

8 Signals-boosting models

This chapter covers

- Aggregating user signals to create popularity-based ranking model
- Normalizing signals for noisy query input
- Fighting signal spam in crowdsourced signals
- Applying time decays to prioritize recent signals
- Blending multiple signal types together into one model
- Choosing query-time versus index-time boosting

In chapter 4, we covered three different categories of reflected intelligence: signals boosting (popularized relevance), collaborative filtering (personalized relevance), and learning to rank (generalized relevance). In this chapter, we'll dive deeper into the first of these, implementing signals boosting to enhance the relevance ranking of your most popular queries and documents.

In most search engines, a relatively small number of queries tend to make up a large portion of the total query volume. These popular queries, called *head queries*, also tend to lead to more signals (such as clicks and purchases in an e-commerce use case), which enable stronger inferences about the popularity of top search results.

Signals-boosting models directly harness these stronger inferences and are the key to ensuring your most important and highest-visibility queries are best tuned to return the most relevant documents.

8.1 Basic signals boosting

In section 4.2.2, we built our first signals-boosting model on the RetroTech dataset, enabling a significant boost in relevance for the most frequently searched and clicked search results. In this section, we'll quickly recap the process of creating a simple signals-boosting model, which we'll build upon in the upcoming sections to tackle some more advanced needs.

You'll recall from section 4.2.2 that signals-boosting models aggregate useful user behavioral signals on documents (such as click signals) that occur as the result of a specific query. We used a search for `ipad` and boosted each document based on how many times it was previously clicked in the results for that search. Figure 8.1 demonstrates the before (no signals boosting) and after (signals boosting on) search results for the query `ipad`.

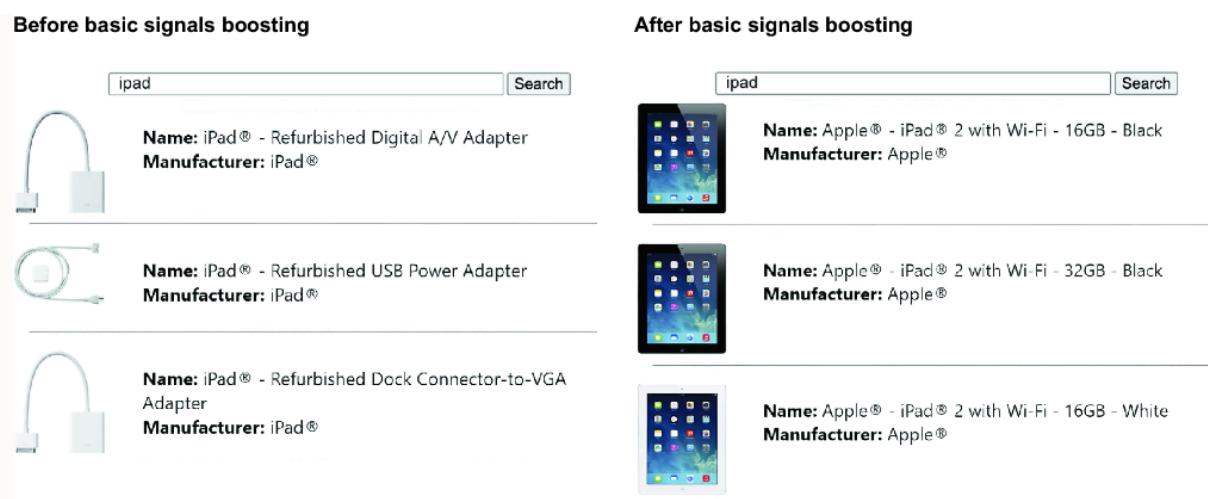


Figure 8.1 Before and after applying a signals-boosting model. Signals boosting improves relevance by pushing the most popular items to the top of the search results.

The signals-boosting model that led to the improved relevance in figure 8.1 is a basic signals-boosting model. It looks at each document ever clicked for a given query and applies a boost equal to the total number of past clicks on that document for that query.

While this basic signals-boosting model covered in section 4.2.2 provides greatly improved relevance, it is unfortunately susceptible to some data biases and even manipulation. In section 8.2, we'll discuss some techniques for removing noise in the signals to maximize the quality of your signals-boosting models and reduce the likelihood of undesirable biases.

8.2 Normalizing signals

It is important to normalize incoming user queries prior to aggregation so that variations are treated as the same query. Given that end users can enter any arbitrary text as a query, the aggregated signals are inherently noisy. The basic signals-boosting model from chapter 4 (and recapped in section 8.1) does no normalization. It generates aggregated boosts for each query and document pair, but since incoming queries haven't been normalized into a common form, variations of a query will be treated as entirely separate queries. The following listing produces a list of top queries that boost the most popular iPad model in their search results.

Listing 8.1 Aggregating signals and retrieving relevant queries

```
def create_boosting_collection(collection_name):
    basic_signals_aggregation_query = """ #1
    SELECT q.target AS query, c.target AS doc, #1
    COUNT(c.target) AS boost #1
    FROM signals c LEFT JOIN signals q #1
    ON c.query_id = q.query_id #1
    WHERE c.type = 'click' AND q.type = 'query' #1
    GROUP BY q.target, doc #1
    ORDER BY boost DESC #1
    """

    collection = engine.get_collection(collection_name) #2
    return aggregate_signals(collection, "basic_signals_boosts", #2
                             basic_signals_aggregation_query) #2

def search_for_boosts(query, collection, query_field="query"): #3
    boosts_request = {"query": query, #3
                      "query_fields": [query_field], #3
                      "return_fields": ["query", "doc", "boost"], #3
                      "limit": 20, 3((COL-16)) #3
                      "order_by": [("boost", "desc")]} #3
    response = collection.search(**boosts_request) #3
    return response["docs"] #3

signals_boosting_collection = create_boosting_collection("signals")
query = "885909457588" #4
signals_docs = search_for_boosts(query, signals_boosting_collection, "doc") #5
show_raw_boosted_queries(signals_docs) #5
```

#1 Defines the signals aggregation query

#2 Runs the aggregation from the signals collection to the basic_signals_boosts collection

#3 Loads signals boosts for the specified query and collection

#4 The most popular iPad model

#5 Returns the list of top signals boosts for the specified document

Output:

```
Raw Boosted Queries
"iPad" : 1050
"ipad" : 966
"Ipad" : 829
"iPad 2" : 509
"ipad 2" : 347
"Ipad2" : 261
"ipad2" : 238
"Ipad 2" : 213
"I pad" : 203
"i pad" : 133
"IPad" : 77
"Apple" : 76
"I pad 2" : 60
"apple ipad" : 55
"Apple iPad" : 53
"ipads" : 43
"tablets" : 42
"apple" : 41
"iPads" : 38
"i pad 2" : 38
```

This listing aggregates all signals by their queries and stores each query along with the number of occurrences into a new collection. You can see from the output that many variations of the same queries exist in the basic signals-boosting model. The biggest culprit of the variations seems to be case-sensitivity, as we see `iPad`, `ipad`, `Ipad`, and `IPad` as common variants. Spacing appears to be another problem, with `ipad 2` versus `i pad 2` versus `ipad2`. We even see singular versus plural representations in `ipad` versus `ipads`.

Keyword search fields usually normalize queries to be case-insensitive, use stemming to ignore plural versions of terms, and split on case changes and letter-to-number transitions between words. It is likewise useful to normalize signals, as keeping separate query terms and boosts for variations that are non-distinguishable by the search engine can be counterproductive. Failing to normalize terms diffuses the value of your signals, since the signals are divided across variations of the same keywords with lower boosts, as opposed to being coalesced into more meaningful queries with stronger boosts.

It is up to you to figure out how sophisticated your query normalization should be prior to signals aggregation, but even just lowercasing incoming queries to make the signals aggregation case-insensitive can go a long way. The following listing demonstrates the same basic signals aggregation as before, but this time with the queries lowercased first.

Listing 8.2 Case-insensitive signals aggregation

```
normalized_signals_aggregation_query = """
SELECT LOWER(q.target) AS query,  #1
c.target AS doc, COUNT(c.target) AS boost  #2
FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
WHERE c.type = 'click' AND q.type = 'query'
GROUP BY LOWER(q.target), doc  #2
ORDER BY boost DESC
"""

normalized_collection = \
    aggregate_signals(signals_collection, "normalized_signals_boosts",
                      normalized_signals_aggregation_query)

query = "885909457588"  #3
signals_documents = search_for_boosts(query, normalized_collection, "doc")
show_raw_boosted_queries(signals_documents)
```

#1 Normalizing case by lowercasing each query

#2 Grouping by normalized query increases the count of signals for those queries, increasing the signals boost

#3 The most popular iPad model

Output:

```
Raw Boosted Queries
"ipad" : 2939
"ipad 2" : 1104
"ipad2" : 540
"i pad" : 341
"apple ipad" : 152
"ipads" : 123
"apple" : 118
"i pad 2" : 99
"tablets" : 67
"tablet" : 61
"ipad 1" : 52
"apple ipad 2" : 27
"hp touchpad" : 26
"ipaq" : 20
"i pad2" : 19
"wi" : 19
"apple computers" : 18
"apple i pad" : 15
"ipad 2 16gb" : 15
"samsung galaxy" : 14
```

The list of raw boosted queries is already looking much cleaner! Not only is there less redundancy, but notice that the strength of the signals boosts has increased, because more signals are being attributed to a canonical form of the query (the lowercased version).

Lowercasing the queries and maybe removing whitespace or extraneous characters is often sufficient normalization for queries prior to aggregating signals. The important takeaway

from this section, though, is that the signals-boosting model becomes stronger the more you can ensure that identical queries are treated as the same when aggregated.

Variations in queries aren't the only kind of noise we need to worry about in our data. In the next section, we'll talk about how we can overcome significant potential problems caused by spam in our user-generated click signals.

8.3 Fighting signal spam

Anytime we use crowdsourced data, such as click signals, to influence the behavior of the search engine, we need to ask ourselves "How might the data inputs be manipulated to create an undesirable result?" In this section, we'll demonstrate how the search engine can be spammed with click signals to manipulate search results, and how you can stop it.

8.3.1 Using signal spam to manipulate search results

Let's imagine we have a user who, for whatever reason, really hates *Star Wars* and thinks that the most recent movies are complete garbage. They feel so strongly, in fact, that they want to ensure any searches for `star wars` return a physical trash can for purchase as the top search result. This user knows a thing or two about search engines and has noticed that your killer relevance algorithms seem to be using user signals and signals boosting. Figure 8.2 shows the default response for the query `star wars`, with signals boosting bringing the most popular products to the top of the search results.

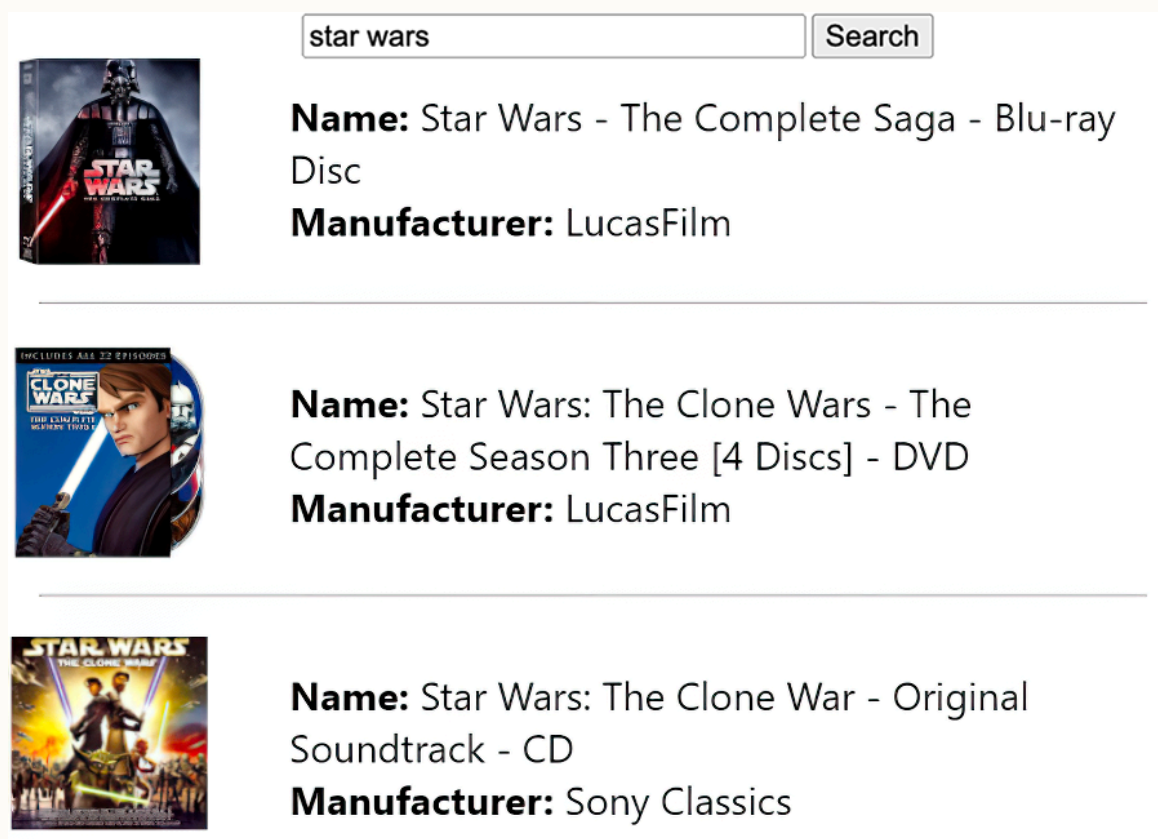


Figure 8.2 The most popular search results for the query `star wars`, with signals boosting turned on. These are the expected results when there is no malicious signal spam.

