CHAPTER 23
Recommender Systems

O nature, nature, why art thou so dishonest, as ever to send men with these false recommendations into the world!

—Henry Fielding

Another common data problem is producing *recommendations* of some sort. Netflix recommends movies you might want to watch. Amazon recommends products you might want to buy. Twitter recommends users you might want to follow. In this chapter, we'll look at several ways to use data to make recommendations.

In particular, we'll look at the dataset of `users_interests` that we've used before:

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

And we'll think about the problem of recommending new interests to a user based on her currently specified interests.

Manual Curation

Before the internet, when you needed book recommendations you would go to the library, where a librarian was available to suggest books that were relevant to your interests or similar to books you liked.

Given DataSciencester's limited number of users and interests, it would be easy for you to spend an afternoon manually recommending interests for each user. But this method doesn't scale particularly well, and it's limited by your personal knowledge and imagination. (Not that I'm suggesting that your personal knowledge and imagination are limited.) So let's think about what we can do with *data*.

Recommending What's Popular

One easy approach is to simply recommend what's popular:

```
from collections import Counter

popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests)
```

which looks like:

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # ...
]
```

Having computed this, we can just suggest to a user the most popular interests that he's not already interested in:

```
from typing import List, Tuple

def most_popular_new_interests(
        user_interests: List[str],
        max_results: int = 5) -> List[Tuple[str, int]]:
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests.most_common()
                   if interest not in user_interests]
    return suggestions[:max_results]
```

So, if you are user 1, with interests:

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

then we'd recommend you:

```
[('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

If you are user 3, who's already interested in many of those things, you'd instead get:

```
[('Java', 3), ('HBase', 3), ('Big Data', 3),
 ('neural networks', 2), ('Hadoop', 2)]
```

Of course, "lots of people are interested in Python, so maybe you should be too" is not the most compelling sales pitch. If someone is brand new to our site and we don't know anything about them, that's possibly the best we can do. Let's see how we can do better by basing each user's recommendations on her existing interests.

User-Based Collaborative Filtering

One way of taking a user's interests into account is to look for users who are somehow *similar* to her, and then suggest the things that those users are interested in.

In order to do that, we'll need a way to measure how similar two users are. Here we'll use cosine similarity, which we used in Chapter 21 to measure how similar two word vectors were.

We'll apply this to vectors of 0s and 1s, each vector `v` representing one user's interests. `v[i]` will be 1 if the user specified the *i*th interest, and 0 otherwise. Accordingly, "similar users" will mean "users whose interest vectors most nearly point in the same direction." Users with identical interests will have similarity 1. Users with no identical interests will have similarity 0. Otherwise, the similarity will fall in between, with numbers closer to 1 indicating "very similar" and numbers closer to 0 indicating "not very similar."

A good place to start is collecting the known interests and (implicitly) assigning indices to them. We can do this by using a set comprehension to find the unique interests, and then sorting them into a list. The first interest in the resulting list will be interest 0, and so on:

```
unique_interests = sorted({interest
                           for user_interests in users_interests
                           for interest in user_interests})
```

This gives us a list that starts:

```
assert unique_interests[:6] == [
    'Big Data',
    'C++',
    'Cassandra',
    'HBase',
    'Hadoop',
    'Haskell',
    # ...
]
```

Next we want to produce an "interest" vector of 0s and 1s for each user. We just need to iterate over the `unique_interests` list, substituting a 1 if the user has each interest, and a 0 if not:

```
def make_user_interest_vector(user_interests: List[str]) -> List[int]:
    """
    Given a list of interests, produce a vector whose ith element is 1
    if unique_interests[i] is in the list, 0 otherwise
    """
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

And now we can make a list of user interest vectors:

```
user_interest_vectors = [make_user_interest_vector(user_interests)
                         for user_interests in users_interests]
```

Now `user_interest_vectors[i][j]` equals 1 if user `i` specified interest `j`, and 0 otherwise.

Because we have a small dataset, it's no problem to compute the pairwise similarities between all of our users:

```
from scratch.nlp import cosine_similarity

user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_vectors]
                     for interest_vector_i in user_interest_vectors]
```

after which `user_similarities[i][j]` gives us the similarity between users `i` and `j`:

```
# Users 0 and 9 share interests in Hadoop, Java, and Big Data
assert 0.56 < user_similarities[0][9] < 0.58, "several shared interests"

# Users 0 and 8 share only one interest: Big Data
assert 0.18 < user_similarities[0][8] < 0.20, "only one shared interest"
```

In particular, `user_similarities[i]` is the vector of user `i`'s similarities to every other user. We can use this to write a function that finds the most similar users to a given user. We'll make sure not to include the user herself, nor any users with zero similarity. And we'll sort the results from most similar to least similar:

```
def most_similar_users_to(user_id: int) -> List[Tuple[int, float]]:
    pairs = [(other_user_id, similarity)                      # Find other
             for other_user_id, similarity in                 # users with
                enumerate(user_similarities[user_id])         # nonzero
             if user_id != other_user_id and similarity > 0]  # similarity.

    return sorted(pairs,                                      # Sort them
                  key=lambda pair: pair[-1],                  # most similar
                  reverse=True)                               # first.
```

For instance, if we call `most_similar_users_to(0)` we get:

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

How do we use this to suggest new interests to a user? For each interest, we can just add up the user similarities of the other users interested in it:

```
from collections import defaultdict

def user_based_suggestions(user_id: int,
                           include_current_interests: bool = False):
    # Sum up the similarities
    suggestions: Dict[str, float] = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity

    # Convert them to a sorted list
    suggestions = sorted(suggestions.items(),
                         key=lambda pair: pair[-1],  # weight
                         reverse=True)

    # And (maybe) exclude already interests
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

If we call `user_based_suggestions(0)`, the first several suggested interests are:

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 #...
]
```

These seem like pretty decent suggestions for someone whose stated interests are "Big Data" and database-related. (The weights aren't intrinsically meaningful; we just use them for ordering.)

This approach doesn't work as well when the number of items gets very large. Recall the curse of dimensionality from Chapter 12—in large-dimensional vector spaces most vectors are very far apart (and also point in very different directions). That is,

when there are a large number of interests the "most similar users" to a given user might not be similar at all.

Imagine a site like Amazon.com, from which I've bought thousands of items over the last couple of decades. You could attempt to identify similar users to me based on buying patterns, but most likely in all the world there's no one whose purchase history looks even remotely like mine. Whoever my "most similar" shopper is, he's probably not similar to me at all, and his purchases would almost certainly make for lousy recommendations.

Item-Based Collaborative Filtering

An alternative approach is to compute similarities between interests directly. We can then generate suggestions for each user by aggregating interests that are similar to her current interests.

To start with, we'll want to *transpose* our user-interest matrix so that rows correspond to interests and columns correspond to users:

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_vectors]
                        for j, _ in enumerate(unique_interests)]
```

What does this look like? Row `j` of `interest_user_matrix` is column `j` of `user_interest_matrix`. That is, it has 1 for each user with that interest and 0 for each user without that interest.

For example, `unique_interests[0]` is Big Data, and so `interest_user_matrix[0]` is:

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

because users 0, 8, and 9 indicated interest in Big Data.

We can now use cosine similarity again. If precisely the same users are interested in two topics, their similarity will be 1. If no two users are interested in both topics, their similarity will be 0:

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                          for user_vector_j in interest_user_matrix]
                         for user_vector_i in interest_user_matrix]
```

For example, we can find the interests most similar to Big Data (interest 0) using:

```
def most_similar_interests_to(interest_id: int):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
             for other_interest_id, similarity in enumerate(similarities)
             if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
```

```
                    key=lambda pair: pair[-1],
                    reverse=True)
```

which suggests the following similar interests:

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

Now we can create recommendations for a user by summing up the similarities of the interests similar to his:

```
def item_based_suggestions(user_id: int,
                           include_current_interests: bool = False):
    # Add up the similar interests
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_vectors[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity

    # Sort them by weight
    suggestions = sorted(suggestions.items(),
                         key=lambda pair: pair[-1],
                         reverse=True)

    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

For user 0, this generates the following (seemingly reasonable) recommendations:

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
```

```
    ('C++', 0.4082482904638631),
    ('artificial intelligence', 0.4082482904638631),
    ('Python', 0.2886751345948129),
    ('R', 0.2886751345948129)]
```

Matrix Factorization

As we've seen, we can represent our users' preferences as a `[num_users, num_items]` matrix of 0s and 1s, where the 1s represent liked items and the 0s unliked items.

Sometimes you might actually have numeric *ratings*; for example, when you write an Amazon review you assign the item a score ranging from 1 to 5 stars. You could still represent these by numbers in a `[num_users, num_items]` matrix (ignoring for now the problem of what to do about unrated items).

In this section we'll assume we have such ratings data and try to learn a model that can predict the rating for a given user and item.

One way of approaching the problem is to assume that every user has some latent "type," which can be represented as a vector of numbers, and that each item similarly has some latent "type."

If the user types are represented as a `[num_users, dim]` matrix, and the transpose of the item types is represented as a `[dim, num_items]` matrix, their product is a `[num_users, num_items]` matrix. Accordingly, one way of building such a model is by "factoring" the preferences matrix into the product of a user matrix and an item matrix.

(Possibly this idea of latent types reminds you of the word embeddings we developed in Chapter 21. Hold on to that idea.)

Rather than working with our made-up 10-user dataset, we'll work with the MovieLens 100k dataset, which contains ratings from 0 to 5 for many movies from many users. Each user has only rated a small subset of the movies. We'll use this to try to build a system that can predict the rating for any given (user, movie) pair. We'll train it to predict well on the movies each user has rated; hopefully then it will generalize to movies the user hasn't rated.

To start with, let's acquire the dataset. You can download it from *http://files.grouplens.org/datasets/movielens/ml-100k.zip*.

Unzip it and extract the files; we'll only use two of them:

```
# This points to the current directory, modify if your files are elsewhere.
MOVIES = "u.item"   # pipe-delimited: movie_id|title|...
RATINGS = "u.data"  # tab-delimited: user_id, movie_id, rating, timestamp
```

As is often the case, we'll introduce a `NamedTuple` to make things easier to work with:

```
from typing import NamedTuple

class Rating(NamedTuple):
    user_id: str
    movie_id: str
    rating: float
```

The movie ID and user IDs are actually integers, but they're not consecutive, which means if we worked with them as integers we'd end up with a lot of wasted dimensions (unless we renumbered everything). So to keep it simpler we'll just treat them as strings.

Now let's read in the data and explore it. The movies file is pipe-delimited and has many columns. We only care about the first two, which are the ID and the title:

```
import csv
# We specify this encoding to avoid a UnicodeDecodeError.
# See: https://stackoverflow.com/a/53136168/1076346.
with open(MOVIES, encoding="iso-8859-1") as f:
    reader = csv.reader(f, delimiter="|")
    movies = {movie_id: title for movie_id, title, *_ in reader}
```

The ratings file is tab-delimited and contains four columns for `user_id`, `movie_id`, `rating` (1 to 5), and `timestamp`. We'll ignore the timestamp, as we don't need it:

```
# Create a list of [Rating]
with open(RATINGS, encoding="iso-8859-1") as f:
    reader = csv.reader(f, delimiter="\t")
    ratings = [Rating(user_id, movie_id, float(rating))
               for user_id, movie_id, rating, _ in reader]

# 1682 movies rated by 943 users
assert len(movies) == 1682
assert len(list({rating.user_id for rating in ratings})) == 943
```

There's a lot of interesting exploratory analysis you can do on this data; for instance, you might be interested in the average ratings for *Star Wars* movies (the dataset is from 1998, which means it predates *The Phantom Menace* by a year):

```
import re

# Data structure for accumulating ratings by movie_id
star_wars_ratings = {movie_id: []
                     for movie_id, title in movies.items()
                     if re.search("Star Wars|Empire Strikes|Jedi", title)}

# Iterate over ratings, accumulating the Star Wars ones
for rating in ratings:
    if rating.movie_id in star_wars_ratings:
        star_wars_ratings[rating.movie_id].append(rating.rating)
```

```
# Compute the average rating for each movie
avg_ratings = [(sum(title_ratings) / len(title_ratings), movie_id)
               for movie_id, title_ratings in star_wars_ratings.items()]

# And then print them in order
for avg_rating, movie_id in sorted(avg_ratings, reverse=True):
    print(f"{avg_rating:.2f} {movies[movie_id]}")
```

They're all pretty highly rated:

```
4.36 Star Wars (1977)
4.20 Empire Strikes Back, The (1980)
4.01 Return of the Jedi (1983)
```

So let's try to come up with a model to predict these ratings. As a first step, let's split the ratings data into train, validation, and test sets:

```
import random
random.seed(0)
random.shuffle(ratings)

split1 = int(len(ratings) * 0.7)
split2 = int(len(ratings) * 0.85)

train = ratings[:split1]              # 70% of the data
validation = ratings[split1:split2]   # 15% of the data
test = ratings[split2:]               # 15% of the data
```

It's always good to have a simple baseline model and make sure that ours does better than that. Here a simple baseline model might be "predict the average rating." We'll be using mean squared error as our metric, so let's see how the baseline does on our test set:

```
avg_rating = sum(rating.rating for rating in train) / len(train)
baseline_error = sum((rating.rating - avg_rating) ** 2
                     for rating in test) / len(test)

# This is what we hope to do better than
assert 1.26 < baseline_error < 1.27
```

Given our embeddings, the predicted ratings are given by the matrix product of the user embeddings and the movie embeddings. For a given user and movie, that value is just the dot product of the corresponding embeddings.

So let's start by creating the embeddings. We'll represent them as `dicts` where the keys are IDs and the values are vectors, which will allow us to easily retrieve the embedding for a given ID:

```
from scratch.deep_learning import random_tensor

EMBEDDING_DIM = 2

# Find unique ids
```

```
user_ids = {rating.user_id for rating in ratings}
movie_ids = {rating.movie_id for rating in ratings}

# Then create a random vector per id
user_vectors = {user_id: random_tensor(EMBEDDING_DIM)
                for user_id in user_ids}
movie_vectors = {movie_id: random_tensor(EMBEDDING_DIM)
                 for movie_id in movie_ids}
```

By now we should be pretty expert at writing training loops:

```
from typing import List
import tqdm
from scratch.linear_algebra import dot

def loop(dataset: List[Rating],
         learning_rate: float = None) -> None:
    with tqdm.tqdm(dataset) as t:
        loss = 0.0
        for i, rating in enumerate(t):
            movie_vector = movie_vectors[rating.movie_id]
            user_vector = user_vectors[rating.user_id]
            predicted = dot(user_vector, movie_vector)
            error = predicted - rating.rating
            loss += error ** 2

            if learning_rate is not None:
                #     predicted = m_0 * u_0 + ... + m_k * u_k
                # So each u_j enters output with coefficent m_j
                # and each m_j enters output with coefficient u_j
                user_gradient = [error * m_j for m_j in movie_vector]
                movie_gradient = [error * u_j for u_j in user_vector]

                # Take gradient steps
                for j in range(EMBEDDING_DIM):
                    user_vector[j] -= learning_rate * user_gradient[j]
                    movie_vector[j] -= learning_rate * movie_gradient[j]

            t.set_description(f"avg loss: {loss / (i + 1)}")
```

And now we can train our model (that is, find the optimal embeddings). For me it worked best if I decreased the learning rate a little each epoch:

```
learning_rate = 0.05
for epoch in range(20):
    learning_rate *= 0.9
    print(epoch, learning_rate)
    loop(train, learning_rate=learning_rate)
    loop(validation)
loop(test)
```

This model is pretty apt to overfit the training set. I got the best results with `EMBEDDING_DIM=2`, which got me an average loss on the test set of about 0.89.

If you wanted higher-dimensional embeddings, you could try regularization like we used in "Regularization" on page 194. In particular, at each gradient update you could shrink the weights toward 0. I was not able to get any better results that way.

Now, inspect the learned vectors. There's no reason to expect the two components to be particularly meaningful, so we'll use principal component analysis:

```
from scratch.working_with_data import pca, transform

original_vectors = [vector for vector in movie_vectors.values()]
components = pca(original_vectors, 2)
```

Let's transform our vectors to represent the principal components and join in the movie IDs and average ratings:

```
ratings_by_movie = defaultdict(list)
for rating in ratings:
    ratings_by_movie[rating.movie_id].append(rating.rating)

vectors = [
    (movie_id,
     sum(ratings_by_movie[movie_id]) / len(ratings_by_movie[movie_id]),
     movies[movie_id],
     vector)
    for movie_id, vector in zip(movie_vectors.keys(),
                                transform(original_vectors, components))
]

# Print top 25 and bottom 25 by first principal component
print(sorted(vectors, key=lambda v: v[-1][0])[:25])
print(sorted(vectors, key=lambda v: v[-1][0])[-25:])
```

The top 25 are all highly rated, while the bottom 25 are mostly low-rated (or unrated in the training data), which suggests that the first principal component is mostly capturing "how good is this movie?"

It's hard for me to make much sense of the second component; and, indeed the two-dimensional embeddings performed only slightly better than the one-dimensional embeddings, suggesting that whatever the second component captured is possibly very subtle. (Presumably one of the larger MovieLens datasets would have more interesting things going on.)

For Further Exploration

- Surprise is a Python library for "building and analyzing recommender systems" that seems reasonably popular and up-to-date.
- The Netflix Prize was a somewhat famous competition to build a better system to recommend movies to Netflix users.