

DMML 2024: Final Project

Francesco Peria, Maria Paola Sforza Fogliani, Andrea Vitali

```
In [2]: import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt
```

```
In [3]: df = pd.read_csv('ravdess_features.csv', sep=',', skipinitialspace=True)  
pd.set_option('display.float_format', '{:.5f}'.format)
```

A) Data Understanding & Preparation

1. Data Semantics

```
In [4]: df.head()
```

```
Out[4]:   modality  vocal_channel  emotion  emotional_intensity  statement  repetition  actor  sex  channels  sample_width  ...  stft_m  
0    audio-only      speech    fearful        normal  Dogs are  
                                             sitting by  
                                             the door  2nd  2.00000    F      1      2  ...  0.00000  
1    audio-only      speech     angry        normal  Dogs are  
                                             sitting by  
                                             the door  1st  16.00000    F      1      2  ...  0.00000  
2    audio-only        NaN     happy       strong  Dogs are  
                                             sitting by  
                                             the door  2nd  16.00000    F      1      2  ...  0.00000  
3    audio-only        NaN  surprised        normal  Kids are  
                                             talking by  
                                             the door  1st  14.00000    F      1      2  ...  0.00000  
4    audio-only      song     happy       strong  Dogs are  
                                             sitting by  
                                             the door  2nd  2.00000    F      1      2  ...  0.00000
```

5 rows × 38 columns

```
In [5]: df.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2452 entries, 0 to 2451
Data columns (total 38 columns):
 #   Column           Non-Null Count Dtype  
 --- 
 0   modality         2452 non-null   object  
 1   vocal_channel    2256 non-null   object  
 2   emotion          2452 non-null   object  
 3   emotional_intensity 2452 non-null   object  
 4   statement         2452 non-null   object  
 5   repetition        2452 non-null   object  
 6   actor             1326 non-null   float64 
 7   sex               2452 non-null   object  
 8   channels          2452 non-null   int64  
 9   sample_width      2452 non-null   int64  
 10  frame_rate        2452 non-null   int64  
 11  frame_width       2452 non-null   int64  
 12  length_ms         2452 non-null   int64  
 13  frame_count       2452 non-null   float64 
 14  intensity          1636 non-null   float64 
 15  zero_crossings_sum 2452 non-null   int64  
 16  mfcc_mean          2452 non-null   float64 
 17  mfcc_std           2452 non-null   float64 
 18  mfcc_min           2452 non-null   float64 
 19  mfcc_max           2452 non-null   float64 
 20  sc_mean            2452 non-null   float64 
 21  sc_std              2452 non-null   float64 
 22  sc_min              2452 non-null   float64 
 23  sc_max              2452 non-null   float64 
 24  sc_kur              2452 non-null   float64 
 25  sc_skew             2452 non-null   float64 
 26  stft_mean           2452 non-null   float64 
 27  stft_std            2452 non-null   float64 
 28  stft_min            2452 non-null   float64 
 29  stft_max            2452 non-null   float64 
 30  stft_kur            2452 non-null   float64 
 31  stft_skew           2452 non-null   float64 
 32  mean                2452 non-null   float64 
 33  std                 2452 non-null   float64 
 34  min                 2452 non-null   float64 
 35  max                 2452 non-null   float64 
 36  kur                 2452 non-null   float64 
 37  skew                2452 non-null   float64 

dtypes: float64(25), int64(6), object(7)
memory usage: 728.1+ KB

```

Il dataset è composto da **2452 record e 38 attributi**. In esso sono descritte le caratteristiche di diversi file audio in cui alcuni attori (uomini o donne) pronunciano una di due frasi ("Kids are talking by the door", "Dogs are sitting by the door"), parlando o cantando, ed esprimendo una particolare emozione; per ogni record, sono inoltre riportate alcune statistiche acustiche (relative a: *Mel-Frequency Cepstral Coefficients*, *spectral centroid*, e *stft chromagram*).

Eploriamo più nel dettaglio la semantica delle variabili, iniziando con quelle **non numeriche**:

```
In [6]: non_numerical_columns = df.select_dtypes(exclude='number')
non_numerical_columns
```

	modality	vocal_channel	emotion	emotional_intensity	statement	repetition	sex
0	audio-only	speech	fearful	normal	Dogs are sitting by the door	2nd	F
1	audio-only	speech	angry	normal	Dogs are sitting by the door	1st	F
2	audio-only	Nan	happy	strong	Dogs are sitting by the door	2nd	F
3	audio-only	Nan	surprised	normal	Kids are talking by the door	1st	F
4	audio-only	song	happy	strong	Dogs are sitting by the door	2nd	F
...
2447	audio-only	speech	calm	strong	Kids are talking by the door	1st	M
2448	audio-only	speech	calm	normal	Dogs are sitting by the door	1st	M
2449	audio-only	song	sad	strong	Dogs are sitting by the door	2nd	M
2450	audio-only	speech	surprised	normal	Kids are talking by the door	1st	M
2451	audio-only	Nan	neutral	normal	Dogs are sitting by the door	2nd	M

2452 rows × 7 columns

Valori unici per ogni feature:

```
In [7]: for column in non_numerical_columns:
```

```
print(column, ":", non_numerical_columns[column].unique(), sep="")  
modality: ['audio-only']  
vocal_channel: ['speech' 'nan' 'song']  
emotion: ['fearful' 'angry' 'happy' 'surprised' 'neutral' 'calm' 'sad' 'disgust']  
emotional_intensity: ['normal' 'strong']  
statement: ['Dogs are sitting by the door' 'Kids are talking by the door']  
repetition: ['2nd' '1st']  
sex: ['F' 'M']
```

In [8]: `non_numerical_columns.nunique()`

Out[8]:

modality	1
vocal_channel	2
emotion	8
emotional_intensity	2
statement	2
repetition	2
sex	2
dtype:	int64

Vediamo che la maggior parte delle variabili categoriche sono **binarie**; tra queste, alcune (e.g. 'sex') sono **simmetriche**, e altre (e.g. 'emotional_intensity') **ordinali**. La variabile 'modality' ha un unico valore in tutto il dataset, mentre 'vocal_channel' presenta dei valori mancanti; entrambe queste caratteristiche saranno trattate in fase di data preparation.

Ad eccezione di 'modality', le variabili categoriche – e soprattutto quelle binarie – sono ben **bilanciate** (c'è la stessa proporzione tra i due statement, 'M' ed 'F' sono quasi egualmente distribuiti, etc.):

In [9]: `for column in non_numerical_columns:
 print(non_numerical_columns[column].value_counts())
 print("\n")`

modality
audio-only 2452
Name: count, dtype: int64

vocal_channel
speech 1335
song 921
Name: count, dtype: int64

emotion
fearful 376
angry 376
happy 376
calm 376
sad 376
surprised 192
disgust 192
neutral 188
Name: count, dtype: int64

emotional_intensity
normal 1320
strong 1132
Name: count, dtype: int64

statement
Dogs are sitting by the door 1226
Kids are talking by the door 1226
Name: count, dtype: int64

repetition
2nd 1226
1st 1226
Name: count, dtype: int64

sex
M 1248
F 1204
Name: count, dtype: int64

Analizziamo ora le variabili **numeriche**:

In [10]: `numerical_columns = df.select_dtypes(include='number')`

numerical_columns

	actor	channels	sample_width	frame_rate	frame_width	length_ms	frame_count	intensity	zero_crossings_sum	mfcc_
0	2.00000	1	2	48000	2	3737	179379.00000	-36.79343	16995	-33.
1	16.00000	1	2	48000	2	3904	187387.00000	NaN	13906	-29.
2	16.00000	1	2	48000	2	4671	224224.00000	-32.29074	18723	-30.
3	14.00000	1	2	48000	2	3637	174575.00000	-49.01984	11617	-36.
4	2.00000	1	2	48000	2	4404	211411.00000	-31.21450	15137	-31.
...
2447	23.00000	1	2	48000	2	4605	221021.00000	NaN	9871	-30.
2448	23.00000	1	2	48000	2	4171	200200.00000	-43.34290	8963	-31.
2449	23.00000	1	2	48000	2	5239	251451.00000	NaN	9765	-26.
2450	NaN	1	2	48000	2	3737	179379.00000	-45.75126	9716	-28.
2451	23.00000	1	2	48000	2	3837	184184.00000	-40.01804	9427	-29.

2452 rows × 31 columns

Valori unici:

In [11]: numerical_columns.nunique()

```
Out[11]: actor          24
channels        2
sample_width    1
frame_rate      1
frame_width     2
length_ms       95
frame_count     158
intensity       989
zero_crossings_sum  2176
mfcc_mean       2451
mfcc_std        2449
mfcc_min        2451
mfcc_max        2449
sc_mean         2451
sc_std          2451
sc_min          1431
sc_max          2423
sc_kur          2451
sc_skew         2451
stft_mean       2451
stft_std        2451
stft_min        1431
stft_max        1
stft_kur        2451
stft_skew       2451
mean            2450
std             2451
min             2148
max             2166
kur             2451
skew            2451
dtype: int64
```

Conteggio dei valori:

In [12]:

```
for column in numerical_columns:
    print(numerical_columns[column].value_counts())
    print("\n")
```

actor	
22.00000	65
12.00000	63
14.00000	62
20.00000	61
8.00000	61
19.00000	60
13.00000	60
2.00000	58
16.00000	58
24.00000	58
5.00000	58
10.00000	56
21.00000	55

```
11.00000 55
6.00000 55
17.00000 55
4.00000 52
3.00000 51
1.00000 51
7.00000 51
23.00000 51
9.00000 51
15.00000 44
18.00000 35
Name: count, dtype: int64
```

```
channels
1 2446
2 6
Name: count, dtype: int64
```

```
sample_width
2 2452
Name: count, dtype: int64
```

```
frame_rate
48000 2452
Name: count, dtype: int64
```

```
frame_width
2 2446
4 6
Name: count, dtype: int64
```

```
length_ms
3737 82
3604 69
3570 69
3504 67
3537 67
..
5472 1
5973 1
3003 1
5906 1
5806 1
Name: count, Length: 95, dtype: int64
```

```
frame_count
168168.00000 65
179379.00000 62
176176.00000 58
171371.00000 57
184184.00000 54
..
296296.00000 1
145745.00000 1
142542.00000 1
145746.00000 1
278679.00000 1
Name: count, Length: 158, dtype: int64
```

```
intensity
-47.38644 7
-46.78717 7
-45.01264 7
-46.17248 7
-36.52282 7
..
-29.91007 1
-29.13289 1
-34.55979 1
-45.64908 1
-29.51279 1
Name: count, Length: 989, dtype: int64
```

```
zero_crossings_sum
10667 4
```

```
12913      4
9531       3
14005      3
12769      3
...
11266      1
10147      1
17605      1
13830      1
9716       1
Name: count, Length: 2176, dtype: int64
```

```
mfcc_mean
-27.67493    2
-33.48595    1
-32.75184    1
-30.87260    1
-30.88487    1
...
-29.02738    1
-30.78250    1
-30.98836    1
-32.05165    1
-29.01924    1
Name: count, Length: 2451, dtype: int64
```

```
mfcc_std
139.45705    2
138.63795    2
149.35985    2
128.14398    1
102.01124    1
...
171.00209    1
161.13136    1
168.47473    1
169.90286    1
149.18895    1
Name: count, Length: 2449, dtype: int64
```

```
mfcc_min
-844.52500    2
-755.22345    1
-729.23010    1
-784.70020    1
-747.66090    1
...
-839.23114    1
-891.49050    1
-890.57623    1
-925.34860    1
-799.51010    1
Name: count, Length: 2451, dtype: int64
```

```
mfcc_max
227.80377    2
182.00595    2
183.27990    2
171.69092    1
206.03415    1
...
228.63307    1
221.48242    1
237.42914    1
204.68369    1
219.52780    1
Name: count, Length: 2449, dtype: int64
```

```
sc_mean
4923.91703    2
5792.55074    1
5279.16305    1
5898.30658    1
4223.68350    1
...
4889.58252    1
6715.37764    1
4612.97540    1
```

```
5575.58672    1  
6082.67612    1  
Name: count, Length: 2451, dtype: int64
```

```
sc_std  
2699.05623    2  
3328.05546    1  
3801.31582    1  
3706.57671    1  
3081.61776    1  
...  
3942.52893    1  
4279.15119    1  
3767.33907    1  
3841.91348    1  
3963.72512    1  
Name: count, Length: 2451, dtype: int64
```

```
sc_min  
0.00000      1021  
929.40279     2  
988.01203     1  
1196.10746     1  
1612.97297     1  
...  
1031.73371     1  
1218.76376     1  
996.95054     1  
901.73863     1  
760.82255     1  
Name: count, Length: 1431, dtype: int64
```

```
sc_max  
12000.00012    3  
12000.00016    3  
12000.00022    3  
12000.00003    3  
11999.99990    3  
...  
9658.84719     1  
11124.09875     1  
12000.00002     1  
9040.81609     1  
12199.77342     1  
Name: count, Length: 2423, dtype: int64
```

```
sc_kur  
-1.18352      2  
-1.12077      1  
-1.44495      1  
-1.55695      1  
-0.79018      1  
...  
-1.48638      1  
-1.74124      1  
-1.38422      1  
-1.38026      1  
-1.50139      1  
Name: count, Length: 2451, dtype: int64
```

```
sc_skew  
0.13669      2  
0.25094      1  
0.51984      1  
0.16814      1  
0.59820      1  
...  
0.49911      1  
-0.24263      1  
0.29682      1  
0.17207      1  
0.14757      1  
Name: count, Length: 2451, dtype: int64
```

```
stft_mean  
0.53313      2  
0.41525      1
```

```
0.43085    1  
0.49617    1  
0.34001    1  
..  
0.45694    1  
0.63298    1  
0.43676    1  
0.52262    1  
0.55495    1  
Name: count, Length: 2451, dtype: int64
```

```
stft_std  
0.29877    2  
0.33553    1  
0.35921    1  
0.31962    1  
0.35216    1  
..  
0.35636    1  
0.29868    1  
0.35303    1  
0.31691    1  
0.32079    1  
Name: count, Length: 2451, dtype: int64
```

```
stft_min  
0.00000    1021  
0.00250    2  
0.00335    1  
0.00024    1  
0.00070    1  
..  
0.00399    1  
0.00301    1  
0.00319    1  
0.00463    1  
0.00156    1  
Name: count, Length: 1431, dtype: int64
```

```
stft_max  
1.00000    2452  
Name: count, dtype: int64
```

```
stft_kur  
-1.16946    2  
-1.21502    1  
-1.47879    1  
-1.33480    1  
-1.07067    1  
..  
-1.50661    1  
-0.99726    1  
-1.42076    1  
-1.25520    1  
-1.25767    1  
Name: count, Length: 2451, dtype: int64
```

```
stft_skew  
-0.00762    2  
0.40351    1  
0.22509    1  
0.06777    1  
0.63525    1  
..  
0.15323    1  
-0.55023    1  
0.15074    1  
-0.07033    1  
-0.23776    1  
Name: count, Length: 2451, dtype: int64
```

```
mean  
-0.00000    2  
0.00000    2  
0.00000    1  
0.00000    1  
0.00002    1
```

```
..  
-0.00000    1  
0.00000    1  
0.00000    1  
-0.00000    1  
0.00000    1  
Name: count, Length: 2450, dtype: int64
```

```
std  
0.00679    2  
0.01448    1  
0.02351    1  
0.00970    1  
0.01774    1  
..  
0.00790    1  
0.00311    1  
0.00446    1  
0.00286    1  
0.01000    1  
Name: count, Length: 2451, dtype: int64
```

```
min  
-0.11392    4  
-0.06915    4  
-0.99881    3  
-0.10233    3  
-0.02753    3  
..  
-0.07480    1  
-0.06235    1  
-0.02402    1  
-0.21024    1  
-0.08151    1  
Name: count, Length: 2148, dtype: int64
```

```
max  
0.99881    14  
0.05518    4  
0.06226    4  
0.05478    3  
0.05307    3  
..  
0.14847    1  
0.18530    1  
0.39636    1  
0.87207    1  
0.10303    1  
Name: count, Length: 2166, dtype: int64
```

```
kur  
9.42733    2  
9.40606    1  
6.52938    1  
21.93122   1  
4.29562    1  
..  
2.94969    1  
12.94743   1  
8.55983    1  
15.01436   1  
12.97318   1  
Name: count, Length: 2451, dtype: int64
```

```
skew  
0.38833    2  
0.27315    1  
0.49945    1  
0.47879    1  
0.04838    1  
..  
0.01393    1  
-0.08540   1  
0.04711    1  
0.00921    1  
1.03208    1  
Name: count, Length: 2451, dtype: int64
```

Statistiche descrittive preliminari sulle colonne numeriche (le riprenderemo più nel dettaglio nella prossima sezione):

In [13]: `numerical_columns.describe()`

	actor	channels	sample_width	frame_rate	frame_width	length_ms	frame_count	intensity	zero_crossings_sur
count	1326.00000	2452.00000	2452.00000	2452.00000	2452.00000	2452.00000	2452.00000	1636.00000	2452.00000
mean	12.58220	1.00245	2.00000	48000.00000	2.00489	4092.15131	193587.18801	-37.62533	12885.3140
std	6.91624	0.04942	0.00000	0.00000	0.09883	598.32153	36825.36906	8.45198	3665.3195
min	1.00000	1.00000	2.00000	48000.00000	2.00000	2936.00000	-1.00000	-63.86461	4721.0000
25%	7.00000	1.00000	2.00000	48000.00000	2.00000	3604.00000	172972.00000	-43.53987	10362.5000
50%	13.00000	1.00000	2.00000	48000.00000	2.00000	4004.00000	190591.00000	-37.07274	12383.5000
75%	19.00000	1.00000	2.00000	48000.00000	2.00000	4538.00000	217817.00000	-31.59131	14966.0000
max	24.00000	2.00000	2.00000	48000.00000	4.00000	6373.00000	305906.00000	-16.35395	30153.0000

8 rows × 31 columns

Alcune osservazioni:

- la variabile '**actor**' non è da considerarsi continua, bensì un identificativo; non è quindi utile ai fini dell'analisi (vedremo, inoltre, che presenta valori mancanti);
- '**sample_width**', '**frame_rate**' e '**stft_max**' non variano mai in tutto il dataset;
- '**mean**' ha una varianza prossima allo 0;
- '**channels**' e '**frame_width**' hanno 2446 record con un certo valore, e 6 con un altro; possiamo inoltre verificare che si tratta degli stessi:

In [14]: `df[(df['channels'] == 2) & (df['frame_width'] == 4)]`

	modality	vocal_channel	emotion	emotional_intensity	statement	repetition	actor	sex	channels	sample_width	...	stf
287	audio-only	speech	fearful	normal	Kids are talking by the door	2nd	NaN	F	2	2	...	0.0
778	audio-only	speech	calm	normal	Kids are talking by the door	2nd	1.00000	M	2	2	...	0.0
1045	audio-only	speech	surprised	normal	Dogs are sitting by the door	2nd	1.00000	M	2	2	...	0.0
1336	audio-only	song	neutral	normal	Kids are talking by the door	1st	24.00000	F	2	2	...	0.0
1348	audio-only	speech	happy	normal	Dogs are sitting by the door	1st	20.00000	F	2	2	...	0.0
1809	audio-only	speech	calm	normal	Dogs are sitting by the door	2nd	5.00000	M	2	2	...	0.0

6 rows × 38 columns

Tutte queste variabili potranno quindi essere **eliminate** in fase di data preparation. Vediamo inoltre che gli altri attributi hanno scale differenti; per compararli sarà perciò necessaria una normalizzazione.

2. Distribution of the variables and statistics

Statistiche descrittive sul dataset – in particolare: **media**, **mediana**, **deviazione standard**, **moda**:

In [15]: `df.describe()`

Out[15]:

	actor	channels	sample_width	frame_rate	frame_width	length_ms	frame_count	intensity	zero_crossings_sum
count	1326.00000	2452.00000	2452.00000	2452.00000	2452.00000	2452.00000	2452.00000	1636.00000	2452.0000
mean	12.58220	1.00245	2.00000	48000.00000	2.00489	4092.15131	193587.18801	-37.62533	12885.3140
std	6.91624	0.04942	0.00000	0.00000	0.09883	598.32153	36825.36906	8.45198	3665.3195
min	1.00000	1.00000	2.00000	48000.00000	2.00000	2936.00000	-1.00000	-63.86461	4721.0000
25%	7.00000	1.00000	2.00000	48000.00000	2.00000	3604.00000	172972.00000	-43.53987	10362.5000
50%	13.00000	1.00000	2.00000	48000.00000	2.00000	4004.00000	190591.00000	-37.07274	12383.5000
75%	19.00000	1.00000	2.00000	48000.00000	2.00000	4538.00000	217817.00000	-31.59131	14966.0000
max	24.00000	2.00000	2.00000	48000.00000	4.00000	6373.00000	305906.00000	-16.35395	30153.0000

8 rows × 31 columns

Media:

In [16]: `df.mean(numeric_only=True)`

```
Out[16]: actor              12.58220
channels           1.00245
sample_width       2.00000
frame_rate        48000.00000
frame_width        2.00489
length_ms         4092.15131
frame_count       193587.18801
intensity          -37.62533
zero_crossings_sum 12885.31403
mfcc_mean          -28.76918
mfcc_std           136.77723
mfcc_min           -758.90938
mfcc_max            199.18251
sc_mean             5170.10140
sc_std              3365.45339
sc_min              551.83412
sc_max              11830.46186
sc_kur              -1.14264
sc_skew              0.34844
stft_mean           0.47585
stft_std            0.33137
stft_min            0.00227
stft_max             1.00000
stft_kur            -1.24793
stft_skew            0.11289
mean                0.00000
std                 0.02050
min                -0.16487
max                 0.17984
kur                 11.20300
skew                -0.04825
dtype: float64
```

Mediana:

In [17]: `df.median(numeric_only=True)`

```
Out[17]: actor           13.00000
channels          1.00000
sample_width      2.00000
frame_rate        48000.00000
frame_width       2.00000
length_ms         4004.00000
frame_count       190591.00000
intensity         -37.07274
zero_crossings_sum 12383.50000
mfcc_mean         -28.68111
mfcc_std          136.52381
mfcc_min          -760.98307
mfcc_max          201.69718
sc_mean           5122.71226
sc_std             3433.83537
sc_min             707.31926
sc_max             12000.29265
sc_kur             -1.30894
sc_skew            0.34762
stft_mean          0.47574
stft_std           0.33422
stft_min           0.00019
stft_max           1.00000
stft_kur           -1.29211
stft_skew          0.12606
mean               -0.00000
std                0.01388
min                -0.10378
max                0.10973
kur                9.82869
skew               0.00426
dtype: float64
```

Deviazione standard:

```
In [18]: df.std(numeric_only=True)
```

```
Out[18]: actor           6.91624
channels          0.04942
sample_width      0.00000
frame_rate        0.00000
frame_width       0.09883
length_ms         598.32153
frame_count       36825.36906
intensity         8.45198
zero_crossings_sum 3665.31958
mfcc_mean         4.46189
mfcc_std          20.45169
mfcc_min          99.94545
mfcc_max          26.00211
sc_mean           875.18544
sc_std             580.47903
sc_min             508.02589
sc_max             1004.95598
sc_kur             0.57265
sc_skew            0.35301
stft_mean          0.08255
stft_std           0.02377
stft_min           0.00483
stft_max           0.00000
stft_kur           0.21178
stft_skew          0.33076
mean               0.00004
std                0.02102
min                0.17544
max                0.19554
kur                6.61486
skew               0.45492
dtype: float64
```

La moda ha al massimo 5 parimeriti:

```
In [19]: df.mode(numeric_only=True).iloc[:10]
```

Out[19]:	actor	channels	sample_width	frame_rate	frame_width	length_ms	frame_count	intensity	zero_crossings_sum	mfcc_n
0	22.00000	1.00000	2.00000	48000.00000	2.00000	3737.00000	168168.00000	-47.38644	10667.00000	-27.6
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-46.78717	12913.00000	
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-46.17248		NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-45.01264		NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	-36.52282		NaN

5 rows × 31 columns



Conteggio dei parimeriti:

```
In [20]: df.mode(numeric_only=True).count()
```

```
Out[20]: actor           1
channels          1
sample_width      1
frame_rate        1
frame_width       1
length_ms         1
frame_count       1
intensity         5
zero_crossings_sum  2
mfcc_mean         1
mfcc_std          3
mfcc_min          1
mfcc_max          3
sc_mean           1
sc_std            1
sc_min            1
sc_max            5
sc_kur             1
sc_skew            1
stft_mean         1
stft_std          1
stft_min          1
stft_max          1
stft_kur          1
stft_skew          1
mean              2
std               1
min              2
max              1
kur              1
skew              1
dtype: int64
```

Visualizzazione più compatta delle **statistiche descrittive**:

```
In [21]: stat_df = pd.DataFrame({'Mean': df.mean(numeric_only=True),
                               'Median': df.median(numeric_only=True),
                               'Mode': df.mode(numeric_only=True).iloc[0],
                               'Standard Deviation': df.std(numeric_only=True)})
```

stat_df

Out[21]:

	Mean	Median	Mode	Standard Deviation
actor	12.58220	13.00000	22.00000	6.91624
channels	1.00245	1.00000	1.00000	0.04942
sample_width	2.00000	2.00000	2.00000	0.00000
frame_rate	48000.00000	48000.00000	48000.00000	0.00000
frame_width	2.00489	2.00000	2.00000	0.09883
length_ms	4092.15131	4004.00000	3737.00000	598.32153
frame_count	193587.18801	190591.00000	168168.00000	36825.36906
intensity	-37.62533	-37.07274	-47.38644	8.45198
zero_crossings_sum	12885.31403	12383.50000	10667.00000	3665.31958
mfcc_mean	-28.76918	-28.68111	-27.67493	4.46189
mfcc_std	136.77723	136.52381	138.63795	20.45169
mfcc_min	-758.90938	-760.98307	-844.52500	99.94545
mfcc_max	199.18251	201.69718	182.00595	26.00211
sc_mean	5170.10140	5122.71226	4923.91703	875.18544
sc_std	3365.45339	3433.83537	2699.05623	580.47903
sc_min	551.83412	707.31926	0.00000	508.02589
sc_max	11830.46186	12000.29265	11999.99990	1004.95598
sc_kur	-1.14264	-1.30894	-1.18352	0.57265
sc_skew	0.34844	0.34762	0.13669	0.35301
stft_mean	0.47585	0.47574	0.53313	0.08255
stft_std	0.33137	0.33422	0.29877	0.02377
stft_min	0.00227	0.00019	0.00000	0.00483
stft_max	1.00000	1.00000	1.00000	0.00000
stft_kur	-1.24793	-1.29211	-1.16946	0.21178
stft_skew	0.11289	0.12606	-0.00762	0.33076
mean	0.00000	-0.00000	-0.00000	0.00004
std	0.02050	0.01388	0.00679	0.02102
min	-0.16487	-0.10378	-0.11392	0.17544
max	0.17984	0.10973	0.99881	0.19554
kur	11.20300	9.82869	9.42733	6.61486
skew	-0.04825	0.00426	0.38833	0.45492

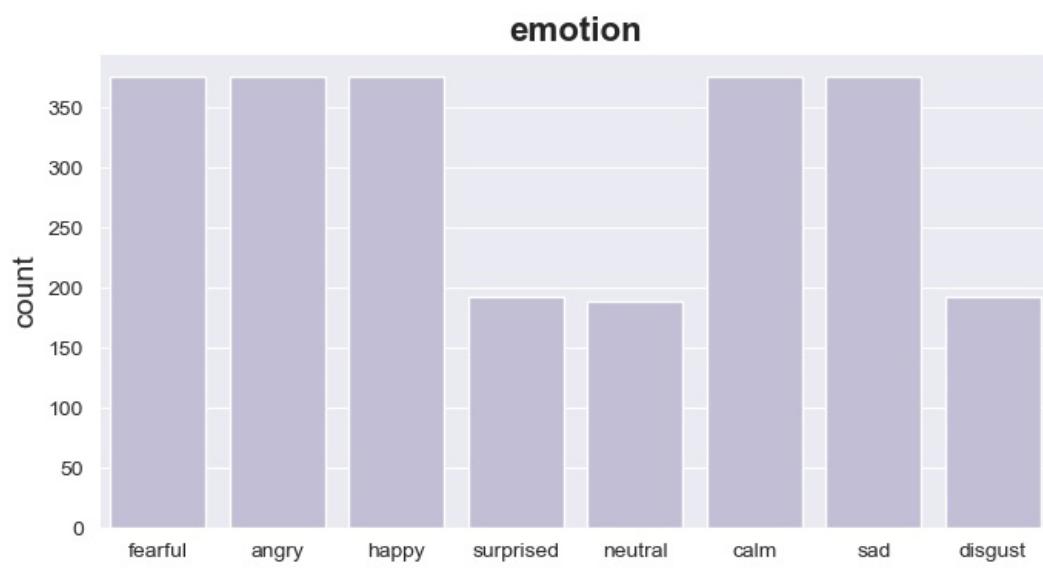
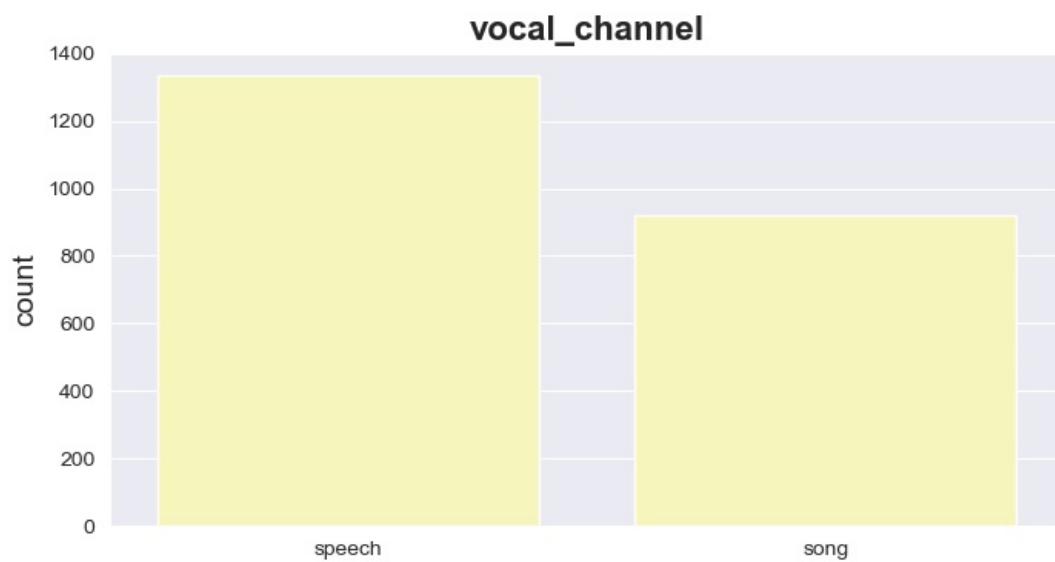
Usiamo alcune **visualizzazioni** per capire meglio le distribuzioni – prima esplorando le variabili **singolarmente**, e focalizzandoci poi su alcune delle loro **interazioni**.

In [22]: `import seaborn as sns`

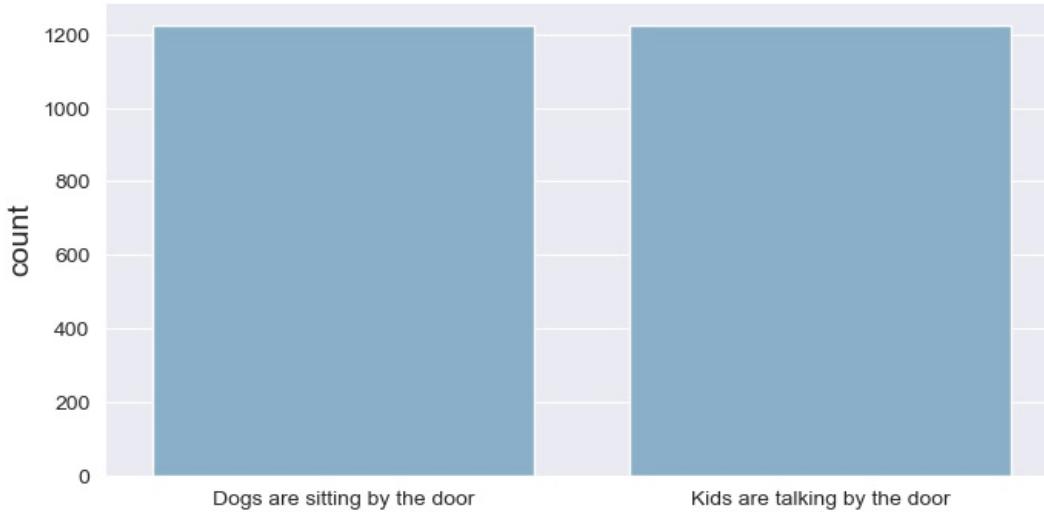
Iniziamo dalle variabili **categoriche**, usando quindi dei **bar chart**; per le ragioni viste in precedenza, escludiamo dalla visualizzazione 'modality':

In [23]: `palette = sns.color_palette("Set3")
sns.set_style("darkgrid")`

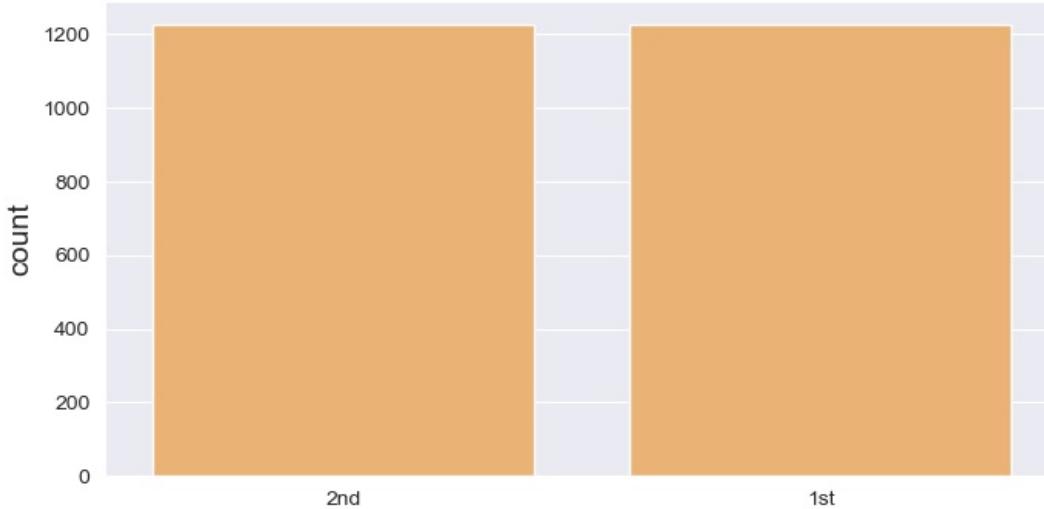
```
for i, col in enumerate(non_numerical_columns):
    if col != 'modality':
        plt.figure(figsize=(8, 4))
        color = palette[i]
        sns.countplot(x=col, data=df, color=color)
        plt.title(col, fontsize=16, fontweight='bold')
        plt.ylabel('count', fontsize=14)
        plt.xlabel('')
        plt.show()
```



statement



repetition



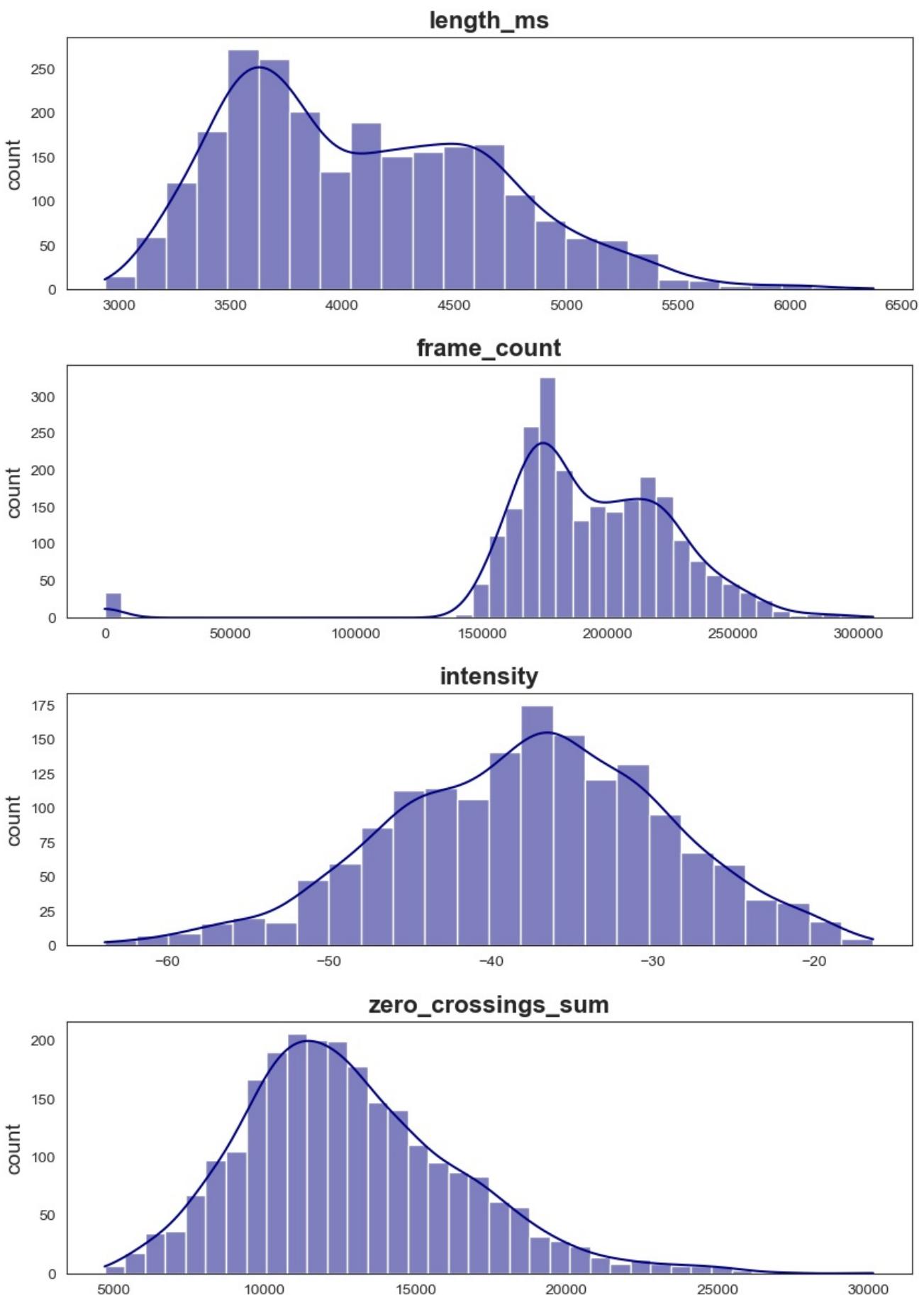
sex

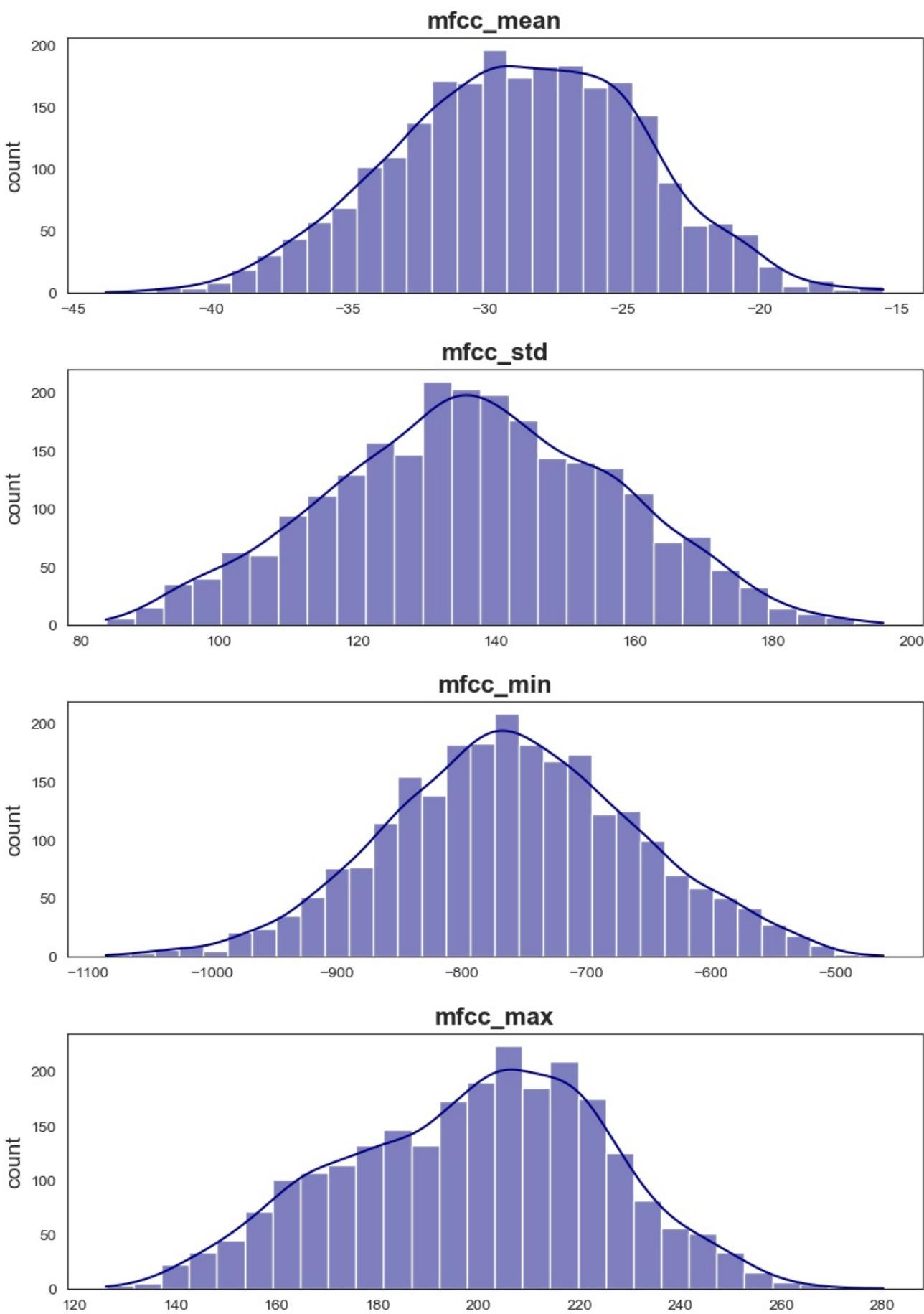


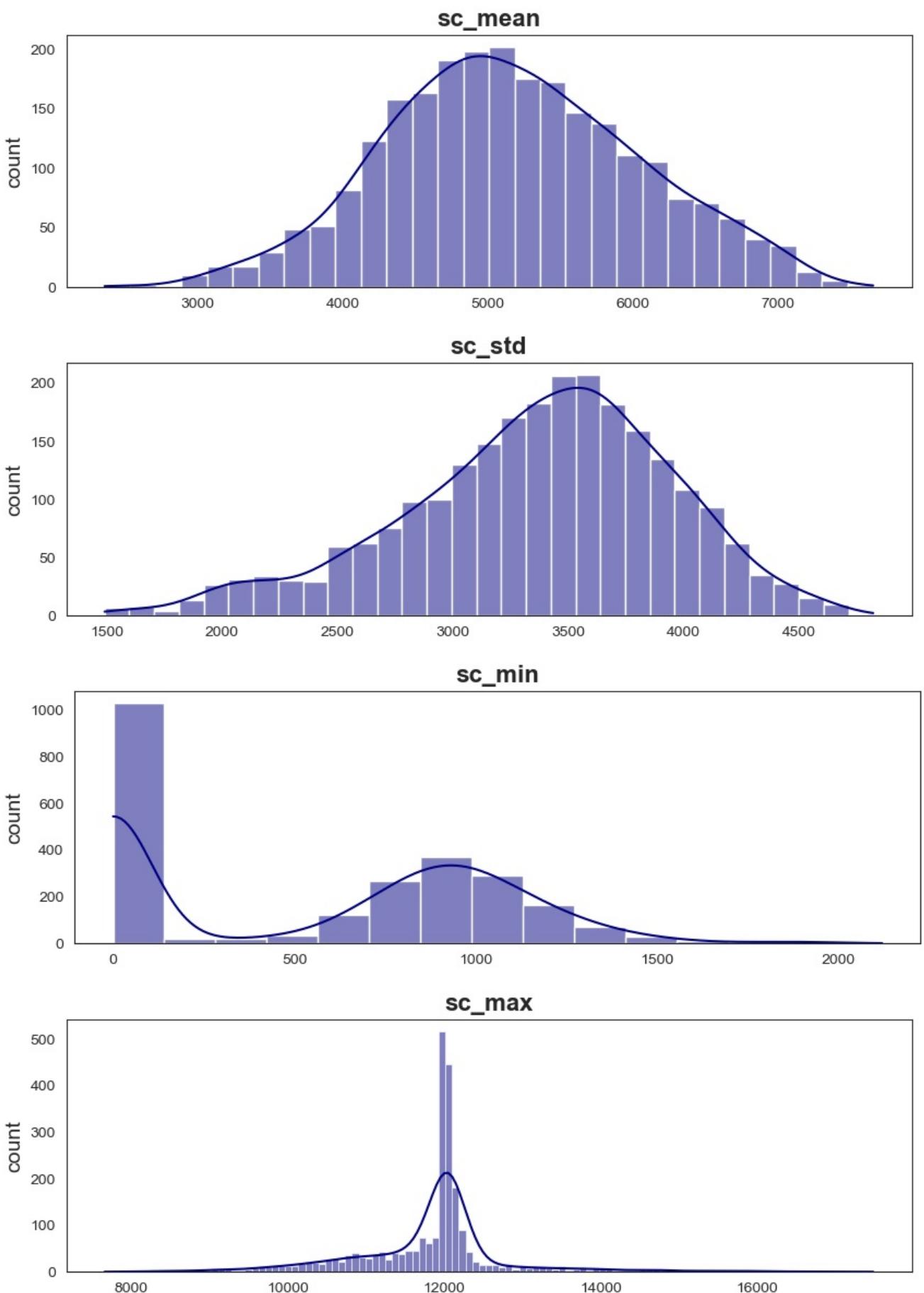
Per la visualizzazione delle variabili **continue** usiamo invece degli **istogrammi**; sempre a causa delle ragioni viste in precedenza, escludiamo dalla visualizzazione gli attributi 'actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max':

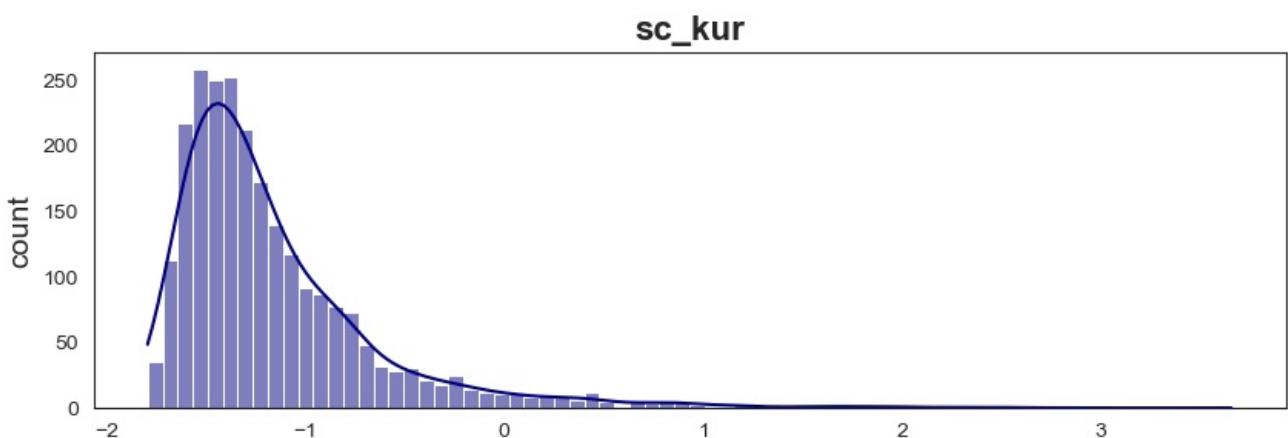
```
In [24]: sns.set_style("white")

for col in numerical_columns:
    if col not in ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max']:
        plt.figure(figsize=(10, 3))
        sns.histplot(df[col], kde=True, color='navy')
        plt.title(col, fontsize=16, fontweight='bold')
        plt.ylabel('count', fontsize=14)
        plt.xlabel('')
        plt.show()
```

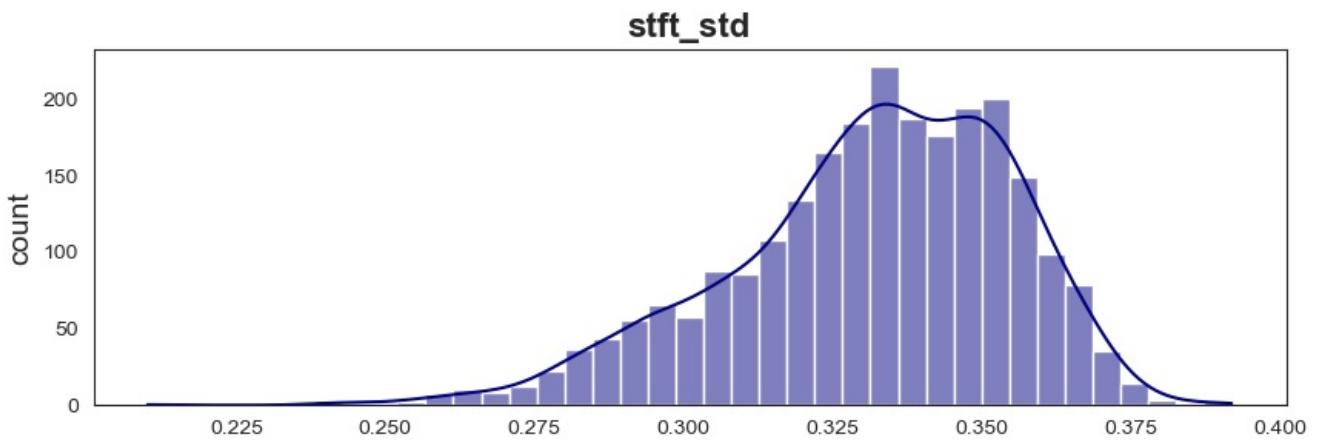
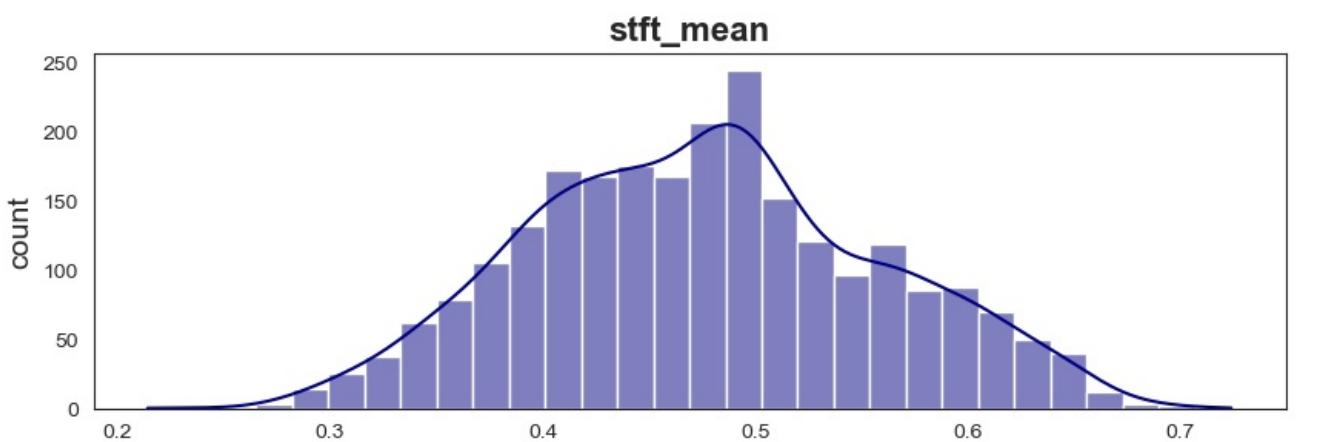


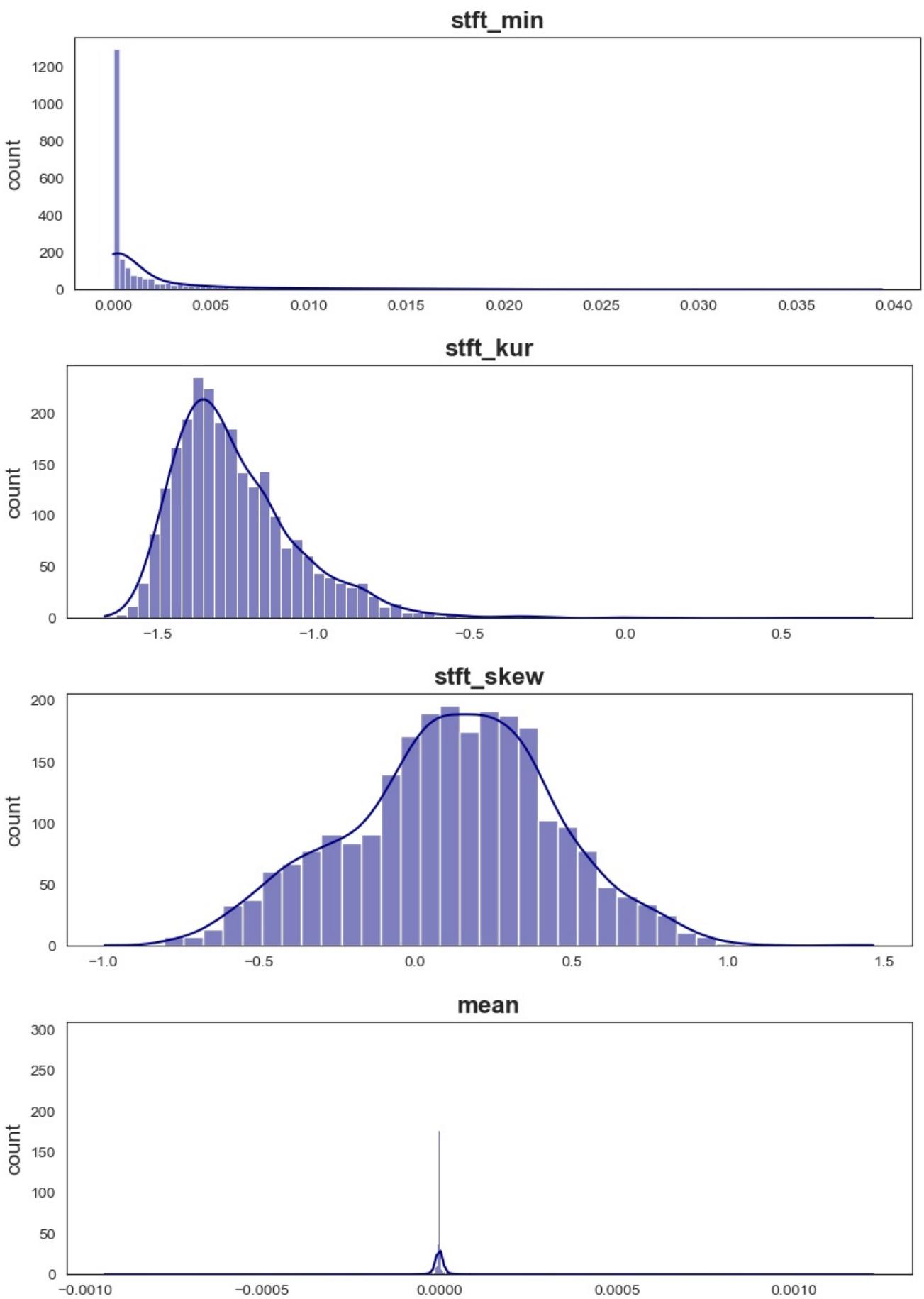


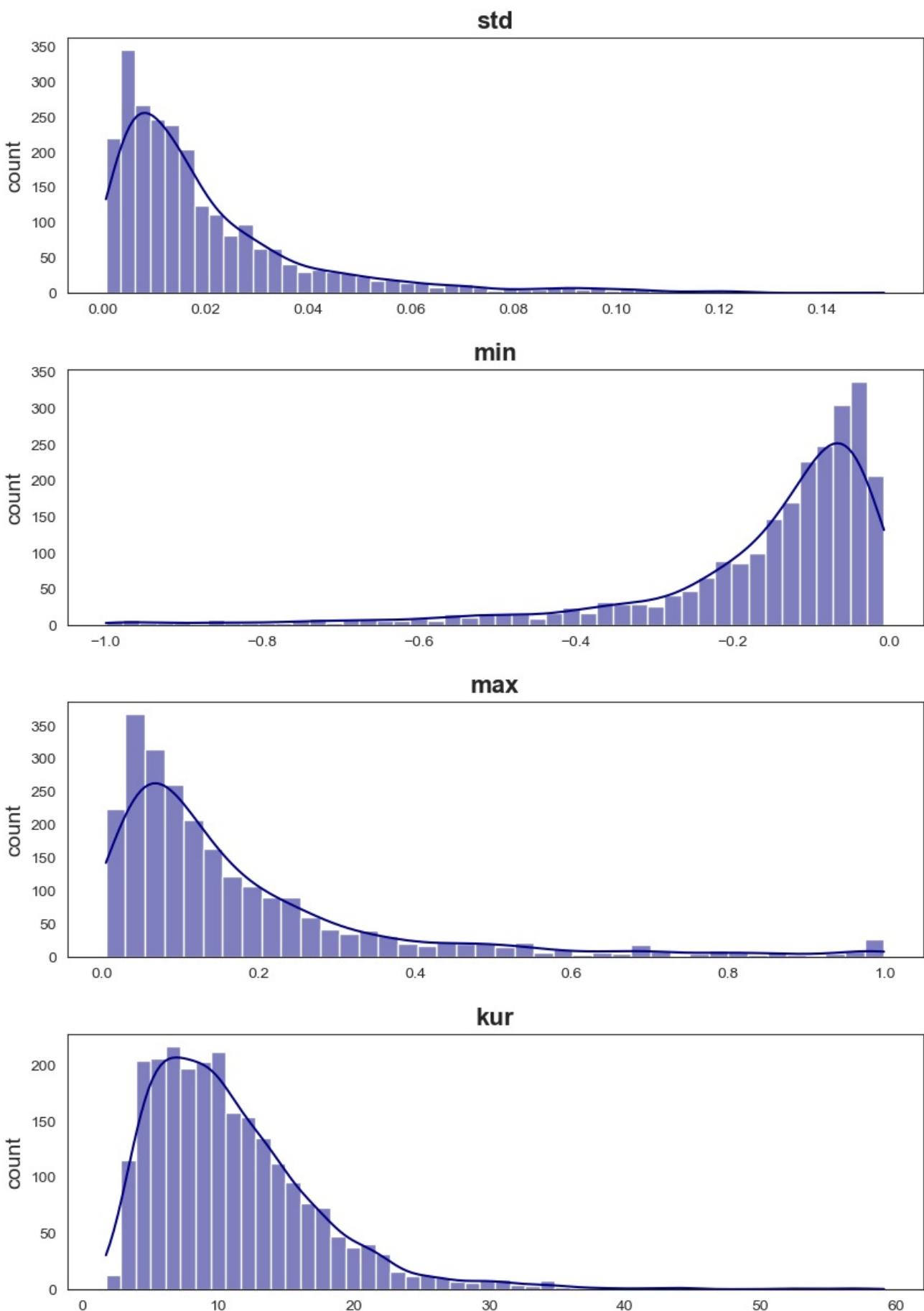


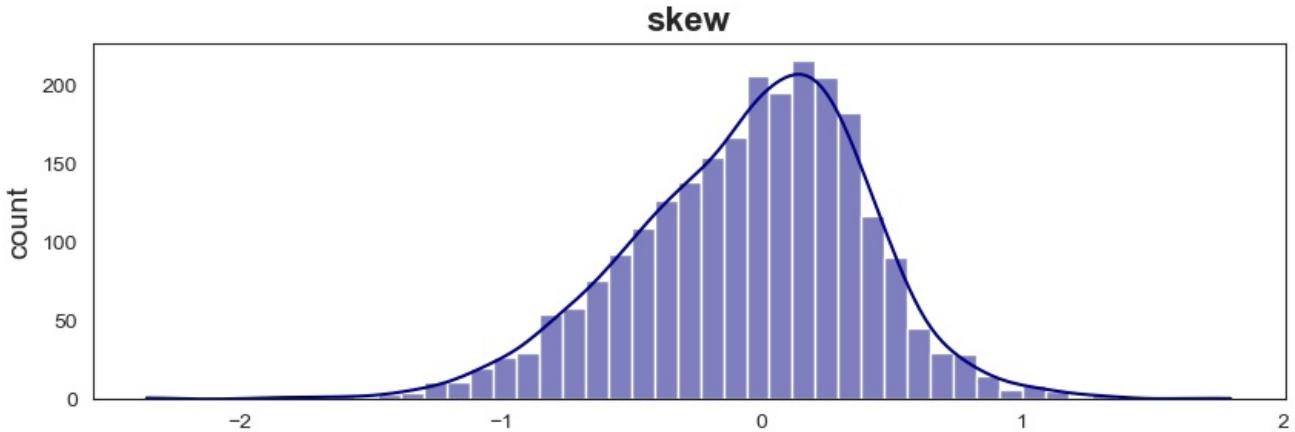


sc_skew.....





