

# 9 Personalized search

## This chapter covers

- The personalization spectrum between search and recommendations
- Implementing collaborative filtering and personalization using latent features from users' signals
- Using embeddings to create personalization profiles
- Multimodal personalization from content and behavior
- Applying clustering-based personalization guardrails
- Avoiding the pitfalls of personalized search

The better your search engine understands your users, the more likely it will be able to successfully interpret their queries. In chapter 1, we introduced the three key contexts needed to properly interpret query intent: content understanding, domain understanding, and user understanding. In this chapter, we'll dive into the user understanding context.

We've already focused on learning domain-specific context from documents (chapter 5) and on the most popular results according to many different users (chapter 8), but it's not always reasonable to assume that the “best” result is agreed upon across all users. Whereas signals-boosting models find the most *popular* answers across all users, personalized search instead attempts to learn about each *specific* user's interests and to return search results catering to those interests.

For example, when searching for restaurants, a user's location clearly matters. When searching for a job, each user's employment history (previous job titles, experience level, salary range) and location may matter. When searching for products, particular brand affinities, colors of appliances, complementary items purchased, and similar personal tastes may matter.

In this chapter, we'll use user signals to learn latent features describing users' interests. *Latent features* are features hidden within data, but which can be inferred about users or items by modeling the data. These latent features will be used to generate product recommendations and boosts to personalized search results. We'll also use content-based embeddings to relate products and we'll use embeddings of the products each user interacts with to generate vector-based personalization profiles to personalize search results.

Finally, we'll cluster products by their embeddings to generate personalization guardrails to ensure that users don't see personalized search results based on products from unrelated categories.

Personalization should be applied to search results very carefully. It's easy to frustrate users by overriding their explicit intent (usually specified as search keywords) with assumptions based on their previous search activity. We'll dive into the nuances of balancing the benefits of better-personalized search against potential user frustration caused by an engine trying too hard to read their minds. Not all searches should be personalized, but when it's done well, you'll see how it can greatly improve the search experience.

## 9.1 Personalized search vs. recommendations

Search engines and recommendation engines represent two ends of a personalization spectrum, which we introduced in chapter 1 (see figure 1.5). We also discussed the dimensions of user intent in chapter 1 (see figure 1.7), noting that fully understanding user intent requires content understanding, user understanding, and domain understanding. Figure 9.1 resurfaces these two mental models.

While keyword search represents only content understanding, and collaborative recommendations represent only user understanding, they both can and should be combined when possible. *Personalized search* lies at the intersection between keyword search and collaborative recommendations.

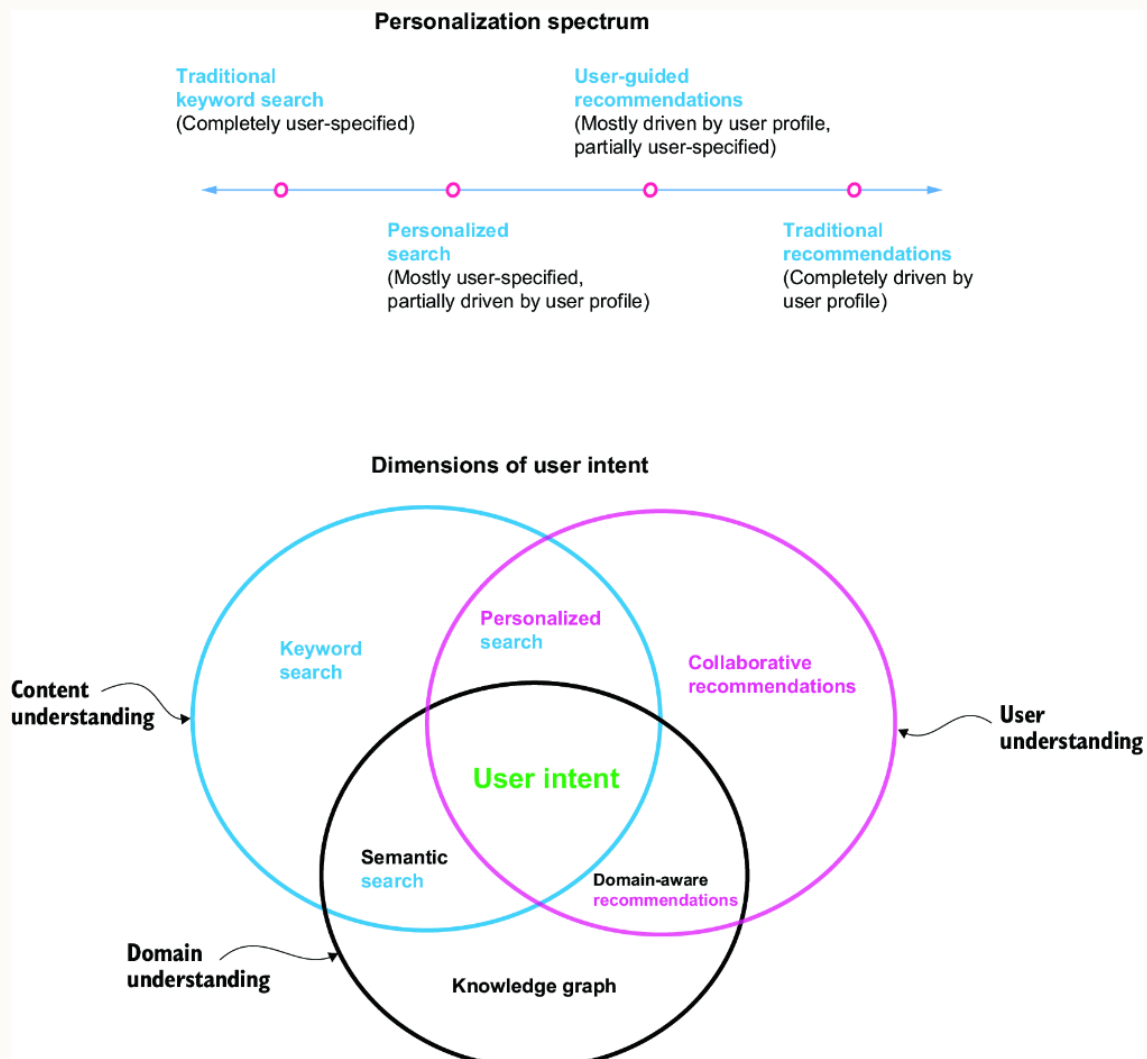
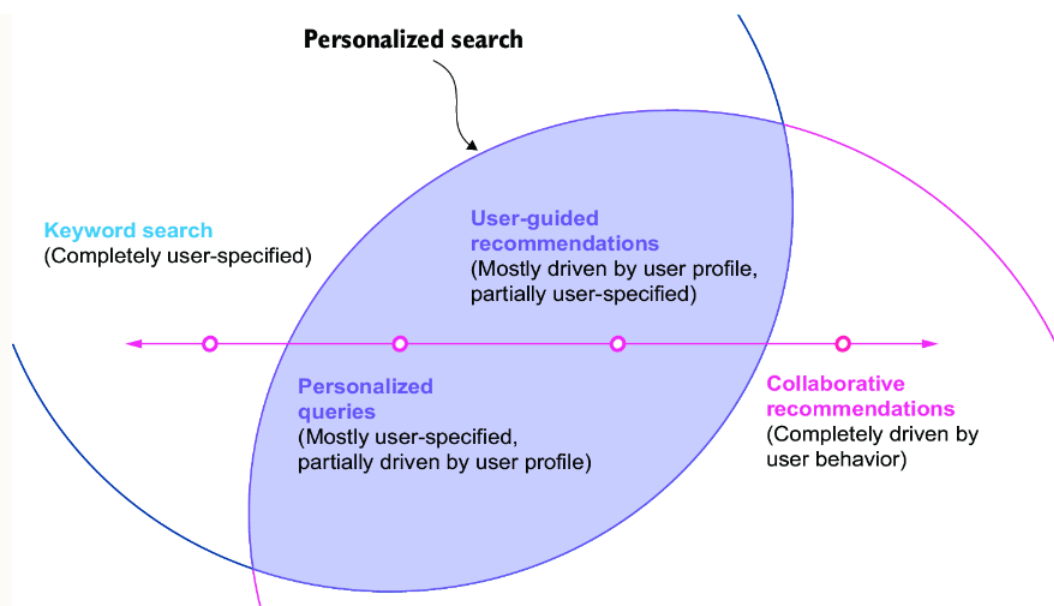


Figure 9.1 The personalization spectrum and the dimensions of user intent

Figure 9.2 superimposes the personalization spectrum on top of the diagram of the dimensions of user intent to paint a more nuanced picture of how personalized search fits along the personalization spectrum.



**Figure 9.2 Personalized search lies at the intersection between keyword search and collaborative recommendations**

The key differentiating factor between search engines and recommendation engines is that search engines are typically guided by users and match their explicitly entered queries, whereas recommendation engines typically accept no direct user input and instead recommend content based on already known or inferred knowledge. The reality, however, is that these two kinds of systems form two sides of the same coin. The goal in both cases is to understand what a user is looking for and deliver relevant results to meet that user's information need. In this section, we'll discuss the broad spectrum of capabilities that lie within the personalization spectrum between search and recommendation systems.

### 9.1.1 Personalized queries

Let's imagine we're running a restaurant search engine. Our user, Michelle, is on her phone in New York at lunchtime, and she types in a keyword search for `steamed bagels`. She sees top-rated steamed bagel shops in Greenville, South Carolina (USA); Columbus, Ohio (USA); and London (UK).

What's wrong with these search results? Well, in this case, the answer is clear—Michelle is looking for lunch in New York, but the search engine is showing her results hundreds to thousands of kilometers away. But Michelle never *told* the search engine she only wanted to see results in New York, nor did she tell the search engine that she was looking for a lunch place close by because she wants to eat now. Nevertheless, the search engine should be able to infer this information and personalize the search results accordingly.

Consider another scenario—Michelle is at the airport after a long flight, and she searches on her phone for `driver`. The top results that come back are for a golf club for hitting the ball off a tee, followed by a link to printer drivers, followed by a screwdriver. If the search engine knows Michelle's location, shouldn't it be able to infer her intended meaning—that she is searching for a ride?

Using our job search example from earlier, let's assume Michelle goes to her favorite job search engine and types in `nursing jobs`. Like our restaurant example earlier, wouldn't it be ideal if nursing jobs in New York showed up at the top of the list? What if she later types `jobs in Seattle`? Wouldn't it be ideal if—instead of seeing random jobs in Seattle (doctor, engineer, chef, etc.)—nursing jobs now showed up at the top of the list, since the engine previously learned that she is a nurse?

Each of these is an example of a personalized query: the combining of both an explicit user query *and* an implicit understanding of the user's intent and preferences into a search that serves results specifically catering to that user. Doing this kind of personalized search well is tricky, as you must carefully balance your understanding of the user without overriding anything they explicitly want to query. When it's done well, though, personalized queries can significantly improve search relevance.

### 9.1.2 User-guided recommendations

Just as it's possible to sprinkle an implicit understanding of user-specific attributes into an explicit keyword search to generate personalized search results, it's also possible to enable user-guided recommendations by allowing user-overrides of the inputs into automatically generated recommendations.

It is becoming increasingly common for recommendation engines to allow users to see and edit their recommendation preferences. These preferences usually include a list of items the user interacted with before by viewing, clicking, or purchasing them. Across a wide array of use cases, these preferences could include both specific item preferences, like favorite movies, restaurants, or places, as well as aggregated or inferred preferences, like clothing sizes, brand affinities, favorite colors, preferred local stores, desired job titles and skills, preferred salary ranges, and so on. These preferences make up a user profile: they define what is known about a customer, and the more control you can give a user to see, adjust, and improve this profile, the better you'll be able to understand your users and the happier they'll likely be with the results.

## 9.2 Recommendation algorithm approaches

In this section, we'll discuss the different types of recommendation algorithms.

