Spotif-A.I: A spotify genre guesser

Table of Contents

1. Introduction

Spotify itself doesn’t provide genres for a particular track, only for artists.

The goal of our project was to train a model that can assign genres to a spotify track.

To try and achieve this, we used 2 data sources: the spotify api and last.fm api

The spotify api provides different kinds of features which can be used for machine learning. From last.fm it is possible to get the appropriate genres of a track (these are manually labeled by users).

We used the genres we got from last.fm as our target and the corresponding spotify tracks as our input.

2. Data Gathering

We used the following features from the spotify api:

- name: Name of the track
- duration: Track length in ms
- artist_genres: Artist genres

- acousticness: Confidence measure from 0.0 to 1.0 of whether the track is acoustic
- pitches: dominance of every pitch
- loudness: overall loudness in dB
- energy: perceptual measure of intensity and activity
- dancebility: How suitable a track is for dancing
- mode: Modality (Major or minor) of a track
- instrumentalness: Precision whether a track contains vocals
- key: Pitches using standard pitch class notation
- liveness: Presence of audience in the recording
- temp: estimated bpm
- time_signature: How many beats are in each bar
- valence: Musical positiveness

The first approach was to fetch songs via the search function of the api. Problem is the results are limited to a few popular tracks. The next approach was to use some large spotify playlists where all genres where included. The loading needs to be paged because only 50 tracks can be loaded at once.

```python
def playlist_tracks(self, playlist_id: str, offset: int = 0):
    completed = False
    limit = 50

    while not completed:
        print('loading playlist', playlist_id, 'offset:', offset)
        try:
            playlist_page: Union[PlaylistTrackPaging, dict] = self.spotify.playlist_items(
                playlist_id=playlist_id,
                offset=offset,
                limit=limit
            )
            if len(playlist_page.items) == 0:
                completed = True
            else:
                yield [item.track for item in playlist_page.items if isinstance(item.track, FullPlaylistTrack)]
                completed = False

            offset += limit
        except Exception as e:
            print('Error while fetching playlist tracks: ', e)
            completed = False
            sleep(3)
```

The genres itself get scraped from last.fm because the api provides inconsistent results. On multiple occasions the API would return no tags, even though the web-UI shows multiple tags on a track.

```python
def get_tags(self, artist: str, track: str) -> Set[str]:
    artist = urllib.parse.quote(artist)
    track = urllib.parse.quote(track)
    url = f"https://www.last.fm/music/{artist}/_/{track}/+tags"
    r = requests.get(url)
    tags_html = BeautifulSoup(r.content, features="html.parser")
    tag_links = tags_html.find_all("a", href=True)
    return {t.text for t in tag_links if t["href"].startswith("/tag/") and t.text != ""}
```

The data is then saved into a Mongo Atlas instance. There are multiple reasons why an online database is used:

- Multiple teammates can load data into the database
- The charting abilities of Mongo Atlas
- Not having to have to track the data in Git

3. Data preprocessing

Initially the pitches have different lengths depending on the track length and the size of the segments. We need to unify all pitches to the same lengths and reduce the frequency.

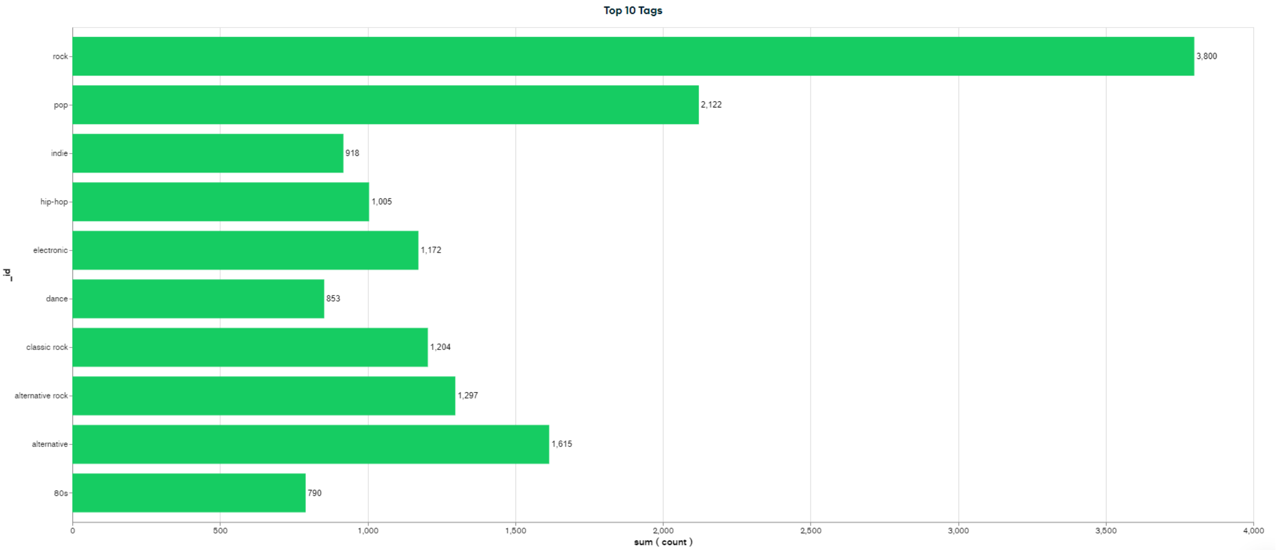
```
df["pitches"] = pitches
for sym in pitch_symbol:
    df.insert(len(df.columns), f"{sym}_max", [max_of_pitches(item, sym) for item in df["pitches"]])
    df.insert(len(df.columns), f"{sym}_min", [min_of_pitches(item, sym) for item in df["pitches"]])
temp_df = pd.DataFrame()

for index, row in enumerate(df["pitches"].values):
    song = pitches_to_dataframe(row)
    song["timestamp"] = song["timestamp"].apply(pd.to_timedelta, unit='s')
    resampled: pd.DataFrame = song.set_index("timestamp").resample(
        f"{song.iloc[-1]['timestamp'].total_seconds() * 10 // 1}ms").mean().interpolate()[:100]
    d: pd.DataFrame = pd.DataFrame()
    for col in resampled.columns:
        if col == "timestamp":
            continue
        d = pd.concat([d, pd.DataFrame({f"{col}_{row_idx}": [val] for row_idx, val in enumerate(resampled[col].values)},
                                       index=[df.index[index]])], axis=1)
    temp_df = pd.concat([temp_df, d])

df = pd.concat([df, temp_df], axis=1)
df.drop(columns=["pitches"], inplace=True)  # drop unprocessed pitches
```

Additionally, the number of tags needs to be reduced. Otherwise, we have way too many targets for our machine learning models. That’s why we only used the top 10 genres of our data and set all other tags to misc (as advised).

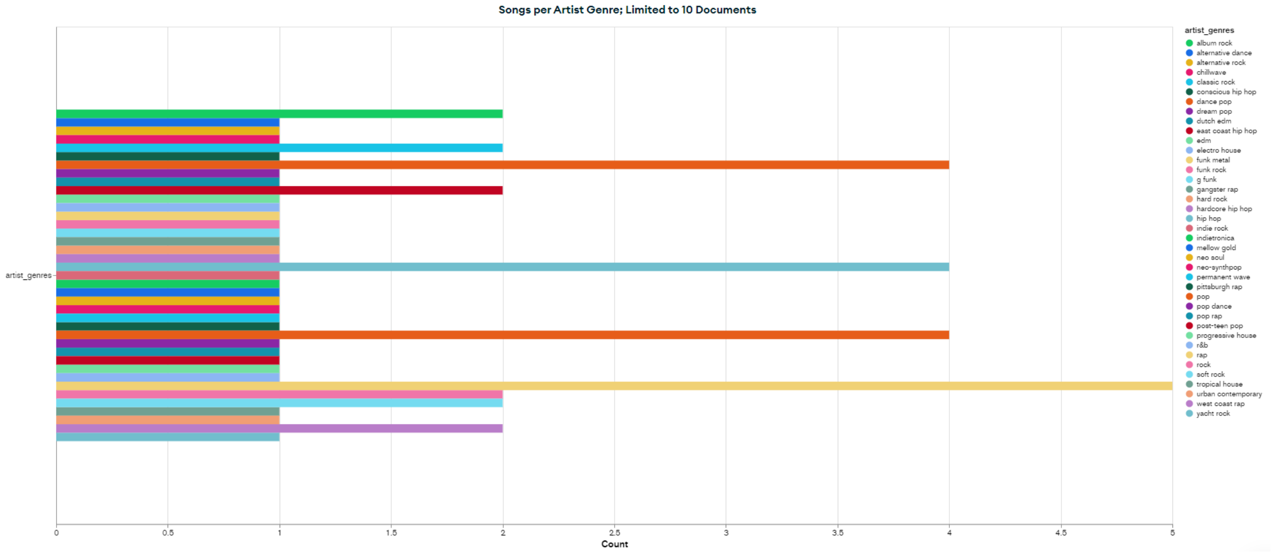
To reduce bias / try to balance our data we also decided to remove the rock genre.

4. Data Visualization

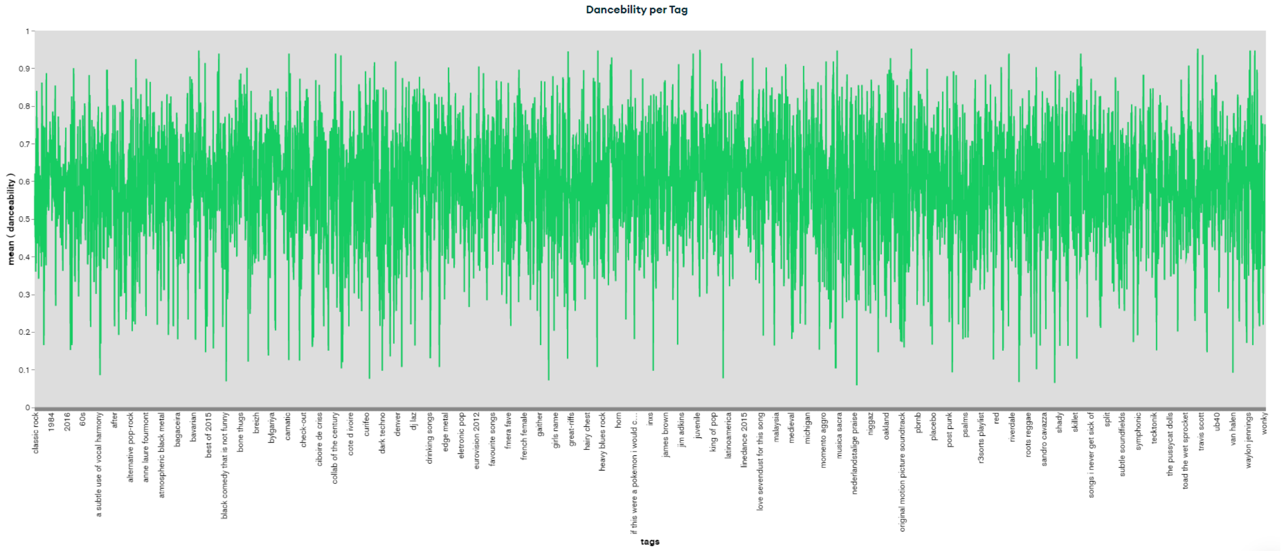
Here you can see the distribution of our top 10 genres.



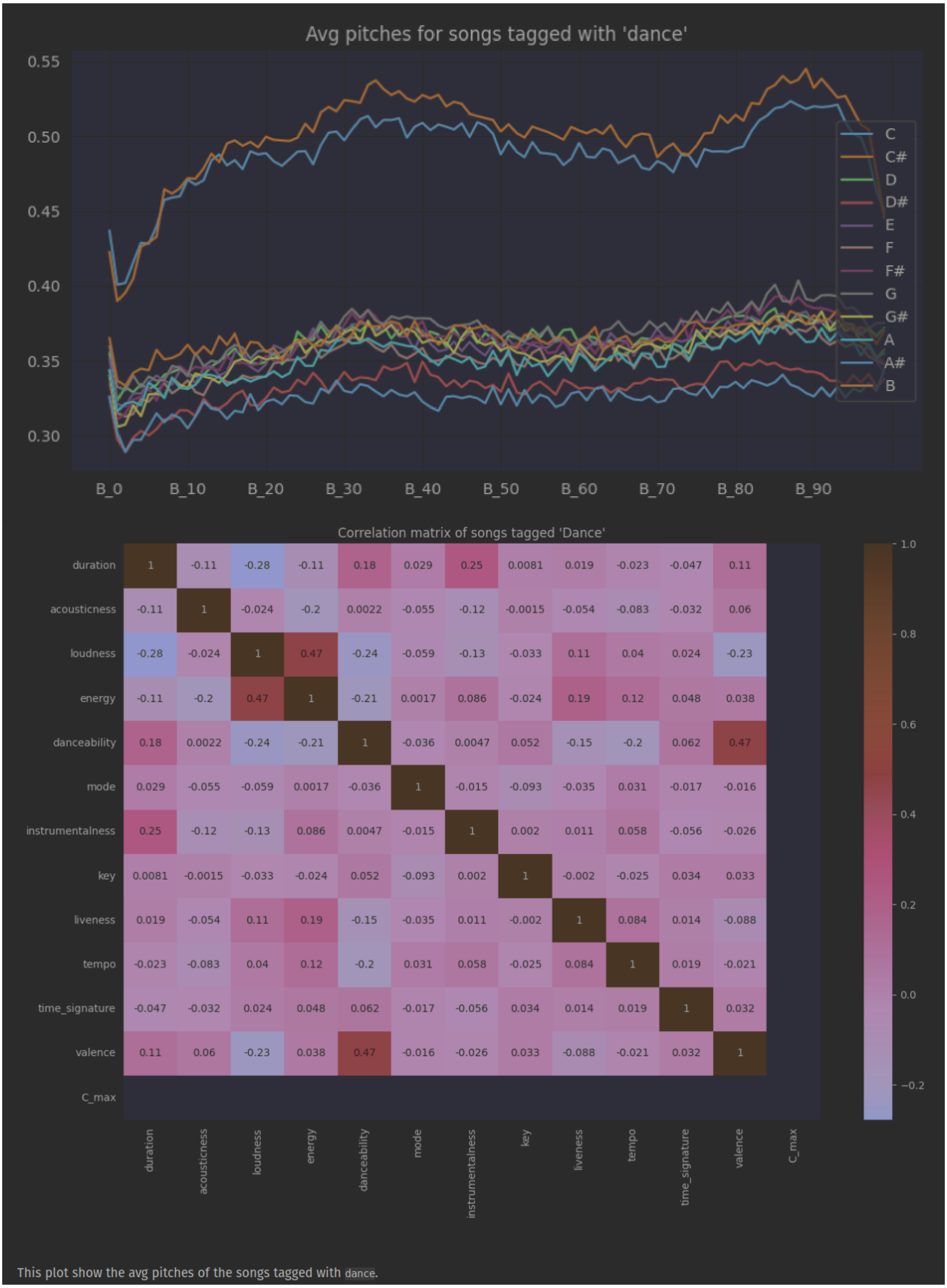
These are our different artist genres.



This is a chart of the different tags with the dancebility feature.



Average pitches and correlation matrix for dance.



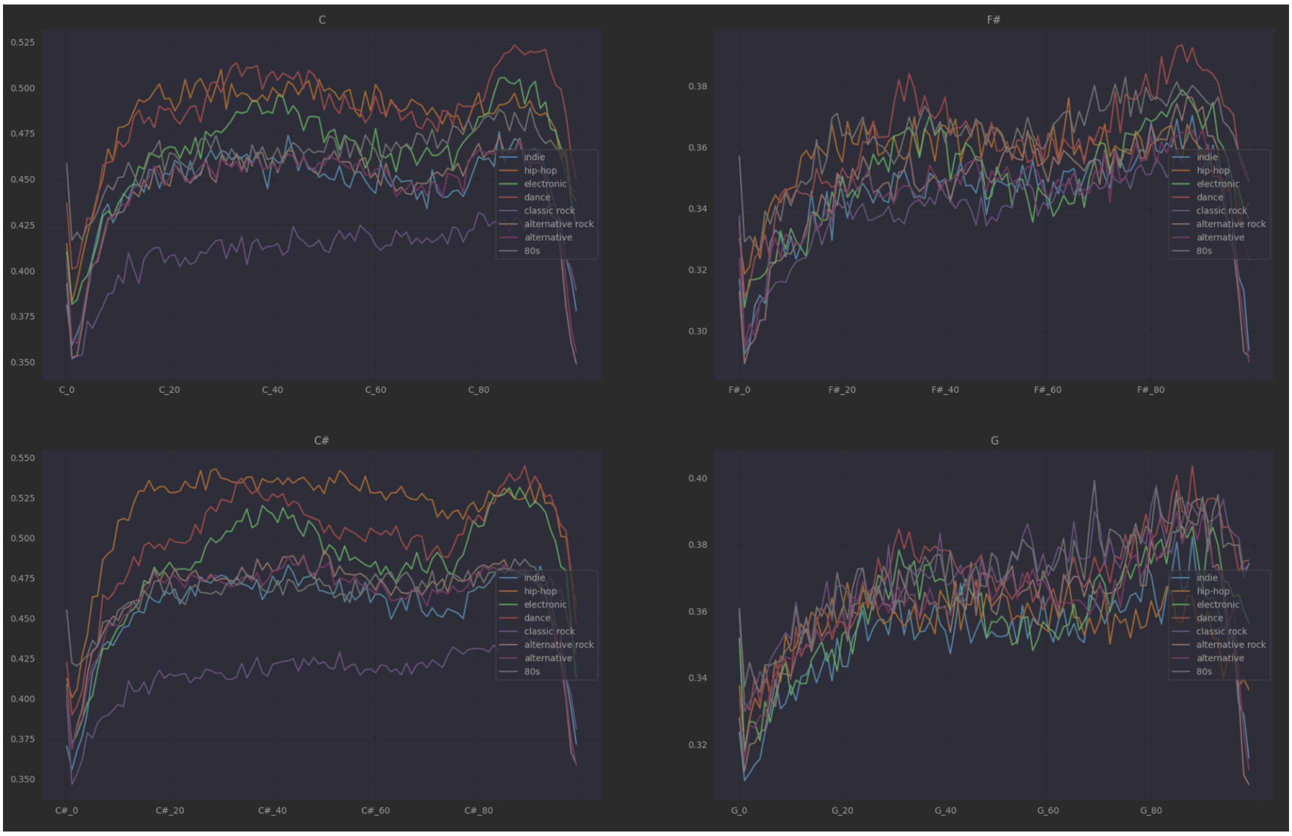
Avg pitches for songs tagged with 'dance'

