Overview

The purpose of this report is to document and benchmark different classes of recommendation system algorithms our team applied with the objective of improving LastFMs current recommendation system (non-personalized top 10 recommender). Our ultimate goal with this project is to replace LastFMs simplistic recommender system with a more sophisticated model that provides diverse personalized recommendations that will significantly boost the quality of UX, subsequently improving LastFMs product. The recommendation techniques that will be explored and benchmarked include:

- Collaborative Filtering
- Content-Based Recommendation
- Hybrid Recommenders

Collaborative Filtering Models Findings Report

Collaborative filtering hinges on the idea that we can predict a users preferences based on the preferences of users as well as predict the rating a user would giveitems based on their interactions with other items. It uses matrices of rating scores indexed by users and items respectively, and then computes a similarity measure: `cosine similarity` or `pearson correlation`, between vectors of ratings. The largest drawback to collaborative filtering models is that if they lack information for a new product or user, this poses the cold-start issue. This means predictions cannot be made as they would for users and items with more complete information on interaction.

Data Preprocessing for `Surprise`

In terms of data wrangling, there was a minimal amount of manipulation required in order to produce a dataframe which would be compatible with classes from the Surprise library. Surprise requires data to follow a three-column structure of user ID, item id (in this case `artistID`), and a rating. As the `user_artist.dat` file already complied with this requirement no columns were dropped.

However, instead of a discrete rating column, the `user_artist` data contains a continuous feature `weight` which indicates the number of songs played per artist for each unique user. A little visualization demonstrated a large amount of skew in this data:

We can see in the figure above how left-skewed the `weight` data is. However, due to the scale of the horizontal axis, which represents frequency, the right-tail distribution caused by the skew is obfuscated from the chart. In fact the maximum `weight` is actually `352698` plays, but only has a frequency of one, therefore does not appear in this plot.

Having a continuous measure like `weight` is not compatible with `Surprise`, which requires some kind of discrete rating scale such as 1-5. Therefore, the next step was too discretize or categorize the `weight` column to make it conform to a scale. A 1-5 rating scale was used as the distribution of `weight` could be equally

divided in five quantiles. The code chunk below demonstrates how this discretization was achieved using the `qcut` function from `Pandas`.

```
#discretize weights using qcuts
user_artists_df['weight_quantiles'] = pd.qcut(user_artists_df['weight'],
                            q=[0,.2,.4,.6,.8,1],
                            labels=False,
                            precision=0)
```

Upon categorizing the `weight` and renaming it `weight_quantiles` the observations are almost equally distributed within these categories the exact values counts can be seen below:

- 0 18770
- 3 18581
- 4 18548
- 2 18469
- 1 18466

`Train`/`Test` Split + Initializing Surprise Objects

Next we perform a traditional train/test split of the `user_artists` data with a test-set size of `0.3`. Next is a crucial step when working with the `Surprise` library. The `Reader` class is used to parse our brand new ratings (`weight_quantiles`) based on given scale (1-5 of course). Secondly, a full dataset object is created for subsequent cross-validation as well as `Surprise`-specific trainset and testset objects. The code for this can be seen below:

```
#create reader object
reader = surprise.Reader(rating_scale=(1,5)) #1:5 scale

#create surprise train and test set objects
data =
surprise.Dataset.load_from_df(UA_df_cf[["userID","artistID","weight_quantiles"]],
reader)
UA_train = surprise.Dataset.load_from_df(UA_train, reader).build_full_trainset()
UA_test = list(UA_test.itertuples(index=False, name=None))
```

Baseline Predictions

The next step is to explore what collaborative filtering algorithms or models from `Surprise` package give us a decent baseline or benchmark estimation by evaluating predictions based on the following metrics:

- `RMSE`
- `MAE`
- `NDCG`
- `F1-Score`

User-based KNNBasic:

```
#get baseline KNN score
from surprise import KNNBasic

# create options dict; use cosine similarity on user_based data
options = {'name':'cosine', 'user_based':True}

ubKNN = KNNBasic(k=20, min_k=2, sim_options=options, random_state=123)

#create cosine similarity matrix
ubKNN.fit(UA_train)\
.compute_similarities()
```

```
Impossible: 0.2671
```

Impossibility here indicates the porportion of the predictions that weren't possible due to cold-start issues (incomplete data). This is an important caveat considering this is one of the biggest weaknesses of collaborative filtering methods.

Baseline Results:

Item-based KNNBasic:

```
#get baseline KNN score
from surprise import KNNBasic

# create options dict; use cosine similarity on user_based data
options = {'name':'cosine', 'user_based':False}

ibKNN = KNNBasic(k=20, min_k=2, sim_options=options, random_state=123)

#create cosine similarity matrix
ibKNN.fit(UA_train)\
.compute_similarities()
```

```
Impossible: 0.2012
```

Baseline Results:

BaselineOnly (ALS)

```
from surprise import BaselineOnly

#alternating least squares (ALS) with 30 iterations
options = {"method": "als", "n_epochs": 30}
als = BaselineOnly(bsl_options=options)

# fit on training set
als.fit(UA_train)

als_preds = als.test(UA_test)
als_accuracy = accuracy.rmse(als_preds)
```

```
Impossible: 0.0000
```

Baseline Results:

 image

SVD (Matrix Factorization

```
from surprise import SVD

# 20 factors non-bias
svd = SVD(n_factors=20, biased=True, random_state=42)

# fit on training set
svd.fit(UA_train)

svd_preds = svd.test(UA_test)
svd_accuracy = accuracy.rmse(svd_preds)
```

```
Impossible: 0.0000
```

Baseline Results:

 image

CoClustering

```
from surprise import CoClustering

clust = CoClustering(n_cltr_u=10, n_cltr_i=10, n_epochs=50, random_state=42)

clust.fit(UA_train)

coclust_preds = clust.test(UA_test)
accuracy.rmse(coclust_preds)
```

```
Impossible: 0.0000
```

Baseline Results:

Baseline Consensus:

By comparing evaluation metrics we can see there's a clear winner in terms of baseline perfomance, `SVD`. Which trumps all other models across all evaluation metrics besides `recall`.

Hyperparameter Tuning

Given we've identified which model works best with our data. Next we perform a cross-validated grid search to find which hyperparameter combinations produce the best results. The grid search we ran can be seen in the code chunk below.

```
grid search cross validation

param_grid = {'n_factors':[20,50,150,200],
              #'n_epochs':[20,50],
              #'lr_all':[0.005, 0.001, 0.002],
              #'biased':[True],
              #'reg_all':[0.01,0.02]}

3-fold grid searched cv
SVD_cv_grid = GridSearchCV(SVD,
                           param_grid=param_grid,
                           measures=['rmse','mae'],
                           cv=3)
SVD_cv_grid.fit(data)
```

Final Model `train`, `predict`, and `evaluate`

```
# with tuned hyperparameters
svd_best = SVD(n_factors=20, n_epochs=20,lr_all=0.005, biased=True, reg_all=0.01,
random_state=42)

# fit on training set
svd_best.fit(UA_train)
```

We can see from the above evaluation metrics we have indeed lowered the `RMSE` of the baseline predictions by a very small relatively negligible margin. Through this benchmarking evaluation process we have select the `SVD` model as the most effective collaborative filtering model for this problem.

Content Based Recommendation System

The matrix for Content based recommendation systems is created using two data frames: user_taggedartists.dat and tags.dat. We further discuss steps executed on these datasets in order to create the content matrix.

Data preprocessing and content matrix for content-based recommendation system

To being, we create a full-date variable column on user_taggedartists.dat. We can see that dates when users tagged artists are mostly frm 2000 and on.

 

We then proceed to create qualitative variables for the same data frame, and a recency variable column later, by categorizing dates artists were tagged by a user as "Very Old" if tagged before January 1970, "Old" if tagged before Jaunary 1984, "New" if tagged before January 2010 and "Very New" from January 2010 and further.

We then merge the two data frames, one containing a tag value of each tag ID, and the other one containg UserID, Artist ID and Recency value.

To create content matrix, we pivot the merged table twice in order to turn categorical variables in dummy variables. Then, two pivoted tables merged together are combined into a content matrix, with ArtistID as an index and Genre and Recency as variables. The resulting matrix is:

```
cb.head()
```

Applying content based model

We initiate the model at NN = 10, filtering the matrix for 10 nearest neighbors with non-negative similarity. Then, we fit on content using content matrix, and on ratings using train dataset.

```
# init content-based
cb_mod = ContentBased(NN=10)

# fit on content
cb_mod.fit(cb)

# fit on train_ratings
cb_mod.fit_ratings(UA_train)

cb_pred = cb_mod.test(UA_test)
```

We get the following evaluation results:

```
# compute metrics for CB RS
cb_res = eval.evaluate(cb_pred, topn=5, rating_cutoff=3.5).rename(columns=
{'value':'Content_based_10'})
cb_res
```

- RMSE 0.891378
- MAE 0.669223
- Recall 0.386871
- Precision 0.760339
- F1 0.512814
- NDCG@5 0.872620

Hybrid Recommender Systems

Hybrid recommendation systems are not a fully independent class of algorithms. Rather they blend methods together in an effort to mitigate the drawbacks of individual techniques. In our case, we've experimented with a hybrid of our two best performing models: The SVD model fit with tuned hyperparameters and the content based model seen in the previous section. We also implemented a`Random Forest` hybrid model which is trained on the predictions of both the content based and SVD models and as well as the actual rating values as a target variable and then makes predictions of its own. However, since hybrid systems aggregate different techniques they can inherit the drawbacks of the models used as well.

SVD and Content Based Hybrid (Weighting)

We applied a weighting method to prioritize the ratings of the better-performing model (content based), when aggregating the predicted ratings. No additional preprocessing was required for this. The code chunk below demonstrates how the weighting technique takes `0.6` (60%) of a content based prediction and then adds the remaining `0.4` (40%) to create a new hybrid prediction, can be seen below:

```
#Combine predictions (weights)

#extract predictions content-based and item-based
df_pred_cb, df_pred_svd = pd.DataFrame(cb_pred), pd.DataFrame(svd_preds)

df_hybrid = df_pred_cb.copy()
df_hybrid['est'] = (np.array(df_pred_cb['est'])*0.6) +
(np.array(df_pred_svd['est'])*0.4)
```

Model Performance:

Below we can see that ultimately this results in a lower `RMSE` than either of the individuals models produced of `0.871708`. However, we see some trade off for some of the other metrics such as `MAE`,`Recall`, and `NDCG@5`, which did not perform better than the individual models with respect to each indivual metric that was just enumerated.

Random Forest Hybrid

Data Preprocesing:

The first step here is creating a new train test split to evaluate this models performance. Next we create a new data frame that contains three columns. A target column with the actual labels for each oberservation as well as a column for the predictions of both the SVD model and the content based model respectively. Finally, these are split into a df `X` which contains the features, and a vector `y` which contains the labels. The code for this can be seen in the code chunk below:

```
#Random Forest Hybrid
df_cb_train, df_cb_test = df_pred_cb[:19500], df_pred_cb[19500:]
df_svd_train, df_svd_test = df_pred_svd[:19500], df_pred_svd[19500:]

rf_data = df_cb_train[['r_ui', 'est']].rename(columns={'r_ui':'target',
'est':'cb_pred'})
rf_data['svd_pred'] = df_svd_train['est']

rf_test = df_cb_test[['r_ui', 'est']].rename(columns={'r_ui':'target',
'est':'cb_pred'})
rf_test['svd_pred'] = df_svd_test['est']

from sklearn.ensemble import RandomForestRegressor

X, y = rf_data.loc[:,rf_data.columns != 'target'], np.array(rf_data['target'])
X_test = rf_test.loc[:,rf_test.columns != 'target']

#fit random forest model
rf_model = RandomForestRegressor(max_depth=4, n_estimators=100).fit(X,y)

#predict
rf_pred = rf_model.predict(X_test)

# transform in surprise format
df_rf = df_cb_test.copy()
df_rf['est'] = rf_pred
```

Conclusion

After trying different recommendation systems to the LastFM dataset, evaluating and comapring them, we ended up going with the hybrid system as we believe that it would be a better solution since it will take both the tags of the artists and similaritites of these artists and users into account. Hence, it could be a good solutions to problems such as overspecialization, providing users with more novelty and accurate predictions.

As an enhancement to the LastFM's non-personalized recommendation system, and with the ubiquity of data collection, we saw that different recommandation strategies can be applied based on the context of each

experience such as the webpage to deploy it on, as well as the user's recency.

To do so, we thought about categorizing our users into 3 different types:

- First-time visitor: for this type of users, that we don't have any historical data of their musical taste and preferences, we can display `Top artists` on the homepage. The system will give more weight to recent ratings over historical ones and updates scores every time a data feed is synchronized (*non-personalised method*).
- Seasonal visitor: for users that don't have frequent activities on the plateform, we may based our recommendations to them on more-specific strategy which is `Most Popular in Category` strategy, that not only promote the most popular artists but includes artist from music genres that he already liked and music genres that are similar to them (*item-based method*).
- Frequent visitor: this category, will have the `most personnalised recommendation` since we already collected an important amount of data about their behaviours. The system that we can deploy will surface suggestions of artists for our users based on the user's persona similarity with other users, the similaritties of artists in addition to the genres and release date affinity (*hybrid method*)