7 Interpreting query intent through semantic search

This chapter covers

- The mechanics of query interpretation
- Implementing an end-to-end query intent pipeline to parse, enrich, transform, and search
- Tagging and classifying query terms and phrases
- Augmenting queries using knowledge graph traversals
- Interpreting the semantics of domain-specific query patterns

In chapters 5 and 6, we used content and signals to interpret the domain-specific meaning of incoming user queries. We discussed phrase identification, misspelling detection, synonym discovery, query intent classification, related-terms expansion, and even query-sense disambiguation. We mostly discussed these techniques in isolation to demonstrate how they each work independently.

In this chapter, we'll put all those techniques into practice, integrating them into a unified query interpretation framework. We'll show an example search interface that accepts real queries, interprets them, rewrites them to better express the end user's intent, and then returns ranked results.

We should note that multiple paradigms have evolved for implementing semantic search, including embedding-based query interpretation and question answering (returning extracted or generated answers instead of documents) using large language models (LLMs) and pretrained Transformers. These typically involve encoding queries into vectors, searching for an approximate nearest neighborhood of vectors, and then performing a vector similarity calculation to rank documents. The ranked documents are often then analyzed to summarize, extract, or generate answers. We'll cover these LLM-based approaches for semantic search and question answering in chapters 13–15.

We'll focus in this chapter on the mechanics of integrating each of the AI-powered search strategies you've already learned to deliver an end-to-end semantic query pipeline. We'll implement the pipeline in four phases:

- *Parsing* the user's query
- *Enriching* the parsed query with improved context
- *Transforming* the query to optimize for relevance in our target search engine
- *Searching* using the optimized query

These steps don't have to be implemented linearly (sometimes they repeat, and sometimes steps can be skipped), and they can also be broken down further (for example, searching could be broken down into matching, ranking, and reranking). Having this consistent framework through which we can integrate any combination of AI-powered search techniques will nevertheless be invaluable as you mix and match approaches in your own search applications.

7.1 The mechanics of query interpretation

There is no one "right way" to build a query interpretation framework. Every organization building an intelligent search platform is likely to build something a bit different, depending on their business needs and the expertise of their search team. There are, however, some consistent themes across implementations worth exploring:

- *Pipelines*—Both when indexing documents and processing queries, it's useful to model all of the necessary parsing, interpretation, and ranking logic as modular stages in a workflow. This allows for easy experimentation by swapping out, rearranging, or adding processing stages within the pipeline at any time.
- *Models*—Whether you are fine-tuning a complex deep-learning-based LLM (chapters 13–14), a learning-to-rank model (chapters 10–12), a signals-boosting or personalization model (chapters 8–9), or a knowledge graph containing synonyms, misspellings, and related terms (chapters 5–6), proper query interpretation requires plugging in the right models in the right order within the indexing and query pipelines.
- *Conditional fallbacks*—You'll never be able to interpret every query perfectly. You could have many models helping interpret one query, while having no clue what another query means. It's usually best to start with a base or "fallback" model (usually keyword-based) that can handle any query imperfectly, and to layer in more sophisticated models on top, to enhance precision of interpretation. Additionally, if no results are found, it might be useful to return recommendations to ensure the searcher sees *something* that might be useful, even if it is not exactly what they were seeking.

Figure 7.1 shows an example query pipeline demonstrating each of these themes of combining pipeline stages, models, and conditional fallbacks.

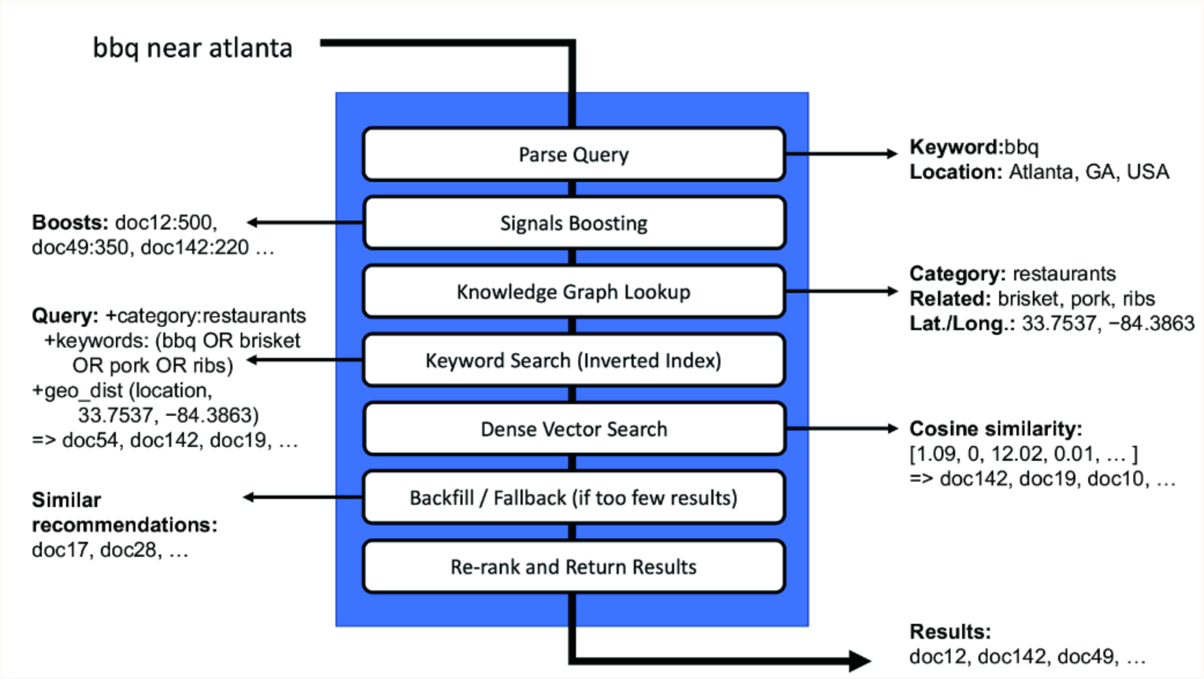

bbq near atlanta
Parse Query
Signals Boosting
Knowledge Graph Lookup
Keyword Search (Inverted Index)
Dense Vector Search
Backfill / Fallback (if too few results)
Re-rank and Return Results
Keyword:bbq
Location: Atlanta, GA, USA
Boosts: doc12:500, doc49:350, doc142:220 …
Category: restaurants
Related: brisket, pork, ribs
Lat./Long.: 33.7537, −84.3863
Query: +category:restaurants +keywords: (bbq OR brisket OR pork OR ribs) +geo_dist (location, 33.7537, −84.3863) => doc54, doc142, doc19, …
Cosine similarity: [1.09, 0, 12.02, 0.01, …] => doc142, doc19, doc10, …
Similar recommendations: doc17, doc28, …
Results: doc12, doc142, doc49, …

Figure 7.1 An example query interpretation pipeline

Figure 7.1 takes in the query `bbq near atlanta` and begins with a parse-query stage that performs entity extraction on known keywords, locations, or other known terms from the query. It then hits a signals-boosting stage, which checks with a signals-boosting model (which was introduced in chapter 4 and will be covered in much greater detail in chapter 8) to boost the most popular documents for the given query.

Three different, but complementary, approaches are commonly used to interpret individual keywords and relate them to each other, all of which are included in the example pipeline:

- *Lexical search*, such as BM25 ranking on Boolean query matches in an inverted index
- *Knowledge graph search*, such as ranking the entities found within a query and their relationships with the most similar entities in your index using a semantic knowledge graph (SKG) or explicitly built knowledge graph
- *Dense vector search*, such as cosine similarity of vectors found with an approximate nearest neighborhood of embeddings

Of the three, the most common "default" matching layer tends to be lexical search on an inverted index, as this approach allows matching on *any* term that exists in the corpus of documents, whether that term is understood or not. The knowledge graph and dense vector approaches both rely on being able to relate the terms in each query to concepts or entities, and this simply can't be done in all cases.

In fact, BM25 ranking often outperforms dense vector approaches on embeddings even from state-of-the-art LLMs unless those language models are initially trained or fine-tuned on domain-specific content (this may change over time, as pretrained LLMs continue to become more robust). We'll dive into using LLMs starting in chapters 9 and 13 for personalized search and semantic search, and we'll spend time fine-tuning and using LLMs for more advanced search functionality like question answering and generative search in chapters 14 and 15. We'll keep our focus in this chapter primarily on demonstrating the mechanics of integrating lexical search and knowledge graphs.

The figure 7.1 pipeline ends with a backfill/fallback stage, which can be useful for inserting additional results in the case that none of the previous stages were able to return a full result set. This can be as simple as returning recommendations instead of search results (covered in chapter 9), or it could involve returning a partially matched query with lower precision.

The final results of all pipeline stages are then combined and reranked as needed to yield a final set of relevance-ranked search results. The reranking stage can be simple, but it will often be implemented using *machine-learned ranking* through a ranking classifier. We'll dive into building and automating the training of learning to rank models in chapters 10–12.

While the example pipeline in this section may provide good results in many cases, the specific logic of a pipeline should always depend on the needs of your application. In the next section, we'll set up an application to search local business reviews, and then we'll implement a unified pipeline capable of semantic search over this domain.

7.2 Indexing and searching on a local reviews dataset

We’re going to create a search engine that aggregates product and business reviews from across the web. If a business has a physical location (restaurant, store, etc.), we want to find all of the reviews associated with the business and make them available for searching.

The following listing shows the ingestion of our crawled local reviews data into the search engine.

Listing 7.1 Loading and indexing the reviews dataset

```
reviews_collection = engine.create_collection("reviews")
reviews_data = reviews.load_dataframe("data/reviews/reviews.csv")
reviews_collection.write(reviews_data)
```

Output:

```
Wiping "reviews" collection
Creating "reviews" collection
Status: Success

Loading Reviews...
root
 |-- id: string (nullable = true)
 |-- business_name: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- content: string (nullable = true)
 |-- categories: string (nullable = true)
 |-- stars_rating: integer (nullable = true)
 |-- location_coordinates: string (nullable = true)

Successfully written 192138 documents
```

The data model for a review can be seen in the listing’s output. Each review has the business name, location information, review content, review rating (number of stars from 1 to 5), and categories of the type of entity being reviewed.

Once the data has been ingested, we can run a search. In this chapter, we provide a more interactive application than in prior chapters, launching a web server to power a dynamic search interface. Running listing 7.2 will launch the web server.

