In [1]:

```
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn.apionly as sns
sns.set_context("poster") # can set to poster to make really large
```

In [2]:

```
sns.set()
```

For information on how this data was gotten from spotify and processed please see:

https://github.com/tclements/Spotify-Project (https://github.com/tclements/Spotify-Project)

Spotify Playlist Data EDA (for milestone 3)

In [3]:

```
data = pd.read_csv('spotify.csv')
data.drop('Unnamed: 0',axis=1,inplace=True)
data['explicit_num'] = data['explicit']*1
data.head()
```

Out[3]:

	acousticness	album	album_href	
0	0.02070	Kids with Cars	https://api.spotify.com/v1/albums/0H4Nrf3GiTuj...	0H4Nrf3(
1	0.08690	Montevallo	https://api.spotify.com/v1/albums/0V7c0hnrLUFJ...	0V7c0hn
2	0.00908	Love You Like That	https://api.spotify.com/v1/albums/3Q0jr9uSc24I...	3Q0jr9uS
3	0.11200	The B-Sides, 2011-2014	https://api.spotify.com/v1/albums/4B1IUKpspjSV...	4B1IUKp
4	0.03440	747	https://api.spotify.com/v1/albums/1JHNRou038Cf...	1JHNRou

5 rows × 38 columns

In [4]:

```
data.describe()
```

Out[4]:

	acousticness	danceability	duration_ms	energy	followers	inst
count	85189.000000	85189.000000	8.531300e+04	85189.000000	8.531300e+04	851
mean	0.365891	0.549410	2.695307e+05	0.560932	2.649094e+05	0.15
std	0.359562	0.178117	3.250850e+05	0.281970	7.022352e+05	0.30
min	0.000000	0.000000	1.155000e+03	0.000000	0.000000e+00	0.00
25%	0.026800	0.436000	1.906930e+05	0.334000	6.471000e+03	0.00
50%	0.224000	0.569000	2.247730e+05	0.599000	5.787800e+04	0.00
75%	0.725000	0.679000	2.718000e+05	0.803000	2.360200e+05	0.06
max	0.996000	0.985000	2.022006e+07	1.000000	1.810417e+07	1.00

In [5]:

```
data.keys()
```

Out[5]:

```
Index(['acousticness', 'album', 'album_href', 'album_id', 'album_uri
',
       'artist', 'artist_href', 'artist_id', 'artist_uri', 'collabor
ative',
       'danceability', 'description', 'duration_ms', 'energy', 'expl
icit',
       'followers', 'instrumentalness', 'key', 'liveness', 'loudness
', 'mode',
       'playlist_href', 'playlist_id', 'playlist_name', 'playlist_ur
i',
       'popularity', 'public', 'snapshot_id', 'speechiness', 'tempo'
,
       'time_signature', 'track_href', 'track_id', 'track_name',
       'track_number', 'track_uri', 'valence', 'explicit_num'],
      dtype='object')
```

Getting genres of first artist on playlist

In [6]:

```
import json
with open('artist_genre.json','r') as fp:
    artist_genre = json.load(fp)

split_genre = []
keys = list(artist_genre)
for key in keys:
    g = artist_genre[key]
    for genre in g:
        split_genre.append(genre)
vals,counts = np.unique(split_genre,return_counts = True)
ind = np.argsort(counts)[::-1]
results = []
index=0
for v,c in zip(vals[ind],counts[ind]):
    results.append([v,c])
    index=index+1
```

In [7]:

```
results[:50]
```

Out[7]:

```
[['dance pop', 572],
 ['modern rock', 566],
 ['pop', 562],
 ['rock', 527],
 ['folk-pop', 466],
 ['indie folk', 452],
 ['pop rock', 449],
 ['mellow gold', 445],
 ['indie pop', 444],
 ['indie rock', 443],
 ['indietronica', 432],
 ['pop rap', 428],
 ['alternative rock', 421],
 ['tropical house', 402],
 ['indie r&b', 398],
 ['soft rock', 396],
 ['classic rock', 378],
 ['chamber pop', 362],
 ['stomp and holler', 356],
 ['neo mellow', 355],
 ['singer-songwriter', 350],
 ['edm', 345],
 ['rap', 338],
 ['album rock', 337],
 ['quiet storm', 336],
 ['folk rock', 335],
 ['urban contemporary', 334],
 ['roots rock', 325],
 ['electro house', 317],
 ['pop christmas', 316],
 ['funk', 316],
 ['new wave pop', 313],
 ['r&b', 312],
 ['alternative metal', 312],
 ['classic funk rock', 311],
 ['adult standards', 309],
 ['new americana', 308],
 ['indie poptimism', 302],
 ['soul', 297],
 ['blues-rock', 295],
 ['brill building pop', 292],
 ['hip hop', 292],
 ['motown', 291],
 ['neo-psychedelic', 290],
 ['pop punk', 285],
 ['disco', 282],
 ['dance rock', 279],
 ['garage rock', 276],
 ['modern blues', 276],
 ['hard rock', 275]]
```

In [8]:

```
genres = list(["pop", 'rock', 'folk', 'mellow', 'indie', 'rap', 'house', 'r&b',
'stomp and holler', 'singer-songwriter', 'edm', 'quiet storm', 'contemporary', '
christmas', 'funk', 'metal', 'alternative', 'adult standards', 'americana', 'sou
l', 'blues', 'hip hop', 'motown', 'punk', 'disco','other'])

def genre_search(artist,genres=genres):
    genre_list = np.zeros(len(genres))
    for i,genre in enumerate(genres):
        genre_list[i] = any(genre in s for s in artist)*1
    genre_list[i] = (genre_list.sum() == 0)*1
    return genre_list
```

In [9]:

```
keys = list(artist_genre)
genre_data = pd.DataFrame(np.zeros((len(keys),len(genres))),columns=genres)
genre_data['artist_id'] = keys

for key in keys:
    genre_data.loc[genre_data.artist_id == key,genres] = genre_search(artist_gen
re[key])
data_with_genre = pd.merge(data,genre_data,how='left',on='artist_id',sort=False)
data_with_genre.fillna(0,inplace=True)
print(genre_data.shape)
print(data.shape)
print(data_with_genre.shape)
data_with_genre.head()
data = data_with_genre.copy()
```

```
(20533, 27)
(85313, 38)
(85313, 64)
```

Grouping by Playlist

In [10]:

```
# choose predictors

cont_predictors = ['acousticness','danceability','duration_ms','energy','instrum
entalness','liveness','loudness','popularity','speechiness','tempo','valence']
cat_predictors = ['explicit','mode','time_signature','key'] # getting rid of col
laborative because all false
cat_predictors = cat_predictors+genres
other_predictors = ['artist','album','description','playlist_name','track_name']
missing = ['genre']
response = ['followers']
index = ['playlist_name']
```

In [11]:

```
# group continuous predictors by playlist -- mean

df_cont = data[cont_predictors].copy()
df_cont['playlist_name'] = data['playlist_name']
df_cont = df_cont.groupby('playlist_name').agg(lambda x:np.mean(x))
df_cont.head()
```

Out[11]:

	acousticness	danceability	duration_ms	energy	instrum
playlist_name					
#FlashbackFriday	0.154854	0.652320	253792.900000	0.673036	0.0023
#Hotmess	0.036124	0.522000	214560.041667	0.790167	0.1166
#JazzAppreciationMonth	0.878450	0.488650	244925.600000	0.199132	0.2837
#MakeitHappen	0.110002	0.647433	220558.700000	0.704100	0.0012
#MetalMonday	0.000478	0.365700	302320.466667	0.939467	0.3313

In [12]:

```
# group categorical predictors by playlist -- vote

df_cat = data[cat_predictors].copy()
df_cat['playlist_name'] = data['playlist_name']
df_cat['followers'] = data['followers']
df_cat = df_cat.groupby('playlist_name').agg(lambda x:x.value_counts().index[0])
df_cat['num_tracks'] = data['playlist_name'].value_counts()
df_cat['explicit'] = df_cat['explicit']*1
df_cat.head()
```

Out[12]:

	explicit	mode	time_signature	key	pop	rock	folk	mellow
playlist_name								
#FlashbackFriday	0	1.0	4.0	1.0	1.0	0.0	0.0	0.0
#Hotmess	0	1.0	4.0	9.0	1.0	1.0	0.0	0.0
#JazzAppreciationMonth	0	1.0	4.0	0.0	0.0	0.0	0.0	0.0
#MakeitHappen	0	1.0	4.0	0.0	1.0	0.0	0.0	0.0
#MetalMonday	0	1.0	4.0	11.0	0.0	0.0	0.0	0.0

5 rows × 32 columns

In [13]:

```
df_cat[genres].sum()
```

Out[13]:

```
pop                 764.0
rock                549.0
folk                184.0
mellow              116.0
indie               236.0
rap                 140.0
house                81.0
r&b                  57.0
stomp and holler     35.0
singer-songwriter    58.0
edm                  39.0
quiet storm          30.0
contemporary         75.0
christmas           202.0
funk                 52.0
metal                96.0
alternative         147.0
adult standards      27.0
americana            22.0
soul                 75.0
blues                86.0
hip hop              68.0
motown               29.0
punk                 59.0
disco                18.0
other               285.0
dtype: float64
```

In [14]:

```
df = pd.concat([df_cont,df_cat],axis=1)
```

In [15]:

```
df.head()
```

Out[15]:

	acousticness	danceability	duration_ms	energy	instrum
playlist_name					
#FlashbackFriday	0.154854	0.652320	253792.900000	0.673036	0.0023
#Hotmess	0.036124	0.522000	214560.041667	0.790167	0.1166
#JazzAppreciationMonth	0.878450	0.488650	244925.600000	0.199132	0.2837
#MakeitHappen	0.110002	0.647433	220558.700000	0.704100	0.0012
#MetalMonday	0.000478	0.365700	302320.466667	0.939467	0.3313

5 rows × 43 columns

In [16]:

```
### Looking at features by track
```

In [17]:

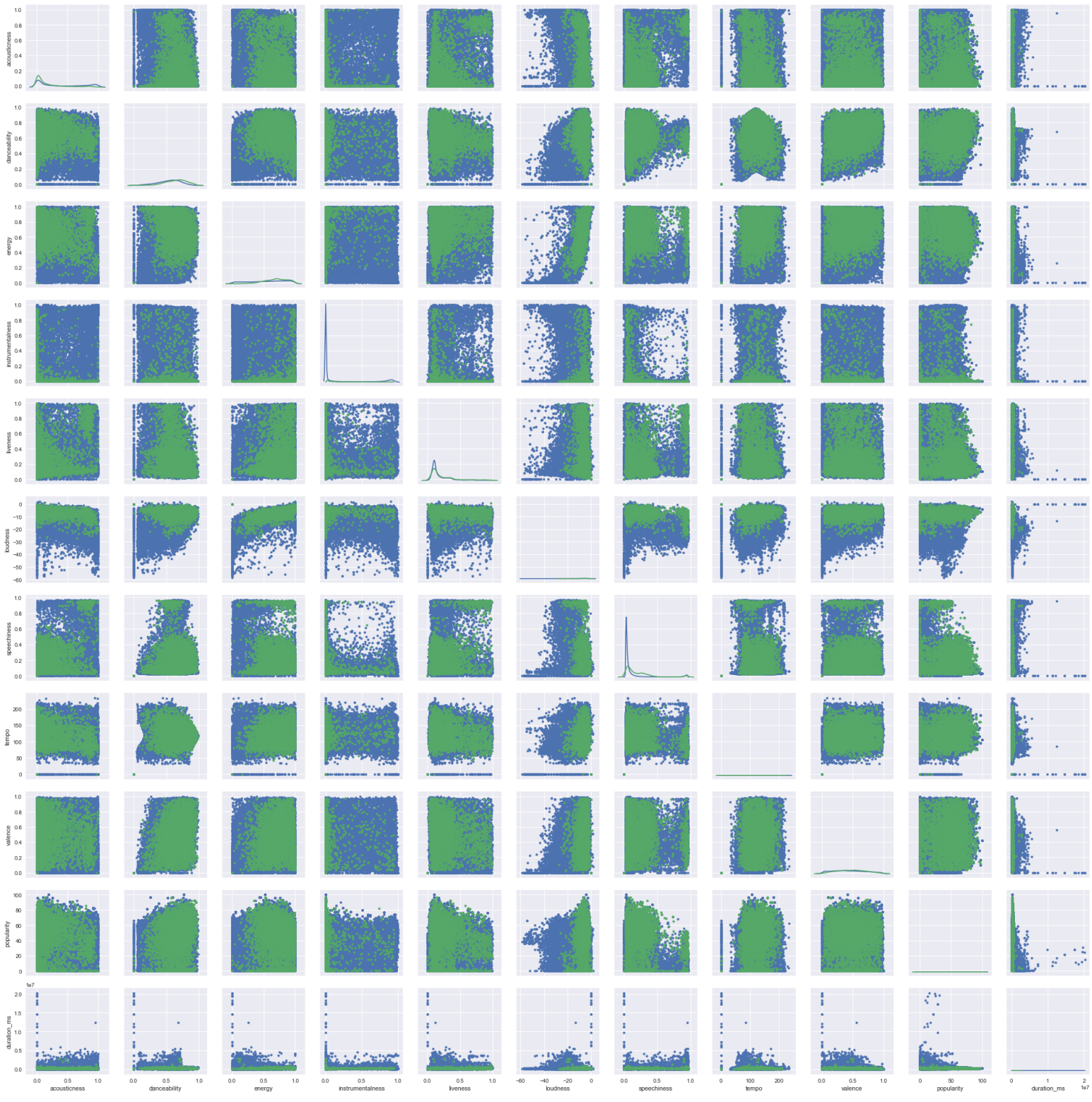
```
audio_features = list(['acousticness','danceability','energy','instrumentalness'
, 'key', 'liveness',
                       'loudness', 'mode','speechiness', 'tempo','time_signature
','valence'])
song_charact = list(['followers','track_number','duration_ms','explicit_num','po
pularity'])
song_descrip = list(['album','artist','playlist_name','track_name','description'
])
```

In [18]:

```
p = sns.PairGrid(data, vars=['acousticness','danceability','energy','instrumentalness', 'liveness',
                             'loudness', 'speechiness', 'tempo','valence','popularity','duration_ms'],hue='explicit_num')
p.map_diag(sns.kdeplot) # kernal density instead of scatter plot
p.map_offdiag(plt.scatter, s=15)
```

Out[18]:

<seaborn.axisgrid.PairGrid at 0x110ea5198>



Distributions:

Accousticness is bimodal, instrumentalness/speechiness are sort of bimodal and have large prevalence of low values, liveliness is right skewed, there are some outlier songs in terms of duration (also tend to be low/mid popularity songs).

Interactions:

Some predictors appear to have strong correlations while others don't. For example, loudness distribution is influenced by all the other variables, while valence only seems to be influenced by loudness and danceability to a small degree.

Explicit:

Explicit definitely should have interaction terms, whereas mode doesn't seem to impact distribution of other audio features. None of the variables seem strongly correlated. Explicit songs tend to be loud, shorter and it impacts some predictor interactions strongly (i.e. intrumentalness and acousticness).

Looking at features by playlist

Followers

In [19]:

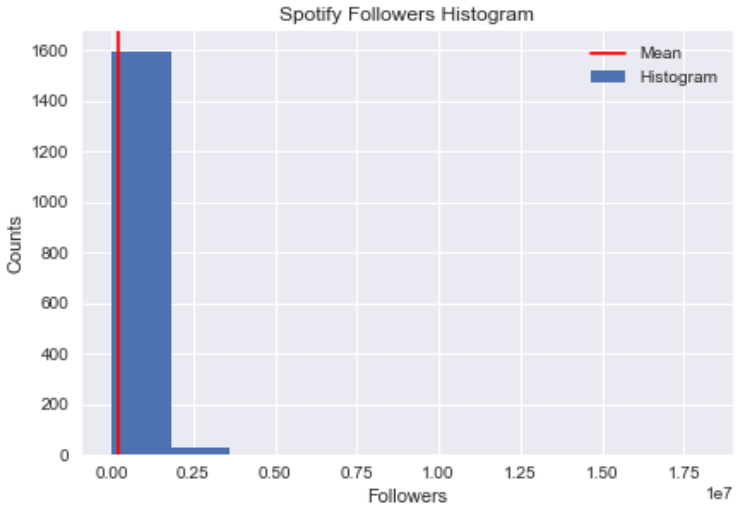
```
df.followers.describe()
```

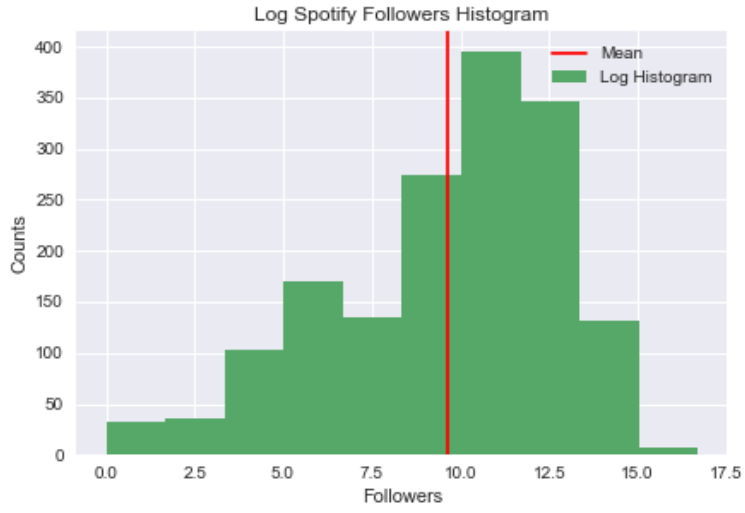
Out[19]:

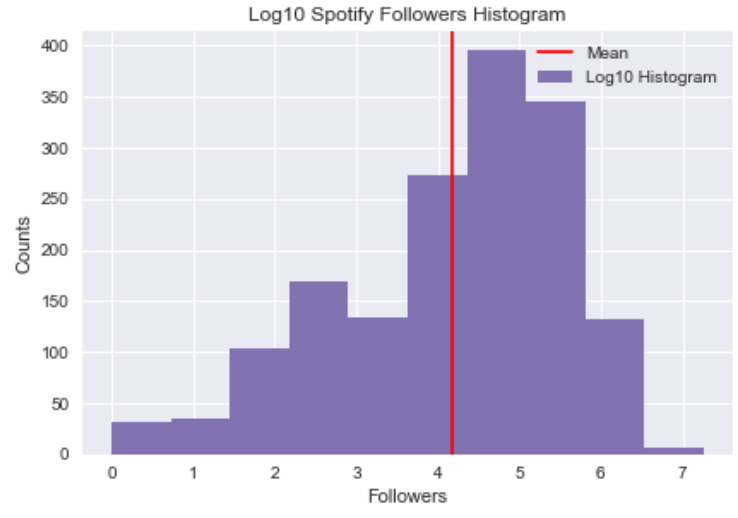
```
count    1.628000e+03
mean     2.144965e+05
std      6.778990e+05
min      0.000000e+00
25%      2.059500e+03
50%      3.206900e+04
75%      1.672992e+05
max      1.810417e+07
Name: followers, dtype: float64
```

In [20]:

```
# followers

plt.figure(figsize=(15,10))
plt.subplot(2,2,1)
plt.hist(df.followers.values,label='Histogram')
plt.axvline(df.followers.mean(), 0, 1.0, color='red', label='Mean')
plt.xlabel("Followers")
plt.ylabel("Counts")
plt.title("Spotify Followers Histogram")
plt.legend();
plt.subplot(2,2,2)
plt.hist(np.log(df.followers.values+1),label='Log Histogram',color=sns.color_pal
ette()[1])
plt.axvline(np.mean(np.log(df.followers.values+1)), 0, 1.0, color='red', label='
Mean')
plt.xlabel("Followers")
plt.ylabel("Counts")
plt.title("Log Spotify Followers Histogram")
plt.legend();
plt.subplot(2,2,3)
plt.hist(np.log10(df.followers.values+1),label='Log10 Histogram',color=sns.color
_palette()[3])
plt.axvline(np.mean(np.log10(df.followers.values+1)), 0, 1.0, color='red', label
='Mean')
plt.xlabel("Followers")
plt.ylabel("Counts")
plt.title("Log10 Spotify Followers Histogram")
plt.legend();
```







Playlist followers is very right skewed. After taking natural log the data becomes relatively normally distributed, so this would serve to make assumption of normality valid.

