"People Who Like This Also Like ... " Part 2

Finding Similar Music using Matrix Factorization (/matrix-factorization/)

In a previous post I wrote about how to build a 'People Who Like This Also Like ...' feature (/distance-metrics/) for displaying lists of similar musicians. My goal was to show how simple Information Retrieval techniques can do a good job calculating lists of related artists. For instance, using BM25 distance on The Beatles shows the most similar artists being John Lennon and Paul McCartney.

One interesting technique I didn't cover was using Matrix Factorization methods to reduce the dimensionality of the data before calculating the related artists. This kind of analysis can generate matches that are impossible to find with the techniques in my original post.

This post is a step by step guide on how to calculate related artists using a couple of different matrix factorization algorithms. The code is written in Python using Pandas (http://pandas.pydata.org/) and SciPy (https://www.scipy.org/) to do the calculations and D3.js (https://d3js.org/) to interactively visualize the results.

As part of writing this post, I also open sourced a high performance python version of the Implicit Alternating Least Squares (http://github.com/benfred/implicit) matrix factorization algorithm. Most of the code here can be found in the examples directory of that project.

Loading the Data

For the post here I'm using the same Last.fm dataset (http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-360K.html) as in my first post. This can be loaded into a sparse matrix with only a couple lines of code using Pandas:

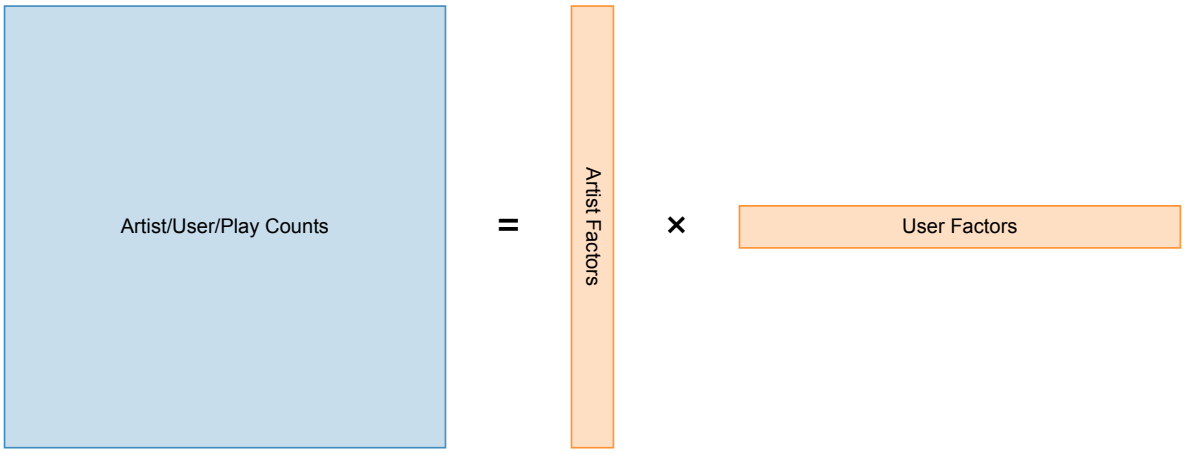
```
# read in triples of user/artist/playcount from the input dataset
data = pandas.read_table("usersha1-artmbid-artname-plays.tsv",
                         usecols=[0, 2, 3],
                         names=['user', 'artist', 'plays'])

# map each artist and user to a unique numeric value
data['user'] = data['user'].astype("category")
data['artist'] = data['artist'].astype("category")

# create a sparse matrix of all the artist/user/play triples
plays = coo_matrix((data['plays'].astype(float),
                    (data['artist'].cat.codes,
                     data['user'].cat.codes)))
```

The matrix returned here has 300,000 artists and 360,000 users with around 17 million entries total. Each entry is the number of times the user played the artist, with the data collected from the Last.fm API way back in 2008.

Matrix Factorization

One technique thats commonly used for this problem is to project the matrix of user-artist-plays in to a low rank approximation, and then compute distances in that space.

