In [209]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import NearestNeighbors
from scipy.sparse import csr_matrix
import tqdm
from tqdm import tqdm
from IPython.display import clear_output
import matplotlib.pyplot as plt
%matplotlib inline
import os

subset100 = pd.read_csv("../raw_data/track_meta_100subset_new.csv")
```

Train-val-test split

In [205]:
```
# Train-val-test split (20%)
train, test = train_test_split(subset100, test_size=0.2, random_state=42, stratify = subset100['Playl
istid'])
```

In [206]:
```
test.head()
```

Out[206]:

	Playlistid	Trackid	Artist_Name	Track_Name	Album_Name	Track_Duration	Artist_uri	
557	38828	35	Bastille	Pompeii	Bad Blood	214147	spotify:artist:7EQ0qTo7fWT7DPxmxtSYEc	spotify:track:3gb
556	38828	34	Britney Spears	Womanizer	Circus (Deluxe Version)	224400	spotify:artist:26dSoYclwsYLMAKD3tpOr4	spotify:track:4f
2414	229646	7	Soft Cell	Tainted Love	Non-Stop Erotic Cabaret	153762	spotify:artist:6aq8T2RcspxVOGgMrTzjWc	spotify:track:0cG
1771	186672	28	Imagine Dragons	Radioactive	Night Visions	186813	spotify:artist:53XhwfbYqKCa1cC15pYq2q	spotify:track:6E
516	37634	17	LANY	WHERE THE HELL ARE MY FRIENDS	WHERE THE HELL ARE MY FRIENDS	216180	spotify:artist:49tQo2QULno7gxHutgccqF	spotify:track:4

5 rows × 28 columns

Create co-occurence matrix

In [207]:
```
# Create Binary Sparse Matrix
co_mat = pd.crosstab(train.Playlistid, train.Track_uri)
co_mat = co_mat.clip(upper=1)
```

ALS Definition

In [204]:

```
class RecommenderSystem():
    """Represents the scheme for implementing a recommender system."""
    def __init__(self, training_data, *params):
        """Initializes the recommender system.

        Note that training data has to be provided when instantiating.
        Optional parameters are passed to the underlying system.
        """
        raise NotImplementedError

    def train(self, *params):
        """Starts training. Passes optional training parameters to the system."""
        raise NotImplementedError

    def score(self, user_id, item_id):
        """Returns a single score for a user-item pair.

        If no prediction for the given pair can be made, an exception should be raised.
        """
        raise NotImplementedError

class ALSRecommenderSystem(RecommenderSystem):
    """Provides a biased ALS-based implementation of an implicit recommender system."""
    def __init__(self, training_data, biased, latent_dimension, log_dir=None, confidence=20):
        """Initializes the recommender system.

        Keyword arguments:
        training_data: Data to train on.
        biased: Whether to include user- and item-related biases in the model.
        latent_dimension: Dimension of the latent space.
        log_dir: Optional pointer to directory storing logging information.
        confidence: Confidence value that should be assigned to pairs where interaction
                    was present. Since the data includes single interactions only, simply
                    assigining 1 for non-interactions and this value otherwise suffices.
                    Should be greater than 1.
        """
        self.biased = biased
        self.confidence = confidence
        self.latent_dimension = latent_dimension
        self.U = None
        self.V = None
        self.log_dir = log_dir
```

```
        self.C_users, self.P_users, self.C_items, self.P_items, self.mapping_users, self.mapping_item
s = self._build_matrices(training_data, confidence)
        self.user_dim, self.item_dim = self.P_users.shape

    def _build_matrices(self, activity, confidence):
        """Build the initial matrices."""
        distinct_users = len(set(activity['user']))
        distinct_items = len(set(activity['items']))
        C_users = np.ones(shape=(distinct_users, distinct_items))
        P_users = np.zeros(shape=(distinct_users, distinct_items))
        C_items = np.ones(shape=(distinct_items, distinct_users))
        P_items = np.zeros(shape=(distinct_items, distinct_users))

        mapping_users = {}
        mapping_items = {}
        user_ct = 0
        items_ct = 0

        for index, row in activity.iterrows():
            user, items = row
            if not user in mapping_users:
                mapping_users[user] = user_ct
                user_ct += 1
            if not items in mapping_items:
                mapping_items[items] = items_ct
                items_ct += 1
            user_index, items_index = mapping_users[user], mapping_items[items]
            C_users[user_index, items_index] = confidence
            P_users[user_index, items_index] = 1
            C_items[items_index, user_index] = confidence
            P_items[items_index, user_index] = 1
        return C_users, P_users, C_items, P_items, mapping_users, mapping_items

    def save(self, directory):
        """Saves current matrices to the given directory."""
        np.save(os.path.join(directory, 'U.npy'), self.U)
        np.save(os.path.join(directory, 'V.npy'), self.V)
        #np.save(os.path.join(directory, 'training_data.npy'), self.training_data)
        np.save(os.path.join(directory, 'params.npy'), np.array([self.confidence]))
        if self.biased:
            np.save(os.path.join(directory, 'user_biases.npy'), self.user_biases)
            np.save(os.path.join(directory, 'item_biases.npy'), self.item_biases)
```

```
    def load(self, directory):
        """Loads matrices from the given directory."""
        self.U = np.load(os.path.join(directory, 'U.npy'))
        self.V = np.load(os.path.join(directory, 'V.npy'))
        self.training_data = np.load(os.path.join(directory, 'training_data.npy'))
        self.confidence = np.load(os.path.join(directory, 'params.npy')).flatten()
        if self.biased:
            self.user_biases = np.load(os.path.join(directory, 'user_biases.npy'))
            self.item_biases = np.load(os.path.join(directory, 'item_biases.npy'))

        self.C_users, self.P_users, self.C_items, self.P_items, self.mapping_users, self.mapping_item
s = self._build_matrices(self.training_data, self.confidence)
        self.user_dim, self.item_dim = self.P_users.shape

    def _single_step(self, lbd):
        """Executes a single optimization step using (biased) ALS, with lbd as regularization facto
r."""
        C_users, P_users, C_items, P_items, mapping_users, mapping_items = self.C_users, self.P_users
, self.C_items, self.P_items, self.mapping_users, self.mapping_items
        biased = self.biased

        # Update U.
        if biased: # Expand matrices to account for biases.
            U_exp = np.hstack((self.user_biases.reshape(-1,1), self.U))
            V_exp = np.hstack((np.ones_like(self.item_biases).reshape(-1,1), self.V))
            kdim = self.latent_dimension + 1
        else: # We work with copies here to make it safer to abort within updates.
            U_exp = self.U.copy()
            V_exp = self.V.copy()
            kdim = self.latent_dimension
        Vt = np.dot(np.transpose(V_exp), V_exp)
        for user_index in tqdm(range(self.user_dim)):
            C = np.diag(C_users[user_index])
            d = np.dot(C, P_users[user_index] - (0 if not biased else self.item_biases))
            val = np.dot(np.linalg.inv(Vt + np.dot(np.dot(V_exp.T, C - np.eye(self.item_dim)), V_exp)
 + lbd*np.eye(kdim)), np.transpose(V_exp))
            U_exp[user_index] = np.dot(val, d)
        if biased:
            self.user_biases = U_exp[:,0]
            self.U = U_exp[:,1:]
        else:
            self.U = U_exp
```

```
        # Update V.
        if biased:
            U_exp = np.hstack((np.ones_like(self.user_biases).reshape(-1,1), self.U))
            V_exp = np.hstack((self.item_biases.reshape(-1,1), self.V))
        else: # We work with copies here to make it safer to abort within updates.
            U_exp = self.U.copy()
            V_exp = self.V.copy()

        Ut = np.dot(np.transpose(U_exp), U_exp)
        for item_index in tqdm(range(self.item_dim)):
            C = np.diag(C_items[item_index])
            d = np.dot(C, P_items[item_index] - (0 if not biased else self.user_biases))
            val = np.dot(np.linalg.inv(Ut + np.dot(np.dot(U_exp.T, C-np.eye(self.user_dim)), U_exp) +
 lbd*np.eye(kdim)), np.transpose(U_exp))
            V_exp[item_index] = np.dot(val, d)
        if biased:
            self.item_biases = V_exp[:, 0]
            self.V = V_exp[:,1:]
        else:
            self.V = V_exp

    def compute_loss(self, lbd):
        """Computes loss value on the training data.

        Returns a tuple of total loss and prediction loss (excluding regularization loss).
        """
        C_users, P_users, C_items, P_items, mapping_users, mapping_items = self.C_users, self.P_users
, self.C_items, self.P_items, self.mapping_users, self.mapping_items
        main_loss = 0
        # Main loss term.
        for user_index in range(self.user_dim):
            for item_index in range(self.item_dim):
                pred = np.dot(self.U[user_index].T, self.V[item_index])
                if self.biased:
                    pred += self.user_biases[user_index] + self.item_biases[item_index]
                loss = self.C_users[user_index, item_index] * (P_users[user_index, item_index]-pred)*
*2
                main_loss += loss

        # Regularization term.
        reg_loss = 0
        if lbd > 0:
            for user_index in range(self.user_dim):
```

```
                reg_loss += np.sum(self.U[user_index]**2) + (0 if not self.biased else self.user_bias
es[user_index]**2)
            for item_index in range(self.item_dim):
                reg_loss += np.sum(self.V[item_index]**2) + (0 if not self.biased else self.item_bias
es[item_index]**2)
            reg_loss *= lbd
        return main_loss + reg_loss, main_loss

    def train(self, lbd, iterations=20, verbose=True):
        """
        Trains the recommendation system.

        Keyword arguments:
        lbd: Regularization factor.
        iterations: Number of iterations to run ALS.
        verbose: Whether to plot and output training loss.
        """
        if self.U is None or self.V is None:
            self.U = np.random.normal(size=(self.user_dim, self.latent_dimension))
            self.V = np.random.normal(size=(self.item_dim, self.latent_dimension))
            self.user_biases = np.zeros(self.user_dim)
            self.item_biases = np.zeros(self.item_dim)
            self.history_losses = []
            self.history_main_losses = []
            self.history_avg_score = []
            self.history_avg_rank = []

        it = 0
        while(it < iterations):
            self._single_step(lbd)
            loss, main_loss = self.compute_loss(lbd)
            self.history_losses.append(loss)
            self.history_main_losses.append(main_loss)

            if verbose:
                clear_output(wait=True)
                print('LOSS:', loss, 'MAIN LOSS:', main_loss)

                plt.figure(figsize=(5,5))
                plt.title('training loss (lower is better)')
                plt.plot(range(len(self.history_losses)), self.history_losses)
                plt.plot(range(len(self.history_main_losses)), self.history_main_losses, color='orang
e')
```

```
                plt.plot(range(len(self.history_main_losses)), np.array(self.history_losses) - np.arr
ay(self.history_main_losses), color='green')
                plt.legend(['total loss', 'data loss', 'regularizing loss'])
                if self.log_dir is not None:
                    plt.savefig(os.path.join(self.log_dir, 'log.png'), bbox_inches='tight', format='p
ng')
                plt.show()
            it += 1

    def reset(self):
        """Resets the recommendation system's internal state."""
        self.U = None
        self.V = None

    def score(self, user_id, items_id):
        """Returns the scoring of item_id for user_id."""
        if self.U is None or self.V is None:
            raise ValueError('system has to be trained first')
        if user_id not in self.mapping_users:
            raise ValueError('user unknown')
        if items_id not in self.mapping_items:
            raise ValueError('item unknown')

        user_index = self.mapping_users[user_id]
        items_index = self.mapping_items[items_id]
        pred = np.dot(self.U[user_index], self.V[items_index])
        if self.biased: # Include applicable biases.
            pred += self.user_biases[user_index] + self.item_biases[items_index]
        return pred
```

Training

In [210]:
```
res = []
for i, row in co_mat.iterrows():
    for track in row[row == 1].index.values:
        res.append((i, track))
res = pd.DataFrame(np.array(res), columns=['user', 'items'])
res.head()
```

Out[210]:

	user	items
0	430	spotify:track:0E1NL6gkv5aQKGNjJfBE3A
1	430	spotify:track:0OuPMjmicFfmnB3SFFqdgQ
2	430	spotify:track:0Tlv1rjOG6Wbc02T4p3y7o
3	430	spotify:track:1CtOCnWYflwVgIKiR2Lufw
4	430	spotify:track:1lNGwNQX4lrvDwgETwyPjR

In [224]:
```
rs = ALSRecommenderSystem(res, True, latent_dimension=20)
```

In [ ]:
```
rs.train(0.0001, iterations=30)
```

In [186]:
```
def als_similar_songs_playlist(model, orig_df, target_playlist_id, cand_list_size):
    """
    Input:
    model: the recommendation system that was trained on the training set with latent factors
    orig_df: original df with tracks as rows, but with playlistid and other features (e.g., train)
    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 of tracks in tr
aining playlist of target playlist
    score_allsongs = list(map(lambda x: model.score(str(target_playlist_id), x), orig_df["Track_uri"
]))
    rec_inx = np.argsort(score_allsongs)[::-1]

    cand_list = orig_df.iloc[rec_inx]['Track_uri']
    unique_cand_list = cand_list.drop_duplicates()#list(set(cand_list)) # drop duplciated tracks

    tracks_in_target_playlist = orig_df.loc[orig_df["Playlistid"] == target_playlist_id, "Track_uri"]

    cand_list2 = unique_cand_list.loc[~unique_cand_list.isin(tracks_in_target_playlist)] # remove son
gs that are in the
    cand_list3 = cand_list2[:cand_list_size]
    return list(cand_list3)
```

Making Predictions

In [ ]:
```
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)
```

Metrics

In [191]:
```
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 [193]:
```
### 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
```

Model Performance

In [226]:
```
rps = []
ndcgs = []
for pid in co_mat.index:
    ps = als_similar_songs_playlist(rs, train, pid, nholdout(pid, train)*15)
    vs = test[test.Playlistid == pid].Track_uri # ground truth
    rps.append(r_precision(ps, vs))

    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 [236]:
```
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:  0.23969273359366242
Avg. NDCG:  0.1504029917610824
Total Sum:  0.1950478626773724
```

In [ ]: