MACHINE LEARNING
FINAL ASSIGNMENT
COURSE 2

The jupyter notebook with the code is at the end of the report

I. INTRODUCTION

i. BACKGROUND

The data I want to analyze for this last assignment is about music. Since Spotify is one of the biggest music platform out there I will utilize a dataframe containing data from it. The dataframe I will describe in the next section is comprehensive of more then 100 thousands songs. The final objective of this little project is to be able to see which are the most important features that influence the song popularity which will be our target variable. I will try to build a model with Lasso Regression given its higher interpretation power, but, out of curiosity I will try also other models such as Ridge Regression and Elastic Net.

II. THE DATA

i. DESCRIPTION OF THE DATASET

The dataset I will use is comprehensive of 170653 songs collected from a random selection of users from the Spotify free database. The dataset is in a csv format and contains 19 features:

1. ***Valence* –** Measuring the degree of positiveness of a song (Range: 0 to 1)
2. ***Year* –** The release year of the track (Range: 1921 to 2020)
3. ***Acousticness* –** Measuring how acoustic a track is. (Range: 0 to 1)
4. ***Artists* –** The list of artist credited for the song
5. ***Danceability* –** Measure how much the track is danceble (Range: 0 to 1)
6. ***Duration*_ms –** The length of the track in milliseconds
7. ***Energy* –** Measure how energetic a track is (Range: 0 to 1)
8. ***Explicit* –** Whether a track contains explicit content or notation (0=No, 1=Yes)
9. ***Id* –** Identification of the track generated by Spotify
10. ***Instrumentalness* –** The relative ratio of the track being instrumental (Range: 0 to 1)
11. ***Key* –** The primary musical key of the track encoded as integers (All keys are encoded as values ranging from 0 to 11, starting with C as 0, C# as 1 etc.)
12. ***Liveness* –** The relative duration of track sounding as a live performance (Range: 0 to 1)
13. ***Loudness* –** Relative loudness of the track (Range: -60 to 0)
14. ***Mode* –** Whether a track start with a major cord progression or notation (0=m, 1=M)
15. ***Name* –** Title of the track
16. ***Popularity* –** Present popularity of the song in the US (Range: 0 to 100)
17. ***Release Date* –** The release date of the track (yyyy-mm-dd)
18. ***Speechiness* –** The relative length in a track containing human voice (Range: 0 to 1)
19. ***Tempo* –** The tempo of the song

ii. DATA CLEANING

The data appear already well presented. There are some features with categorical values but most of them are numerical. First lets focus on those features that, reasonably, would not be useful to predict the popularity of a song. I would say that the song '*name*' can be removed as well as the '*id*' and also the '*release_date*' since it is the same as the year column and the elements are of type object. Therefore lets remove the columns aforementioned. All the other might be all necessary. For skewed features I have intention to try to use both the log1p but also the square-root transformations, which doesn't take negative numbers. Therefore, I decided to make the elements in the '*loudness*' feature all positive by taking their absolute value. I also created a new, smaller dataset, '*data_small*' for the purpose of calling the .describe() function. The data_small will be without features such as '*mode*' or '*year*' for which it would be useless to call .describe() on.

iii. FEATURE ENGINEERING

The feature containing the artist name can be important to predict the popularity of a song. However this feature, '*artists*', is of object type which means we need to encode it in order to add it to our model. I will do an ordinal encoding of this feature which means every unique artists name will have its own unique number. Afterwards I will drop the original artists column and keep only the '*artist_enc*'. Now all features are of numerical type and there are no missing values.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170653 entries, 0 to 170652
Data columns (total 16 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   valence           170653 non-null  float64
 1   year              170653 non-null  int64
 2   acousticness      170653 non-null  float64
 3   danceability      170653 non-null  float64
 4   duration_ms       170653 non-null  int64
 5   energy            170653 non-null  float64
 6   explicit          170653 non-null  int64
 7   instrumentalness  170653 non-null  float64
 8   key               170653 non-null  int64
 9   liveness          170653 non-null  float64
 10  loudness          170653 non-null  float64
 11  mode              170653 non-null  int64
 12  popularity        170653 non-null  int64
 13  speechiness       170653 non-null  float64
 14  tempo             170653 non-null  float64
 15  artists_enc       170653 non-null  int64
dtypes: float64(9), int64(7)
memory usage: 20.8 MB
```

Figure 1: Feature list with variable type

Next I will search the float64 columns for skewing setting the skew limit at 0.75. There are four features that present a significant skewing. All those features have a right skew making a log1p transformation optimal.

	Skew
speechiness	4.047848
liveness	2.154382
instrumentalness	1.631114
loudness	1.052758

Figure 2: Skewed columns

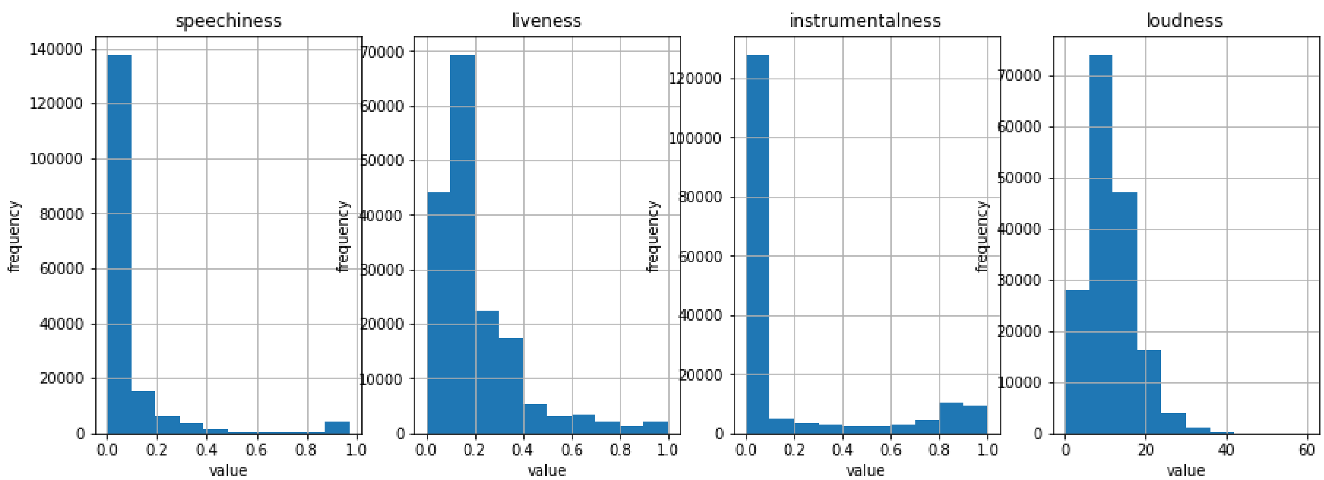


Figure 3: Histogram of the skewed columns

We can see that the speechiness and instrumentalness are two features that presents a huge peak on the lower values. Showing that most of the songs are not instrumental and most of them have words. These two columns will be hard to normalize. Lets see what happens after the normalization with log1p and with the square root method.

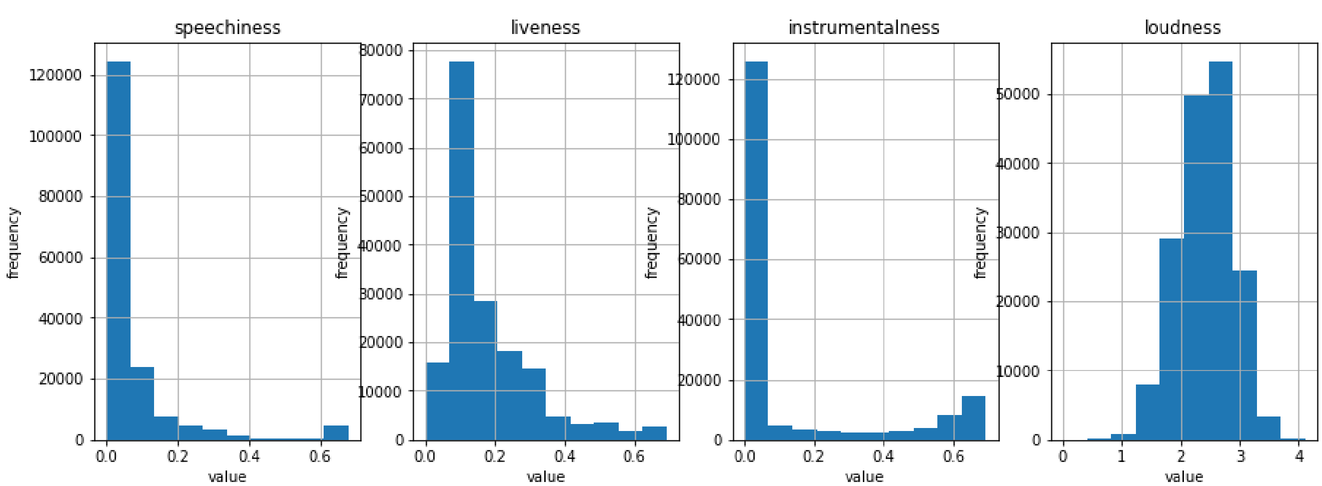


Figure 4: Histogram after log1p transformation

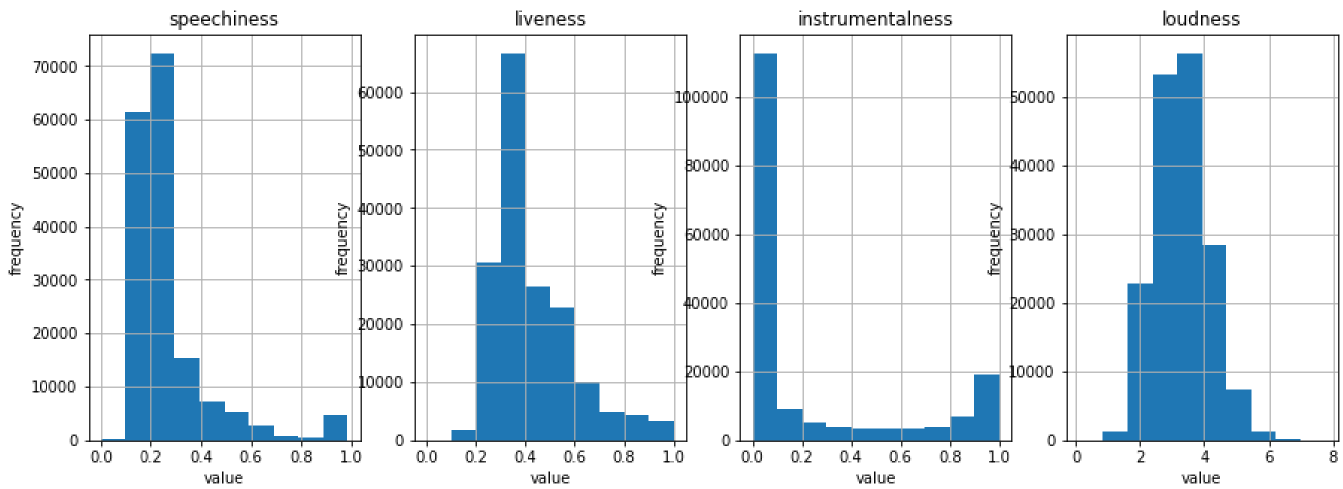


Figure 4: Histogram after square-root transformation

I tried to use the D’Agostino K^2 method to evaluate the p-value to see how normalized where the distribution of these two columns after the transformations. Unfortunately it was not helpful since the D’Agostino method was indifferent to both transformations giving always a p-value of 0.0 for all features. However to have at least a hint if these two transformations were useful I calculated the skew values after both transformation in order to see if they had any effect and which one resulted in a better normalization.

	Skew
speechiness	4.047848
liveness	2.154382
instrumentalness	1.631114
loudness	1.052758

Original skewed data

```
speechiness           3.587121
liveness              1.748512
instrumentalness      1.540390
loudness             -0.229451
dtype: float64
```

After Log1p transformation

```
speechiness           2.857476
liveness              1.302035
instrumentalness      1.309775
loudness              0.326961
dtype: float64
```

After squre root transformation

Figure 5: Comparison between the original skewed data and after the transformations

Overall the square root transformation seems to have done a better job then the log1p transformation. However the data is still significantly skewed right. Thinking it through, I decided to drop the instrumentalness feature because more then 85% of the songs have values between 0 and 0.1 which means its almost a constant that is not going to be very effective in our model.

iv. DATA EXPLORATION

Lets see a little bit the relations between our features themselves and between some features and the target variable. Also lets separate the plotted data in explicit and non-explicit songs. I want to do this separation for two main reasons. First, it will be interesting to see the difference from explicit and non-explicit songs and second, given the large amount of data, splitting it I think will result in more clear and meaningful plots. First lets explore some features vs other features. I am interested in seeing the relation between Energy vs Valence and Danceability vs Valence.

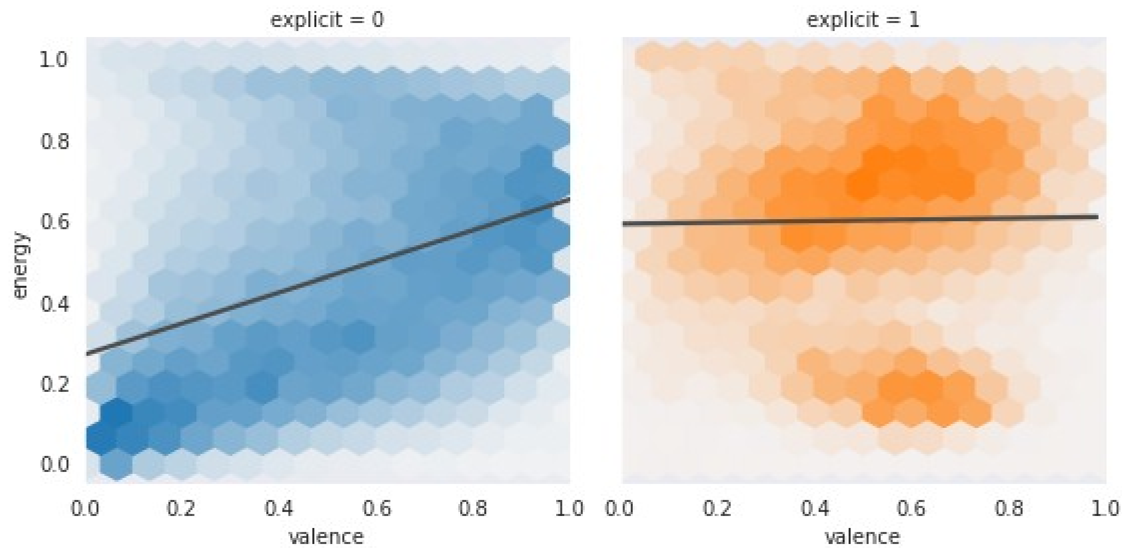


Figure 6: Energy vs Valence

Non-explcit song have a high concentration at the two extreme of the valence range which slight more frequency to the sad side (valence 0). Also it is interesting to see that for non-explicit very sad songs the energy is very low while for happy songs the energy is higher. On the other hand for explicit songs the valence is more focused in the center in middle values. We can identify two different groups, one more expanded characterized by a pretty high energy level average and another one less expended characterized by a low energy level

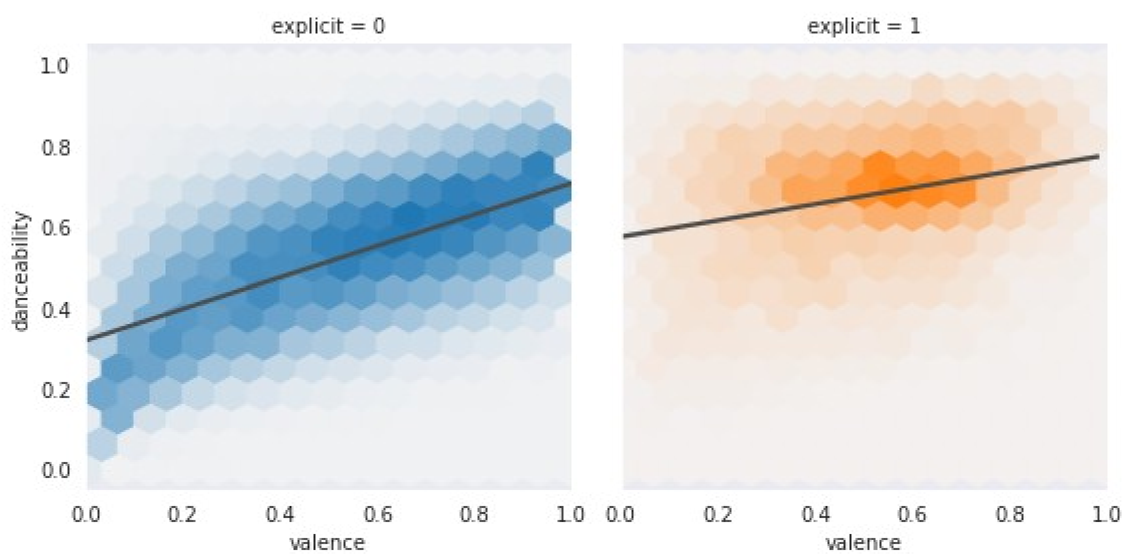


Figure 7: Danceability vs Valence

We can see that as for non-explicit songs the happier they get the more danceable they become while the explicit song presents are concentrated in the mid valence values and have a pritty high average danceability and increasing from sad to happy songs.

Now I will start to observe the relations between the features and the target variable (popularity). Lets start with a basic popularity of 5 to avoid having to consider unknown songs that will only make our plots more confusing.

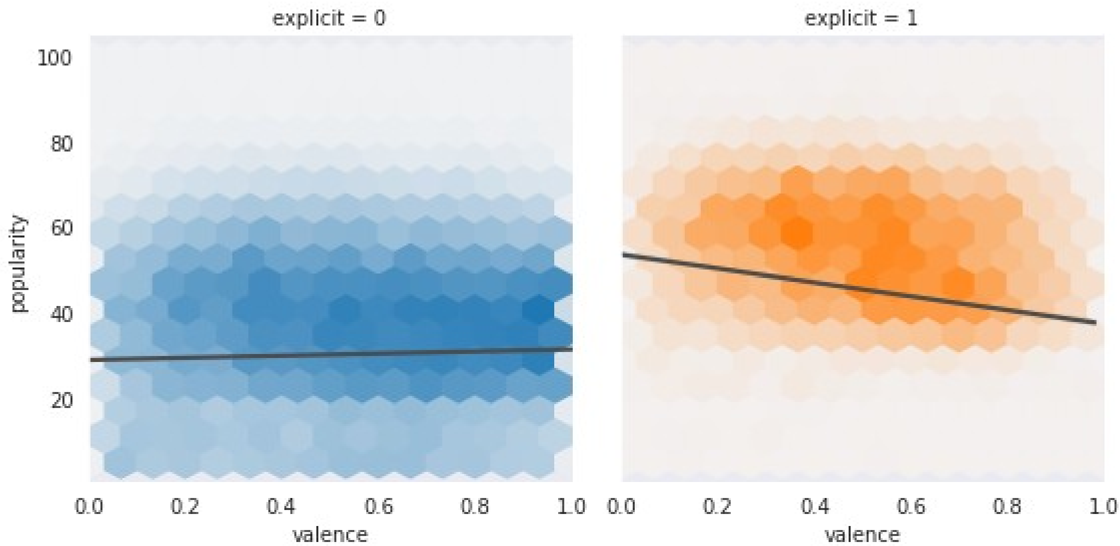


Figure 8: Popularity vs Valence

Lets keep in mind that there are 156220 non explicit songs and 14433 explicit ones. However, it looks like the non explicit songs are most likely to be under 60 of popularity and stacked towards the positive/happy feeling while the explicit tend to be more popular with popularity between 40 and 80 with almost no presence under the 40 popularity. Moreover, explicit songs tend to be concentrated in the middle of the valence(positiveness) range and their popularity decreases the more happy the song is. Which means that an explicit song is more popular is it is sad than happy. Kind of makes sense.

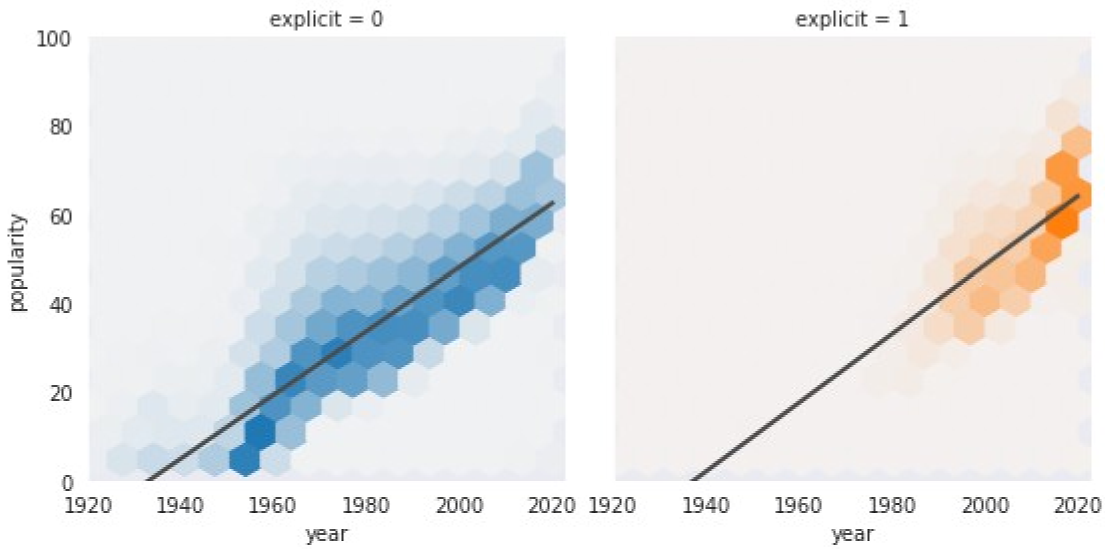


Figure 9: Popularity vs Year

This is interesting we can see that before 1990 songs were practically all non-explicit indicating a change of the morality and what was considered unaccepable before 1990. We can also see that while non-explicit songs tend to be pretty stable in time, maybe with a little decline towards 2020, the explicit traks seems to grow in popularity as time progresses.

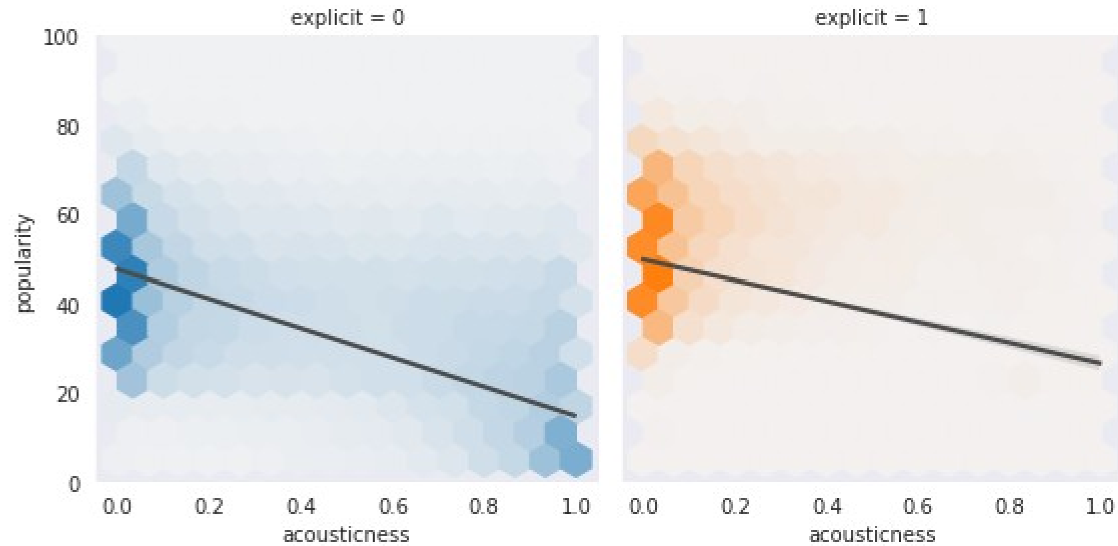


Figure 10: Popularity vs Acousticness

We can see that almost all songs are in the non-acustic sides for both explicit and non-explicit. Of course, the only precence of tracks with a good acousticness character can be seen only in the non-explicit plot, as it should be. However the popularity of the songs decreases as the more acustic the track is.

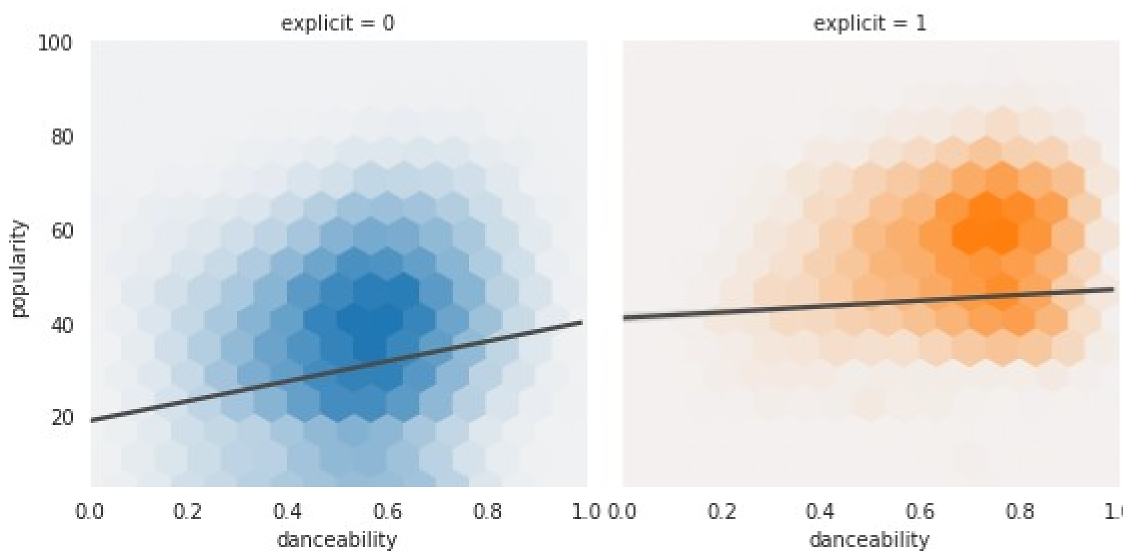


Figure 10: Popularity vs Danceability

For non-explicit songs popularity tend to increase as the danceability increases while the majority of the tanks are in the middle ranges of danceability. The popularity of explicit songs, on the other hand, dont vary much with the danceability and the majority of the tracks are, on average, more danceable then the non-explicit ones.

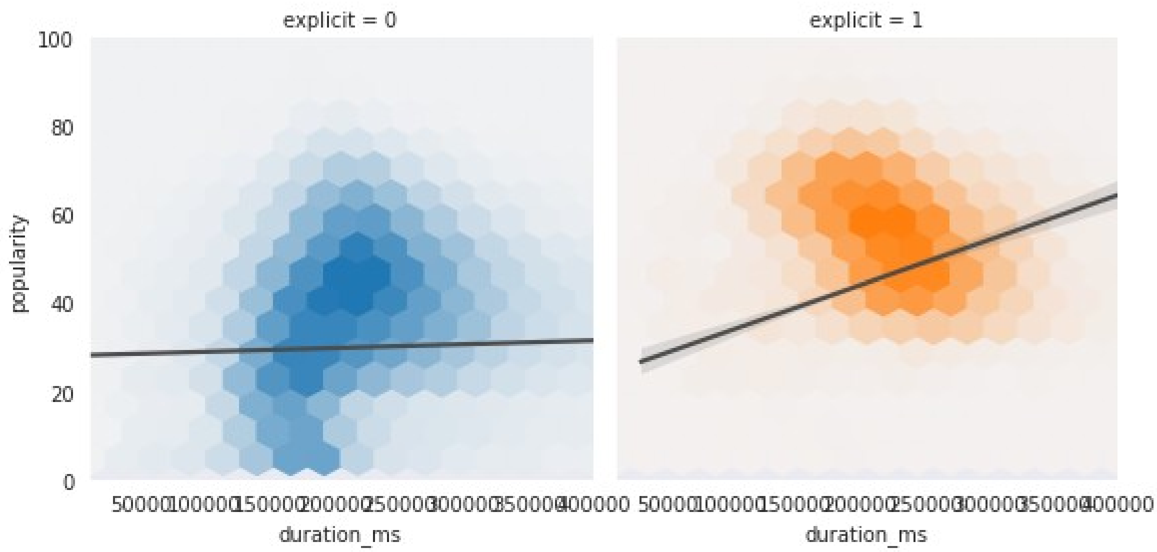


Figure 11: Popularity vs Duration_ms

On average, explicit and non-explicit songs tend to have comparable time-lenght. We can see that explicit songs seems to be a little more popular the non-explicit ones with similar track duration.

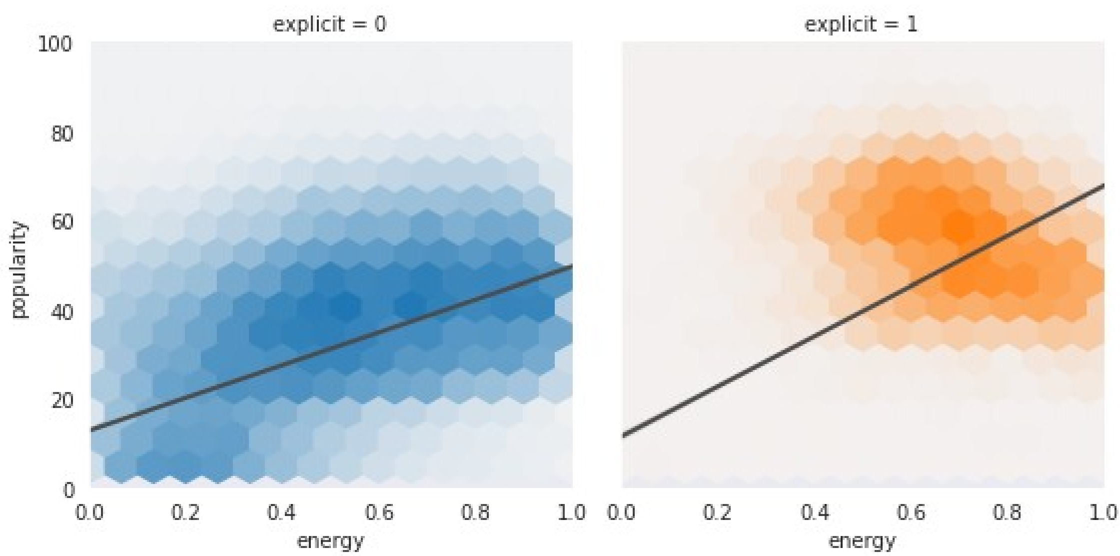


Figure 12: Popularity vs Energy

For both cases the more a song is energetic the more popular it is.It is interesting to see that Non-explicit songs are diffused across all energy levels while explcit ones are focused on the high energy end.

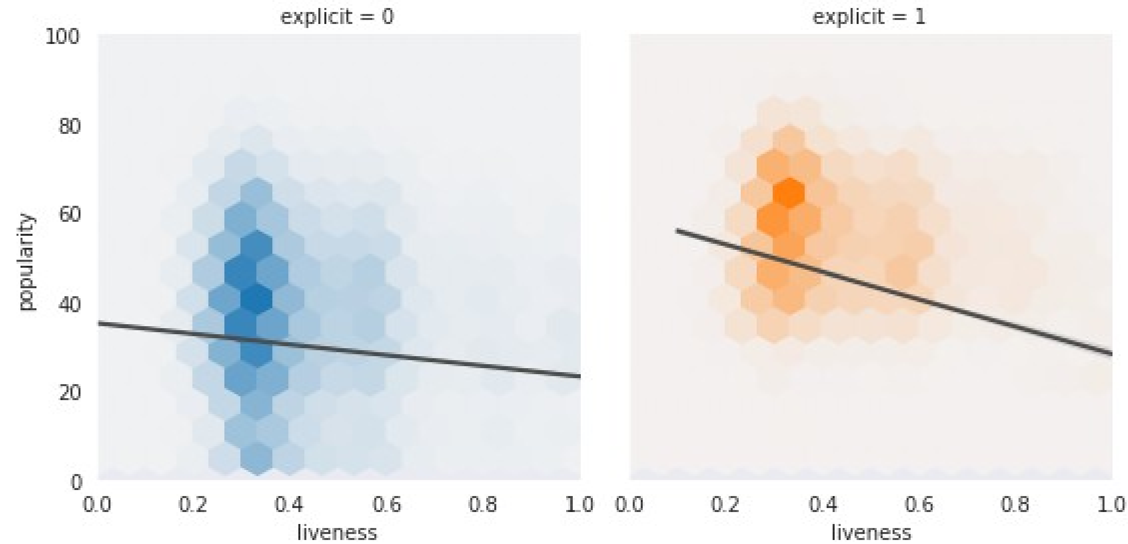


Figure 13: Popularity vs Liveness

We see that both non-explicit and explicit songs have similar level of liveness feeling and have very similar behaviors.

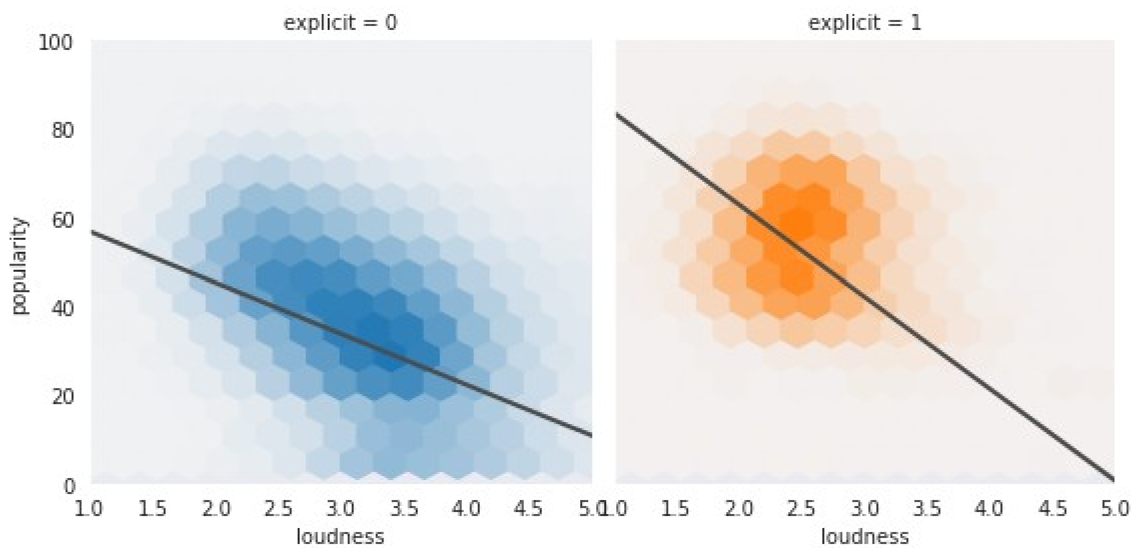


Figure 14: Popularity vs Loudness

In the features engineering section I took the absolute value for the loudness because it was ranging from -60 and 0 and it would not have been possible to do a square root transformation of the feature. Therefore the behavior we see here should be mirrored on the y axis. The popularity increases with an increment in the loudness feature. Also we see that explicit songs tend to have a higher loudness then non-explicit songs.

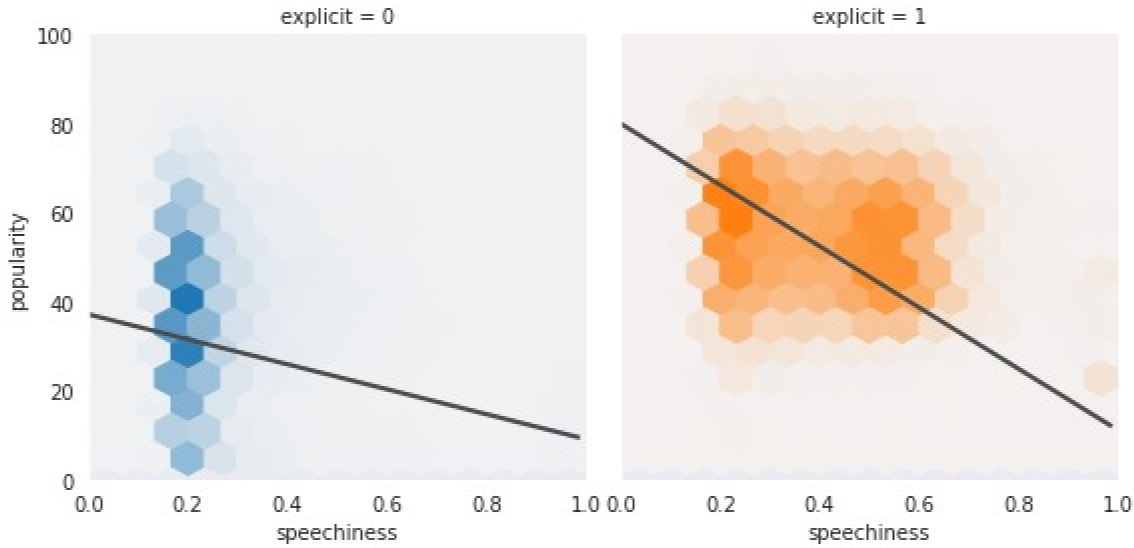


Figure 15: Popularity vs Speechiness

This is interesting because we see that non-explicit songs have a speechiness concentrated around the 0.2 mark which is pretty low. However the explicit songs are more speechy but still the popularity of the track diminishes as speechiness increases.

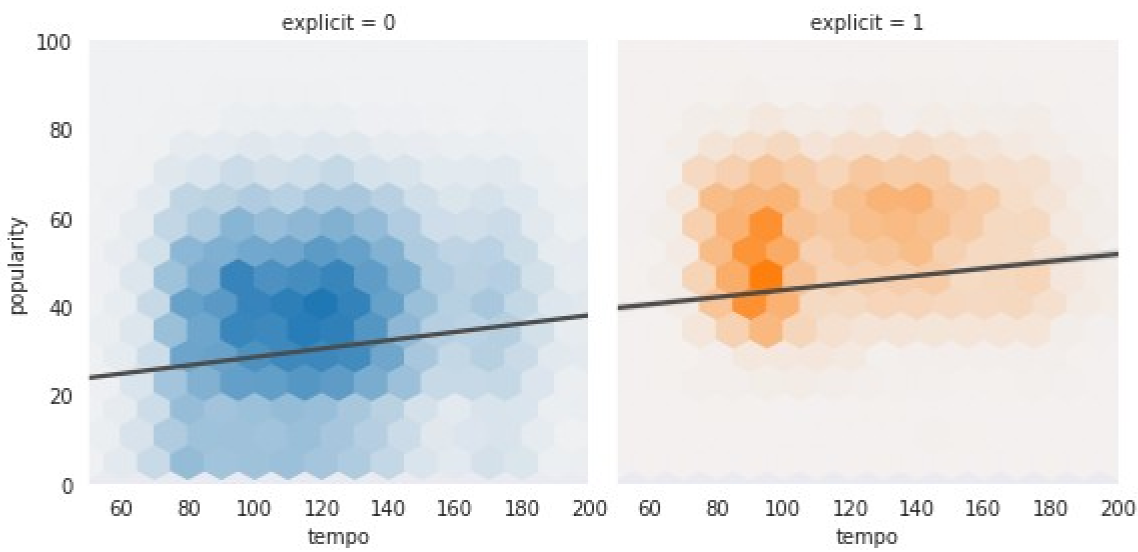


Figure 15: Popularity vs Tempo

Most of the non-explicit songs have a tempo range between 80 and 140 bpm while the explicit tracks are more frequent in the 80 to 100 bpm. For both explicit and non-explicit the song popularity tend to increase with increased tempo.

III. MACHINE LEARNING METHODS

First off, I will start with simple linear regression to warm up and see what if this method at least decent in predicting my target variable '*song popularity*'. First off I will create a dataset called '*X*' where I will have all the features minus the target variable and a dataset calle '*y*' where I will have only the target variable. Subsequently I created the training and testing splits. I used a test size of 30% of the dataset and a random-state to ensure the same test and train splits, of 43210.

1 Simple Linear Regression

R^2	Mean Square Error
0.7533323362083683	117.63853180989686

Vanilla Linear Regression has a $R^2 > 75\%$ which is not that bad. Anyways lets see if we and do better with Ridge and Lasso Regression.

■ Ridge Regression

R^2	Mean Square Error	Polynomial Degree	Alpha
0.776354576393793	106.5443078393529	3	132.728558

In the ridge regression the features have been scaled to a common scale using the StandardScaler, Moreover I applied the polynomial features to take into account the squared features and all the interaction terms between them. To find the best hyper-parameters I decided to use the GridSearchCV function. To make this work I first created a Kfold cross validation object for three main reasons. Firstly, for specify that I wanted the dataset split in 5 train/test sets, for ensure that the train tests split where shuffled and to specify the random-state to be the same I used for the vanilla linear regression (43210). Subsequently I used a pipeline to feed my hyper-parameters to the model. The polynomial features have been set from 1 to 3. Unfortunately I could not set higher polynomial degrees because my laptop does not have enough memory to handle such models. However we can see that this model has a better predictive power since the R^2 score is higher compared to the simple linear regression.

■ Lasso Regression

R^2	Mean Square Error	Polynomial Degree	Alpha
0.7640169734051945	112.4219213832434	2	0.080000

In this case I used GridSearchCV too to find the best hyper-parameters. The lasso regression here presents a predictive accuracy R^2 lower then the ridge regression as expected since this model is more focused on interpretation. However this model is still better then the vanilla linear regression since it has a higher R^2. Similarly as the ridge regression I had to limit the choice of polynomial degrees to 1 and 2. Degree 3 was prohibitive for my laptop. Also the alpha set chosen was very small, from 0.08 to 0.01. The optimal alpha is probably even lower but the model would get too complex to be computed by my machine.

- Elastic-Net Regression

R^2	Mean Square Error	Polynomial Degree	Alpha	L1_ratio
0.7640025753707842	112.4287805828896	2	0.01	0.8

The elastic net regression is a model that is in between a ridge and lasso regression. For running this model I used a gridsearchCV as I did for all the previous models. Using a pipeline I fed the model different hyper-parameters. The polynomial features degrees considered were 1 and 2, the alpha values 0.01 to 0.1 and the L1_ration of 0.5, 0.6, 0.7, 0.8. The best score is R^2 is around 0.76 which is pretty similar to the score obtained with the ridge regression. The best polynomial degree hyper-parameters here is degree 2, however, similarly to the previous regressions, I could not use a higher degree due to computational limits. Same issue with the alpha which should be smaller to have a better model but the increased complexity that would arise from a lower alpha would have not been able to compute with my machine.

IV. RECOMMENDED METHOD

If the final objective is the interpretation of the features I recommend the lasso regression method. This method gives a better R^2 score then the simple linear regression but a lower one compared to the ridge regression. However the lasso regression is characterized by a better interpetability of the data while penalizing the accuracy of the prediction.

V. SUMMARY AND FINDINGS

The dataframe under analysis is a collection of more then 100 thousand songs taken from Spotify. Every song is listed with 19 different features. The feature taken as target variable is the popularity of the songs. The project objective was to learn which where the most important features that would be useful in predict the popularity of a song.

2	acousticness	16.189867
13	artists_enc	-0.033159
3	danceability	-2.394422
4	duration_ms	0.771700
5	energy	-0.272737
6	explicit	-0.412394
7	key	-0.000000
8	liveness	0.023451
9	loudness	-0.360195
10	mode	-0.958898
11	speechiness	-0.117398
12	tempo	-1.574637
0	valence	0.000000
1	year	-0.074702

Figure 16 shows the coefficient that the *gridsearch* optimized lasso regression has found for each feature used to predict the traget variable '*popularity*'. Surprisingly we see that the acousticness feature has a incredibly high impact on the prediction while all the other feature have pretty much the same importance. We can see that some features have been set to zero such as the key and the valence features.

The last method implemented was the elastic net. Actually I made a pipeline that would feed to the method 4 different L1_ratios. However it surprised me to see that the best R^2 score was obtained with the highest L1_ratio. I would have though that the best prediction would come from the model with the less lasso-regularization in favor of the L2-regularization which is more focused on prediction then interpretability.

VI. FUTURE SUGGESTION

For future study I would like to find a way to better normalize the features that presented a fairly high right skew and that I was not able to normalize well. I suggest analyzing both ridge and lasso regression more in detail. I would try to increase the polynomial degree beyond 3 if possible and probe many different lower alphas for the lasso regression model. Finally ,it would be interesting to apply the machine learning method trained here to predict song popularity on another dataset.

Final_course2

January 22, 2021

1 Machine Learning Final Exercise - 2nd Course

1.1 Working with a Spotify dataset

I think this is going to be interesting. I will try to predict song popularity base on different features. I will focus on interpretation. But maybe I will also try a more predictive model

Importring libraries

[261]:
```
import pandas as pd
import numpy as np
from scipy.stats import binom, normaltest, shapiro
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV, KFold,␣
 ↪cross_val_predict
from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler, PolynomialFeatures,␣
 ↪LabelEncoder, MinMaxScaler
from sklearn.pipeline import Pipeline

%pylab inline
%matplotlib inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

Importing the data.

[4]:
```
filepath = 'spotify_data.csv'
data_main = pd.read_csv(filepath)
data_main.head()
```

[4]:
```
   valence  year  acousticness  \
0   0.0594  1921         0.982
1   0.9630  1921         0.732
2   0.0394  1921         0.961
3   0.1650  1921         0.967
4   0.2530  1921         0.957
```

```
                                             artists  danceability  \
0  ['Sergei Rachmaninoff', 'James Levine', 'Berli…         0.279
1                                     ['Dennis Day']         0.819
2  ['KHP Kridhamardawa Karaton Ngayogyakarta Hadi…         0.328
3                                   ['Frank Parker']         0.275
4                                     ['Phil Regan']         0.418

   duration_ms  energy  explicit                      id  instrumentalness  \
0       831667   0.211         0  4BJqT0PrAfrxzMOxytFOIz          0.878000
1       180533   0.341         0  7xPhfUan2yNtyFG0cUWkt8          0.000000
2       500062   0.166         0  1o6I8BglA6ylDMrIELygv1          0.913000
3       210000   0.309         0  3ftBPsC5vPBKxYSee08FDH          0.000028
4       166693   0.193         0  4d6HGyGT8e121BsdKmw9v6          0.000002

   key  liveness  loudness  mode  \
0   10     0.665   -20.096     1
1    7     0.160   -12.441     1
2    3     0.101   -14.850     1
3    5     0.381    -9.316     1
4    3     0.229   -10.096     1

                                                name  popularity release_date  \
0  Piano Concerto No. 3 in D Minor, Op. 30: III. …           4         1921
1                            Clancy Lowered the Boom           5         1921
2                                          Gati Bali           5         1921
3                                          Danny Boy           3         1921
4                        When Irish Eyes Are Smiling           2         1921

   speechiness    tempo
0       0.0366   80.954
1       0.4150   60.936
2       0.0339  110.339
3       0.0354  100.109
4       0.0380  101.665
```

[5]:
```
data_main.shape
```

[5]:
```
(170653, 19)
```

Get to know the dataframe a little bit

Lets see how many songs there are that have the same artist

[6]:
```
data_main.artists.value_counts()
```

[6]:
```
['         ']                                        1211
['          ']                                       1068
```

```
['Francisco Canaro']                                    942
['Frank Sinatra']                                       630
['Ignacio Corsini']                                     628
                                                       …
['Sara Ramirez']                                          1
['Judy Garland', 'Leo Diamond Harmonica Quartet']         1
['Richard Wagner', 'Georg Kulenkampff', 'Franz Rupp']     1
['Pushpavalli']                                           1
['Gucci Mane', 'Jeezy', 'Boo']                            1
Name: artists, Length: 34088, dtype: int64
```

I'm actually curious to see if there are many songs with the same title

[7]:
```
data_main.name.value_counts()
```

[7]:
```
White Christmas                                            73
Winter Wonderland                                          63
Summertime                                                 56
Jingle Bells                                               53
Overture                                                   46
                                                           ..
Overture to Act I of Lohengrin                              1
Screamager                                                  1
Whatever Will Be, Will Be (Que Sera, Sera) - Single Version 1
Something Wonderful                                         1
Las Margaritas                                              1
Name: name, Length: 133638, dtype: int64
```

ah ! thats funny :)

Ok. Lets move on. Lets see the dataframe structure

[8]:
```
print('Column Names')
print(data_main.columns.tolist())

print('\n Number of rows')
print(data_main.shape[0])

print('\n Number of columns')
print(data_main.shape[1])

print('\n Data type per column')
print(data_main.dtypes)
```

```
Column Names
['valence', 'year', 'acousticness', 'artists', 'danceability', 'duration_ms',
'energy', 'explicit', 'id', 'instrumentalness', 'key', 'liveness', 'loudness',
'mode', 'name', 'popularity', 'release_date', 'speechiness', 'tempo']
```

```
 Number of rows
170653

 Number of columns
19

 Data type per column
valence             float64
year                  int64
acousticness        float64
artists              object
danceability        float64
duration_ms           int64
energy              float64
explicit              int64
id                   object
instrumentalness    float64
key                   int64
liveness            float64
loudness            float64
mode                  int64
name                 object
popularity            int64
release_date         object
speechiness         float64
tempo               float64
dtype: object
```

Making a copy as for backup

[9]:
```
data_backup = data_main.copy()
```

And one where we will work on.

[10]:
```
data=data_main.copy()
```

1.2 Data Cleaning

First lets focus on those features that, reasonably, would not be useful to predict the popularity of a song. I would say that the song name can be removed as well as the id and the release_date since it is the same as the year column and it the elements are object type. All the other might be all necessary. Still dont know, we should build an interpretative model to see that.

[11]:
```
data_main.shape
```

[11]:
```
(170653, 19)
```

[12]:
```
data=data.drop(['name', 'id', 'release_date'], axis=1)
```

[13]:
```
data.shape
```

[13]: (170653, 16)

[14]:
```
data.head()
```

[14]:
```
   valence  year  acousticness  \
0   0.0594  1921         0.982
1   0.9630  1921         0.732
2   0.0394  1921         0.961
3   0.1650  1921         0.967
4   0.2530  1921         0.957

                                             artists  danceability  \
0  ['Sergei Rachmaninoff', 'James Levine', 'Berli…        0.279
1                                     ['Dennis Day']         0.819
2  ['KHP Kridhamardawa Karaton Ngayogyakarta Hadi…        0.328
3                                   ['Frank Parker']         0.275
4                                     ['Phil Regan']         0.418

   duration_ms  energy  explicit  instrumentalness  key  liveness  loudness  \
0       831667   0.211         0          0.878000   10     0.665   -20.096
1       180533   0.341         0          0.000000    7     0.160   -12.441
2       500062   0.166         0          0.913000    3     0.101   -14.850
3       210000   0.309         0          0.000028    5     0.381    -9.316
4       166693   0.193         0          0.000002    3     0.229   -10.096

   mode  popularity  speechiness    tempo
0     1           4       0.0366   80.954
1     1           5       0.4150   60.936
2     1           5       0.0339  110.339
3     1           3       0.0354  100.109
4     1           2       0.0380  101.665
```

Lets also get transform all the 'loudness' feature elements to positive numbers

[15]:
```
data.loudness=data.loudness.abs()
```

Lets create a smaller dataset where to apply the .describe() function without features such as 'mode' or 'year' for which it would be useless.

[16]:
```
data_small=data.drop(['year', 'key', 'mode'], axis=1)
```

[17]:
```
data_small.describe()
```

[17]:
```
             valence   acousticness   danceability   duration_ms  \
count  170653.000000  170653.000000  170653.000000  1.706530e+05
mean        0.528587       0.502115       0.537396  2.309483e+05
```

```
std        0.263171       0.376032       0.176138  1.261184e+05
min        0.000000       0.000000       0.000000  5.108000e+03
25%        0.317000       0.102000       0.415000  1.698270e+05
50%        0.540000       0.516000       0.548000  2.074670e+05
75%        0.747000       0.893000       0.668000  2.624000e+05
max        1.000000       0.996000       0.988000  5.403500e+06

              energy       explicit  instrumentalness       liveness  \
count  170653.000000  170653.000000     170653.000000  170653.000000
mean        0.482389       0.084575          0.167010       0.205839
std         0.267646       0.278249          0.313475       0.174805
min         0.000000       0.000000          0.000000       0.000000
25%         0.255000       0.000000          0.000000       0.098800
50%         0.471000       0.000000          0.000216       0.136000
75%         0.703000       0.000000          0.102000       0.261000
max         1.000000       1.000000          1.000000       1.000000

            loudness     popularity    speechiness          tempo
count  170653.000000  170653.000000  170653.000000  170653.000000
mean       11.468323      31.431794       0.098393     116.861590
std         5.697272      21.826615       0.162740      30.708533
min         0.007000       0.000000       0.000000       0.000000
25%         7.183000      11.000000       0.034900      93.421000
50%        10.580000      33.000000       0.045000     114.729000
75%        14.615000      48.000000       0.075600     135.537000
max        60.000000     100.000000       0.970000     243.507000
```

1.3 Feature Engineering

Lets take the 'data' dataset and encode the categorical feature 'artist'.

[18]:
```
data_enc=data.copy()
```

[19]:
```
enc = LabelEncoder()
```

[20]:
```
data_enc['artists_enc'] = enc.fit_transform(data_enc.artists)

print(data_enc[['artists_enc', 'artists']].head())
```

```
   artists_enc                                            artists
0        26839  ['Sergei Rachmaninoff', 'James Levine', 'Berli…
1         7382                                     ['Dennis Day']
2        16378  ['KHP Kridhamardawa Karaton Ngayogyakarta Hadi…
3        10077                                   ['Frank Parker']
4        23719                                     ['Phil Regan']
```

[21]:
```
data_enc=data_enc.drop(['artists'], axis=1)
```

[22]:
```
data_enc.head()
```

[22]:
```
   valence  year  acousticness  danceability  duration_ms  energy  explicit  \
0   0.0594  1921         0.982         0.279       831667   0.211         0
1   0.9630  1921         0.732         0.819       180533   0.341         0
2   0.0394  1921         0.961         0.328       500062   0.166         0
3   0.1650  1921         0.967         0.275       210000   0.309         0
4   0.2530  1921         0.957         0.418       166693   0.193         0

   instrumentalness  key  liveness  loudness  mode  popularity  speechiness  \
0          0.878000   10     0.665    20.096     1           4       0.0366
1          0.000000    7     0.160    12.441     1           5       0.4150
2          0.913000    3     0.101    14.850     1           5       0.0339
3          0.000028    5     0.381     9.316     1           3       0.0354
4          0.000002    3     0.229    10.096     1           2       0.0380

     tempo  artists_enc
0   80.954        26839
1   60.936         7382
2  110.339        16378
3  100.109        10077
4  101.665        23719
```

Before traing our method lets see if training sets have columns that are skewed and need transformation

Lets split the data in train and test sets with a random state set at 140

[23]:
```
#train, test = train_test_split(data_enc, test_size=0.3, random_state=140)a
data_enc.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170653 entries, 0 to 170652
Data columns (total 16 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   valence           170653 non-null  float64
 1   year              170653 non-null  int64
 2   acousticness      170653 non-null  float64
 3   danceability      170653 non-null  float64
 4   duration_ms       170653 non-null  int64
 5   energy            170653 non-null  float64
 6   explicit          170653 non-null  int64
 7   instrumentalness  170653 non-null  float64
 8   key               170653 non-null  int64
 9   liveness          170653 non-null  float64
 10  loudness          170653 non-null  float64
 11  mode              170653 non-null  int64
```

```
 12   popularity        170653 non-null  int64
 13   speechiness       170653 non-null  float64
 14   tempo             170653 non-null  float64
 15   artists_enc       170653 non-null  int64
dtypes: float64(9), int64(7)
memory usage: 20.8 MB
```

[24]:
```
mask = data_enc.dtypes == np.float
float_cols = data_enc.columns[mask]
```

[25]:
```
skew_limit = 0.75
skew_vals = data_enc[float_cols].skew()

skew_cols = (skew_vals
             .sort_values(ascending=False)
             .to_frame()
             .rename(columns={0:'Skew'})
             .query('abs(Skew) > {0}'.format(skew_limit)))

skew_cols
```

[25]:

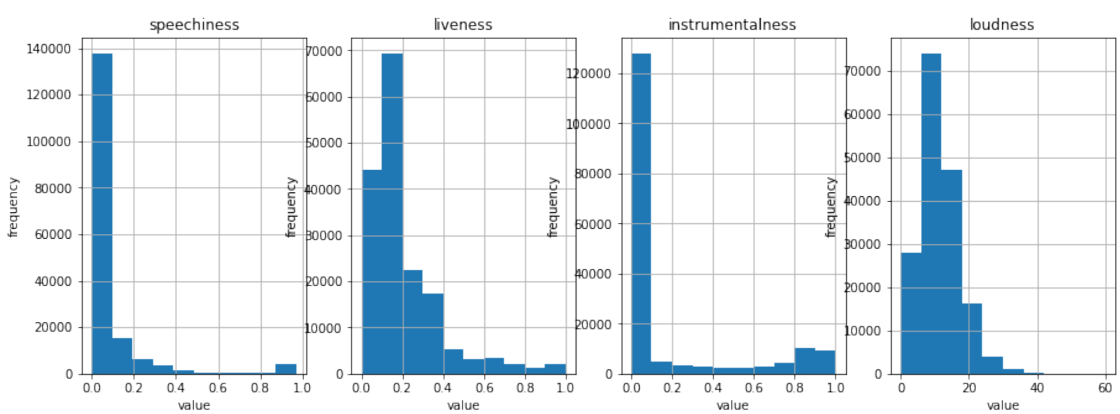
	Skew
speechiness	4.047848
liveness	2.154382
instrumentalness	1.631114
loudness	1.052758

Lets have a look

[26]:
```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(15, 5))

cols={ax1:'speechiness', ax2:'liveness', ax3:'instrumentalness', ax4:'loudness'}

for loc, name in cols.items():
    data_enc[name].hist(ax=loc)
    loc.set(title=name, ylabel='frequency', xlabel='value')
```



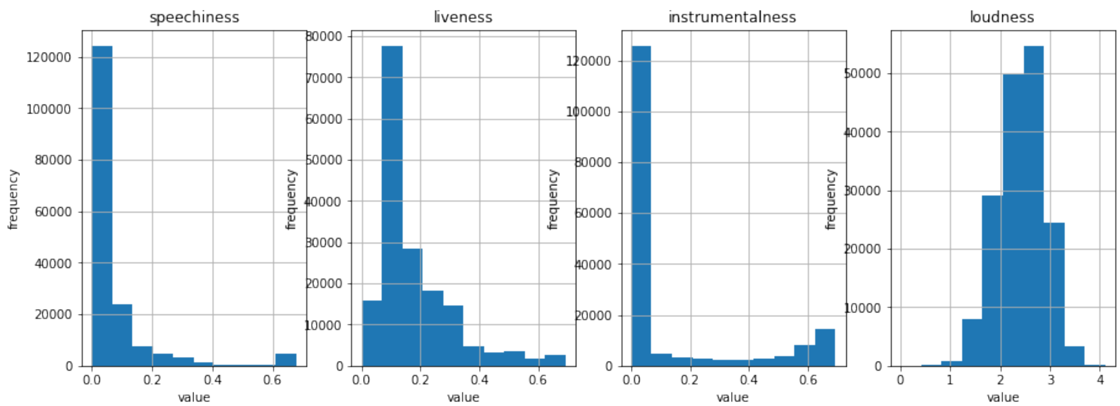
All these needs transofrmation

Sine everyting is right skewed lets do a log1p transformation

[27]:
```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(15, 5))

cols={ax1:'speechiness', ax2:'liveness', ax3:'instrumentalness', ax4:'loudness'}

for loc, name in cols.items():
    data_enc[name].apply(np.log1p).hist(ax=loc)
    loc.set(title=name, ylabel='frequency', xlabel='value')
```

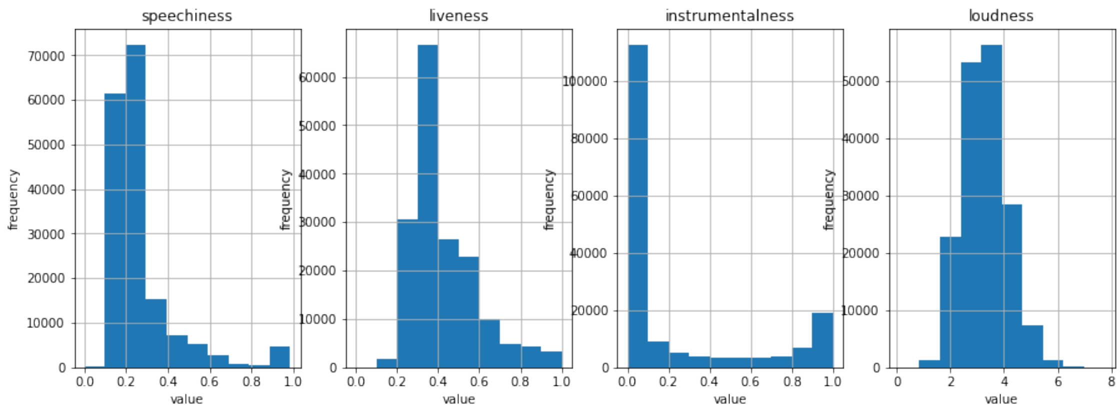


Lets try sqrt tranfromation too

[28]:
```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(15, 5))

cols={ax1:'speechiness', ax2:'liveness', ax3:'instrumentalness', ax4:'loudness'}

for loc, name in cols.items():
    data_enc[name].apply(np.sqrt).hist(ax=loc)
    loc.set(title=name, ylabel='frequency', xlabel='value')
```



Not great, but better then before. Lets see which is better. I did a k d'agostino normaltest but the p values where 0 for both transformations. So I will just check the skew again to see which transformation reduced it the most

[29]:
```
data_enc_log1p=data_enc.copy()
data_enc_sqrt=data_enc.copy()
```

[30]:
```
for col in skew_cols.index.values:
    data_enc_log1p[col] = data_enc[col].apply(np.log1p)
    data_enc_sqrt[col]=data_enc[col].apply(np.sqrt)
```

[31]:
```
normaltest(data_enc_log1p.loudness.values)
```

[31]:
```
NormaltestResult(statistic=1526.8000505110372, pvalue=0.0)
```

[32]:
```
trans_cols=['speechiness', 'liveness', 'instrumentalness', 'loudness']
```

[33]:
```
skew_limit = 0.75
skew_vals_log1p = data_enc_log1p[trans_cols].skew()
skew_vals_sqrt = data_enc_sqrt[trans_cols].skew()

skew_cols_log1p = (skew_vals_log1p
                  .sort_values(ascending=False)
                  .to_frame()
                  .rename(columns={0:'Skew'})
                  .query('abs(Skew) > {0}'.format(skew_limit)))

skew_cols_sqrt = (skew_vals_sqrt
                  .sort_values(ascending=False)
                  .to_frame()
                  .rename(columns={0:'Skew'})
                  .query('abs(Skew) > {0}'.format(skew_limit)))
```

[34]:
```
skew_vals_log1p
```

[34]:
```
speechiness         3.587121
liveness            1.748512
instrumentalness    1.540390
loudness           -0.229451
dtype: float64
```

[35]:
```
skew_vals_sqrt
```

[35]:
```
speechiness         2.857476
liveness            1.302035
instrumentalness    1.309775
loudness            0.326961
dtype: float64
```

Apparetly the sqrt transformation did a better job in normalizing the columns then the log1p. Lets apply the sqrt transformation to the data_enc dataframe instead on the test data_enc_sqrt

[36]:
```
for col in skew_cols.index.values:
    data_enc[col]=data_enc[col].apply(np.sqrt)
```

[37]:
```
data_enc.head()
```

[37]:
```
   valence  year  acousticness  danceability  duration_ms  energy  explicit  \
0   0.0594  1921         0.982         0.279       831667   0.211         0
1   0.9630  1921         0.732         0.819       180533   0.341         0
2   0.0394  1921         0.961         0.328       500062   0.166         0
3   0.1650  1921         0.967         0.275       210000   0.309         0
4   0.2530  1921         0.957         0.418       166693   0.193         0

   instrumentalness  key  liveness  loudness  mode  popularity  speechiness  \
0          0.937017   10  0.815475  4.482856     1           4     0.191311
1          0.000000    7  0.400000  3.527180     1           5     0.644205
2          0.955510    3  0.317805  3.853570     1           5     0.184120
3          0.005263    5  0.617252  3.052212     1           3     0.188149
4          0.001296    3  0.478539  3.177420     1           2     0.194936

     tempo  artists_enc
0   80.954        26839
1   60.936         7382
2  110.339        16378
3  100.109        10077
4  101.665        23719
```

Lets drop the instrumentalness feature because it doesnt look very useful

[38]:
```
data_enc=data_enc.drop(['instrumentalness'], axis=1)
```

[39]:
```
data_enc.head()
```

[39]:
```
   valence  year  acousticness  danceability  duration_ms  energy  explicit  \
0   0.0594  1921         0.982         0.279       831667   0.211         0
1   0.9630  1921         0.732         0.819       180533   0.341         0
2   0.0394  1921         0.961         0.328       500062   0.166         0
3   0.1650  1921         0.967         0.275       210000   0.309         0
4   0.2530  1921         0.957         0.418       166693   0.193         0

   key  liveness  loudness  mode  popularity  speechiness    tempo  \
0   10  0.815475  4.482856     1           4     0.191311   80.954
1    7  0.400000  3.527180     1           5     0.644205   60.936
2    3  0.317805  3.853570     1           5     0.184120  110.339
3    5  0.617252  3.052212     1           3     0.188149  100.109
4    3  0.478539  3.177420     1           2     0.194936  101.665

   artists_enc
0        26839
1         7382
2        16378
3        10077
4        23719
```

1.4 Data Exploration

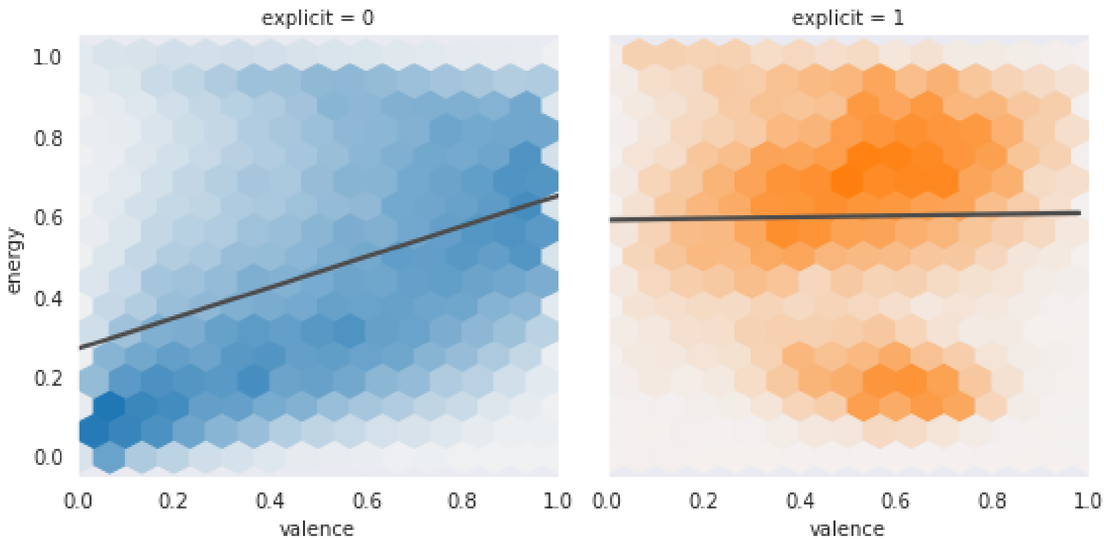
Lets see a little be the relations between our features and between features and target variable. Also lets separete the plotted data in explicit and non-explcit songs. I want to do this separation for two main reasons. First, it will be interesting to see the difference from explicit and non-explicit songs and second, given the large amount of data, splitting it I think will result in more clear and meaningful plots.

[124]:
```
def hexbin(x, y, color, **kwargs):
    cmap = sns.light_palette(color, as_cmap=True)
    plt.hexbin(x, y, gridsize=15, cmap=cmap, **kwargs)
```

First lets explore some features vs other features

Valence vs Energy for non-explicit (0) and explicit(1) songs

[132]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'valence', 'energy', extent=[0, 1, 0, 1])
g.map(sns.regplot, 'valence', 'energy', scatter=False, x_estimator=np.mean,␣
 ↪color='.3')
```

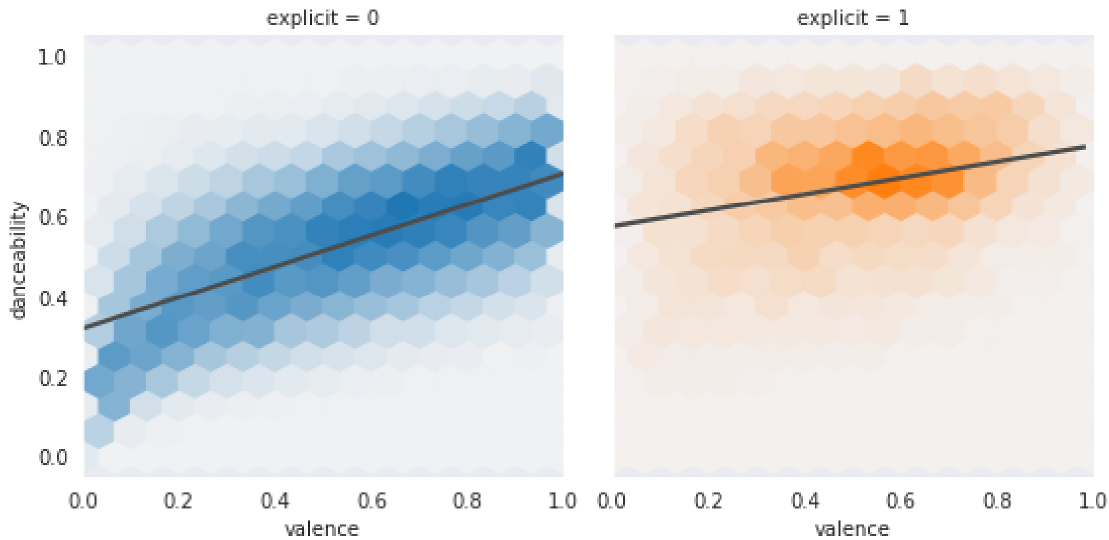


Non-explcit song have a high concentration at the two extreme of the valence range which slight more frequency to the sad side (valence 0). Also it is interesting to see that for non-explicit very sad songs the energy is very low while for happy songs the energy is higher. On the other hand for explicit songs the valence is more focused in the center in middle values. We can identify two different groups, one more expanded characterized by a pretty high energy level average and another one less expended characterized by a low energy level

Valence vs Danceability for non-explicit (0) and explicit(1) songs

[135]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'valence', 'danceability', extent=[0, 1, 0, 1])
g.map(sns.regplot, 'valence', 'danceability', scatter=False, x_estimator=np.
 ↪mean, color='.3')
```

[135]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe7e4506898>
```



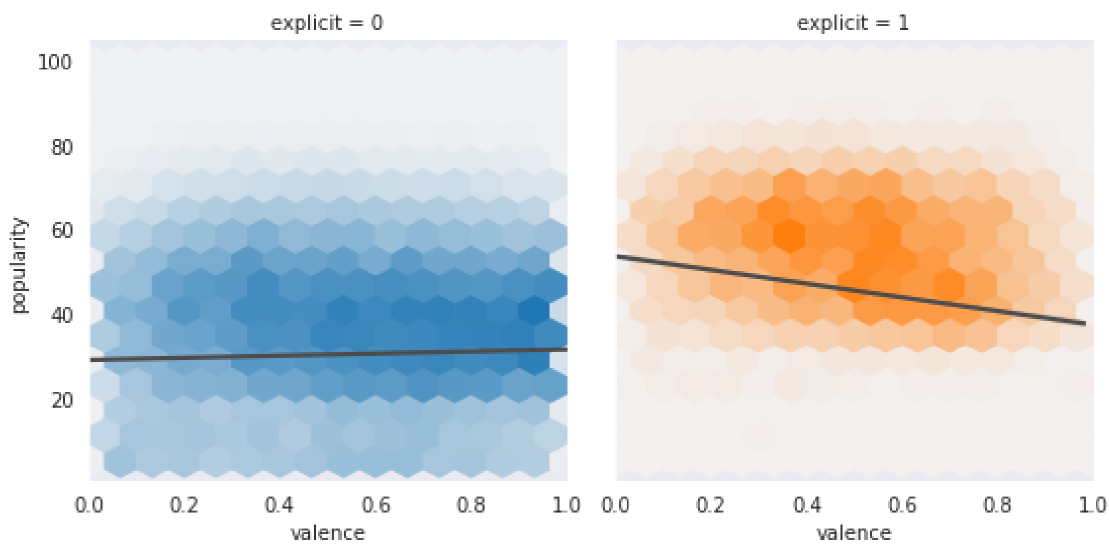
We can see that as for non-explicit songs the happier they get the more danceable they become while the explicit song presents are concentrated in the mid valence values and have a pritty high average danceability and increasing from sad to happy songs.

Lets start with a basic popularity of 5 to avoid having to consider unknown songs that will only make our plots more confusing

Valence vs Popularity for non-explicit (0) and explicit(1) songs

[136]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'valence', 'popularity', extent=[0, 1, 5, 100])
g.map(sns.regplot, 'valence', 'popularity', scatter=False, x_estimator=np.mean,␣
 ↪color='.3')
```

[136]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe7e44416a0>
```



[98]:
```
data_enc.explicit.value_counts()
```

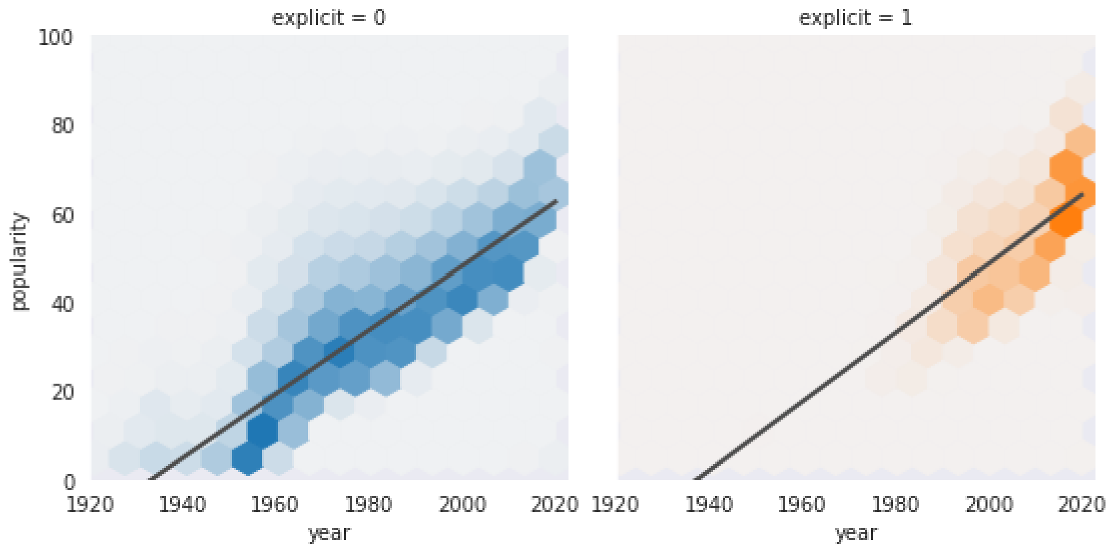
[98]:
```
0    156220
1     14433
Name: explicit, dtype: int64
```

Lets keep in mind that there are 156220 non explicit songs and 14433 explicit ones. However, it looks like the non explicit songs are most likely to be under 60 of popularity and stacked towards the positive/happy feeling while the explicit tend to be more popular with popularity between 40 and 80 with almost no presence under the 40 popularity. Moreover, explicit songs tend to be concentrated in the middle of the valence(positiveness) range and their popularity decreases the more happy the song is. Which means that an explicit song is more popular is it is sad than happy. Kind of makes sense.

Year vs Popularity for non-explicit (0) and explicit(1) songs

[181]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'year', 'popularity', extent=[1921, 2020, 5, 100])
g.map(sns.regplot, 'year', 'popularity', scatter=False, x_estimator=np.mean,␣
 ↪color='.3')
g.set(xlim=(1920,2023), ylim=(0, 100))
```

[181]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe7e358b208>
```

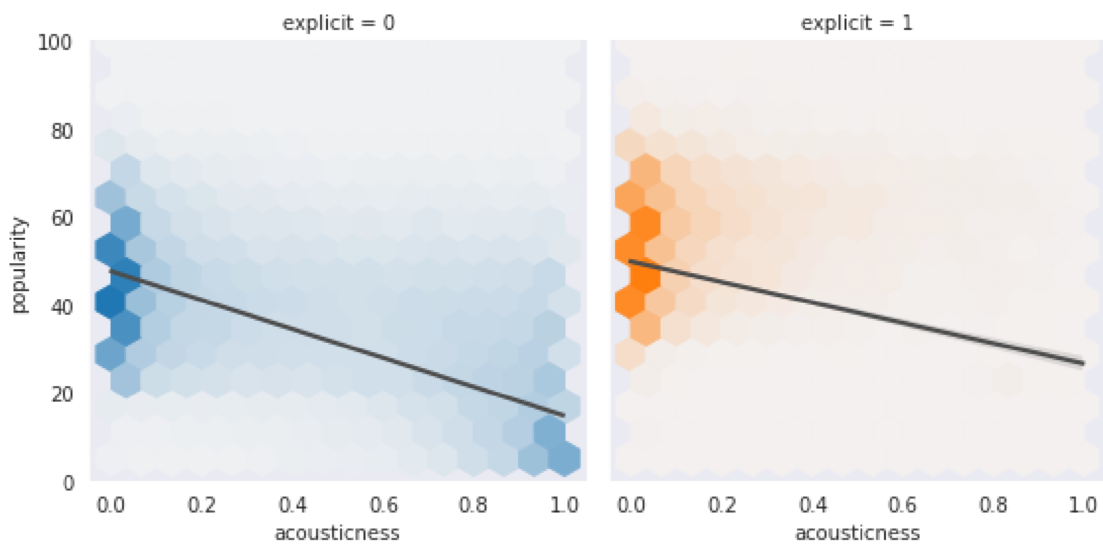


This is interesting we can see that before 1990 songs were practically all non-explicit indicating a change of the morality and what was considered unaccepable before 1990. We can also see that while non-explicit songs tend to be pretty stable in time, maybe with a little decline towards 2020, the explicit traks seems to grow in popularity as time progresses.

Acousticness vs Popularity for non-explicit (0) and explicit(1) songs

[183]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'acousticness', 'popularity', extent=[0, 1, 5, 100])
g.map(sns.regplot, 'acousticness', 'popularity', scatter=False, x_estimator=np.
 ↪mean, color='.3')
g.set(xlim=(-0.05,1.05), ylim=(0, 100))
```

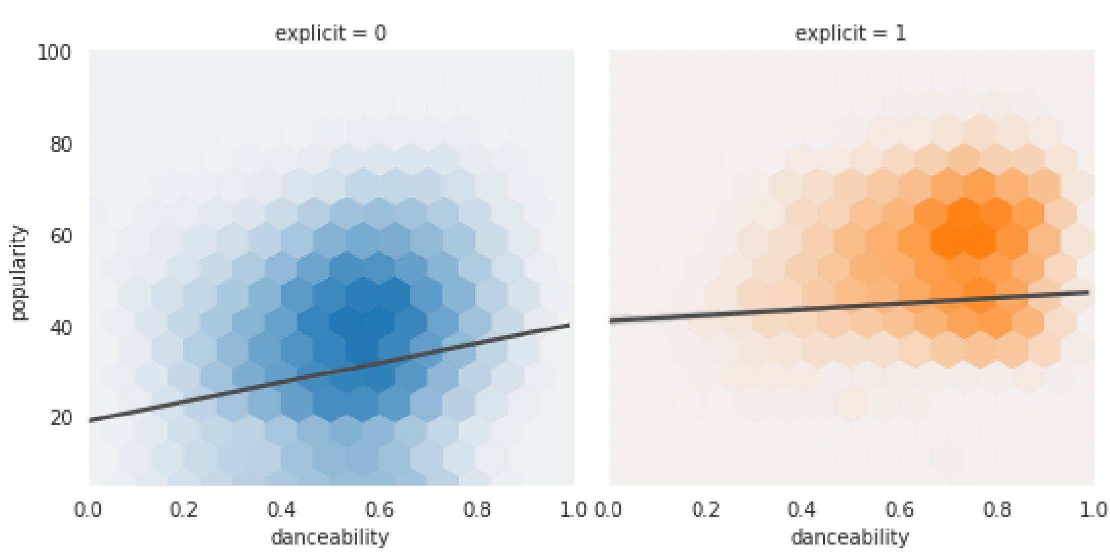
[183]: <seaborn.axisgrid.FacetGrid at 0x7fe7e2b3e1d0>



We can see that almost all songs are in the non-acustic sides for both explicit and non-explicit. Of course, the only precence of tracks with a good acousticness character can be seen only in the non-explicit plot, as it should be. However the popularity of the songs decreases as the more acustic the track is.