Recommendation engine implementations come in different flavors depending on what data is available to drive their recommendations. Some systems only have user behavioral signals and very little content or information about the items being recommended, whereas other systems have rich content about items, but very few user interactions with the items. We'll cover content-based, behavior-based, and multimodal recommenders.

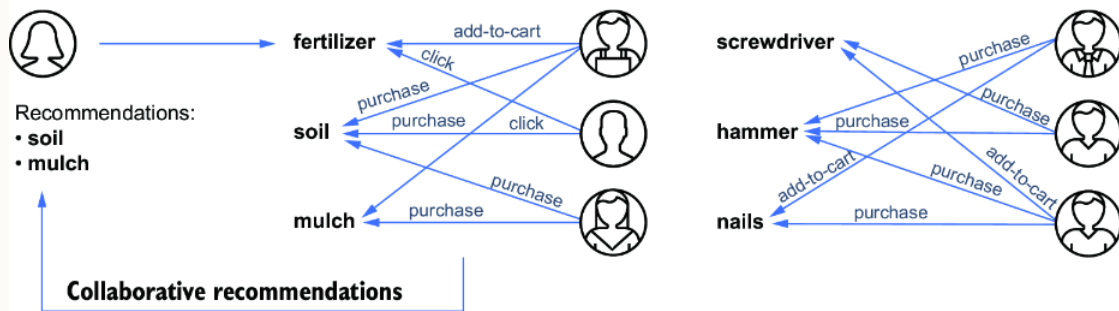
### 9.2.1 Content-based recommenders

Content-based recommendation algorithms recommend new content based on attributes shared between different entities (often between users and items, between items and items, or between users and users). For example, imagine a job search website. Jobs may have properties on them like "job title", "industry", "salary range", "years of experience", and "skills". Users will have similar attributes on their profile or resume/CV. Based upon these properties, a content-based recommendation algorithm can figure out which of these features are most important and can then rank the best matching jobs for any given user based on the user's desired attributes. This is what's known as a user-item (or user-to-item) recommender.

Similarly, if a user likes a particular job, it is possible to use this same process to recommend similar jobs based on how well those jobs match the attributes of the first job. This type of recommendation is popular on product details pages, where a user is already looking at an item and it may be desirable to help them explore related items. This kind of recommendation algorithm is known as an item-item (or item-to-item) recommender.

Figure 9.3 demonstrates how a content-based recommender might use attributes about items with which a user has previously interacted to match similar items for that user. In this case,





**Figure 9.4 Recommendations based on collaborative filtering, a technique using the overlap between behavioral signals across multiple users**

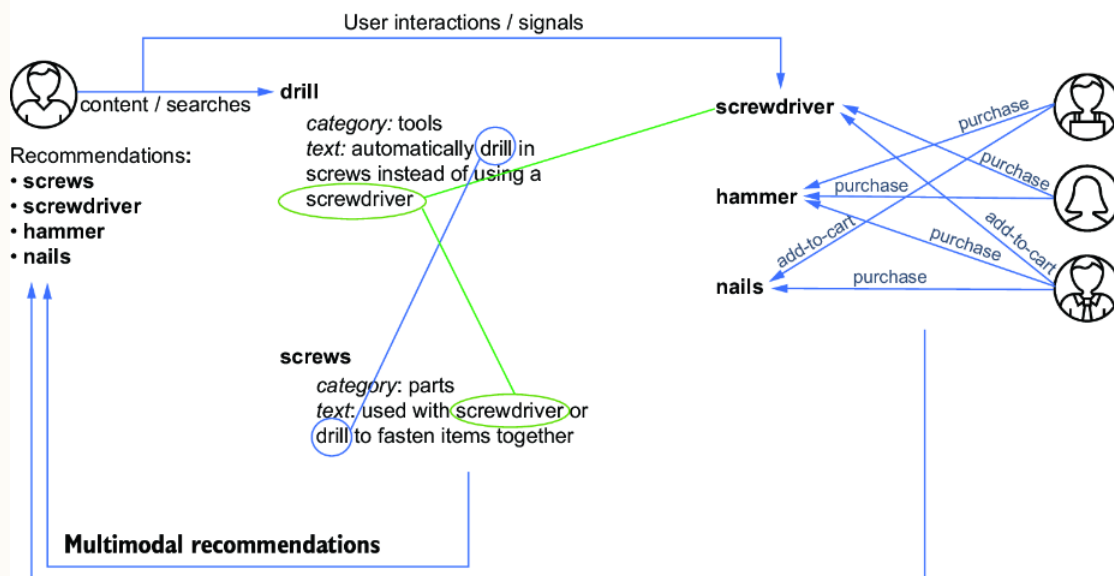
We'll implement an end-to-end collaborative filtering example in section 9.3, covering the process of discovering latent user and item features from user behavioral signals and using those to generate item recommendations for users. Because collaborative filtering is completely crowdsourced, it is immune to data quality problems with your documents or associated content attributes that may be missing or incorrect.

Unfortunately, the same dependence upon user behavioral signals that makes collaborative filtering so powerful also turns out to be its weakness. What happens when there are only a few interactions with a particular item—or possibly none at all? The answer is that the item either never gets recommended (when there are no signals), or it will be likely to generate poor recommendations or show up as a bad match for other items (when there are few signals). This situation is known as the *cold-start problem*, and it's a major challenge for behavior-based recommenders. To solve this problem, you typically need to combine behavior-based recommenders with content-based recommenders, as we'll discuss next.

### 9.2.3 Multimodal recommenders

*Multimodal recommenders* (also sometimes called *hybrid recommenders*) combine both content-based and behavior-based recommender approaches. Since collaborative filtering tends to work best for items with many signals, but works poorly when few or no signals are present, it is often most effective to use content-based features as a baseline and then layer a collaborative filtering model on top. This way, if few signals are present, the content-based matcher will still return results, whereas if there are many signals, the collaborative filtering algorithm will take greater prominence when ranking results. Incorporating both approaches can give you the best of both worlds: high-quality crowdsourced matching, while avoiding the cold-start problem for newer and less-well-discovered content. Figure 9.5 demonstrates how this can work in practice.





**Figure 9.5 Multimodal recommendations combine both content-based matching and collaborative filtering into a hybrid matching algorithm.**

You can see in figure 9.5 that the user could interact with either the drill (which has no signals) or the screwdriver (which has previous signals from other users, as well as content), and the user would receive recommendations in both cases. This provides the benefit that signals-based collaborative filtering can be used, while also enabling content-based matching for items with insufficient signals.

We'll implement a collaborative filtering model in the next section, followed by a hybrid personalized search system in section 9.4.

## 9.3 Implementing collaborative filtering

In this section, we'll implement a collaborative filtering algorithm. We'll use user-item interaction signals and demonstrate how to learn latent (hidden) features from those signals that represent users' preferences. We'll then use those learned preferences to generate recommendations.

Pure collaborative filtering, as in figure 9.2, allows us to learn the similarity between items based entirely on user-interaction patterns with those items. This is a powerful concept, as it allows learning about items without any knowledge of the items themselves (such as titles, text, or other attributes).

### 9.3.1 Learning latent user and item features through matrix factorization

Collaborative filtering often uses a technique called *matrix factorization* to learn latent features about items based on user interactions. Latent features are features that are not directly observed but are inferred from other observed features. For example, assume you have four users with the following movie purchase history:

- User 1—*Avengers: Endgame*, *Black Panther*, and *Black Widow*
- User 2—*Black Widow*, *Captain Marvel*, and *Black Panther*
- User 3—*Black Widow*, *The Dark Knight*, and *The Batman*
- User 4—*The Little Mermaid*, *The Lion King*, and *Toy Story*
- User 5—*Frozen*, *Toy Story*, and *The Lion King*

Are there patterns to these purchases? If you know the titles or descriptions, you could infer the following:

Users 1–3:

- All of these are movies about superheroes.
- Most of them were made by Marvel Studios, though some were made by Warner Brothers (DC Comics).
- They are all action movies.
- They are not suitable for small children due to violence and/or language.

Users 4–5:

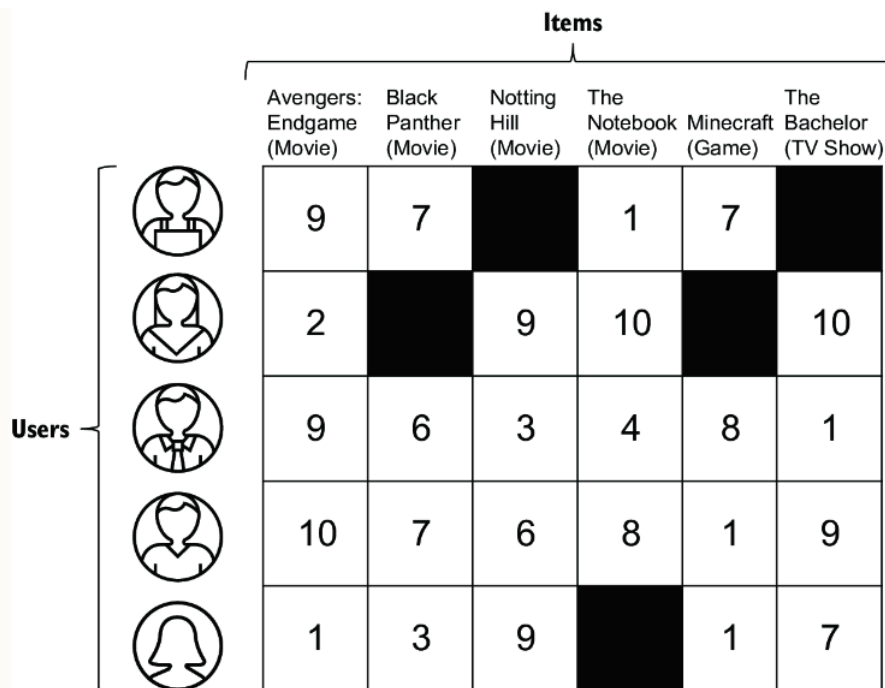
- All of them are animated movies.
- All of them are suitable for small children.
- All of them are made by Disney/Pixar.

Imagine you don't have access to anything other than the product IDs, though. By using matrix factorization, it is possible to observe how users interact with items and to infer latent features about those items. If the features listed in the previous bullet points are the most predictive of the purchasing behavior of similar users, they are likely to be represented in the latent features learned by matrix factorization. Matrix factorization is also likely to discover other features that are not as obvious.

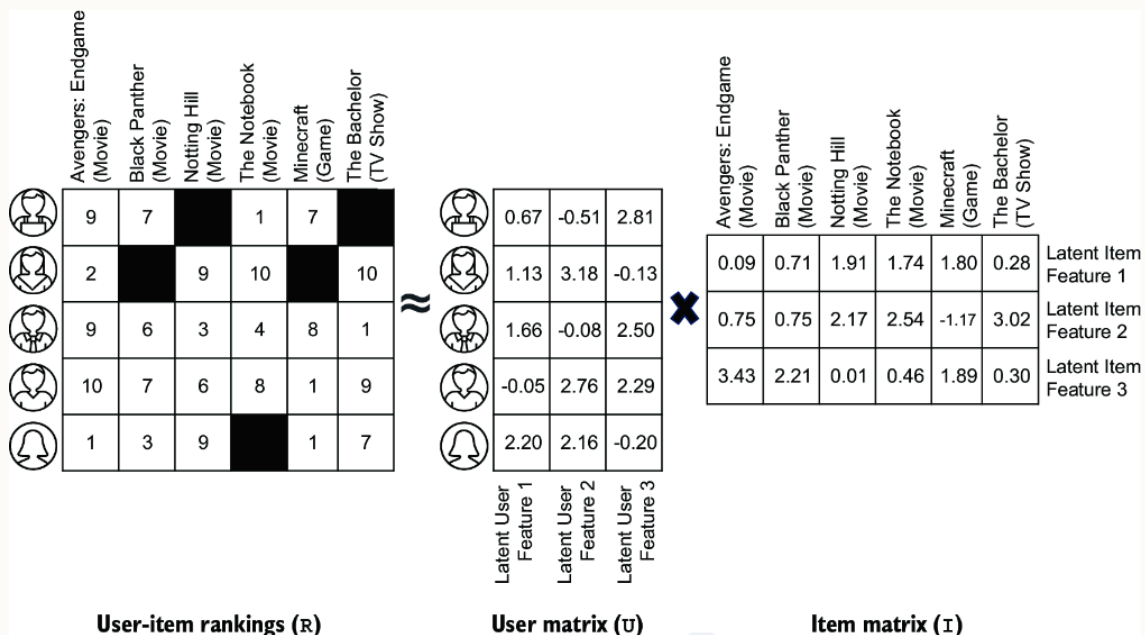
As a different example, in the RetroTech dataset, user signals may show one group of users purchasing stainless-steel microwaves, stainless-steel refrigerators, and stainless-steel dishwashers. Another group of users may be purchasing black microwaves, black refrigerators, and black dishwashers. By clustering the user-item interactions together, it's possible to statistically determine a latent feature that separates these items by color. Additionally, one group may be purchasing televisions, PlayStations, and DVD players, and another group may be purchasing iPhones, phone cases, and screen protectors. By clustering these behaviors together, we can differentiate these product categories (home theaters versus mobile phones) into one or more latent features.

