REPORT

LAB OF INFORMATION SYSTEM ANALYTICS

GROUP MEMBERS:

NIKLAS SELCH, MAT: **887250**
LORENZO NICHELE, MAT: **887066**
ANDREA SALUSSO, MAT: **885032**
MATTIA DONADEI, MAT: **884911**

INDEX:

PROJECT INTRODUCTION:

With the growth of the internet and social media, music data is growing at an enormous rate!

Music analytics has a wide canvas covering all aspects related to music. Machine learning has taken a central role in the progress of many domains including music analytics.

Faster computational speed and increasing number of online users have resulted in a dramatic increase in music consumption.

It is getting more and more difficult for the general public, especially non-experts, to find and retrieve music from the millions of songs available online. Currently, there are several musical retrieval and similar artist recommendation apps.

We decided to start working with machine learning by managing music datasets to discover interesting patterns and give us the opportunity to expand our knowledge in this tricky but very interesting world

DATASETS USED:

We used three different datasets, here below you can find the links:

https://www.kaggle.com/datasets/imuhammad/audio-features-and-lyrics-of-spotify-songs?resource=download → load

https://www.kaggle.com/datasets/imuhammad/audio-features-and-lyrics-of-spotify-songs?resource=download → lcs

https://www.kaggle.com/datasets/neisse/scrapped-lyrics-from-6-genres → pop

DATA LOADING & LIBRARIES:

We proceeded following order to accomplish our goals: firstly we imported all the modulus and secondly we focused on some data-cleaning stuff.

```
import numpy as np
import pandas as pd
import copy
from tqdm import tqdm
import warnings
warnings.filterwarnings("ignore")
from scipy import stats
```

The most important libraries we used are *Numpy* and *Pandas*:

1. Numpy: NumPy is a Python library used for working with arrays. It also has functions for working in the domain of linear algebra, fourier transform, and matrices.

2. Pandas is an open source Python package that is most widely used for data science/data analysis and machine learning tasks. It is built on top of another package named Numpy, which provides support for multi-dimensional arrays.

```
#ML stuff
from sklearn.neighbors import KNeighborsClassifier
from sklearn.utils import shuffle
from sklearn import *
from sklearn.cluster import *
import tensorflow
import keras
import sklearn
from sklearn import metrics
from sklearn.preprocessing import scale
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import PCA
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import confusion_matrix
```

From ML we imported *SKlearn,* which is the most useful library for machine learning in Python.

The sklearn library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction. Scikit-learn comes loaded with a lot of features: Supervised learning algorithms, Cross-validation, Unsupervised learning algorithms, Various toy datasets, Feature extraction.

```
#plot stuff
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
from matplotlib import style
from IPython.core.display import Image
from mlxtend.plotting import plot_decision_regions
from collections import Counter, defaultdict
from PIL import Image
from nltk.corpus import stopwords, movie_reviews
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator
stopwords = set(STOPWORDS)
```

It's suitable to underline two important libraries such as *Matplolib* and *Seaborn*:

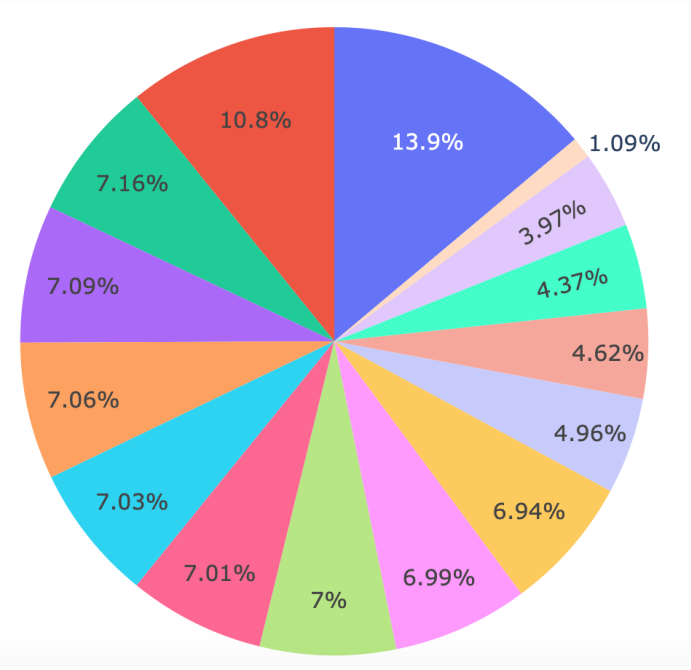
1. Matplotlib: is a comprehensive library for creating static, animated, and interactive visualizations in Python.

2. Seaborn: Is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

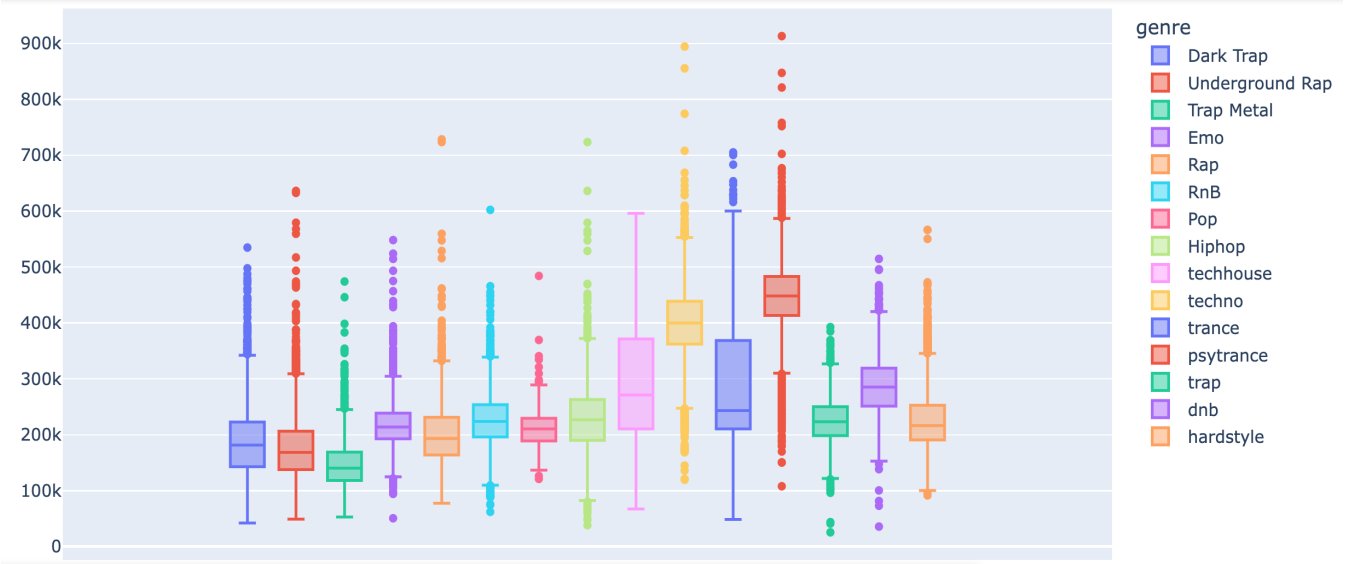
GENERAL ANALYSIS:

Spotify genre distribution:

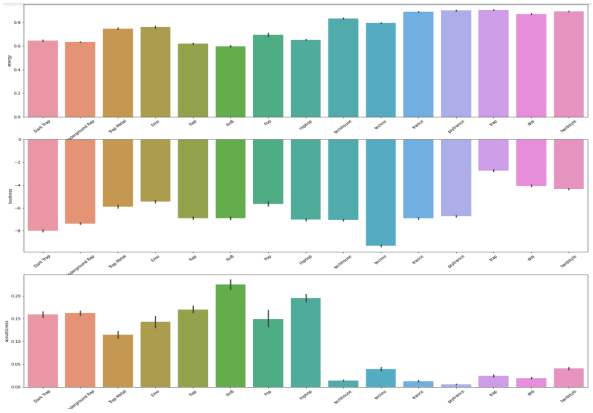
We found out that the three most *distributed* genres are, in that order: *Underground Rap, Dark Trap, and Hip-hop*.
From the Pie Chart below is possible to see the distribution of all other genres:



Genre difference among features:
We underlined the difference between the genres among some features, for example,

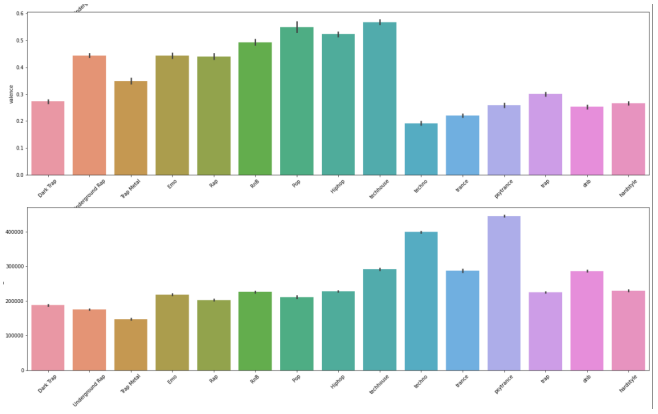


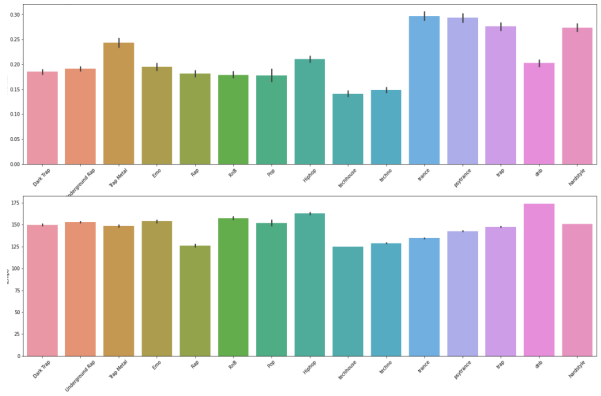
we discovered that the genre with the lower level of *duration* is Trap Metal.

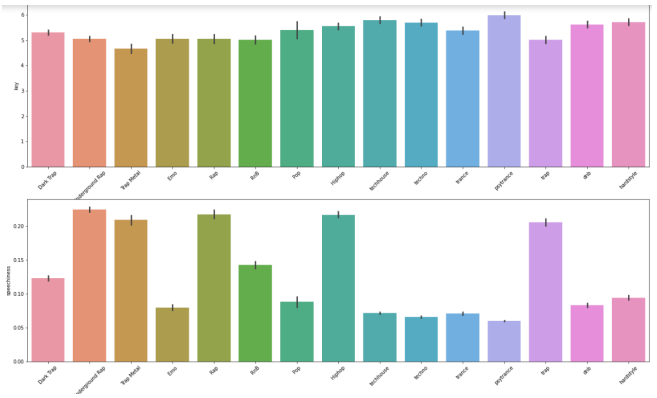


Music duration by genre:
Moreover, we discovered that by mean the genre with the *longest song* is Psytrance.

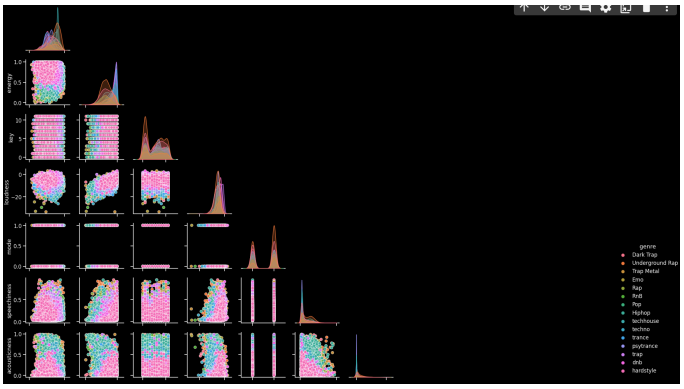
Feature correlation:
We tried two types of *correlation methods*: the first one, the *airport method* was not very clear at all, so we tried to make a *heatmap* to be more precise. Indeed we learned, for example, that *danceability* is strictly correlated with the energy of the song.

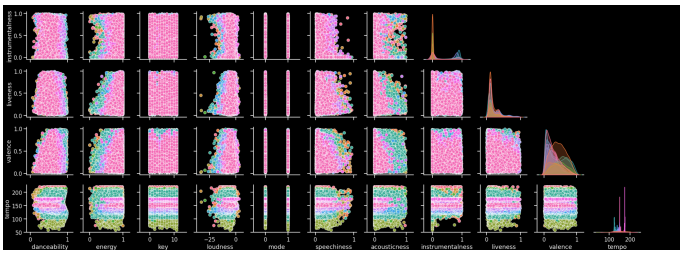






MUSIC RECOMMENDATION SYSTEM:





Feature engineering:

Feature engineering is the process of selecting, manipulating, and transforming raw data into features that can be used in supervised learning.

To make machine learning work well on new tasks, it might be necessary to design and train better features.

We used our data to calculate the *distances* between the songs and our feature's data varies, then we created a function to normalize it.
We got all the numerical columns and normalized them.

```
def normalize_column(col):
    max_d = data[col].max()
    min_d = data[col].min()
    data[col] = (data[col] - min_d)/(max_d - min_d)
```

```
num_types = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64', 'float128']
num = data.select_dtypes(include=num_types)

data. dropna()


for col in num.columns:
    normalize_column(col)
```

There was a probability that songs from the different genres could have had quite similar characteristics, and that was not fine.
For example, Sfera Ebbasta songs wouldn't be an accurate recommendation for Queen songs.

That's why we created a new feature, which would differentiate the songs from different groups.
We used *KMeans clusterization* with 10 clusters for this goal.

The K-means clustering algorithm is used to find groups that have not been explicitly labeled in the data. This can be used to confirm business assumptions about what types of groups exist or to identify unknown groups in complex data sets.

```
from sklearn.cluster import KMeans

data.dropna()

km = KMeans(n_clusters=10, )
cat = km.fit_predict(num)


data['cat'] = cat
normalize_column('cat')
```

Recommendation system:

We created a *class* in order to make the recommendations for our songs.
To find the difference between the songs, we calculated the *Manhattan distance* between all of them and, as a result, we chose the songs with the smallest distances.

But before it is crucial to define what a class is:
It’s a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together.
Creating a new class creates a new type of object, allowing new instances of that type to be made.

Moreover, Manhattan distance is made in order to measure the dissimilarity between two vectors and is commonly used in many machine learning algorithms.

```
class SpotifyRecommender():
    def __init__(self, rec_data):
        #our class should understand which data to work with
        self.rec_data_ = rec_data

    #if we need to change data
    def change_data(self, rec_data):
        self.rec_data_ = rec_data

    #function which returns recommendations, we can also choose the amount of songs to be recommended
    def get_recommendations(self, song_name, amount=1):
        distances = []
        #choosing the data for our song
        song = self.rec_data_[(self.rec_data_.song_name.str.lower() == song_name.lower())].head(1).values[(
        #dropping the data with our song
        res_data = self.rec_data_[self.rec_data_.song_name.str.lower() != song_name.lower()]
```

```
        for r_song in tqdm(res_data.values):
            dist = 0
            for col in np.arange(len(res_data.columns)):
                #indeces of non-numerical columns
                if not col in [5, 11, 12, 13, 14, 15, 18,19]:
                    #calculating the manhettan distances for each numerical feature
                    dist = dist + np.absolute(float(song[col]) - float(r_song[col]))
            distances.append(dist)
        res_data['distance'] = distances
        #sorting our data to be ascending by 'distance' feature
        res_data = res_data.sort_values('distance')
        columns = ['genre', 'song_name']
        return res_data[columns][:amount]
```

At the end of this recommendation system, we created the object of our *recommender*, and then we tried it!

```
recommender.get_recommendations('Head Straight', 3)
```

```
100%|██████████| 42303/42303 [00:02<00:00, 15378.25it/s]
```

	genre	song_name
11933	Trap Metal	Following the Breadcrumb Trail by Proxy
5424	Underground Rap	eighty
7174	Underground Rap	As Above so Look out Below

```
recommender.get_recommendations('Venom', 2)
```