Correlation matrix of songs tagged 'Dance'

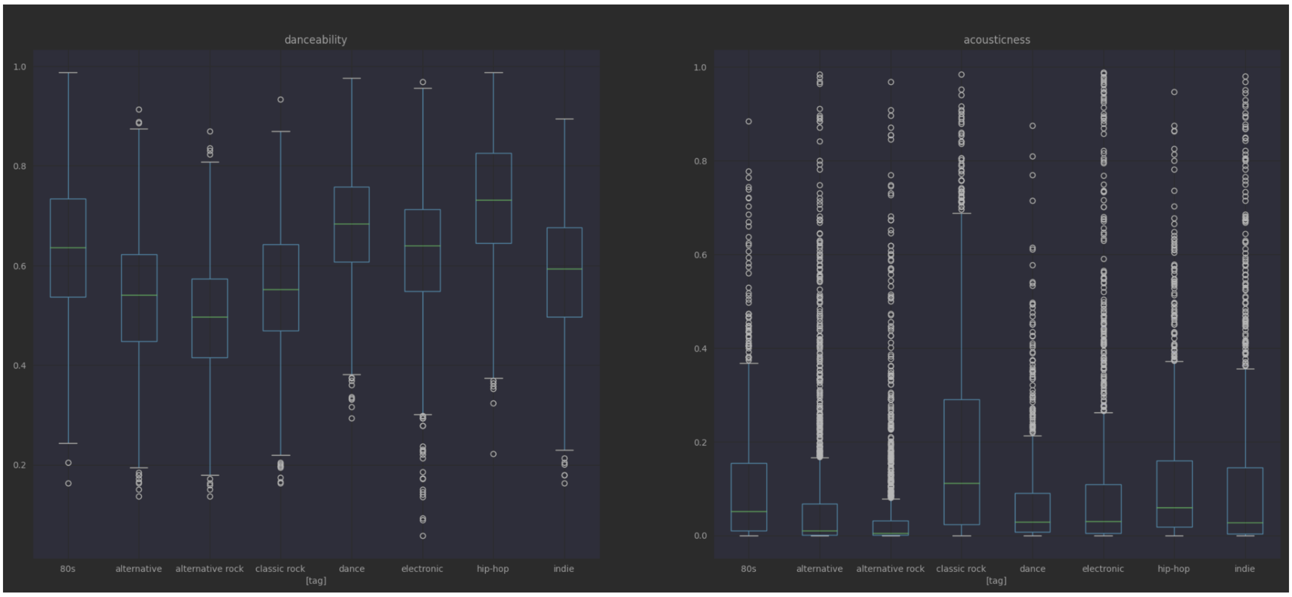
	duration	acousticness	loudness	energy	danceability	mode	instrumentalness	key	liveness	tempo	time_signature	valence	C_max
duration	1	-0.11	-0.28	-0.11	0.18	0.029	0.25	0.0081	0.019	-0.023	-0.047	0.11	
acousticness	-0.11	1	-0.024	-0.2	0.0022	-0.055	-0.12	-0.0015	-0.054	-0.083	-0.032	0.06	
loudness	-0.28	-0.024	1	0.47	-0.24	-0.059	-0.13	-0.033	0.11	0.04	0.024	-0.23	
energy	-0.11	-0.2	0.47	1	-0.21	0.0017	0.086	-0.024	0.19	0.12	0.048	0.038	
danceability	0.18	0.0022	-0.24	-0.21	1	-0.036	0.0047	0.052	-0.15	-0.2	0.062	0.47	
mode	0.029	-0.055	-0.059	0.0017	-0.036	1	-0.015	-0.093	-0.035	0.031	-0.017	-0.016	
instrumentalness	0.25	-0.12	-0.13	0.086	0.0047	-0.015	1	0.002	0.011	0.058	-0.056	-0.026	
key	0.0081	-0.0015	-0.033	-0.024	0.052	-0.093	0.002	1	-0.002	-0.025	0.034	0.033	
liveness	0.019	-0.054	0.11	0.19	-0.15	-0.035	0.011	-0.002	1	0.084	0.014	-0.088	
tempo	-0.023	-0.083	0.04	0.12	-0.2	0.031	0.058	-0.025	0.084	1	0.019	-0.021	
time_signature	-0.047	-0.032	0.024	0.048	0.062	-0.017	-0.056	0.034	0.014	0.019	1	0.032	
valence	0.11	0.06	-0.23	0.038	0.47	-0.016	-0.026	0.033	-0.088	-0.021	0.032	1	
C_max													

This plot show the avg pitches of the songs tagged with `dance`.

Average pitches for multiple genres. For example classic rock is very distinguishable when looking at the c pitch.



A boxplot of some spotify features:



5. Model Training

5.1. Problems in first iteration

In our first iteration we couldn’t manage to get any meaningful results. We tried the following models with `GridSearchCV` to also include a hyperparameter search:

- tree.ExtraTreeClassifier
- neighbors.KNeighborsClassifier

- neural_network.MLPClassifier
- neighbors.RadiusNeighborsClassifier
- linear_model.RidgeClassifier

But the best we could achieve was an `accuracy` of `4%` with `19% precision`.

5.1.1. Target imbalance & too many targets

The dataset that was gathered contains some massive imbalances. About one third of the songs contain `rock` as target and one fourth contain `pop`. This obviously leads to problematic models, that wrongly predict rock as target.

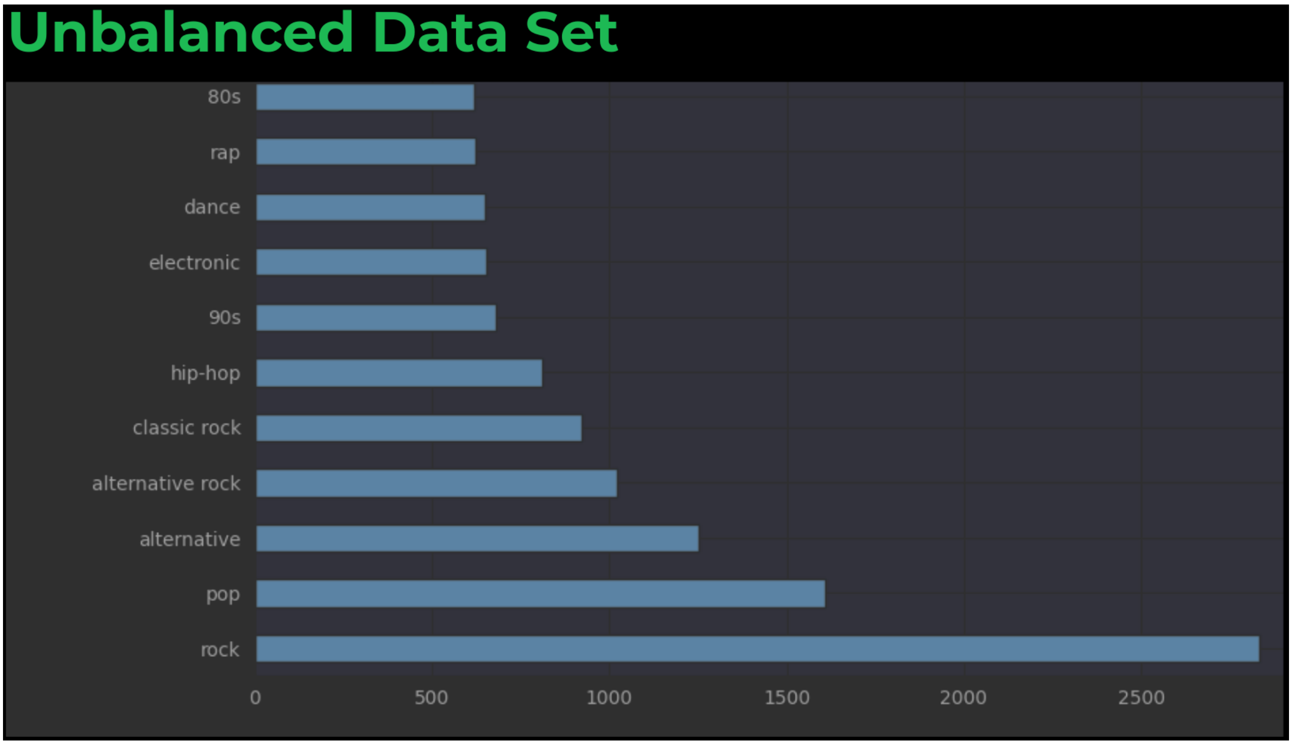


Figure 1. first iteration data imbalance

Although we had over `10.000` songs, we also had `4941` tags i.e. `4941` targets. Which essentially means we either have too many tags or not enough songs.