Figure 9.6 demonstrates an example user-item interaction matrix for a few products and users. The numbers are ratings representing how strongly a user (y-axis) is interested in an item (x-axis), with a purchase being weighted higher than an add-to-cart action, and an add-to-cart signal being weighted higher than a click. The empty cells represent no interaction between the user and the item.





Given the user-item interaction matrix, our goal is to figure out *why* particular items are preferred by each user. We assume that some combination of user interests and item similarities explain these preferences. Matrix factorization, therefore, takes the user-item ratings matrix and breaks it into two separate matrices—one mapping each user to a set of features, and one mapping each item to a set of features.

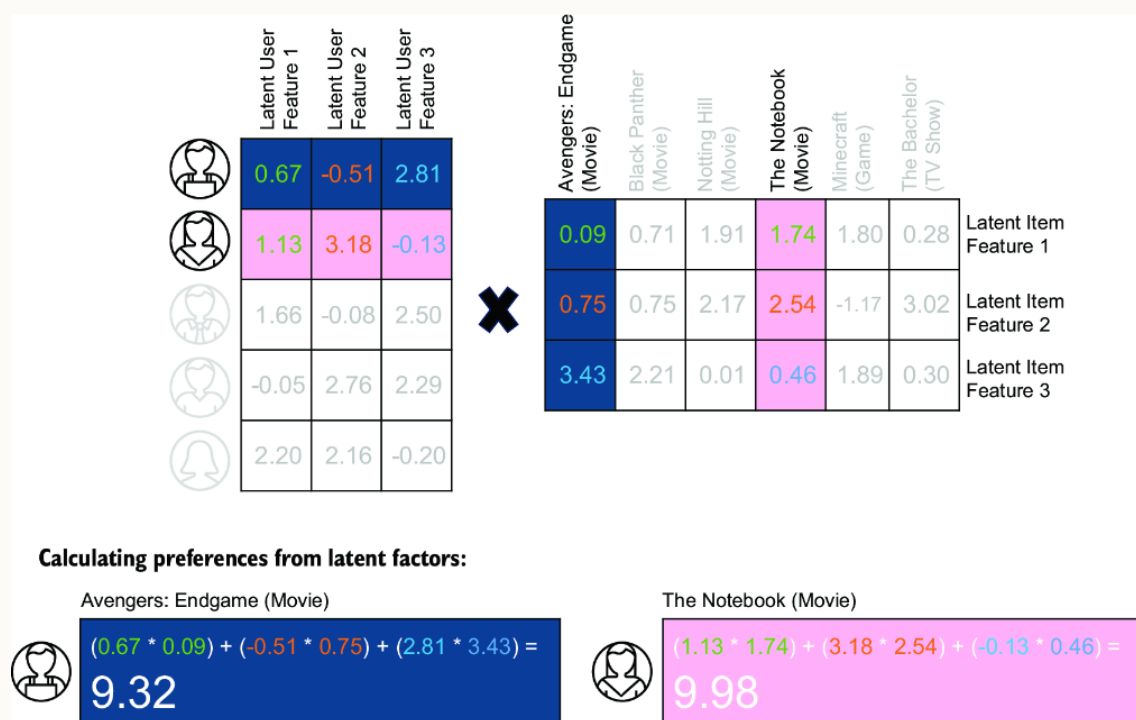


Each row in the user matrix (  $U$  ) is a vector representing one user, with each column representing one of three latent user features (labeled Latent User Feature 1, Latent User Feature 2, and Latent User Feature 3 ). In the item matrix (  $I$  ), each column is a vector representing one item, with each row representing one of three latent item features (la-

beled Latent Item Feature 1, Latent Item Feature 2, and Latent Item Feature 3).

We don't have names for these latent features or know exactly what they represent, but they are discovered mathematically and are predictive of actual user-item interests. The number of latent features is a hyperparameter that can be tuned, but it is set to 3 in this example. This means that each user is represented by a vector with three dimensions (latent features), and each item is also represented by a vector with three dimensions (latent features).

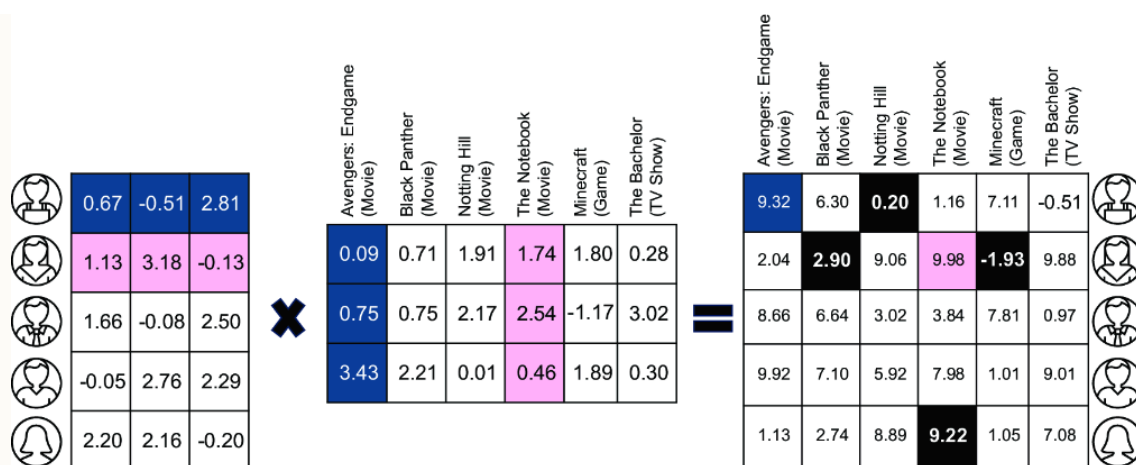
Once matrices  $U$  and  $I$  are learned, they can thereafter be used independently to predict the similarity between any user and item (by comparing users in  $U$  with items in  $I$ ), between any two users (by comparing a user in  $U$  with another user in  $U$ ), or between any two items (by comparing an item in  $I$  with another item in  $I$ ). We will focus only on the user-item similarity as a means of personalizing recommendations for each user. Figure 9.8 demonstrates how to generate an item rating prediction for any user.



**Figure 9.8** Calculating a user-item preference from the factorized matrices. Multiply each latent user feature value (first, second, and third values in the row for the user) by the corresponding latent item feature value (first, second, and third values in the column for the item), and then sum the results. This is the predicted user-item preference for the chosen user and item.

For the first user (first row in  $U$ ), we can generate a predicted rating for the movie *Avengers: Endgame* (first column in  $I$ ) by performing a dot product between the first row of the user matrix  $U$  (0.67, -0.51, 2.81) and the first column of the item matrix  $I$  (0.09, 0.75, 3.43), which results in a predicted rating of  $(0.67 * 0.09) + (-0.51 * 0.75) + (2.81 * 3.43) = 9.32$ . Likewise, for the second user (second row in  $U$ ), we can generate a predicted rating for the movie *The Notebook* (fourth column in  $I$ ) by performing a dot product between the second row of the user matrix  $U$  (1.13, 3.18, -0.13) and the fourth column of the item matrix  $I$  (1.74, 2.54, 0.46), which results in a predicted rating of 9.98.

While performing individual predictions between a single user and item may be helpful in some cases, such as for generating real-time recommendations immediately after an incremental user interaction, it is often more useful to generate a full user-item matrix  $R'$  of predicted ratings for all users and items. Figure 9.9 demonstrates a final user-item matrix  $R'$  generated (on the far-right) by performing a dot product of the user matrix  $U$  with the item matrix  $I$ .



**Figure 9.9 Reconstituted user-item matrix  $R'$ , with previous calculations from figure 9.8 highlighted. Note that the empty values from the original user-item matrix  $R$  are now filled in with predicted values (highlighted in black).**

When taking the dot product of the user matrix and the item matrix ( $U \cdot I$ ), the resulting user-item matrix  $R'$  should be as close as possible to the original user-item matrix  $R$ . Minimizing the difference between the original matrix  $R$  and predicted matrix  $R'$  is the training optimization goal of matrix factorization. The closer the two matrices are, the better the model's ability to predict similar personalized recommendations in the future.

In practice, the latent features don't perfectly represent all the potentially relevant features. By training with a loss function that reduces the difference between the original  $R$  and predicted  $R'$ , however, the model will maximize the chances of representing  $R$  and thus be able to best predict future recommendations based upon past user-item interactions.

### 9.3.2 Implementing collaborative filtering with Alternating Least Squares

One popular algorithm for pure collaborative filtering (based only on user interaction with items) is *Alternating Least Squares* (ALS). ALS is an iterative algorithm that performs matrix factorization by alternating between learning the latent features of items and the latent features of users.

The logic behind ALS is that latent features in a user-item ratings matrix are a combination of the user's latent features and the items' latent features. While the relative weights between users and items for each latent feature are not known upfront, it is possible to begin learning user feature weights by initially using random item weights and freezing them (keeping them constant). As the user feature weights begin to coalesce, they can be frozen and used as inputs when learning the item feature weights. ALS then continues to alternate between further training the user features matrix (with the latest item feature weights frozen) and the item features matrix (with the latest user feature weights frozen). This process is repeated for a configurable number of iterations until the weights of both matrices are well-balanced and optimized. By alternating between learning the latent features of items and users, ALS can iteratively learn the best combined weights of both matrices to improve the predictive power of the model.

The number of latent features learned using matrix factorization is a hyperparameter, called the *rank*. The higher the rank, the more granular the features you can learn, but you also tend to need more data points to reliably learn more granular features. While you won't be able to apply a label to each latent feature (features are just represented as numbers), it's still possible to discover meaningful categories in the data that best predict similar items. ALS is a popular algorithm for collaborative filtering because it is relatively easy to implement and can scale to large datasets.

In this section, we'll discuss how to implement ALS using Spark to generate a recommendations model based on user-item interactions. We'll use the RetroTech dataset, since it contains user-item interactions for a set of products. We'll use user-item interactions to learn latent features about both users and items, and then we'll use those latent features to generate future recommendations.

We'll start by generating a list of implicit preferences for each user-item pair using Spark's built-in ALS implementation. Listing 9.1 generates a `user_product_implicit_preferences` collection, assigning a rating based on the strength of the user interaction.

#### Listing 9.1 Generating implicit user-item ratings from user signals

```
click_weight = 1  #1
add_to_cart_weight = 0  #1
purchase_weight = 0  #1

signals_collection = engine.get_collection("signals")

mixed_signal_types_aggregation = f"""
SELECT user, product,
       (click_boost + add_to_cart_boost + purchase_boost) AS rating
FROM (
  SELECT user, product,
         SUM(click) AS click_boost,
         SUM(add_to_cart) AS add_to_cart_boost,
         SUM(purchase) AS purchase_boost
  FROM (
    SELECT s.user, s.target AS product,
           IF(s.type = 'click', {click_weight}, 0) AS click,
           IF(s.type = 'add-to-cart', {add_to_cart_weight}, 0) AS add_to_cart,
           IF(s.type = 'purchase', {purchase_weight}, 0) AS purchase
    FROM signals s
    WHERE (s.type != 'query')) AS raw_signals
  GROUP BY user, product) AS per_type_boosts"""

signals_agg_collection = \ #2
  aggregate_signals(signals_collection, "user_product_implicit_preferences", #2
                   mixed_signal_types_aggregation) #2
```

#1 Only click signals are currently weighted, but weights can be set per signal type.

#2 Aggregates all signals to generate a single rating per user-item pair

We modeled support for clicks, add-to-cart, and purchase signals, though we only assigned a weight of 1 to clicks and 0 to both add-to-cart and purchase signals. We did this to keep the math more straightforward for the ALS algorithm, but you can experiment with turning on add-to-cart or purchase signals by increasing their weights to a positive number. These weights are somewhat arbitrary, but the idea is to differentiate the strength of the user's product interest based on their level of interaction. You could also simplify by just assigning a rating of 1 for each user-item pair if you don't have confidence that more interactions by a user necessarily indicates a stronger rating or that the weights you've chosen are meaningful.

With our user-item ratings prepared, we'll generate a dataframe from the prepared collection to train and test the model. Our dataset contains less than 50,000 products, and we'll be using all of them in listing 9.2; however, you may want to modify the `top_product_count_for_recs` to a substantially lower number if you want to run through it quickly. Depending on your hardware and Docker resource configuration, it could take anywhere from several

minutes to several days to run. For a quick (but low-quality) run, consider testing with 1,000 products initially ( `top_product_count_for_recs=1000` ) and then scaling up as you feel comfortable.

### Listing 9.2 Preparing the user-product-ratings data for training

```
create_view_from_collection(signals_agg_collection,
                           "user_product_implicit_preferences")

top_product_count_for_recs = 50000 #1
user_preference_query = f"""
SELECT user, product, rating #2
FROM user_product_implicit_preferences
WHERE product IN (
    SELECT product FROM (
        SELECT product, COUNT(user) user_count
        FROM user_product_implicit_preferences
        GROUP BY product
        ORDER BY user_count DESC #3
        LIMIT {top_product_count_for_recs} #3
    ) AS top_products)
ORDER BY rating DESC"""

user_prefs = spark.sql(user_preference_query)
```

**#1** Decreasing the number of products can speed up training, but with reduced accuracy.

**#2** Returns the user, product, and rating

**#3** Limits the number recommendations to the most popular products

Our dataframe contains three columns: `user`, `product`, and `rating`. For performance reasons, many machine learning algorithms (including Spark's ALS implementation, which we will be using) prefer to deal with numeric IDs instead of strings. Spark contains a `StringIndexer` helper object that can be used to convert string IDs to numeric IDs, and a corresponding `IndexToString` object that can be used to convert the numeric IDs back to string IDs. Listing 9.3 integrates this ID conversion into our dataframe.

### Listing 9.3 Converting IDs to integers for Spark's ALS algorithm

```
def order_preferences(prefs):
    return prefs.orderBy(col("userIndex").asc(),
                          col("rating").desc(),
                          col("product").asc())

def strings_to_indexes(ratings, user_indexer,
                      product_indexer):
    transformed = product_indexer.transform( #1
        user_indexer.transform(ratings)) #1
    return order_preferences(transformed)

def indexes_to_strings(ratings, user_indexer,
                      product_indexer):
    user_converter = IndexToString(inputCol="userIndex", #2
                                   outputCol="user", #2
                                   labels=user_indexer.labels) #2
    product_converter = IndexToString(inputCol="productIndex", #2
                                      outputCol="product", #2
                                      labels=product_indexer.labels) #2
    converted = user_converter.transform( #3
        product_converter.transform(ratings)) #3
    return order_preferences(converted)

user_indexer = StringIndexer(inputCol="user", #4
                             outputCol="userIndex").fit(user_prefs) #4
product_indexer = StringIndexer(inputCol="product", #5
                                outputCol="productIndex").fit(user_prefs) #5

indexed_prefs = strings_to_indexes(user_prefs, user_indexer, product_indexer)
indexed_prefs.show(10)
```

**#1 Transforms user and product columns into index columns for the dataframe**

**#2 The numeric index-to-string mappings for product and user**

**#3 Performs the index-to-string transformation for the user identifier**

**#4 Maps the string user field to an integer index named userIndex**

**#5 Maps the string product field to an integer index named productIndex**

Output:

```
+-----+-----+-----+-----+-----+
|  user|  product|rating|userIndex|productIndex|
+-----+-----+-----+-----+-----+
|u159789|008888345435|    1|      0.0|      5073.0|
|u159789|014633196870|    1|      0.0|      4525.0|
|u159789|018713571687|    1|      0.0|     10355.0|
|u159789|024543718710|    1|      0.0|       263.0|
|u159789|025192979620|    1|      0.0|     12289.0|
|u159789|025193102324|    1|      0.0|      9650.0|
|u159789|085391163121|    1|      0.0|      9196.0|
|u159789|720616236029|    1|      0.0|      2781.0|
|u159789|801213001996|    1|      0.0|     28736.0|
|u159789|813985010007|    1|      0.0|      5819.0|
+-----+-----+-----+-----+-----+
only showing top 10 rows
```

As you can see from listing 9.3, our dataframe now contains two additional columns: `userIndex` and `productIndex`. We'll use these numeric IDs going forward in the ALS implemen-

tation code, before we call the `indexes_to_strings` function at the very end to convert back to our original string IDs.

Now that our user-item preferences dataframe is prepared, it's time to invoke the ALS algorithm. ALS requires three parameters: `userCol`, `itemCol`, and `ratingCol`, which correspond to the `userIndex`, `productIndex`, and `rating` columns in our dataframe. We'll also set a few other parameters, including the following:

- `maxIter=3` (the maximum number of iterations to run)
- `rank=10` (the number of latent features to learn)
- `regParam=0.15` (the regularization parameter)
- `implicitPrefs=True` (whether to treat the ratings as implicit or explicit)
- `coldStartStrategy=drop` (how to handle new users or items that were not present in the training data)

Listing 9.4 demonstrates how to invoke ALS with these parameters.

#### Listing 9.4 Training an ALS model using Spark

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row

als = ALS(maxIter=3, rank=10, regParam=0.15, implicitPrefs=True,
          userCol="userIndex", itemCol="productIndex", ratingCol="rating",
          coldStartStrategy="drop", seed=0)

(training_data, test_data) = \ #1
    user_prefs.randomSplit([0.95, 0.05], 0) #1

training_data = strings_to_indexes(training_data, user_indexer, product_indexer)
test_data = strings_to_indexes(test_data, user_indexer, product_indexer)

model = als.fit(training_data) #2
predictions = model.transform(test_data) #3
evaluator = RegressionEvaluator(metricName="rmse", #3
                                labelCol="rating", #3
                                predictionCol="prediction") #3
rmse = evaluator.evaluate(predictions) #3
print(f"Root-mean-square error = {rmse}") #3
```

#1 Splits the preferences, 95% as training data and 5% as test data

#2 Trains the ALS model with the user preferences in the training set

#3 Measures the trained model against the user preferences in the test set

Output:

```
Root-mean-square error = 1.0007877733299877
```

You have now trained a recommendation model! We split the data into a training set (95%) and a test set (5%), built the ALS model, and then ran an evaluator to calculate a root mean square error (RMSE) loss function to measure the quality of the model. The RMSE is a measure of how far off the predicted ratings are from the actual ratings, so the lower the RMSE, the better the model. The absolute value of the RMSE is less important than the relative value across different model training passes, as the calculation depends on the scale used in the underlying data. If you increase the `maxIter`, find an optimal `rank`, and increase the `top_product_count_`



for `recs` when preparing the user-product-ratings data, you'll likely see the RMSE decrease a bit due to improvement in the model.

Now that the model is trained, we can use it to generate recommendations. Listing 9.5 demonstrates how to generate item recommendations for all users. We'll generate 10 recommendations for each user and display the top 5 users' recommendations.

#### Listing 9.5 Generating user-item recommendations from the ALS model

```
indexed_user_recs = model.recommendForAllUsers(10) \
                        .orderBy(col("userIndex").asc())
indexed_user_recs.show(5, truncate=64)
```

Output:

```
+-----+-----+
|userIndex|                                recommendations|
+-----+-----+
|      0|[ {6, 0.022541389}, {13, 0.015104328}, {36, 0.010634022}, {20,...|
|      1|[ {13, 0.009001873}, {3, 0.007981183}, {23, 0.0050935573}, {31...|
|      2|[ {9, 0.06319133}, {17, 0.04681776}, {3, 0.041046627}, {14, 0....|
|      3|[ {17, 0.0145240165}, {14, 0.01413305}, {12, 0.012459144}, {39...|
|      4|[ {14, 0.006752351}, {4, 0.004651022}, {10, 0.004487163}, {17,...|
+-----+-----+
only showing top 5 rows
```

Note that the format of the recommendations is a bit awkward. We're stuck with the `userIndex` instead of our original `user`, and the `recommendations` column is an array of structs, with each struct containing a `productIndex` and a `rating`. Let's clean this up by converting each user-item recommendation into a row and replacing the `userIndex` and `productIndex` values with our original `user` and `product` IDs. Listing 9.6 demonstrates how to do this.

#### Listing 9.6 Converting recommendations into a final, cleaned-up format

```
column_exploder = explode("recommendations").alias("productIndex_rating")
user_item_recs = indexed_user_recs.select("userIndex", column_exploder) \
                                .select("userIndex", col("productIndex_rating.*"))
user_item_recs = indexes_to_strings(user_item_recs, user_indexer,
                                    product_indexer)
user_item_recs = user_item_recs.select("user", "product",
                                       col("rating").alias("boost"))
```

In this listing, we first `explode` the recommendations into separate rows for each recommendation with `rec.productIndex` and `rec.rating` columns. After selecting `userIndex` onto each row, we select `rec.productIndex` as `productIndex` and `rec.rating` as `rating`. Finally, we convert back to `user` and `product` from `userIndex` and `productIndex`, and we return `user`, `product`, and `boost`.

Let's save our recommendations to a collection for future use. This will enable us to serve the recommendations instantly from the search engine or to use them as boosts to personalize search results. Listing 9.7 writes our user-item recommendations dataframe to a `user_item_recommendations` collection in the search engine, following a data format similar to the one we used in chapter 8 to represent signal boosts.

Listing 9.7 Indexing the recommendations into the search engine

```
recs_collection = engine.create_collection("user_item_recommendations")
recs_collection.write(user_item_recs)
```

You have now generated item recommendations for users based on their interactions with items, and you’ve saved them for future use in the `user_item_recommendations` collection in the search engine. Next, we’ll demonstrate how we can serve these recommendations and use them to personalize search results.

9.3.3 Personalizing search results with recommendation boosting

With user-item recommendations generated, we can now personalize search results. The only difference between our collection schema for the `signals_boosts` collection in chapter 8 and the `user_item_recommendations` collection here is the replacement of the `query` column with a `user` column. In other words, whereas signals boosting is based on matching a particular keyword query and applying associated item relevance boosts, personalization is based on matching a particular user and applying associated item relevance boosts.

With our recommendations collection now populated from listing 9.7, we can either serve the recommendations directly (no keyword query) or use the recommendations to personalize search results by boosting them based on the user’s recommendations.

PURE COLLABORATIVE RECOMMENDATIONS

Serving recommendations directly is straightforward, so we’ll start there. Listing 9.8 shows recent signals for one of our users, for whom we’ll demonstrate these personalization techniques.

Listing 9.8 Interaction history for our target user

```
def signals_request(user_id):
    return {"query": "*",
            "return_fields": ["signal_time", "type", "target"],
            "order_by": [("signal_time", "asc")],
            "filters": [("user", user_id)]}

user_id = "u478462" #1
signals_collection = engine.get_collection("signals")

request = signals_request(user_id)
previous_signals = signals_collection.search(**request)["docs"]
print_interaction_history(user_id, previous_signals)Previous Product Interactions for user u478462
+-----+-----+-----+-----+
|signal_time|      type|      target|                                name|
+-----+-----+-----+-----+
|05/20 06:05|    query|      apple|                                apple|
|05/20 07:05|  click|885909457588|Apple® - iPad® 2 with Wi-Fi - 16GB...|
|05/20 07:05|add-to-cart|885909457588|Apple® - iPad® 2 with Wi-Fi - 16GB...|
|05/20 07:05| purchase|885909457588|Apple® - iPad® 2 with Wi-Fi - 16GB...|
|05/25 06:05|    query|    macbook|                                macbook|
|05/25 07:05|  click|885909464043|Apple® - MacBook® Air - Intel® Cor...|
+-----+-----+-----+-----+
```

#1 User for whom we’ll be personalizing results

Based on the user’s history, it’s clear that they are interested in Apple products, tablets, and computers. The following listing demonstrates how to serve up recommendations for this user from our `user_item_recommendations` collection.

#### Listing 9.9 Serving recommendations using a signals boosting query

```
def get_query_time_boosts(user, boosts_collection):
    request = {"query": "*",
               "return_fields": ["product", "boost"],
               "filters": [("user", user)] if user else [],
               "limit": 10,
               "order_by": [("boost", "desc")]}

    response = boosts_collection.search(**request)
    signals_boosts = response["docs"]
    return " ".join(f'"{b["product"]}"^{b["boost"]} * 100'
                    for b in signals_boosts)

def search_for_products(query, signals_boosts):
    request = product_search_request(query if query else "*") #1
    if signals_boosts:
        request["query_boosts"] = ("upc", signals_boosts)
    return products_collection.search(**request)

user = "u478462"
boosts = get_query_time_boosts(user, recs_collection)
response = search_for_products("", boosts) #2

print(f"Boost Query:\n{boosts}")
display_product_search("", response["docs"])
```

#1 Function omitted for brevity; it can be seen in listing 4.3.

#2 Queries the recommendation collection for indexed product boosts

Figure 9.10 shows the output from listing 9.9. At the top, you’ll notice a “Boost Query” listed, showing the top-recommended products for the user, along with their relative boost for the user (which was calculated as `rating * 100`). Under that boost query, you’ll see the boosted search results for this blank keyword search, which are the raw recommendations for the user.

Boost Query:  
"885909457588"^83.317953 "022265004289"^19.800967 "024543742180"^8.756707 "635753493559"^6.914275  
"045496880484"^6.1463382 "635753493573"^5.834811 "885909457595"^5.7118796 "885370315080"^5.6894064  
"612572171585"^5.5108927 "885909395095"^5.2595586



**Name:** Apple® - iPad® 2 with Wi-Fi - 16GB - Black  
**Manufacturer:** Apple®



**Name:** Samsung - Galaxy Tab 10.1 - 16GB - Metallic Gray  
**Manufacturer:** Samsung



**Name:** Samsung - Galaxy Tab 10.1 - 32GB - Metallic Gray  
**Manufacturer:** Samsung



**Name:** Apple® - iPad® 2 with Wi-Fi - 32GB - Black  
**Manufacturer:** Apple®

**Figure 9.10 Recommendations for a user based only on collaborative filtering**

The recommendations boost a 16 GB iPad to the top, which makes sense given that the user previously searched for and clicked on the 16 GB iPad, with another Apple iPad (a 32 GB model) ranked fourth. You also see other tablets made by competing manufacturers with similar configurations within the top recommendations. This is a good example of how collaborative filtering can help surface items that might not directly match the user's previous interactions (with only an Apple laptop and iPads), but which may still be relevant to the user's interests (similar tablets to iPads).

Recommendations like these can be useful to integrate alongside traditional search results, or possibly to even insert into a set of search results. But it's also possible to use them as boosts to your keyword ranking algorithm to personalize search results, which we'll explore next.

#### **PURE KEYWORD SEARCH VS. PERSONALIZED SEARCH**

Instead of serving recommendations independently of keyword search, it can also be useful to blend them in as additional signals in your search ranking algorithm to personalize the results. Going back to our last example, imagine that our user who is interested in iPads and MacBooks from Apple performs a keyword search for `tablet`. How would this look different than if the tablet recommendations were used to personalize the search results? Listing 9.10 runs the query before and after applying signals boosts based on the user's personalized recommendations.

## Listing 9.10 Non-personalized vs. personalized search results

```
query = "tablet"
response = search_for_products(query, None) #1
print(f"Non-personalized Query")
display_product_search(query, response["docs"])

response = search_for_products(query, boosts) #2
print(f"Personalized Query")
display_product_search(query, response["docs"])
```

#1 Non-personalized search results (keyword search only)

#2 Personalized search results (keyword + user-item recommendations boosting)

Figure 9.11 shows the output of the non-personalized query for `tablet`, whereas figure 9.12 shows the output once the recommendation boosts have been applied to personalize the search results.

The personalized search results are likely much more relevant for this user than the non-personalized results. It's worth noting that in our implementation, the personalization is *only* applied as a relevance boost. This means products that don't match the user's explicit query will not be returned, and all items that do match the query will still be returned; the only difference is the ordering of the products, as items personalized to the user should now show up on the first page.

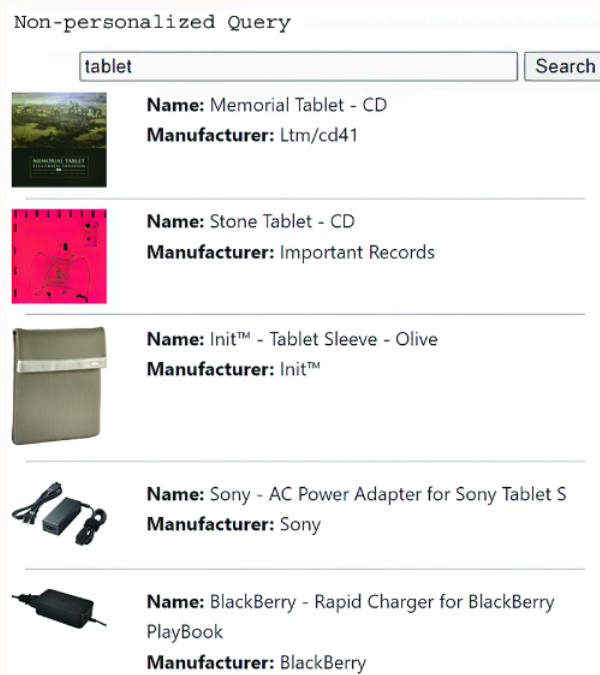
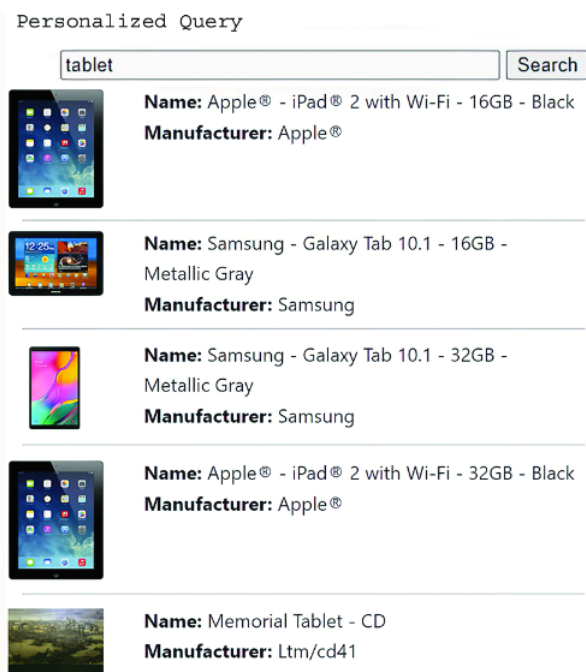


Figure 9.11 Traditional keyword search for `tablet` with no personalization applied



**Figure 9.12 Personalized search for `tablet`, where the user has shown an interest in the Apple brand**

Also note that after the boosted recommendations (tablets from the recommendation example) the fifth search result is an item from the non-personalized search results, the “Memorial Tablet” titled CD.

This implies two things:

- If you are personalizing search results and not just serving pure recommendations, you’ll likely want to generate more than 10 recommendations per user, particularly since recommendations only show up if they also match a user’s explicit query.
- The non-personalized relevance algorithm is still critical. If signals boosting based on the query (per chapter 8) was applied in addition to recommendations boosting (based on the user), you would see popular tablets at the top (and not the tablet sleeve and CD), with the personalized tablets then moving up higher among the popular results because of the personalization.

We’ve now learned how collaborative filtering works through matrix factorization, we’ve implemented recommendations based on a collaborative filtering algorithm (ALS), and we’ve demonstrated how to use those recommendations to personalize search results. In the next section, we’ll explore another technique for personalization based on document embeddings.

## 9.4 Personalizing search using content-based embeddings

In the previous section, we used user signals to learn personalized boosts for particular items. These boosts were generated by learning latent features about users and items using matrix factorization on user-item interaction patterns.

You could also use these latent factors directly to cluster users or items together. Unfortunately, there isn’t a great way to reliably map queries into particular clusters of items based only on user-interaction signals without having already seen the corresponding queries before (the cold-start problem, again). Thankfully, it is quite rare for a search engine to *not* have additional knowledge of items such as titles, descriptions, and other attributes.

In this section, we’ll look at a hybrid approach using both content-based understanding and user-interaction patterns to build an evolving user profile to personalize search results.

### 9.4.1 Generating content-based latent features

We’ve covered many techniques for utilizing fields to filter and boost on explicit attributes in documents. Chapters 5–7, in particular, focused on generating knowledge graphs and parsing domain-specific entities to help with context-dependent relevance.

While those techniques can certainly be useful for implementing personalized search (and we encourage you to experiment with them), we’re going to explore a different approach in this section. Instead of using explicit attributes, we’re going to use latent features learned from the content of documents to generate personalized search results. We’ll use a large language model (LLM) to generate embeddings for each document, and then we’ll use those embeddings along with user interactions with documents to build an evolving user profile. Finally, we’ll use that user profile to personalize search results.

Figure 9.13 demonstrates conceptually how using an LLM to generate embeddings for documents works. Similar to how we used matrix factorization in section 9.3.1 to create a matrix that mapped each item to its list of latent features, we’ll use an LLM to generate a vector of latent features for each document. We’ll extract these latent features based on the text of the document mapped into a vector space that has already been learned by the LLM. Don’t worry for now about the mechanics of *how* the LLM is trained—we will cover that in depth in chapters 14 and 15. Just know that it is trained on a large corpus of text, and it learns how to map words and phrases into a vector space using some number of latent features that represent the meaning of the text. Each of the dimensions in the vector space represents a latent feature, and the value of each dimension represents how strongly that latent feature is represented in the text.

		Items						
Latent features (content-based)	1	4.1	4.1	8.2	7.7	1.9	4.0	≈ size?
	2	9.8	1.2	2.2	0.8	1.2	0.3	≈ color?
	3	9.9	9.9	3.2	1.7	9.2	0.2	≈ computer-like?
	...	...	...	...	...	...	...	
	768	3.0	3.0	7.3	5.9	4.3	0.3	≈ price?

**Figure 9.13** Item embeddings from an LLM. Each dimension in the vector space represents a latent feature, and the value of each dimension represents how strongly that latent feature is represented in the text for that item.

The values in figure 9.13 are for illustration purposes and are not the actual values that would be generated by our LLM. We have assigned simplified labels to the features to describe what they seem to represent (“size”, “color”, “computer-like”, and “cost”), but in a real-world scenario, these features would be unlabeled and would represent more complex latent features combining many different aspects learned by the LLM’s deep neural network during its training process.

For our examples, we’ll be using the `all-mpnet-base-v2` LLM, a publicly available model (<https://huggingface.co/sentence-transformers/all-mpnet-base-v2>) that serves as a good general-purpose LLM for semantic search and clustering over sentences and short paragraphs, like those in our RetroTech dataset. It is a lightweight model (only 768 dimensions) that was trained on over 1.17 billion sentence pairs from across the web, providing a good general-knowledge base.



The following listing retrieves the fields we need to pass to the LLM.

### Listing 9.11 Retrieving product data to generate embeddings