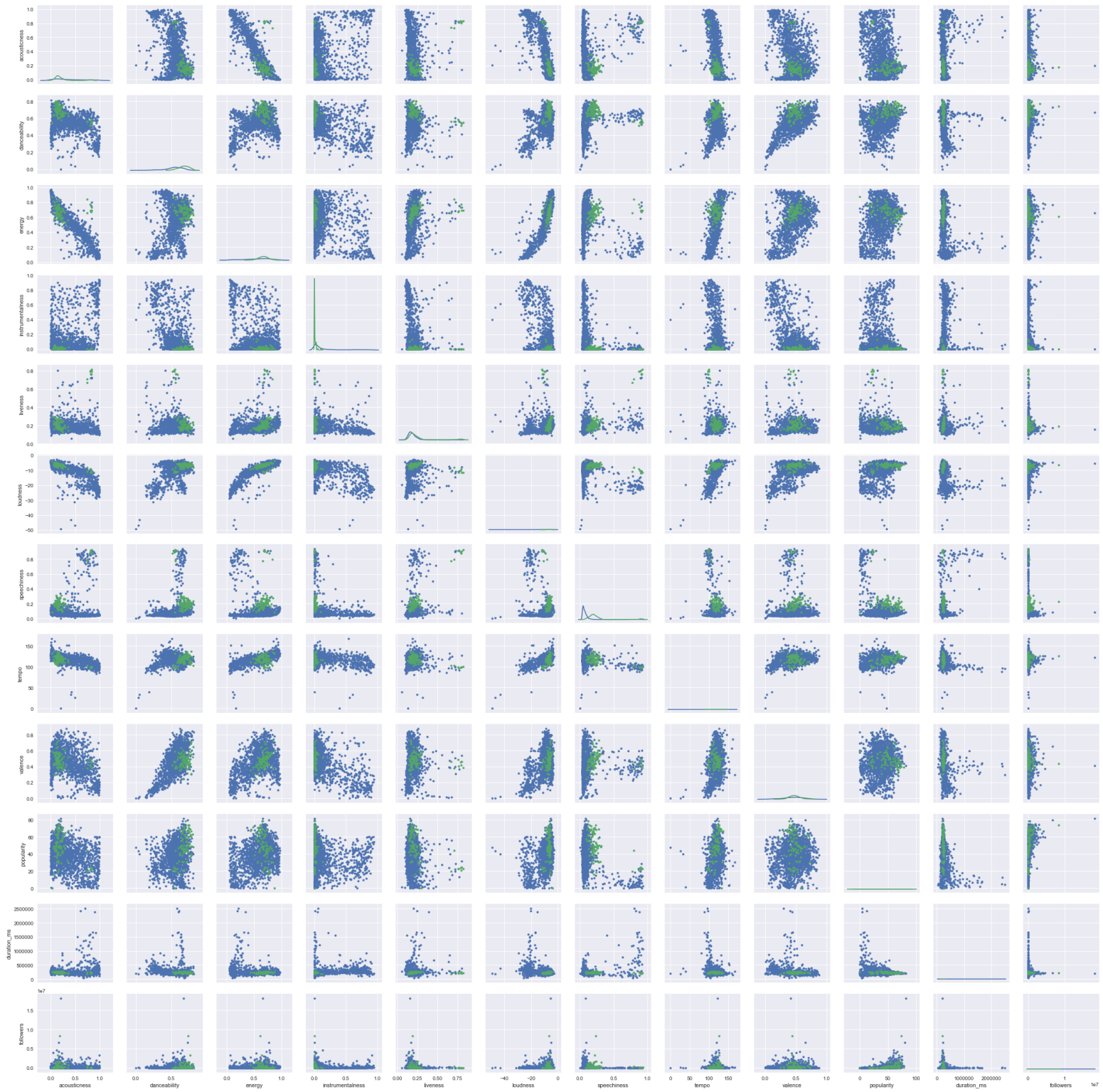
Quantitative Predictors

In [21]:

```
p = sns.PairGrid(df, vars=['acousticness','danceability','energy','instrumentaln
ess', 'liveness',
                           'loudness', 'speechiness', 'tempo','valence','popularity'
,'duration_ms','followers'],hue='explicit')
p.map_diag(sns.kdeplot) # kernal density instead of scatter plot
p.map_offdiag(plt.scatter, s=15)
```

Out[21]:

<seaborn.axisgrid.PairGrid at 0x12c1c4240>



Quantitative predictors:

Some predictors seem correlated when averaged by playlist - for example, accousticness and energy, where sounds with more acousticness tend to have less energy. Some predictors tend to have only a few extreme outliers like liveliness, loudness, tempo and duration.

Explicit:

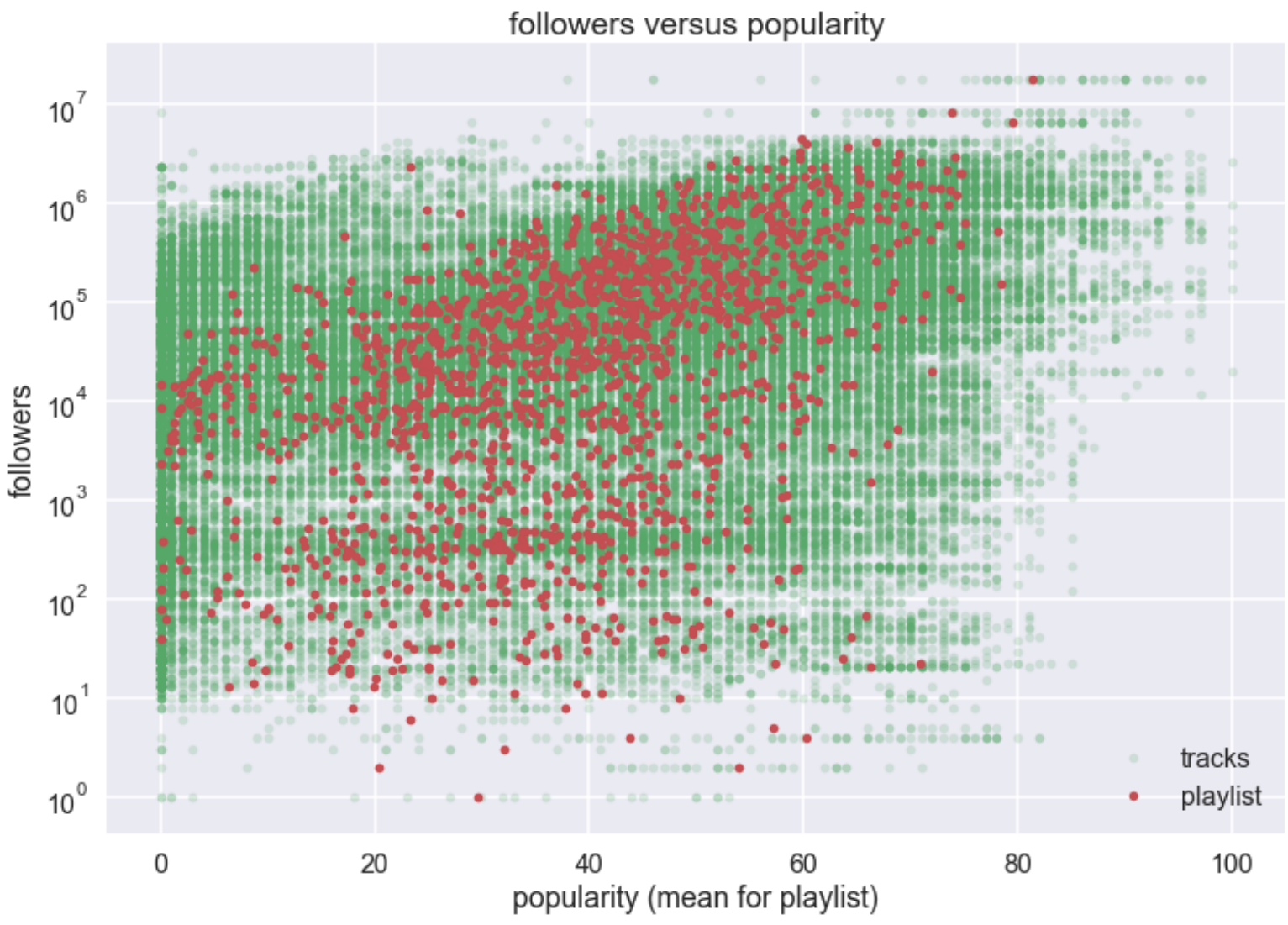
Playlists with majority explicit songs (shown in green on top paired grid plots) generally have a different distribution of predictors than playlists with non-majority explicit songs. For example, explicit songs are louder and tend to have more energy/danceability - similar to what was observed for explicitness in individual tracks just more extreme separation when playlists are considered (with majority explicitness).

In [22]:

```
def quantPlot(x,y='followers'):
    sns.set_context("poster",font_scale=1)
    plt.plot(data[x],data[y],'.',label='tracks',alpha=0.2,color=sns.color_palett
e()[1]);
    plt.plot(df[x],df[y],'.',label='playlist',color=sns.color_palette()[2]);
    plt.xlabel(x+' (mean for playlist)');
    plt.ylabel(y); plt.yscale('log')
    plt.title('followers versus '+x);
    plt.legend();
```

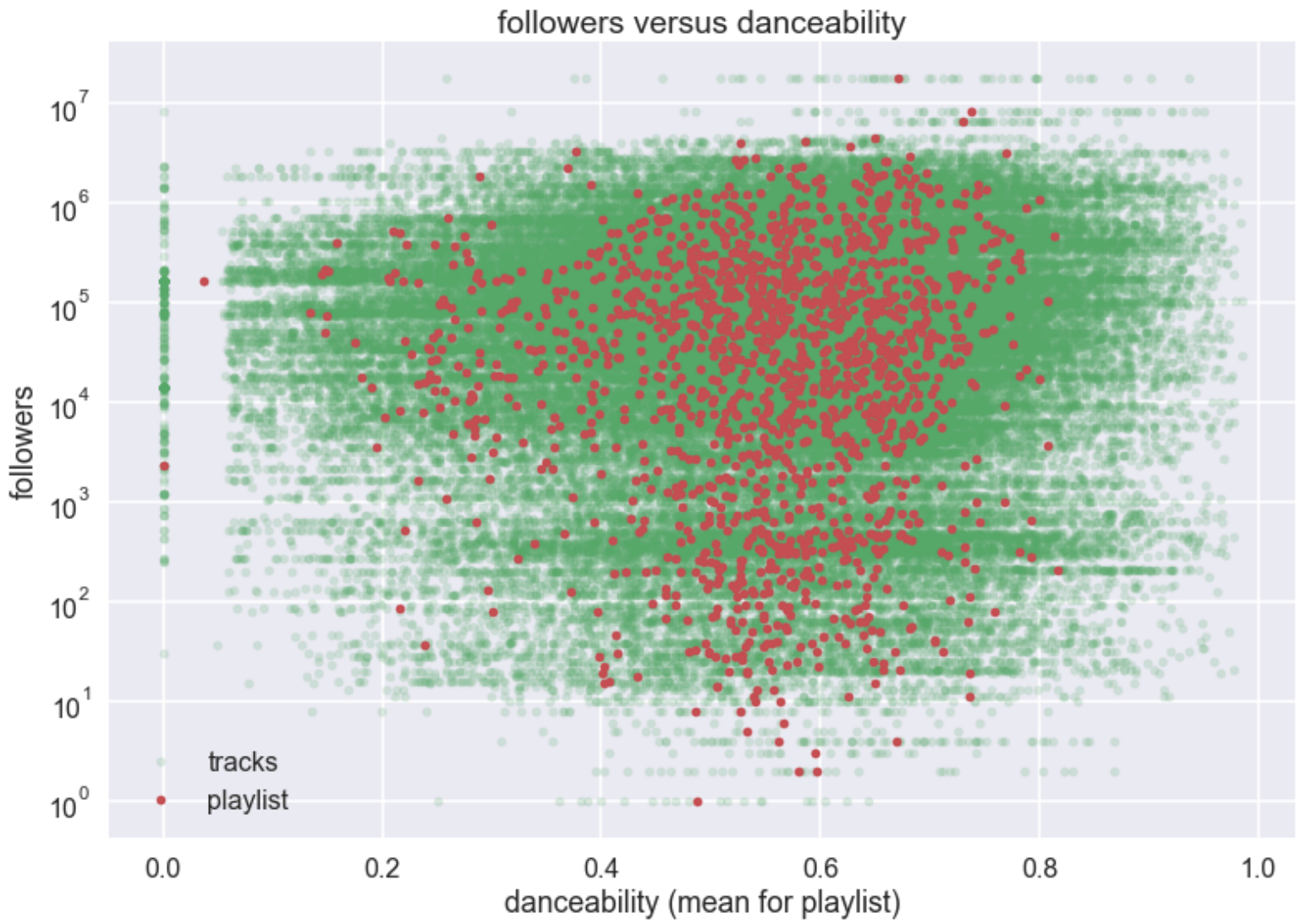
In [23]:

```
quantPlot('popularity')
```



In [24]:

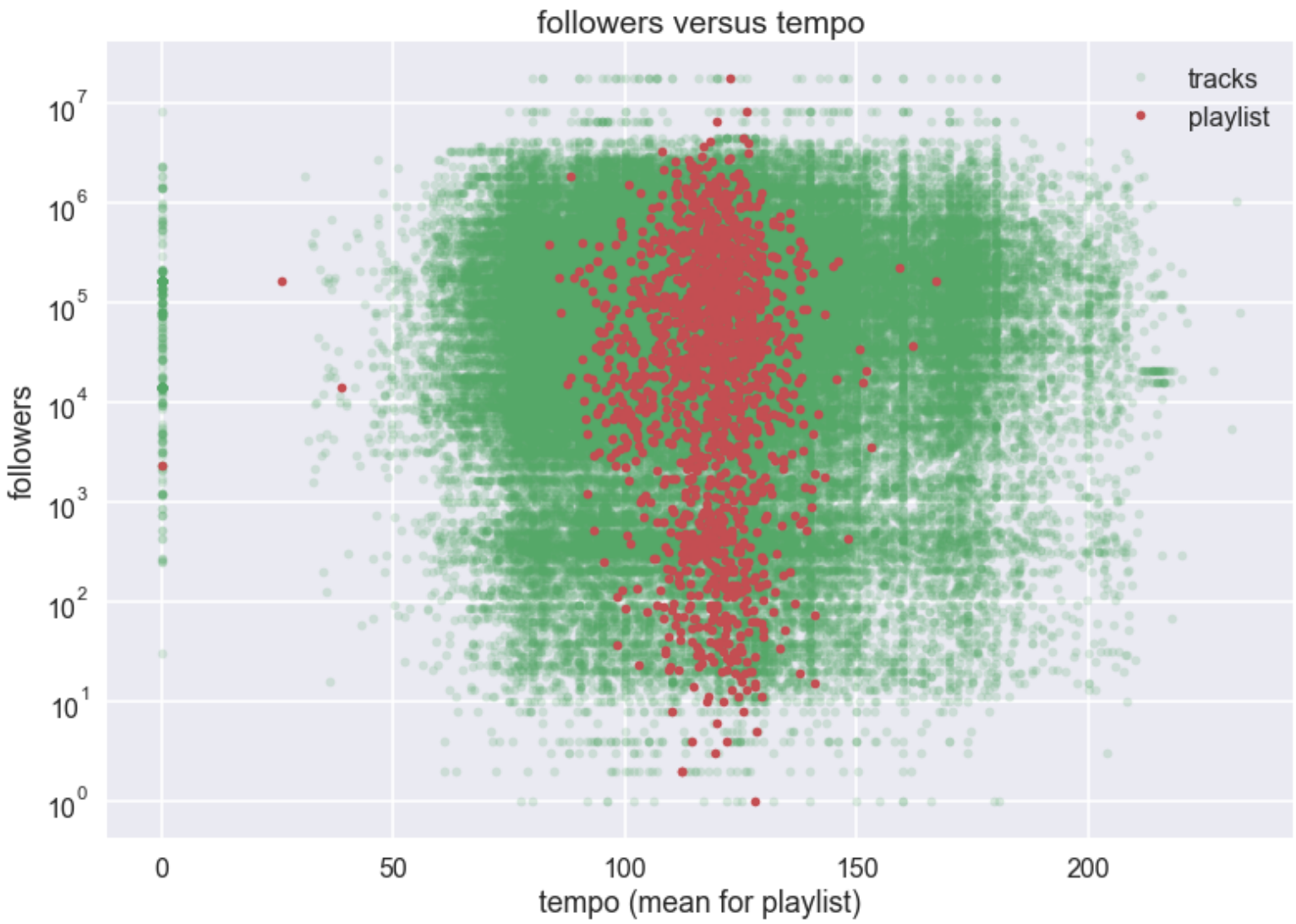
```
quantPlot('danceability')
```



This doesn't say much to me except that we're looking at relatively 'danceable' playlists; the cloud is most dense above danceability = 0.5.