[173]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'danceability', 'popularity', extent=[0, 1, 5, 100])
g.map(sns.regplot, 'danceability', 'popularity', scatter=False, x_estimator=np.
 ↪mean, color='.3')
g.set(xlim=(0,1), ylim=(0, 100))
```

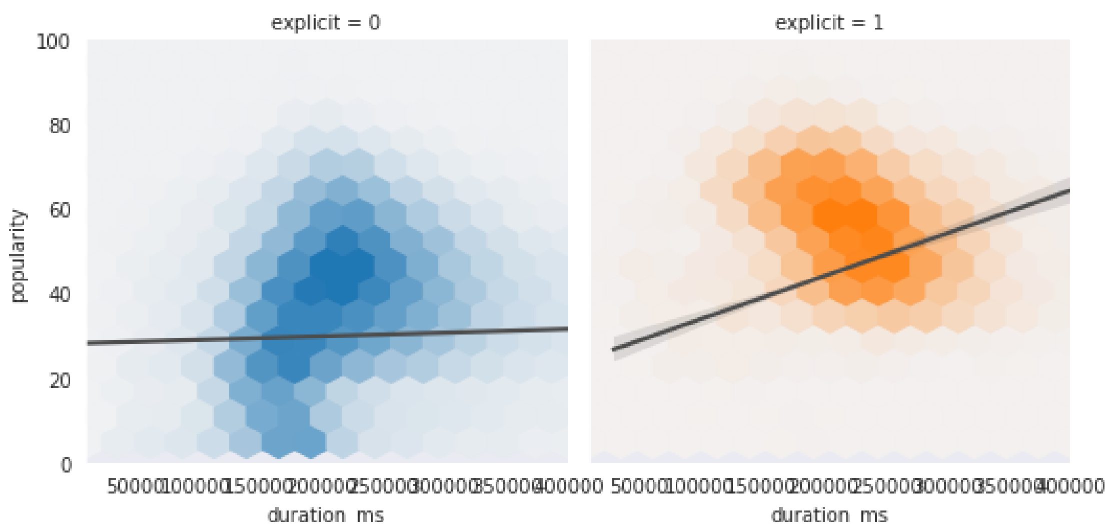
[173]: <seaborn.axisgrid.FacetGrid at 0x7fe7e2c3c9e8>



For non-explicit songs popularity tend to increase as the danceability increases while the majority of the tanks are in the middle ranges of danceability. The popularity of explicit songs, on the other hand, dont vary much with the danceability and the majority of the tracks are, on average, more danceable then the non-explicit ones.

[184]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'duration_ms', 'popularity', extent=[1e4, 4e5, 5, 100])
g.map(sns.regplot, 'duration_ms', 'popularity', scatter=False, x_estimator=np.
↪mean, color='.3')
g.set(xlim=(1e4,4e5), ylim=(0, 100))
```

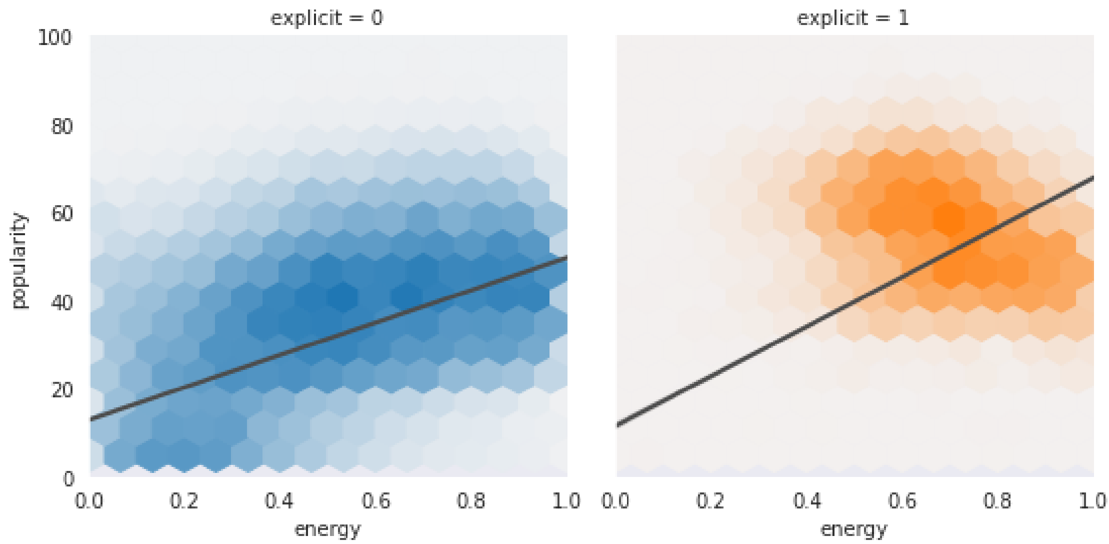
[184]: <seaborn.axisgrid.FacetGrid at 0x7fe7e2a70e48>



On average, explicit and non-explicit songs tend to have comparable time-lenght. We can see that explicit songs seems to be a little more popular the non-explicit ones with similar track duration.

[185]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'energy', 'popularity', extent=[0, 1, 5, 100])
g.map(sns.regplot, 'energy', 'popularity', scatter=False, x_estimator=np.mean,␣
↪color='.3')
g.set(xlim=(0,1), ylim=(0, 100))
```

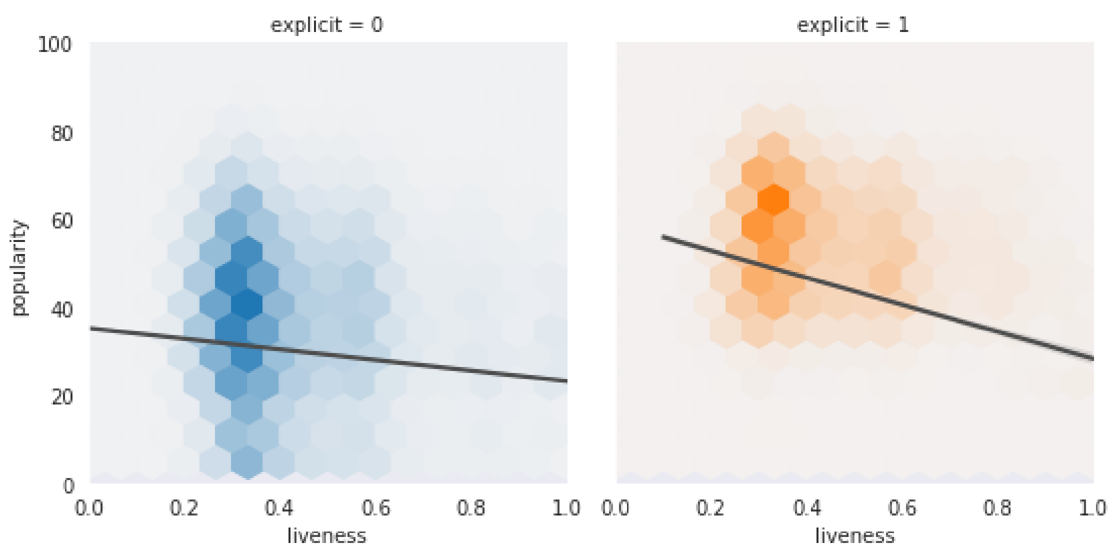
[185]: <seaborn.axisgrid.FacetGrid at 0x7fe7e29b35f8>



For both cases the more a song is energetic the more popular it is.It is interesting to see that Non-explicit songs are diffused across all energy levels while explcit ones are focused on the high energy end.

[186]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'liveness', 'popularity', extent=[0, 1, 5, 100])
g.map(sns.regplot, 'liveness', 'popularity', scatter=False, x_estimator=np.
 ↪mean, color='.3')
g.set(xlim=(0,1), ylim=(0, 100))
```

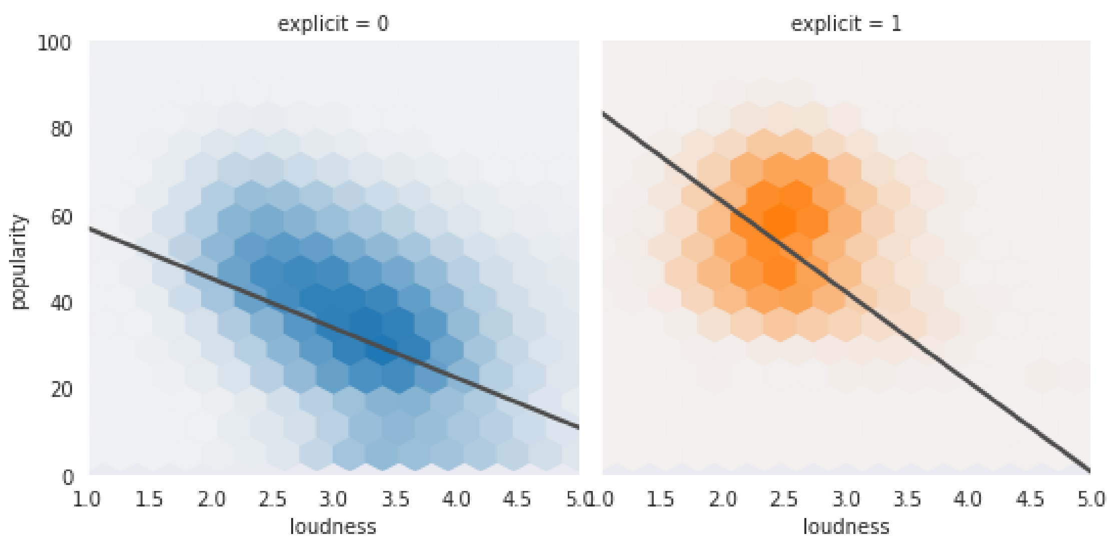
[186]: <seaborn.axisgrid.FacetGrid at 0x7fe7e28f7978>



We see that both non-explicit and explicit songs have simialar level of liveness feeling and have very similar behaviours.

[187]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'loudness', 'popularity', extent=[1, 5, 5, 100])
g.map(sns.regplot, 'loudness', 'popularity', scatter=False, x_estimator=np.
 ↪mean, color='.3')
g.set(xlim=(1,5), ylim=(0, 100))
```

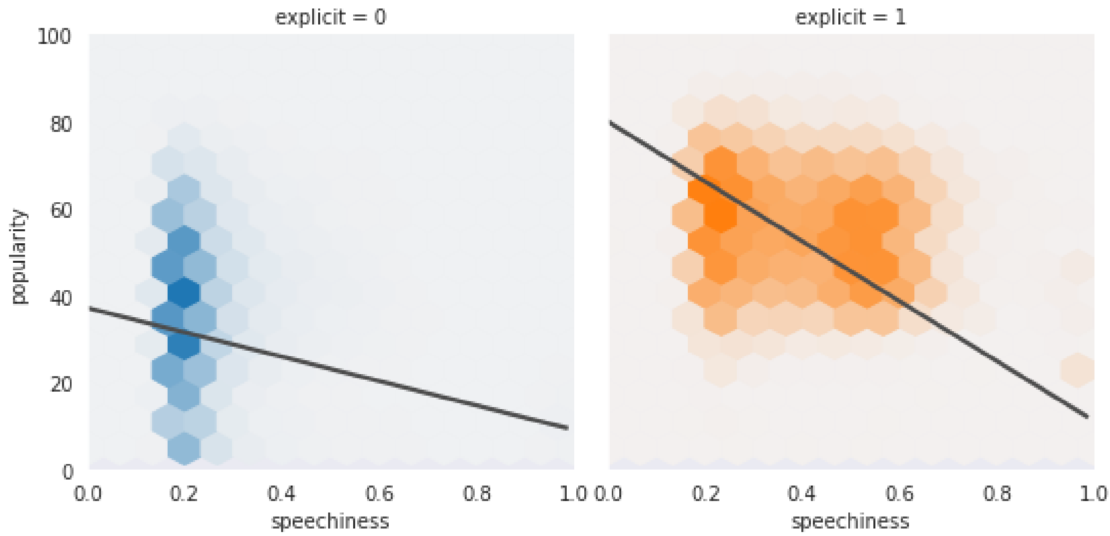
[187]: <seaborn.axisgrid.FacetGrid at 0x7fe7e282fda0>



In the freature engineering I took the absolute value for the laudness because it was ranging from -60 and 0 and it would not have been possible to do a square root transformation on that fearture. Therefore the behavior we see here should be mirrored on the y axis. The the popularity increases with an increment in the loudness feature. Also we see that explicit songs tend to have a higher loudness then non-explicit songs.

[188]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'speechiness', 'popularity', extent=[0, 1, 5, 100])
g.map(sns.regplot, 'speechiness', 'popularity', scatter=False, x_estimator=np.
 ↪mean, color='.3')
g.set(xlim=(0,1), ylim=(0, 100))
```

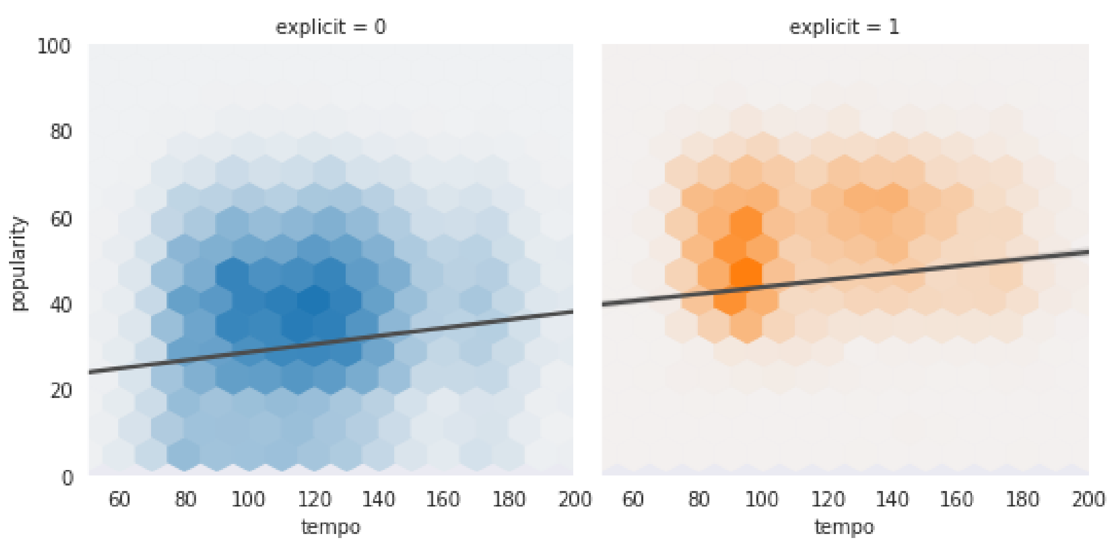
[188]: <seaborn.axisgrid.FacetGrid at 0x7fe7e2769400>



This is interesting because we see that non-explicit songs have a speechiness concentrated around the 0.2 mark which is pretty low. However the explicit songs are more speechy but still the popularity of the track diminishes as speechiness increases.