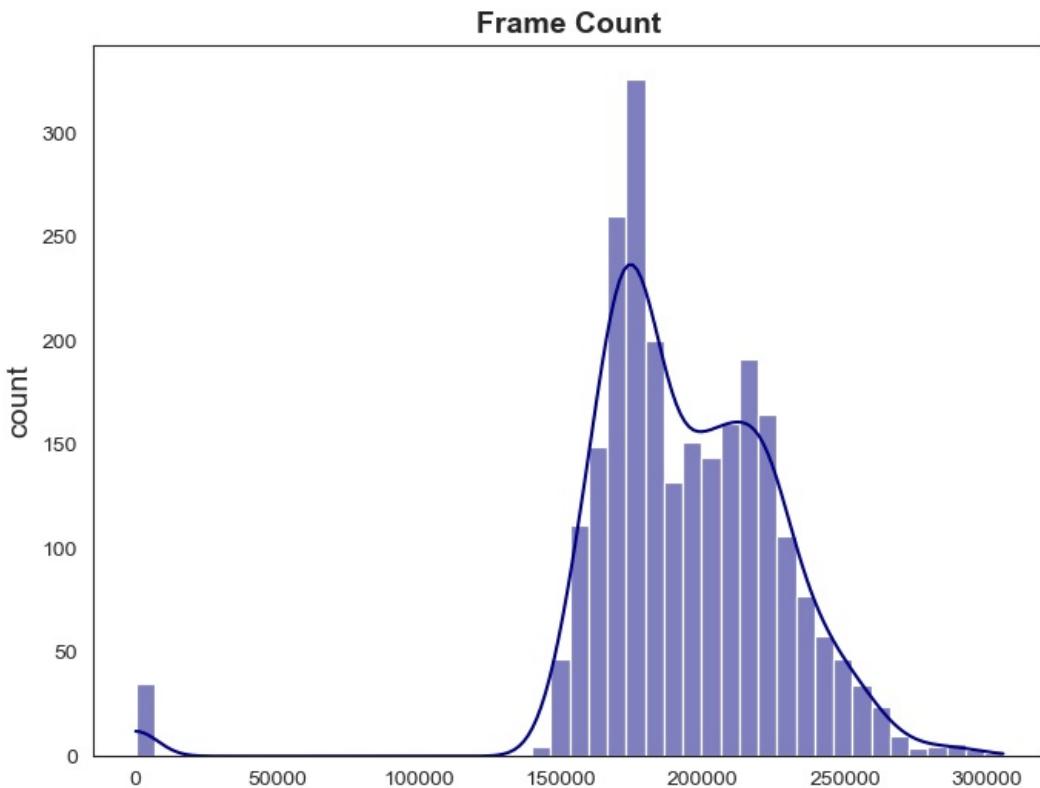




Analizziamo più nel dettaglio alcuni degli istogrammi:

```
In [25]: plt.figure(figsize=(8, 6))
sns.histplot(df['frame_count'], color='navy', kde=True)
plt.ylabel('count', fontsize=14)
plt.xlabel('')
plt.title('Frame Count', fontsize=14, fontweight='bold')

plt.show()
```



L'attributo 'frame_count' presenta alcuni **valori inferiori a 0**; data la semantica con cui è presentato nella descrizione del dataset ("numero di frame nel sample"), non dovrebbe averne:

```
In [26]: len(df[df['frame_count'] < 0])
```

```
Out[26]: 35
```

Anche aumentando di molto la soglia, solo questi 35 record hanno valori così piccoli:

```
In [27]: len(df[df['frame_count'] < 100000])
```

```
Out[27]: 35
```

Infatti, il secondo valore unico più piccolo è 140941:

```
In [28]: unique_fc = df['frame_count'].unique()
sorted_unique_fc = sorted(unique_fc)
sorted_unique_fc[:10]
```

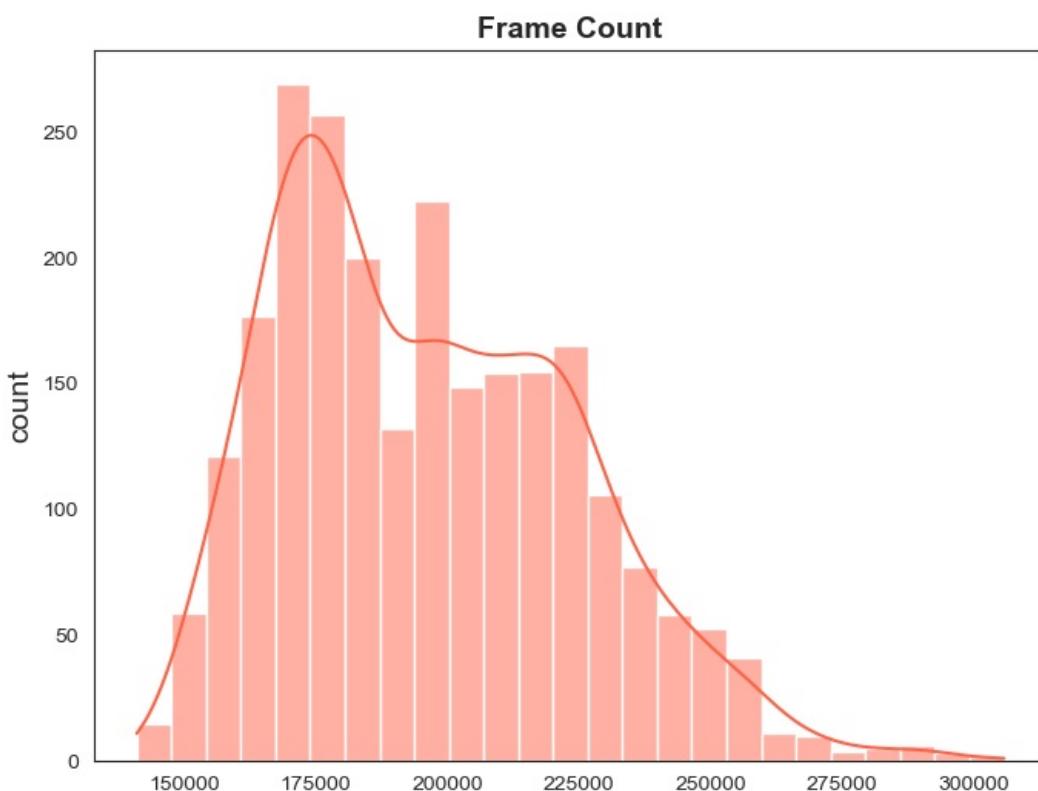
```
Out[28]: [-1.0,
 140941.0,
 142542.0,
 144144.0,
 145745.0,
 145746.0,
 147347.0,
 147348.0,
 148948.0,
 148949.0]
```

Pensiamo quindi che si tratti di **errori**; li rimpiazziamo con la **media** che la variabile avrebbe se non ci fossero i valori < 0:

```
In [29]: mean_fc = df['frame_count'][df['frame_count'] > 0].mean()
df.loc[df['frame_count'] < 0, 'frame_count'] = mean_fc
```

E ne plottiamo l'istogramma aggiornato:

```
In [30]: plt.figure(figsize=(8, 6))
sns.histplot(df['frame_count'], color='tomato', kde=True)
plt.ylabel('count', fontsize=14)
plt.xlabel('')
plt.title('Frame Count', fontsize=14, fontweight='bold')
plt.show()
```



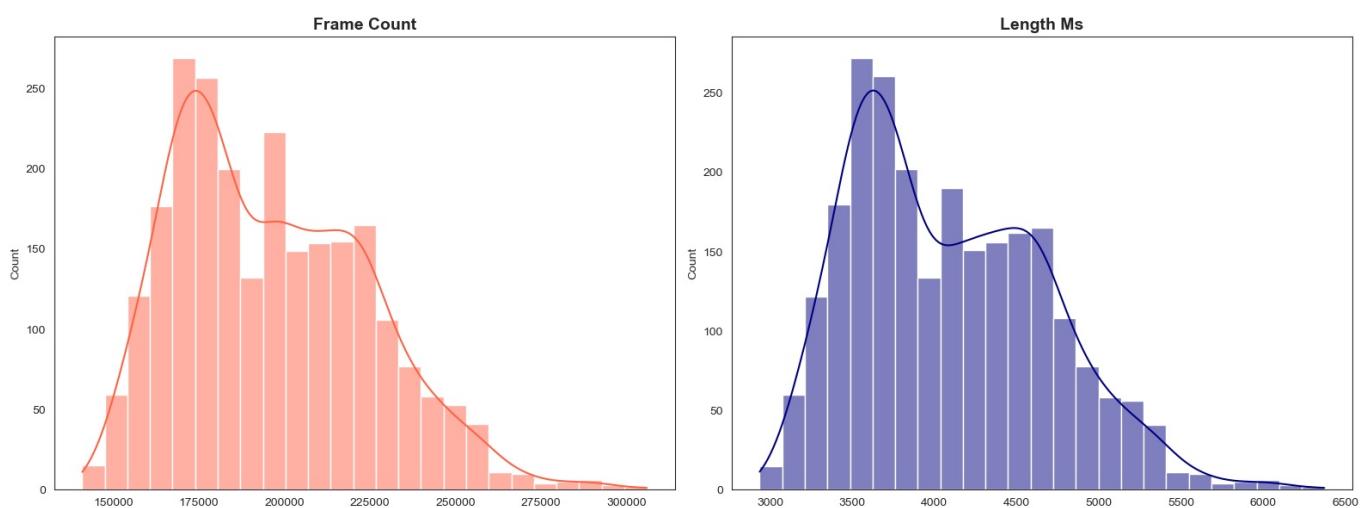
La nuova distribuzione della variabile è **molti simile a quella di 'length_ms'**; semanticamente, questo è infatti un risultato che ci aspetteremmo:

```
In [31]: fig, axes = plt.subplots(1, 2, figsize=(16, 6))

sns.histplot(df['frame_count'], ax=axes[0], color='tomato', kde=True)
axes[0].set_title('Frame Count', fontsize=14, fontweight='bold')
axes[0].set_xlabel('')

sns.histplot(df['length_ms'], ax=axes[1], color='navy', kde=True)
axes[1].set_title('Length Ms', fontsize=14, fontweight='bold')
axes[1].set_xlabel('')

plt.tight_layout()
plt.show()
```



Le due variabili **correlano** quasi perfettamente:

```
In [32]: df['length_ms'].corr(df['frame_count'])
```

```
Out[32]: 0.9929036807367501
```

Torneremo nel seguito al trattamento di queste e di diverse delle altre variabili. Per ora, possiamo fare alcune osservazioni a partire dagli istogrammi:

- 'sc_min' e 'stft_min' hanno entrambe 1021 valori a 0; si tratta in effetti degli stessi record:

```
In [33]: len(df[(df['sc_min'] == 0) & (df['stft_min'] == 0)])
```

```
Out[33]: 1021
```

- Molte delle altre variabili (e.g. 'sc_kur', 'stft_std', 'stft_kur', 'std', 'min', 'max', 'kur') presentano **distribuzioni asimmetriche**; ad esempio, 'sc_kur' ha un massimo di 3.65, ma meno del 10% dei valori è > -0.5:

```
In [34]: df['sc_kur'].max()
```

```
Out[34]: 3.65795320733888
```

```
In [35]: len(df[df['sc_kur'] > -0.5]) / len(df['sc_kur'])
```

```
Out[35]: 0.09787928221859707
```

- La variabile 'mean' ha una varianza vicina allo 0:

```
In [36]: df['mean'].var()
```

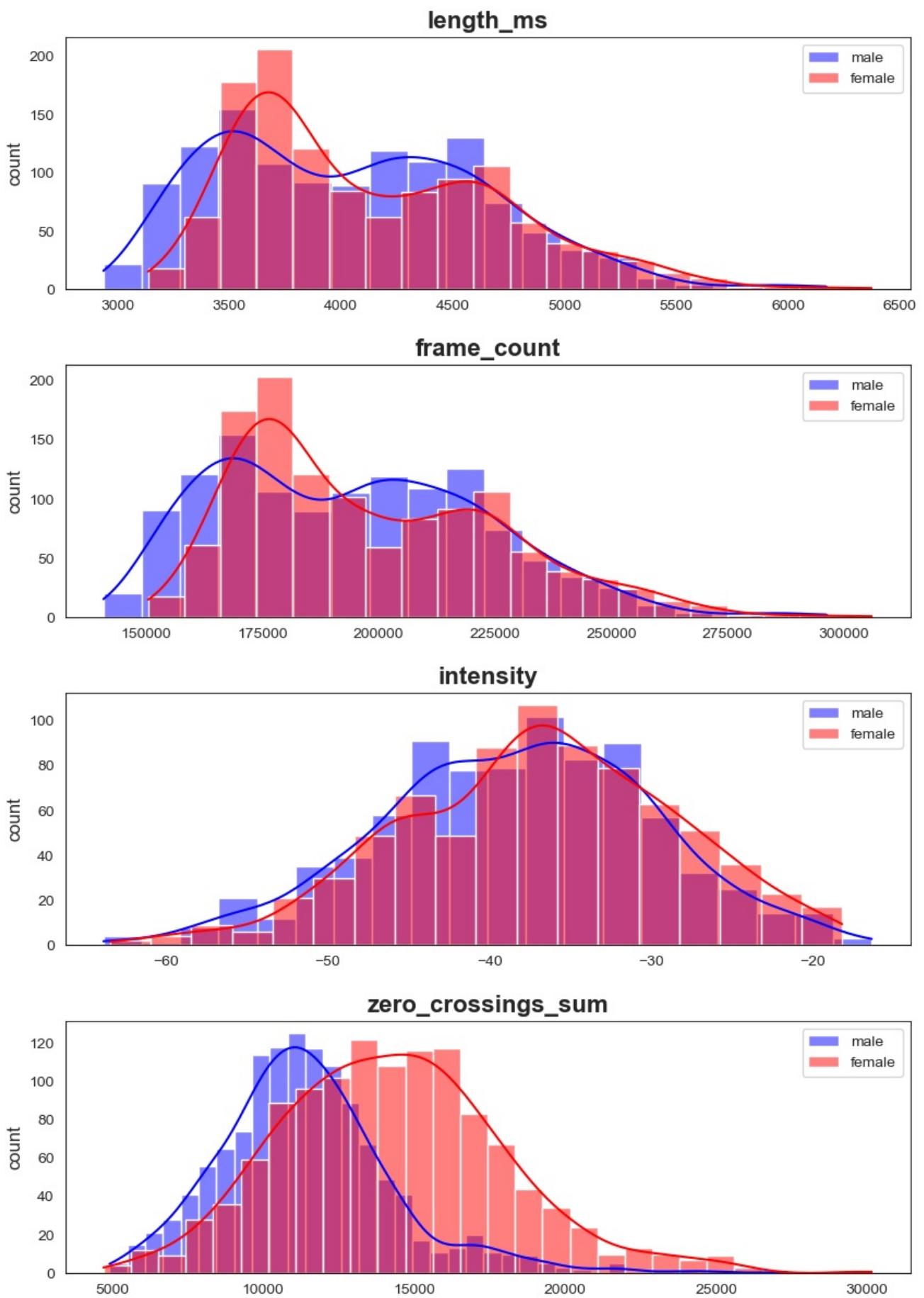
```
Out[36]: 1.8212297990049587e-09
```

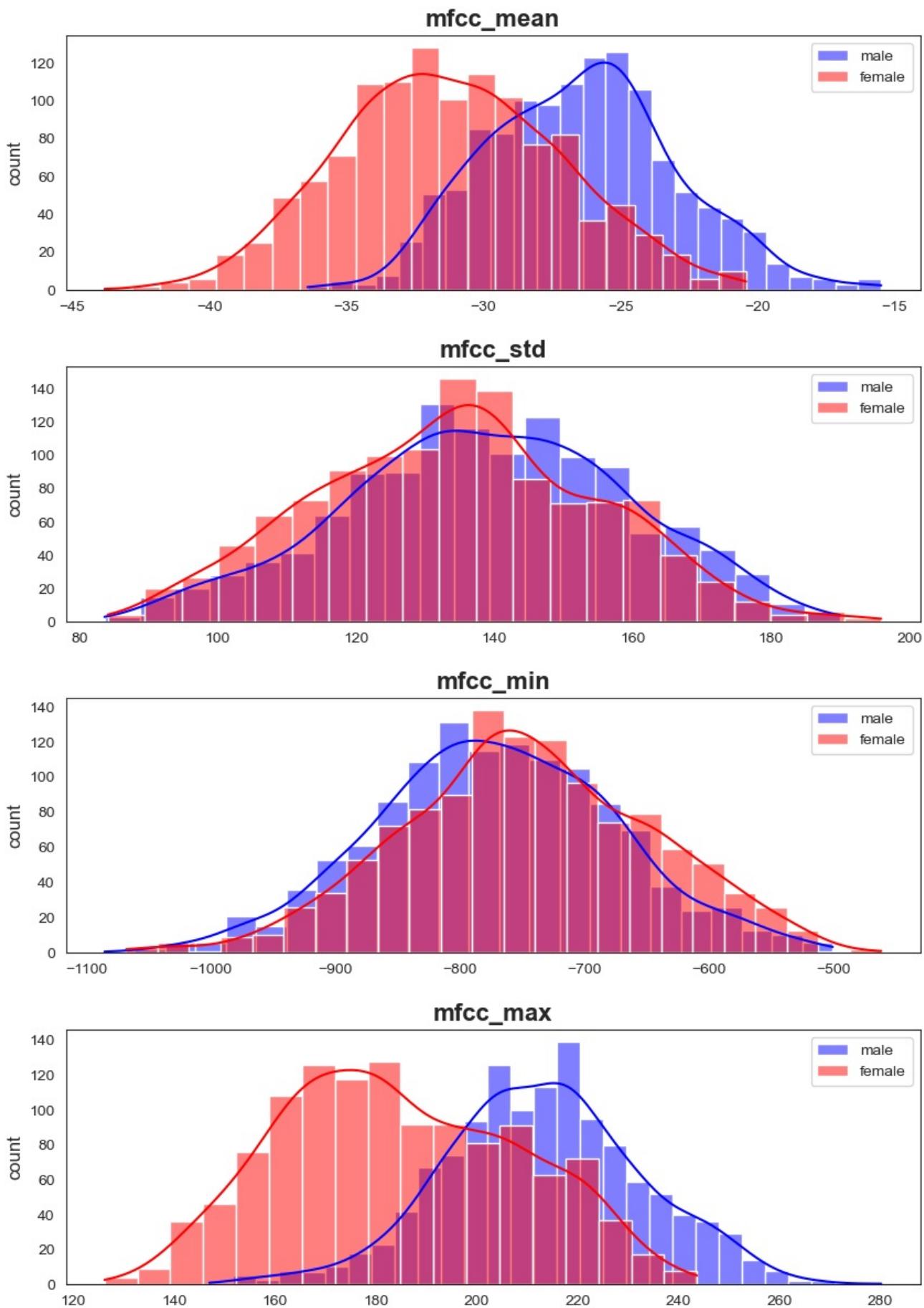
Dopo aver esplorato le variabili singolarmente, analizziamo ora alcune delle loro **interazioni**. In particolare, ci focalizziamo su 'sex' ed 'emotion', perché saranno i target della nostra classificazione.

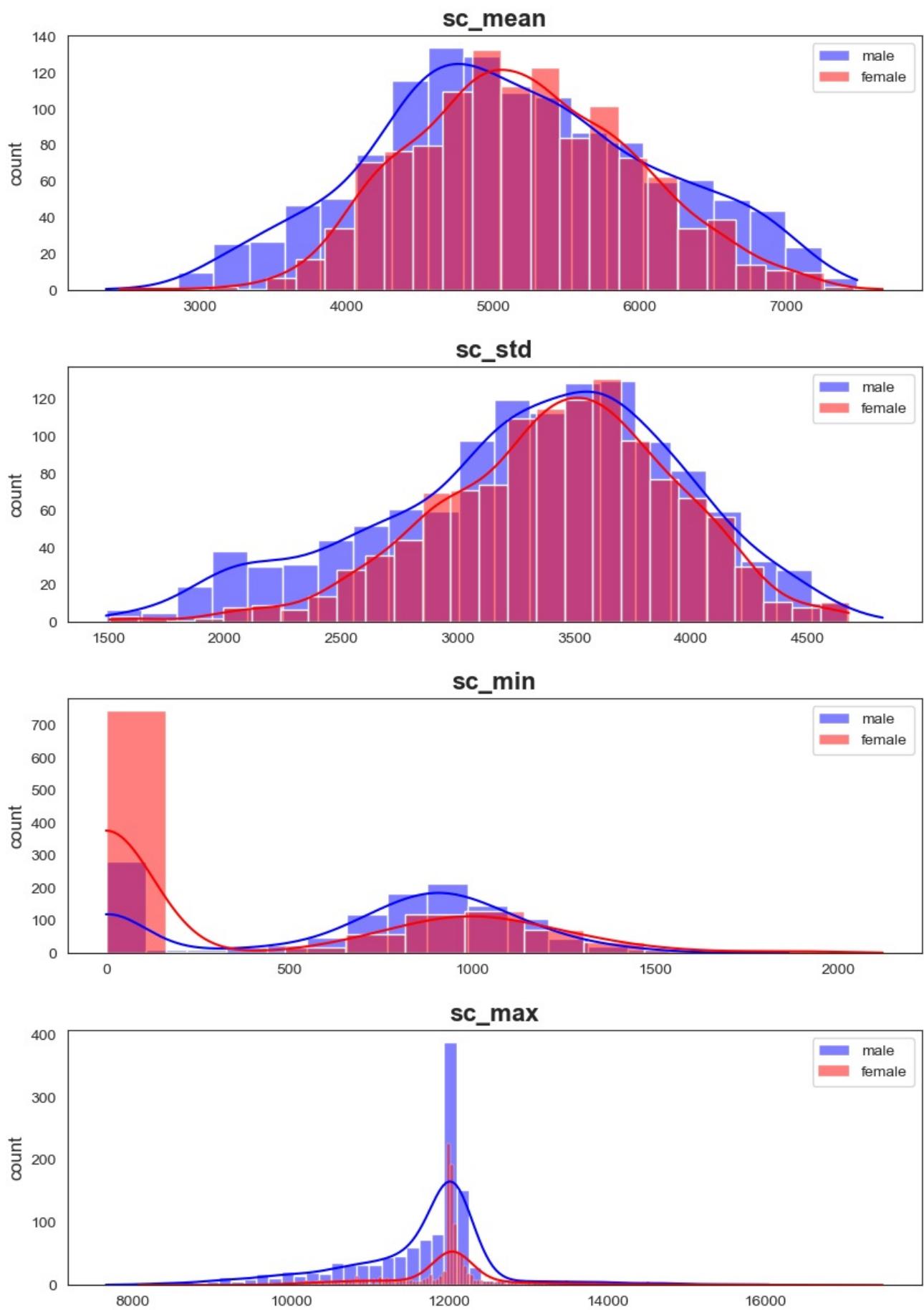
Sex:

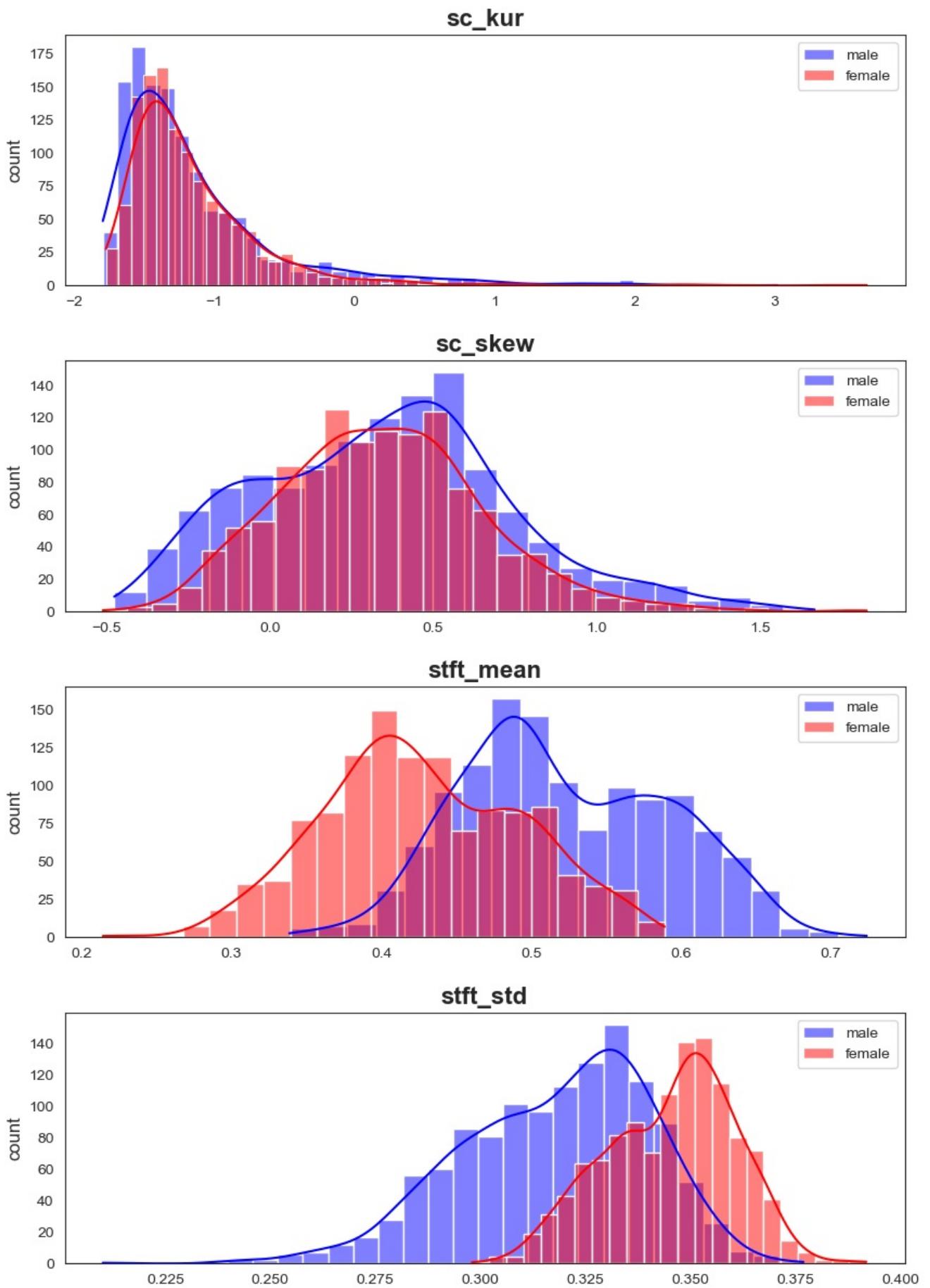
```
In [37]: male = df['sex'] == 'M'
female = df['sex'] == 'F'
```

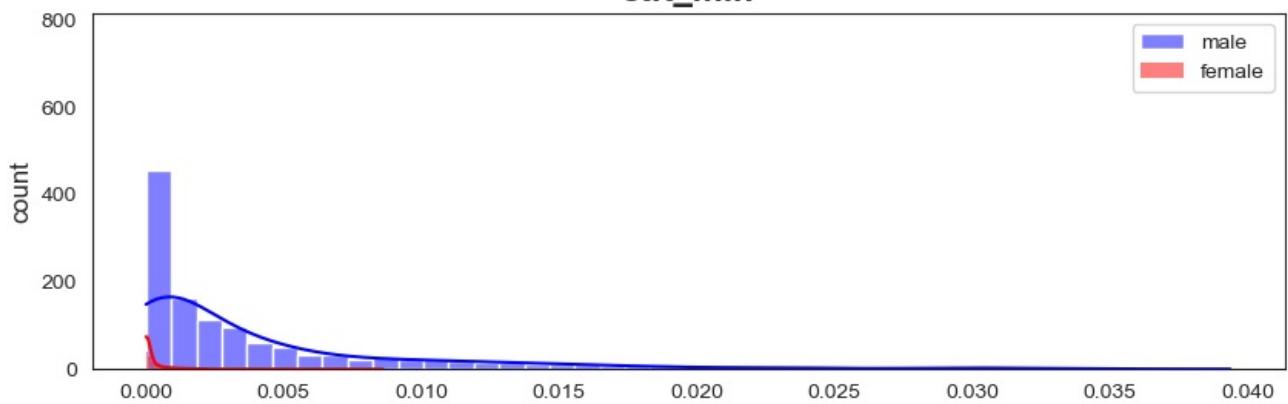
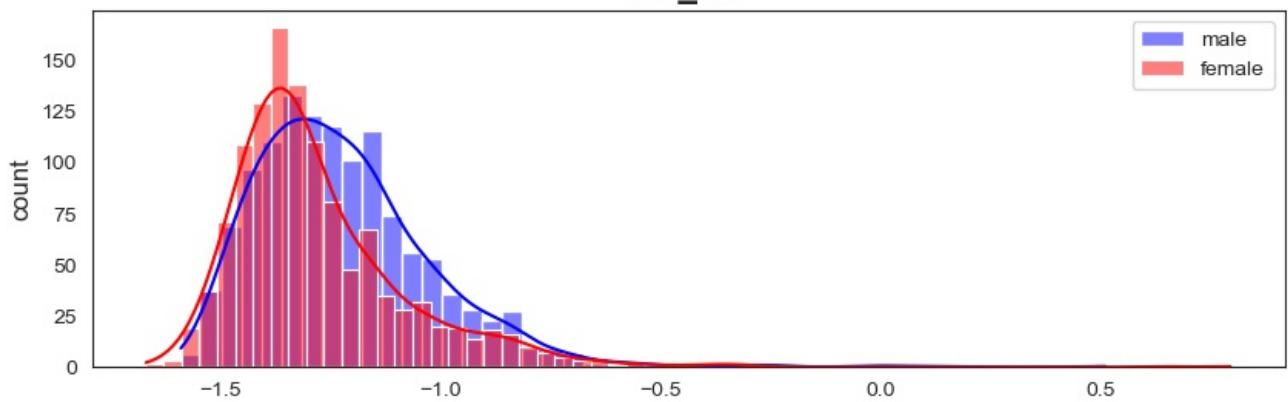
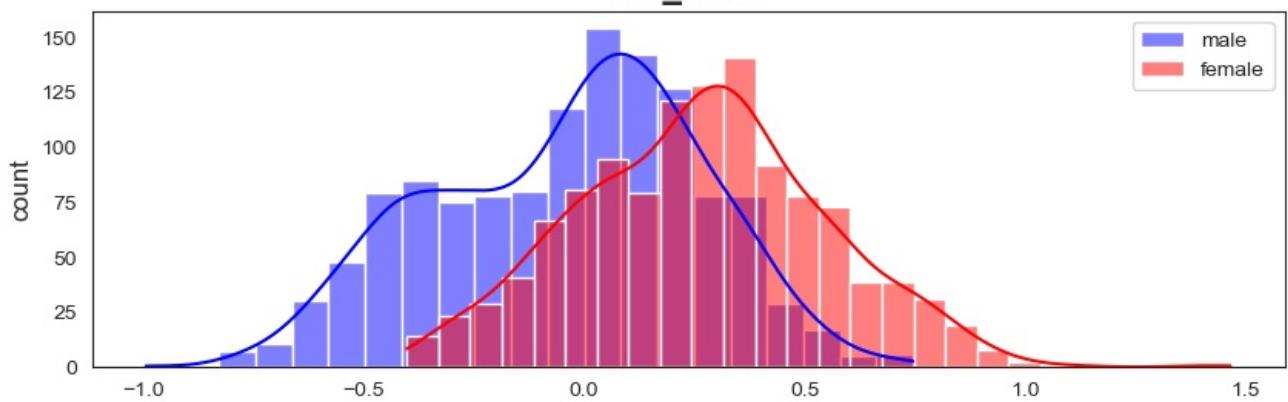
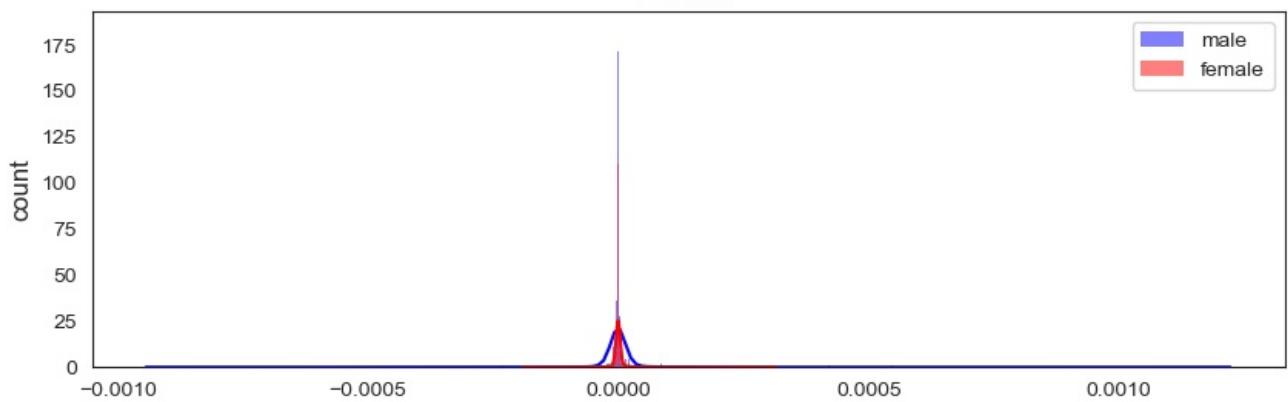
```
In [38]: for col in numerical_columns:
    if col not in ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max']:
        plt.figure(figsize=(10, 3))
        sns.histplot(df[male][col], label='male', kde=True, alpha=0.5, color='b')
        sns.histplot(df[female][col], label='female', kde=True, alpha=0.5, color='r')
        plt.title(col, fontsize=16, fontweight='bold')
        plt.ylabel('count', fontsize=12)
        plt.xlabel('')
        plt.legend()
        plt.show()
```

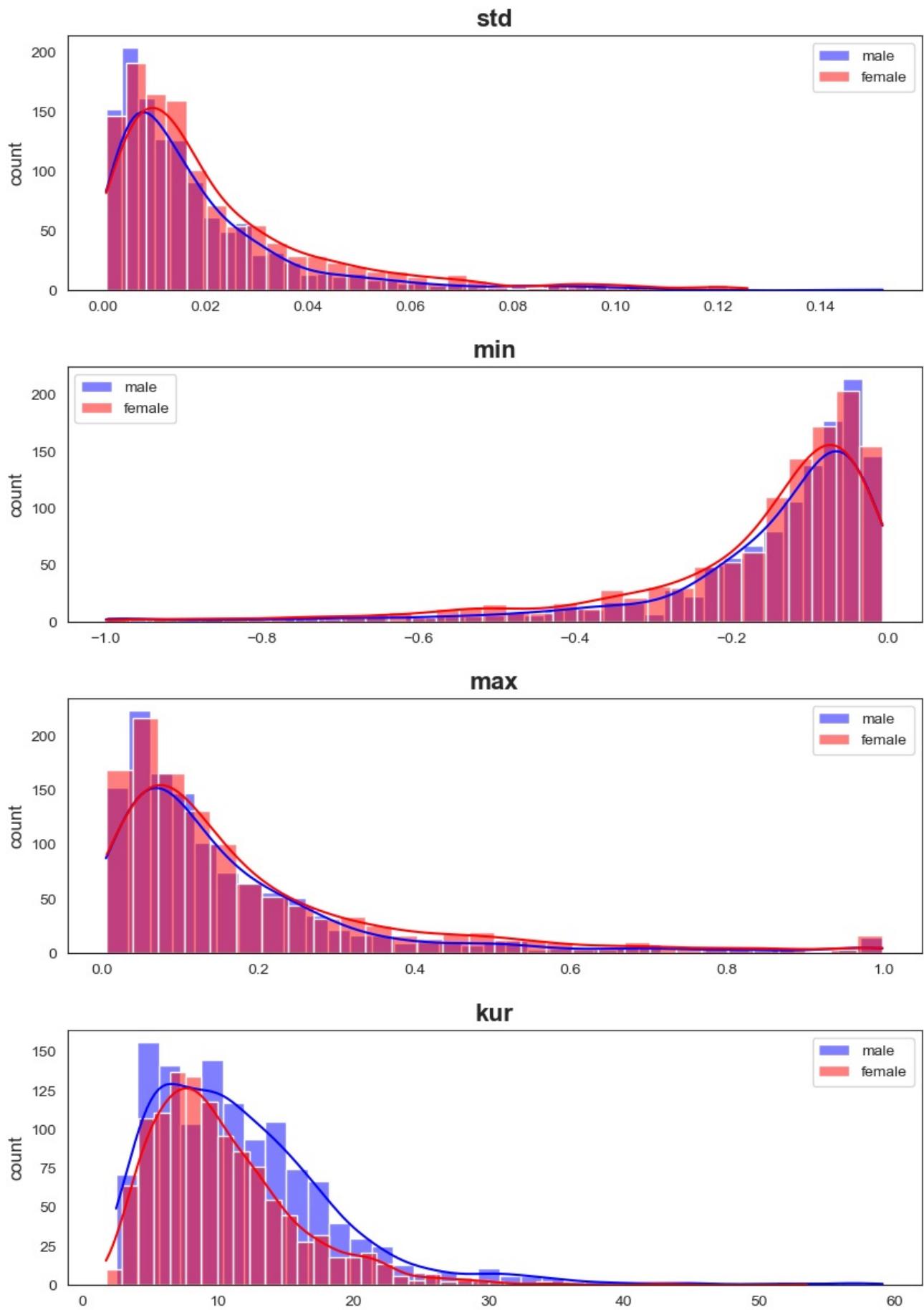


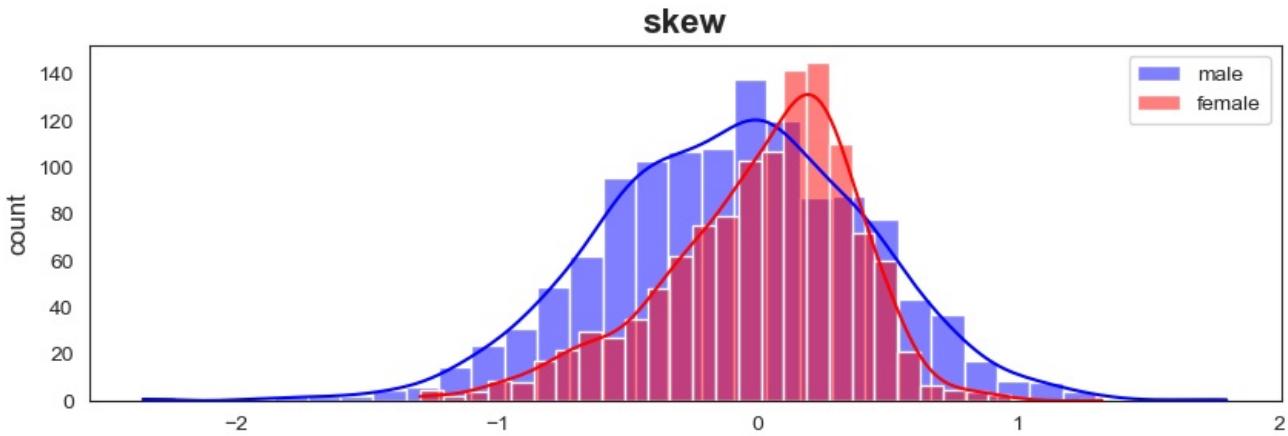






stft_min**stft_kur****stft_skew****mean**





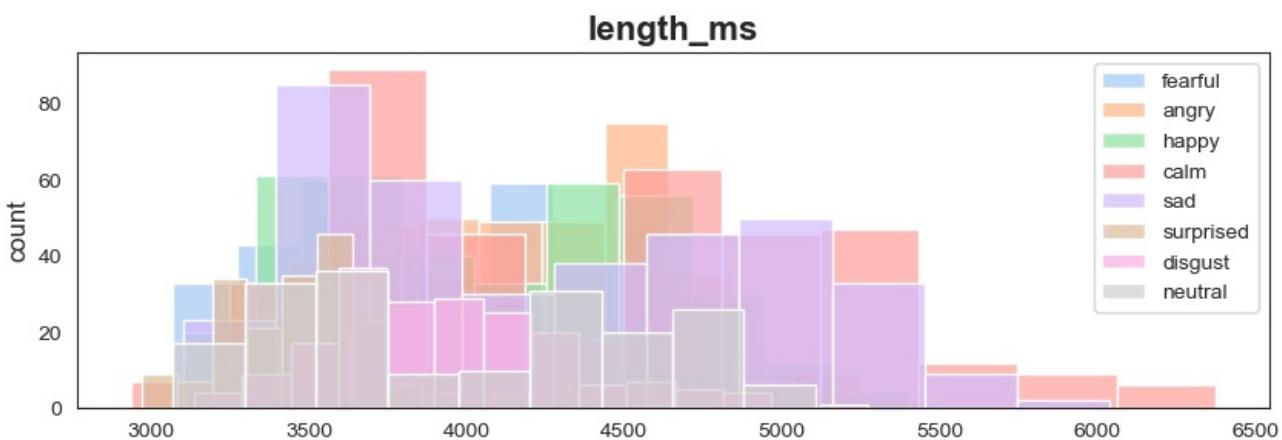
Vediamo che 'M' e 'F' sembrano avere distribuzioni **abbastanza distinte** rispetto ad alcuni degli attributi (ad esempio: 'stft_min', 'stft_std', 'zero_crossings_sum', 'mfcc_max').

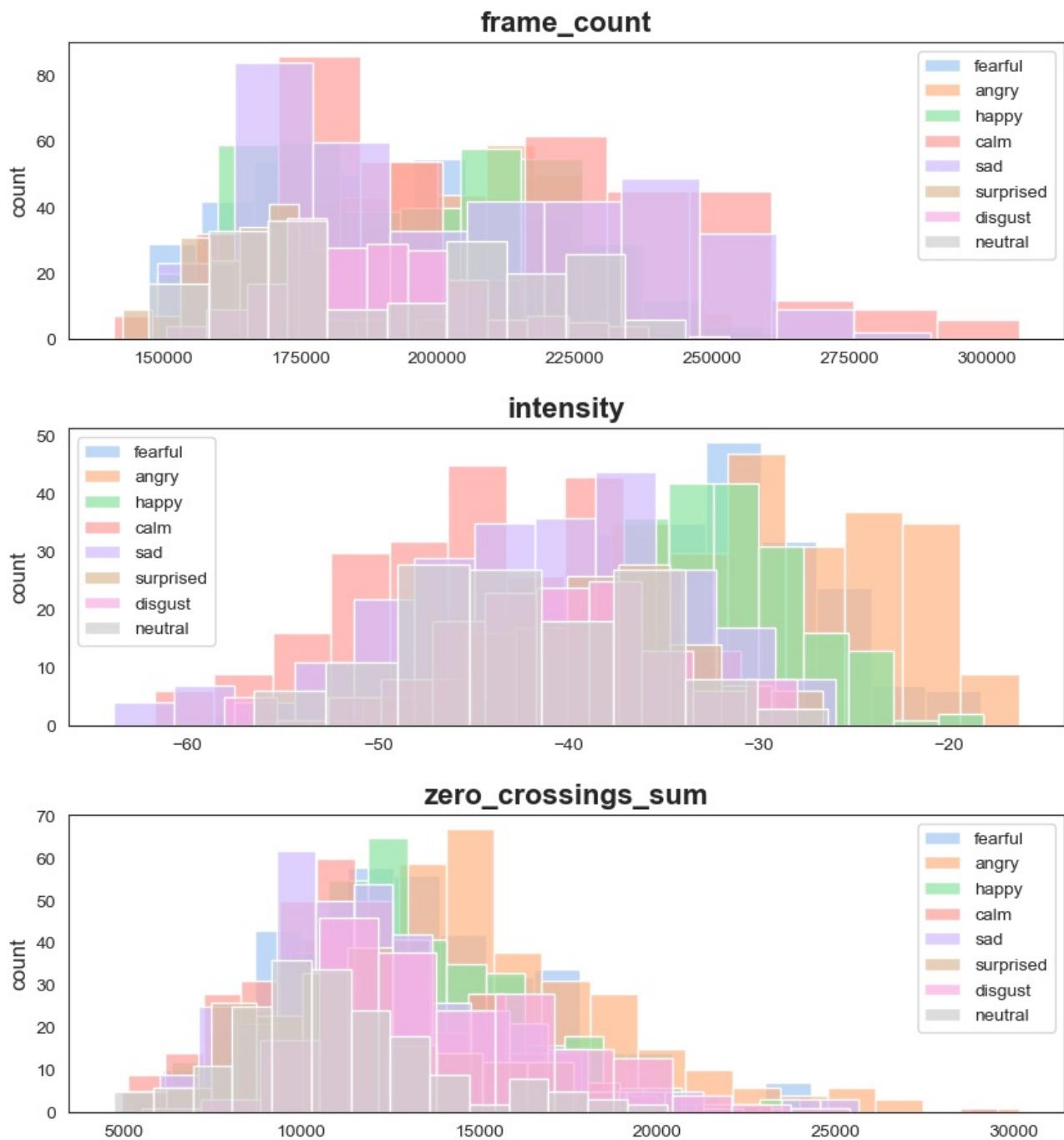
Emotion:

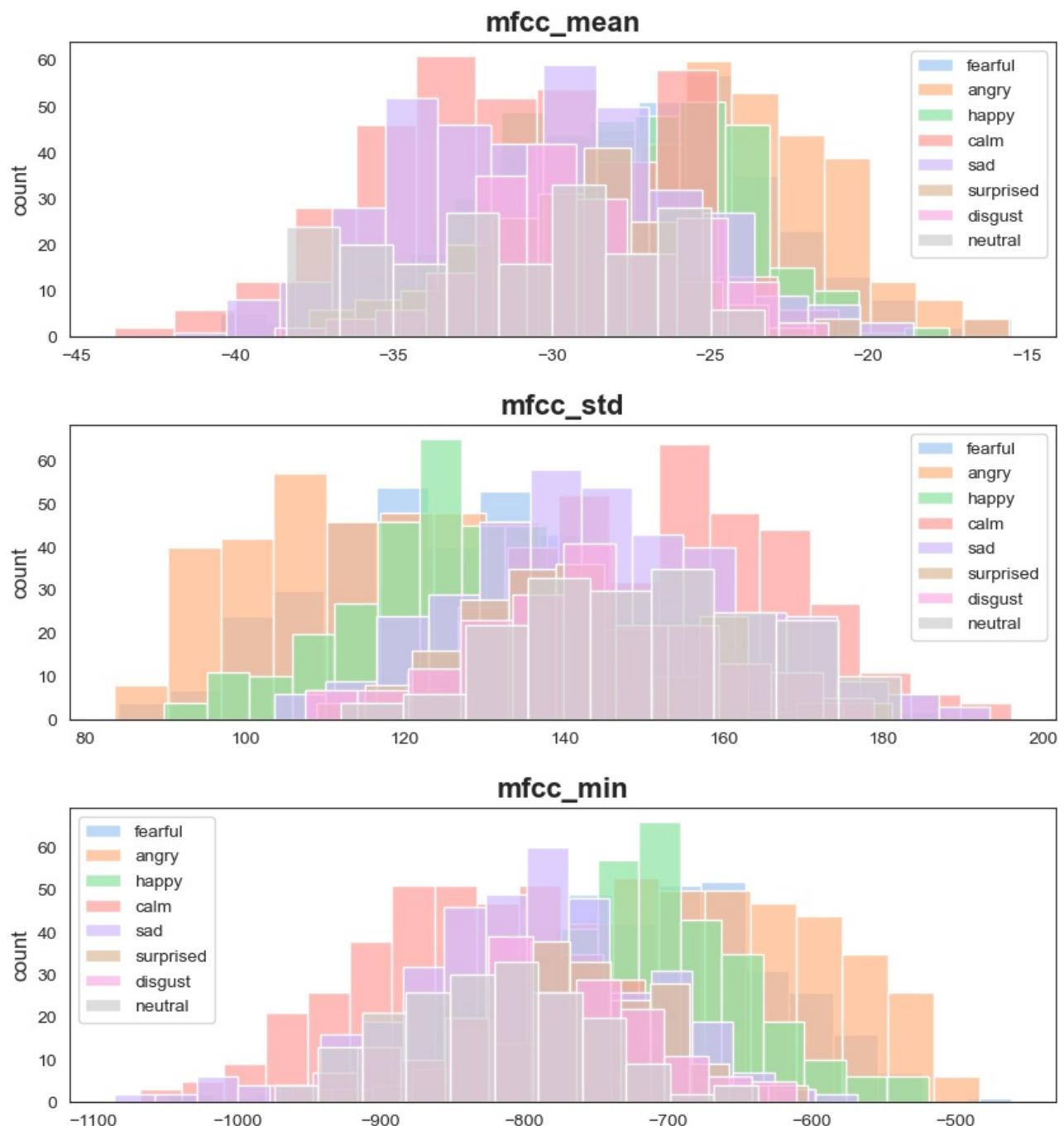
```
In [39]: fearful = df['emotion'] == 'fearful'
angry = df['emotion'] == 'angry'
happy = df['emotion'] == 'happy'
calm = df['emotion'] == 'calm'
sad = df['emotion'] == 'sad'
surprised = df['emotion'] == 'surprised'
disgust = df['emotion'] == 'disgust'
neutral = df['emotion'] == 'neutral'
```

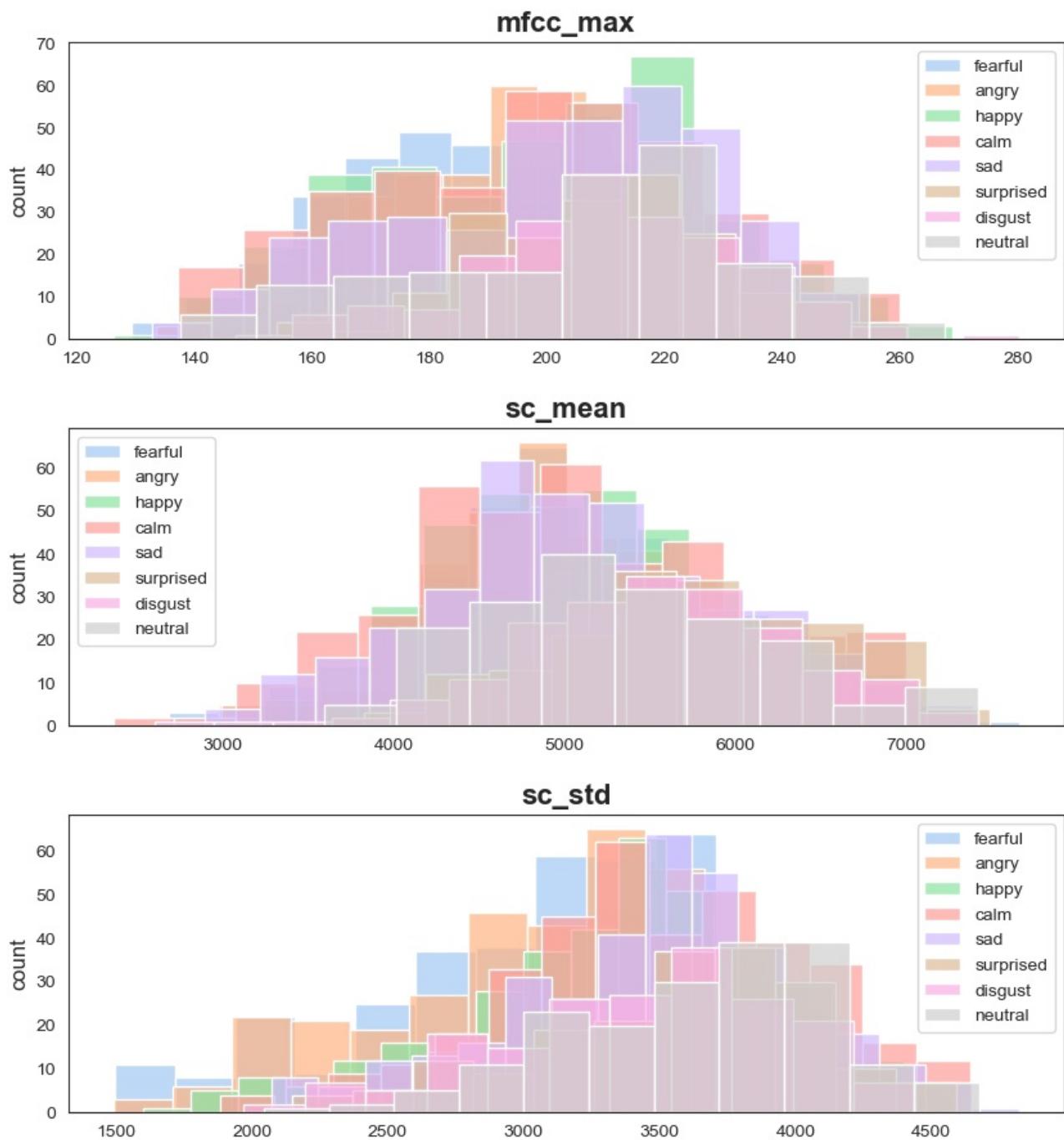
```
In [40]: pastel_colors = sns.color_palette("pastel", 8)

for col in numerical_columns:
    if col not in ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max']:
        plt.figure(figsize=(10, 3))
        sns.histplot(df[fearful][col], label='fearful', alpha=0.7, color=pastel_colors[0])
        sns.histplot(df[angry][col], label='angry', alpha=0.7, color=pastel_colors[1])
        sns.histplot(df[happy][col], label='happy', alpha=0.7, color=pastel_colors[2])
        sns.histplot(df[calm][col], label='calm', alpha=0.7, color=pastel_colors[3])
        sns.histplot(df[sad][col], label='sad', alpha=0.7, color=pastel_colors[4])
        sns.histplot(df[surprised][col], label='surprised', alpha=0.7, color=pastel_colors[5])
        sns.histplot(df[disgust][col], label='disgust', alpha=0.7, color=pastel_colors[6])
        sns.histplot(df[neutral][col], label='neutral', alpha=0.7, color=pastel_colors[7])
        plt.title(col, fontsize=16, fontweight='bold')
        plt.ylabel('count', fontsize=12)
        plt.xlabel('')
        plt.legend(facecolor = 'white')
        plt.show()
```

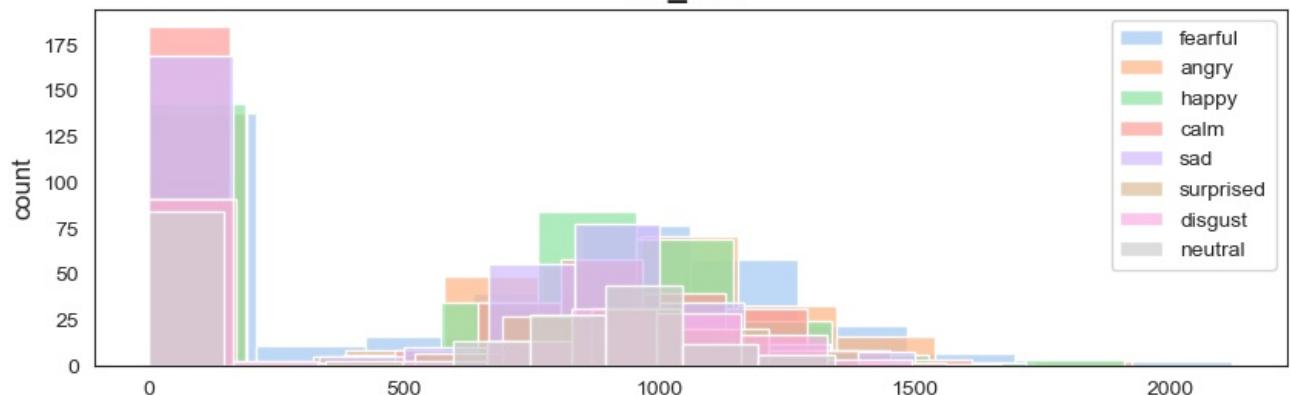




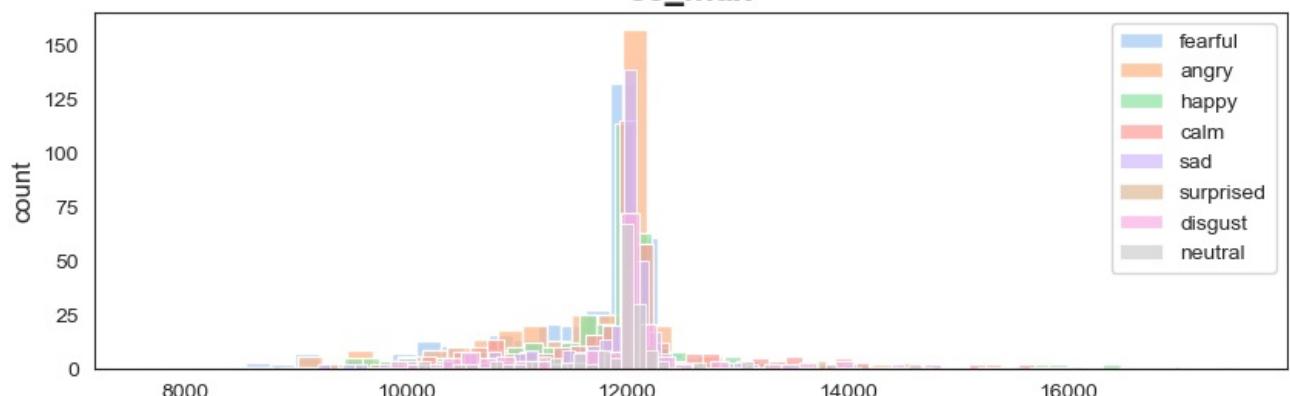




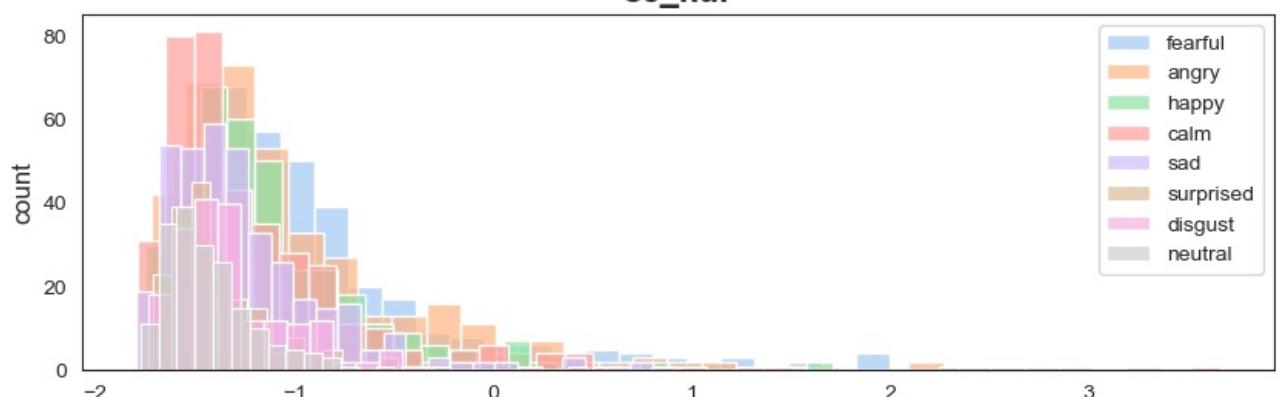
sc_min

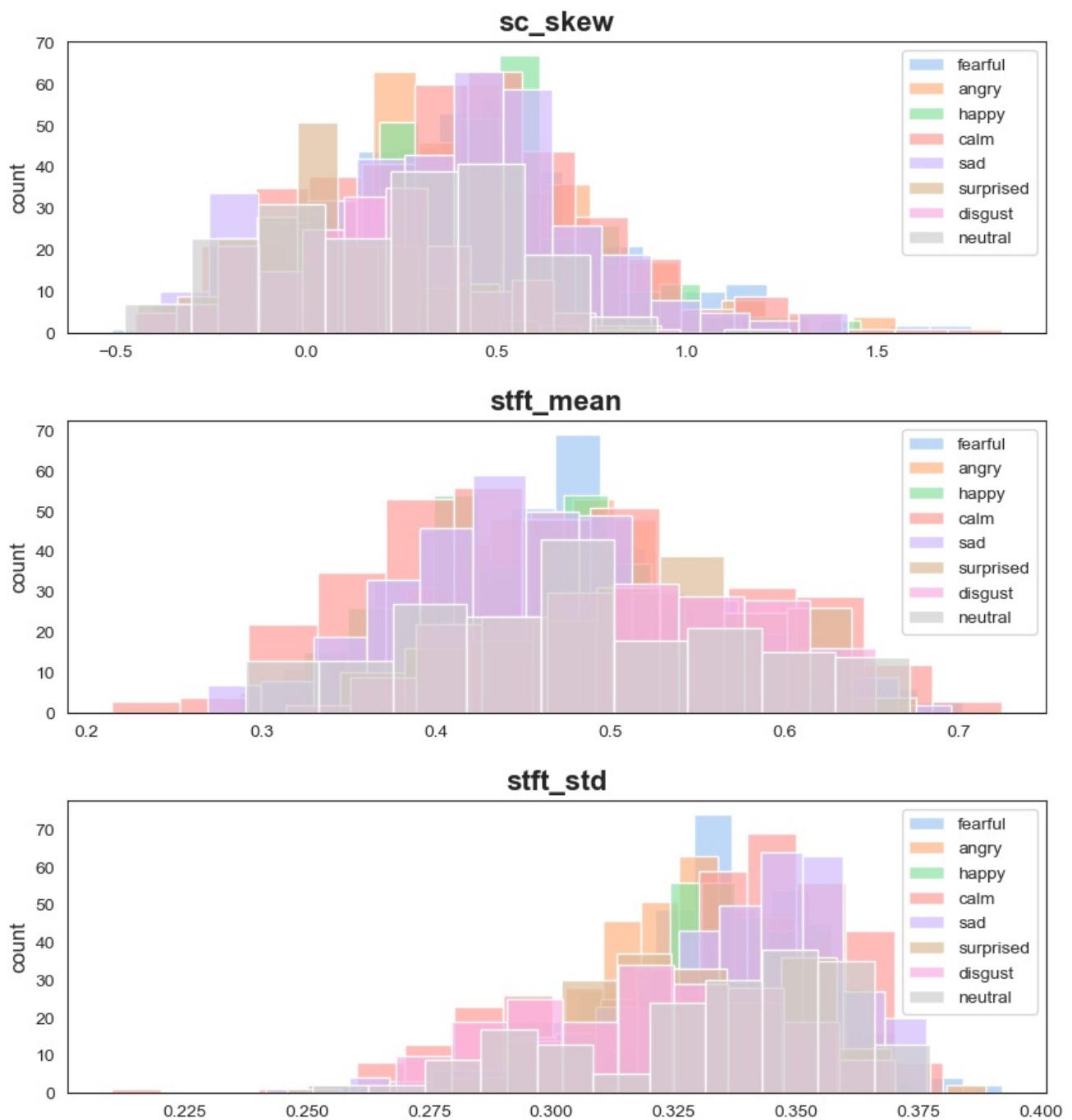


sc_max

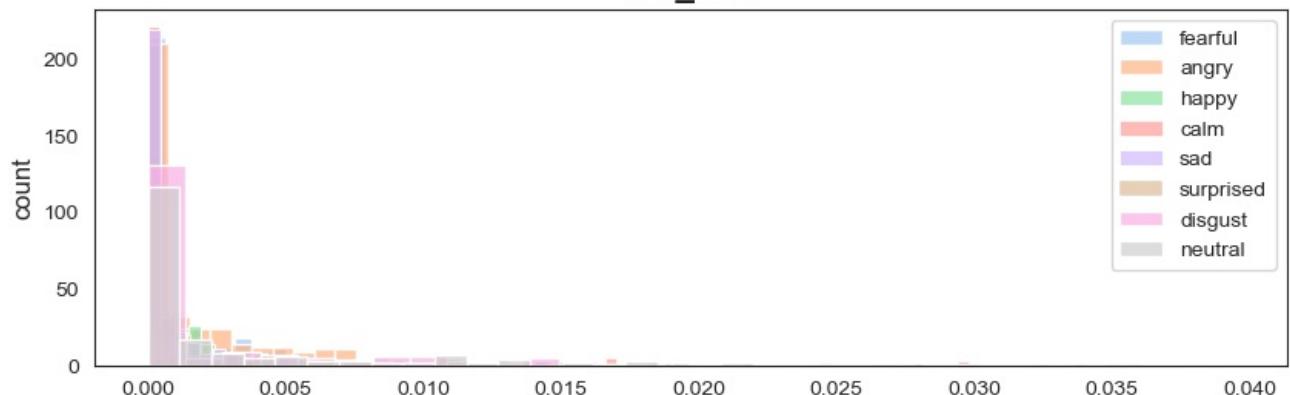


sc_kur

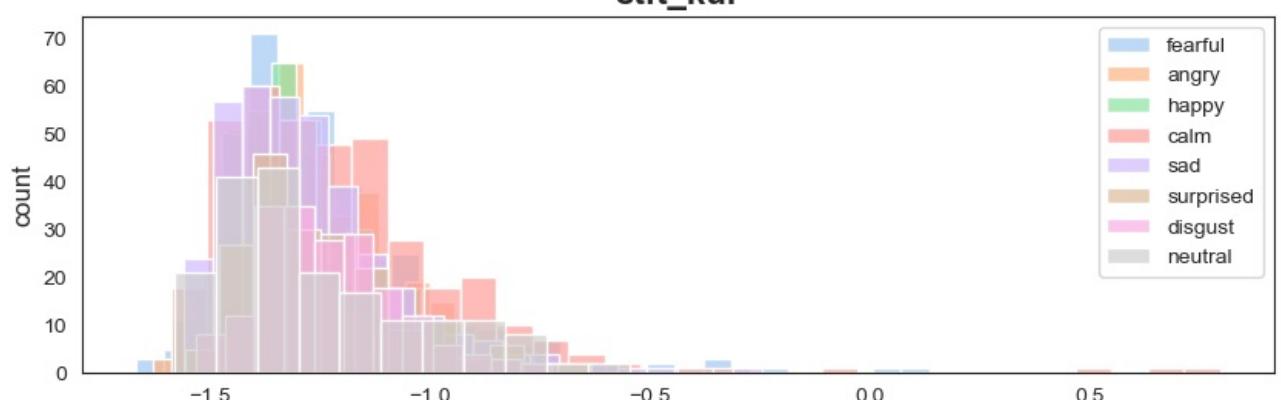




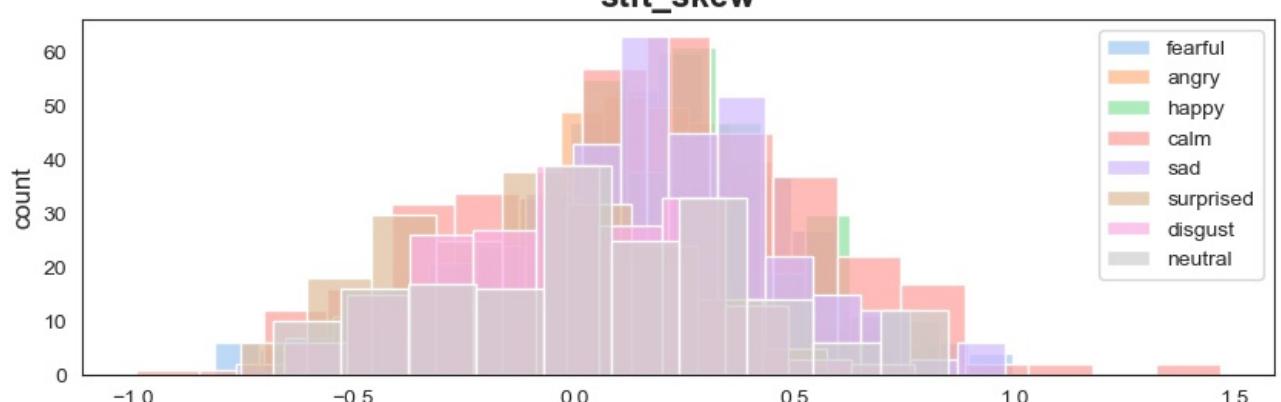
stft_min

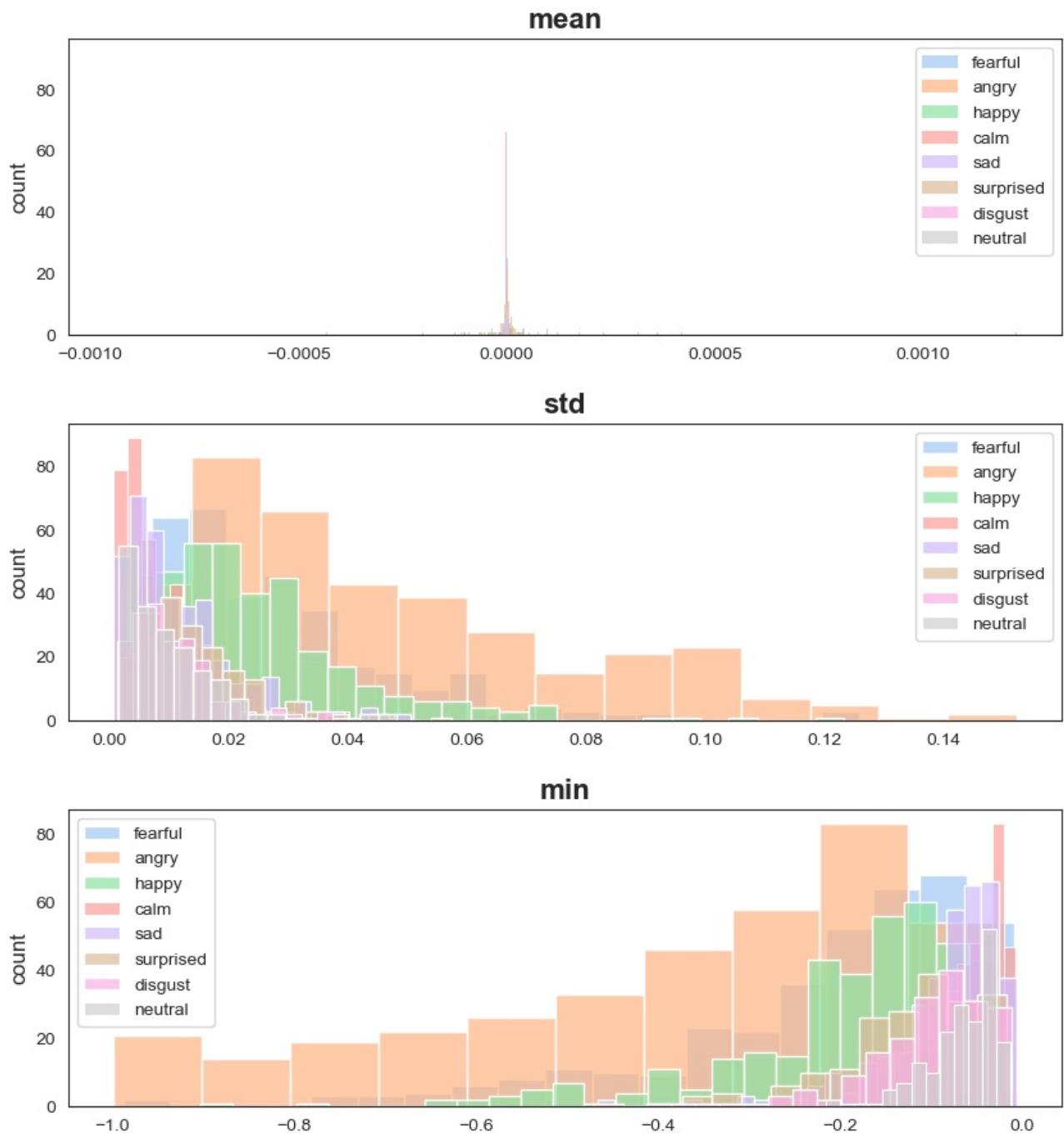


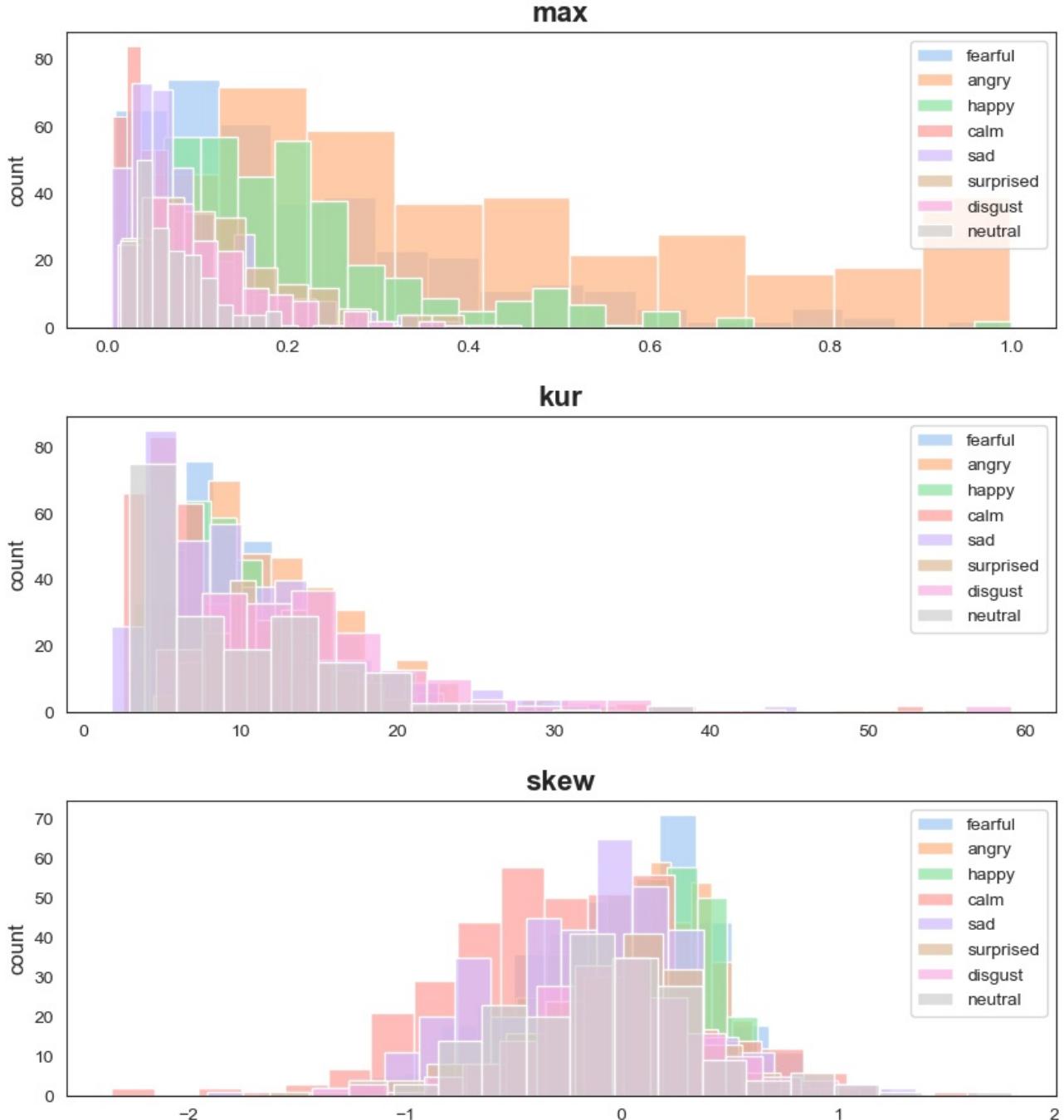
stft_kur



stft_skew





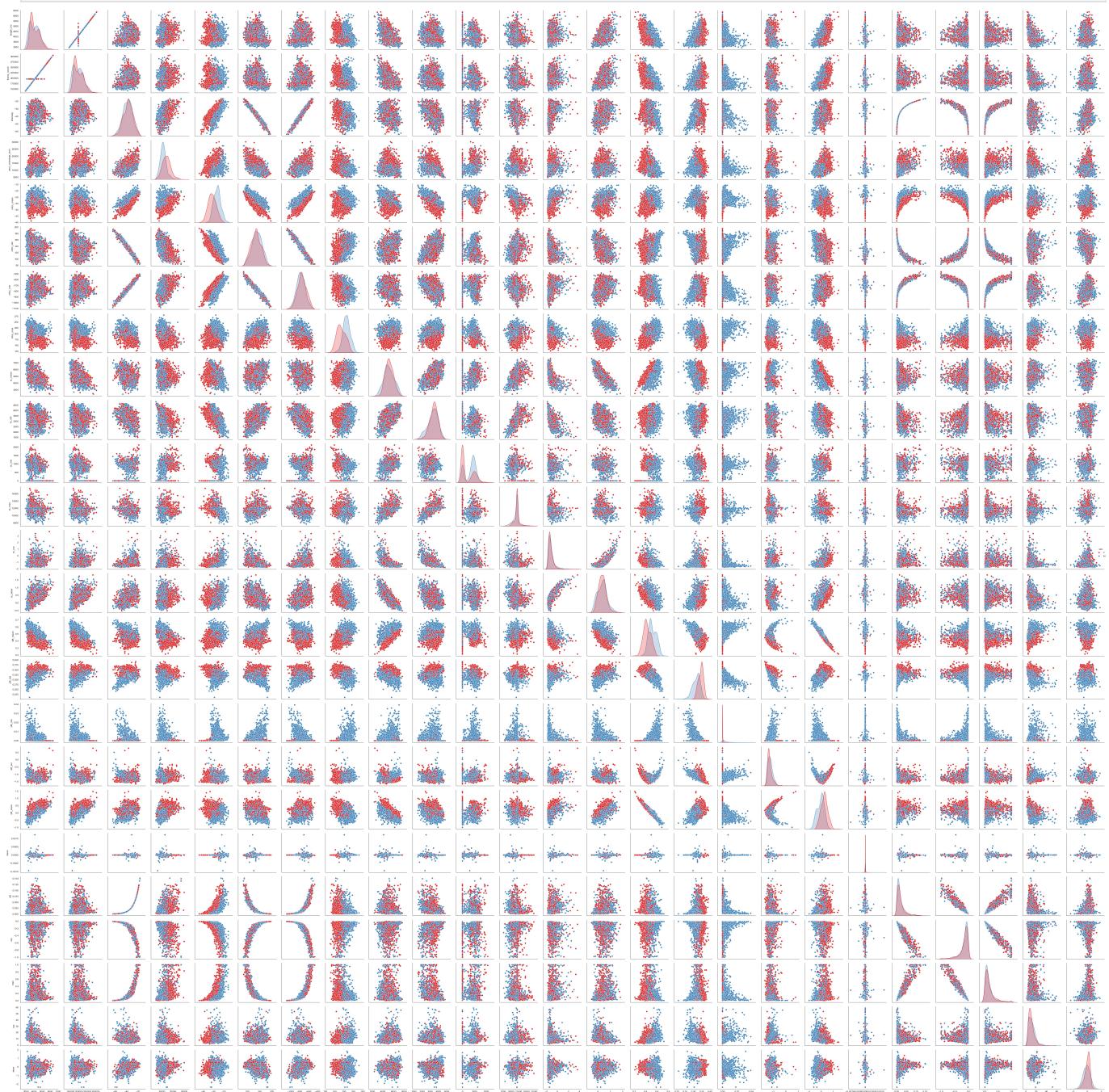


Le distribuzioni sono prevedibilmente **meno nette** nel caso delle emozioni, anche se alcuni sentimenti semanticamente opposti – come 'angry' e 'calm' – sembrano distinguersi in alcune delle variabili (ad esempio: 'std', 'min', 'max').

Queste considerazioni sulle interazioni di 'sex' ed 'emotion' con le variabili continue risultano evidenti anche attraverso la visualizzazione generale fornita dalle **scatter matrix**. Da esse possiamo inoltre notare che esistono **forti legami** tra molti degli attributi; alcuni di questi, come ad esempio quello tra 'intensity' e 'std', verranno esaminati in dettaglio nel seguito.

```
In [41]: num_col_list = [c for c in numerical_columns if c not in
                  ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max']]
sns.pairplot(df, palette='Set1', vars=num_col_list, hue='sex')
```

```
plt.tight_layout()  
plt.show()
```



```
In [42]: num_col_list = [c for c in numerical_columns if c not in  
                  ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max']]  
  
sns.pairplot(df, palette='Set1', vars=num_col_list, hue='emotion')  
plt.tight_layout()  
plt.show()
```



Ed attraverso un **plot a coordinate parallele** ('emotion'):

```
In [43]: par_col_list = [c for c in numerical_columns if c not in
                  ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max', 'mean']]

emotions_list = list(df['emotion'].unique())

colormap = plt.colormaps['Set1']
colors_list = [colormap(i) for i in range(len(emotions_list))]

colors_dict = dict(zip(emotions_list, colors_list))
```

```
In [44]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(df[par_col_list].values)

df_numeric_scaled = pd.DataFrame(X_scaled, columns = par_col_list, index = df.index)
df_numeric_scaled['emotion'] = df['emotion']
```

```
In [45]: plt.figure(figsize = (18, 8))

x = range(len(par_col_list))

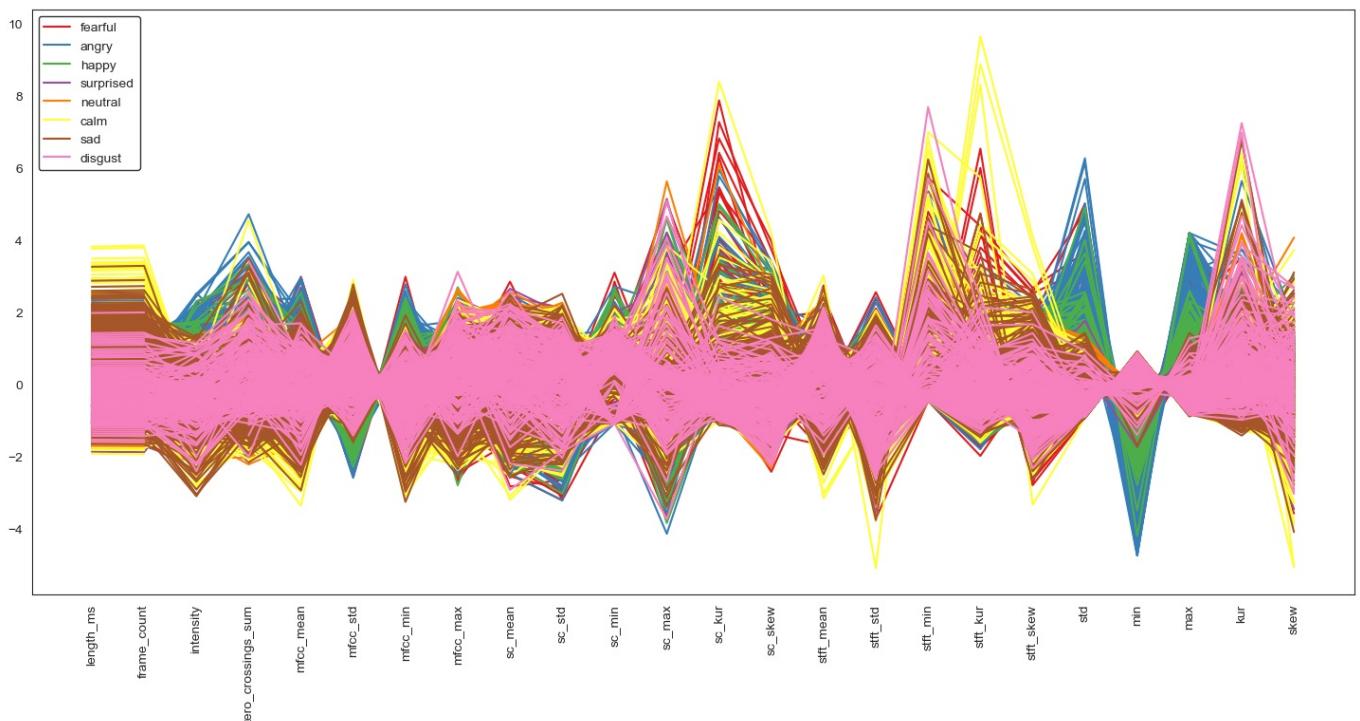
for j in emotions_list:
    df_temp = df_numeric_scaled[df_numeric_scaled['emotion'] == j]
    for i in range(df_temp.shape[0]):
        if i == 0:
            plt.plot( x, df_temp[par_col_list].iloc[i].values, c = colors_dict[j], label = j )
        else:
```

```

plt.plot( x, df_temp[par_col_list].iloc[i].values, c = colors_dict[j] )

plt.xticks(x, par_col_list, rotation = 90)
plt.legend(facecolor = 'white', edgecolor = 'black')
plt.show()

```



Molte delle variabili numeriche rappresentano **famiglie di metriche acustiche**: mfcc (Mel-Frequency Cepstral Coefficients), sc (spectral centroid), stft (stft chromagram); le statistiche riportate nel dataset ne mostrano i minimi, i massimi, le medie, etc. registrati nel corso dei vari sample. Raggruppiamo perciò queste metriche, e ne mostriamo le interazioni con le emozioni tramite dei **box plot**:

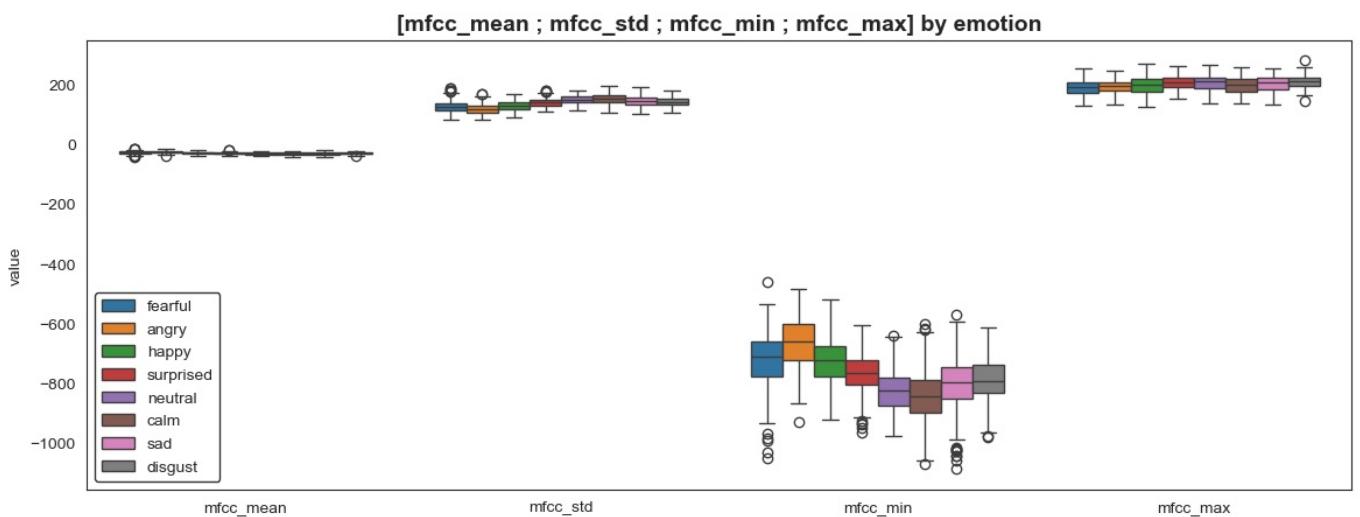
```

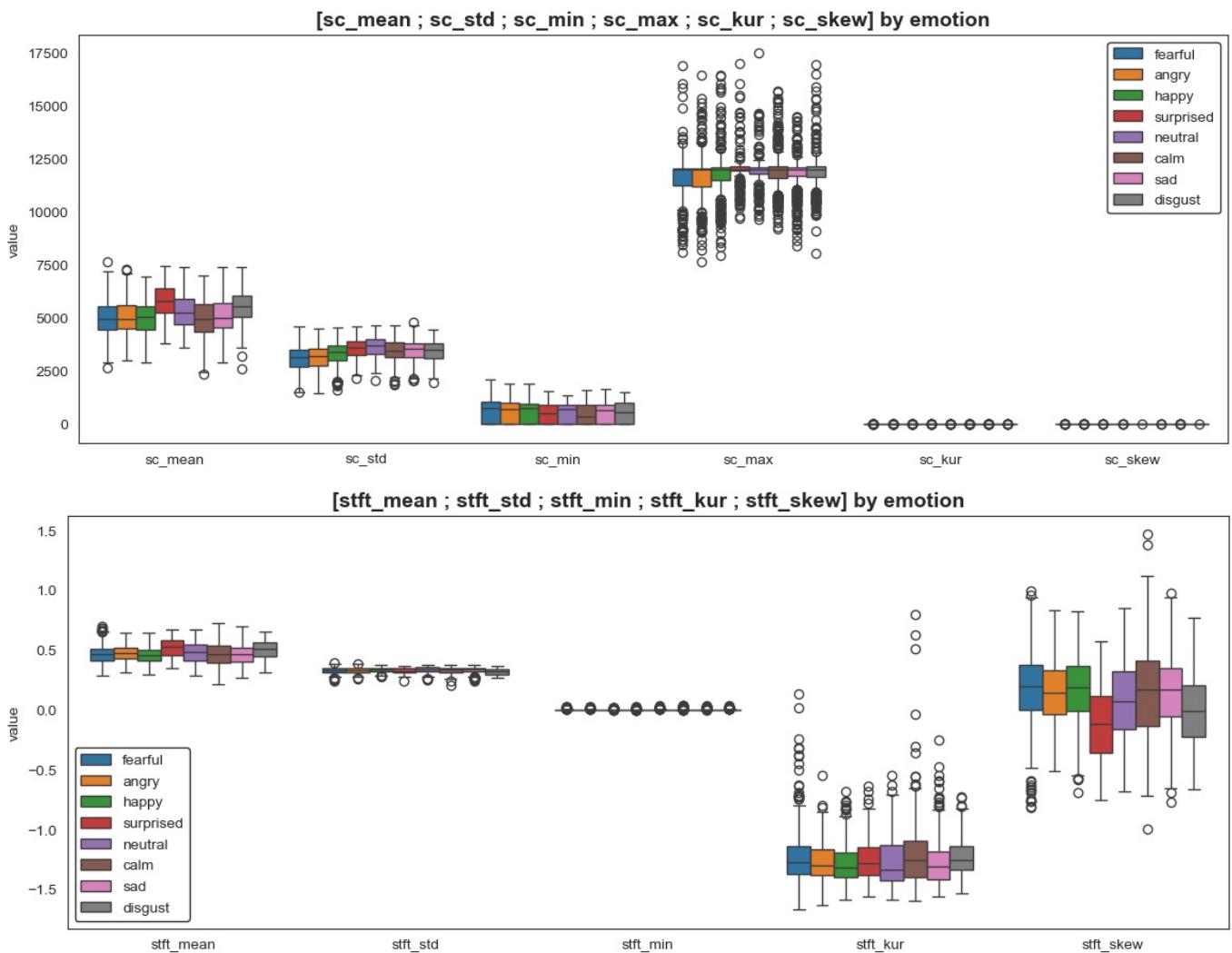
In [46]: target = 'emotion'

mfcc_metrics = ['mfcc_mean', 'mfcc_std', 'mfcc_min', 'mfcc_max']
sc_metrics = ['sc_mean', 'sc_std', 'sc_min', 'sc_max', 'sc_kur', 'sc_skew']
stft_metrics = ['stft_mean', 'stft_std', 'stft_min', 'stft_kur', 'stft_skew']

for metrics in [mfcc_metrics, sc_metrics, stft_metrics]:
    fig, ax = plt.subplots(figsize = (14, 5))
    df_temp = df[metrics + [target]].melt(id_vars = target)
    sns.boxplot(x = 'variable', y = 'value', hue = target, data = df_temp, ax = ax)
    ax.set_title('[' + ' ; '.join(metrics) + '] by ' + target, fontsize = 14, fontweight='bold')
    plt.legend(facecolor = 'white', edgecolor = 'black', fontsize = 10)
    plt.xlabel('')
    plt.show()

```





Come visualizzazione conclusiva di questa sezione, usiamo dei **violin plot** per mostrare le relazioni tra 'emotion' e 'sex', rispetto a tutte le variabili continue:

```
In [47]: hue = 'sex'
target = 'emotion'

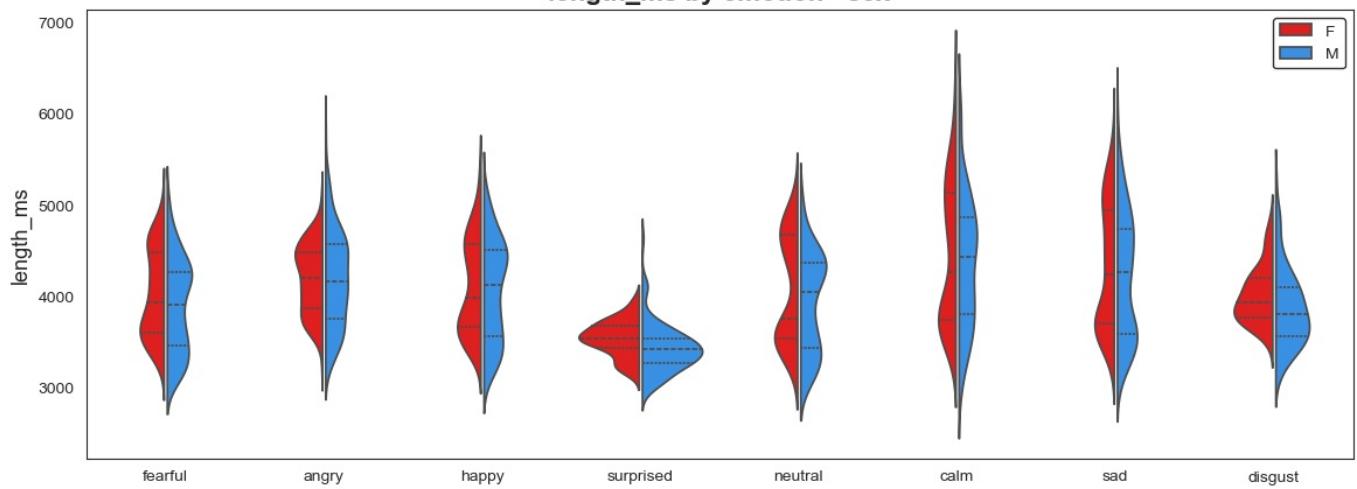
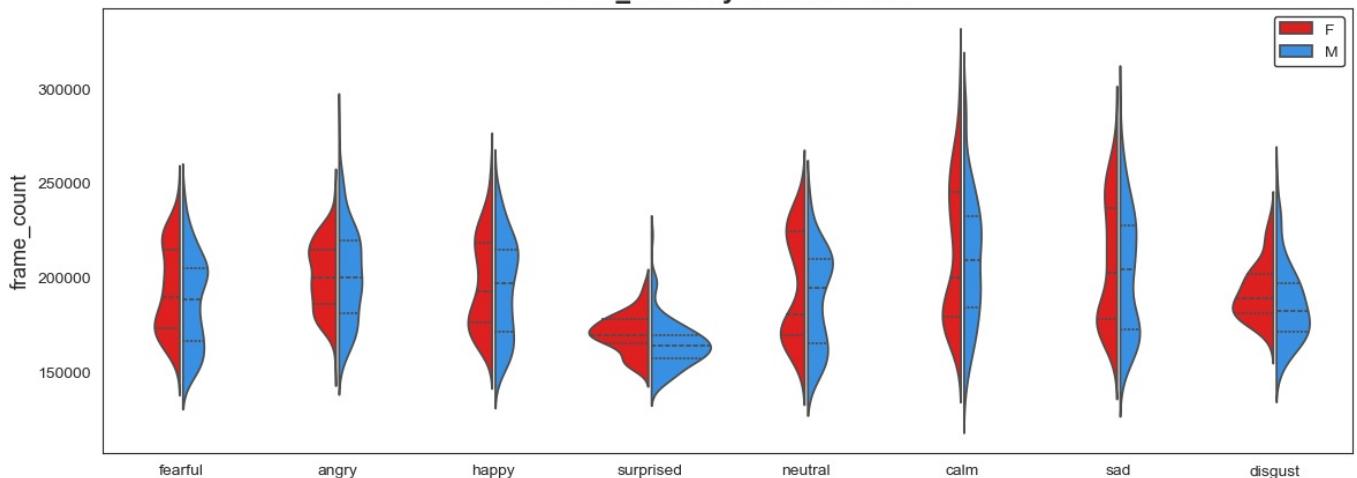
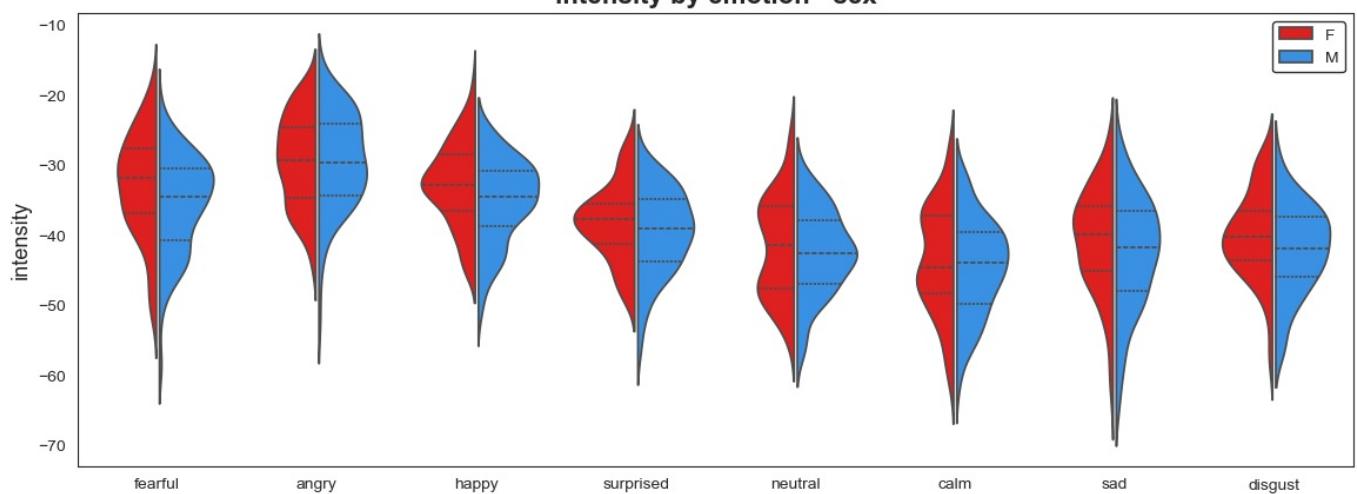
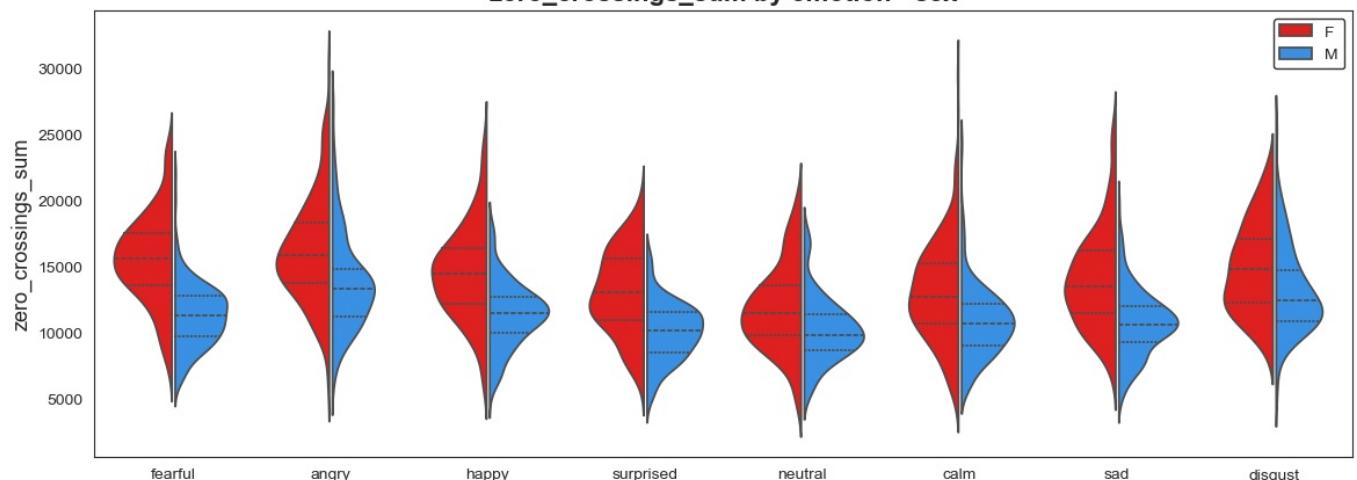
palette = {'M': 'dodgerblue', 'F': 'red'}

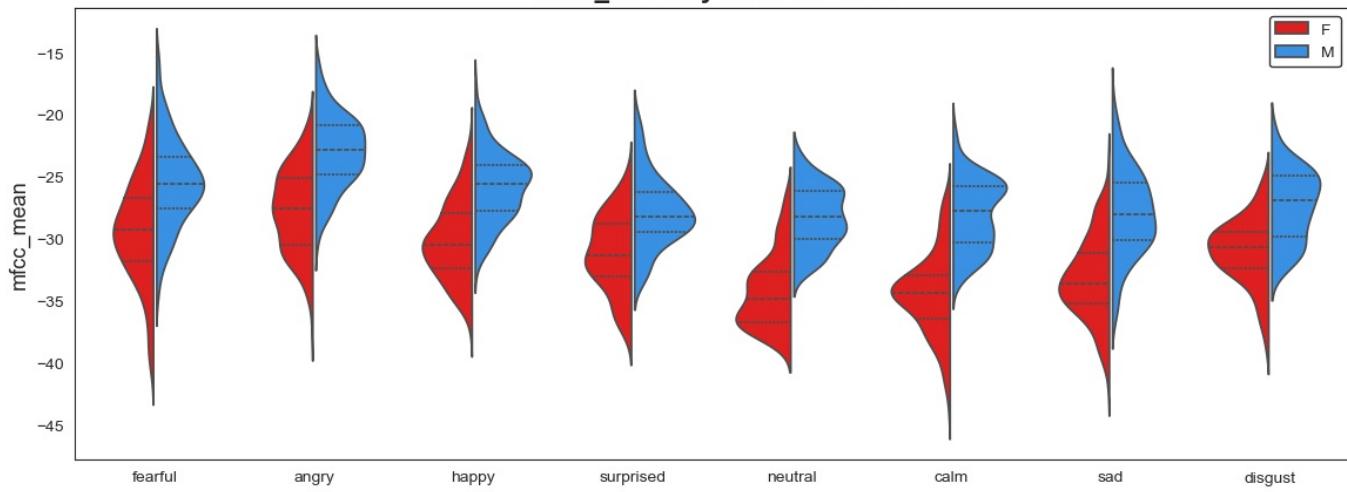
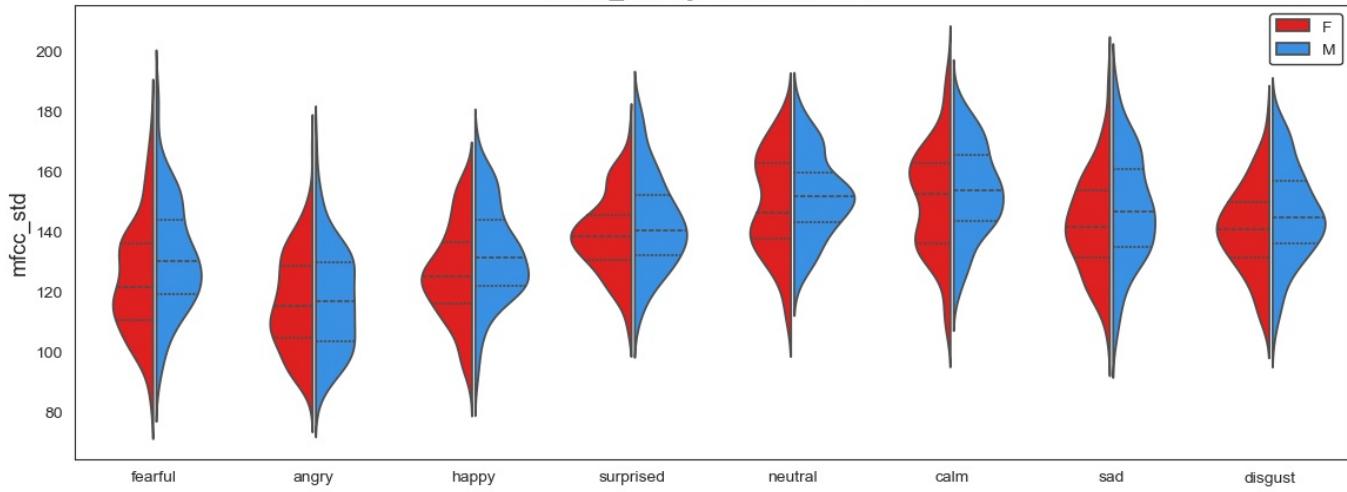
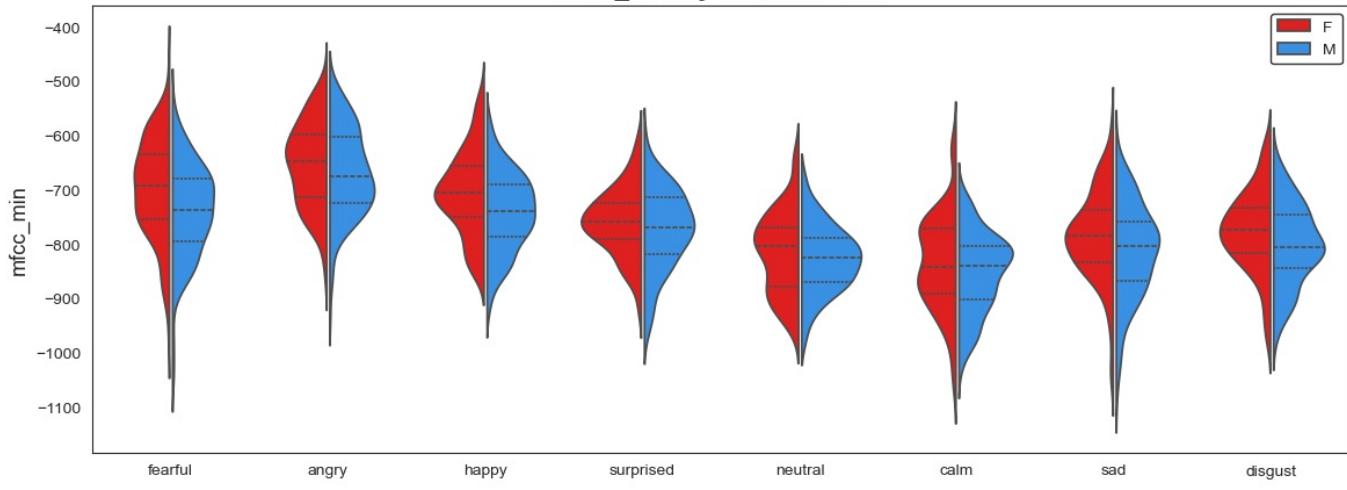
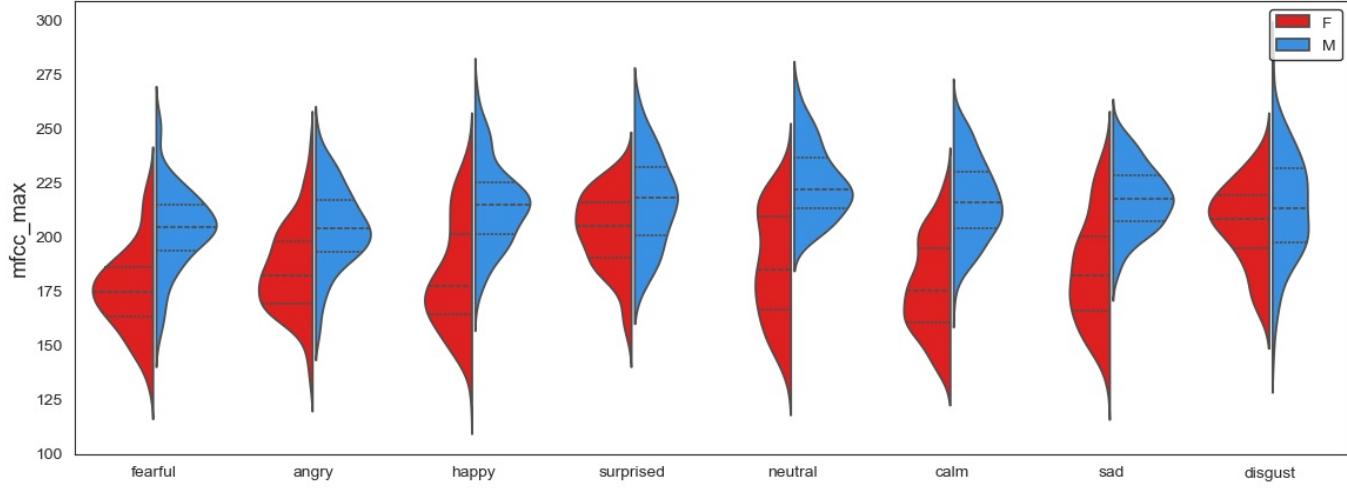
for col in num_col_list:

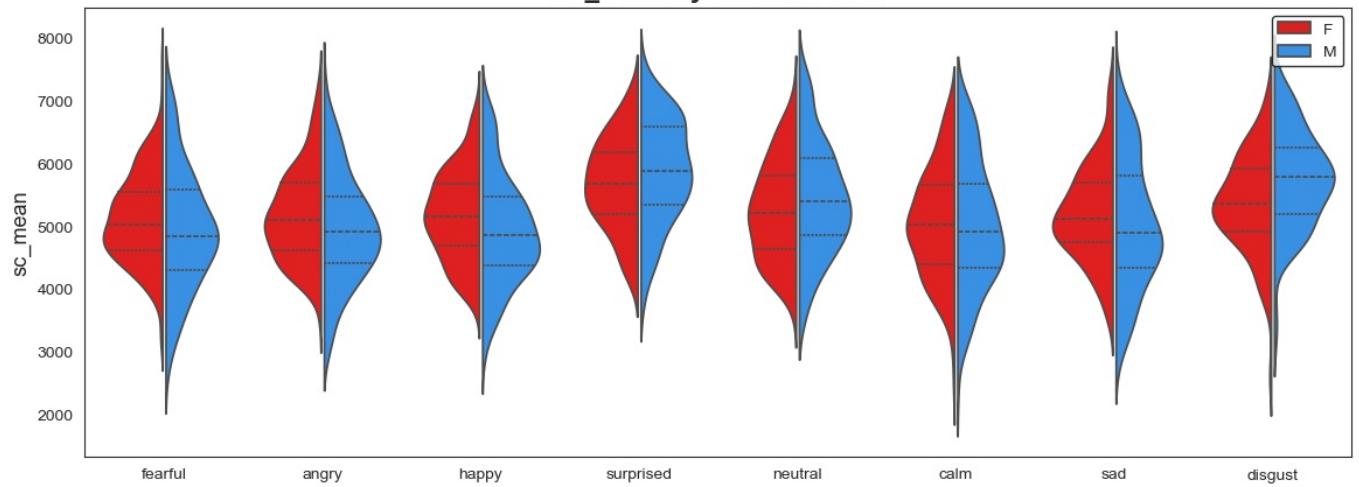
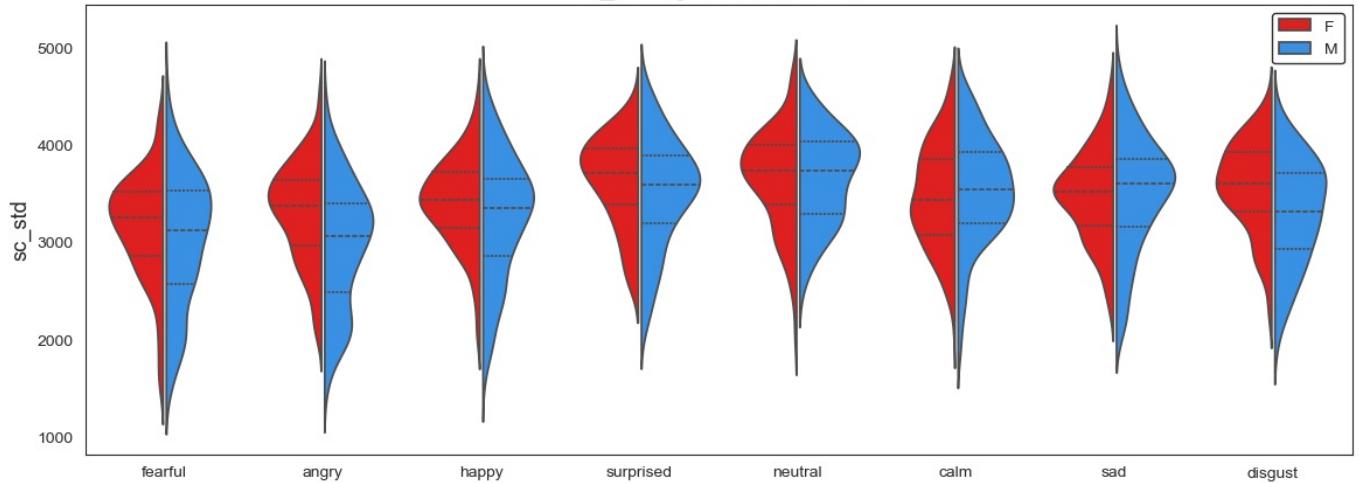
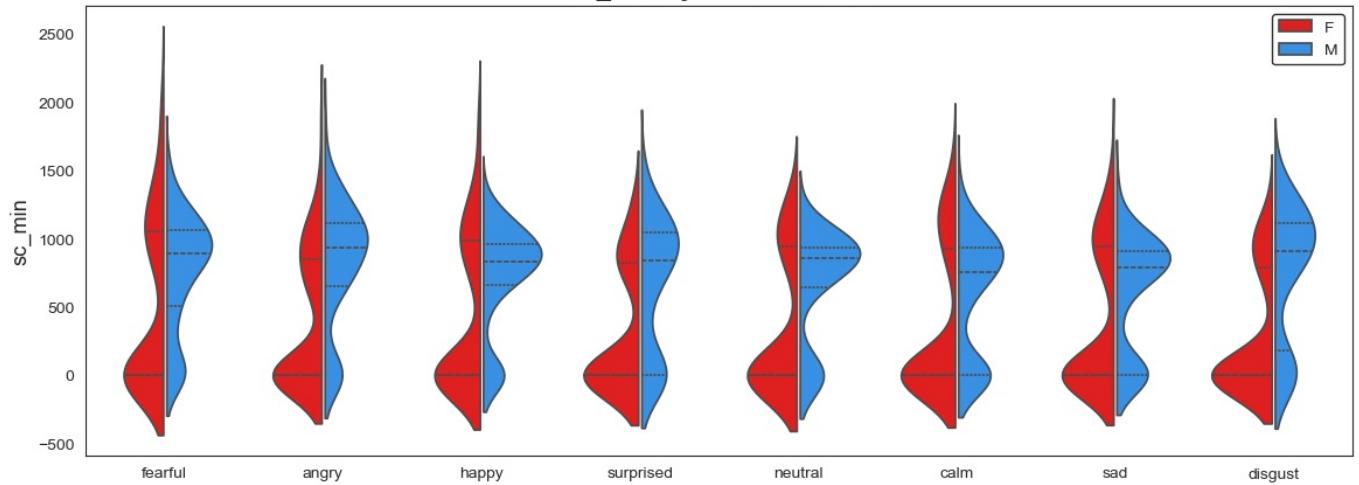
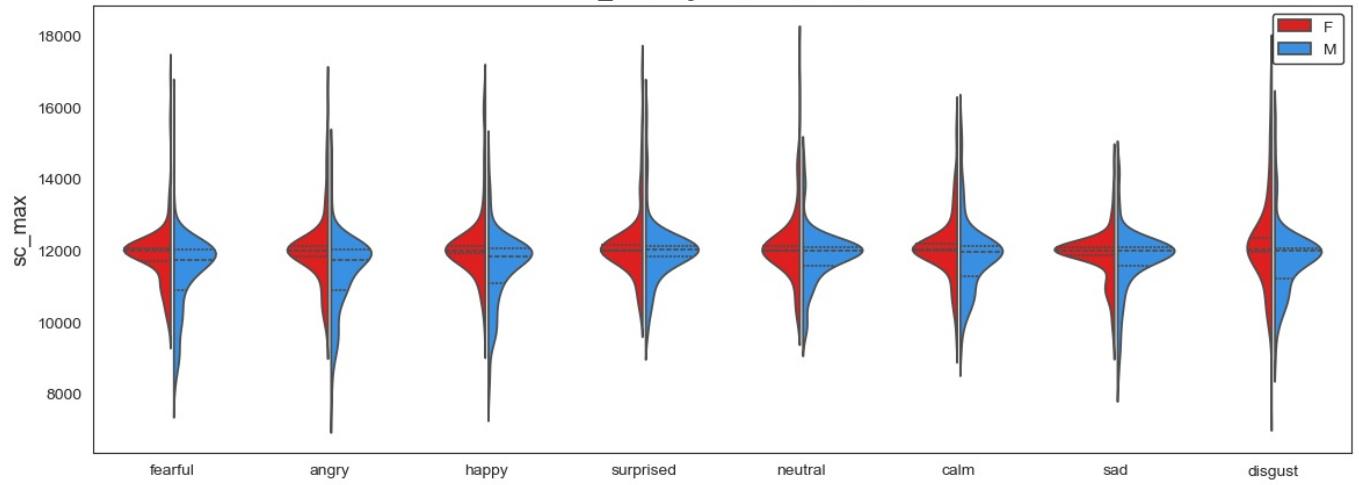
    fig, ax = plt.subplots(figsize = (14, 5))

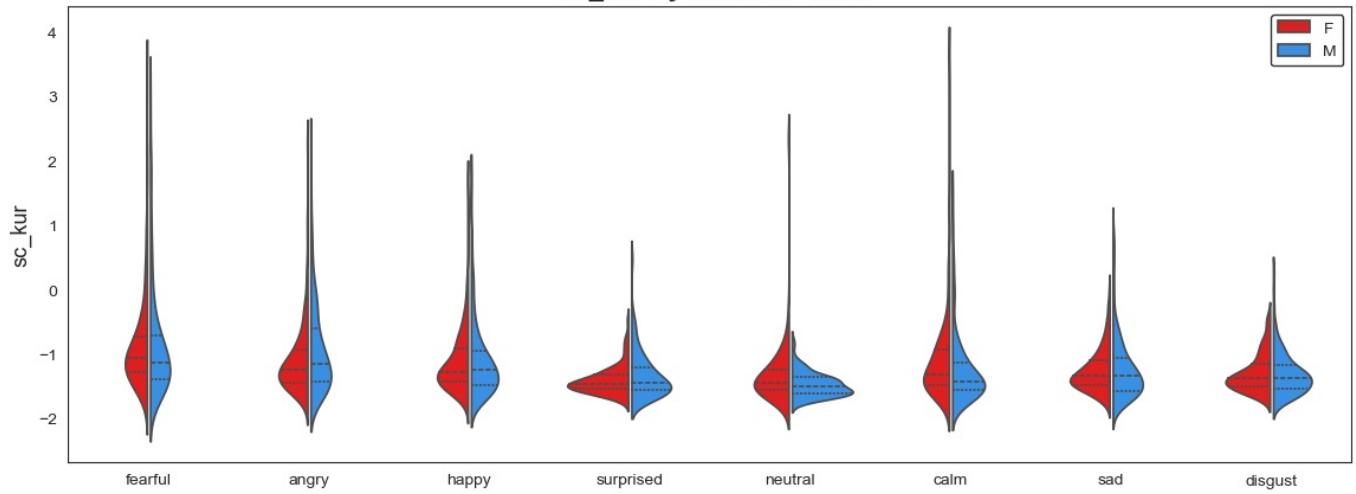
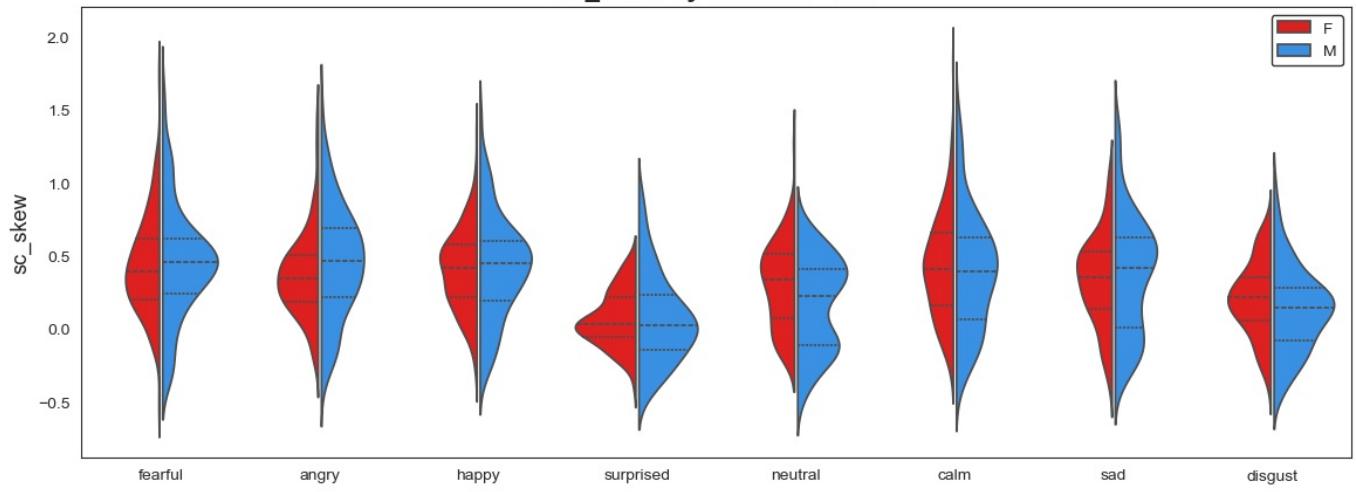
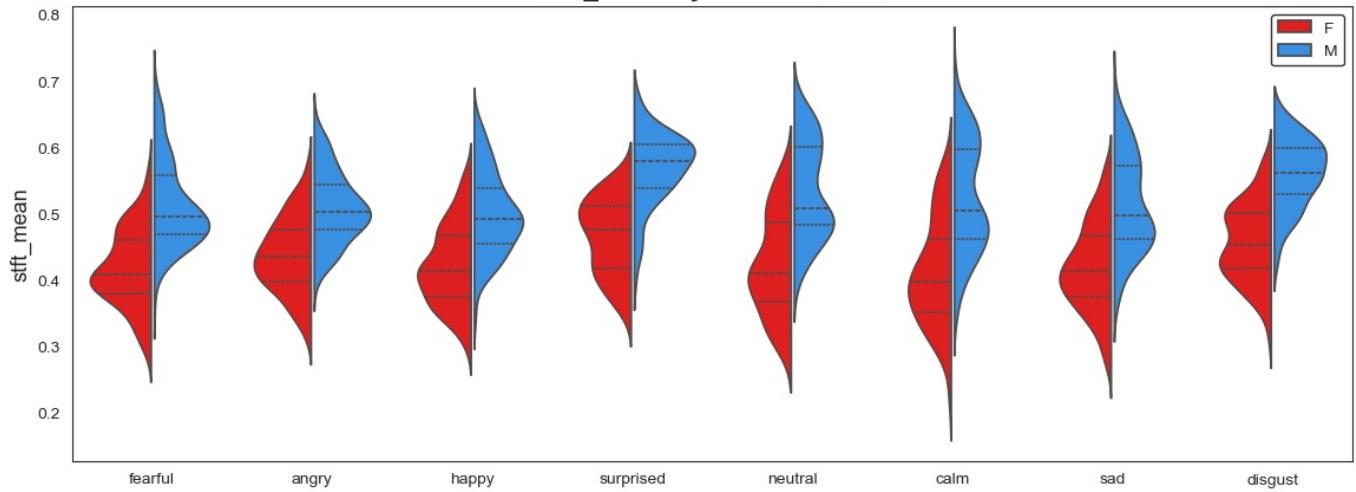
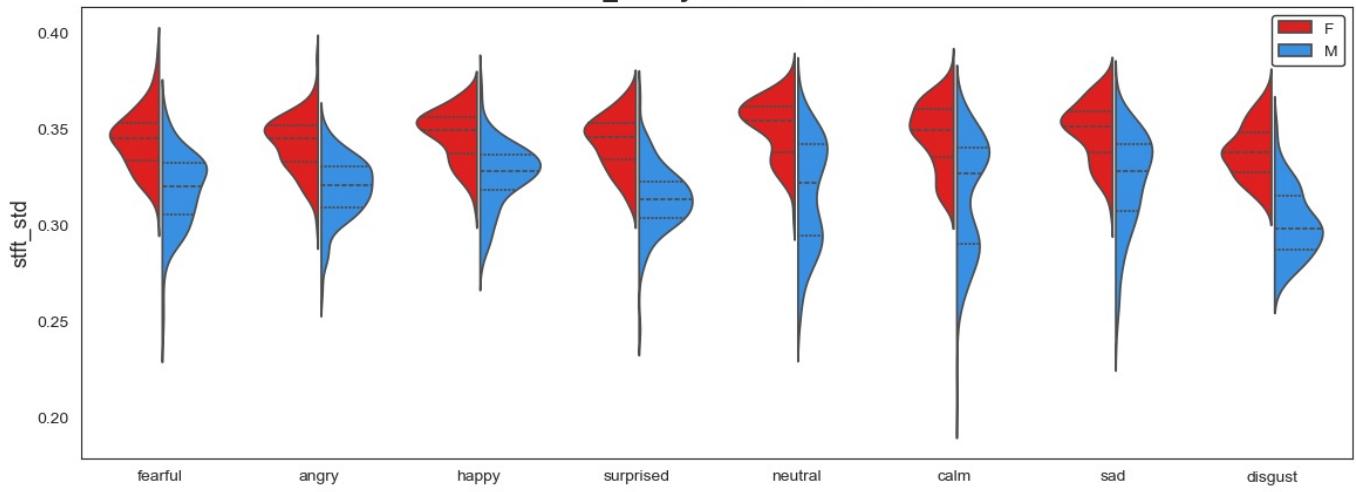
    df_temp = df[[col, target, hue]].melt(id_vars = [target, hue])

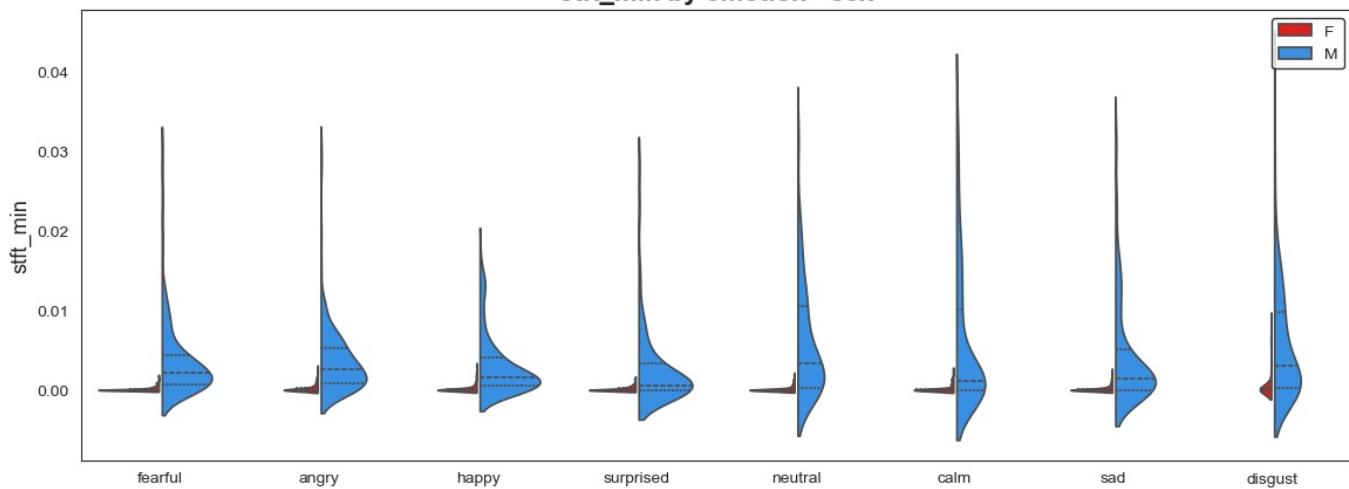
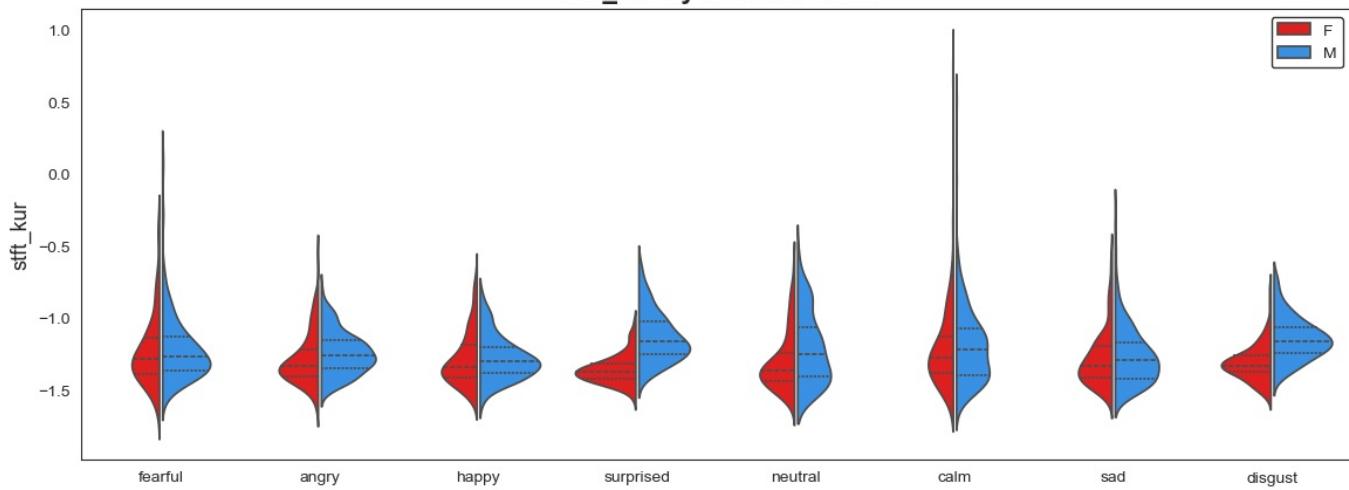
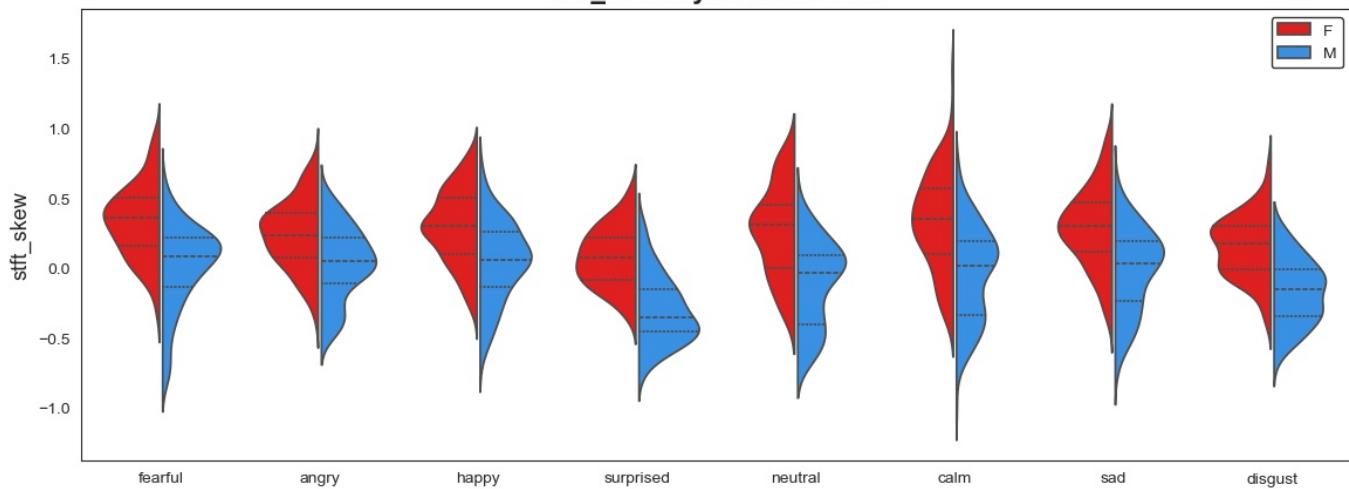
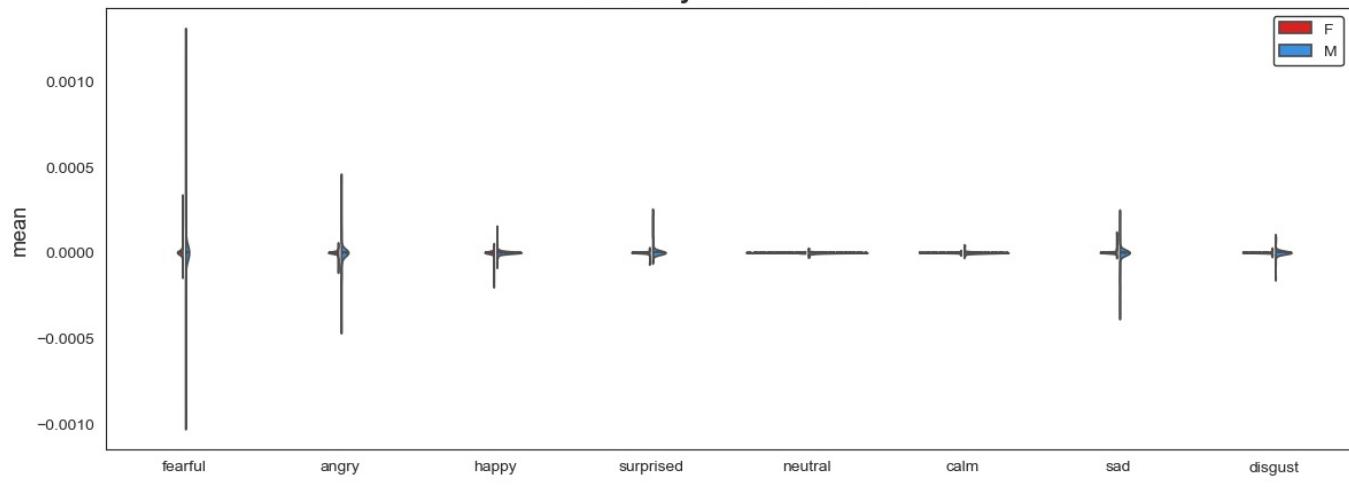
    sns.violinplot(x = target, y = 'value', hue = hue, data = df_temp, split = True,
                    inner = 'quart', gap = 0.05, palette=palette, ax = ax)
    ax.set_title(f'{col} by {target} - {hue}', fontsize = 16, fontweight='bold')
    plt.legend(facecolor = 'white', edgecolor = 'black', fontsize = 10)
    ax.set_ylabel(col, fontsize = 13)
    plt.xlabel('')
    plt.show()
```

length_ms by emotion - sex**frame_count by emotion - sex****intensity by emotion - sex****zero_crossings_sum by emotion - sex**

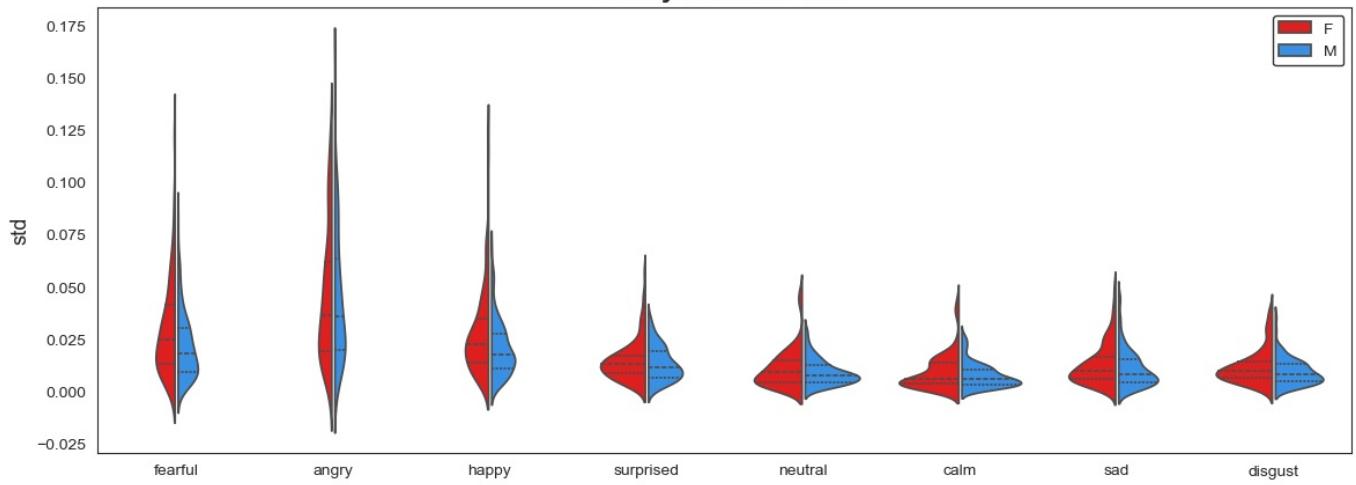
mfcc_mean by emotion - sex**mfcc_std by emotion - sex****mfcc_min by emotion - sex****mfcc_max by emotion - sex**

sc_mean by emotion - sex**sc_std by emotion - sex****sc_min by emotion - sex****sc_max by emotion - sex**

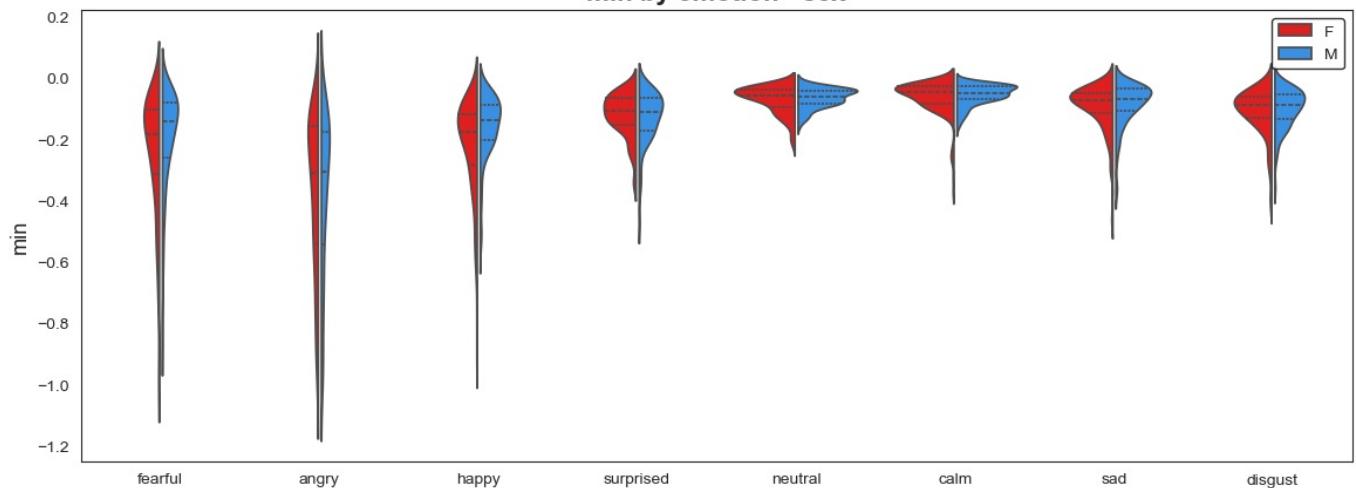
sc_kur by emotion - sex**sc_skew by emotion - sex****stft_mean by emotion - sex****stft_std by emotion - sex**

stft_min by emotion - sex**stft_kur by emotion - sex****stft_skew by emotion - sex****mean by emotion - sex**

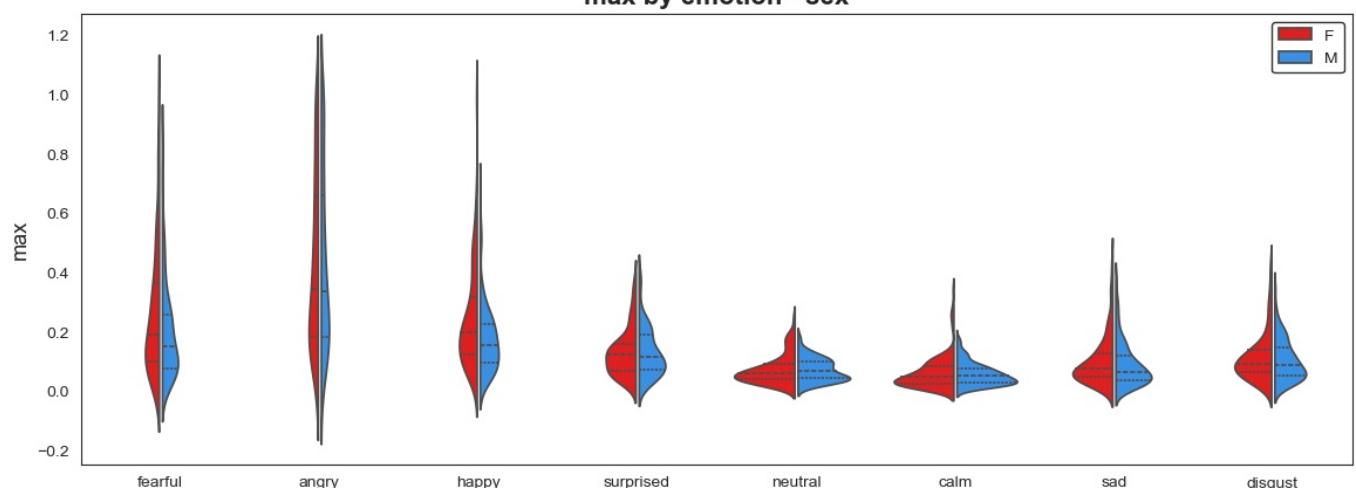
std by emotion - sex



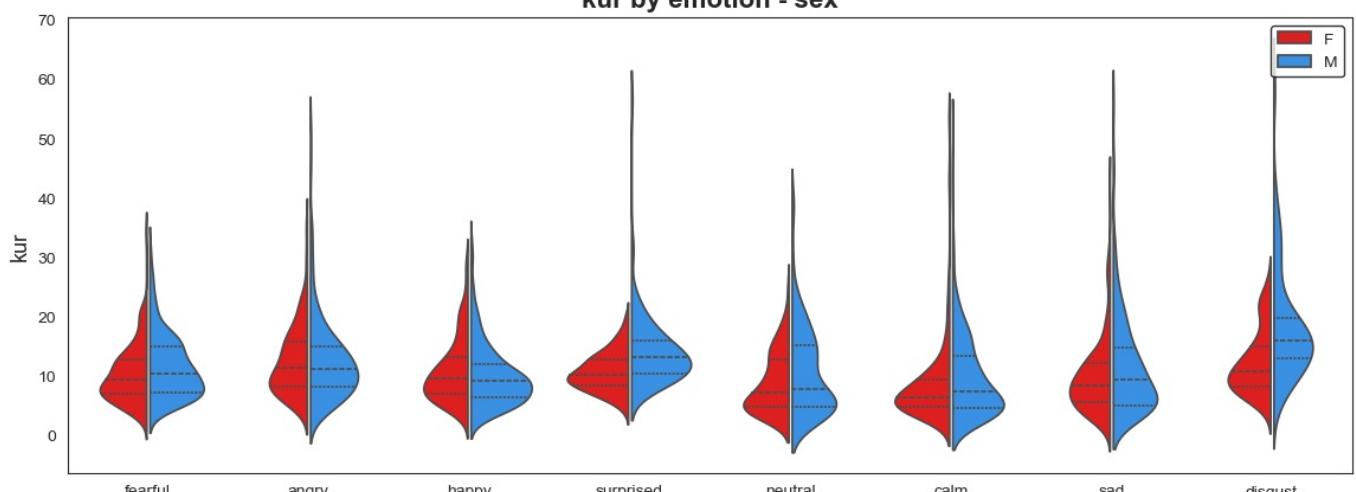
min by emotion - sex

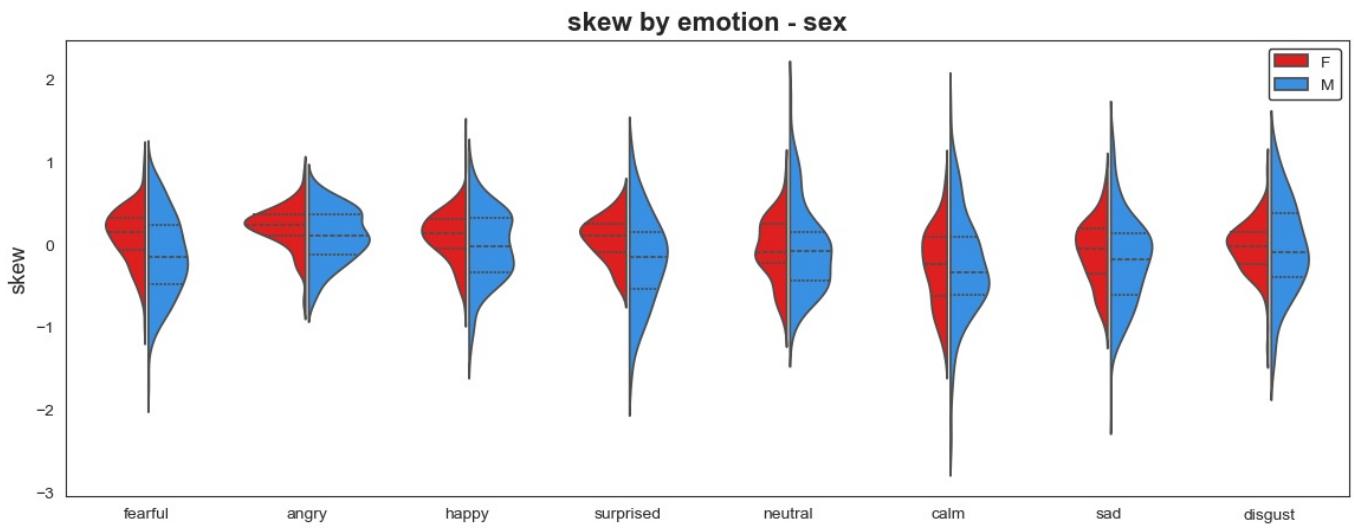


max by emotion - sex



kur by emotion - sex





3. Assessing data quality & variable transformations

Errori nel dataset: come mostrato sopra, la variabile 'frame_count' presentava alcuni valori inferiori < 0; dato il suo significato, e il resto della sua distribuzione, questi possono essere considerati errori.

