CS 181 - Machine Learning (Spring 2016)
Practical 3: Preferences in Streaming Music

Team BridgingtheGAK

Gioia Dominedò (40966234) Amy Lee (60984077) Kendrick Lo (70984997)

May 5, 2016

For this project we used machine learning methods to predict the number of times different users will listen to tracks by different artists over a given time horizon. Our analysis was based on: basic demographic attributes that were provided for most of the 233K users, attributes for the 2K artists that were scraped from MusicBrainz[1] and Wikipedia, and a training set of the historical number of plays of 4.1M user-artist pairs.

At the time of submission, we had uploaded 12 different sets of predictions to Kaggle and achieved a best mean absolute error (MAE) of 137.50, which put us above the user median baseline. Though we tried a number of more sophisticated modeling techniques, our best results were achieved by adjusting the user median baseline to include a small amount of artist bias.

1 Technical Approach

Given the volume of data – with over 4.1M user-artist pairs in the training set and 4.15M pairs in the test set – we focused our efforts for this practical on judiciously creating features and on dimensionality reduction techniques. The biggest challenges that we faced were time and memory constraints. Given additional time, we would have liked to explore using parallelization (e.g. via technologies such as Apache Spark) to facilitate the algorithms and approaches described below.

We spent a considerable amount of time on this practical experimenting with various modeling techniques. The primary modeling techniques attempted included collaborative filtering kNN, K-means clustering, matrix factorization, PCA, random forest, and linear regression. We also spent a considerable amount of time on feature engineering, including scraping the MusicBrainz website and Wikipedia for artist features. Excerpts of our Python code are included in Appendix E.

1.1 Exploratory Data Analysis

We started by examining the data quality and the distribution of the provided data on user attributes, artist attributes, and play counts. The combined training and test sets included 233,186 unique users and 2,000 unique artists.

[1] https://musicbrainz.org

The user data included basic demographic attributes: gender, age, and country. All users had a country attribute, with over 20% of users hailing from the United States. Figure 1 shows the distribution of the top 20 user countries; to put this into perspective, the total dataset included 239 unique user countries.

In contrast, we found that gender and age were not present for a significant proportion of users. Specifically, almost 20% of users had missing ages, and almost 10% had missing genders (see Figure 2). Forty users had ages of -1337, which most likely represented a code for missing data. Additional suspicious entries included one 666-year-old, one 1,002-year-old and two 121-year-olds – with the latter being potentially plausible! Ignoring these outliers, we found that the majority of users are between 15-45 years of age (see Figure 3). Moving on to gender: approximately 65% of users are male and 25% are female, with the balance unspecified (see Figure 4).

The provided artist data only included the artists' MusicBrainz IDs and names (10 of which were missing). This led us to consider the possibility of using the IDs to scrape additional artist attributes from the MusicBrainz website and from Wikipedia.

We also looked at the distribution of the number of artists each user had played at least once. Figure 5 shows that out of the 2,000 artists, most users had play counts greater than 0 for about 10-30 artists, with both the mean and median at about 18, and the distribution looking approximately normal. In contrast, Figure 6 shows the distribution of the number of users that had played each artists at least once. This distribution shows an exponentially decreasing trend, with the median and mean user counts at about 1,097 and 2,077, respectively.

Finally, we examined the distribution of play counts, and noted that the majority of user-artist pairs have a relatively low play count (Figure 7 shows the exponentially decreasing trend).

1.2 Feature Engineering

While we were provided with basic demographic features on the users, we did not have any additional artist information with which to work. We therefore scraped additional data from both the MusicBrainz website (using the provided MusicBrainz IDs) and Wikipedia.

Artist "tags". Tags are typically related to either genres or artist locations (see Figure 8 for the 20 most common tags). The majority of artists only had a few tags, though we were surprised to see that some artists were associated with several hundred tags (see Figure 9). As a starting point, we converted these tags into binary indicator variables for all of the artists.

Artist year of birth/formation year. We considered Wikipedia as an additional source on artist information, and used it to scrape the year of birth for individual artists and the formation year for groups. Figure 10 shows the resulting distributions and, in particular, the fat left tails of both distributions.

1.3 Modeling Approaches

Validation. Wherever possible, we used validation to check how well our models generalized before submitting their predictions to Kaggle. In these cases, we used a common 70-30 split. However, the size of the data and the computational cost of the more complex models that we tested meant that it was not always possible (or practical) to go through the additional validation step. We specify below the cases where we had to rely on the MAE on the entire training set, instead of a training-validation split.

Where we did use a validation set, we noticed that we were able to achieve very similar Kaggle scores when compared to the error computed on the validation set. Accordingly, we were able to rely more heavily on the results of using the validation set to estimate out-of-sample error when determining whether it would be worthwhile training on the entire training set to make a Kaggle submission.

Baseline. We started by examining the provided code and baseline predictions. This baseline calculates the median number of plays for all users (i.e. aggregating all artists whose music they have listened to), and uses this value to predict the number of plays for new artists. If the algorithm encounters a user in the test set that is not present in the training set, then it uses the global median play count for the entire training set as a baseline prediction. This approach results in a MAE of 129.34 on the (entire) training set and 137.79 on Kaggle.

Adjusted baseline with user and artist biases. Our first attempt to improve on the baseline was a modified version that adjusts for systematic tendencies of certain users to listen to more or fewer plays than others, and of certain artists to receive more or fewer plays than others. This yields the following modified prediction:

global median + (user median - global median) + (artist median - global median)

The final two terms in this equation correspond to the user-specific and artist-specific biases in the play counts – in other words, the extent to which their play counts tend to systematically diverge from the global median.

We found that the above approach underperformed compared to the original user median approach (MAE of 135.33 on the validation set and 142.62 on Kaggle), so we experimented with different weightings of the user and artist biases. We found that the following prediction optimized validation error:

*global median + 0.99 * (user median - global median) + 0.0999 * (artist median - global median)*

This approach resulted in a MAE of 129.23 on the training set and 137.50 on Kaggle, beating the user median baseline.

"Basic" linear regression. Given the promising results of the adjusted baseline approach, we attempted to run a simple linear regression with a beta parameter for each artist and each user. This was achieved by contrast-coding a column for each artist and user into our design matrix. We achieved a validation set accuracy of 190.0 (vs the baseline of 136.0) by running this approach on a smaller sample of 1,000 training data points. We expected that this validation set accuracy would improve as we expanded the training data set to include more users and artists, as this would produce "better informed" model coefficients; however, we were constrained by computing resources, as we were not able to store a numpy matrix of 4M x (233K + 2K) dimensions in memory.

kNN collaborative filtering. We focused our experiments with the k-nearest neighbors algorithm on the relationship between artists rather than between users. The intuition behind this was that a user's play count pattern was likely to be comparable for similar artists. On a practical level, we also chose to begin with an artist-artist based model because the $\binom{2000}{2} \approx 2M$ artist similarities were far more tractable than the $\binom{233K}{2} \approx 27B$ user similarities. With additional time and computing resources, we would have liked to also explore the user-user option, as the artist-artist model assumes that only a given user is similar enough to make a predicted play count, whereas there are likely to be similar patterns across users that could be used to extrapolate missing play count data.

