12 Overcoming ranking bias through active learning

This chapter covers

- Harnessing live user interactions to gather feedback on a deployed LTR model
- A/B testing search relevance solutions with live users
- Using active learning to explore potentially relevant results beyond the top results
- Balancing *exploiting* user interactions while *exploring* what else might be relevant

So far, our learning to rank (LTR) work has taken place in the lab. In previous chapters, we built models using automatically constructed training data from user clicks. In this chapter, we'll take our model into the real world for a test drive with (simulated) live users!

Recall that we compared an automated LTR system to a self-driving car. Internally, the car has an engine: the end-to-end model retraining on historical judgments as discussed in chapter 10. In chapter 11, we compared our model's training data to self-driving car directions: what *should* we optimize to automatically learn judgments based on previous interactions with search results? We built training data and overcame key biases inherent in click data.

In this chapter, our focus is on moving our ranking models from the lab to production. We'll deploy and monitor our models as they receive user traffic. We'll see where the model does well and understand whether the work in the previous two chapters failed or succeeded. This means exploring a new kind of testing to validate our model: *A/B testing*. In *A/B testing* we randomly assign live users to different models and examine business outcomes (sales, etc.), to see which models perform best. You might be familiar with A/B testing in other contexts, but here we'll zero in on the implications for an automated LTR system.

Live users help us not just validate our system, they also aid in escaping dangerous negative feedback loops our models can find themselves in, as shown in figure 12.1.

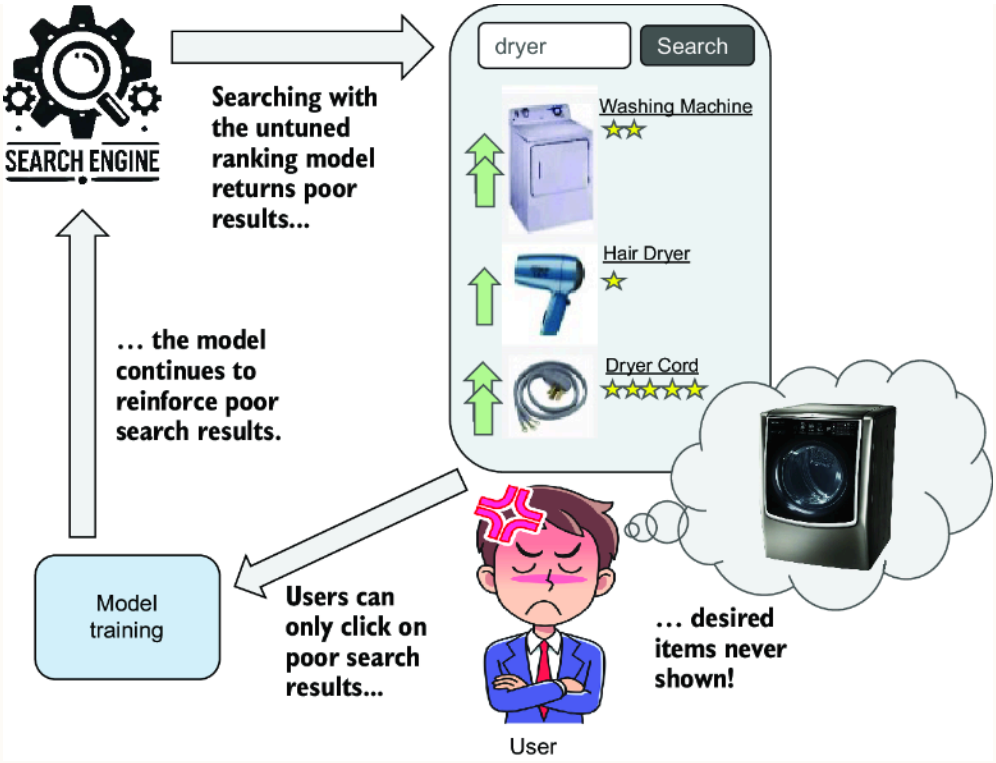

Figure 12.1 Presentation bias's negative feedback loop. Users never click on what the search engine never returns, so relevance models can never grow beyond the current model's knowledge.

In figure 12.1, our model can only learn what's relevant *within the results shown to the user*. In other words, we have an unfortunate chicken-and-egg problem: the model is trained based on what users deem relevant, but what users deem relevant is based on what the model shows them. Good LTR attempts to optimize for results with the most positive interaction signals, but users will only click on what's right in front of them. How could LTR possibly get better when the training data seems hopelessly biased toward the search engine's current ranking? This bias of implicitly derived training data reflecting back the previously displayed results is called *presentation bias*.

After we explore A/B testing, we'll fight presentation bias for the rest of the chapter using active learning. An *active learning* system is one that can interactively gather new labeled data from users to answer new questions. In our case, our active learning algorithm will determine blind spots leading to ranking bias, prompt users to interact with new results exploring those blind spots, and utilize the user interactions as new training data to correct the blind spots. Much like a self-driving car that has only learned one suboptimal path, we'll have to strategically explore alternative promising paths—in our case, additional types of search results—to learn new patterns for what's relevant to users. In figure 12.2, we see the automated LTR loop augmented with this blind-spot exploration.

Before we get to this all-important subject, we must first wrap everything we learned in chapters 10 and 11 into a few lines of code. Then we'll be able to iterate quickly, exploring A/B testing and overcoming presentation bias.

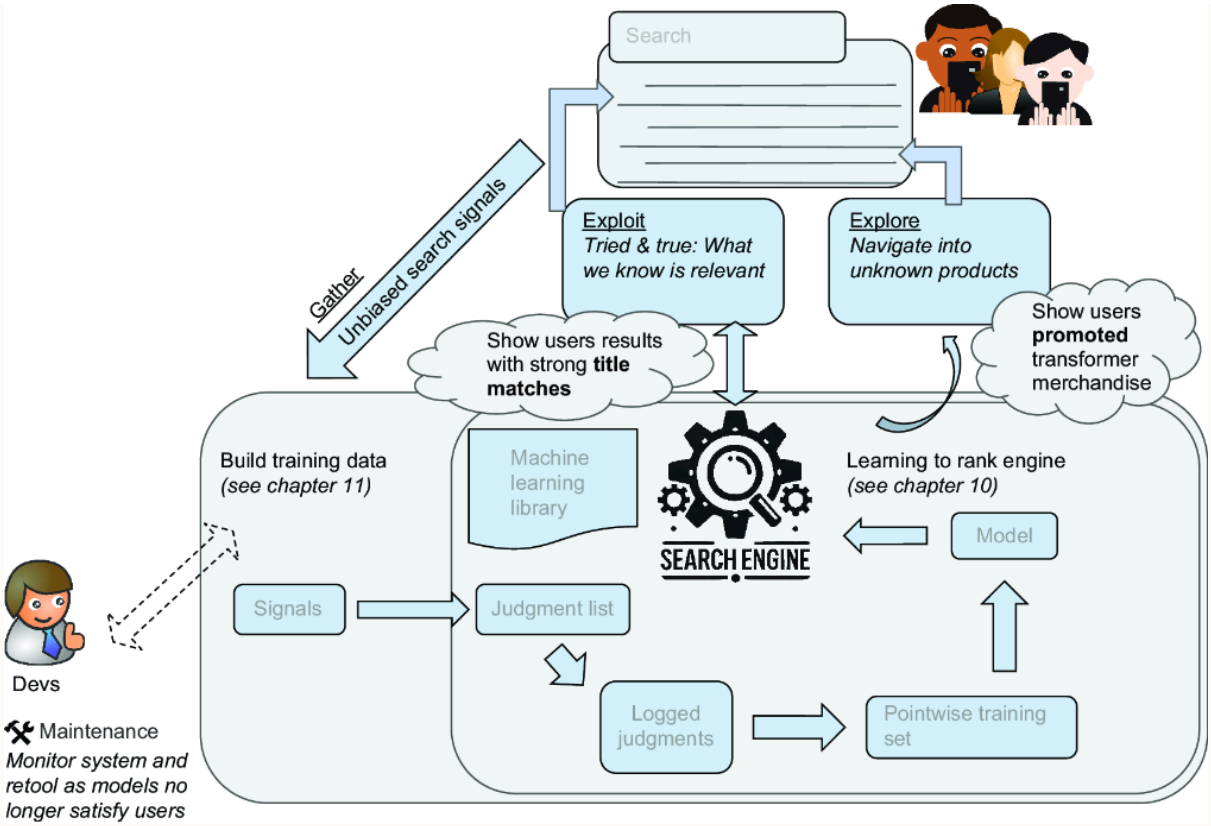

Figure 12.2 Automated LTR meets live users. To be useful, our automated LTR system must overcome presentation bias by exploring yet-to-be-seen results with users to expand training data coverage.