Inaccuratezze sintattiche: nessuna delle variabili categoriche sembra presentare inaccuratezze sintattiche; possiamo verificarlo mostrandone i valori unici (ovvero, ad esempio, nessuna emozione ha refusi nel nome):

```
In [48]: for column in non_numerical_columns:
    print(column, ":", non_numerical_columns[column].unique(), sep="")
```

```
modality: ['audio-only']
vocal_channel: ['speech' nan 'song']
emotion: ['fearful' 'angry' 'happy' 'surprised' 'neutral' 'calm' 'sad' 'disgust']
emotional_intensity: ['normal' 'strong']
statement: ['Dogs are sitting by the door' 'Kids are talking by the door']
repetition: ['2nd' '1st']
sex: ['F' 'M']
```

Le inconsistenze semantiche sono difficili da individuare in questo dataset (non avendo, ad esempio, il nome degli attori). Allo stesso modo risulta arduo esprimersi sulla completezza dei dati; possiamo però notare, come già accennato in precedenza, che le variabili categoriche sembrano ben bilanciate ('M' vs. 'F', 'speech' vs. 'song', etc.). Inoltre, possiamo assicurarcoci che il dataset non presenti **duplicati**:

```
In [49]: len(df)
```

```
Out[49]: 2452
```

```
In [50]: df = df.drop_duplicates(subset=None, keep='first', inplace=False, ignore_index=True)
```

```
In [51]: len(df)
```

```
Out[51]: 2452
```

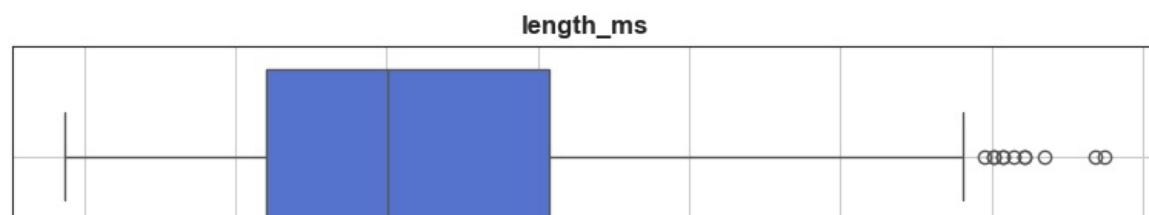
Attraverso l'uso di boxplot, possiamo mostrare che molte delle variabili continue presentano degli **outliers**:

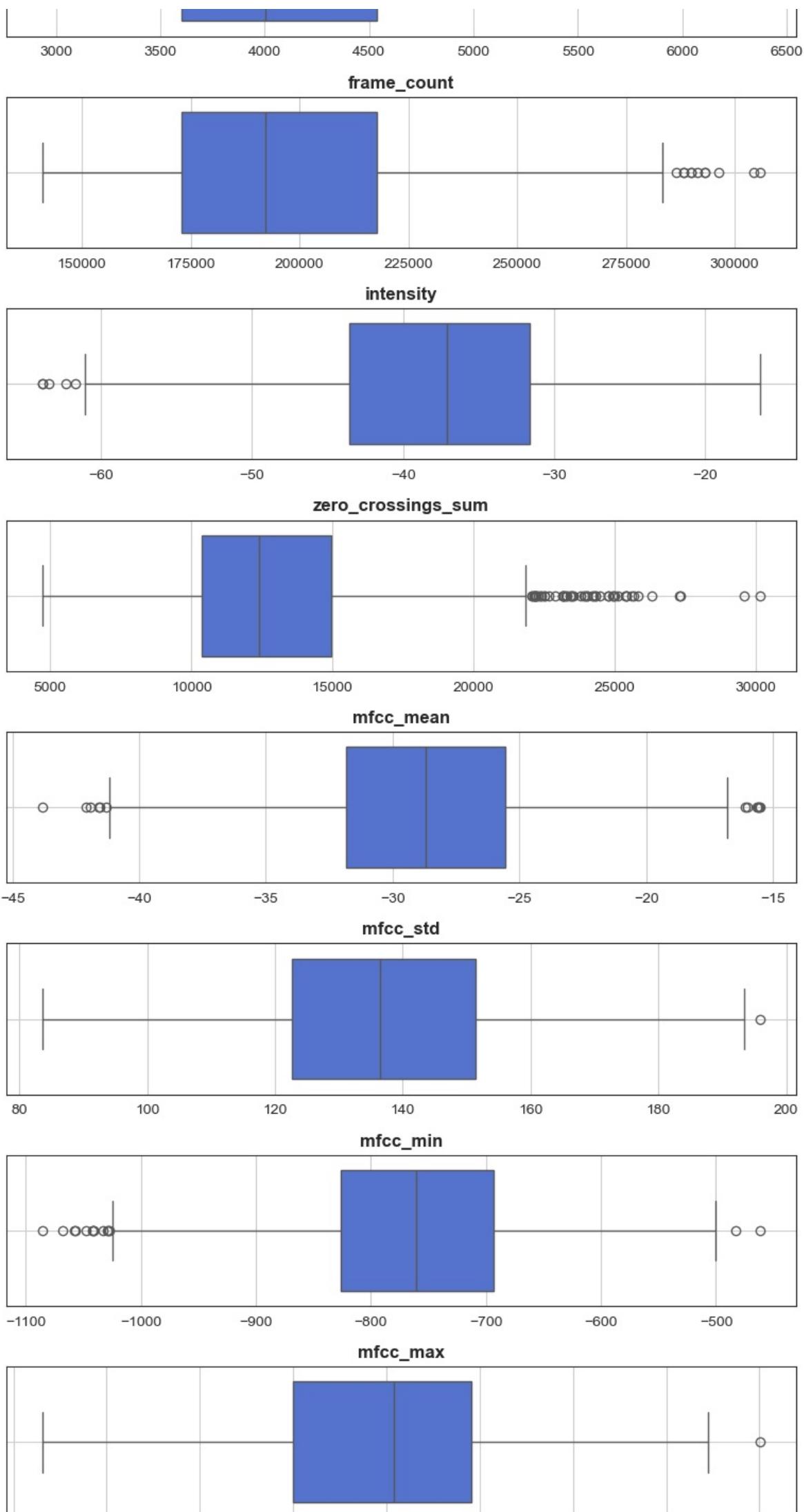
```
In [52]: box_col = [c for c in numerical_columns.columns if c not in
              ['actor', 'channels', 'sample_width', 'frame_rate', 'frame_width', 'stft_max']]

fig, axs = plt.subplots(len(box_col), 1, figsize=(8, len(box_col)*2))

for i, col in enumerate(box_col):
    ax = sns.boxplot(data=df, x=col, ax=axs[i], color='royalblue')
    ax.set_title(f'{col}', fontweight='bold')
    ax.set_xlabel('')
    ax.grid(True)

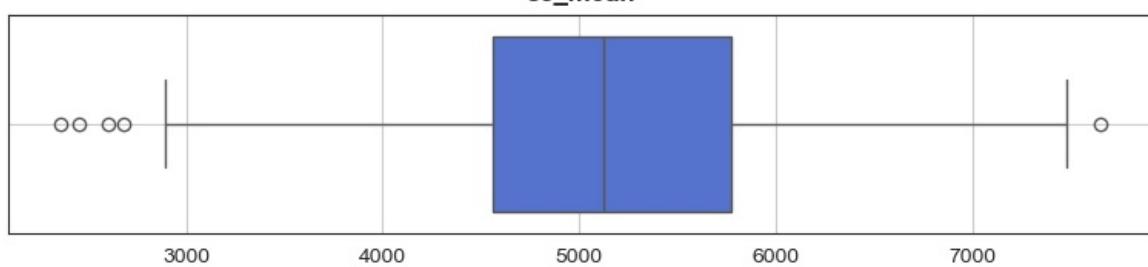
plt.tight_layout()
plt.show()
```



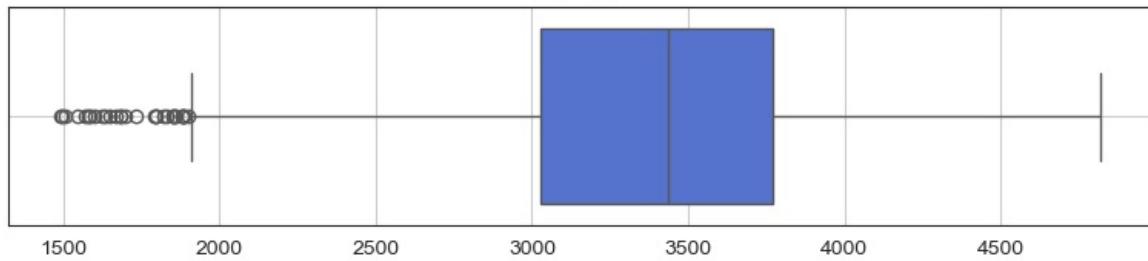


120 140 160 180 200 220 240 260 280

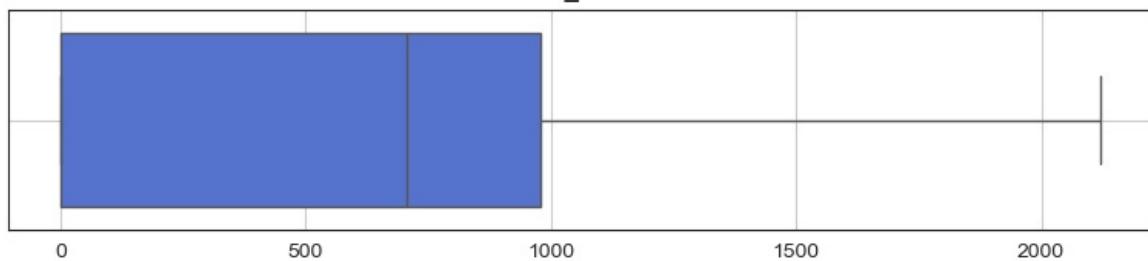
sc_mean



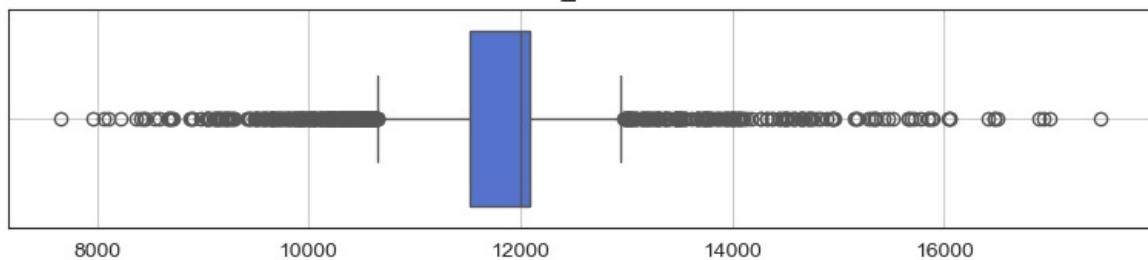
sc_std



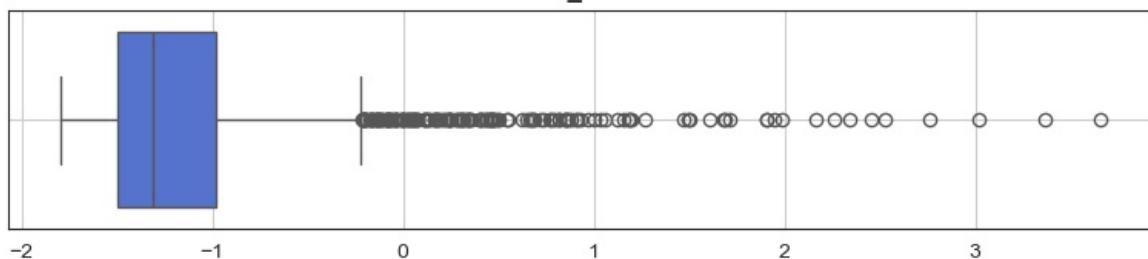
sc_min



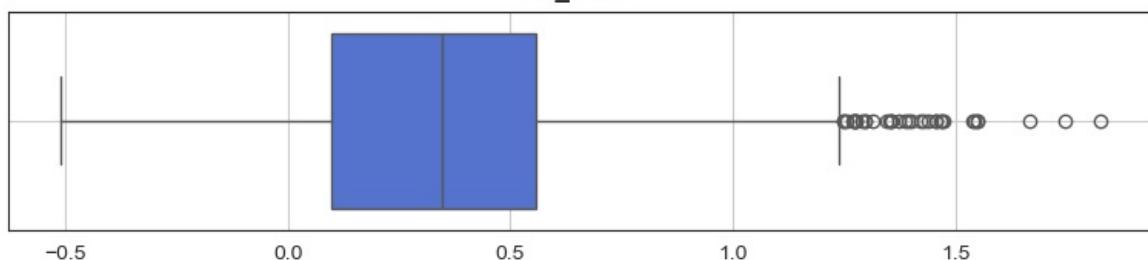
sc_max



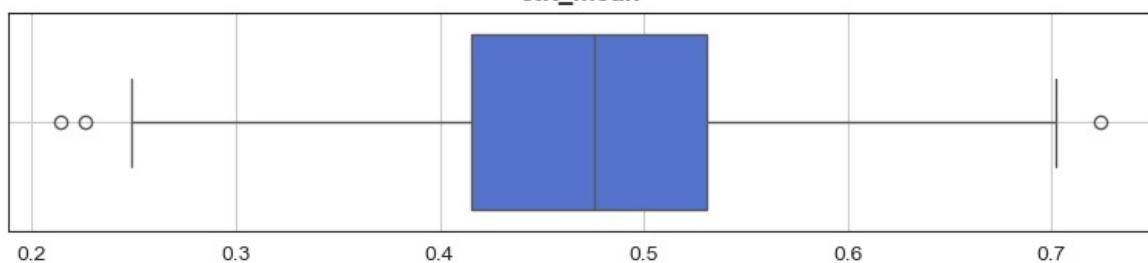
sc_kur



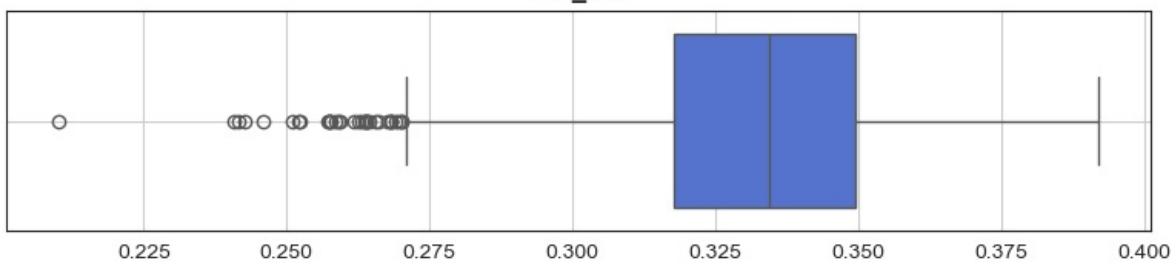
sc_skew



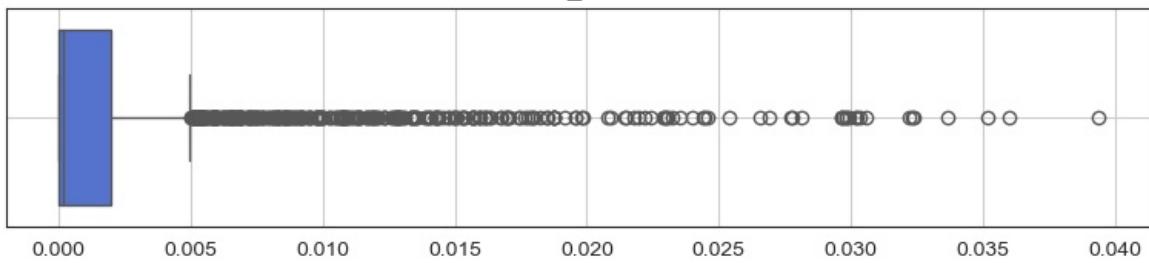
stft_mean



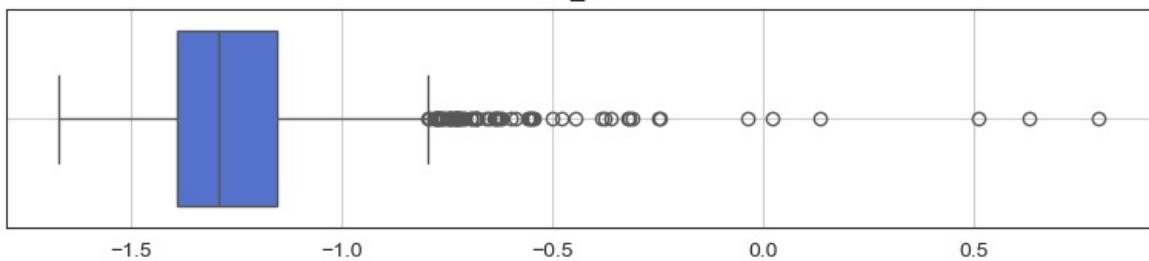
stft_std



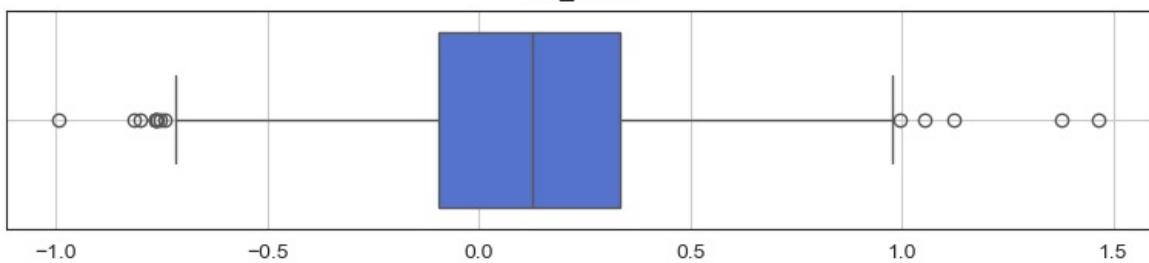
stft_min



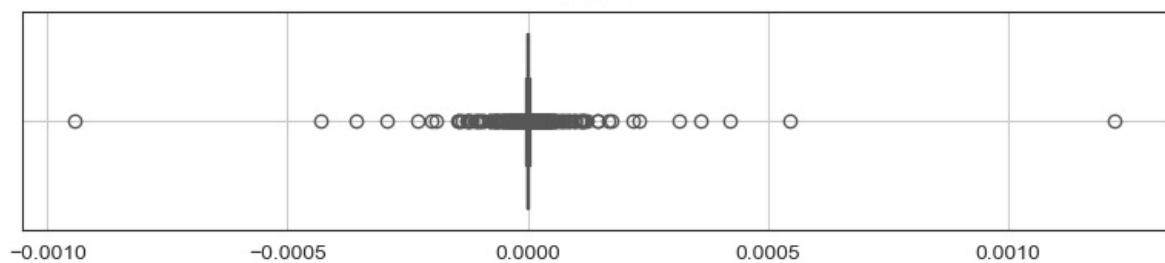
stft_kur



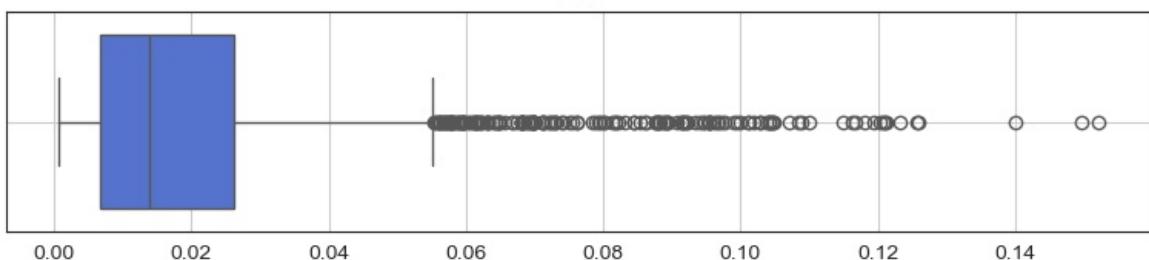
stft_skew



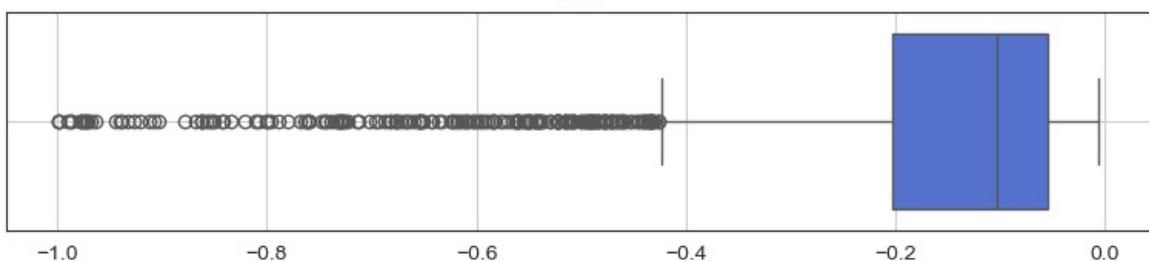
mean



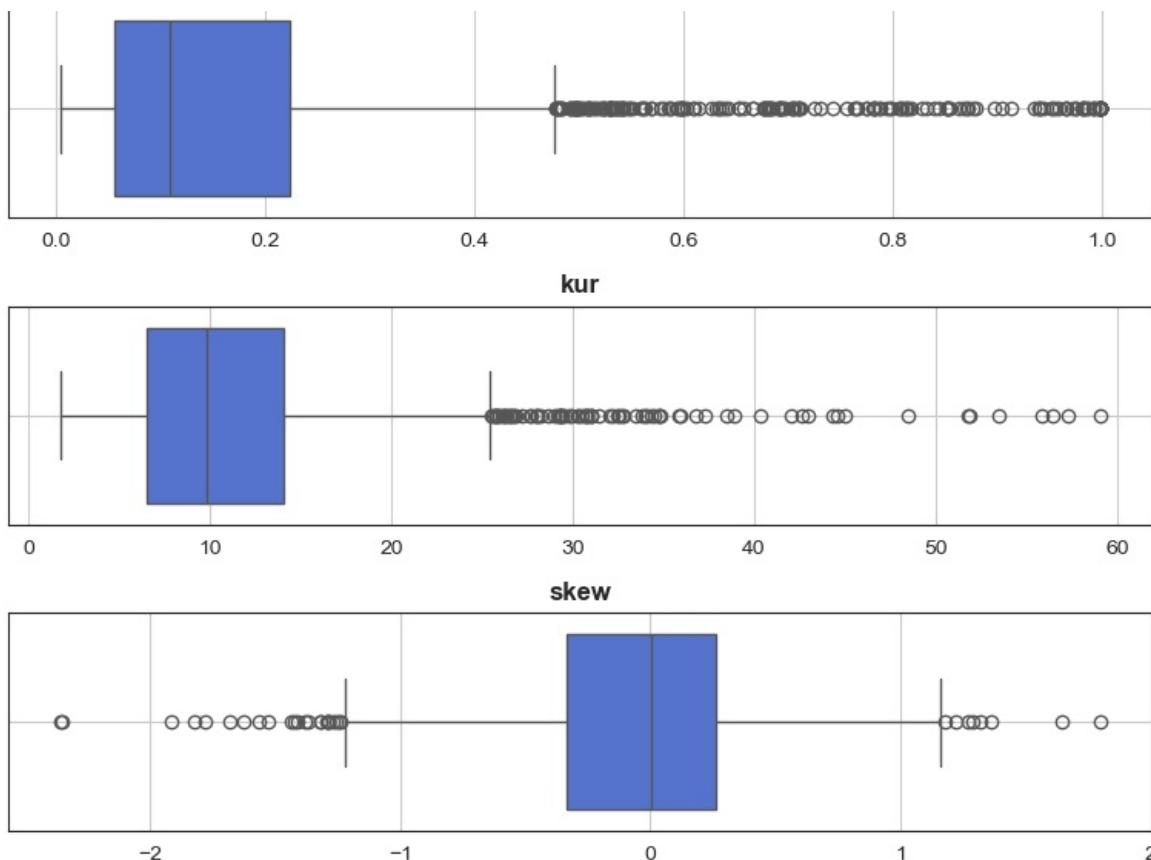
std



min



max



Troviamo gli outlier individuando i dati che sono più di **1.5 volte l'IQR sotto al primo quartile o sopra al terzo**, rispetto a ogni variabile continua:

```
In [53]: outliers_dict = {}

for column in numerical_columns.columns:
    feature = numerical_columns[column]

    q1 = np.quantile(feature, 0.25)
    q3 = np.quantile(feature, 0.75)
    iqr = q3 - q1

    outliers = []
    for v in feature:
        if v < q1 - 1.5 * iqr or v > q3 + 1.5 * iqr:
            outliers.append(True)
        else:
            outliers.append(False)

    outliers_dict[column] = outliers
```

Conteggio degli outlier per ogni feature:

```
In [54]: for column, outlier_value in outliers_dict.items():
    counter = sum(outlier_value)
    print(column, ":", counter, sep="")
```

```
actor: 0
channels: 6
sample_width: 0
frame_rate: 0
frame_width: 6
length_ms: 11
frame_count: 46
intensity: 0
zero_crossings_sum: 52
mfcc_mean: 12
mfcc_std: 1
mfcc_min: 14
mfcc_max: 1
sc_mean: 5
sc_std: 30
sc_min: 0
sc_max: 439
sc_kur: 152
sc_skew: 35
stft_mean: 3
stft_std: 33
stft_min: 336
stft_max: 0
stft_kur: 69
stft_skew: 14
mean: 536
std: 172
min: 207
max: 207
kur: 85
skew: 32
```

Non conoscendo nel dettaglio la semantica delle variabili, non siamo in grado di discernere se gli outlier siano errori nel dataset o data point genuini di cui dobbiamo tenere conto; per evitare di scartare record che dovremmo invece prendere in considerazione, e puntando a un'analisi più completa possibile, decidiamo quindi di tenerli, e valutare poi il loro impatto attraverso tecniche che risentono in diversa misura della loro presenza. Gli outlier individuati sopra sono comunque solo relativi alle variabili prese singolarmente; servirà quindi un'analisi multidimensionale per trovare i data point anomali rispetto a più attributi.

Per quanto riguarda i **missing values**, invece, questi sembrano essere effettive mancanze nel dataset, e non semplicemente dati a cui l'attributo in questione non si applica. Le colonne che presentano valori mancanti sono tre:

```
In [55]: df.isnull().sum()
```

```
Out[55]: modality          0  
vocal_channel      196  
emotion            0  
emotional_intensity 0  
statement          0  
repetition         0  
actor              1126  
sex                0  
channels           0  
sample_width       0  
frame_rate         0  
frame_width        0  
length_ms          0  
frame_count        0  
intensity          816  
zero_crossings_sum 0  
mfcc_mean          0  
mfcc_std           0  
mfcc_min           0  
mfcc_max           0  
sc_mean            0  
sc_std             0  
sc_min             0  
sc_max             0  
sc_kur             0  
sc_skew            0  
stft_mean          0  
stft_std           0  
stft_min           0  
stft_max           0  
stft_kur           0  
stft_skew          0  
mean               0  
std                0  
min               0  
max               0  
kur               0  
skew              0  
dtype: int64
```

```
In [56]: df.isnull().sum() / len(df)
```

```
Out[56]: modality      0.00000  
vocal_channel  0.07993  
emotion        0.00000  
emotional_intensity 0.00000  
statement      0.00000  
repetition     0.00000  
actor          0.45922  
sex            0.00000  
channels        0.00000  
sample_width    0.00000  
frame_rate      0.00000  
frame_width     0.00000  
length_ms       0.00000  
frame_count     0.00000  
intensity       0.33279  
zero_crossings_sum 0.00000  
mfcc_mean       0.00000  
mfcc_std        0.00000  
mfcc_min        0.00000  
mfcc_max        0.00000  
sc_mean          0.00000  
sc_std           0.00000  
sc_min           0.00000  
sc_max           0.00000  
sc_kur           0.00000  
sc_skew          0.00000  
stft_mean        0.00000  
stft_std         0.00000  
stft_min         0.00000  
stft_max         0.00000  
stft_kur         0.00000  
stft_skew        0.00000  
mean             0.00000  
std              0.00000  
min              0.00000  
max              0.00000  
kur              0.00000  
skew             0.00000  
dtype: float64
```

Come già accennato, la variabile 'actor' è un identificativo; non ha alcun ruolo nei nostri modelli, quindi possiamo accantonarla per il

momento. Passiamo invece a 'vocal_channel':

```
In [57]: df['vocal_channel'].value_counts(normalize=True)
```

```
Out[57]: vocal_channel
speech    0.59176
song      0.40824
Name: proportion, dtype: float64
```

I valori mancanti della variabile sono relativamente pochi, quindi non c'è un grosso rischio di modificarne la distribuzione originaria. Abbiamo perciò deciso di sostituire i missing values sfruttando la **proporzione dei suoi due valori unici**:

```
In [58]: vc_proportion = df['vocal_channel'].value_counts(normalize=True)
vc_null_num = df['vocal_channel'].isnull().sum()
replacements = np.random.choice(vc_proportion.index, size=vc_null_num, p=vc_proportion.values)

df.loc[df['vocal_channel'].isnull(), 'vocal_channel'] = replacements
```

La situazione è molto diversa per 'intensity'; la variabile riporta un totale di 816 missing values, che corrispondono al 33,28% delle osservazioni iniziali. Pertanto, data la numerosità del campione mancante, abbiamo cercato di effettuare il replacing attraverso un **modello di regressione**. La scelta è supportata dal fatto che altre strategie, come ad esempio il sampling da una distribuzione di probabilità, avrebbero potuto portare ad una sostituzione imprecisa.

Come features del regressore abbiamo deciso di utilizzare tutti gli attributi del dataset **ad eccezione di 'sex' ed 'emotion'**, in quanto queste variabili saranno successivamente predette, quindi utilizzarle in fase di replacing dei missing potrebbe introdurre un bias all'interno del dataset.

Dato che le features: ['vocal_channel', 'sex', 'emotional_intensity', 'repetition', 'statement'] sono **categoriche binarie**, possiamo effettuare la sostituzione dei valori sfruttando un **mapping 0 - 1**. Di fatto, nel caso di variabili binarie, questa strategia non è dissimile dall'effettuare un ONE-HOT-ENCODING, con l'unica differenza che in quest'ultimo si generano due colonne dummy che sono una il complemento a 1 dell'altra. Questo fa sì che entrambe abbiano lo stesso contenuto informativo, e che quindi una delle due possa essere scartata; il risultato è perciò identico a fare un semplice mapping 0 - 1.

```
In [59]: dict_map = {'speech': 0,
                 'song': 1,
                 'F': 0,
                 'M': 1,
                 'normal': 0,
                 'strong': 1,
                 '1st': 0,
                 '2nd': 1,
                 'Dogs are sitting by the door': 0,
                 'Kids are talking by the door': 1}
```

```
In [60]: for col in ['vocal_channel', 'sex', 'emotional_intensity', 'repetition', 'statement']:
    df[col] = df[col].apply(lambda x: dict_map[x])

df[['vocal_channel', 'sex', 'emotional_intensity', 'repetition', 'statement']]
```

```
Out[60]:   vocal_channel  sex  emotional_intensity  repetition  statement
0             0     0                  0           1         0
1             0     0                  0           0         0
2             0     0                  1           1         0
3             0     0                  0           0         1
4             1     0                  1           1         0
...
2447          0     1                  1           0         1
2448          0     1                  0           0         0
2449          1     1                  1           1         0
2450          0     1                  0           0         1
2451          0     1                  0           1         0
```

2452 rows × 5 columns

```
In [61]: df.head()
```

	modality	vocal_channel	emotion	emotional_intensity	statement	repetition	actor	sex	channels	sample_width	...	stft_m
0	audio-only	0	fearful	0	0	1	2.00000	0	1	2	...	0.00000
1	audio-only	0	angry	0	0	0	16.00000	0	1	2	...	0.00000
2	audio-only	0	happy	1	0	1	16.00000	0	1	2	...	0.00000
3	audio-only	0	surprised	0	1	0	14.00000	0	1	2	...	0.00000
4	audio-only	1	happy	1	0	1	2.00000	0	1	2	...	0.00000

5 rows × 38 columns



Come detto in precedenza, isoliamo le colonne ['emotion', 'sex'], che non verranno utilizzate come predittori al fine di non creare un bias nell'operazione di resampling; escludiamo inoltre gli attributi che verranno poi eliminati dal dataset per le ragioni discusse nel data understanding.

```
In [62]: list_col_reg = [col for col in df.columns.to_list() if col not in ['emotion', 'intensity', 'sex', 'actor', 'sample_width', 'frame_rate', 'frame_width', 'modality', 'channels', 'stft_max', 'mean']]  
df[list_col_reg].head()
```

	vocal_channel	emotional_intensity	statement	repetition	length_ms	frame_count	zero_crossings_sum	mfcc_mean	mfcc_std
0	0	0	0	1	3737	179379.00000	16995	-33.48595	134.65486
1	0	0	0	0	3904	187387.00000	13906	-29.50211	130.48563
2	0	1	0	1	4671	224224.00000	18723	-30.53246	126.57711
3	0	0	1	0	3637	174575.00000	11617	-36.05956	159.72516
4	1	1	0	1	4404	211411.00000	15137	-31.40600	122.12582

5 rows × 27 columns



A questo punto andiamo a creare gli array **X** e **y**, che corrisponderanno alle sole osservazioni che non presentano missing values.

```
In [63]: # Valori not NaN su cui addestrare e valutare il regressore:  
mask_not_nan = df['intensity'].notna()  
  
# Valori NaN da rimpiazzare attraverso la previsione del regressore precedentemente addestrato e valutato:  
mask_nan = df['intensity'].isna()  
  
X = df.loc[mask_not_nan, list_col_reg].values  
y = df.loc[mask_not_nan, 'intensity'].values  
  
X[0:3, :], y[0:3]
```

```
Out[63]: (array([[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
       1.00000000e+00,  3.73700000e+03,  1.79379000e+05,
       1.69950000e+04, -3.34859470e+01,  1.34654860e+02,
      -7.55223450e+02,  1.71690920e+02,  5.79255074e+03,
       3.32805546e+03,  0.00000000e+00,  1.35419590e+04,
      -1.12076852e+00,  2.50940162e-01,  4.15250450e-01,
       3.35533140e-01,  0.00000000e+00, -1.21502497e+00,
       4.03514030e-01,  1.44816360e-02, -1.28631590e-01,
       1.38946530e-01,  9.40606124e+00,  2.73153374e-01],
      [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
       1.00000000e+00,  4.67100000e+03,  2.24224000e+05,
       1.87230000e+04, -3.05324630e+01,  1.26577110e+02,
      -7.26060360e+02,  1.65456530e+02,  4.83074304e+03,
       3.33213130e+03,  0.00000000e+00,  1.20077512e+04,
      -1.13015274e+00,  4.36699155e-01,  3.79757520e-01,
       3.52269680e-01,  0.00000000e+00, -1.24294652e+00,
       4.70350013e-01,  2.43171750e-02, -1.37481690e-01,
       1.66351320e-01,  4.88124056e+00,  3.02658999e-01],
      [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00,
       0.00000000e+00,  3.63700000e+03,  1.74575000e+05,
       1.16170000e+04, -3.60595550e+01,  1.59725160e+02,
      -8.42946350e+02,  1.90036090e+02,  5.37644648e+03,
       4.05366307e+03,  0.00000000e+00,  1.20482239e+04,
      -1.49776486e+00,  9.88022841e-02,  4.07276720e-01,
       3.60552070e-01,  0.00000000e+00, -1.44531811e+00,
       2.74755933e-01,  3.56098640e-03, -2.73742680e-02,
       2.40783700e-02,  1.30402594e+01, -8.10143267e-02]]),
 array([-36.79343187, -32.29073734, -49.01983891]))
```

Dopo aver creato gli array X e y, passiamo a suddividere il dataset in **training set** e **test set** con proporzione 70% - 30%.

```
In [64]: from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

L'idea è quella di addestrare un **Decision Tree Regressor** sul training set, andando ad ottimizzare i parametri con una **Random Search Cross Validation**. Impostiamo un numero di fold della CV pari a 10 e reiteriamo la CV per 10 volte.

I parametri che abbiamo deciso di ottimizzare sono:

- massima profondità dell'albero;
- minimo numero di samples che devono essere presenti all'intero di un nodo per giustificare uno split;
- minimo numero di samples che possono stare all'interno di una foglia;
- parametro di regolarizzazione alpha che introduce una penalizzazione sul numero di foglie generate a seguito dei vari split, e dunque permette di regolare il trade-off tra bias (albero troppo poco dettagliato) e variance (albero in completo overfitting e quindi troppo ricco di foglie).

La metrifica utilizzata in fase di validazione del modello è il Mean Squared Error (MSE).

```
In [65]: from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeRegressor

# definisci la hyperparameter grid:
dict_params = {'max_depth': list(np.arange(1, 10, 3)),
               'min_samples_split': np.arange(5, 21, 3),
               'min_samples_leaf': np.arange(5, 21, 3),
               'ccp_alpha': np.arange(0, 0.2, 0.02)}

# crea il DecisionTreeRegressor:
reg = DecisionTreeRegressor()

rand = RandomizedSearchCV(reg, dict_params, cv = 10, scoring = 'neg_mean_squared_error', refit = True, n_iter = rand.fit(X_train, y_train))
```

```
Out[65]: >      RandomizedSearchCV
> estimator: DecisionTreeRegressor
    > DecisionTreeRegressor
```

Di seguito osserviamo in ordine:

- migliori parametri, ovvero quelli che hanno mediamente permesso di ottenere un MSE più basso durante le varie iterazioni della 10-fold CV;
- miglior MSE ottenuto durante la 10-fold CV;
- miglior Tree ottenuto durante la 10-fold CV.

```
In [66]: rand.best_params_
```

```
Out[66]: {'min_samples_split': 11,
           'min_samples_leaf': 17,
           'max_depth': 7,
           'ccp_alpha': 0.04}
```

```
In [67]: best_validation_metric = rand.best_score_
best_validation_metric
```

```
Out[67]: -0.554803985356809
```

```
In [68]: reg = rand.best_estimator_
reg
```

```
Out[68]: ▾ DecisionTreeRegressor
DecisionTreeRegressor(ccp_alpha=0.04, max_depth=7, min_samples_leaf=17,
                      min_samples_split=11)
```

A questo punto possiamo fare il miglior Tree sull'intero dataset di training e usare il modello addestrato per andare a prevedere le osservazioni di test e vedere quanto la previsione `y_pred` si discosta dal reale valore `y_test`.

```
In [69]: reg.fit(X_train, y_train) # fit su intero training
y_pred = reg.predict(X_test) # predizione sul test set
```

```
In [70]: y_test[0:10], y_pred[0:10]
```

```
Out[70]: (array([-51.92743685, -44.37569489, -20.01804365, -30.79117597,
                  -29.77833341, -32.83096674, -21.15136076, -20.40644107,
                  -36.1405807 , -38.07453254]),
array([-51.36697177, -44.07733557, -20.36167671, -31.38662567,
      -29.44401329, -33.475995 , -20.36167671, -20.36167671,
      -36.40105338, -37.92074047]))
```

Di seguito riportiamo alcune metriche che per i task di regressione aiutano a capire se il modello è accurato:

- Mean Absolute Error;
- Mean Squared Error --> loss function scelta per il modello;
- BIAS --> permette di capire se il modello sta facendo overshooting oppure il contrario;
- R2 --> varianza spiegata del modello.

```
In [71]: # Mean Absolute Error
MAE = np.mean(np.abs(y_pred - y_test))
MSE = np.mean((y_pred - y_test)**2)
BIAS = np.mean((y_pred - y_test))
R2 = 1 - np.var(y_pred - y_test)/np.var(y_test)

print(f'Metriche di errore ottenute in fase di test:\nMAE: {MAE}\nMSE: {MSE}\nBIAS: {BIAS}\nR^2: {R2}')
```

Metriche di errore ottenute in fase di test:
MAE: 0.5693526351795175
MSE: 0.5259450230307586
BIAS: -0.014702059445458344
R^2: 0.9927433899392226

Il modello ha dei risultati talmente buoni da risultare anomali. Con una varianza spiegata del 99% circa, abbiamo quasi la certezza che **sussista una relazione esatta, deterministica** tra la variabile `intensity` e almeno una delle variabili scelte in fase di training.

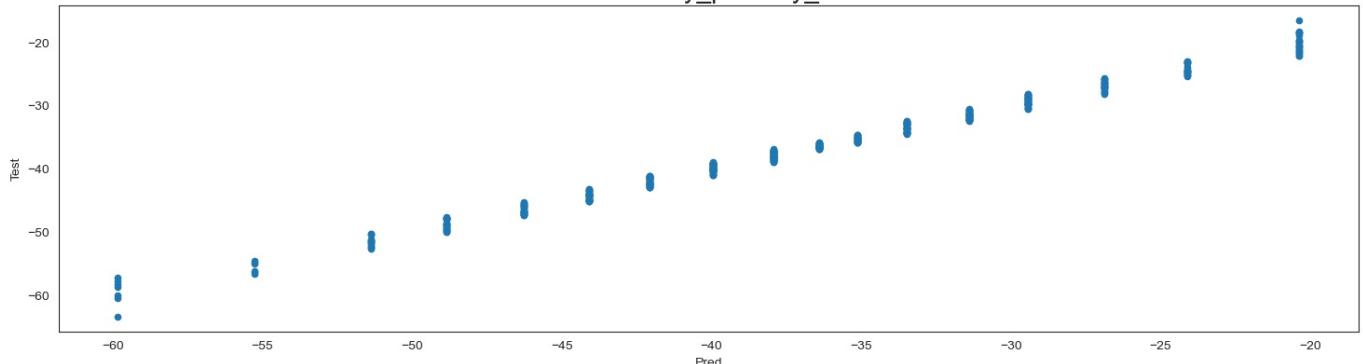
Questa affermazione è avvalorata dai seguenti due plot, che mettono in relazione `y_pred` e `y_test`:

```
In [72]: df_temp = pd.DataFrame({'Pred': y_pred, 'Test': y_test})

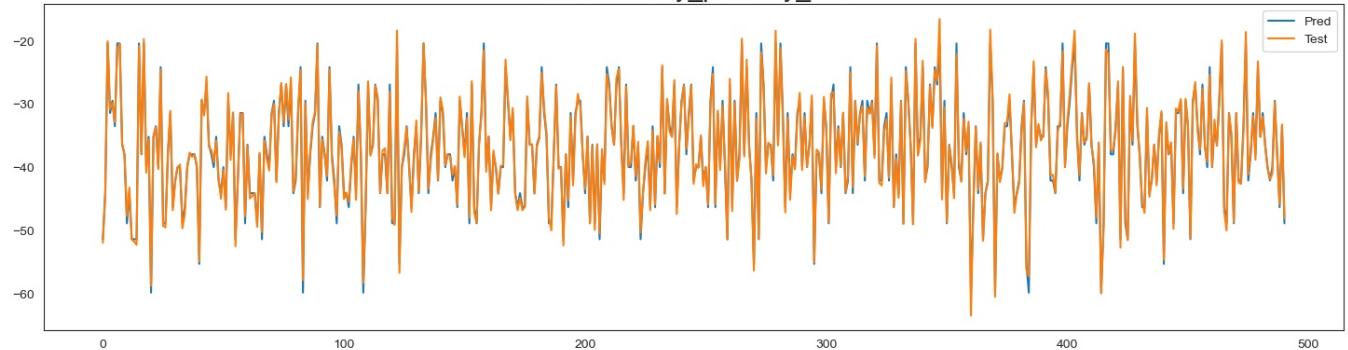
fig, ax = plt.subplots(2, figsize = (18, 10))

df_temp.plot(kind = 'scatter', x = 'Pred', y = 'Test', ax = ax[0])
df_temp.plot(ax = ax[1])
ax[0].set_title('Scatter tra y_pred e y_test', fontsize = 20)
ax[1].set_title('Andamento y_pred e y_test', fontsize = 20)
plt.legend()
plt.show()
```

Scatter tra y_pred e y_test



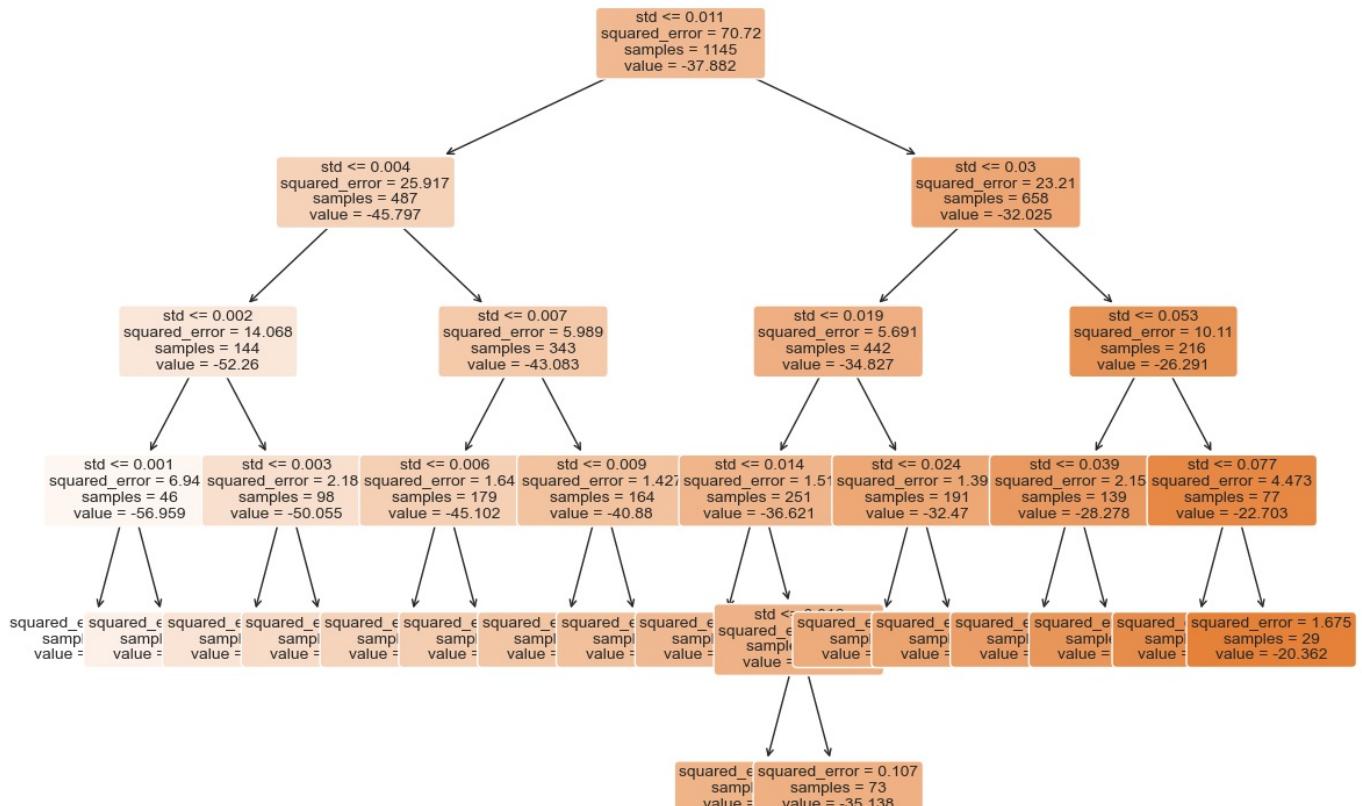
Andamento y_pred e y_test



Cerchiamo di capire il motivo di questo comportamento andando ad osservare su quali attributi l'albero esegue i primi split.

```
In [73]: from sklearn.tree import plot_tree

plt.figure(figsize=(14, 10))
plot_tree(reg,
          feature_names=list_col_reg,
          filled=True,
          rounded=True,
          fontsize=10,
          #max_depth=3
         )
plt.show()
```



Vediamo chiaramente che tutti gli split sono in corrispondenza dello stesso attributo **std** e quindi la relazione deve essere attribuita a tale variabile.

Dal seguente grafico osserviamo che la **relazione supposta esiste** ed è **non lineare**, più specificamente del tipo $f(x) = x^b$ con $b < 1$.

Dato che l'implementazione con l'algoritmo **CART** del Decision Tree esegue split axis parallel, notiamo che l'approssimazione della funzione da parte del Decision Tree è "**a scalini**" e quindi non adatta al problema.

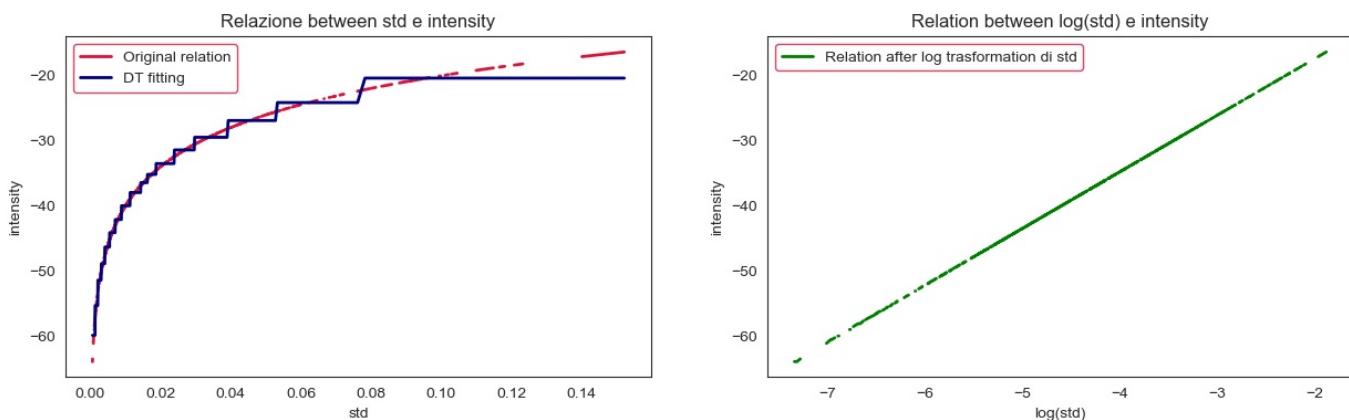
La strategia migliore sarebbe quella di catturare tale relazione attraverso una **trasformazione log della variabile std**, in modo da visualizzare una relazione lineare tra $\log(\text{std})$ e **intensity**.

```
In [76]: temp = df[['std', 'intensity']].copy(deep = True)
temp['DT_fitting'] = reg.predict(df.loc[:, list_col_reg].values)
temp['log_std'] = np.log(df['std'])
temp = temp.sort_values(by = ['std'])

fig, ax = plt.subplots(1, 2, figsize = (15, 4))
ax[0].plot(temp['std'], temp['intensity'], c = 'crimson', label = 'Original relation', lw = 2, linestyle = '-.')
ax[0].plot(temp['std'], temp['DT_fitting'], c = 'navy', label = 'DT fitting', lw = 2, linestyle = '-.')
ax[0].set_title('Relazione between std e intensity')
ax[0].set_xlabel('std')
ax[0].set_ylabel('intensity')
ax[0].legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)

ax[1].plot(temp['log_std'], temp['intensity'], c = 'green', label = 'Relation after log trasformation di std',
           ax[1].set_title('Relation between log(std) e intensity')
ax[1].set_xlabel('log(std)')
ax[1].set_ylabel('intensity')
ax[1].legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)

plt.show()
```



Come dimostrato nel plot esiste una relazione lineare esatta del tipo $\text{intensity} = m \cdot \log(\text{std}) + q$.

Dunque andiamo ad estrarre i parametri m e q di questa retta e proviamo a ricostruire la relazione originaria tra **std** e **intensity**.

```
In [77]: from sklearn.linear_model import LinearRegression

In [78]: X = np.log(df.loc[mask_not_nan, 'std'].values.reshape(-1, 1))
y = df.loc[mask_not_nan, 'intensity'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

```
In [79]: linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
print(f'Intercetta: {linear_model.intercept_}\nCoefficiente Angolare: {linear_model.coef_[0]}')

Intercetta: 0.0613349472312521
Coefficiente Angolare: 8.703817022740086
```

Otteniamo che: $\text{intensity} = 8.7 \cdot \log(\text{std}) + 0.06$

Ovviamente calcolando le metriche di accuratezza, il bias e la varianza spiegata del modello, otteniamo i massimi valori ottenibili.

```
In [80]: y_pred = linear_model.predict(X_test)

# Mean Absolute Error
MAE = np.mean(np.abs(y_pred - y_test))
MSE = np.mean((y_pred - y_test)**2)
BIAS = np.mean((y_pred - y_test))
R2 = 1 - np.var(y_pred - y_test)/np.var(y_test)

print(f'Metriche di errore ottenute in fase di test:\nMAE: {MAE}\nMSE: {MSE}\nBIAS: {BIAS}\nR^2: {R2}' )
```

Metriche di errore ottenute in fase di test:

MAE: 0.011410262462256057

MSE: 0.000377189979039145

BIAS: -0.00039563557075306214

R^2: 0.9999947958256035

A questo punto è evidente che non abbia molto senso proseguire l'analisi portandosi dietro entrambe le variabili **intensity** e **std**, dato che abbiamo visto che l'una è esattamente ottenibile a partire dall'altra. Quindi possiamo tranquillamente eliminare una delle due variabili e tenere l'altra: il contenuto informativo dei dati non subirà variazioni.

Tuttavia per **completezza** si preferisce completare la procedura di **replacing dei missing values**, al termine della quale **una delle due variabili verrà eliminata**.

In [81]:

```
import copy

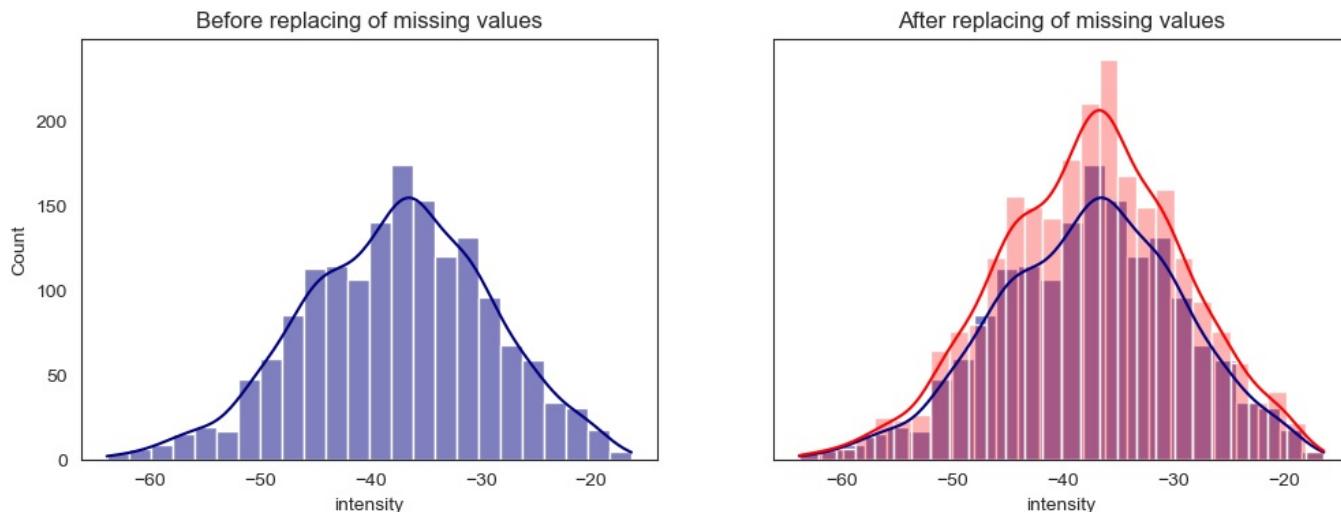
old_intensity = df['intensity'].copy(deep = True)
df.loc[mask_nan, 'intensity'] = linear_model.predict( np.log(df.loc[mask_nan, 'std']).values.reshape(-1, 1) )
```

In [82]:

```
fig, ax = plt.subplots(1, 2, figsize = (12, 4), sharey = True)

sns.histplot(old_intensity, ax = ax[0], kde = True, color = 'navy')
sns.histplot(old_intensity, ax = ax[1], kde = True, color = 'navy')
sns.histplot(df['intensity'], ax = ax[1], alpha = 0.3, kde = True, color = 'red')
ax[0].set_title('Before replacing of missing values')
ax[1].set_title('After replacing of missing values')

plt.show()
```



Come accennato in precedenza, procediamo all'eliminazione della variabile **intensity**:

In [83]:

```
df.drop(columns = ['intensity'], inplace = True)
```

4. Pairwise correlations and eventual elimination of variables

Continuando con la fase di data preparation, possiamo notare che non sembra necessario un sampling, dato che il numero di record non è troppo elevato. D'altra parte, è invece utile una riduzione del numero degli attributi al fine di mitigare la 'curse of dimensionality'; utilizzeremo a questo scopo tecniche di **feature selection** e **feature projection**.

Iniziamo rimuovendo gli attributi che non apportano contributi informativi, viste le ragioni discusse nel data understanding. Ricapitolandole:

- tra le variabili categoriche: '**modality**' ha un solo valore in tutto il dataset:

In [84]:

```
df['modality'].nunique()
```

Out[84]:

```
1
```

- tra quelle continue: '**sample_width**', '**frame_rate**', '**stft_max**' hanno varianza pari a 0:

In [85]:

```
df[['sample_width', 'frame_rate', 'stft_max']].var()
```

Out[85]:

```
sample_width    0.000000
frame_rate     0.000000
stft_max       0.000000
dtype: float64
```

- 'actor' è solo un identificativo:

```
In [86]: df['actor'].unique()
```

```
Out[86]: array([ 2., 16., 14., nan, 12., 6., 22., 20., 8., 23., 7., 13., 3.,
       17., 21., 1., 15., 5., 19., 9., 11., 24., 18., 4., 10.])
```

- 'channels' e 'frame_width' hanno lo stesso valore in tutto il dataset, fatta eccezione per 6 record, gli stessi per entrambe le variabili:

```
In [87]: len(df[(df['channels'] == 2) & (df['frame_width'] == 4)])
```

```
Out[87]: 6
```

```
In [88]: len(df[(df['channels'] != 2) & (df['frame_width'] != 4)])
```

```
Out[88]: 2446
```

Procediamo quindi al drop:

```
In [89]: df = df.drop(['modality', 'sample_width', 'frame_rate', 'stft_max', 'actor', 'channels', 'frame_width'], axis=1)
```

- a differenza delle variabili eliminate in precedenza, 'mean' ha 2450 valori unici; tuttavia, la sua varianza è molto prossima allo 0:

```
In [90]: df['mean'].nunique()
```

```
Out[90]: 2450
```

```
In [91]: df['mean'].var()
```

```
Out[91]: 1.8212297990049588e-09
```

La eliminiamo quindi con un **Variance Threshold**; sceglioamo come soglia 1.8212297990049588e-06 perché non vogliamo escludere altre feature, come ad esempio 'stft_min':

```
In [92]: df.var(numeric_only=True)
```

```
Out[92]: vocal_channel          0.24183
emotional_intensity          0.24863
statement                      0.25010
repetition                     0.25010
sex                            0.25002
length_ms                      357988.64825
frame_count                     813199784.87234
zero_crossings_sum             13434567.65492
mfcc_mean                       19.90843
mfcc_std                        418.27157
mfcc_min                        9989.09344
mfcc_max                        676.10957
sc_mean                          765949.56097
sc_std                           336955.90887
sc_min                           258090.30540
sc_max                           1009936.51318
sc_kur                           0.32793
sc_skew                          0.12461
stft_mean                        0.00681
stft_std                         0.00057
stft_min                         0.00002
stft_kur                         0.04485
stft_skew                        0.10940
mean                            0.00000
std                             0.00044
min                            0.03078
max                            0.03824
kur                            43.75636
skew                           0.20696
dtype: float64
```

```
In [93]: from sklearn.feature_selection import VarianceThreshold
```

```
numerical_columns = df.select_dtypes(include='number')
selector = VarianceThreshold(threshold=1.8212297990049588e-06)
```

```
numerical_reduced = selector.fit_transform(numerical_columns)
```

```
numerical_reduced_df = pd.DataFrame(numerical_reduced, columns=numerical_columns.columns[selector.get_support()])
```

```
In [94]: numerical_reduced_df.var()
```

```
Out[94]: vocal_channel          0.24183
emotional_intensity        0.24863
statement                  0.25010
repetition                 0.25010
sex                         0.25002
length_ms                  357988.64825
frame_count                813199784.87234
zero_crossings_sum         13434567.65492
mfcc_mean                  19.90843
mfcc_std                   418.27157
mfcc_min                   9989.09344
mfcc_max                   676.10957
sc_mean                     765949.56097
sc_std                      336955.90887
sc_min                      258090.30540
sc_max                      1009936.51318
sc_kur                      0.32793
sc_skew                     0.12461
stft_mean                  0.00681
stft_std                   0.00057
stft_min                   0.00002
stft_kur                   0.04485
stft_skew                  0.10940
std                          0.00044
min                          0.03078
max                          0.03824
kur                          43.75636
skew                         0.20696
dtype: float64
```

```
In [95]: non_numerical_columns = df.select_dtypes(exclude='number')

df = pd.concat([non_numerical_columns, numeric_reduced_df], axis=1)
```

Il dataframe è quindi passato da **38 a 29 colonne**, nessuna di questi con valori mancanti:

```
In [96]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2452 entries, 0 to 2451
Data columns (total 29 columns):
 #   Column           Non-Null Count  Dtype  
 ---  --  
 0   emotion          2452 non-null    object 
 1   vocal_channel    2452 non-null    float64
 2   emotional_intensity  2452 non-null    float64
 3   statement         2452 non-null    float64
 4   repetition        2452 non-null    float64
 5   sex               2452 non-null    float64
 6   length_ms        2452 non-null    float64
 7   frame_count      2452 non-null    float64
 8   zero_crossings_sum  2452 non-null    float64
 9   mfcc_mean         2452 non-null    float64
 10  mfcc_std          2452 non-null    float64
 11  mfcc_min          2452 non-null    float64
 12  mfcc_max          2452 non-null    float64
 13  sc_mean           2452 non-null    float64
 14  sc_std            2452 non-null    float64
 15  sc_min            2452 non-null    float64
 16  sc_max            2452 non-null    float64
 17  sc_kur            2452 non-null    float64
 18  sc_skew           2452 non-null    float64
 19  stft_mean         2452 non-null    float64
 20  stft_std          2452 non-null    float64
 21  stft_min          2452 non-null    float64
 22  stft_kur          2452 non-null    float64
 23  stft_skew         2452 non-null    float64
 24  std               2452 non-null    float64
 25  min               2452 non-null    float64
 26  max               2452 non-null    float64
 27  kur               2452 non-null    float64
 28  skew              2452 non-null    float64
dtypes: float64(28), object(1)
memory usage: 555.7+ KB
```

Esploriamo ora le **correlazioni** tra le variabili:

```
In [100...]: corr_matrix = df.corr(numeric_only=True)
corr_matrix.style.background_gradient(cmap='coolwarm')
```

Out[100...]	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	frame_count	zero_crossings_
vocal_channel	1.000000	-0.003409	-0.004147	0.000829	0.025720	0.735581	0.730971	0.150
emotional_intensity	-0.003409	1.000000	0.000000	0.000000	-0.000256	0.114027	0.112906	0.254
statement	-0.004147	0.000000	1.000000	-0.000000	0.000000	0.029536	0.031539	-0.162
repetition	0.000829	0.000000	-0.000000	1.000000	-0.000000	0.015751	0.014837	0.016
sex	0.025720	-0.000256	0.000000	-0.000000	1.000000	-0.071584	-0.071873	-0.389
length_ms	0.735581	0.114027	0.029536	0.015751	-0.071584	1.000000	0.992904	0.329
frame_count	0.730971	0.112906	0.031539	0.014837	-0.071873	0.992904	1.000000	0.326
zero_crossings_sum	0.150020	0.254869	-0.162901	0.016961	-0.389077	0.329517	0.326649	1.000
mfcc_mean	0.041158	0.305904	0.019976	0.011494	0.539941	0.011321	0.010193	0.130
mfcc_std	-0.273710	-0.416725	0.022503	-0.018511	0.108791	-0.302735	-0.301194	-0.559
mfcc_min	0.189327	0.403562	-0.010737	0.013579	-0.126288	0.205996	0.205063	0.500
mfcc_max	-0.272968	-0.213957	-0.002525	-0.002847	0.545416	-0.347253	-0.344543	-0.448
sc_mean	-0.490620	-0.118081	-0.083286	-0.010039	-0.042652	-0.546234	-0.539614	-0.072
sc_std	-0.139776	-0.256610	-0.075070	-0.013379	-0.100327	-0.182306	-0.178340	-0.373
sc_min	-0.044178	0.076052	-0.012920	0.008414	0.301492	-0.077001	-0.077952	0.190
sc_max	-0.119548	-0.090582	0.003062	-0.010509	-0.203118	-0.118781	-0.114388	-0.040
sc_kur	0.231057	0.233992	0.075559	0.005785	0.019193	0.292127	0.290117	0.186
sc_skew	0.537546	0.197154	0.076676	0.009429	0.007658	0.621148	0.616003	0.140
stft_mean	-0.461034	-0.081052	-0.001038	-0.012592	0.574827	-0.561395	-0.556985	-0.400
stft_std	0.375834	0.031236	-0.057050	0.007671	-0.578974	0.409380	0.407534	0.119
stft_min	-0.184459	-0.085916	-0.000847	0.005653	0.422499	-0.245511	-0.243661	-0.195
stft_kur	-0.098856	0.004877	0.079564	-0.004754	0.129566	-0.072309	-0.072402	0.072
stft_skew	0.489754	0.107097	0.011607	0.014438	-0.462716	0.602057	0.596986	0.449
std	0.132173	0.400001	-0.025725	0.018927	-0.083554	0.167134	0.166655	0.470
min	-0.030327	-0.416768	0.005634	-0.009472	0.052830	-0.070644	-0.071574	-0.420
max	0.037932	0.417563	0.001904	0.008832	-0.053374	0.072309	0.072823	0.411
kur	-0.486250	0.038873	0.168459	-0.028316	0.118022	-0.460723	-0.456365	-0.118
skew	0.036377	0.123895	0.032291	-0.005024	-0.122251	0.064473	0.067136	0.201

Come osservato in precedenza, dopo la rimozione degli errori da **'frame_count'** (ovvero i record con valori < 0), la variabile ha una forte correlazione positiva con **'length_ms'**; ciò risulta sensato anche dal punto di vista semantico. Possiamo quindi eliminare una delle due variabili – naturalmente, sceglieremo di mantenere la feature a cui non abbiamo apportato modifiche:

```
In [101...]: df = df.drop('frame_count', axis=1)
```

```
In [103...]: df.columns
```

```
Out[103...]: Index(['emotion', 'vocal_channel', 'emotional_intensity', 'statement',
       'repetition', 'sex', 'length_ms', 'zero_crossings_sum', 'mfcc_mean',
       'mfcc_std', 'mfcc_min', 'mfcc_max', 'sc_mean', 'sc_std', 'sc_min',
       'sc_max', 'sc_kur', 'sc_skew', 'stft_mean', 'stft_std', 'stft_min',
       'stft_kur', 'stft_skew', 'std', 'min', 'max', 'kur', 'skew'],
      dtype='object')
```

5. Dimensionality Reduction: Principal Component Analysis (PCA)

Proseguiamo adesso l'analisi con la **Principal Component Analysis**, una tecnica che consente di **ridurre la dimensionalità del problema**, andando a proiettare i dati nello spazio delle componenti principali PC_1, \dots, PC_n , ovvero delle **combinazioni lineari** degli attributi originari, generatrici dello spazio che permette di osservare i dati dalla **prospettiva di massima variabilità**. Questo perché le componenti principali sono le direzioni di massima variabilità dei dati.