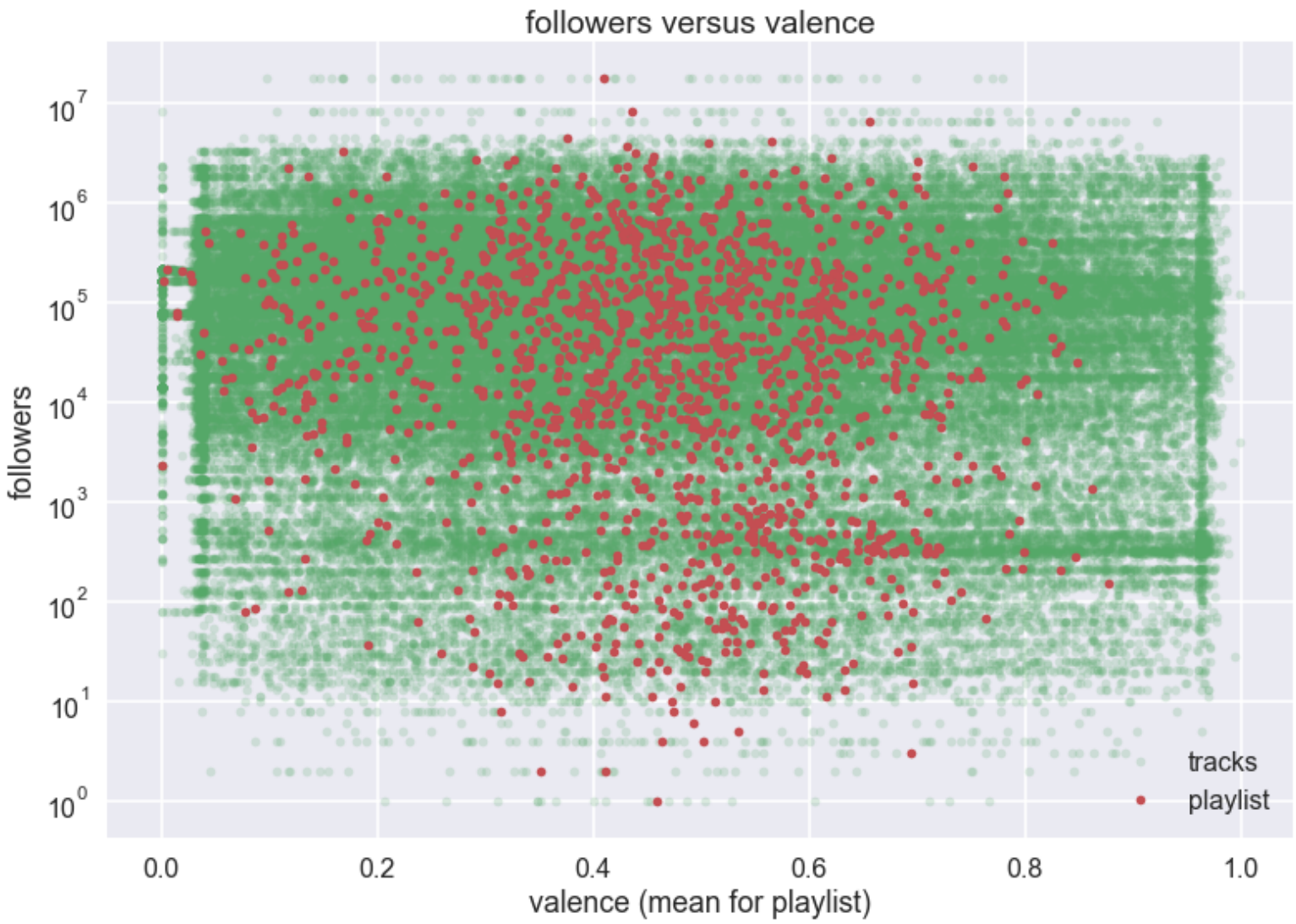
In [25]:

```
quantPlot('tempo')
```



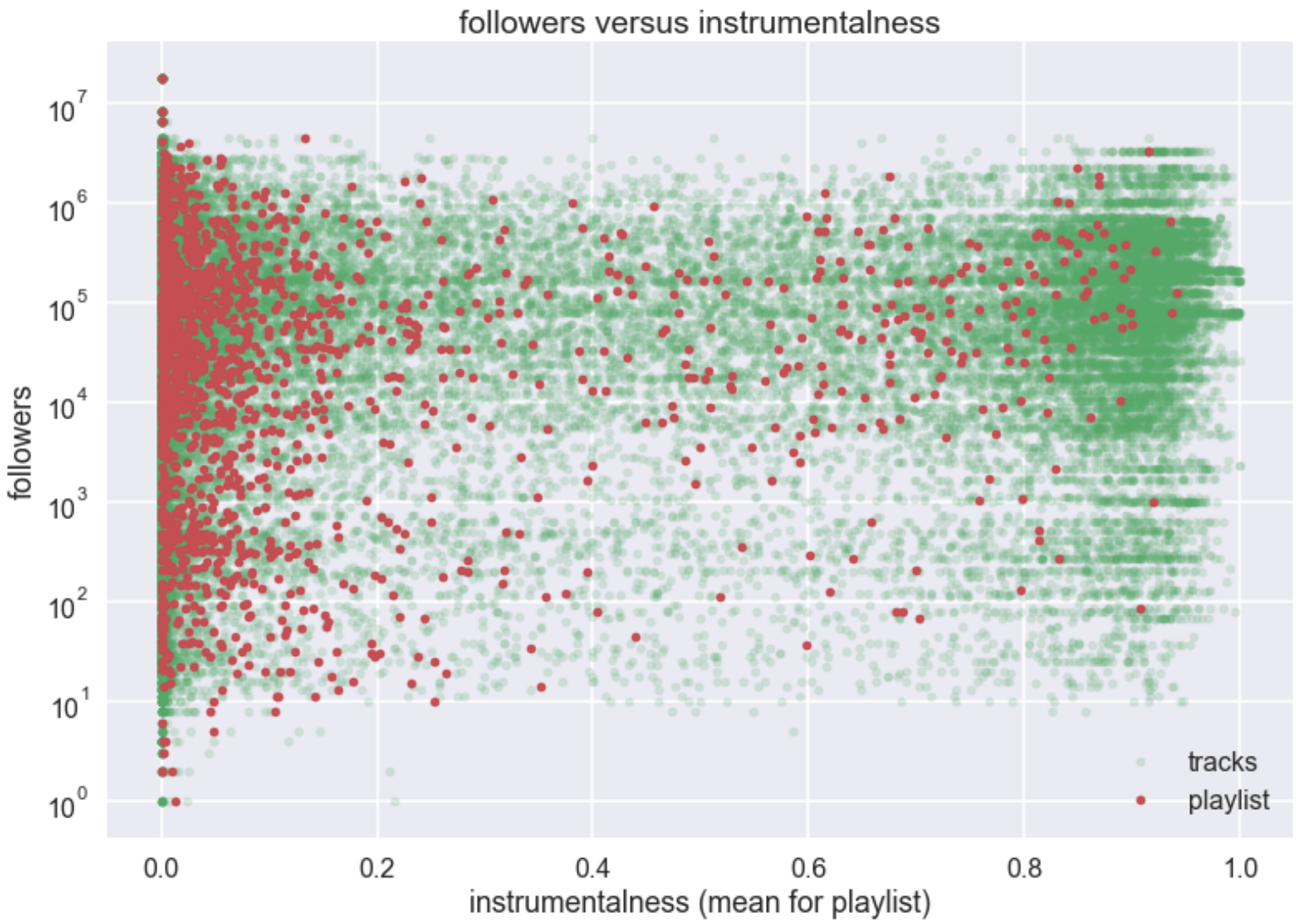
In [26]:

```
quantPlot('valence')
```



In [27]:

```
quantPlot('instrumentalness')
```



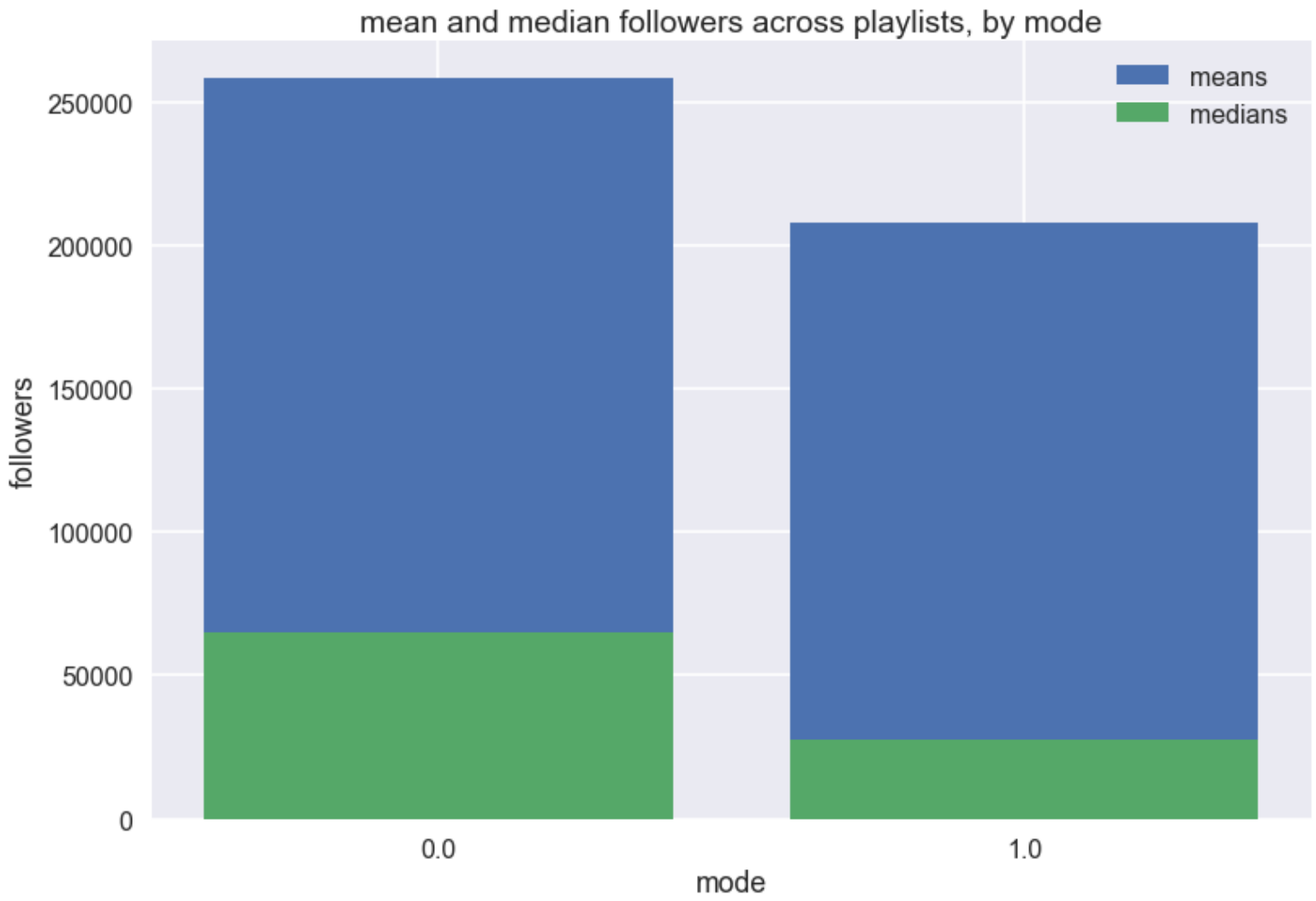
Valence, tempo, energy and danceability don't seem that correlated with number of playlist followers. More extreme values of intrumentalness tend to be common for playlists with larger numbers of followers. Playlists with more followers tend to have a songs with higher popularity, although there also appears to be a divide where playlists with a lower number of followers don't follow this relationship as well.

Categorical predictors

In [28]:

```
# what about mode?

# mean
mnmode_df = df[['mode','followers']]
mnmode_df = mnmode_df.groupby('mode').agg(lambda x:np.mean(x))

# median
mdmode_df = df[['mode','followers']]
mdmode_df = mdmode_df.groupby('mode').agg(lambda x:np.median(x))

# combine
mode_df = mdmode_df
mode_df['mean_followers'] = mnmode_df['followers']
mode_df['median_followers'] = mdmode_df['followers']
mode_df = mode_df.drop('followers',axis=1)
print(mode_df.head())

index = np.arange(2)
bar_width = 0.4
plt.bar(index,mode_df['mean_followers'].values,label='means')
plt.bar(index,mode_df['median_followers'].values,label='medians')
objects = ['0.0', '1.0']
plt.xticks(index,objects);
plt.xlabel('mode')
plt.ylabel('followers')
plt.title('mean and median followers across playlists, by mode')
plt.legend();
```

```
      mean_followers  median_followers
mode
0.0     258441.161137            64871
1.0     207952.883557            27291
```



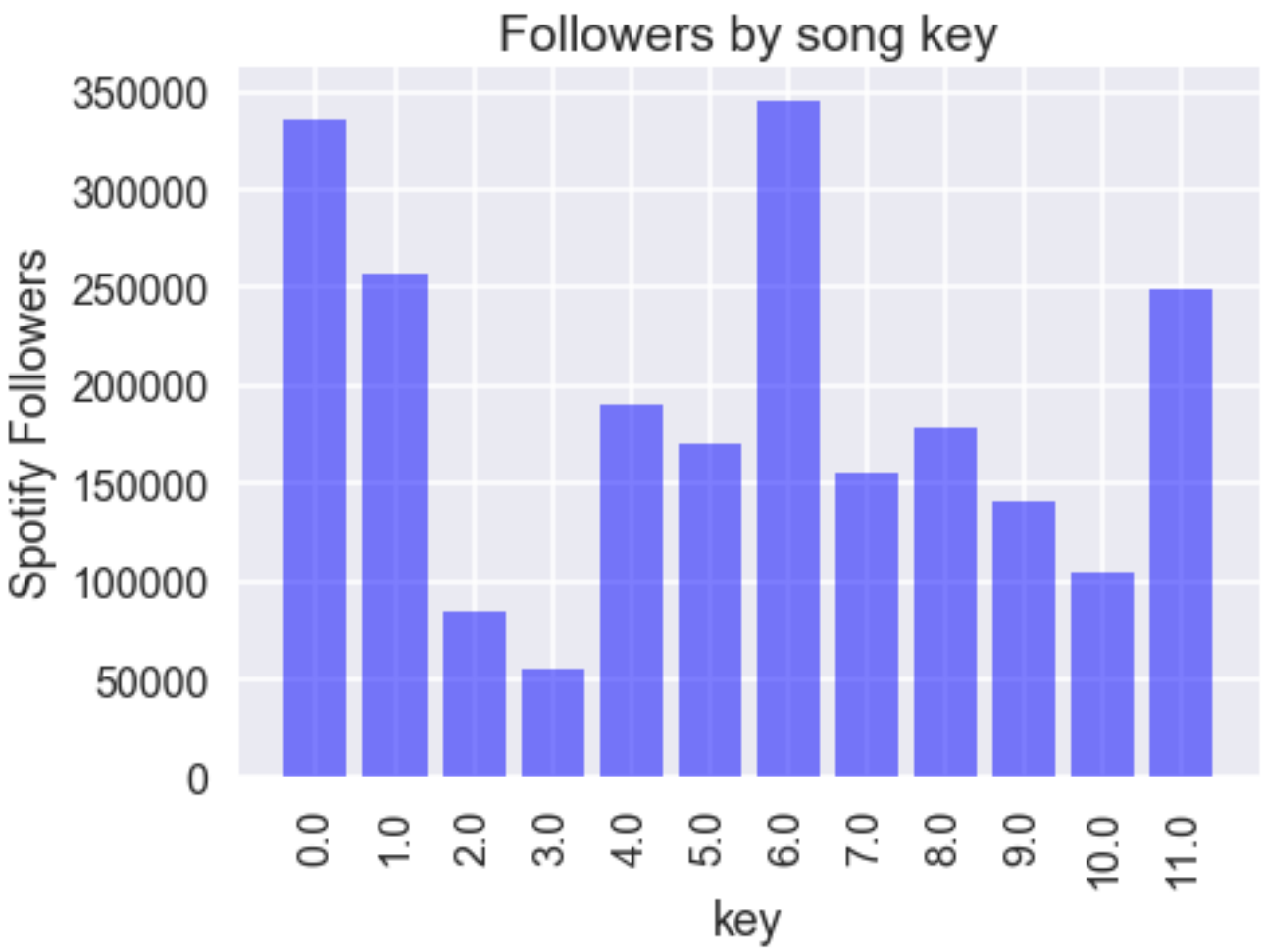
Mode seems helpful. If I remember correctly, mode 0 is major. So people tend to prefer songs in major keys?

In [29]:

```
def bars(var,title,xlab=True,color='blue',data=df):
    sort_var = data.groupby(data[var])
    x=sort_var['followers'].mean()
    ticks=np.arange(len(x))
    plt.bar(ticks,x,color=color,alpha=0.5)
    plt.ylabel('Spotify Followers')
    if xlab==True:
        plt.xticks(ticks,list(x.index),rotation='vertical')
        plt.xlabel(var)
    else:
        plt.xticks([])
    plt.title(title)
    #plt.show()
```

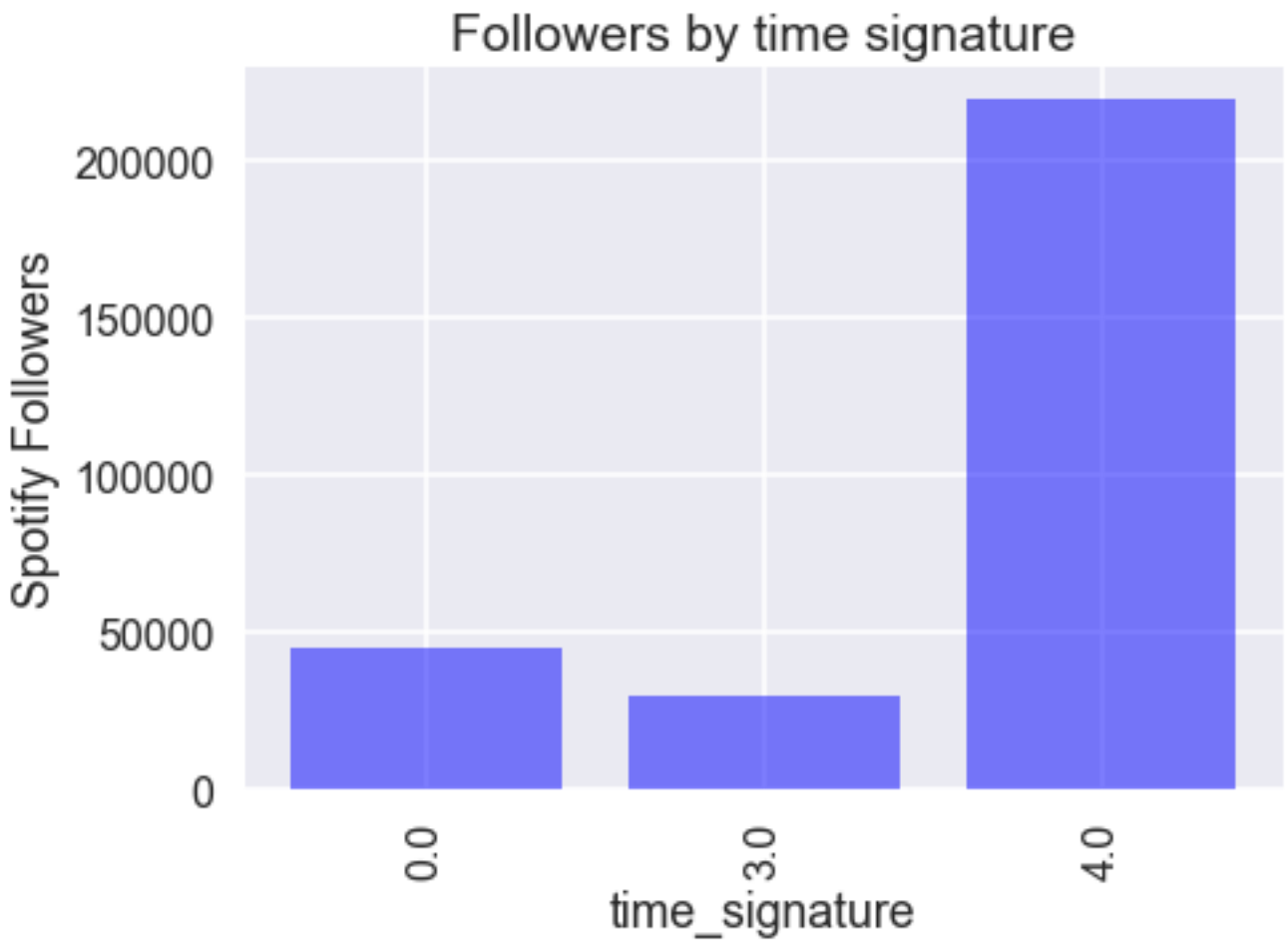
In [30]:

```
plt.figure(figsize=(7, 5))
bars('key','Followers by song key')
plt.show()
```



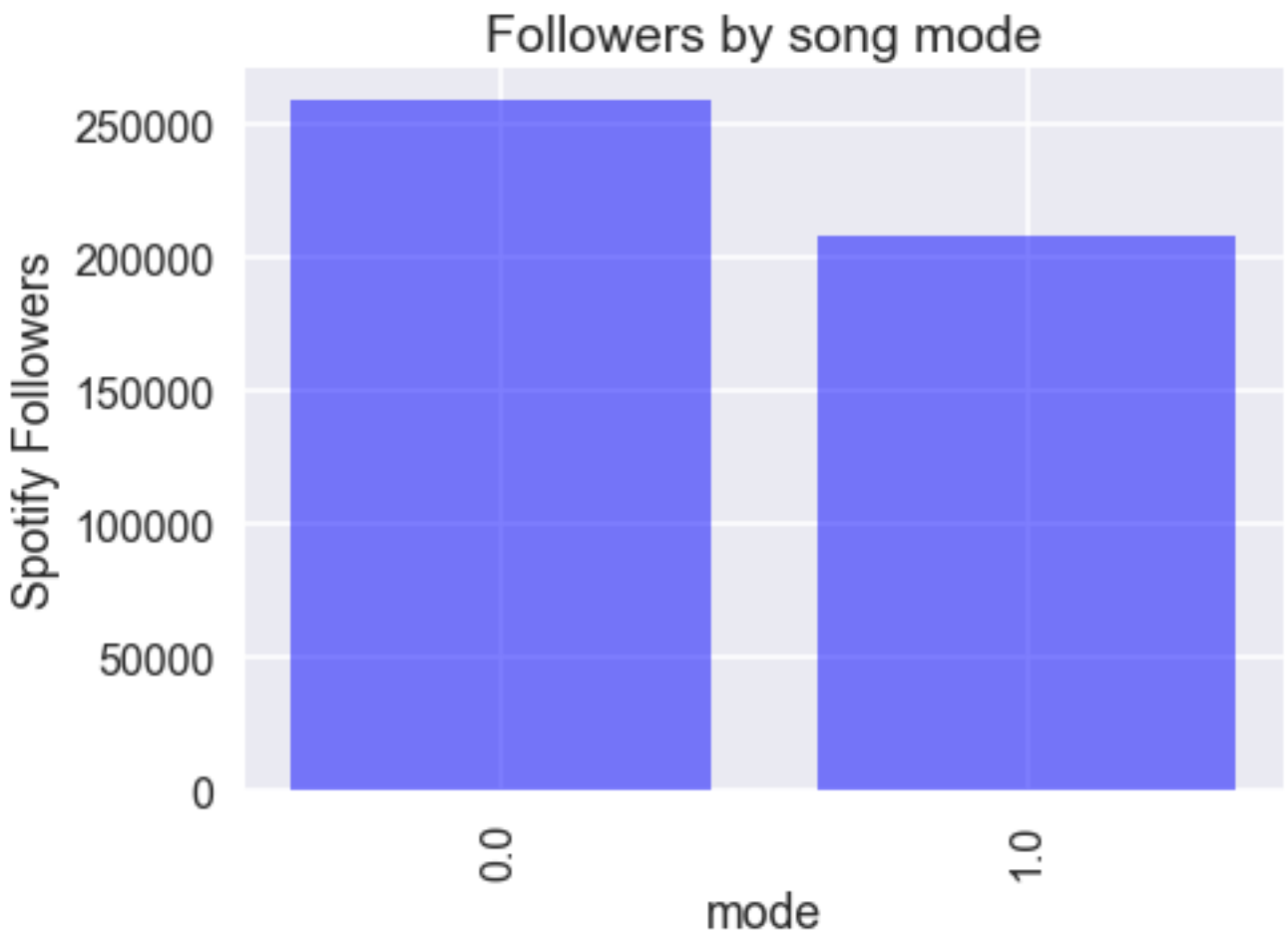
In [31]:

```
plt.figure(figsize=(7, 5))
bars('time_signature','Followers by time signature')
plt.show()
```