12.1 Our automated LTR engine in a few lines of code

Before we begin to A/B test, let's gather all our knowledge from chapters 10 and 11 into a small handful of reusable Python helper functions. First, we'll define a function to let us rebuild training data from raw session clicks using a simplified dynamic Bayesian network (SDBN) click model (all of chapter 11). Next, we'll create an equally simple snippet of code to train a model with that training data (all of chapter 10). We'll very quickly sum up these functions before diving into A/B testing and overcoming presentation bias in the rest of the chapter.

12.1.1 Turning clicks into training data (chapter 11 in one line of code)

In chapter 11, we turned clicks into training data and explored the SDBN click model, which can overcome biases in how users click on search results. We will reuse much of the code from chapter 11 as we explore additional biases and automate the end-to-end LTR process in this chapter.

As a reminder, our click model turns raw clicks into training labels or *grades* mapping how relevant a document is for a keyword. The raw input we need to build the training data includes a query string, the rank of the result as displayed, the document in that position, and whether it was clicked. We can see that data stored in this dataframe:

```
id  sess_id  query      rank  doc_id        clicked
0   50002    blue ray   0.0   600603141003  True
1   50002    blue ray   1.0   827396513927  False
2   50002    blue ray   2.0   24543672067   False
3   50002    blue ray   3.0   719192580374  False
4   50002    blue ray   4.0   885170033412  True
```

Given this input, we can wrap all of chapter 11 into a reusable function that computes our training data. Recall that we use the term *judgment list* or *judgments* to refer to our training data. We can see our judgments computation in the following listing.

Listing 12.1 Generating training data from sessions (chapter 11)

```
training_data = generate_training_data(sessions, prior_weight=10,
                                       prior_grade=0.2)
display(training_data)
```

Output (truncated):

```
query             doc_id          clicked  examined  grade     beta_grade
blue ray          27242815414     42       42        1.000000  0.846154
                  827396513927    1304     3359      0.388211  0.387652
                  883929140855    140      506       0.276680  0.275194
                  885170033412    568      2147      0.264555  0.264256
                  24543672067     665      2763      0.240680  0.240534
...
transformers dvd  47875819733     24       1679      0.014294  0.015394
                  708056579739    23       1659      0.013864  0.014979
                  879862003524    23       1685      0.013650  0.014749
                  93624974918     19       1653      0.011494  0.012628
                  47875839090     16       1669      0.009587  0.010721
```

The `generate_training_data` function takes in all the user search `sessions`, along with the `prior_weight`, indicating how strong the prior should be weighted (defaults to `10`), and the `prior_grade`, specifying the default probability of a result's relevance when we have no evidence (defaults to `0.2`). See section 11.3.2 for a refresher on how these values influence the SDBN calculation.

Let's briefly revisit what we learned in chapter 11 by looking at listing 12.1. As you can see in the output, we compute a dataframe where each query-document pair has corresponding `clicked` and `examined` counts. Clicks are what they sound like: the sum of raw clicks this product received for this query. Recall that `examined` corresponds to the number of times the click model thinks the user noticed the result.

The `grade` and `beta_grade` statistics are the training labels. These correspond to the probability that a document is relevant for the query. Recall that `grade` simply divides `clicked` by `examined`: the naive, first implementation of the SDBN click model. However, we learned in chapter 11 that it would be better to account for how much information we have (see section 11.3). We don't want one click with one examine `(1 / 1 = 1.0)` to be counted as strongly as a hundred-clicks with a hundred examines `(100 / 100 = 1.0)`. For this reason, `beta_grade` places a higher weight on results with more information (preferring the hundred-clicks example). We'll therefore use `beta_grade` as opposed to `grade` when retraining LTR models.

This data served as training data for the LTR models we trained in chapter 10. Next, let's see how we can easily take this training data, train a model, and deploy it.

12.1.2 Model training and evaluation in a few function calls

In addition to regenerating training data, we also need to retrain our model before deploying it for live users. In this section, we'll explore the convenience functions for our core LTR model training engine. This will set us up to quickly experiment with models through the rest of this chapter.

We'll wrap model training and offline evaluation in a few simple lines.

Listing 12.2 Training and evaluating the model on a few features

```
def train_and_evaluate_model(sessions, model_name, features, log=False):
  training_data = generate_training_data(sessions)
  train, test = split_training_data(training_data, 0.8)
  train_and_upload_model(train, model_name, features=features, log=log)
  evaluation = evaluate_model(test, model_name, training_data, log=log)
  return evaluation

feature_set = [
  ltr.generate_query_feature(feature_name="long_description_bm25",
                             field_name="long_description"),
  ltr.generate_query_feature(feature_name="short_description_constant",
                             field_name="short_description",
                             constant_score=True)]

evaluation = train_and_evaluate_model(sessions, "ltr_model_variant_1",
                                      feature_set)
display(evaluation)
```

Evaluation for `ltr_model_variant_1`:

```
{"dryer": 0.03753076750950996,
 "blue ray": 0.0,
 "headphones": 0.0846717500031762,
 "dark of moon": 0.0,
 "transformers dvd": 0.0}
```

With the help of listing 12.2, let's briefly revisit what we learned in chapter 10. We define a `feature_set` with two features for LTR: one to search against the `long_ description` field, and another to search against the `short_description` field. We must choose carefully, hoping to find features that meaningfully predict relevance and that can be learned from the training data in listing 12.1. We then split the `training_data` into `train` and `test` sets and use the `train` set to train and upload the model.

But how do we know if our model successfully learned from the training data? Splitting the judgments and excluding the `test` set during model training reserves some of the training data for evaluating the trained model. You're like a professor giving the student (here the model) a final exam. You might give students many sample problems to study for the test (the `train` set). But to see if students truly learned the material, as opposed to just memorizing it, you'd give them a final exam with different questions (the `test` set). This helps you evaluate

whether the student understands what you've taught them before sending them off into the real world.
Of course, success in the classroom does not always equate to success in the real world. Graduating our model into the real world, with live users in an A/B test, might show it does not perform as well as we hoped!

Finally, what is the statistic next to each test query? How do we evaluate the students' success on the test queries? Recall from chapter 10 that we simply used precision (the proportion of relevant queries). This statistic sums the top `N` grades and divides by `N` (for us `N` = 10), which is effectively the average relevance grade. We recommend exploring other statistics for model training and evaluation that are biased toward getting the top positions correct, such as Discounted Cumulative Gain (DCG), Normalized DGC (NDCG), or Expected Reciprocal Rank (ERR). For our purposes, we'll stay with the simpler precision statistic.

Just judging by the relevance metrics for our test queries in listing 12.2, our model does quite poorly in offline testing. By improving offline metrics, we should see a significant improvement with live users in an A/B test.

12.2 A/B testing a new model

In this section, we'll simulate running an A/B test and compare listing 12.2's model to a model that seems to perform better in the lab. We'll reflect on the results of the A/B test, setting us up to complete the automated LTR feedback loop we introduced in chapter 11. We'll finish by reflecting on what didn't go so well, spending the remainder of the chapter on adding "active learning", a crucial, missing piece to our automated LTR feedback loop.

12.2.1 Taking a better model out for a test drive

Our original LTR model hasn't performed very well, as we saw in the output of listing 12.2. In this section, we'll train a new model, and once it looks promising, we'll deploy it in an A/B test against the model we trained in listing 12.2.