Gli **attributi da usare nella PCA** sono i **continui**, opportunamente **standardizzati** per evitare che le differenze di scala inficino la bontà del risultato.

```
In [104...]: numeric_cols = ['length_ms',
       'zero_crossings_sum',
       'mfcc_mean', 'mfcc_std',
       'mfcc_min', 'mfcc_max',
       'sc_mean',
```

```

'sc_std',
'sc_min',
'sc_max',
'sc_kur',
'sc_skew',
'stft_mean',
'stft_std',
'stft_min',
'stft_kur',
'stft_skew',
'std',
'min',
'max',
'kur',
'skew']

```

Dalla teoria sappiamo che le direzioni di massima variabilità dei dati sono gli **autovettori della matrice di covarianza**, quindi un buon punto di partenza per comprendere se l'approccio può risultare fruttuoso è quello di calcolare la **matrice di correlazione** (che altro non è che la **matrice di covarianza scalata** rispetto alle deviazioni standard dei predittori coinvolti).

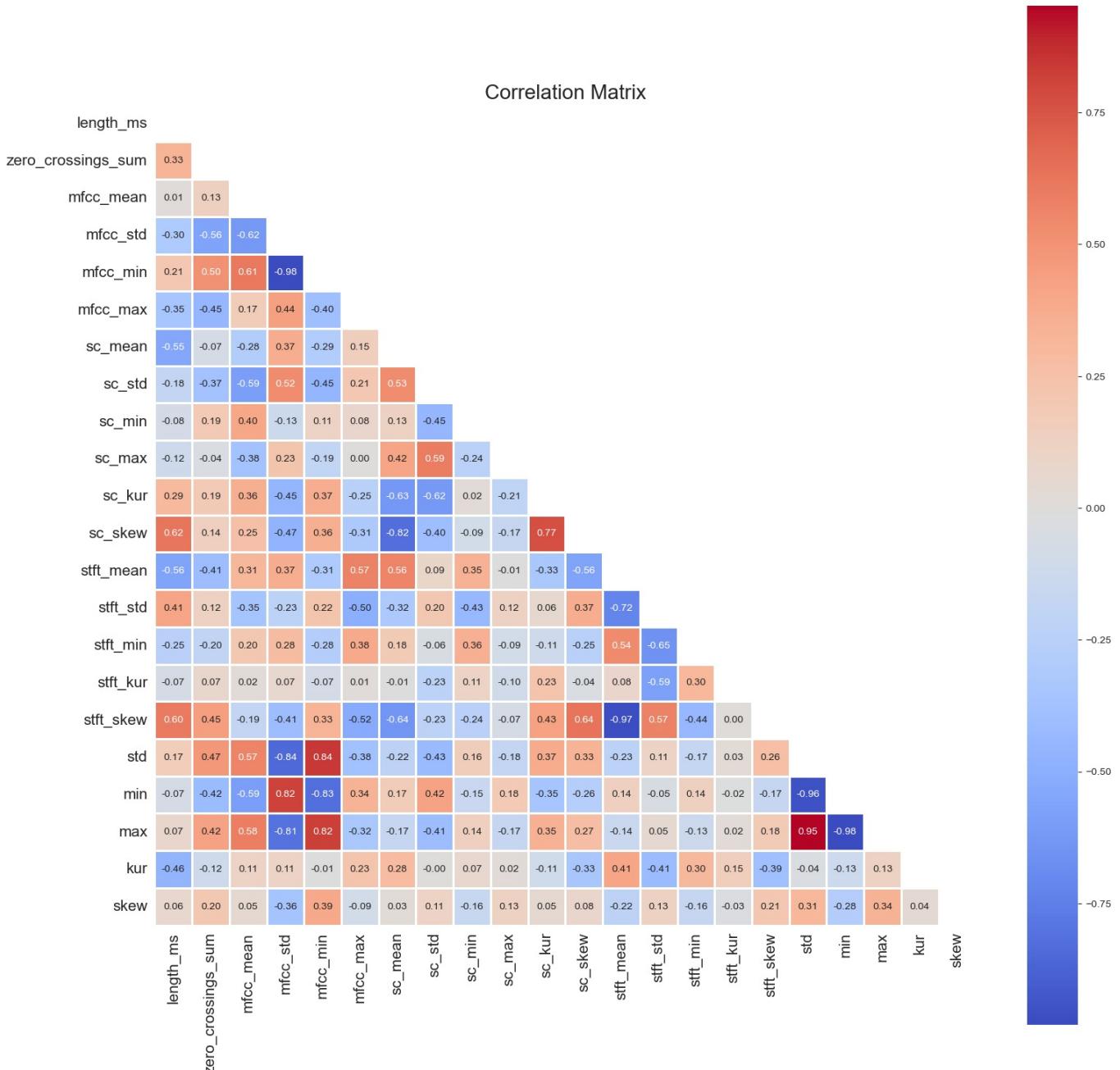
In [108]:

```

plt.figure(figsize=(18, 18))
corr_matrix = df[numeric_cols].corr(method = 'pearson', numeric_only=True)
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
sns.heatmap(corr_matrix, mask=mask, cmap="coolwarm", annot=True, fmt=".2f", square = True, linewidths = 0.8)
plt.title('Correlation Matrix', fontsize = 20)
plt.xticks(fontsize = 15)
plt.yticks(fontsize = 15)

plt.show()

```



Notiamo diverse correlazioni significative che risultano un buon presupposto per capire di più sulla struttura dei dati.

```
In [109]: from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

Nella cella sottostante prima **standardizziamo** i fattori considerati con uno **standard scaler**, che semplicemente esegue questa operazione:

$$Z = \frac{X - \mu}{\sigma}, \text{ per ogni predittore } X, \text{ con } (\mu, \sigma) \text{ rispettivamente media e deviazione standard del vettore } X$$

Dopodiché proiettiamo ogni osservazione, nello spazio delle componenti principali:

```
In [110]: X = df[numeric_cols].values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
In [111]: n_comp = 6

pca = PCA(n_components = n_comp)
pca.fit(X_scaled)

# Direzioni principali / eigenvectors della matrice di covarianza
prin_dir = pca.components_
prin_dir.shape
```

```
Out[111]: (6, 22)
```

Notiamo che abbiamo ottenuto una matrice con un numero di righe pari a 6 (valore da noi scelto e che giustificheremo in seguito), che sono le direzioni di variabilità principali e un numero di colonne che corrisponde ai fattori originari coinvolti.

Quindi questa matrice ci consente di leggere la relazione tra:

- **Fattori originari**
- **Componenti principali**

in termini di:

- **Segno** --> se un fattore originario cresce nella stessa direzione di crescita della componente principale o meno
- **Magnitudo** --> la grandezza della singola entry del DataFrame sottostante mi consente di capire quanto ciascun fattore originario "entra", ovvero è allineato a ciascuna componente principale

```
In [113]: df_principal_directions = pd.DataFrame(prin_dir.T,
                                              columns = ['EigenV' + str(j+1) for j in range(prin_dir.shape[0])],
                                              index = numeric_cols)

print('Principal directions explored:\n')
df_principal_directions
```

```
Principal directions explored:
```

Out[113...]

	EigenV1	EigenV2	EigenV3	EigenV4	EigenV5	EigenV6
length_ms	0.19029	-0.21519	0.18030	0.04122	-0.13937	-0.38954
zero_crossings_sum	0.21040	0.02007	-0.11875	0.44598	-0.27742	-0.13222
mfcc_mean	0.16502	0.34770	0.10739	-0.29785	-0.02479	-0.18165
mfcc_std	-0.33141	-0.12438	0.12550	0.05246	0.03933	0.07157
mfcc_min	0.30943	0.14586	-0.19008	-0.07664	-0.00769	-0.02527
mfcc_max	-0.20746	0.15073	0.12800	-0.31461	0.19061	-0.32657
sc_mean	-0.22772	0.12132	-0.37850	0.20325	-0.15599	-0.02400
sc_std	-0.21519	-0.20219	-0.32494	-0.08270	0.15333	-0.19501
sc_min	0.01141	0.27641	0.14201	0.15005	-0.47337	-0.22436
sc_max	-0.10371	-0.13615	-0.30882	0.12776	0.21301	-0.30724
sc_kur	0.23154	0.02090	0.29676	0.00857	0.30816	0.07954
sc_skew	0.25723	-0.13657	0.27984	-0.16248	0.19563	-0.13124
stft_mean	-0.22879	0.33444	0.00033	-0.17093	-0.05735	-0.05541
stft_std	0.14564	-0.36033	-0.17317	-0.24186	-0.13931	0.14996
stft_min	-0.14326	0.25343	0.19584	0.13316	0.06885	-0.26322
stft_kur	-0.01292	0.14233	0.21000	0.57841	0.34849	0.03963
stft_skew	0.24714	-0.28390	0.08646	0.20483	0.07113	-0.01392
std	0.29564	0.18874	-0.19044	-0.01207	0.00786	0.00873
min	-0.27570	-0.22873	0.20942	0.03984	-0.05469	-0.11358
max	0.27463	0.22649	-0.21861	-0.03659	0.08803	0.06890
kur	-0.09192	0.23934	-0.10974	0.05303	0.33820	0.38774
skew	0.11059	-0.00331	-0.28802	0.04213	0.36797	-0.46239

proiettiamo ogni osservazione X_j , nello spazio delle componenti principali:

In [114...]

```
# Osservazioni nel SR delle componenti principali
X_pca = pca.transform(X_scaled)
```

Nella cella sottostante troviamo giustificazione al numero di componenti principali sino ad ora scelto, ovvero 6.

Infatti ogni **direzione di variabilità (autovettore della matrice di covarianza)** si porta dietro un certo **autovalore λ** . Dividendo ciascun autovalore per la somma di tutti i possibili autovalori associati a tutte le direzioni di variabilità, otteniamo la **proporzione di varianza spiegata** da ciascuna componente principale:

$$\text{Explained - Variance - Ratio}_j = \frac{\lambda_j}{\lambda_1 + \dots + \lambda_d}$$

In [115...]

```
# percentage of explained variance (lambda_k) / (lambda_1 + ... + lambda_d)
prop_explained_variance = pca.explained_variance_ratio_
print('Proportion of explaine variance of each component:\n', prop_explained_variance, '\n\n')
```

Proportion of explaine variance of each component:
[0.33710647 0.19594283 0.11604689 0.06379365 0.05722639 0.04114389]

Considerando le prime 6 componenti principali totalizziamo una **proporzione di varianza spiegata cumulata** pari all'80%.

Questo significa che abbiamo notevolmente semplificato il problema in quanto **da oltre 20 fattori originari**, siamo passati a **sole 6 componenti principali, perdendo un contenuto informativo del solo 20%** circa.

$$\text{Explained - Variance - Ratio}_{\text{First-}k-\text{Components}} = \frac{\lambda_1 + \dots + \lambda_6}{\lambda_1 + \dots + \lambda_d} = 0.81$$

In [117...]

```
# cumulative percentage of explained variance (lambda_1 + lambda_k) / (lambda_1 + ... + lambda_d)
cum_prop_explained_variance = np.cumsum(prop_explained_variance)
print('Proportion of explaine variance of the first k-component:\n', cum_prop_explained_variance)
```

Proportion of explaine variance of the first k-component:
[0.33710647 0.5330493 0.64909619 0.71289893 0.77011622 0.81126011]

Di seguito visualizziamo lo **Scree Plot** che permette di percepire il gain di contenuto informativo che ciascuna componente principale si

porta dietro.

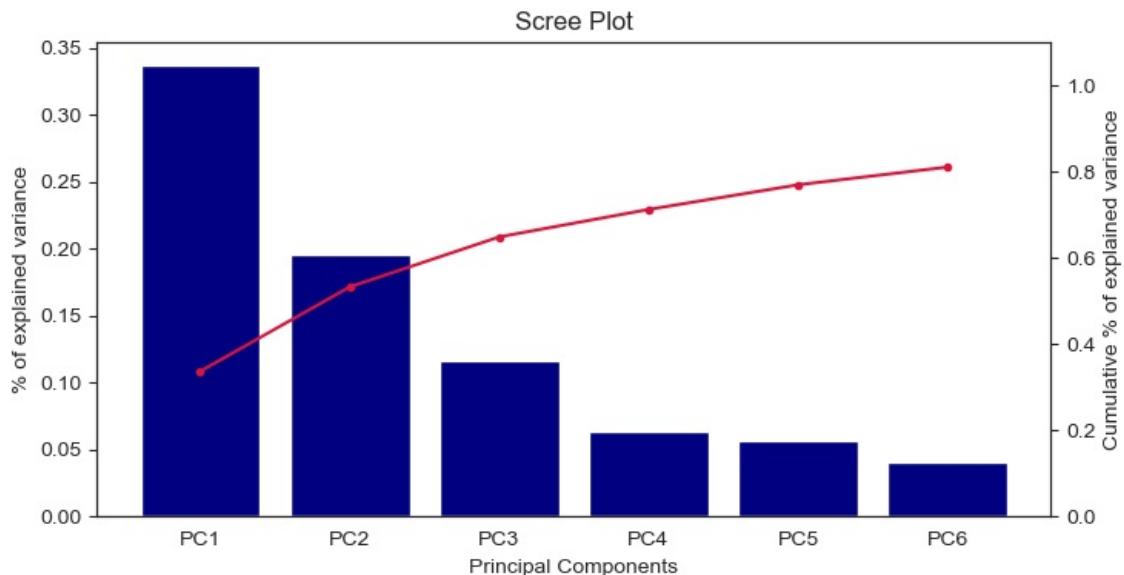
```
In [118]: fig, ax = plt.subplots(figsize = (8, 4))

axt = ax.twinx()

labels = ['PC' + str(j) for j in range(1, len(prop_explained_variance) + 1)]
ax.bar(labels, prop_explained_variance, color = 'navy')

axt.plot(labels, cum_prop_explained_variance, color = 'crimson', marker = '.')
ax.set_xlabel('Principal Components')
ax.set_ylabel('% of explained variance')
axt.set_ylabel('Cumulative % of explained variance')
ax.set_title('Scree Plot')

axt.set_ylim([0, 1.1])
axt.grid(False)
plt.show()
```



Cerchiamo adesso di dare un'interpretazione alle **Componenti Principali**, osservando la disposizione dei punti all'interno dello spazio che esse generano, con particolare focus sulla particolare **emotion** di ogni osservazione

```
In [121]: from mpl_toolkits.mplot3d import Axes3D

x = 0
y = 1
z = 2

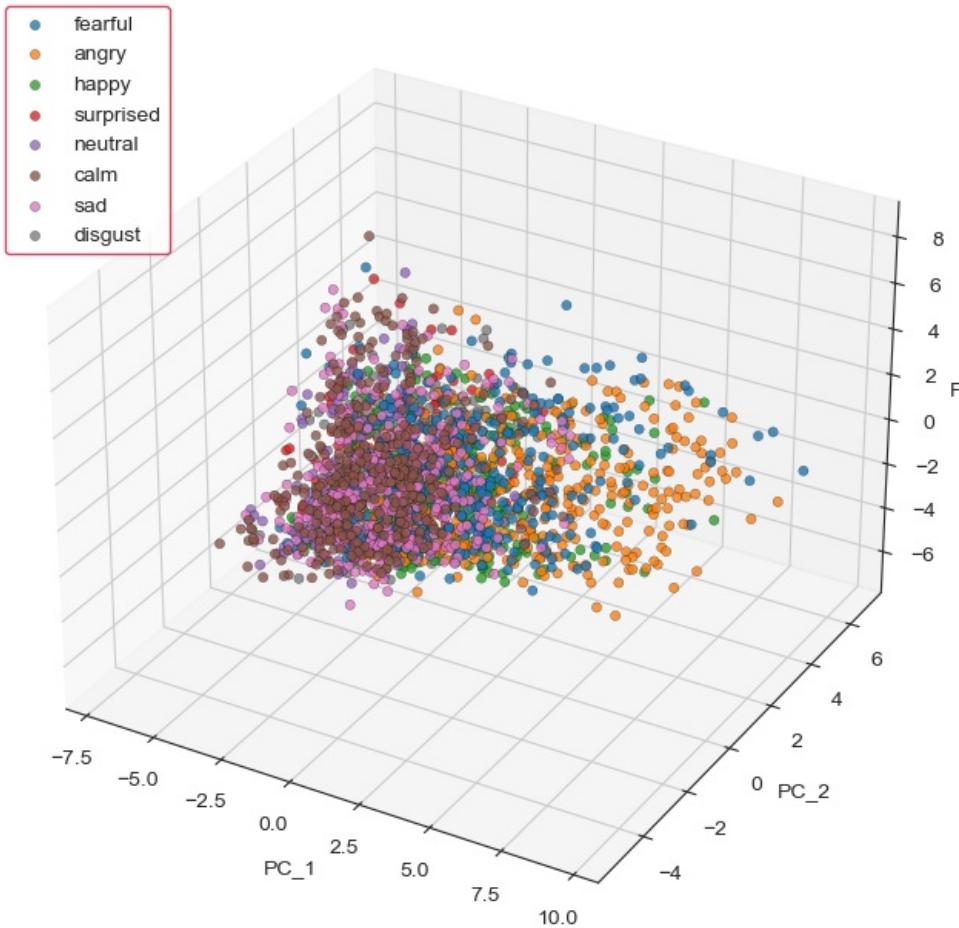
fig = plt.figure(figsize = (30, 8))
ax = fig.add_subplot(111, projection = '3d')

# X_pca e df hanno gli stessi indici
for emotion in df['emotion'].unique():
    mask = df['emotion'] == emotion
    scatter = ax.scatter(X_pca[mask, x],
                         X_pca[mask, y],
                         X_pca[mask, z],
                         label = emotion,
                         s = 20, alpha = 0.8, edgecolor = 'black', linewidth = 0.2, marker = 'o')

ax.set_xlabel(f'PC_{x + 1}')
ax.set_ylabel(f'PC_{y + 1}')
ax.set_zlabel(f'PC_{z + 1}')
ax.set_title(f'3D scatter of PC_{x+1} - PC_{y+1} - PC_{z+1}', fontsize = 14)
ax.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10, loc = 'upper left')

plt.show()
```

3D scatter of PC_1 - PC_2 - PC_3



Dal precedente plot osserviamo che:

- Emozioni che corrispondono ad una **forte intensità vocale** come **angry**, **fearful** ed **happy** sono in corrispondenza di **valori alti e positivi di PC_1**
- L'emozione **fearful** in realtà ha una forte dispersione in quanto la troviamo anche per **valori centrali di PC_1**. Volendo azzardare un'interpretazione potremmo dire che durante gli stati di paura, le persone modulano la voce passando da un registro vocale molto basso, tipico degli **stati calmi** a un registro vocale molto alto, tipico degli **stati di rabbia / euforia**. La predominanza dell'uno o dell'altro nell'interpretazione dell'attore potrebbe aver generato questo effetto
- Emozioni caratterizzate da **bassa intensità vocale** come **calm**, **sad**, **neutral** si trovano per valori bassi della PC_1, come anche **surprised**
- Emozioni caratterizzate da **alta intensità vocale** come **happy** ed **angry** sembrano prediligere valori alti di PC_2 mentre **neutral** e **sad** valori diametralmente opposti della PC_2

Per validare queste prime intuizioni, facciamo uno step aggiuntivo e cerchiamo di capire *come i fattori originari si dispongono rispetto alle componenti principali*. Per farlo realizziamo il **biplot**, un grafico che permette di visualizzare i **predittori originari** come vettori nello spazio delle **componenti principali**.

In conclusione, come le componenti principali possono essere viste come combinazione lineare dei **predittori originari**, anche i predittori originari possono essere espressi come combinazione lineare delle **componenti principali**. Questo è molto utile a tracciare il link tra i due spazi vettoriali e **aggiungere contenuto semantico alle componenti principali**.

In [122]:

```
# %%BIPLOT

# PCs = X @ V, quindi X = PCs @ V.T. Considering only 2 PCs PC1 and PC2:
# PC1 @ V1 + PC2 @ V2 = X and for each factor Xj
# Xj = y1 * V1_j + y2 * V2_j
# So we can write the original factors as linear combinations of the principal components basis,
# having the principal_directions' coefficients as weights

def biplot(PC, principal_directions, list_variable_names, by_class_values = None, idx_1 = 0, idx_2 = 1):

    plt.figure(figsize=[12, 8])

    # idx_1, idx_2 are indexes of the PCs to be plotted (es. PC1 and PC_3)
    PC_1 = PC[:, idx_1] # first principal component
    PC_2 = PC[:, idx_2] # second principal component
```

```

PC_1 = PC_1 / (max(PC_1) - min(PC_1))
PC_2 = PC_2 / (max(PC_2) - min(PC_2))

for j in np.unique(by_class_values):
    mask = by_class_values == j
    plt.scatter(PC_1[mask], PC_2[mask], s = 20, label = str(j), edgecolor = 'black', linewidth = 0.2)

n = principal_directions.shape[0] # it's equal to the number of original factors

for j in range(n):
    plt.arrow(x = 0, y = 0, # coordinates of the arrow base
              dx = principal_directions[j, idx_1], # PC1 coordinate of arrow
              dy = principal_directions[j, idx_2], # PC2 coordinate of arrow
              color='black', head_width = 0.01, lw = 0.2)

    plt.text(principal_directions[j, idx_1] * 1.1, principal_directions[j, idx_2] * 1.1,
             list_variable_names[j], color='black', fontsize = 10)

plt.title("Biplot of factors in the principal components' space", fontsize = 16)
plt.xlabel('PC_1' + str(idx_1 + 1))
plt.ylabel('PC_2' + str(idx_2 + 1))
plt.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)

```

Semantica PC_1

Nei precedenti plot osserviamo che i fattori più allineati (con verso positivo) a PC_1 sono:

- zero_crossings_sum
- skew
- mfcc_min
- sc_skew
- sc_kur

Mentre con verso negativo:

- mfcc_std
- mfcc_max
- sc_mean

Inoltre confermiamo che solo **angry**, **happy** e **fearful** arrivano a toccare elevati valori di PC_1, mentre dal lato diametralmente opposto abbiamo le emozioni "deboli" come **sad**, **neutral** e **calm**.

Semantica PC_2

Nei precedenti plot osserviamo che i fattori più allineati (con verso positivo) a PC_2 sono:

- mfcc_mean
- sc_min
- stft_kur

Mentre con verso negativo:

- stft_std

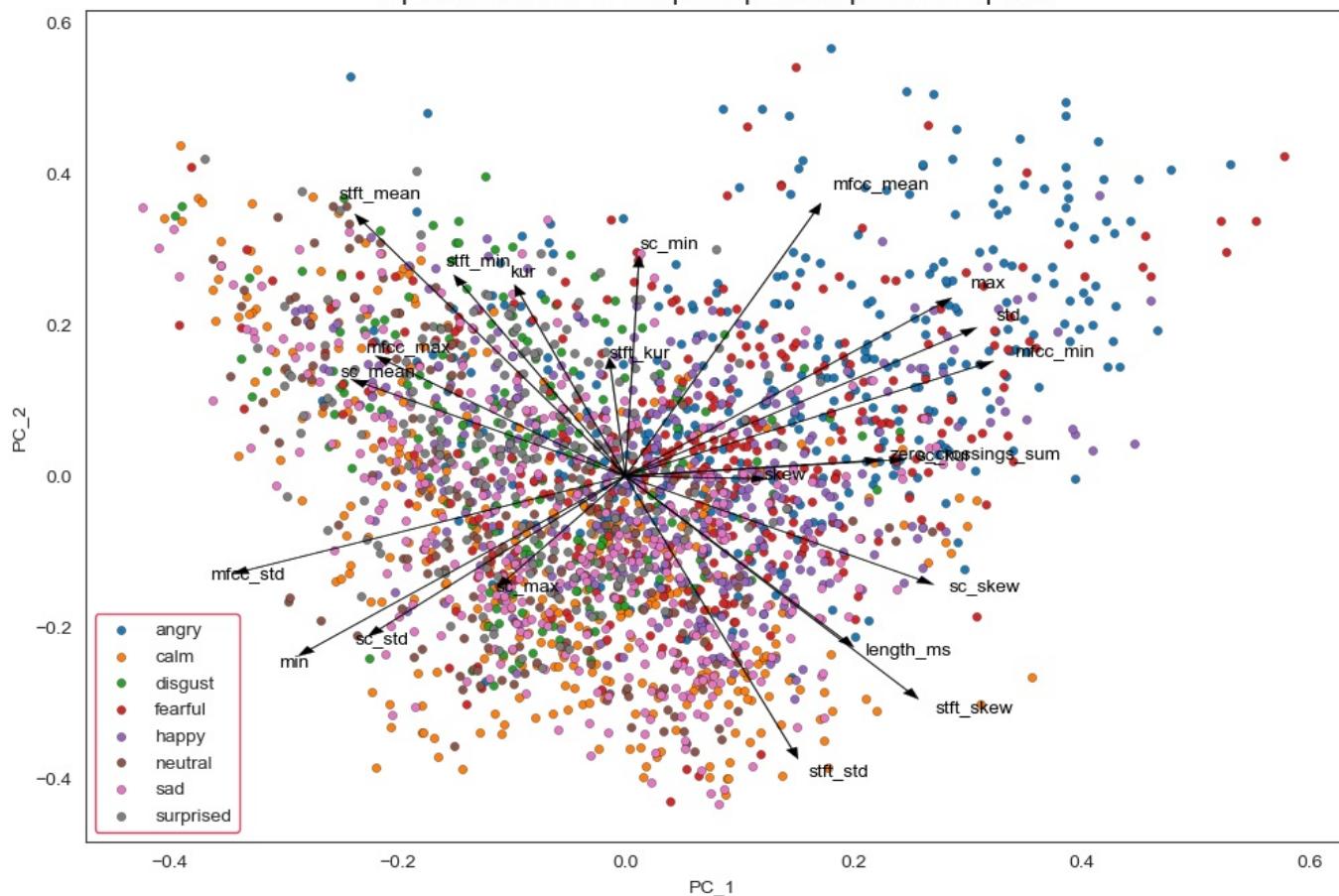
Ancora una volta solo **angry**, **happy** e **fearful** arrivano a toccare elevati valori di PC_2, mentre dal lato diametralmente opposto abbiamo le emozioni "deboli" come **sad**, **neutral** e **calm**, a cui si aggiunge anche **disgust**.

```

In [124]: biplot(PC = X_pca,
            # with the transposition the eigenvectors V are on columns
            principal_directions = prin_dir.T,
            list_variable_names = numeric_cols,
            by_class_values = df['emotion'].values,
            # insert index of principal direction
            idx_1 = 0, idx_2 = 1)

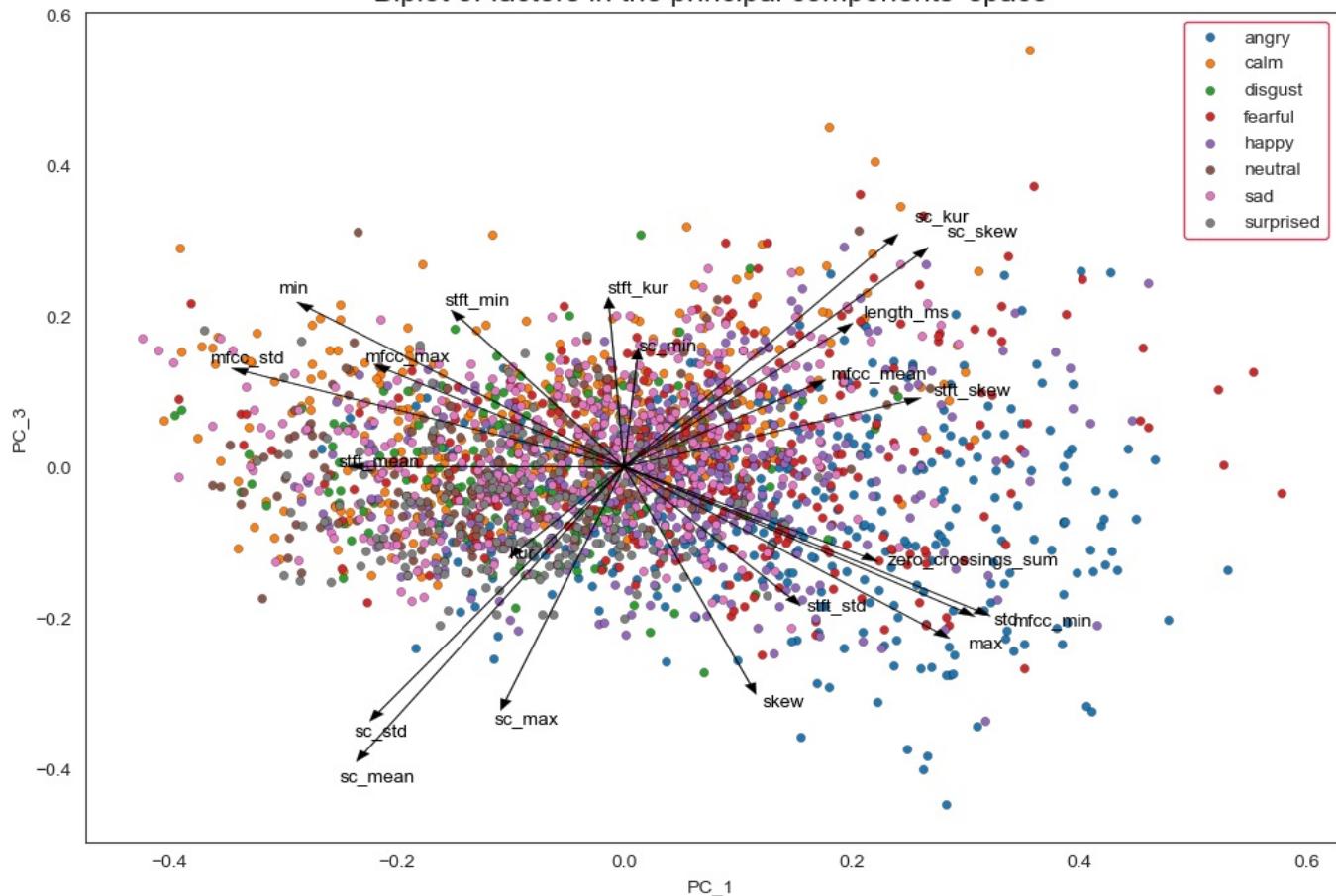
```

Biplot of factors in the principal components' space



```
In [125]: biplot(PC = X_pca,
           # with the transposition the eigenvectors V are on columns
           principal_directions = prin_dir.T,
           list_variable_names = numeric_cols,
           by_class_values = df['emotion'].values,
           idx_1 = 0, idx_2 = 2)
```

Biplot of factors in the principal components' space



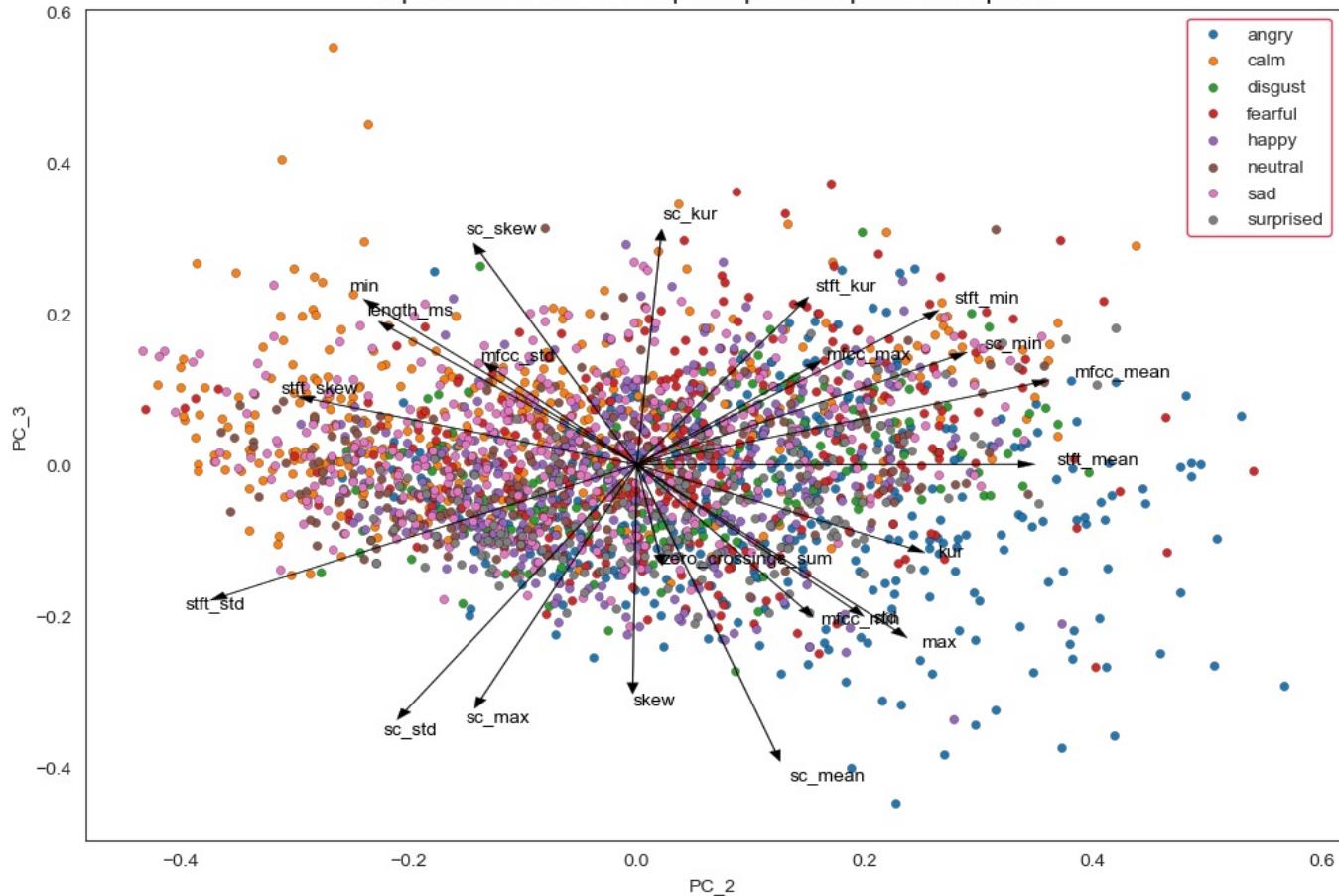
```
In [133]: biplot(PC = X_pca,
           # with the transposition the eigenvectors V are on columns
```

```

principal_directions = prin_dir.T,
list_variable_names = numeric_cols,
by_class_values = df['emotion'].values,
idx_1 = 1, idx_2 = 2)

```

Biplot of factors in the principal components' space



Possiamo usare quanto sopra per visualizzare un'altra dimensione, ovvero il **sex** (0 = 'F' e 1 = 'M').

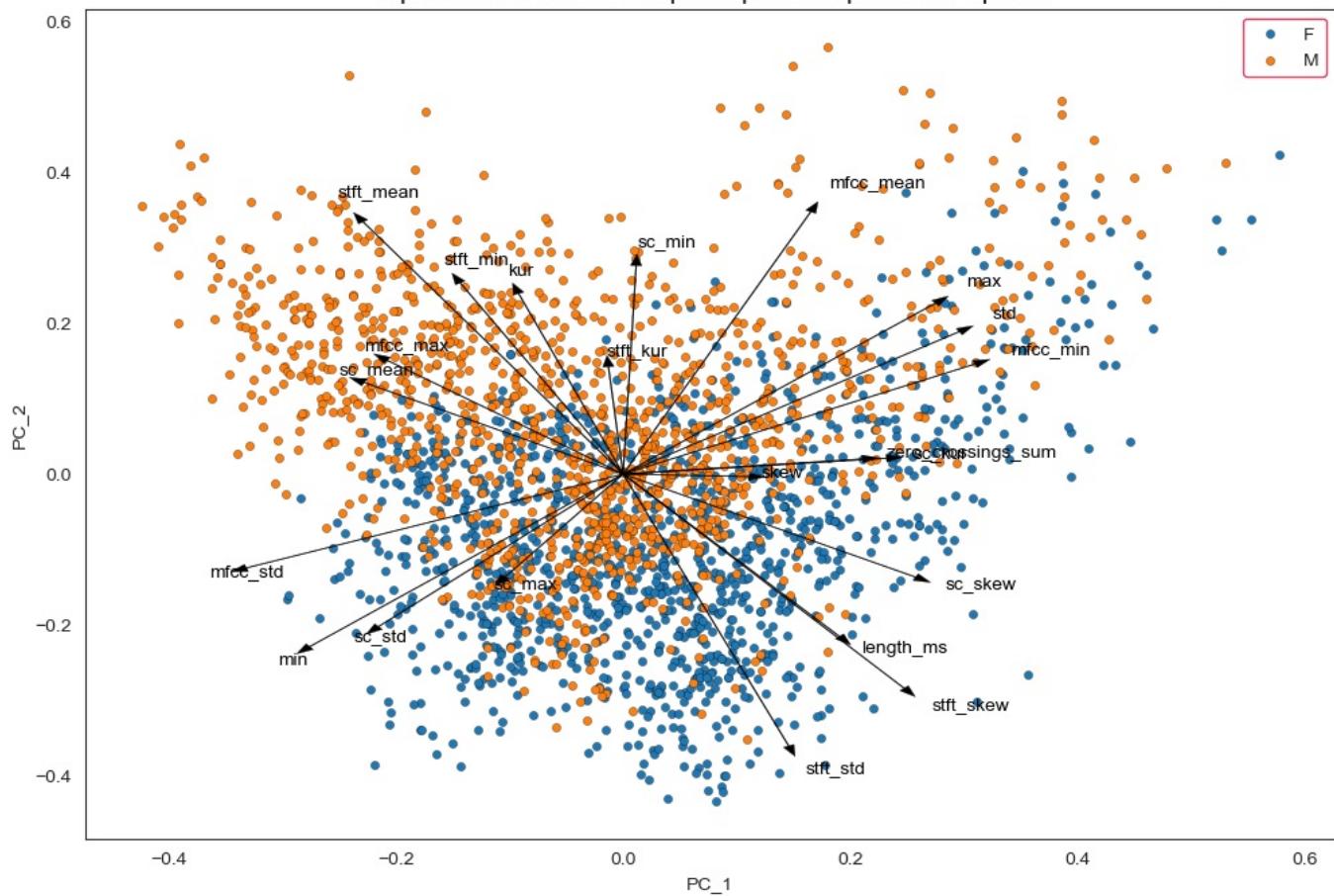
Sembra che PC_2 e PC_3 siano le dimensioni in cui possiamo meglio distinguere il **sex** e gli attributi che dominano questa separazione sono:

- stft_mean, mfcc_mean, mfcc_max, sc_min, stft_min --> valori elevati per uomini e bassi per le donne
- stft_skew, stft_std --> l'opposto

Questo lo avevamo già osservato in precedenza nei **violin plot**.

```
In [134]: biplot(PC = X_pca,
           # with the transposition the eigenvectors V are on columns
           principal_directions = prin_dir.T,
           list_variable_names = numeric_cols,
           by_class_values = df['sex'].apply(lambda x: {0: 'F', 1: 'M'}[x]).values,
           # insert index of principal direction
           idx_1 = 0, idx_2 = 1)
```

Biplot of factors in the principal components' space



6. Exporting Data Understanding & Preparation Output

```
In [132]: df.to_csv('A) Data Understanding & Preparation Output.csv', index = False)
```

```
In [ ]:
```

Processing math: 100%

DMML 2024: Final Project

Francesco Peria, Maria Paola Sforza Fogliani, Andrea Vitali

Importazione funzioni e processazioni propedeutiche

Per prima cosa importiamo il dataset che rappresenta l'output della sezione **A) Data Understanding & Preparation**

```
In [1]: import numpy as np
import pandas as pd
pd.set_option('display.max_columns', 100)
import matplotlib.pyplot as plt
import seaborn as sns
```

```
In [2]: df = pd.read_csv('A) Data Understanding & Preparation Output.csv', sep=',', skipinitialspace=True)
df.head()
```

```
Out[2]:   emotion  vocal_channel  emotional_intensity  statement  repetition  sex  length_ms  zero_crossings_sum  mfcc_mean  mfcc_std
0    fearful         0.0            0.0             0.0       1.0    0.0    3737.0      16995.0     -33.485947  134.65486
1     angry         0.0            0.0             0.0       0.0    0.0    3904.0      13906.0     -29.502108  130.48563
2    happy          0.0            1.0             0.0       1.0    0.0    4671.0      18723.0     -30.532463  126.57711
3  surprised         0.0            0.0             1.0       0.0    0.0    3637.0      11617.0     -36.059555  159.72516
4    happy          1.0            1.0             0.0       1.0    0.0    4404.0      15137.0     -31.405996  122.12582
```

Inoltre aggiungiamo alcuni step della PCA perché ci serviranno per l'interpretazione dei clusters.

```
In [3]: numeric_cols = ['length_ms',
                   'zero_crossings_sum',
                   'mfcc_mean', 'mfcc_std',
                   'mfcc_min', 'mfcc_max',
                   'sc_mean',
                   'sc_std',
                   'sc_min',
                   'sc_max',
                   'sc_kur',
                   'sc_skew',
                   'stft_mean',
                   'stft_std',
                   'stft_min',
                   'stft_kur',
                   'stft_skew',
                   'std',
                   'min',
                   'max',
                   'kur',
                   'skew']
```

```
In [4]: from sklearn.preprocessing import StandardScaler
X = df[numeric_cols].values
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
In [5]: from sklearn.decomposition import PCA
n_comp = 6
pca = PCA(n_components = n_comp)
pca.fit(X_scaled)

# Direzioni principali / eigenvectors della matrice di covarianza
prin_dir = pca.components_
prin_dir.shape
```

```
Out[5]: (6, 22)
```

```
In [6]: # Osservazioni nel SR delle componenti principali
X_pca = pca.transform(X_scaled)
```

```
In [7]: # %%BIPLOT
```

```

# PCs = X @ V, quindi X = PCs @ V_T. Considering only 2 PCs PC1 and PC2:
# PC1 @ V1 + PC2 @ V2 = X and for each factor Xj
# Xj = y1 * V1_j + y2 * V2_j
# So we can write the original factors as linear combinations of the principal components basis,
# having the principal_directions' coefficients as weights

def biplot(PC, principal_directions, list_variable_names, by_class_values = None, idx_1 = 0, idx_2 = 1):

    plt.figure(figsize=[10, 6])

    # idx_1, idx_2 are indexes of the PCs to be plotted (es. PC1 and PC_3)
    PC_1 = PC[:, idx_1] # first principal component
    PC_2 = PC[:, idx_2] # second principal component

    PC_1 = PC_1 / (max(PC_1) - min(PC_1))
    PC_2 = PC_2 / (max(PC_2) - min(PC_2))

    for j in np.unique(by_class_values):
        mask = by_class_values == j
        plt.scatter(PC_1[mask], PC_2[mask], s = 20, label = str(j), edgecolor = 'black', linewidth = 0.2)

    n = principal_directions.shape[0] # it's equal to the number of original factors

    for j in range(n):
        plt.arrow(x = 0, y = 0, # coordinates of the arrow base
                  dx = principal_directions[j, idx_1], # PC1 coordinate of arrow
                  dy = principal_directions[j, idx_2], # PC2 coordinate of arrow
                  color='black', head_width = 0.01, lw = 0.2)

        plt.text(principal_directions[j, idx_1] * 1.1, principal_directions[j, idx_2] * 1.1,
                 list_variable_names[j], color='black', fontsize = 10)

    plt.title("Biplot of factors in the principal components' space", fontsize = 16)
    plt.xlabel('PC_' + str(idx_1 + 1))
    plt.ylabel('PC_' + str(idx_2 + 1))
    plt.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)

```

B) CLUSTERING

1. Center-Based Clustering: K-Means

Procediamo adesso ad eseguire un **K-means clustering** che coinvolge tutte le **variabili numeriche continue** utilizzate precedentemente nella PCA

In [8]: df.columns

Out[8]: Index(['emotion', 'vocal_channel', 'emotional_intensity', 'statement',
 'repetition', 'sex', 'length_ms', 'zero_crossings_sum', 'mfcc_mean',
 'mfcc_std', 'mfcc_min', 'mfcc_max', 'sc_mean', 'sc_std', 'sc_min',
 'sc_max', 'sc_kur', 'sc_skew', 'stft_mean', 'stft_std', 'stft_min',
 'stft_kur', 'stft_skew', 'std', 'min', 'max', 'kur', 'skew'],
 dtype='object')

In [9]: numeric_cols = ['length_ms',
 'zero_crossings_sum',
 'mfcc_mean', 'mfcc_std',
 'mfcc_min', 'mfcc_max',
 'sc_mean',
 'sc_std',
 'sc_min',
 'sc_max',
 'sc_kur',
 'sc_skew',
 'stft_mean',
 'stft_std',
 'stft_min',
 'stft_kur',
 'stft_skew',
 'std',
 'min',
 'max',
 'kur',
 'skew']

In [10]: # Dataset di riferimento per clustering
df_cluster = df[numeric_cols + ['sex', 'emotion']].copy(deep = True)

Per evitare che il risultato sia affetto da differenze di scala, standardizzo il dataset ancora una volta con lo **Standard Scaler**

```
In [11]: from sklearn.preprocessing import StandardScaler  
  
# X è matrice già scalata con StandardScaler con le sole colonne numeriche da considerare come base per il cluster  
X = df_cluster[numeric_cols].values  
  
scaler = StandardScaler()  
X = scaler.fit_transform(X)  
X
```

```
Out[11]: array([[-0.59370043,  1.12146408, -1.05733963, ..., -0.2091685 ,  
                 -0.27170749,  0.70663217],  
                [-0.31452936,  0.27852801, -0.16429762, ...,  0.42569433,  
                 1.26425838,  0.74841435],  
                [ 0.96765158,  1.59300617, -0.39526838, ..., -0.06898976,  
                 -0.95588585,  0.77150368],  
                ...,  
                [ 1.91716757, -0.85148114,  0.59043138, ..., -0.42240248,  
                 -0.94332397,  0.09327539],  
                [-0.59370043, -0.86485242,  0.11799329, ..., -0.65390028,  
                 0.40108606, -0.13789756],  
                [-0.42653212, -0.94371565, -0.05605405, ..., -0.39289938,  
                 0.26766114,  2.37522233]])
```

Per determinare il numero corretto di clusters da considerare, andiamo a variarne iterativamente il numero, fino ad arrivare a un valore sufficientemente alto, come ad esempio 30 che rappresenta un valore di soglia discrezionale. Ad ogni iterazione salviamo in due liste il valore della **silhouette media** e il valore di **SSE (Sum of Squared Error)** ottenuto con un certo k.

L'idea è che quando la **silhouette media** è **sufficientemente alta** e il **SSE** è **sufficientemente basso** allora avremo trovato il k ideale, che permette di raggiungere il trade-off tra le due metriche.

Per il calcolo della silhouette possiamo dapprima calcolare la **matrice delle distanze D** e successivamente passarla alla **funzione silhouette_score** mettendo il parametro **metric = 'precomputed'**.

```
In [12]: from scipy.spatial.distance import pdist, squareform  
  
# Give the distance from each observation to each other observation in the dataset  
D = squareform(pdist(X))  
D
```

```
Out[12]: array([[0.        , 3.66167418, 2.94395837, ..., 5.46555793, 5.2764969 ,  
                 4.82515379],  
                [3.66167418, 0.        , 3.92486327, ..., 4.35324002, 4.96188907,  
                 4.26220702],  
                [2.94395837, 3.92486327, 0.        , ..., 4.76955893, 6.2976186 ,  
                 5.87639466],  
                ...,  
                [5.46555793, 4.35324002, 4.76955893, ..., 0.        , 4.88642629,  
                 4.78186685],  
                [5.2764969 , 4.96188907, 6.2976186 , ..., 4.88642629, 0.        ,  
                 3.2973708 ],  
                [4.82515379, 4.26220702, 5.87639466, ..., 4.78186685, 3.2973708 ,  
                 0.        ]])
```

```
In [13]: from sklearn.cluster import KMeans  
from sklearn.metrics import silhouette_score, silhouette_samples  
  
# liste in cui appendere gli score di SSE e SILHOUETTE MEDIA  
sse_list = []  
sil_list = []  
  
max_k = 30  
for k in range(2, max_k + 1):  
    kmeans = KMeans(n_clusters = k, n_init = 20, random_state = 0) # run per n_init volte  
    kmeans.fit(X)  
    sse_list.append(kmeans.inertia_)  
    sil_list.append(silhouette_score(D, kmeans.labels_, metric = 'precomputed'))
```

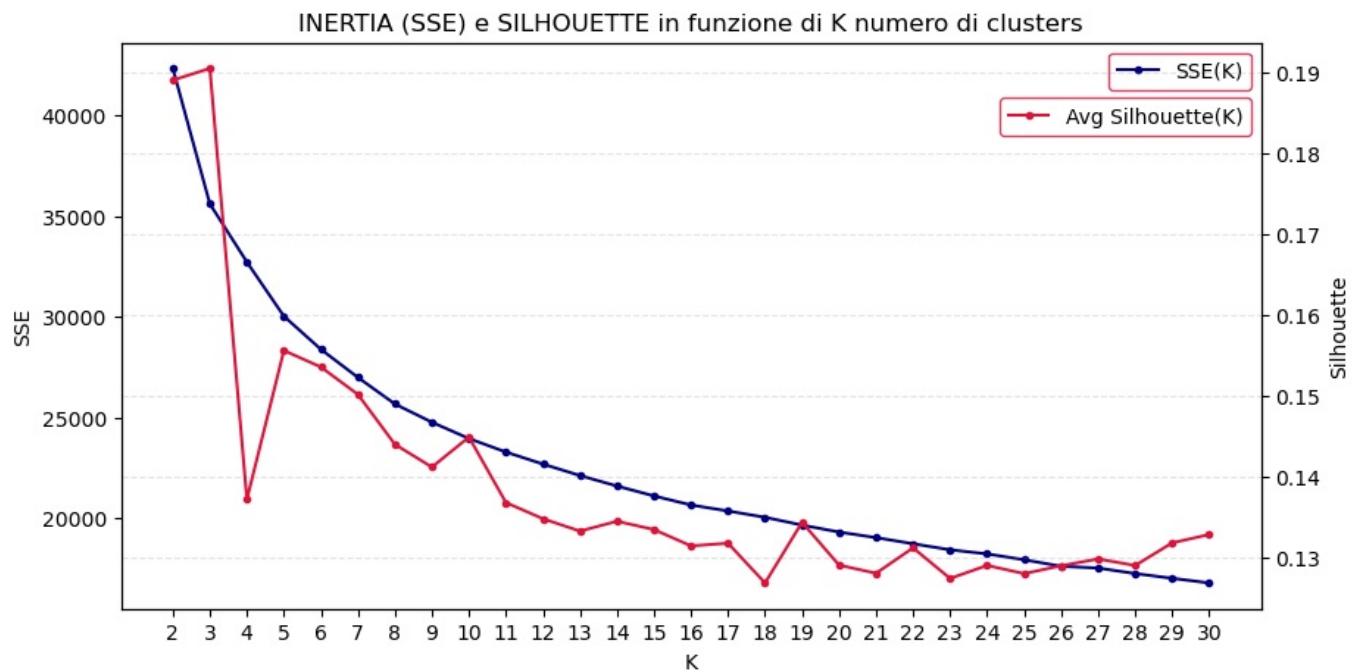
```
In [14]: fig, ax = plt.subplots(figsize = (10, 5))  
  
# INERTIA --> SSE  
ax.plot(range(2, max_k + 1), sse_list, marker = '.', c = 'navy', label = 'SSE(K)')  
ax.set_xlabel('K')  
ax.set_ylabel('SSE')  
plt.xticks(range(2, max_k + 1))  
  
# SILHOUETTE  
axt = ax.twinx()  
axt.plot(range(2, max_k + 1), sil_list, marker='.', c = 'crimson', label = 'Avg Silhouette(K)')  
axt.set_ylabel('Silhouette')
```

```

plt.xticks(range(2, max_k + 1))
plt.grid(color = 'grey', alpha = 0.2, ls = '--')
plt.title('INSERTIA (SSE) e SILHOUETTE in funzione di K numero di clusters')

ax.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)
axt.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10, bbox_to_anchor=(1, 0.92))
plt.show()

```



Sembra che il miglior numero di clusters sia tra 5 e 7. Tuttavia si preferisce scegliere 5 in quanto potrebbe essere più semplice interpretare un numero inferiore di cluster e i guadagni relativi di SSE nel passaggio da 5 a 7 sembrano irrisoni (meno dell'1%).

```
In [15]: kmeans = KMeans(n_clusters = 5, n_init = 10, random_state = 0)
kmeans.fit(X)
```

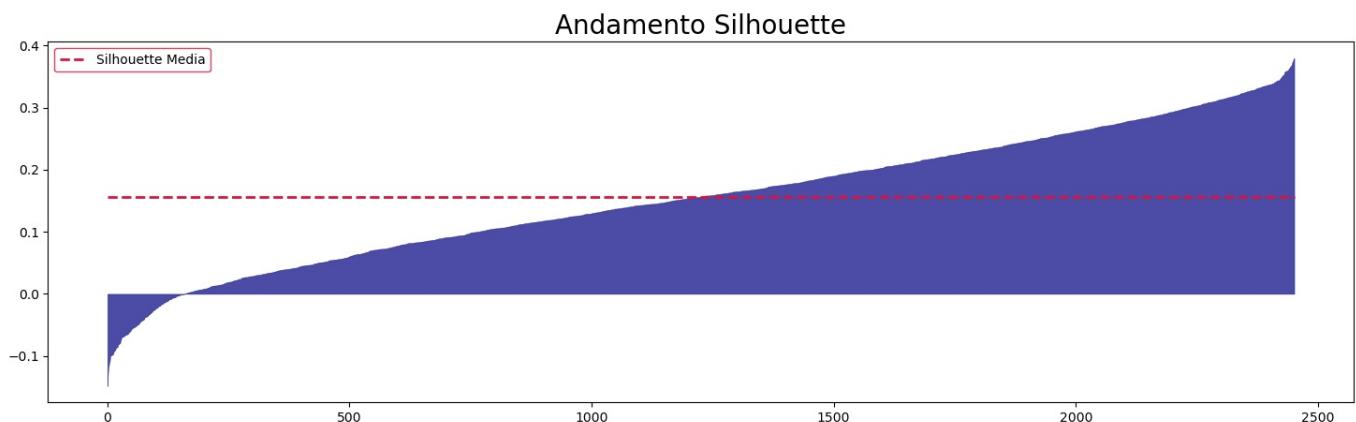
```
Out[15]: ▾ KMeans
KMeans(n_clusters=5, n_init=10, random_state=0)
```

Adesso cerchiamo di validare il risultato ottenuto visualizzando la silhouette per ciascuna delle osservazioni a seguito della scelta del k, per vedere se questo ha portato ad osservazioni clusterizzate particolarmente male, che quindi verranno evidenziate da un **elevato valore di silhouette**.

```
In [16]: sil_samples = sorted(silhouette_samples(D, kmeans.labels_, metric = 'precomputed'))

fig, ax = plt.subplots(figsize = (18, 5))

ax.fill_between(range(len(sil_samples)), sil_samples, color = 'navy', alpha = 0.7, lw = 0.4)
ax.plot([0, len(sil_samples)], np.mean(sil_samples)*np.ones(2), color = 'crimson', lw = 2, linestyle = '--', label='Silhouette Media')
ax.set_title('Andamento Silhouette', fontsize = 20)
ax.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10, loc = 'upper left')
plt.show()
```



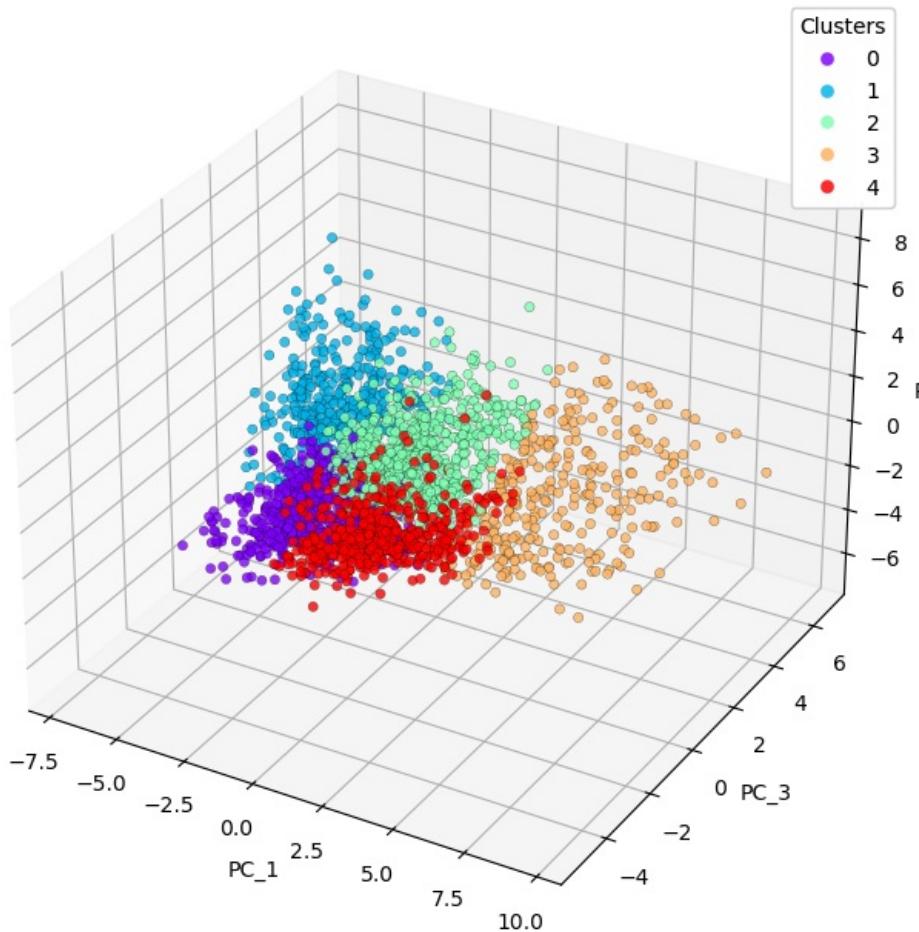
Il grafico conferma che la **silhouette** è negativa solo per un piccolo campione di osservazioni, inferiore al 10%.

Cerchiamo adesso una conferma della scelta effettuata andando ad analizzare specificamente ogni cluster. Andiamo quindi a plottare i risultati nello spazio delle **componenti principali**:

```
In [17]: x = 0  
y = 1  
z = 2
```

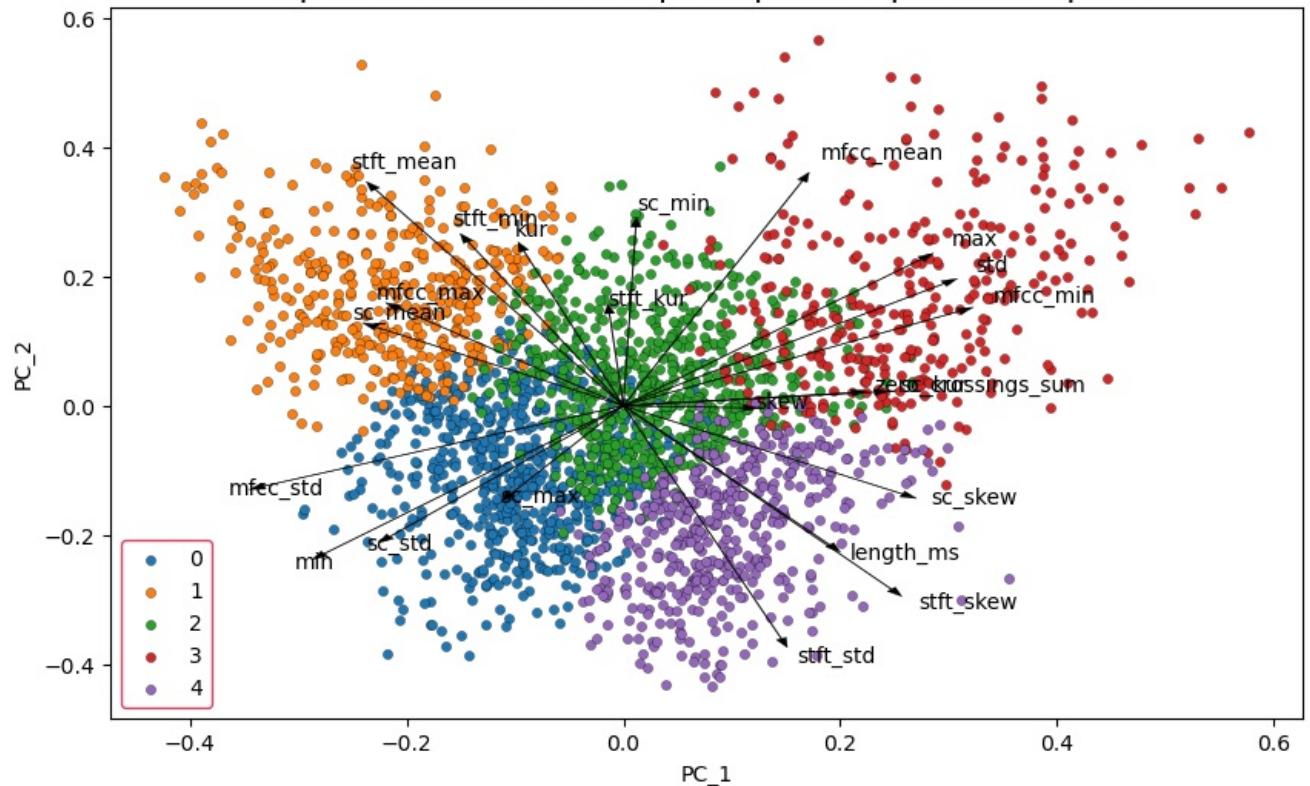
```
fig = plt.figure(figsize = (30, 8))  
ax = fig.add_subplot(projection = '3d')  
scatter = ax.scatter(X_pca[:, x], X_pca[:, y], X_pca[:, z], c = kmeans.labels_, cmap='rainbow',  
                     s = 20, alpha = 0.8, edgecolor = 'black', linewidth = 0.2, marker = 'o')  
  
ax.set_xlabel(f'PC_{x + 1}')  
ax.set_ylabel(f'PC_{y + 2}')  
ax.set_zlabel(f'PC_{z + 3}')  
ax.set_title('Clustering in PC space', fontsize = 14)  
  
legend1 = ax.legend(*scatter.legend_elements(), title="Clusters", loc = 'best')  
ax.add_artist(legend1)  
  
plt.show()
```

Clustering in PC space



```
In [18]: biplot(PC = X_pca,  
            # with the transposition the eigenvectors V are on columns  
            principal_directions = prin_dir.T,  
            list_variable_names = numeric_cols,  
            by_class_values = kmeans.labels_,  
            idx_1 = 0, idx_2 = 1)
```

Biplot of factors in the principal components' space



Di seguito alcune considerazioni preliminari:

- Le emozioni caratterizzate da una **forte intensità vocale**, come **angry**, **happy** e **fearful**, sono le sole a toccare elevati valori di PC_1, quindi avranno una maggiore incidenza nei clusters 2, 3 e 4.
- Al contrario le emozioni dove l'**intensità vocale è bassa**, come **sad**, **neutral** e **calm** si troveranno maggiormente nei cluster 0, 1 e 2.
- I cluster 0, 3 e 4 sono maggiormente rappresentati da donne, mentre 1, 2 da uomini.

Cerchiamo adesso riscontro dell'intuizione grafica andando ad analizzare le statistiche per ogni cluster. Per prima cosa visualizziamo una tabella riassuntiva incrociando le variabili numeriche analizzate con i cluster ottenuti:

```
In [19]: df_cluster['KM_clusters'] = kmeans.labels_
df_cluster.head()
```

	length_ms	zero_crossings_sum	mfcc_mean	mfcc_std	mfcc_min	mfcc_max	sc_mean	sc_std	sc_min	sc_max
0	3737.0	16995.0	-33.485947	134.654860	-755.22345	171.69092	5792.550744	3328.055457	0.0	13541.95902
1	3904.0	13906.0	-29.502108	130.485630	-713.98560	205.00770	5197.620555	4040.931570	0.0	12000.29044
2	4671.0	18723.0	-30.532463	126.577110	-726.06036	165.45653	4830.743037	3332.131300	0.0	12007.75117
3	3637.0	11617.0	-36.059555	159.725160	-842.94635	190.03609	5376.446478	4053.663073	0.0	12048.22389
4	4404.0	15137.0	-31.405996	122.125824	-700.70276	161.13400	5146.012474	3442.724109	0.0	12025.58270

Di seguito l'**andamento di media e deviazione standard** per i diversi attributi continui al variare del cluster.

```
In [20]: cluster_means = df_cluster.drop(columns = ['sex', 'emotion']).groupby('KM_clusters').mean().T
cluster_means.columns = [str(j) + '_mu' for j in cluster_means.columns]

cluster_std = df_cluster.drop(columns = ['sex', 'emotion']).groupby('KM_clusters').std().T
cluster_std.columns = [str(j) + '_std' for j in cluster_std.columns]

cluster_stats = pd.merge(cluster_means, cluster_std, left_index = True, right_index = True)
cluster_stats = cluster_stats[sorted(cluster_stats.columns)]

cluster_stats
```

	0_mu	0_std	1_mu	1_std	2_mu	2_std	3_mu	3_std
length_ms	3823.632558	403.142785	3552.037406	322.223371	4214.521463	513.086872	4188.455414	492.546464
zero_crossings_sum	11724.232558	3093.508657	10445.845387	2433.834678	12625.674086	2925.369685	16347.398089	4172.490572
mfcc_mean	-32.104590	3.191596	-28.601587	2.666362	-25.557483	2.277526	-23.387920	3.072005
mfcc_std	149.114706	14.490241	157.729974	13.814607	130.781464	10.684172	103.648295	7.445623
mfcc_min	-814.149578	76.856519	-849.491221	77.462645	-736.018061	55.629831	-597.626464	41.320402
mfcc_max	204.265113	18.549848	224.435588	19.341438	206.838019	18.157928	178.618373	21.229211
sc_mean	5580.158869	700.016775	6048.012886	670.852732	4747.337609	662.474962	4842.284922	844.518948
sc_std	3856.760836	346.410730	3551.368717	464.543624	3029.185491	479.065910	2809.764224	617.435559
sc_min	241.006708	444.487305	840.704042	379.911237	824.356848	318.413616	712.523691	569.324816
sc_max	12323.065099	845.244596	11969.151251	848.092994	11322.278057	965.564887	11387.807714	1176.919349
sc_kur	-1.422035	0.219885	-1.474855	0.182696	-1.066218	0.495942	-0.601164	0.871491
sc_skew	0.202336	0.225212	-0.043976	0.222581	0.458627	0.286313	0.576265	0.370450
stft_mean	0.473078	0.053085	0.596700	0.037672	0.489451	0.043722	0.451344	0.063985
stft_std	0.341540	0.016118	0.295201	0.017651	0.328695	0.015182	0.332275	0.018110
stft_min	0.000324	0.000886	0.009113	0.008396	0.002140	0.002301	0.001100	0.001931
stft_kur	-1.370606	0.099318	-1.046928	0.215536	-1.317924	0.124583	-1.213191	0.221114
stft_skew	0.067188	0.198436	-0.344787	0.197225	0.102138	0.196305	0.239141	0.284183
std	0.009495	0.006542	0.006781	0.005289	0.019475	0.008974	0.064473	0.024013
min	-0.079262	0.058793	-0.066038	0.056030	-0.154489	0.080401	-0.539403	0.193506
max	0.085925	0.067282	0.072187	0.060931	0.162944	0.088878	0.599322	0.211464
kur	11.724938	5.986940	16.593395	8.752359	9.634628	4.888467	12.562091	5.763623
skew	-0.021743	0.426531	-0.208001	0.640954	-0.174690	0.413566	0.221783	0.269413

Di seguito il supporto, ovvero il numero di osservazioni che risiedono in ogni cluster, al variare di **sesso** ed **emotion**.

Ricordare che:

- Sex = 0 --> Female
- Sex = 1 --> Male

In [21]:	cluster_count = pd.crosstab(index = df_cluster['sex'], columns = df_cluster['KM_clusters'], values = df_cluster['mfcc_mean'], aggfunc = 'count').fillna(0)
cluster_count	

KM_clusters	0	1	2	3	4
sex					
0.0	450	26	128	179	421
1.0	195	375	501	135	42

In [22]:	cluster_count = pd.crosstab(index = df_cluster['emotion'], columns = df_cluster['KM_clusters'], values = df_cluster['mfcc_mean'], aggfunc = 'count').fillna(0)
cluster_count	

KM_clusters	0	1	2	3	4
emotion					
angry	49.0	25.0	90.0	173.0	39.0
calm	116.0	84.0	67.0	1.0	108.0
disgust	75.0	58.0	40.0	5.0	14.0
fearful	57.0	36.0	128.0	80.0	75.0
happy	77.0	38.0	129.0	46.0	86.0
neutral	66.0	48.0	31.0	0.0	43.0
sad	111.0	60.0	105.0	6.0	94.0
surprised	94.0	52.0	39.0	3.0	4.0

Questa tabella permette di confermare le supposizioni grafiche viste in precedenza:

- Le donne (**Sex = 0**) prevalgono nei **cluster 0, 3 e 4**
- Gli uomini (**Sex = 1**) prevalgono nei **cluster 1 e 2**
- Il **cluster 3** è il più bilanciato sull'attributo Sex
- Le emozioni caratterizzate da una forte intensità vocale, come **angry, fearful ed happy** prevalgono nei **cluster 2, 3**
- Le emozioni **deboli** come **calm, neutral, sad, surprised** prevalgono nei **cluster 0 e 2**
- Il **cluster 1** è molto **bilanciato** in termini di emozioni

```
In [23]: cluster_count = pd.crosstab(index = [df_cluster['sex'], df_cluster['emotion']],
                                    columns = df_cluster['KM_clusters'],
                                    values = df_cluster['mfcc_mean'],
                                    aggfunc = 'count').fillna(0)

cluster_count
```

KM_clusters	0	1	2	3	4	
sex	emotion					
0.0	angry	39.0	0.0	23.0	84.0	38.0
	calm	77.0	10.0	7.0	1.0	89.0
	disgust	62.0	4.0	12.0	4.0	14.0
	fearful	36.0	1.0	23.0	52.0	72.0
	happy	51.0	2.0	20.0	33.0	78.0
	neutral	42.0	4.0	4.0	0.0	42.0
	sad	71.0	4.0	22.0	3.0	84.0
	surprised	72.0	1.0	17.0	2.0	4.0
1.0	angry	10.0	25.0	67.0	89.0	1.0
	calm	39.0	74.0	60.0	0.0	19.0
	disgust	13.0	54.0	28.0	1.0	0.0
	fearful	21.0	35.0	105.0	28.0	3.0
	happy	26.0	36.0	109.0	13.0	8.0
	neutral	24.0	44.0	27.0	0.0	1.0
	sad	40.0	56.0	83.0	3.0	10.0
	surprised	22.0	51.0	22.0	1.0	0.0

Per semplicità di analisi plottiamo graficamente le **distribuzioni delle variabili** al variare dei **cluster** sfruttando diversi **violinplot**, uno per ogni variabile.

Inoltre aggiungiamo il **violino Overall** relativo alla distribuzione totale di ogni specifica variabile, senza differenziazione sul cluster.

```
In [24]: target = 'KM_clusters'

for col in numeric_cols:

    fig, ax = plt.subplots(figsize = (14, 5))
```

```

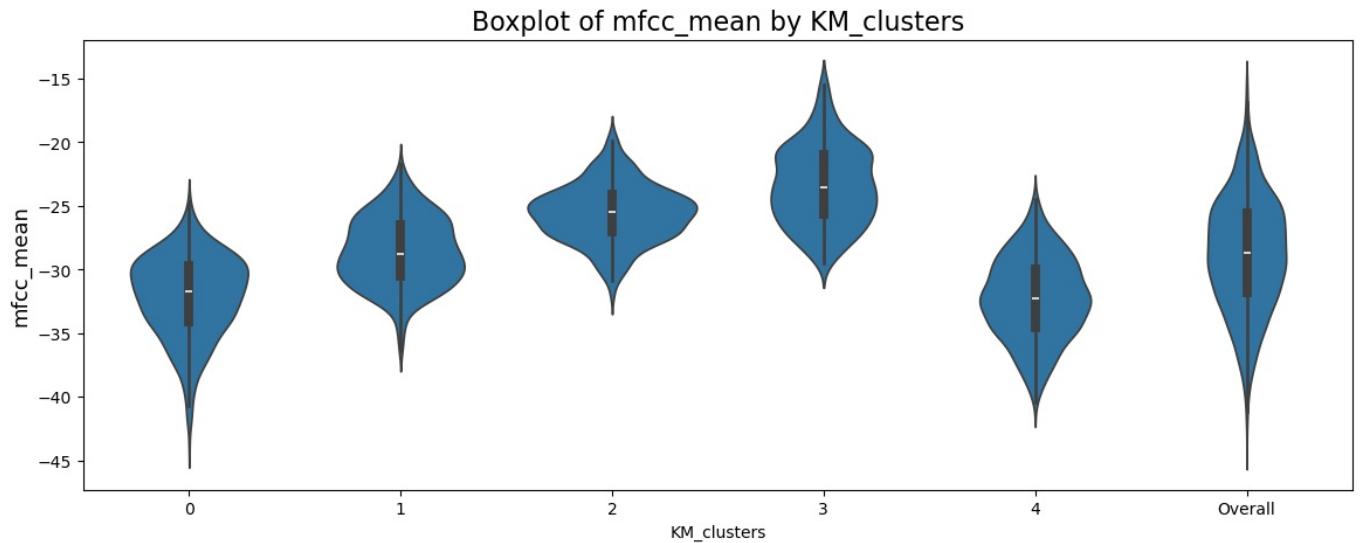
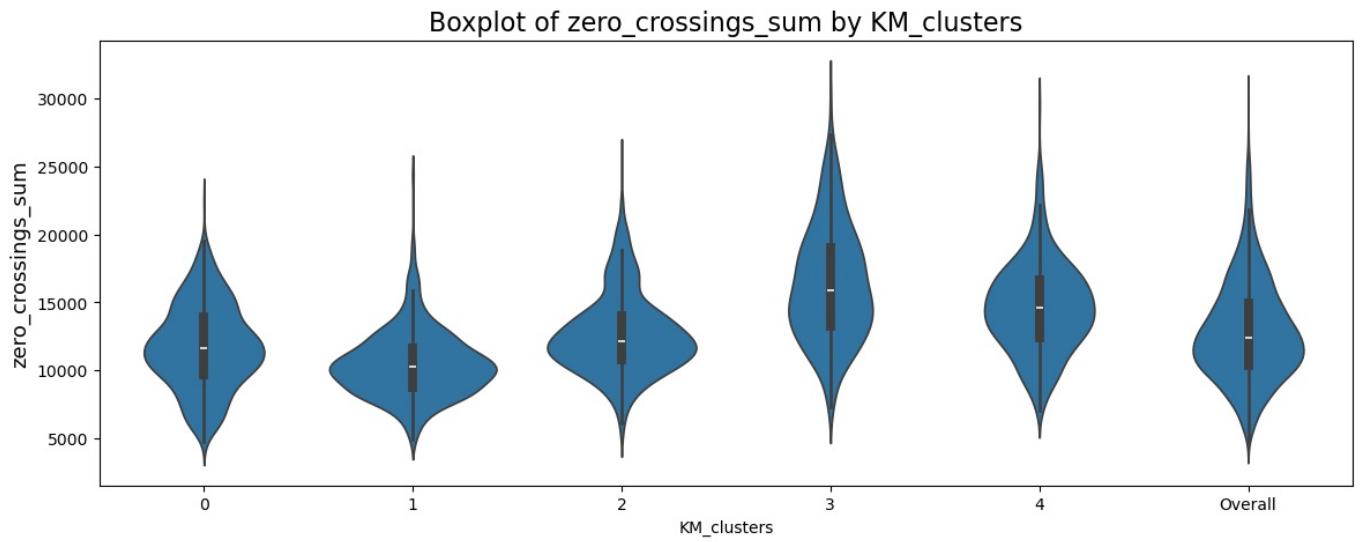
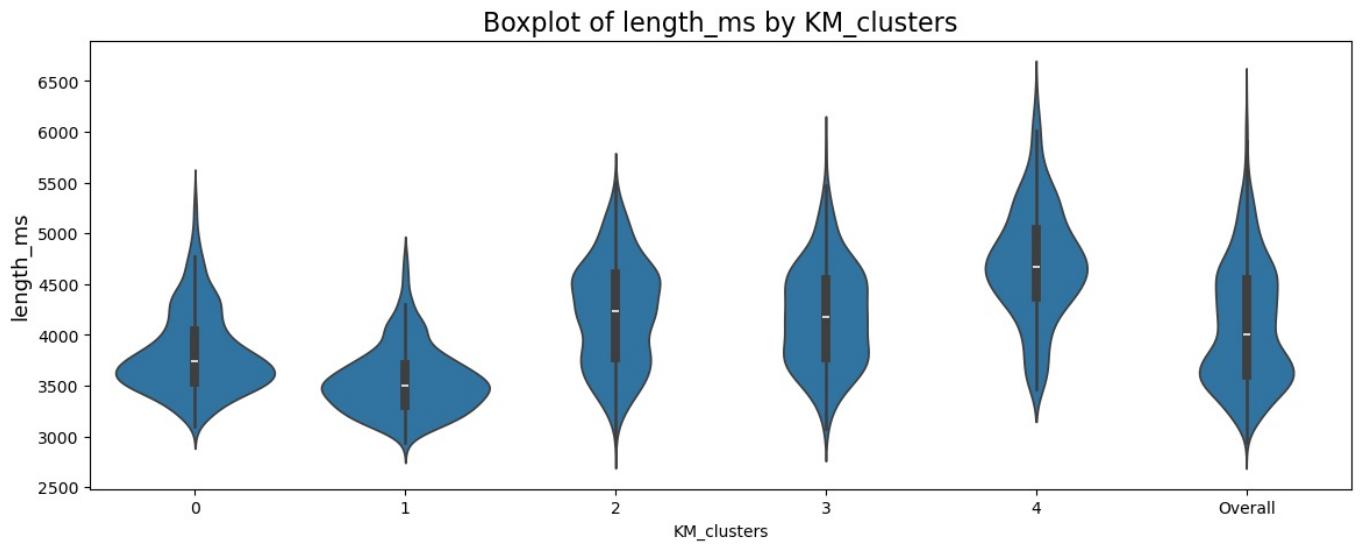
df_temp = df_cluster[[col, target]].melt(id_vars = [target])
df_temp = df_temp.sort_values(by = target)

# Mi ricreò la distribuzione della variabile non suddivisa sui cluster, ma su tutto il dataset originario
overall_dist = df[col].copy(deep = True).rename('value').to_frame()
overall_dist['variable'] = col
overall_dist[target] = 'Overall'
overall_dist = overall_dist[[target, 'variable', 'value']]
df_temp = pd.concat([df_temp, overall_dist], axis = 0).reset_index()

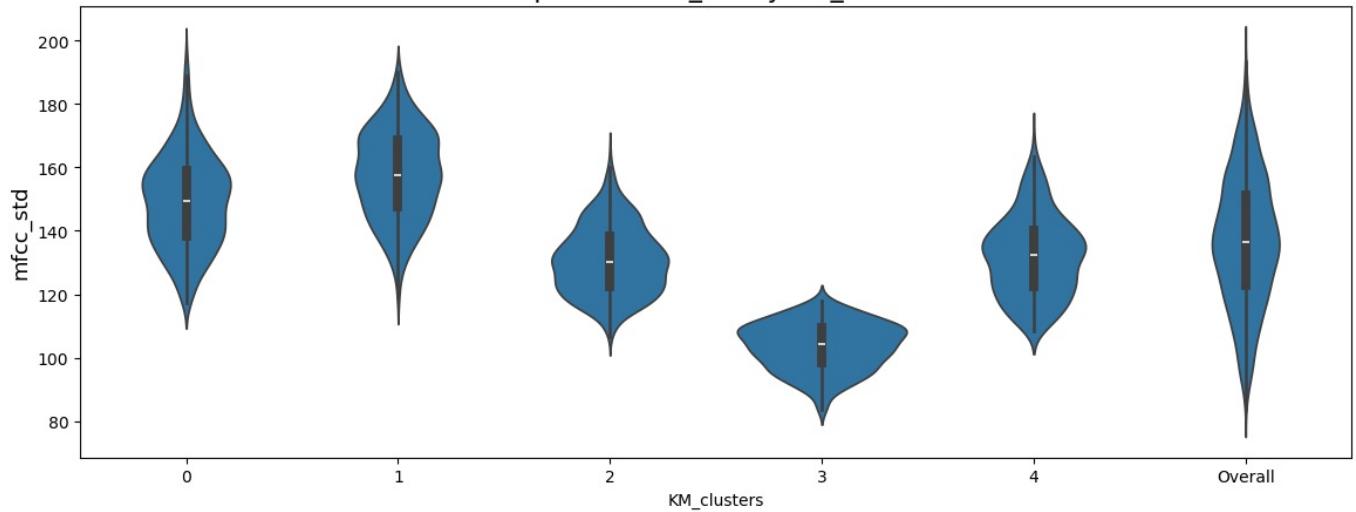
sns.violinplot(x = target, y = 'value', data = df_temp, ax = ax, split = False, inner = 'box')
ax.set_title(f'Boxplot of {col} by {target}', fontsize = 16)
ax.set_ylabel(col, fontsize = 13)

plt.show()

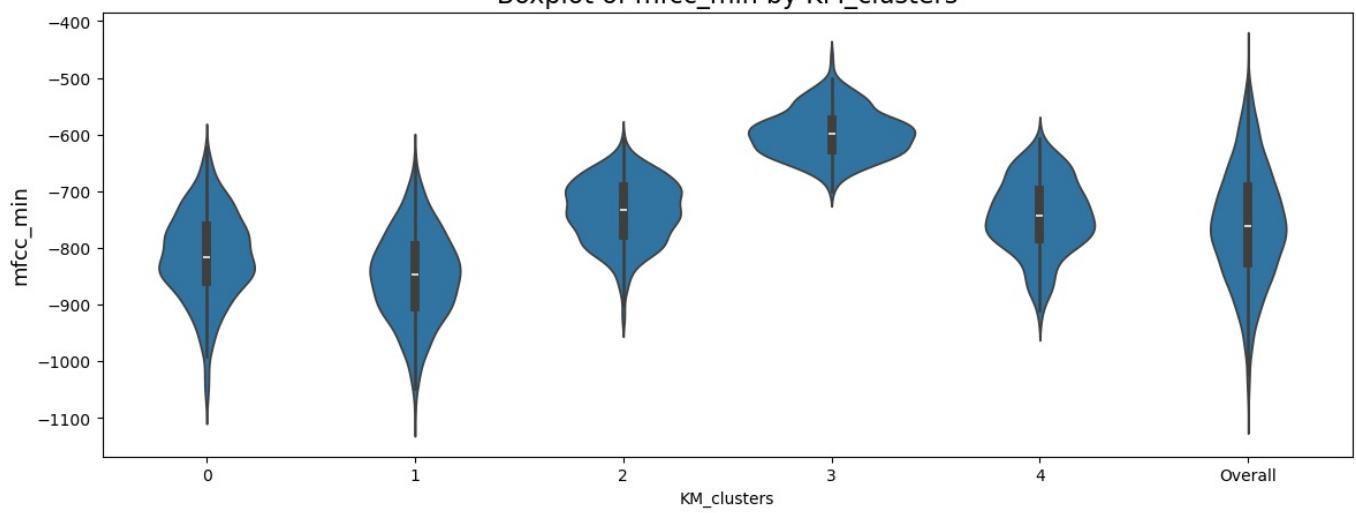
```



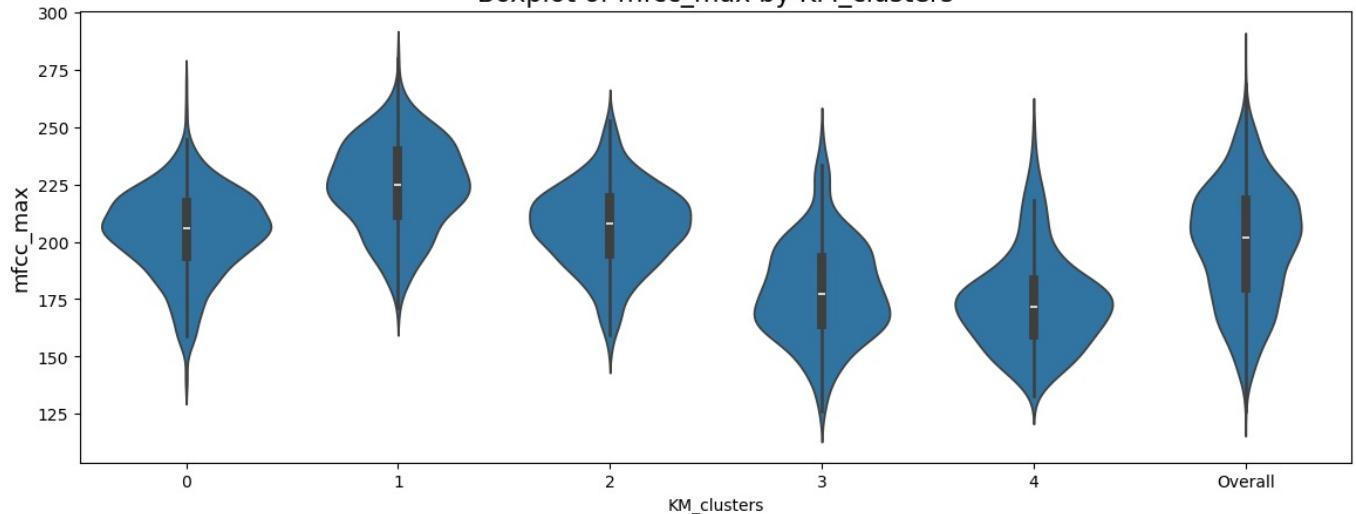
Boxplot of mfcc_std by KM_clusters



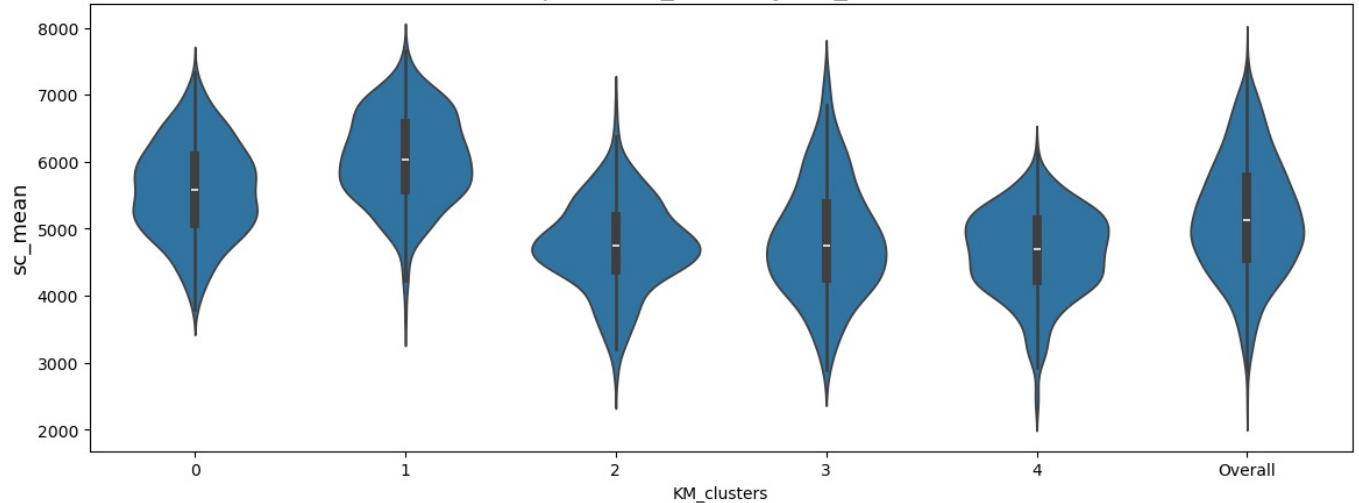
Boxplot of mfcc_min by KM_clusters



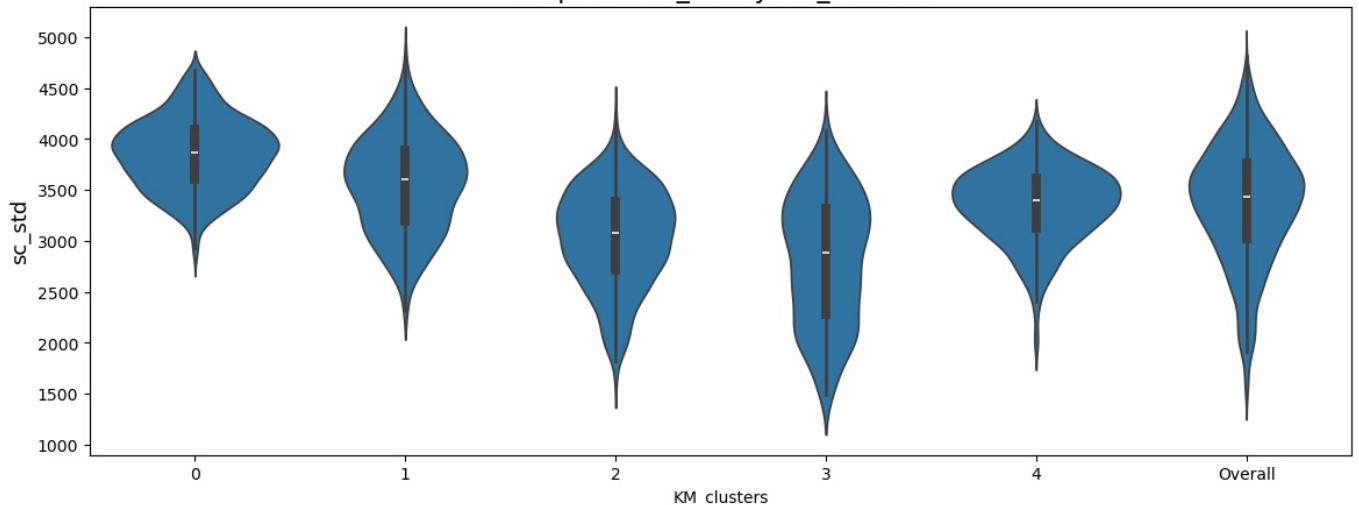
Boxplot of mfcc_max by KM_clusters



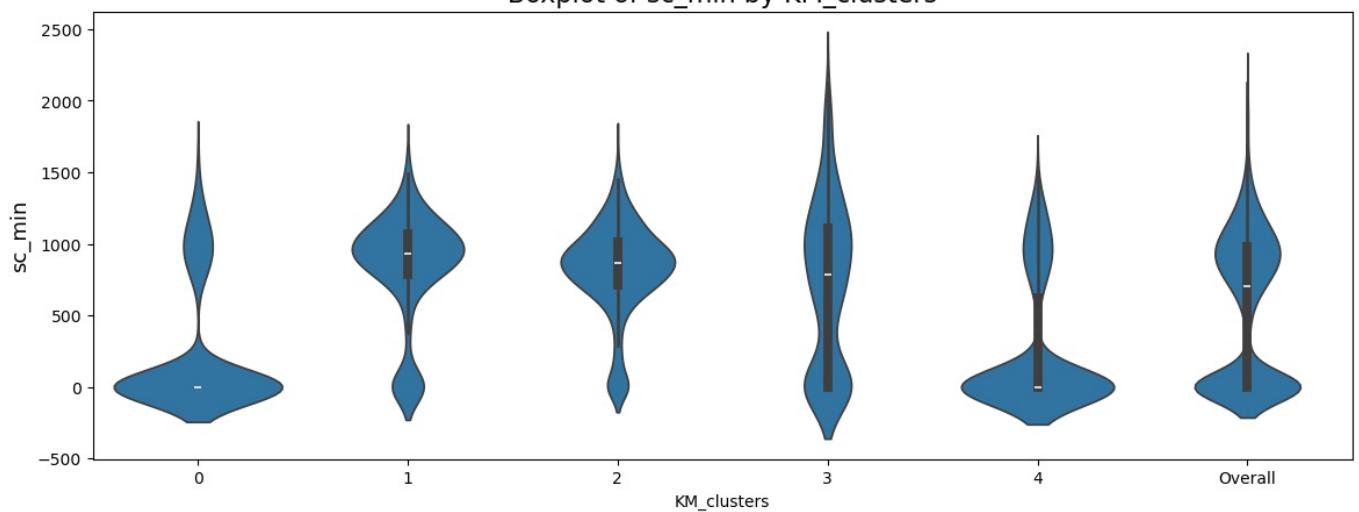
Boxplot of sc_mean by KM_clusters



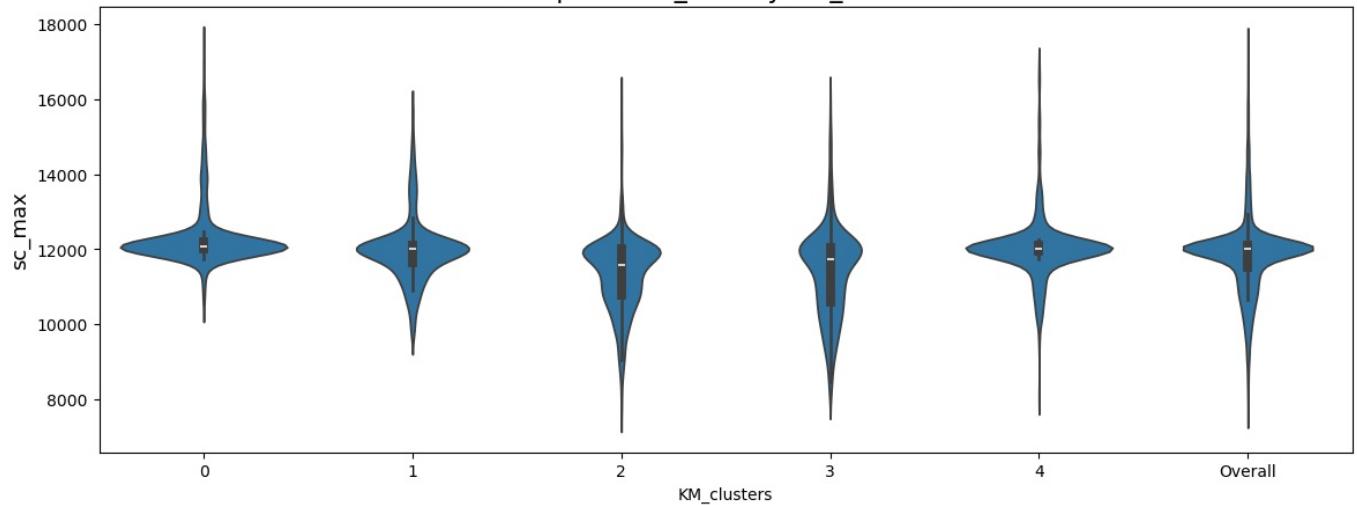
Boxplot of sc_std by KM_clusters



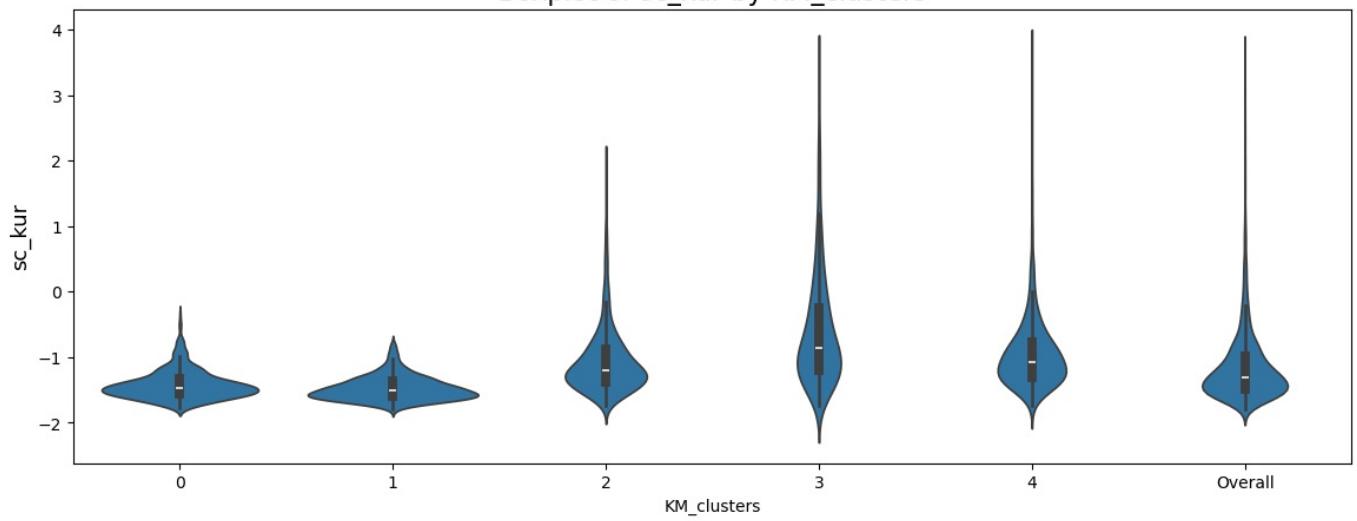
Boxplot of sc_min by KM_clusters



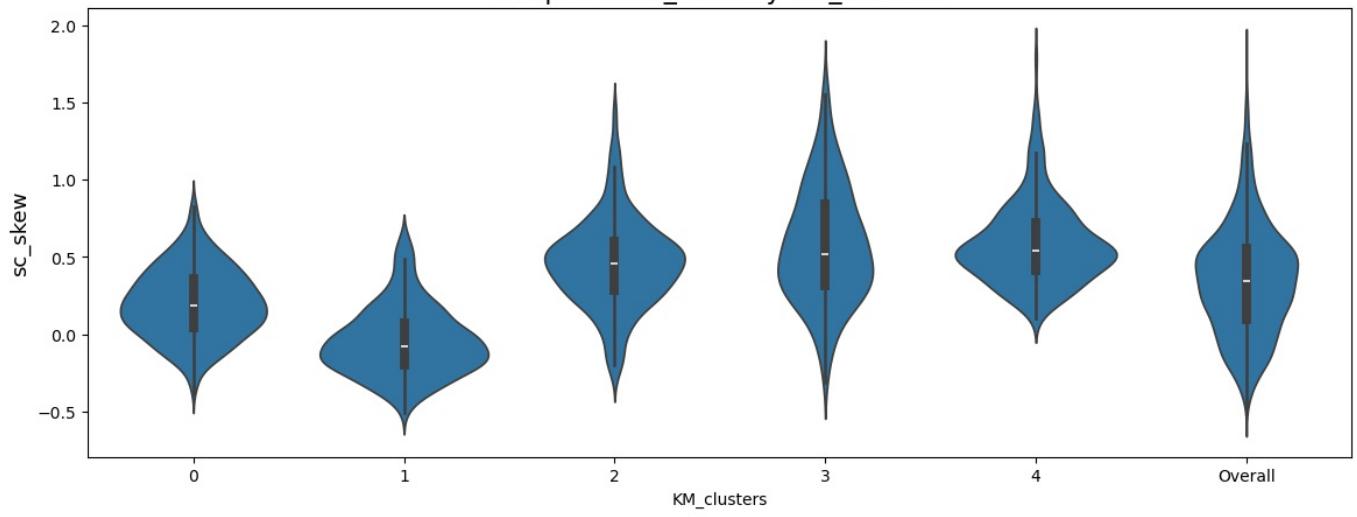
Boxplot of sc_max by KM_clusters



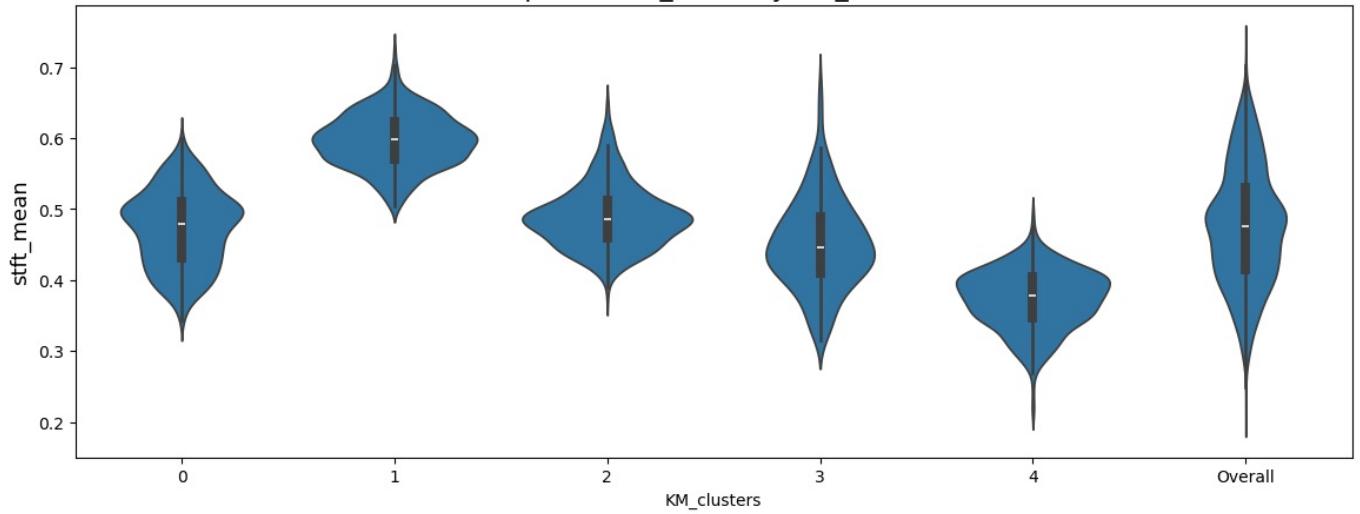
Boxplot of sc_kur by KM_clusters



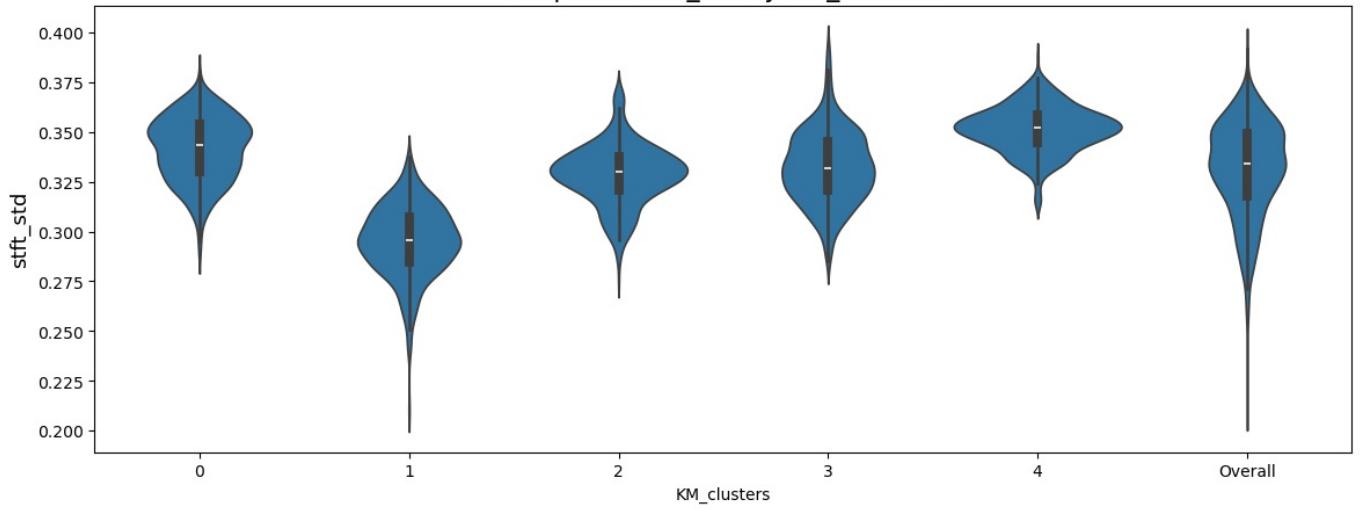
Boxplot of sc_skew by KM_clusters



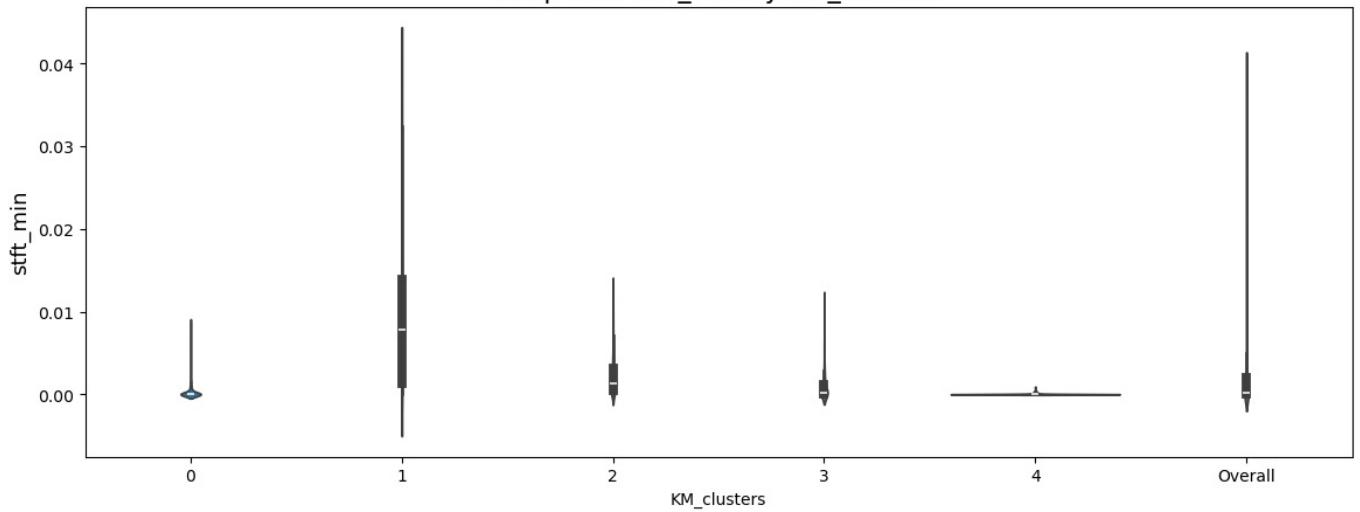
Boxplot of stft_mean by KM_clusters



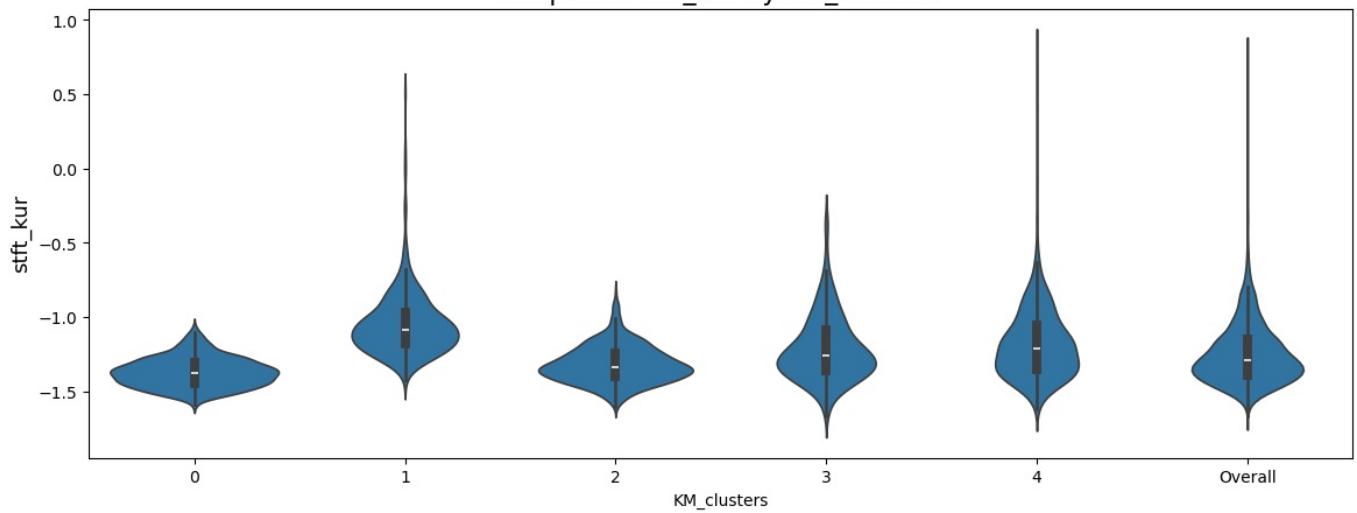
Boxplot of stft_std by KM_clusters



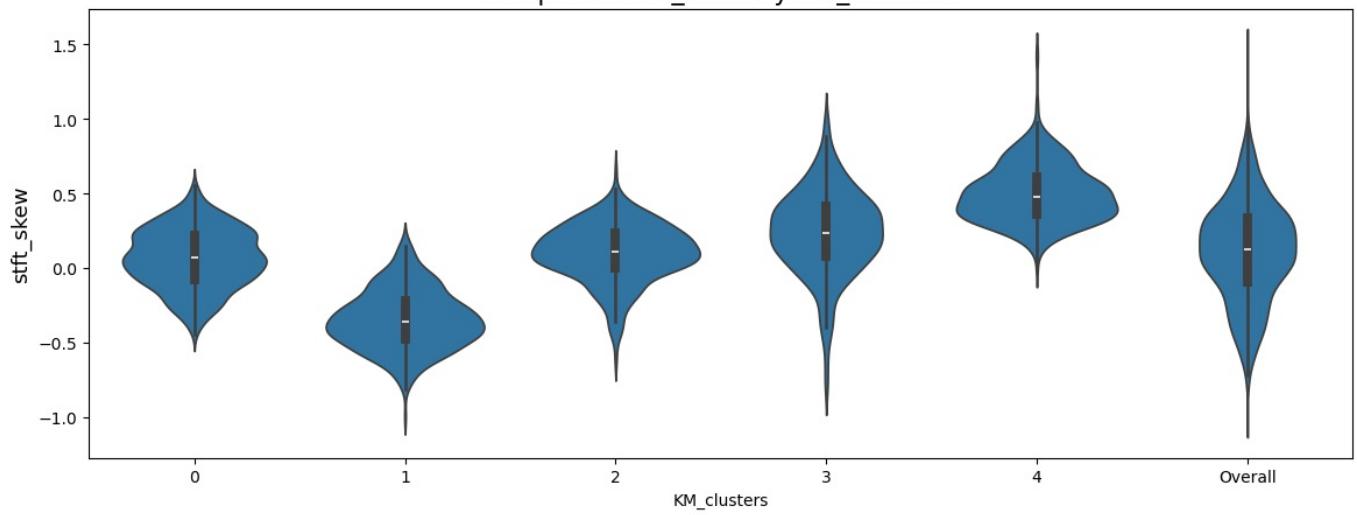
Boxplot of stft_min by KM_clusters



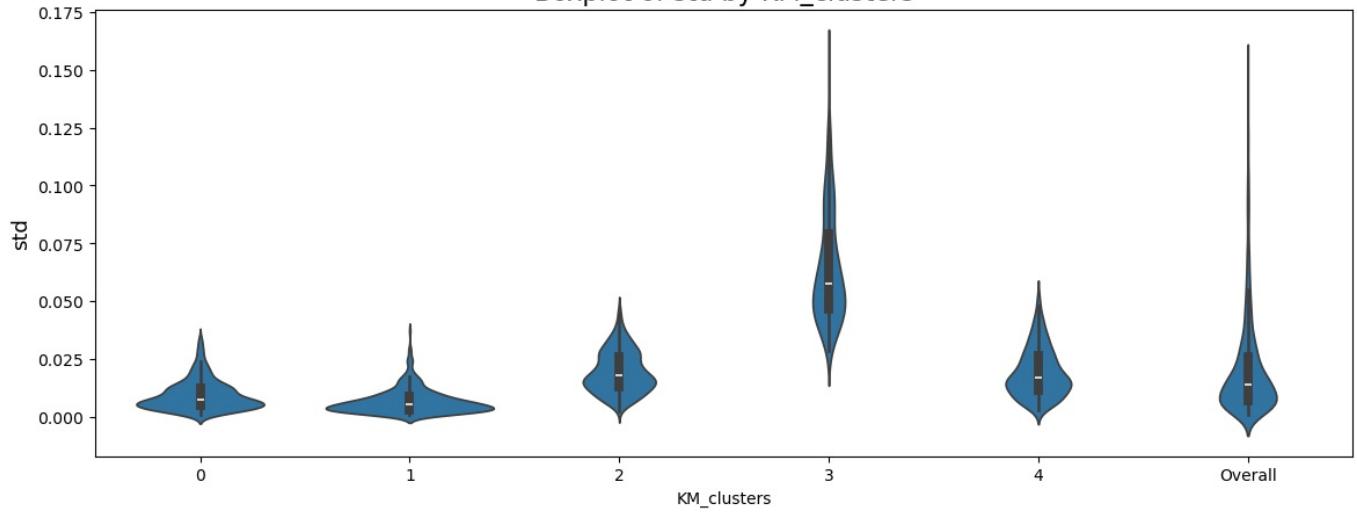
Boxplot of stft_kur by KM_clusters



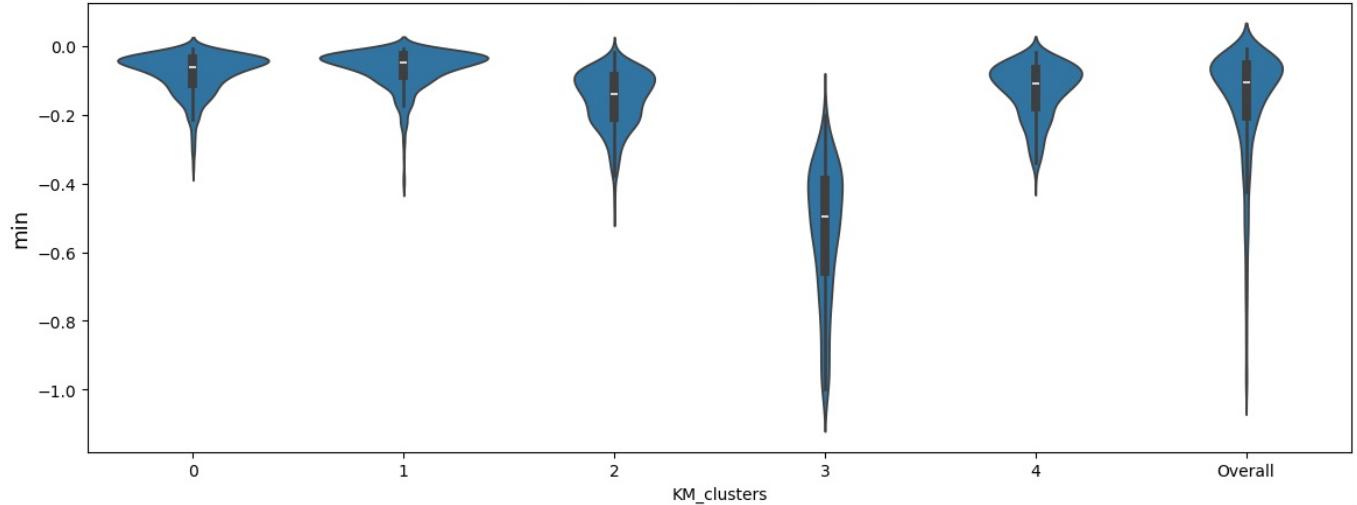
Boxplot of stft_skew by KM_clusters



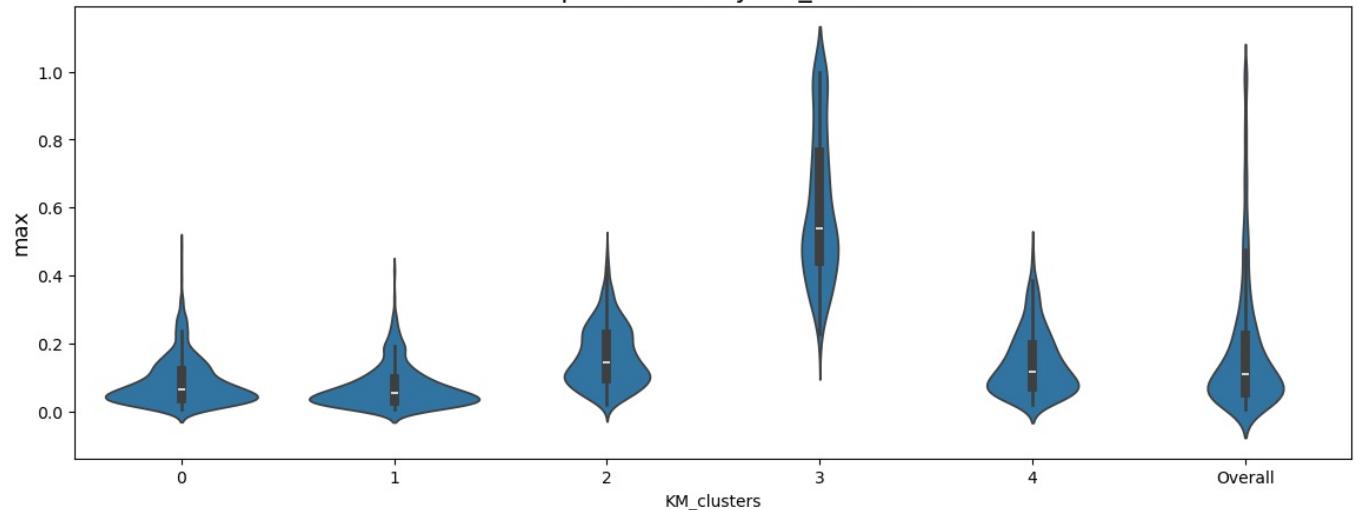
Boxplot of std by KM_clusters



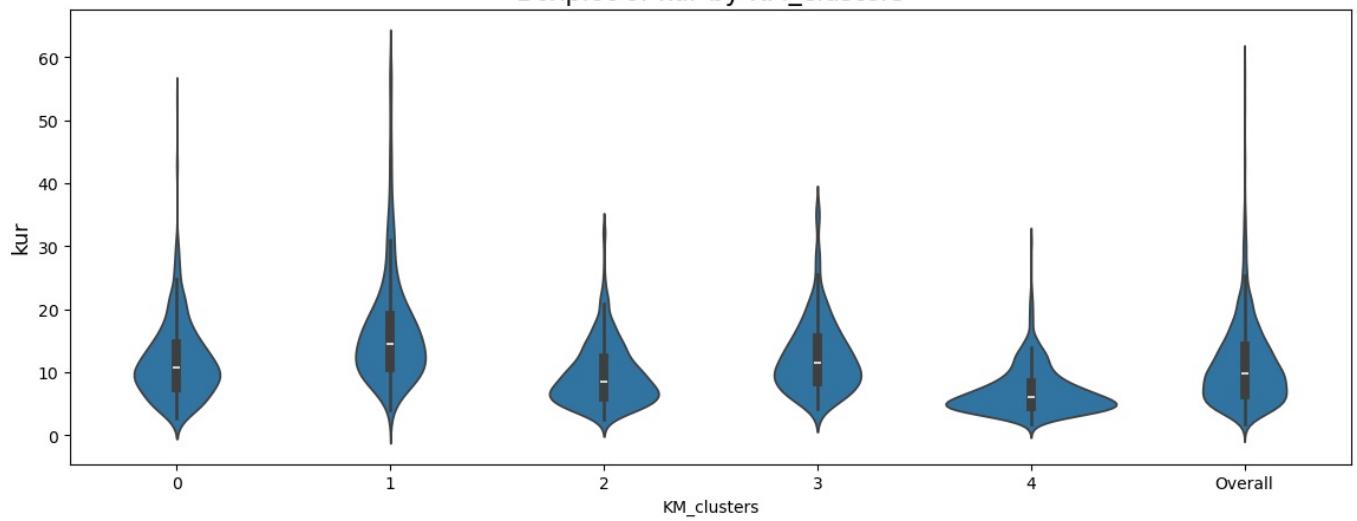
Boxplot of min by KM_clusters



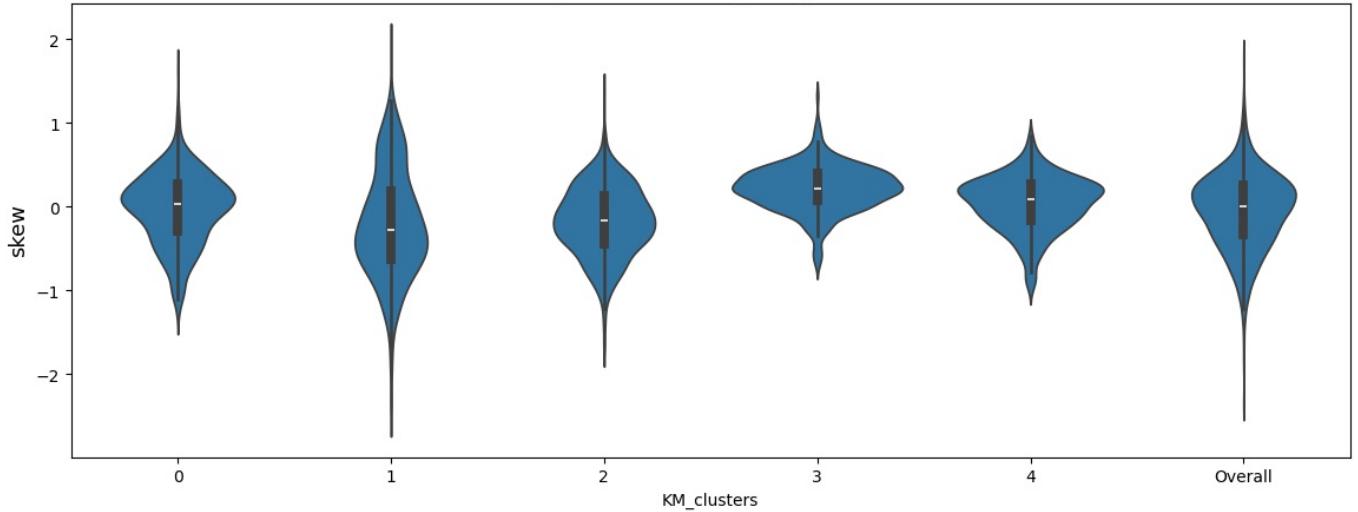
Boxplot of max by KM_clusters



Boxplot of kur by KM_clusters



Boxplot of skew by KM_clusters



Adesso aggiungiamo una dimensione all'analisi appena fatta.

Il proposito dei seguenti grafici è esplorare le **distribuzioni delle variabili originarie** al variare dei cluster, ma mettendo una lente di ingrandimento sul comportamento di:

- **Sex**
- **Emotion**

all'interno dei vari clusters.

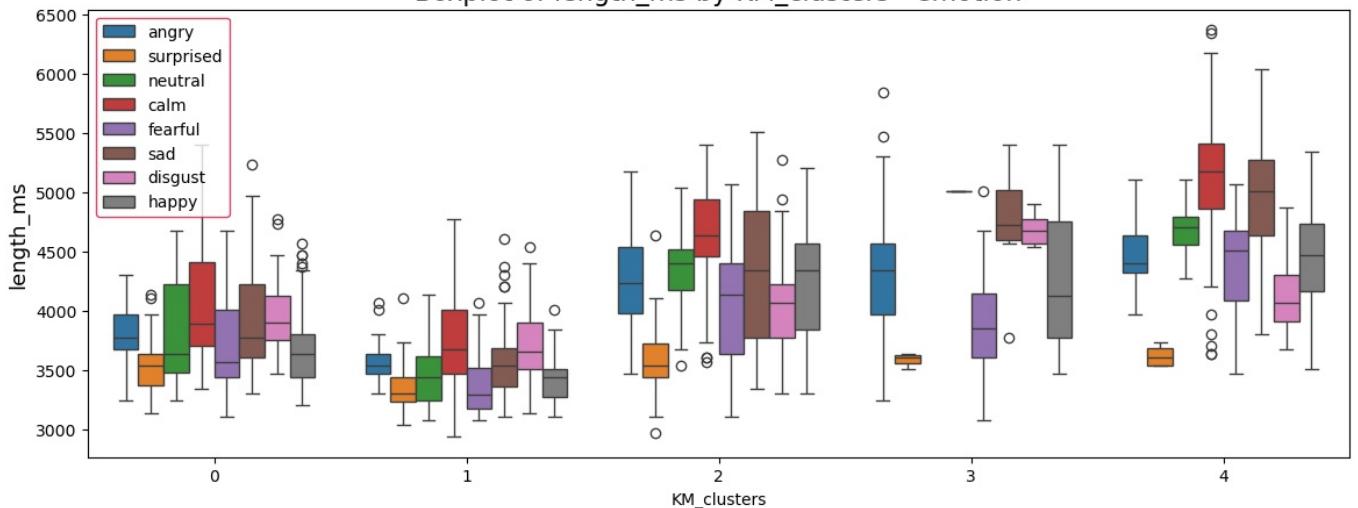
```
In [25]: hue = 'emotion'
target = 'KM_clusters'

for col in numeric_cols:
    fig, ax = plt.subplots(figsize = (14, 5))