There are a few reasons why there is such a vast amount of target variables. Often, genres will occur in their hyphenated form and with a dash. For example, `hip-hop` and `hip hop`. Other sources for useless tags are that the tags will contain the artists of a song and custom tags, that are used to map tracks to a playlist for example. These churn tags lead to many targets that are either useless or redundant.

5.1.2. Solution

We decided to try and resolve this by restricting our targets to the `top 9`, without `rock`, appearing genres and wrap aggregate all other genres into a `misc` genre. This was done by preprocessing the tags in our `MongoDB` instance and providing a view to further work with them. We also added some more songs. `Rock` was removed as target, because it is overrepresented in our dataset. There are multiple tags, that are a subgenre of rock. Therefore, `rock` as standalone tag was removed, but other subgenres were kept.

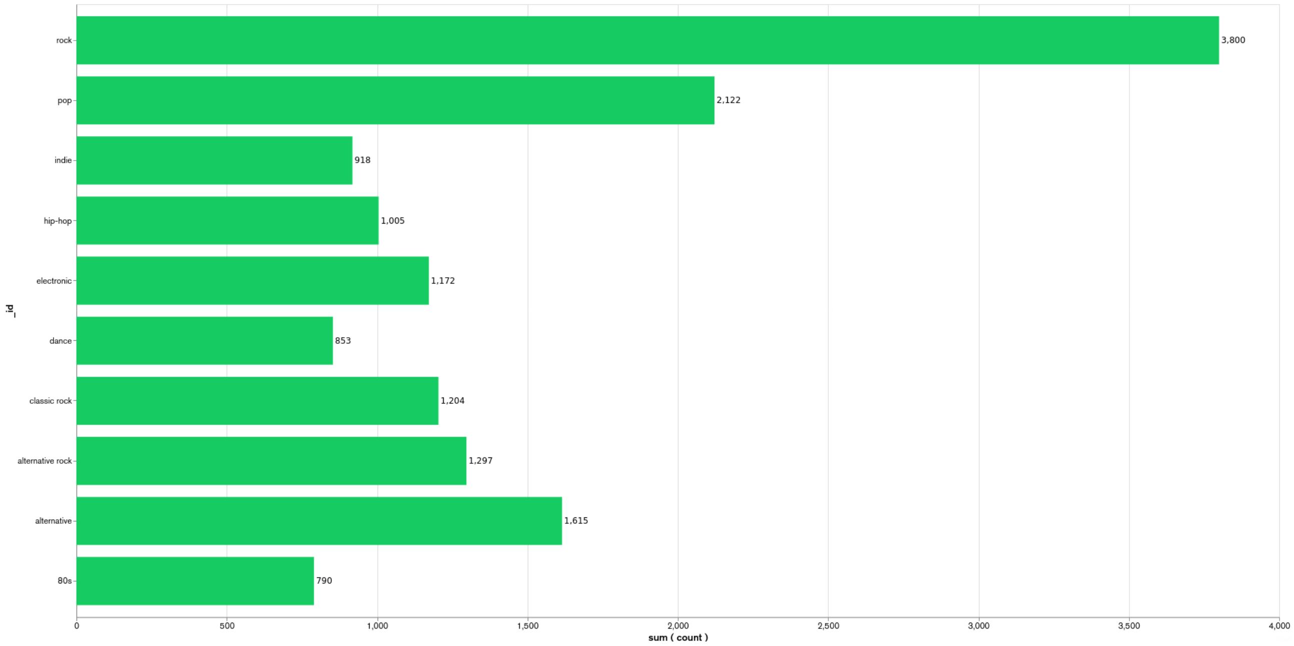
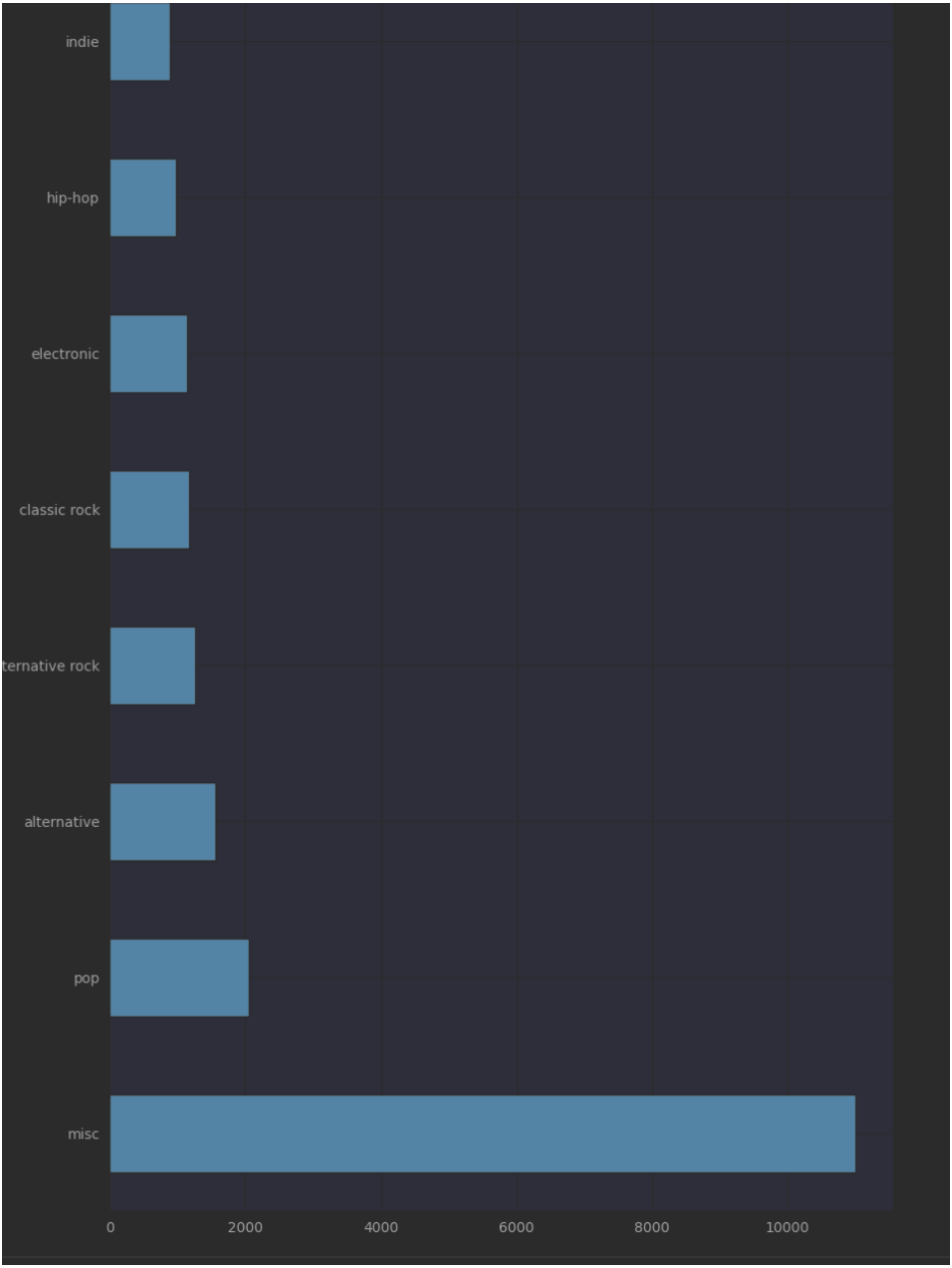


Figure 2. top 10 genres

Which led to the following distribution after classifying everything apart from the top 10 and rock as misc.