The user decides that since your search engine ranking is based upon popular items, they will spam the search engine with a bunch of searches for `star wars`. They will follow up each search with fake clicks on the Star Wars-themed trash can they found, attempting to make the trash can show up at the top of the search results.

In order to simulate this scenario, we'll run a simple script in the following listing to generate 5,000 queries for `star wars` and 5,000 corresponding clicks on the trash can after running that query.

Listing 8.3 Generating spam query and click signals

```
signals_collection = engine.get_collection("signals")
spam_user = "u8675309"
spam_query = "star wars"
spam_signal_boost_doc_upc = "45626176" #1

signal_docs = []
for num in range(5000): #2
    query_id = f"u8675309_0_{num}" #2
    query_signal = { #2
        "query_id": query_id, #2
        "user": spam_user, #2
        "type": "query", #2
        "target": spam_query, #2
        "signal_time": datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ"), #2
        "id": f"spam_signal_query_{num}" #2
    }
    click_signal = { #2
        "query_id": query_id, #2
        "user": spam_user, #2
        "type": "click", #2
        "target": spam_signal_boost_doc_upc, #2
        "signal_time": datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ"), #2
        "id": f"spam_signal_click_{num}" #2
    }
    signal_docs.extend([click_signal, query_signal]) #2

signals_collection.add_documents(signal_docs) #2

spam_signals_collection = \ #3
    aggregate_signals(signals_collection, "signals_boosts_with_spam", #3
        normalized_signals_aggregation_query) #3
```

#1 Document for the trash can the spammer wants to move to the top of the search results

#2 Generates and sends 5,000 query and click signals to the search engine

#3 Generates and sends 5,000 query and click signals to the search engine

#4 Runs the signals aggregation to generate the signals-boosting model including the spammy signals

Listing 8.3 sends thousands of spammy query and click signals to our search engine, modeling the same outcome we would see if a user searched and clicked on a particular search result thousands of times. The listing then reruns the basic signals aggregation.

To see the effect of the malicious user's spammy click behavior on our search results, the following listing runs a search for the query `star wars`, now incorporating the manipulated signals-boosting model.

Listing 8.4 Search results affected by spam user signals


```
def boosted_product_search_request(query, collection, boost_field=None):
    signals_documents = search_for_boosts(query, collection) #1
    signals_boosts = create_boosts_query(signals_documents) #1
    boosted_request = product_search_request(query) #2
    if boost_field: #2
        signals_boosts = (boost_field, signals_boosts) #2
    boosted_request["query_boosts"] = signals_boosts #2
    return boosted_request

query = '"star wars"'
boosted_request = boosted_product_search_request(query,
                                                spam_signals_collection, "upc")
response = products_collection.search(**boosted_request)
display_product_search(query, response["docs"])
```


#1 Loads signals boosts from the signals-boosting model that included the spammy signals

#2 Boosts the star wars query using the signals-boosting model


Figure 8.3 shows the new manipulated search results generated from listing 8.4, with the Star Wars trash can in the top spot.



Name: Trash Can (Star Wars Themed)
Manufacturer: Jay Franco & Sons



Name: Star Wars - The Complete Saga - Blu-ray Disc
Manufacturer: LucasFilm



Name: Star Wars: The Clone Wars - The Complete Season Three [4 Discs] - DVD
Manufacturer: LucasFilm

Figure 8.3 Search results manipulated by a user spamming the search engine with fake signals to affect the top result. The user was able to modify the top result just by clicking on it many times.

The spammer was successful, and these manipulated search results will now be seen by every subsequent visitor who searches for `star wars` on the RetroTech website! Looks like we're going to need to make our signals-boosting model more robust to combat this kind of signal spam.

8.3.2 Combating signal spam through user-based filtering

If you are going to use crowdsourced data like user signals to influence the search engine ranking, it's important to take steps to minimize users' ability to manipulate your signals-based ranking algorithm.

To combat the Star Wars trash can problem we just demonstrated, the simplest technique is to ensure that duplicate clicks by the same user only get one “vote” in the signals-boosting aggregation. That way, whether a malicious user clicks one time or a million times, their clicks only count as one signal and therefore have no material effect on the signals-boosting model. The following listing reworks the signals aggregation query to only count unique click signals from each user.

Listing 8.5 Deduplicating noisy user signals

```
anti_spam_aggregation_query = ""
SELECT query, doc, COUNT(doc) AS boost FROM (
  SELECT c.user unique_user, LOWER(q.target) AS query, c.target AS doc,
    MAX(c.signal_time) AS boost  #1
  FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
  WHERE c.type = 'click' AND q.type = 'query'
  GROUP BY unique_user, LOWER(q.target), doc)  #2
GROUP BY query, doc
ORDER BY boost DESC""

anti_spam_collection = \
  aggregate_signals(signals_collection, "signals_boosts_anti_spam",
    anti_spam_aggregation_query)
```

#1 Signal date is the most recent signal from the user if there are duplicates.

#2 Group by user to limit each user to one “vote” per query/doc pair in the signals-boosting model.

If we rerun the `star wars` query from listing 8.3 with this new `signals_boosts_anti_spam` model, we'll see that our normal search results have returned and look the same as in figure 8.2. This is because the extra, spammy signals from our malicious user have all been reduced to a single bad signal, as shown in table 8.1.

Table 8.1 The 5,000 spammy signals have been deduplicated to 1 signal in the antispam signals-boosting model.

model	query	doc	boost
Before spam signals (normalized_signals_boosts)	star wars	400032015667	0 (no signals yet)
After spam signals (normalized_signals_boosts)	star wars	400032015667	5000
After spam signals cleanup (signals_boosts_anti_spam)	star wars	400032015667	1

You can see that the aggregated signals count in the `signals_boosts_anti_spam` model has a total much closer to the `normalized_signals_boosts` model that we built before the spam signals were generated. Since each user is limited to one signal per query/document pair in the `signals_boosts_anti_spam` model, the ability for users to manipulate the signals-boosting model is now substantially reduced.

You could, of course, identify any user accounts that appear to be spamming your search engine and remove their signals entirely from your signals-boosting aggregation, but reducing the reach of the signals through deduplication is simpler and often accomplishes the same end goal of restoring a good, crowdsourced relevance ranking.

In listing 8.5, we used user IDs as the key identifier to deduplicate spammy signals, but any identifier will work here: user ID, session ID, browser ID, IP address, or even some kind of browser fingerprint. As long as you find some value to uniquely identify users or to otherwise identify low-quality traffic (like bots and web scrapers), you can use that information to deduplicate signals. If none of those techniques work, and you have too much noise in your click signals, you can also choose to only look at click signals from known (authenticated) users who you can presumably be much more confident are legitimate traffic.