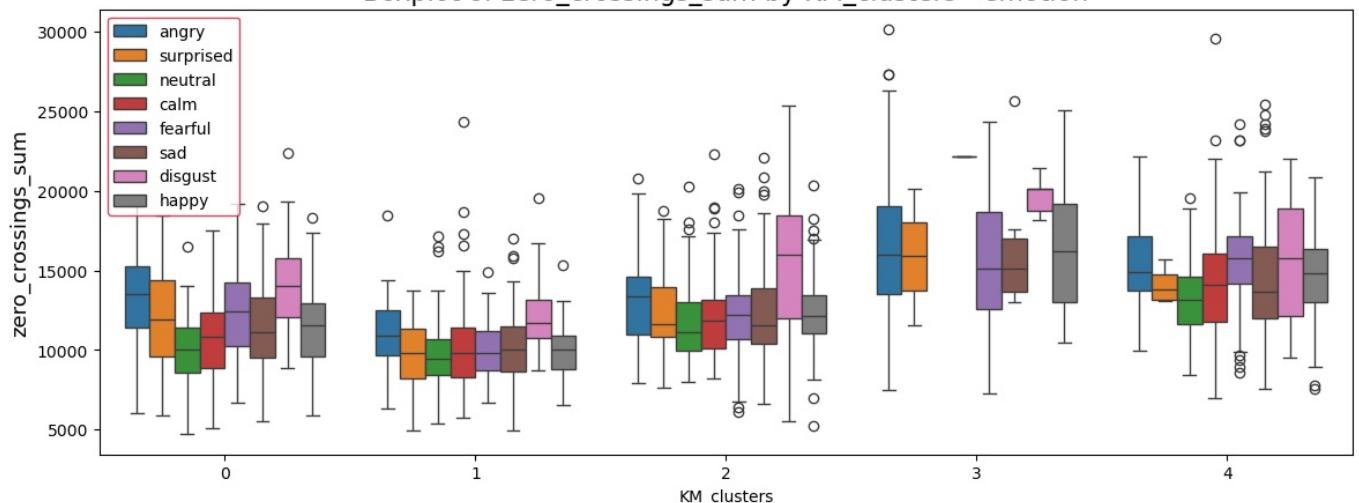
    df_temp = df_cluster[[col, target, hue]].melt(id_vars = [target, hue])

    sns.boxplot(x = target, y = 'value', hue = hue, data = df_temp, ax = ax)
    #sns.violinplot(x = target, y = 'value', hue = hue, data = df_temp, ax = ax, split = False, inner = 'box')
    ax.set_title(f'Boxplot of {col} by {target} - {hue}', fontsize = 16)
    plt.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)
    ax.set_ylabel(col, fontsize = 13)
plt.show()
```

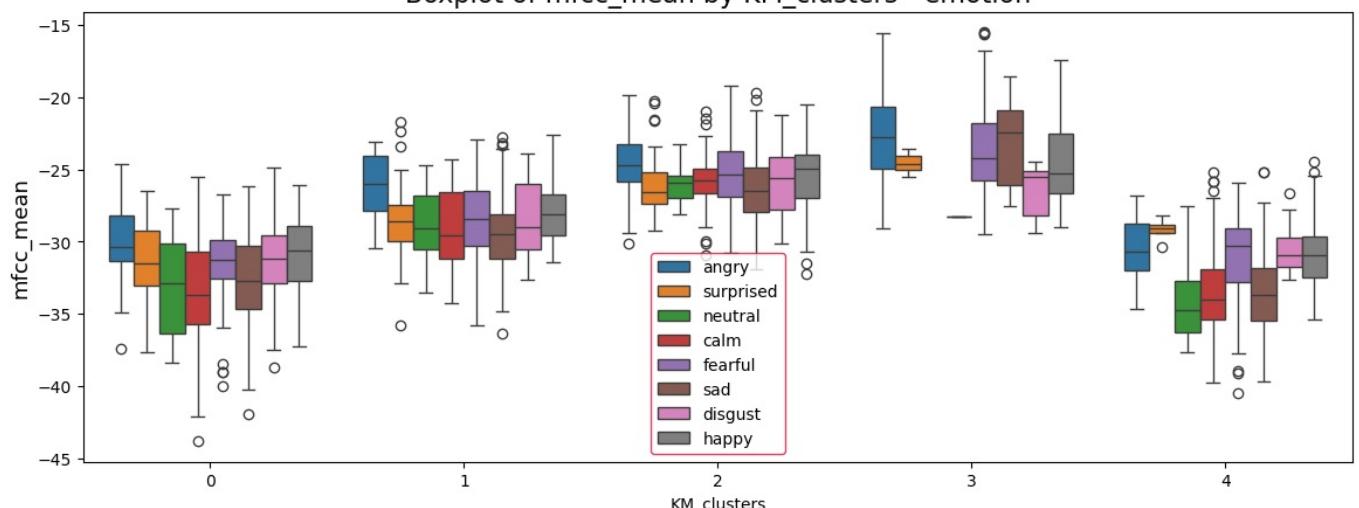
Boxplot of length_ms by KM_clusters - emotion



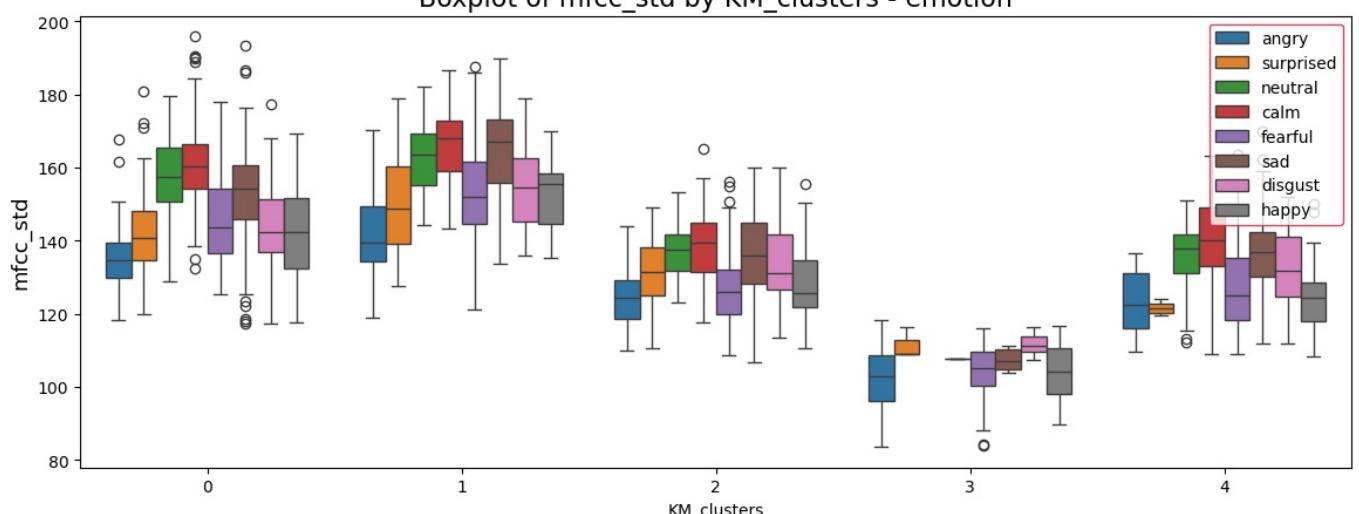
Boxplot of zero_crossings_sum by KM_clusters - emotion



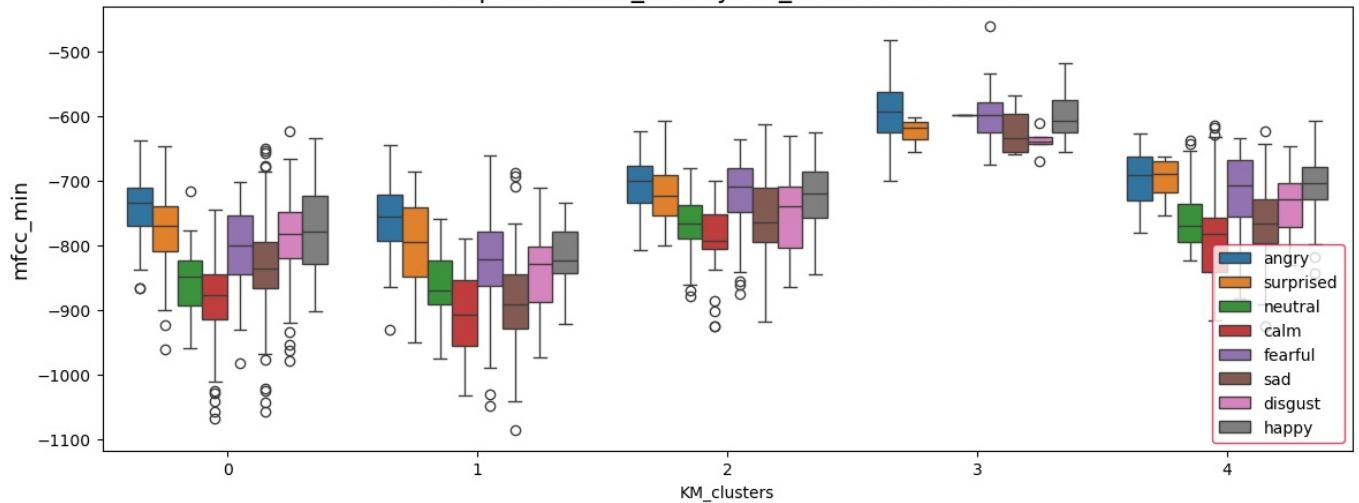
Boxplot of mfcc_mean by KM_clusters - emotion



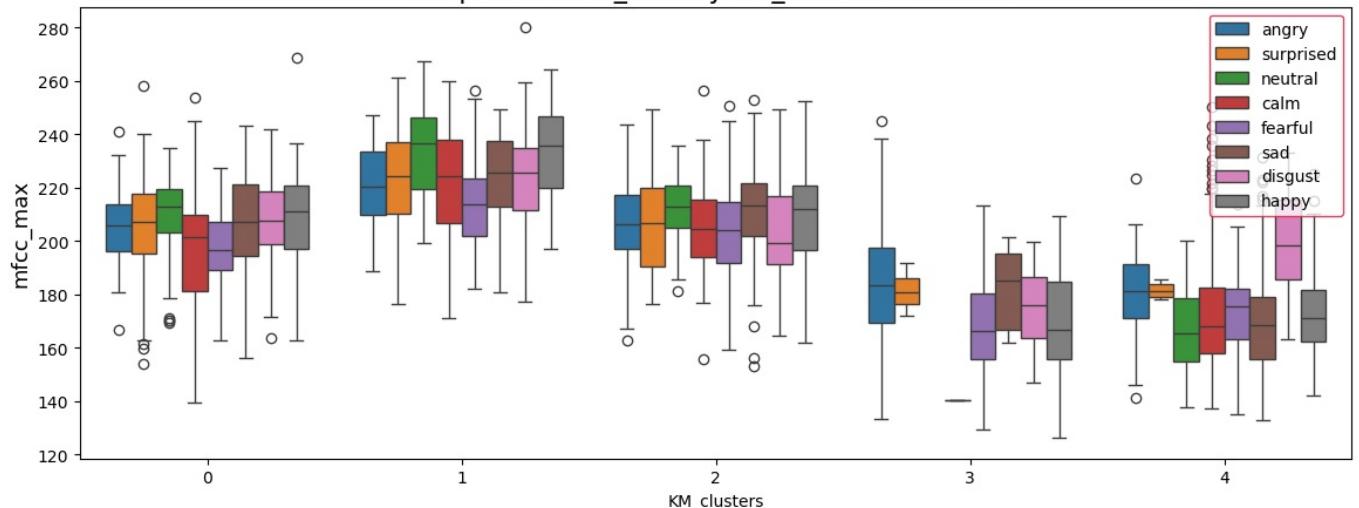
Boxplot of mfcc_std by KM_clusters - emotion



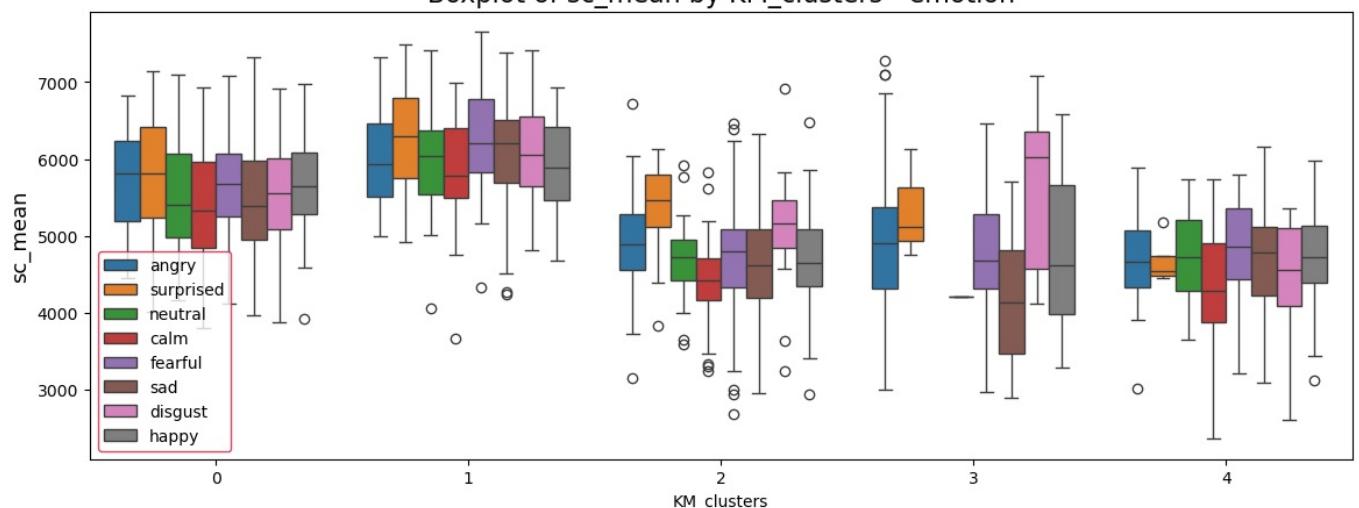
Boxplot of mfcc_min by KM_clusters - emotion



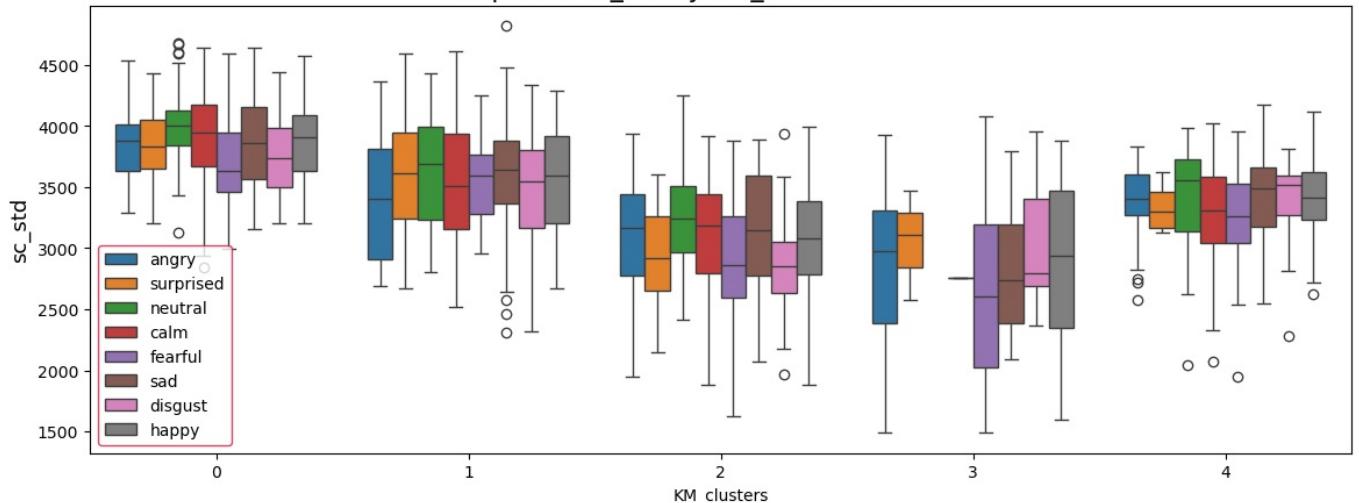
Boxplot of mfcc_max by KM_clusters - emotion



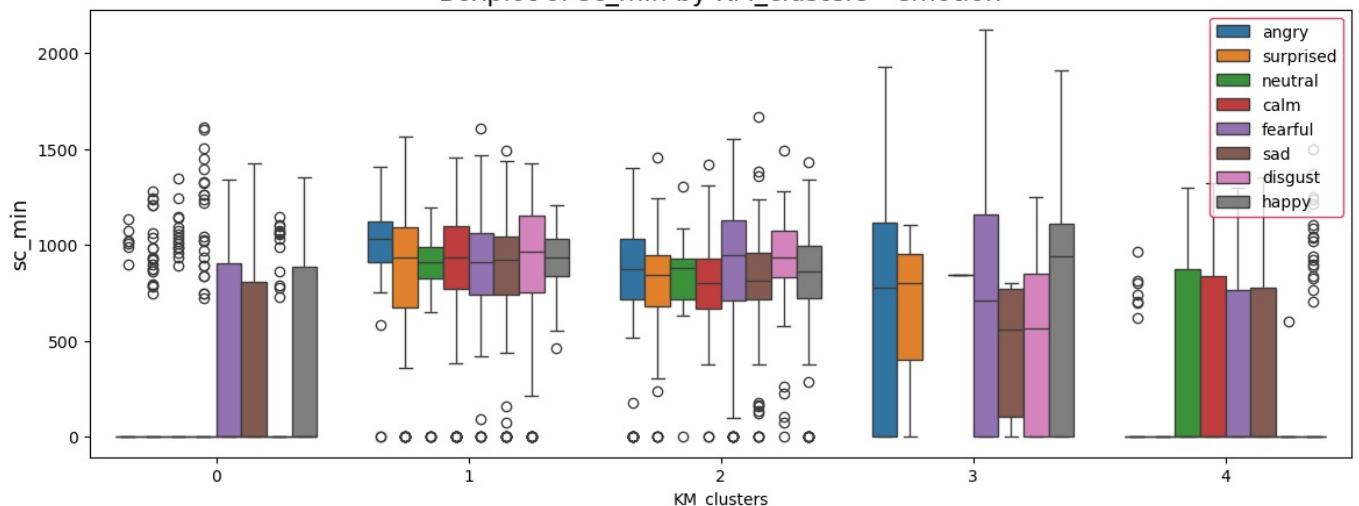
Boxplot of sc_mean by KM_clusters - emotion



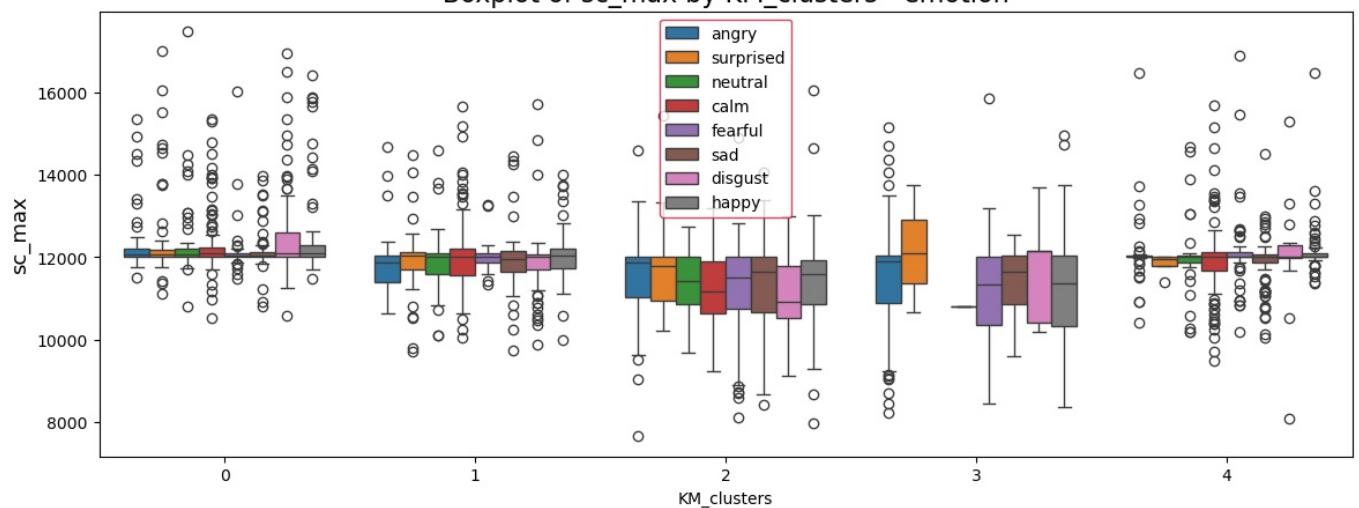
Boxplot of sc_std by KM_clusters - emotion



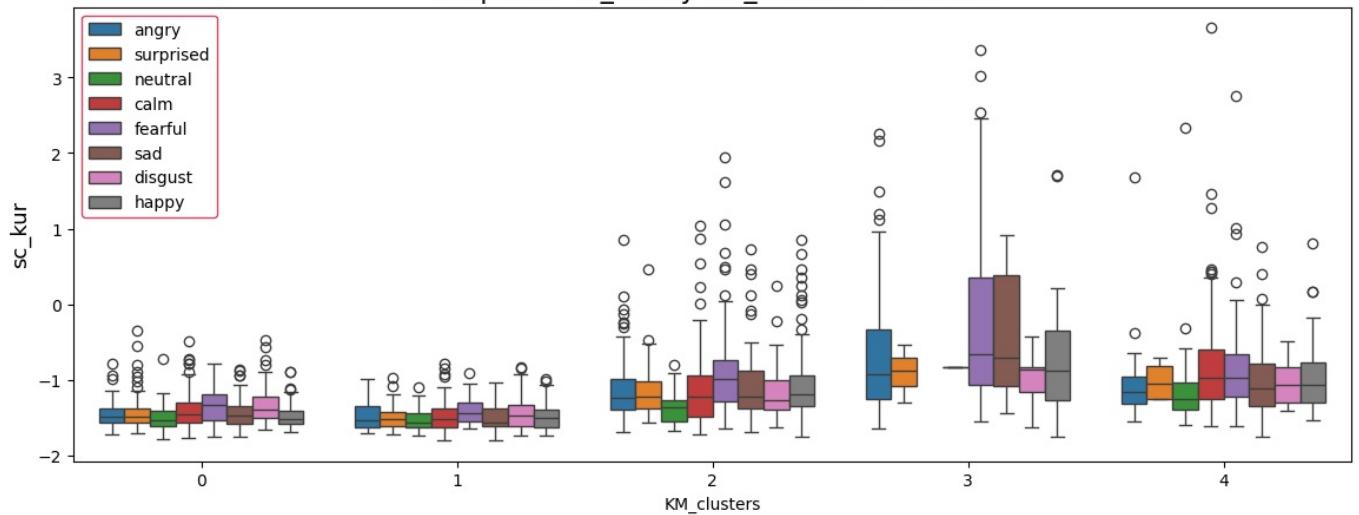
Boxplot of sc_min by KM_clusters - emotion



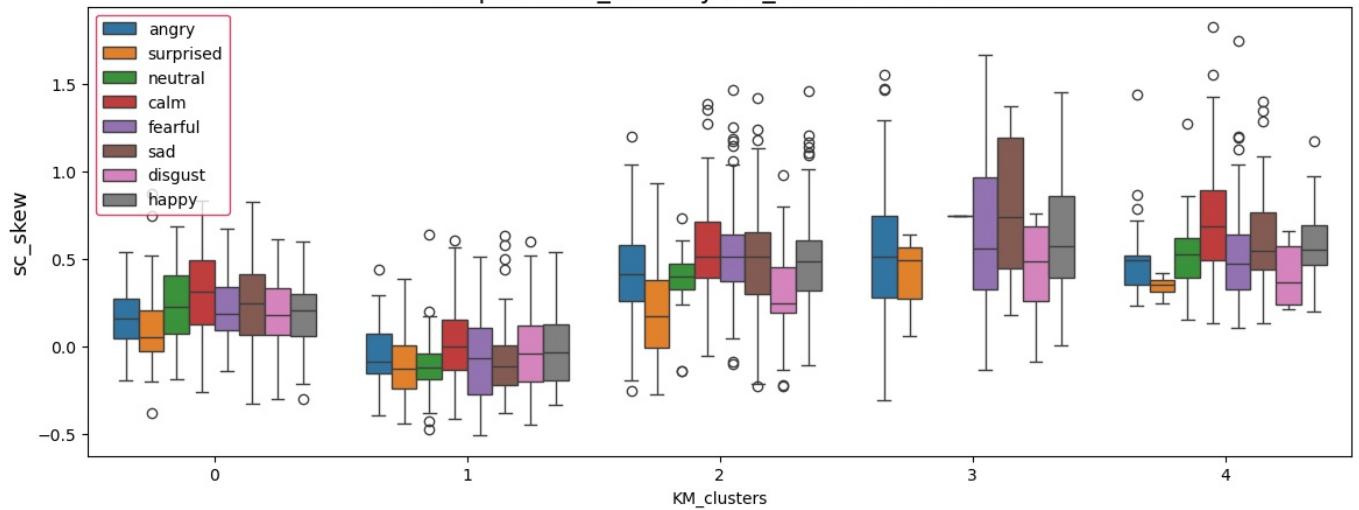
Boxplot of sc_max by KM_clusters - emotion



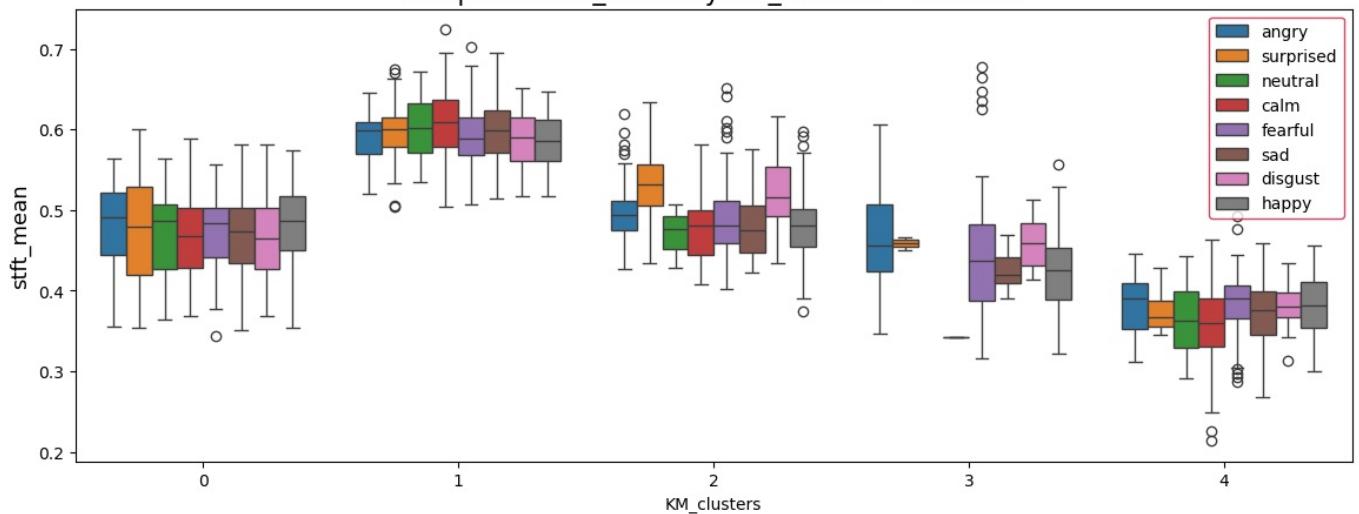
Boxplot of sc_kur by KM_clusters - emotion

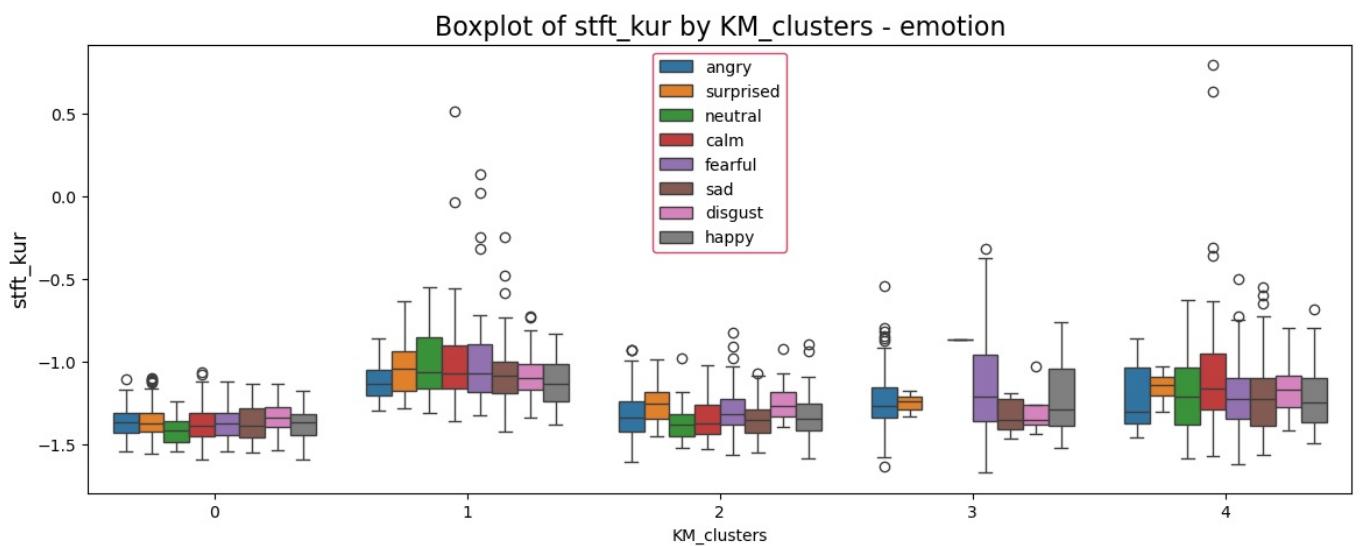
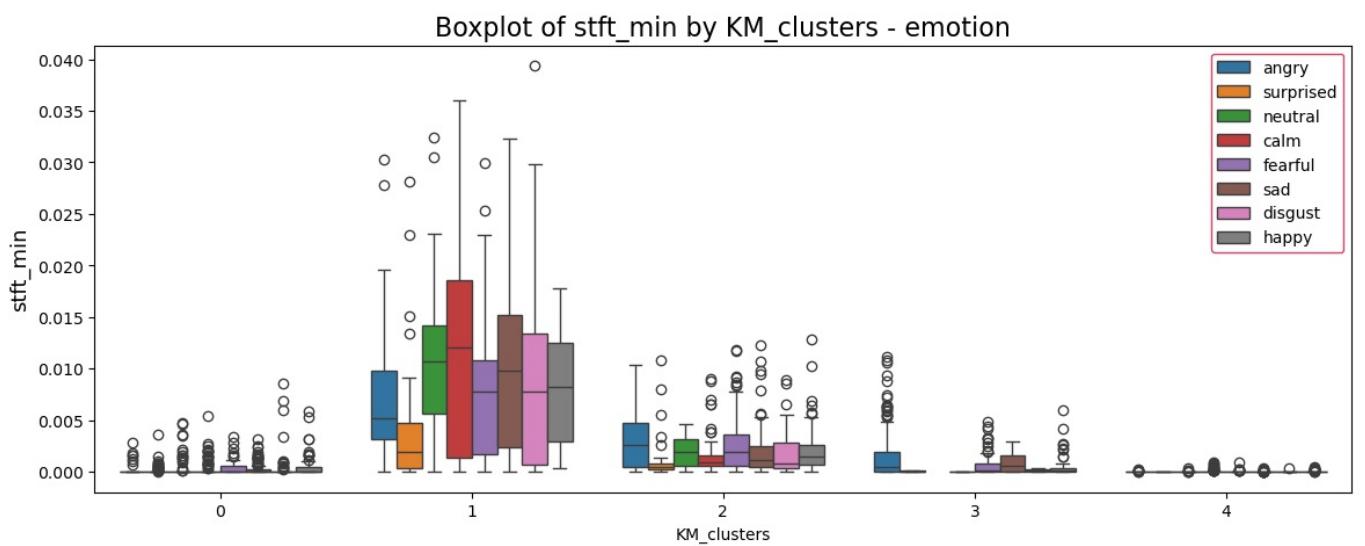
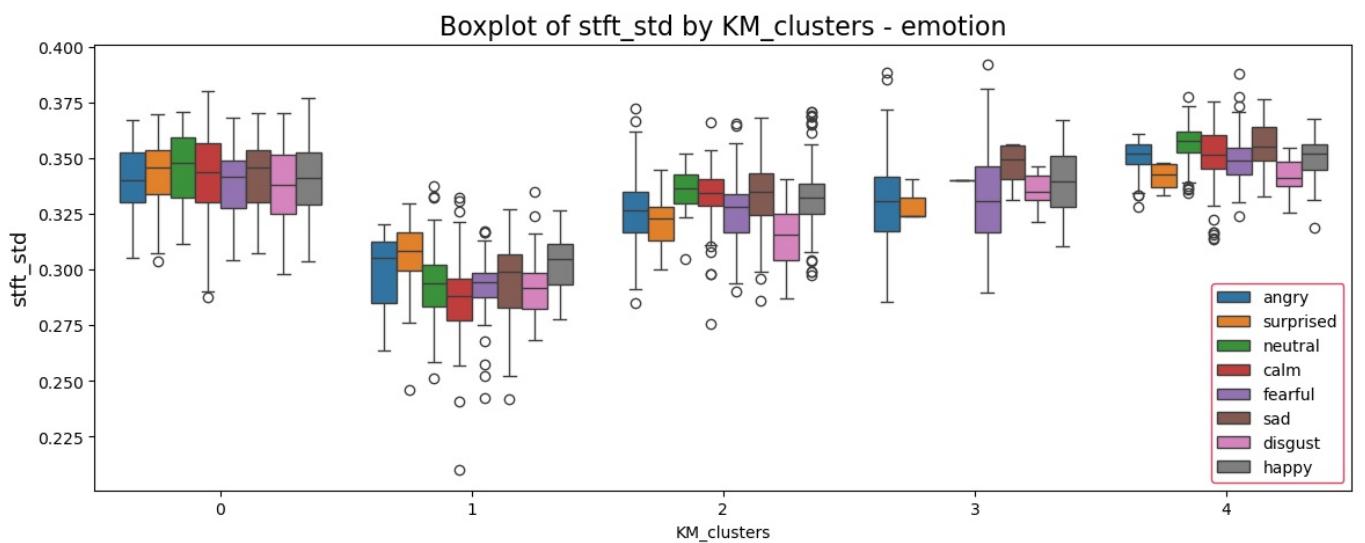


Boxplot of sc_skew by KM_clusters - emotion

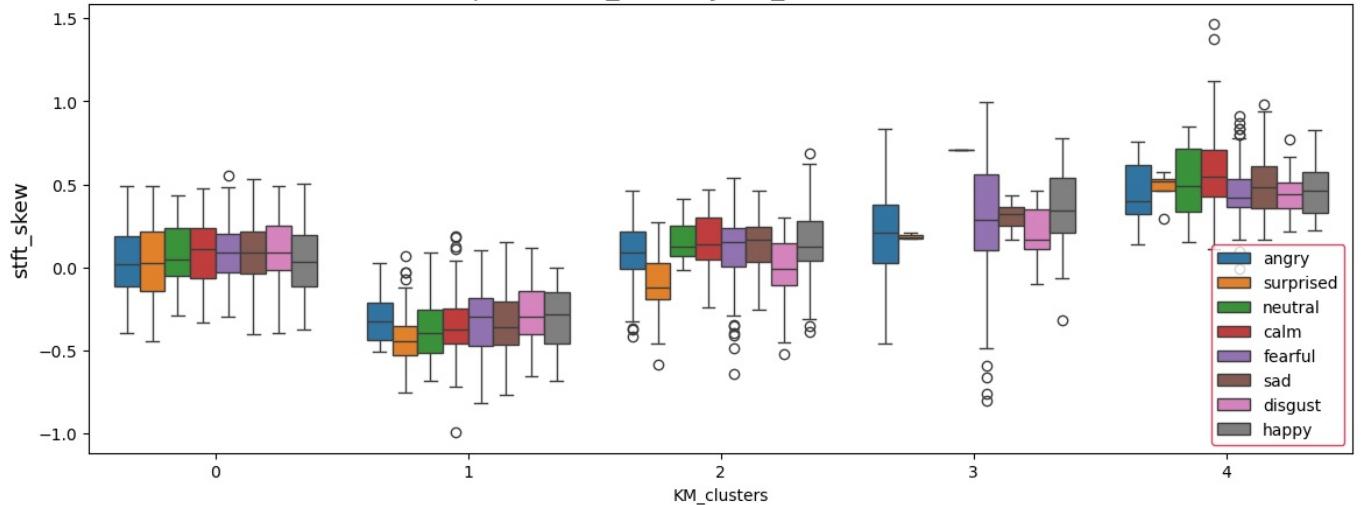


Boxplot of stft_mean by KM_clusters - emotion

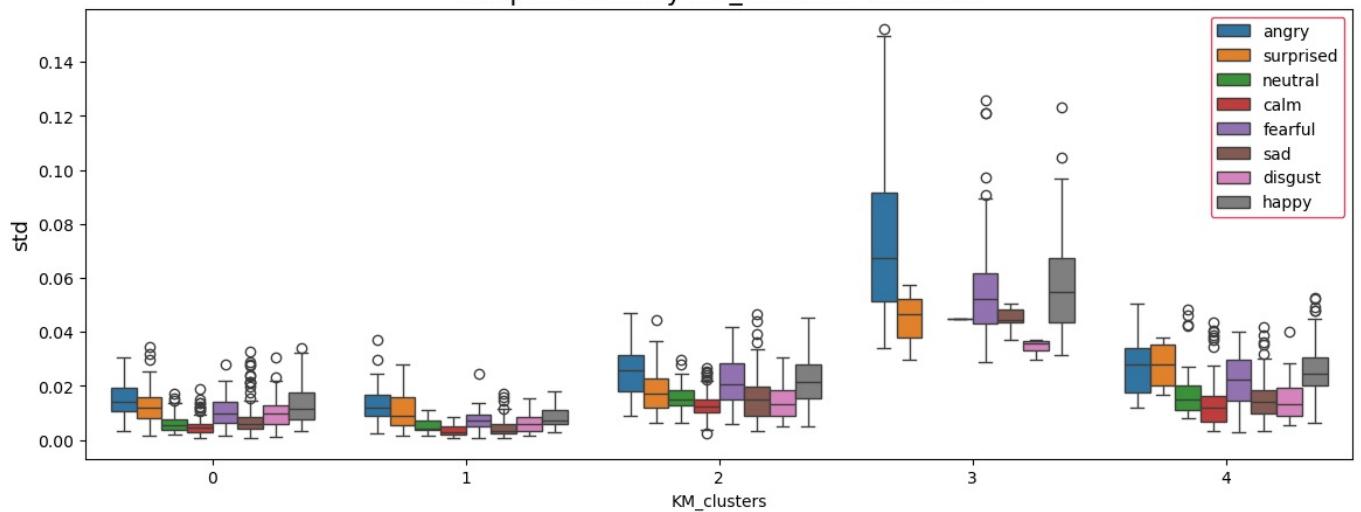




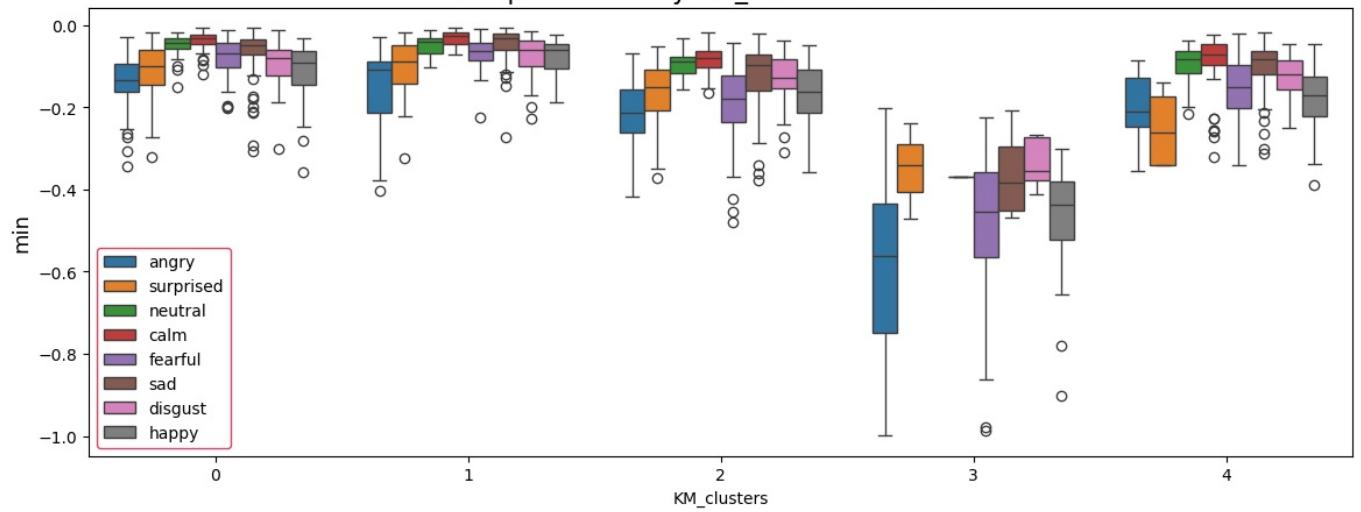
Boxplot of stft_skew by KM_clusters - emotion



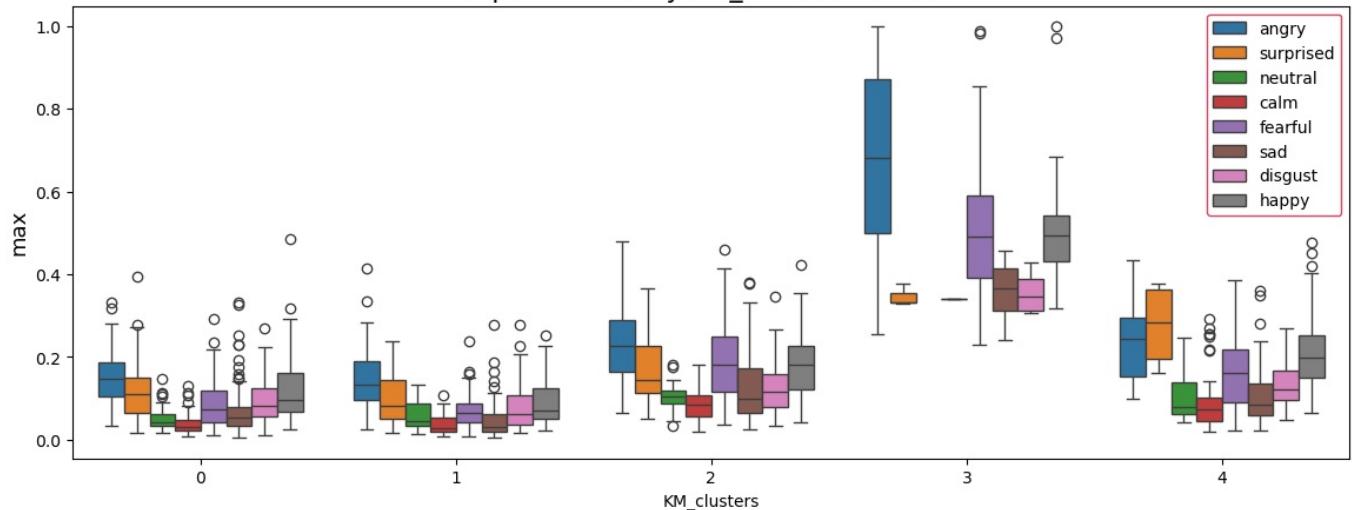
Boxplot of std by KM_clusters - emotion



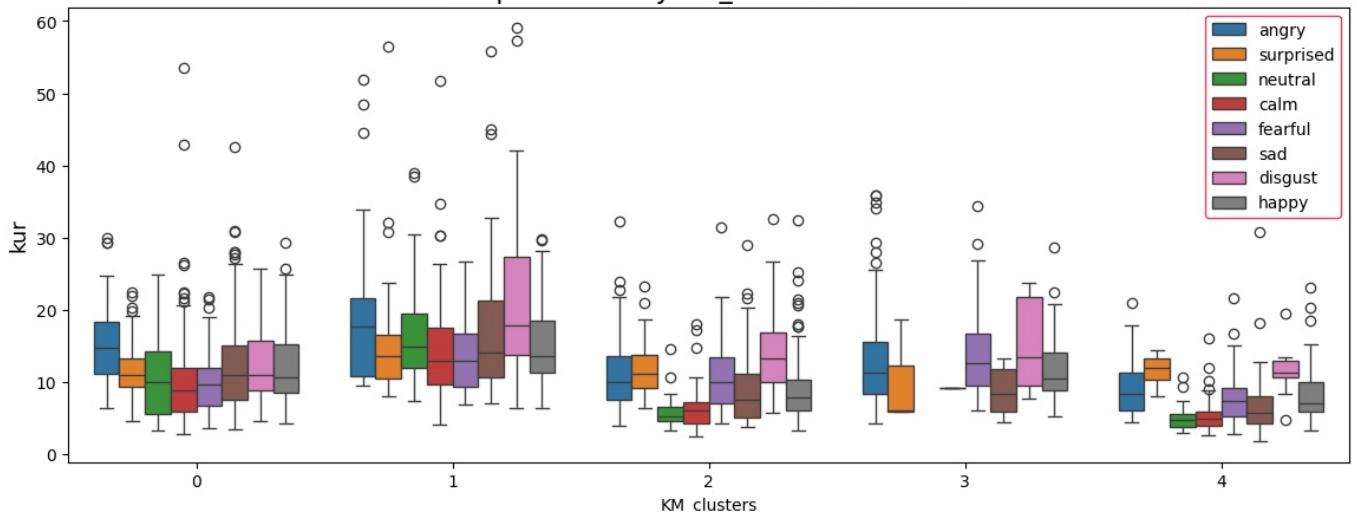
Boxplot of min by KM_clusters - emotion



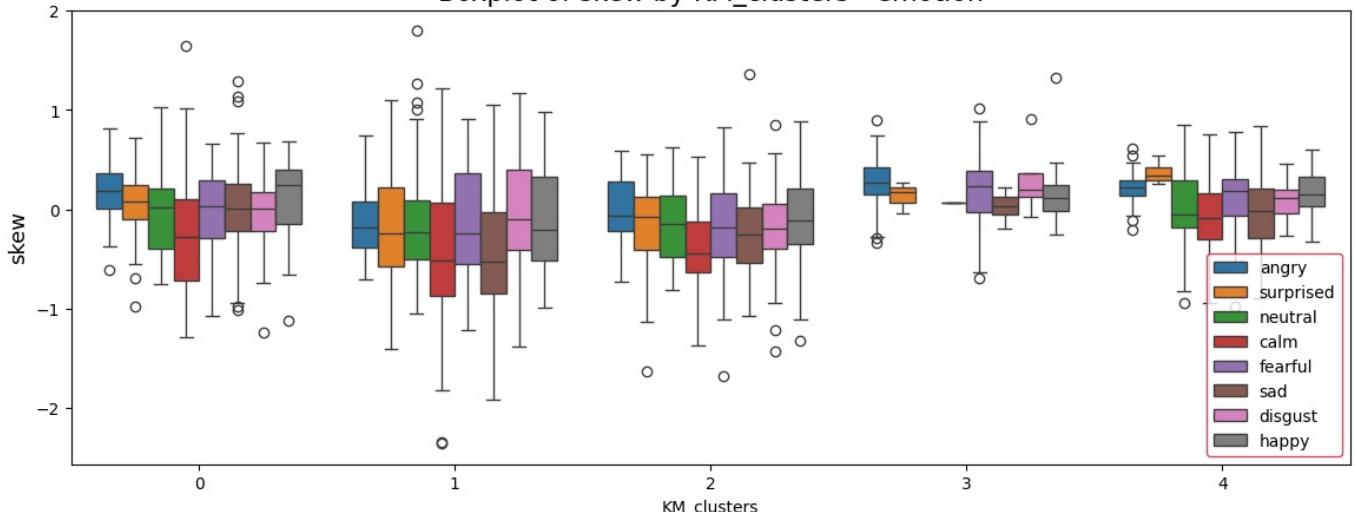
Boxplot of max by KM_clusters - emotion



Boxplot of kur by KM_clusters - emotion



Boxplot of skew by KM_clusters - emotion



```
In [26]: hue = 'sex'
target = 'KM_clusters'

for col in numeric_cols:

    fig, ax = plt.subplots(figsize = (14, 5))

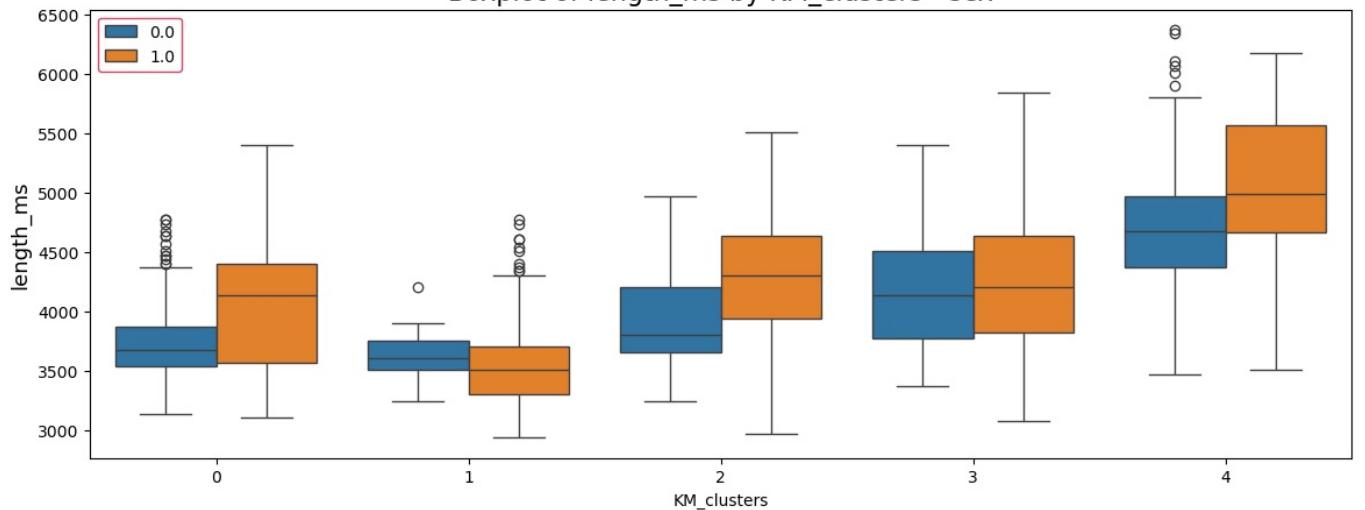
    df_temp = df_cluster[[col, target, hue]].melt(id_vars = [target, hue])
```