Let's look at the following improved model.

Listing 12.3 A new model improved by changing the features

```
feature_set = [
  ltr.generate_fuzzy_query_feature(feature_name="name_fuzzy",
                                   field_name="name"),
  ltr.generate_bigram_query_feature(feature_name="name_bigram",
                                    field_name="name"),
  ltr.generate_bigram_query_feature(feature_name="short_description_bigram"
                                    field_name="short_description")]

evaluation = train_and_evaluate_model(sessions, "ltr_model_variant_2",
                                      feature_set)
display(evaluation)
```

Evaluation for `ltr_model_variant_2`:

```
{"dryer": 0.07068309073137659,     #Before: 0.038
 "blue ray": 0.0,                          # 0.0
 "headphones": 0.06540945492120899,        # 0.085
 "dark of moon": 0.2576592004029579,       # 0.0
 "transformers dvd": 0.10077083021678328} # 0.0
```

In the preceding listing, we define a `feature_set` containing three features: `name_fuzzy`, which performs a fuzzy search against the `name` field, `name_bigram`, which performs a two-word phrase search on the `name` field, and `short_description_ bigram`, which performs a two-word phrase search against the `short_description` field. Like before, this model is trained, deployed, and evaluated. Notice the output of listing 12.3—on the same set of test queries, our model seems to perform much better. This seems promising! Indeed, we've chosen a set of features that seems to capture the text-matching aspects of relevance better.

The astute reader might notice we've kept the test queries the same as in listing 12.2. We've intentionally done this for clarity. It's good enough to teach you fundamental AI-powered search skills. In real life, however, we would want a truly random test/train split to better evaluate the model's performance. We might even take things further, performing *cross-validation*—the resampling and training of many models on different test/train dataset splits to ensure the models generalize well without overfitting to the training data. If you'd like to dive deeper into offline model evaluation, we recommend a more general machine learning book, such as *Machine Learning Bootcamp* by Alexey Grigorev (Manning, 2021).

Perhaps your search team feels the model trained in listing 12.3 has promise and is good enough to deploy to production. The team's hopes are up, so let's see what happens when we deploy to production for further evaluation with live users.

12.2.2 Defining an A/B test in the context of automated LTR

By the end of chapter 11, we had developed an end-to-end LTR retraining process: we could take incoming user signals to generate a click model, use the click model to generate judgments, use the judgments to train an LTR model, and then deploy the LTR model to production to gather more signals to restart the process. With this LTR retraining loop set up, we can easily deploy promising new ranking models.

We haven't actually deployed our LTR models to production yet, though. We've only developed the theoretical models. How do we know whether what we built in the lab performs well in the real world? It's quite a different thing to handle live, real-world scenarios.

In this section, we'll explore the results of A/B testing with (simulated) live users. Because this is a book with a codebase you're running locally, it's unfortunately not possible for us to have real users hitting our application. Therefore, the "live" user traffic we'll use will just be simulated from within our codebase. For our purposes, this traffic simulation is similar enough to live user interactions to successfully demonstrate the active learning process.

We'll see how the A/B test serves as the ultimate arbiter of our automated LTR system's success. It will enable us to correct problems in our offline automated LTR model training so the feedback loop can progressively grow more reliable.

You may know about A/B tests, but here we'll demonstrate how they factor into an automated LTR system. As illustrated in figure 12.3, an *A/B test* randomly assigns users to two *variants*. Each *variant* contains a distinct set of application features. This might include anything from different button colors to new relevance ranking algorithms. Because users are randomly assigned to the variants, we can more reliably infer which variant performs best on chosen business outcomes, such as sales, time spent on the app, user retention, or whatever else the business might choose to prioritize.

When running an A/B test, you will usually make one of the variants a *control group* representing the current default algorithm. Having a control allows you to measure the improvement of other models. It's also common to perform multivariate testing, whereby multiple variants or combinations of variants are tested simultaneously. More advanced testing strategies can be implemented, like multi-arm bandit testing, where the test continually shifts live traffic toward the currently best-performing variants, or signals-based backtesting, where you use historical data to simulate the A/B test to predict the best variant offline before even showing results to live users.

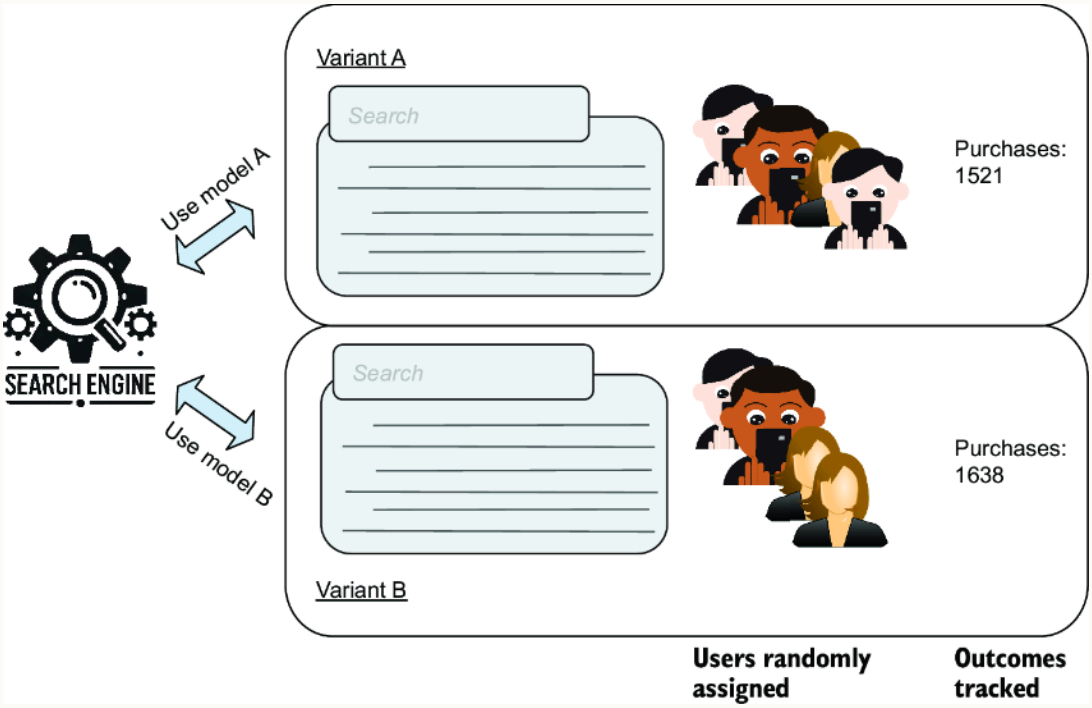

Figure 12.3 A search A/B test. Search users are randomly assigned to two relevance algorithms (here two LTR models) with outcomes tracked.

12.2.3 Graduating the better model into an A/B test

Next, we'll deploy our promising new model, `ltr_model_variant_2` from listing 12.3, into an A/B test. We'll then explore the implications of the test results. Hopes are high, and your team thinks this model might knock the socks off the competition: the poorly performing `ltr_model_variant_1` from listing 12.2.

In this section, we'll simulate an A/B test, assigning 1,000 users randomly to each model. In our case, these simulated users have specific items they want to buy. If they see those items, they'll make a purchase and leave our store happy. If they don't, they might browse around, but they'll most likely leave without making a purchase. Our search team, of course, doesn't know what users hope to buy—this information is hidden from us. We only see a stream of clicks and purchases, which, as we'll see, is heavily influenced by presentation bias.

In listing 12.4, we have a population of users seeking the newest Transformers movies by searching for `transformers dvd`. We'll stay focused on this single query during our discussion. Of course, with a real A/B test, we'd look over the full query set, and the user population wouldn't be this static. By zeroing in on one query, though, we can more concretely understand the implications of our A/B test for automated LTR. For a deeper overview of good A/B testing experimentation, we recommend the book *Experimentation for Engineers: From A/B testing to Bayesian optimization* by David Sweet (Manning, 2023).