The general idea is as follows: we can predict the play count of new artists for a given user as the play count of the closest "neighbors" (i.e. similar artists that the user has already listened to). We originally planned to determine the similarity measure between a given pair of artists by finding the users that they have in common (i.e. all users with a play count greater than zero for both artists) and calculating the Pearson correlation coefficient based on the play count of their common user support. However, we found this calculation to be incredibly slow – in fact, we calculated that it would require 23 days to finish processing!

In order to speed things up, we used principal component analysis (PCA) to reduce the dimensionality of the number of users for each artist. We then took advantage of numpy's fast matrix correlation function (`corrcoef`), which returned a 2,000 x 2,000 matrix of product-moment correlation coefficients using the reduced artist-user matrix. This allowed us to calculate the predicted rating p_{ij} for user i and artist j as follows[2]:

$$pij = \bar{r}_i + \kappa \textstyle\sum_{k=1}^{n} w(i,p)(\tilde{r}_{pj} - \tilde{r}_p) \ ,$$

where $\bar{r}_i$ is the average artist rating for user i, $w(i,p)$ is a similarity weighting (in this case, the Pearson correlation coefficients), and κ is a normalizing factor that ensures that the absolute value of the weights sum to zero. Despite this optimization, our kNN algorithm still took a full second to process three user-artist pairs – which equates to approximately 2 weeks for the full 4.1MM artist-user pairs.

Finally, we also experimented with using the median of the k nearest neighbors' play counts instead of the weighted predicted play count based on the Pearson correlation similarity in the formula above. We tested the following approach: if a user had played songs by fewer than 4 artists, we simply took the median of all the actual play counts for that particular user. If, instead, a user had played songs by a greater number of artists, we then took a certain top portion of the nearest neighbors (the proportions tested varied between 50% and just the single nearest neighbor – "nearest neighors" meaning the most similar artists based on the pre-calculated similarity coefficient). Unfortunately, this approach either matched or underperformed the modified baseline from the previous section.

With additional time, we would have experimented further with the user-based neighborhood model by combining the work from the K-means clustering approach discussed below and by only calculating and searching for similarities between users in the same user classes, thus reducing the computation volume of user-user comparisons.

K-means clustering. In addition to k-nearest neighbors, we also attempted to implement k-means clustering on the users and artists, including both "regular" and k-means++ initializations.

[2] http://goldberg.berkeley.edu/pubs/eigentaste.pdf

The aim was to reduce the users and artists into classes, in order to be able to deal with "similar" people as groups. The features and adjustments that we experimented with are detailed in Appendix C. In addition, it was also necessary to clean the artist profile data to correct for the large number of missing data points.

We had originally tried to use the user and artist clusters to find the single nearest neighbor within the training set and return that neighbor's play count; however, we estimated that it would take over 500 days to assign predictions to all test values! In order to reduce dimensionality, we instead built a (user clusters) x (artist clusters) lookup table to store the median number of plays per user-artist type combination and returned that value instead. We expect that this "averaging" does not work well for extreme points, but it did allow us to run the algorithm to completion.

We experimented with various feature combinations (see Table 2), and in all cases used the "elbow method" to determine the optimal number of user and artist clusters – though we found that there was not always an unambiguous optimum (see Figures 11 and 12). Our preferred model used a 4x5 lookup table, and was able to achieve results that were competitive with – but not materially better than – those from the user median baselines. We noticed that adding features did not improve performance due to the curse of dimensionality, as points were less likely to be clustered when there were too many features. As an example, Figure 13 plots the clusters and means for two features, before adding the artist age and genre features.

Matrix factorization We experimented with latent factor models such as non-negative matrix factorization, which was of particular interest due to its ability to handle sparse matrices well. Our user-artist matrix is sparse due to the fact that most users only listen to a few artists' songs (about 10% of artists, according to Figure 5) and most artists do not have many users who played their songs (the median is approximately 1,097 users, compared to the total 233K). Combined with its ability to identify similar users and artists as a function of latent factors, we believed that this technique would achieve more accurate predictors. However, we once again came up against issues surrounding computing time, as the technique takes polynomial time and our user-artist matrix has about 400M scalars.

"Majority" classifier. While exploring matrix factorization, we realized how sparse our matrix was – with only a small percentage of users playing each artist's songs, and an even tinier percentage of artists played by each user. We therefore attempted a "majority classifier" approach that predicts 0 plays for all artist-user combinations in the test set. This achieved a MAE of 253.56 on Kaggle. Well, it was worth a try!

Principal component analysis. We attempted to alleviate the computational difficulties created by the size of the datasets by using principal component analysis (PCA). To do this, we built a 2K x 233K artist matrix and a 233K x 2K user matrix. In the former, each user represents a "feature", such that any given artist's row will show the play counts for all users; in the latter, each artist represents a "feature", such that any given user's row will show its play count for all artists.

We carried out the following steps: first, we standardized the data using scikit-learn's StandardScaler; then, we using RandomizedPCA (due to its improved performance over PCA for a large number of features) to obtain reduced dimensional forms of both matrices; finally, we used the reduced matrices to construct our training and test design matrices by concatenating the appropriate user and artist matrix rows for each unique user-artist pair. In addition, we also included feature columns for the user-specific median play count, artist-specific median, and the global median.

Finally, we ran both linear regression and random forest on this shrunk design matrix. On the full dataset, random forest yielded a MAE of 66.82 on the training data set and 170.89 on the validation set, suggesting severe overfitting (see Table 3). Given these results, we did not attempt to predict on the test data.

With additional time, we would have liked to explore combining the PCA features with the provided user attributes and the scraped artist attributes, to see whether this would improve performance.

"Basic" random forest. Finally, we experimented with a "basic" random forest approach. For this model, we built a design matrix based on the users' demographic attributes of gender, age, and country (with the latter converted into binary indicator variables), the artists' tags (again, converted to binary indicator variables), and the users' and artists' median plays.

We initially attempted to train a random forest model using this full design matrix, but this resulted in out-of-memory errors. We therefore worked to reduce the size of our design matrix by removing binary indicator variables associated with less than a certain number of users or artists. We were finally able to train a random forest model by dropping all countries associated with less than 1K users, and all tags associated with less than 20 artists.

Our first attempt, using the default random forest parameters, yielded a MAE of 69.11 on the (entire) training set and 181.34 on Kaggle. This indicated severe overfitting to the training data. Given the training time required, we decided to skip cross-validation and simply train the model on the entire training set with a larger number of estimators. We attempted to do so with 100 estimators; however, the model was still running after nearly 24 hours and we were not able to complete this experiment.