```
query = "SELECT DISTINCT name, string(upc), short_description FROM products"
spark.sql(query).createOrReplaceTempView("products_samples")
product_names = dataframe.select("name").rdd.flatMap(lambda x: x).collect()
product_ids = dataframe.select("upc").rdd.flatMap(lambda x: x).collect()
```

To generate embeddings, we first use Spark to create a new `products_samples` table containing a subset of fields useful for generating embeddings and identifying the associated products. Listing 9.12 demonstrates how we can generate embeddings for each product using the `all-mpnet-base-v2` LLM and the `Sentence_Transformers` library. We'll generate a `product_embeddings` object containing a 768-dimension vector for each product, along with a `product_names` object containing the name of each product and a `product_ids` object containing the ID of each product.

### Listing 9.12 Generating product embeddings

```
from sentence_transformers import SentenceTransformer #1
transformer = SentenceTransformer("all-mpnet-base-v2") #1
... #2

def get_embeddings(texts, model, cache_name, ignore_cache=False):
    ... #2
    embeddings = model.encode(texts) #3
    ... #2
    return embeddings

product_embeddings = get_embeddings(product_names,
    transformer, cache_name="all_product_embeddings")
```

#1 Loads the all-mpnet-base-v2 LLM

#2 Optimization code to cache generated embeddings is omitted for brevity.

#3 Generates the 768-dimension vector embedding for all products

Because we're using the out-of-the-box `all-mpnet-base-v2` model, loading and generating embeddings for all our products is as simple as the code in listing 9.12. Because the process of generating embeddings for all products can take a while, the notebooks additionally contain some omitted code optimizations to cache and reuse embeddings to save extra processing time.

If we want to compare the similarity of two products, we can directly compare their vectors using dot product or cosine similarity calculations. The 768 features in the vector are pre-trained latent features of each document, similar to the latent features represented in the item feature matrix in figure 9.7. This means that we can now

- Generate embeddings for any item or query to get a vector representation of that item or query.
- Perform semantic search starting with any query embedding and find the closest (cosine or dot product) other embeddings.
- Use an item's embedding to generate recommendations for other items by finding the ones with the most similar (cosine or dot product) embeddings.

But what about generating user-based recommendations or personalized search results? In figure 9.7, we not only factored out latent item features, but we also factored out latent user features. The whole idea behind collaborative filtering in section 9.2 is that similar users interact

with similar items precisely *because* the items share features that overlap with those users’ interest. In other words, a vector representing a user’s interests should be similar to the vectors representing the items for which the user has expressed interest.

To personalize search results based on embedding vectors, we thus need to generate a vector representing the user’s interests. One way to do this is by taking the average of the vectors representing the items with which the user has interacted. This is a simple way to generate a vector representing *all* the user’s past interests, and it works surprisingly well in practice.

Unfortunately, personalizing *every* future search based on every past search can be a bit too aggressive, as users often perform unrelated searches for different types of items at different times. To avoid unhelpful over-personalization in these cases, it can be useful to first apply some guardrails across different item categories, which we’ll cover next.

### 9.4.2 Implementing categorical guardrails for personalization

The fact that someone searches for an item doesn’t always mean they want to see similar items. But if they do want personalization, it is usually a very bad idea to apply personalization across conceptual or categorical boundaries. For example, if someone watches the movie *The Terminator*, which contains violent time-traveling robots, it doesn’t mean they want to purchase a robot vacuum cleaner or a gun. As a concrete example from our dataset, imagine that someone previously expressed interest in a “Hello Kitty Water Bottle”, a “GE Electric Razor (Black)”, “GE Bright White Light Bulbs”, and a “Samsung Stainless-steel Refrigerator”. If they subsequently perform a search for `microwave`, which of the items from figure 9.14 would be most appropriate to recommend?

Previous Interactions:

Hello Kitty Water bottle

GE Electric Razor (Black)


GE Bright White Light Bulbs

Samsung Stainless-steel Refrigerator

No personalization guardrails

microwave

Search




Name:

Hello Kitty - 0.7 Cu. Ft. Compact Microwave

Manufacturer:

Hello Kitty




Name:

Panasonic - 1.2 Cu. Ft. Mid-Size Microwave - Stainless-Steel

Manufacturer:

Panasonic




Name:

Panasonic - 1.2 Cu. Ft. Mid-Size Microwave - Stainless-Steel

Manufacturer:

Panasonic



Name:

Panasonic - 1.2 Cu. Ft. Mid-Size Microwave - White

Manufacturer:


Panasonic

VS.

With categorical personalization guardrails

microwave

Search




Name:

Samsung - 1.8 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer:

Samsung




Name:

Samsung - 1.8 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer:

Samsung




Name:

Samsung - 1.6 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer:

Samsung



Name:

Samsung - 1.7 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer:

Samsung

Figure 9.14 Personalization guardrails can help prevent unrelated past interests from unexpectedly influencing future searches

While the user previously looked at “white” lights and a “black” electric shaver, there is no good reason to apply those color preferences to the unrelated category of “kitchen appliances”. Additionally, it is questionable whether the interest in a “Hello Kitty Water Bottle” would transfer over to an interest in a “Hello Kitty microwave”, or whether looking at “light bulbs” and an “electric shaver” made by the company “GE” would in any way translate into a user having a brand affinity for “GE” when looking at “kitchen appliances”. Given that this particular user has already shown an interest in another appliance (a refrigerator that is “stainless-steel”) made by the company “Samsung”, however, it is very reasonable to assume they would be more interested in other “stainless-steel” appliances made by “Samsung” (or at least by other companies beyond “GE”), such as the microwave for which they are now searching.

Personalization should be applied with a light touch. It is easy to make mistakes and to apply personalization in ways that are not helpful to the user (or are even frustrating and counter-productive), so it's usually better to err on the side of caution and ensure that personalization is only applied when it is likely to be helpful. One simple way to do this is to only apply personalization within similar categories as the query. This is one way of applying *guardrails* to personalization, and it is a very effective way to avoid applying personalization in ways that are likely to be unhelpful to the user.

While your data may or may not have an explicit category field to filter on, it's also possible to dynamically generate categories by clustering items together based on their similarity. This can be done by taking the embeddings for all items and clustering them to dynamically create a data-driven set of categories. The following listing demonstrates a simple method for generating clusters of items from their embeddings.

#### Listing 9.13 Generating dynamic categories from clustered products

```
def get_clusters(data, algorithm, args):
    return algorithm(**args).fit(data)

def assign_clusters(labels, product_names):
    clusters = defaultdict(lambda:[], {})
    for i in range(len(labels)):
        clusters[labels[i]].append(product_names[i])
    return clusters

args = {"n_clusters": 100, "n_init": 10, "random_state": 0} #1
algo = get_clusters(product_embeddings, cluster.KMeans, args) #1
labels = algo.predict(product_embeddings)
clusters = assign_clusters(labels, product_names) #2
```

#1 Generates 100 clusters using a KMeans clustering algorithm

#2 Assigns each product name to its corresponding cluster label

To ensure that our clustering worked well, we can inspect the top words in each cluster to ensure they are related and form a coherent category. Listing 9.14 demonstrates code to identify the top words in each cluster and to generate a 2D visualization of the clusters using principal component analysis (PCA) to map the 768-dimension embeddings down to two dimensions for visualization purposes.

### Listing 9.14 Inspecting popular terms from each product cluster

```
import collections, numpy as np, matplotlib.pyplot as plt
from adjustText import adjust_text
from sklearn.decomposition import PCA

plt.figure(figsize=(15, 15))
pca = PCA(100, svd_solver="full") #1
centers = algo.cluster_centers_ #1
plot_data = pca.fit_transform(centers) #1

points = []
for i, cluster_name in enumerate(plot_data): #2
    plt.scatter(plot_data[i,0], plot_data[i, 1], #2
               s=30, color="k") #2
    label = f"{i}_{\"_\".join(top_words(clusters[i], 2))}" #3
    points.append(plt.text(plot_data[i, 0], #4
                          plot_data[i, 1], #4
                          label, size=12)) #4
adjust_text(points, arrowprops=dict(arrowstyle="-", #5
                                    color="gray", alpha=0.3)) #5