5.2. Training

For the training of our tuned tags, we decided to use the following models and compare there performance:

- neighbors.KNeighborsClassifier
- tree.DecisionTreeClassifier
- ensemble.ExtraTreesClassifier
- neural_network.MLPClassifier

5.2.1. Feature Preprocessing

Like always, before we start training, we create a *held-back test set* to use as a safety line and to perform the final evaluation of our model.

However, before we can create the held-back test set, we need to use the `MultilabelBinarizer` to transform our targets (genre tags) into numerical values so that the models can work with them.

For good measure we also use a `StandardScaler` to scale the input data for our models, as some models tend to perform better with this.

Listing 1. held-back test set

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.preprocessing import StandardScaler

# transform tags with MultiLabelBinarizer
mlb = MultiLabelBinarizer()
y = mlb.fit_transform([*df['tags']])

X = df.drop(columns=['tags', 'artist_names', 'name', "artist_genres"])

# scale input
scaler = StandardScaler()
X = scaler.fit_transform(X)

# create held-back test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=234634754)  # 70/30 split
```

In order to train a `DecisionTreeClassifer`, we also needed to calculate the `class_weights` of our targets:

Listing 2. class_weights calculation

```python
import numpy as np
from collections import Counter
from sklearn.utils import class_weight
flat_labels = [label for sublist in df['tags'] for label in sublist]
label_counts = Counter(flat_labels)
class_weights = class_weight.compute_class_weight('balanced', classes=np.unique(flat_labels), y=flat_labels)
class_weights_dict = dict(zip(np.unique(flat_labels), class_weights))

