Photo credit: **Slash Gear**

Nick Becker

Music Recommendations with Collaborative Filtering and Cosine Distance

☐ 13 minute read

I’ve been using a lot of products with recommendation engines lately, so I decided it would be cool to build one myself. Recommender

Data Scientist at
Enigma Technologies

☐ New York, NY

☐ LinkedIn

☐ Github

systems can be loosely broken down into three categories: **content based systems**, **collaborative filtering systems**, and **hybrid systems** (which use a combination of the other two).

Content based recommender systems use the features of items to recommend other similar items. For example, if I'm browsing for solid colored t-shirts on Amazon, a content based recommender might recommend me other t-shirts or solid colored sweatshirts because they have similar features (sleeves, single color, shirt, etc.).

Collaborative filtering based systems use the actions of users to recommend other items. In general, they can either be user based or item based. User based collaborating filtering uses the patterns of users similar to me to recommend a product (users like me also looked at *these other* items). Item based collaborative filtering uses the patterns of users who browsed the same item as me to recommend me a product (users who looked at my item also looked at *these other* items).

For this post, I'm going to build an item based collaborative filtering system. I'll leave the user based collaborative filtering recommender for another post.

Finding a Dataset for Recommendations

While googling around for a good dataset, I stumbled upon a page from 2011 with a bunch of cool datasets. Since I use Spotify and Pandora all the time, I figured I'd choose a music dataset.

The Last.fm data are from the Music Technology Group at the Universitat Pompeu Fabra in Barcelona, Spain. The data were scraped by Òscar Celma using the Last.fm API, and they are available free of charge for non-commercial use. So, thank you Òscar!

The Last.fm data are broken into two parts: the activity data and the profile data. The activity data comprises about 360,000 individual users's Last.fm artist listening information. It details how many times a Last.fm user played songs by various artists. The profile data contains each user's country of residence. We'll use `read.table` from `pandas` to read in the tab-delimited files.

```
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix

# display results to 3 decimal points, not in scientific notation
```

```
pd.set_option('display.float_format', lambda x: '%.3f' % x)
```

```
user_data = pd.read_table('/users/nickbecker/Downloads/lastfm-dataset-360K/u
                          header = None, nrows = 2e7,
                          names = ['users', 'musicbrainz-artist-id', 'artist
                          usecols = ['users', 'artist-name', 'plays'])
user_profiles = pd.read_table('/users/nickbecker/Downloads/lastfm-dataset-36
                          header = None,
                          names = ['users', 'gender', 'age', 'country', 'sig
                          usecols = ['users', 'country'])
```

Let’s take a quick look at the two datasets.

```
user_data.head()
```

	users	artist-name	plays
0	00000c289a1829a808ac09c00daf10bc3c4e223b	betty blowtorch	2137
1	00000c289a1829a808ac09c00daf10bc3c4e223b	die Ärzte	1099
2	00000c289a1829a808ac09c00daf10bc3c4e223b	melissa etheridge	897
3	00000c289a1829a808ac09c00daf10bc3c4e223b	elvenking	717

	users	artist-name	plays
4	00000c289a1829a808ac09c00daf10bc3c4e223b	juliette & the licks	706

```
user_profiles.head()
```

	users	country
0	00000c289a1829a808ac09c00daf10bc3c4e223b	Germany
1	00001411dc427966b17297bf4d69e7e193135d89	Canada
2	00004d2ac9316e22dc007ab2243d6fcb239e707d	Germany
3	000063d3fe1cf2ba248b9e3c3f0334845a27a6bf	Mexico
4	00007a47085b9aab8af55f52ec8846ac479ac4fe	United States

With the datasets loaded in memory, we can start doing some data work and eventually make recommendations.

Filtering to Only Popular Artists

Since we’re going to be doing item-based collaborative filtering, our recommendations will be based on user patterns in listening to artists. Lesser known artists will have views from fewer viewers, making the pattern more noisy. This would probably result in bad recommendations (or at least ones highly sensitive to an individual person who ***loves*** one obscure artist. To avoid this problem, we'll just look at the popular artists.

To find out which artists are popular, we need to know the total play count of every artist. Since our user play count data has one row per artist per user, we need to aggregate it up to the artist level. With `pandas`, we can group by the artist name and then calculate the sum of the `plays` column for every artist. If the `artist-name` variable is missing, our future reshaping and analysis won't work. So I'll start by removing rows where the artist name is missing just to be safe.

```
if user_data['artist-name'].isnull().sum() > 0:
    user_data = user_data.dropna(axis = 0, subset = ['artist-name'])
```

```
artist_plays = (user_data.
     groupby(by = ['artist-name'])['plays'].
     sum().
     reset_index().
     rename(columns = {'plays': 'total_artist_plays'})
     [['artist-name', 'total_artist_plays']]
    )
```

```
artist_plays.head()
```

	artist-name	total_artist_plays
0	04)]	6
1	2	1606
2	58725ab=>	23
3	80lİ yillarin tÜrkÇe sÖzlÜ aŞk Şarkilari	70
4	amy winehouse	23

Now we can merge the total play count data into the user activity data, giving us exactly what we need to filter out the lesser known artists.

```
user_data_with_artist_plays = user_data.merge(artist_plays, left_on = 'artis
user_data_with_artist_plays.head()
```

	users	artist-name	plays	total_artist_plays
0	00000c289a1829a808ac09c00daf10bc3c4e223b	betty blowtorch	2137	25651
1	00000c289a1829a808ac09c00daf10bc3c4e223b	die Ärzte	1099	3704875

	users	artist-name	plays	total_artist_plays
2	00000c289a1829a808ac09c00daf10bc3c4e223b	melissa etheridge	897	180391
3	00000c289a1829a808ac09c00daf10bc3c4e223b	elvenking	717	410725
4	00000c289a1829a808ac09c00daf10bc3c4e223b	juliette & the licks	706	90498

Picking a threshold for popular artists

With nearly 300,000 different artists, it's almost a guarantee most artists have been played only a few times. Let's look at the descriptive statistics.

```
print artist_plays['total_artist_plays'].describe()
```

```
count     292364.000
mean       12907.037
std       185981.313
min            1.000
25%           53.000
50%          208.000
75%         1048.000
```

```
max     30466827.000
Name: total_artist_plays, dtype: float64
```

As expected, the median artist has only been played about 200 times. Let’s take a look at the top of the distribution.

```
print artist_plays['total_artist_plays'].quantile(np.arange(.9, 1, .01)),
```

```
0.900     6138.000
0.910     7410.000
0.920     9102.960
0.930    11475.590
0.940    14898.440
0.950    19964.250
0.960    28419.880
0.970    43541.330
0.980    79403.440
0.990   198482.590
Name: total_artist_plays, dtype: float64
```

So about 1% of artists have roughly 200,000 or more plays, 2% have 80,000 or more, and 3% have 40,000 or more. Since we have so many artists, we’ll limit it to the top 3%. This is an arbitrary threshold for popularity, but it gives us about 9000 different artists. It seems to me that including more than that might push us into the problem of noisy listening patterns we're trying to avoid.

```
popularity_threshold = 40000
user_data_popular_artists = user_data_with_artist_plays.query('total_artist_
user_data_popular_artists.head()
```

	users	artist-name	plays	total_artist_plays
1	00000c289a1829a808ac09c00daf10bc3c4e223b	die Ärzte	1099	3704875
2	00000c289a1829a808ac09c00daf10bc3c4e223b	melissa etheridge	897	180391
3	00000c289a1829a808ac09c00daf10bc3c4e223b	elvenking	717	410725
4	00000c289a1829a808ac09c00daf10bc3c4e223b	juliette & the licks	706	90498
5	00000c289a1829a808ac09c00daf10bc3c4e223b	red hot chili peppers	691	13547741

Filtering to US Users Only

Since I'm in Washington, D.C., I'll limit the user data to just those from the United States. This will reduce the number of users and artists considerably, improving the computational efficiency significantly.

First, I'll merge in the user profile data that has the user's country. Then I'll filter the data to only users in the United States.

```
combined = user_data_popular_artists.merge(user_profiles, left_on = 'users',
usa_data = combined.query('country == \'United States\'')
usa_data.head()
```

	users	artist-name	plays	total_artist_plays	country
156	00007a47085b9aab8af55f52ec8846ac479ac4fe	devendra banhart	456	2366807	United States
157	00007a47085b9aab8af55f52ec8846ac479ac4fe	boards of canada	407	6115545	United States
158	00007a47085b9aab8af55f52ec8846ac479ac4fe	cocorosie	386	2194862	United States
159	00007a47085b9aab8af55f52ec8846ac479ac4fe	aphex twin	213	4248296	United States
160	00007a47085b9aab8af55f52ec8846ac479ac4fe	animal collective	203	3495537	United States

Before doing any analysis, we should make sure the dataset is internally consistent. Every user should only have a play count variable once for each artist. So we'll check for instances where rows have the same `users` and `artist-name` values.

```
if not usa_data[usa_data.duplicated(['users', 'artist-name'])].empty:
    initial_rows = usa_data.shape[0]

    print 'Initial dataframe shape {0}'.format(usa_data.shape)
    usa_data = usa_data.drop_duplicates(['users', 'artist-name'])
    current_rows = usa_data.shape[0]
    print 'New dataframe shape {0}'.format(usa_data.shape)
    print 'Removed {0} rows'.format(initial_rows - current_rows)
```

```
Initial dataframe shape (2788019, 5)
New dataframe shape (2788013, 5)
Removed 6 rows
```

Implemeting the Nearest Neighbor Model

Reshaping the Data

For K-Nearest Neighbors, we want the data to be in an `m x n` array, where `m` is the number of artists and `n` is the number of users. To reshape the dataframe, we'll `pivot` the dataframe to the wide format with artists as rows and users as columns. Then we'll fill the missing observations with `0`s since we're going to be performing linear algebra operations (calculating distances between vectors). Finally, we transform the values of the dataframe into a scipy sparse matrix for more efficient calculations.

```
wide_artist_data = usa_data.pivot(index = 'artist-name', columns = 'users',
wide_artist_data_sparse = csr_matrix(wide_artist_data.values)
```

Fitting the Model

Time to implement the model. We'll initialize the `NearestNeighbors` class as `model_knn` and `fit` our sparse matrix to the instance. By specifying the `metric = cosine`, the model will measure similarity bectween artist vectors by using cosine similarity.

```
from sklearn.neighbors import NearestNeighbors
```

```
model_knn = NearestNeighbors(metric = 'cosine', algorithm = 'brute')
model_knn.fit(wide_artist_data_sparse)
```

Making Recommendations

And we're finally ready to make some recommendations!

```
query_index = np.random.choice(wide_data.shape[0])
distances, indices = model_knn.kneighbors(wide_artist_data.iloc[query_index,

for i in range(0, len(distances.flatten())):
    if i == 0:
        print 'Recommendations for {0}:\n'.format(wide_artist_data.index[que
    else:
        print '{0}: {1}, with distance of {2}:'.format(i, wide_artist_data.i
```

```
Recommendations for tony bennett:

1: frank sinatra, with distance of 0.226917809755:
2: keiko matsui, with distance of 0.39397227453:
3: andy williams, with distance of 0.477163884119:
4: chic, with distance of 0.488077997533:
5: cherry poppin daddies, with distance of 0.4908909547:
```

Pretty good! Frank Sinatra and Andy Williams are obviously good recommendations for Tony Bennett. I'd never heard of Keiko Matsui or Cherry Poppin Daddies, but they both seem like good recommendations after listening to their music. Chic, though, doesn't seem as similar to me as the other artists do (Chic sounds a little more disco-y).

Why would our model recommend Chic? Since we're doing item-based collaborative filtering with K-Nearest Neighbors on the actual play count data, outliers can have a big influence. If a few users listened to Tony Bennett and Chic a ***ton***, our distance metric between vectors will be heavily influenced by those individual observations.

So is this good? Maybe. Depending on our goal, we might want **super-fans** to have disproportionate weight in the distance calculation. But could we represent the data differently to avoid this feature?