2 Results

As discussed in the previous section, we attempted to run eight different classes of models (excluding the provided baseline) but were consistently faced with memory and/or computational constraints. In addition, the models that we were able to run to completion either underperformed the baseline or were not able to beat it by a significant margin. Table 1 summarizes the models that we built and, for those that we were able to finish running, MAE scores on the training set, validation set and/or Kaggle.

At time of submission, we had uploaded 12 set of predictions to Kaggle. Our best-performing model performed slightly better than the baseline (137.50 vs 137.79). The two models that we selected as our final entries for the Kaggle competition both used features based on user and artist biases.

3 Discussion

The discussion above details our thought process and the techniques with which we experimented. Our main takeaways are listed below.

- **Model complexity:** We experimented with eight different modeling techniques of varying complexity. Certain models (e.g. kNN collaborative filtering) were very promising in theory, but impossible to execute in practice due to the significant compute time that would have been necessary to run to completion. On the other end of the spectrum were the adjusted baseline models, which turned out to be our best-performing models despite their relative simplicity. It is very likely that the more complex models would have ultimately outperformed the adjusted baselines, given sufficient time and computing resources, but it is worth noting the power of a simple modeling approach in a constrained situation such as this one.

- **Dimensionality:** Our biggest challenge was reducing the dimensionality of the data in order to avoid or minimize memory and/or computational constraints, while still retaining information. The best-performing adjusted baseline models performed well because they "cross-bred" information from other artists and users, while still maintaining information specific to the target user and artist – all in a manageable feature matrix.

- **Choosing parameters:** Previous practicals have highlighted the importance of tuning the parameters for individual models (e.g. the number of estimators in a random forest). In this practical, the focus was on choosing the appropriate parameters for dimensionality reduction when creating the feature matrix. For example, our combined PCA and random forest approach looked promising, but there was clearly too much loss of information from the necessary PCA dimensionality reduction. Given additional time, we could have experimented with lower degrees of dimensionality reduction; alternatively, we could have added the scraped artist data and compared the results of PCA on that expanded dataset.

- **Ensemble techniques:** This practical highlighted the importance of experimenting with different techniques and combining the key takeaways of each method attempted. For example, we can infer from the results of the adjusted baseline model that user-user based neighborhood model could have potentially performed very well; given more time, we could have even combined it with the k-means modeling approach in order to reduce the volume of computation.