The idea is to take the original play count matrix, and then reduce that down to two much smaller matrices that approximate the original when multiplied together:



Instead of representing each artist as a sparse vector of the play counts of all 360,000 possible users, after factorizing the matrix each artist will be represented by say a 50 dimensional dense vector.

By reducing the dimensionality of the data like this, we're in effect compressing the input matrix down to two much smaller matrices. This compression forces generalization of the input data, and that generalization leads to a better understanding of the data.

Latent Semantic Analysis

One simple way to factorize the input matrix is to compute the Singular Value Decomposition (https://en.wikipedia.org/wiki/Singular_value_decomposition) on an appropriately weighted matrix (1):

```
artist_factors, _, user_factors = scipy.sparse.linalg.svds(bm25_weight(plays), 50)
```

The SVD is one of those amazingly useful techniques, and can also be used for things like Principal Component Analysis (https://en.wikipedia.org/wiki/Principal_component_analysis) or Multidimensional Scaling (http://localhost:4000/multidimensional-scaling/). I was going include a summary of how it works, but Jeremy Kun recently wrote an excellent overview of the SVD (https://jeremykun.com/2016/04/18/singular-value-decomposition-part-1-perspectives-on-linear-algebra/) , and I'm not going to even try beating that. For the purposes of this post, we just need to know that the SVD generates a low rank approximation of the input matrix, and we can use that low rank representation to generate insights.

Using the SVD like this is called Latent Semantic Analysis (https://en.wikipedia.org/wiki/Latent_semantic_analysis) (LSA). All thats really involved is getting the top most related artists by cosine distance in this factorized space, which can be done by:

```
class TopRelated(object):
    def __init__(self, artist_factors):
        # fully normalize artist_factors, so can compare with only the dot product
        norms = numpy.linalg.norm(artist_factors, axis=-1)
        self.factors = artist_factors / norms[:, numpy.newaxis]

    def get_related(self, artistid, N=10):
        scores = self.factors.dot(self.factors[artistid])
        best = numpy.argpartition(scores, -N)[-N:]
        return sorted(zip(best, scores[best]), key=lambda x: -x[1])
```

Latent Semantic Analysis got its name because after factorizing the matrix, latent hidden structure in the input data can be exposed - which can be thought of as revealing the semantic meaning of the input data.

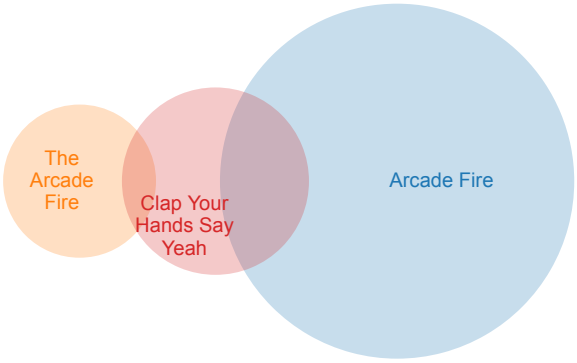
For instance, 'Arcade Fire' and 'The Arcade Fire' have no common listeners in the dataset I'm using: it was generated by Last.fm histories and people either used one label or the other in their music libraries.

This means that all of the direct methods of comparing artists think that these two bands are entirely different. However, both of these labels refer to the same band - a fact that LSA manages to pick up on since they get ranked as being the most similar to each other:

Similar to 'Arcade Fire' by Implicit ALS:

Artist	Implicit ALS
The Arcade Fire	0.968
Clap Your Hands Say Yeah	0.947
The National	0.921
The Shins	0.907
Midlake	0.880

more ...



You can also see the same effect with "Guns N Roses" vs "Guns N' Roses" and "Nick Cave and the Bad Seeds" vs "Nick Cave & the Bad Seeds". Feel free to enter other artists, but keep in mind this dataset is from 2008 so more modern artists aren't represented here.

While LSA successfully generalizes some aspects of our data, the results here aren't all that good. Take a look at the results for Bob Dylan for an example.

To do a better job while still maintaining this ability to generalize, we need to come up with a better model.