# Create a list of class weight dictionaries for each label
class_weights_list = []
for i in range(y.shape[1]):
    label_column = y[:,i]
    label_counts = Counter(label_column)
    class_weights = class_weight.compute_class_weight('balanced', classes=np.unique(label_column), y=label_column)
    class_weights_list.append(dict(zip(np.unique(label_column), class_weights)))
```

5.2.2. Feature Selection

Some models can’t effectively use 1241 features, so for them, we need to reduce the amount of features. Other models like MLP however can use all features, so we do not remove the other features.

5.2.3. KNeighborsClassifier

The KNN classifier was tested in following configurations, the best parameters are marked bold in the table:

Parameter	Values		
weights	`uniform`	**`distance`**	
algorithm	**`ball_tree`**	`kd_tree`	
leaf_size	**`1`**	`3`	
p	**`1`**	`2`	`4`
metric	`manhattan`	`cosine`	**`euclidean`**

Table 1. GridSearchCV for KNN

The best KNN classifier achieved following stats:

Metric	Value
Accuracy	34%
Precision	91%
Recall	66%
F1-Score	71%

Table 2. KNN Performance Evaluation

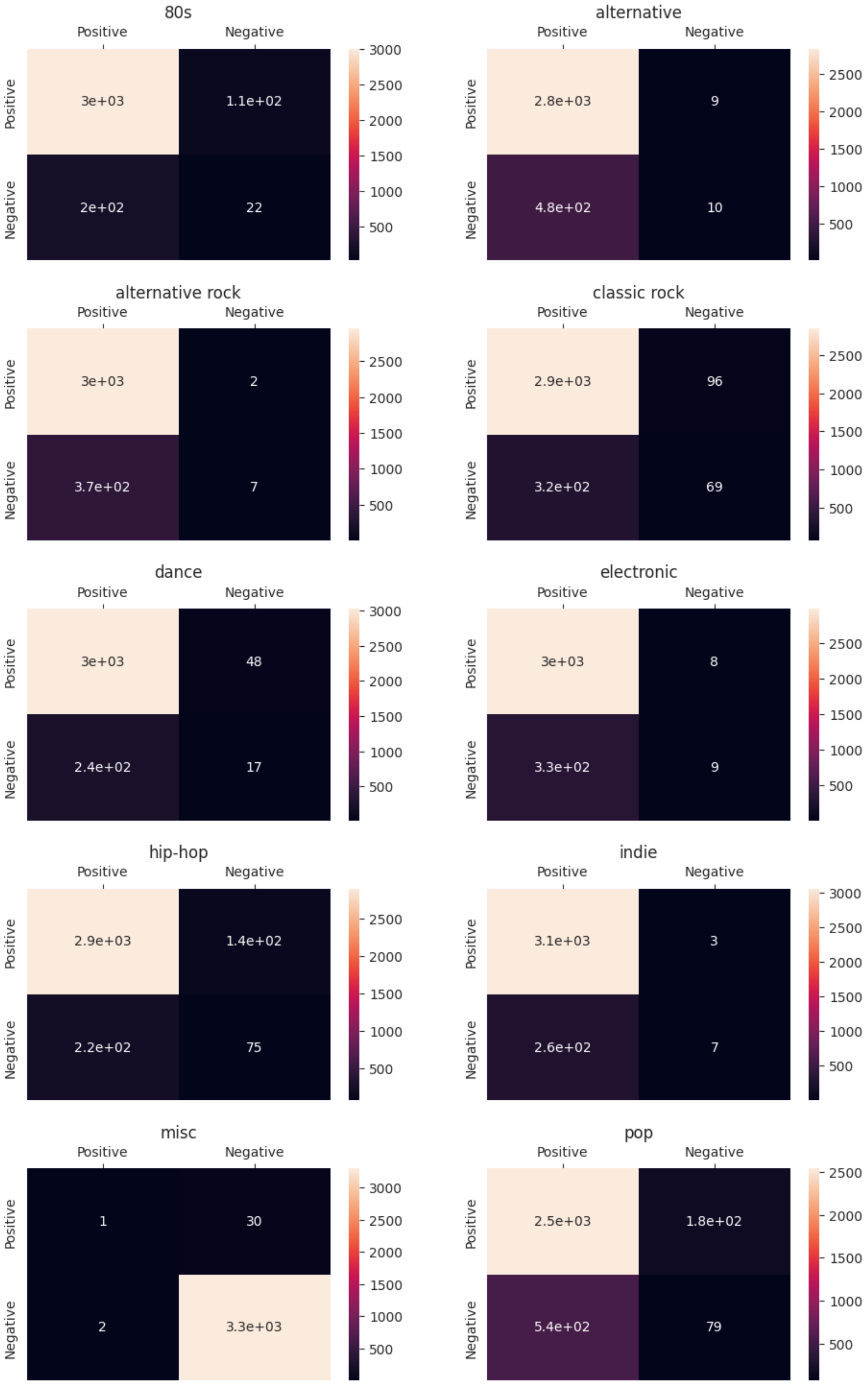


Figure 3. Confusion Matrix for each tag

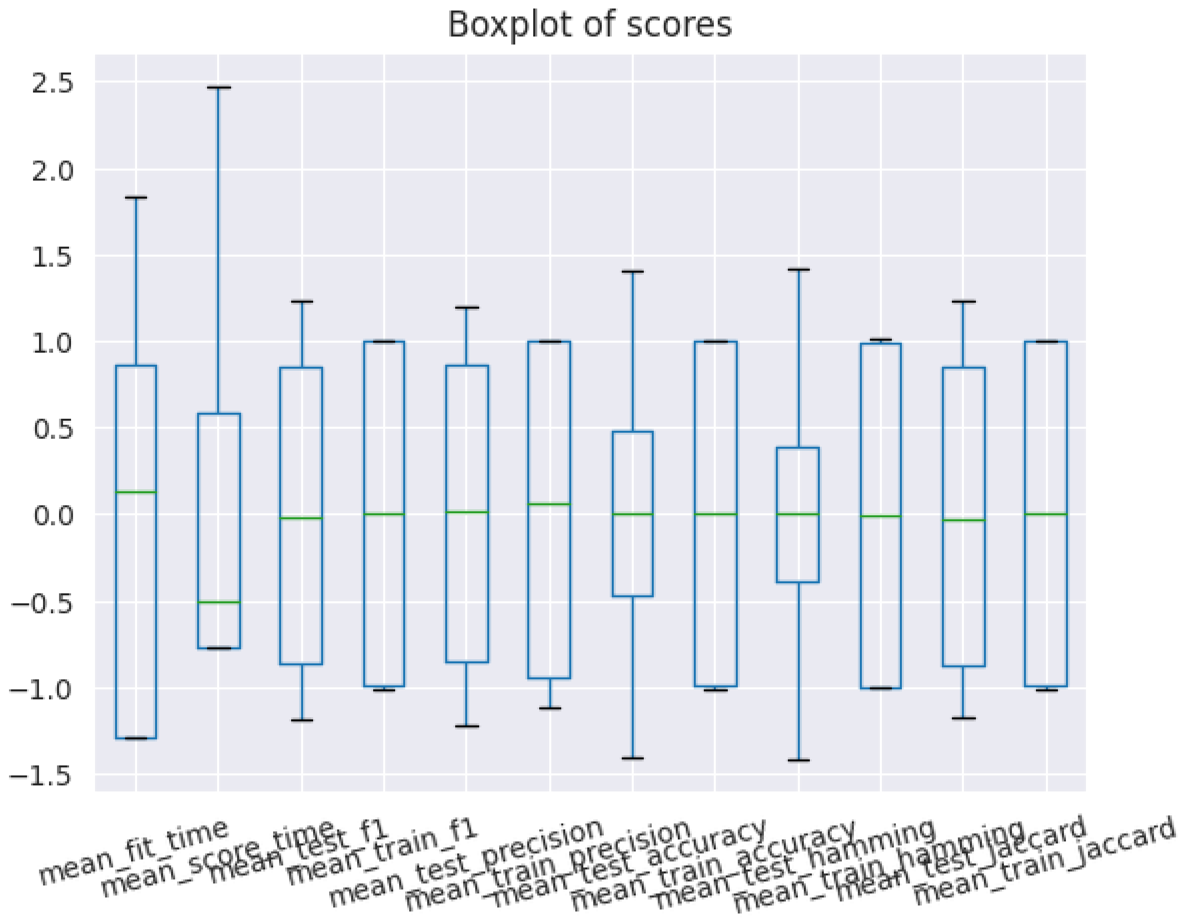


Figure 4. Performance Boxplot

5.2.4. DecisionTreeClassifier

The Decision Tree classifier was tested in following configurations, the best parameters are marked bold in the table:

Note: The class weights list are the ratios in which each class occurs

Parameter	Values				
criterion	**gini**				
splitter	**best**		random		
max_depth	1	**2**	8	16	48
min_samples_leaf	1	10		**30**	
min_weight_fraction_leaf	0.0	**0.0001**		0.0001^10	
max_features	None	**sqrt**		log2	
max_leaf_nodes	None	10	**100**	1000	
min_impurity_decrease	**0.0**	0.0001		0.0001^10	
class_weight	**class_weights_list**				

Parameter	Values		
ccp_alpha	`0.0`	`0.0001`	`0.0001^10`

Table 3. GridSearchCV for DecisionTreeClassifier

Metric	Value
Accuracy	07%
Precision	56%
Recall	67%
F1-score	58%

Table 4. Performance Report

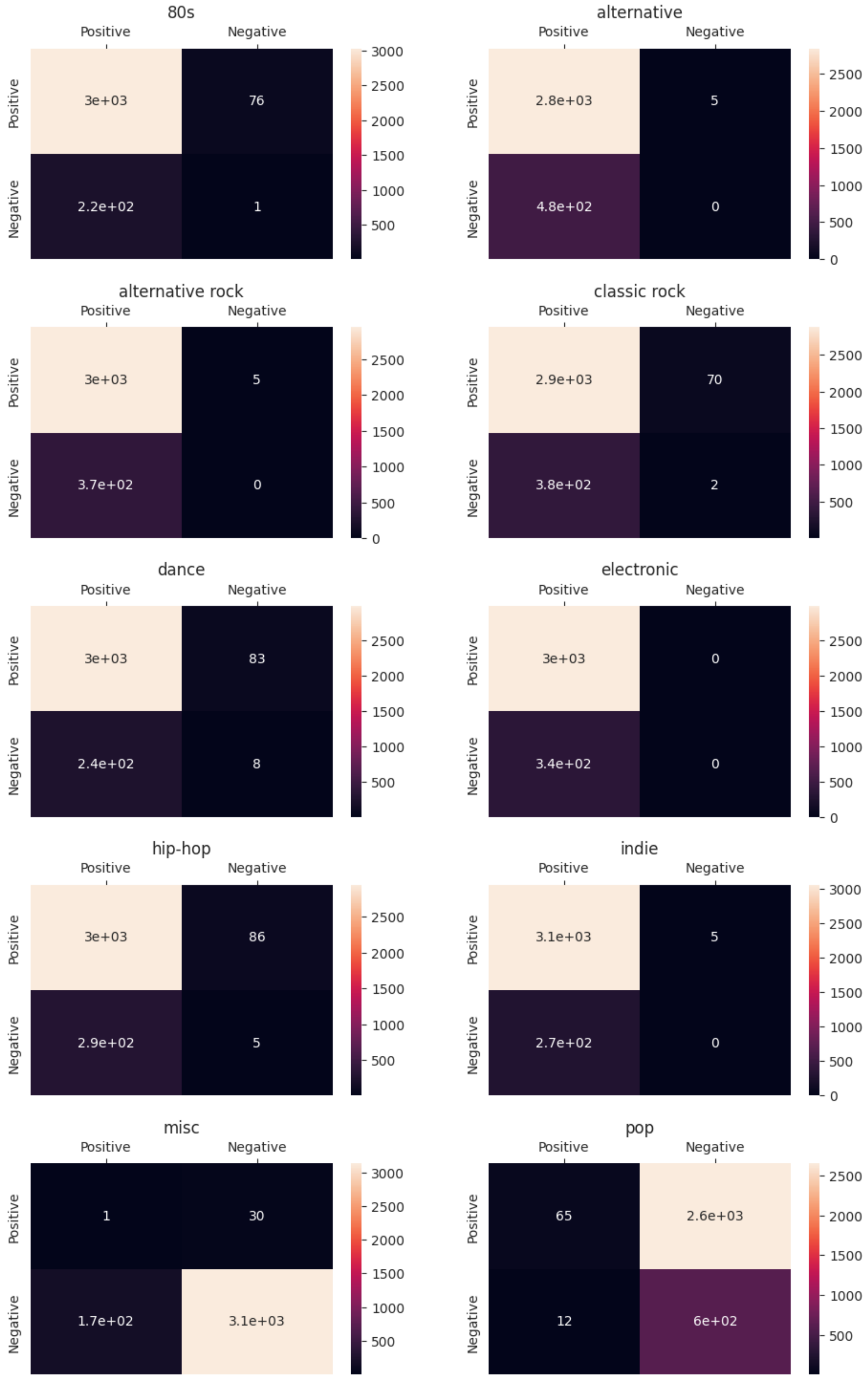


Figure 5. Confusion Matrix for each tag

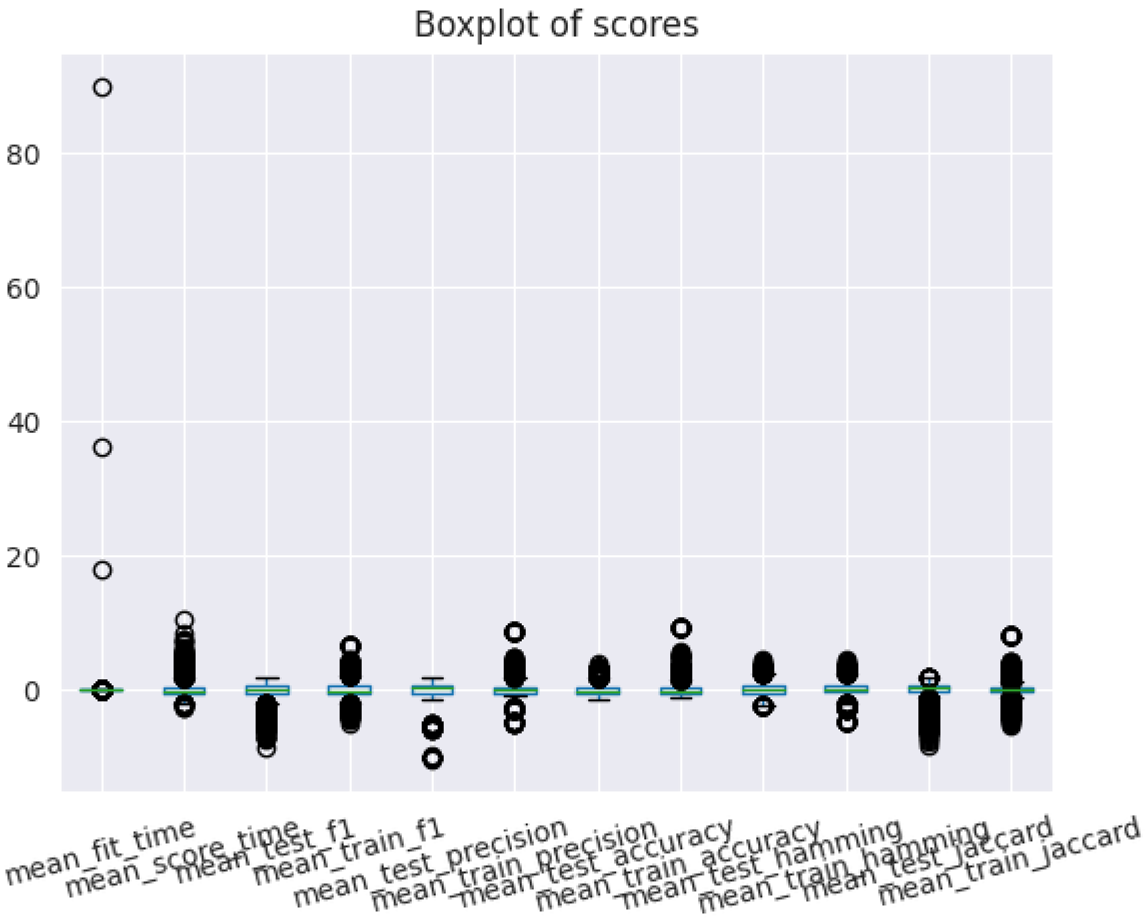


Figure 6. Performance Boxplot

5.2.5. ExtraTreesClassifier

Note: The class weights list are the ratios in which each class occurs

Parameter	Values				
n_estimators	100	150	200	300	400
criterion	gini	entropy		log_loss	
max_depth	12	24	36	48	64
min_samples_split	2	8		32	
max_features	sqrt	log2		None	
n_jobs	-2				
class_weight	class_weights_list		None		

Table 5. ExtraTreesClassifier Parameters

5.2.6. RandomForestClassifier

The random forest was trained with following parameters, the bold parameters are the ones that produce the strongest model:

Parameter	Values			
n_estimators	**100**	200	300	
max_depth	**None**	5	10	20
min_samples_split	**2**		3	
min_samples_leaf	**1**		2	
max_features	**sqrt**		log2	

Table 6. RandomForestClassifier Parameters

Metric	Value
Accuracy	38%
Precision	99%
Recall	64%
F1-score	74%

Table 7. Performance Report

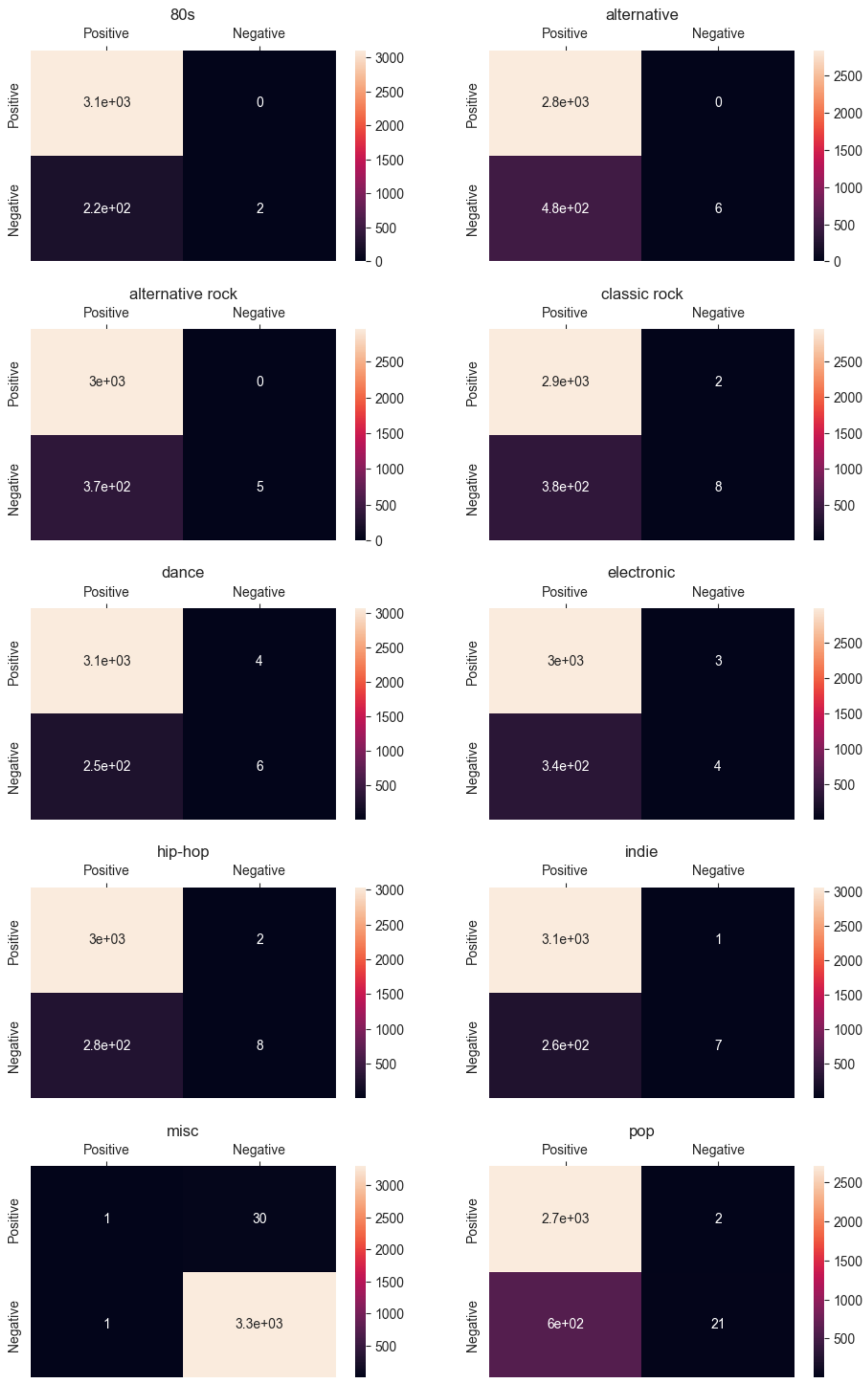


Figure 7. Confusion Matrix for each tag

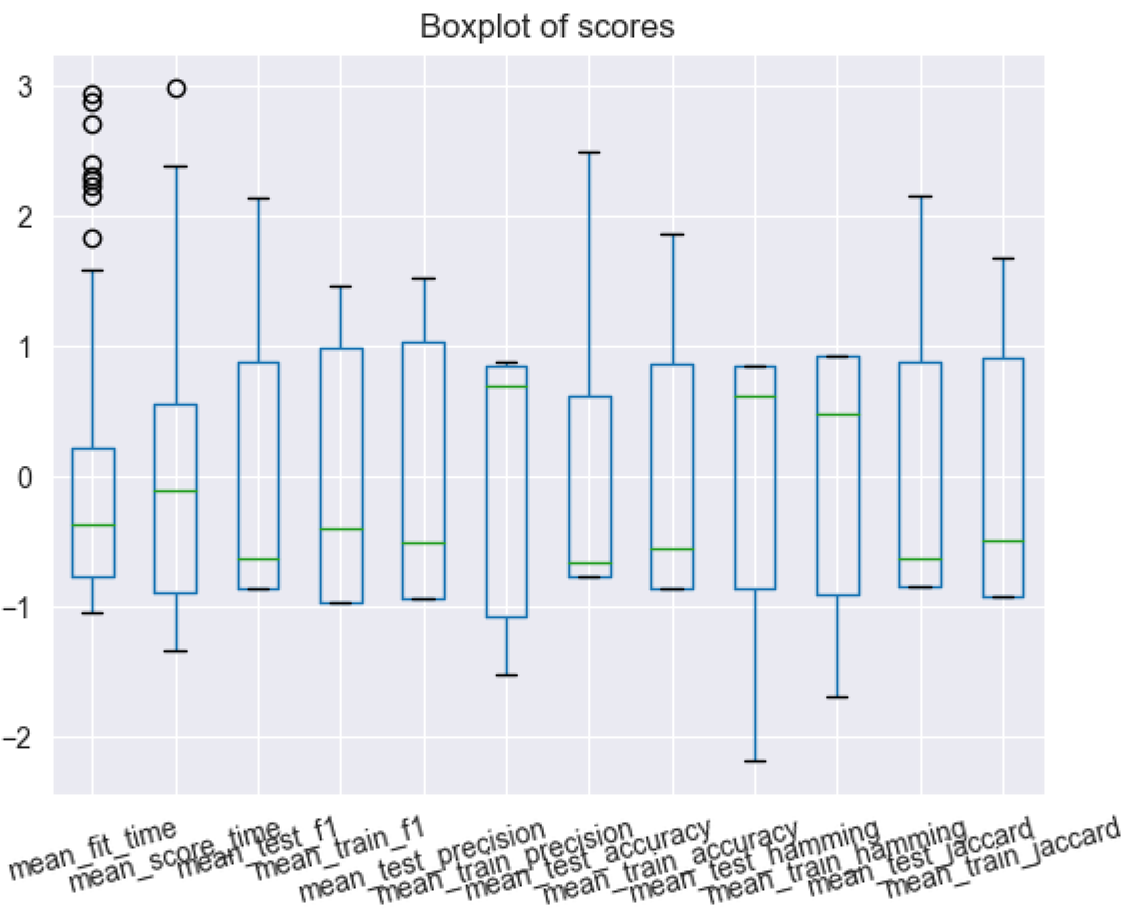


Figure 8. Performance Boxplot

5.2.7. MultilayerPerceptronClassifier