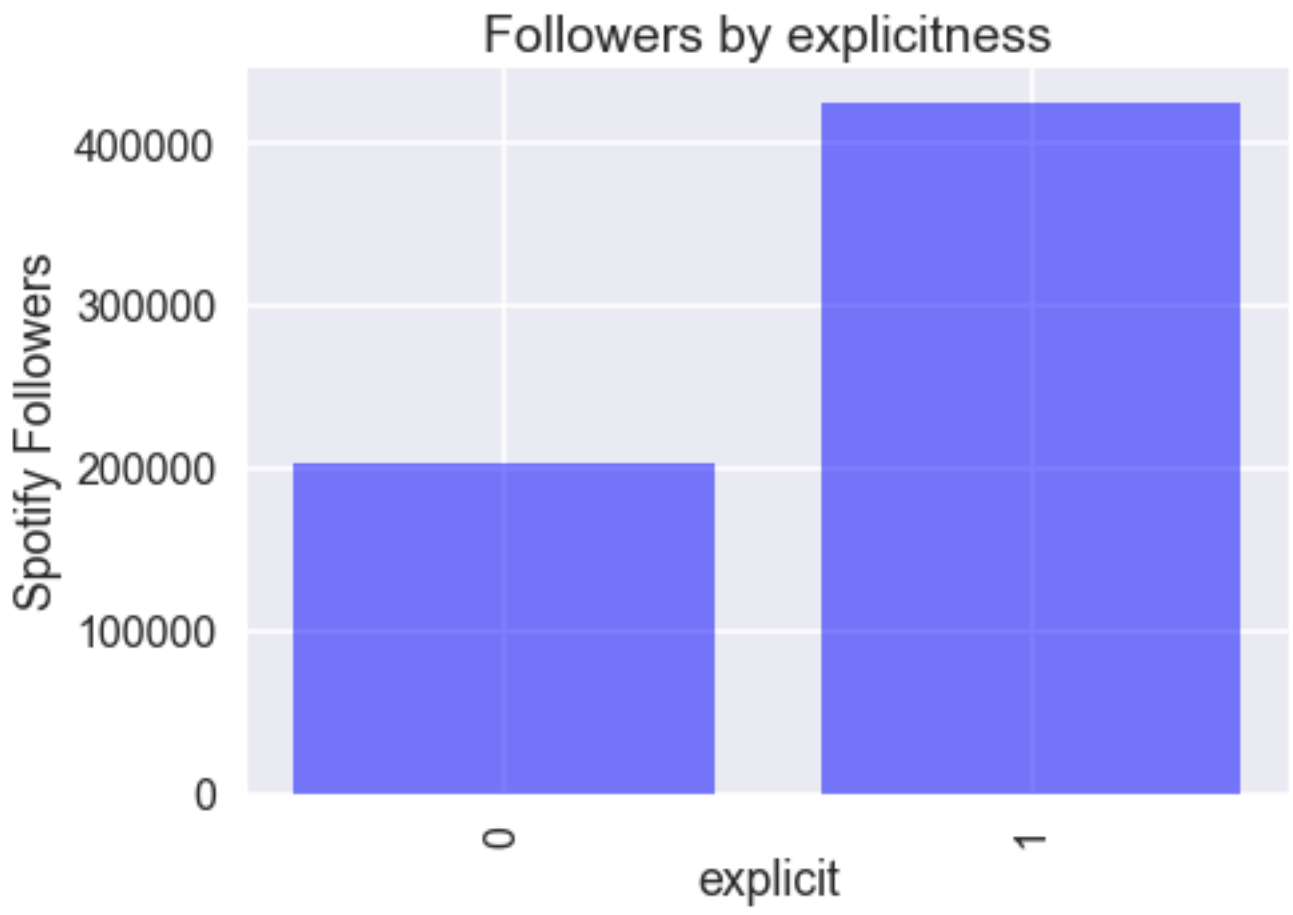
In [32]:

```
plt.figure(figsize=(7, 5))
bars('mode','Followers by song mode')
plt.show()
```



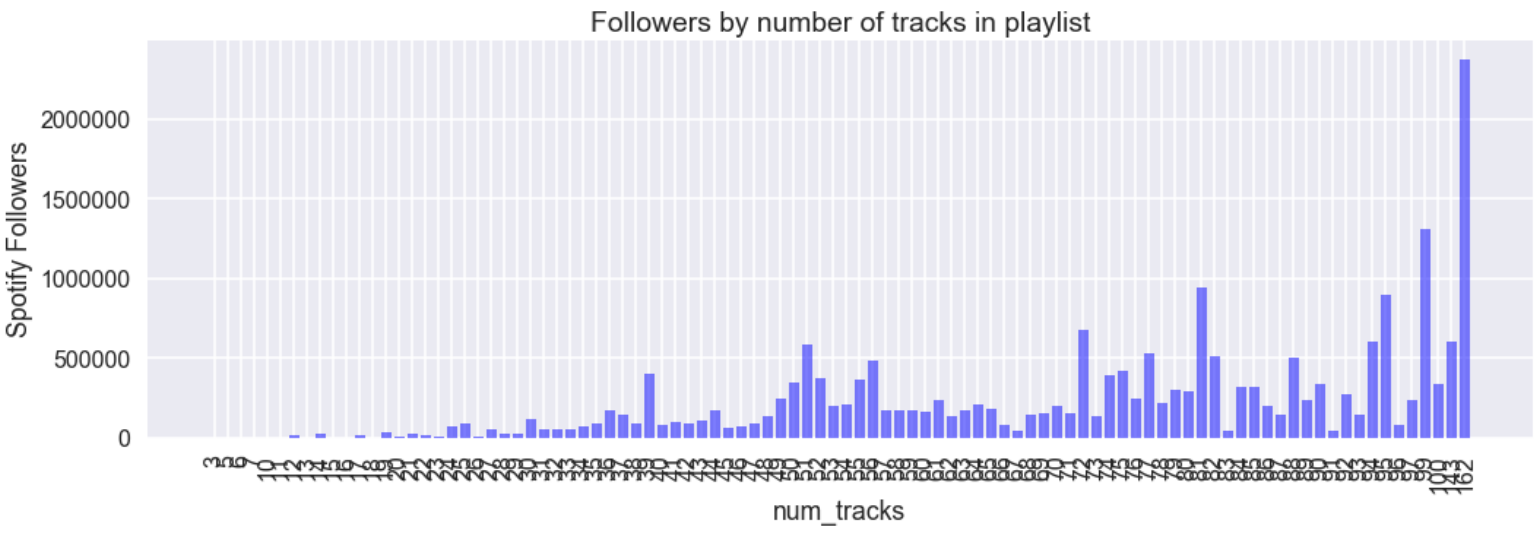
In [33]:

```
plt.figure(figsize=(7, 5))
bars('explicit','Followers by explicitness')
plt.show()
```



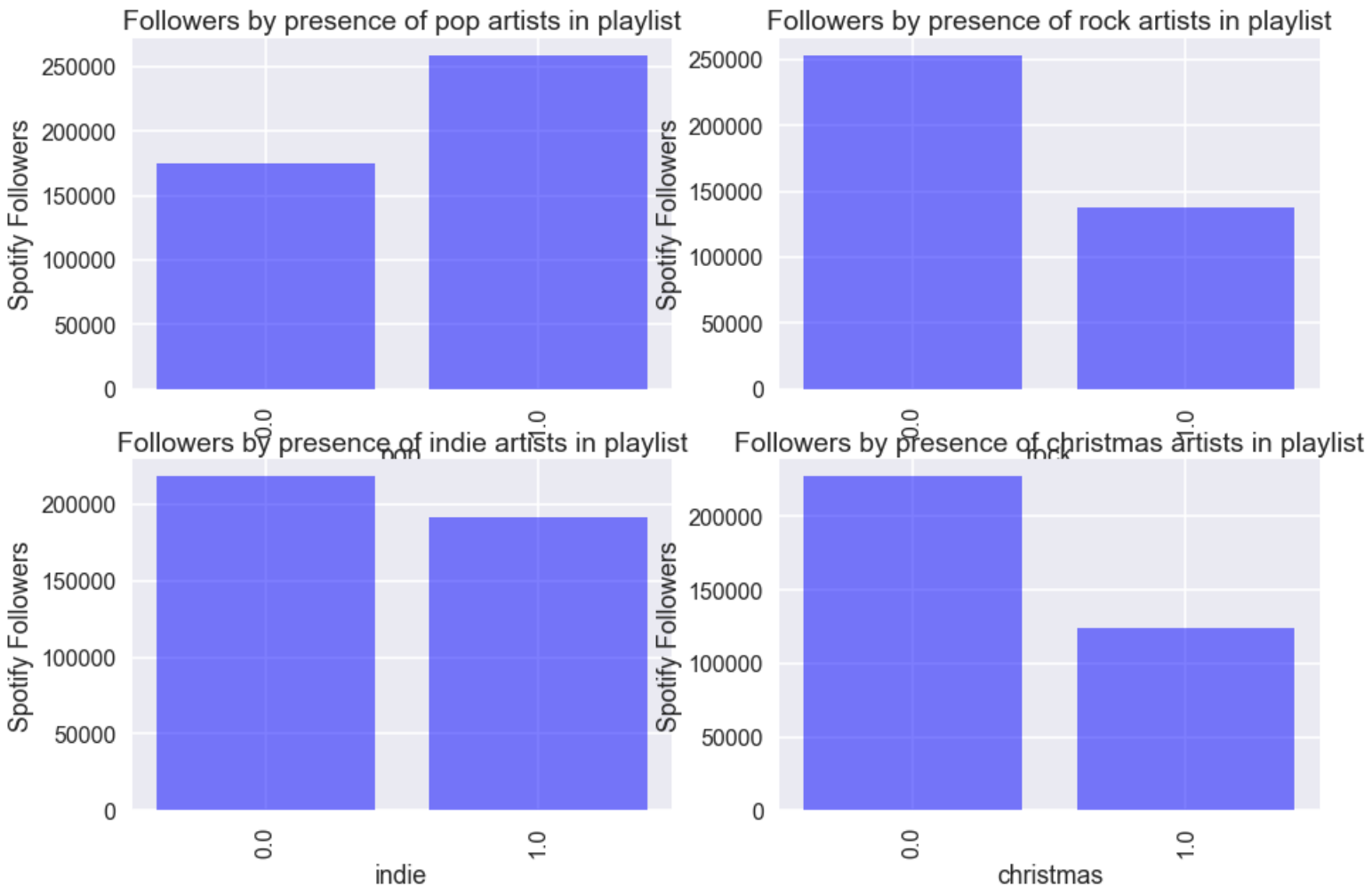
In [34]:

```
plt.figure(figsize=(17, 5))
bars('num_tracks','Followers by number of tracks in playlist')
plt.show()
```



In [35]:

```
plt.figure(figsize=(15,10))
plt.subplot(2,2,1)
bars('pop','Followers by presence of pop artists in playlist')
plt.subplot(2,2,2)
bars('rock','Followers by presence of rock artists in playlist')
plt.subplot(2,2,3)
bars('indie','Followers by presence of indie artists in playlist')
plt.subplot(2,2,4)
bars('christmas','Followers by presence of christmas artists in playlist')
plt.show()
```



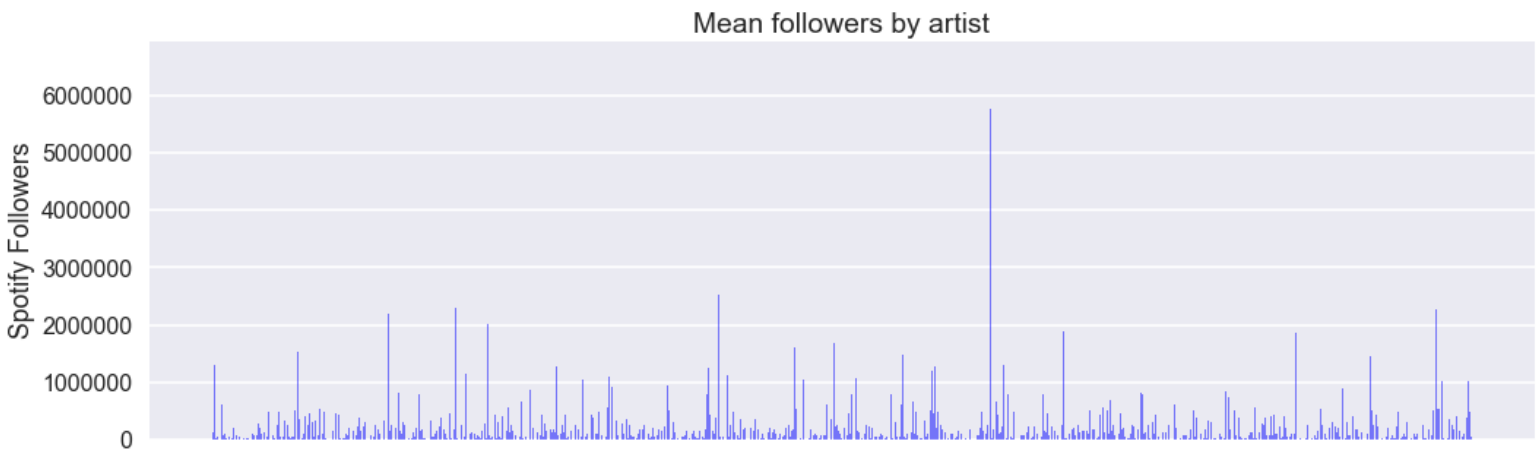
Mode and explicitness seem to impact the number of followers for a given playlist. Playlists with few tracks tend to have lower followers while playlists with lots of tracks tend to have more followers. Playlists with a majority of high tempo songs tend to have more followers. The number of followers also seems to vary with dominant song key in a given playlist.

Word Prevalence as predictor

We made indicator variables for the occurence specific words in track and album titles. First we established which words occurred most often in these titles and then determined which of these words seemed significant in defining whether a person would follow a playlist.

In [36]:

```
plt.figure(figsize=(17, 5))
bars('artist','Mean followers by artist',xlab=False,color='blue',data=data)
plt.show()
```



In [37]:

```
data[other_predictors].describe()
```

Out[37]:

	artist	album	description	playlist_name	track_name
count	85313	85313	85313	85313	85313
unique	20448	39240	1473	1628	52525
top	Various Artists	Spotify Sessions	0	New Music Friday	Home
freq	349	239	6092	162	71

In [38]:

```
def word_search(var,data=data):
    #print("Number of playlists mentioning "+str(var)+":",data['playlist_name'].
str.contains(var).sum())
    print("Number of track names mentioning "+str(var)+":",data['track_name'].st
r.contains(var).sum())
    #print("Number of descriptions mentioning "+str(var)+":",data['description']
.str.contains(var).sum())
    print("Number of ablum titles mentioning "+str(var)+":",data['album'].str.co
ntains(var).sum())
```

In [39]:

```
# For albums
word_search('Best')
word_search('best')
print('---')
word_search('Deluxe')
word_search('deluxe')
print('---')
word_search('feat.')
word_search('Feat.')
word_search('eat.')
print('---')
word_search('Remastered')
word_search('remastered')
print('---')
word_search('Remix')
word_search('remix')
```

```
Number of track names mentioning Best: 214
Number of ablum titles mentioning Best: 2219
Number of track names mentioning best: 12
Number of ablum titles mentioning best: 6
---
Number of track names mentioning Deluxe: 5
Number of ablum titles mentioning Deluxe: 2925
Number of track names mentioning deluxe: 0
Number of ablum titles mentioning deluxe: 0
---
Number of track names mentioning feat.: 2560
Number of ablum titles mentioning feat.: 1103
Number of track names mentioning Feat.: 114
Number of ablum titles mentioning Feat.: 101
Number of track names mentioning eat.: 3664
Number of ablum titles mentioning eat.: 3434
---
Number of track names mentioning Remastered: 1750
Number of ablum titles mentioning Remastered: 1269
Number of track names mentioning remastered: 2
Number of ablum titles mentioning remastered: 4
---
Number of track names mentioning Remix: 1348
Number of ablum titles mentioning Remix: 1071
Number of track names mentioning remix: 9
Number of ablum titles mentioning remix: 0
```

In [40]:

```
def common_words(data,N_top=25):
    print('Track Name')
    print(pd.Series(' '.join(data.loc[data.track_name !=0 ,'track_name']).lower(
).split()).value_counts()[:N_top]/len(data))
    print('--' *30)
    print('Album Name')
    print(pd.Series(' '.join(data.loc[data.album !=0 ,'album']).lower().split())
.value_counts()[:N_top]/len(data))
```

In [41]:

```
common_words(data)
```

```
Track Name
the           0.134118
-             0.131082
you           0.054974
in            0.051376
of            0.049348
i             0.043030
a             0.042936
me            0.038271
love          0.035786
to            0.033500
my            0.028155
(feat.        0.026186
on            0.025776
and           0.025670
version       0.024451
it            0.021884
for           0.021532
remastered    0.019458
&             0.016633
your          0.014992
is            0.013902
remix         0.013527
live          0.013363
from          0.013163
all           0.013023
dtype: float64
------------------------------------------------------------
Album Name
the         0.217845
of          0.086833
-           0.052735
&           0.034707
a           0.032469
in          0.031531
and         0.030992
you         0.026303
to          0.025963
```

```
best         0.025658
for          0.022798
version)     0.022365
love         0.021146
i            0.020700
edition)     0.017582
me           0.017406
/            0.016328
on           0.015390
(deluxe      0.014945
from         0.014582
(deluxe)     0.014558
hits         0.014066
my           0.013843
vol.         0.012999
ep           0.012894
dtype: float64
```

In [42]:

```
test = data.copy()
test['track_feat'] = data['track_name'].str.contains('Feat.')*1
test['track_remast'] = data['track_name'].str.contains('Remastered')*1
test['track_remix'] = data['track_name'].str.contains('Remix')*1
test['album_feat'] = data['track_name'].str.contains('Feat.')*1
test['album_deluxe'] = data['track_name'].str.contains('Deluxe')*1
test['album_best'] = data['track_name'].str.contains('Best')*1
```

In [43]:

```
word_features = ['playlist_name','track_feat','track_remast','track_remix','albu
m_feat','album_deluxe','album_best']
test = test[word_features]
word_df = test.groupby('playlist_name').agg(lambda x:x.value_counts().index[0])
word_df.head()
```

Out[43]:

	track_feat	track_remast	track_remix	album_feat	album_
playlist_name					
#FlashbackFriday	0	0	0	0	0
#Hotmess	0	0	0	0	0
#JazzAppreciationMonth	0	0	0	0	0
#MakeitHappen	0	0	0	0	0
#MetalMonday	0	0	0	0	0

In [44]:

```
word_df.sum(axis=0)
```

Out[44]:

```
track_feat       0
track_remast    13
track_remix      1
album_feat       0
album_deluxe     0
album_best       0
dtype: int64
```

In [45]:

```
word = (test.groupby('playlist_name').sum() > 0)*1
df_words = pd.concat([df,word],axis=1)
df_words.head()
```

Out[45]:

	acousticness	danceability	duration_ms	energy	instr
playlist_name					
#FlashbackFriday	0.154854	0.652320	253792.900000	0.673036	0.00
#Hotmess	0.036124	0.522000	214560.041667	0.790167	0.11
#JazzAppreciationMonth	0.878450	0.488650	244925.600000	0.199132	0.28
#MakeitHappen	0.110002	0.647433	220558.700000	0.704100	0.00
#MetalMonday	0.000478	0.365700	302320.466667	0.939467	0.33

5 rows × 49 columns

In [46]:

```
cat_predictors.append('num_tracks')
print(pd.get_dummies(df_words['time_signature']).sum())
print(pd.get_dummies(df_words['key']).sum())
```

```
0.0       4
3.0      38
4.0    1586
dtype: int64
0.0     351
1.0     268
2.0     192
3.0       8
4.0      67
5.0     121
6.0      56
7.0     238
8.0      40
9.0     158
10.0     40
11.0     89
dtype: int64
```

In [47]:

```
df_words = pd.concat([df_words,pd.get_dummies(df_words['time_signature'],prefix=
'time_sign')],axis=1)
df_words.drop('time_signature',axis=1,inplace=True)
df_words = pd.concat([df_words,pd.get_dummies(df_words['key'],prefix='key')],axi
s=1)
df_words.drop('key',axis=1,inplace=True)

# removing playlist with zero followers, some of these playlists don't seem like
they fit the population
# they're very specific and it seems like people would listen but not FOLLOW the
m
df_words = df_words[df_words.followers!=0] # removing
```

Saving as CSV (ONLY RERUN if want to overwrite CSV)

In [48]:

```
df_words.to_csv('df_by_playlist.csv')
```

GBRTree Model

In [49]:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
```

Data prep

1. load data
2. train test split
3. interaction terms
4. take natural log of followers

In [50]:

```
data = pd.read_csv('df_by_playlist.csv',index_col=0)
```

In [51]:

```
names = list(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'popularity', 'speechiness',
       'tempo', 'valence', 'explicit', 'mode', 'num_tracks',
       'track_feat', 'track_remast', 'track_remix', 'album_feat',
       'album_deluxe', 'album_best', 'time_sign_0.0', 'time_sign_3.0',
       'time_sign_4.0', 'key_0.0', 'key_1.0', 'key_2.0', 'key_3.0', 'key_4.0',
       'key_5.0', 'key_6.0', 'key_7.0', 'key_8.0', 'key_9.0', 'key_10.0','key_11
.0'])
```

In [52]:

```
# splitting data
xtrain, xtest, ytrain, ytest = train_test_split(data.drop('followers',axis=1),da
ta['followers'],test_size=0.2,random_state=1)
```

In [53]:

```
xtrain.head()
```

Out[53]:

	acousticness	danceability	duration_ms	energy	instrumentalness
playlist_name					
This Is: Schumann	0.980080	0.282758	406096.500000	0.074308	0.666462
Might and Myth: Power Metal	0.000698	0.415224	315269.836735	0.933776	0.080616
Indie Rock Brain Boost	0.119103	0.567800	208275.480000	0.781240	0.062042
2004	0.182080	0.691000	237680.780000	0.684580	0.001216
Alternative Beats	0.118359	0.594626	224036.032967	0.709527	0.079324

5 rows × 61 columns

In [54]:

```
quant_predictors = list([predictor for predictor in names if xtrain.loc[:,predic
tor].unique().sum()!=1])

def interact(x,variable,predictors=quant_predictors):
    for predictor in predictors:
        x[variable+'_'+predictor] = x[variable]*x[predictor]
    return x
```

In [55]:

```
# using explicit interaction terms (even though GBRT sort of already can detect
these)
#from sklearn.preprocessing import PolynomialFeatures
#poly = PolynomialFeatures(degree=2)
#poly.fit_transform()

xtrain = interact(xtrain,'loudness')
xtrain = interact(xtrain,'danceability',list(['speechiness','tempo']))
xtest = interact(xtest,'loudness')
xtest = interact(xtest,'danceability',list(['speechiness','tempo']))
```

In [56]:

```
# Taking log of response variable to deal with outliers (right skewed distributi
on)
# Otherwise, since each progressive model fit focuses on steepest gradient, we w
ould always favor fitting these outliers
# Also don't want to worry about whether train or test set gets most extreme out
lier
ytrain = np.log(ytrain)
ytest = np.log(ytest)
```

Gradient Boosted Regression Model

In [57]:

```
# Gradient Boosted Regression
# learning rate: want to get smaller as get closer to "bottom"
# FITTING STUMPs, if more than 1 predictor - need more depth
#also learning rate =1, just using gradient, if learning rate<1 you're slowing y
ourself down by multiplying "residual" by fraction
```

In [58]:

```
def deviance_plot(est, X_test, y_test, ax=None, label='', train_color='#2c7bb6',
                  test_color='#d7191c', alpha=1.0, ylim=(0, 10)):
    """Deviance plot for ``est``, use ``X_test`` and ``y_test`` for test error.
"""
    n_estimators = len(est.estimators_)
    test_dev = np.empty(n_estimators)

    for i, pred in enumerate(est.staged_predict(X_test)):
        test_dev[i] = est.loss_(y_test, pred) # can put in any loss you want (MS
E, etc.)

    if ax is None:
        fig = plt.figure()
        ax = plt.gca()

    ax.plot(np.arange(n_estimators) + 1, test_dev, color=test_color, label='Test
%s' % label,
            linewidth=2, alpha=alpha)
    ax.plot(np.arange(n_estimators) + 1, est.train_score_, color=train_color,
            label='Train %s' % label, linewidth=2, alpha=alpha)
    ax.set_ylabel('Error')
    ax.set_xlabel('n_estimators')
    ax.set_ylim(ylim)
    return test_dev, ax
```

Fitting for learning rate, max_depth, min_samples_leaf, max_features

In [59]:

```
param_grid = {'learning_rate': [0.1,0.01],
              'max_depth': [4,8,12],
              'min_samples_leaf': [3,5,7],
              'max_features': [0.3,0.6]}
```

In [60]:

```
gb = GradientBoostingRegressor(n_estimators=600, loss='huber',random_state=1) # 
huber weights outliers less
gb_cv = GridSearchCV(gb, param_grid, cv=3, n_jobs=-1)
gb_cv.fit(xtrain, ytrain)
```

Out[60]:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=GradientBoostingRegressor(alpha=0.9, criterion='fri
edman_mse', init=None,
             learning_rate=0.1, loss='huber', max_depth=3,
             max_features=None, max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=600,
             presort='auto', random_state=1, subsample=1.0, verbose=
0,
             warm_start=False),
       fit_params=None, iid=True, n_jobs=-1,
       param_grid={'learning_rate': [0.1, 0.01], 'max_depth': [4, 8,
12], 'min_samples_leaf': [3, 5, 7], 'max_features': [0.3, 0.6]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

In [61]:

```
gb_cv.best_estimator_
```

Out[61]:

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=
None,
             learning_rate=0.01, loss='huber', max_depth=12,
             max_features=0.3, max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=7, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=600,
             presort='auto', random_state=1, subsample=1.0, verbose=
0,
             warm_start=False)
```

In [62]:

```
bp = gb_cv.best_params_
print('Best Params: \n',bp)
print('Score: ',gb_cv.score(xtrain, ytrain))
print('MSE: ',mean_squared_error(ytrain, gb_cv.predict(xtrain)))
print('MAE: ',mean_absolute_error(ytrain, gb_cv.predict(xtrain)))
```

```
Best Params:
 {'learning_rate': 0.01, 'max_depth': 12, 'max_features': 0.3, 'min_
samples_leaf': 7}
Score:  0.971840005606
MSE:  0.256314964823
MAE:  0.216755414681
```

Now fitting for just learning rate using 5-fold CV - want to avoid overfitting

In [63]:

```
param_grid2 = {'learning_rate': [0.1,0.01,0.001]}
gb2 = GradientBoostingRegressor(n_estimators=1000,
                                loss="huber",
                                max_depth=bp['max_depth'],
                                max_features=bp['max_features'],
                                min_samples_leaf=bp['min_samples_leaf'],random_s
tate=1)
gb2_cv = GridSearchCV(gb2,param_grid2,cv=5)
gb2_cv.fit(xtrain,ytrain)
```

Out[63]:

```
GridSearchCV(cv=5, error_score='raise',
       estimator=GradientBoostingRegressor(alpha=0.9, criterion='fri
edman_mse', init=None,
             learning_rate=0.1, loss='huber', max_depth=12,
             max_features=0.3, max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=7, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=1000,
             presort='auto', random_state=1, subsample=1.0, verbose=
0,
             warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'learning_rate': [0.1, 0.01, 0.001]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

In [64]:

```
gb2_cv.best_estimator_
```

Out[64]:

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=
None,
             learning_rate=0.01, loss='huber', max_depth=12,
             max_features=0.3, max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=7, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=1000,
             presort='auto', random_state=1, subsample=1.0, verbose=
0,
             warm_start=False)
```

In [65]:

```
bp2 = gb2_cv.best_params_
print('Best Params: \n',bp2)
print('Score: ',gb2_cv.score(xtrain, ytrain))
print('MSE: ',mean_squared_error(ytrain, gb2_cv.predict(xtrain)))
print('MAE: ',mean_absolute_error(ytrain, gb2_cv.predict(xtrain)))
```

```
Best Params:
 {'learning_rate': 0.01}
Score:  0.978977903172
MSE:  0.191345137845
MAE:  0.13786446553
```

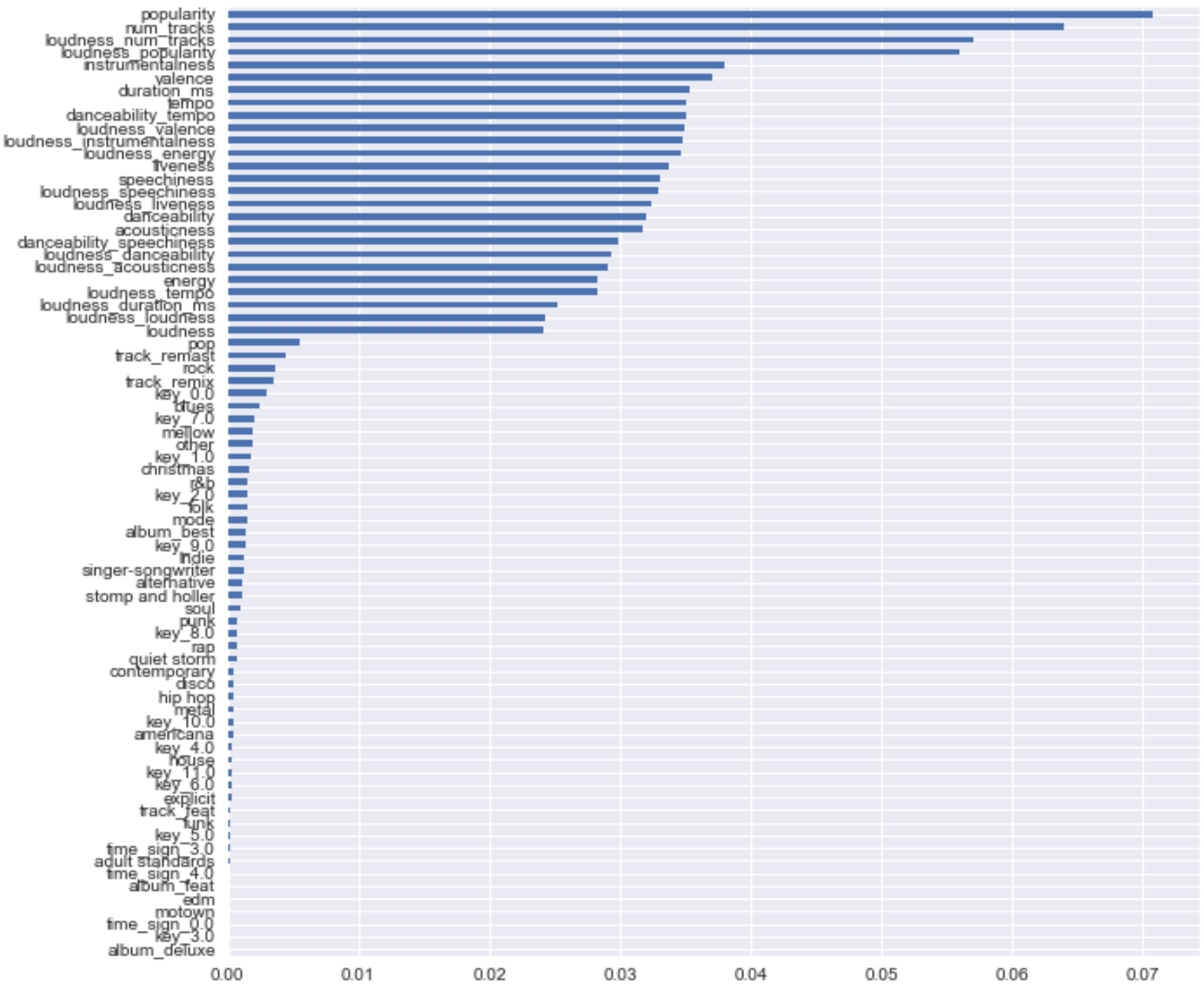
In [66]:

```
sns.set()
```

In [67]:

```
names_wInteraction = list(xtrain.keys().values)
indicators = list([predictor for predictor in names_wInteraction if xtrain.loc[:
,predictor].unique().sum()==1])

print('Relative importance of all variables')
feature_imp = gb2_cv.best_estimator_.feature_importances_
plt.figure(figsize=(10,10))
pd.Series(feature_imp, index=names_wInteraction).sort_values().plot(kind="barh")
;
```

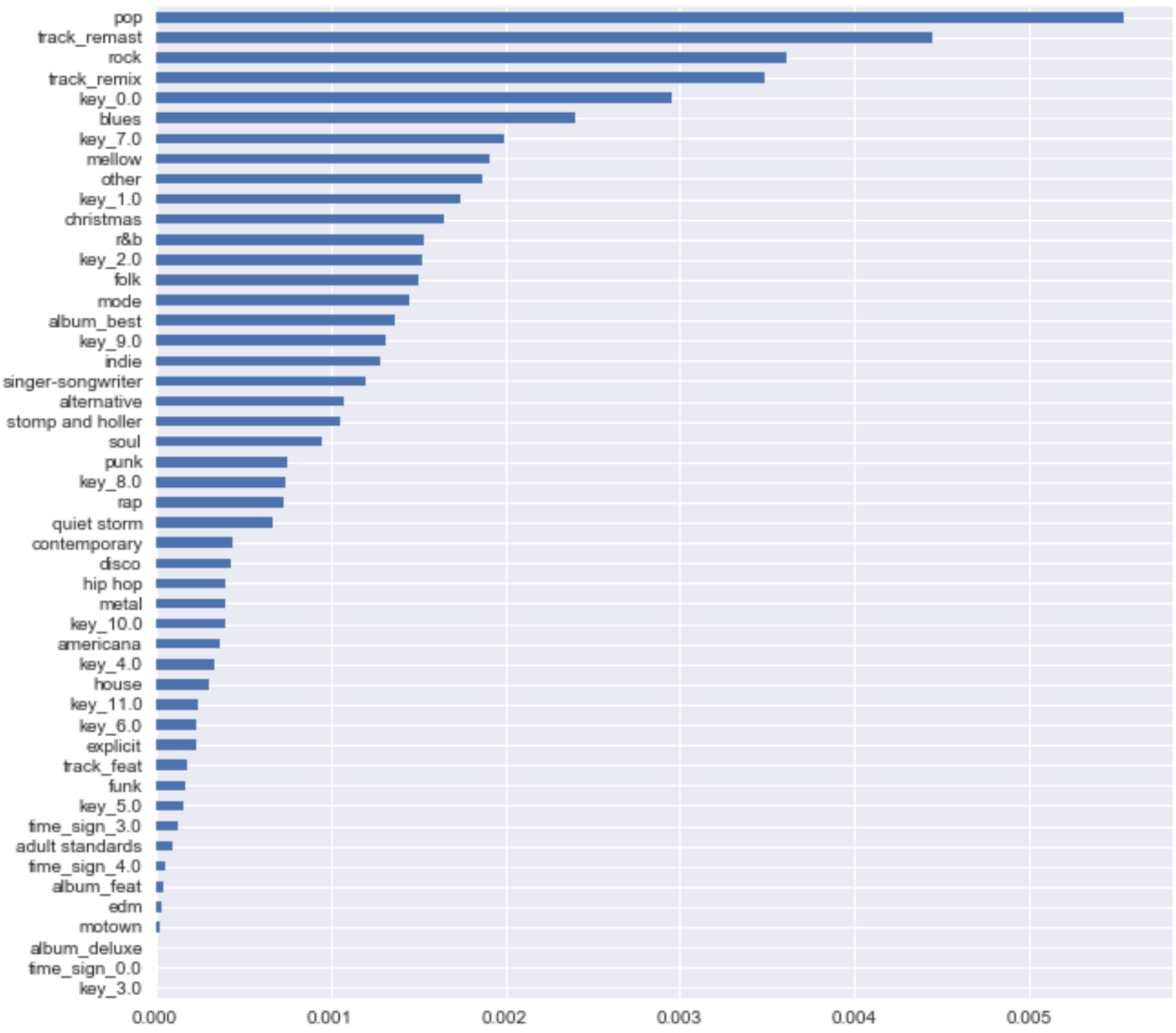
Relative importance of all variables



In [68]:

```
print('Relative importance of indicator variables')
plt.figure(figsize=(10,10))
pd.Series(feature_imp[feature_imp<0.01],index=indicators).sort_values().plot(kin
d='barh');
```

```
Relative importance of indicator variables
```

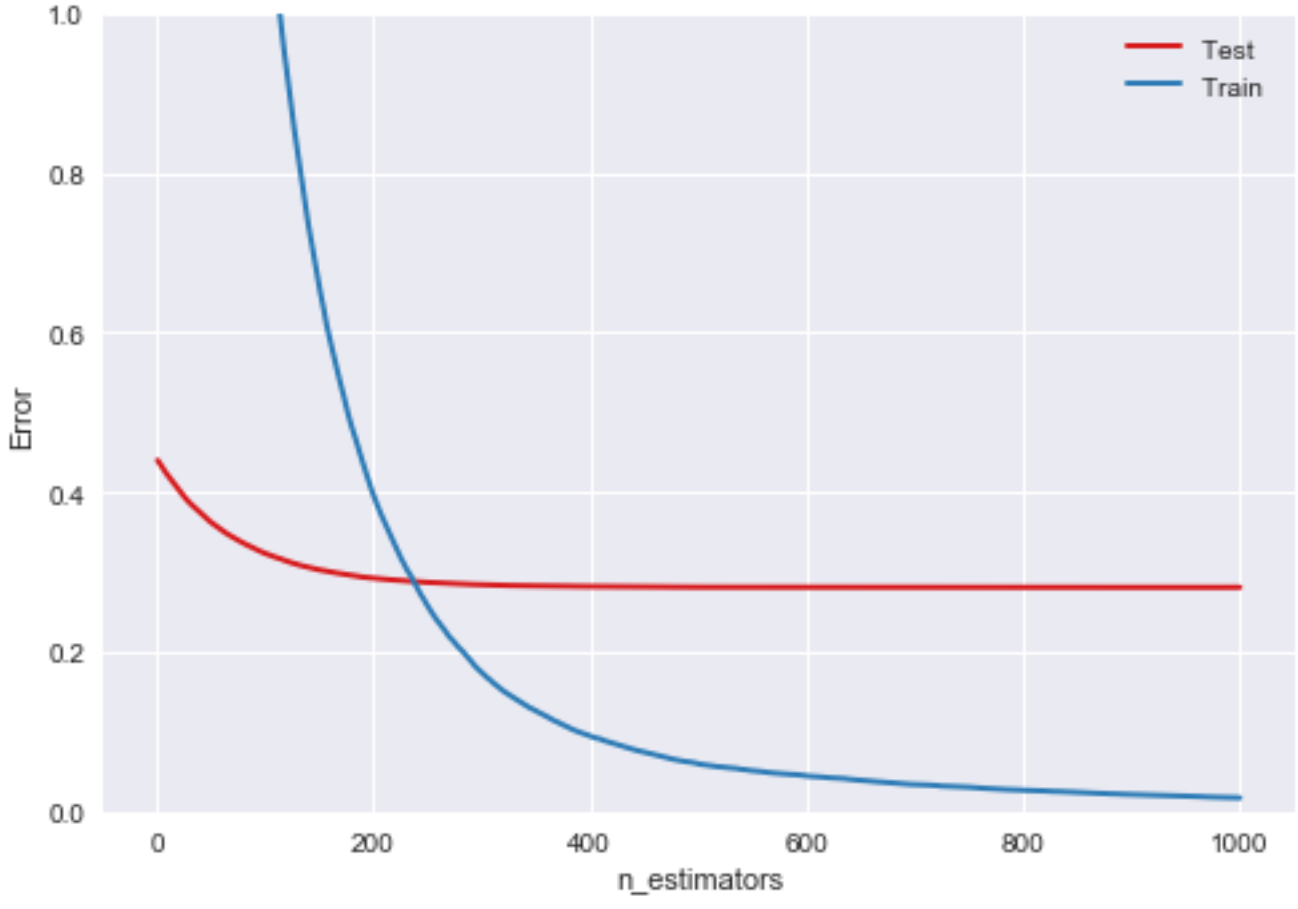


In [ ]:

GBRTree Model Evaluation

In [69]:

```
deviance_plot(gb2_cv.best_estimator_, xtest, ytest,ylim=(0,1));
plt.legend();
```



In [70]:

```
# Cross validation score
gbCV = GradientBoostingRegressor(n_estimators=1000,
                                 loss="huber", learning_rate=bp2['learning_rate']
,
                                 max_depth=bp['max_depth'],
                                 max_features=bp['max_features'],
                                 min_samples_leaf=bp['min_samples_leaf'],random_s
tate=1)
gbCV_score = cross_val_score(gbCV,xtrain,ytrain)
```

In [71]:

```
print('Cross validation scores: ', gbCV_score)
print('Mean CV score: ', gbCV_score.mean())
```

```
Cross validation scores:  [ 0.56443524  0.58574286  0.49900643]
Mean CV score:  0.549728176777
```

In [72]:

```
plt.figure(figsize=(8,6))
plt.scatter(ytrain, gb_cv.predict(xtrain)-ytrain,alpha=0.7,label='Fitting all pa
rams')
plt.scatter(ytrain, gb2_cv.predict(xtrain)-ytrain,alpha=0.7,label='After learnin
g rate re-fit')
plt.axhline(c='red')
plt.xlabel('True # Log Followers')
plt.ylabel('Residual')
plt.title('GBRTree Model Residuals for training set')
plt.legend();
```