4 Appendix

A Figures

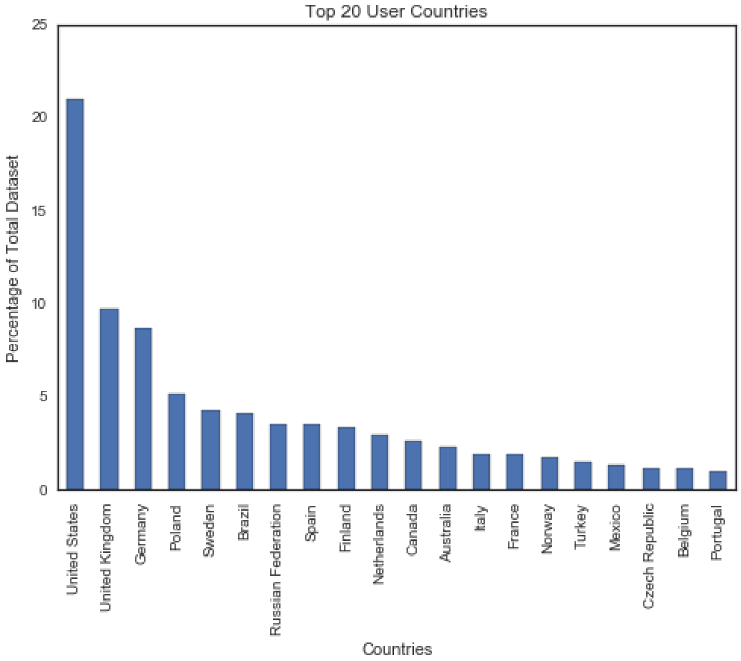

Figure 1: Top 20 User Countries

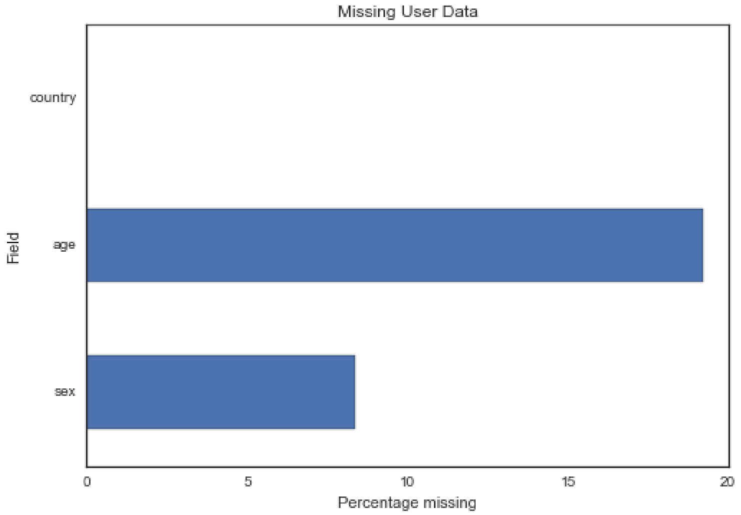

Figure 2: Percentage of Missing User Data By Attribute

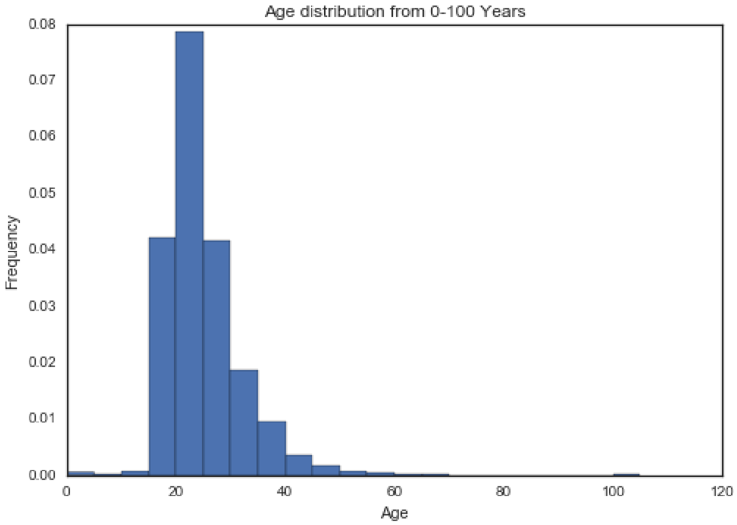

Figure 3: Distribution of User Age

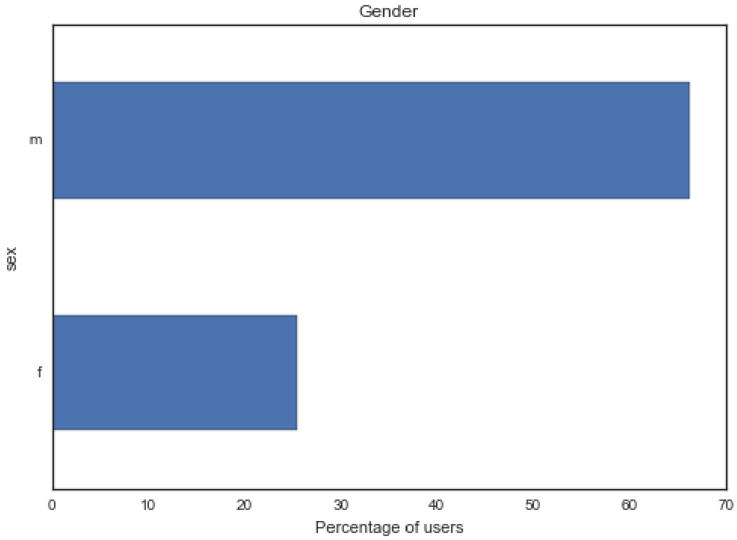

Figure 4: Distribution of User Gender (Excluding Missing Data)

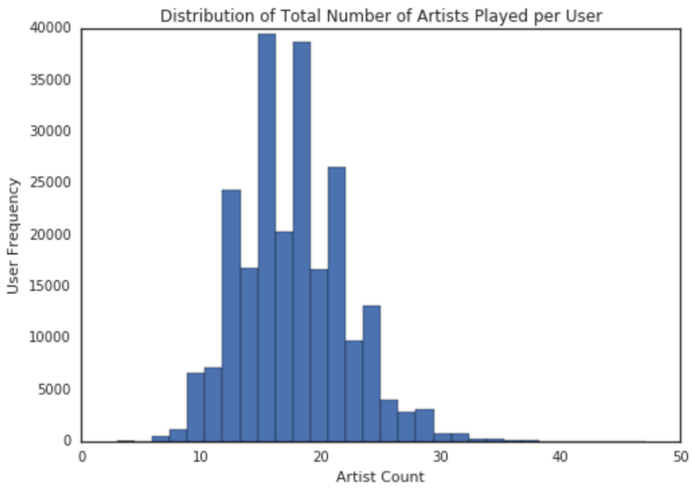

Figure 5: Distribution of Number of Users per Artist with Play Counts

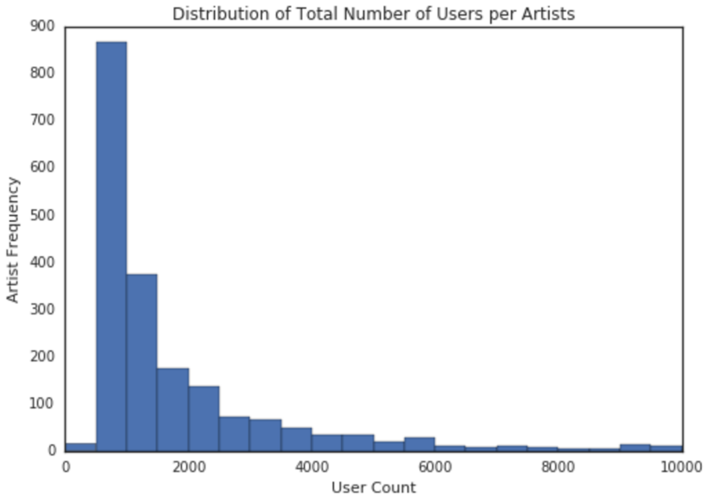

Figure 6: Distribution of Number of Artists Played Per User

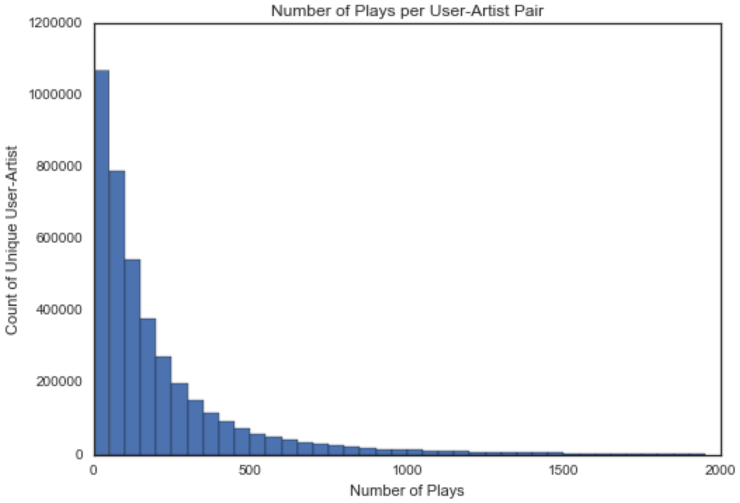

Figure 7: Distribution of Play Counts

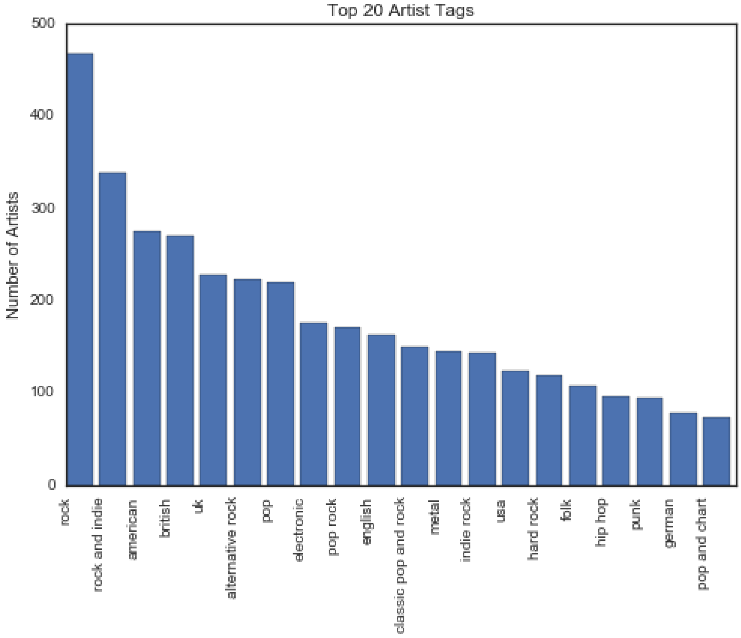

Figure 8: Top 20 Artist Tags

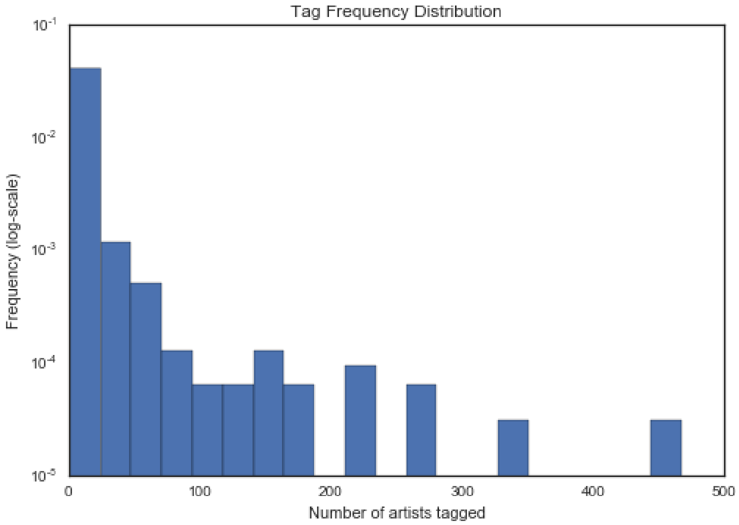

Figure 9: Artist Tag Frequency

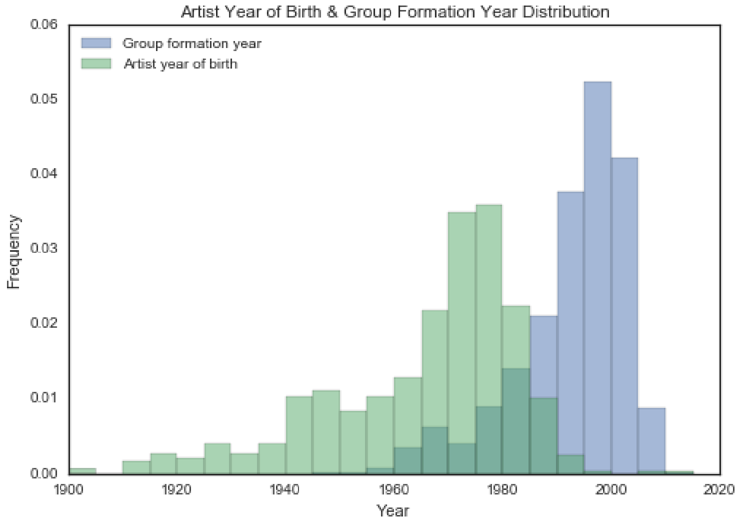

Figure 10: Distribution of Artist/Group Ages

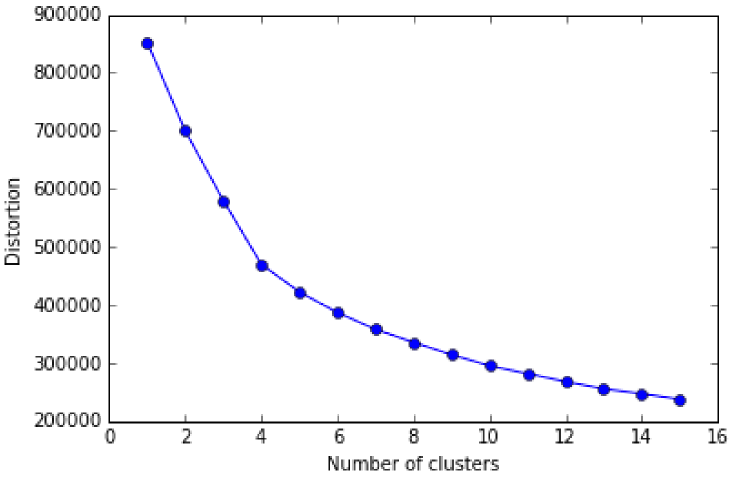

Figure 11: Plotting Distortion Values for Various K = No. User Clusters

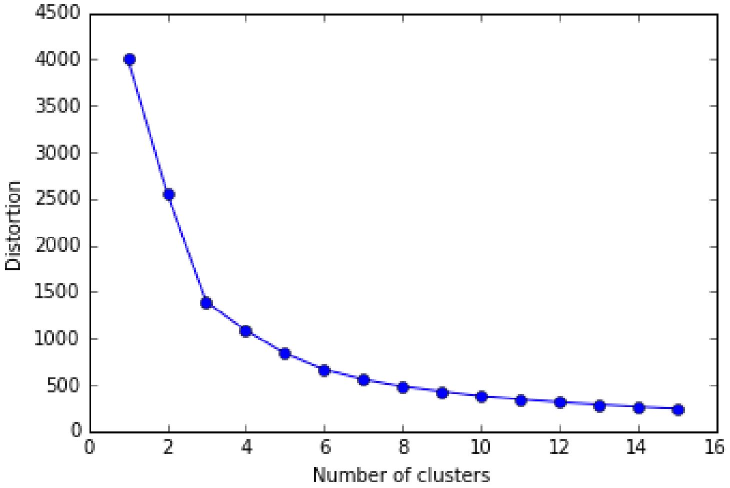

Figure 12: Plotting Distortion Values for Various K = No. Artist Clusters

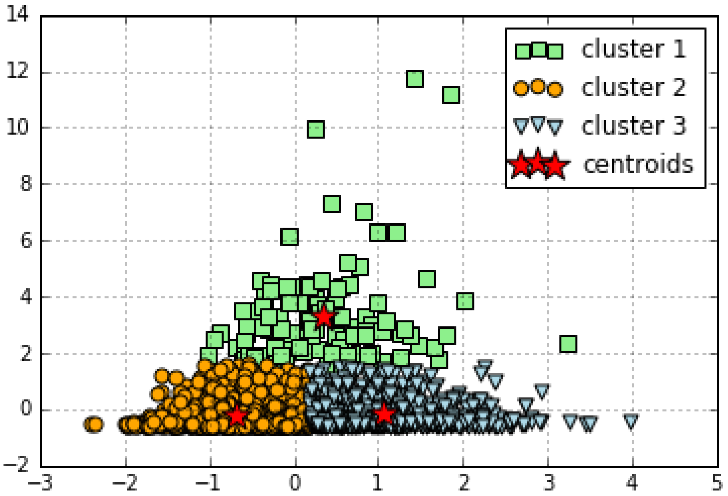

Figure 13: Clusters and Means for 2 Artist Features

B Model Summary

Model	Ran successfully?	Training MAE	Validation MAE	Kaggle MAE
Baseline (user median)	Yes	129.34	-	137.79
$1 \ast$ user bias $+ 1 \ast$ artist bias	Yes	135.33	-	142.62
$0.8 \ast$ user bias $+ 0.2 \ast$ artist bias	Yes	133.42	-	-
$1 \ast$ user bias $+ 0.08 \ast$ artist bias	Yes	129.24	-	137.61
$0.99 \ast$ user bias $+ 0.0999 \ast$ artist bias	Yes	129.23	-	137.50
kNN collaborative filtering	Not on full dataset	-	-	-
K-means clustering	Some variations		143.31	
Matrix factorization	No	-	-	-
Random Forest with PCA	Yes	66.82	170.89	-
Linear Regression with PCA	Yes	164.45	169.34	-
“Basic” random forest	Not on all features	69.11	-	181.34
“Basic” linear regression	Not on full dataset	157.95	-	-
“Majority” Classifier	Yes	230.59	-	253.56

Table 1: Attempted Model Approaches

C K-Means Clustering

C.1 Features

- User clusters:
  - Gender (provided): converted to binary indicator variables for male, female, missing.
  - Age (provided): identified all values less than 5 or greater than 95 as missing; created an additional indicator variable for missing values; replaced missing values with the average; standardized.
  - Country (provided): replaced by the rank number of users; standardized.
  - Total number of plays per user (new): used as a proxy for frequent vs occasional listeners; standardized.
  - Total number of artists listened per user (new): used as a proxy for broad vs narrow taste; standardized.
- Artist clusters:
  - Total number of plays per artist (new): used as a proxy for popular vs lesser-known artists; standardized.
  - Total number of users that listened to artist (new): used as a proxy for broad vs narrow fan bases; standardized.
  - Age of artist (new): scraped from Wikipedia; standardized.
  - Year band formed (new): scraped from Wikipedia; standardized.
  - Tag/genre (new): scraped from MusicBeatz; converted to 1,363 indicator variables.

C.2 Model Configurations

User clusters	Artist clusters	Validation MAE
4	3	180.65
4	4	180.35
4	5	180.46
4	6	180.40
4	7	180.30
4	8	180.20
4	9	180.13
4	10	180.16
4	15	179.97
4	20	179.92
6	3	172.63
6	4	172.35
6	5	172.46
6	6	172.40
6	7	172.30
6	8	172.22
6	9	172.15
6	10	172.18
6	15	172.00
6	20	171.95
8	3	174.10
8	4	173.82
8	5	173.92
8	6	173.86
8	7	173.76
8	8	173.68
8	9	173.61
8	10	173.64
8	15	173.46
8	20	173.41
10	3	175.08
10	4	174.79
10	5	174.91
10	6	174.84
10	7	174.73
10	8	174.64
10	9	174.58
10	10	174.61
10	15	174.42
10	20	174.37
15	3	169.39
15	4	169.15
15	5	169.25
15	6	169.17
15	7	169.08
15	8	169.00
15	9	168.95
15	10	168.96
15	15	168.79
15	20	168.76
20	3	168.78
20	4	168.54
20	5	168.64
20	6	168.56
20	7	168.47
20	8	168.39
20	9	168.34
20	10	168.36
20	15	168.19
20	20	168.15
25	3	166.31
25	4	166.10
25	5	166.19
25	6	166.10
25	7	166.01
25	8	165.94
25	9	165.88
25	10	165.89
25	15	165.75
25	20	165.70
30	3	164.17
30	4	163.99
30	5	164.08
30	6	164.00

User clusters	Artist clusters	Validation MAE
30	7	163.90
30	8	163.81
30	9	163.76
30	10	163.78
30	15	163.64
30	20	163.61
50	3	161.87
50	4	161.66
50	5	161.75
50	6	161.66
50	7	161.57
50	8	161.51
50	9	161.46
50	10	161.48
50	15	161.34
50	20	161.32
55	20	160.06
60	20	159.89
75	20	157.80
100	20	156.14
100	25	156.08
200	20	151.15
200	25	151.12
500	20	146.22
500	25	146.31
1,000	20	144.12
1,000	25	144.25
2,000	20	143.31

Table 2: K-Means Clustering Models

D Principal Component Analysis

Number of trees	Maximum features	Minimum samples	Training MAE	Validation MAE
10	1.0	1	68.49	177.93
15	0.6	5	93.51	160.55
30	0.6	5	93.13	156.78
30	0.6	10	111.00	153.87
30	0.8	10	109.95	154.49
30	0.6	20	126.33	151.18
60	0.6	15	129.08	151.90