The MLP classifier was tested in following configurations, the best parameters are marked bold in the table:

<table>
<tr><th>Parameter</th><th colspan="6">Values</th></tr>
<tr><td><b>hidden_layer_sizes</b></td><td colspan="2"><b>(150,150,150)</b></td><td colspan="2">(100,100,100)</td><td colspan="2">(150,200,150)</td></tr>
<tr><td><b>activation</b></td><td colspan="3"><b>relu</b></td><td colspan="3">tanh</td></tr>
<tr><td><b>solver</b></td><td colspan="3"><b>adam</b></td><td colspan="3">lbfgs</td></tr>
<tr><td><b>learning_rate_init</b></td><td colspan="2">0.01</td><td colspan="2"><b>0.001</b></td><td colspan="2">0.0001</td></tr>
</table>

Table 8. GridSearchCV for MLPClassifier

Metric	Value
Accuracy	29%
Precision	81%
Recall:	71%
F1-score	70%

Table 9. Performance Report

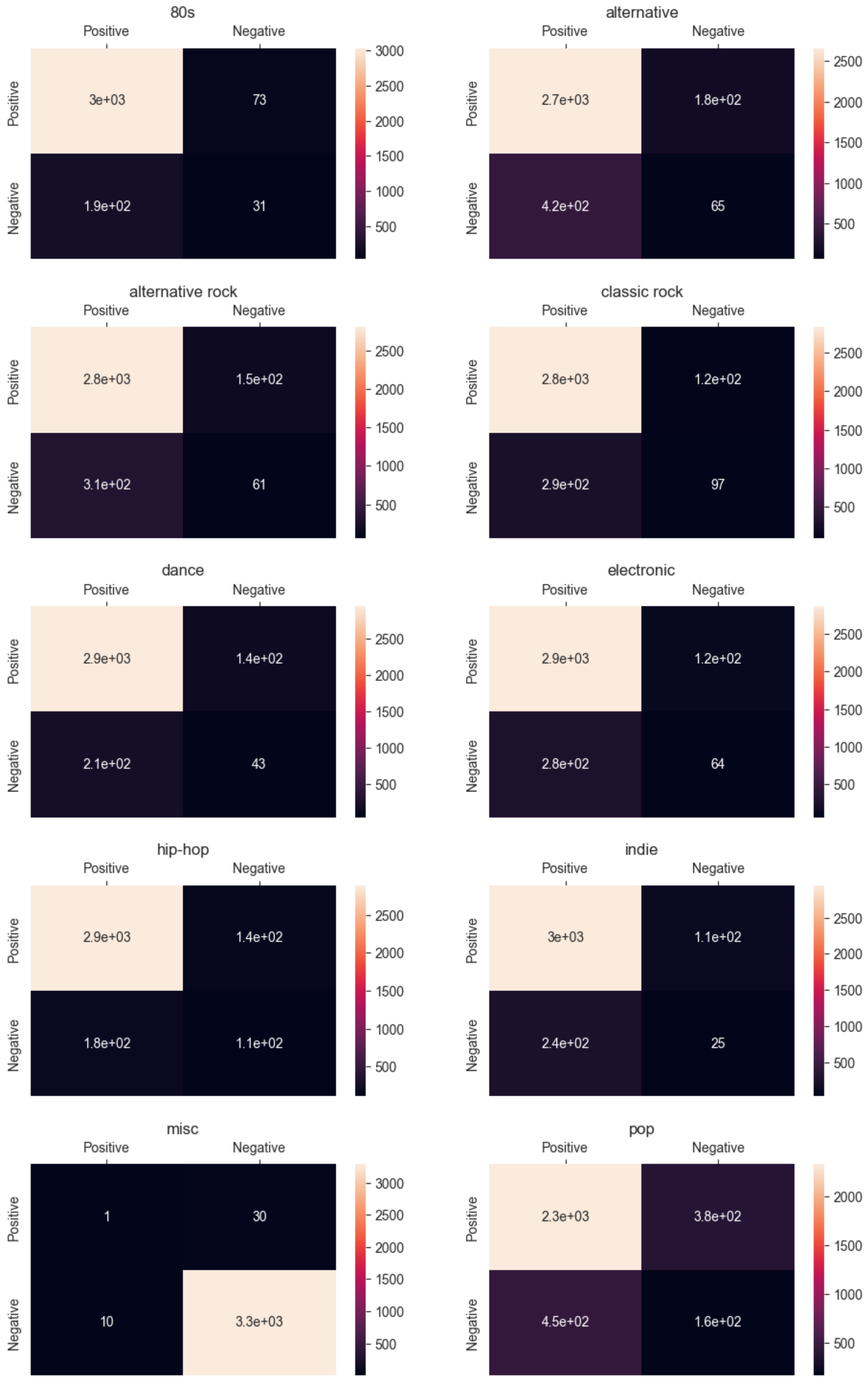


Figure 9. Confusion Matrix for each tag

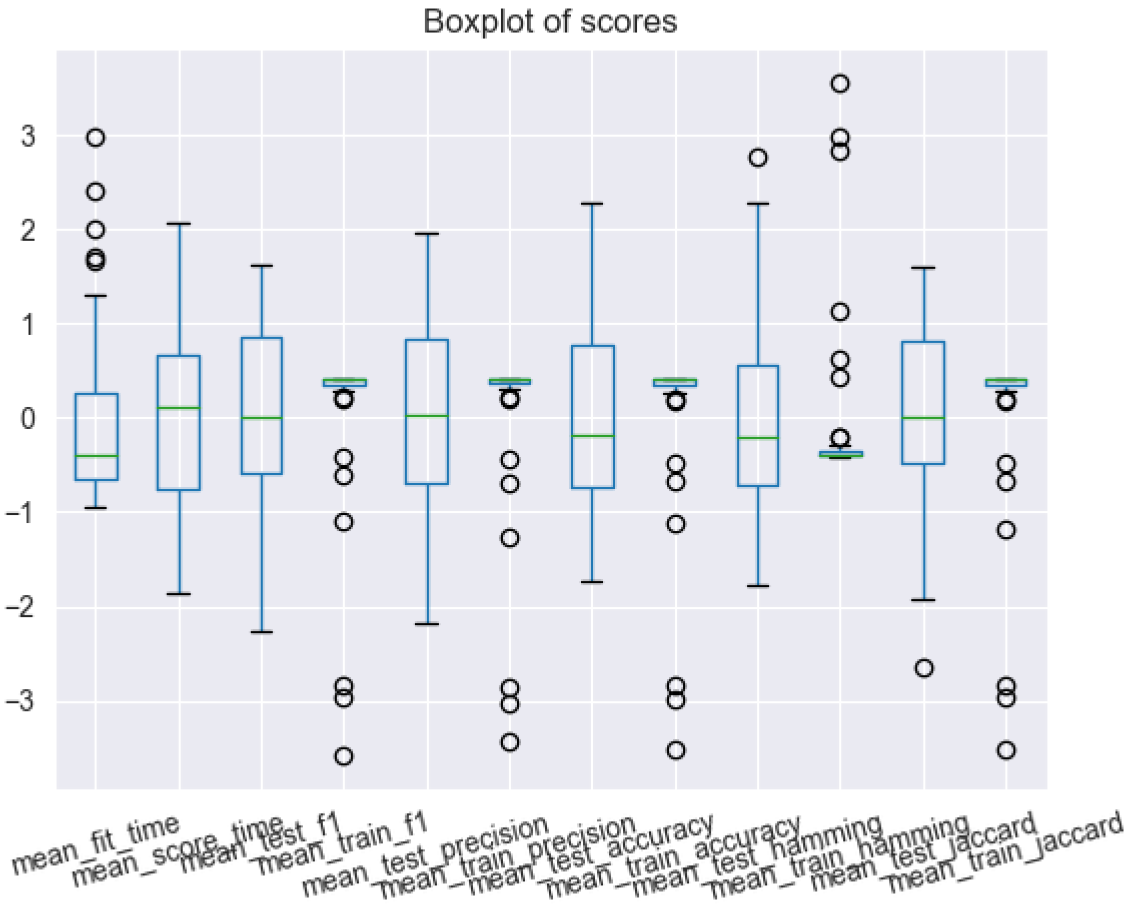


Figure 10. Performance Boxplot

5.3. Results

In conclusion, the goal of this task was to see if a machine learning model could learn to recognize the genre of a song using data from both Spotify and LastFM. However, the tags from LastFM were highly unbalanced and numerous, making it necessary to select only ten of them for the model. Despite this, the data remained highly skewed and the best accuracy achieved was only around 40%. These scores were achieved by the model learning algorithms ..., probably due to ... .

This suggests that while the model was able to learn from the data, a more balanced and representative dataset would be necessary for it to perform well on all songs on Spotify.