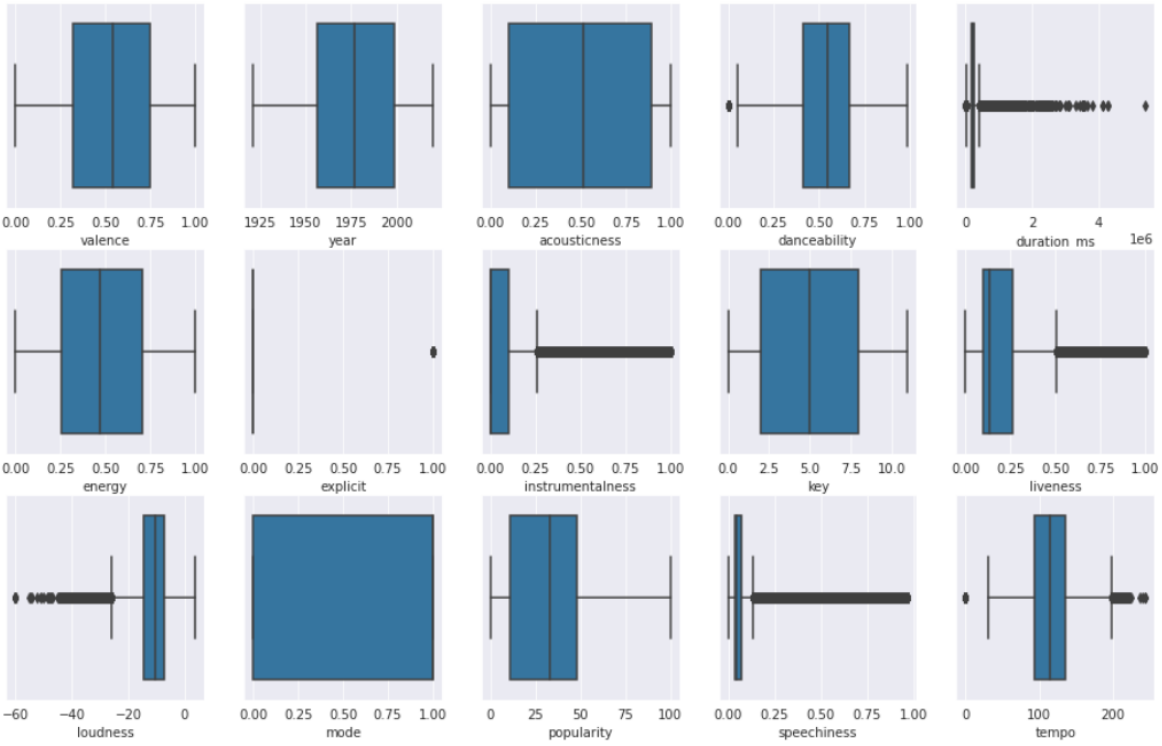
```
100%|██████████| 42299/42299 [00:02<00:00, 15176.67it/s]
```

	genre	song_name
8281	Underground Rap	Venom (feat. Shakewell)
11757	Trap Metal	Crash

SPOTIFY SONG POPULARITY PREDICTION:

Given the data about Spotify songs, We tried to predict the *popularity* of a given song. We used a *linear regression model* to make our predictions, but we focused on *outlier detection* and *removal*.

Outlier detection:
We started from creating a *plot* for the outlier detection.
Outlier detection is a technique to identify a group of data points that are different from other data points within a dataset.



Firstly we created a function for basically counting the outliers.
A Z-score is a numerical measurement that describes a value's relationship to the mean of a group of values.
Z-score is measured in terms of standard deviations from the mean. If a Z-score is 0, it indicates that the data point's score is identical to the mean score.
We wanted to get the z-score of a specified threshold and the z-score for each value in the data frame and then we compared them with the count of the outlier in each column.

```
def get_outlier_counts(df, threshold):
    df = df.copy()

    # Get the z-score for specified threshold
    threshold_z_score = stats.norm.ppf(threshold)

    # Get the z-scores for each value in df
    z_score_df = pd.DataFrame(np.abs(stats.zscore(df)), columns=df.columns)

    # Compare df z_scores to the threshold and return the count of outliers in each co
    return (z_score_df > threshold_z_score).sum(axis=0)
```

Then we created a function that had the goal to remove the outlier's examples after getting their indices.

```
def remove_outliers(df, threshold):
    df = df.copy()

    # Get the z-score for specified threshold
    threshold_z_score = stats.norm.ppf(threshold)

    # Get the z-scores for each value in df
    z_score_df = pd.DataFrame(np.abs(stats.zscore(df)), columns=df.columns)
    z_score_df = z_score_df > threshold_z_score

    # Get indices of the outliers
    outliers = z_score_df.sum(axis=1)
    outliers = outliers > 0
    outlier_indices = df.index[outliers]

    # Drop outlier examples
    df = df.drop(outlier_indices, axis=0).reset_index(drop=True)

    return df
```

We continued by doing some preprocessing which refers to the technique of preparing (cleaning and organizing) the raw data to make it suitable for building and training Machine Learning models.

```python
def preprocess_inputs(df, outliers=True, threshold=0.95):
    df = df.copy()

    # Remove outliers if specified
    if outliers == False:
        df = remove_outliers(df, threshold)

    # Split df into X and y
    y = df['track_popularity'].copy()
    X = df.drop('track_popularity', axis=1).copy()

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=1)

    # Scale X with a standard scaler
    scaler = StandardScaler()
    scaler.fit(X_train)

    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)

    return X_train, X_test, y_train, y_test
```

Then we created two *test data*: one with outliers and one without outliers.
In the last steps of this part of the work we built up the two models that gave us very similar results. This is because in our dataset we discovered that we had a small number of outliers.
Below there are the two models:

```python
# With outliers

l = list()
acc = list()
for i in range(1,50,1):

  outlier_model = KNeighborsClassifier(n_neighbors=i)
  outlier_model.fit(outlier_X_train, outlier_y_train)

  outlier_model_acc = outlier_model.score(outlier_X_test, outlier_y_test)
  l.append(outlier_model)
  acc.append(outlier_model_acc)
outlier_model= l[acc.index(max(acc))]
print("The Best amount of Neighbors is:  "+str(acc.index(max(acc))+1)+"    with a ACC of: "+str(max(acc)*1(
```

```python
# Without outliers
l = list()
acc = list()
for i in range(1,50,1):
  model = KNeighborsClassifier(n_neighbors=i)
  model.fit(X_train, y_train)

  model_acc = model.score(X_test, y_test)
  l.append(model)
  acc.append(model_acc)
model= l[acc.index(max(acc))]
print("The Best amount of Neighbors is:  "+str(acc.index(max(acc))+1)+"    with a ACC of: "+str(max(acc)*1(
```

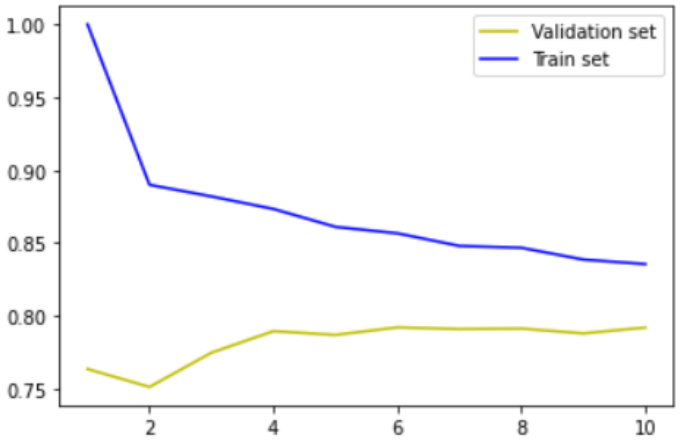
The results of the models with outliers and the ones without outliers showed us a fair percentage of *accuracy*, respectively around *49%* and *51%*.
So in the end, we can affirm that this song's popularity prediction works pretty well!

PREDICTING GENRE BASED ON SOUND FEATURES:

Our plan in this section was to predict the genre of a song based on the sound features. We used a supervised learning algorithm. Because of it's great accuracy we decided to make use of the KNN algorithm.

```
val_accuracies = []
train_accuracies = []
KNN_models = []
X_train, X_test, Y_train, Y_test = train_test_split(data, load['genre'], test_size=0.2)
X_train, X_val, Y_train, Y_val = train_test_split(data, load['genre'], test_size=0.2)
# We try neighbors from 1 to 11
for i in range(1,11):
    neigh = KNeighborsClassifier(n_neighbors=i)
    neigh.fit(X_train, Y_train)
    total = 0
    for actual,pred in zip(neigh.predict(X_val), Y_val):
        total += actual == pred
    val_accuracies.append(total/len(X_val))
    total = 0
    for actual,pred in zip(neigh.predict(X_train), Y_train):
        total += actual == pred
    train_accuracies.append(total/len(X_train))
    KNN_models.append(neigh)
```

We found the number of neighbors which offered us the best accuracy.



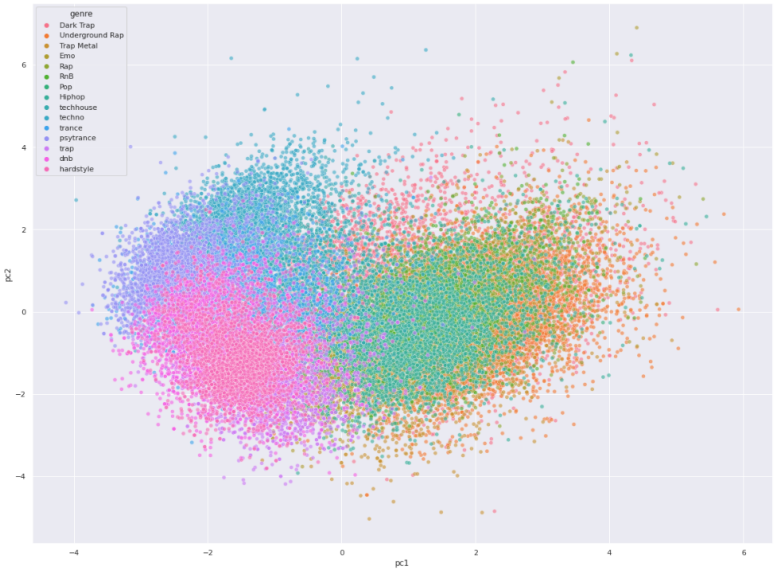

```
for i in range(10):
    select_model = KNN_models[i]
    total = 0
    for actual,pred in zip(select_model.predict(X_test), Y_test):
        total += actual == pred
    print('Model {} accuracy:'.format(i+1), total/len(X_test))
```

```
Model 1 accuracy: 0.9533152109679707
Model 2 accuracy: 0.8652641531733838
Model 3 accuracy: 0.8613639049757712
Model 4 accuracy: 0.8624276090296655
Model 5 accuracy: 0.8487176456683607
Model 6 accuracy: 0.8507268644368278
Model 7 accuracy: 0.8420990426663515
Model 8 accuracy: 0.843990072095497
Model 9 accuracy: 0.8359531970216286
Model 10 accuracy: 0.8329984635385889
```

After having analyzed the different amounts of neighbors we saw that *1* is the *best performing number*. When we increased this number the accuracy fell.

For the following part we decided to make use of a Principal Component Analysis (PCA). This allowed us to remove all the features that were correlated with each other. On top of that, we simplified the multidimensional features, in order to represent them in a graphical representation.

```
print(PCA(n_components=2).fit(data).explained_variance_ratio_)
load['pc1'] = pca[:, 0]
load['pc2'] = pca[:, 1]
sns.set(rc={'figure.figsize': (20,15)})
sns.scatterplot(data=load, x='pc1', y='pc2', hue='genre', alpha=0.6)
```

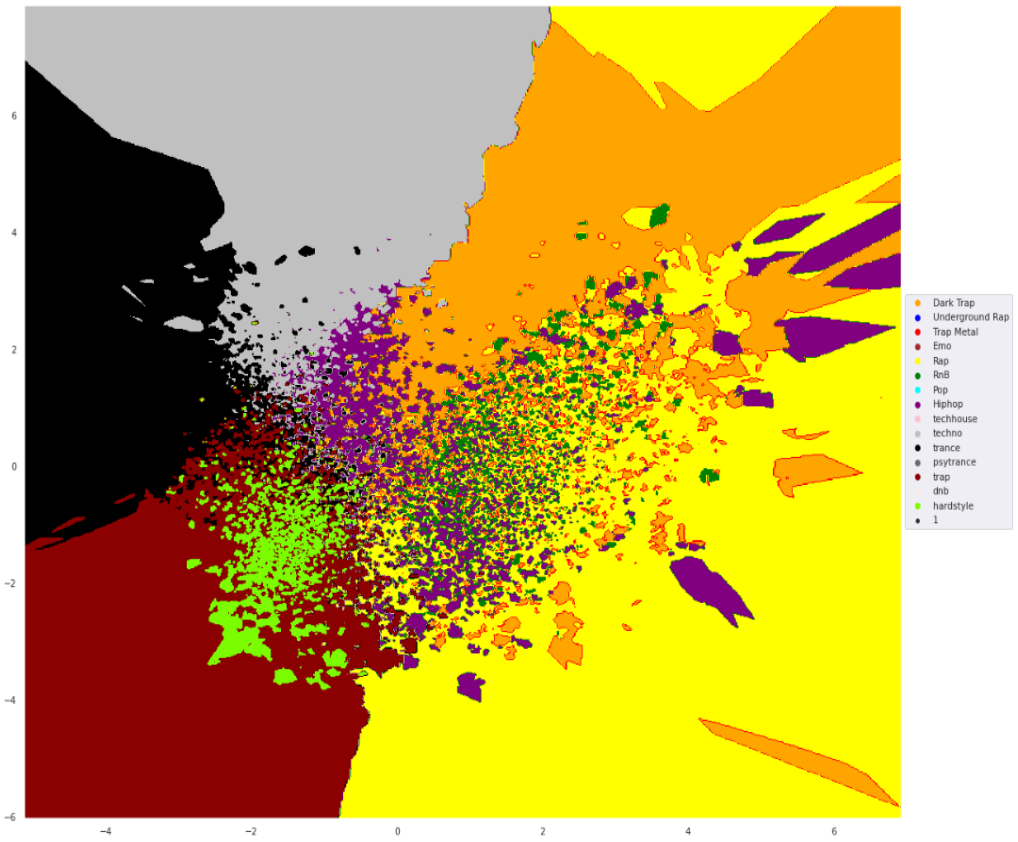


As we can see from the graph, on one hand, genres like *hardstyle* or *rnb, psytrance* or *Trance* have a positive variance (similar values), on the other hand, genres like *Dark Trap* or *Underground Trap* have negative variance (different values).

```
h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['orange', 'blue', 'red', 'brown', 'yellow', 'green', 'aqua'
cmap_bold = ['orange', 'blue', 'red', 'brown', 'yellow', 'green', 'aqua', 'purple', 'pin

# We train the model under pca to plot 2d boundaries
clf = KNeighborsClassifier(n_neighbors=5).fit(pca, load['genre'])

codemap = {}
for i,genre_name in enumerate(load['genre']):
    codemap[genre_name] = i

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = pca[:, 0].min() - 1, pca[:, 0].max() + 1
y_min, y_max = pca[:, 1].min() - 1, pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = np.array([codemap[genre_name] for genre_name in Z])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(figsize=(20,16))
plt.contourf(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
pts = sns.scatterplot(x=pca[:, 0], y=pca[:, 1], hue=load['genre'],
                palette=cmap_bold, alpha=0, edgecolor="black", size=1)
pts.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.show()
```

By training the KNN classifier with the data we obtained from the PCA algorithm, we were able to predict the genre of a song with an accuracy of *95%*

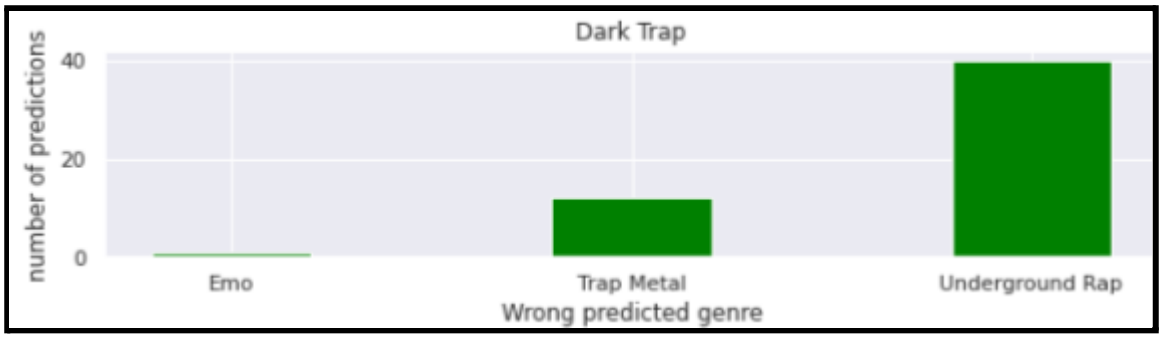
This graph shows us a graphical representation of our previously trained KNN algorithm. When the features of a new song are in the field of a genre, it's gonna be classified as this genre.
Also interesting to see is that genres which look similar are located close to each other. For example all Electronic music genres, such as Techno and Trance are in the top right corner.




```
Predicted:  Rap              Real:  Rap
Predicted:  psytrance              Real:  psytrance
Predicted:  psytrance              Real:  psytrance
Predicted:  trance            Real:  trance
Predicted:  Hiphop            Real:  Hiphop
Predicted:  Emo              Real:  Emo
Predicted:  Dark Trap              Real:  Dark Trap
Predicted:  Trap Metal              Real:  Trap Metal
Predicted:  RnB              Real:  RnB
Predicted:  dnb              Real:  dnb
Predicted:  trap              Real:  trap
Predicted:  techno            Real:  techno
Predicted:  hardstyle              Real:  hardstyle
Predicted:  trap              Real:  trap
Predicted:  dnb              Real:  dnb
Predicted:  Underground Rap              Real:  Underground Rap
Predicted:  Underground Rap              Real:  Underground Rap
Predicted:  techhouse              Real:  trance
Predicted:  hardstyle              Real:  hardstyle
Predicted:  techhouse              Real:  techhouse
```

This is an example of how we used the trained model to predict based on sound features the genre. As we can see most of the cases are right, but what happened with the wrong predicted fractions?

We created a specific analysis for the cases we predicted wrong and had a look at the outcome.



1. Here we took into consideration the songs which were Dark Trap, but got classified wrong. As we can see Underground Rap and Trap metal make the biggest proportion of this.

2. Then we had a look at what the wrong predictions for tech house were. We noticed that other electronic genres such as Techno and Trance are represented in this graph. The same happened with Hip Hop.