For each run of the `a_b_test` function in listing 12.4, a model is assigned at random. Then the `simulate_live_user_session` function simulates a user searching with the query and selected model, scanning the results, possibly clicking and making a purchase. Unbeknownst to us, our user population has hidden preferences behind their queries, which are simulated in `simulate_live_user_session`. We run `a_b_test` 1,000 times, collecting the purchases made by users that use each model.

Listing 12.4 Simulated A/B test for the query `transformers dvd`

```
def a_b_test(query, model_a, model_b):
  draw = random.random()
  model_name = model_a if draw < 0.5 else model_b  #1
  purchase_made = simulate_live_user_session(query,  #2
                                             model_name)  #2
  return (model_name, purchase_made)

def simulate_user_a_b_test(query, model_a, model_b, number_of_users=1000):
  purchases = {model_a: 0, model_b: 0}
  for _ in range(number_of_users):  #3
    model_name, purchase_made = a_b_test(query, model_a, model_b)  #3
    if purchase_made:  #4
      purchases[model_name] += 1  #4
  return purchases

results = simulate_user_a_b_test("transformers dvd",
                                 "ltr_model_variant_1",
                                 "ltr_model_variant_2")
display(results)
```

#1 Randomly assigns each user to model a or b
#2 Simulates a user's searching and purchasing behavior
#3 Simulates the number_of_users being tested
#4 Counts the total number of purchases made by each model

Output:

```
{"ltr_model_variant_1": 21,
 "ltr_model_variant_2": 15}
```

As we see in the output of listing 12.4, `ltr_model_variant_2` (our golden student), actually performs *worse* in this A/B test! How can this be? What could have gone wrong for it to have such good offline test metric performance but poor outcomes in the real world?

For the rest of this chapter, we'll dive into what's happening and attempt to address the problem. Thus, you'll learn how live users can increase the accuracy of your automated LTR system,

allowing you to retrain with confidence!

12.2.4 When "good" models go bad: What we can learn from a failed A/B test

As we saw in listing 12.4, a lot can change when our model enters the real world. In this section, we'll reflect on the implications of the A/B test we just ran to see what next steps would be appropriate.

When a model performs well in the lab but fails an A/B test, it means that while we may have built a "correct" LTR model, we built it to the wrong specification. We need to correct problems with the training data itself: the judgments generated from our click model.

But how might problems creep into our click-model-based judgments? We saw two problems in chapter 11: *position bias* and *confidence bias*. Depending on your goals, UX, and domain, additional biases can creep in. In e-commerce, users might be enticed to click an item that's on sale, skewing the data toward those items. In a research setting, one article might provide a richer summary in the search results than another. Some biases blur the line between "bias" and actual relevance for that domain. A product with a missing image, for example, might get fewer clicks. It might be technically identical to another "relevant" product with an image, but, to users, a product missing an image seems less trustworthy and thus won't be clicked. Is that a bias or simply an actual indicator of relevance for this domain, where product trustworthiness is a factor?

To make better judgments, should clicks be ignored or discounted, and should we instead use other behavioral signals? Perhaps follow-on actions after clicking, such as clicking a "like" button, adding an item to a cart, or clicking a "read more" button, ought to be included? Perhaps we should ignore "cheap" or accidental clicks when the user immediately hits the back button after clicking?

Considering post-click actions can be valuable. However, we must ask how strongly search ranking influences events like a purchase or add-to-cart, or whether they are attributable to other factors. For example, a lack of purchases could indicate a problem with a product display page, or with a complex checkout process, not just with the search result's relevance for a specific query.

We might use an outcome like total purchases, in aggregate over all queries for each test group, to evaluate an A/B test. As long as all other variables in the app remain unchanged, except the ranking algorithm, then we know any significant difference in purchases across test groups must be caused by the one thing we changed. However, in the specific query-to-document relationship, causality gets complicated. Any single product may have very few purchases (many people view a $2,000 television, but very few buy). The data may simply lack enough quantity to know whether the purchase is exclusively related to a product's specific relevance for a query.

Accounting for all the variations in search UX, domains, and behaviors would fill many books and still fall short. The search space constantly evolves, with new ways of interacting with search results coming in and out of fashion. For most scenarios, using clicks and standard click models will suffice. Clicks in search UIs have been heavily studied. Still, arriving at good judgments is both an art and science; you may find a slight modification to a click model that accounts for extra signals is important to your domain and may provide tremendous gains in

how your model performs in an A/B test. You can spend as much time perfecting your click model as you do developing your search engine.

However, there is one universally pernicious training data problem that challenges all click models: presentation bias. *Presentation bias* occurs when our models can't learn what's relevant from user clicks because the results never show up to be clicked in the first place. We'll dive into this difficult problem next and learn how to overcome this bias by simultaneously optimizing both for what our models have already learned and for what they still need to learn.

12.3 Overcoming presentation bias: Knowing when to explore vs. exploit

Regardless of whether we use clicks or more complex signals, users never interact with what they can't see. In other words, underneath automated LTR is a chicken and egg problem: if the relevant result is never returned by the original, poorly tuned system, how could any click-based machine learning system learn that the result is relevant?

In this section, you'll learn about one machine learning technique that selects documents to explore *despite those results having no click data*. This final missing piece of our automated LTR system helps us not just build models optimized for the training data, but actively participates in its own learning to grow the breadth of the available training data. We call a system that participates in its own learning an *active learning* system.

Figure 12.4 demonstrates presentation bias. The items on the right side of the figure could feasibly be relevant for our query, but, without traffic, we have no data to know either way. It would be nice to send some traffic to these results to learn whether users find them relevant.

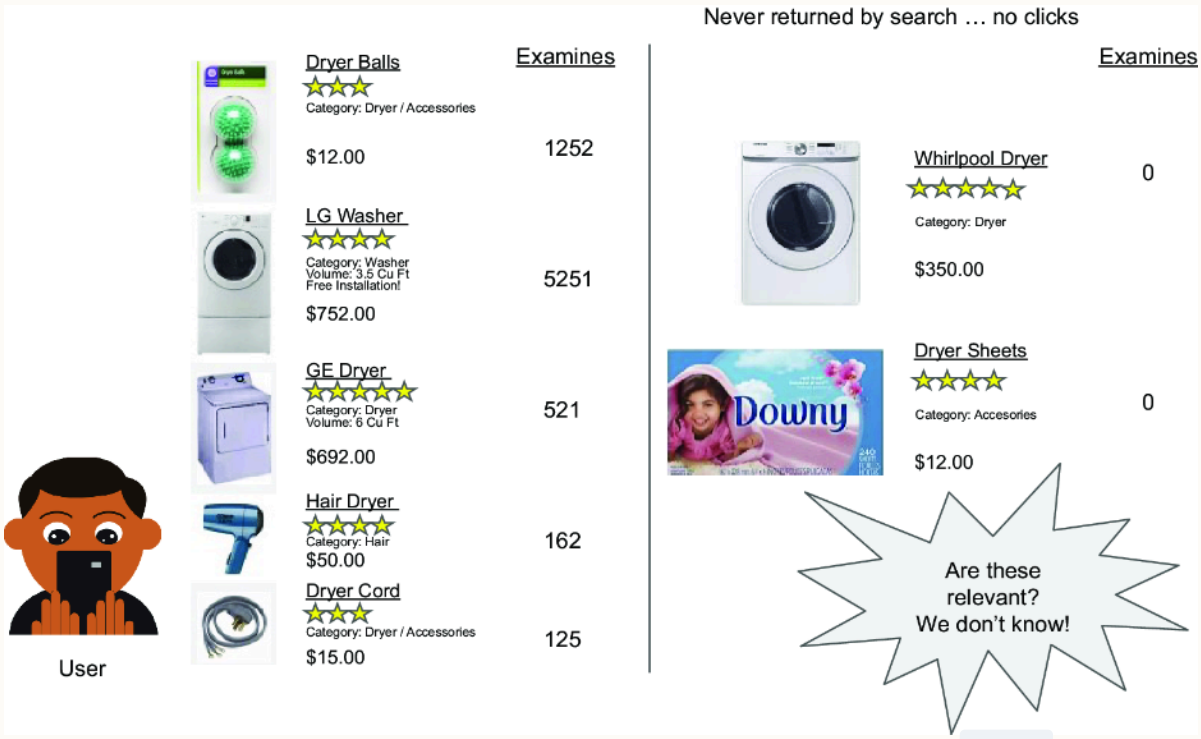

Figure 12.4 Presentation bias (versus unexplored results) for the query `dryer`

To overcome presentation bias, we must carefully balance *exploiting* our model's current, hard-earned knowledge and *exploring* beyond that knowledge. This is the *explore versus exploit* tradeoff. Exploring lets us gain knowledge, growing the coverage of our click model to new and different types of documents. However, if we always explore, we will never take advantage of our knowledge. When we *exploit*, we optimize for what we currently know per-

forms well. Exploiting corresponds to our current LTR model that aligns with our training data. Knowing how to systematically balance exploring and exploiting is key, and it's something we'll discuss in the next few sections using a machine learning tool (a Gaussian process) built for this purpose.