Table 3: Random Forest Tuning

E Code Base

We have included below sample excerpts of code that was used to generate our features and models.

```
=================================================================
PCA & STANDARDIZATION

artist_matrix = train2.pivot(index=’artist’, columns=’user’, values=’plays’)
user_matrix = train2.pivot(index=’user’, columns=’artist’, values=’plays’)

# you’ll want to fill in the nan’s with 0’s too:
train_matrix = train_matrix.fillna(value=0)

# standardize first for PCA
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.decomposition import RandomizedPCA

std_scale = preprocessing.StandardScaler().fit(train_matrix)
train_matrix_std = std_scale.transform(train_matrix)

# alternatively, can combine these 2 steps into fit_transform
def pcaTrainAndPredict(n_components, train_matrix, test_matrix):
    # this message is relevant if passing in artist_matrix, reverse labels if using user_matrix
    print "Extracting the top %d users from %d artists" % (n_components, train_matrix.shape[0])
    pca = RandomizedPCA(n_components=n_components, whiten=True).fit(train_matrix)
    X_train_pca = pca.transform(train_matrix)
    X_test_pca = pca.transform(test_matrix)

    return X_train_pca, X_test_pca

=================================================================
CROSS VALIDATION & MODEL FITTING

def cv_optimize(model, parameters, train_x, train_y, n_folds=5, score_func=None):
   """
   Function
   --------
   cv_optimize

   Inputs
   ------
   model: an instance of a scikit-learn classifier
   parameters: a parameter grid dictionary that is passed to GridSearchCV
   train_x: a samples-features matrix in the scikit-learn style
   train_y: the response vectors of 1s and 0s (+ives and -ives)
   n_folds: the number of cross-validation folds (default 5)
   score_func: a score function we might want to pass (default python None)

   Returns
   -------
   The best estimator from the GridSearchCV, after the GridSearchCV has been used to fit the model.

   Notes
   -----
   see do_classify and the code below for an example of how this is used
   """

    if score_func is not None:
       gs = GridSearchCV(model, param_grid=parameters, cv=n_folds, scoring=score_func)
   else:
       gs = GridSearchCV(model, param_grid=parameters, cv=n_folds)
       gs.fit(train_x, train_y)

   print "BEST", gs.best_params_, gs.best_score_, gs.grid_scores_
   best = gs.best_estimator_

   return best



def fit_model(model, train_x, train_y, test_x=None, test_y=None, title=None, tracker=None):
    """
    Generic function that is used to fit a model and compute out-of-sample accuracy.

    Inputs
    ----------
    model: Any Scikit Learn predictive model (with or without custom parameters)
    train_x: Dataframe containing training set X-variables
    train_y: Dataframe containing training set Y-variables
    train_x: Dataframe containing test set X-variables
    train_y: Dataframe containing test set Y-variables

    title: title to use as key in tracker
    tracker: dictionary to store results
    """
    # fit model
    model.fit(train_x, train_y)

    # calculate training set accuracy
    train_acc = accuracy_score(train_y, model.predict(train_x))

    if tracker is not None:
        tracker[title] = [model, train_acc]

    # calculate test set accuracy
    if test_x is not None:
    test_acc = accuracy_score(test_y, model.predict(test_x))

    print ’############’
    print model
    print ’----------’
    print ’Training set accuracy = %0.4f’ % train_acc
    print ’Validation set accuracy = %0.4f’ % test_acc
    print ’----------’
    print ’############’

    if tracker:
        tracker[title].append(test_acc)

    else:
        print ’############’
        print model
        print ’----------’
        print ’Training set accuracy = %0.4f’ % train_acc
        print ’----------’
        print ’############’

    # return fitted model and training/test accuracy
    return model
```

```
=================================================================
CROSS-VALIDATION LOOP FOR K-MEANS CLUSTERING

### Cross validation parameters
ulist = [4] # [50, 100, 200] # [4, 6, 8, 10, 15, 20, 25, 30, 50] [60, 75]
```

```
alist = [3] # [10, 20, 50, 100] # [50, 100, 200, 500] # [3, 4, 5, 6, 7, 8, 9, 10, 15, 20]

for u in ulist:
    unew = True
    for a in alist:
        anew = True

        # users
        if unew:
            unew = False
            print "clustering users"; sys.stdout.flush()
            km = KMeans(n_clusters=u, init=’k-means++’, n_init=10, max_iter=50000, tol=1e-05, random_state=0)
            y_km = km.fit_predict(new_matrix)
            classdict = {}
            for i in xrange(U):
                classdict[dfprofs.user[i]] = y_km[i]

        # artists
        if anew:
            anew = False
            print "clustering artists"; sys.stdout.flush()
            km = KMeans(n_clusters=a, init=’k-means++’, n_init=10, max_iter=50000, tol=1e-05, random_state=0)
            y_km = km.fit_predict(new_artmatrix)
            artdict = {}
            for i in xrange(V):
                artdict[dfarts.artist[i]] = y_km[i]

        # retrain
        print "retraining on training data"; sys.stdout.flush()
        lookup = {}

        for i in xrange(u):
            for j in xrange(a):
                lookup[(i, j)] = list([])

        for i in xrange(N_train):
            lookup[(classdict[train.user[i]], artdict[train.artist[i]])] += list([train.plays[i]])

        for key in lookup.keys():
            if len(lookup[key])==0:
                # lookup[key] = 118.0  # impute global median
                lookup[key] = medplays[key[0]]  # impute user median
            else:
                lookup[key] = np.median(lookup[key])

        # revalidate
        print "validating"; sys.stdout.flush()
        N_valid = valid.shape[0]
        preds = np.zeros(N_valid)

        for i in xrange(N_valid):
            preds[i] = lookup[(classdict[valid.user[i]], artdict[valid.artist[i]])]

        actual = np.array(valid.plays)
        MAE = np.mean(np.abs(preds - actual))
        maedict2[(u, a)] = MAE

        print "user clusters: %i, artist clusters: %i, MAE: %.2f" % (u, a, MAE)
        sys.stdout.flush()

=================================================================
FEATURE SCRAPING & PARSING

# artist data: {id, name}
artist_data = load_artist_data(’data/artists.csv’)
```

```
idx = artist_data.keys()[0]
num_artists = len(artist_data)
print ’Sample artist data:’, idx, artist_data[idx]
print ’Number of artists:’, num_artists

artist_list = artist_data.keys()

# scrape data from tags page

timer = time.time()
artist_tags_raw = dict()

for artist in artist_list:

    # overview page
    req = requests.get(’https://musicbrainz.org/artist/’ + artist + ’/tags’)
    soup = BeautifulSoup(req.text, ’html.parser’)
    artist_tags_raw[artist] = soup
    time.sleep(1)

print ’Runtime: %0.2f seconds’ % (time.time() - timer)

# parse scraped data

timer = time.time()
artist_tags_parsed = dict()

for artist in artist_list:

    content = artist_tags_raw[artist].find(’div’, attrs={’id’: ’content’})
    try:
        tags = content.find(’div’, attrs={’id’: ’all-tags’}).find_all(’a’)
        result = [t.get_text() for t in tags]
        artist_tags_parsed[artist] = result
    except AttributeError:
        artist_tags_parsed[artist] = [None]

print ’Runtime: %0.2f seconds’ % (time.time() - timer)

# save parsed data
fd = open(’data/artist_tags.json’, ’w’)
json.dump(artist_tags_parsed, fd)
fd.close()

=================================================================
FEATURE GENERATION

# training data
df_train = load_training_data(’data/train.csv’)
num_train = len(df_train)
’Number of users in training data:’, num_train

# user data
df_user = load_user_data(’data/profiles.csv’)
num_users = len(df_user)
’Number of users:’, num_users

# convert to categorical variables
df_user[’user_sex_m’] = np.array(df_user[’user_sex’].values == ’m’, dtype=np.int)
del df_user[’user_sex’]

countries = list(set(df_user[’user_country’].values))

# create country indicator variables
for c in countries:
    df_user[’user_country_’ + str(c)] = np.array((df_user[’user_country’].values == str(c)), dtype=np.int)
