Madhu Vijay
Jordan Abbasi
Tyler VanValkenburg
April 8, 2016

Practical 3: Predicting Music Tastes

1 Technical Approach

1.1 Feature Creation and Data Representation

For this project, we represented the provided data set as an $n \times m$ matrix, where $n = 233{,}286$ is the number of distinct users and m is the number of features corresponding to them. We included one feature for each of the 2000 artists in the data set. The value of each such feature was set to the number of times that its particular user had played that particular artist, where the data was available; if the data was not available in the training set, the feature was set to zero.

In addition to tracking artist plays, we included binary features corresponding to the demographic characteristics of each user. Most notably, we included binary indicator features denoting the user's sex, the user's age (such that the feature for age i was set to 1 if and only if the user was i years old), and the user's country of origin. In cases where learning run time became a significant burden, we trimmed these additional features in order to increase efficiency; for example, for the NMF algorithm, we included indicator features for every ten years of age (e.g. a binary feature that equaled 1 for all users in their 20s), rather than for each year. Ultimately, the number of features consistently lied within the range of 2000 to 2500; for instance, in the final SVD algorithm, we used $m = 2353$ features (of which 2000 corresponded to specific artists).

Maintaining this matrix of roughly 500 million elements efficiently was no simple task. First and foremost, in order to perform computations efficiently, we typically had to use a sparse representation. This makes sense due to the number of nonzero elements in the data matrix being so small, and to achieve it we used one of two methods in different cases. When we used inbuilt factorization packages, we used scipy.sparse in order to store sparse matrices efficiently and use them as direct inputs to inbuilt functions. In cases where we manually implemented the training and predicting process, we found it most convenient to store each row of the matrix as a dictionary containing only the nonzero elements, requiring much less storage space than the full matrix would otherwise occupy.

For algorithms like NMF, it was beneficial to have features that were roughly standardized, in order to keep the loss function scale-independent. However, standardizing the features to have a zero mean and unit variance would upset the sparsity of the data, and so render the algorithms much less tractable. As a compromise, we used scikit-learn's MaxAbsScaler class, which scales each feature by a constant such that its maximum is 1 and the sparsity of the data representation remains intact.

1.2 General Approach

Our general approach was roughly inspired by a paper that was mentioned in this practical's guidelines, entitled "Matrix Factorization Techniques for Recommender Systems". Based on the assumption that any person's musical tastes tend to center around a few genres, we decided on unsupervised clustering as an effective way to analyze the data. Specifically, we focused on matrix factorization, as our objective was not the cluster assignments themselves, but rather the ability to reduce dimensionality and predict unknown dimensions based on existing knowledge. As a high-level overview, matrix factorization algorithms attempt to decompose a $n \times m$ data matrix into the product of an $n \times K$ matrix and a $K \times m$ matrix (with $K < m$) in order to project the data onto a K-dimensional subspace. We tested multiple factorization algorithms of this variety, and they are detailed below.

1.3 SVD

The matrix factorization algorithm that we developed most comprehensively was singular value decomposition (SVD). Though an exact SVD exists for any matrix, a lower-dimensional approximate SVD can be used for clustering and dimensionality reduction. We approached this method in two ways. The first approach was inspired by an article we read about one of the more successful approaches in the Netflix Prize competition. This approach involved producing an approximate SVD *manually*, via explicit implementation of gradient descent with regularization. Despite it's obvious tediousness, this approach was extremely well-suited to our particular task, for reasons that will be expanded on later.

Despite our success with the above approach, for the sake of completeness we also tried applying scikit-learn's TruncatedSVD class on the sparse, scaled data matrix. This model was relatively convenient to use: the model operates directly on the data rather than explicitly computing any covariance matrix, which makes it particularly suited for sparse matrices as their respective covariance matrices are not guaranteed to be sparse or tractable.