MLP Model

In [73]:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import cross_val_score
```

Data Prep

1. Load data
2. Interaction terms and log(followers)
3. Standardize predictors

In [74]:

```
# Load data
data = pd.read_csv('df_by_playlist.csv',index_col=0)
# Train/test split
xtrain, xtest, ytrain, ytest = train_test_split(data.drop('followers',axis=1),da
ta['followers'],test_size=0.2,random_state=1)
```

In [75]:

```
# Interaction terms
names = list(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'popularity', 'speechiness',
       'tempo', 'valence', 'explicit', 'mode', 'num_tracks',
       'track_feat', 'track_remast', 'track_remix', 'album_feat',
       'album_deluxe', 'album_best', 'time_sign_0.0', 'time_sign_3.0',
       'time_sign_4.0', 'key_0.0', 'key_1.0', 'key_2.0', 'key_3.0', 'key_4.0',
       'key_5.0', 'key_6.0', 'key_7.0', 'key_8.0', 'key_9.0', 'key_10.0','key_11
.0'])

quant_predictors = list([predictor for predictor in names if xtrain.loc[:,predic
tor].unique().sum()!=1])

def interact(x,variable,predictors=quant_predictors):
    for predictor in predictors:
        x[variable+'_'+predictor] = x[variable]*x[predictor]
    return x

xtrain = interact(xtrain,'loudness')
xtrain = interact(xtrain,'danceability',list(['speechiness','tempo']))
xtest = interact(xtest,'loudness')
xtest = interact(xtest,'danceability',list(['speechiness','tempo']))
ytrain = np.log(ytrain)
ytest = np.log(ytest)
```

In [76]:

```
# Standardizing predictors
scaler = StandardScaler()
scaler.fit(xtrain)
xtrain = pd.DataFrame(scaler.transform(xtrain),columns = list(xtrain.keys().valu
es),index = list(xtrain.index.values))
xtest = pd.DataFrame(scaler.transform(xtest),columns = list(xtest.keys().values)
,index = list(xtest.index.values))
xtrain.head()
```

Out[76]:

	acousticness	danceability	duration_ms	energy	instrumentalness	liv
This Is: Schumann	2.252434	-2.149302	0.787979	-2.216925	2.278406	-0.8
Might and Myth: Power Metal	-1.250416	-1.089451	0.268752	1.593815	-0.282559	0.23
Indie Rock Brain Boost	-0.826927	0.131290	-0.342900	0.917497	-0.363757	-0.2
2004	-0.601684	1.117000	-0.174799	0.488922	-0.629649	-0.4
Alternative Beats	-0.829588	0.345925	-0.252802	0.599535	-0.288209	-0.4

5 rows × 75 columns

Fitting MLP model - using CV for regularizing parameter

In [77]:

```
param_grid = {'alpha': [0.1,0.01,0.001,0.0001,0.00001]}

mlp = MLPRegressor(random_state=1,tol=0.001,max_iter=1000)
mlp_cv = GridSearchCV(mlp, param_grid, cv=3, n_jobs=-1)
mlp_cv.fit(xtrain, ytrain)
```

Out[77]:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=MLPRegressor(activation='relu', alpha=0.0001, batch
_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=1000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle
=True,
       solver='adam', tol=0.001, validation_fraction=0.1, verbose=Fa
lse,
       warm_start=False),
       fit_params=None, iid=True, n_jobs=-1,
       param_grid={'alpha': [0.1, 0.01, 0.001, 0.0001, 1e-05]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

In [78]:

```
mlp_cv.best_estimator_
```

Out[78]:

```
MLPRegressor(activation='relu', alpha=0.1, batch_size='auto', beta_1
=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=1000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle
=True,
       solver='adam', tol=0.001, validation_fraction=0.1, verbose=Fa
lse,
       warm_start=False)
```

In [79]:

```
print('Best Params: \n',mlp_cv.best_params_)
print('Score: ',mlp_cv.score(xtrain, ytrain))
print('MSE: ',mean_squared_error(ytrain, mlp_cv.predict(xtrain)))
print('MAE: ',mean_absolute_error(ytrain, mlp_cv.predict(xtrain)))
```

```
Best Params:
 {'alpha': 0.1}
Score:  0.895823144689
MSE:  0.948227710231
MAE:  0.668010547524
```

Adjusting other params

1. Number of hidden layers
2. activation function

In [80]:

```
param_grid = {'activation': ['identity','logistic','tanh','relu'],
              'hidden_layer_sizes': [(50,),(100,),(200,)]}

mlp2 = MLPRegressor(random_state=1,tol=0.001,max_iter=1000,alpha=0.1)
mlp2_cv = GridSearchCV(mlp2, param_grid, cv=3, n_jobs=-1)
mlp2_cv.fit(xtrain, ytrain)
```

Out[80]:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=MLPRegressor(activation='relu', alpha=0.1, batch_si
ze='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='constant',
       learning_rate_init=0.001, max_iter=1000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle
=True,
       solver='adam', tol=0.001, validation_fraction=0.1, verbose=Fa
lse,
       warm_start=False),
       fit_params=None, iid=True, n_jobs=-1,
       param_grid={'activation': ['identity', 'logistic', 'tanh', 'r
elu'], 'hidden_layer_sizes': [(50,), (100,), (200,)]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

In [81]:

```
print('Best Params: \n',mlp2_cv.best_params_)
print('Score: ',mlp2_cv.score(xtrain, ytrain))
print('MSE: ',mean_squared_error(ytrain, mlp2_cv.predict(xtrain)))
print('MAE: ',mean_absolute_error(ytrain, mlp2_cv.predict(xtrain)))
```

```
Best Params:
 {'activation': 'logistic', 'hidden_layer_sizes': (200,)}
Score:  0.522356645505
MSE:  4.34755553897
MAE:  1.58186544369
```

Model evaluation

In [82]:

```
mlpCV = MLPRegressor(random_state=1,tol=0.001,max_iter=1000,alpha=0.1,activation
='logistic',
                     hidden_layer_sizes=(200,))

mlpCV_score = cross_val_score(mlpCV,xtrain,ytrain)
```

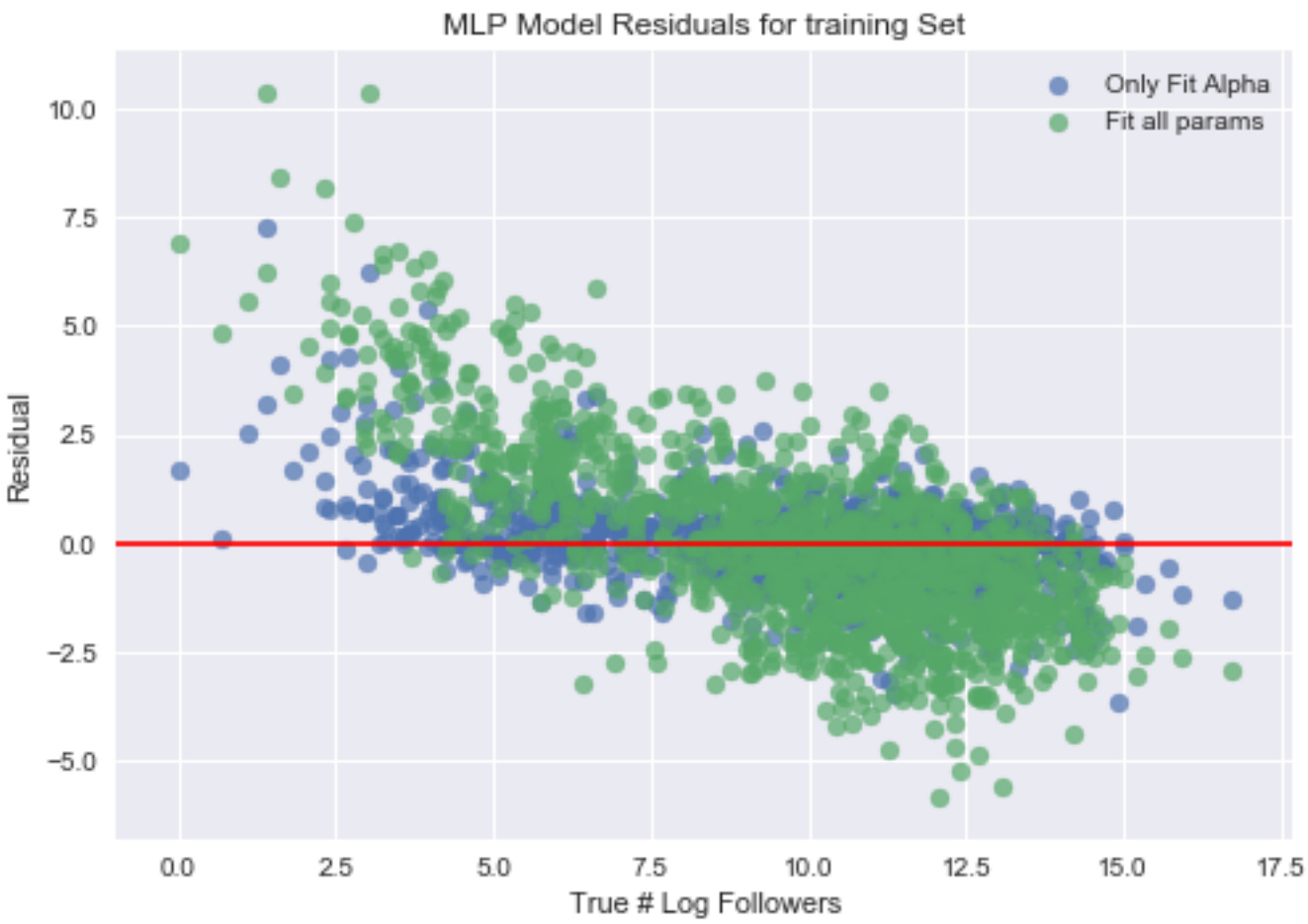
In [83]:

```
print('Cross validation scores: ', mlpCV_score)
print('Mean CV score: ', mlpCV_score.mean())
```

```
Cross validation scores:  [ 0.4332229   0.48759502  0.39099289]
Mean CV score:  0.437270268864
```

In [84]:

```
plt.scatter(ytrain, mlp_cv.predict(xtrain)-ytrain,alpha=0.7,label='Only Fit Alph
a')
plt.scatter(ytrain, mlp2_cv.predict(xtrain)-ytrain,alpha=0.7,label='Fit all para
ms')
plt.axhline(c='red')
plt.xlabel('True # Log Followers')
plt.ylabel('Residual')
plt.title('MLP Model Residuals for training Set')
plt.legend();
```



KNN Model

In [85]:

```
import sys
pd.set_option('display.max_rows', 999)
pd.set_option('display.width', 500)
pd.set_option('display.notebook_repr_html', True)
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import cross_val_score,KFold

import warnings
warnings.filterwarnings("ignore")

matplotlib.rcParams['figure.figsize'] = (10.0, 5.0)
```

Load and clean DataFrame

- Predict log of number of followers for each playlist.
- Standardize columns to be between 0 and 1.
- Take 80/20 test/train split
- Create interaction terms for loudness and danceability
- Remove playlists that have zero followers

In [86]:

```
data = pd.read_csv('df_by_playlist.csv',index_col=0)
data = data.loc[data['followers'] > 0]
y = np.log(data['followers'].values)
y = y.reshape(len(y),1)
names = list(data)
names.remove('followers')
```

Functions to create cross-terms for quantitative predictors and plot KNN results

In [87]:

```
quant_predictors = list([predictor for predictor in names if data.loc[:,predicto
r].unique().sum()!=1])

def interact(x,variable,predictors=quant_predictors):
    for predictor in predictors:
        x[variable+'_'+predictor] = x[variable]*x[predictor]
    return x

def plot_knn(knn_train_scores,knn_test_scores,best_k,title_ = ''):
    plt.plot(ks,knn_train_scores,label = 'Training score')
    plt.plot(ks,knn_test_scores,label = 'Testing score')
    plt.xlabel('K nearest neighbors')
    plt.ylabel('R^2 Score')
    plt.title('KNN Regression {}, Best k = {}'.format(title_,best_k))
    plt.legend()
    plt.show()
```

KNN Regression

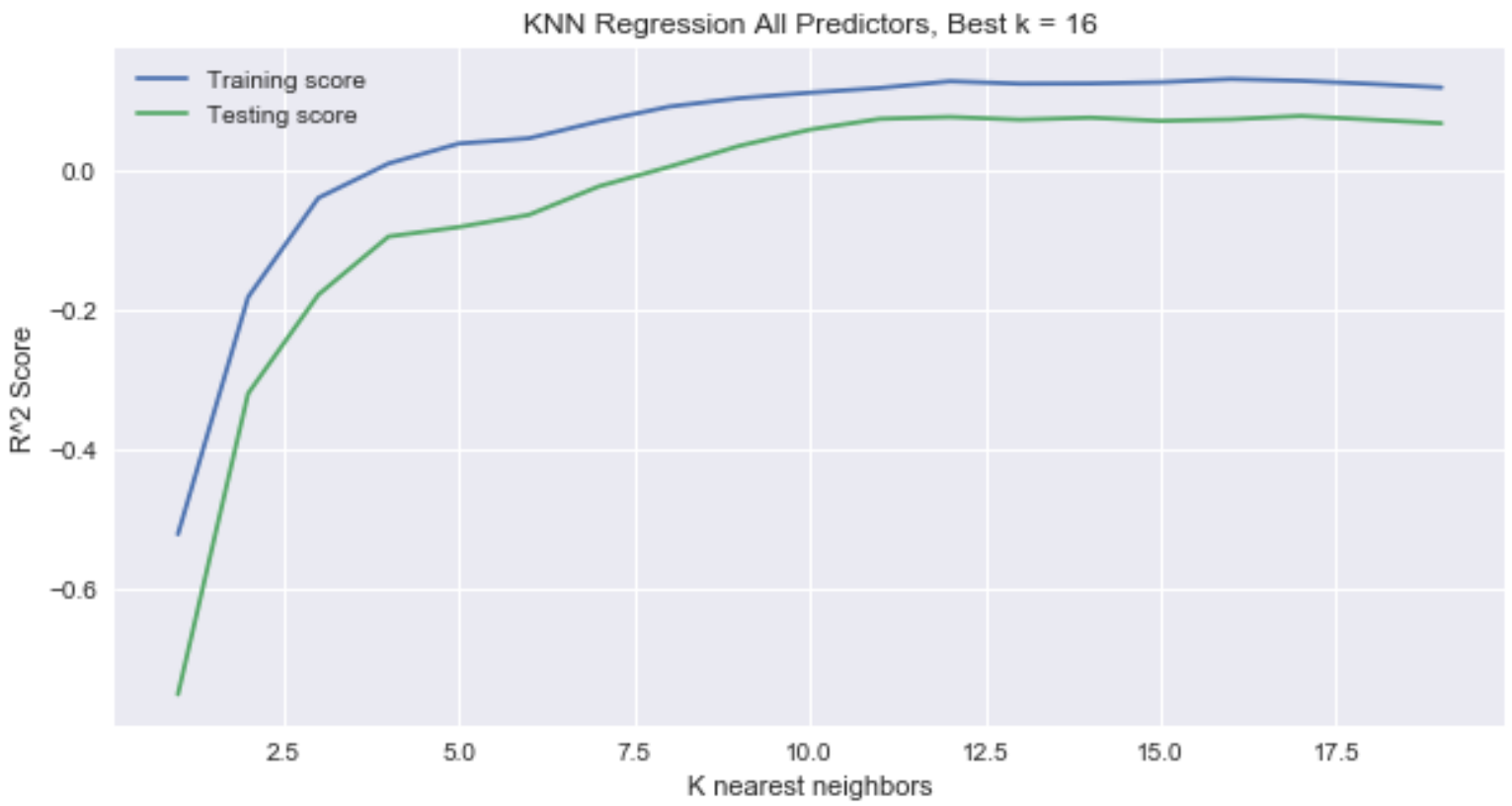
- Begin with baseline KNN using all attributes, including genres, to predict followers in a playlist
- Next use a se
- Finally, use a subset of predictors with interaction terms
- Use cross-validation for the training and testing sets
- We will use the R^2 score of the testing set as the determination of the best model

Regression with all predictors

In [88]:

```
X = data.copy()

# normalize and split
X = (X - X.min()) / (X.max() - X.min())
xtrain, xtest, ytrain, ytest = train_test_split(X,y,test_size=0.2,random_state =
1)
#xtrain = (xtrain - xtrain.min()) / (xtrain.max() - xtrain.min())
#xtest = (xtest - xtrain.min()) / (xtrain.max() - xtrain.min())

all_train_scores, all_test_scores = [],[]
ks = np.arange(1,20)

for k in ks:
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(xtrain,ytrain)
    kf = KFold(n_splits=3,random_state=29)
    train_scores = cross_val_score(knn,xtrain,ytrain,scoring="r2", cv=kf)
    test_scores = cross_val_score(knn,xtest,ytest,scoring="r2", cv=kf)
    all_train_scores.append(np.mean(train_scores))
    all_test_scores.append(np.mean(test_scores))

k_arg = np.argmax(all_train_scores)
best_k = ks[k_arg]
print('KNN Logistic Regression Training Score k = {}: {}'.
      format(best_k,all_train_scores[k_arg]))
print('KNN Logistic Regression Testing Score k = {}: {}\n'.
      format(best_k,all_test_scores[k_arg]))
plot_knn(all_train_scores,all_test_scores,best_k,'All Predictors')
```

```
KNN Logistic Regression Training Score k = 16: 0.13265345272682447
KNN Logistic Regression Testing Score k = 16: 0.0746517717064404
```

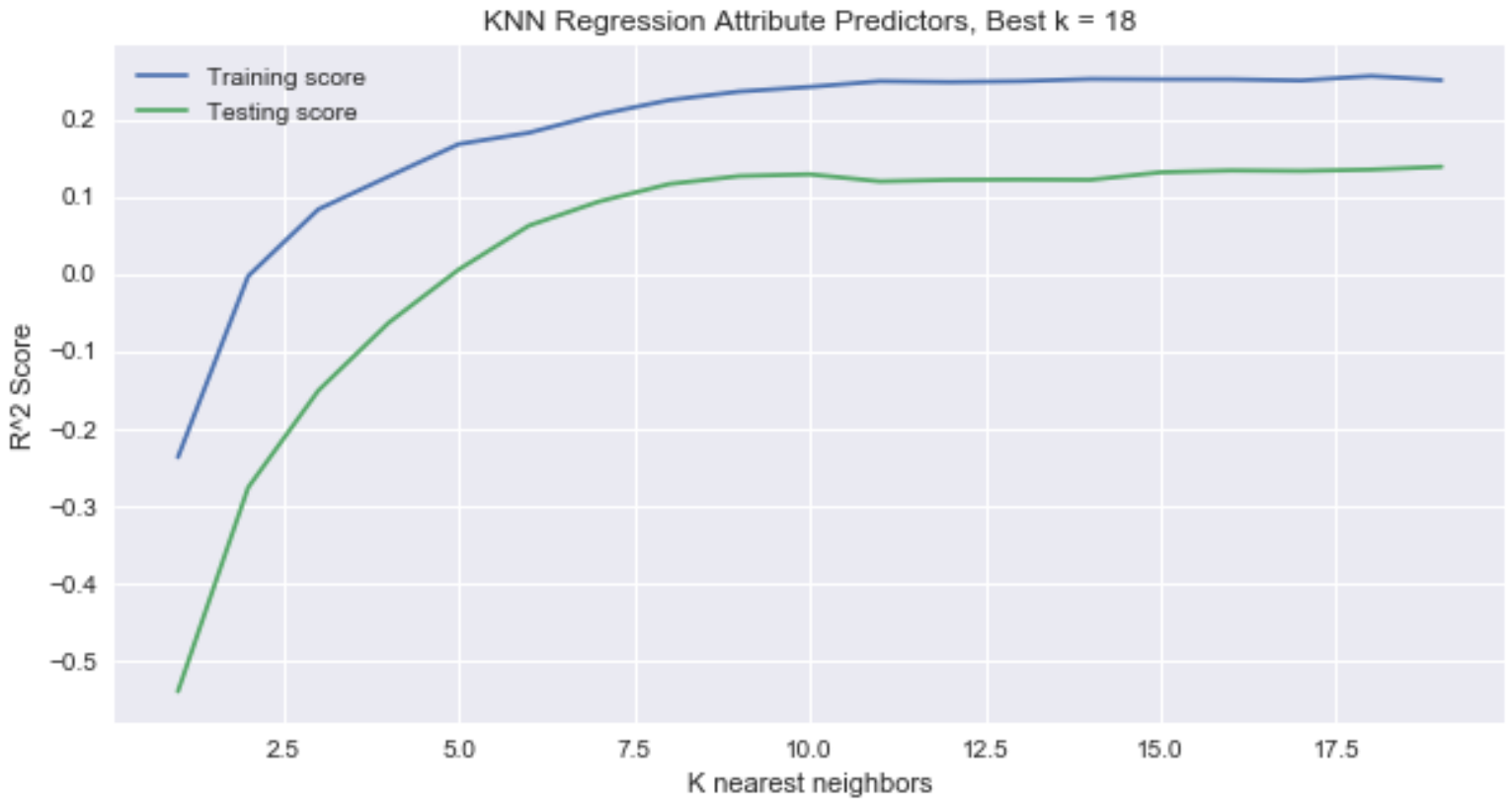


Including all columns gives a what looks like a poor R^2 score. We suspect our one-hot encoded columns for genre and key/time signature are not helping the regression. To test this, include only quantitative predictors for our next regression.

In [89]:

```
cols = ['acousticness','danceability','duration_ms','energy','instrumentalness',
'liveness',
 'loudness','popularity','speechiness','tempo','valence']
X = data[cols]

# normalize and split
X = (X - X.min()) / (X.max() - X.min())
xtrain, xtest, ytrain, ytest = train_test_split(X,y,test_size=0.2,random_state =
1)

sub_train_scores, sub_test_scores = [],[]
ks = np.arange(1,20)

for k in ks:
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(xtrain,ytrain)
    kf = KFold(n_splits=3,random_state=29)
    train_scores = cross_val_score(knn,xtrain,ytrain,scoring="r2", cv=kf)
    test_scores = cross_val_score(knn,xtest,ytest,scoring="r2", cv=kf)
    sub_train_scores.append(np.mean(train_scores))
    sub_test_scores.append(np.mean(test_scores))

k_arg = np.argmax(sub_train_scores)
best_k = ks[k_arg]
print('KNN Logistic Regression Training Score k = {}: {}'.
      format(best_k,sub_train_scores[k_arg]))
print('KNN Logistic Regression Testing Score k = {}: {}\n'.
      format(best_k,sub_test_scores[k_arg]))
plot_knn(sub_train_scores,sub_test_scores,best_k,'Attribute Predictors')
```

```
KNN Logistic Regression Training Score k = 18: 0.2553687282551847
KNN Logistic Regression Testing Score k = 18: 0.13435814191197218
```



Using only quantitative predictors improves the testing R^2 substatially. To further improve our model, let's add interaction terms.

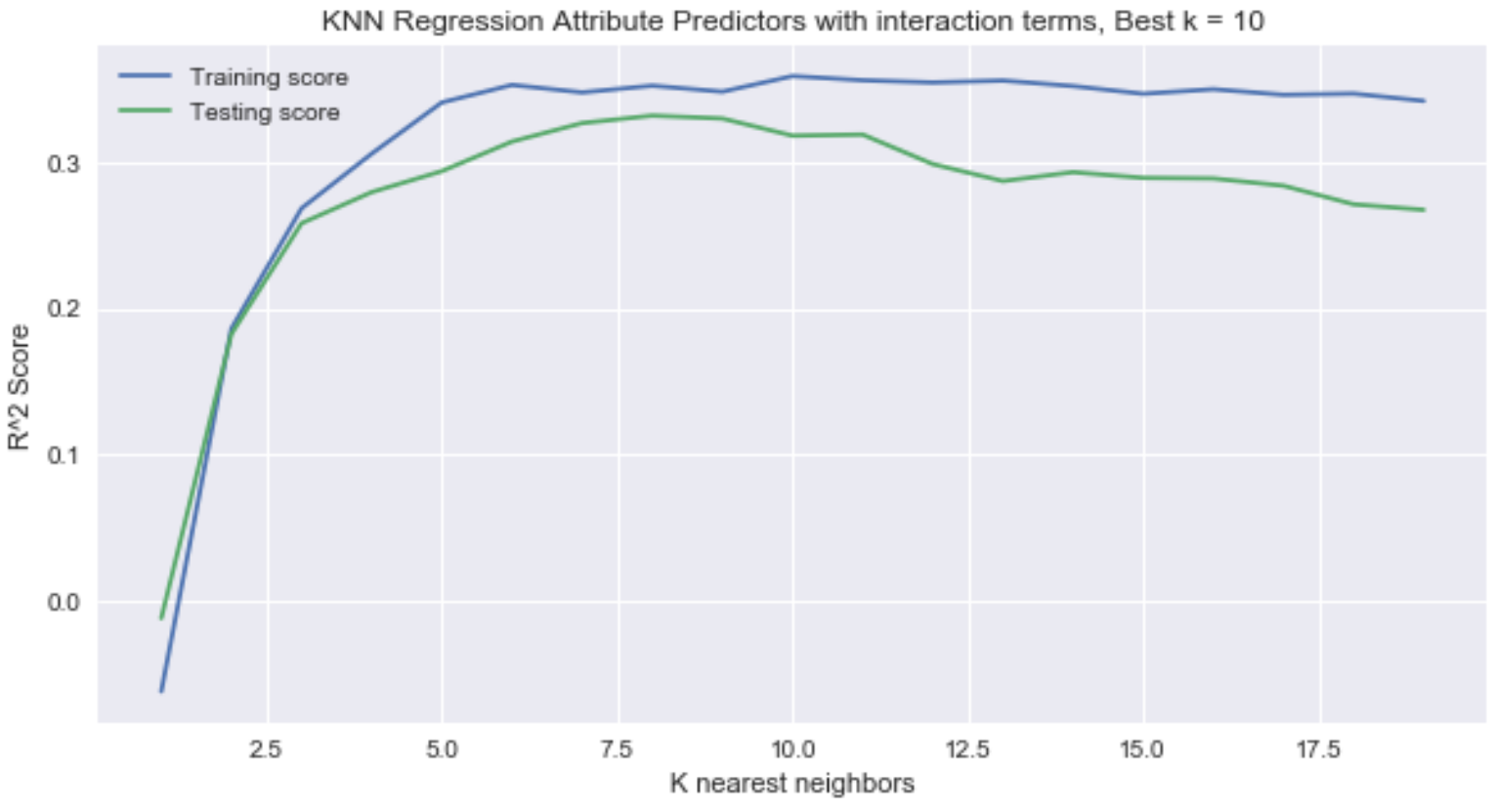
Interaction Terms

- Include interaction terms with loudness and danceability

In [90]:

```
drop_cols = ['pop','rock','folk','mellow','indie','rap','house','r&b','stomp and
holler','singer-songwriter','edm',
 'quiet storm','contemporary','christmas','funk','metal','alternative','adult st
andards','americana',
 'soul','blues','hip hop','motown','punk','disco','other','time_sign_0.0','time_
sign_3.0','time_sign_4.0',
 'key_0.0','key_1.0','key_2.0','key_3.0','key_4.0','key_5.0','key_6.0','key_7.0'
,'key_8.0','key_9.0','key_10.0',
 'key_11.0','track_feat','track_remast','track_remix','album_feat', 'album_delux
e','album_best','followers']

X = data.drop(drop_cols,1)
X = interact(X,'loudness')
X = interact(X,'danceability',list(['speechiness','tempo']))

# normalize and split
X = (X - X.min()) / (X.max() - X.min())
xtrain, xtest, ytrain, ytest = train_test_split(X,y,test_size=0.2,random_state =
29)

# split
int_train_scores, int_test_scores = [],[]
ks = np.arange(1,20)

for k in ks:
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(xtrain,ytrain)
    kf = KFold(n_splits=3,random_state=29)
    train_scores = cross_val_score(knn,xtrain,ytrain,scoring="r2", cv=kf)
    test_scores = cross_val_score(knn,xtest,ytest,scoring="r2", cv=kf)
    int_train_scores.append(np.mean(train_scores))
    int_test_scores.append(np.mean(test_scores))

k_arg = np.argmax(int_train_scores)
best_k = ks[k_arg]
print('KNN Logistic Regression Training Score k = {}: {}'.
      format(best_k,int_train_scores[k_arg]))
print('KNN Logistic Regression Testing Score k = {}: {}\n'.
      format(best_k,int_test_scores[k_arg]))
plot_knn(int_train_scores,int_test_scores,best_k,'Attribute Predictors with inte
raction terms')
```

```
KNN Logistic Regression Training Score k = 10: 0.3587650023982401
KNN Logistic Regression Testing Score k = 10: 0.318073224375351
```

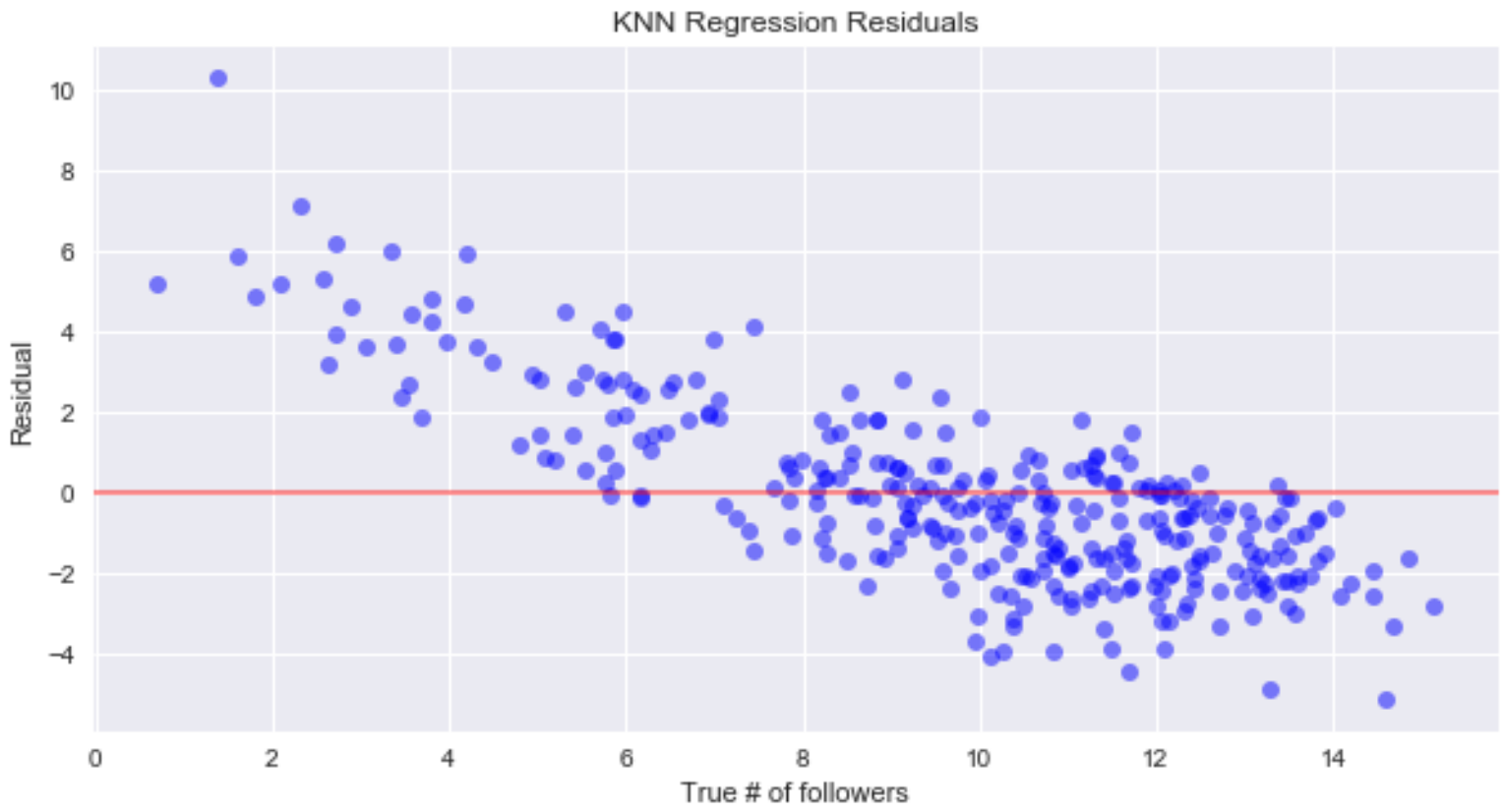


By adding interaction terms, our training R^2 score improves from 0.25 to 0.35. Finally, we'll look at the residuals in our prections.

Model Evaluation

In [91]:

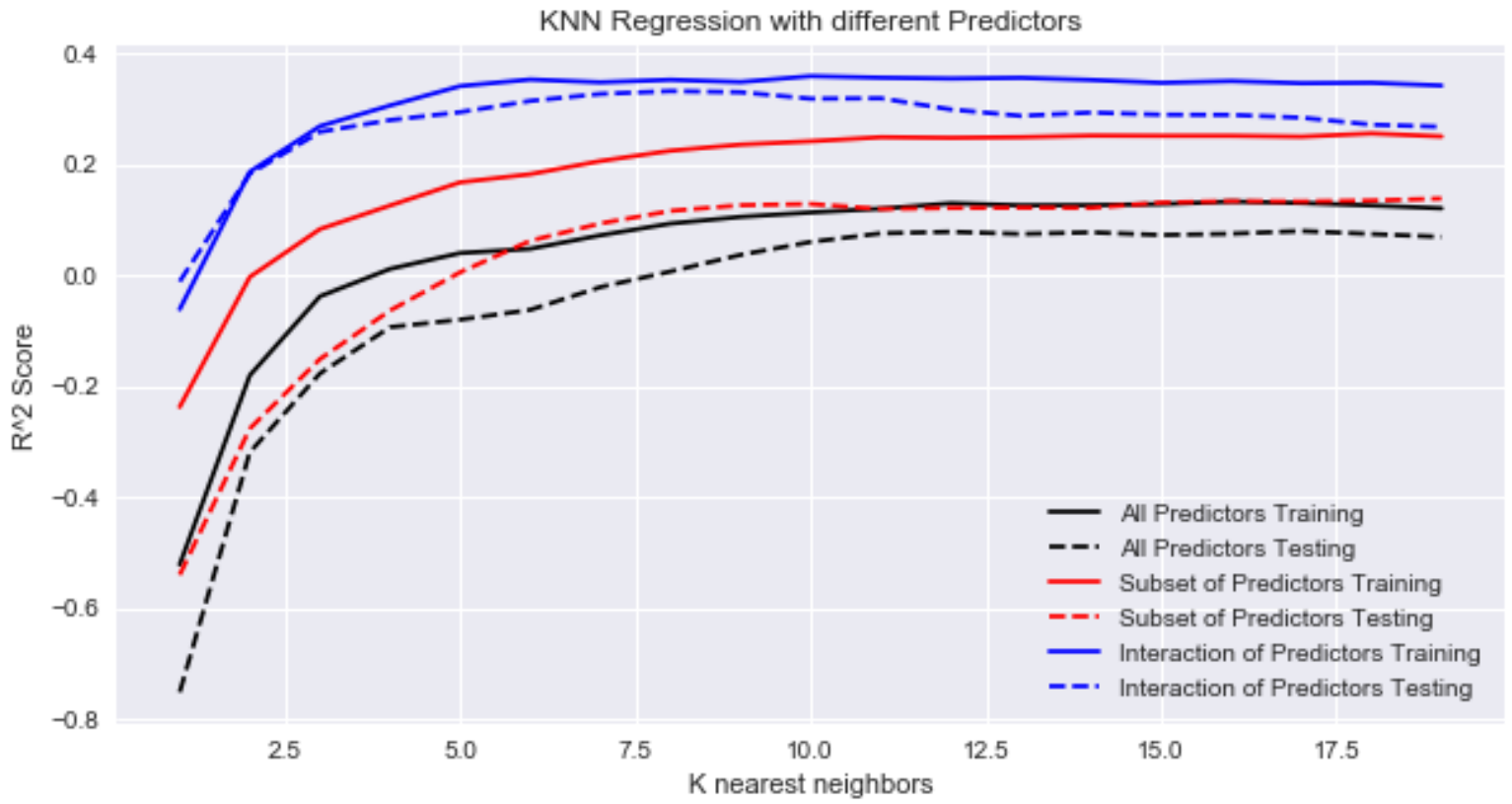
```
knn = KNeighborsRegressor(n_neighbors=best_k)
knn.fit(xtrain,ytrain)
plt.plot(ytest,knn.predict(xtest) - ytest,'bo',alpha = 0.5)
plt.axhline(0,color = 'r',alpha = 0.5)
plt.xlabel('True # of followers')
plt.ylabel('Residual')
plt.title('KNN Regression Residuals')
plt.savefig('KNN_RESIDUAL.png')
plt.show()
```



The model is overpredicting the number of followers for playlists with a small number of followers and underpredicting on playlists with a high number of followers.

In [92]:

```
# plot all training and testing scores

plt.plot(ks,all_train_scores,'k-',label = 'All Predictors Training')
plt.plot(ks,all_test_scores,'k--',label = 'All Predictors Testing')
plt.plot(ks,sub_train_scores,'r-',label = 'Subset of Predictors Training')
plt.plot(ks,sub_test_scores,'r--',label = 'Subset of Predictors Testing')
plt.plot(ks,int_train_scores,'b-',label = 'Interaction of Predictors Training')
plt.plot(ks,int_test_scores,'b--',label = 'Interaction of Predictors Testing')
plt.xlabel('K nearest neighbors')
plt.ylabel('R^2 Score')
plt.title('KNN Regression with different Predictors')
plt.legend()
plt.savefig('KNN_REGRESSION.png')
plt.show()
```



In [ ]:

In [ ]:

Ridge Regression Model

In [93]:

```
import time
import timeit
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
import statsmodels.api as sm
```

Preprocessing

In [94]:

```
# load data csv

df = pd.read_csv('df_by_playlist.csv')
print('dataframe has shape', df.shape)
```

```
dataframe has shape (1602, 63)
```

In [95]:

```
names = list(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'popularity', 'speechiness',
       'tempo', 'valence', 'explicit', 'mode', 'num_tracks',
       'track_feat', 'track_remast', 'track_remix', 'album_feat',
       'album_deluxe', 'album_best', 'time_sign_0.0', 'time_sign_3.0',
       'time_sign_4.0', 'key_0.0', 'key_1.0', 'key_2.0', 'key_3.0', 'key_4.0',
       'key_5.0', 'key_6.0', 'key_7.0', 'key_8.0', 'key_9.0', 'key_10.0','key_11
.0'])

# normalize the (un-normalized) continuous data

colnames = ['duration_ms','popularity','tempo','num_tracks']
for name in colnames:
    df[name] = df[name]/max(df[name])
```

In [96]:

```
# add some interaction terms

quant_predictors = list([predictor for predictor in names if df.loc[:,predictor]
.unique().sum()!=1])

def interact(x,variable,predictors=quant_predictors):
    for predictor in predictors:
        x[variable+'_'+predictor] = x[variable]*x[predictor]
    return x

df = interact(df,'loudness')
df = interact(df,'danceability',list(['speechiness','tempo']))
df = interact(df,'loudness')
df = interact(df,'danceability',list(['speechiness','tempo']))

print('dataframe has shape', df.shape)
```

```
dataframe has shape (1602, 77)
```

In [97]:

```
# add polynomial features for continuous data

polydf = pd.DataFrame(df[quant_predictors])

deg = 2

poly = PolynomialFeatures(degree=deg)
polydftrans = poly.fit_transform(polydf)
target_feature_names = ['x'.join(['{}^{}'.format(pair[0],pair[1]) for pair in tu
ple if pair[1]!=0]) for tuple in [zip(polydf.columns,p) for p in poly.powers_]]
polydf = pd.DataFrame(polydftrans, columns = target_feature_names)

df = df.join(polydf)
df = df.drop(colnames,axis=1)
print('dataframe with continuous variable polynomial features has size', df.shap
e)
```

```
dataframe with continuous variable polynomial features has size (160
2, 164)
```

In [98]:

```
# 80/20 train/test split

Xtr, Xte, ytr, yte = train_test_split(df.drop(['followers','playlist_name'],axis
=1), df['followers'], test_size=0.2, random_state=1)

print('training set: predictors dataframe has shape', Xtr.shape)
print('test set: predictors dataframe has shape', Xte.shape)
```

```
training set: predictors dataframe has shape (1281, 162)
test set: predictors dataframe has shape (321, 162)
```

Fit model

In [99]:

```
# do base line fit using only popularity

X_test = Xte['popularity^1']; X_test = sm.add_constant(X_test)
y_test = np.log(yte)
X_train = Xtr['popularity^1']; X_train = sm.add_constant(X_train)
y_train = np.log(ytr)

start = timeit.default_timer()
model = RidgeCV(alphas=(1e-5,1e-4,1e-3,1e-2,1e-1,1,1e1,1e2,1e3,1e4,1e5))
model.fit(X_train,y_train)

predtrain = model.predict(X_train)
predtest = model.predict(X_test)
stop = timeit.default_timer()

