6.0172443	2013.0
0	Life As We Know It	0.0	4.353118	2010.0
0	The Cheyene Social Club	3.4286604	3.1086721	1970.0

A machine learning algorithm might examine feature values from Table 10.3 and converge on a good ranking function. From just the data in Table 10.3, it seems such an algorithm might produce a ranking function with a higher weight for the `title` feature and lower weights for the other features.

Before we get to the algorithms, however, we need to examine the feature logging workflow in a production search system.

10.4.1 Storing Features

Solr LTR tracks the features logged in a *feature store* - a list of named features. It's crucial that we log features for training in a manner consistent with how the model will be later executed for relevance ranking. Much of the LTR plugin's job is to help store and manage features and keep things consistent.

As shown in Listing 10.5, creating a feature store in Solr is a simple HTTP PUT.

Listing 10.5 Creating three features for our Learning to Rank, a title field relevance score (`title_bm25`), an overview field relevance score (`overview_bm25`), and the value of the `release_year` field.

```
feature_set = [
    {
      "name" : "title_bm25",
      "store": "movies",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "title:(${keywords})"
      }
    },
    {
      "name" : "overview_bm25",
      "store": "movies",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "overview:(${keywords})"
      }
    },
    {
      "name" : "release_year",
      "store": "movies",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "{!func}release_year"
}}]

requests.put('http://aips-solr:8983/solr/tmdb/schema/feature-store',
             json=feature_set)
```

The first two features are parameterized: they each take the search keywords (i.e. `social network`, `star wars`) and execute a search on the corresponding field, returning the BM25 score (covered in chapter 3) as the feature value. The final feature simply retrieves the release year of the movie (purely a document feature). Note the syntax used in `params` is simply a Solr query, letting you leverage the full power of Solr's extensive Query DSL to craft features.

10.4.2 Logging Features from our Corpus

For each unique query in our judgment list, we need to extract the features for the query's graded documents. For `social network` in the mini judgment list above, we have one relevant document (`37799`) and three irrelevant documents (`267752`, `38408`, and `28303`).

An example of feature logging for the query `social network` is shown in Listing 10.6.

Listing 10.6 Logging the values from our feature store for the docs present in our judgment list for the query `social network`.

```
logging_solr_query = {
    "fl": "id,title,[features store=movies efi.keywords=\"social network\"]",
    'q': "id:37799 OR id:267752 id:38408 OR id:28303", #social network graded documents from judgement l
    'rows': 10,
    'wt': 'json'
}

resp = requests.post('http://aips-solr:8983/solr/tmdb/select',
                     data=logging_solr_query)
resp.json()
```

Results

```
'docs': [{'id': '38408',
  'title': 'Life As We Know It',
  '[features]': 'title_bm25=0.0,overview_bm25=4.353118,release_year=2010.0'},
 {'id': '37799',
  'title': 'The Social Network',
  '[features]': 'title_bm25=8.243603,overview_bm25=3.8143613,release_year=2010.0'},
 {'id': '28303',
  'title': 'The Cheyenne Social Club',
  '[features]': 'title_bm25=3.4286604,overview_bm25=3.1086721,release_year=1970.0'},
 {'id': '267752',
  'title': '#chicagoGirl',
  '[features]': 'title_bm25=0.0,overview_bm25=6.0172443,release_year=2013.0'}]}}
```

The key part of this query needed for Solr's LTR calculations is the syntax in square brackets passed to `fl` (field list). This adds a computed field to each document response, containing the feature vector, for the specified feature store (here `movies`). The `efi` parameter stands for *external feature information* and is used to pass query keywords (here `social network`) and any additional query-time information needed to compute each feature. Notice in the response, we retrieve each of the 4 documents we requested, along with every feature computed for that document.

With some mundane Python string munging and book-keeping (code omitted). We can fill-in the features for query `social network` in our training set from this response. We end up with Listing 10.7, a partially filled out training set. Features are filled in for the query `social network`; features are still needed for the query `star wars`

Listing 10.7 The judgment list, with features appended for just the query `social network`

```
[Judgment(grade=1,qid=1,keywords=social network,doc_id=37799,features=[8.243603, 3.8143613,
                                                                       2010.0],weight=1,
 Judgment(grade=0,qid=1,keywords=social network,doc_id=267752,features=[0.0, 6.0172443,
                                                                        2013.0],weight=1,
 Judgment(grade=0,qid=1,keywords=social network,doc_id=38408,features=[0.0, 4.353118,
                                                                        2010.0],weight=1,
 Judgment(grade=0,qid=1,keywords=social network,doc_id=28303,features=[3.4286604, 3.1086721,
                                                                        1970.0],weight=1,
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=11,features=[],weight=1,
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=1892,features=[],weight=1,
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=54138,features=[],weight=1,
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=85783,features=[],weight=1,
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=325553,features=[],weight=1]
```

In Listing 10.7, as we might expect, the first feature value corresponds to the first feature in our feature store (`title_bm25`); the second value to the second feature in our feature store (`overview_bm25`), and so on. We would then repeat Listing 10.6 for the query `star wars` the corresponding graded documents, resulting in a fully logged training set as shown in Listing 10.8 .

Listing 10.8 The judgment list with features appended for both the queries `star wars` and `social network`.

```
[Judgment(grade=1,qid=1,keywords=social network,doc_id=37799,features=[8.243603, 3.8143613,
                                                                       2010.0],weight=1,
 Judgment(grade=0,qid=1,keywords=social network,doc_id=267752,features=[0.0, 6.0172443,
                                                                        2013.0],weight=1,
 Judgment(grade=0,qid=1,keywords=social network,doc_id=38408,features=[0.0, 4.353118,
                                                                        2010.0],weight=1,
 Judgment(grade=0,qid=1,keywords=social network,doc_id=28303,features=[3.4286604, 3.1086721,
                                                                        1970.0],weight=1,
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=11,features=[6.7963624, 0.0, 1977.0],weight=1,
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=1892,features=[0.0, 1.9681965, 1983.0],weight=1,
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=54138,features=[2.444128, 0.0, 2013.0],weight=1,
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=85783,features=[3.1871135, 0.0, 1952.0],weight=1,
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=325553,features=[0.0, 0.0, 2003.0],weight=1]
```

Just two queries with a few judgments per query isn't that interesting, however. In this chapter, we'll use a judgment list with about a hundred movie queries, each with about 40 movies graded as relevant / irrelevant. Code for loading and logging features for this larger training set essentially repeats the Solr request shown in Listing 10.6. We won't repeat the lengthy code here (you can find it in the accompanying notebook for this chapter). The end result of this feature logging looks just like Listing 10.8, just created from a much larger judgment list.

We'll next move on to consider how to handle the problem of ranking as a machine learning problem.

10.5 Step 3 - Transforming LTR to a Traditional Machine Learning Problem

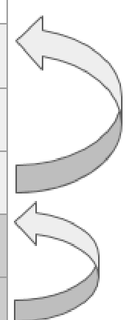
In this section we're going to explore ranking as a machine learning problem. This will help us understand how to apply existing, classic machine learning knowledge to our Learning to Rank task.

The task of Learning to Rank is to look over many relevant / irrelevant training examples for a query, and to then build a model to bring relevant ones to the top, and push irrelevant ones to the bottom. Each training example doesn't have much value by itself, but only in relation to how its ordered alongside its peers in a query. Figure 10.3 shows this task, with two queries. The goal is to find a scoring function that can use the features to correctly order results.

Ranking is NOT direct prediction

(Ranking optimizes order of a query grouping of examples)

qid	Grade	title_bm25	overview_bm25
1	0	0.15	0.00
1	0	0.00	0.87
1	1	11.15	9.04
2	0	0.98	3.5
2	1	8.75	5.67
2	0	0.95	4.34

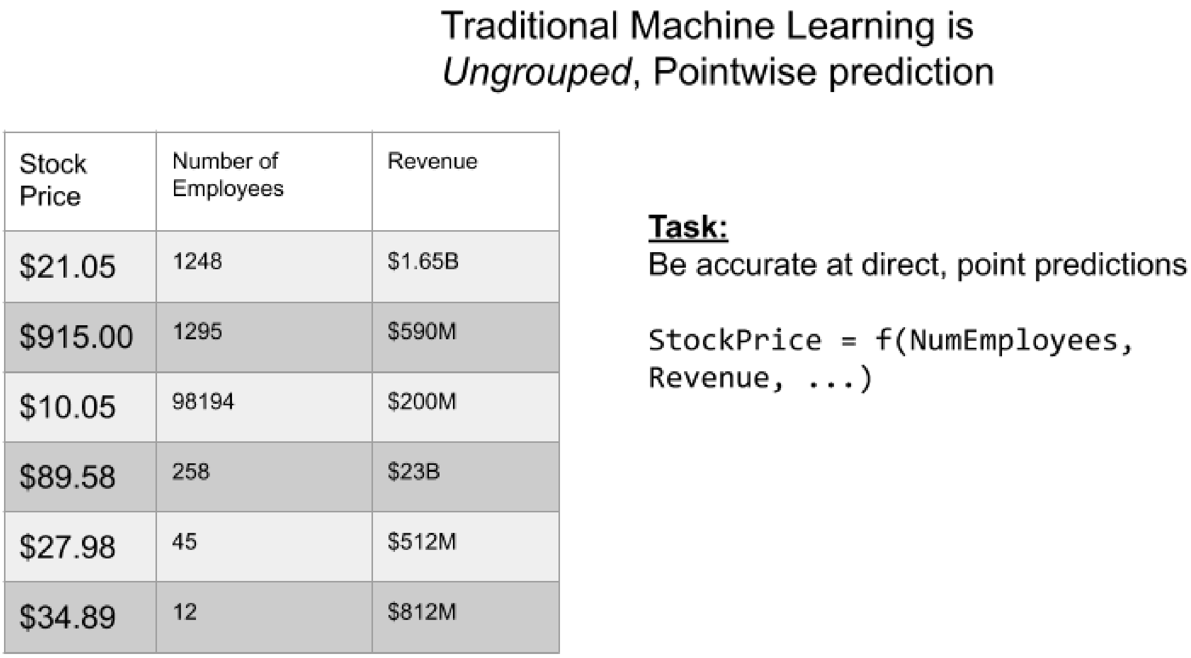


Task:
Sort relevant above irrelevant

`Score = S(title_bm25, overview_bm25, ...)`

Figure 10.3 Learning to Rank is about making each query the ideal order, not predicting individual points

Contrast LTR with a more classic *point-wise machine learning* task. A task like predicting a company's stock price given revenue or number of employees. Here, we can evaluate the model's accuracy on each training example in isolation. We know, just by looking at one company, how well we predicted that company's stock price. Compare Figure 10.4 showing a point-wise task to Figure 10.3. Notice in Figure 10.4 the learned function attempts to predict the stock price directly. Whereas with LTR, the function's output is only meaningful for ordering items relative to their peers for a query.



Stock Price	Number of Employees	Revenue
$21.05	1248	$1.65B
$915.00	1295	$590M
$10.05	98194	$200M
$89.58	258	$23B
$27.98	45	$512M
$34.89	12	$812M

Figure 10.4 Point-wise Machine Learning tries to optimize predictions of individual points (such as a stock price or the temperature)

LTR appears to be a very different animal than point-wise machine learning. But most LTR methods use clever alchemy to transmogrify ranking into a point-wise machine learning task like stock price prediction.

We'll take a look at this magic trick in the next section by exploring a popular LTR model named SVMRank.

10.5.1 SVMRank: Transforming Ranking to Binary Classification

SVMRank transforms relevance into a binary classification problem. *Binary Classification* simply means classifying between two classes (like 'relevant' and 'irrelevant'; or 'adult' or 'child') using the available features.

An *SVM* or *Support Vector Machine* is one method of performing binary classification. It's best to review a general machine learning book if you're interested in a deeper, more complete introduction to SVMs. Intuitively an SVM finds the best, most generalizable line (or really hyperplane) to draw between the two classes. If trying to build a model to predict whether an animal is a dog or cat, we might look the heights and weights of known dogs or cats and draw a line separating the two classes as shown in Figure 10.5.

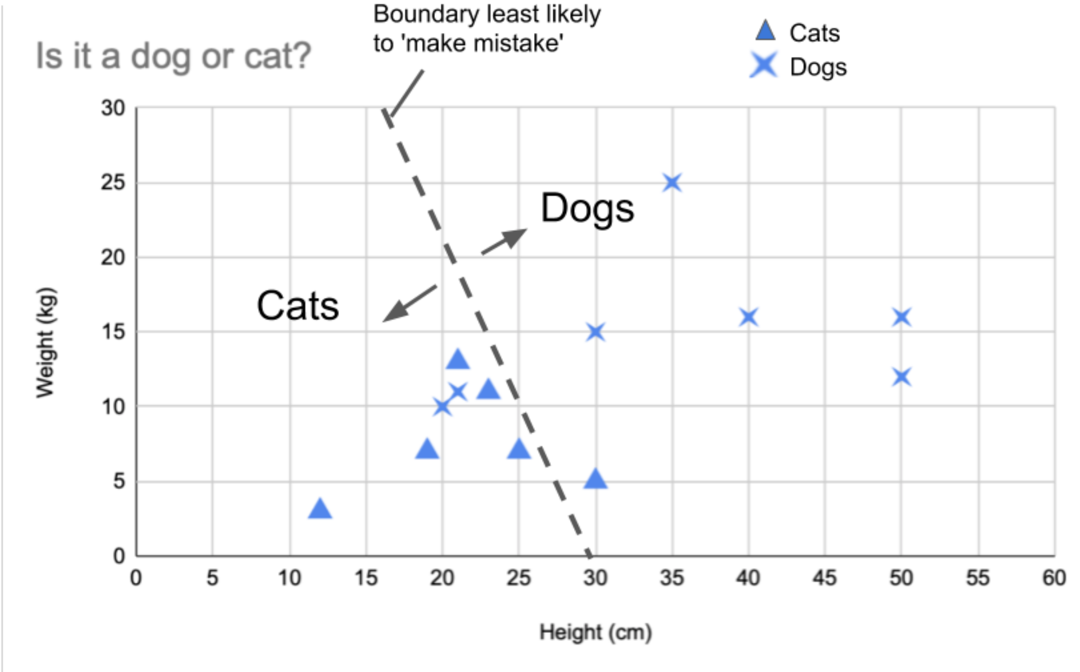


Figure 10.5 SVM Example: Is an animal a dog or a cat?

A good line drawn between the classes, or *separating hyperplane*, attempts to minimize the mistakes it makes in the training data (fewer dogs on the cat side and vice versa). We also want a hyperplane that is *generalizable*, meaning that it will probably do a good job of classifying animals that weren't seen during training. After all, what good is a model if it can't make predictions about new data? It's not very AI powered!

Another detail to know about SVMs is they can be sensitive to the range of our features. For example, imagine if our height feature was millimeters instead of centimeters like in Figure 10.6. It forces the data to stretch out on the x-axis, and the separating hyperplane looks quite different!

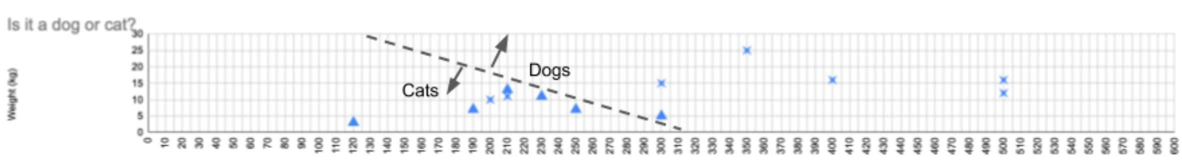


Figure 10.6 Separating hyperplane impacted by range of one of the features.

SVMs work best when our data is *normalized*. Normalization means we've scaled every feature to a consistent range. We'll normalize our data by equating 0 to the mean. So if the average `release_year` is 1990, movies released in 1990 will get a 0 in the normalized `release_year` feature. We'll also map +1/-1 to one standard deviation above or below the mean. So if the standard deviation of movie release years is `22` years, then movies in 2012 turn into a `1.0`; movies in 1968 turn into a `-1.0`. This helps makes features a bit more comparable when finding a separating hyperplane.

With that brief background out of the way, could an SVM help us somehow separate relevant from irrelevant documents? Yes! That's exactly what SVMRank hopes to do. Let's walk through how SVMRank works.

10.5.2 Transforming our LTR Training Data to Binary Classification

We'll next explore how to transform ranking into a problem suitable for an SVM.

Before we get started, let's inspect the fully logged training set logged at the end of step 2 for our two favorite queries, `star wars` and `social network`. In this section, we'll focus on just two features (`title_bm25` and `overview_bm25`) to help us explore feature relationships graphically. Figure 10.7 shows these two features for every graded `star wars` and `social network` document, labeling some prominent movies from the training set.

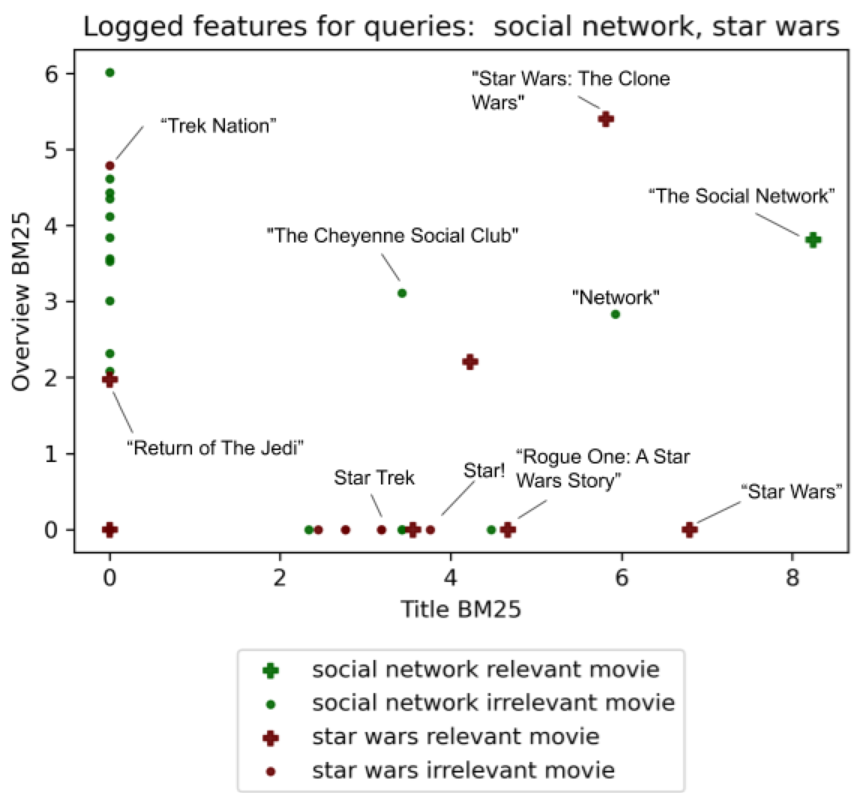


Figure 10.7 Logged feature scores for social network and star wars queries

FIRST, NORMALIZE THE LTR FEATURES

We're first going normalize each feature. We include a function `normalize_features` in the code for this chapter. Listing 10.9 takes the logged output from Step 2 and normalizes features into `normed_judgments`.

Listing 10.9 Normalize our logged LTR training set to a normalized one

```
means, std_devs, normed_judgments = normalize_features(logged_judgments)
logged_judgments[360], normed_judgments[360]
```

Results

```
(Judgment(grade=1,qid=11,keywords=social network,doc_id=37799,features=[8.243603, 3.8143613,
                                                                       2010.0],weight=1,
 Judgment(grade=1,qid=11,keywords=social network,doc_id=37799,features=[4.482941696779275,
                                                                        2.100049660821875,
                                                                       0.8347155270825962],
                                                                       weight=1)
```

You can see the output from Listing 10.9 shows first the logged BM25 scores for title and overview (8.243603, 3.8143613), alongside the release year (2010). These features are normalized, where 8.243603 `title_bm25` corresponds to 4.4829 standard deviations above the mean `title_bm25`, and so on for each feature.

We plot the normalized features in Figure 10.8. This looks very similar to Figure 10.7, with only the scale on the axis differing.

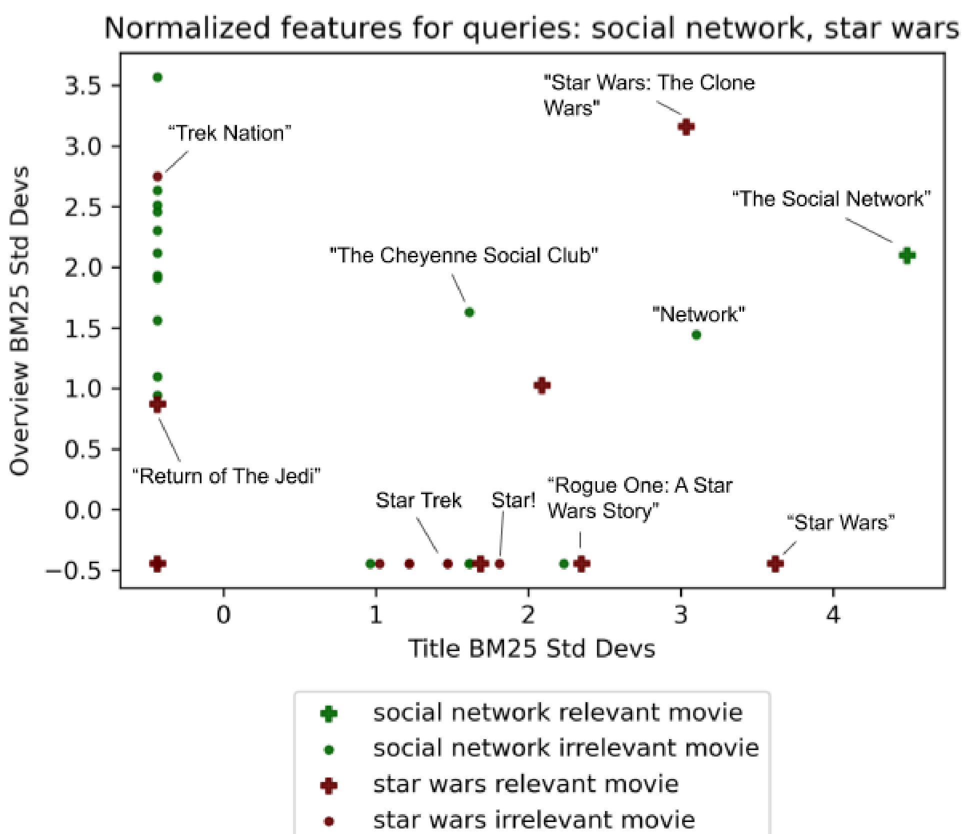


Figure 10.8 Normalized star wars and social network graded movies. Each increment in the graph is a standard deviation above or below the mean.

Next we'll turn ranking into a binary classification learning problem.

SECOND, COMPUTE THE PAIR-WISE DIFFERENCES

With normalized data, our SVM should not be biased by features that just so happen to have very large ranges (such as `release_year`). We're ready to transform the task into binary classification.

SVM Rank uses a *pair-wise* transformation to reformulate LTR to a binary classification problem. *Pair-wise* simply means turning ranking into the task of minimizing out of order pairs for a query.

In the rest of this section, we'll carefully walk through SVMRank's pair-wise algorithm, outlined in the psuedocode in Listing 10.10. Before we do that, let's discuss the algorithm at a high level. This algorithm takes every query's judgment, and compares it to every other judgment for that same query. It computes the feature differences (`feature_deltas`) between every relevant and irrelevant pair for that query. When adding to `feature_deltas`, if the first judgment is more relevant than the second, it's labeled with a `+1` in `predictor_deltas` - and vice versa. In the end, the algorithm builds a brand new training set from the old one: the `feature_deltas` and `predictor_deltas`, which is more suitable to binary classification.

Listing 10.10 Psuedoocde for SVMRank training data transformation. (Note this algorithm is implemented in the training sample code as `pairwise_transform`)

```
foreach query in queries:
   foreach judged_document_1 in query.judgments:
      foreach judged_document_2 in query.judgments:
         if judged_document_1.grade > judged_document_2.grade:
            predictor_deltas.append(+1) # +1 for 1 more relevant than 2
            feature_deltas.append( judged_document_1.features - judged_document_2.features)
         else if  judged_document_1.grade < judged_document_2.grade:
            predictor_deltas.append(-1) # -1 for 1 less relevant than 2
            feature_deltas.append( judged_document_1.features - judged_document_2.features)
```

Figure 10.9 plots the resulting `feature_deltas` for the `social network` and `star wars` data from Figure 10.8, with some prominent pair-wise differences highlighted.

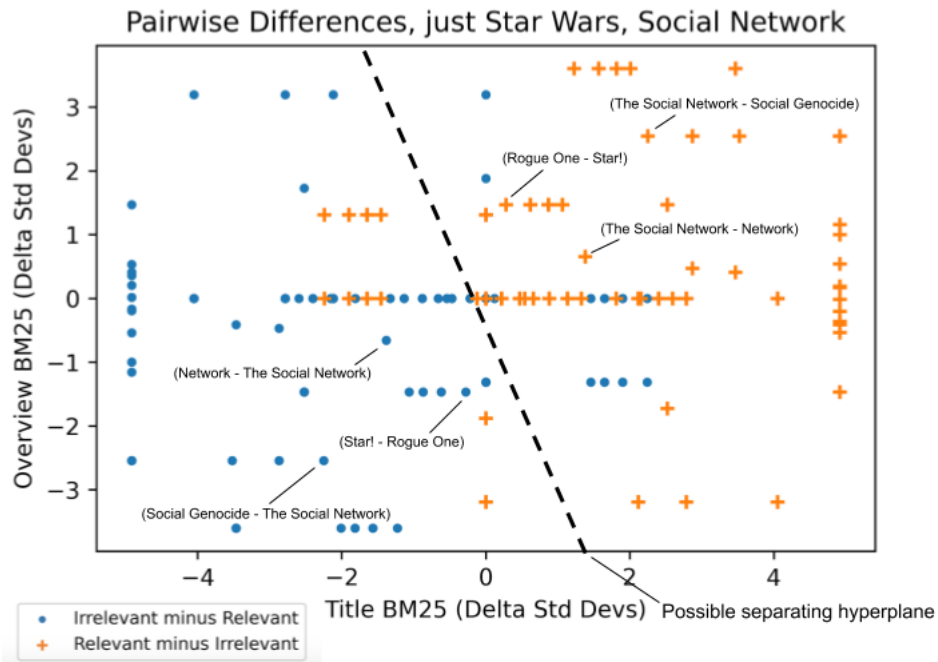


Figure 10.9 Pair-wise differences after SVMRank's transformation for social network and star wars documents along with a candidate separating hyperplane.

You'll notice the relevant minus irrelevant pair-wise deltas (+) tend to be towards the upper right. Meaning relevant documents have a higher `title_bm25` and `overview_bm25` when compared to irrelevant ones.

If we walk through a few examples step-by-step, we might see how this algorithm constructs the data points in Figure 10.9. First let's compare two documents ("Network" and "The Social Network") within the query `social network` as shown in In Figure 10.10.

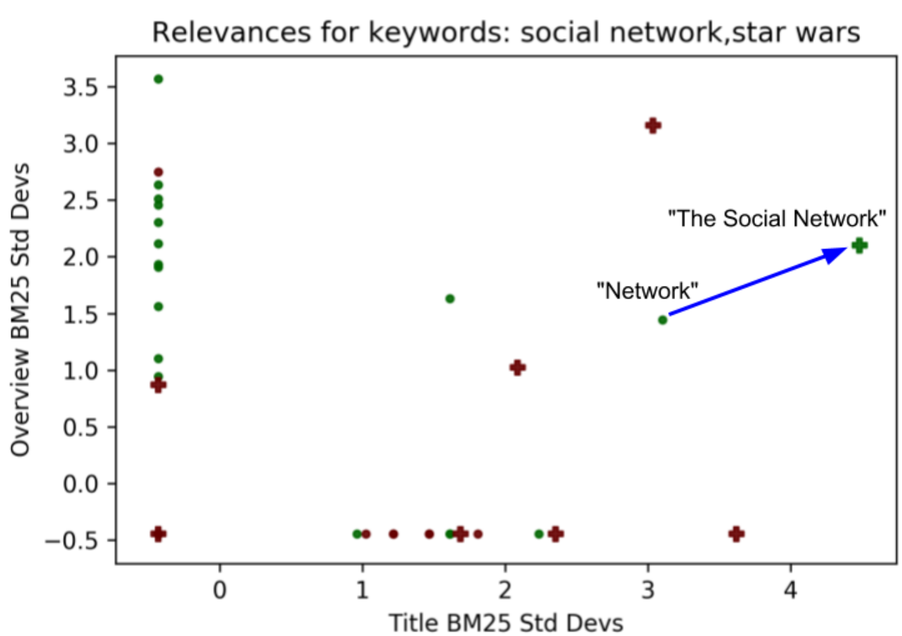


Figure 10.10 Comparing "Network" to "The Social Network" for query social network

The features for "The Social Network" are:

```
[4.483, 2.100]  # title_bm25 is 4.483 stddevs above mean, overview_bm25 is 2.100 stddevs above mean
```

The features for "Network" are:

```
[3.101, 1.443]  # title_bm25 is 3.101 stddevs above mean, overview_bm25 is 1.443 stddevs above mean
```

We then insert the delta between "The Social Network" and "Network" in Listing 10.11.

Listing 10.11 Labeling "The Social Network" vs "Network" delta into feature_deltas

```
predictor_deltas.append(+1)
feature_deltas.append( [4.483, 2.100] - [3.101, 1.443]) # appends [1.382, 0.657]
```

Restating Listing 10.11 in English, we might say, here is one example of a document that’s more relevant than another. In this case, "The Social Network"'s `title_bm25` is 1.382 standard deviations higher than "Network"'s; `overview_bm25` is 0.657 standard deviations higher. Indeed, note the + for "The Social Network - Network" in Figure 10.9 showing the point [1.382, 0.657] amongst the deltas.

The algorithm would also note "The Network" is less relevant than "The Social Network", as shown in Figure 10.11.

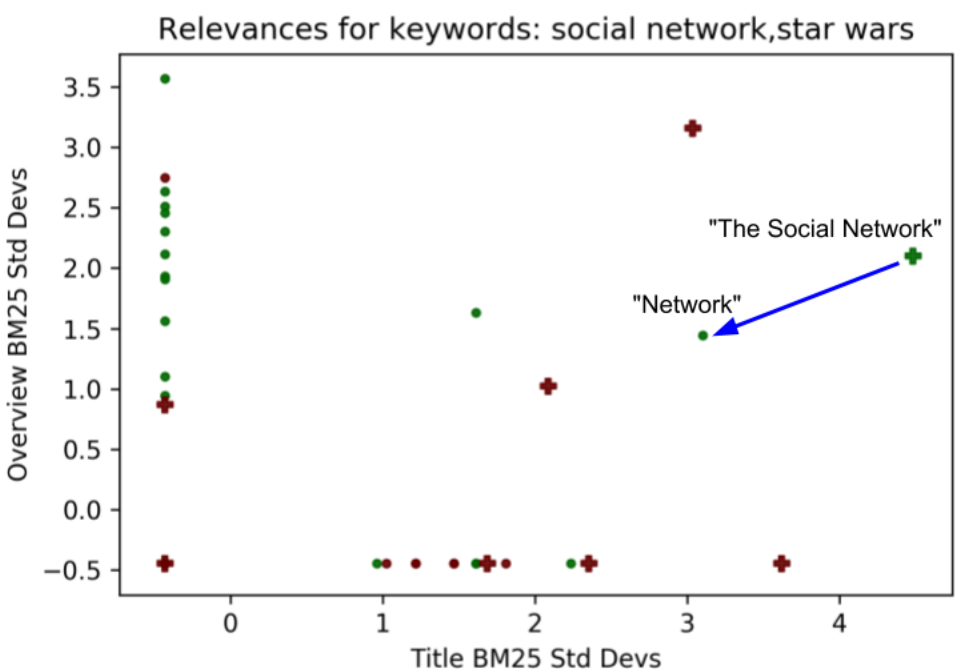


Figure 10.11 Comparing "Network" to "The Social Network" for the query social network

Just as in Listing 10.11, we capture in code this difference in relevance between these two documents. This time, however, in the opposite direction (irrelevant minus relevant).

```
predictor_deltas.append(-1)
feature_deltas.append([3.101, 1.443] - [4.483, 2.100] ) # [-1.382, -0.657]
```

In Figure 10.12, we move onto another relevant-irrelevant comparison of two documents for the query `social network`, appending another comparison to the new training set.

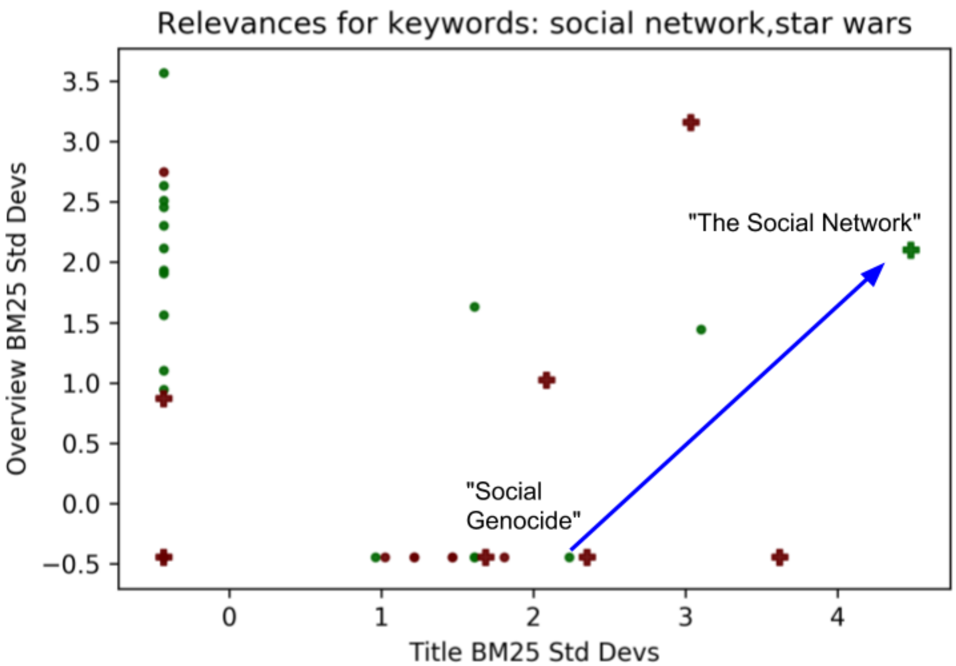


Figure 10.12 Comparing 'Social Genocide' to 'The Social Network' for the query social network

We show appending both directions of the comparison from Figure 10.12 in Listing 10.12, adding both a positive value (when the more relevant document is listed first) and a negative value (when the less relevant document is listed first).

Listing 10.12 Adding "Social Genocide" and "The Social Network" to point-wise training data

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append( [4.483, 2.100] - [2.234, -0.444]) # [2.249, 2.544]
# Negative example
predictor_deltas.append(-1)
feature_deltas.append([2.234, -0.444] - [4.483, 2.100]  ) # [-2.249, -2.544]
```

Once we iterate every pair-wise difference between documents matching the query `social network` to create a point-wise training set, we move on to also logging differences for other queries. Figure 10.13 shows differences for a second query, this time comparing the relevance of documents matching the query `star wars`.

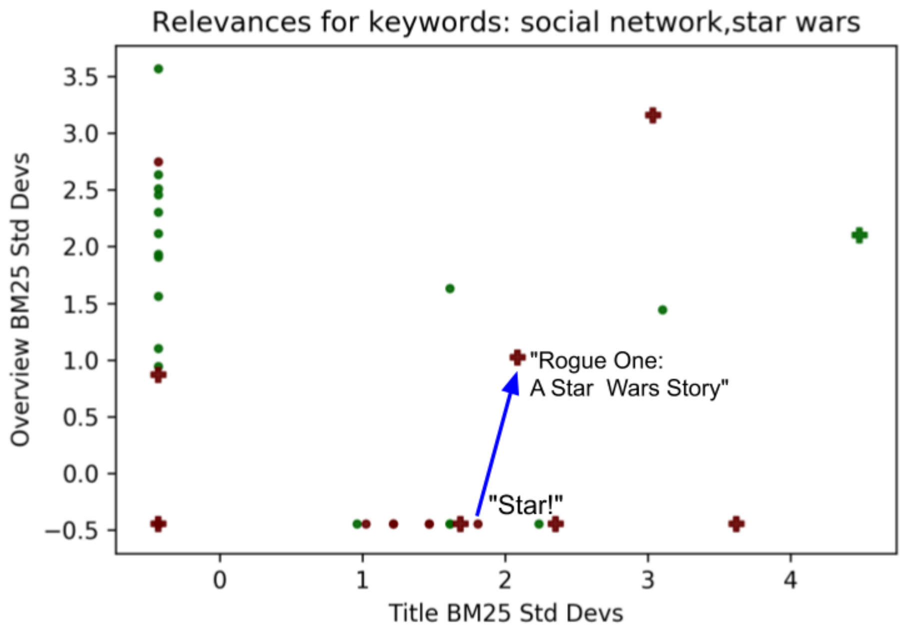


Figure 10.13 Comparing 'Rogue One: A Star Wars Movie' to 'Star!' for the query 'star wars'.

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append( [2.088, 1.024] - [1.808, -0.444]) # Rogue One - "Star!"
# Negative example
predictor_deltas.append(-1)
feature_deltas.append([1.808, -0.444] - [2.088, 1.024]) # "Star!" - "Rogue One"
```

We continue this process of calculating differences between feature values for relevant vs. irrelevant documents, until we have calculated all the pairwise differences together for our training and test queries.

You could see back in Figure 10.9 that the positive examples show a positive `title_bm25` delta, and possibly a slightly positive `overview_bm25` delta. This becomes even more clear if if we calculate deltas over the full dataset of 100 queries, as shown in Figure 10.14.

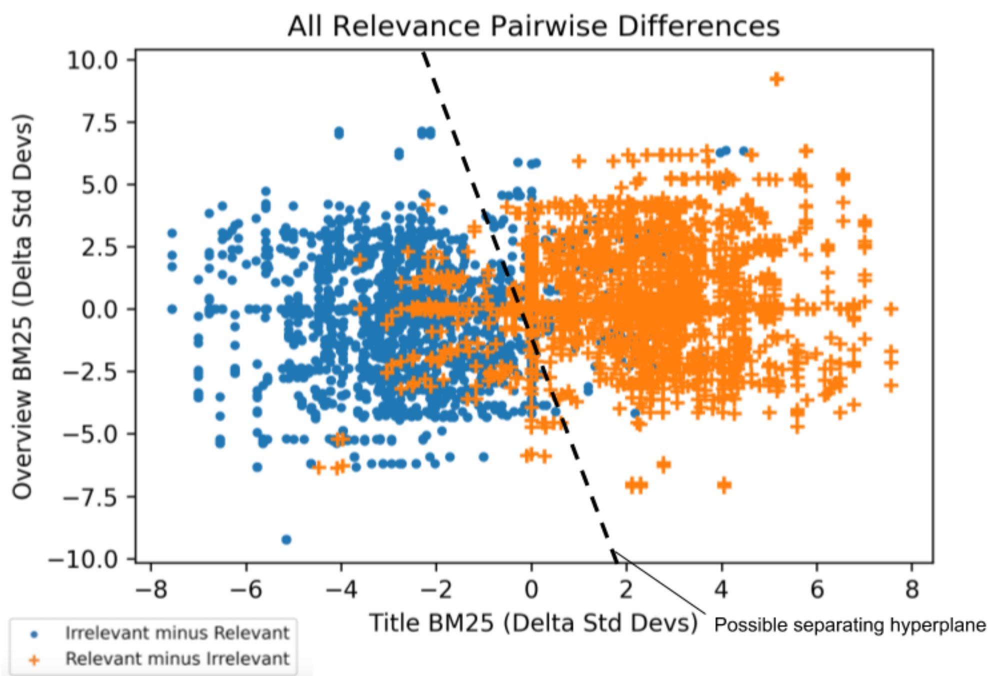


Figure 10.14 Full training set with candidate separating hyperplane

Interesting! It is now very easy to visually identify that a larger Title BM25 score match is highly correlated with a document being relevant for a query, and that having a higher Overview BM25 score is at least somewhat positively correlated.

It's worth taking a step back and asking whether this formulation of ranking is appropriate for your domain. Other LTR methods have their own ways of performing this trick of mapping pair-wise comparisons into classification problems, but it's important to get under the hood to see how your chosen solution performs. Other methods, like LambdaMART, perform their own tricks, but they may alternatively directly optimize for classic search relevance ranking metrics like precision or Discounted Cummulative Gain (DGC).

Next up, we'll cover how to train a robust model to capture the patterns in our fully-transformed ranking data set.

10.6 Step 4 — Training (and testing!) the model

Machine Learning clearly requires a lot of data preparation, but you've finally arrived at the section where we actually train a model! With the `feature_deltas` and `predictor_deltas` from the last section, we now have a training set suitable for training a classic machine learning model.

10.6.1 Turning a Seperating Hyperplane's Vector into a Scoring Function

We've seen how SVMRank's separating hyperplane can classify and separate irrelevant examples from the relevant ones. You might be thinking, "that's cool, but our task was actually to find 'optimal' weights for our features, not just to classify documents!"

That's a great point, but thankfully the separating hyperplane also gives us just what we need to learn optimal weights. Any hyperplane is defined by the vector orthogonal to the plane. So when an SVM machine learning library does it's work, it actually gives us a sense of the weights that each feature should have, as shown in Figure 10.15.

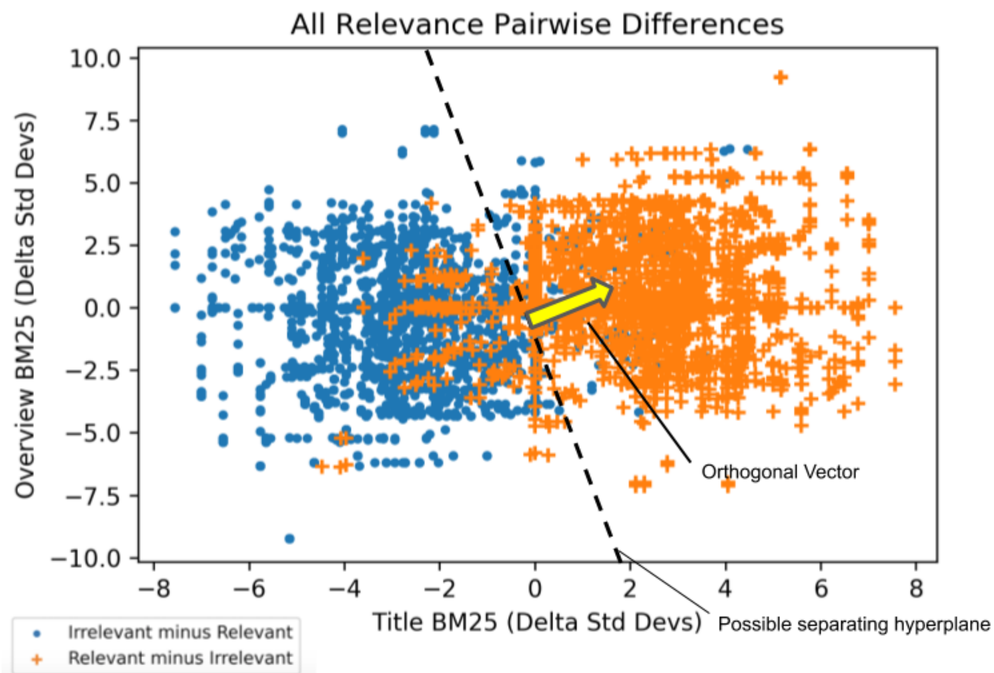


Figure 10.15 Full training set with candidate separating hyperplane

Think about what this orthogonal vector represents. This vector points in the direction of relevance! It says relevant examples are this way, and irrelevant ones are in the opposite direction. This vector *definitely* points to `title_bm25` having a strong influence on relevance, with some smaller influence from `overview_bm25`. This vector might be something like:

```
[0.65, 0.40]
```

If we train an SVM, as in Listing 10.13, the model gives us the separating hyperplane.

Listing 10.13 Training a linear SVM with scikit learn

```
from sklearn import svm
model = svm.LinearSVC(max_iter=10000, verbose=1)      # Create a Linear Model with sklearn
model.fit(feature_deltas, predictor_deltas)           # fit to the deltas using an SVM,
model.coef_
```

Results:

```
array([[0.40512169, 0.29006365, 0.14451721]])
```

Listing 10.13 trains an SVM to separate the `predictor_deltas` (remember they're `+1` and `-1` s) using the corresponding `feature_deltas` (the deltas in normalized `title_bm25`, `overview_bm25` and `release_year` features). The resulting model is a vector orthogonal to the separating hyperplane. As expected, it shows a strong weight on `title_bm25`, a more modest one on `overview_bm25`, and a weaker weight on `release_year`.

10.6.2 Taking the model for a test drive

How does this model work as a ranking function? Let's suppose the user types in the query `wrath of khan`. How might this model score the document "Star Trek II: The Wrath of Khan" relative to the keywords `wrath of khan` using this model? The unnormalized feature vector indicates a strong title and overview match for `wrath of khan`:

```
[5.9217176, 3.401492, 1982.0]
```

Normalizing it, each feature value is this many standard deviations above/below the features' means:

```
[3.099, 1.825, -0.568]
```

We simply mulitply each normalized feature with its corresponding `coef_` value. Summing them together, gives us a feature score:

```
(3.099 * 0.405) + (1.825 * 0.290) + (-0.568 * 0.1445) = 1.702
```

How would this model rank "Star Trek III: The Search for Spock" relative to "Star Trek II: Wrath of Khan"? Hopefully not nearly as highly! Indeed it doesn't:

```
[0.0,0.0,1984.0] # Raw Features
[-0.432, -0.444, -0.468] # Normalized features
(-0.432 * 0.405) + (-0.444 * 0.290) + (-0.468 * 0.1445) = -0.371
```

The model seems to be getting what we suspect to be the right answer towards the top.

10.6.3 Validating the Model

Kicking the tires on 1-2 queries helps us spot problems, but we'd prefer a more systematic way of checking if the model is generalizable.

One difference between LTR and classic machine learning is we usually evaluate queries, not individual data points, to prove our model is effective. So we'll perform a test/training split at the query level. It will let us spot types of queries with problems. We'll evaluate using a simple precision metric, counting the proportion of results in the top N (with N=4 in our case) that are relevant. You should choose the relevance metric best suited to your own use case, however.

First we will randomly put our queries into a test or training set, as shown in Listing 10.14.

Listing 10.14 Simple test/training split at the query level.

```
all_qids = list(set([j.qid for j in normed_judgments]))
random.shuffle(all_qids)
proportion_train=0.1

test_train_split_idx = int(len(all_qids) * proportion_train)
test_qids=all_qids[:test_train_split_idx]
train_qids=all_qids[test_train_split_idx:]

train_data = []; test_data=[]
for j in normed_judgments:
    if j.qid in train_qids:
        train_data.append(j)
    elif j.qid in test_qids:
        test_data.append(j)
```

With training data split, we can perform the pairwise transform trick from Step 3. We can then retrain on just the training data in Listing 10.15.

Listing 10.15 Train just on training data

```
train_data_features, train_data_predictors = pairwise_transform(train_data)

from sklearn import svm
model = svm.LinearSVC(max_iter=10000, verbose=1)
model.fit(train_data_features, train_data_predictors)
model.coef_[0]
```

Now we have held back the test data. Just like a good teacher, we don’t want to give the "student" all the answers. We want to see if the model has learned anything beyond rote memorization of training examples.

In Listing 10.16 we evaluate our model using the test data. This code loops over every test query and ranks every test judgment using the model (`rank` omitted). It then computes precision for the top 4 judgements.

Listing 10.16 Train just on training data

```
def eval_model(test_data, model):

    tot_prec = 0
    num_queries = 0

    for qid, query_judgments in groupby(test_data, key=lambda j: j.qid):
        query_judgments = list(query_judgments)

        ranked = rank(query_judgments, model)

        tot_relevant = 0
        for j in ranked[:4]:
            if j.grade == 1:
                tot_relevant += 1
        query_prec = tot_relevant/4.0
        tot_prec += query_prec
        num_queries += 1

    return tot_prec / num_queries
```

On multiple runs, you should expect a precision of approximately 0.3 - 0.4 . Not bad for our first iteration, where we just guessed at a few features (`title_bm25`, `overview_bm25`, and `release_year`)!

In LTR, you can always look back at previous steps to see what might be improved. This precision test is the first time we've been able to systematically evaluate our model, so it's a natural time to revisit the features to see how the precision might be improved in subsequent runs. Go all the way back up to Step 2. See what examples are on the wrong side of the separating hyperplane. For example, if you look at Figure 10.8, the 3rd Star Wars movie, "Return of the Jedi", fits a pattern of a relevant document that doesn't have a keyword match in the title. In the absence of a title, what other features might be added, to help capture that a movie belongs in a specific collection like Star Wars? Perhaps there is a TMDB movie property indicating this we could experiment with?

We'll consider deeper questions of model, feature, and especially training data quality in the next chapter. For now, though, let's take the model we just built and see how to deploy it to production.

10.7 Steps 5 and 6 - Upload a model and search

Let's finally tell Solr about our model.

Originally we presented our objective to be finding 'ideal' boosts for a manual ranking function like the one in Listing 10.2:

```
q="title:(${keywords})^10 overview:(${keywords})^20 {{!func}}release_year^0.01"
```

This manual function indeed multiplies each feature by a weight (the boost), and sums the results. But it turns out we actually don't want Solr to multiply the *raw* feature values. Instead we need the feature values to be normalized.

Luckily, Solr LTR lets us store a linear model along with feature normalization statistics. We saved off the `means` and `std_devs` of each feature, which Solr can use to normalize values for any document being evaluated. We just need to provide this information to Solr when storing a model, as we do in Listing 10.17.

Listing 10.17 Uploading model to Solr with normalization and weights for each feature.

```
PUT http://aips-solr:8983/solr/tmdb/schema/model-store
{
  "store": "movies",
  "class": "org.apache.solr.ltr.model.LinearModel",
  "name": "movie_titles",
  "features": [
    {
      "name": "title_bm25",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "1.5939970007512951",
          "std": "3.689972140122766"
        }
      }
    },
    {
      "name": "overview_bm25",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "1.4658440933160637",
          "std": "3.2978986984657808"
        }
      }
    },
    {
      "name": "release_year",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "1993.3349740932642",
          "std": "19.964916628520722"
        }
      }
    }
  ],
  "params": {
    "weights": {
      "title_bm25": 0.4491741700850341,
      "overview_bm25": 0.31118624504321124,
      "release_year": 0.13971710392127754
    }
  }
}
```

Note that the model is associated with a feature store so it can lookup feature names to compute them during evaluation.

Finally, we can issue a search using Solr's LTR query parser in Listing 10.18.

Listing 10.18 Rank 60,000 documents with our model to search for `harry potter`

```
request = {
    "fields": ["title", "id", "score"],
    "limit": 5,
    "params": {
      "q": "{!ltr reRankDocs=60000 model=movie_model efi.keywords=\"harry potter\"}",

    }
}

resp = requests.post('http://aips-solr:8983/solr/tmdb/select', json=request)

resp.json()["response"]["docs"]
```

Results:

```
{'id': '570724', 'title': ['The Story of Harry Potter'], 'score': 2.786719, '_score': 2.786719}
{'id': '116972', 'title': ['Discovering the Real World of Harry Potter'], 'score': 2.5646381,
                           '_score': 2.5646381}
{'id': '672', 'title': ['Harry Potter and the Chamber of Secrets'], 'score': 2.3106465,
                           '_score': 2.3106465}
{'id': '671', 'title': ["Harry Potter and the Philosopher's Stone"], 'score': 2.293983,
                           '_score': 2.293983}
{'id': '393135', 'title': ['Harry Potter and the Ten Years Later'], 'score': 2.2162843,
                           '_score': 2.2162843}
```

Notice in Listing 10.21 the use of the term 'rerank'. As this implies, LTR always happens as a second phase on top of another query. In Listing 10.21 there's not an initial query. So what happens in this case? The model is applied to the documents in the order they were indexed (essentially random). You can see from the results, the model seems effective at our query.

Usually, we don't want to rerank over such a large set of results. We'd rather have a quick, baseline match with simple scoring (such as BM25 or a simple edismax query), and THEN choose to do a more expensive rerank leveraging LTR over a smaller number of candidate documents. Listing 10.19 demonstrates how to implement a baseline search, reranking the top 500 using our LTR model.

Listing 10.19 Rerank top 500 with `movie_model` to search for `harry potter` after the top results from the initial, quick ranking calculation.

```
request = {
    "fields": ["title", "id", "score"],
    "limit": 5,
    "params": {
      "rq": "{!ltr reRankDocs=500 model=movie_model efi.keywords=\"harry potter\"}",
      "qf": "title overview",
      "defType": "edismax",
      "q": "harry potter"
    }
}

resp = requests.post('http://aips-solr:8983/solr/tmdb/select', json=request)
resp.json()["response"]["docs"]
```

10.7.1 A note on LTR performance

Good engineering requires balancing different constraints, not a single optimization. Your model has to run in a real-life system, answering live user queries. That's the difference between practical LTR and that done for a Kaggle competition or in an academic setting. You'll need to find the sweet spot between performance and relevance. Some quick pointers you'll need to consider

1. *Model complexity* - the more complex the model, likely the more accurate. A simpler model will be fast, though perhaps less accurate. Here we've stuck to a very simple model (a set of linear weights). Imagine a complex deep-learning model, how well would that work?
2. *Rerank depth* - the deeper you rerank, the more you might find additional documents that could be hidden gems. On the other hand, the deeper you rerank, the more CPU cycles your model spends scoring results in your live search engine cluster
3. *Feature complexity* - if you compute very complex features at query time, they might help your model. However they'll slow down evaluation and search response time.
4. *Number of features* - a model with many features might lead to higher relevance. However it will also take more time to compute every feature on each document, so ask yourself which features are crucial. Many academic LTR systems use hundreds. Practical LTR systems usually boil these down to the dozens. You will almost always see diminishing returns as you continue adding additional features, so choosing the right cut-off threshold on number of features is important

It is also very difficult to implement a successful LTR project that balances speed, reliability, and relevance without close collaboration between search engineers and data scientists.

10.8 Rinse and Repeat

Congrats! You've done one full cycle of Learning to Rank! Like many data problems though, you'll likely need to continue iterating on the problem. There's always something new you can do to improve. We'll continue iterating and improving how we do LTR in the next chapter.

On your second iteration, you might begin thinking through some of the following considerations:

1. *New and better features*. Are there types of queries or examples on which the model performs poorly? Such as `title` searches where there's no `title` mention (`star wars` is not mentioned in the title of "Return of the Jedi". What features could capture these?). Could we incorporate lessons from chapters 1-9 to construct more advanced features?
2. *Training data coverage of all features*. The flipside of more features is more training data. As you increase features you'd like to try, you should be wondering whether your training data has enough examples of relevant / irrelevant documents across each different combination of your features. Otherwise your model won't know how to leverage features to solve the problem.
3. *Different model architecture*. We used a relatively simple model that expects features to linearly correlate with relevance. Yet relevance can often be non-linear. A shopper

searching for iPads might expect the most recent Apple iPad release, except when they add the word 'cable', as in 'iPad cable'. For that query, they might just want the cheapest cable they can find.

In the next chapter, we will indeed 'rinse and repeat'. We’ll gain confidence in our iterations, pushing the boundaries on how much can be automated as our maturity increases.

10.9 Summary

- Learning to Rank builds general ranking functions using robust machine learning techniques
- In Solr LTR, features generally correspond to Solr queries. Solr LTR lets you store and log features for the purposes of training, and later applying, a ranking model.
- Features can either be properties of queries (like number of terms), properties of documents (like a popularity), or relationship between queries and documents (like BM25 or other relevance scores)
- LTR can require transforming a LTR training set into a different, point-wise training set that reformulates the problem into a classic machine learning problem
- SVMRank creates simple linear weights on normalized feature values, a good first step on your LtR journey
- LTR should be evalauted by doing multiple test/training splits at the query level, using metrics like precision
- Once a LTR model is loaded into your search engine, be sure to consider performance (as in speed) tradeoffs with relevance. Real-life search systems require both!

Appendix A: Running the Code Examples

A

This Appendix covers:

- How this book's source code examples are packaged
- Pulling the AI-Powered Search source code
- Building and running the examples
- Working with Jupyter
- Working with Docker

During your journey through *AI-Powered Search*, we'll walk through a lot of code and running software examples demonstrating the techniques within this book. This appendix will show you how to easily setup and run the accompanying source code so that you can interact and experiment with live, running examples as you work through the material.

A.1 Overall Structure of Code Examples

In order to build an AI-powered search system, it is necessary to integrate many components and libraries. For our core search engine, we will leverage Apache Solr, which internally leverages Apache Zookeeper. For significant data processing and machine learning tasks, we'll leverage systems like Apache Spark and Tensorflow. We'll leverage Python as our primary programming language for code examples, and will thus need to install and manage many Python library dependencies, in addition to other system dependencies (like Java) which several of our systems require. Of course, we also need the ablity to actually execute our code examples and see the results in a user-friendly way, which we'll accomplish through the use of Jupyter notebooks.

Instead of having you install dozens of software libraries and hundreds of dependencies to make this all work, however, we are making this process as easy as possible for readers by packaging all of the examples in this book into Docker containers which are already fully-configured and and ready to use. This means that there is a single prerequisite you must install before running the code examples in this book: Docker.

Docker enables the creation and running of tiny containers - fully-functioning virtual machines that run only a light-weight operating system with all of the needed software and dependencies already installed and configured.

Once all of the services are running, all of the code listings in the book will be available through Jupyter notebooks, which will serve as the interface for walking through and experimenting with the code examples and seeing the resulting outputs.

A.2 Pulling the source code

The source code accompanying this book is available at: github.com/treygrainger/ai-powered-search

To pull the code, either use an installed Git client or open up a terminal into your preferred development folder and run one of the following commands:

```
git clone https://github.com/treygrainger/ai-powered-search.git
```

or

```
git clone git@github.com:treygrainger/ai-powered-search.git
```

You should now have a new folder in your current directory called `ai-powered-search`, which contains all of the source code for the book.

If you do not have Git installed or can't pull the code through one of the above commands, there is also an option to download the source code as a zip through your web browser, which you would then just need to unzip into your development folder.

Feel free to rename or move this `ai-powered-search` folder if you wish; throughout the rest of this book, we'll simply refer to this directory using the variable `$AIPS_HOME`.

A.3 Building and running the code

As previously mentioned, Docker is the one key dependency you must install on your system in order to build and run the AI-Powered Search examples. We will not cover this installation process here as it is system-dependent and changes from time to time, so please visit www.docker.com for download and installation instructions.

Once you have Docker installed, all you need to do is run the following command:

```
cd $AIPS_HOME/docker
docker-compose up
```

TIP The `docker-compose up` command in the foreground of your console, which allows you to see all logs streaming by in real-time, but which also means your containers will all die if you close the console. If you would like to instead run the containers in the background and continue using your console, you can pass in the `-d` or `--detach` parameter (`docker-compose -d`). If you launch like this, be sure to explicitly run `docker-compose down` when you are finished to stop the containers from running indefinitely in the background consuming resources.