Listing 7.2 Launching the web server and loading the search page

```
start_reviews_search_webserver()

%%html
<iframe src="http://localhost:2345/search" width=100% height="800"/>
```

Figure 7.2 shows the loaded search interface from listing 7.2. You can run the embedded search page from the Jupyter notebook, but if you're running on your local computer on port `2345`, you can also just navigate to http://localhost:2345/search in your web browser to get a better experience.

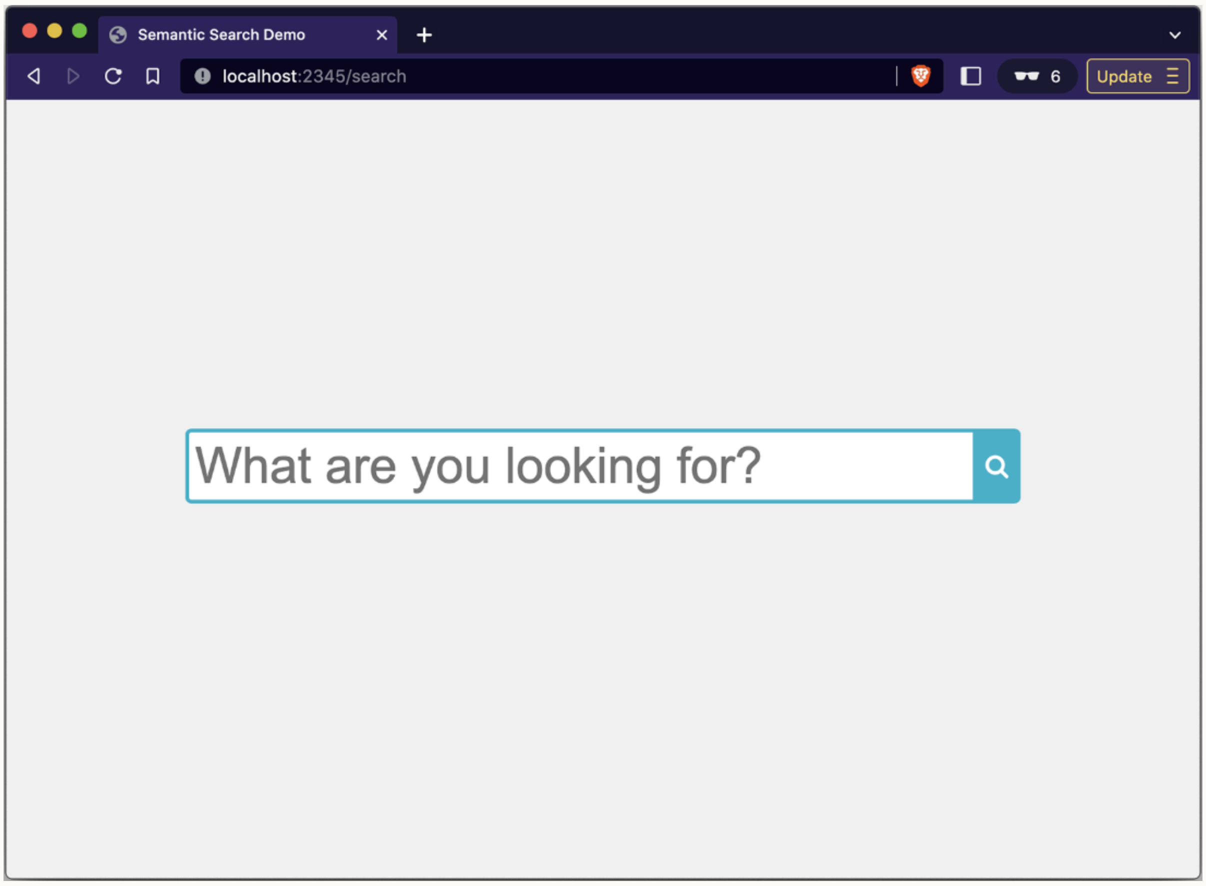
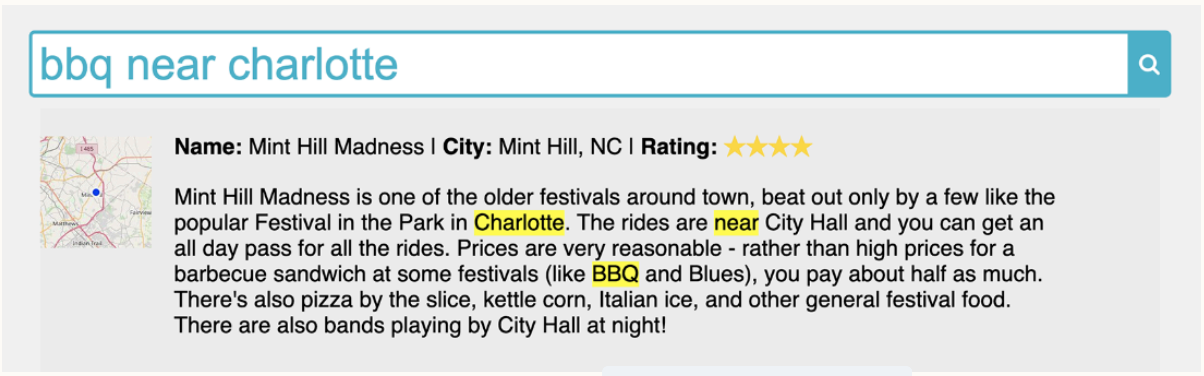


Figure 7.2 Visiting the review search page from a local web browser

Let's first try a simple query for `bbq near charlotte`. For now, let's pretend you haven't gone through the knowledge graph learning process (chapter 6) and don't yet know how to apply an SKG (chapter 5) to your query interpretation. In this case, we're just doing out-of-the-box lexical keyword matching. Figure 7.3 shows the top lexical search result for the query `bbq near charlotte`.



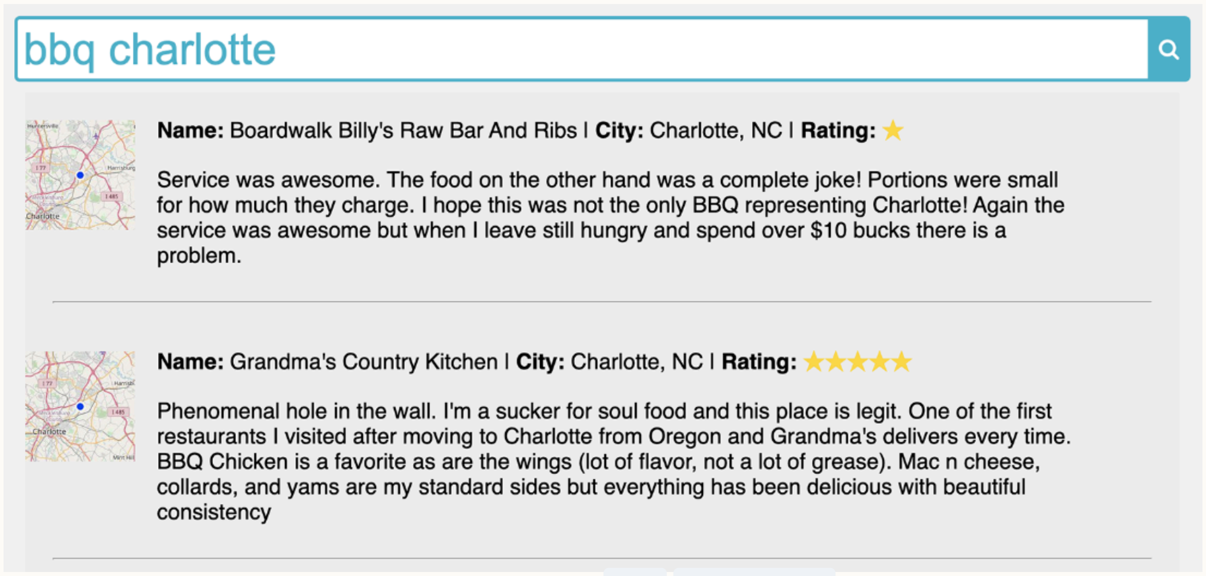
Name: Mint Hill Madness | **City:** Mint Hill, NC | **Rating:** ★★★★

Mint Hill Madness is one of the older festivals around town, beat out only by a few like the popular Festival in the Park in Charlotte. The rides are near City Hall and you can get an all day pass for all the rides. Prices are very reasonable - rather than high prices for a barbecue sandwich at some festivals (like BBQ and Blues), you pay about half as much. There's also pizza by the slice, kettle corn, Italian ice, and other general festival food. There are also bands playing by City Hall at night!

Figure 7.3 Basic lexical keyword search for `bbq near charlotte`, only matching keywords

In our reviews dataset, this is the only review that matches our query, even though multiple BBQ (also known as barbecue) restaurants exist in or near the city of Charlotte, NC, USA. The reason only this result is returned is that it's the only document containing all three terms (`bbq`, `near`, and `charlotte`). If you look at the review, it is not even for a restaurant that serves barbecue—it's actually for a festival whose review just happens to reference another festival with "BBQ" in the name!

The main problem here is that most relevant restaurants do not contain the word `near`. Figure 7.4 shows that more results do exist if we take out the word `near` and search instead for `bbq charlotte`.


bbq charlotte

Name: Boardwalk Billy's Raw Bar And Ribs | **City:** Charlotte, NC | **Rating:** ★

Service was awesome. The food on the other hand was a complete joke! Portions were small for how much they charge. I hope this was not the only BBQ representing Charlotte! Again the service was awesome but when I leave still hungry and spend over $10 bucks there is a problem.