One final way to mitigate signal spam is to find a way to separate the important signal types from the noisy ones that can be easily manipulated. For example, generating signals from running queries and clicking on search results is easy. Signals from purchasing a product are much harder to manipulate, as they require users to log in or enter payment information before a purchase will be recorded. The odds of someone maliciously purchasing 5,000 Star Wars trash cans are quite low, because there are multiple financial and logistical barriers to doing this.

Not only is it valuable to weight purchases as stronger signals than clicks from the standpoint of fighting spam, it is also valuable from a relevance standpoint, because purchases are more clear indicators of intent. In the next section, we'll walk through how we can combine different signal types into a signals-boosting model that considers the relative importance of each different signal type.

8.4 Combining multiple signal types

Thus far we've only worked with two signal types—queries and clicks. For some search engines (such as web search engines), click signals may be the only good source of crowdsourced data available to build a signals-boosting model. Many different signal types exist, however, which can provide additional and often much better inputs for building a signals-boosting model.

In our RetroTech dataset, we have several kinds of signals that are common to e-commerce use cases:

- query
- click
- add-to-cart
- purchase

While clicks in response to queries are helpful, they don't necessarily imply a strong interest in the product, as someone could just be browsing to see what's available. If someone adds a product to their shopping cart, this typically represents a much stronger signal of interest than a click. A purchase is an even stronger signal that a user is interested in a product, as the user is willing to pay money to receive the item for which they searched.

While some e-commerce websites may receive enough traffic to ignore click signals entirely and only focus on add-to-cart and purchase signals, it is often more useful to include all signal types when calculating signals boosts. Thankfully, combining multiple signal types is as simple as assigning relative weights as multipliers to each signal type when performing the signals aggregation:

```
signals_boost = (1 * sum(click_signals)) +  
                (10 * sum(add_to_cart_signals)) +  
                (25 * sum(purchase_signals))
```

By counting each click as 1 signal, each add-to-cart as 10 signals, and each purchase as 25 signals, a purchase carries 25 times as much weight and an add-to-cart carries 10 times as much weight as a click in the signals-boosting model. This helps reduce noise from less reliable signals and boosts more reliable signals while still making use of the large volume of less reliable signals in cases where better signals are less prevalent (like new or obscure items).

The following listing demonstrates a signals aggregation designed to combine different signal types with different weights.

Listing 8.6 Combining multiple signal types with different weights

```
mixed_signal_types_aggregation_query = ""  
SELECT query, doc, ((1 * click_boost)  #1  
    + (10 * add_to_cart_boost) +  #1  
    (25 * purchase_boost)) AS boost FROM (  #1  
SELECT query, doc,  
    SUM(click) AS click_boost,  #2  
    SUM(add_to_cart) AS add_to_cart_boost,  #2  
    SUM(purchase) AS purchase_boost FROM (  #2  
    SELECT lower(q.target) AS query, cap.target AS doc,  
        IF(cap.type = 'click', 1, 0) AS click,  
        IF(cap.type = 'add-to-cart', 1, 0) AS add_to_cart,  
        IF(cap.type = 'purchase', 1, 0) AS purchase  
    FROM signals cap LEFT JOIN signals q on cap.query_id = q.query_id  
    WHERE (cap.type != 'query' AND q.type = 'query')  
    ) raw_signals  
GROUP BY query, doc) AS per_type_boosts""  
  
type_weighted_collection = \  
    aggregate_signals(signals_collection, "signals_boosts_weighted_types",  
        mixed_signal_types_aggregation_query)
```

#1 Multiple signals are combined with different relative weights to calculate a total boost value.

#2 Each signal type is summed independently before being combined.

You can see from the SQL query that the overall boost for each query/document pair is calculated by counting all clicks with a weight of 1, counting all add-to-cart signals and multiplying them by a weight of 10, and counting all purchase signals and multiplying them by a weight of 25.

These suggested weights of 10x for add-to-cart signals and 25x for purchase signals should work well in many e-commerce scenarios, but these relative weights are also fully configurable for each domain. Your website may be set up such that almost everyone who adds a product to their cart purchases the product (for example, a grocery store delivery app, where the only purpose of using the website is to fill a shopping cart and purchase). In these cases, you could find that adding an item to a shopping cart adds no additional value, but that *removing* an item from a shopping cart should potentially carry a penalty, indicating that the product is a bad match for the query.

In this case, you may want to introduce the idea of *negative signals boosts*. Just as we’ve discussed clicks, add-to-carts, and purchases as signals of user intent, your user experience may also have numerous ways to measure user dissatisfaction with your search results. For example, you might have a thumbs-down button or a remove-from-cart button, or you may be able to track product returns after a purchase. You may even want to count documents in the search results that were skipped over and record a “skip” signal for those documents to indicate that the user saw them but didn’t show interest. We’ll cover the topic of managing clicked versus skipped documents in chapter 11 when we discuss click modeling.

Thankfully, handling negative feedback is just as easy as handling positive signals: instead of just assigning increasingly positive weights to signals, you can also assign increasingly negative weights to negative signals. Here’s an example:

```
positive_signals = (1 * sum(click_signals)) +  
                  (25 * sum(purchase_signals)) +  
                  (10 * sum(add_to_cart_signals)) +  
                  (0.025 * sum(seen_doc_signals))  
  
negative_signals = (-0.025 * sum(skipped_doc_signals)) +  
                  (-20 * sum(remove_from_cart_signals)) +  
                  (-100 * sum(returned_item_signals)) +  
                  (-50 * sum(negative_post_about_item_in_review_signals))  
  
type_based_signal_weight = positive_signals + negative_signals
```

This simple, linear function provides a highly configurable signals-based ranking model, taking in multiple input parameters and returning a ranking score based on the relative weights of those parameters. You can combine as many useful signals as you want into this weighted signals aggregation to improve the robustness of the model. Of course, tuning the weights of each of the signal types to achieve an optimal balance may take some effort. You can do this manually, or you can use a machine learning technique called learning to rank to do this. We’ll explore learning to rank in chapters 10 and 11.

Not only is it important to weight different kinds of signals relative to each other, but it can also sometimes be necessary to weight the *same* kind of signals differently against each other.

In the next section, we'll discuss one key example of doing this: assigning higher value to more recent interactions.

8.5 Time decays and short-lived signals

Signals don't always maintain their usefulness indefinitely. In the last section, we showed how signals-boosting models can be adjusted to weight different kinds of signals as more important than others. In this section, we'll address a different challenge—factoring in the “temporal value” of signals as they age and become less useful.

Imagine three different search engine use cases:

- An e-commerce search engine with stable products
- A job search engine
- A news website

For an e-commerce search engine, like RetroTech, the documents (products) often stay around for years, and the best products are often those that have a long track record of interest.

In a job search engine, the documents (jobs) may only stick around for a few weeks or months until the job is filled, and then they disappear forever. While the documents are present, however, newer clicks or job applications aren't necessarily any more important as signals than older interactions.

In a news search engine, while the news articles stick around forever, newer articles are generally much more important than older articles, and newer signals are also more important than older signals, as people's interests change on a daily, if not hourly, basis.

Let's dive into these use cases and demonstrate how to best handle the time-sensitivity of signals when performing signals boosting.

8.5.1 Handling time-insensitive signals

In our RetroTech use case, our documents are intentionally old, having been around for a decade or more, and interest in them likely only increases as the products become older and more “retro”. As a result, we don't often have massive spikes in popularity for items, and newer signals don't necessarily carry significantly more importance than older signals. This type of use case is a bit atypical, but plenty of search use cases do deal with “static” document sets like this. The best solution in this case is the strategy we've already taken in this chapter: process all signals within a reasonable period of months or years and give them equal weight. When all time periods carry the same weight, the signals-boosting model likely doesn't need to be rebuilt very often, since the model changes slowly over time. The frequent processing of signals is unnecessary computational overhead.

In a job search use case, however, the scenario is very different. For the sake of argument, let's say that on average it takes 30 days to fill a job opening. This means the document representing that job will only be present in the search engine for 30 days, and any signals collected for that document are only useful for signals boosting during that 30-day window. When a job is posted, it will typically be very popular for the first few days, since it is new and is likely to attract many existing job seekers, but all interactions with that job at any point during the 30 days are just as useful. In this case, all click signals should get equal weight, and all job ap-

plication signals should likewise receive an equal weight (at a weight higher than the click signals). Given the very short lifetime of the documents, however, it is important that all signals be processed as quickly as possible to make the best use of their value.

Use cases with short-lived documents, like in the job search use case, aren't usually the best candidates for signals boosting, as the documents may be deleted by the time the signals-boosting model becomes any good. As a result, it can often make more sense to look at personalized models (like collaborative filtering, covered in chapter 9) and generalizable relevance models (like learning to rank, covered in chapters 10 and 11) for these use cases.

In both the RetroTech and the job search use cases, the signals were just as useful for the entire duration of the document's existence. In the news search use case, which we'll look at next, the value of the signals declines over time.

8.5.2 Handling time-sensitive signals

In a news search engine use case, the most recently published news usually gets the most interaction, so more recent signals are considerably more valuable than older signals. Some news items may be very popular and relevant for days or longer, but generally the signals from the last ten minutes are more valuable than the signals from the last hour, which are more valuable than the signals from the last day, and so on. News search is an extreme use case where signals both need to be processed quickly and where more recent signals need to be weighted as substantially more important than older signals.

One easy way to model this is by using a decay function, such as a half-life function, which cuts the weight assigned to a signal by half (50%) over equally spaced time spans. For example, a decay function with a half-life of 30 days would assign 100% weight to a signal that happens "now", 75% weight to a signal from 15 days ago, 50% weight to a signal from 30 days ago, 25% weight to a signal from 60 days ago, 12.5% weight to a signal from 90 days ago, and so on. The math for calculating a signal's time-based weight using a decay function is

```
starting_weight × 0.5(signal_age / half_life)
```

When applying this calculation, the `starting_weight` will usually be the relative weight of a signal based upon its type, such as a weight of 1 for clicks, 10 for add-to-cart signals, and 25 for purchase signals. If you are not combining multiple signal types, then the `starting_weight` will just be 1.

The `signal_age` is how old the signal is, and the `half_life` is how long it takes for the signal to lose half of its value. Figure 8.4 demonstrates how this decay function affects signal weights over time for different half-life values.

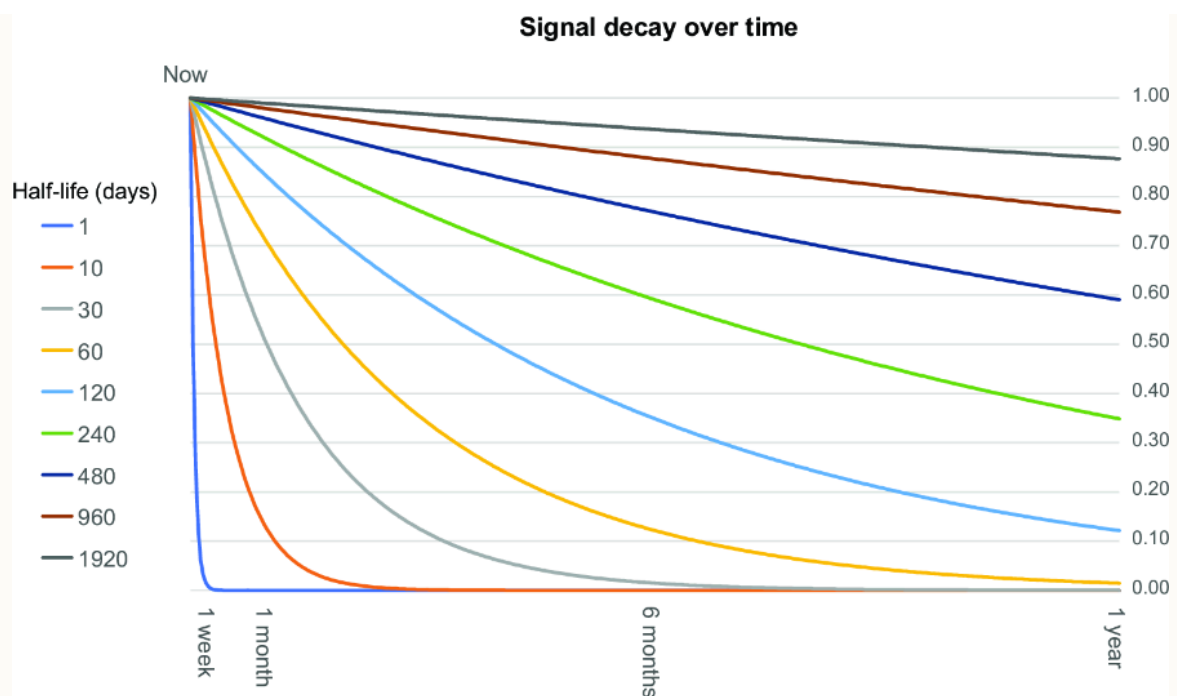


Figure 8.4 Signal decay over time based upon various half-life values. As the half-life increases, individual signals maintain their boosting power for longer.

The 1-day half-life is very aggressive and is impractical in most use cases, as it is unlikely you would be able to collect enough signals in a day to power meaningful signals boosting, and the likelihood of your signals becoming irrelevant that quickly is low.

The 30-day, 60-day, and 120-day half-lives are effective at aggressively discounting older signals while still keeping residual value from the discounted signals over a six to twelve month period. If you have very long-lived documents, you could push out even longer, making use of signals over the course of many years. The following listing demonstrates an updated signal aggregation query that implements a half-life of 30 days for each signal.

Listing 8.7 Applying time decay to the signals-boosting model

```
half_life_days = 30
target_date = '2024-06-01' #1
signal_weight = 1 #2

time_decay_aggregation = f"""
SELECT query, doc, sum(time_weighted_boost) AS boost FROM (
  SELECT user, query, doc, {signal_weight} * #3
    POW(0.5, (DATEDIFF('{target_date}', signal_time) / #3
      {half_life_days})) #3
  AS time_weighted_boost FROM (
    SELECT c.user AS user, lower(q.target) AS query, c.target AS doc,
      MAX(c.signal_time) as signal_time #4
    FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
    WHERE c.type = 'click' AND q.type = 'query'
      AND c.signal_time <= '{target_date}' #5
    GROUP BY c.user, q.target, c.target #4
  ) AS raw_signals
  ) AS time_weighted_signals
GROUP BY query, doc
ORDER BY boost DESC"""

time_weighted_collection = \
  aggregate_signals(signals_collection, "signals_boosts_time_weighted",
    time_decay_aggregation)
```

#1 The latest possible signal date. This should be now() in a live system, but it can be set to a fixed date for a frozen dataset like RetroTech.

#2 A function could be added here to differentiate weights for different signal types.

#3 The half-life calculation

#4 Gets the most recent unique signal per user, query, and product combination

#5 Only includes signals up to the target date

This decay function has a few unique configurable parameters:

- It contains a `half_life_days` parameter, which calculates a weighted average using a configurable half-life, which we've set as 30 days to start.
- It contains a `signal_weight` parameter, which can be replaced with a function returning a weight by signal type, as shown in the last section ("click" = 1, "add-to-cart" = 10, "purchase" = 25, etc.).
- It contains a `target_date` parameter, which is the date at which a signal gets the full value of 1. Any signals before this date will be decayed based on the half-life, and any signals after this date will be ignored (filtered out).

Your `target_date` will usually be the current date, so that you are making use of your most up-to-date signals and assigning them the highest weight. However, you could also apply it to past periods if your documents have seasonal patterns that repeat monthly or yearly.

While our product documents don't change very often, and the most recent signals aren't necessarily any more valuable than older signals, there are potentially annual patterns we could find in a normal e-commerce dataset. For example, certain types of products may be more popular around major holidays like Mother's Day, Father's Day, and Black Friday. Likewise, searches for something like a "shovel" may take on a different meaning in the summer (shov-

el for digging dirt) versus the winter (shovel for removing snow from the sidewalk). If you explore your signals, any number of trends may emerge for which time sensitivity should affect how your signals are weighted.

Ultimately, signals are a lagging indicator. They reflect what your users just did, but they are only useful as predictions of future behavior if the patterns learned are likely to repeat themselves.

Having now explored techniques for improving our signals models through query normalization, mitigating spam and relevance manipulation, combining multiple signal types with different relative weights, and applying time decays to signals, you should be able to flexibly implement the signals-boosting models most appropriate for your use case. When rolling out signals boosting at scale, however, there are two different approaches you can take to optimize for flexibility versus performance, which we'll cover in the next section.

8.6 Index-time vs. query-time boosting: Balancing scale vs. flexibility

All the signals-boosting models in the chapter have been demonstrated using *query-time boosting*, which loads signals boosts for each query from a separate sidecar collection at query time and modifies the query to add the boosts before sending it to the search engine. It's also possible to implement signals-boosting models using *index-time boosting*, where boosts are added directly to documents for the queries to which those boosts apply. In this section, we'll show you how to implement index-time boosting and discuss the benefits and tradeoffs of query-time boosting versus index-time boosting.

8.6.1 Tradeoffs when using query-time boosting

Query-time boosting, as we've seen, turns each query into a two-step process. Each incoming user query is looked up in a signals-boosting collection, and any boosted documents found are used to modify the user's query. Query-time boosting is the most common way to implement signals boosting, but it comes with benefits and drawbacks.

BENEFITS OF QUERY-TIME BOOSTING

Query-time boosting's primary architectural characteristic is that it keeps the main search collection (`products`) and the signals-boosting collection (`*_signals_boosts`) separate. This separation provides several benefits:

- It allows the signals for each query to be updated incrementally by only modifying the one document representing that query.
- It allows boosting to be turned on or off easily by just not doing a lookup or modifying the user's query.
- It allows different signals-boosting algorithms to be swapped in at any time.

Ultimately, the flexibility to change signals boosts at any point in time based on the current context is the major advantage of query-time signals boosting. This enables easier incorporation of real-time signals and easier experimentation with different ranking functions.

DRAWBACKS OF QUERY-TIME BOOSTING

Although it's flexible, query-time boosting also introduces some significant downsides affecting query performance, scale, and relevance, which may make it inappropriate for certain use cases:

- It requires an extra search to look up boosts before the boosted search is executed, adding more processing (executing two searches) and latency (the final query must wait on the results of the signals lookup query before being processed).
- It leads to unfortunate tradeoffs between relevance (boosting all relevant documents) and scalability (limiting the number of boosted documents to keep query times and query throughput reasonable).
- It makes pagination inefficient and potentially inaccurate, as increasing the number of boosts to account for the increasing document offset while paging through results will slow the query down and possibly result in documents being pushed to earlier pages (missed by the user) or later pages (seen as duplicates by the user).

The first downside is straightforward, as each query essentially becomes two queries executed back-to-back, increasing the total search time. The second downside may not be as obvious. In query-time boosting, we look up a specific number of documents to boost higher in the search results for a query. In our `ipad` search example from figure 8.1 (see listing 4.7 for the code), the boost for the query ultimately becomes

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

This boost contains 10 documents, but only because that is the number of boosts we requested. Assuming we only showed ten documents on the first page, the whole first page will look good, but what if the user navigates to page 2? In this case, no boosted documents will be shown, because only the first 10 documents with signals for the query were boosted!

To boost documents for the second page, we would need to ensure we have at least enough document boosts to cover the full first two pages, which means increasing from 10 boosts to 20 boosts (modifying the `limit` parameter to 20 on the boost lookup query):