Implicit Alternating Least Squares

Recommender systems frequently use matrix factorization models to generate personalized recommendations for users. (https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf) These models have been found to work well on recommending items, and can be easily reused for calculating related artists.

Many of the MF models used in recommender systems assume explicit data, where the user has rated both things they like and dislike using something like a 5 star rating scale. They typically work by treating the missing data as an unknown, and then minimizing the reconstruction error using SGD.

The data here is implicit though - we can assume that a user listening to an artist means they like it, but we don't have the corresponding signal that a user doesn't like an artist. Implicit data is usually more plentiful and easier to collect than explicit data - and even when you have the user give 5 star ratings the vast majority of those ratings are going to be positive only (/rating-set-distributions) so you need to account for implicit behaviour anyways.

This means we can't just treat the missing data as unknowns, instead we have to treat a a user not listening to an artist as being a signal that the user might not like that artist.

This presents a couple of challenges in learning a factorized representation.

The first challenge is in doing this factorization efficiently: by treating the unknowns as negatives, the naive implementation would look at every single entry in our input matrix. Since the dimensionality here is roughly 360K by 300K - there are over 100 billion total entries to consider, compared to only 17 million non zero entries.

The second problem is that we can't be certain that a user not listening to an artist actually means that they don't like it. There could be other reasons for the artist not being listened to, especially considering that we only have the top 50 most played artists for each user in the dataset.

The Collaborative Filtering for Implicit Feedback Datasets (http://yifanhu.net/PUB/cf.pdf) paper accounts for both of these challenges in an elegant way.

To handle the case where we're not confident about our negative data, this approach learns a factorized matrix representation using different confidence levels on binary preferences: unseen items are treated as negative with a low confidence, where present items are treated as positive with a much higher confidence.

The goal then is to learn user factors X_u and artist factors Y_i by minimizing a confidence weighted sum of squared errors loss function:

$$loss = \sum_u \sum_i C_{ui}(P_{ui} - X_u Y_i)^2 + \lambda(\|X_u\|^2 + \|Y_i\|^2)$$

C_{ui} is the confidence that we have that the user likes the artist, P_{ui} is a binary value indicating if the user listened to the artist or not, and the λ is a basic L2 Regularizer (https://en.wikipedia.org/wiki/Regularization_(mathematics)) to reduce overfitting.

To minimize the user factors, we fix the item factors constant and then take the derivative of the loss function to calculate X_u directly:

$$X_u = (Y^T C_u Y + \lambda I)^{-1}(Y^T C_U P_u)$$

The item factors are calculated in a similar way, and the entire thing is minimized by alternating back and forth until it converges (hence the 'Alternative Least Squares' name).

The clever part of this paper is in how it learns over all data, but only has to do work on the non-zero items. Since P_u is sparse (the negative preferences have a 0 value), $Y^T C_u P_u$ can be easily calculated. To calculate $Y^T C_u Y$ they note that it's equal to $Y^T Y + Y^T (C_u - I)Y$. By setting the confidences for negative items to 1, $(C_u - I)$ is sparse, and $Y^T Y$ can be precalculated for all users.