Name: Grandma's Country Kitchen | **City:** Charlotte, NC | **Rating:** ★★★★★

Phenomenal hole in the wall. I'm a sucker for soul food and this place is legit. One of the first restaurants I visited after moving to Charlotte from Oregon and Grandma's delivers every time. BBQ Chicken is a favorite as are the wings (lot of flavor, not a lot of grease). Mac n cheese, collards, and yams are my standard sides but everything has been delicious with beautiful consistency

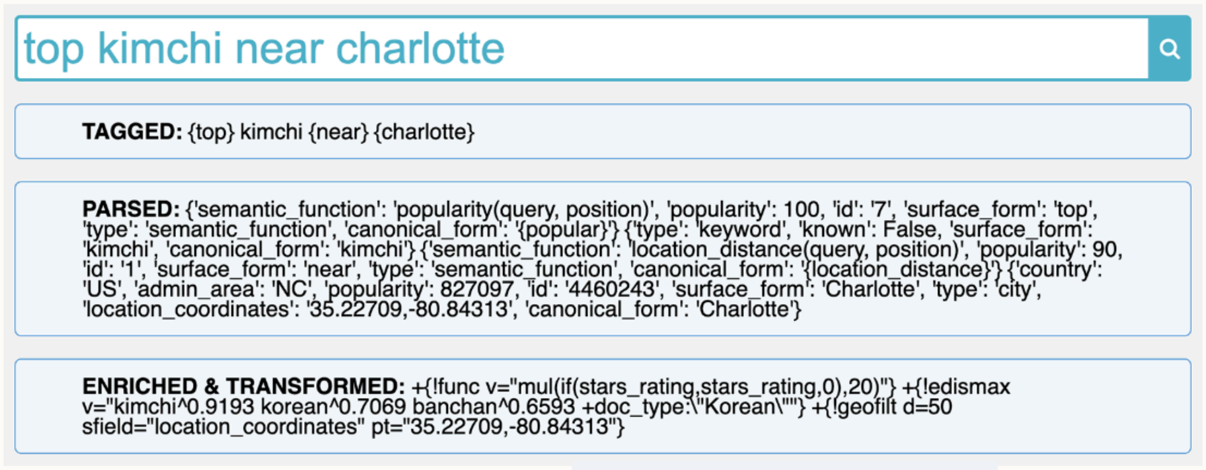
Figure 7.4 Basic lexical keyword search for `bbq charlotte`. More results matching with the word "near" removed.

Both of the top results contain the term "bbq", but the first one has a low (1-star) rating, and the second one mentions "bbq chicken" (chicken with barbecue sauce on it) but not "bbq" (barbecue), which would typically refer more to smoked meat like pulled pork, pulled chicken, rib, or brisket. Additionally, while the results are all in the city Charlotte, NC, this is only because they match the keyword `charlotte` in the review text, which means many good results are missing that didn't reference the city by name in the review. It is clear from the results that the search engine hasn't properly interpreted the user's query intent.

We can do so much better than this! You've already learned how to extract domain-specific knowledge and how to classify queries (for example, `bbq` implies a restaurant), so we just need to integrate these techniques and learned models end-to-end.

7.3 An end-to-end semantic search example

The last section showed the shortcomings of relying on pure keyword search. How might we improve upon the search engine's ability to interpret queries? Figure 7.5 demonstrates the results of a reasonably specific query that traditional keyword search would struggle to correctly interpret: `top kimchi near charlotte`.


top kimchi near charlotte

TAGGED: {top} kimchi {near} {charlotte}

PARSED: {'semantic_function': 'popularity(query, position)', 'popularity': 100, 'id': '7', 'surface_form': 'top', 'type': 'semantic_function', 'canonical_form': '{popular}'} {'type': 'keyword', 'known': False, 'surface_form': 'kimchi', 'canonical_form': 'kimchi'} {'semantic_function': 'location_distance(query, position)', 'popularity': 90, 'id': '1', 'surface_form': 'near', 'type': 'semantic_function', 'canonical_form': '{location_distance}'} {'country': 'US', 'admin_area': 'NC', 'popularity': 827097, 'id': '4460243', 'surface_form': 'Charlotte', 'type': 'city', 'location_coordinates': '35.22709,-80.84313', 'canonical_form': 'Charlotte'}

ENRICHED & TRANSFORMED: +{!func v="mul(if(stars_rating,stars_rating,0),20)"} +{!edismax v="kimchi^0.9193 korean^0.7069 banchan^0.6593 +doc_type:\"Korean\""} +{!geofilt d=50 sfield="location_coordinates" pt="35.22709,-80.84313"}

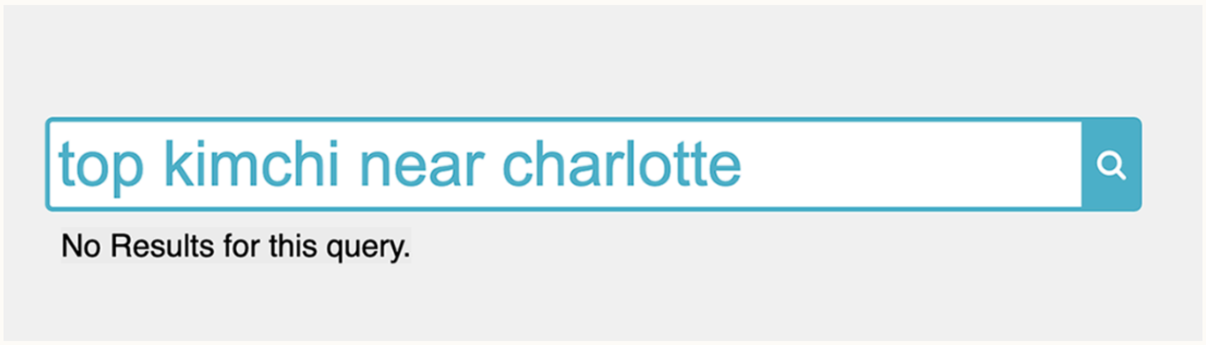
Figure 7.5 Semantic search for the query `top kimchi near charlotte`

This query is interesting, because only one keyword ("kimchi") actually contains a traditional keyword for relevance ranking purposes. The keyword "top" really means "most popular" or "highest rated", and the phrase "near charlotte" indicates a geographical filter to apply to search results. You can see in the figure that the original query is parsed as `{top} kimchi {near} {charlotte}`. We're using this curly-brace syntax to indicate that the terms "top", "near", and "charlotte" were all identified from our knowledge graph, whereas "kimchi" was not tagged and is, therefore, an unknown.

After these keywords and phrases are parsed, you can see that they are enriched and transformed into the following search-engine-specific syntax (Solr):

- *top*: `+{!func v="mul(if(stars_rating,stars_rating,0),20)"}`. This syntax will boost all documents based on their reviews (1 to 5 stars), multiplying by 20 to generate a score between 0 and 100.
- *kimchi*: `+{!edismax v="kimchi^0.9193 korean^0.7069 banchan^0.6593 +doc_type:\"Korean\""}`. This is an expansion of the unknown term "kimchi" using the SKG expansion approach from chapter 5. The SKG, in this case, identifies "Korean" as the category in which to filter results, and the top related terms to "kimchi" are "korean" and "banchan".
- *near charlotte*: `+{!geofilt d=50 sfield="location_coordinates" pt="35.22709,-80.84313"}`. This geographical filter limits results to documents only within a 50 km radius of the latitude/longitude of Charlotte, NC, USA.

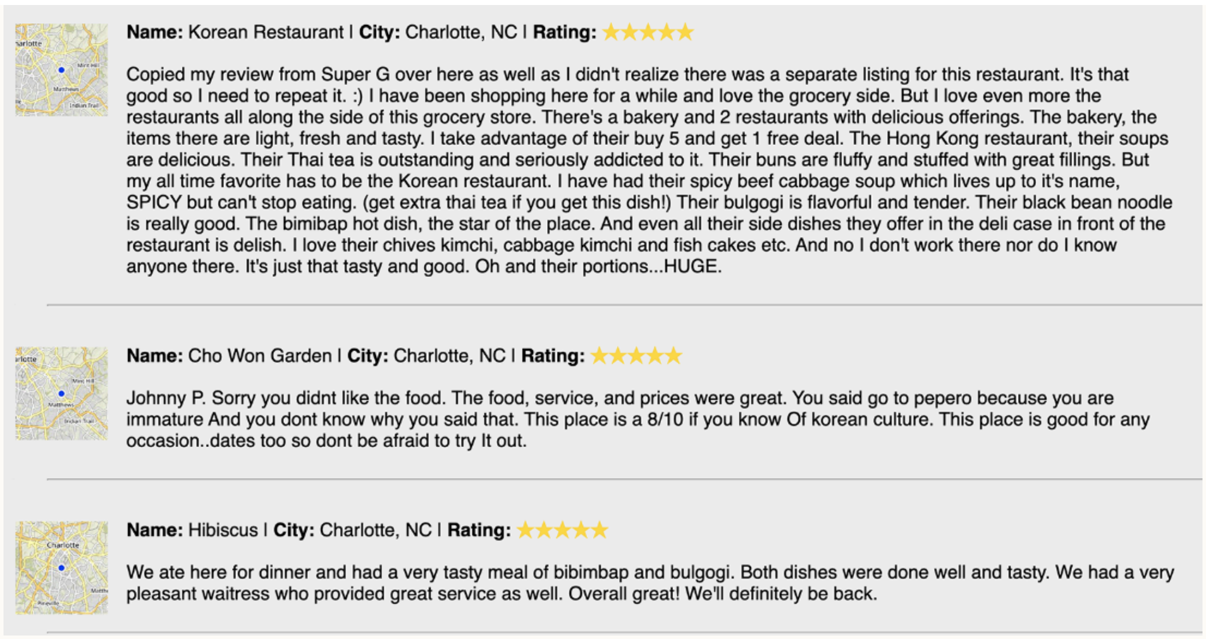
Had the original query been executed as a traditional lexical search *without* the query interpretation layer, no results would have matched, as shown in figure 7.6.



No Results for this query.

Figure 7.6 Traditional lexical search returns no results due to no documents containing all keywords

However, figure 7.7 demonstrates the results after executing the semantically parsed and enriched version.



Name: Korean Restaurant | **City:** Charlotte, NC | **Rating:** ★★★★★

Copied my review from Super G over here as well as I didn't realize there was a separate listing for this restaurant. It's that good so I need to repeat it. :) I have been shopping here for a while and love the grocery side. But I love even more the restaurants all along the side of this grocery store. There's a bakery and 2 restaurants with delicious offerings. The bakery, the items there are light, fresh and tasty. I take advantage of their buy 5 and get 1 free deal. The Hong Kong restaurant, their soups are delicious. Their Thai tea is outstanding and seriously addicted to it. Their buns are fluffy and stuffed with great fillings. But my all time favorite has to be the Korean restaurant. I have had their spicy beef cabbage soup which lives up to it's name, SPICY but can't stop eating. (get extra thai tea if you get this dish!) Their bulgogi is flavorful and tender. Their black bean noodle is really good. The bimibap hot dish, the star of the place. And even all their side dishes they offer in the deli case in front of the restaurant is delish. I love their chives kimchi, cabbage kimchi and fish cakes etc. And no I don't work there nor do I know anyone there. It's just that tasty and good. Oh and their portions...HUGE.

Name: Cho Won Garden | **City:** Charlotte, NC | **Rating:** ★★★★★

Johnny P. Sorry you didnt like the food. The food, service, and prices were great. You said go to pepero because you are immature And you dont know why you said that. This place is a 8/10 if you know Of korean culture. This place is good for any occasion..dates too so dont be afraid to try It out.

Name: Hibiscus | **City:** Charlotte, NC | **Rating:** ★★★★★

We ate here for dinner and had a very tasty meal of bibimbap and bulgogi. Both dishes were done well and tasty. We had a very pleasant waitress who provided great service as well. Overall great! We'll definitely be back.

Figure 7.7 Semantic search example returns relevant results by better interpreting and executing the query

The results look quite good! You'll notice that

- There are many results (instead of zero).
- All results have *top* ratings (5 stars).
- All results are in *Charlotte*.
- Some results match even without having the main keyword ("kimchi"), and they are clearly for Korean restaurants that would serve kimchi because similar terms were used in the review.

We'll spend the remainder of the chapter walking through how we can implement this level of semantic query interpretation, starting with a high-level query interpretation pipeline.

7.4 Query interpretation pipelines

While we often need to integrate multiple models and diverse query-understanding approaches in a query pipeline, most query pipelines share a similar set of high-level phases:

1. *Parsing*—Extracting key entities and their logical relationships from the query
2. *Enriching*—Generating an understanding of the query context, queried entities, and their semantic relationships
3. *Transforming*—Rewriting the user's query for the search engine to optimize recall and ranking
4. *Searching*—Executing the transformed query and returning ranked results

You can think of each phase as a different type of pipeline stage. As in the example pipeline in section 7.1, some pipelines may invoke multiple stages to parse or enrich a query, and some pipelines may even run multiple conditional searches and merge results.

In the coming subsections, we'll implement each phase to demonstrate the inner workings of our end-to-end semantic search example from section 7.3.

7.4.1 Parsing a query for semantic search

As you saw in section 3.2.5, most keyword search engines perform some form of Boolean parsing on incoming queries by default. The query `statue` `of` `liberty` thus becomes a query for `statue` `AND` `of` `AND` `liberty`, where any document containing all three words ("statue", "of", "liberty") will match, assuming a default query operator of `AND`.

This Boolean matching alone does not yield great results, but when paired with BM25 ranking (discussed in section 3.2.1), it can yield great results for a naive algorithm that doesn't have any true understanding of the terms within the domain.

In contrast to this Boolean parsing, it's also possible to convert an entire query into a numerical vector embedding, as discussed in section 3.1.1. We'll cover dense vector search using LLMs and embeddings later in chapters 13–14. One benefit of using LLMs and embedding-based query interpretation is that these techniques provide a better representation of the query as one unit of meaning. The logical structure of the query can sometimes be lost using this approach, though, so it may not work well for scenarios where your Boolean logic must be preserved or you must ensure certain keywords appear in the search results.

A final way to parse a query is by extracting the known terms and phrases from a knowledge graph. We took this approach in our end-to-end example in section 7.3. One benefit of this approach is that, in addition to offering fine-grained control over known vocabulary, it also allows for the interpretation of specific phrases and trigger words (`top`, `in`, `near`) to be modeled explicitly, to reflect their functional meaning instead of just keyword matching. The downside of this approach is that any terms or phrases *not* existing in the knowledge graph can't be as easily extracted and interpreted.

Since we'll dive into LLMs in later chapters, we'll focus in this chapter on explicit query parsing using a knowledge graph, as explicit parsing enables significant customization for a domain, it's cheap to implement, and it enables us to incorporate all the other AI techniques we've already learned.

IMPLEMENTING A SEMANTIC QUERY PARSER

The first step when semantically interpreting a query is identifying the terms and phrases in the query (the *parsing* phase). In chapter 6, we walked through how to identify important domain-specific terms and phrases from our content and user behavioral signals. These can be used as the known entities list for powering entity extraction on incoming queries.

Because there can be potentially millions of entities in the known phrases list, an efficient structure such as a *finite state transducer* (FST) makes it possible to perform entity extraction at this scale in just milliseconds. We won't get into the nuances of how FSTs work here, but they enable very compact compression of many term sequences and very quick lookups on those term sequences, enabling lightning-fast entity extraction.

Our example search engine, Apache Solr, implements a *text tagger* request handler, which is purpose-built for fast entity extraction. It enables you to index any number of terms into

a lookup index, so you can build that index into an FST and extract terms from that index on any incoming stream of text.
In chapter 6, we generated lists of domain-specific phrases, which also included alternative spellings. We can map all these terms into a specially configured `entities` collection, along with any spelling variations, to enable seamless entity extraction from incoming queries. The following listing explores several types of entity data within our `entities` collection.

Listing 7.3 Entity data used for tagging and extraction

```
entities_dataframe = from_csv("data/reviews/entities.csv", log=False)
display_entities(entities_dataframe, limit=20)
```

Output:

```
Entities
+---+--------------------+--------------------+-----------------+----------+
| id|        surface_form|      canonical_form|             type|popularity|
+---+--------------------+--------------------+-----------------+----------+
|  1|                near| {location_distance}|semantic_function|        90|
|  2|                  in| {location_distance}|semantic_function|       100|
|  3|                  by| {location_distance}|semantic_function|        90|
|  4|                  by|{text_within_one_...|semantic_function|        10|
|  5|                near|     {text_distance}|semantic_function|        10|
|  6|             popular|           {popular}|semantic_function|       100|
|  7|                 top|           {popular}|semantic_function|       100|
|  8|                best|           {popular}|semantic_function|       100|
|  9|                good|           {popular}|semantic_function|       100|
| 10|              violet|              violet|            color|       100|
| 11|       violet crowne|       violet crowne|            brand|       100|
| 12|violet crowne cha...|violet crowne cha...|    movie_theater|       100|
| 13|        violet crown|       violet crowne|            brand|       100|
| 14|violet crown char...|violet crowne cha...|    movie_theater|       100|
| 15|            haystack| haystack conference|            event|       100|
| 16|       haystack conf| haystack conference|            event|       100|
| 17| haystack conference| haystack conference|            event|       100|
| 18|            heystack| haystack conference|            event|       100|
| 19|       heystack conf| haystack conference|            event|       100|
| 20| heystack conference| haystack conference|            event|       100|
+---+--------------------+--------------------+-----------------+----------+
only showing top 20 rows

... Entities continued
+---+----------------------------------------------+
|id |semantic_function                             |
+---+----------------------------------------------+
|1  |location_distance(query, position)            |
|2  |location_distance(query, position)            |
|3  |location_distance(query, position)            |
|4  |text_within_one_edit_distance(query, position)|
|5  |text_distance(query, position)                |
|6  |popularity(query, position)                   |
|7  |popularity(query, position)                   |
|8  |popularity(query, position)                   |
|9  |popularity(query, position)                   |
+---+----------------------------------------------+
```

The fields represented within the table for entities in listing 7.3 include

- `surface_form`—The specific text of any spelling variation we want to match on future queries.
- `canonical_form`—The "official" version of any term that may have potentially multiple surface forms.
- `type`—A classification (category) for the term within our domain.
- `popularity`—Used to prioritize different meanings of the same surface form.

- `semantic_function`—Only present for entities of type “semantic_function”. This is used to inject programmatic handling of special combinations of keywords.

In most cases, the `surface_form` and `canonical_form` will be the same, but our entity extractor will always match on a `surface_form` and map it to a `canonical_form`, so this mechanism is used to map multiple variations of an entity’s spelling to one official or “canonical” version. This can be used to handle misspellings (“amin” ⇒ “admin”), acronyms and initialisms (“cto” ⇒ “chief technology officer”), ambiguous terms (“cto” ⇒ “chief technology officer” versus “cto” ⇒ “cancelled-to-order”), or even mapping terms to specific interpretation logic (semantic functions) like “near” ⇒ `{location_ distance}`.

The “semantic_function” type is a special type that we’ll explore in section 7.4.2; it enables nonlinear, conditional query parsing rules. For example, “if the word *near* is followed by an entity that has a *geographical location*, interpret that section of the query as a geography filter”.

In the event of an ambiguous term, multiple entries will exist containing the same surface form mapped to different canonical forms. In this case, the `popularity` field will specify a relative value indicating which meaning is more common (the higher, the more popular).

This format is also extensible—you can add a `vector` field representing the semantic meaning of a canonical form, or a `related_terms` field that contains other terms with similar meanings. This would enable caching a static representation of the meaning of the `canonical_form`, which can be much more efficient at query time than referencing external models or knowledge graphs for known terms on every request.

INVOKING THE ENTITY EXTRACTOR

In addition to the `reviews` collection created in listing 7.1, we also need to create an `entities` collection containing the known entities to be extracted. This collection will serve as an explicit knowledge graph containing all of the entities from listing 7.3, as well as a list of all major cities in the world. The next listing configures and populates the `entities` collection.

Listing 7.4 Creating the `entities` collection

```
entities_collection = engine.create_collection("entities")  #1
entities_dataframe = from_csv("data/reviews/entities.csv")
cities_dataframe = cities.load_dataframe("data/reviews/cities.csv")
entities_collection.write(entities_dataframe)  #2
entities_collection.write(cities_dataframe)  #2
```

#1 Creates the entities collection and configures it to hold our explicit knowledge graph of entities to extract from queries
#2 Explicit entities and city entities are indexed to the entities collection to be used for entity extraction.

Output:

```
Wiping "entities" collection
Creating "entities" collection
Status: Success
Loading data/reviews/entities.csv
Schema:
root
 |-- id: integer (nullable = true)
 |-- surface_form: string (nullable = true)
 |-- canonical_form: string (nullable = true)
 |-- type: string (nullable = true)
 |-- popularity: integer (nullable = true)
 |-- semantic_function: string (nullable = true)

Loading Geonames...
Successfully written 21 documents
Successfully written 137581 documents
```

One configuration point we should highlight is the setting up of entity extraction, which occurs within `engine.create_collection("entities")`. In the default case where Solr is being used to serve the explicit knowledge graph for entity extraction from queries, Solr's text tagger functionality is enabled by internally making the following configuration changes:

- Adding an `/entities/tag` endpoint using the `TaggerRequestHandler` in Solr. We can pass queries to this endpoint to perform entity extraction of any entities found in the `entities` collection.
- Adding a `tags` field type in the schema that is configured to use an in-memory FST, enabling compact and fast tagging from a collection of potentially millions of entities in milliseconds.
- Adding a `name_tag` field that the `surface_form` field is copied into. The `name_tag` field is a `tags` field type and is used by the `/entities/tag` endpoint to match entities from the query.

If your search engine has native text tagging capabilities, the configuration will differ, but the following listing shows the code corresponding to these changes for the default implementation of the text tagger using Apache Solr.

Listing 7.5 Configuring the Solr text tagger for entity extraction

```
add_tag_type_commands = [{
  "add-field-type": {
    "name": "tag",  #1
    "class": "solr.TextField", #1
    "postingsFormat": "FST50", #1
    "omitNorms": "true",
    "omitTermFreqAndPositions": "true",
    "indexAnalyzer": {
      "tokenizer": {"class": "solr.StandardTokenizerFactory"},
      "filters": [
        {"class": "solr.EnglishPossessiveFilterFactory"},
        {"class": "solr.ASCIIFoldingFilterFactory"},
        {"class": "solr.LowerCaseFilterFactory"},
        {"class": "solr.ConcatenateGraphFilterFactory",  #2
         "preservePositionIncrements": "false"}]},  #2
    "queryAnalyzer": {
      "tokenizer": {"class": "solr.StandardTokenizerFactory"},
      "filters": [{"class": "solr.EnglishPossessiveFilterFactory"},
                  {"class": "solr.ASCIIFoldingFilterFactory"},
                  {"class": "solr.LowerCaseFilterFactory"}]}}
  },

  {"add-field": {"name": "name_tag", "type": "tag",  #3
                 "stored": "false"}},  #3
  {"add-copy-field": {"source": "surface_form",  #4
                      "dest": ["name_tag"]}}] #4

add_tag_request_handler_config = {
  "add-requesthandler": {  #5
    "name": "/tag",  #5
    "class": "solr.TaggerRequestHandler",  #5
    "defaults": {
      "field": "name_tag",  #5
      "json.nl": "map",
      "sort": "popularity desc",  #6
      "matchText": "true",
      "fl": "id,surface_form,canonical_form,type,semantic_function,
      ↪popularity,country,admin_area,*_p"
    }}}
```

#1 The tag field type is configured using the Lucene FST50 index format, enabling fast in-memory matching using an FST.

#2 The ConcatenateGraphFilter is a special filter used by the text tagger to facilitate entity extraction.

#3 We add the name_tag field, which we'll use to tag queries against the index.

#4 The name_tag field is populated using the surface_form value.

#5 A /tag request handler is configured to use values indexed in the name_tag field as entities to extract from incoming queries.

#6 If multiple entities match (polysemy), return the most popular one by default.

With the `entities` collection created, the text tagger configured, and the entities all indexed, we're now ready to perform entity extraction on a query. In the following listing, we run a query for `top kimchi near charlotte`.

Listing 7.6 Extracting entities for a given query

```
query = "top kimchi near charlotte"
entities_collection = engine.get_collection("entities")
extractor = get_entity_extractor(entities_collection)
query_entities = extractor.extract_entities(query)
print(query_entities)
```

Output:

```
{"query": "top kimchi near charlotte",
 "tags": [
  {"startOffset": 0, "endOffset": 3, "matchText": "top", "ids": ["7"]},
  {"startOffset": 11, "endOffset":15, "matchText":"near", "ids":["1","5"]},
  {"startOffset": 16, "endOffset": 25, "matchText": "charlotte",
   "ids": ["4460243", "4612828", "4680560", "4988584", "5234793"]}],
 "entities": [
  {"id":"1", "surface_form":"near", "canonical_form":"{location_distance}",
   "type": "semantic_function", "popularity": 90,
   "semantic_function": "location_distance(query, position)"},
  {"id": "5", "surface_form": "near", "canonical_form": "{text_distance}",
   "type": "semantic_function", "popularity": 10,
   "semantic_function": "text_distance(query, position)"},
  {"id": "7", "surface_form": "top", "canonical_form": "{popular}",
   "type": "semantic_function", "popularity": 100,
   "semantic_function": "popularity(query, position)"},
  {"id":"4460243", "canonical_form":"Charlotte", "surface_form":"Charlotte",
   "admin_area": "NC", "popularity": 827097, "type": "city",
   "location_coordinates": "35.22709,-80.84313"},
  {"id":"4612828", "canonical_form":"Charlotte", "surface_form":"Charlotte",
   "admin_area": "TN", "popularity": 1506, "type": "city",
   "location_coordinates": "36.17728,-87.33973"},
  {"id":"4680560", "canonical_form":"Charlotte", "surface_form":"Charlotte",
   "admin_area": "TX", "popularity": 1815, "type": "city",
   "location_coordinates": "28.86192,-98.70641"},
  {"id":"4988584", "canonical_form":"Charlotte", "surface_form":"Charlotte",
   "admin_area": "MI", "popularity": 9054, "type": "city",
   "location_coordinates": "42.56365,-84.83582"},
  {"id":"5234793", "canonical_form":"Charlotte", "surface_form":"Charlotte",
   "admin_area": "VT", "popularity": 3861, "type": "city",
   "location_coordinates": "44.30977,-73.26096"}]}
```

The response includes three key sections:

- `query`—The query that has been tagged
- `tags`—A list of text phrases found within the incoming query, along with the character offsets within the text (start and end positions) and a list of all possible entity matches (canonical forms) for each tag (surface form)
- `entities`—The list of doc IDs for matching entities that may correspond with one of the matched tags

We've previously described ambiguous terms, where one surface form can map to multiple canonical forms. In our example, the first tag is `{'startOffset': 0, 'endOffset': 3, 'matchText': 'top', 'ids': ['7']}`. This states that the text "top" was matched starting at position `0` and ending at position `3` in the input `top kimchi near charlotte`. It also lists only one entry in the `ids`, meaning that there is only one possible meaning (canonical representation). For the other two tags, however, multiple `ids` are listed, making them ambiguous tags:

- `{"startOffset": 11, "endOffset": 15, "matchText": "near", "ids": ["1", "5"]}`
- `{"startOffset": 16, "endOffset": 25, "matchText": "charlotte", "ids": ["4460243", "4612828", "4680560", "4988584", "5234793"]}`

This means that there were two canonical forms (two `ids` listed) for the surface form "near" and five canonical forms for the surface form "charlotte". In the `entities` section, we can also see all of the different entity records associated with the lists of `ids` in the tags.

In this chapter, we'll keep things simple by always using the canonical form with the highest `popularity`. For cities, we supplied the city's population in the `popularity` field, which means that the "charlotte" selected is Charlotte, NC, USA (the most populated Charlotte in the world). For our other entities, the popularity was specified manually in entities.csv from listing 7.3. You can alternatively specify the popularity using the signals-boosting value (if you derived your entities from signals, which will be covered in detail in chapter 8) or using the count of documents containing the entity in your index as a proxy for popularity.

You may also find it beneficial to use user-specific context or query-specific context to choose the most appropriate entity. For example, if you are disambiguating locations, you could boost the popularity with a geographical distance calculation so locations closer to the user receive a higher weight. If the entity is a keyword phrase, you can alternatively use an SKG to classify the query or load a term vector and boost the canonical form that is a better conceptual match for the overall query.

With our `query_entities` available from the knowledge graph, we can now generate a user-friendly version of the original query with the tagged entities identified. The following listing implements this `generate_tagged_query` function.

Listing 7.7 Generating a tagged query

```
def generate_tagged_query(extracted_entities):  #1
  query = extracted_entities["query"]
  last_end = 0
  tagged_query = ""
  for tag in extracted_entities["tags"]:
    next_text = query[last_end:tag["startOffset"]].strip()
    if len(next_text) > 0:
      tagged_query += " " + next_text
    tagged_query += " {" + tag["matchText"] + "}"  #2
    last_end = tag["endOffset"]
  if last_end < len(query):
    final_text = query[last_end:len(query)].strip()
    if len(final_text):
      tagged_query += " " + final_text
  return tagged_query

tagged_query = generate_tagged_query(query_entities)
print(tagged_query)
```

#1 Reconstructs the query with tagged entities
#2 Wraps known entities in braces to offset them from regular keywords

Output:

```
{top} kimchi {near} {charlotte}
```

From this tagged query, we can now see that the keywords “top”, “near”, and “charlotte” map to known entities, while “kimchi” is an unknown keyword. This format is a useful user-friendly representation of the query, but it is too simple to represent the metadata associated with each entity. Because we need to process the entities and their semantic interactions programmatically to enrich the query, we will also implement a more structured representation of the semantically parsed query, which we’ll call a `query_tree`.

Instead of a pure text query, this `query_tree` is a structure of strongly typed nodes within the query represented as JSON objects. Listing 7.8 demonstrates the `generate_query_tree` function, which returns a query tree from the incoming entity extraction data (`query_entities`).

Listing 7.8 Generating a typed query tree from a user query

```
def generate_query_tree(extracted_entities):
  query = extracted_entities["query"]
  entities = {entity["id"]: entity for entity  #1
              in extracted_entities["entities"]}  #1
  query_tree = []
  last_end = 0

  for tag in extracted_entities["tags"]:
    best_entity = entities[tag["ids"][0]]  #2
    for entity_id in tag["ids"]:  #2
      if (entities[entity_id]["popularity"] >  #2
          best_entity["popularity"]):  #2
        best_entity = entities[entity_id] #2

    next_text = query[last_end:tag["startOffset"]].strip()
    if next_text:
      query_tree.append({"type": "keyword",  #3
                         "surface_form": next_text,  #3
                         "canonical_form": next_text})  #3
    query_tree.append(best_entity)  #4
    last_end = tag["endOffset"]

  if last_end < len(query):  #5
    final_text = query[last_end:len(query)].strip()  #5
    if final_text:  #5
      query_tree.append({"type": "keyword",  #5
                         "surface_form": final_text,  #5
                         "canonical_form": final_text})  #5
  return query_tree

parsed_query = generate_query_tree(query_entities)
display(parsed_query)
```

#1 Creates a mapping of entity IDs to entities
#2 Chooses the most popular canonical form for the entity by default
#3 Assigns a keyword type to any untagged text as a fallback
#4 Adds the entity object to the query tree in the appropriate position in the query
#5 Any text after the last tagged entity will also be considered a keyword.

Output:

```
[{"semantic_function": "popularity(query, position)", "popularity": 100,
  "id": "7", "surface_form": "top", "type": "semantic_function",
  "canonical_form": "{popular}"},
 {"type": "keyword", "surface_form": "kimchi", "canonical_form": "kimchi"},
 {"semantic_function":"location_distance(query, position)", "popularity":90,
  "id": "1", "surface_form": "near", "type": "semantic_function",
  "canonical_form": "{location_distance}"},
 {"country": "US", "admin_area": "NC", "popularity": 827097,
  "id": "4460243", "surface_form": "Charlotte", "type": "city",
  "location_coordinates": "35.22709,-80.84313",
  "canonical_form": "Charlotte"}]
```

We now have multiple representations of the query and tagged entities:

- `tagger_data`—Output of listing 7.6
- `tagged_query`—Output of listing 7.7
- `parsed_query`—Output of listing 7.8

The `parsed_query` output is a serialization of the underlying `query_tree` object, fully representing all keywords and entities along with their associated metadata. At this point, the initial *parsing* phase that maps the query into typed entities is complete, and we can begin to use the relationships between the entities to better enrich the query.

7.4.2 Enriching a query for semantic search

The *enriching* phase of our query interpretation pipeline focuses on understanding the relationships between entities in a query and how to best interpret and represent them for optimal search results relevance.

Much of this book has already been, and will continue to be, focused on the enriching phase. Chapter 4 introduced crowdsourced relevance, which is a way to enrich specific keyword phrases with information about which documents are the most relevant, based on prior user interactions. Chapter 5 focused on knowledge graphs, which provide a way to enrich specific keyword phrases with topic classifications and to find other terms that are highly related. In chapter 6, we implemented algorithms to find synonyms, misspellings, and related terms that can be used to enrich queries by augmenting or replacing parsed terms with better, learned versions. Upcoming chapters on signals boosting, personalization, and dense vector search on embeddings will likewise introduce new ways to interpret parsed entities and enrich queries to optimize relevance.

These techniques are all tools in your toolbox, but the best way to combine them for any specific implementation will be domain-specific, so we'll avoid overly generalizing in our examples. Instead, we'll focus on a simple end-to-end implementation to tie things together in a way that other models can be easily plugged into. Our simple implementation will consist of two components:

- A *semantic function* implementation that enables dynamic and nonlinear semantic rules to be injected for each domain
- An SKG to find related terms for unknown keywords and query classifications

You already have the tools you need to extend the query-parsing framework to handle other enrichment types from previous chapters. For example, you can use the mappings from surface form to canonical form to handle all alternate representations learned in chapter 6. Likewise, by adding additional fields to each entity in the `entities` collection, you can inject signals boosts, related terms, query classifications, or vectors, making them available for use as soon as queries are parsed.

Let's kick off our enrichment implementation with a discussion of semantic functions.

IMPLEMENTING SEMANTIC FUNCTIONS

A *semantic function* is a nonlinear function that can be applied during query parsing and enrichment to better interpret the meaning of the surrounding terms. Our previous example of `top kimchi near charlotte` contains two terms that map to semantic functions: “top” and “near”. The term “top” has a very domain-specific meaning: prioritize documents with the highest rating (number of stars on the review). Likewise, the term “near” isn’t a keyword that should be matched; instead, it modifies the meaning of the succeeding terms to attempt to marshal them into a geographic location. From listing 7.3, you will see the following entities referencing semantic functions:

```
Semantic Function Entities
+-------+-------------------+----------+----------------------------------+
|surface|canonical_form     |popularity|semantic_function                 |
+-------+-------------------+----------+----------------------------------+
|near   |{location_distance}|90        |location_distance(query, position)|
|in     |{location_distance}|100       |location_distance(query, position)|
|by     |{location_distance}|90        |location_distance(query, position)|
|by     |{text_within_on...}|10        |text_within_one_edit_distance(...)|
|near   |{text_distance}    |10        |text_distance(query, position)    |
|popular|{popular}          |100       |popularity(query, position)       |
|top    |{popular}          |100       |popularity(query, position)       |
|best   |{popular}          |100       |popularity(query, position)       |
|good   |{popular}          |100       |popularity(query, position)       |
+-------+-------------------+----------+----------------------------------+
```

You’ll note that the surface forms “top”, “popular”, “good”, and “best” all map to the `{popularity}` canonical form, which is represented by the `popularity(query, position)` semantic function in the following listing.

Listing 7.9 A semantic function that accounts for popularity

```
def popularity(query, position):
  if len(query["query_tree"]) -1 > position:  #1
    query["query_tree"][position] = {
      "type": "transformed",
      "syntax": "solr",
      "query": '+{!func v="mul(if(stars_rating,stars_rating,0),20)"}'}  #2
    return True  #3
  return False  #3
```

#1 Another query tree node must follow the popularity node (“top mountain” and not “mountain top”).
#2 Replace the {popularity} node in the query tree with a new node that represents a relevance boost for popularity.
#3 Returns whether the semantic function was triggered. If False, then another overlapping function with lower precedence could be attempted.

This popularity function allows us to apply semantic interpretation logic to manipulate the query tree. Had the query tree ended with the keyword “top”, the function would have returned `False` and no adjustments would have been made. Likewise, if another function had been assigned higher precedence (as specified in the `entities` collection), it might have removed the `{popularity}` entity before its function was even executed.

The `location_distance` function is a bit more involved, as shown in the next listing.

Listing 7.10 A semantic function that accounts for location

```
def location_distance(query, position):
  if len(query["query_tree"]) -1 > position:  #1
    next_entity = query["query_tree"][position + 1] #1
    if next_entity["type"] == "city": #2

      query["query_tree"].pop(position + 1)  #3
      query["query_tree"][position] = {  #4
        "type": "transformed",  #4
        "syntax": "solr",  #4
        "query": create_geo_filter(  #4
          next_entity['location_coordinates'],  #4
          "location_coordinates", 50)}  #4
      return True
  return False  #5

def create_geo_filter(coordinates, field, distance_KM):
  return f'+{!geofilt d={distance_KM} sfield="{field}" pt="{coordinates}"}'
```

#1 The function must modify the next entity to succeed.
#2 The next entity must be of a location type (city).
#3 Remove the next entity, since it is a location that will now be replaced by a radius-based filter.
#4 Add in the replacement entity with the radius filter.
#5 If the next entity was not a city, then don't apply the function.

As you can see, our implementation of semantic functions allows for any arbitrary logic to be conditionally applied when interpreting queries. If you want, you can even call external knowledge graphs or other data sources to pull in additional information to better interpret the query.

You may have noticed that the surface forms "near", "in", and "by" all map to the `{location_distance}` canonical form, which is represented by the `location_distance(query, position)` function. This function works well if one of those terms is followed by a location, but what if someone searches for `chief near officer`? In this context, the end user may have meant "find the term *chief* close to the term *officer* within a document"—essentially an edit distance search. Note that there is also an entity mapping "near" ⇒ `{text_distance}` that can be invoked conditionally for this fallback use case if the `{location_distance}` entity's semantic function returns `False`.

Semantic functions can be implemented in many different ways, but our example implementation provides a highly configurable way to code dynamic semantic patterns into your query interpretation pipeline to best tie into the many different AI-powered search approaches available to your search application. We show this implementation in the `process_semantic_functions` function in the following listing, which loops through the query tree to invoke all matched semantic functions.

Listing 7.11 Processing all semantic functions within the query tree

```
def process_semantic_functions(query_tree):
  position = 0  #1
  while position < len(query_tree):  #2
    node = query_tree[position]  #2
    if node["type"] == "semantic_function":
      query = {"query_tree": query_tree}  #1
      command_successful = eval(node["semantic_function"]) #3
      if not command_successful:  #4
        node["type"] = "invalid_semantic_function"  #4
    position += 1
  return query_tree
```

#1 The query and position variables are passed to the semantic function on eval.
#2 Iterates through all items in the query tree, searching for semantic functions to execute
#3 Dynamically evaluates semantic functions, which augment the query tree
#4 Updates the type of any unsuccessful semantic functions

Since the semantic functions are stored as part of their entity from the `entities` collection, we perform late binding on these functions (using Python's `eval` function). This allows you to plug new semantic functions into the `entities` collection at any time without needing to modify the application code.

Because semantic functions may or may not succeed depending on the surrounding context nodes, each semantic function must return `True` or `False` to allow the processing logic to determine how to proceed with the rest of the query tree.

INTEGRATING AN SKG

In this section, we'll integrate an SKG (discussed in chapter 5) into our query enrichment process.

Your `entities` collection is likely to contain many learned entities using the techniques from chapter 6. You may also use an SKG or other approach to classify known entities or to generate lists of related terms. If you do, we recommend adding the classifications and related terms as additional fields in the `entities` collection to cache the responses for quicker lookup at query time.

For our implementation, we'll invoke an SKG in real time to enrich *unknown* terms. This approach injects related keywords for all unknown keyword phrases in a query, which can generate a lot of false positives. You'll likely want to be more conservative in any production implementation, but implementing this is useful for learning and experimentation purposes. The following listing demonstrates how to look up keyword phrases and traverse our reviews collection as an SKG.

Listing 7.12 Getting related terms and categories from the SKG

```
def get_enrichments(collection, keyword, limit=4):
  enrichments = {}
  nodes_to_traverse = [{"field": "content",  #1
                        "values": [keyword],  #1
                        "default_operator": "OR"},  #1
                       [{"name": "related_terms",  #2
                         "field": "content", #2
                         "limit": limit},  #2
                        {"name": "doc_type", #3
                         "field": "doc_type", #3
                         "limit": 1}]]  #3
  skg = get_semantic_knowledge_graph(collection)
  traversals = skg.traverse(*nodes_to_traverse)
  if "traversals" not in traversals["graph"][0]["values"][keyword]:
    return enrichments  #4

  nested_traversals = traversals["graph"][0]["values"] \
                                [keyword]["traversals"]

  doc_types = list(filter(lambda t: t["name"] == "doc_type", #5
                          nested_traversals))  #5
  if doc_types:  #5
    enrichments["category"] = next(iter(doc_types[0]["values"]))  #5

  related_terms = list(filter(lambda t: t["name"] == "related_terms",  #6
                              nested_traversals))  #6
  if related_terms:  #6
    term_vector = ""  #6
    for term, data in related_terms[0]["values"].items():  #6
      term_vector += f'{term}^{round(data["relatedness"], 4)} '  #6
    enrichments["term_vector"] = term_vector.strip()  #6

  return enrichments

query = "kimchi"  #7
get_enrichments(reviews_collection, query)  #7
```

#1 The starting node for the SKG traversal is a query for the passed-in keyword in the content field.

#2 Returns the top 4 related terms for the keyword

#3 Returns the top 1 doc_type (category) for the keyword

#4 Returns empty when no enrichments are found

#5 Returns discovered categories from the traversal

#6 Constructs a boosted query from discovered related terms boosted by their relatedness

#7 Gets enrichments for the keyword "kimchi"

The output of listing 7.12 for the keyword "kimchi" is as follows:

```
{"category": "Korean",
 "term_vector": "kimchi^0.9193 korean^0.7069 banchan^0.6593 bulgogi^0.5497"}
```

Here are some sample SKG outputs for other potential keywords:

- *bbq*:

```
{"category": "Barbeque",
 "term_vector": "bbq^0.9191 ribs^0.6187 pork^0.5992 brisket^0.5691"}
```

- *korean bbq*:

```
{"category": "Korean",
 "term_vector": "korean^0.7754 bbq^0.6716 banchan^0.5534 sariwon^0.5211"}
```

- *lasagna*:

```
{"category": "Italian",
 "term_vector": "lasagna^0.9193 alfredo^0.3992 pasta^0.3909
              ↪italian^0.3742"}
```

- *karaoke*:

```
{"category": "Karaoke",
 "term_vector": "karaoke^0.9193 sing^0.6423 songs^0.5256 song^0.4118"}
```

- *drive through*:

```
{"category": "Fast Food",
 "term_vector": "drive^0.7428 through^0.6331 mcdonald's^0.2873
               ↪window^0.2643"}
```

To complete our *enriching* phase, we need to apply the `get_enrichments` function and the previously discussed `process_semantic_functions` function to our query tree.

Listing 7.13 Enriching the query tree nodes

```
def enrich(collection, query_tree):
  query_tree = process_semantic_functions(query_tree)  #1
  for item in query_tree:


    if item["type"] == "keyword":  #2
      enrichments = get_enrichments(collection, item["surface_form"]) #2
      if enrichments:  #3
        item["type"] = "skg_enriched"  #3
        item["enrichments"] = enrichments  #3
  return query_tree
```

#1 Loops through the query tree and processes all semantic functions
#2 Takes all unknown keyword phrases, and looks them up in the SKG
#3 If enrichments are found, apply them to the node.

This `enrich` function encompasses the entire enrichment phase, processing all semantic functions and then enriching all the remaining unknown keywords using the SKG. Before we move on to the transformation phase, however, let's quickly look at an alternative approach to the SKG-based keyword expansion we've implemented.

7.4.3 Sparse lexical and expansion models

We've covered two main approaches to search so far in this book: *lexical search*—matching and ranking based on specific terms or attributes in a query—and *semantic search*—matching and ranking based on the meaning of the query. You've also been introduced to two main approaches for representing queries: as sparse vectors (vectors with very few nonzero values) and dense vectors (vectors with mostly nonzero values). Lexical keyword search is usually implemented using an inverted index, which holds a sparse vector representation of every document with a dimension for every term in the index. Semantic search is likewise usually implemented using a dense vector representation searching on embeddings.

SPARSE VECTOR VS. DENSE VECTOR VS. LEXICAL SEARCH VS. SEMANTIC SEARCH

Due to computational cost, dense vector representations usually have a limited number of dimensions (hundreds to thousands) that densely compress a semantic representation of the data, whereas sparse vector representations can easily have hundreds of thousands to tens of millions of dimensions that represent more identifiable terms or attributes. Lexical keyword search is usually implemented using an inverted index, which holds a sparse vector representation of every document with a dimension for every term in the index. Semantic search is likewise usually implemented using a dense vector representation searching on embeddings. Due to these trends, many people mistakenly treat the term "semantic search" as synonymous with dense vector search on embeddings, but this throws away the rich history of much more explainable and flexible sparse-vector and graph-based semantic search approaches. This chapter highlights some of those approaches, with chapters 13–15 covering the dense vector search techniques in more depth.

As you've seen already in this chapter, however, semantic search can also be implemented using sparse vectors, and within the context of typical lexical queries. While we've implemented semantic query parsing that operates directly on user queries, we've also generated query expansions using an SKG to generate sparse vectors of terms and weights to power semantic search.

Other techniques exist for this kind of query expansion, such as SPLADE (Sparse Lexical and Expansion). Instead of using an inverted index as its language model, the SPLADE approach (**https://arxiv.org/pdf/2107.05720**) uses a prebuilt language model to generate contextualized tokens. We won't use SPLADE (or SPLADE V2 or subsequent versions) because it wasn't released under a license enabling commercial use, but listing 7.14 demonstrates sample output from an alternative open source implementation (SPLADE++) for the same example queries we just tested with the SKG approach in section 7.4.2.

Listing 7.14 Expanding queries with SPLADE++

```
from spladerunner import Expander
expander = Expander('Splade_PP_en_v1', 128) #1
queries = ["kimchi", "bbq", "korean bbq",
           "lasagna", "karaoke", "drive through"]

for query in queries:
  sparse_vec = expander.expand(query,  #2
                  outformat="lucene")[0]  #3
  print(sparse_vec)
```

#1 Specifies the SPLADE++ model name and maximum sequence length
#2 Generates the sparse lexical vector
#3 Returns token labels (strings) instead of token IDs (integers)

Here are the outputs from the SPLADE++ expansion:

- *kimchi*:

```
{"kim": 3.11, "##chi": 3.04, "ki": 1.52, ",": 0.92, "who": 0.72,
 "brand": 0.56, "genre": 0.46, "chi": 0.45, "##chy": 0.45,
 "company": 0.41,  "only": 0.39, "take": 0.31, "club": 0.25,
 "species": 0.22, "color": 0.16, "type": 0.15, "but": 0.13,
 "dish": 0.12, "hotel": 0.11, "music": 0.09, "style": 0.08,
 "name": 0.06, "religion": 0.01}
```

- *bbq*:

```
{"bb": 2.78, "grill": 1.85, "barbecue": 1.36, "dinner": 0.91,
 "##q": 0.78, "dish": 0.77, "restaurant": 0.65, "sport": 0.46,
 "food": 0.34, "style": 0.34, "eat": 0.24, "a": 0.23, "genre": 0.12,
 "definition": 0.09}
```

- *korean bbq*:

```
{"korean": 2.84, "korea": 2.56, "bb": 2.23, "grill": 1.58, "dish": 1.21,
 "restaurant": 1.18, "barbecue": 0.79, "kim": 0.67, "food": 0.64,
 "dinner": 0.39, "restaurants": 0.32, "japanese": 0.31, "eat": 0.27,
 "hotel": 0.16, "famous": 0.11, "brand": 0.11, "##q": 0.06, "diner": 0.02}
```

- *lasagna*:

```
{"las": 2.87, "##ag": 2.85, "##na": 2.39, ",": 0.84, "she": 0.5,
 "species": 0.34, "hotel": 0.33, "club": 0.31, "location": 0.3,
 "festival": 0.29, "company": 0.27, "her": 0.2, "city": 0.12,
 "genre": 0.05}
```

- *karaoke*:

```
{"kara": 3.04, "##oke": 2.87, "music": 1.31, "lara": 1.07,
 "song": 1.03, "dance": 0.97, "style": 0.94, "sara": 0.81,
 "genre": 0.75, "dress": 0.48, "dish": 0.44, "singer": 0.37,
 "hannah": 0.36, "brand": 0.31, "who": 0.29, "culture": 0.21,
 "she": 0.17, "mix": 0.17, "popular": 0.12, "girl": 0.12,
 "kelly": 0.08, "wedding": 0.0}
```

- *drive through*:

```
{"through": 2.94, "drive": 2.87, "driving": 2.34, "past": 1.75,
 "drives": 1.65, "thru": 1.44, "driven": 1.22, "enter": 0.81,
 "drove": 0.81, "pierce": 0.75, "in": 0.72, "by": 0.71, "into": 0.64,
 "travel": 0.59, "mark": 0.51, ";": 0.44, "clear": 0.41,
 "transport": 0.41, "route": 0.39, "within": 0.36, "vehicle": 0.3,
 "via": 0.15}
```

Note that the `outputformat=lucene` parameter results in the tokens (keywords or partial keywords) being returned instead of the integer IDs of the tokens, because seeing the tokens helps us better interpret the results.

When comparing this output with the previously shown SKG output for the same queries, you may notice the following differences:

- The SKG output returns actual terms in the index, whereas the SPLADE-style output returns tokens from the LLM. This means that you can use the SKG output ("lasagna", "alfredo", "pasta") directly to search on the fields on your documents, whereas the SPLADE tokens (`las`, `##ag`, `na##`) will need to be generated from SPLADE for all documents and indexed in order for a SPLADE-style query to match on the right tokens at query time.
- The SKG sparse vectors tend to look cleaner and more relevant to the dataset (restaurant reviews) for in-domain terms. For example, for the query `bbq`, the SKG returns `{"bbq": 0.9191, "ribs":0.6186, "pork":0.5991, "brisket" :0.569}`, whereas SPLADE returns `{'bb': 2.78, 'grill': 1.85, 'barbecue': 1.36, 'dinner': 0.91, '##q': 0.78, 'dish': 0.77, 'restaurant': 0.65, 'sport': 0.46, 'food': 0.34, …}`. This underperformance of the SPLADE model relative to the SKG model is mostly due to SPLADE not being trained on the data in the search index, whereas the SKG uses the data in the search index directly as its language model. Fine-tuning the SPLADE-based model would help to close this gap.
- The SKG model is more flexible, as it can return relationships across multiple dimensions. Notice in the last section that we not only returned the sparse vector of related terms, but also a classification of the query.
- Both the SPLADE and SKG models are context-aware. SPLADE natively weights each token based on the entire context of the query (or document) being encoded, and the SKG request can likewise (optionally) use any passed-in context for the query or document to contextualize the weights of its tokens. SPLADE-based models tend to shine more with longer well-known contexts (like general documents), whereas the SKG model is more optimized for shorter, domain-specific contexts (like in-domain queries), but they both

work and represent novel techniques for sparse-vector-based or lexically oriented semantic search.

We've chosen to use an SKG-based approach instead of SPLADE in this chapter due to its ability to also classify queries and further contextualize queries for query sense disambiguation, but similar concepts apply for implementing sparse-vector-based semantic search regardless of which model you choose, so it's good to be familiar with multiple techniques.

In the next section, we'll walk through the transformation of the enriched query tree into a search-engine-specific query syntax for sending to the search engine.

7.4.4 Transforming a query for semantic search

With the user's query now parsed and enriched, it's time to transform the query tree into an appropriate search-engine-specific syntax.

During this *transforming* phase, we invoke an adapter to convert the query tree to the most useful engine-specific representation of the query—Solr in our default implementation. In the case of our semantic functions (the `popularity` and `location_distance` functions), we already injected this engine-specific syntax (`{"type":"transformed", "syntax":"solr"}`) directly into the enriched nodes in the query tree. We could have alternatively abstracted this a bit by creating a generic intermediate representation of each semantic function's output and then waiting until the transforming phase to convert to engine-specific syntax (Solr, OpenSearch, etc.), but we chose to avoid the intermediate representations to keep the examples simpler. If you run the code using a different engine (as explained in appendix B), you'll see the syntax for that engine in the transformed nodes instead.

The following listing shows a `transform_query` function that takes an enriched query tree and transforms each of its nodes into search-engine-specific nodes.

Listing 7.15 Transforming a query tree into engine-specific syntax

```
def transform_query(query_tree):
  for i, item in enumerate(query_tree):
    match item["type"]:
      case "transformed":  #1
        continue  #1
      case "skg_enriched":  #2
        enrichments = item["enrichments"]  #2
        if "term_vector" in enrichments:  #2
          query_string = enrichments["term_vector"]  #2
          if "category" in enrichments: #2
            query_string += f' +doc_type:"{enrichments["category"]}"'  #2
          transformed_query =   #2
          ↪'+{!edismax v="' + escape_quotes(query_string) + '"}'  #2
        else:  #2
          continue  #2
      case "color":  #3
        transformed_query = f'+colors:"{item["canonical_form"]}"'
      case "known_item" | "event":  #3
        transformed_query = f'+name:"{item["canonical_form"]}"'
      case "city":  #3
        transformed_query = f'+city:"{str(item["canonical_form"])}"'
      case "brand":  #3
        transformed_query = f'+brand:"{item["canonical_form"]}"'
      case _:  #4
        transformed_query = "+{!edismax v=\"" +  #4
        ↪escape_quotes(item["surface_form"]) + "\"}"  #4
    query_tree[i] = {"type": "transformed",  #5
                     "syntax": "solr",  #5
                     "query": transformed_query}  #5
  return query_tree

enriched_query_tree = enrich(reviews_collection, query_tree)
processed_query_tree = transform_query(enriched_query_tree)
display(processed_query_tree)
```

#1 If a query tree element has already been transformed to search-engine-specific syntax, there is no need to process it further.

#2 Generates an enriched query for enriched nodes

#3 Handles the transformation of other potential query-tree elements with custom type-handling logic

#4 For all other types without custom transformation logic, just search on their surface form.

#5 Denotes each transformed query tree node with the engine-specific syntax and query

Output:

```
[{"type": "transformed",
  "syntax": "solr",
  "query": "+{!func v=\"mul(if(stars_rating,stars_rating,0),20)\"}"},
 {"type": "transformed",
  "syntax": "solr",
  "query": "{!edismax v=\"kimchi^0.9193 korean^0.7069 banchan^0.6593
  ↪+doc_type:\\\"Korean\\\"\"}"},
 {"type": "transformed",
  "syntax": "solr",
  "query": "+{!geofilt d=50 sfield=\"location_coordinates\"
  ↪pt=\"35.22709,-80.84313\"}"}]
```

At this point, all nodes in the query tree have been transformed into `{'type': 'transformed', 'syntax': engine}` nodes, which means they internally contain the search-engine-specific syntax needed to generate a final query to the configured engine. We're now ready to convert the query tree into a string and send the request to the search engine.

7.4.5 Searching with a semantically enhanced query

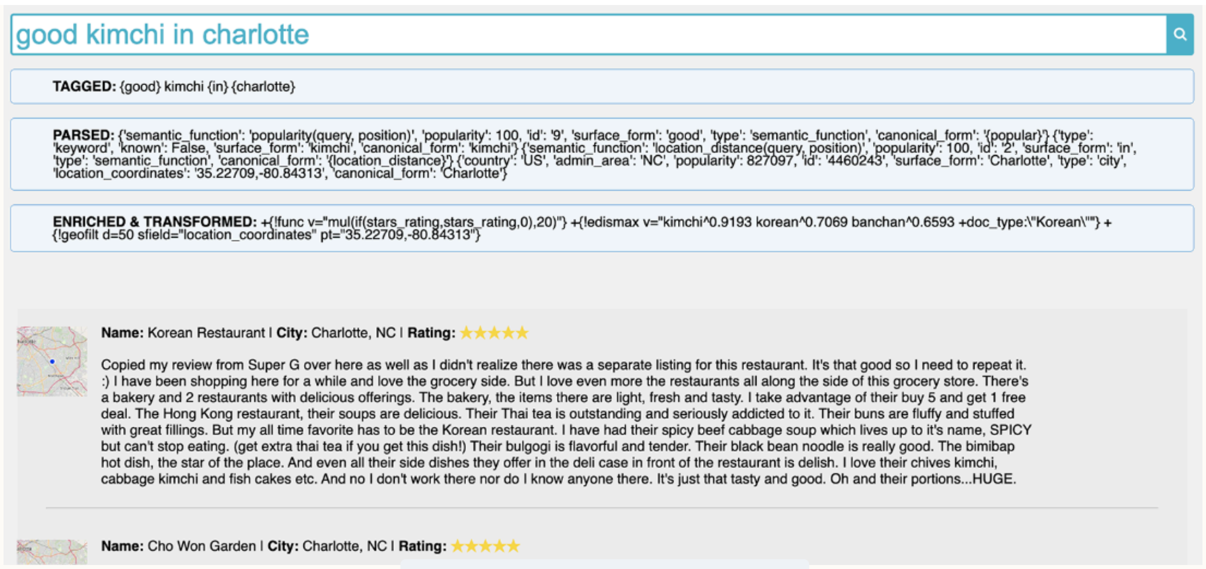
The final step of our semantic search process is the *searching* phase. We convert the fully transformed `query_tree` to a query, run the query against the search engine, and return the results to the end user.

Listing 7.16 Running the query

```
def to_query(query_tree):
  return [[node["query"] for node in query_tree]


transformed_query = to_query(query_tree)
reviews_collection = engine.get_collection("reviews")
reviews_collection.search(query=transformed_query)
```

The search results for our query `top kimchi near charlotte` will return exactly what was shown in our end-to-end example in section 7.3. Since we know we can now handle variations of semantic functions ("in" versus "near" for locations, "good" versus "popular" versus "top" for popularity), we'll show the output for a slightly modified query: `good kimchi in charlotte`. If you contrast the output for this variant, shown in figure 7.8, with the output for the original query of `top kimchi near charlotte`, you'll see that they yield the exact same transformed query and final set of search results as figures 7.5 and 7.7 earlier in the chapter.



good kimchi in charlotte

TAGGED: {good} kimchi {in} {charlotte}

PARSED: {'semantic_function': 'popularity(query, position)', 'popularity': 100, 'id': '9', 'surface_form': 'good', 'type': 'semantic_function', 'canonical_form': '{popular}'} {'type': 'keyword', 'known': False, 'surface_form': 'kimchi', 'canonical_form': 'kimchi'} {'semantic_function': 'location_distance(query, position)', 'popularity': 100, 'id': '2', 'surface_form': 'in', 'type': 'semantic_function', 'canonical_form': '{location_distance}'} {'country': 'US', 'admin_area': 'NC', 'popularity': 827097, 'id': '4460243', 'surface_form': 'Charlotte', 'type': 'city', 'location_coordinates': '35.22709,-80.84313', 'canonical_form': 'Charlotte'}

ENRICHED & TRANSFORMED: +{!func v="mul(if(stars_rating,stars_rating,0),20)"} +{!edismax v="kimchi^0.9193 korean^0.7069 banchan^0.6593 +doc_type:\"Korean\""} + {!geofilt d=50 sfield="location_coordinates" pt="35.22709,-80.84313"}

Name: Korean Restaurant | **City:** Charlotte, NC | **Rating:** ★★★★★

Copied my review from Super G over here as well as I didn't realize there was a separate listing for this restaurant. It's that good so I need to repeat it. :) I have been shopping here for a while and love the grocery side. But I love even more the restaurants all along the side of this grocery store. There's a bakery and 2 restaurants with delicious offerings. The bakery, the items there are light, fresh and tasty. I take advantage of their buy 5 and get 1 free deal. The Hong Kong restaurant, their soups are delicious. Their Thai tea is outstanding and seriously addicted to it. Their buns are fluffy and stuffed with great fillings. But my all time favorite has to be the Korean restaurant. I have had their spicy beef cabbage soup which lives up to it's name, SPICY but can't stop eating. (get extra thai tea if you get this dish!) Their bulgogi is flavorful and tender. Their black bean noodle is really good. The bimibap hot dish, the star of the place. And even all their side dishes they offer in the deli case in front of the restaurant is delish. I love their chives kimchi, cabbage kimchi and fish cakes etc. And no I don't work there nor do I know anyone there. It's just that tasty and good. Oh and their portions...HUGE.

Name: Cho Won Garden | **City:** Charlotte, NC | **Rating:** ★★★★★

Figure 7.8 Search results for `good kimchi in charlotte`, interpreted as semantically identical to the results for the `top` `kimchi` `near` `charlotte` end-to-end example

Congratulations, you've now implemented an end-to-end semantic search pipeline that can semantically parse, enrich, transform, and run searches. This chapter didn't introduce any fancy new machine-learning algorithms but instead provided a concrete implementation of how the many models, algorithms, and other techniques you've learned throughout the book can be integrated into an end-to-end system.

Throughout the remainder of the book, we'll continue to explore more advanced approaches that can plug into this framework to enhance relevance ranking and improve query intent understanding.

Summary

- Query interpretation requires appropriately mixing query pipelines with learned models while ensuring an adequate fallback model for matching unknown keywords.
- Matching only on keywords can sometimes work, but it results in poor matches when the intent expressed by connecting words isn't understood (`top`, `near`, etc.). One way to solve this is by implementing domain-specific semantic functions to overcome those limitations.
- A semantic query parser, which is aware of known terms and phrases learned within your domain, allows you to move from a keyword-based search to a semantic search on entities and their relationships.
- Extracting known entities enables the seamless integration of models into your query interpretation pipeline, using mappings between surface forms of keywords to canonical representations of entities generated from your learned models (signals boosts, alternative spellings, related terms, and other knowledge graph data).
- Semantic search involves *parsing* known entities, *enriching* with learned models, *transforming* the query to optimize matching and relevance for the target search engine, and then *searching* and returning results to the end user. We'll continue to explore more advanced techniques to plug into these phases in the coming chapters.

Previous chapter

6 Using context to learn domain-specific language

Next chapter

Part 3 Reflected intelligence