12.3.1 Presentation bias in the RetroTech training data

Let's first analyze the current training data to get the lay of the land. What kinds of search results does the training data lack? Where is our knowledge incomplete? Another way of saying "presentation bias" is that there are potentially relevant search results excluded from the training data: blind spots we must detect and fight against. Once we've defined those blind spots, we can better correct them. This will set us up to retrain with a more robust model.

In listing 12.5, we create a new feature set named `explore_feature_set`, in which we have three simple features: `long_description_match`, `short_description_match`, and `name_match`, telling us whether a given field match occurs or not. These correspond to features our model has already learned. In addition, we've added a `has_ promotion` feature. This feature becomes a `1.0` if the product is on sale and is being promoted through marketing channels. We haven't explored this feature before; perhaps it's a blind spot?

Listing 12.5 Analyzing missing types of documents

```
def get_latest_explore_feature_set():
  return [
    ltr.generate_query_feature(  #1
      feature_name="long_description_match",  #1
      field_name="long_description",  #1
      constant_score=True),  #1
    ltr.generate_query_feature(  #1
      feature_name="short_description_match",  #1
      field_name="short_description",  #1
      constant_score=True),  #1
    ltr.generate_query_feature(  #1
      feature_name="name_match",  #1
      field_name="name", constant_score=True),  #1
    ltr.generate_query_feature(  #2
      feature_name="has_promotion",  #2
      field_name="has_promotion", value="true")]  #2

def get_logged_transformers_judgments(sessions, features):
  training_data = generate_training_data(sessions)  #3
  logged_judgments = generate_logged_judgments(training_data,  #4
                                               features, "explore")  #4
  logged_judgments = logged_judgments \  #5
      [logged_judgments["query"] == "transformers dvd"]  #5
  return logged_judgments

explore_features = get_latest_explore_features()
logged_transformers_judgments = get_logged_transformers_judgments(sessions,
                                                                  explore_features)
display(logged_transformers_judgments)
```

#1 Features corresponding to fields already used to train the LTR model.
#2 New feature we're exploring for blind spot: promotions
#3 Builds SDBN judgments from the current raw sessions
#4 Logs the feature values and returns the SDBN judgments joined with the feature values
#5 Examines the properties of the current "transformers dvd" training data

Output:

```
doc_id       query             grade  long_desc*_match   name_match
                                          short_desc*_match  has_promotion
-------------------------------------------------------------------------------
97363560449  transformers dvd  0.34   0.0   0.0          1.0   0.0
97361312804  transformers dvd  0.34   0.0   0.0          1.0   0.0
97361312743  transformers dvd  0.34   0.0   0.0          1.0   0.0
97363455349  transformers dvd  0.34   0.0   0.0          1.0   0.0
...
708056579739 transformers dvd  0.01   1.0   1.0          1.0   0.0
879862003524 transformers dvd  0.01   1.0   1.0          1.0   0.0
93624974918  transformers dvd  0.01   0.0   0.0          1.0   0.0
47875839090  transformers dvd  0.01   1.0   0.0          1.0   0.0
```

We can see some gaps in our training data's knowledge in the output of listing 12.5:

- Every item includes a name match.

- No promotions (`has_promotion=0`) are present.
- There's a range of `long_description_match` and `short_description_match` values.

Intuitively, if we want to expand our knowledge, we should show users searching for `transformers dvd` something completely outside the box from what's in the output of listing 12.5. That would mean showing users a promoted item, and possibly one with no name match. In other words, we need to get search out of its own echo chamber by explicitly diversifying what we show to the user, moving away from what's in the training data. The only question is, how much of a risk are we willing to take to improve our knowledge? We don't want to blanket the search results with random products just to broaden our training data.

What we've done so far has not been systematic: we've only analyzed a single query to see what was missing. How might we automate this? Next up, we'll discuss one method for automating exploration using a tool called a Gaussian process.

12.3.2 Beyond the ad hoc: Thoughtfully exploring with a Gaussian process

A *Gaussian process* is a statistical model that makes predictions and provides a probability distribution capturing the certainty of that prediction. In this section, we'll use a Gaussian process to select areas for exploration. Later in this chapter, we'll create a more robust way of finding gaps in our data.

GAUSSIAN PROCESS BY EXAMPLE: EXPLORING A NEW RIVER BY EXPLOITING EXISTING KNOWLEDGE

To get an intuition for Gaussian processes, let's use a concrete example of real-life exploration. Working through this example will enable you to think more intuitively about how we might, mathematically, make explore versus exploit tradeoffs.

Imagine you're a scientist planning to survey a rarely explored river deep in the wilderness. As you plan your trip, you have only spotty river depth observations from past expeditions to know when it's safe to travel. For example, one observation shows the river is two meters deep in April; another time in August it's one meter deep. You'd like to pick a date for your expedition that optimizes for ideal river conditions (i.e., not monsoon season, but also not during a dry spell). However, you're also a scientist—you'd like to make observations during yet-unobserved times of the year to increase your knowledge of the river. Figure 12.5 shows river depth measurements made throughout the year.

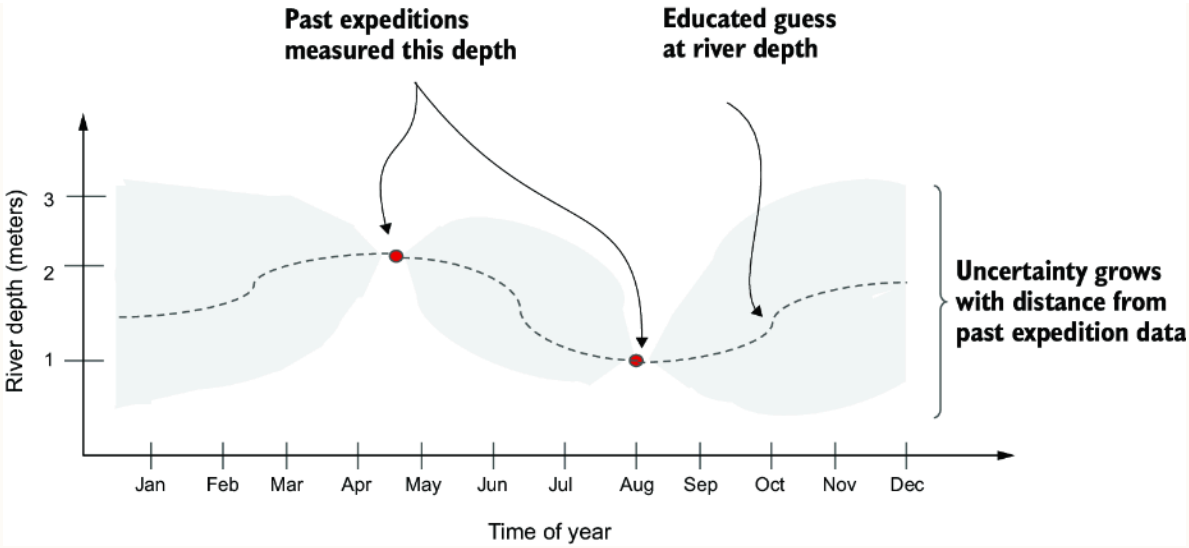

Figure 12.5 Exploring a river, with uncertainty of the river's depth increasing as we get further away from past observations. How would we pick a time of year that is both safe and maximally increases our knowledge of the river's depth?