USING SVM TO PREDICT DIFFERENT SOUND MODES:

What does the mode of a Song mean ?

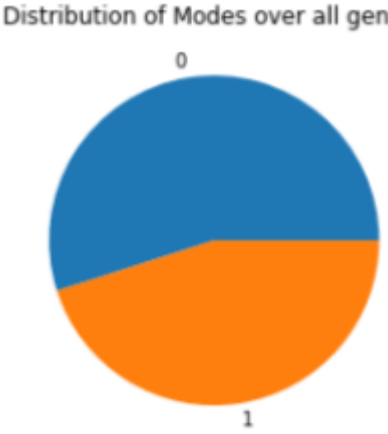
Mode, in music, is any of several ways of ordering the notes of a scale according to the intervals they form with the tonic, thus providing a theoretical framework for the melody. A mode is the vocabulary of a melody; it specifies which notes can be used and indicates which have particular importance.

Why we used SVMs?
Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression, and outlier detection.
We decided to choose SVMs because of the following reasons:

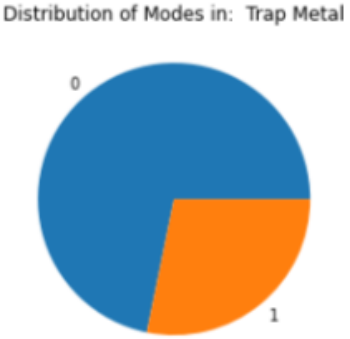
- Effective in high dimensional spaces: The mode of a song depends on a bunch of different aspects.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

First We understood if the Mode was depended at all on the features our Dataset was offering us:

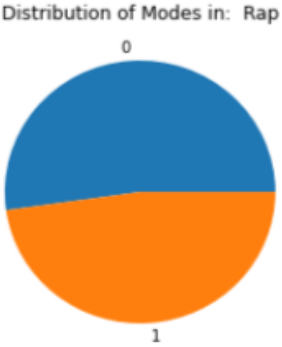
```
x =load["mode"].value_counts()
plt.pie(x,labels=[0,1])
plt.title("Distribution of Modes over all genres")
plt.figure(figsize=(20,10))
```



Indeed we can see that Mode 1 is slightly more represented in our data set. More precisely 56% of all data are Mode 1.



Also between different genres there were differences in the distribution of Modes.



Then, we setted up our support *Vector Machine algorithm*. After playing around the kernel and its degree we reached a maximum accuracy of *77%.*

```
y = load["mode"]
x = data.drop(["mode"],axis = 1)


x_train, x_test, y_train, y_test = sklearn.model_selection.train_test_split(x, y, test_size=0.2)


clf = svm.SVC(kernel="poly",degree=5)

clf.fit(x_train, y_train)

y_pred = clf.predict(x_test)

acc = metrics.accuracy_score(y_test, y_pred)

print(acc)
```

```
  76.819234712
```

LYRICS BASED GENRE PREDICTION:

In this section, we tried to analyze if we could distinguish between different genres by focusing on the *lyrics* of the Song.
As well we tried to compare the *accuracy* of genre predictions based on lyrics and based on sound features.

Data preparation:
Since our original Dataset about Spotify did not include any lyrics, we used an additional Dataset, called lsc.

```
lcs.head()
```

	Unnamed: 0	artist	song	text
0	0	Pearl Jam	Not For You	Restless soul enjoy your youth \nLike Muhamma...
1	1	Stevie Ray Vaughan	Crossfire	Day by day,night after night, \nBlinded by th...
2	2	Nine Inch Nails	Metal	We're in the building where they make us grow ...
3	3	Savage Garden	You Can Still Be Free	Cool breeze and autumn leaves \nSlow motion d...
4	4	Billie Holiday	Love For Sale	Love for sale. \nAppetizing, young love for s...

Because in later parts of this section we wanted to compare the *accuracy* of a lyric-based prediction with a song feature-based prediction we *merged* both datasets.

Since there were not all songs included in both Datasets, the total number of rows was drastically reduced.

This might reduce the accuracy of the predictions.

Some genres were not represented at all anymore. This is because electronic music such as techno, trance, or Techhouse normally does not have lyrics.

```
print(len(lcs))
lcs = pd.merge(lcs, load, on="song_name")
print(len(lcs))
```

```
6739
1774
```

Create sub-Dataframes:
First of all, we decided to just analyze 2 different genres at the same time.
This comes along with several advantages:

1. *Distinguish between genre accuracy:*
   Since we already dealt with a limited number of Data. Focusing just on two genres might have increased our accuracy. With each algorithm, we also identified which genres we performed well in and which we failed to perform in.

2. *Diversity:*
   We performed multiple analyses, taking into account that we used just two different labels, always on small Datasets which are differently structured. This led us to a different result for each label pair we used. Results will be compared at a later point of the work. Below there are several examples for different shaped sub Datasets while using label pairs:

```
Total Data:  503  Data in Genre:  Dark Trap:  431  Data in Genre:  Pop:  72
Total Data:  686  Data in Genre:  Dark Trap:  431  Data in Genre:  RnB:  255
Total Data:  434  Data in Genre:  Underground Rap:  317  Data in Genre:  Rap:  117
```

3. *More options for algorithms*:
   By just using two different labels, we used the SVM model. As well as some graphs to make everything more understandable.

Function disti():

Let's introduce the Function **disti(g1,g2,df,cloud=False,head=False).**
It serves multiple functionalities and applies different Machine learning algorithms.
It takes 3 different positional arguments which are g1 and g2, the two genres that will be used as labels later on. As well as df, the Dataframe we are going to use.
Last but not least, there are also two optional parameters to create graphs.

Below, there is a small explanation of the different functionalities of the function.