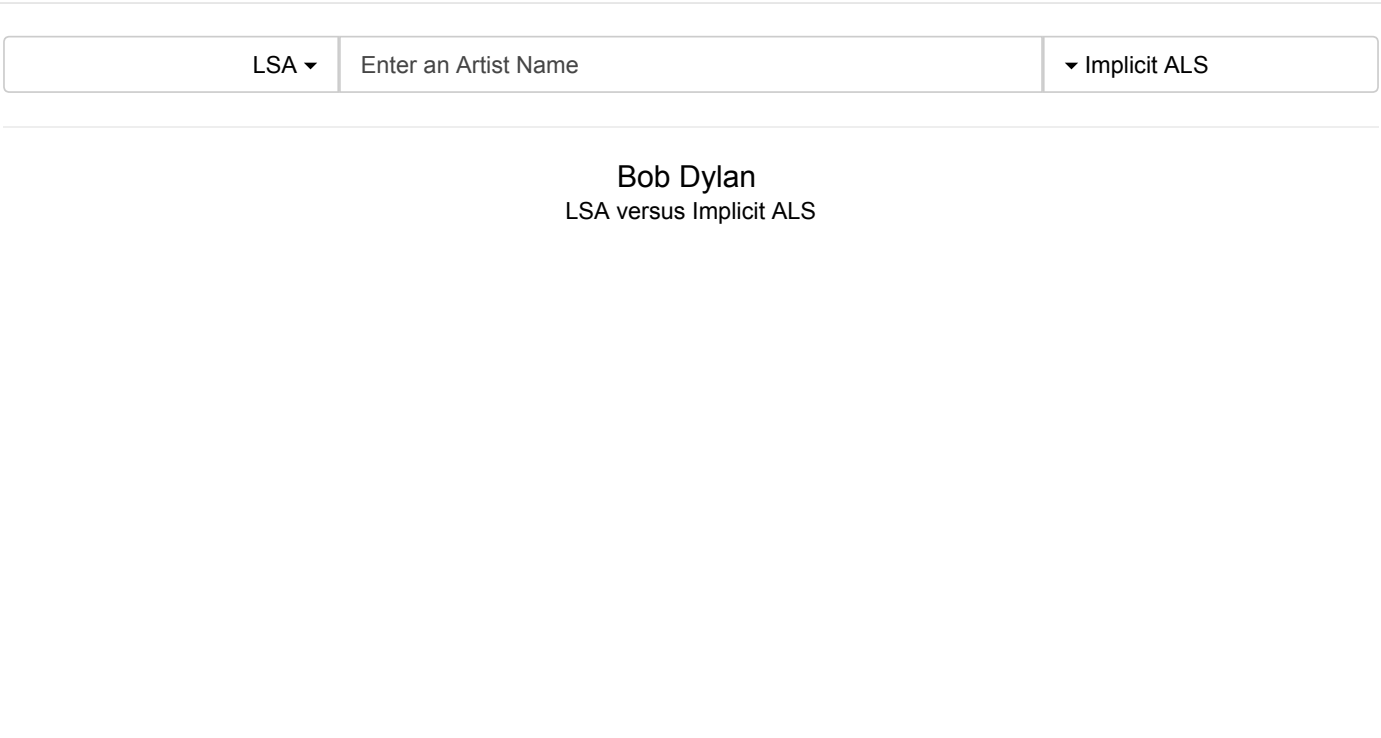
Putting the entire algorithm together in python, and you get code like:

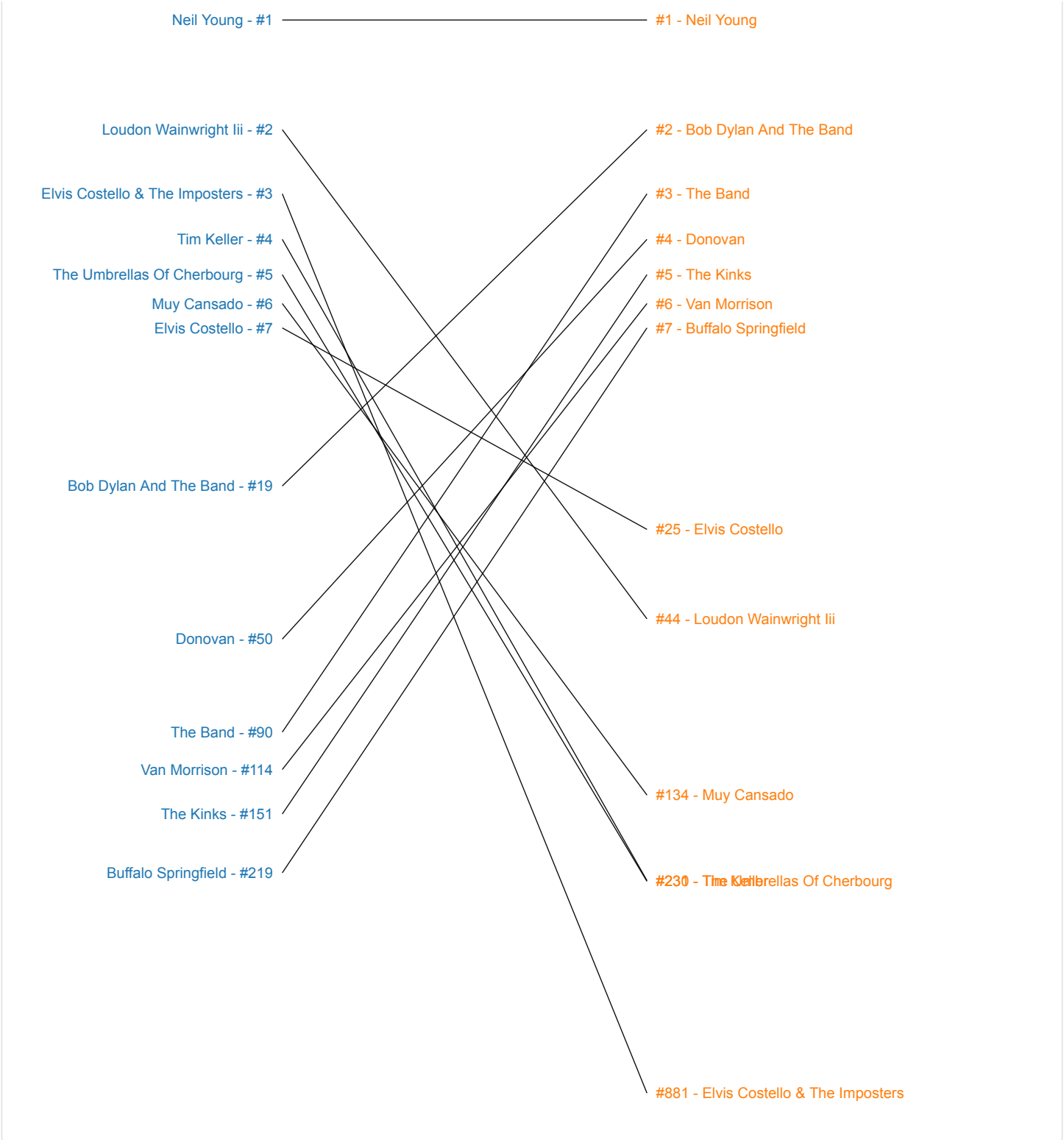
```
def alternating_least_squares(Cui, factors, regularization, iterations=20):
    users, items = Cui.shape

    X = np.random.rand(users, factors) * 0.01
    Y = np.random.rand(items, factors) * 0.01

    Ciu = Cui.T.tocsr()
    for iteration in range(iterations):
        least_squares(Cui, X, Y, regularization)
        least_squares(Ciu, Y, X, regularization)

    return X, Y

def least_squares(Cui, X, Y, regularization):
    users, factors = X.shape
    YtY = Y.T.dot(Y)

    for u in range(users):
        # accumulate YtCuY + regularization * I in A
        A = YtY + regularization * np.eye(factors)

        # accumulate YtCuPu in b
        b = np.zeros(factors)

        for i, confidence in nonzeros(Cui, u):
            factor = Y[i]
            A += (confidence - 1) * np.outer(factor, factor)
            b += confidence * factor

        # Xu = (YtCuY + regularization * I)^-1 (YtCuPu)
        X[u] = np.linalg.solve(A, b)
```

To call this, I'm using the same weighting for the confidence matrix as we used in LSA, and then calculating related artists in the same way:

```
artist_factors, user_factors = alternating_least_squares(bm25_weight(plays), 50)
```

This method leads to significantly better results than just using LSA. Take a look at a slopegraph (http://charliepark.org/slopegraphs/) comparing the results for Bob Dylan as an example:





The irrelevant results returned by LSA here are pushed out of the head of the list and are replaced by relevant results.

The nice thing about Implicit ALS is that it still generalizes the input successfully. As an example, both Guns N' Roses and Nick Cave And The Bad Seeds each still bring up their synonyms, just without some of the marginal results being returned by LSA.

Performance

The are a couple performance challenges in computing related artists like this.