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109
"843404073153"^73 "635753493559"^62 "885909457601"^62 "885909472376"^61
"610839379408"^29 "884962753071"^28 "635753490879"^27 "885909457632"^26
"885909393404"^26 "716829772249"^23 "821793013776"^21 "027242798236"^15
"600603132827"^14 "886111271283"^14 "722868830062"^13 "092636260712"^13
```

You can mostly solve this problem by increasing the number of boosts looked up every time someone navigates to the “next” page, but this will aggressively slow down subsequent queries, as page 3 will require looking up and applying 30 boosts, page 10 will require 100 boosts, and so on. For a use case where only a small number of boosted documents exist for each query, this is not a big problem, but for many use cases, there may be hundreds or thousands of documents that would benefit from being boosted. In our query for `ipad`, there are more than 200 documents that contain aggregated signals, so most of those documents will

never be boosted unless someone pages very deep into the search results. At that point, the queries are likely to be slow and could even time out.

Only including a subset of the boosts presents another problem: search results aren't always strictly ordered by the boost value! We've assumed that requesting the top 10 boosts will be enough for the first page of 10 results, but the boost is only one of the factors that affects relevance. It could be that documents further down in the boost list have a higher base relevance score and that if their boosts were also loaded, they would jump up to the first page of search results.

As a result, when a user navigates from page 1 to 2 and the number of boosts loaded increases, unwanted reranking may occur, where results might jump up to page 1 and never be seen or jump down to page 2 and be seen again as duplicates.

Even if these results are much more relevant than search results without signals boosting applied, it doesn't make for an optimal user experience. Index-time signals boosting can help overcome these drawbacks, as we'll show in the next section.

8.6.2 Implementing index-time signals boosting

Index-time signals boosting turns the signals-boosting problem on its head—instead of boosting popular documents for queries at query time, we boost popular queries for documents at indexing time. This is accomplished by adding popular queries to a field in each document, along with their boost value. Then, at query time, we simply search on the new field, and if the field contains the term from our query, it gets automatically boosted based on the boost value indexed for the term.

When implementing index-time boosting, we use the exact same signals aggregations to generate pairs of documents and boost weights for each query. Once those signals boosts have been generated, we just have to add one additional step to our workflow: updating the products collection to add a field to each document containing each term for which the document should be boosted, along with that term's associated signals boost. The following listing demonstrates this additional step.

Listing 8.8 Indexing boosts into the main product collection

```
from aips.data_loaders import index_time_boosts

boosts_collection = engine.get_collection("normalized_signals_boosts")
create_view_from_collection(boosts_collection, #1
                             boosts_collection.name) #1

boosted_products_collection = \
    engine.get_collection("products_with_signals_boosts")
create_view_from_collection(boosted_products_collection, #2
                             boosted_products_collection.name) #2

boosted_products = index_time_boosts.load_dataframe( #3
    boosted_products_collection, #3
    boosts_collection) #3

boosted_products_collection.write(boosted_products) #4
```

#1 Loads a previously generated signals-boosting model

#2 Registers the product table so we can load from it and save back to it with boosts added

#3 Inserts all keywords with signals boosts for each document into a new `signals_boosts` field on the document

#4 Saves the products back to the boosted products collection, with the updated `signals_boosts` added

The code reads all previously generated signals boosts for each product document and then maps the queries and boosts into a new `signals_boosts` field on that document. The `signals_boosts` field has a comma-separated list of terms (user queries) with a corresponding weight for each term.

When using Solr as your search engine (the default), this `signals_boosts` field is a specialized field containing a `DelimitedPayloadBoostFilter` filter, which allows for terms (queries) to be indexed with associated boosts that can be used to influence query-time scoring. For example, for the most popular iPad, the product document will now be modified to look as follows:

```
{...
  "id": "885909457588",
  "name": "Apple® - iPad® 2 with Wi-Fi - 16GB - Black"
  "signals_boosts": "ipad|2939,ipad 2|1104,ipad2|540,i pad|341,apple ipad|
    152,ipads|123,apple|118,i pad 2|99,tablets|67,..."
  ...
}
```

This format for specifying index-time term boosts will vary from search engine to search engine. At query time, this `signals_boosts` field will be searched upon, and if the query is present in the field, the relevance score for that document will be boosted by the indexed payload for the matched query. The following listing demonstrates how to perform a query utilizing index-time signals boosts.

Listing 8.9 Ranking search results with index-time boosts

```
def get_boosted_search_request(query, boost_field):
    request = product_search_request(query)
    request["index_time_boost"] = (boost_field, query)  #1
    return request

query = "ipad"
boosted_query = get_boosted_search_request(query, "signals_boosts")
response = boosted_products_collection.search(**boosted_query)
display_product_search(query, response["docs"])
```

#1 Boosting the relevance score based upon the indexed signals boosts for the query

While support for querying on index-time boosted terms is handled differently by various search engines, in the case of Solr (our default search engine), this translates internally into a `boost` parameter value of `payload("signals_boosts", "ipad", 1, "first")` being added to the search request to boost documents by the payload attached to the *first* match of the query `ipad` in the `signals_boosts` field (or `1` if no payload was indexed). See section 3.2 if you want a refresher on influencing ranking like this through functions and multiplicative boosts.

Figure 8.5 shows the results of this index-time signals boosting. As you can see, the results now look similar to the query-time signals-boosting output shown previously in figure 8.1.

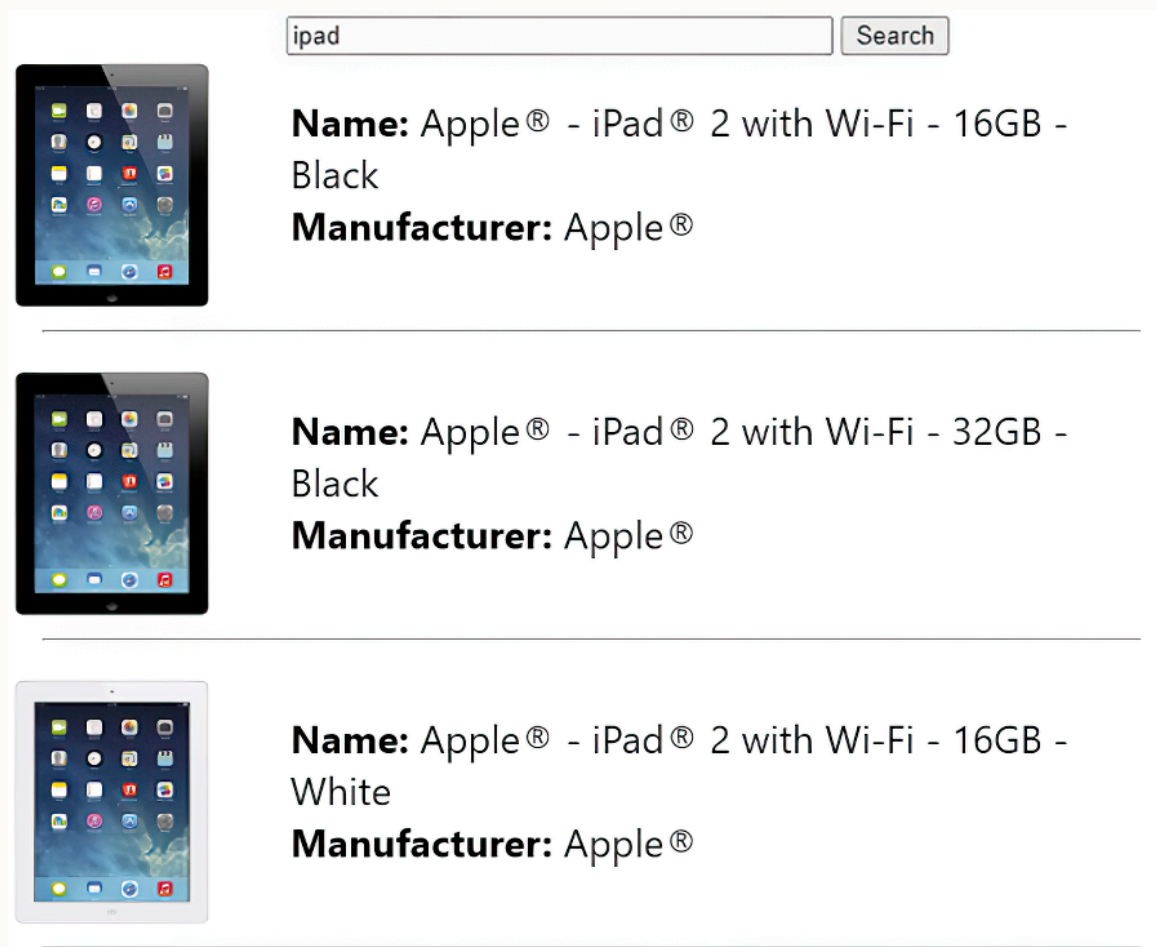


Figure 8.5 Index-time signals boosting, demonstrating similar results as query-time index boosting

The relevance scores will likely not be identical when performing index-time boosting versus query-time boosting, as the math is different when scoring an indexed payload versus a

boosted query term. The relative ordering of results should be very similar, though. The index-time signals boosting will also apply to all documents with a matching signals-boost payload, whereas query-time signals boosting only applies to the top-*N* documents that are explicitly boosted in the query.

8.6.3 Tradeoffs when implementing index-time boosting

Index-time boosting, as we've seen, moves most of the work of performing signals boosting from the query execution phase to the indexing phase of your searches. This solves some of the problems inherent in query-time boosting, but it also introduces some new challenges, which we'll discuss in this section.

BENEFITS OF INDEX-TIME BOOSTING

Index-time boosting solves most of the drawbacks of query-time boosting:

- The query workflow is simpler and faster because it doesn't require doing two queries—one to look up the signals boosts and another to run a boosted query using those signals boosts.
- Each query is more efficient and faster as the number of boosted documents increases, because the query is a single keyword search against the `signals_boosts` field as opposed to a longer query containing the increasing list of boosted documents.
- Results paging is no longer a problem, because *all* documents matching the query are boosted, not just the top-*N* that can be efficiently loaded and added to the query.

Given these characteristics, index-time boosting can substantially improve the relevance and consistency of results ordering by ensuring all queries receive consistent and complete boosting of all their matching documents. It can also substantially improve query speed by removing query terms (the doc boosts) and eliminating extra lookups prior to execution of the main query.

DRAWBACKS OF INDEX-TIME BOOSTING

If index-time boosting solves all the problems of query-time boosting, why wouldn't we always use index-time signals boosting?

The main drawback of index-time boosting is that since the boost values for a query are indexed onto each document (each document contains the terms for which that document should be boosted), adding or removing a keyword from the signals-boosting model requires reindexing all documents associated with that keyword. If signals-boosting aggregations are updated incrementally (on a per-keyword basis), this means potentially reindexing all the documents within your search engine on a continuous basis. If your signals-boosting model is updated in batch for your entire index, this means potentially reindexing all your documents every time your signals-boosting model is regenerated.

This kind of indexing pressure adds operational complexity to your search engine. To keep query performance fast and consistent, given this indexing pressure, you may want to separate the indexing of documents onto separate servers from where the search indexes are hosted for serving queries.

SEPARATION OF CONCERNS: INDEXING VS. QUERYING

When performing high-volume indexing, it may be architecturally best to isolate your indexing servers from your querying servers if possible. Otherwise, memory or CPU pressure from busy indexing operations can affect query latency and throughput. Not all search engines support this separation of concerns between indexing and query servers, but many do.

Elasticsearch and OpenSearch, for instance, support this separation of concerns using the concept of *follower indexes*, while Solr does it through supporting different *replica types*. All three of these engines have the concepts of *shards*, which are partitions containing a subset of the documents in a collection, and *replicas*, which are exact copies of all the data belonging to their shard. Each shard has a leader that is responsible for receiving updates and forwarding them to all replicas.

By default, leaders send all document updates to every replica of a shard, and each replica then (redundantly) indexes the document so that it is available immediately to be searched on that replica. Unfortunately, the tradeoff for immediate availability is that high-volume indexing will consume resources on every replica, which can slow down query performance across the entire search engine.

By changing the replica type in Solr for replicas on query servers from `NRT` (near-real-time) to either `TLOG` (transaction log) or `PULL`, you'll instruct replicas to pull prebuilt index files from the shard's leader (which is already indexing documents) instead of performing duplicate indexing. Similarly in Elasticsearch and OpenSearch, if you configure a follower index, the servers hosting the follower index will replicate prebuilt index files from the leader index instead of redundantly indexing the document. Some other search engines and vector databases have similar abilities to segregate indexing operations and query operations across servers that you can explore.

If you plan to do index-time signals boosting and expect to be constantly reindexing signals at high volumes, you should strongly consider isolating index and query-time operations. This ensures your query performance isn't negatively affected by the significant additional indexing overhead from ongoing signals-boosting aggregations.

The other drawback of index-time boosting is that making changes to your signals-boosting function can require more planning. For example, if you want to change your weight for click versus purchase signals from 1:25 to 1:20, you'll need to create a `signals_boosts_2` field with the new weights, reindex all your documents adding the new boosts, and then flip over your query to use the new field instead of the original `signals_boosts` field. Otherwise, your boost values and ranking scores will fluctuate inconsistently until all your documents' scores have been updated.

If those drawbacks can be worked around, however, implementing index-time signals boosting can solve all the drawbacks of query-time signals boosting, leading to better query performance, full support for results paging, and the use of all signals from all documents as opposed to just a sampling from the most popular documents.

As we've seen in this chapter, signals boosting allows for popularized relevance—boosting the most important items for specific queries. In the next chapter, we'll implement personalized

relevance—adjusting ranking on a per-user basis, using each user’s signals (relative to other users) to learn their specific interests.

Summary

- Signals boosting is a type of ranking algorithm that aggregates user signal counts per query and uses those counts as relevance boosts for that query in the future. This ensures the most popular items for each query are pushed to the top of the search results.
- Normalizing queries by treating different variations (case, spelling, etc.) as the same query helps clean up noise in user signals and builds a more robust signals-boosting model.
- Crowdsourced data is subject to manipulation, so it is important to explicitly prevent spam and malicious signals from affecting the quality of your relevance models.
- You can combine different signal types into a single signals-boosting model by assigning relative weights to each type and doing a weighted sum of values across signal types. This enables you to give more relevance to stronger signals (positive or negative) and reduce noise from weaker signals.
- Introducing a time-decay function enables recent signals to carry more weight than older signals, allowing older signals to phase out over time.
- Signals-boosting models can be productionized using query-time signals boosting (more flexible) or index-time signals boosting (more scalable and more consistent relevance ranking).

Previous chapter

< [Part 3 Reflected intelligence](#)

Next chapter

[9 Personalized search](#) >