Binary Play Count Data

Previously, we used the actual play counts as values in our artist vectors. Another approach would be convert each vector into a binary (1 or 0): either a user played the song or they did not. We can do this by applying the `sign` function in `numpy` to each column in the dataframe.

```
wide_artist_data_zero_one = wide_artist_data.apply(np.sign)
wide_artist_data_zero_one_sparse = csr_matrix(wide_artist_data_zero_one.valu

save_sparse_csr('/users/nickbecker/Python_Projects/lastfm_sparse_artist_matr
```

```
model_nn_binary = NearestNeighbors(metric='cosine', algorithm='brute')
model_nn_binary.fit(wide_artist_data_zero_one_sparse)
```

Let’s make a quick comparison. Which recommendations for Tony Bennett look better?

```
distances, indices = model_nn_binary.kneighbors(wide_artist_data_zero_one.il

for i in range(0, len(distances.flatten())):
    if i == 0:
        print 'Recommendations with binary play data for {0}:\n'.format(wide
    else:
        print '{0}: {1}, with distance of {2}:'.format(i, wide_artist_data_z
Recommendations with binary play data for tony bennett:


1: nat king cole, with distance of 0.771590841231:
2: dean martin, with distance of 0.791135426625:
3: frank sinatra, with distance of 0.815388695965:
4: bobby darin, with distance of 0.818033228367:
5: doris day, with distance of 0.81859043384:
```

These are great, too. At least for Tony Bennett, the binary data representation recommendations look just as good if not better. Someone who likes Tony Bennett might also like Nat King Cole or Frank Sinatra. The distances are higher, but that’s due to squashing the data by using the sign function.

Again, it’s not obvious which method is better. Since ultimately it’s the users’s future actions that indicate which recommender system is better, it’s a perfect candidate for A/B Testing. For now, I’ll stick with the binary data representation model.

Recommending Artists on Command with Fuzzy Matching

Previously we picked query artists at random. But really, we want to make recommendations for a specific artist on command. Since some artists’s names are ambiguous or commonly mispelled, we’ll include fuzzy matching part in the process so we don’t need exact name matches.

So we can do this anytime we want, we’ll define a function `print_artist_recommendations` to do it.

```
from fuzzywuzzy import fuzz

def print_artist_recommendations(query_artist, artist_plays_matrix, knn_mode
    """
    Inputs:
    query_artist: query artist name
    artist_plays_matrix: artist play count dataframe (not the sparse one, th
    knn_model: our previously fitted sklearn knn model
    k: the number of nearest neighbors.

    Prints: Artist recommendations for the query artist
    Returns: None
    """
    query_index = None
    ratio_tuples = []

    for i in artist_plays_matrix.index:
        ratio = fuzz.ratio(i.lower(), query_artist.lower())
        if ratio >= 75:
            current_query_index = artist_plays_matrix.index.tolist().index(i
            ratio_tuples.append((i, ratio, current_query_index))

    print 'Possible matches: {0}\n'.format([(x[0], x[1]) for x in ratio_tupl

    try:
        query_index = max(ratio_tuples, key = lambda x: x[1])[2] # get the i
    except:
        print 'Your artist didn\'t match any artists in the data. Try again'
        return None

    distances, indices = knn_model.kneighbors(artist_plays_matrix.iloc[query

    for i in range(0, len(distances.flatten())):
        if i == 0:
            print 'Recommendations for {0}:\n'.format(artist_plays_matrix.in
        else:
            print '{0}: {1}, with distance of {2}:'.format(i, artist_plays_m

    return None
```