plt.show()
```

**#1 Performs PCA to reduce embeddings down to two dimensions for visualization**

**#2 Loops through each cluster and plots it on the graph**

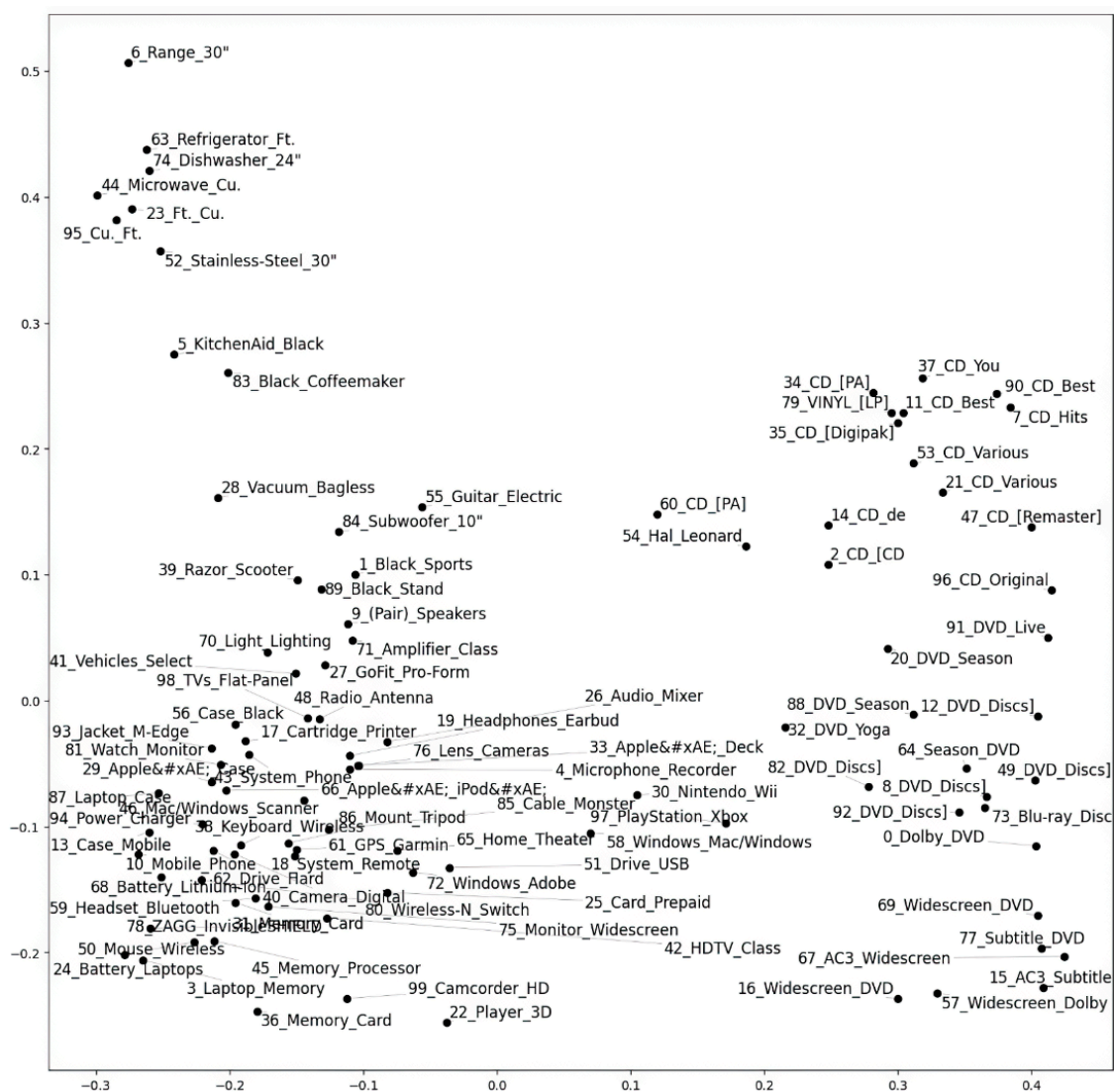
**#3 The top words function gets the most common words from a cluster.**

**#4 Adds a text label for each cluster with the cluster ID and top-N words in each cluster**

**#5 Display improvement: adjusts the text labels to minimize overlap**

Figure 9.15 shows the output of listing 9.14. Each dot represents a cluster, with the text label for each cluster including the cluster ID and the top words in that cluster.

While figure 9.15 may appear chaotic, representing all 100 clusters into which our nearly 50,000 products are categorized, you can see clear patterns in the semantic space. The top left of the graph contains kitchen appliances, and music tends to be at the top right of the remaining populated area of the graph (CDs in the top right, musical instruments and speakers in the top middle), and items related to video and data storage tend to be at the bottom of the graph (DVDs and Blu-ray at the bottom right, home theaters and cameras in the bottom middle, computer memory cards and storage in the bottom left along with other computer peripherals). Feel free to inspect the various categories and relationships between the clusters, but realize that they have been mapped down from 768 dimensions into 2 dimensions, so much of the richness represented by the KMeans clustering algorithm will be lost in the visualization.



**Figure 9.15 Clusters generated by KMeans clustering of all product embeddings, to be used for categorizing all queries and products**

Now that we have clusters available to categorize products (and signals corresponding to interacted-with products), we need to ensure that we can map queries into the correct clusters.

There are multiple ways to do this:

- *Model-driven*—Just pass the query through the LLM, and use the resulting embedding vector to find the closest categories.
- *Behavior-driven*—Use query signals and corresponding interaction signals (such as clicks) to determine the most likely categories for popular queries.
- *Content-driven*—Run a keyword or semantic search, and find the top categories in the results.
- *Hybrid*—Use any combination of these approaches.

The behavior-driven approach follows the signals-boosting methodology from chapter 8, but it aggregates by the categories associated with the top boosted documents instead of by queries. The content-driven approach enables you to use the other semantic search techniques explored in chapters 5–7. For simplicity, we’ll use the model-driven approach here and give deference to the LLM to determine the meaning of the query. The following listing demonstrates three different approaches for deriving the top categories for a query based on the embedding vectors.

### Listing 9.15 Comparing techniques for mapping queries to clusters

```
import sentence_transformers, heapq

def get_top_labels_centers(query, centers, n=2): #1
    query_embedding = transformer.encode([query], convert_to_tensor=False)
    similarities = sentence_transformers.util.cos_sim(
        query_embedding, centers)
    sim = similarities.tolist()[0]
    return [sim.index(i) for i in heapq.nlargest(n, sim)]

def get_query_cluster(query): #2
    query_embedding = transformer.encode([query], convert_to_tensor=False)
    return algo.predict(query_embedding)

def get_cluster_description(cluser_num):
    return "_".join(top_words(clusters[cluser_num], 5))

query = "microwave"
kmeans_predict = get_query_cluster(query)[0] #3
print("K-means Predicted Cluster:")
print(f"    {kmeans_predict} ({get_cluster_description(kmeans_predict)})")

closest_sim = get_top_labels_centers(query, centers, 1)[0] #4
print(f"\nCosine Predicted Cluster:")
print(f"    {closest_sim} ({get_cluster_description(closest_sim)})")

knn_cosine_similarity = get_top_labels_centers(query, centers, 5) #5
print(f"\nKNN Cosine Predicted Clusters: {knn_cosine_similarity}")
for n in knn_cosine_similarity:
    print(f"    {n} ({get_cluster_description(n)})")
```

**#1 Gets the top N clusters based on cosine similarity with the cluster centroids**

**#2 Gets the cluster based on the KMeans model's prediction**

**#3 Option 1: Predicts the nearest cluster (KMeans)**

**#4 Option 2: Finds the most-similar cluster (cosine similarity)**

**#5 Option 3 (recommended): Finds N-most-similar clusters (cosine similarity)**

Output:

```
K-means Predicted Cluster:
    44 (Microwave_Cu._Ft._Stainless-Steel_Oven)

Cosine Predicted Cluster:
    44 (Microwave_Cu._Ft._Stainless-Steel_Oven)

KNN Cosine Predicted Clusters: [44, 52, 5, 83, 6]
    44 (Microwave_Cu._Ft._Stainless-Steel_Oven)
    52 (Stainless-Steel_30"_Black_Range_Cooktop)
    5 (KitchenAid_Black_White_Stand_Mixer)
    83 (Black_Coffeemaker_Maker_Coffee_Stainless-Steel)
    6 (Range_30"_Self-Cleaning_Freestanding_Stainless-Steel)
```

In listing 9.15, we see three predictions being calculated: nearest cluster (K-means), most-similar cluster (cosine similarity), and the *N*-most-similar clusters (cosine similarity). The `get_top_labels_centers` function calculates the top *N* clusters based on the cosine similarity with the cluster centroids. The clustering function `get_query_cluster` calculates a cluster based on a K-means prediction.



The output from these three approaches demonstrates an important point. While the query is for `microwave`, we know that the categories were generated dynamically and may have overlap between products. The K-means model and cosine similarity approaches both choose category `44 (Microwave_Cu._Ft._Stainless-Steel_Oven)` in this example. While you're likely to find better results by relying on the cosine similarity to measure semantic similarity versus the K-means prediction, the categories returned from each are likely to be closely related. Thus, any personalization would benefit by being applied *across* each of the relevant categories instead of only one. Products can be split across multiple, related categories, and meaningful categories can be arbitrarily split based upon the number of items and nuances of the item descriptions.

To overcome the overlaps between similar categories, we recommend using the top-*N* cosine predicted clusters (Knn, option 3) instead of filtering to a single cluster. In the results from listing 9.15, this miscellaneous approach returns five related categories: `44` ("microwaves"), `52` ("stoves"), `5` ("miscellaneous appliances"), `83` ("counter appliances"), and `33` ("ovens").

We'll next use these predicted categories, along with the embeddings from a user's previous interactions, to personalize search results.

### 9.4.3 Integrating embedding-based personalization into search results

The final step in our personalization journey is to execute the personalized search. We could accomplish this in many different ways:

- Perform a weighted average between the query vector (embedding for `microwave`) and the vectors for the user's previous interactions within the predicted clusters. This would generate a single vector representing a personalized version of the user's query, so all results would be personalized.
- Perform a standard search, but then boost the results based on the average of the embeddings from a user's previous interactions within the predicted clusters. This would be a hybrid keyword- and vector-based ranking function, where the keyword search would be the primary driver of the results, but the user's previous interactions would be used to boost related results higher.
- Do one of the above, but then only personalize a few items in the search results instead of all the results. This follows a light-touch mentality so as not to disturb all of the user's search results, while still injecting novelty to enable the user to discover personalized items they may not have otherwise found.
- Perform a standard search (keyword or vector), but then rerank the results based on the weighted average between the query vector and the vectors for the user's previous interactions within the predicted clusters. This uses the original search to find the candidate results using the default relevance algorithm, but those results are reranked to boost personalized preferences higher.

We'll demonstrate the last technique, as it is easy to replicate across any search engine, since the personalization/reranking pass can be done as a final step after the original search. This technique will thus work well with both traditional search engines and vector databases.

Listing 9.16 demonstrates the two key functions we'll use to generate our personalization vector: a `get_user_embeddings` function that looks up the embeddings for a list of products and also returns the cluster associated with each product, and a `get_personalization_vector` function that can combine embeddings between a query and all relevant user-item interaction vectors.



## Listing 9.16 Functions for generating personalization vectors

```
def top_clusters_for_embedding(embedding, n=2):
    similarities = sentence_transformers.util.cos_sim(embedding, centers)
    sim = similarities.tolist()[0]
    return [sim.index(i) for i in heapq.nlargest(n, sim)]

def get_user_embeddings(products=[]): #1
    values = []
    embeddings = get_indexed_product_embeddings()
    for p in products:
        values.append([embeddings[p],
                       top_clusters_for_embedding(embeddings[p], 1)[0]])
    return pandas.DataFrame(data=numpy.array(values), index=products,
                           columns=["embedding", "cluster"])

def get_personalization_vector(query=None, #2
                               user_items=[],
                               query_weight=1, #3
                               user_items_weights=[]): #3
    query_embedding = transformer.encode(query) if query else None

    if len(user_items) > 0 and len(user_items_weights) == 0: #4
        user_items_weights = numpy.full(shape=len(user_items),
                                         fill_value=1 / len(user_items))

    embeddings = []
    embedding_weights = []
    for weight in user_items_weights: #3
        embedding_weights.append(weight) #3
    for embedding in user_items:
        embeddings.append(embedding)
    if query_embedding.any():
        embedding_weights.append(query_weight) #3
        embeddings.append(query_embedding)

    return numpy.average(embeddings, weights=numpy.array(embedding_weights),
                        axis=0).astype("double") if len(embeddings) else None
```

#1 Returns a dataframe with the embedding and guardrail cluster for each product

#2 Returns a vector that combines (weighted average) an embedding for the query with the embeddings for the passed-in user\_items

#3 You can optionally specify a query\_weight and user\_item\_weights to influence how much each embedding influences the personalization vector.

#4 By default, the weight is split 1:1 (50% each) between the query embedding and the user\_items\_weight.

With the ability to combine embeddings and to look up the guardrail cluster for any product, it's time to generate a personalization vector for a user based on their incoming query and past product interactions. We'll generate a personalization vector with guardrails as well as one without guardrails to compare the results side by side.

Listing 9.17 demonstrates how to generate personalization vectors. In this case, the user has previously interacted with two products: a Hello Kitty water bottle and a stainless-steel electric range. They are now running a new query for the keyword `microwave`.

### Listing 9.17 Generating personalization vectors from user queries

```
product_interests = ["7610465823828", #hello kitty water bottle
                     "36725569478"]   #stainless steel electric range

user_embeddings = get_user_embeddings(product_interests)
query = "microwave"

unfiltered_personalization_vector = #1
    get_personalization_vector(query=query, #1
    user_items=user_embeddings['embedding'].to_numpy()) #1
print("\nPersonalization Vector (No Cluster Guardrails):")
print(format_vector(unfiltered_personalization_vector))

query_clusters = get_top_labels_centers(query, #2
                                       centers, n=5) #2
print("\nQuery Clusters ('microwave'):\n" + str(query_clusters))

clustered = user_embeddings.cluster.isin(query_clusters) #3
products_in_cluster = user_embeddings[clustered] #3
print("\nProducts Filtered to Query Clusters:\n" + str(products_in_cluster))

filtered_personalization_vector = get_personalization_vector(query=query, #4
    user_items=filtered['embedding'].to_numpy()) #4
print("\nFiltered Personalization Vector (With Cluster Guardrails):")
print(format_vector(filtered_personalization_vector))
```

#1 Personalization vector with no guardrails (uses query and all past item interactions)

#2 Gets the top 5 clusters for the query to use as guardrails

#3 Filters down to only items in the guardrail query clusters

#4 Generates a personalization vector with guardrails (uses query and only items related to the query)

Output:

```
Products Interactions for Personalization:
product      embedding                                     cluster
7610465823828 [0.06417941, 0.04178553, -0.0017139615, -0.020...  1
36725569478   [0.0055417763, -0.024302201, -0.024139373, -0....  6

Personalization Vector (No Cluster Guardrails):
[0.016, -0.006, -0.02, -0.032, -0.016, 0.008, -0.0, 0.017, 0.011, 0.007 ...]