[190]:
```
with sns.axes_style('dark'):
    g = sns.FacetGrid(data_enc, hue='explicit', col='explicit', height=4)
g.map(hexbin, 'tempo', 'popularity', extent=[50, 200, 5, 100])
g.map(sns.regplot, 'tempo', 'popularity', scatter=False, x_estimator=np.mean,␣
 ↪color='.3')
g.set(xlim=(50,200), ylim=(0, 100))
```

[190]: <seaborn.axisgrid.FacetGrid at 0x7fe7e25d14e0>



Most of the non-explicit songs have a tempo range between 80 and 140 bpm while the explicit tracks are more frequent in the 80 to 100 bpm. For both explicit and non-explicit the song popularity tend to increase with increased tempo.

1.5 Machine Learning

First of all lest take our target variable 'popularity' out from the other features.

[194]:
```
X = data_enc.drop('popularity', axis=1)
y = data_enc['popularity']
```

Lets get going with the train/test splits

[195]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=43210)
```

[196]:
```
#Lets save all the R2 scores, MSE, and Alphas for the test set from all the␣
↪methods in a pandas series
errors=[]
```

[197]:
```
LR = LinearRegression()

LR = LR.fit(X_train, y_train)
y_train_pred = LR.predict(X_train)
y_test_pred = LR.predict(X_test)


#Saving it to the 2 arrays
errors.append(pd.Series({'MSE' : mean_squared_error(y_test,  y_test_pred), 'R2'␣
↪: r2_score(y_test,  y_test_pred), 'Alphas' : 'No alpha'}, name='LR'))
```

[198]:
```
print('R2:', r2_score(y_test,  y_test_pred), '\nMSE:',␣
↪mean_squared_error(y_test,  y_test_pred), '\nalpha:', 'No alpha')
```

```
R2: 0.7533323362083683
MSE: 117.63853180989686
alpha: No alpha
```

Vanilla Linear Regression has a R2 > 75% which is acceptable. Anyways lets see if we can do better with Ridge and Lasso Regression.

First lets scale introduce Polynomial features and Standard scaler. Lest try using the a pipeline because its amazing and elegant

Lets define a Kfold cross validation object so we ensure we are suffling and that we have the same randomstate

[199]:
```
kf = KFold(shuffle=True, random_state=43210, n_splits=5)
```

[202]:
```
estimator_ridge = Pipeline([("scaler", StandardScaler()),
                ("polynomial_features", PolynomialFeatures()),
                ("ridge_regression", Ridge())])

#After few tries I evaluated that the best alpha is around 100-110
#Unfortunatly I cannot go above polynomial degree 3 because my laptop doesnt␣
↪have enough memory to allocate the arrays
params = {
    'polynomial_features__degree': [1, 2, 3],
    'ridge_regression__alpha': np.geomspace(100, 110, 20)
}

grid_ridge = GridSearchCV(estimator_ridge, params, cv=kf)
grid_ridge.fit(X, y)
```

[202]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=43210, shuffle=True),
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('polynomial_features',
                                        PolynomialFeatures()),
                                       ('ridge_regression', Ridge())]),
             param_grid={'polynomial_features__degree': [1, 2, 3],
                         'ridge_regression__alpha': array([100.        ,
100.50289281, 101.00831463, 101.51627818,
       102.02679624, 102.53988166, 103.05554735, 103.57380628,
       104.09467151, 104.61815612, 105.14427331, 105.67303629,
       106.20445839, 106.73855298, 107.27533348, 107.81481342,
       108.35700636, 108.90192595, 109.4495859 , 110.        ])})
```

[203]:
```
grid_ridge.best_score_, grid_ridge.best_params_
```

[203]:
```
(0.7740622730533511,
 {'polynomial_features__degree': 3,
  'ridge_regression__alpha': 106.20445839236804})
```

Lets predict

[204]:
```
y_predict_ridge = grid_ridge.predict(X)
```

[205]:
```
print('R2', r2_score(y, y_predict_ridge), '\nMSE', mean_squared_error(y, ␣
↪y_predict_ridge), '\nalpha', 132.728558)
```

```
R2 0.776354576393793
MSE 106.5443078393529
alpha 132.728558
```

Saving them to the arrays

[206]:
```
errors.append(pd.Series({'MSE' : mean_squared_error(y,  y_predict_ridge), 'R2' :
↪ r2_score(y, y_predict_ridge), 'Alphas' : 132.728558}, name='RR'))
```

Using other scaler method such as MinMaxScaler leads to more complex model with lower R2.

Lets set a new estimator with a Lasso Regression

[224]:
```
estimator_lasso = Pipeline([("scaler", StandardScaler()),
             ("polynomial_features", PolynomialFeatures()),
             ("lasso_regression", Lasso(max_iter=5000))])

#After few tries I evaluated that the best alpha is lower then 0,08 but my␣
↪laptop cannot handle models more complex then than that.
#Unfortunatly I cannot go above polynomial degree 3 because my laptop doesnt␣
↪have enough memory to allocate the arrays
params = {
    'polynomial_features__degree': [1, 2],
    'lasso_regression__alpha': np.geomspace(0.008, 0.01, 10)
}

grid_lasso = GridSearchCV(estimator_lasso, params, cv=kf)
grid_lasso.fit(X, y)
```

[224]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=43210, shuffle=True),
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('polynomial_features',
                                        PolynomialFeatures()),
                                       ('lasso_regression',
                                        Lasso(max_iter=5000))]),
             param_grid={'lasso_regression__alpha': array([0.008     ,
0.00820083, 0.0084067 , 0.00861774, 0.00883408,
       0.00905584, 0.00928318, 0.00951622, 0.00975511, 0.01      ]),
                         'polynomial_features__degree': [1, 2]})
```

[226]:
```
grid_lasso.best_score_, grid_lasso.best_params_
```

[226]:
```
(0.7636240439341375,
 {'lasso_regression__alpha': 0.008000000000000004,
  'polynomial_features__degree': 2})
```

[227]:
```
y_predict_lasso = grid_lasso.predict(X)
```

[228]:
```
r2_score(y, y_predict_lasso)
```

[228]:
```
0.7640169734051945
```

[229]:
```
mean_squared_error(y,  y_predict_lasso)
```

[229]: 112.4219213832434

[256]:
```
grid_lasso.best_estimator_.named_steps['lasso_regression'].coef_
```

[256]:
```
array([ 0.00000000e+00, -7.47018688e-02,  1.61898669e+01, -2.39442216e+00,
        7.71700220e-01, -2.72736779e-01, -4.12394442e-01, -0.00000000e+00,
        2.34514861e-02, -3.60194707e-01, -9.58898312e-01, -1.17397973e-01,
       -1.57463720e+00, -3.31587010e-02, -3.24445181e-01, -5.58055206e-01,
       -1.02849956e-01, -2.92665332e-01,  5.08264395e-01,  4.50691146e-01,
        1.07756953e+00, -2.44496309e-01, -0.00000000e+00,  6.42930010e-02,
        6.41340230e-01, -5.49328682e-02,  0.00000000e+00,  1.34841278e-01,
       -2.40710045e-02,  4.86627553e-01,  1.52605860e+00, -7.27468927e-01,
       -3.75416542e-01, -3.47965477e-01,  1.04093322e+00, -1.64847116e-02,
       -5.91957155e-02, -2.53136837e-01, -1.06550114e-01, -3.59521356e-01,
       -1.23143689e-01,  3.25697400e-01, -1.91867845e+00,  1.48876659e-01,
       -1.55424850e-01, -7.42652084e-02,  8.93063049e-02, -6.63752676e-02,
       -2.88744173e-02,  1.11850069e+00,  9.50121014e-02, -1.83910899e-01,
        6.10301051e-02, -3.25254585e-02, -2.64265873e-01, -2.90873536e-01,
       -7.24194630e-01,  3.03394082e-01, -1.78241718e-03,  6.49625677e-02,
       -6.27490610e-01, -0.00000000e+00,  6.98932015e-03, -1.04187761e-01,
       -9.81345109e-02,  7.71334271e-03,  7.28285461e-02, -2.77962499e-02,
        4.38856899e-02,  1.66266426e-02,  1.78778707e-01,  3.00879383e-02,
        0.00000000e+00, -2.08146509e-02,  1.80561388e-02, -4.94842041e-01,
       -1.87945899e-01, -9.94898604e-02,  1.72340885e-02, -2.53270976e-02,
        1.21394178e-01,  4.98446592e-01, -3.24008413e-01,  1.87000920e-01,
       -1.10930017e-01,  2.33671983e-02,  5.91519213e-02,  1.34189081e-01,
        5.04786950e-02,  7.38324535e-02,  1.49316152e-01, -4.30237068e-02,
       -6.49728640e-02, -0.00000000e+00,  1.59664853e-02,  6.63670508e-02,
        4.66771081e-02,  4.94695771e-02, -2.55595675e-02, -4.38883963e-02,
       -0.00000000e+00,  5.15903895e-02,  1.65804219e-02, -3.85010382e-02,
       -0.00000000e+00, -1.56103517e-01, -7.02585417e-03, -1.18571997e-01,
       -2.22884891e-01,  1.88557488e-01,  0.00000000e+00, -8.27572876e-04,
        4.61287251e-02,  1.09550403e-02,  1.25088789e-01,  0.00000000e+00,
        2.70446572e-02,  1.54847212e-01, -6.53517728e-02,  4.82418818e-02])
```

[281]:
```
pd.DataFrame(zip(X.columns, grid_lasso.best_estimator_.
 ↪named_steps['lasso_regression'].coef_)).sort_values(by=1)
```

[281]:

	0	1
3	danceability	-2.394422
12	tempo	-1.574637
10	mode	-0.958898
6	explicit	-0.412394
9	loudness	-0.360195
5	energy	-0.272737
11	speechiness	-0.117398
1	year	-0.074702

```
13   artists_enc  -0.033159
0        valence   0.000000
7            key  -0.000000
8       liveness   0.023451
4    duration_ms   0.771700
2   acousticness  16.189867
```

[231]:
```
errors.append(pd.Series({'MSE' : mean_squared_error(y,  y_predict_lasso), 'R2' :
↪ r2_score(y, y_predict_lasso), 'Alphas' : 0.08 }, name='RR'))
```

Lets try an elastic net

[268]:
```
estimator_elastic = Pipeline([("scaler", StandardScaler()),
            ("polynomial_features", PolynomialFeatures()),
            ("elastic_regression", ElasticNet())])

#After few tries I evaluated that the best alpha is lower then 0,08 but my␣
↪laptop cannot handle models more complex then than that.
#Unfortunatly I cannot go above polynomial degree 3 because my laptop doesnt␣
↪have enough memory to allocate the arrays
params = {
    'polynomial_features__degree': [1, 2],
    'elastic_regression__alpha': np.geomspace(0.01, 0.10, 10),
    'elastic_regression__l1_ratio' : [0.5, 0.6, 0.7, 0.8]
}

grid_elastic = GridSearchCV(estimator_elastic, params, cv=kf)
grid_elastic.fit(X, y)
```

[268]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=43210, shuffle=True),
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('polynomial_features',
                                        PolynomialFeatures()),
                                       ('elastic_regression', ElasticNet())]),
             param_grid={'elastic_regression__alpha': array([0.01      ,
0.0129155 , 0.01668101, 0.02154435, 0.02782559,
       0.03593814, 0.04641589, 0.05994843, 0.07742637, 0.1       ]),
                         'elastic_regression__l1_ratio': [0.5, 0.6, 0.7, 0.8],
                         'polynomial_features__degree': [1, 2]})
```

[269]:
```
grid_elastic.best_score_, grid_elastic.best_params_
```

[269]:
```
(0.7636123064961626,
 {'elastic_regression__alpha': 0.01,
  'elastic_regression__l1_ratio': 0.8,
  'polynomial_features__degree': 2})
```

[270]:
```
y_predict_elsatic = grid_elastic.predict(X)
```

[271]:
```
r2_score(y, y_predict_elsatic)
```

[271]: 0.7640025753707842

[272]:
```
mean_squared_error(y,  y_predict_elsatic)
```

[272]: 112.42878058288966

[273]:
```
errors = pd.concat(errors, axis = 1)
```

[274]:
```
errors
```

[274]:

	LR	RR	RR
MSE	117.638532	106.544308	112.421921
R2	0.753332	0.776355	0.764017
Alphas	No alpha	132.728558	0.080000