The Implicit ALS code I posted above here isn't terribly fast. Since each user factor can be calculated independently of each other, this algorithm is embarrassingly parallel and well suited to multithreading - which pure Python code doesn't do well at.

To overcome this problem, I published a fast python version (https://github.com/benfred/implicit) that uses Cython and OpenMP to parallelize the computation. On this task it has a comparable running time to (https://github.com/benfred/implicit/blob/master/examples/benchmark.py) the multithreaded C++ version that Quora just published in their QMF library (https://github.com/quora/qmf).

By parallelizing computation, this library can factorize the Last.fm dataset in about 40 seconds on a c4.8xlarge (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/c4-instances.html#c4-instances-cpu-support) EC2 instance (50 factors, 15 iterations).

The problem is that using these factors to calculate all the most related artists takes about an hour once the factorization is done. To do the exact nearest neighbour calculation is a $O(N^2)$ calculation since we need to compare each artist to each other artist.

It's possible to do a much faster job using an approximate nearest neighbours search. The annoy (http://github.com/spotify/annoy) package uses a random hyperplane splitting approach to build up a forest of search trees to efficiently calculate the nearest neighbours.

The annoy package comes with Python bindings. To generate the approximate nearest neighbours using cosine distance using these bindings is pretty simple:

```
class ApproximateTopRelated(object):
    def __init__(self, artist_factors, treecount=20):
        index = annoy.AnnoyIndex(artist_factors.shape[1], 'angular')
        for i, row in enumerate(artist_factors):
            index.add_item(i, row)
        index.build(treecount)
        self.index = index

    def get_related(self, artistid, N=10):
        neighbours = self.index.get_nns_by_item(artistid, N)
        return sorted(((other, 1 - self.index.get_distance(artistid, other))
                       for other in neighbours), key=lambda x: -x[1])
```

The results using this method are largely identical, and reduce the time to compute over all artists from an hour to about 30 seconds - which includes the time to build up the index in the first place.

Erik Bernhardsson has a series (http://erikbern.com/2015/09/24/nearest-neighbor-methods-vector-models-part-1/) of excellent posts (http://erikbern.com/2015/10/20/nearest-neighbors-and-vector-models-epilogue-curse-of-dimensionality/) on how annoy works (http://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces/) if you're interested in reading more on this.

Conclusion

Matrix Factorization methods like Implicit ALS are typically used to generate personalized results - but there are some upsides to using these models for the much simpler task of generating lists of related artists.

In fact, Spotify uses a matrix factorization technique called Logistic Matrix Factorization (http://stanford.edu/~rezab/nips2014workshop/submits/logmat.pdf) to generate their lists of related artists. This method has a similar idea to Implicit ALS: it's a confidence weighted factorization on binary preference data - but uses a logistic loss instead of a least squares loss. The paper has some examples where Logistic Matrix Factorization does a better job calculating similar artists than Implicit ALS (2).

There are many other matrix factorization methods that can be used instead of the couple of talked about here though. For instance, Bayesian Personalized Ranking (http://www.algo.uni-konstanz.de/members/rendle/pdf/Rendle_et_al2009-Bayesian_Personalized_Ranking.pdf), and Collaborative Less-is-More Filtering (http://www.ci.tuwien.ac.at/~alexis/Publications_files/climf-recsys12.pdf) both attempt to learn a factorized representation that optimizes the ranking of artists for each user.

The real power with these models are in generating personalized recommendations for each user - which I'm hoping to talk more about in a future post.

Footnote 1

In techniques like LSA, it's common to use TFIDF to weight the input matrix before doing the matrix factorization. However from the previous post, we know that BM25 weighting produces good results in calculating similarity between artists - so I'm using that weighting instead here. See the example app in my Implicit library (https://github.com/benfred/implicit/blob/master/examples/lastfm.py) for code on how to do this efficiently.

Footnote 2

The results here for Implicit ALS are much better than those reported in the Logistic Matrix Factorization paper (aside from Mumford & Sons, who got popular after the Last.fm dataset was released). This is mainly because of how I am weighting the confidence matrix - by using a simple linear weighting I also get terrible results.

Published on 03 May 2016