10 Learning to rank for generalizable search relevance

This chapter covers

- An introduction to machine-learned ranking, also known as learning to rank (LTR)
- How LTR differs from other machine learning methods
- Training and deploying a ranking classifier
- Feature engineering, judgment lists, and integrating machine-learned ranking models into a search engine
- Validating an LTR model using a train/test split
- Performance tradeoffs for LTR-based ranking models

It's a random Tuesday. You review your search logs, and the searches range from the frustrated runner's `polar m430 running watch charger` query to the worried hypochondriac's `weird bump on nose - cancer?` to the curious cinephile's `william shatner first film`. Even though these may be one-off queries, you know each user expects nothing less than amazing search results.

You feel hopeless. You know many query strings, by themselves, are distressingly rare. You have very little click data to know what's relevant for these searches. Every day gets more challenging: trends, use cases, products, user interfaces, and even user terminology evolve. How can anyone hope to build search that amazes when users seem to constantly surprise us with new ways of searching?

Despair not, there is hope! In this chapter, we'll introduce generalizable relevance models. These models learn the underlying patterns that drive relevance ranking. Instead of memorizing that the article entitled "Zits: bumps on nose" is the answer for the query `weird bump on nose - cancer?` we observe the underlying pattern—that a strong title match corresponds to high probability of relevance. If we can learn these patterns and encode them into a model, we can give relevant results *even for search queries we've never seen*.

This chapter explores *learning to rank* (LTR): a technique that uses machine learning to create generalizable relevance ranking models. We'll prepare, train, and search with LTR models using the search engine.

10.1 What is LTR?

Let's explore what LTR does. We'll see how LTR creates generalizable ranking models by finding patterns that predict relevance. We'll then explore more of the nuts and bolts of building a model.

10.1.1 Moving beyond manual relevance tuning

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 those in the following listing. This document comes from TheMovieDB (tmdb) corpus (**http://themoviedb.org**), which we'll use in this chapter. If you wish to follow along with the code for this chapter, use this chapter's first notebook to index the tmdb dataset.

Listing 10.1 A document for the movie *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}
```

Through endless iterations and tweaks, we might arrive at a generalizable movie ranking function that looks something like the next listing.

Listing 10.2 A generalizable ranking function using manual boosts

```
keywords = "some example keywords"
{"query": f"title:({keywords})^10 overview:({keywords})^20
↪{!func}release_year^0.01"}
```

Manually optimizing the feature weights of general ranking functions like this to work over many queries can take significant effort, but such optimizations are perfect for machine learning.

This is where LTR comes in—it takes our proposed relevance factors and learns an optimal ranking function. LTR 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 in listing 10.2. With this relatively simple task, we'll see the full lifecycle of developing an LTR solution.

10.1.2 Implementing LTR in the real world

As we continue to define LTR at a high level, let's quickly clarify where LTR fits into the overall picture of a search system. Then we can look at the kinds of data we'll need to build an LTR model.

We'll focus on building LTR for production search systems, which can be quite different from a research context. We not only need relevant results, but results returned suitably quickly, with mainstream, well-understood search techniques.

Conceptually, invoking LTR usually involves three high-level steps:

1. Training an LTR model
2. Deploying the model to production
3. Using the model to rank (or rerank) search results

Most modern search engines support deploying ranking models directly into the search engine, allowing the LTR model to be invoked efficiently "where the data lives". Usually, LTR models are significantly slower at ranking than basic keyword-based ranking functions like BM25, so LTR models are often only invoked for subsequent-pass ranking (or reranking) on a subset of the top search results ranked by an initial, faster ranking function. Pushing the LTR model into the engine (if supported) prevents the need to return hundreds or thousands of documents and their metadata from the search engine to an external model service for reranking, which can be slow and inefficient compared to doing the work in-engine and at scale.

For this reason, our `ltr` library in this chapter implements pluggable support for deploying and invoking each supported search engine or vector database's native LTR model integration capabilities when available. The code in each listing will work with any supported engine (see appendix B to change it), but the listing output you'll see in this chapter will reflect Solr's LTR implementation, since Solr is configured by default. If you change the engine, you'll see the output from your chosen engine when you run the Jupyter notebooks.

Solr was one of the first major open source search engines to natively support LTR model serving, with the capabilities later being ported to a community-developed Elasticsearch LTR plugin (**https://github.com/o19s/elasticsearch-learning-to-rank**) and then forked to the OpenSearch LTR plugin (**https://github.com/opensearch-project/opensearch-learning-to-rank-base**). As such, the Elasticsearch and OpenSearch LTR plugins implement nearly identical concepts as those in Solr. Vespa implements phased ranking (reranking) and the ability to invoke models during each phase, and Weaviate also implements various reranking capabilities. Other engines that support native LTR will follow similar patterns.

Figure 10.1 outlines the workflow for developing a practical LTR solution.

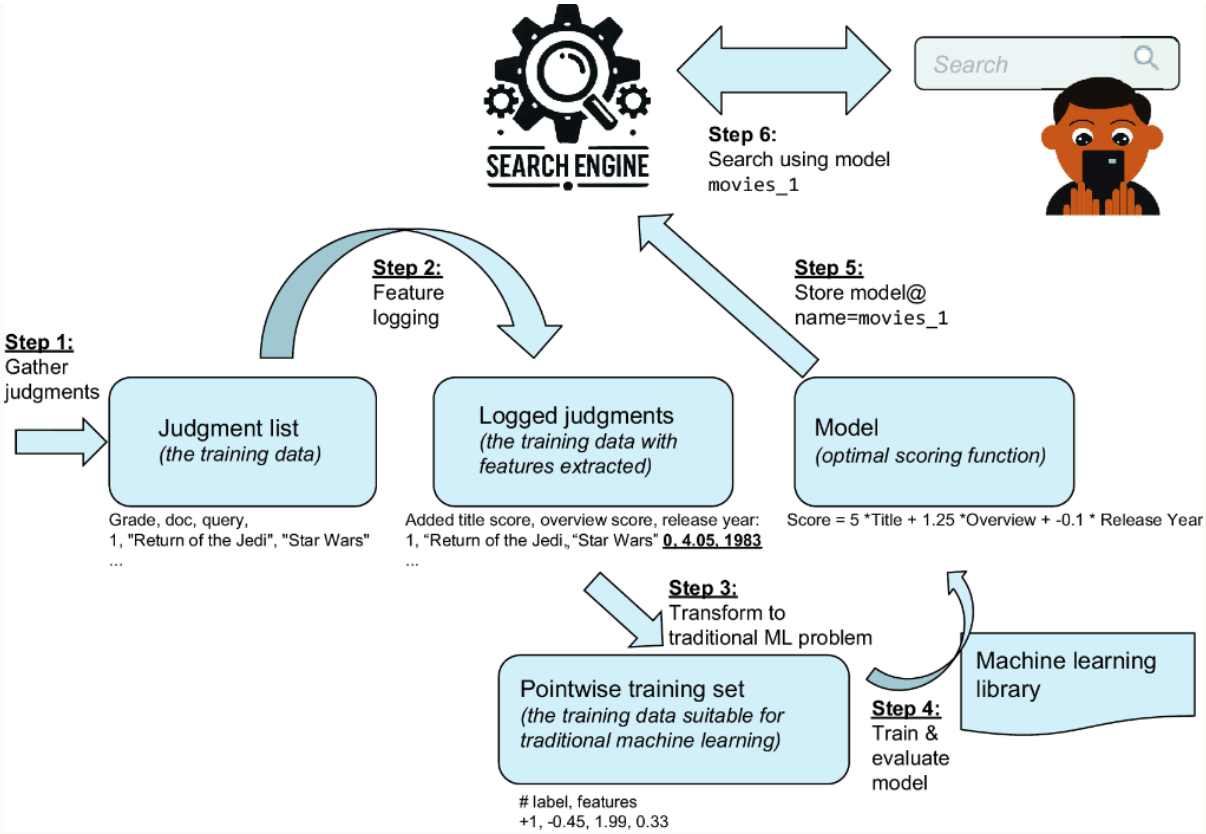

Figure 10.1 LTR systems transform our training data (judgment lists) into models that generalize relevance ranking. This type of system lets us find the underlying patterns in our training data.

You may notice similarities between LTR and traditional machine learning–based classification or regression system workflows. But the exceptions are what make it interesting. Table 10.1 maps definitions between traditional machine learning objectives and LTR.

Table 10.1 Traditional machine learning vs. LTR

Concept	Traditional machine learning	LTR
Training data	Set of historical or "true" examples the model should try to predict, e.g., stock prices on past days, like "Apple" was $125 on June 6th, 2021.	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` .
Feature	The data we can use to predict the training data, e.g., Apple had 147,000 employees and revenue of $90 billion.	Data used so that relevant results rank higher than irrelevant ones and, ideally, values the search engine can compute quickly. Our features are search queries like `title:({keywords})` from listing 10.2.
Model	The algorithm that takes features as input to make a prediction. Given that Apple has 157,000 employees on July 6th, 2021, with $95 billion in revenue, the model might predict a stock price of $135 for that date.	Combines the ranking features (search queries) together to assign a relevance *score* to each potential search result. Results are sorted by score descending, hopefully placing more relevant results first.

This chapter follows the steps in figure 10.1 to train an LTR model:

1. *Gather judgments* —We derive judgments from clicks or other sources. We'll cover this step in depth in chapter 11.
2. *Feature logging* —To train a model, we must combine the judgments with features to see the overall pattern. This step requires us to ask the search engine to store and compute queries representing the features.
3. *Transform to a traditional machine learning problem* —You'll see that most LTR really is about translating the ranking task into something that looks more like the "traditional machine learning" column in table 10.1.
4. *Train and evaluate the model* —Here we construct our model and confirm that it is, indeed, generalizable, and thus will perform well for queries it hasn't seen.
5. *Store the model* —We upload the model to our search infrastructure, tell the search engine which features to use as input, and enable it for users to use in their searches.
6. *Search using the model* —We finally can execute searches using the model!

The rest of the chapter will walk through each of these steps in detail to build our first LTR implementation. Let's get cracking!

10.2 Step 1: A judgment list, starting with the training data

You already saw what LTR is at a high level, so let's get into the nitty-gritty. Before implementing LTR, we must first learn about the data used to train an LTR model: the judgment list.

A *judgment list* is a list of relevance labels or *grades*, each indicating the 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` indicates an irrelevant document and a `1` indicates a relevant one.

Using the `Judgment` class provided with this book's code, we'll label *The Social Network* as relevant for the query `social network` by creating a `Judgment`:

```
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 10.3, we have `social network` and `star wars` as two different queries, with movies graded as relevant or irrelevant.

Listing 10.3 Labeling movie judgments as relevant or irrelevant

```
sample_judgments = [
  # for 'social network' query
  Judgment(1, "social network", 37799),  # The Social Network
  Judgment(0, "social network", 267752), # #chicagoGirl
  Judgment(0, "social network", 38408),  # Life As We Know It
  Judgment(0, "social network", 28303),  # The Cheyenne Social Club

  # for 'star wars' query
  Judgment(1, "star wars", 11),     # Star Wars
  Judgment(1, "star wars", 1892),   # Return of the Jedi
  Judgment(0, "star wars", 54138),  # Star Trek Into Darkness
  Judgment(0, "star wars", 85783),  # The Star
  Judgment(0, "star wars", 325553)  # Battlestar Galactica
]
```

You can see that we labeled *Star Trek into Darkness* and *Battlestar Galactica* as irrelevant for the query `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. To get training data in bulk, we usually derive these labels from click traffic using a type of algorithm known as a *click model*. As this step is so foundational, we'll dedicate all of chapter 11 to diving deeper into the topic. In this chapter, however, we'll start with manually labeled judgments so we can initially focus on the mechanics of LTR.

Each judgment also has 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, so if you inspect `sample_judgments[0].features`, it's currently empty (`[]`).

Let's use the search engine to gather some features.

10.3 Step 2: Feature logging and engineering

Feature engineering requires identifying patterns between document attributes and relevance. For example, we might hypothesize that "relevant results in our judgments correspond to strong title matches". In this case, "title match" would be a feature we'd need to define. In this section, you'll see what features (like "title match") are and how to use a modern search engine to engineer and extract these features from a corpus.

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 building blocks we use to build a ranking function. You've already seen a manual ranking function with features in listing 10.2: the keyword score in the `title` field is one such feature, as are the `release_year` and `overview` keyword scores:

```
{"query": f"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 the commute distance in a 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 of each component of listing 10.2 for the query `social network`, we would arrive at something like table 10.2.

Table 10.2 Features logged for the keywords `social network` for relevant (`grade=1`) and 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 Cheyenne Social Club	3.4286604	3.1086721	1970.0

A machine learning algorithm might examine the feature values from table 10.2 and converge on a good ranking function. From just the data in table 10.2, 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.

10.3.1 Storing features in a modern search engine

Modern search engines that support LTR help us store, manage, and extract features. Engines like Solr, Elasticsearch, and OpenSearch track features 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 search engine will execute the model.

As shown in listing 10.4, we generate and upload features to the search engine. We use a generic feature store abstraction in the book's codebase, allowing us to generate various search-based features and upload them as a *feature set* to the feature store of a supported search engine. Here we create three features: a title field relevance score `title_bm25`, an overview field relevance score `overview_bm25`, and the value of the `release_year` field. BM25 here corresponds to the BM25-based scoring defined in chapter 3, which will be our default method for scoring term matches in text fields.

Listing 10.4 Creating three features for LTR

```
feature_set = [
  ltr.generate_query_feature(feature_name="title_bm25",
                             field_name="title"),
  ltr.generate_query_feature(feature_name="overview_bm25",
                             field_name="overview"),
  ltr.generate_field_value_feature(feature_name="release_year",
                                   field_name="release_year")]

ltr.upload_features(features=feature_set, model_name="movie_model")
display(feature_set)
```

Engine-specific feature set definition (for `engine=solr`):

```
[{"name": "title_bm25",  #1
  "store": "movies",  #2
  "class": "org.apache.solr.ltr.feature.SolrFeature",
  "params": {"q": "title:(${keywords})"}},  #3
 {"name": "overview_bm25", #4
  "store": "movies",
  "class": "org.apache.solr.ltr.feature.SolrFeature",
  "params": {"q": "overview:(${keywords})"}},
 {"name": "release_year", #5
  "store": "movies",
  "class": "org.apache.solr.ltr.feature.SolrFeature",
  "params": {"q": "{!func}release_year"}}]  #6
```

#1 The name of the feature
#2 The feature store where the feature will be saved
#3 A parametrized feature taking the keywords (e.g., star wars) and searching the title field
#4 Another feature that searches against the overview field
#5 A document-only feature, the release_year of the movie
#6 params are the same params for a Solr query, allowing you to use the full power of Solr's extensive Query DSL to craft features.

The output of listing 10.4 shows the feature set that's uploaded to the search engine—in this case, a Solr feature set. This output will obviously look different based on which search engine implementation you configure (as discussed in appendix B). The first two features are parameterized: they each take the search keywords (`social network`, `star wars`) and execute a search on the corresponding field. The final one is a field value feature utilizing the release year of a movie, which will boost more recent movies higher.

10.3.2 Logging features from our search engine corpus

With features loaded into the search engine, our next focus will be to log features for every row in our judgment list. After we get this last bit of plumbing out of the way, we will then train a model that can observe relationships between each relevant and irrelevant document for each query.

For each unique query in our judgment list, we need to extract the features for the query's graded documents. For the query `social network` in the sample judgment list from list-

ing 10.3, we have one relevant document (37799) and three irrelevant documents (267752, 38408, and 28303).
The following listing shows an example of feature logging for the query `social network`.

Listing 10.5 Logging feature values for `social network` results

```
ids = ["37799", "267752", "38408", "28303"]  #1
options = {"keywords": "social network"}
ltr.get_logged_features("movie_model", ids,  #2
                        options=options, )  #2
                        fields=["id", "title"])  #2
display(response)
```

#1 Relevant and irrelevant documents for the "social network" query
#2 Queries the search engine for feature values contained in the movies feature store

Engine-specific search request (for `engine=solr`):

```
{"query": "{!terms f=id}37799,267752,38408,28303",
 "fields": ["id", "title",
   '[features store=movies efi.keywords="social network"]']}  #1
```

#1 Example Solr query syntax to retrieve feature values from each returned document

Documents with logged features:

```
[{"id": "37799",
  "title": "The Social Network",
  "[features]": {"title_bm25": 8.243603, #1
                 "overview_bm25": 3.8143613,  #1
                 "release_year": 2010.0}},  #1
 {"id": "267752",
  "title": "#chicagoGirl",
  "[features]": {"title_bm25": 0.0,
                 "overview_bm25": 6.0172443,
                 "release_year": 2013.0}},
 {"id": "38408",
  "title": "Life As We Know It",
  "[features]": {"title_bm25": 0.0,
                 "overview_bm25": 4.353118,
                 "release_year": 2010.0}},
 {"id": "28303",
  "title": "The Cheyenne Social Club",
  "[features]": {"title_bm25": 3.4286604,
                 "overview_bm25": 3.1086721,
                 "release_year": 1970.0}}]
```

#1 Each feature value logged for this movie for the "social network" query

Notice that the search request (for Solr in this case) in listing 10.5 has a return field containing square brackets. This syntax tells Solr to return an extra field on each document containing the feature data defined in the feature store (the `movies` feature store in this case). The `efi` parameter stands for *external feature information*, and it's used here to pass the keyword query (`social network`) and any additional query-time information needed to

compute each feature. The response contains the four requested documents with their corresponding features. These parameters will be different for each search engine, but the concepts will be similar.
With some mundane Python data transformation, we can fill in the features for the query `social network` in our training set from this response. In listing 10.6, we apply feature data to judgments for the query `social network`:

Listing 10.6 Judgments with logged features for query `social network`

```
[Judgment(grade=1, keywords="social network", doc_id=37799, qid=1,
          features=[8.243603, 3.8143613, 2010.0], weight=1),  #1
 Judgment(0, "social network", 267752, 1, [0.0, 6.0172443, 2013.0], 1),
 Judgment(0, "social network", 38408, 1, [0.0, 4.353118, 2010.0],1), #2
 Judgment(0, "social network", 28303, 1, [3.4286604, 3.1086721, 1970.0], 1)]
```

#1 Judgment for the movie The Social Network relative to the "social network" query, including the logged feature values
#2 An irrelevant document for the "social network" query (note the low first feature value, the title_bm25 score of 0.0)

In listing 10.6, 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. Let's repeat the process of logging features for judgments for the query `star wars`.

Listing 10.7 Logged judgments for the query `star wars`

```
[Judgment(1, "star wars", 11, 2, [6.7963624, 0.0, 1977.0], 1),
 Judgment(1, "star wars", 1892, 2, [0.0, 1.9681965, 1983.0], 1),
 Judgment(0, "star wars", 54138, 2, [2.444128, 0.0, 2013.0], 1),
 Judgment(0, "star wars", 85783, 2, [3.1871135, 0.0, 1952.0], 1),
 Judgment(0, "star wars", 325553, 2, [0.0, 0.0, 2003.0], 1)]
```

With the ability to generate logged judgments, let's expand the judgment list to 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 search engine request shown in listing 10.5. The end result of this feature logging looks just like listing 10.7, but created from a much larger judgment list.

We'll move on next to consider how to handle the problem of ranking as a machine learning problem.

10.4 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 well-known, traditional machine learning concepts to our LTR task.

The task of LTR is to look over many relevant and irrelevant training examples for a query and then build a model to bring more relevant documents to the top (and conversely push less relevant documents down). Each training example doesn't have much value by itself;

what matters is how it's ordered alongside its peers in a query. Figure 10.2 shows this task, with two queries. The goal is to find a scoring function that can use the features to correctly order results.

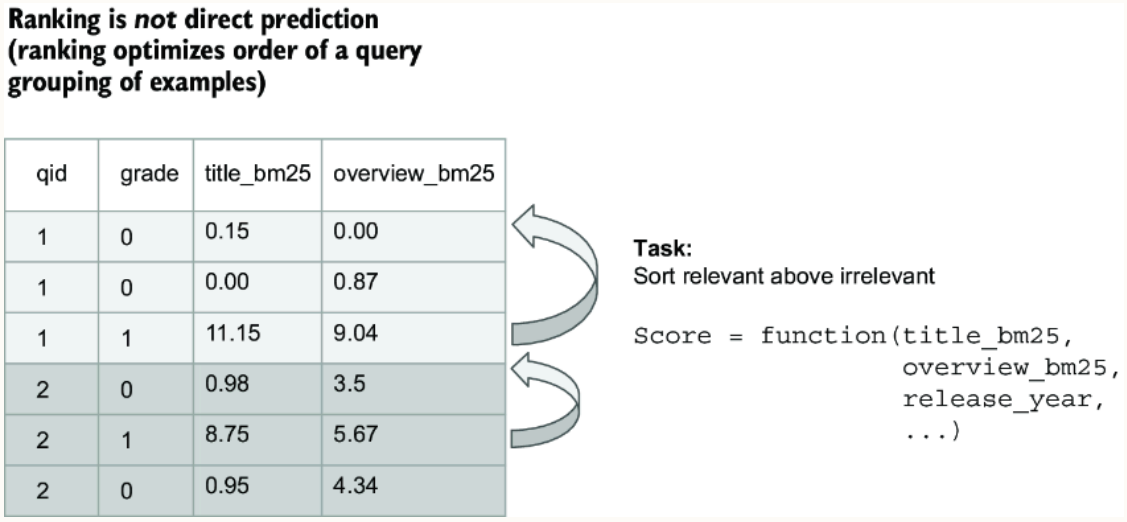

Figure 10.2 LTR is about placing each query's result set in the ideal order, not about predicting individual relevance grades. That means we need to look at each query as a case unto itself.

Contrast LTR with a more traditional pointwise machine learning task: a task like predicting a company's stock price as mentioned in table 10.2 earlier. *Pointwise machine learning* means that we can evaluate the model's accuracy on each example in isolation, predicting its absolute value as opposed to its relative value versus other examples. We know, just by looking at one company, how well we predicted that company's stock price. Compare figure 10.3 showing a pointwise task to figure 10.2. Notice in figure 10.3 that 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.

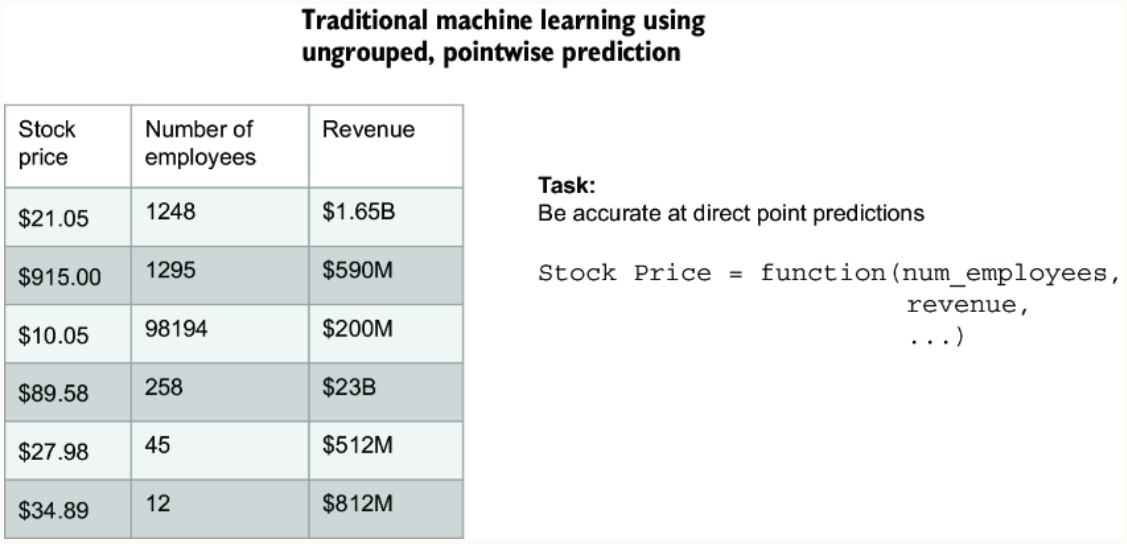

Figure 10.3 Pointwise machine learning tries to optimize predictions of individual points (such as a stock price or the temperature). Search relevance is a different problem than pointwise prediction. Instead, we need to optimize a ranking of examples grouped by a search query.

LTR targets a very different objective (ranking multiple results) than pointwise machine learning (predicting specific values of results). Most LTR methods use clever alchemy to transmogrify this "ranking of pairs" task into a classification task per document that learns to predict which features and feature weights best separate "relevant" from "irrelevant" documents. This transformation is the key to building a generalizable LTR model that can

operate on specific documents as opposed to only pairs of documents. We'll look at one model's method for transforming the ranking task in the next section by exploring a popular LTR model named SVMrank.

10.4.1 SVMrank: Transforming ranking to binary classification

At the core of LTR is the model: the actual algorithm that learns the relationship between relevance/irrelevance and the features like `title_bm25`, `overview_bm25`, etc. In this section, we'll explore one such model, SVMrank, first understanding what "SVM" stands for and then how it can be used to build a great, generalizable LTR model.

SVMrank transforms relevance into a binary classification problem. *Binary classification* simply means classifying items as one of two classes (like "relevant" versus "irrelevant", "adult" versus "child", "dog" versus "cat") using the available features.

An *SVM* or *support vector machine* is one method of performing binary classification. We won't go in-depth into SVMs, as you need not be a machine learning expert to follow the discussion. Nevertheless, if you want to get a deeper overview of SVMs, you can look at a book such as *Grokking Machine Learning* by Luis Serrano (Manning, 2021).

Intuitively, an SVM finds the best, most generalizable hyperplane to draw between the two classes. A *hyperplane* is a boundary that separates a vector space into two parts. A 1D point can be a hyperplane separating a 2D line into two parts, just as a line can be a hyperplane separating a 3D space into two parts. A plane is usually a 3D boundary separating a 4D space. All of these, as well as boundaries even greater than three dimensions are generically referred to as hyperplanes.

As an example, if we were trying to build a model to predict whether an animal is a dog or cat, we might look at a 2D graph of the heights and weights of known dogs or cats and draw a line separating the two classes as shown in figure 10.4.

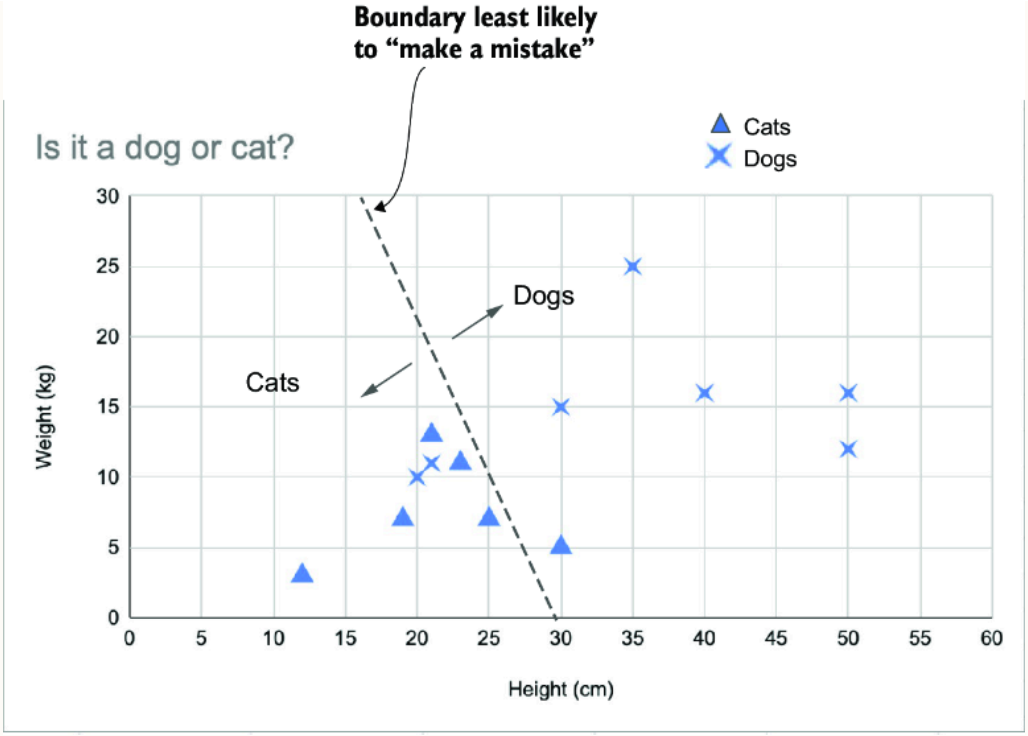

Figure 10.4 SVM example: Is an animal a dog or a cat? This hyperplane (the line here) separates these two cases based on two features: height and weight. Soon you'll see how we might do something similar to separate relevant and irrelevant search results for a query.

A good separating hyperplane drawn between the classes attempts to minimize the mistakes it makes in classifying 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 wouldn't be very AI-powered!

Another detail to know about SVMs is that they can be sensitive to the range of our features. For example, imagine if the `height` feature was millimeters instead of centimeters, like in figure 10.5. It forces the data to stretch out on the *x*-axis, and the separating hyperplane looks quite different!

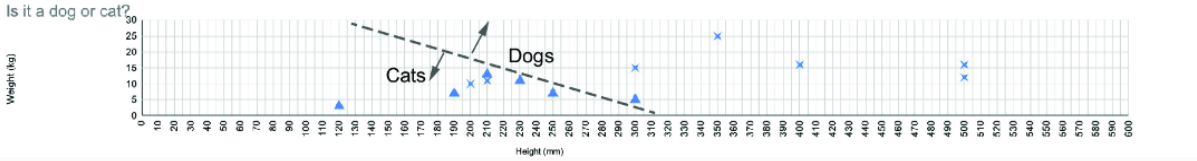

Figure 10.5 Separating hyperplane affected by the range of one of the features. This causes SVMs to be sensitive to the range of features, and thus we need to normalize the features so one feature doesn't create undue influence on the model.

SVMs work best when our data is normalized. *Normalization* just means scaling features to a comparable range. We'll normalize our data by mapping `0` to the mean of the feature values. If the average `release_year` is `1990`, movies released in 1990 will normalize to `0`. We'll also map `+1` and `-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`. We can repeat this for `title_bm25` and `overview_bm25` using those features' means and standard deviations in our training data. This helps make the features a bit more comparable when finding a separating hyperplane.

With that brief background out of the way, let's now explore how SVMrank can create a generalizable model to distinguish relevant from irrelevant documents, even for queries it has never seen before.

10.4.2 Transforming our LTR training task to binary classification

With LTR, we must reframe the task from ranking to a traditional machine learning task. In this section, we'll explore how SVMrank transforms ranking into a binary classification task suitable for an SVM.

Before we get started, let's inspect the fully logged training set from 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 10.6 shows these two features for every graded document for the `star wars` and `social network` queries, labeling some prominent movies from the training set.

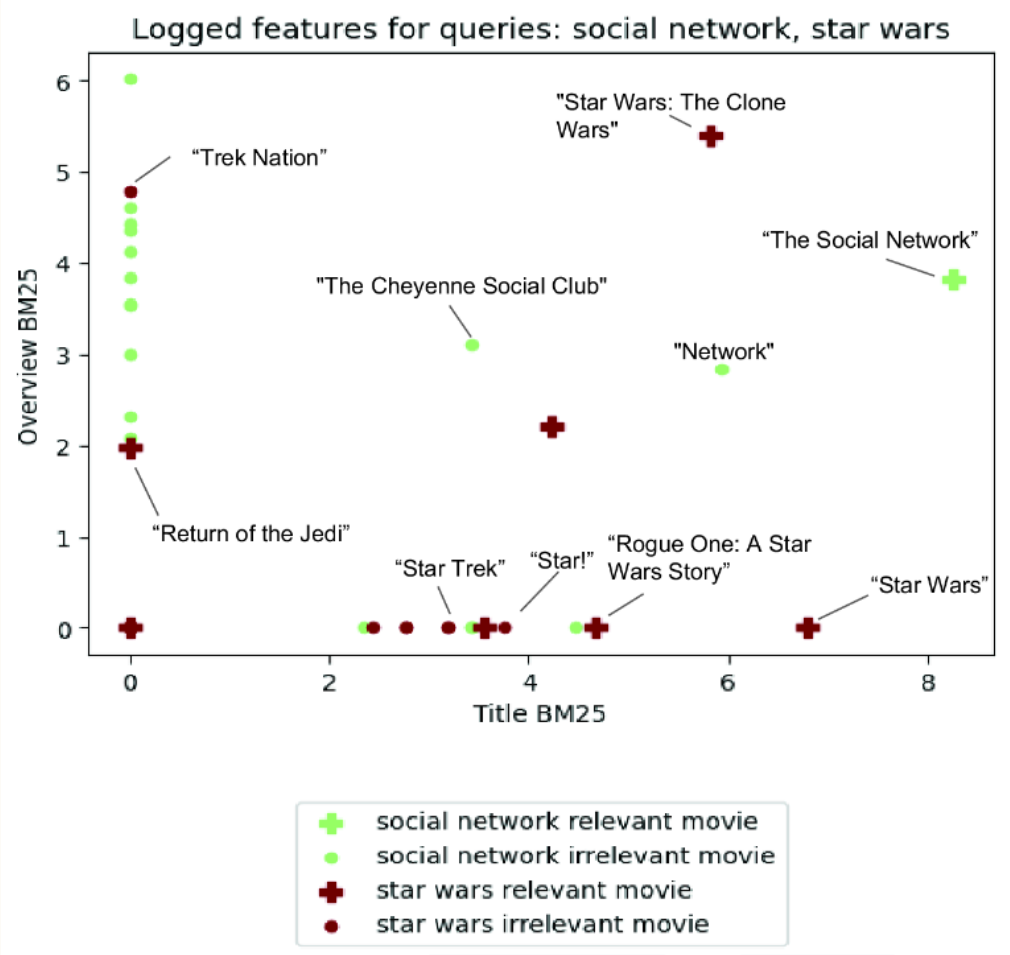

Figure 10.6 Logged feature scores for `social network` and `star wars` queries

FIRST, NORMALIZE THE LTR FEATURES

Our first step is to normalize each feature. The following listing takes the logged output from step 2 and normalizes features into `normed_judgments`.

Listing 10.8 Normalizing logged LTR training data

```
means, std_devs, normed_judgments = normalize_features(logged_judgments)
print(logged_judgments[360])
print(normed_judgments[360])
```

Output:

```
#Judgment(grade, keywords, doc_id,
#         qid, features, weight)
Judgment(1, "social network", 37799,  #1
         11, [8.244, 3.814, 2010.0], 1)  #1
Judgment(1, "social network", 37799,  #2
         11, [4.483, 2.100, 0.835], 1)  #2
```

#1 Unnormalized example, with raw title_bm25, overview_bm25, and release_year

#2 Same judgment, but normalized

You can see that the output from listing 10.8 shows first the logged BM25 scores for title and overview (`8.244`, `3.814`) alongside the release year (`2010`). These features are then normalized, where `8.244` for `title_bm25` corresponds to `4.483` standard deviations above the mean `title_bm25`, and so on for each feature.

We've plotted the normalized features in figure 10.7. This looks very similar to figure 10.6, with only the scale on each axis differing.

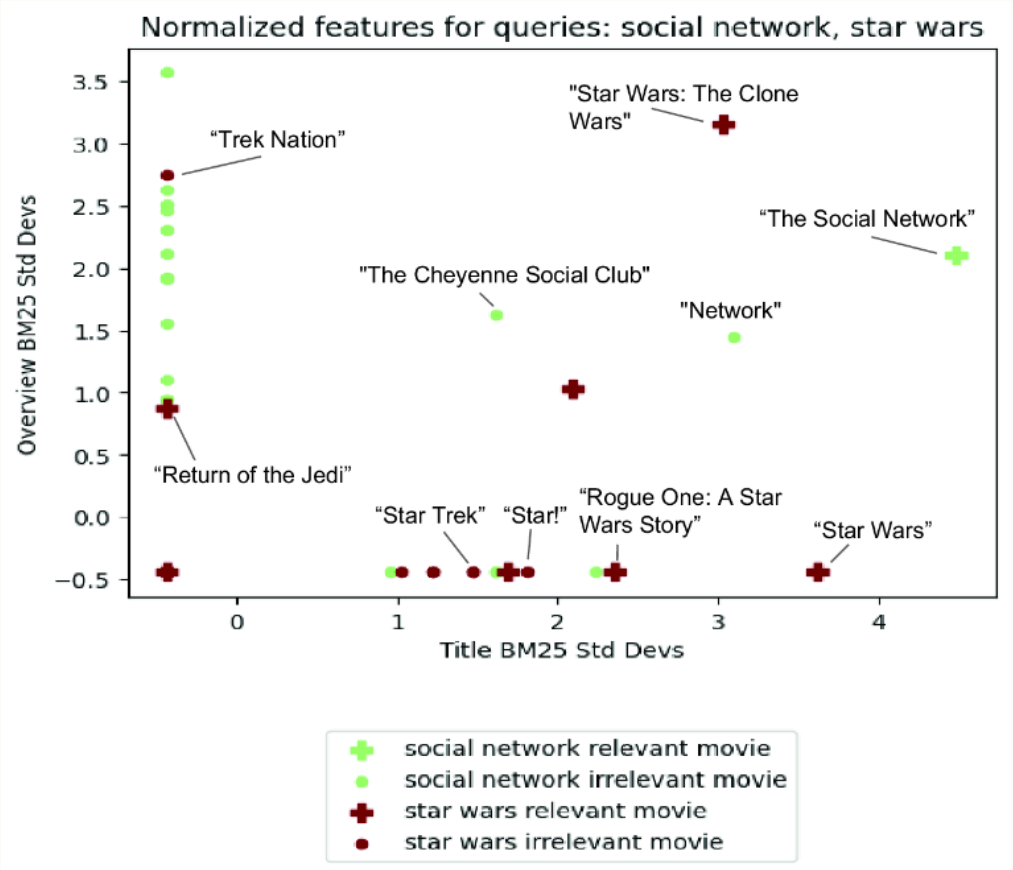

Figure 10.7 Normalized `star wars` and `social network` graded movies. Each increment in the graph is a standard deviation above or below the mean.

Next, we'll turn ranking into a binary classification learning problem to separate the relevant from irrelevant results.

SECOND, COMPUTE THE PAIRWISE DIFFERENCES

With normalized data, we've forced features to a consistent range. Now our SVM should not be biased by features that happen to have very large ranges. In this section, we're ready to transform the task into a binary classification problem, setting the stage for us to train our model.

SVMrank uses a pairwise transformation to reformulate LTR to a binary classification problem. *Pairwise* 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 pairwise algorithm, outlined in listing 10.9. The SVMrank algorithm takes every judgment for each query 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`. If the first judgment is less relevant, it is labeled with a `-1`. This pairwise transform algorithm yields training data (the `feature_deltas` and `predictor_deltas`) needed for binary classification.

Listing 10.9 Transforming features into pairwise data for SVMrank

```
for doc1_judgment in query_judgments:
  for doc2_judgment in query_judgments:
    j1_features = numpy.array(doc1_judgment.features)
    j2_features = numpy.array(doc2_judgment.features)

    if doc1_judgment.grade > doc2_judgment.grade:
      predictor_deltas.append(+1)  #1
      feature_deltas.append(j1_features -  #2
                            j2_features)  #2
    elif doc1_judgment.grade < doc2_judgment.grade:
      predictor_deltas.append(-1)  #3
      feature_deltas.append(j1_features - #4
                            j2_features)  #4
```

#1 Stores a label of +1 if doc1 is more relevant than doc2.
#2 Stores the feature deltas
#3 Stores a label of –1 if doc1 is less relevant than doc2.
#4 Stores the feature deltas

Figure 10.8 plots the pairwise differences and highlights important points.

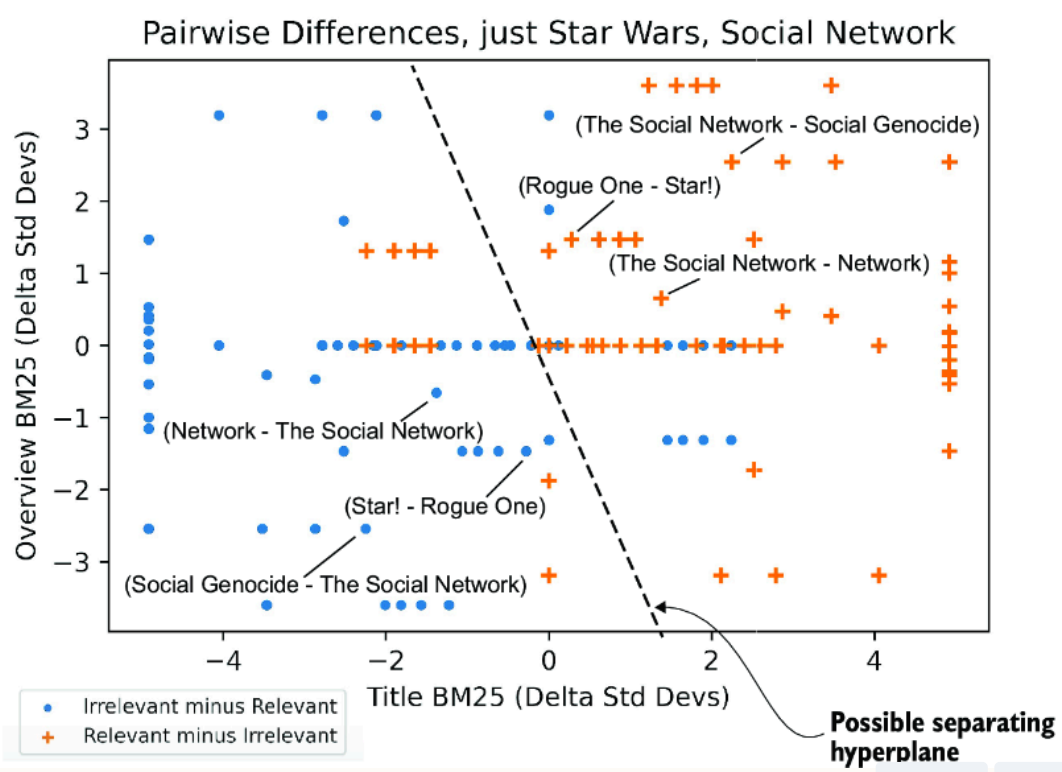

Figure 10.8 Pairwise differences after SVMrank's transformation for `social network` and `star wars` documents, along with a candidate separating hyperplane.

You'll notice that the positive pairwise deltas (+) tend to be toward the upper right. This means relevant documents have a higher `title_bm25` and `overview_bm25` when compared to irrelevant ones.

That's a lot to digest! Let's walk through a few examples carefully, step-by-step, to see how this algorithm constructs the data points in figure 10.9. This algorithm compares relevant and irrelevant documents for each query, comparing two documents (*Network* and *The Social Network*) within the query `social network` as shown in figure 10.9.

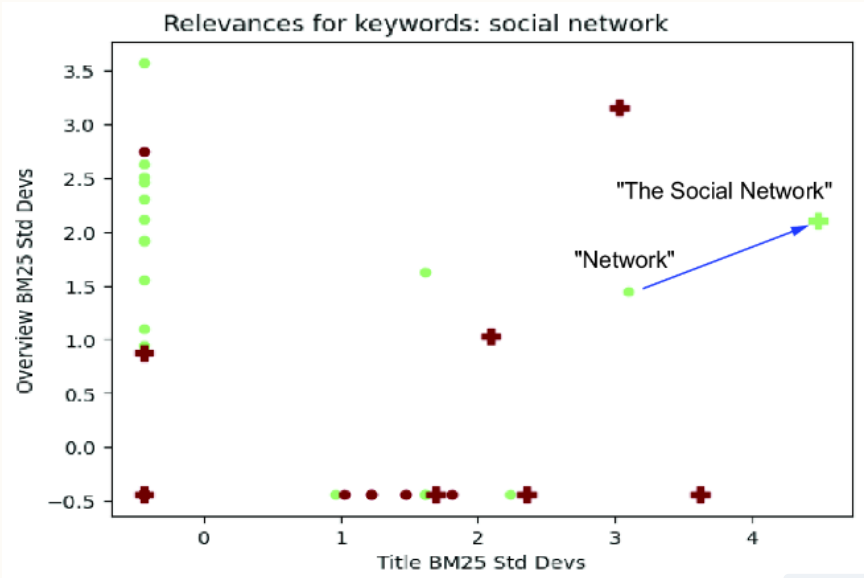

Figure 10.9 Comparing *Network* to *The Social Network* for the query `social network`

These are the features for *The Social Network*:

```
#[title_bm25, overview_bm25]  #1
[4.483, 2.100]  #1
```

#1 title_bm25 is 4.483 standard deviations above the mean, and overview_bm25 is 2.100 standard deviations above the mean.

These are the features for *Network*:

```
#[title_bm25, overview_bm25]  #1
[3.101, 1.443]  #1
```

#1 title_bm25 is 3.101 standard deviations above the mean, and overview_bm25 is 1.443 standard deviations above the mean.

We then insert the delta between *The Social Network* and *Network* in the following listing.

Listing 10.10 Calculating and storing the feature delta

```
predictor_deltas.append(+1)
feature_deltas.append([4.483, 2.100] - [3.101, 1.443]) #1
```

#1 Adds [1.382, 0.657] to feature_deltas

To restate listing 10.10, we might say that here is one example of a movie, *The Social Network*, that's more relevant than the movie *Network* for this query `social network`. Interesting! Let's look at what makes them different. Of course, "difference" in math means subtraction, which we'll do here. Ah yes, after taking the difference we see *The Social Network*'s `title_bm25` is `1.382` standard deviations higher than *Network*'s; similarly, the `overview_bm25` is `0.657` standard deviations higher. Indeed, note the `+` for *The Social Network* minus *Network* in figure 10.8 showing the point `[1.382, 0.657]` amongst the deltas.

The algorithm would also note that *Network* is less relevant than *The Social Network* for the query `social network`, as shown in figure 10.10.

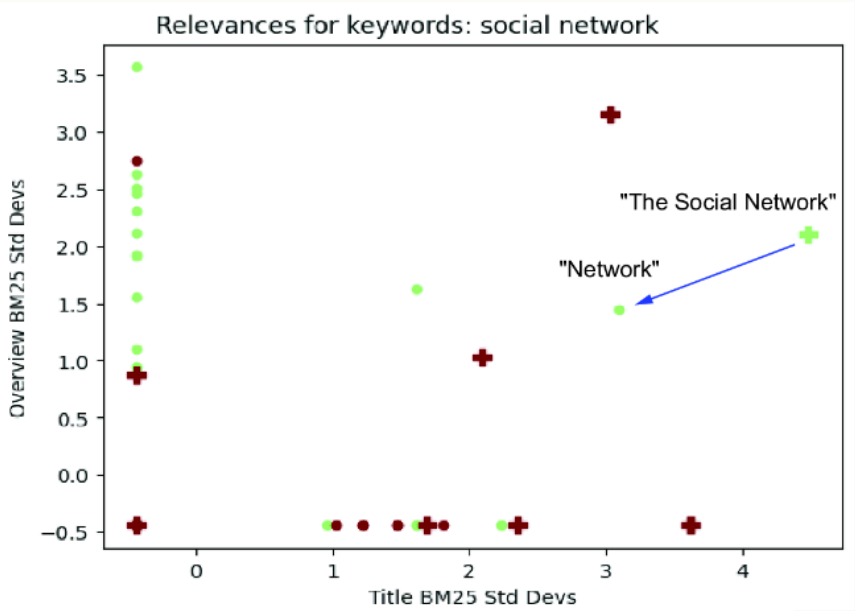

Figure 10.10 Comparing *Network* to *The Social Network* for the query `social network`

Just as in listing 10.9, our code captures this difference in relevance between these two documents, but this time in the opposite direction (irrelevant-minus-relevant). So it's no surprise that we see the same values, but in the negative.

```
predictor_deltas.append(-1)
feature_deltas.append([3.101, 1.443] - [4.483, 2.100])  #1
```

#1 Evaluates to [−1.382, −0.657]

In figure 10.11, we move on to another relevant-irrelevant comparison of two documents for the query `social network`, appending another comparison to the new training set.

Listing 10.11 shows appending both positive deltas (with the more relevant document listed first) and negative deltas (with the less relevant document listed first) for the highlighted pair of documents compared in figure 10.11.

Listing 10.11 Adding the positive and negative deltas

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append([4.483, 2.100] - [2.234, -0.444])  #1

# Negative example
predictor_deltas.append(-1)
feature_deltas.append([2.234, -0.444] - [4.483, 2.100])  #2
```

#1 Evaluates to [2.249, 2.544]
#2 Evaluates to [−2.249, −2.544]

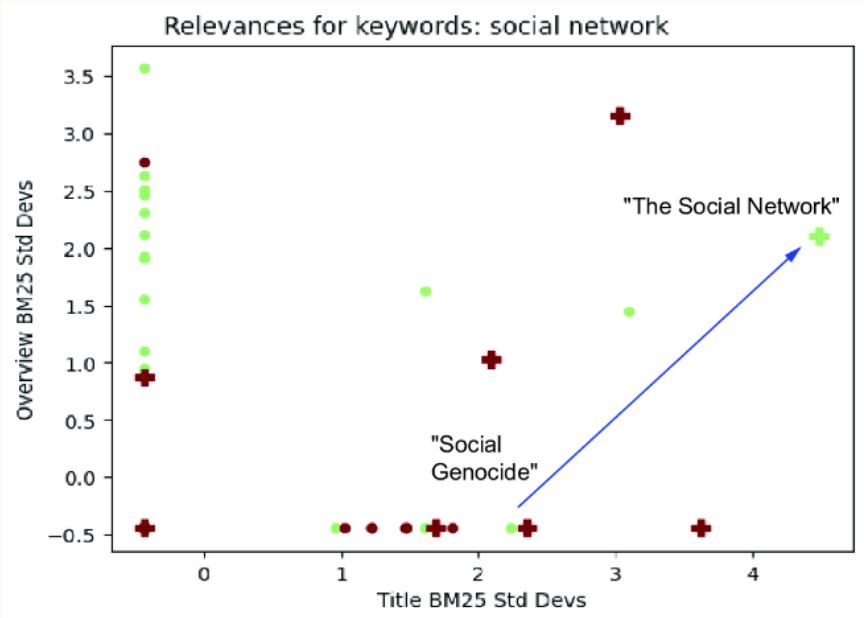

Figure 10.11 Comparing *Social Genocide* to *The Social Network* for the query `social network`

Once we iterate through every pairwise difference between documents matching the query `social network` to create a pointwise training set, we can move on to also logging differences for other queries. Figure 10.12 shows differences for a second query, this time comparing the relevance of documents matching the query `star wars`.

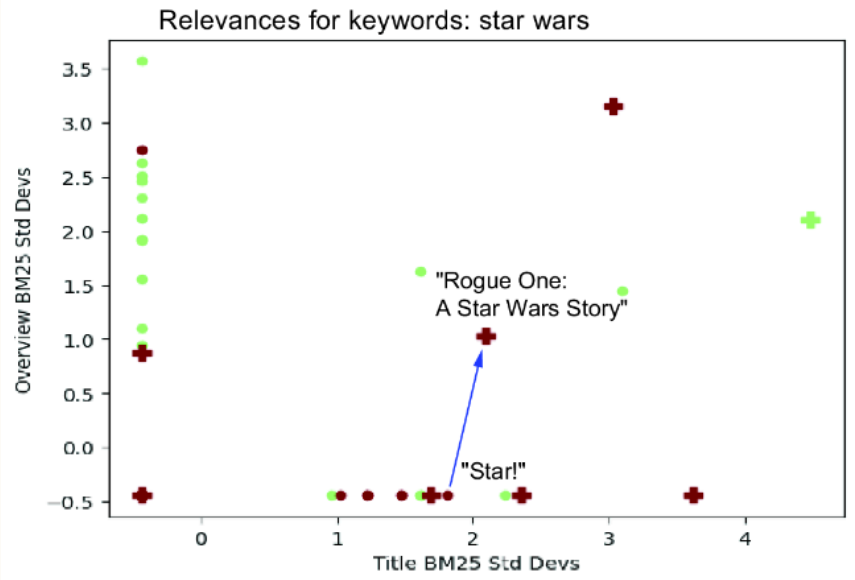

Figure 10.12 Comparing *Rogue One: A Star Wars Movie* to *Star!* for the query `star wars`. We've moved on from `social network` and have begun to look at patterns within another query.

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append([2.088, 1.024] - [1.808, -0.444])  #1

# Negative example
predictor_deltas.append(-1)
feature_deltas.append([1.808, -0.444] - [2.088, 1.024])  #2
```

#1 Rogue One features minus Star! features

#2 Star! features minus Rogue One features

We continue this process of calculating differences between feature values for relevant versus irrelevant documents until we have calculated all the pairwise differences for our training and test queries.

You can see back in figure 10.8 that the positive examples show a positive `title_bm25` delta, and possibly a slightly positive `overview_bm25` delta. This becomes even more clear if we calculate deltas over the full dataset of 100 queries, as shown in figure 10.13.

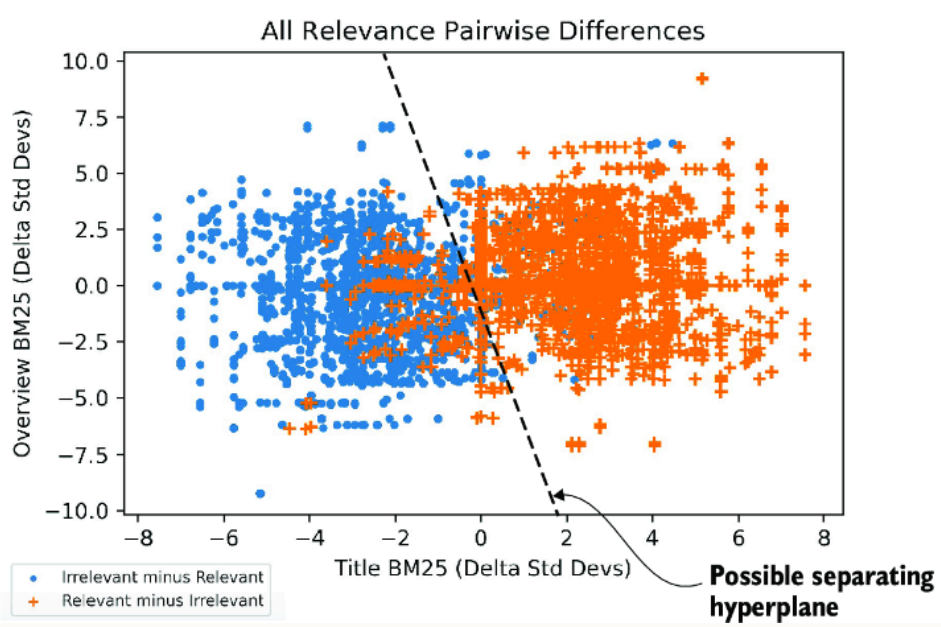

Figure 10.13 Full training set with a hyperplane separating relevant from irrelevant documents. We see a pattern! Relevant documents have a higher `title_bm25` and perhaps a modestly higher `overview_bm25`.

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 now and asking whether this formulation of ranking is appropriate for your domain. Different LTR models have their own method of mapping pairwise comparisons into classification problems as needed. As another example, LambdaMART—a popular LTR algorithm based on boosted trees—uses pairwise swapping and measures the change in *discounted cumulative gain* (DCG).

Next up, we'll train a robust model to capture the patterns in our fully transformed ranking dataset.

10.5 Step 4: Training (and testing!) the model

Good machine learning clearly requires a lot of data preparation. Luckily, you've 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 ranking classifier. This model will let us predict when documents might be relevant, even for queries and documents it hasn't seen yet.

10.5.1 Turning a separating hyperplane's vector into a scoring function

We've seen how SVMrank's separating hyperplane can classify and differentiate irrelevant examples from the relevant ones. That's useful, but you may remember that our task is to find *optimal* weights for our features, not just to classify documents. Let's therefore look at how we can *score* search results using this hyperplane.

It turns out that the separating hyperplane also gives us 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 its work, it gives us a sense of the weights that each feature should have, as shown in figure 10.14.

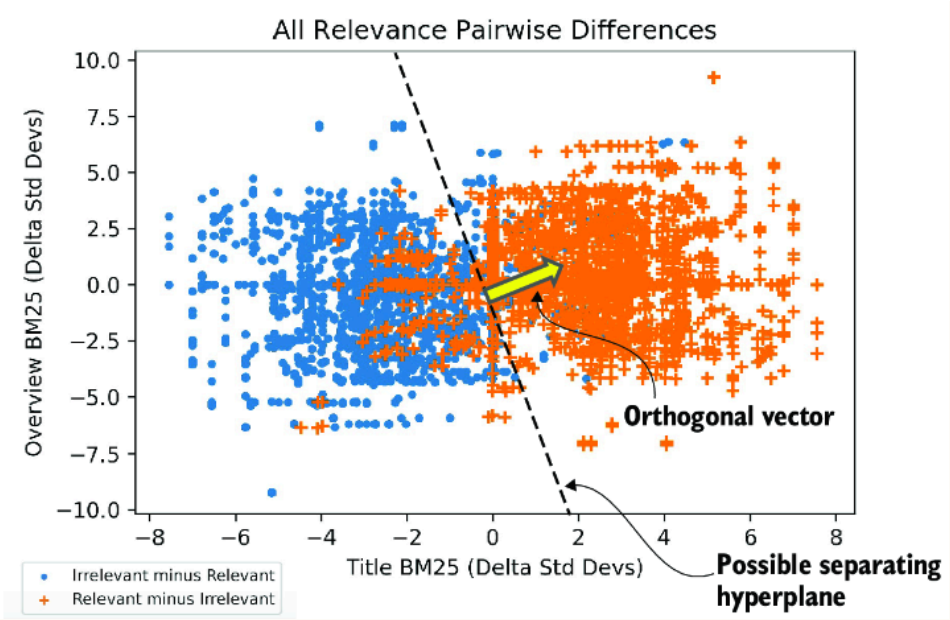

Figure 10.14 Full training set with a candidate-separating hyperplane, showing the orthogonal vector defining the hyperplane.

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 coming from `overview_bm25`. This vector might be something like:

```
[0.65, 0.40]
```

We used the pairwise transform algorithm in listing 10.9 to compute the deltas needed to perform classification between irrelevant and relevant examples. If we train an SVM on this data, as in the following listing, the model gives us the vector defining the separating hyperplane.

Listing 10.12 Training a linear SVM with scikit-learn

```
from sklearn import svm
model = svm.LinearSVC(max_iter=10000  #1
model.fit(feature_deltas, predictor_deltas)  #2
display(model.coef_) #3
```

#1 Creates a linear model with sklearn
#2 Fits to deltas using an SVM
#3 The vector that defines the separating hyperplane

Output:

```
array([0.40512169, 0.29006328, 0.14451715])
```

Listing 10.12 trains an SVM to separate the `predictor_deltas` (remember they're `+1` and `-1`) using the corresponding `feature_deltas` (the deltas in the 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 `ti-`

`tle_bm25`, a more modest one on `overview_bm25`, and a weaker weight on `release_year`.

10.5.2 Taking the model for a test drive

How does this model work as a ranking function? Let's suppose the user enters the query `wrath of khan`. How might this model score the document *Star Trek II: The Wrath of Khan* relative to this query? The unnormalized feature vector indicates a strong title and overview match for this query.

```
[5.9217176, 3.401492, 1982.0] #1
```

#1 Raw features for "Star Trek II"

Normalizing it, each feature value is this many standard deviations above or below each feature's mean:

```
[3.099, 1.825, -0.568]  #1
```

#1 Normalized features for "Star Trek II"

We simply multiply each normalized feature with its corresponding `coef_` value. Summing them then gives us a relevance score:

```
(3.099 * 0.405) + (1.825 * 0.290) + (-0.568 * 0.1445) = 1.702  #1
```

#1 Relevance score calculation for "Star Trek II"

How would this model rank *Star Trek III: The Search for Spock* relative to *Star Trek II: The Wrath of Khan* for our query `wrath of khan`? Hopefully not nearly as highly! Indeed, it doesn't:

```
[0.0, 0.0, 1984.0]  #1
[-0.432, -0.444, -0.468]  #2
(-0.432 * 0.405) + (-0.444 * 0.290) + (-0.468 * 0.1445) = -0.371  #3
```

#1 Raw features for "Star Trek III"
#2 Normalized features for "Star Trek III"
#3 Relevance calculation for "Star Trek III"

The model seems to be correctly predicting the most relevant answer.

10.5.3 Validating the model

Testing a couple of 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 traditional machine learning is that we usually evaluate queries and entire result sets, not individual data points, to prove our model is effective. We'll perform a test/training split at the query level. This 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 *K* (with `k=5` in our case) that are relevant. You should choose the relevance metric best suited to your own use case.

First, we will randomly put our queries into a test or training set, as shown in the following listing.

Listing 10.13 Simple test/training split at the query level

```
all_qids = list(set([j.qid for j in normed_judgments]))
random.shuffle(all_qids)  #1
proportion_train = 0.1  #1
  #1
split_idx = int(len(all_qids) * proportion_train)  #1
test_qids = all_qids[:split_index]  #2
train_qids = all_qids[split_index:]  #2

train_data = []; test_data=[]
for j in normed_judgments:
  if j.qid in train_qids:  #2
    train_data.append(j)  #2
  elif j.qid in test_qids:  #2
    test_data.append(j)  #2
```

#1 Identifies a random 10% of the judgments to go into the training set
#2 Places each judgment into training data (10%) or test set (90%)

With the training data split out, we can perform the pairwise transform trick from step 3. We can then retrain on just the training data.

Listing 10.14 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)  #1
display(model.coef_[0])
```

#1 Fits only to training data

Output:

```
array([0.37486809, 0.28187458, 0.12097921])
```

So far, 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 the training examples.

In the next listing, we evaluate our model using the test data. This code loops over every test query and ranks every test judgment using the model. It then computes the precision for the top four judgments.

Listing 10.15 Can our model generalize beyond the training data?

```
def score_one(features, model):
  score = 0.0
  for idx, f in enumerate(features):
    this_coef = model.coef_[0][idx].item()
    score += f * this_coef
  return score

def rank(query_judgments, model):
  for j in query_judgments:
    j.score = score_one(j.features, model)
  return sorted(query_judgments, key=lambda j: j.score, reverse=True)

def evaluate_model(test_data, model, k=5):
  total_precision = 0
  unique_queries = groupby(test_data, lambda j: j.qid)
  num_groups = 0
  for qid, query_judgments in unique_queries: #1
    num_groups += 1
    ranked = rank(list(query_judgments), model)  #2
    total_relevant = len([j for j in ranked[:k]
                          if j.grade == 1])  #3
    total_precision += total_relevant / float(k)
  return total_precision / num_groups

evaluation = evaluate_model(test_data, model)
print(evaluation)
```

#1 For each test query
#2 Scores each judgment and ranks this query using the model
#3 Compute the precisions for this query

Evaluation:

```
0.36
```

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 back at figure 10.8, the third 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 property within the TMDB dataset that we could experiment with.

For now, though, let's take the model we just built and see how we can deploy it to production.

10.6 Steps 5 and 6: Upload a model and search

In this section, we'll finally upload our model so that it can be applied to rank future search results. We'll then discuss both applying the model to rank all documents, as well as applying it to rerank an already-run and likely more efficient initial query. Finally, we'll discuss some of the performance implications of using LTR models in production.

10.6.1 Deploying and using the LTR model

Originally, we presented our objective as finding *ideal* boosts for a hardcoded ranking function like the one in listing 10.2:

```
{"query": f"title:({keywords})^10 overview:({keywords})^20
↪{!func}release_year^0.01"}
```

This boosted query indeed multiplies each feature by a weight (the boost) and sums the results. But it turns out that we don't want the search engine to multiply the *raw* feature values. Instead, we need the feature values to be normalized.

Many search engines let us store a linear ranking model along with feature normalization statistics. We saved the `means` and `std_devs` of each feature, which will be used to normalize values for any document being evaluated. These coefficients are associated with each feature when uploading the model, as shown in the next listing.

Listing 10.16 Generating and uploading a linear model

```
model_name = "movie_model"
feature_names = ["title_bm25", "overview_bm25", "release_year"]
linear_model = ltr.generate_model(model_name, feature_names,
                                  means, std_devs, model.coef_[0])
response = ltr.upload_model(linear_model)
display(linear_model)
```

Generated linear model (for `engine=solr`):

```
{"store": "movies",  #1
 "class": "org.apache.solr.ltr.model.LinearModel",
 "name": "movie_model",
 "features": [
   {"name": "title_bm25",  #2
    "norm": {"class": "org.apache.solr.ltr.norm.StandardNormalizer",
             "params": {"avg": "0.7245440735518126",  #3
                        "std": "1.6772600303613545"}}},  #3
   {"name": "overview_bm25",
    "norm": {"class": "org.apache.solr.ltr.norm.StandardNormalizer",
             "params": {"avg": "0.6662927508611409",
                        "std": "1.4990448120673643"}}},
   {"name": "release_year",
    "norm": {"class": "org.apache.solr.ltr.norm.StandardNormalizer",
             "params": {"avg": "1993.3349740932642",
                        "std": "19.964916628520722"}}}],
 "params": {
   "weights": {
     "title_bm25": 0.3748679655554891,  #4
     "overview_bm25": 0.28187459845467566,  #4
     "release_year": 0.12097924576841014}}}  #4
```

#1 Feature store to locate the features
#2 Which feature to execute before evaluating this model
#3 How to normalize this feature before applying the weight
#4 The weight of each feature in the model

The `response` from listing 10.16 is Solr-specific and will change depending on which search engine you have configured. Next, we can issue a search using the uploaded LTR model, as shown in the following listing.

Listing 10.17 Ranking all documents with LTR model for `harry potter`

```
request = {"query_fields": ["title", "overview"],
           "return_fields": ["title", "id", "score"],
           "rerank_query": "harry potter",
           "log": True}
response = ltr.search_with_model("movie_model", **request)
display(response["docs"])
```

Engine-specific search request (for `engine=solr`):

```
{"fields": ["title", "id", "score"],
 "limit": 5,
 "query": "{!ltr reRankDocs=9999999  #1
 ↪model=movie_model efi.keywords=\"harry potter\"}"}
```

#1 Executes our model over the maximum number of documents with the specified parameters

Returned documents:

```
[{"id": "570724", "title": "The Story of Harry Potter", "score": 2.4261155},
 {"id": "116972", "title": "Discovering the Real World of Harry Potter",
  "score": 2.247846},
 {"id": "672", "title": "Harry Potter and the Chamber of Secrets",
  "score": 2.017499},
 {"id": "671", "title": "Harry Potter and the Philosopher's Stone",
  "score": 1.9944705},
 {"id": "54507", "title": "A Very Potter Musical",
  "score": 1.9833609}]
```

In listing 10.17, the LTR model ranks all the documents in the corpus using the keywords in the `rerank_query` parameter as input to the model. Since no initial `query` parameter is specified in the request, no matching filter is applied to the collection before the search results (all documents) are ranked by the LTR model. Though scoring such a large number of documents with the model will lead to nontrivial latency, it allows us to test the model directly, absent of any other matching parameters.

Notice in listing 10.17 the use of the term "rerank" in the `rerank_query` parameter. As this term implies, LTR usually happens as a second ranking phase on results first calculated by a more efficient algorithm (such as BM25 and/or an initial Boolean match). This is to reduce the number of documents that must be scored by the more expensive LTR model. The following listing demonstrates executing a baseline search and then reranking the top `500` results with the LTR model.

Listing 10.18 Searching for `harry potter` and reranking with the model

```
request = {"query": "harry potter",
           "query_fields": ["title", "overview"],
           "return_fields": ["title", "id", "score"],
           "rerank_query": "harry potter",
           "rerank_count": 500,
           "log": True}
response = ltr.search_with_model("movie_model", **request)
display(response["docs"])
```

Engine-specific search request (for `engine=solr`):

```
{"query": "harry potter", #1
 "fields": ["title", "id", "score"],
 "limit": 5,
 "params": {
   "rq": "{!ltr reRankDocs=500 model=movie_model  #2
   ↪efi.keywords=\"harry potter\"}", #2
   "qf": ["title", "overview"], #1
   "defType": "edismax"}}  #1
```

#1 First-pass Solr query—a simple keyword query with BM25 ranking
#2 Reranks only the top 500 documents

Returned documents:

```
[{"id": "570724", "title": "The Story of Harry Potter", "score": 2.4261155},
 {"id": "116972", "title": "Discovering the Real World of Harry Potter",
  "score": 2.247846},
 {"id": "672", "title": "Harry Potter and the Chamber of Secrets",
  "score": 2.017499},
 {"id": "671", "title": "Harry Potter and the Philosopher's Stone",
  "score": 1.9944705},
 {"id": "54507", "title": "A Very Potter Musical", "score": 1.9833605}]
```

This request is much faster, and it still yields the same top results when performing the cheaper initial BM25 ranking on the filtered `query` followed by the more expensive LTR-based reranking on just the top `500` results.

10.6.2 A note on LTR performance

As you can see, many steps are required to build a real-world LTR model. Let's close the chapter with some additional thoughts on practical performance constraints in LTR systems:

- *Model complexity* —The more complex the model, the more accurate it *might* be. A simpler model can be faster and easier to understand, 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? Would the complexity be worth it? Would it be as generalizable (or could it possibly be more generalizable)?
- *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 compute cycles your model spends scoring results in your live search engine cluster.
- *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.
- *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 dozens. You will almost always see diminishing returns for relevance ranking and rising compute and latency costs as you continue adding additional features, so prioritizing which features to include is important.

CROSS-ENCODERS

A cross-encoder is a specialized kind of machine-learned ranking model. Cross-encoders are trained to score the relevance of two pieces of input (usually text), such as a query and a document. They use a Transformer architecture to combine both pieces of input into a single representation, which is then used in search to rank the relevance of the document for the query based upon interpreting both the query and document within their shared semantic context. Cross-encoders are ranking classifiers, like other LTR models, but they are unique in that they are pretrained on a large amount of data and are generally only focused on the textual similarity between the query and document instead of other features like popularity, recency, or user behavior. While they can be fine-tuned on your dataset, they are often used as is, since they are already trained on a large amount of data and can generalize well to new textual inputs.

Cross-encoders are very easy to use out of the box, and they're often the easiest way to get started with machine-learned ranking without having to do your own training. Cross-encoders tend to be slow, so they're not typically used to rerank large numbers of documents. Our focus in this chapter and in the coming chapter is on more flexible models that can use reflected intelligence, including those trained on your users' judgments and implicit judgments from user signals, but it's good to be familiar with cross-encoders, as they are a popular choice for many search teams, particularly when just getting started. We'll cover cross-encoders in more detail, with example code, in section 13.7.

10.7 Rinse and repeat

Congrats! You've done one full cycle of LTR! 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.

On your second iteration, you might consider the following:

- *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?
- *Training data coverage of all features*—The flip side of more features is more training data. As you increase the features you'd like to try, you should be wondering whether your training data has enough examples of relevant and irrelevant documents across each different combination of your features. Otherwise, your model won't know how to use features to solve the problem.
- *Different model architectures*—We used a relatively simple model that expects features to linearly and independently correlate with relevance, but relevance can often be nonlinear and multidimensional. A shopper searching for `ipad` might expect the most recent Apple iPad release, except when they add the word "cable", making the query `ipad cable`. For that query, the shopper might just want the cheapest cable they can find instead of the most recent. In this case, there may be "recency" and "price" features that activate

depending on specific keyword combinations, necessitating a more complicated model architecture.

In the next chapter, we will focus on the foundation of good LTR: great judgments!

Summary

- Learning to rank (LTR) builds generalized ranking functions that can be applied across all searches, using robust machine learning techniques.
- LTR features generally correspond to search queries. Search engines that support LTR often let you store and log features for use when training, and later applying, a ranking model.
- We have tremendous freedom in what features we use to generalize relevance. Features could be properties of queries (like the number of terms), properties of documents (like popularity), or relationships between queries and documents (like BM25 or other relevance scores).
- To do LTR well and apply well-known machine learning techniques, we typically reformulate the relevance ranking problem into a traditional, pointwise machine learning problem.
- SVMrank creates simple linear weights on normalized feature values, a good first step on your LTR journey.
- To be truly useful, we need our model to generalize beyond what it's learned. We can confirm an LTR model's ability to generalize by setting some judgments aside in a test dataset and not using them during training. After training, we can then evaluate the model on that previously unseen test dataset to confirm the model's ability to generalize.
- Once an 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.

Previous chapter
< 9 Personalized search

Next chapter
11 Automating learning to rank with click models >