R2train = r2_score(y_train,predtrain)
R2test = r2_score(y_test,predtest)
#print('Ridge alpha value is',model.alpha_)
print('R2 on training set is',R2train)
print('R2 on test set is',R2test)
print('took',stop - start,'seconds')
```

```
R2 on training set is 0.203491368866
R2 on test set is 0.126163807149
took 0.0047395068686455 49 seconds
```

In [100]:

```
# do fit using all predictors

X_test = Xte; X_test = sm.add_constant(X_test)
y_test = np.log(yte)
X_train = Xtr; X_train = sm.add_constant(X_train)
y_train = np.log(ytr)

start = timeit.default_timer()
model = RidgeCV(alphas=(1e-5,1e-4,1e-3,1e-2,1e-1,1,1e1,1e2,1e3,1e4,1e5))
model.fit(X_train,y_train)

predtrain = model.predict(X_train)
predtest = model.predict(X_test)
stop = timeit.default_timer()

R2train = r2_score(y_train,predtrain)
R2test = r2_score(y_test,predtest)
print('Ridge alpha value is',model.alpha_)
print('R2 on training set is',R2train)
print('R2 on test set is',R2test)
print('took',stop - start,'seconds')
```

```
Ridge alpha value is 0.1
R2 on training set is 0.56276084744
R2 on test set is 0.484277549712
took 0.0447507060598582 seconds
```

Evaluate Ridge Model

In [101]:

```
# cross_val_score on training set

CVscore = cross_val_score(model,X_train,y_train)
print('CV scores are',CVscore)
print('mean CV score is', np.mean(CVscore))
```

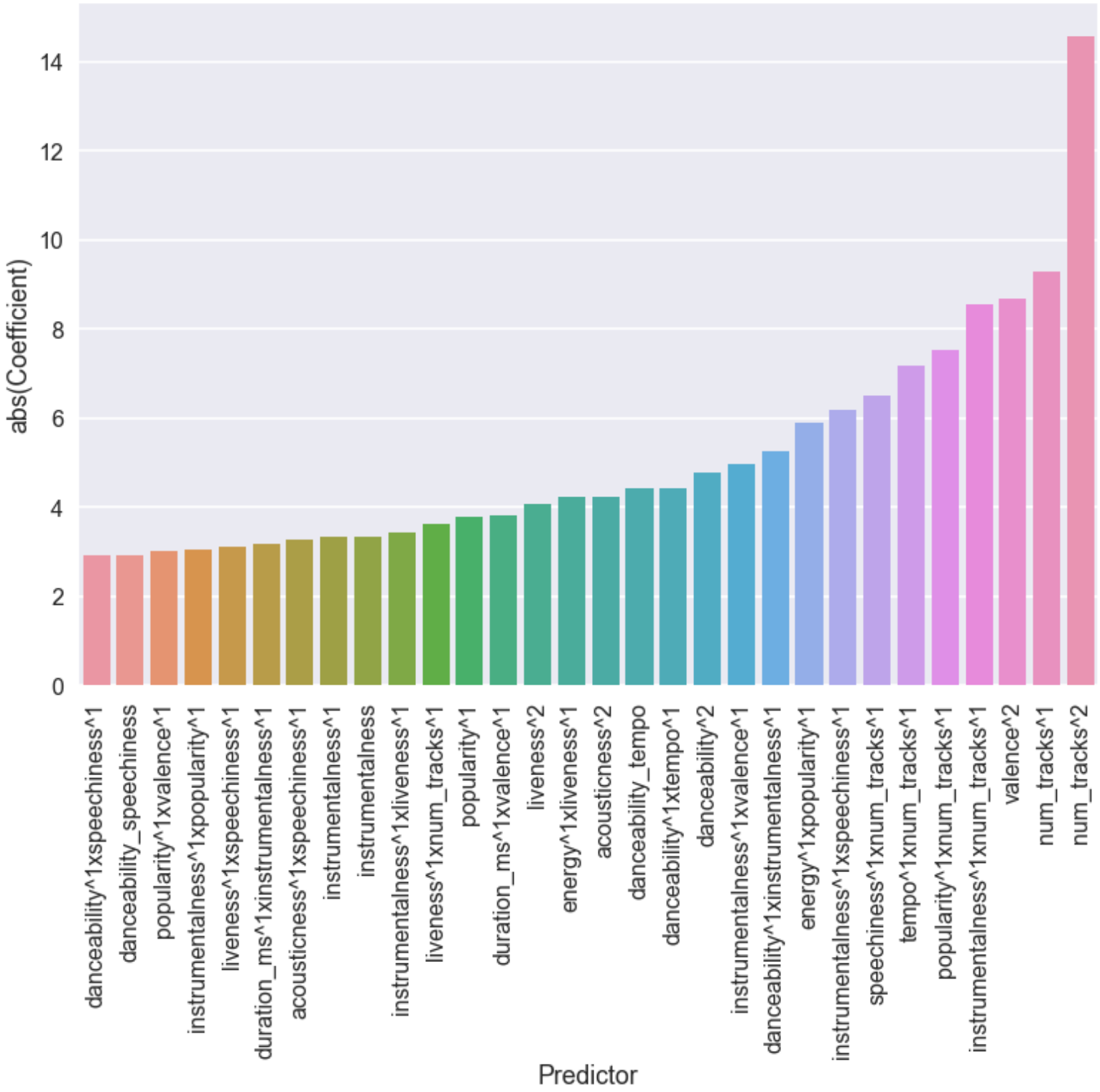
```
CV scores are [ 0.46384187  0.47141176  0.40788327]
mean CV score is 0.447712301003
```

In [102]:

```
# check out importance of predictors

# important predictors
sns.set_context("poster",font_scale=1)

num=30
predictors = list(X_train)
coefficients = abs(model.coef_)
df_ = pd.DataFrame(coefficients,predictors)
df_ = df_.sort_values(0)
fig = sns.barplot(x=df_.index[-num:],y=df_[0][-num:],data=df_)

for item in fig.get_xticklabels():
    item.set_rotation(90)

fig.set(xlabel='Predictor', ylabel='abs(Coefficient)')
```

Out[102]:

[<matplotlib.text.Text at 0x14d47a400>, <matplotlib.text.Text at 0x14c4bca20>]

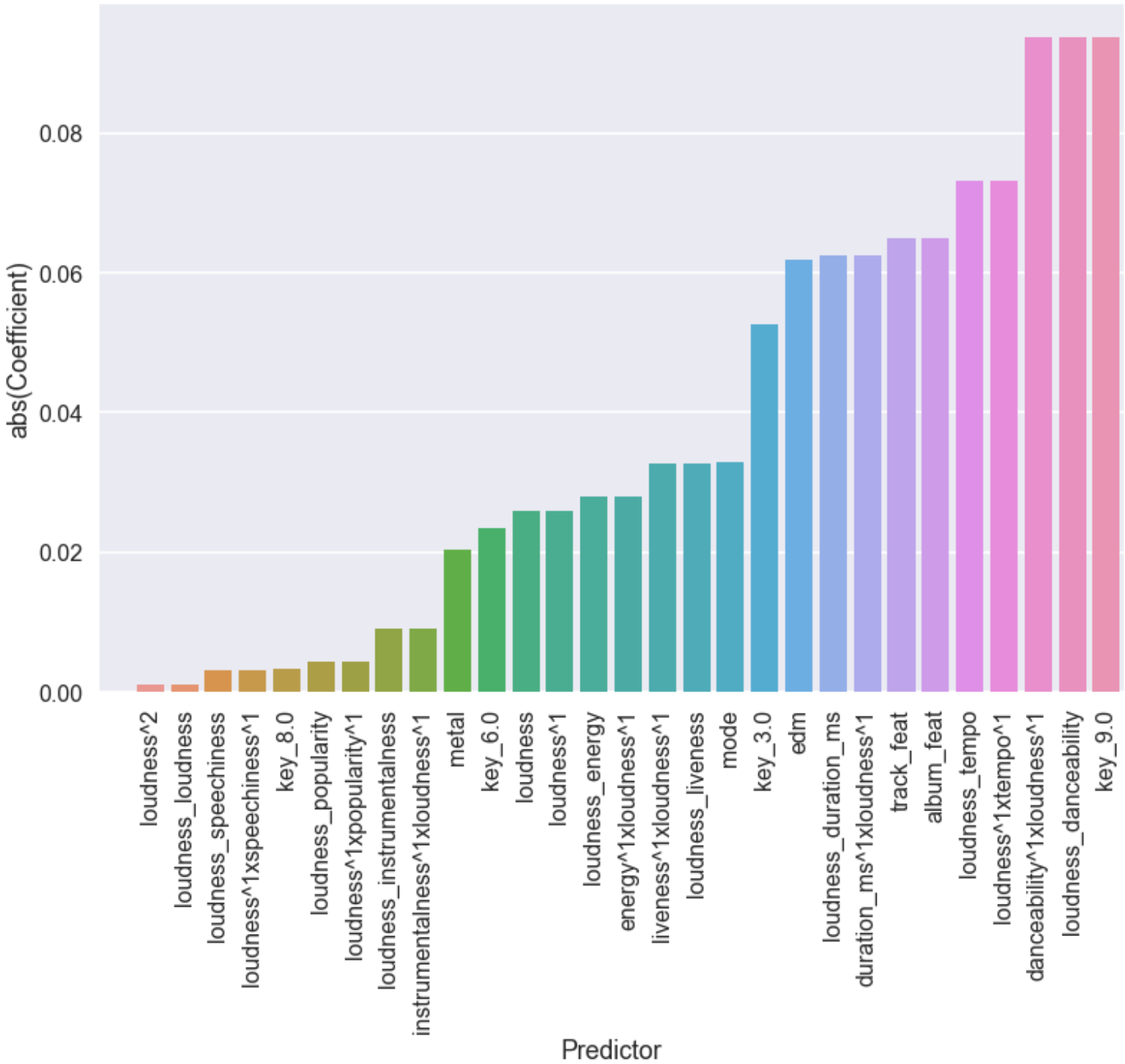


In [103]:

```
# unimportant predictors

# important predictors
sns.set_context("poster",font_scale=1)

num=30
predictors = list(X_train)
coefficients = abs(model.coef_)
df_ = pd.DataFrame(coefficients,predictors)
df_ = df_.sort_values(0)
fig = sns.barplot(x=df_.index[0:num],y=df_[0][0:num],data=df_)

for item in fig.get_xticklabels():
    item.set_rotation(90)

fig.set(xlabel='Predictor', ylabel='abs(Coefficient)')
```

Out[103]:

[<matplotlib.text.Text at 0x15e32ffd0>, <matplotlib.text.Text at 0x1
5bbf9a58>]



Winning Model Evaluation

In [104]:

```
model = GradientBoostingRegressor(n_estimators=1000,
                                  loss="huber", learning_rate=bp2['learning_rate']
,
                                  max_depth=bp['max_depth'],
                                  max_features=bp['max_features'],
                                  min_samples_leaf=bp['min_samples_leaf'],random_s
tate=1)
model.fit(xtrain,ytrain)
```

Out[104]:

```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=
None,
             learning_rate=0.01, loss='huber', max_depth=12,
             max_features=0.3, max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=7, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=1000,
             presort='auto', random_state=1, subsample=1.0, verbose=
0,
             warm_start=False)
```

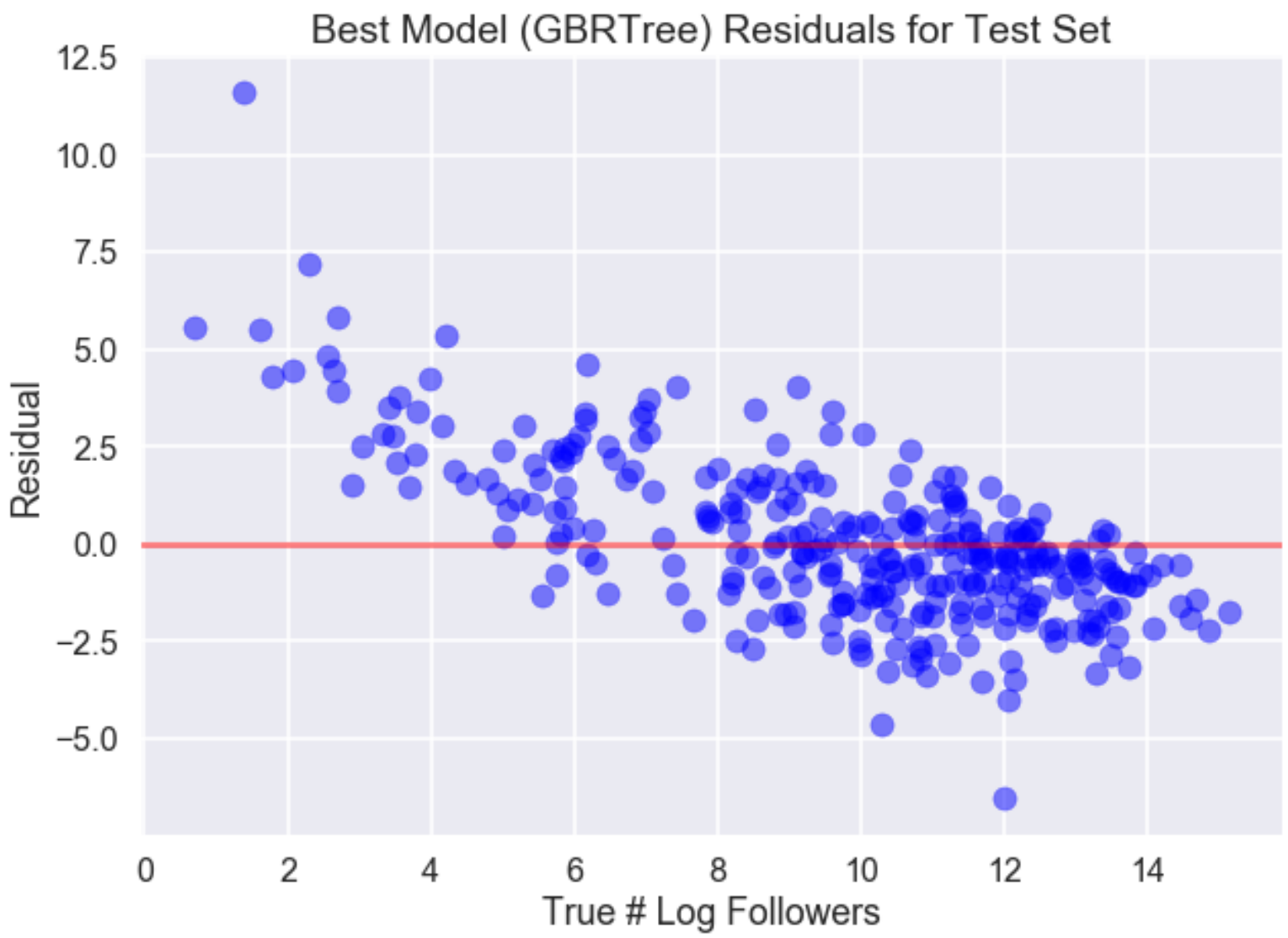
In [105]:

```
print('Score (test set): ',model.score(xtest, ytest))
print('MSE (test set): ',mean_squared_error(ytest, model.predict(xtest)))
print('MAE (test set): ',mean_absolute_error(ytest, model.predict(xtest)))
```

```
Score (test set):  0.540180575032
MSE (test set):  4.25311557203
MAE (test set):  1.54966736183
```

In [109]:

```
# Residual Plot
plt.figure(figsize=(10,7))
plt.scatter(ytest, model.predict(xtest).reshape(-1,1)-ytest,c='b',alpha=0.5)
plt.axhline(c='red',alpha=0.5)
plt.xlabel('True # Log Followers')
plt.ylabel('Residual')
plt.title('Best Model (GBRTree) Residuals for Test Set')
plt.legend();
```



Many of the playlists with larger residuals also have a low number of followers

In [107]:

```
data[data.followers<5]
```

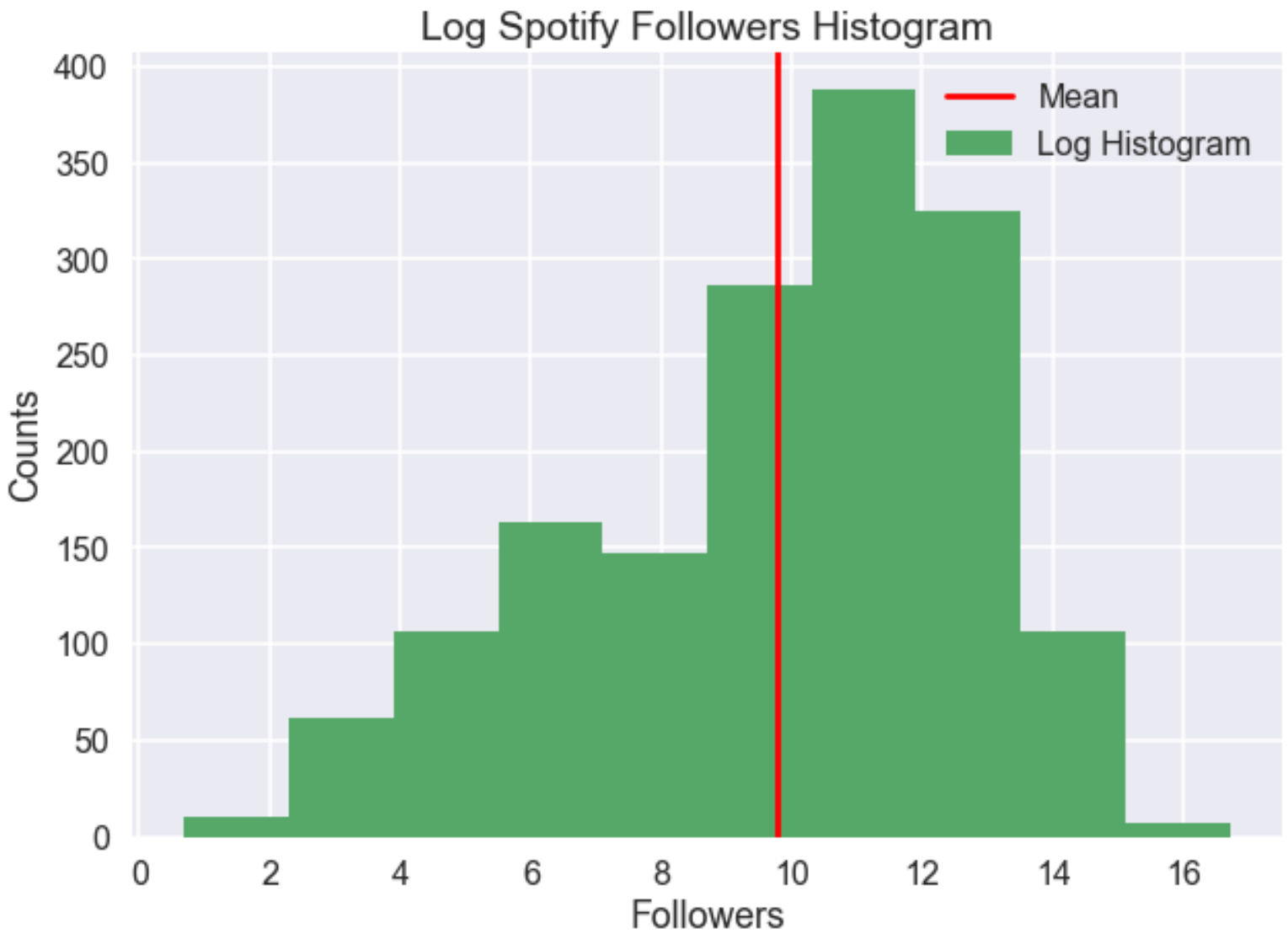
Out[107]:

	acousticness	danceability	duration_ms	energy	instrumentalness
playlist_name					
Honeysuckle Lazy	0.360725	0.562000	227349.550000	0.581900	0.002448
Intervju med Seinabo Sey	0.557667	0.580333	277483.000000	0.362667	0.000000
On to Something Good	0.200703	0.595850	210925.350000	0.805950	0.002166
Rise of the Underdog	0.014342	0.487300	230109.400000	0.845300	0.012336
dw-c	0.206085	0.669720	217227.500000	0.618930	0.003239
dw_g	0.381130	0.597000	229589.066667	0.523767	0.009258

6 rows × 62 columns

In [110]:

```
plt.figure(figsize=(10,7))
plt.hist(np.log(df.followers.values+1),label='Log Histogram',color=sns.color_pal
ette()[1])
plt.axvline(np.mean(np.log(df.followers.values+1)), 0, 1.0, color='red', label='
Mean')
plt.xlabel("Followers")
plt.ylabel("Counts")
plt.title("Log Spotify Followers Histogram")
plt.legend();
```



In [ ]: