1000_playlist_content_filtering_colab

December 12, 2018

1 Content filtering using cosine similarity of tracks

The following notebook illustrates our content filtering approach that uses track similarity (measured by cosine distance) to recommend tracks to playlists.

Cosine similarity measures the orientation of two n-dimensional sample vectors irrespective to their magnitude. It is calculated by the dot product of two numeric vectors, and it is normalized by the product of the vector lengths. The output value ranges from 0 to 1, with 1 as the highest similarity.

We compute a similarity matrix for tracks by using sklearn pairwise distance method, with cosine similairty:

$cos(\boldsymbol{t}rack_1, \boldsymbol{t}rack_2) = \frac{\boldsymbol{t}rack_1 \cdot \boldsymbol{t}rack_2}{||\boldsymbol{t}rack_1|| \cdot ||\boldsymbol{t}rack_2||}$

```
In [1]: from google.colab import drive
        drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/co
```

```
In [0]: import pandas as pd
        import numpy as np
        from sklearn.model_selection import train_test_split
        from sklearn.neighbors import NearestNeighbors
        from sklearn.utils import shuffle

        subset100 = pd.read_csv("gdrive/My Drive/track_meta_100subset_new.csv")
        subset100 = shuffle(subset100)
```

1.1 1. Data Processing

1.1.1 1.1 Train-test split

We split the data into training and test set by 80-20.

```
In [0]: train, test = train_test_split(subset100, test_size=0.2, random_state=42, stratify = su
        # train, val = train_test_split(train, test_size=0.2, random_state=42, stratify = trai
```

1.1.2 1.2 Data cleaning

We drop some non-numeric features in order to calculate the cosine similarity matrix.

In [0]:
```
# Drop features here
features_drop = ["Playlistid","Playlist","Album", "Track", "Artist", "Trackid", "Artis
train_cleaned, test_cleaned = train.drop(features_drop, axis =1), test.drop(features_d
train = train.reset_index(drop=True)
train_cleaned = train_cleaned.reset_index(drop=True)
```

In [5]:
```
train_cleaned.head()
```

Out[5]:
```
   Track_Duration  acousticness  artist_popularity  danceability  energy  \
0          363521        0.4030                 88         0.712   0.838
1          268004        0.1750                 78         0.814   0.779
2          205040        0.1510                 68         0.324   0.776
3          205733        0.0603                 80         0.687   0.793
4          185306        0.0178                 67         0.549   0.981

   instrumentalness  key  liveness  loudness  mode  speechiness    tempo  \
0          0.000000    7    0.8090    -2.679     1       0.3130  148.138
1          0.000671   11    0.0605    -3.271     1       0.2350   93.430
2          0.917000    0    0.0728    -6.784     1       0.0346  101.964
3          0.000000    2    0.5820    -4.254     1       0.1660  107.045
4          0.000002   11    0.4380    -3.558     0       0.0590   82.331

   time_signature  valence
0               4    0.607
1               4    0.544
2               3    0.317
3               4    0.751
4               4    0.873
```

1.1.3 1.3 Create a cosine-similarity matrix

In [6]:
```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import MinMaxScaler

# Standardize the data
scaler = MinMaxScaler()
scaler.fit(train_cleaned)
train_scaled = scaler.transform(train_cleaned)
test_scaled = scaler.transform(test_cleaned)

train_scaled_cos_matrix = cosine_similarity(train_scaled)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/preprocessing/data.py:323: DataConversionWarnin
  return self.partial_fit(X, y)
```

The shape of the cosine matrix shows 1970 unique tracks (in 100 playlists) in the training set.

```
In [7]: train_scaled_cos_matrix.shape
```

```
Out[7]: (2463, 2463)
```

We wrote a function to compute prediction set per playlist.

The function takes in a pre-calculated track cosine similarity matrix, training set, the target playlist id and the prediction set size (which we pre-determine it to be test set size * 15). It returns a list of tracks (prediction list) to recommend per playlist. The prediction list contains top k similar songs (based on cosine similarity) per track in the playlist.

```
In [0]: def cos_similar_songs_playlist(cos_matrix, orig_df, target_playlist_id, cand_list_size
            """
            Input:
            cos_matrix: cosine matrix of the tracks
            orig_df: original df with tracks as rows, but with playlistid and other features (
            target_playlist_id: id of the target playlist
            target_playlist_inx: index of playlist in the training set
            cand_list_size: candidate list of songs to recommend size (= test-set size * 15)

            Output:
            k_song_to_recommend: the most similar tracks per track
            """
            target_track_inx = np.where(train["Playlistid"] == target_playlist_id)[0] # index 
            candidate_cos_matrix = cos_matrix

            ## For each song in the playlist, find k similar songs
            cand_list = []
            # cand_list_size = k*15
            k = np.floor(cand_list_size/len(target_track_inx)) # round(cand_list_size/len(targ
            k_rest = cand_list_size - k*len(target_track_inx)
            # e.g., for a candidate list size of 30, get 3 songs for each track first
            for inx, i in enumerate(target_track_inx):
                candidate_song_rec = candidate_cos_matrix[i, ] #ith row of matrix
                candidate_song_rec_inx = np.argsort(candidate_song_rec)
                unique_candidate_song_sorted = train['Track_uri'][candidate_song_rec_inx][::-1
                tracks_in_target_playlist = train.loc[train["Playlistid"] == target_playlist_i
                song_to_recommend = np.array(unique_candidate_song_sorted.loc[~unique_candidat

                if (k_rest != 0 & inx <= k_rest): # 30-24 = 6; for the first 6 tracks recommen
                    k_song_to_recommend = song_to_recommend[:int(k+1)]
                else:
                    k_song_to_recommend = song_to_recommend[:int(k)]

                if inx == 0:
                    cand_list = k_song_to_recommend
                else:
                    cand_list = np.append(cand_list, k_song_to_recommend)
            return list(cand_list) # turn np array into list
```

1.2 2. Model Performance

1.2.1 2.1 Metrics

In [0]:
```
def nholdout(playlist_id, df):
    '''Pass in a playlist id to get number of songs held out in val/test set'''

    return len(df[df.Playlistid == playlist_id].Track_uri)
```

In [0]:
```
def r_precision(prediction, val_set):
# prediction should be a list of predictions
# val_set should be pandas Series of ground truths
    score = np.sum(val_set.isin(prediction))/val_set.shape[0]
    return score
```

In [0]:
```
### NDCG Code Source: https://gist.github.com/bwhite/3726239
def dcg_at_k(r, k, method=0):
    r = np.asfarray(r)[:k]
    if r.size:
        if method == 0:
            return r[0] + np.sum(r[1:] / np.log2(np.arange(2, r.size + 1)))
        elif method == 1:
            return np.sum(r / np.log2(np.arange(2, r.size + 2)))
        else:
            raise ValueError('method must be 0 or 1.')
    return 0.


def ndcg_at_k(r, k, method=0):
    dcg_max = dcg_at_k(sorted(r, reverse=True), k, method)
    if not dcg_max:
        return 0.
    return dcg_at_k(r, k, method) / dcg_max
```

1.2.2 2.2 Model Test-Set Performance on 100 playlists

In [0]:
```
unique_playlistid = train['Playlistid'].drop_duplicates()
```

In [0]:
```
rps = []
ndcgs = []
for pid in unique_playlistid: # loop through each playlist
#     print(pid)
    ps = cos_similar_songs_playlist(train_scaled_cos_matrix, train, pid, nholdout(pid,
    vs = test[test.Playlistid == pid].Track_uri # ground truth

#     print(r_precision(ps, vs))
    rps.append(r_precision(ps, vs)) # append individual r-precision score

    # NDCG
```

```
            r = np.zeros(len(ps))
            for i, p in enumerate(ps):
                if np.any(vs.isin([p])):
                    r[i] = 1
            ndcgs.append(ndcg_at_k(r, len(r)))
```

```
In [14]: avg_rp = np.mean(rps)
         avg_ndcg = np.mean(ndcgs)
         print('Avg. R-Precision: ', avg_rp)
         print('Avg. NDCG: ', avg_ndcg)
         print('Total Sum: ', np.mean([avg_rp, avg_ndcg]))
```

```
Avg. R-Precision:  0.04585610327638191
Avg. NDCG:  0.051839478084263604
Total Sum:  0.04884779068032276
```

1.2.3 2.3 Model Performance on 1000 playlists

```
In [0]: subset1k_seed = pd.read_csv("gdrive/My Drive/track_meta_milestone3.csv", index_col="Unn
        np.random.seed(123)
```

```
In [0]: subset1k_id = np.random.choice(subset1k_seed['Playlistid'].unique(), size = 1000, repla
        subset1k = subset1k_seed[subset1k_seed['Playlistid'].isin(subset1k_id)]
```

2.3.1 Data Processing on 1k playlists

```
In [17]: train, test = train_test_split(subset1k, test_size=0.2, random_state=42, stratify = su
         # Drop features here
         features_drop = ["Playlistid","Playlist","Album", "Track", "Artist", "Trackid", "Artis
         train_cleaned, test_cleaned = train.drop(features_drop, axis =1), test.drop(features_d
         train = train.reset_index(drop=True)
         train_cleaned = train_cleaned.reset_index(drop=True)

         # Standardize the data
         scaler = MinMaxScaler()
         scaler.fit(train_cleaned)
         train_scaled = scaler.transform(train_cleaned)
         test_scaled = scaler.transform(test_cleaned)

         train_scaled_cos_matrix = cosine_similarity(train_scaled)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/preprocessing/data.py:323: DataConversionWarning
  return self.partial_fit(X, y)
```

```
In [19]: subset1k.shape
```

```
Out[19]: (34825, 28)
```

```
In [0]: unique_1000playlistid = train['Playlistid'].drop_duplicates()
```

```
In [0]: rps = []
        ndcgs = []
        for pid in unique_1000playlistid: # loop through each playlist
        #     print(pid)
            ps = cos_similar_songs_playlist(train_scaled_cos_matrix, train, pid, nholdout(pid,
            vs = test[test.Playlistid == pid].Track_uri # ground truth

        #     print(r_precision(ps, vs))
            rps.append(r_precision(ps, vs)) # append individual r-precision score

            # NDCG
            r = np.zeros(len(ps))
            for i, p in enumerate(ps):
                if np.any(vs.isin([p])):
                    r[i] = 1
            ndcgs.append(ndcg_at_k(r, len(r)))
```

```
In [22]: avg_rp = np.mean(rps)
         avg_ndcg = np.mean(ndcgs)
         print('Avg. R-Precision: ', avg_rp)
         print('Avg. NDCG: ', avg_ndcg)
         print('Total Sum: ', np.mean([avg_rp, avg_ndcg]))
```

```
Avg. R-Precision:  0.03337335854232656
Avg. NDCG:  0.042258525693232844
Total Sum:  0.0378159421177797
```