Query Clusters ('microwave'):
[44, 52, 5, 83, 6]

Products Filtered to Query Clusters:
product      embedding                                     cluster
36725569478   [0.0055417763, -0.024302201, -0.024139373, -0....  6

Filtered Personalization Vector (With Cluster Guardrails):
[0.002, -0.023, -0.026, -0.037, -0.025, 0.002, -0.009, 0.007, 0.033, -0 ...]
```

Listing 9.17 performs a four-step process for generating a personalization vector for a user's query:

1. Get the list of product interactions, along with the associated product embeddings and clusters.
2. Find the  $N$  (5 in this case) most similar clusters for the query.

3. Filter the list of user interactions down to only items in the query clusters.
4. Generate the personalization vector ( `filtered_personalization_vector` ) by combining the query and filtered user-item interaction vectors. (Note: we also generated an `unfiltered_personalization_vector` that does not apply categorical guardrails, for later side-by-side comparison.)

The final `filtered_personalization_vector` could be used directly for a vector search across embeddings, as it represents an embedding for the query that has been pulled toward the user's interests in the 768-dimension embedding vector space. In our case, we are going to run an independent search for the query instead, and then use the `filtered_personalization_vector` to rerank the top results. The following listing demonstrates this search and reranking process.

#### Listing 9.18 Using the personalization vector to rerank results

```
def rerank_with_personalization(docs, #1
                               personalization_vector): #1
    embeddings = get_indexed_product_embeddings()
    result_embeddings = numpy.array(
        [embeddings[docs[x]["upc"]]
         for x in range(len(docs))]).astype(float)
    similarities = sentence_transformers.util.cos_sim(
        personalization_vector, result_embeddings).tolist()[0]
    reranked = [similarities.index(i)
                 for i in heapq.nlargest(len(similarities), similarities)]
    reranked, _ = zip(*sorted(enumerate(similarities),
                              key=itemgetter(1), reverse=True))
    return [docs[i] for i in reranked]

query = "microwave"
request = product_search_request(query, {"limit": 100})

response = products_collection.search(**request)
docs = response["docs"]
print("No Personalization:")
display_product_search(query, docs[0:4]) #2

print("Global Personalization (No Category Guardrails):")
reranked_search_results_no_guardrails = \ #3
    rerank_with_personalization(docs, #3
                               unfiltered_personalization_vector) #3
display_product_search(query, reranked_search_results_no_guardrails[0:4])

print("Contextual Personalization (with Category Guardrails):")
reranked_search_results_with_guardrails = \ #4
    rerank_with_personalization(docs, #4
                               filtered_personalization_vector) #4
display_product_search(query, reranked_search_results_with_guardrails[0:4])
```

#1 Reranks all search results based upon cosine similarity to the personalized query vector

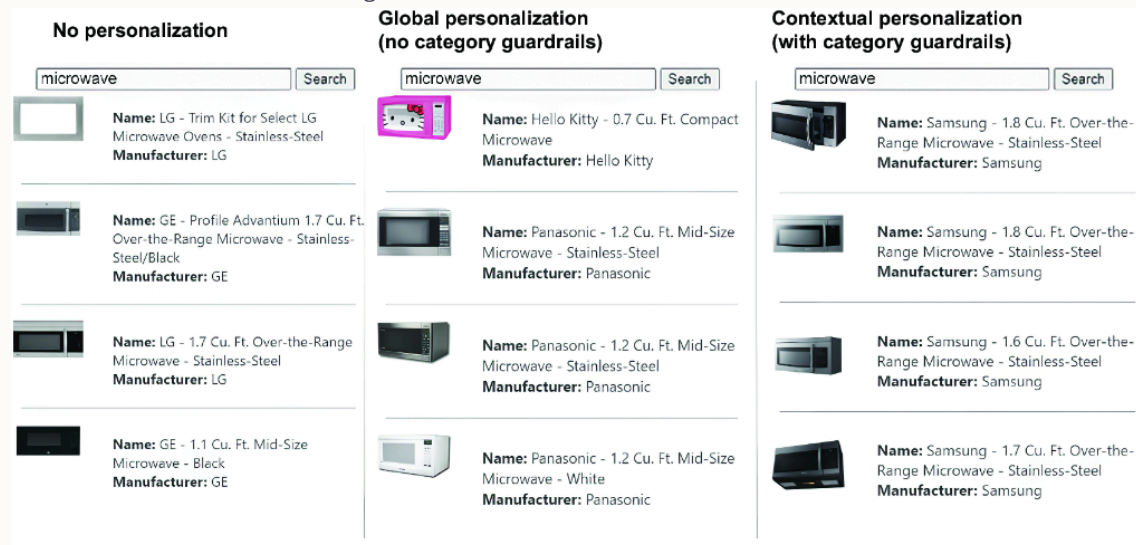
#2 Displays the original search results (no personalization)

#3 Personalized search with no guardrails (uses `unfiltered_personalization_vector`)

#4 Personalized search with guardrails (uses `filtered_personalization_vector`)

Listing 9.18 walks through the entire process of applying personalization vectors to rerank the search results. The `rerank_with_personalization` function takes the original search results and a personalization vector and then reranks the search results based on the cosine similarity between the personalization vector and the embedding vectors for each search result.

We invoke reranking twice for comparison purposes: once with and once without guardrails applied to the personalization vector. The final sets of ranked results are each passed to the `display_product_search` function to render the three result sets compared in figure 9.16: the non-personalized search results, personalized search results with no guardrails, and personalized search results with guardrails.



**Figure 9.16 Comparing non-personalized, always personalized (no guardrails), and contextually personalized (with guardrails) search results**

On the left, we see the original search results for `microwave`, including a microwave cover, some stainless-steel microwaves, and a basic microwave. In the middle, we see personalized search results with no categorical guardrails. The user's personalization vector includes embeddings for a stainless-steel microwave, as well as a Hello Kitty water bottle. As you can see, the Hello Kitty microwave jumped straight to the top of the results, even though the user has previously looked at a stainless-steel refrigerator, and their interest in a water bottle is unlikely to translate into an interest in a Hello Kitty microwave. On the right, we see personalization with guardrails applied. We see that all of these results are now for stainless-steel microwaves, reflecting the user's previous interest in a stainless-steel refrigerator, which was automatically identified as a similar category.

You have now implemented an end-to-end personalized search algorithm. Personalized search can significantly improve relevance when implemented carefully and with a light touch, but it is important to not frustrate your users by over-personalizing. In the next section, we'll review some of the pitfalls and challenges with personalization that you'll need to keep in mind to avoid potential user frustration.

## 9.5 Challenges with personalizing search results

Throughout this chapter, we've highlighted many of the challenges with personalizing search results. While personalization can be a powerful tool for driving more relevant search results, it is important to be aware of the potential pitfalls and to ensure that personalization is only applied when it is likely to be helpful to the user. We touched on the following key challenges in this chapter:

- *The cold-start problem*—When using collaborative filtering, users who have interacted with no items lack any information on which to base personalization. For such users, it's important to fall back to non-personalized search results. Combining a content-based filtering approach (search or attribute-based matching) with collaborative filtering can help overcome the cold-start problem.
- *Guardrails are important*—Applying personalization across categorical boundaries is generally a bad idea. Otherwise, as a user switches context to look at unrelated items, search re-

sults are going to look strange and be counterproductive. Looking at “white paper” or “white light bulbs” doesn’t mean a user wants to later see “white” refrigerators when searching for appliances. Similarly, liking the movie *The Terminator* doesn’t mean someone wants to purchase a gun or a robot vacuum. When personalizing search results, it is important to understand the relevant scope in which learned user preferences should be applied. Modeling related categories for items and queries and restricting personalization to only using items related to the query is a good way to avoid these problems.

- *Over-personalization is frustrating* —When someone types in a search query, they expect the search engine to return the most relevant results for their specific query. While applying personalization can be very helpful in certain use cases (e.g., location personalization in a restaurant), it can also be very frustrating if the amount of personalization interferes with the user’s control over the search experience. As an extreme case, imagine if every query were boosted by features from every previous query or item interaction; the search experience would quickly degrade into an unusable mess that would prevent the user from finding what they’re looking for. Consider only personalizing a few of the top results instead of the entire set of search results so that if the personalization is ever wrong, the non-personalized results are still available. Also, consider providing a way for users to turn off personalization if they find it frustrating.
- *Feedback loops are critical* —User interests change over time. Whether you’re showing recommendations or building personalization profiles for search, users need to be able to provide feedback to the system to help it learn and adapt to their changing interests. This can be done by allowing users to provide explicit feedback (e.g., thumbs up or thumbs down) on recommendations, or by just continuing to collect implicit feedback from behavioral signals (clicks, purchases, and so on) and using newer interactions to update the personalization profile. In either case, it is important to provide a way for users to provide feedback to the system so that it can learn and adapt to their changing interests over time.
- *Privacy can be a concern* —Because personalization is based on previous user-interaction patterns, showing personalized recommendations and search results means collecting and exposing a user’s past behavior. Imagine a movie-streaming service suggesting violent or adult-themed movies, a bookstore suggesting romance novels or self-improvement titles, or a grocery store boosting junk food and alcohol. This could both be embarrassing and demoralizing for the user, eroding both trust and confidence in the service. It is important to be transparent about what signals are being collected and how they are being used. It is also important to provide a way for users to opt out of personalization if they are concerned about their privacy.
- *Apply personalization with a light touch* —Most search engines do not personalize search results, leaving the user in full control of expressing their interests through their current query. Deviating from this paradigm can be beneficial in many cases, but it is important to ensure that personalization is only applied when it is likely to be helpful to the user. One strategy for ensuring a light touch is to apply personalization only to the top few results. It is usually better to err on the side of caution with personalization and apply it very conservatively. Most users will be less frustrated by a lack of personalization than by a search engine that tries too hard to read their minds and gets it wrong.

Out of all the techniques in AI-powered search, personalization is both one of the most underutilized ways to better understand user intent and one of the most challenging. While recommendation engines are prevalent, the personalization spectrum between search and recommendations is more nuanced and less explored. So long as personalized search is implemented with care, it can be a powerful tool to drive more relevant search results and save the user time discovering the items that best meet their particular interests.

## Summary

- Personalized search sits in the middle of the personalization spectrum between keyword search (driven by explicit user input) and collaborative recommendations (driven by implicit input derived from user behavior).
- Collaborative recommendations can be learned entirely from user-interaction patterns across documents, but they suffer from the cold-start problem. Combining collaborative filtering with content-based attributes can overcome the cold-start problem and drive more flexible personalized search experiences.
- Representing documents and users as embedding vectors enables building dynamic personalization profiles that can be used to drive better-personalized search results.
- Clustering products by their embedding vectors can be used to generate dynamic categories to serve as guardrails for personalized search, ensuring that users are not shown results that are personalized too far outside of their interests.
- Incorporating feedback loops to learn from user interactions is important, as long as user privacy is preserved and it is applied with a light touch to avoid over-personalizing.
- Personalized search can drive more relevant search results, but it's important to balance the benefits of personalization with the potential for user frustration if the personalization is too aggressive. Striking the right balance can drive significant improvements to your search engine's understanding of user intent.

Previous chapter

< [8 Signals-boosting models](#)

Next chapter

[10 Learning to rank for generalizable search relevance](#)

>