del df_user[’user_country’]

# drop columns with too few users
for c in countries:
    if np.sum(df_user[’user_country_’ + str(c)].values) <= 1000:
        del df_user[’user_country_’ + str(c)]

# artist data
df_artist = load_artist_data(’data/artists.csv’)
num_artists = len(df_artist)
’Number of artists:’, num_artists

artist_list = df_artist[’artist_id’].values
len(artist_list)

# reload parsed data
with open(’data/artist_tags.json’, ’r’) as fd:
    artist_tags_parsed = json.load(fd)

# flatten list to identify unique genres
tags = sorted(list(set(list(itertools.chain.from_iterable(artist_tags_parsed.values())))))
len(tags)

# add genres to pandas dataframe
for t in tags:
    df_artist[’genre_’ + str(t)] = np.zeros(num_artists, dtype=np.int)

# fill in for each artist
for artist in artist_list:
    for t in artist_tags_parsed[artist]:
        df_artist.loc[df_artist.artist_id==artist, ’genre_’ + str(t)] = 1

# drop columns with too few artists

for t in tags:
    if str(t) == ’None’:
        del df_artist[’genre_’ + str(t)]
    elif np.sum(df_artist[’genre_’ + str(t)].values) <= 20:
        del df_artist[’genre_’ + str(t)]

# add artist features
df_train = df_train.merge(df_artist, left_on=’artist_id’, right_on=’artist_id’, how=’left’,
suffixes=(’’, ’_’))

# add user features
df_train = df_train.merge(df_user, left_on=’user_id’, right_on=’user_id’, how=’left’,
suffixes=(’’, ’_’))

# calculate user and artist medians
user_median = df_train[[’user_id’, ’plays’]].groupby(’user_id’).median()
user_median.reset_index(inplace=True)
user_median.columns = [’user_id’, ’user_median_plays’]
artist_median = df_train[[’artist_id’, ’plays’]].groupby(’artist_id’).median()
artist_median.reset_index(inplace=True)
artist_median.columns = [’artist_id’, ’artist_median_plays’]

# add to dataframe
df_train = df_train.merge(user_median, left_on=’user_id’, right_on=’user_id’, how=’left’,
suffixes=(’’, ’_’))
df_train = df_train.merge(artist_median, left_on=’artist_id’, right_on=’artist_id’, how=’left’,
suffixes=(’’, ’_’))

# remove non-features
del df_train[’artist_id’]
del df_train['artist_name']
del df_train['user_id']

# extract response variable
resp = df_train.plays.values
del df_train['plays']

# training data
df_test = load_test_data('data/test.csv')
num_test = len(df_test)
'Number of users in test data:', num_test

# add artist and user features
df_test = df_test.merge(df_artist, left_on='artist_id', right_on='artist_id', how='left',
suffixes=('', '_'))
df_test = df_test.merge(df_user, left_on='user_id', right_on='user_id', how='left',
suffixes=('', '_'))

# add to dataframe
df_test = df_test.merge(user_median, left_on='user_id', right_on='user_id', how='left',
suffixes=('', '_'))
df_test = df_test.merge(artist_median, left_on='artist_id', right_on='artist_id', how='left',
suffixes=('', '_'))

# remove non-features
del df_test['artist_id']
del df_test['artist_name']
del df_test['user_id']

# extract id for kaggle submission
idx = df_test.idx.values
del df_test['idx']

# avoid out-of memory errors
df_user = None
df_artist = None
user_median = None
artist_median = None

# check that columns line up
np.all(df_train.columns.values == df_test.columns.values)

=================================================================
REGRESSION MODELS

# random forest regressor
# note: no train-validation split for practical reasons
model = RandomForestRegressor(n_estimators=100)
model.fit(df_train, resp)
model_pred = model.predict(df_train)
model_err = np.sum(np.abs(model_pred - resp)) / num_train
model_pred_test = model.predict(df_test)
write_predictions('data/test_RF.csv', idx, model_pred_test)
joblib.dump(model, 'data/model_RF.pkl');
'Training error: %0.2f' % model_err

=================================================================

COLLABORATIVE FILTERING ARTIST-BASED NEIGBHBORHOOD MODEL

def pearson_sim(artist1_plays, artist2_plays, n_common, min_common=2):
    if n_common < min_common:
    return 0

    artist1_adj_plays = artist1_plays.apply(lambda row: row['plays'] - row['user_median'], axis=1).values
    artist2_adj_plays = artist2_plays.apply(lambda row: row[’plays’] - row[’user_median’], axis=1).values

    if len(artist1_adj_plays) != len(artist2_adj_plays):
        raise "ERROR! Not same number of plays for artist1 and artist2"

    rho, pvalue = pearsonr(artist1_adj_plays, artist2_adj_plays)

    if math.isnan(rho):
        return 0

    return rho

def get_artists_plays(artist_id, df, set_of_users):
   """
   given an artist id and a set of user ids, return the sub-dataframe of their
   plays.
   """
   mask = (df.user.isin(set_of_users)) & (df.artist==artist_id)
   plays = df[mask]
   # remove any users that are duplicates
   plays = plays[plays.user.duplicated()==False]
   return plays

def calculate_similarity(df, artist1, artist2, supports, similarity_func):
    # find common users
    common_users = supports[a1][a2]

    n_common=len(common_users)
    if artist1==artist2:
        return 1., n_common

    #get plays - ensure they’re in the same order
    artist1_plays = get_artists_plays(artist1, df, common_users).sort_values(by=’user’)
    artist2_plays = get_artists_plays(artist2, df, common_users).sort_values(by=’user’)

    sim = similarity_func(artist1_plays, artist2_plays, n_common)
    return sim, n_common

def knearest(artist_id, dict_of_artists_to_plays, corr_matrix, common_support, k, reg):
    """
    Given a artist_id, get a sorted list of the
    k most similar artists from the set of artists.
    """
    num_artists = len(dict_of_artists_to_plays)
    if num_artists < 4:
        k = num_artists
    else:
        k = num_artists//3

    similars=[]
    for other_artist_id in dict_of_artists_to_plays.keys():
        if other_artist_id == artist_id:
            continue

        if other_artist_id != artist_id:
            sim = corr_matrix[artist_id][other_artist_id]
            plays = dict_of_artists_to_plays[other_artist_id]

        simdist = (1. - sim)/2.
        similars.append((plays, simdist))

        # sort by sim distance distance
        similars = sorted(similars, key=itemgetter(1))
        return similars[0:k]
```

```
def predict_plays(dict_of_artists_to_plays, train_df, artist_id, user_id, corr_matrix, common_support, k, reg):

    knear = knearest(artist_id, dict_of_artists_to_plays, corr_matrix, common_support, k=k, reg=reg)

    nearest = []
    for other_artist in knear:
        plays, simdist = other_artist
        nearest.append(plays)

    return np.median(nearest)

def predict(test_df, train_df, corr_matrix, common_support, k=2, reg=3):
    zips = zip(test_df.user, test_df.artist)
    preds = []

    for uid, aid in zips:
        dfuser = train_df[train_df.user==uid]
        artist_to_plays = dict(zip(dfuser.artist.values, dfuser.plays.values))

    predicted_plays = predict_plays(artist_to_plays, train_df, aid, uid, corr_matrix, common_support, k, reg)
    preds.append(predicted_plays)

    return np.array(preds)

=================================================================

LINEAR REGRESSION PREPROCESSING

y_mini = train_mini[’plays’].values
x_mini = train_mini[[’global_median’, ’user_median’, ’artist_median’]].values
user_ids =  train_mini.user.values
users_npa = np.array(users.user)
users_npa = dict(zip(users_npa, range(len(users_npa))))

u_tc = np.zeros([len(x_mini), len(users_npa)])

# loop through user_ids in training datafaame
for i in range(len(user_ids)):
    # find index of userid in full data frame
    uidx = users_npa[user_ids[i]]
    # update indicator in matrix
    u_tc[i][uidx] = 1

x_mini_users = np.concatenate((x_mini, u_tc), axis=1)

# repeat same process for artists

x_train, x_test, y_train, y_test = train_test_split(x_mini_artists, y_mini, train_size = 0.8)

=================================================================

MATRIX FACTORIZATION

train_matrix = train2.pivot(index=’artist’, columns=’user’, values=’plays’)

from sklearn.decomposition import NMF

model = NMF(beta=0.002, init=’nndsvd’, max_iter=5000, random_state=0,  alpha=0.0002, tol=0.001,
sparseness=None)

W = model.fit_transform(train_matrix)
H = model.components_
nR = np.dot(W, H)
```