This is helpful for seeing live logs, but also means the containers will be stopped as soon as you close the console. If you'd like to start them up to continue running in the back

This command will take a while to run the first time, as it is pulling in all of the software, operating systems, and dependencies needed to build and run the software accompanying this book.

Once the command finishes, however, you will have all of the necessary services (Jupyter, Solr, Zookeeper, Spark, etc.) running in separate Docker containers. Now, to get started, simply open up your web browser and go to:

```
http://localhost:8888/notebooks/welcome.ipynb
```

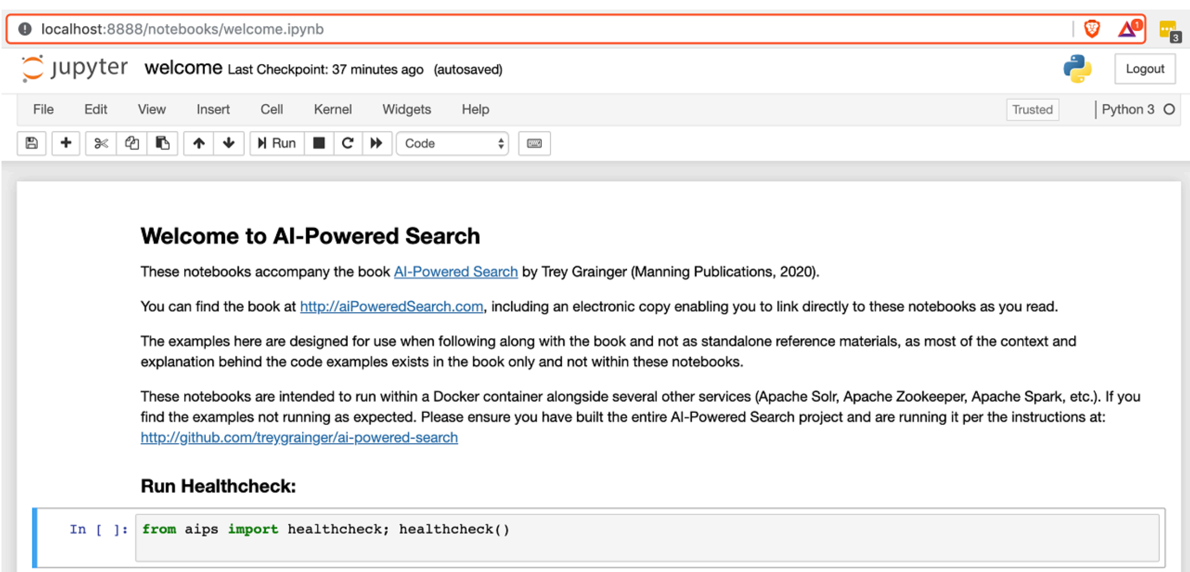


Figure A.1 Welcome screen. Once you see this, the AI-Powered Search containers are built and the Jupyter notebooks are running.

A.4 Working with Jupyter

Once you load the `welcome.ipynb` notebook, you'll see a few data cells on the screen, including an introduction message, a "health check" script, and a table of contents to various notebooks containing executable examples from the book.

If you've never used Jupyter before, it is a tool that lets you mix markup (usually instructions and explanation) and code in your browser, and to edit, run, and interact with the output from the code examples. This makes learning much easier, as you don't have to use command line tools and can instead interact entirely with ready-to-execute examples with the push of a button.

You'll notice a toolbar near the top of the screen (below the menu bar), which allows you to interact with the sections of content (called "cells") within the notebook. You can use these to navigate up and down, to stop and restart the notebook, or to execute each cell sequentially using the "Run" button.

Figure A.2 Running code examples. Clicking "Run" in the toolbar will execute any examples in the current cell (if any) and proceed to the next cell.

In Figure A1.2, clicking "Run" while the healthcheck code cell is highlighted will result in the healthcheck executing to confirm that all Docker containers are running and that the services running within them are healthy and responding.

Figure A1.3 shows the response you will see when everything is running as expected.

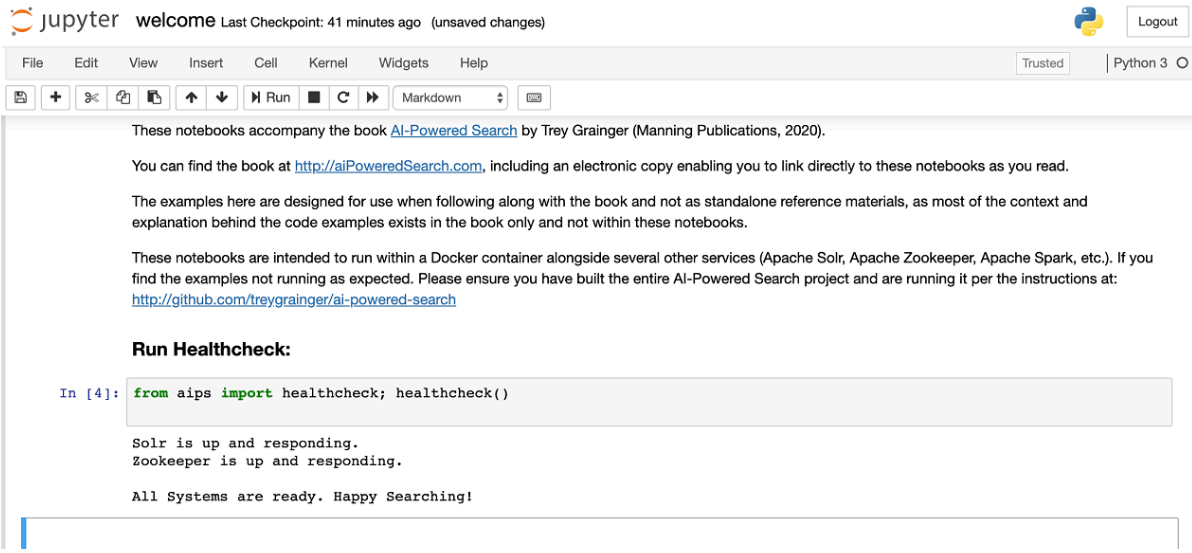


Figure A.3 Healthcheck Success. You should see this message if everything is running correctly.

At this point you can scroll down to the table of contents and proceed through the notebooks for each chapter. Of course, since the explanation behind the examples is contained within the book, you'll probably prefer to work through the examples as you're reading through the book so that you have the appropriate background understanding when running them. The Jupyter notebooks are not intended to be stand-alone examples, so you'll probably want to keep the book close by to provide context.

All Jupyter notebooks are designed to be independently idempotent. This means that, while all steps in a notebook need to be executed in order to guarantee a successful result, that you can always start back over at the beginning of any notebook and it will "reset" to the expected results necessary to make the following steps succeed. If ever you experience errors in a notebook, just go back to the first cell on the page and run through the whole notebook again!

A.5 Working with Docker

While everything in the previous sections should work as expected, it's of course possible you could run into problems along the way. The most likely challenge you'll face is for one of your Docker containers, or the service running inside of it, to fail. It's also possible, if you're making changes to underlying data or config in one of the services, for example, that you could put it in a bad state.

When this happens, you can always tear down your containers and start over. To do this, just run:

```
cd $AIPS_HOME/docker
docker-compose down && docker-compose up
```

Keep in mind that if you're doing anything on your cluster *other* than running through the examples, that any work you've done will be lost. In general, the examples are designed to be transient. If you want to preserve your work across container stops and starts, you can modify the `docker-compose.yaml` file to make your data volumes `external` and thus persistent. Please refer the the Docker documentation if you plan to make changes there, as the mechanisms and APIs can change from time to time.

If you ever modify the code examples or your configuration, it is also possible you may need to rebuild your Docker images. When you run `docker-compose up` the first time, it will build your images and start them, but it doesn't rebuild with changes made since the first build. To rebuild everything prior to starting, you can instead run:

```
cd $AIPS_HOME/docker
docker-compose build && docker-compose up
```

This should give you everything you need to run through all of the notebooks and code in *AI-Powered Search*. Happy searching!

Notes

1. John Rupert Firth (1957). "A synopsis of linguistic theory 1930-1955." In Special Volume of the Philological Society. Oxford: Oxford University Press.

2. Marti Hearst. “Automatic Acquisition of Hyponyms from Large Text Corpora”. 1992. The 15th International Conference on Computational Linguistics

3. Trey Grainger et al. “The Semantic Knowledge Graph: A Compact, Auto-Generated Model for Real-Time Traversal and Ranking of any Relationship within a Domain.” 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA) (2016): 420-429.