Time to try a few sample bands and get some recommendations for music I might like.

```
print_artist_recommendations('red hot chili peppers', wide_artist_data_zero_
```

```
Possible matches: [('red hot chili peppers', 100)]

Recommendations for red hot chili peppers:

1: incubus, with distance of 0.686632912166:
2: the beatles, with distance of 0.693856742888:
3: sublime, with distance of 0.70540037526:
4: foo fighters, with distance of 0.71155686859:
5: coldplay, with distance of 0.716691422348:
6: led zeppelin, with distance of 0.722488787624:
7: nirvana, with distance of 0.724943983169:
8: green day, with distance of 0.734603813118:
9: radiohead, with distance of 0.737372302802:
10: rage against the machine, with distance of 0.740136491957:
```

```
print_artist_recommendations('arctic monkeys', wide_artist_data_zero_one, mo
```

```
Possible matches: [('arctic monkeys', 100)]

Recommendations for arctic monkeys:
```

```
1: the strokes, with distance of 0.746696592481:
2: the kooks, with distance of 0.767492571954:
3: bloc party, with distance of 0.772120302741:
4: franz ferdinand, with distance of 0.774566073856:
5: the killers, with distance of 0.807176759929:
6: radiohead, with distance of 0.812762633074:
7: the fratellis, with distance of 0.814611330462:
8: kings of leon, with distance of 0.815408152181:
9: the beatles, with distance of 0.815680085574:
10: the white stripes, with distance of 0.81607343278:
```

```
print_artist_recommendations('u2', wide_artist_data_zero_one, model_nn_binar
```

```
Possible matches: [('u2', 100)]

Recommendations for u2:

1: r.e.m., with distance of 0.690057376797:
2: coldplay, with distance of 0.697303504846:
3: the beatles, with distance of 0.726085401263:
4: the police, with distance of 0.756131589948:
5: radiohead, with distance of 0.77692434746:
6: pearl jam, with distance of 0.778566201394:
```

```
7: the rolling stones, with distance of 0.782771546531:
8: led zeppelin, with distance of 0.788269370325:
9: dave matthews band, with distance of 0.788520439902:
10: bruce springsteen, with distance of 0.789619233997:
```

```
print_artist_recommendations('dispatch', wide_artist_data_zero_one, model_nn
```

```
Possible matches: [('dispatch', 100)]

Recommendations for dispatch:

1: state radio, with distance of 0.665125909027:
2: o.a.r., with distance of 0.749067403207:
3: jack johnson, with distance of 0.778821492779:
4: dave matthews band, with distance of 0.792342999987:
5: guster, with distance of 0.821963512261:
6: ben harper, with distance of 0.824748164332:
7: slightly stoopid, with distance of 0.837663503717:
8: the john butler trio, with distance of 0.841162765581:
9: donavon frankenreiter, with distance of 0.841397926422:
10: sublime, with distance of 0.846795058358:
```

To be brief, these are fantastic.

Future Ideas

Scaling up to Massive Datasets

Since we're calling `model_nn_binary` every query, we're calculating the distance of each artist vector in our `wide_artist_data_sparse` array to the query artist vector every time we want recommendations. If our dataset is fairly small (like in this post), this isn't an issue. If we had the entirety of Last.fm's user data, we'd be bottlenecked like crazy at query time.

Because we're doing item based collaborative filtering, we can actually avoid this issue. The item vectors change, of course, as users listen to more artists. But, in general, they are pretty static. If we pre-computed an item-item similarity matrix (in our case, every cell would be the cosine-distance between artist `i` and artist `j`), we could just look up the similarity values at query time. This is way faster, and scales extremely well to massive datasets.

If we were doing user based collaboartive filtering, we'd probably need to do more frequent computations since user activity fluctuates so much. If we wanted real-time recommendations based on user behavior similarities (including things like recent browsing history or recent actions), we'd be totally bottlenecked.

Fortunately, there's been great work done on Approximate Nearest Neighbor Search techniques such as locality sensitive hashing. These techniques sacrifice the **guarantee** of finding the nearest neighbors for increases in computational efficiency, and work extremely well with high dimensional data. The Machine Learning: Clustering & Retrieval course on Coursera has a great walk-through of LSH for those curious.

Recommending less popular artists

While our recommendation engine is doing a great job, it's only recommending popular artists (by design). A really cool alternative recommender might recommend us unknown artists given a query artist so we can discover new music.

Recommending lesser known artists is a huge challenge that doesn't fit as well with standard collaborative filtering, so we might want to incorporate feature based recommendations into such a system.

For those interested, the Jupyter Notebook with all the code can be found in the Github repository for this post.

☐ **Tags:** recommender systems

☐ **Updated:** August 31, 2016

Previous **Next**

LEAVE A COMMENT

We were unable to load Disqus. If you are a moderator please see our troubleshooting guide.