1.4 NMF

Nonnegative matrix factorization is a matrix decomposition algorithm that can be used (when all features are nonnegative) for clustering or dimensionality reduction. NMF was useful in this context for a few reasons. Conveniently, the scikit-learn implementation of NMF is specifically designed to work with scipy sparse matrices, rather than requiring that they be cast as dense matrices. Furthermore, nonnengative factorization can sometimes produce factorizations more efficiently because of how it exploits the uniform parity of the input matrix to assist in the factorization algorithm. (Correspondingly, because the $k \times m$ output matrix is now constrained to be nonnegative, the algorithm in general no longer ensures that the k component vectors in $\mathbb{R}^m$ form an orthonormal set, unlike in some other algorithms including PCA.)

1.5 PCA

Finally, we also attempted to use principal component analysis to perform dimensionality reduction on the data. We attempted to use several scikit-learn classes, including PCA, SparsePCA, and MiniBatchSparsePCA. (The advantage of the latter two is that they use L1 regularization to produce sparse components, which makes for easier testing and prediction.) Unfortunately, the scikit-learn PCA implementations all require the use of dense matrices, and casting our data as a dense matrix led to memory and efficiency issues. Thus we decided not to pursue this option any further with the given data set.

1.6 Incorporation of Baselines

While matrix factorization is helpful, we also recognize that the user median baseline is generally very successful at predicting listens. When producing final predictions with our matrix factorization algorithms, we tended to use some convex combination of the matrix factorization prediction, the user median, and the user-bias-adjusted artist median, in order to incorporate all three sources of information into our final prediction. The details and implications of this methodology are discussed in the following sections.

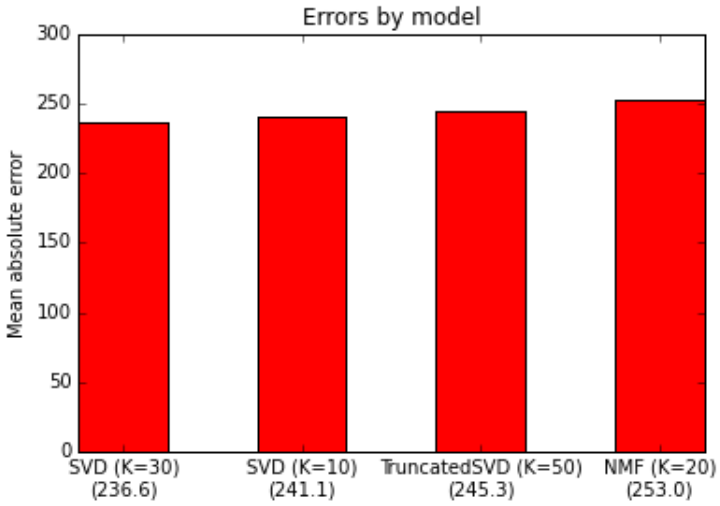
1.7 Validation and Model Comparison

To optimize hyperparameters for each model, we split off a subset of 10% of the data for validation. While 30% would be a more conventional choice, we thought that eliminating too much of the data set

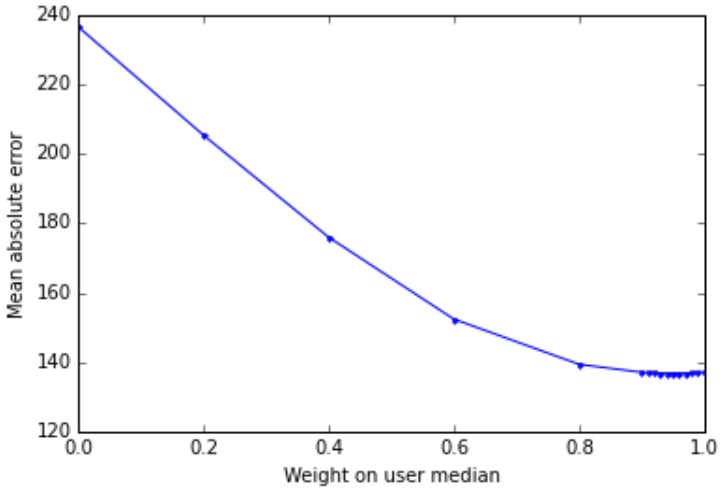
would make the data matrix unrealistically sparse, which might lead to inaccuracies in factorization. The 10% validation set still gave us roughly 400,000 unseen data points, a healthy number with which to optimize hyperparameters. Similarly, to compare different model classes, we computed the loss function (mean absolute error) using validation data.

2 Results

The following bar chart shows the errors from a few of our models. Within each model class, we used validation to choose hyperparameter values. (In particular, we experimented with some different values of K; and the manually implemented SVD allowed several additional avenues for parameter choice, such as the number of gradient descent epochs.) Note that all results in this bar chart are based entirely on matrix factorization predictions, without incorporating user medians. The number in parentheses under each model is the mean absolute error for that model from validation.



After testing various models, we chose to use the manually implemented SVD with $K = 30$ dimensions. We then tried to modify the SVD predictions by using convex combinations of the form $(1-w)p+wu$, where p is the SVD prediction, u is the user median for a particular user, and $w \in [0, 1]$ is a constant weight. The following plot shows the validation results from trying different values of w:



As we can see, the SVD predictions themselves are very imperfect, and are improved significantly

by incorporating the user medians. This result and its implications are discussed below.

Ultimately, our optimal model was a manual SVD with $K = 30$, 10 gradient descent epochs, and $w = 0.95$. This achieved a mean absolute error of 136.85 in validation and 137.42 on the Kaggle leaderboard, which surpassed both benchmarks and put us at 21th place.

3 Discussion

3.1 Comparison of Model Classes

Ultimately, our manual SVD algorithm consistently outperformed other models (including scikit-learn's own TruncatedSVD model). While manual gradient descent seems tedious, it was ultimately much better suited to our task than other matrix factorization packages. This is because we only cared about the loss function evaluated on the *specific elements* of the matrix that were in the training data, *not* on all 500 million elements of the matrix. Using manual gradient descent enabled us to iterate over only the elements that we cared about, producing more useful results. Our SVD algorithm was also more space-efficient than most algorithms: by using a dictionary implementation of the data matrix, it consistently maintained sparsity of the data and made prediction relatively easy.

While the performance differences between the various algorithms do look small on the bar chart above, those differences were still ultimately meaningful. (For example, the best team on the Kaggle leaderboard only beat the user-median benchmark by a mean absolute error of 3.) Therefore, SVD with $K = 30$ was substantially superior to the other models we tested.

3.2 Comparison to Baseline

As we can see in the second plot above, the optimal prediction method was to predict $0.95u + 0.05p$, where u is the user median and p is the SVD model prediction. This implies that our best predictions were very close to the user medians themselves, and only slightly incorporated information from the SVD. However, it is worth noting that the optimal coefficient on u was very consistently 0.95 in SVD validation, even when we varied other parameters significantly. This implies that even though our best model does not add a substantial *amount* of information relative to user medians, it is very *consistently* able to produce improvements over the benchmark. This is consistent with the fact that this convex combination prediction was able to beat the benchmark on Kaggle, albeit not very significantly.

3.3 Potential Enhancements and Avenues for Further Work

Although we were able to successfully use a matrix factorization technique to improve on the baseline predictions, there is significant room for exploration and improvement. First, in terms of feature engineering: We would have liked to scraped and used data on the artists themselves, rather than just the users. For example, MusicBrainz has publicly accessible data on the genre of each artist, and that information could be incorporated to improve the quality of our predictions.

Also, in terms of model choice: We would have liked to have tested other model classes and other types of models. Unfortunately, even within matrix factorization, the class of models we could feasibly use on a personal computer was artificially restricted by the fact that certain models (like PCA) require the use of dense matrices. We would like to try those models (by, for example, running on a cluster), in addition to branching out and trying other unsupervised models altogether.

Code

You can find our code in a git repository at https://github.com/madhuvijay95/cs181-practical3.