How might we choose a date for the expedition? If you observed a river depth of two meters on April 14th, you would guess that the April 15th depth would be very close to two meters. Traveling during that time might be pleasant: you trust the river wouldn't be excessively flooded. However, you wouldn't gain much knowledge about the river. What about trying to go several months away from this observation, like January? January would be too far from April to understand the river's likely depth. We might travel during a treacherous time of the year, but we'd almost certainly gain new knowledge—perhaps far more than we bargained for! With so little data to go on, there may be too much risk exploring at this time of the year.

In figure 12.5 we see an educated guess at the river level, based on an expected correlation between adjacent dates (April 15th and 14th should be very close). Our level of certainty decreases as we move away from direct observations, represented by the widening gray zone.

Figure 12.5 illustrates a Gaussian process. It mathematically captures predictions, along with our uncertainty in each prediction. How does this relate to relevance ranking? Just as nearby dates have similar river levels, similar search results would be similar in their relevance. Consider our explore features from listing 12.5. Those with strong name matches for `transformers` `dvd`, that are not promoted, and with no short/long description matches would likely have similar relevance grades—all moderately relevant. As we move away from these well-trodden examples—perhaps adding promoted items—we grow less certain about our educated guesses. If we go very far, way outside the box, such as a search result with no name match, that is promoted, but that has strong short/long description field matches, our uncertainty grows very high. Just like the scientist considering a trip in January, we have almost no ability to make a good guess about whether those results could be relevant. It might involve too much risk to show those results to users.

We can use Gaussian processes to balance exploiting existing relevance knowledge with riskier exploration to gain knowledge. Gaussian processes use incomplete information to make careful tradeoffs in the likely quality and the knowledge gained. For example, we can trade off ideal river conditions or a likely relevant search result against learning more about river conditions or about the relevance of a new kind of search result. We can carefully choose how far away from known, safe search results we want to venture to gain new knowledge.

In our `transformers dvd` case, what kind of search result would have a high upside, would likely also be relevant/safe to explore, but also would maximally increase our knowledge? Let's train a Gaussian process and find out!

TRAINING AND ANALYZING A GAUSSIAN PROCESS

Let's get hands-on to see how a Gaussian process works. We'll train a Gaussian process on the `transformers dvd` query. We'll then use it to generate the best exploration candidates. You'll see how we can score those exploration candidates to maximally reduce our risk and increase the likelihood we'll gain knowledge.

In the following listing, we train a Gaussian process using the `GaussianProcessRegressor` (aka `gpr`) from `sklearn`. This code creates a Gaussian process that attempts to predict the relevance `grade` as a function of the `explore_feature_set` features we logged.

Listing 12.6 Training a `GaussianProcessRegressor` on our training data

```
from sklearn.gaussian_process import GaussianProcessRegressor

def train_gpr(logged_judgments, feature_names):
  feature_data = logged_judgments[feature_names]  #1
  grades = logged_judgments["grade"]  #2
  gpr = GaussianProcessRegressor()  #3
  gpr.fit(feature_data, grades)  #3
  return gpr
```

#1 Uses the features logged from the explore_feature_set in listing 12.5
#2 Predicts relevance grades
#3 Creates and trains the gpr model

Once we've trained a `GaussianProcessRegressor`, we can use it to make predictions. Remember that a `GaussianProcessRegressor` not only predicts a value, it also returns the probability distribution of that prediction. This helps us gauge the model's certainty.

In listing 12.7, we generate candidate feature values we'd like to possibly explore. With our river exploration example, these values correspond to possible exploration dates for our scientist's expedition. In our case, as each feature can either be `0` or `1`, we look at each possible feature value as a candidate.

Listing 12.7 Predicting a set of candidates to explore

```
def calculate_prediction_data(logged_judgments, feature_names):
  index = pandas.MultiIndex.from_product([[0, 1]] * 4,  #1
                                         names=feature_names)  #1
  with_prediction = pandas.DataFrame(index=index).reset_index()


  gpr = train_gpr(logged_judgments, feature_names)
  predictions_with_std = gpr.predict(  #2
    with_prediction[feature_names], return_std=True)  #2
  with_prediction["predicted_grade"] = predictions_with_std[0]  #3
  with_prediction["predicted_stddev"] = predictions_with_std[1]  #3



  return  with_prediction.sort_values("predicted_stddev", ascending=True)

explore_features = get_latest_explore_features()
logged_transformers_judgments = get_logged_transformers_judgments(sessions,
                                                                  explore_features)
feature_names = [f["name"] for f in explore_features]
prediction_data = calculate_prediction_data(logged_transformers_judgments,
                                                                    feature_names)
display(prediction_data)
```

#1 Generates a candidate matrix with a possible value of 0 or 1 for each feature we want to explore
#2 Predicts grade and standard deviation for those candidates based on the gpr probability distribution
#3 Stores the predicted grade and standard deviation from the gpr

Output:

```
long_description_match  name_match      predicted_grade
         short_description_match  has_promotion         prediction_stddev
0        0               1         0     0.256798        0.000004
1        0               1         0     0.014674        0.000005
1        1               1         0     0.014864        0.000007
0        1               1         0     0.022834        0.000010
1        0               1         1     0.018530        0.000010
0        0               1         1     0.161596        0.632121
1        1               1         1     0.014856        0.632121
```

In the output of listing 12.7, we see a `predicted_grade`—the educated guess from `gpr` on the relevance of that example. We also have `prediction_stddev`, which captures the gray band in figure 12.5—how much uncertainty is in the prediction.

We note in the output of listing 12.7 that the standard deviation is near 0 for the first four products with `name_match=1`. In other words, the `gpr` tends to have more confidence when `name_match=1`. We see after these observations that the standard deviation dramatically increases, as we lack a great deal of knowledge beyond these initial name-matching examples.

The output begins to show the presentation biases we intuitively detected in listing 12.5. We find a tremendous amount of knowledge about the importance of name matches, but little

knowledge about other cases. Which case would be worth exploring with live users and also minimizes the risk that we'll show users something completely strange in the search results? In listing 12.8, we generate and score exploration candidates using an algorithm called *expected improvement*, which predicts the candidates with the highest potential upside. We'll only cover the basic details of this algorithm, so we recommend the article "Exploring Bayesian Optimization" by Agnihotri and Batra if you'd like to learn more (**https://distill.pub/2020/bayesian-optimization**).

Listing 12.8 Calculating expected improvement for exploration

```
def calculate_expected_improvement(logged_judgments, feature_names, theta=0.6):
  data = calculate_prediction_data(logged_judgments, feature_names)
  data["opportunity"] = (data["predicted_grade"] -  #1
           logged_judgments["grade"].mean() - theta) #1
 #1
  data["prob_of_improvement"] = (
    norm.cdf(data["opportunity"] /  #2
             data["predicted_stddev"]))  #2

  data["expected_improvement"] = (  #3
    data["opportunity"] * data["prob_of_improvement"] +  #3
    data["predicted_stddev"] *  #3
    norm.pdf(data["opportunity"] /  #3
             data["predicted_stddev"]))  #3
  return data.sort_values("expected_improvement",  #4
                          ascending=False)  #4

improvement_data = calculate_expected_improvement(
      logged_transformers_judgments, feature_names)
display(improvement_data)
```

#1 Is the predicted grade likely to be above or below the typical grade?
#2 Probability we'll improve over the mean, considering the amount of uncertainty in the prediction
#3 How much there is to gain, given the probability and improvement and the magnitude of the improvement.
#4 Sorts to show the best exploration candidates

Output:

```
long_description_match  name_match  opportunity         expected_improvement
    short_description_match  has_promotion     prob_of_improvement
0   0                    0    1      -0.638497  0.234728  0.121201
0   1                    0    1      -0.725962  0.213214  0.110633
0   0                    0    0      -0.580755  0.232556  0.107853
1   1                    0    1      -0.727500  0.204914  0.101653
0   1                    0    0      -0.722661  0.181691  0.078549
```

The higher the expected improvement, the higher the predicted upside for the exploration candidate. But how does the algorithm quantify this potential upside? It's either the case that we know to a high degree of certainty there's an upside (standard deviation is low and the predicted grade is high) or we know that there's a high degree of uncertainty but the predicted grade is still high enough to take a gamble. We can see this in the following code from listing 12.8:

```
data["expected_improvement"] = (
  data["opportunity"] * data["prob_of_improvement"] +
  (data["prediction_stddev"] * norm.pdf(data["opportunity"] /
                                        data["prediction_stddev"])))
```

An opportunity that is more of a sure thing is covered by this first expression:

```
data["opportunity"] * data["prob_of_improvement"]
```

Meanwhile, an unknown opportunity with wide variability is covered by this second expression (after the `+`):

```
data["prediction_stddev"] * norm.pdf(data["opportunity"] /
                                     data["prediction_stddev"])
```

In the first expression, you'll notice that the opportunity (how much we expect to gain) times the probability that improvement happens corresponds to feeling confident in a good outcome. On the other hand, the second expression depends much more on the standard deviation. The higher the standard deviation *and* opportunity, the more likely it will be selected.

We can calibrate our risk tolerance with a parameter called `theta`: the higher this value, the more we prefer candidates with a higher standard deviation. A high `theta` causes `opportunity` to diminish toward 0. This biases scoring to the second expression—the unknown, higher standard deviation cases.

If we set `theta` too high, our `gpr` selects candidates to learn about without considering whether they might be useful to the user. If `theta` is too low, we don't explore new candidates very much and instead will be biased toward existing knowledge. A high `theta` is analogous to a scientist taking a high degree of risk (like exploring in January) in figure 12.5, while a very low `theta` corresponds to traveling during a risk-averse time (like mid-April). Because we're using this algorithm to augment an existing LTR system, we chose `theta` of `0.6` (a bit high) to give us more knowledge.

In the output of listing 12.8, we see that `gpr` confirms our earlier ad hoc analysis: we should show users items with promotions. These products will more likely yield greater knowledge, with possibly a high upside from the gamble.

Now that we have identified the kind of products we should explore, let's gather products from the search engine to show users. The following listing shows how we might go about selecting products for exploration that we can later intersperse into the existing model's search results.

Listing 12.9 Selecting a product to explore from the search engine

```
def explore(query, logged_judgments, features):  #1
  feature_names = [f["name"] for f in features]
  prediction_data = calculate_expected_improvement(logged_judgments,
                                                   feature_names)
  explore_vector = prediction_data.head().iloc[0][feature_names] #2
  return search_for_explore_candidate(explore_vector, query) #3

explore_features = get_latest_explore_features()
logged_judgments = get_logged_transformers_judgments(sessions,
                                            explore_features)
exploration_upc = explore("transformers dvd", logged_judgments,
                                             explore_features)["upc"]
print(exploration_upc)
```

#1 Explores according to the provided explore vector and selects a random doc from that group
#2 Extracts the best exploration candidate features based on expected_improvement
#3 Searches for a candidate matching the criteria

Output:

```
826663114164   # Transformers: The Complete Series [25th Anniversary ... ]
```

In listing 12.9, we take the best exploration candidate—promoted items—and issue a query to fetch documents with those characteristics. We omit the lower-level translation here of converting the candidate into a query (the `explore_query` function), but you can imagine that if `has_promotion=1.0` in the candidate, that we would issue a query searching for any item with a promotion (`has_promotion=true`), and so on for other features.

We see in the output of listing 12.9 that for the query `transformers dvd`, the randomly selected promoted product for exploration is 826663114164. This corresponds to *Transformers: The Complete Series [25th Anniversary Matrix of Leadership Edition] [16 Discs] - DVD*. Interesting!

What should we do with this document? This comes down to a design decision. A common choice is to slot it into the third position of the results, as this allows the user to still see the previously most relevant results first, but also ensures that the explore result gets high visibility. Note our 826663114164 document in the third slot (`rank=2.0`).

```
doc_id       product_name               sess_id query             rank click
93624974918  Transformers: Revenge O... 100049  transformers dvd  0.0  False
879862003524 Razer - DeathAdder Tran... 100049  transformers dvd  1.0  False
826663114164 Transformers: The Compl... 100049  transformers dvd  2.0  False
708056579739 Nintendo - Transformers... 100049  transformers dvd  3.0  False
```

We've simulated many similar exploration sessions in the accompanying notebook. Since we don't have actual live users hitting our app, the simulation code is implemented just for demonstration purposes so we can integrate explore candidates for active learning within our automated LTR system. In a real production environment, you would be showing results to real users instead of simulating user sessions.

Each session adds a random exploration candidate based on listing 12.9, simulates whether the added exploration result was clicked, and appends it to a new set of sessions: `sessions_with_exploration`. Recall that these sessions serve as the input we need to compute LTR training data (the SDBN-based judgments from chapter 11 that are generated in listing 12.1).

Finally, we have the data needed to rerun our automated LTR training loop. We'll see what happens with these examples added to our training data and how we can fit this exploration into the overall automated LTR algorithm.

12.3.3 Examining the outcome of our explorations

We've explored by showing some outside-the-box search results to (simulated) live users. We now have new sessions appended to the original session data stored in the `sessions_with_exploration` dataframe. In this section, we'll run the session data through our automated LTR functions to regenerate training data and train a model. We'll then run this new model in an A/B test to see the results.

As you'll recall, our automated LTR helpers can regenerate training data using the `generate_training_data` function. We do this in listing 12.10, but this time with our augmented sessions that include exploration data.

Listing 12.10 Regenerating SDBN judgments from new sessions

```
query = "transformers dvd"
sessions_with_exploration = generate_simulated_exploration_sessions(
  query, sessions, logged_transformers_judgments, explore_features)
training_data_with_exploration = \
  generate_training_data(sessions_with_exploration)
display(training_data_with_exploration.loc["transformers dvd"])
```

Output:

```
doc_id        product_name                click  examined  grade  beta_grade
97360724240   Transformers: Revenge of... 43     44        0.977  0.833333
826663114164  Transformers: The Comple... 42     44        0.954  0.814815
97360722345   Transformers/Transformer... 46     55        0.836  0.738462
97363455349   Transformers - Widescree... 731    2113      0.345  0.345266
97361312804   Transformers - Widescree... 726    2109      0.344  0.343558
```

We see in listing 12.10's output that a new product has been included. Note specifically the addition of 826663114164, *Transformers: The Complete Series [25th Anni- versary Matrix of Leadership Edition] [16 Discs] - DVD*. Interestingly, this movie has `has_promotion=true`, meaning it was one of the newly selected explore candidates from the previous section:

```
{"upc": "826663114164",
 "name": "Transformers: The Complete Series [25th Anniversary ...] - DVD",
 "manufacturer": "",
 "short_description": "",
 "long_description": "",
 "has_promotion": True}
```

It seems users are drawn to promoted products, so let's move our `has_promotion` feature from the explore feature set to our main model and retrain to see the effect. In the following listing, we train a model with this new feature added to the mix to see the effect.

Listing 12.11 Rebuilding model using updated judgments

```
promotion_feature_set = [
  ltr.generate_fuzzy_query_feature(feature_name="name_fuzzy",
                                   field_name="name"),
  ltr.generate_bigram_query_feature(feature_name="name_bigram",
                                    field_name="name"),
  ltr.generate_bigram_query_feature(feature_name="short_description_bigram",
                                    field_name="short_description"),
  ltr.generate_query_feature(feature_name="has_promotion",  #1
                             field_name="has_promotion",  #1
                             value="true",  #1
                             constant_score=True)]  #1

evaluation = train_and_evaluate_model(sessions_with_exploration,
                                      "ltr_model_variant_3",
                                      feature_set)
display(evaluation)
```

#1 Adding has_promotion to the feature set we're training our model with

Evaluation for `ltr_model_variant_3`:

```
{"dryer": 0.12737002598513025,     # Before: 0.071
 "blue ray": 0.08461538461538462,          # 0.0
 "headphones": 0.12110565745285455,        # 0.065
 "dark of moon": 0.1492224251599605,       # 0.258
 "transformers dvd": 0.26947504217124457}  # 0.101
```

Wow! When comparing listing 12.11 to the earlier output from listing 12.3, we see that adding a promoted product to the training data creates a significant improvement in most cases in our offline test evaluation. The precision of `transformers dvd`, in particular, jumped significantly! If we issue a search for `transformers dvd`, we see this reflected in our data.

Listing 12.12 Searching for `transformers dvd` using the latest model

```
results = ltr.search_with_model("ltr_model_variant_3",
                                query="transformers dvd",
                                rerank_query="transformers dvd",
                                limit=5)["docs"]
display([doc["name"] for doc in results])
```

Output:

```
["Transformers/Transformers: Revenge of the Fallen: Two-Movie Mega Coll...",
 "Transformers: Revenge of the Fallen - Widescreen - DVD",
 "Transformers: Dark of the Moon - Original Soundtrack - CD",
 "Transformers: The Complete Series [25th Anniversary Matrix of Leaders...",
 "Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii"]
```

However, we know great-looking test results don't always translate to the real world. What happens when we rerun the A/B test from listing 12.4? If you recall, we created an `a_b_test` function that randomly selects a model for a user's search. If the results contained an item the user secretly wanted to purchase, a purchase would likely occur. If we use this function to resimulate an A/B test, we see our new model appears to have hit the jackpot!

Listing 12.13 Rerunning A/B test on new `ltr_model_variant_3` model

```
results = simulate_user_a_b_test(query="transformers dvd",
                                 model_a="ltr_model_variant_1",
                                 model_b="ltr_model_variant_3",
                                 number_of_users=1000)
display(results)
```

Output:

```
{"ltr_model_variant_1": 21,
 "ltr_model_variant_3": 145}
```

Now we see that the new model (`ltr_model_variant_3`) has significantly outperformed the old model (`ltr_model_variant_1`) in the A/B test. We now know that not only did our exploration help us find a theoretical gap in the training data, but that when we tested the new model in a real-world scenario for our target query (`transformers dvd`), it performed significantly better than the old "exploit-only" model. While we focused on a particular query in this chapter, the same process can be applied across many queries and exploration candidates to continue to automatically refine your LTR model with active learning.

We've now implemented an automated LTR system that not only relearns from the latest user signals, but that uses active learning to explore what else might be relevant to live users and then collects the corresponding signals measuring their feedback. This active learning process helps remove blind spots in the training data on an ongoing basis.

12.4 Exploit, explore, gather, rinse, repeat: A robust automated LTR loop

With the final pieces in place, we see how exploring new features helps us to overcome presentation bias. Feature exploration and training data exploration go hand-in-hand, as we learn our presentation biases by understanding the features we lack and may need to engineer into search. In this chapter, we used a simple example with "promotions", but what other, more complex, features might show blind spots in your training data? In this section, let's conclude by augmenting our automated LTR algorithm from chapter 11 to include not just training a model with previous click-model-based training data, but also this new active learning approach to explore beyond the training data's current scope.

Our new auto-exploring automated LTR algorithm can be summarized in these three main steps:

1. *Exploit* —Use known features and train the LTR model for ranking using existing training data.
2. *Explore* —Generate hypothesized, "explore" features to eliminate training data blind spots.
3. *Gather* —With a deployed model and a trained `gpr` model, show explore/exploit search results, and gather clicks to build judgments.

We can summarize the past three chapters by combining them into listing 12.14, shown next. This listing puts all the pieces together (with some internals omitted). Our main decision points in this algorithm are the features used to explore and exploit. We can also go under the hood to change the chosen click model, the LTR model architecture, and our risk tolerance (the `theta` parameter).

Listing 12.14 Summarizing the fully automated LTR algorithm

```
def train_and_deploy_model(sessions, model_name, feature_set):
  judgments = generate_training_data(sessions)
  train, test = split_training_data(judgments, 0.8)
  train_ranksvm_model(train, model_name, feature_set=feature_set)

def ltr_retraining_loop(latest_sessions, iterations=sys.maxsize,
                        retraining_frequency=60 * 60 * 24):  #1
  exploit_feature_set = get_exploit_feature_set()  #2
  train_and_deploy_model(latest_sessions,  #2
                         "exploit",  #2
                         exploit_feature_set)  #2
  for i in range(0, iterations):  #3
    judgments = generate_training_data(latest_sessions)
    train, test = split_training_data(judgments)
    if i > 0:
      previous_explore_model_name = f"explore_variant_{i-1}"
      exploit_model_evaluation = evaluate_model(test_data=test,
        model_name="exploit", training_data=train)
      explore_model_evaluation = evaluate_model(test_data=test,
        model_name=previous_explore_model_name, training_data=train)
      print(f"Exploit evaluation: {exploit_model_evaluation}")
      print(f"Explore evaluation: {explore_model_evaluation}")
      if is_improvement(explore_model_evaluation,  #4
                        exploit_model_evaluation):  #4
        print("Promoting previous explore model")
        train_and_deploy_model(latest_sessions,
                               "exploit",
                               explore_feature_set)

    explore_feature_set = get_latest_explore_feature_set()  #5
    train_and_deploy_model(latest_sessions,  #5
                           f"explore_variant_{i}",  #5
                            explore_feature_set)  #5

    wait_for_more_sessions(retraining_frequency)  #6
    latest_sessions = gather_latest_sessions(  #7
                        "transformers dvd",  #7
                        latest_sessions,  #7
                        explore_feature_set)  #7

ltr_retraining_loop(sessions)
```

#1 Returns the model once once per day
#2 Trains the LTR model on known good features and current training data
#3 Collects the new sessions and repeats the process
#4 Evaluates the current explore variant and promotes it if it's better than the current explore model
#5 Hypothesizes new features to explore for blind spots
#6 Collects user signals until it is time to retrain the model
#7 Collects the new sessions and repeats the process

In this loop, we capture a better automated LTR process. We actively learn our training data's blind spots by theorizing the features that might be behind them. In letting the loop run, we can observe its performance and decide when to promote "explore" features to the full production "exploit" feature set. As we retire old click data, we can also note when old features no

longer matter and when new features become important due to trends and seasonality. Our implementation uses a hand-tuned exploit-and-explore feature set to keep our search relevance team in control of feature engineering, but you could certainly write an algorithm to generate new features or use deep learning to discover latent features or use some other approach based on existing content attributes.

Taken together, these algorithms provide a robust mechanism for approaching an ideal ranking, considering a full spectrum of options that could be shown to users. They let you choose new features to investigate blind spots, arriving at a relevance algorithm that maximizes what users prefer to see in their search results.

Summary

- Performing well in an offline test shows that our features can approximate the training data. However, that's no guarantee of success. An A/B test can show us situations where the training data itself was misleading.
- Training data must be monitored for biases and carefully corrected.
- Presentation bias is one of search's most pernicious relevance problems. Presentation bias happens when our models can't learn what's relevant from user clicks because the results never show up to be clicked in the first place.
- We can overcome presentation bias by making the automated LTR process an active participant in finding blind spots in the training data. Models that do this participate in active learning.
- A Gaussian process is one way to select promising opportunities for exploration. Using a set of features, we can find what's missing in the training and select new items to show to users based on which items are likely to provide the most useful new data points for continued learning. We can experiment with different ways of describing the data via features to find new and interesting blind spots and areas of investigation.
- When we put together exploiting existing knowledge with exploring blind spots, we have a more robust, automated LTR system—reflected intelligence that can automatically explore and exploit features with little internal maintenance.

Previous chapter
< 11 Automating learning to rank with click models

Next chapter
Part 4 The search frontier >