Bag of Words by using Naive_bayes:

We decided to use the *Naive_bayes algorithm* to perform the lyric-based genre prediction. This, mainly because it didn't require as much training data, which was lacking and also because of its relatively fast speed, which became handy when performing the algorithm multiple times.

```
[236]
      def disti(g1,g2,df,cloud=False,head=False):
        df = df.loc[df['genre'].isin([g1, g2])]
        print("Total Data:  "+str(len(df))+"  Data in Genre:  "+str(g1)+":  "+str(len(df.loc[df["genre"]==g1])))
        #based on lyriks
        X_train,X_test,Y_train,Y_test = train_test_split(df["text"],df["genre"],random_state=1)
        count_vector = CountVectorizer()
        le = preprocessing.LabelEncoder()
        naive_bayes = MultinomialNB(fit_prior=True,alpha=0.07)
        training_data = count_vector.fit_transform(X_train)
        testing_data = count_vector.transform(X_test)
        naive_bayes.fit(training_data,Y_train)
        pred = naive_bayes.predict(testing_data)
        acc_NB=accuracy_score(Y_test,pred)
        print("Lyrik based naiveBayes ACC:  "+str(acc_NB))
```

By performing the *MultinomialNB algorithm* on all possible different label pairs we reached an average accuracy of *68%*. As mentioned previously, this number is the sum of all the single acc, divided by the total amount of computations.

KNN Classification:
Since we wanted to compare later on the results of different algorithms, the function disti() also included a KNN classifier. To optimize it, we tried several amounts of *neighbors* and used the best model at the end.
On average the method has an accuracy of *82%*. Which is lower than the acc. of the more advanced classification we used in the part "Predicting genre based on sound features".

```
#KNN

  X = df.drop(["genre","text"],axis=1)
  Y = df["genre"]

  x_train, x_test, y_train, y_test = sklearn.model_selection.train_test_split(X, Y, test_size=0.1)
  acc_KN = 0
  b_m = 0
  b_n = 0
  for i in range(1,10,1):

    model = KNeighborsClassifier(n_neighbors=i)
    model.fit(x_train,y_train)
    acc = model.score(x_test,y_test)
    if acc>acc_KN:
      acc_KN = acc
      b_m = model
      b_n= i
```

SVM classification:

Last but not least the function includes an *SVM classification*, which is possible because we were just analyzing two labels at the same time.
On average the accuracy score is *72%* for this method. As we can see, we used *poly* as a kernel with a degree of 5. This is based on trying around the algorithm with different parameters and comparing manually the outcomes.

```
#SVM based on features
Y = df["genre"]
X = df.drop(["genre","text"],axis = 1)


x_train, x_test, y_train, y_test = sklearn.model_selection.train_test_split(X, Y, test_size=0.1)

clf = svm.SVC(kernel="poly",degree=5)

clf.fit(x_train, y_train)

y_pred = clf.predict(x_test)

acc_SVM = metrics.accuracy_score(y_test, y_pred)

print("song feauture SVM based ACC: "+str(acc_SVM))
```

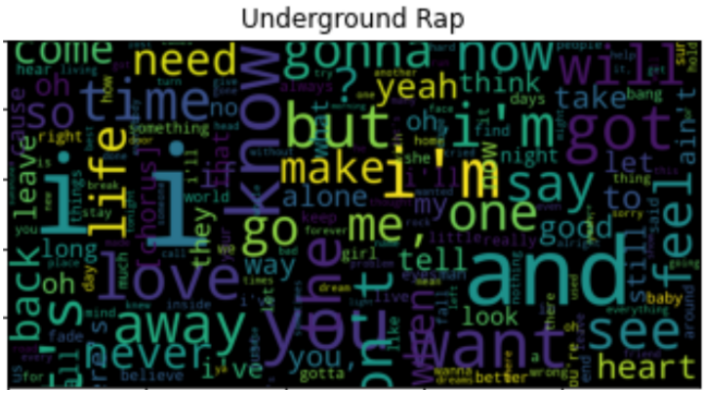
In this section, We had a look at Cases where different algorithms predicted in a non-average way.

Naive Bayes Overperform:

```
disti("Pop","Underground Rap",lcs,cloud=True,head=True)
```

```
Total Data:  389  Data in Genre:  Pop:  72  Data in Genre:  Underground Rap:  317
Lyrik based naiveBayes ACC:  0.8367346938775511
song feauture SVM based ACC: 0.717948717948718
song feauture KNN based ACC: 0.7461538461538462  Amount of Neighbors:  1
```

For example between the genres *Pop* and *Underground Rap*, the NaiveBayes algorithm was performing with a *higher accuracy* than on average. In this case it worked better than SVM and KNN.
This is because between these genres there is a quite big difference in the lyrics. Since both SVM and KNN perform under average performance, we may conclude that there is no big difference between sound features.



NaiveBayes Underperform:

```
disti("Pop","Trap Metal",lcs,cloud=True,head=True)
```

```
Total Data:  158  Data in Genre:  Pop:  72  Data in Genre:  Trap Metal:  86
Lyrik based naiveBayes ACC:  0.575
song feauture SVM based ACC: 0.8125
song feauture KNN based ACC: 0.875  Amount of Neighbors:  1
```

In the Case of *Pop* and *Trap Metal,* we saw that the NaiveBayes algorithm is *below* its average accuracy. This is because the word appearances were similar between these two genres. In contrast, this time the two sound feature Based algorithms were working quite well.

Sound feature accuracy underperforming:

```
disti("Dark Trap","Underground Rap",lcs,cloud=False,head=False)
```

```
Total Data:  748  Data in Genre:  Dark Trap:  431  Data in Genre:  Underground Rap:  317
Lyrik based naiveBayes ACC:  0.7454545454545454
song feauture SVM based ACC: 0.6266666666666667
song feauture KNN based ACC: 0.6933333333333334  Amount of Neighbors:  2
```

In this case, both sound feature-based algorithms were not performing well. This is because of the *high similarity* between the two genres.
Also, the KNN model we created in the part "predicting genre based on sound features" struggled with distinguishing between these two labels. As you can see in the graph below, in most cases when Trap Metal was detected wrong, the algorithm predicted it as Underground Rap.