11 Automating learning to rank with click models

This chapter covers

- Automating learning to rank retraining from user behavioral signals (searches, clicks, etc.)
- Transforming user signals into implicit LTR training data using click models
- Overcoming user's tendency to click items higher in the search results, regardless of relevance
- Handling low-confidence documents with fewer clicks when deriving implicit judgments

In chapter 10, we went step by step through training a learning to rank (LTR) model. Like walking through the mechanics of building a car, we saw the underlying nuts and bolts of LTR model training. In this chapter, we'll treat the LTR training process as a black box. In other words, we'll step away from the LTR internals, instead treating LTR more like a self-driving car, fine-tuning its trip toward a final destination.

Recall that LTR relies on accurate training data in order to be effective. LTR training data describes how users expect search results to be optimally ranked; it provides the directions we'll input into our LTR self-driving car. As you'll see, determining what's relevant based on user interactions comes with many challenges. If we can overcome these challenges and gain high confidence in our training data, though, we can build *automated learning to rank*: a system that regularly retrains LTR to capture the latest user relevance expectations.

As training data is so central to automated LTR, the challenges become not "What model/features/search engine should we use?" but more fundamentally, "What do users want from search?", "How do we turn that into training data?", and "How do we know whether that training data is any good?". By improving our confidence in the answers to these questions, we can put LTR (re)training on autopilot, as shown in figure 11.1.

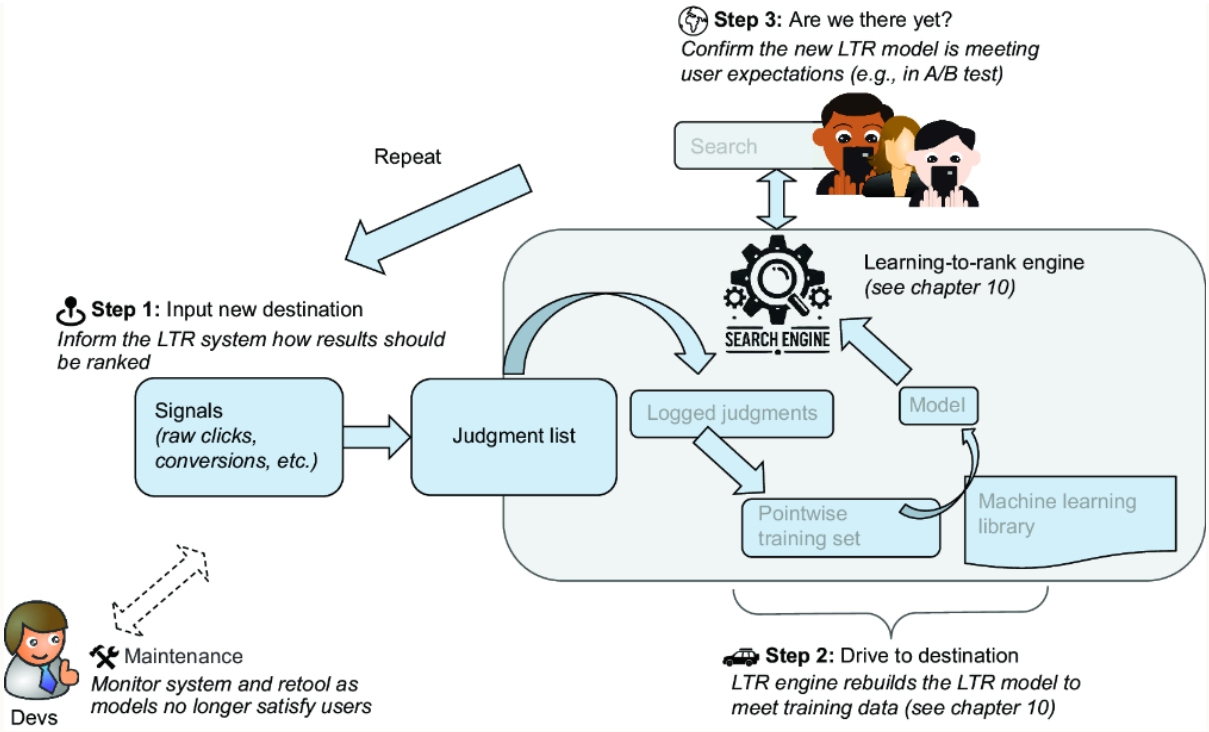

Figure 11.1 An automated LTR system automatically learns and retrains from the user's signals. This helps build models based on what actual users consider relevant over many queries.

Let's briefly walk through each step in the automated LTR process:

1. *Input new destination*—We input training data into the LTR system describing ideal relevance, based on our understanding of user behavioral signals, such as searches, clicks, and conversions (covered in this chapter).
2. *Drive to destination*—Our LTR system retrains an LTR model using the provided training data (as covered in chapter 10).
3. *Are we there yet?*—Is the model truly helping users? And should future models perhaps explore alternate routes (covered in chapter 12)?

Automated LTR repeats steps 1–3 continuously to automatically optimize relevance. The search team monitors the automated LTR's performance and intervenes as needed. This is the *maintenance* portion of figure 11.1. During maintenance, we open the hood to explore new LTR features and other model adjustments. Maintenance could also mean revisiting step 1 to correct our understanding of user behaviors and build more reliable, robust training data. After all, without good training data, we could follow chapter 10 to a T and still fail to satisfy our users.

This chapter starts our exploration of automated LTR by focusing on step 1—inputting a new destination. We'll first define the task of deriving training data from user clicks. We'll then spend the rest of this chapter overcoming some common biases and challenges with search click data. By the end of this chapter, you'll be able to build models with more reliable training data derived from user signals. Chapter 12 will then finish our automated LTR exploration by observing the model interacting with live users, using active learning and Gaussian techniques to overcome trickier presentation biases, and integrating all these components into a final, end-to-end automated LTR system.

11.1 (Re)creating judgment lists from signals

We mentioned that we need to overcome biases when creating LTR training data from clicks. However, before we dig into those biases, we'll explore the implications of using clicks instead of manual labels for LTR training data. We'll then take a naive, first stab at crafting training data in this section, reflecting on what went well or not so well. This will set us up for the rest of the chapter, where we'll explore removing bias from these results (in section 11.2 and beyond).

11.1.1 Generating implicit, probabilistic judgments from signals

Let's lay a foundation for how to use behavioral signals as LTR training data. Then we'll dive into the details of constructing reliable judgment lists.

In chapter 10, we discussed LTR training data, referred to as *judgment lists* or *judgments*. These judgements contain labels or *grades* for how relevant potential search results are for a given query. In chapter 10, we used movies as our example, labeling them with a grade of either `1` (relevant) or `0` (irrelevant), as in the following example.

Listing 11.1 Labeling movies relevant or irrelevant

```
# Judgment(grade, keywords, doc_id)
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
]
```

There are many techniques for generating judgment lists, and this isn't a comprehensive chapter on judgment lists and their many applications. Instead, we'll specifically focus on LTR training data. For this reason, we will only discuss judgments generated from user click signals. We call these *implicit judgments* because they derive from user interactions with the search application as users search and click. This contrasts with *explicit judgments*, where raters directly label search results as relevant/irrelevant.

Implicit judgments are ideal for automating LTR for several reasons:

- *Recency* —We have ready access to user traffic, so we can automate training today's LTR model on the latest user search expectations.
- *More data at less cost* —Setting up a task to capture explicit judgments, even with crowdsourcing, is time consuming and expensive to do well at scale. Deriving implicit judgments

from live user interactions we're already collecting allows us to use the existing user base to do this work for us.
- *Capturing real use cases*—Implicit judgments capture real users doing actual tasks with your search app. Contrast this with an artificial setting where explicit raters think carefully, perhaps unrealistically so, about the abstract task of choosing the most relevant results.

Unfortunately, click data can be noisy. We don't know why a user clicked on a given search result. Further, users are not homogeneous; some will interpret one result as relevant, while others will think otherwise. Search interactions also contain biases that need to be overcome, creating additional uncertainty around a model's calculations, which we'll discuss in detail later in this chapter and the next.

For these reasons, instead of a binary judgment, click models create *probabilistic judgments*. Instead of producing a grade of only `1` (relevant) or `0` (irrelevant), the grade represents the probability (between `0.0` and `1.0`) that a random user would consider the result to be relevant or not. For example, a good click model might restate the judgments from listing 11.1 as something more like the following.

Listing 11.2 Labeling movie query relevance probabilistically

```
# Judgment(grade, keywords, doc_id),
sample_judgments = [
  Judgment(0.99, "social network", 37799),  # The Social Network
  Judgment(0.01, "social network", 267752), # #chicagoGirl
  Judgment(0.01, "social network", 38408),  # Life As We Know It
  Judgment(0.01, "social network", 28303),  # The Cheyenne Social Club
  Judgment(0.99, "star wars", 11),          # Star Wars
  Judgment(0.80, "star wars", 1892),        # Return of the Jedi
  Judgment(0.20, "star wars", 54138),       # Star Trek Into Darkness
  Judgment(0.01, "star wars", 85783),       # The Star
  Judgment(0.20, "star wars", 325553)       # Battlestar Galactica
]
```

Notice the Star Wars movies in listing 10.2—the `grade` has become quite a bit more interesting. *Star Wars* now has a very high probability of relevance (`0.99`). The sequel, *Return of the Jedi*, has a slightly lower probability. Other science fiction movies (*Star Trek Into Darkness* and *Battlestar Galactica*) have ratings a bit higher than `0`, as fans of the Star Wars franchise might also enjoy these movies. *The Star* is completely unrelated—it's a children's animated movie about the first Christmas—so it receives a low `0.01` relevance probability.

11.1.2 Training an LTR model using probabilistic judgments

We just introduced the idea that a relevance grade could be probabilistic. Now let's consider how we can apply the lessons from chapter 10 to train a model using these probabilistic judgments (between `0.0` and `1.0`) instead of binary judgments.

Generally, you might consider these options when training a model:

- *Quantize the grades*—Quite simply, you can set arbitrary cutoffs before training to convert the grades to an acceptable format. You might assign a grade greater than `0.75` as rele-

vant (or `1.00`). Anything less than `0.75` would be considered irrelevant (or `0.00`). Other algorithms, like LambdaMART, accept a range of grades, like `1` to `4`, and these could have discrete cutoffs as well, such as assigning anything less than `0.25` a grade of `1.00`, anything greater than or equal to `0.25` but less than `0.5` a grade of `2.00`, and so on. With these algorithms, you could create 100 such labels, assigning `0.00` a grade of `0`, `0.01` a grade of `1`, and so on, until `1` is assigned a grade of `100` prior to training.

- *Just use the floating-point judgments*—The SVMRank algorithm from chapter 10 subtracted a more relevant item's features from a less relevant item's features (and vice versa) and built a classifier to tell relevant from irrelevant items. We did this with binary judgments, but nothing prevents us from doing this with probabilistic judgments. Here, if *Return of the Jedi* (`grade=0.80`) is considered more relevant than *Star Trek Into Darkness* (`grade=0.20`), we simply note *Return of the Jedi* as more relevant than *Star Trek Into Darkness* (labeling the difference as `+1`). Then we perform the same pairwise subtraction we would perform from chapter 10, subtracting features of *Star Trek Into Darkness* from those of *Return of the Jedi* to create a full training example.

Retraining the model with judgments in this chapter would mostly repeat the code from chapter 10, so we'll instead focus on the mechanics of training a click model. We have included a notebook with a full end-to-end LTR training example (see section 11.4) that integrates the click model we'll arrive at by the end of this chapter into the LTR training process you already explored in chapter 10.

Time to get back to the code and see our first click model!

11.1.3 Click-Through Rate: Your first click model

Now that you've seen the judgments format that a click model generates and how this format can be integrated to train an LTR model, let's take a first, naive pass at building a click model. After that, we'll take a step back to focus on a more sophisticated, general-purpose click model, and we'll then explore some of the core biases inherent in processing query and click signals.

TIP If you'd like to take a deeper dive into this topic, we encourage you to read *Click Models for Web Search* by Chuklin, Markov, and Rijke (Springer, 2015).

To build our click model, we'll return to the RetroTech dataset, as it comes conveniently bundled with user click signals. From these signals, we've also reverse-engineered the kind of raw session data you need to build high-quality judgments. We'll make use of the `pandas` library to perform tabular computations on session data.

In the following listing, we examine a sample search session for the movie *Transformers Dark of The Moon*. This raw session information is your starting point—the bare minimum information needed to develop a judgment list from user signals.

Listing 11.3 Examining a search session

```
query = "transformers dark of the moon"
sessions = get_sessions(query)  #1
print(sessions.loc[3]) #2
```

#1 Selects sessions for the "transformers dark of the moon" query
#2 Examines a single search session shown to the user

Output:

```
sess_id  query                          rank    doc_id          clicked
3        transformers dark of the moon  0.0     47875842328     False
3        transformers dark of the moon  1.0     24543701538     False
...
3        transformers dark of the moon  7.0     97360810042     True
...
3        transformers dark of the moon  13.0    47875841406     False
3        transformers dark of the moon  14.0    400192926087    False
```

Listing 11.3 corresponds to a single search session, with `sess_id=3`, for the query `transformers` `dark` `of` `the` `moon`. This session includes the query, the ranked results seen by the user, and whether each result was clicked. These three elements are the core ingredients needed to build a click model.

Search sessions will frequently differ. Another session, even seconds later, could have a slightly different ranking presented to the user. The search index might have changed, or a new relevance algorithm may have been deployed to production. We encourage you to retry listing 11.3 with another `sess_id` to compare sessions.

Let's convert this data into judgments using our first, simple click model: Click-Through Rate.

BUILDING JUDGMENTS FROM CLICK-THROUGH RATE

We'll start by building a very simple click model to get comfortable with the data, and then we can step back to see the flaws in this first pass. This will allow us to think carefully about the quality of the generated judgments for automated LTR in the rest of this chapter.

Our first click model will be based on *click-through rate* (CTR). CTR is the number of clicks received on a search result divided by the number of times it appeared in search results. If a result is clicked every single time the search engine returns the result, the CTR will be `1`. If it's never clicked, the CTR will be `0`. Sounds simple enough—what could go wrong?

We can look over every result for the query `transformers` `dark of the moon` and consider clicks with respect to the number of sessions in which the `doc_id` was returned. The following listing shows the computation and the resulting CTR value per document.

Listing 11.4 Computing CTR

```
def calculate_ctr(sessions):
  click_counts = sessions.groupby("doc_id")["clicked"].sum()
  sess_counts = sessions.groupby("doc_id")["sess_id"].nunique()
  ctrs = click_counts / sess_counts
  return ctrs.sort_values(ascending=False)

query = "transformers dark of the moon"
sessions = get_sessions(query, index=False)
click_through_rates = calculate_ctr(sessions)
print_series_data(click_through_rates, column="CTR")
```

Output:

```
doc_id         CTR       name
97360810042    0.0824    Transformers: Dark of the Moon - Blu-ray Disc
47875842328    0.0734    Transformers: Dark of the Moon Stealth Force E...
47875841420    0.0434    Transformers: Dark of the Moon Decepticons - N...
...
93624956037    0.0082    Transformers: Dark of the Moon - Original Soun...
47875841369    0.0074    Transformers: Dark of the Moon - PlayStation 3
24543750949    0.0062    X-Men: First Class - Widescreen Dubbed Subtitl...
```

In listing 11.4, for all `sessions` with the query `transformers dark of the moon` (per listing 11.3), we sum the clicks for each `doc_id` as `click_counts`. We also count the number of unique sessions for that document in `sess_counts`. Finally, we compute `ctrs` as `click_counts` / `sess_counts`, giving us our first click model. We see that document 97360810042 has the highest CTR and 24543750949 the lowest.

The preceding listing outputs the *ideal search results* based on the CTR. That is, if our LTR model was trained using this CTR click model to provide the relevance judgments, the search engine would produce this ordering as the optimal ranking. Throughout this chapter and the next, we'll frequently visually display this ideal ranking to understand whether the click model builds reasonable training data (judgments). We can see the CTR-based ideal judgments for `transformers` `dark` `of` `the moon` in figure 11.2.

Click-Through Rate judgments for q=transformers dark of the moon

	ctr	upc	image	name
0	0.0824	97360810042		Transformers: Dark of the Moon - Blu-ray Disc
1	0.0734	47875842328		Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii
2	0.0434	47875841420		Transformers: Dark of the Moon Decepticons - Nintendo DS
3	0.0364	24543701538		The A-Team - Widescreen Dubbed Subtitle AC3 - Blu-ray Disc
4	0.0352	25192107191		Fast Five - Widescreen - Blu-ray Disc

Figure 11.2 Search results ranked by CTR for the query `transformers dark of the moon`

Examining the results of figure 11.2, a couple of things jump out:

- The CTR for our top result (the Blu-ray of the movie *Transformers: Dark of the Moon*) seems rather low (`0.0824`, only a little better than the next judgment at `0.0734`). We might expect the Blu-ray's relevance grade to be much higher than other results.
- The DVD for the movie *Transformers: Dark of The Moon* doesn't even show up. It sits far below seemingly unrelated movies and secondary video games about the movie *Dark of The Moon*. We would expect the DVD to rank higher, maybe as high or higher than the Blu-ray.

But perhaps `transformers dark of the moon` is just a weird query. Let's repeat the process for something completely unrelated, this time for `dryer` in figure 11.3.

Click-Through Rate judgments for q=dryer

	ctr	upc	image	name
0	0.1608	84691226727		GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White
1	0.0816	84691226703		Hotpoint - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White-on-White
2	0.0710	12505451713		Frigidaire - Semi-Rigid Dryer Vent Kit - Silver
3	0.0576	783722274422		The Independent - Widescreen Subtitle - DVD
4	0.0572	883049066905		Whirlpool - Affresh Washer Cleaner

Figure 11.3 Search results ranked by CTR for the query `dryer`. Here we note the strange result for the movie *The Independent* that doesn't seem relevant.

In figure 11.3 we see other odd-looking results:

- The first two results are clothes dryers, which seems good.
- Following the clothes dryers are clothes-dryer parts. Hmm, OK?
- A movie called *The Independent* shows up. This seems completely random. Why would this be rated so highly?
- Next there's a washer accessory, which is kind of related.
- Finally, we see hair dryers, which shows another potential meaning of the word "dryer".

What do you think of the judgments produced by the CTR click model? Think back to what you learned in chapter 10. Remember this is the foundation, the very target, of your LTR model. Do you think these judgments would lead to a good LTR model that would ultimately succeed if put into production?

We also encourage you to ask yourself a more fundamental question: how could we even tell if a judgment list is good? Our subjective interpretation could be as flawed as the data in a click model. We'll consider this more analytically in chapter 12. For this chapter, we'll let our instincts guide us to possible problems.

11.1.4 Common biases in judgments

We've seen so far that we can create probabilistic judgments—those with grades between 0.00 and 1.00—simply by dividing the number of clicks on a product by the number of times that product is returned by search. The output, however, seemed to be a bit wanting, as it included movies unrelated to the Transformers franchise. We also saw a movie placed in the search results for `dryer`!

It turns out that search click data is full of biases. Here, we'll briefly define what we mean by "bias" before exploring each of these biases in the RetroTech click data.

With click models, a *bias* is a reason that raw user click data can have nothing to do with the relevance of search results. Instead, biases define how clicks (or the lack of clicks) reflect user psychology, search user interface design, or noisy data. We can separate biases into two broad groups: nonalgorithmic and algorithmic biases. *Algorithmic biases* are those inherent in the ranking, display, and interaction with search results. *Nonalgorithmic biases* occur for reasons only indirectly related to search ranking.

Algorithmic biases can include the following:

- *Position bias*—Users click on higher-ranked results more than lower-ranked results.
- *Confidence bias*—Documents with little signal data influence judgments the same as documents with much more data.
- *Presentation Bias*—If search never surfaces particular results, users never click them, so the click model won't know whether they're relevant.

Nonalgorithmic biases, on the other hand, are biases like the following:

- *Attractiveness bias*—Some results appear attractive and generate clicks (perhaps due to better images or wording selection), but they turn out to be spammy or just irrelevant.
- *Performance bias*—Users give up on slow searches, get distracted, and end up not clicking anything or clicking only on the earliest-returned results.

Since this book is about *AI-powered* search, we will focus our discussion on *algorithmic* biases in search clickstream data. We'll cover position bias and confidence bias in this chapter. Presentation bias will be covered in chapter 12.

But nonalgorithmic biases matter too! Search is a complex ecosystem that goes beyond relevance rankings. If results are frequently clicked, but follow-up actions like sales or other conversions don't occur, it might not be a ranking problem—perhaps you have a problem with spammy products. Or you might have a problem with the product pages or checkout process. You may find yourself asked to improve "relevance" when the limiting factor is actually the user experience, the content, or the speed of search.

Now that we've reflected on our first click model, let's work to overcome the first bias.

11.2 Overcoming position bias

In the previous section, we saw our first click model in action: a simple CTR click model. This divided the number of times a product was clicked in search by the number of times it was returned in the top results. We saw that this was quite a flawed approach, noting numerous reasons it could be biased. Specifically, we pointed out position bias, confidence bias, and presentation bias as three of the algorithmic biases present in our click model. It's time to begin tackling those problems!

In this section, we'll focus on the first of those algorithmic biases, position bias, digging into the problem and working on a click model designed to overcome it.

11.2.1 Defining position bias

Position bias is present in most search systems. If users are shown search results, they tend to prefer highly ranked search results over lower ones, even when those lower results are in fact more relevant. Joachims, et al. in their paper "Evaluating the Accuracy of Implicit Feedback from Clicks and Query Reformulations in Web Search" (**www.cs.cornell.edu/people/tj/publications/joachims_etal_07a.pdf**) discuss several reasons for position biases to exist:

- *Trust bias* —Users trust that the search engine must know what it's doing, so they interact with higher results more.
- *Scanning behaviors* —Users examine search results in specific patterns, such as top-to-bottom, and often don't explore everything in front of them.
- *Visibility* —Higher ranked results are likely to be rendered on the user's screen, so users need to scroll to see the remaining results.

With these factors in mind, let's see if we can detect position bias in the RetroTech sessions.

11.2.2 Position bias in RetroTech data

How much position bias exists in the sessions in the RetroTech dataset? If we can quantify this, we can consider how exactly we can remedy this problem. Let's assess the bias quickly before we consider a new click model for overcoming these biases.

By looking at all sessions across all queries, we can compute an average CTR per rank. This will tell us how much position bias exists in the RetroTech click data. We do this in the following listing.

Listing 11.5 CTR per rank in search sessions across all queries

```
sessions = all_sessions()
num_sessions = len(sessions["sess_id"].unique())
ctr_by_rank = sessions.groupby("rank")["clicked"].sum() / num_sessions
print(ctr_by_rank)
```

Output:

```
rank
0     0.249727
1     0.142673
2     0.084218
3     0.063073
4     0.056255
5     0.042255
6     0.033236
7     0.038000
8     0.020964
9     0.017364
10    0.013982
```

You can see in listing 11.5 that users click higher positions more. The CTR of results at rank `0` is `0.25`, followed by `0.143` at rank `1`, and so on.

Further, we can see position bias when we compare the CTR judgments from earlier to the typical ranking for each product in a query. If position bias is present, then our judgment's ideal ranking will end up resembling the typical ranking shown to users. We can analyze this by averaging the rank of each document over every session to see where they appear.

The following listing shows the typical search results page for `transformers dark of the moon` sessions.

Listing 11.6 Examining ranking for `transformers` `dark` `of` `the` `moon`

```
def calculate_average_rank(sessions):
  avg_rank = sessions.groupby("doc_id")["rank"].mean()
  return avg_rank.sort_values(ascending=True)

sessions = get_sessions("transformers dark of the moon")
average_rank = calculate_average_rank(sessions)
print_series_data(average_rank, "mean_rank")
```

Output:

```
doc_id        mean_rank    name
400192926087    13.0526    Transformers: Dark of the Moon - Original Soun...
97363532149     12.1494    Transformers: Revenge of the Fallen - Widescre...
93624956037     11.3298    Transformers: Dark of the Moon - Original Soun...
...
25192107191      2.6596    Fast Five - Widescreen - Blu-ray Disc
24543701538      1.8626    The A-Team - Widescreen Dubbed Subtitle AC3 - ...
47875842328      0.9808    Transformers: Dark of the Moon Stealth Force E...
```

In listing 11.6, some documents, like 24543701538 and 47875842328, historically occur toward the top of the search results for this query. They will be clicked more due to position bias. The typical results page, shown in figure 11.4, overlaps quite a lot with the CTR rank from figure 11.2.

Typical search session for q=transformers dark of the moon

	mean_rank	upc	image	name
0	0.9808	47875842328		Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii
1	1.8626	24543701538		The A-Team - Widescreen Dubbed Subtitle AC3 - Blu-ray Disc
2	2.6596	25192107191		Fast Five - Widescreen - Blu-ray Disc
3	3.5344	47875841420		Transformers: Dark of the Moon Decepticons - Nintendo DS
4	4.4444	786936817218		Pirates of the Caribbean: On Stranger Tides - Blu-ray 3D

Figure 11.4 Typical search result page for the `transformers dark of the moon` query. Notice the irrelevant movies like *The A-Team* and *Fast Five* showing up. Also note the high ranking of the Wii game. The high position of these results and the fact that they get clicked more just by showing up higher in the list explains why the CTR model erroneously thinks these are relevant.

Unfortunately, CTR is primarily influenced by position bias. Users click on the odd movies in figure 11.4 because the search engine returns them highly for this query, not because they are relevant. If we train an LTR model just on CTR, we would be asking the LTR model to optimize for what users already see and interact with. We must account for position bias when automating LTR.

Next, let's see how we can overcome position bias in a more robust click model that compensates for position bias.

11.2.3 Simplified dynamic Bayesian network: A click model that overcomes position bias

You've seen the harm position bias can do in action! If we just use clicks directly, we will train our LTR model to reinforce the ranking already shown to users. It's time to introduce a click model that can overcome position bias. We'll start by defining an "examine", a key concept in modeling position bias. We'll then introduce one particular click model that uses this examine concept to adjust raw clicks to overcome position bias.

HOW CLICK MODELS OVERCOME POSITION BIAS WITH AN "EXAMINE" EVENT

The basic CTR calculation doesn't *really* account for how users scan search results. The user likely considers only a few results—biased by position—before deciding to click one or two. If we can capture which results users consciously consider before clicking, we might be able to overcome position bias. Click models do exactly this by defining the concept of an "examine". We'll explore this concept before building a click model that overcomes position bias.

What is an examine? You may be familiar with an *impression*—when a UI element is rendered on the visible part of a user's screen. In click models, we consider instead an *examine*, the probability that a search result was consciously considered by the user. As we know, users often fail to notice something right in front of their eyes. You may have even been that user! Figure 11.5 captures this concept, contrasting impressions with examines.

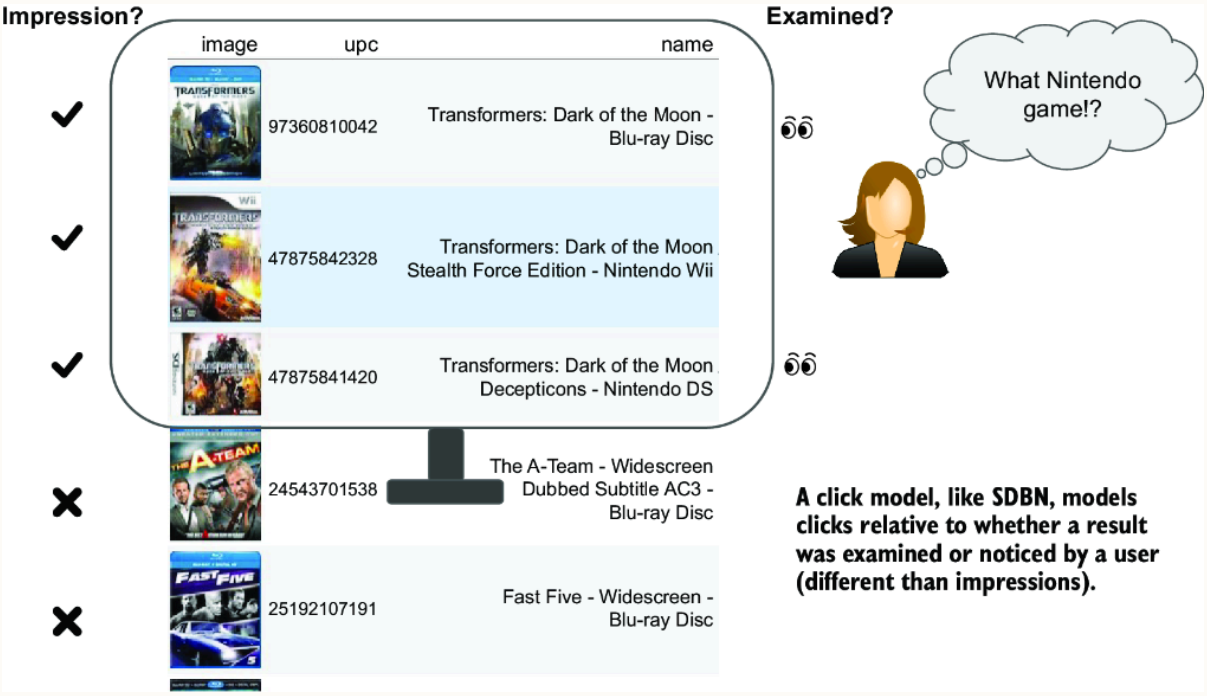

Figure 11.5 Impressions are whatever is rendered in the viewport (the monitor-shaped square) while examines are what the user considers (the results with eyeballs adjacent). Modeling what users examine helps correctly account for how users interact with search results.

You can see in figure 11.5 that the user fails to notice the Nintendo game in the second position, even though it's being rendered on their monitor. If the user didn't examine it, a click model shouldn't penalize the Nintendo game's relevance.

Why does tracking examines help overcome position bias? Examines are how a click model understands position bias. Another way of saying "position bias" is "we think that whether users examine search results depends on the position." As a result, modeling examines correctly is a core activity of most click models. Some click models, like the *position-based model* (PBM) attempt to determine an examine probability per position across all searches. Others, like the *cascading model* or, as we'll see soon, the *dynamic Bayesian network* (DBN) models, assume that if a result was above the last click on the search page, it likely was examined.

For most click models, the top position usually has a higher examine probability than lower ones. This allows click models to adjust for clicks correctly. Items examined frequently and clicked are rewarded and seen as more relevant. Those examined but not clicked are seen as less relevant.

To make this more concrete, let's dive deeper into one of the dynamic Bayesian network click models that uses examines to help overcome position bias.

DEFINING A SIMPLIFIED DYNAMIC BAYESIAN NETWORK

A *simplified dynamic Bayesian network* (SDBN) is a slightly less accurate version of the more complex dynamic Bayesian network (DBN) click model. These click models assume that,

within a search session, the probability that a user examined a document depends heavily on whether it was positioned at or above the lowest clicked document.
SDBN's algorithm first marks the last click of each session and then considers every document at or above this last click as examined. Finally, it computes a relevance grade by simply dividing the total clicks on a document by that document's total examines. We thus get a kind of dynamic CTR, tracking within each user's search session when they likely examined a result, and carefully using this to account for how that user evaluated its relevance. We then use these relevance evaluations in aggregate across sessions to train the SDBN click model.

Let's follow this algorithm step by step. We'll first mark the last click of each session in the following listing.

Listing 11.7 Marking which results were examined in each session

```
def calculate_examine_probability(sessions):
  last_click_per_session = sessions.groupby(  #1
    ["clicked", "sess_id"])["rank"].max()[True]  #1
  sessions["last_click_rank"] = last_click_per_session  #2
  sessions["examined"] = \  #3
    sessions["rank"] <= sessions["last_click_rank"]  #3
  return sessions

sessions = get_sessions("dryer")
probability_data = calculate_examine_probability(sessions).loc[3]
print(probability_data)
```

#1 Computes last_click_per_ session, the max rank where clicked is True per session.
#2 Marks the last click rank in each session
#3 Sets every position at or above the last click to True (otherwise False)

Output (truncated):

```
sess_id  query   rank    doc_id          clicked  last_click_rank  examined
3        dryer   0.0     12505451713     False    9.0              True
3        dryer   1.0     84691226727     False    9.0              True
3        dryer   2.0     883049066905    False    9.0              True
...
3        dryer   8.0     14381196320     True     9.0              True
3        dryer   9.0     74108096487     True     9.0              True
3        dryer   10.0    74108007469     False    9.0              False
3        dryer   11.0    12505525766     False    9.0              False
...
```

In listing 11.7, we find the max rank where `clicked` is `True` by storing it in `last_click_per_session`. We then mark positions at or above `last_click_rank` as examined in our sessions for `dryer`, as you can see in the output for `sess_id=3`.

With every session updated with examines set to `True` or `False`, we now sum the total clicks and examines per document across all sessions.

Listing 11.8 Sum clicks and examines per `doc_id` for this query

```
def calculate_clicked_examined(sessions):
  sessions = calculate_examine_probability(sessions)
  return sessions[sessions["examined"]] \
    .groupby("doc_id")[["clicked", "examined"]].sum()

sessions = get_sessions("dryer")
clicked_examined_data = calculate_clicked_examined(sessions)
print_dataframe(clicked_examined_data)
```

Output (truncated):

```
doc_id        clicked  examined    name
12505451713       355      2707    Frigidaire - Semi-Rigid Dryer Ve...
12505525766       268       974    Smart Choice - 6' 30 Amp 3-Prong...
...
36172950027        97       971    Tools in the Dryer: A Rarities C...
...
883049066905      286      2138    Whirlpool - Affresh Washer Cleaner
883929085118       44       578    A Charlie Brown Christmas - AC3 ...
```

In listing 11.8, `sessions[sessions["examined"]]` filters to examined rows only. Then, for each `doc_id`, we compute the total `clicked` and `examined` counts. You can see that some results, like `doc_id=36172950027`, clearly were examined a lot with relatively few clicks from users.

Finally, we finish the SDBN algorithm in the following listing by computing clicks over examines.

Listing 11.9 Compute final SDBN grades

```
def calculate_grade(sessions):
  sessions = calculate_clicked_examined(sessions)
  sessions["grade"] = sessions["clicked"] / sessions["examined"]
  return sessions.sort_values("grade", ascending=False)

query = "dryer"
sessions = get_sessions(query)
grade_data = calculate_grade(sessions)
print_dataframe(grade_data)
```

Output (truncated):

```
doc_id       clicked  examined  grade     name
856751002097     133       323  0.411765  Practecol - Dryer Balls (2-Pack)
48231011396      166       423  0.392435  LG - 3.5 Cu. Ft. 7-Cycle High-Ef...
84691226727      804      2541  0.316411  GE - 6.0 Cu. Ft. 3-Cycle Electri...
...
12505451713      355      2707  0.131141  Frigidaire - Semi-Rigid Dryer Ve...
36172950027       97       971  0.099897  Tools in the Dryer: A Rarities C...
883929085118      44       578  0.076125  A Charlie Brown Christmas - AC3 ...
```

In the output of listing 11.9, document 856751002097 is seen as the most relevant, with a grade of `0.4118`, or `133` clicks out of `323` examines.

Let's revisit our two queries to see how the ideal results now look for `dryer` and `transformers dark of the moon`. Figure 11.6 shows results for `dryer`, and figure 11.7 shows the `transformers dark of the moon` results.

SDBN judgments for q=dryer

	grade	upc	image	name
0	0.4118	856751002097		Practecol - Dryer Balls (2-Pack)
1	0.3924	48231011396		LG - 3.5 Cu. Ft. 7-Cycle High-Efficiency Washer - White
2	0.3164	84691226727		GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White
3	0.2938	74108007469		Conair - 1875-Watt Folding Handle Hair Dryer - Blue
4	0.2752	12505525766		Smart Choice - 6' 30 Amp 3-Prong Dryer Cord

Figure 11.6 Ideal search results for the query `dryer` according to SDBN. Notice how SDBN seems to promote more results related to washing clothes.

SDBN judgments for q=transformers dark of the moon

	grade	upc	image	name
0	0.6417	97360810042		Transformers: Dark of the Moon - Blu-ray Disc
1	0.4806	400192926087		Transformers: Dark of the Moon - Original Soundtrack - CD
2	0.3951	97363560449		Transformers: Dark of the Moon - Widescreen Dubbed Subtitle - DVD
3	0.3231	97363532149		Transformers: Revenge of the Fallen - Widescreen Dubbed Subtitle - DVD
4	0.2662	93624956037		Transformers: Dark of the Moon - Original Soundtrack - CD

Figure 11.7 Ideal search results for the query `transformers dark of the moon` according to SDBN. We've now surfaced the DVD, Blu-ray movie, and CD soundtrack.

If we subjectively examine figures 11.6 and 11.7, both sets of judgments appear more intuitive than the previous CTR judgments. In our `dryer` example, the emphasis appears to be on washing clothes. There are some accessories (such as the dryer balls) that score roughly the same as the dryers themselves.

For `transformers dark of the moon`, we note the very high grade for the Blu-ray movie. We also see the DVD and CD soundtrack ranking higher than other secondary "Dark of the Moon" items such as video games. Somewhat oddly, the soundtrack CD is ranked higher than the movie DVD—perhaps we should investigate this more.

Of course, as we've said earlier, we're using our intuition for now. In chapter 12, we'll think more objectively about how we might evaluate judgment quality.

With our position bias more under control, we'll now move on to fine-tune our judgments to handle another crucial bias: confidence bias.

11.3 Handling confidence bias: Not upending your model due to a few lucky clicks

In the game of baseball, a player's batting average tells us the proportion of hits they get for every at bat. A great professional player has a batting average greater than 0.3. Consider, however, a lucky little league baseball player stepping up to the plate for their first at bat

and getting a hit. Their batting average is technically 1.0! We can conclude, then, that this young child is a baseball prodigy and will certainly have a great baseball career. Right? Not quite! In this section, we're going to explore the relevance side of this lucky little leaguer. What do we do with results that, perhaps simply out of luck, have been examined only a few times, each resulting in a click? These likely shouldn't get a perfect grade of 1.0. We'll see (and correct) this problem in our data.

11.3.1 The low-confidence problem in click data

Let's look at the data to see where low-confidence data points are biasing the training data. We'll then see how we can compensate for low-confidence problems in the SDBN results. To define the problem, let's look at the SDBN results for `transformers dark of the moon` and another, rarer, query to see common low-confidence situations.

If you recall, it was a bit suspicious that the soundtrack CD for the *Transformers Dark of The Moon* movie ranked so highly according to SDBN. When we examine the raw data underlying the rankings, we can see a possible problem. In the following listing, we reconstruct the SDBN data for `transformers dark of the moon` to debug this problem, combining listings 11.7–11.9.

Listing 11.10 Recomputing SDBN statistics

```
query = "transformers dark of the moon"
sessions = get_sessions(query)
grade_data = calculate_grade(sessions)
print_dataframe(grade_data)
```

Output (truncated):

```
doc_id      clicked  examined  grade     name
97360810042     412       642  0.641745  Transformers: Dark of the Moon -...
400192926087     62       129  0.480620  Transformers: Dark of the Moon -...
97363560449      96       243  0.395062  Transformers: Dark of the Moon -...
...
47875841406      80       626  0.127796  Transformers: Dark of the Moon A...
24543750949      31       313  0.099042  X-Men: First Class - Widescreen ...
47875842335      53       681  0.077827  Transformers: Dark of the Moon S...
```

In the output of listing 11.10, note that the top result, the Blu-ray movie (`doc_id=97360810042`), has far more examines (`642`) than the soundtrack CD (`doc_id=400192926087` with `129` examines). The Blu-ray's grade is more reliable, given it has had many times more opportunities for user interaction, making it less likely to be dominated by noisy, spurious clicks. The CD, on the other hand, has far fewer examines. Shouldn't the Blu-ray's relevance grade be weighed higher, given that it's a more reliable data point compared to the CD with more limited data?

Often, this situation is even starker, particularly when dealing with less common queries. Regardless of the number of queries your search engine receives, some queries are likely to be received many times (*head queries*), some a moderate number of times (*torso queries*),

and some very rarely (*long-tail queries*, or simply *tail queries*). Consider the query `blue ray`. You'll note that this is a common misspelling of "Blu-ray". As a common mistake, it likely mixes documents with a modest number of examines with documents receiving very few. In the following listing, we compute the SDBN statistics for `blue ray`, which suffers from this data sparsity problem.

Listing 11.11 SDBN judgments for a query with sparse data

```
def get_sample_sessions(query):
  sessions = get_sessions(query, index=False)
  sessions = sessions[sessions["sess_id"] < 50050]  #1
  return sessions.set_index("sess_id")

sessions = get_sample_sessions("blue ray")
grade_data = calculate_grade(sessions)
print_dataframe(grade_data)
```

#1 Randomly samples a few sessions to simulate a typical long-tail case

Output (truncated):

```
doc_id        clicked  examined  grade      name
600603132872  1        1         1.000000   Blu-ray Disc Cases (10-Pack)
827396513927  14       34        0.411765   Panasonic - Blu-Ray Player
25192073007   8        20        0.400000   The Blues Brothers - Widescree...
...
25192107191   0        7         0.000000   Fast Five - Widescreen - Blu-r...
23942972389   0        15        0.000000   Verbatim - 10-Pack 6x BD-R Dis...
885170038875  0        5         0.000000   Panasonic - 9" Widescreen Port...
```

Looking at the output of listing 11.11, we see something unsettling. Like the most extreme case of our lucky little league baseball player, the most relevant result, doc 600603132872, receives a grade of `1.0` (perfectly relevant) after being examined by only one user! This grade of `1.0` ranks higher than the next result, which has a grade of `0.411` based on `34` examines. When you consider that doc 600603132872 is a set of Blu-ray cases and 827396513927 is a Blu-ray player, this feels more troubling. Our subjective interpretation might rank the player above the cases. Shouldn't the fact that the second result was examined significantly more count for something?

What we've seen in these examples is *confidence bias*—when a judgment list has many grades based on statistically insignificant, spurious events. We say these spurious events with few examines have low confidence, whereas those with more examines provide a higher level of confidence. No matter your click model, you likely have many situations where queries have only a modest amount of traffic. To automate LTR, you'll need to adjust your training data generation to account for the confidence you have in the data.

Now that you've seen the effect of low-confidence data, we can move on to some solutions you can apply when building your click model.

11.3.2 Using a beta prior to model confidence probabilistically

We've just seen a few problems created by valuing low-confidence data too highly. Without adjusting your models based on your confidence in the data, you won't be able to build a reliable automated LTR system. We could just filter these low-confidence examples out, but can we perhaps do something smarter? We'll discuss an approach to preserving all the clickstream data in this section as we introduce the concept of beta distributions. But first, let's discuss why using all data is generally preferred over simply filtering out the low-confidence examples.

SHOULD WE FILTER OUT LOW-CONFIDENCE JUDGMENTS?

In our click model, should we just remove the low-confidence examples? We don't generally recommend throwing data points away like that.

Filtering training data, such as data points below some minimum threshold of examines, reduces the amount of training data you have. Even with a reasonable threshold, documents for a query are typically examined on a power law distribution. Users examine some documents very frequently, while examining the vast majority very infrequently. A threshold can thus remove too many LTR examples and cause an LTR model to miss important patterns. Even with a threshold, you would be left with the challenge of how to weight medium-confidence examples against high-confidence ones, such as with the `transformers dark of the moon` query from earlier.

Instead of using a hard cutoff, we advocate keeping low-confidence examples and just weighing all examples based on confidence level. We'll do this using a beta distribution on the computed relevance grades. We'll then apply this solution to fix our SDBN click model judgments.

USING THE BETA DISTRIBUTION TO ADJUST FOR CONFIDENCE

Beta distributions help us draw conclusions from our clicks and examines based on probabilities instead of just biased occurrences. However, before we dive straight into using the beta distribution for judgments, let's first examine the usefulness of a beta distribution using our previous, intuitive baseball batting-average analogy.

In baseball, a batting average of 0.295 for a player means that when this player goes to bat, there's roughly a 29.5% chance they will get a hit. But if we wanted to know "What's the batting average for that player, batting in Fenway Park in September on rainy days", we'd probably have very little information to go on. The player may have only batted in those conditions a handful of times. Maybe they made 2 hits out of 3 tries in those conditions. We would conclude their batting average in these cases is 2/3 or 0.67. We know by now that this conclusion would be a mistake: do we really think, based on only 3 chances at bat, we can conclude that the player has an improbably high 66.7% chance of making a hit? A better approach would be to use the 0.295 general batting average as an initial belief, moving slowly away from that assumption as we gradually gain more data on "Fenway Park in September on rainy days" at bats.

The *beta distribution* is a tool used to manage beliefs. It turns a probability, like a batting average or judgment grade, into two values, `a` and `b`, that represent the probability as a dis-

tribution. The `a` and `b` values can be interpreted as follows:

- `a` *(the successes)*—The number of at bats with hits we've observed, or the number of examines with clicks
- `b` *(the failures)*—The number of at bats without hits we've observed, or the number of examines without clicks

With the beta distribution, the property `mean = a / (a + b)` holds, where `mean` is the initial point value, like a batting average. Given a `mean`, we can find many `a` and `b` values that satisfy `mean = a / (a + b)`. After all, `0.295 = 295 / (295 + 705)` as does `0.295 = 1475 / (1475 + 3525)` and so on. Yet each represents a different beta distribution. Keep this property in mind as we move along.

Let's put these pieces together to see how the beta distribution prevents us from jumping to conclusions on spurious click (or batting) data.

We could declare our initial belief about any document's relevance grade as `0.125`. This is like declaring the baseball player's batting average to be 0.295 as our initial belief of their performance. We can use the beta distribution to update the initial belief for specific cases like "Fenway Park in September on rainy days" or a specific document's relevance for a search query.

The first step is to pick an `a` and a `b` that capture our initial belief. For our relevance case, we could choose many values for `a` and `b` that satisfy `0.125 = a / (a + b)`. Suppose we choose `a=2.5, b=17.5` as our relevance belief on documents with no clicks. Graphing this, we would see the distribution in figure 11.8.

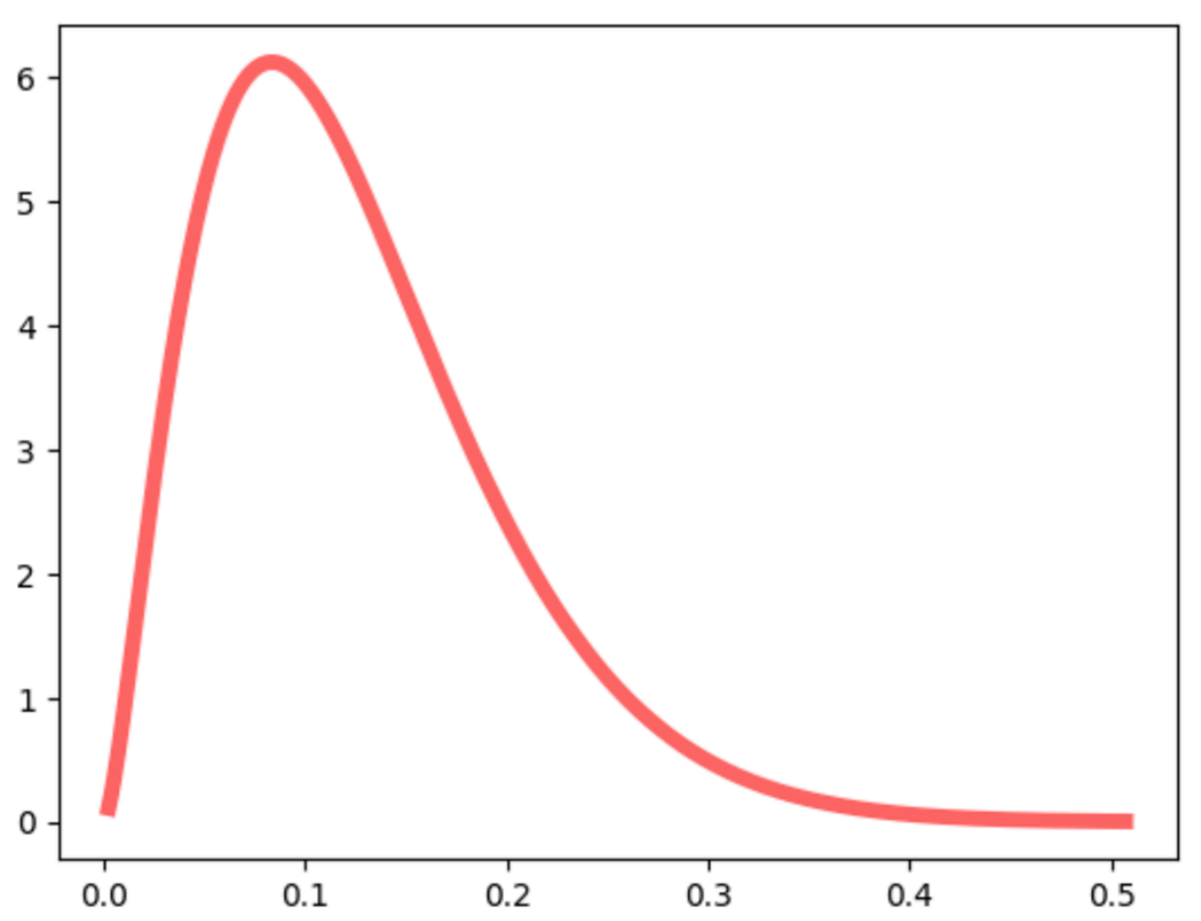

Figure 11.8 Beta distribution for a relevance grade of 0.125. The mean corresponds to our default relevance grade. We see the distribution of most likely relevance grades.

We can observe now what happens when we see a document's first click, incrementing that document's `a` to 3.5. In figure 11.9 we have `a=3.5, b=17.5`.

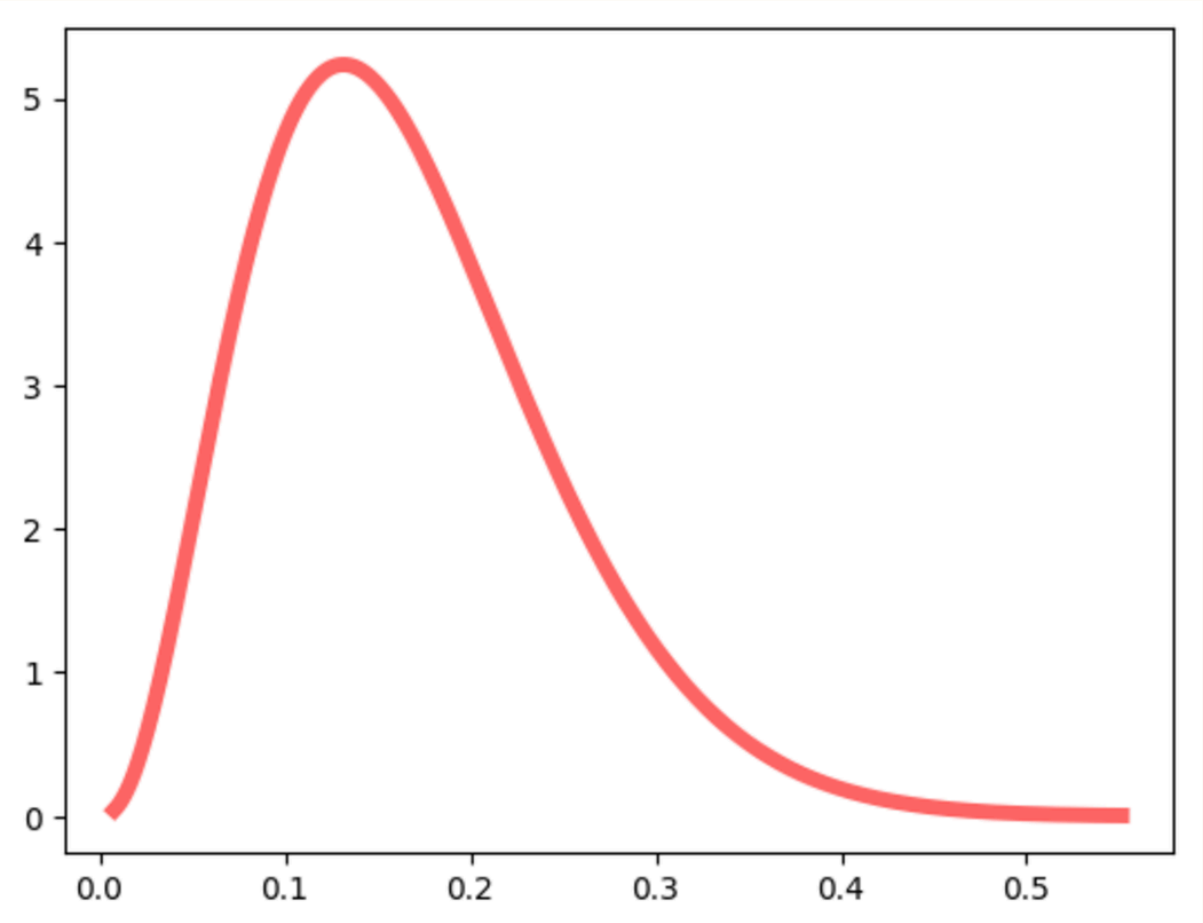

Figure 11.9 The beta distribution for a relevance grade after adding one click is now `0.1667`. Adding a click "pulls" the probability distribution a little toward one direction, updating the initial belief.

The mean relevance grade for the updated distribution is now `3.5 / (17.5 + 3.5)` or `0.1667`, effectively pulling the initial belief a little higher, given its first click. Without the beta distribution, this document would have 1 click and 1 examine, resulting in a grade of `1`.

We refer to the starting point probability distribution (the chosen `a` and `b`) as the *prior distribution*, or just a *prior*. This is our initial belief in what will happen. The distribution after updating `a` and `b` for a specific case, like a document, is a *posterior distribution*, or just a *posterior*. This is our updated belief.

Recall that we said earlier that many initial `a` and `b` values could be chosen. This has significance, as the magnitude of the initial `a` and `b` make our prior weaker or stronger. We could choose any value for `a` and `b` where `a / (a + b) = 0.125`. But note what happens if we choose a very small value `a=0.25, b=1.75`. Then we go to update `a` by incrementing it by 1. The new expected value of the posterior distribution is `1.25 / (1.25 + 1.75)` or ~`0.416`. That's a major effect for just one click. Conversely, using very high `a` and `b` values would make a prior so strong it would barely budge. When you use the beta distribution, you'll want to tune the magnitude of the prior so updates have the desired effect.

Now that you've seen this handy tool for capturing SDBN grades in practice, let's see how the beta distribution can help with our SDBN confidence problems.

USING A BETA PRIOR IN SDBN CLICK MODELS

Let's finish the chapter by updating the SDBN click model using the beta distribution. If you use other click models, like the ones alluded to earlier in this chapter, you'll need to reflect on how confidence can be solved in those cases. The beta distribution might be a useful tool there, as well.

If you recall, the output of SDBN was a count of `clicks` and `examines` for each document. In listing 11.12, we pick up from listing 11.11, which computed the SDBN for the `blue ray` query. We'll choose a prior grade of `0.3` for use with our SDBN model. This is our default grade when we don't have information about the document—possibly derived from the typical grade we see in our judgments. We'll then compute a prior beta distribution (`prior_a` and `prior_b`) using this prior grade.

Listing 11.12 Computing prior beta distribution

```
def calculate_prior(sessions, prior_grade, prior_weight):
  sessions = calculate_grade(sessions)
  sessions["prior_a"] = prior_grade * prior_weight  #1
  sessions["prior_b"] = (1 - prior_grade) * prior_weight  #1
  return sessions

prior_grade = 0.3  #2
prior_weight = 100  #3
query = "blue ray"
sessions = get_sample_sessions(query)
prior_data = calculate_prior(sessions, prior_grade, prior_weight)
print(prior_data)
```

#1 Resulting a and b satisfying prior_grade = prior_a / (prior_a + prior_b)
#2 Default prior relevance grade
#3 How much confidence to place in the prior (prior_weight = a + b)

Output (truncated):

```
doc_id          clicked  examined  grade     prior_a  prior_b
600603132872    1.0      1.0       1.000000  30.0     70.0
827396513927    14.0     34.0      0.411765  30.0     70.0
25192073007     8.0      20.0      0.400000  30.0     70.0
885170033412    6.0      19.0      0.315789  30.0     70.0
...
```

In listing 11.12, with a weight of `100`, you can confirm that `prior_grade = prior_a / (prior_a + prior_b)` or `0.3 = 30 / (30 + 70)`. This has captured an initial probability distribution for our prior.

In listing 11.13, we need to compute a posterior distribution and corresponding relevance grade. We do this by incrementing `prior_a` for clicks (our "successes"), and `prior_b` for examines with no clicks (our "failures"). Finally, we compute an updated grade as the `beta_grade`.

Listing 11.13 Computing posterior beta distribution

```
def calculate_sdbn(sessions, prior_grade=0.3, prior_weight=100):
  sessions = calculate_prior(sessions, prior_grade, prior_weight)
  sessions["posterior_a"] = (sessions["prior_a"] +  #1
                             sessions["clicked"])  #1
 sessions["posterior_b"] = (sessions["prior_b"] +    #2
    sessions["examined"] - sessions["clicked"])  #2
  sessions["beta_grade"] = (sessions["posterior_a"] /  #3
    (sessions["posterior_a"] + sessions["posterior_b"]))  #3
  return sessions.sort_values("beta_grade", ascending=False)

query = "blue ray"
sessions = get_sample_sessions(query)
bluray_sdbn_data = calculate_sdbn(sessions)
print(bluray_sdbn_data)
```

#1 Updates our belief about the document's relevance, increasing from prior_a by the number of clicks
#2 Updates our belief about the document's lack of relevance, increasing from prior_b by examines without clicks
#3 Computes a new grade from posterior_a and posterior_b

Output (truncated):

```
doc_id        cl  ex  grade  pr_a  pr_b  posterior_a posterior_b beta_grade
827396513927  14  34  0.411  30.0  70.0  44.0        90.0        0.328358
25192073007   8   20  0.400  30.0  70.0  38.0        82.0        0.316667
600603132872  1   1   1.000  30.0  70.0  31.0        70.0        0.306931
...
786936805017  1   14  0.071  30.0  70.0  31.0        83.0        0.271930
36725608511   0   11  0.000  30.0  70.0  30.0        81.0        0.270270
23942972389   0   15  0.000  30.0  70.0  30.0        85.0        0.260870
```

In the output of listing 11.13, the column headers `clicked`, `examined`, `prior a`, and `prior_b` were shortened for space to `cl`, `ex`, `pr_a`, and `pr_b`. Notice our new ideal results for the query `blue ray` by sorting on `beta grade`. The `beta grade` values remain closer to the prior grade of `0.3`. Notably, our Blu-ray cases have slid to the third most relevant slot, with the single click not pushing the grade much past `0.3`.

When we repeat this calculation of judgments for `dryer` and `transformers dark of the moon` in figures 11.10 and 11.11, we note that the order is the same, but the grades themselves stay closer to the prior of `0.3` depending on our confidence in the data.

Confidence-adjusted SDBN judgments for q=dryer

	beta_grade	upc	image	name
0	0.3853	856751002097		Practecol - Dryer Balls (2-Pack)
1	0.3748	48231011396		LG - 3.5 Cu. Ft. 7-Cycle High-Efficiency Washer - White
2	0.3158	84691226727		GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White
3	0.2946	74108007469		Conair - 1875-Watt Folding Handle Hair Dryer - Blue
4	0.2775	12505525766		Smart Choice - 6' 30 Amp 3-Prong Dryer Cord

Figure 11.10 Beta-adjusted SDBN results for `dryer`. Notice the grades now are more tightly focused around the prior grade `0.3`, with some above or below this prior.

Confidence-adjusted SDBN judgments for q=transformers dark of the moon

	beta_grade	upc	image	name
0	0.5957	97360810042		Transformers: Dark of the Moon - Blu-ray Disc
1	0.4017	400192926087		Transformers: Dark of the Moon - Original Soundtrack - CD
2	0.3673	97363560449		Transformers: Dark of the Moon - Widescreen Dubbed Subtitle - DVD
3	0.3130	97363532149		Transformers: Revenge of the Fallen - Widescreen Dubbed Subtitle - DVD
4	0.2795	93624956037		Transformers: Dark of the Moon - Original Soundtrack - CD

Figure 11.11 Beta-adjusted SDBN results for `transformers` `dark` `of` `the` `moon`. In figure 11.7, we noted the soundtrack seemed oddly high (`0.48`) in its relevance grade despite fewer clicks than the Blu-ray movie. We now see the soundtrack's relevance closer to the prior of `0.4`.

Figure 11.11 notably shows less confidence in the soundtrack when compared to the SDBN judgments without modeling confidence (figure 11.7). The grade has dropped from `0.4806` to `0.4017`. Notably, the DVD grade following the CD has not changed much, only changing from `0.3951` to `0.3673`, because of our higher confidence in that observation. As more ob-

servations come in, it's likely the CD would even move down in ranking if this pattern continues.

Most of your queries won't be like `dryer` or `transformers dark of the moon`. They'll be more like `blue ray`. To meaningfully work with these queries for LTR, you'll need to be able to handle these "small data" problems, such as having lower confidence.

We are beginning to have a more reasonable training set for automating LTR, but there's still work to do. In the next chapter, we'll move to look at the complete search feedback loop. This includes working on presentation bias. Recall that this is the bias where users never examine what search never returns to them. How can we add surveillance to the automated LTR feedback loop to both overcome presentation bias and ensure that our model—and by extension, the judgments—are working as expected?

Before we examine those topics in the next chapter, though, let's look again at training an LTR model end-to-end so you can experiment with what you've learned so far.

11.4 Exploring your training data in an LTR system

Great work! You've made it through chapters 10 and 11. You now have what you need to develop reasonable LTR training data and train an LTR model. You're likely eager to train a model from your work. Instead of repeating the extensive code from chapter 10 here, we've created an "End-to-End Automated Learning to Rank" notebook in the ch11 folder of the book's codebase (4.end-to-end-auto-ltr.ipynb). It will allow you to experiment with LTR on the RetroTech data (figure 11.12).

In this notebook, you can fine-tune the inner LTR engine—the feature engineering and model creation that attempts to satisfy the training data. You can also explore the implications of altering the automated inputs to this engine: the training data itself. Altogether, this notebook has every step you've learned about so far:

1. Processing raw click session data into judgments, using the SDBN click model and a beta prior
2. Transforming the dataframe into the judgments we used in chapter 10
3. Loading a selection of LTR features to use with the search engine's feature store capabilities
4. Logging these features from the search engine and then performing a pairwise transformation of the data into a suitable training set
5. Training and uploading a model to the search engine's model store
6. Searching and ranking with the model

SDBN Judgments using Beta Distribution

We have about a dozen queries where we've simulated the click stream. Here we compute the SDBN judgments, using a beta distribution, on each of these queries. The code in `generate_training_data` just repeats what we did in this section of the book, just for every query we have data for.

We then convert these to the `Judgments` object we use in Chapter 10

What you should play with

Explore the strength of the prior (`PRIOR_WEIGHT`) as well as the specific default relevance grade, `PRIOR_GRADE` . A stronger `PRIOR_WEIGHT` won't budge much from `PRIOR_WEIGHT` .

If you're feeling more advanced, you can explore different methods of computing the relevance judgments from these sessions, by replacing `sessions_to_sdbn` with your own formula for translating clicks to judgments.

```
PRIOR_GRADE=0.2
PRIOR_WEIGHT=10
```

Figure 11.12 Notebook exploring the full LTR system. You can take the model for a test drive.

We invite you to tune the click model parameters and to think of new features, and different ways of arriving at the final LTR model, discovering which ones seem to yield the best results. While you do this tuning, please be sure to question your own subjective assumptions compared to what the data is showing you.

With out-of-the box tuning, we'll leave you with figure 11.13, showing the current search results for the query `transformers dvd` . Try different queries here. How can you help the model better discriminate between relevant and irrelevant documents? Are the problems you encounter due to the training data used? Or are they due to the features used to construct the model?

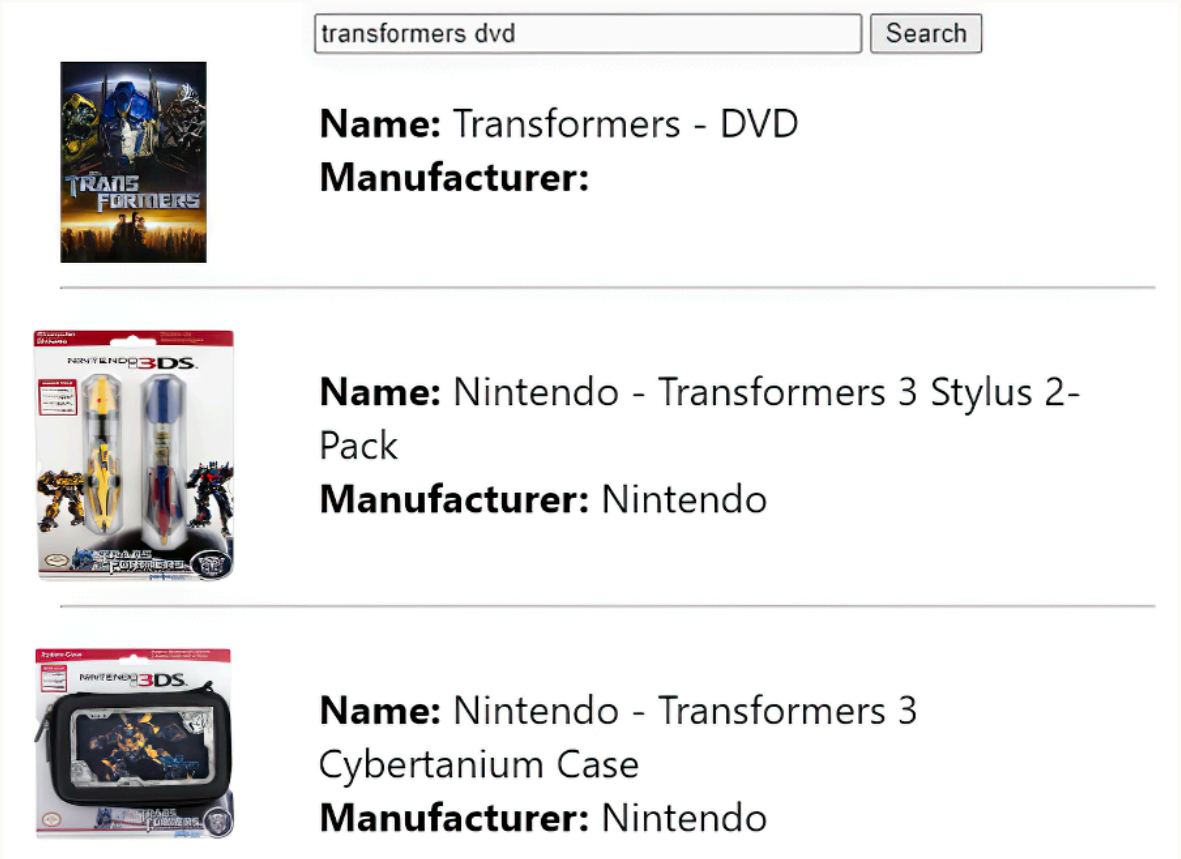

Figure 11.13 How our trained model ranks `transformers dvd` . Do you think you could improve on this?

In the next chapter, we'll finalize the automated LTR system by performing surveillance on the model. Most crucially, we'll consider how to overcome *presentation bias*. Even with the adjustments in this chapter, users will still only ever have a chance to act on what the search shows them. So we still have a feedback loop biased heavily by the current relevance ranking. How can we look out for this problem and overcome it? In the next chapter, we'll consider these problems as our LTR model continues to iteratively incorporate incoming user interactions and actively surface additional promising results.

Summary

- We can automate learning to rank (LTR) if we can reliably transform user click data into relevance judgments using a click model. However, the click model must be designed carefully to reduce bias in the data and ensure the reliability of the automated LTR system when deployed to live users.
- Learned (implicit) relevance judgment lists can be plugged into existing LTR training processes to either replace or augment manually created judgments.
- Raw clicks are usually problematic in automated LTR models due to common biases in how algorithms rank and present search results to users.
- Among the visible search results, position bias says users prefer results ranked near the top. We can overcome position bias by using a click model that tracks the probability that a user has examined a document or a position in the search results.
- Most search applications have a lot of spurious click data. When training data is biased toward these spurious results, we have confidence bias. We can overcome confidence bias by using a beta distribution to create a prior that we update gradually with new observations as they come in.

Previous chapter
< 10 Learning to rank for generalizable search relevance

Next chapter
12 Overcoming ranking bias through active learning >