```

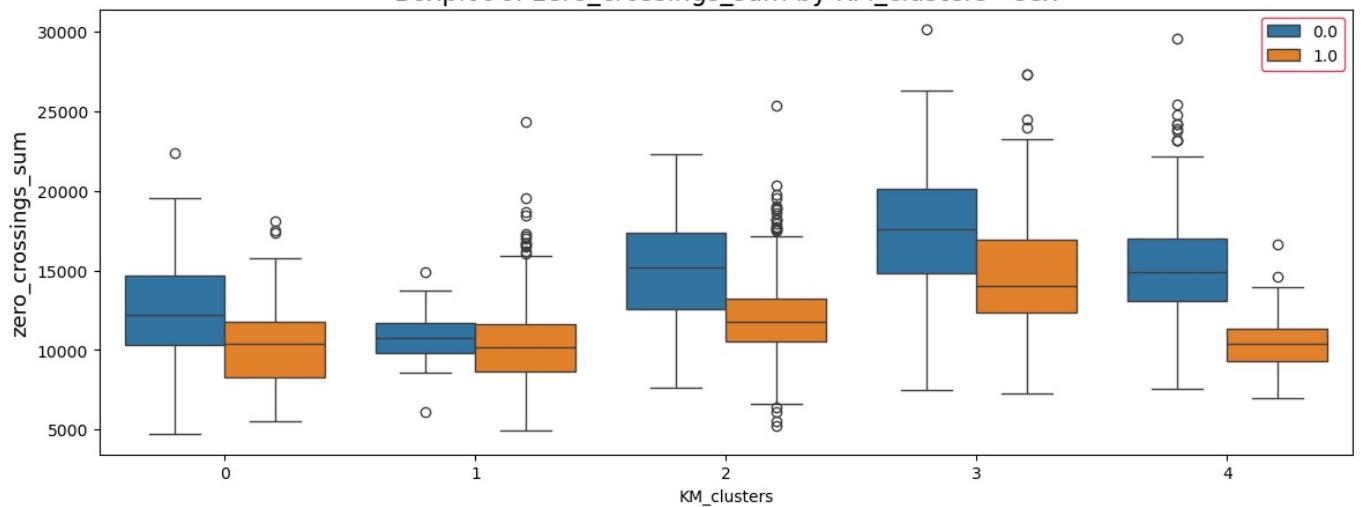
sns.boxplot(x = target, y = 'value', hue = hue, data = df_temp, ax = ax)
#sns.violinplot(x = target, y = 'value', hue = hue, data = df_temp, ax = ax, split = False, inner = 'box')
ax.set_title(f'Boxplot of {col} by {target} - {hue}', fontsize = 16)
plt.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)
ax.set_ylabel(col, fontsize = 13)
plt.show()

```

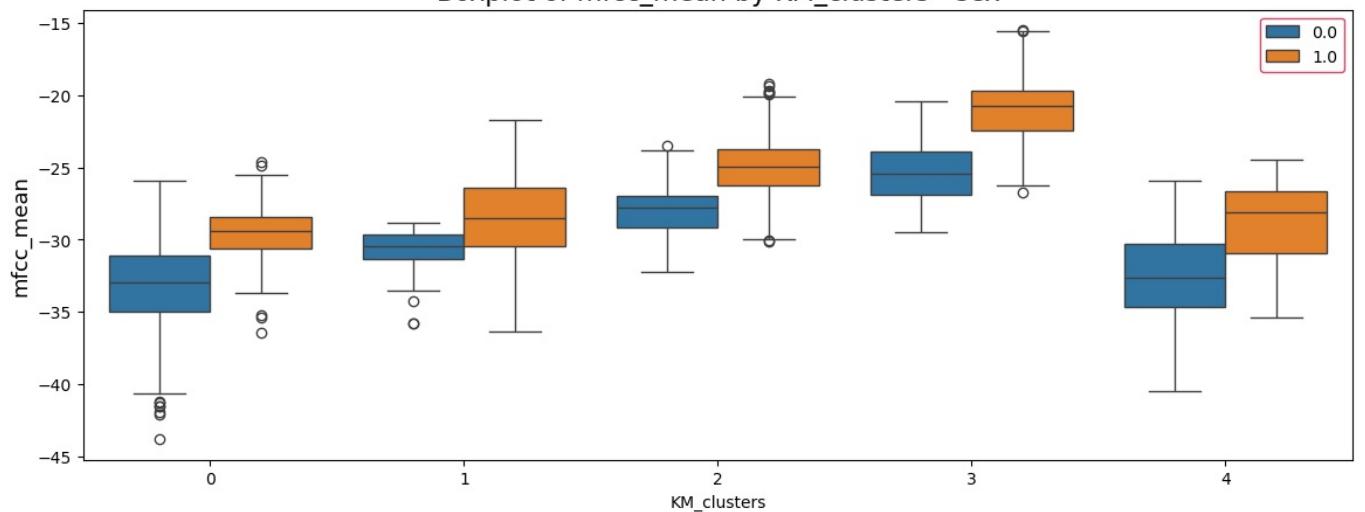
Boxplot of length_ms by KM_clusters - sex



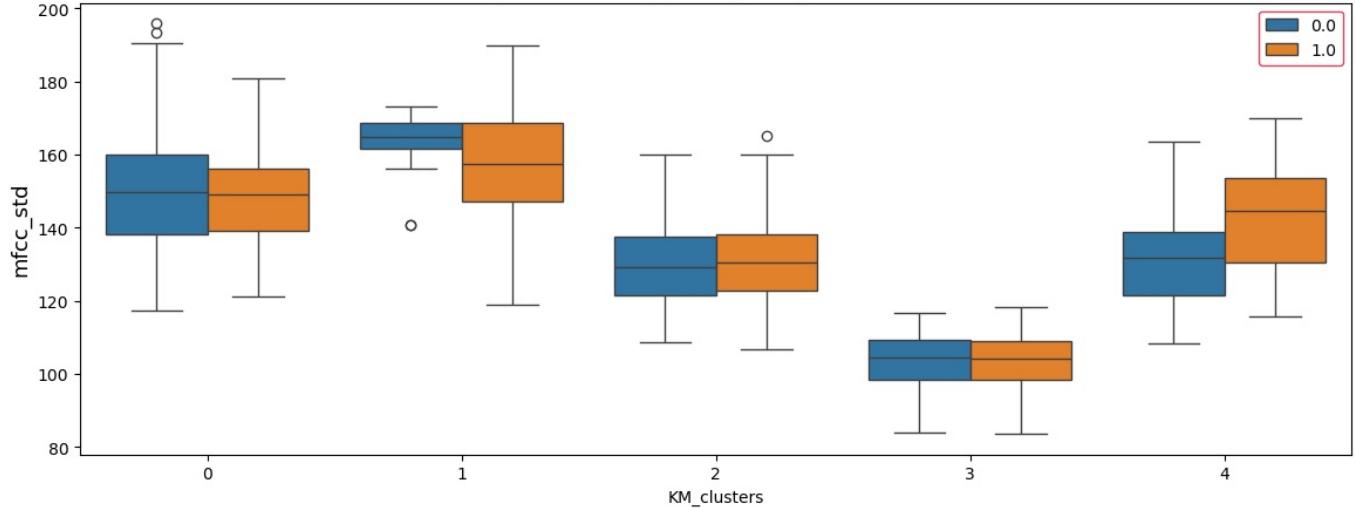
Boxplot of zero_crossings_sum by KM_clusters - sex



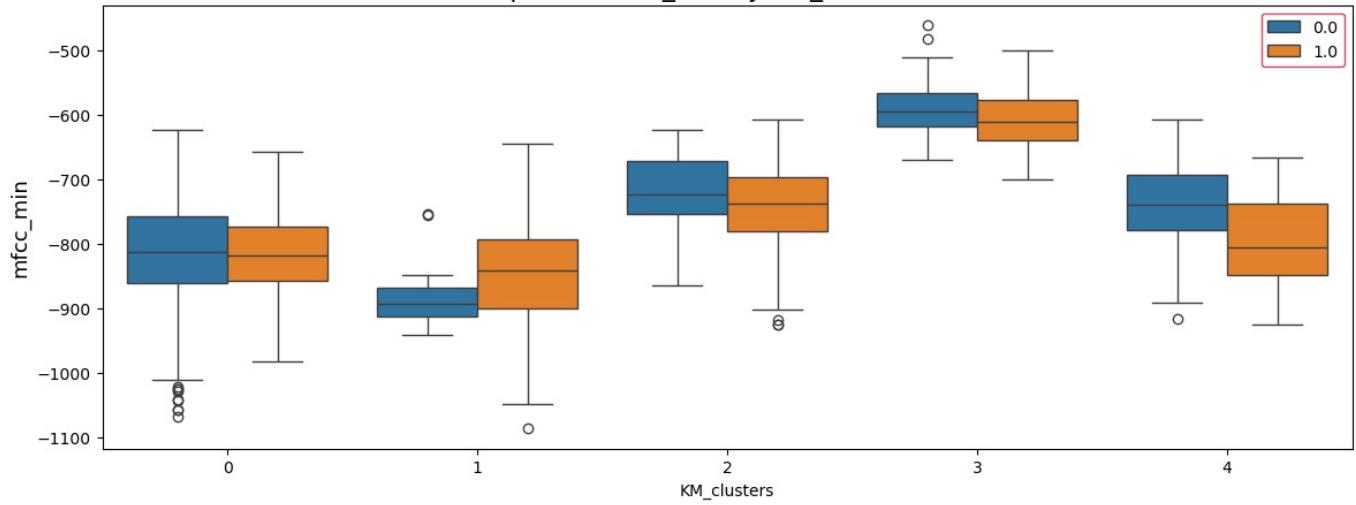
Boxplot of mfcc_mean by KM_clusters - sex



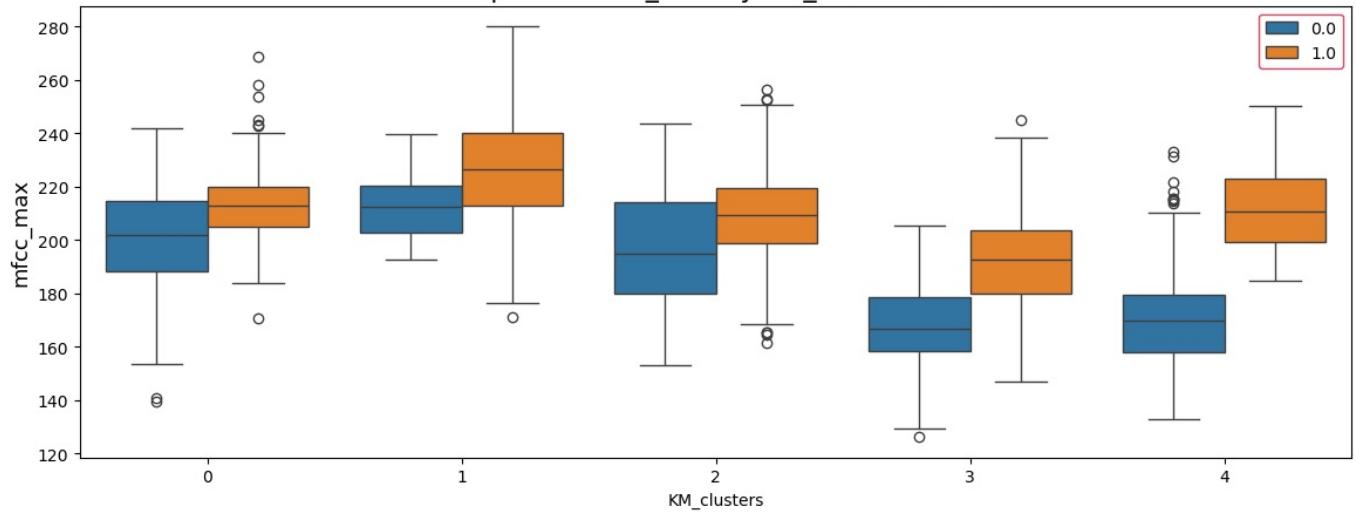
Boxplot of mfcc_std by KM_clusters - sex



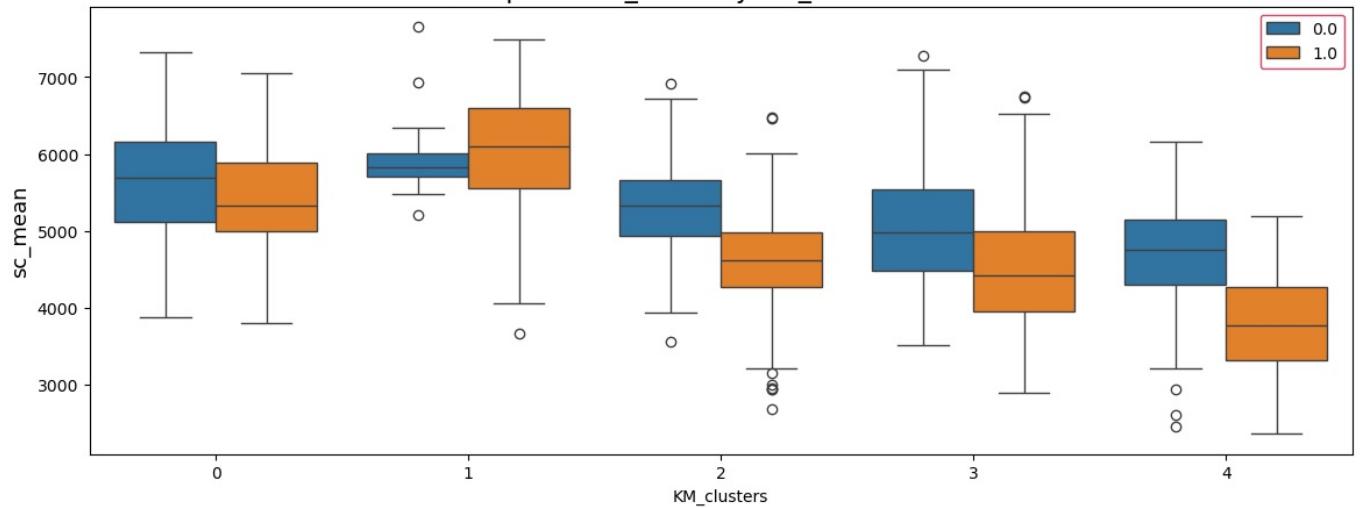
Boxplot of mfcc_min by KM_clusters - sex



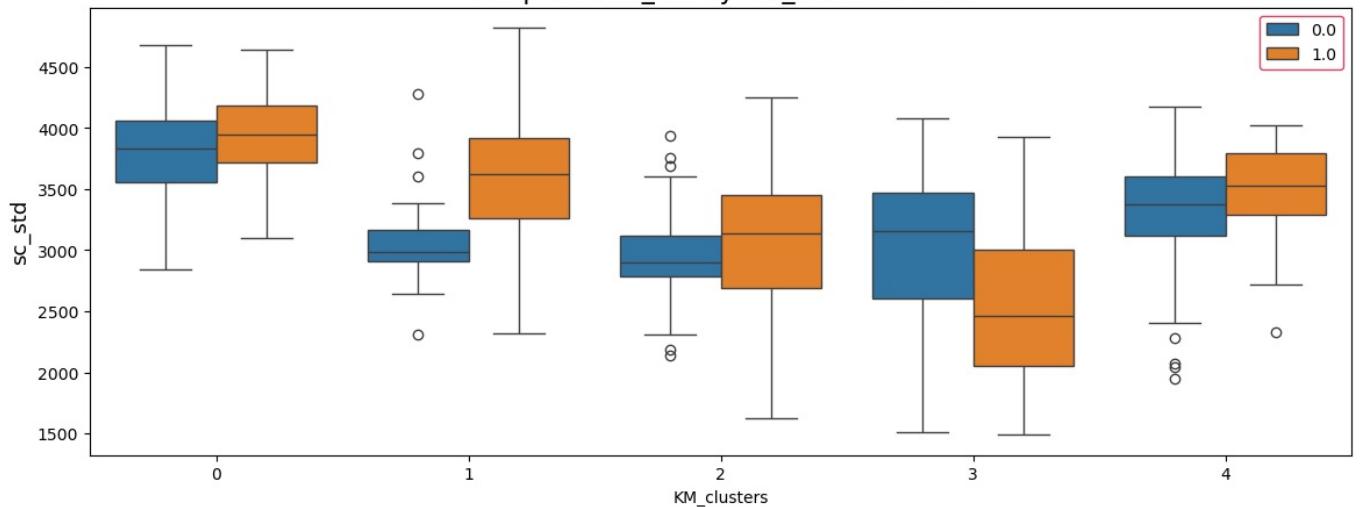
Boxplot of mfcc_max by KM_clusters - sex



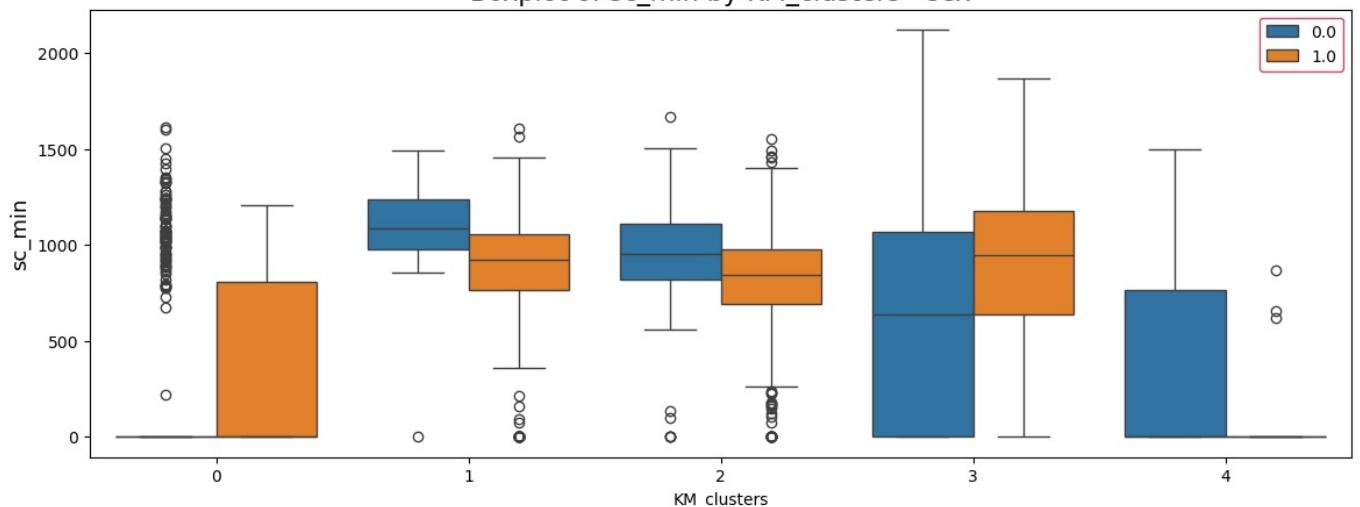
Boxplot of sc_mean by KM_clusters - sex



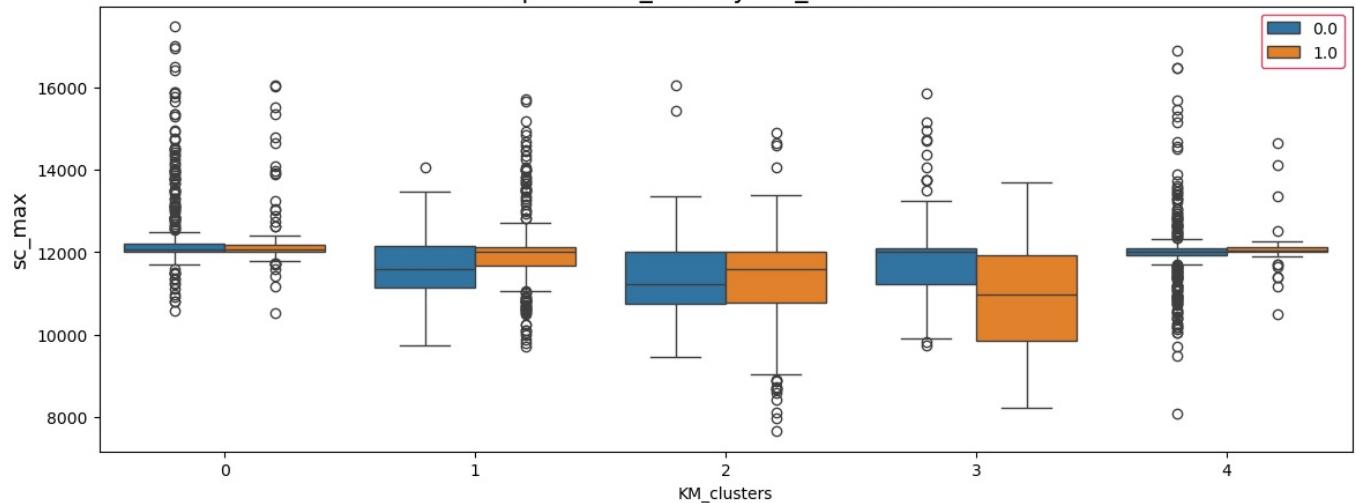
Boxplot of sc_std by KM_clusters - sex



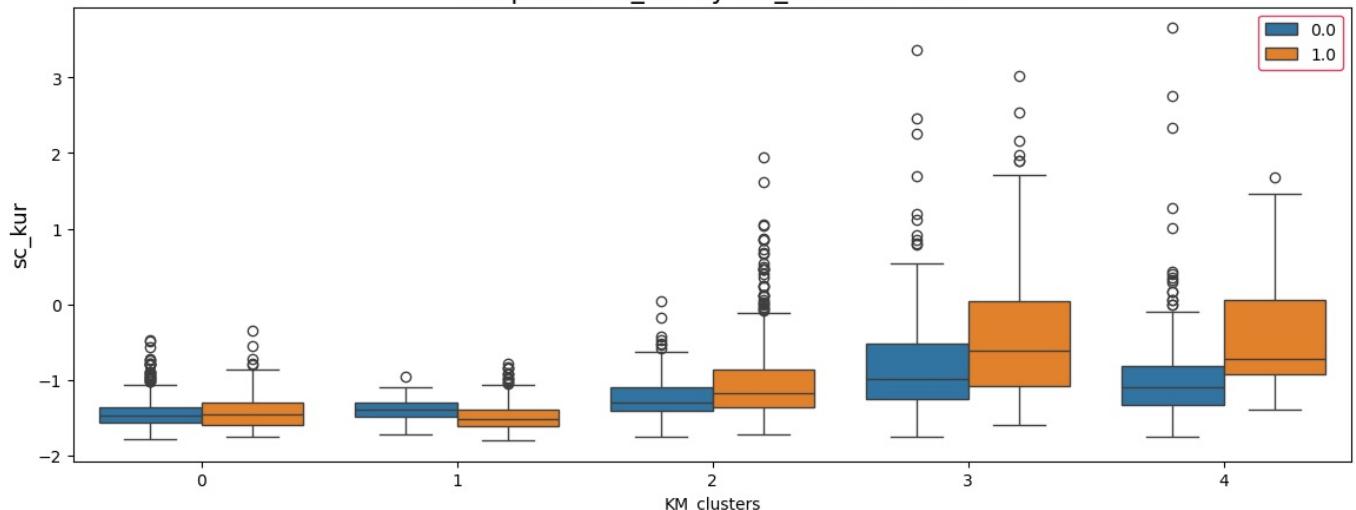
Boxplot of sc_min by KM_clusters - sex



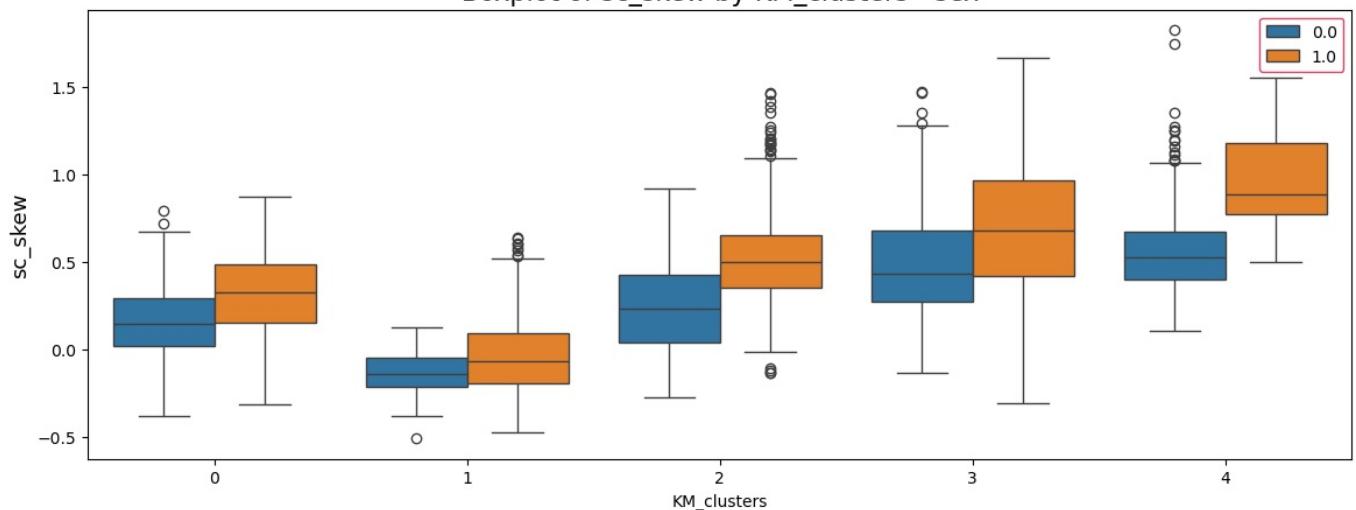
Boxplot of sc_max by KM_clusters - sex



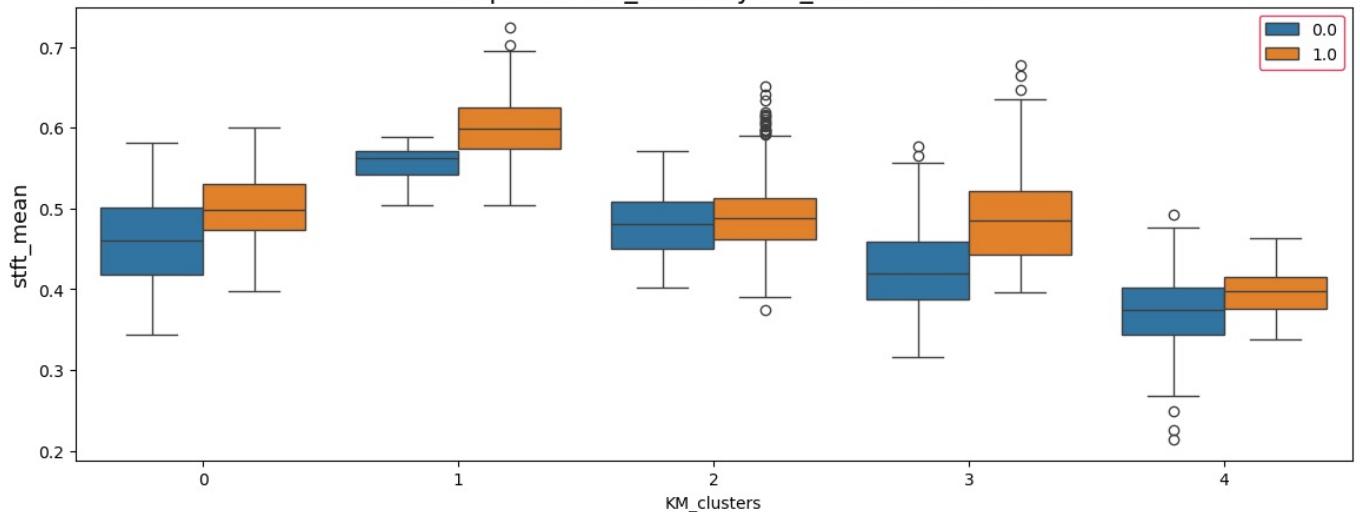
Boxplot of sc_kur by KM_clusters - sex



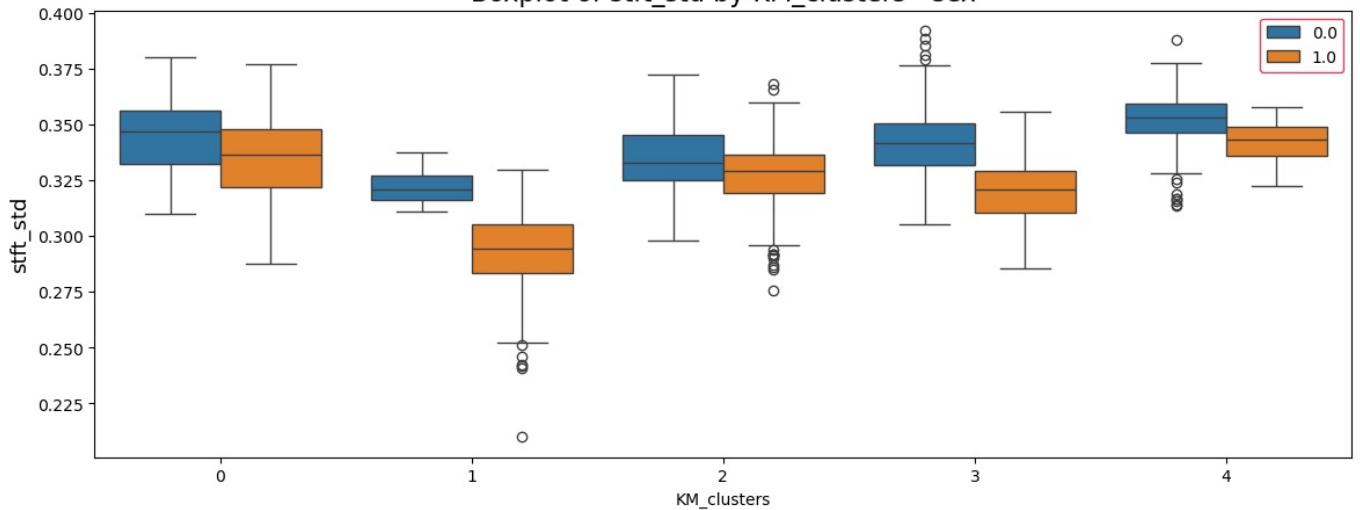
Boxplot of sc_skew by KM_clusters - sex



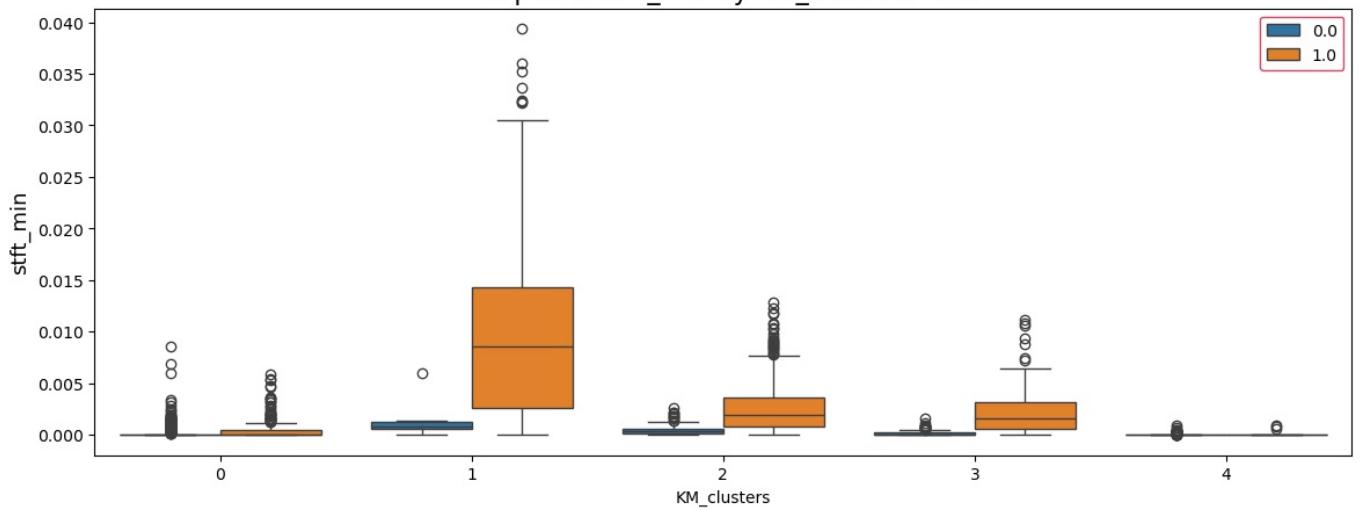
Boxplot of stft_mean by KM_clusters - sex



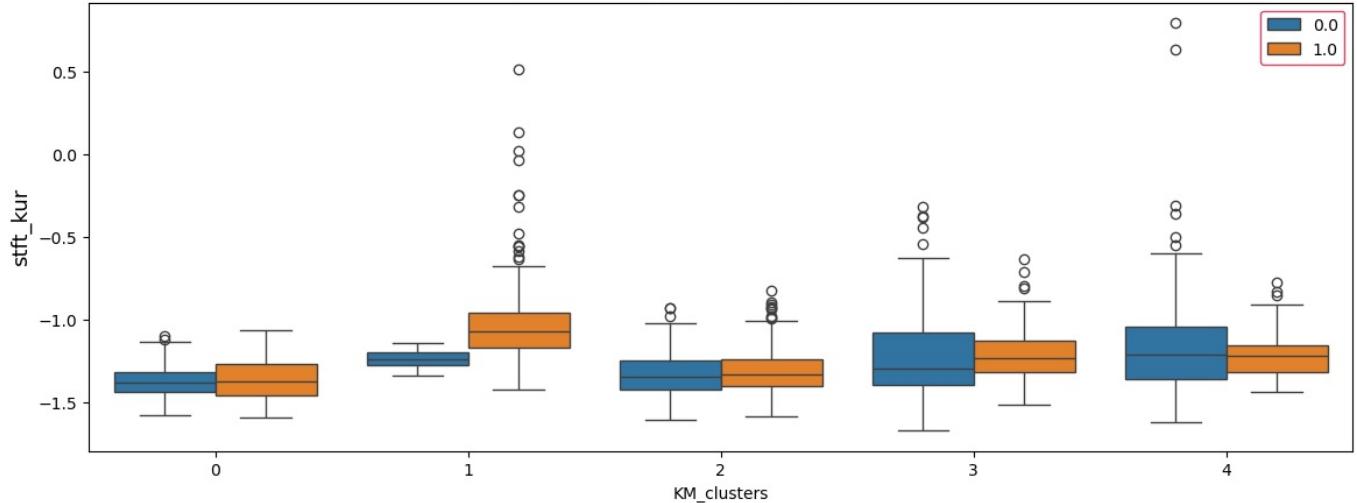
Boxplot of stft_std by KM_clusters - sex



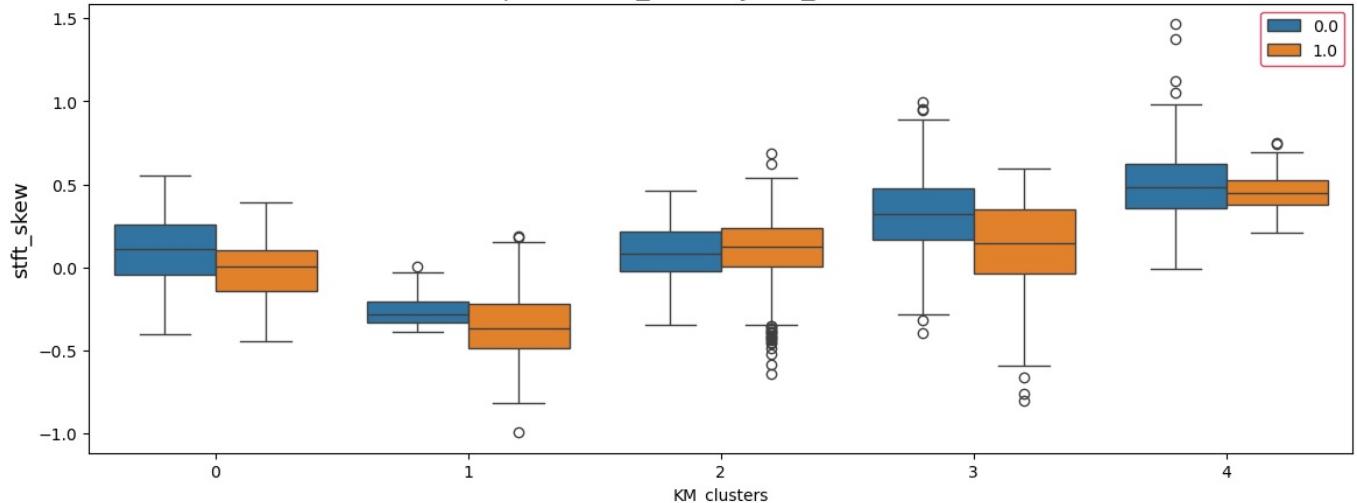
Boxplot of stft_min by KM_clusters - sex



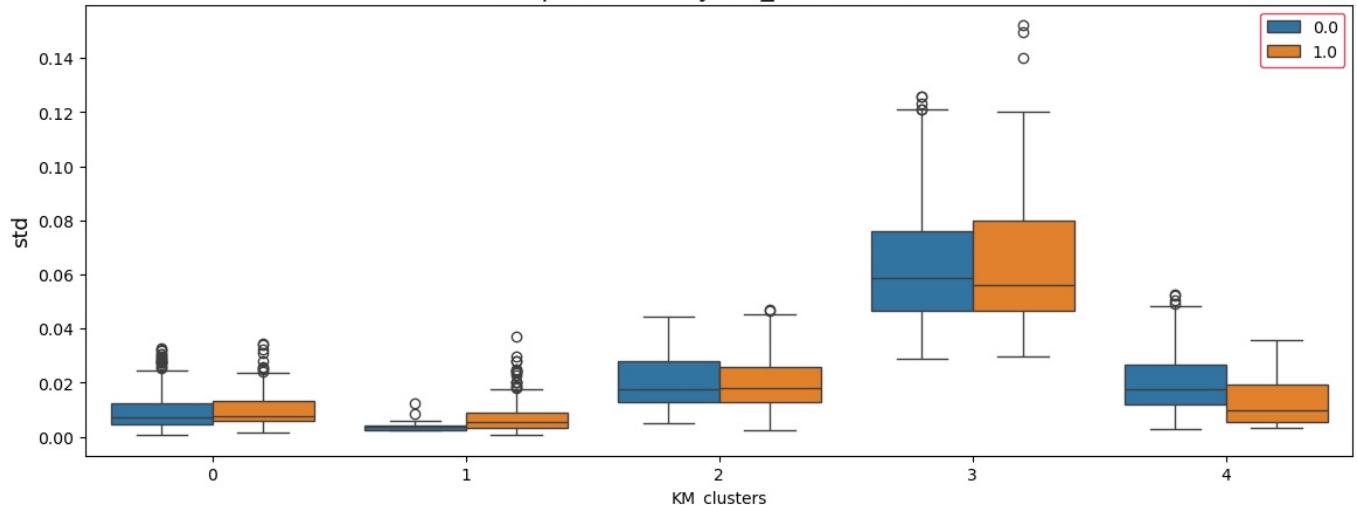
Boxplot of stft_kur by KM_clusters - sex



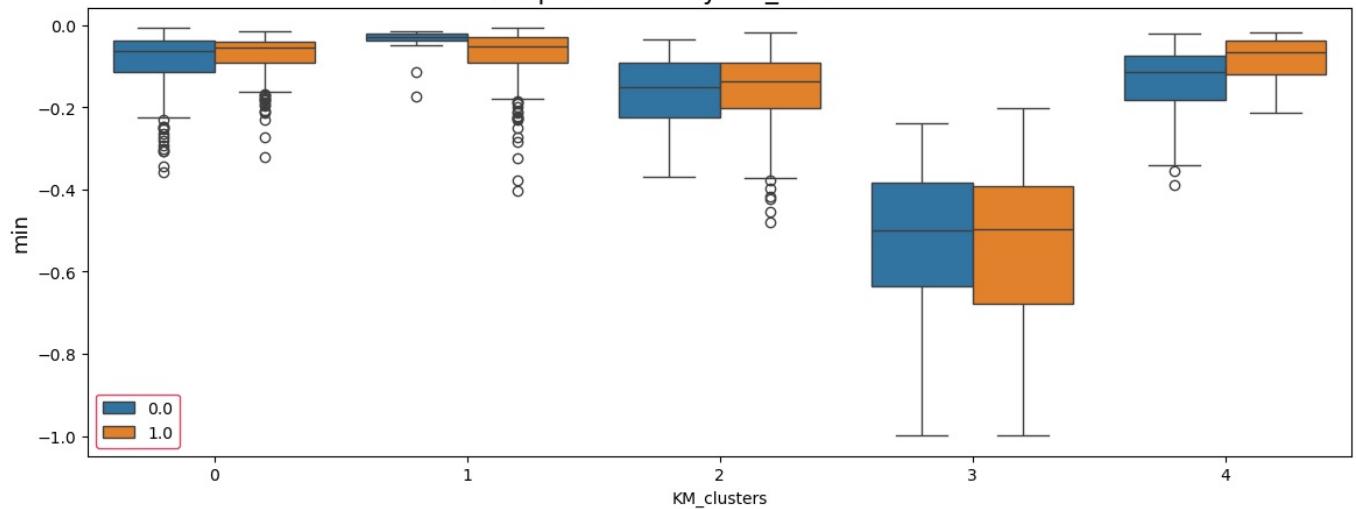
Boxplot of stft_skew by KM_clusters - sex



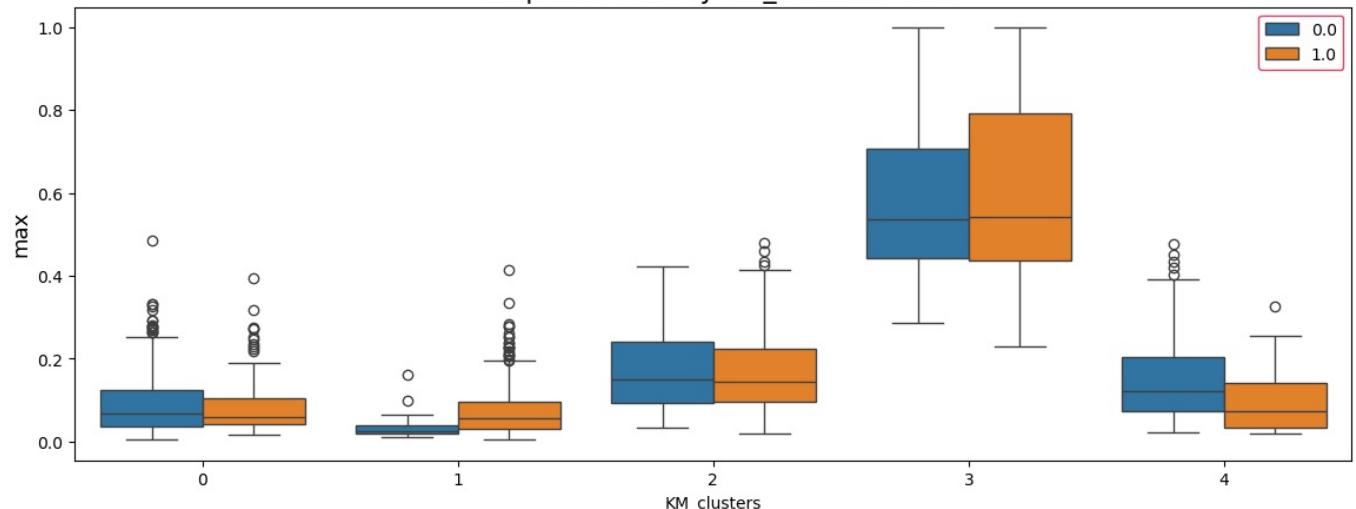
Boxplot of std by KM_clusters - sex



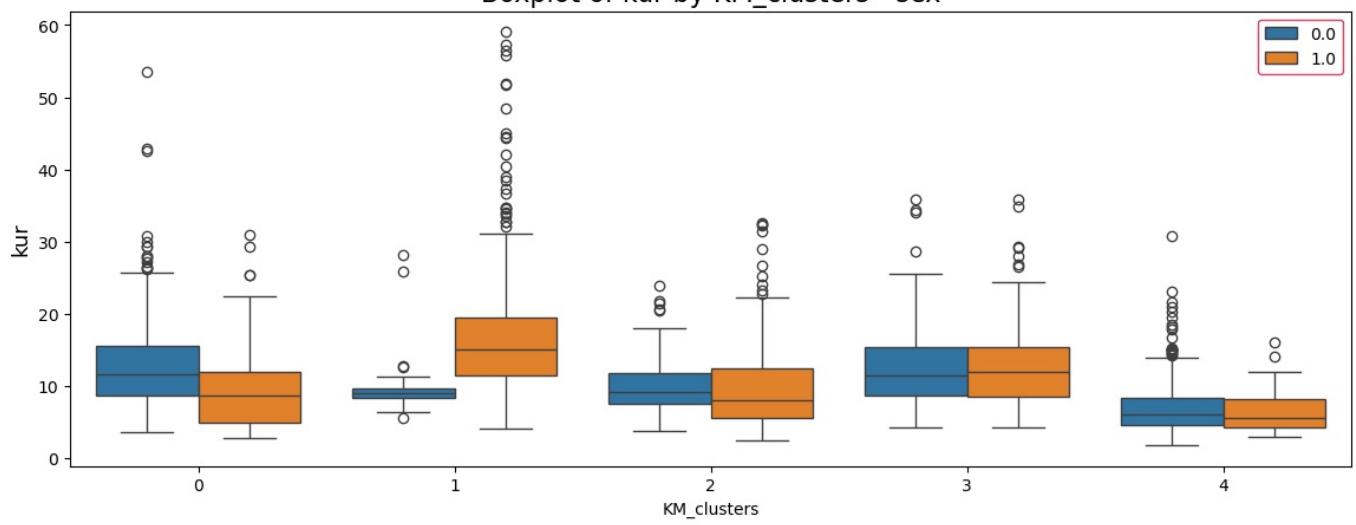
Boxplot of min by KM_clusters - sex

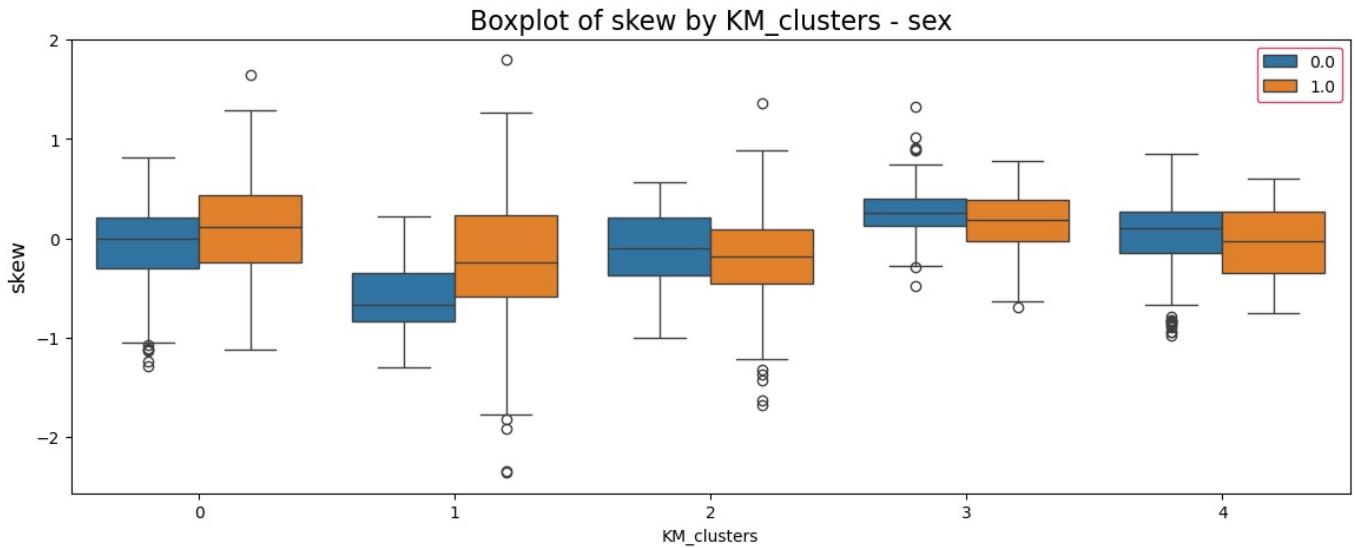


Boxplot of max by KM_clusters - sex



Boxplot of kur by KM_clusters - sex

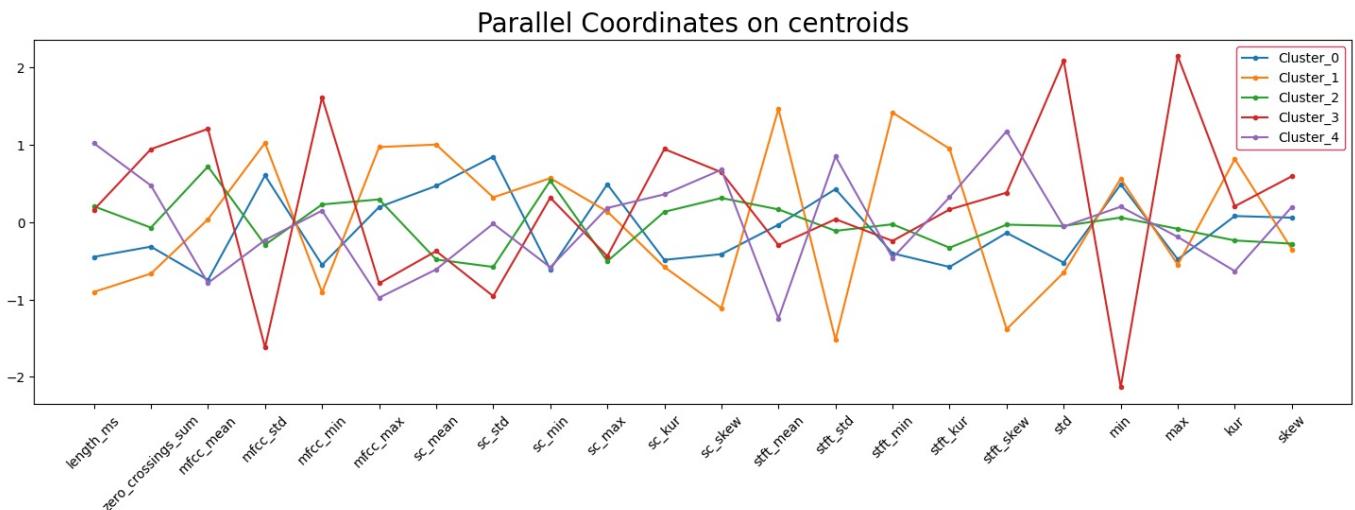




Infine analizziamo il **parallel coordinates** sui centroidi e notiamo che, anche se con oscillazioni di ampiezza diversa, su qualche features abbiamo un comportamento simile dei **centroidi dei cluster 3 e 4** e un comportamento simile dei **centroidi dei cluster 3 e 2**.

```
In [27]: plt.figure(figsize = (18, 5))
for l in np.unique(kmeans.labels_):
    plt.plot(numeric_cols, kmeans.cluster_centers_[l], marker = '.', label = 'Cluster_' + str(l))

plt.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)
# Rotate x-axis labels
plt.xticks(rotation=45)
plt.title('Parallel Coordinates on centroids', fontsize = 20)
plt.show()
```



DBSCAN

Una prima difficoltà che appare è il fatto che la nuvola di punti è piuttosto compatta e a primo impatto non sembrano esserci zone ad alta densità intervallate da zone a bassa densità.

Tuttavia il **DBSCAN** demarca la fine di un cluster e l'inizio di un altro in corrispondenza di un **density drop**. Il rischio è che tutti i punti siano indistintamente inseriti nel solito grande cluster.

Alla luce di tale complessità prima di procedere a tentativi sulla determinazione di:

- MinPts --> minimo numero di punti in un intorno di raggio ϵ per decretare che un punto è **core**

- $\epsilon \rightarrow$ raggio che sancisce la distanza radiale entro cui cercare almeno **MinPts** per decretare che un punto è **core**

è opportuno visualizzare l'andamento della **distanza dall'n-esimo vicino** con **n corrispondente a MinPts**. Questo ci permette di scegliere consapevolmente MinPts ed ϵ , in modo da avere un buon numero di **core points**.

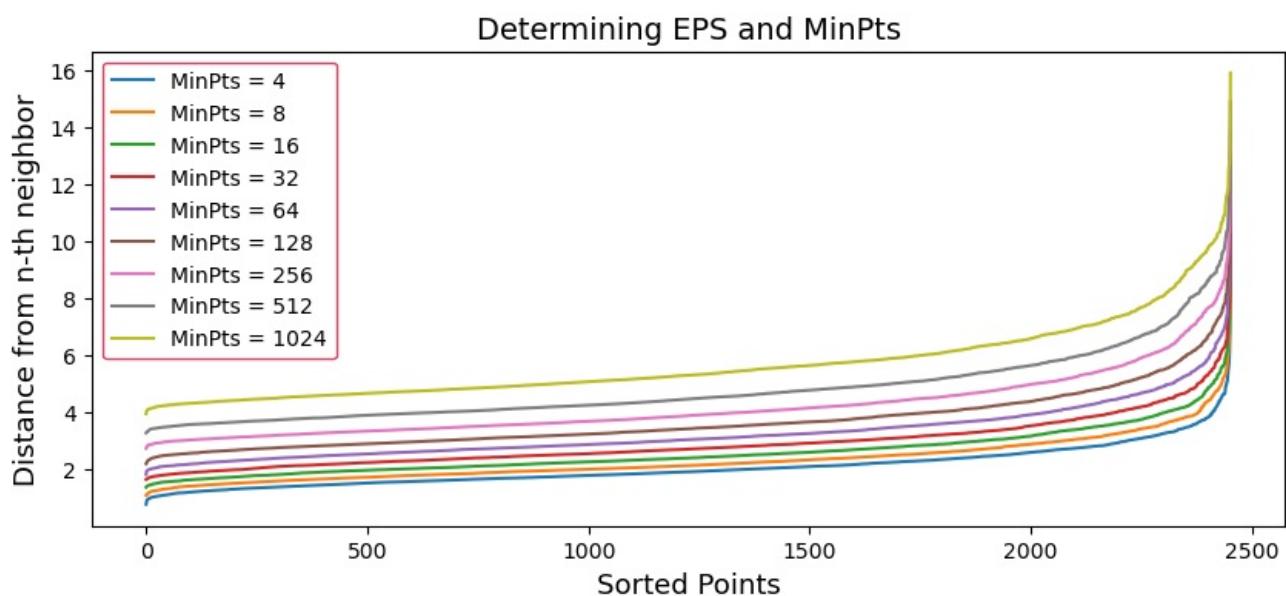
Per farlo sfruttiamo la matrice delle distanze, calcolata con la **distanza euclidea** in quanto gli attributi coinvolti nel clustering sono continui.

```
In [28]: # X è matrice già scalata con StandardScaler con le sole colonne numeriche da considerare come base per il clustering
```

```
Out[28]: array([[-0.59370043,  1.12146408, -1.05733963, ..., -0.2091685 ,  
                 -0.27170749,  0.70663217],  
                [-0.31452936,  0.27852801, -0.16429762, ...,  0.42569433,  
                 1.26425838,  0.74841435],  
                [ 0.96765158,  1.59300617, -0.39526838, ..., -0.06898976,  
                 -0.95588585,  0.77150368],  
                ...,  
                [ 1.91716757, -0.85148114,  0.59043138, ..., -0.42240248,  
                 -0.94332397,  0.09327539],  
                [-0.59370043, -0.86485242,  0.11799329, ..., -0.65390028,  
                 0.40108606, -0.13789756],  
                [-0.42653212, -0.94371565, -0.05605405, ..., -0.39289938,  
                 0.26766114,  2.37522233]])
```

```
In [29]: fig, ax = plt.subplots(figsize = (10, 4))
```

```
for min_pts in [4, 8, 16, 32, 64, 128, 256, 512, 1024]:  
    min_pts_distance = []  
    for i in range(len(X)):  
        neigh_idx = np.argsort(D[i])[min_pts]  
        min_pts_distance.append(D[i, neigh_idx])  
  
    ax.plot(range(len(X)), sorted(min_pts_distance), label=f'MinPts = {min_pts}' )  
  
ax.set_ylabel('Distance from n-th neighbor', fontsize = 13)  
ax.set_xlabel('Sorted Points', fontsize = 13)  
  
ax.set_title('Determining EPS and MinPts', fontsize = 14)  
  
ax.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)  
plt.show()
```



```
In [30]: from sklearn.cluster import DBSCAN
```

```
In [31]: dbscan = DBSCAN(min_samples = 64, eps = 4)  
dbscan.fit(X)
```

```
Out[31]: ▾          DBSCAN  
DBSCAN(eps=4, min_samples=64)
```

```
In [32]: dbscan.labels_
```

```
Out[32]: array([0, 0, 0, ..., 0, 0, 0], dtype=int64)
```

```
In [33]: label, sizes = np.unique(dbSCAN.labels_, return_counts = True)

# zip mi fa fare un ciclo for contemporaneamente su più vettori ma devono avere la stessa lunghezza
for l, s in zip(label, sizes):
    print('Cluster %s, size %s (%.2f)' % (l, s, s/len(X)))
```

Cluster -1, size 83 (0.03)
 Cluster 0, size 2369 (0.97)

```
In [34]: df_cluster['DBScan_clusters'] = dbSCAN.labels_
df_cluster.head()
```

```
Out[34]:   length_ms  zero_crossings_sum  mfcc_mean  mfcc_std  mfcc_min  mfcc_max  sc_mean  sc_std  sc_min  sc_max
0      3737.0        16995.0     -33.485947  134.654860  -755.22345  171.69092  5792.550744 3328.055457  0.0  13541.95902
1      3904.0        13906.0     -29.502108  130.485630  -713.98560  205.00770  5197.620555 4040.931570  0.0  12000.29042
2      4671.0        18723.0     -30.532463  126.577110  -726.06036  165.45653  4830.743037 3332.131300  0.0  12007.75117
3      3637.0        11617.0     -36.059555  159.725160  -842.94635  190.03609  5376.446478 4053.663073  0.0  12048.22389
4      4404.0        15137.0     -31.405996  122.125824  -700.70276  161.13400  5146.012474 3442.724109  0.0  12025.58270
```

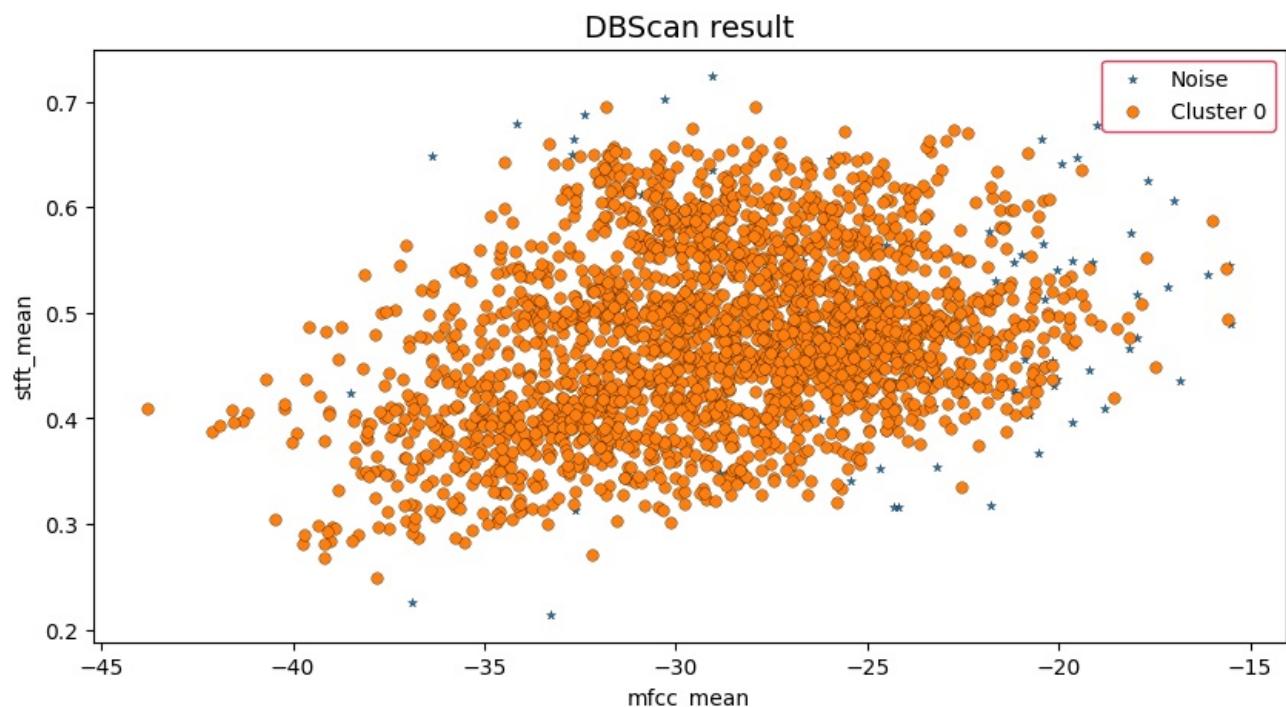
```
In [35]: x = 'mfcc_mean'
y = 'stft_mean'
```

```
fig, ax = plt.subplots(figsize = (10, 5))

for l in sorted(df_cluster['DBScan_clusters'].unique()):
    if l == -1:
        label = 'Noise'
        size = 20
        marker = '*'
    else:
        label = 'Cluster ' + str(l)
        size = 30
        marker = 'o'

    plt.scatter(df_cluster[df_cluster['DBScan_clusters'] == l][x],
                df_cluster[df_cluster['DBScan_clusters'] == l][y], label = label, s = size,
                edgecolor = 'black', linewidth = 0.2, marker = marker)

ax.legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10)
ax.set_xlabel(x)
ax.set_ylabel(y)
ax.set_title('DBScan result', fontsize = 14)
plt.show()
```



Nonostante i numerosi tentativi, non sembra esserci una combinazione di parametri che consenta di ottenere una situazione migliore di quella mostrata in figura. Infatti la condizione alla base del **DBScan** per decretare la fine di un cluster e l'inizio del successivo, ovvero i crollo di densità nei dati, non sembra essere una caratteristica di questo dataset. Pertanto sembra che non sarà possibile ottenere una situazione migliore di questa: **un solo cluster + qualche noise ai bordi della nuvola**.

Calcolo il centro dell'unico cluster ottenuto.

```
In [36]: def calc_centers(X, labels):
    centers = []
    for l in sorted(np.unique(labels)):
        if l == -1: # serve per il DBSCAN in cui il NOISE lo mette in classe -1
            continue
        mask = labels == l
        centers.append(np.mean(X[mask], axis = 0))
    return centers
```

```
In [37]: centers_dbSCAN = calc_centers(X, dbSCAN.labels_)
```

```
Out[37]: [array([-0.5880418 ,  1.05155791, -1.0234462 , -0.07973562,  0.02188472,
       -0.99521152,  0.72258861, -0.02736328, -1.08645385,  1.6581862 ,
       0.01570157, -0.28617435, -0.67690045,  0.15412062, -0.47041363,
       0.1485841 ,  0.81319958, -0.29372915,  0.21569372, -0.21538778,
      -0.25344991,  0.76311381])]
```

Come temuto il **DBScan** fallisce a causa dell'assenza di un drop nella **densità**.

Clustering Gerarchico

```
In [38]: # X è matrice già scalata con StandardScaler con le sole colonne numeriche da considerare come base per il clustering
```

```
Out[38]: array([[-0.59370043,  1.12146408, -1.05733963, ..., -0.2091685 ,
       -0.27170749,  0.70663217],
      [-0.31452936,  0.27852801, -0.16429762, ...,  0.42569433,
       1.26425838,  0.74841435],
      [ 0.96765158,  1.59300617, -0.39526838, ..., -0.06898976,
       -0.95588585,  0.77150368],
      ...,
      [ 1.91716757, -0.85148114,  0.59043138, ..., -0.42240248,
       -0.94332397,  0.09327539],
      [-0.59370043, -0.86485242,  0.11799329, ..., -0.65390028,
       0.40108606, -0.13789756],
      [-0.42653212, -0.94371565, -0.05605405, ..., -0.39289938,
       0.26766114,  2.37522233]])
```

```
In [39]: # Matrice delle distanze sparsa, in forma vettoriale, salva un sacco di spazio.
D_sparse = pdist(X)
D_sparse
```

```
Out[39]: array([3.66167418, 2.94395837, 3.77014013, ..., 4.88642629, 4.78186685,
       3.2973708 ])
```

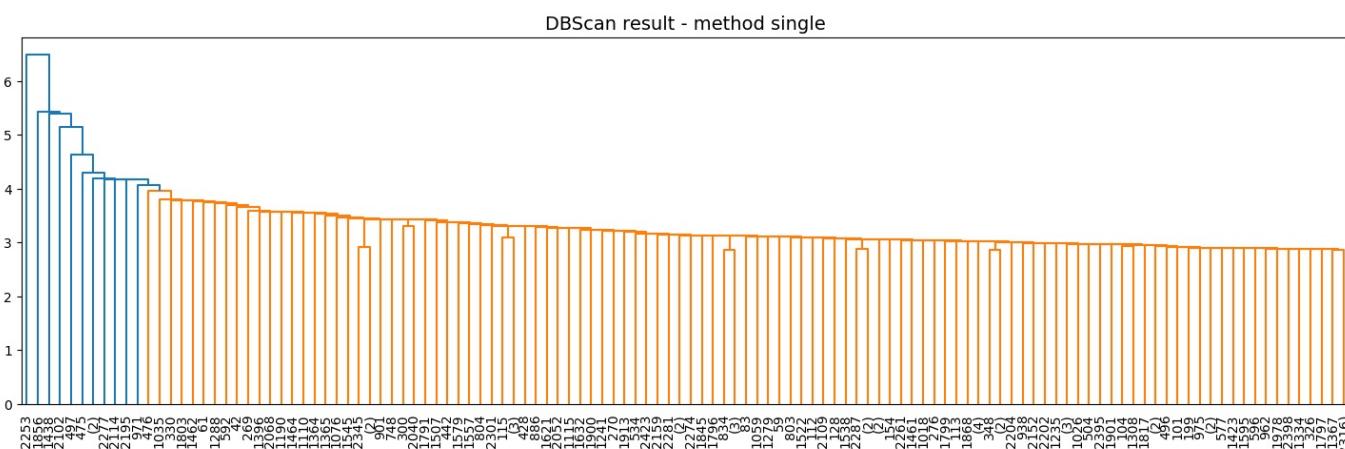
```
In [40]: from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
```

```
In [41]: from scipy.cluster.hierarchy import linkage, dendrogram, fcluster

method = 'single'

single_link = linkage(D_sparse, method = method)

plt.figure(figsize = (18, 5))
dendrogram = dendrogram(single_link, color_threshold = 4, p = 120, truncate_mode = 'lastp')
plt.title(f'DBScan result - method {method}', fontsize = 14)
plt.xticks(fontsize = 10)
plt.show()
```



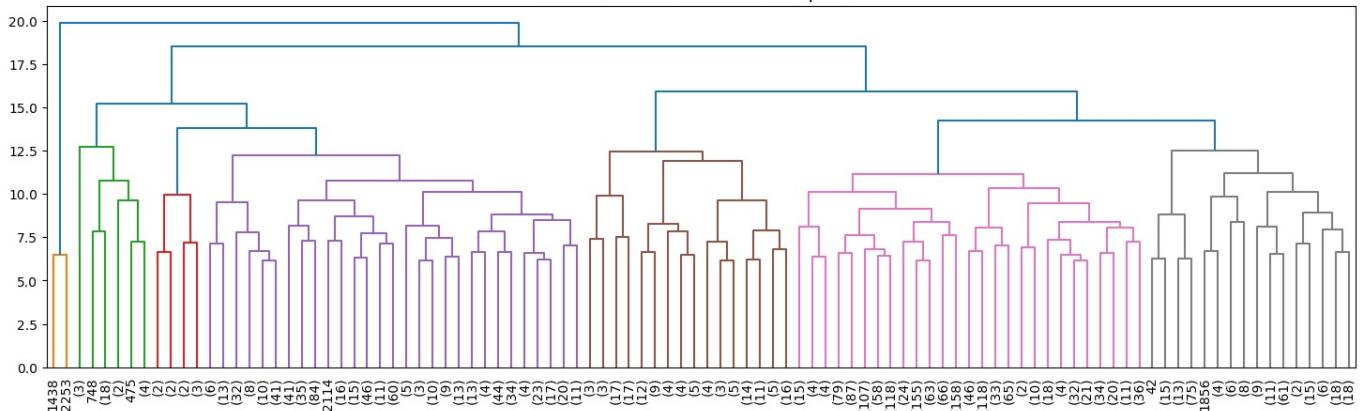
In [42]:

```
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
method = 'complete'

complete_link = linkage(D_sparse, method = method)

plt.figure(figsize = (18, 5))
dendrogram = dendrogram(complete_link, color_threshold = 13, p = 100, truncate_mode = 'lastp')
plt.title(f'DBScan result - method "{method}"', fontsize = 14)
plt.xticks(fontsize = 10)
plt.show()
```

DBScan result - method "complete"



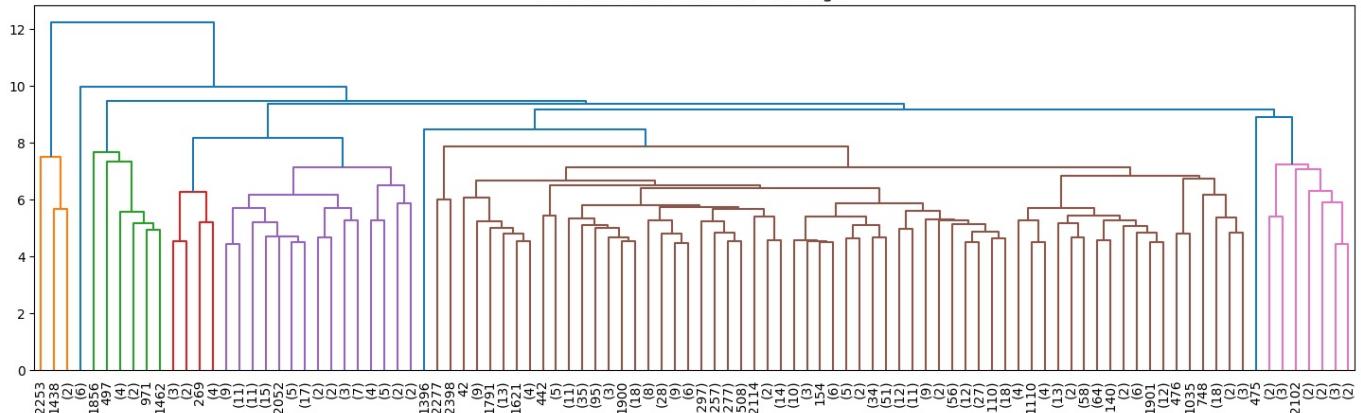
In [43]:

```
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
method = 'average'

avg_link = linkage(D_sparse, method = method)

plt.figure(figsize = (18, 5))
dendrogram = dendrogram(avg_link, color_threshold = 8, p = 100, truncate_mode = 'lastp')
plt.title(f'DBScan result - method "{method}"', fontsize = 14)
plt.xticks(fontsize = 10)
plt.show()
```

DBScan result - method "average"



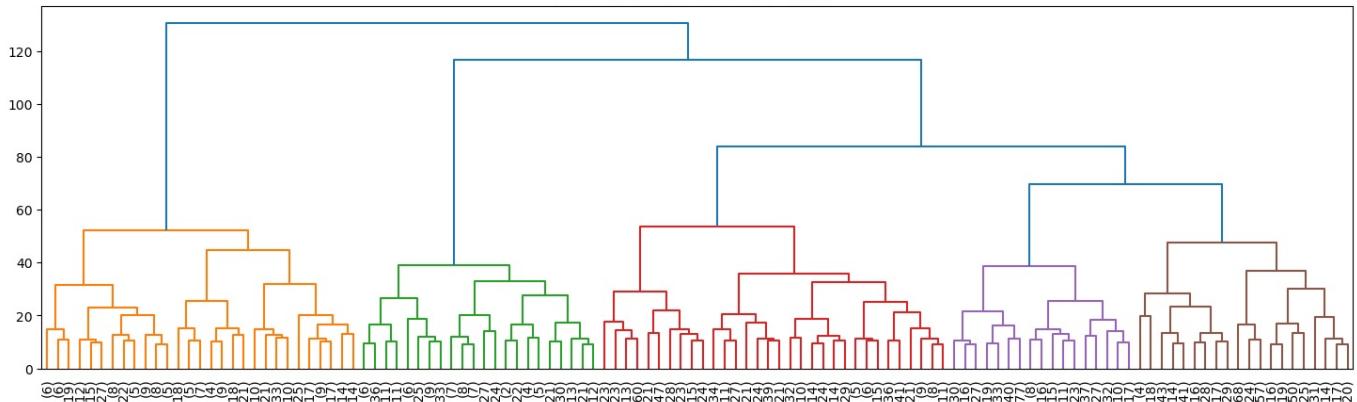
In [44]:

```
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
method = 'ward'

ward_link = linkage(D_sparse, method = method)

plt.figure(figsize = (18, 5))
dendrogram = dendrogram(ward_link, color_threshold = 60, p = 120, truncate_mode = 'lastp')
plt.title(f'DBScan result - method "{method}"', fontsize = 14)
plt.xticks(fontsize = 10)
plt.show()
```

DBScan result - method "ward"



Si nota che per ottenere un numero di clusters paragonabile utilizzando i 4 metodi, si deve impostare un valore di **threshold sulla distanza di taglio** piuttosto differente. In particolare notiamo che nel **single linkage** e nell'**average** tale valore è più contenuto e sotto 10. Saliamo leggermente sopra 10 nel caso del **complete linkage**. Il metodo **ward** è un caso a parte in quanto la threshold impostata è sopra 50.

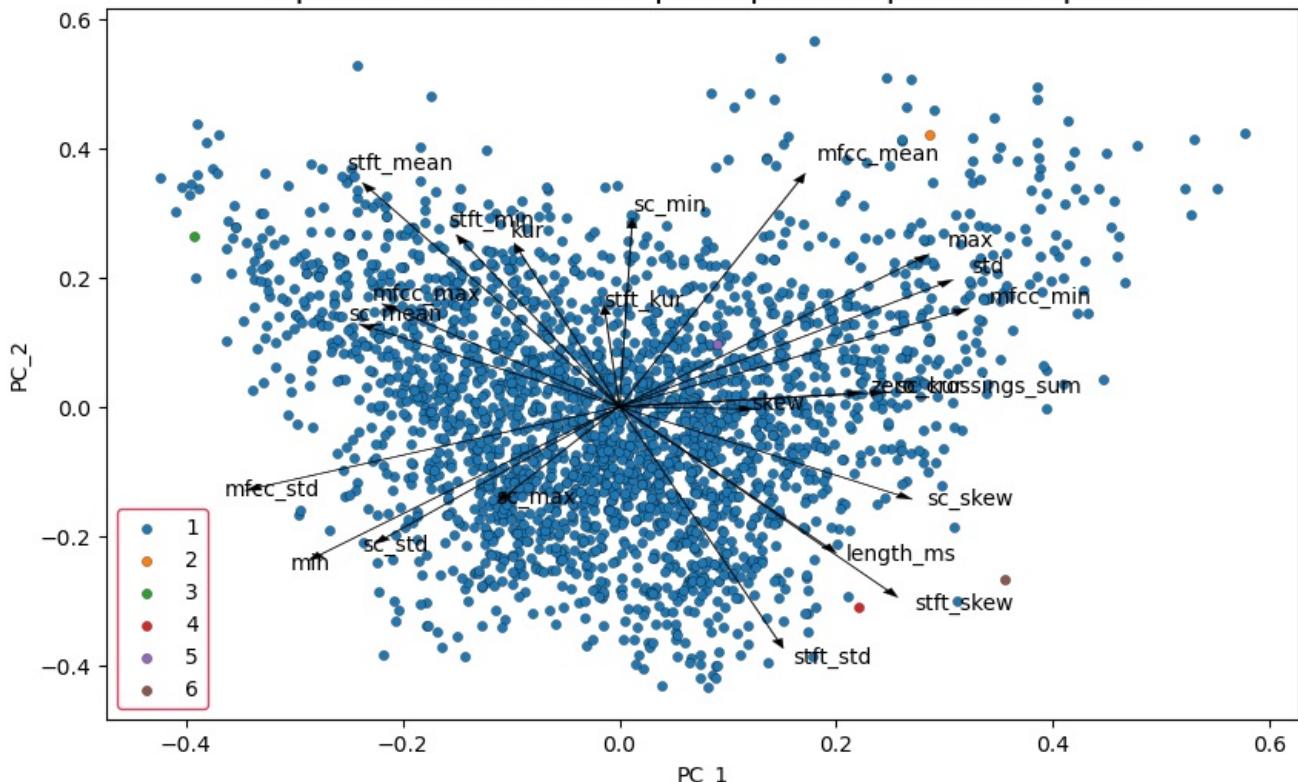
Adesso procediamo con il taglio effettivo per i vari metodi con la funzione **fcluster(links, t)** dove **t** è la **THRESHOLD** su cui ho precedentemente ragionato grazie al parametro **color_threshold**. Visualizziamo ogni volta il risultato per vedere che cluster otteniamo:

```
In [45]: labels_single = fcluster(single_link, t = 4.5, criterion = 'distance')
print(np.unique(labels_single))
```

```
[1 2 3 4 5 6]
```

```
In [46]: biplot(PC = X_pca,
          # with the transposition the eigenvectors V are on columns
          principal_directions = prin_dir.T,
          list_variable_names = numeric_cols,
          by_class_values = df_cluster['SingLink_clusters'],
          idx_1 = 0, idx_2 = 1)
```

Biplot of factors in the principal components' space

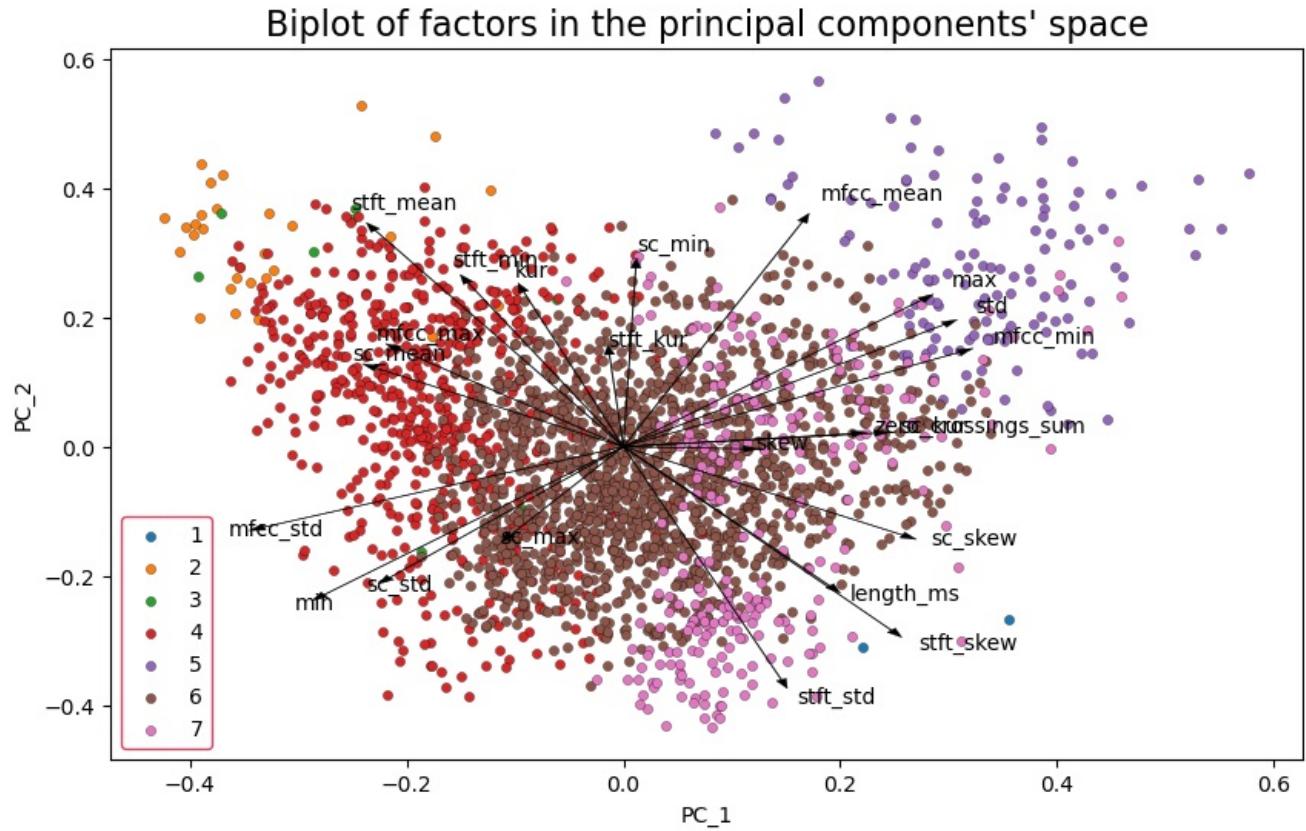


Single Link sembra non funzionare correttamente perché scendendo con il taglio per **aumentare il numero di clusters**, il solo effetto che otteniamo è quello di creare dei **singleton** poco significativi. Questo metodo sembra fallire.

```
In [47]: labels_complete = fcluster(complete_link, t = 13, criterion = 'distance')
print(np.unique(labels_complete))
```

```
[1 2 3 4 5 6 7]
```

```
In [48]: biplot(PC = X_pca,
            # with the transposition the eigenvectors V are on columns
            principal_directions = prin_dir.T,
            list_variable_names = numeric_cols,
            by_class_values = df_cluster['CompLink_clusters'],
            idx_1 = 0, idx_2 = 1)
```



Complete Link sembra fare un lavoro molto migliore.

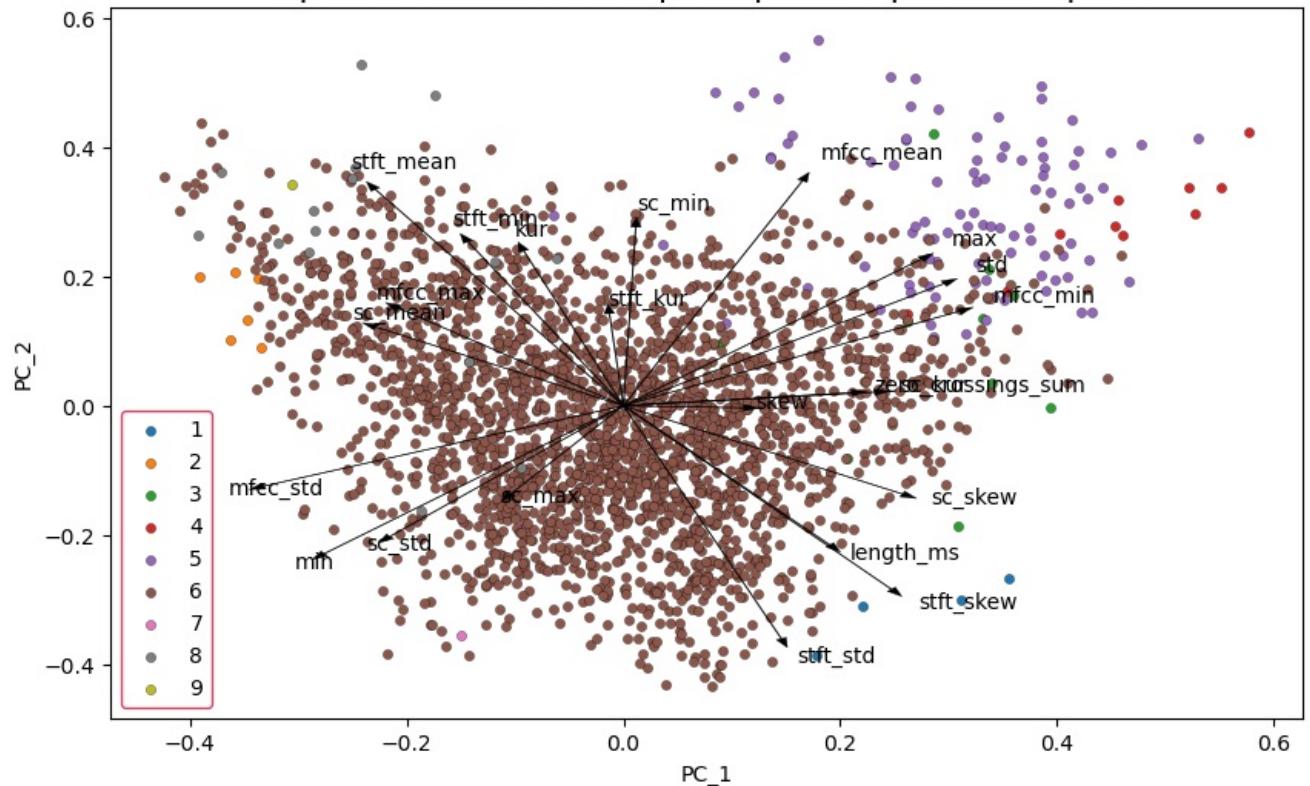
```
In [49]: labels_avg = fcluster(avg_link, t = 8, criterion = 'distance')
print(np.unique(labels_avg))

df_cluster['AvgLink_clusters'] = labels_avg
```

[1 2 3 4 5 6 7 8 9]

```
In [50]: biplot(PC = X_pca,
            # with the transposition the eigenvectors V are on columns
            principal_directions = prin_dir.T,
            list_variable_names = numeric_cols,
            by_class_values = df_cluster['AvgLink_clusters'],
            idx_1 = 0, idx_2 = 1)
```

Biplot of factors in the principal components' space



Average Link tende ad addensare quasi tutti i punti in un grande cluster centrale creando singleton sulla periferia

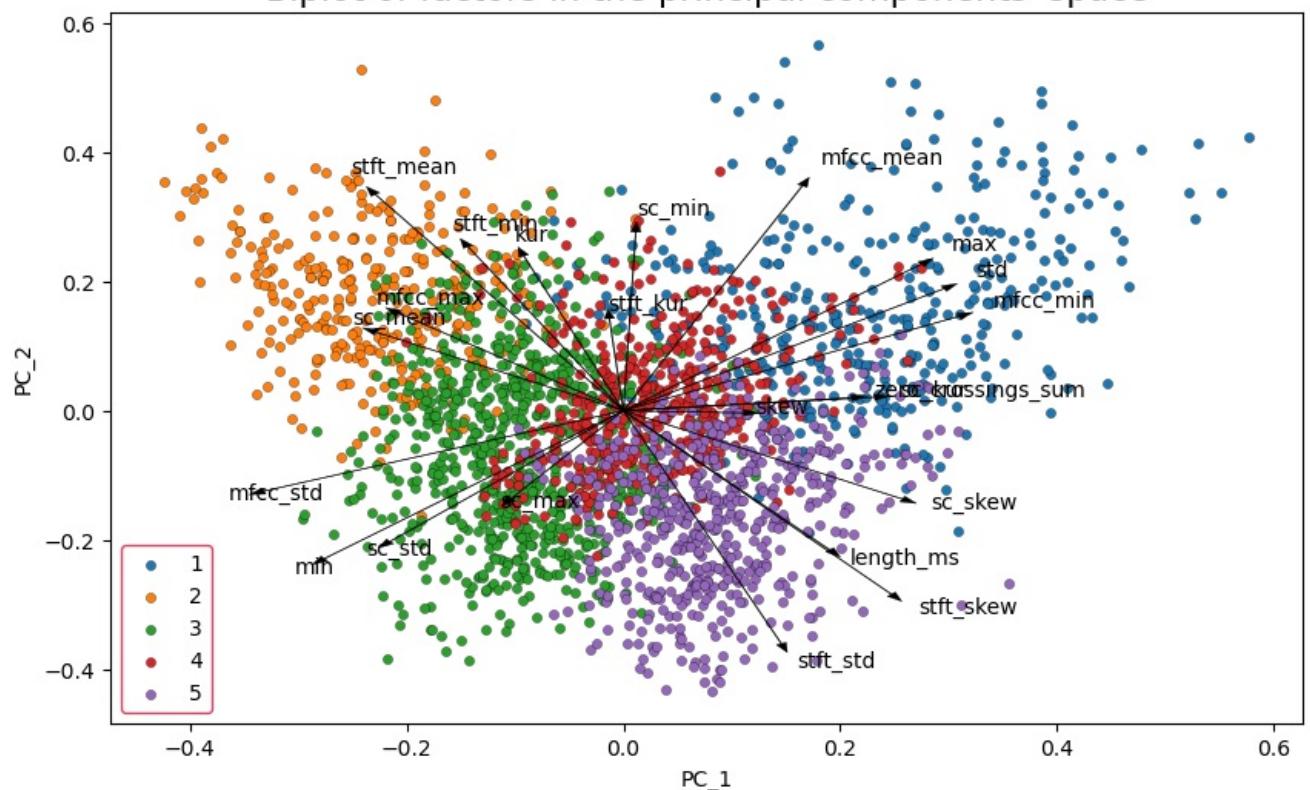
```
In [51]: labels_ward = fcluster(ward_link, t = 60, criterion = 'distance')
print(np.unique(labels_ward))

df_cluster['WardLink_clusters'] = labels_ward

[1 2 3 4 5]
```

```
In [52]: biplot(PC = X_pca,
           # with the transposition the eigenvectors V are on columns
           principal_directions = prin_dir.T,
           list_variable_names = numeric_cols,
           by_class_values = df_cluster['WardLink_clusters'],
           idx_1 = 0, idx_2 = 1)
```

Biplot of factors in the principal components' space



Ward sembra comportarsi molto simile a kmeans, creando cluster piuttosto omogenei e ben distribuiti nello spazio.

Dall'analisi grafica dei cluster, concludiamo che i due migliori algoritmi di clustering gerarchico sono:

- **Ward Link**
- **Complete Link**

In [53]:

```
# Calcolo dei centri
centers_single = calc_centers(X, labels_single)
centers_complete = calc_centers(X, labels_complete)
centers_avg = calc_centers(X, labels_avg)
centers_ward = calc_centers(X, labels_ward)
```

Clustering - Final Evaluation

Per trarre le conclusioni finali sul miglior clustering, abbiamo deciso di fare una valutazione incrociando le seguenti metrich:

- SSE (Sum of Squared Errors) : riflette il concetto di **coesione dei clusters**
- BSS (Between Sum of Squared Error) : riflette il concetto di **separazione dei clusters**
- SILHOUETTE : fa il **match tra coesione e separazione**

Pertanto abbiamo definito tre funzioni che data una **matrice X**, sulla base della matrice delle distanze **squareform(pdist(X))** e date le labels conseguenti ai vari algoritmi, permette di derivare le tre metriche di cui sopra.

In [54]:

```
from scipy.spatial.distance import cdist

def sse_score(X, labels, return_centers = False):
    cluster_centers = [] # mi creo la lista dei centri che vado piano a riempire
    tot_distance = 0.0 # il SSE
    for l in np.unique(labels):
        if l == -1: # Serve per gestire i NOISE del DBSCAN
            continue
        cluster = X[labels == l] # Solo osservazioni che hanno la label l
        center = cluster.mean(axis = 0) # Medio il valore degli attributi di queste osservazioni ottenendo il CEN
        # Mi dà un vettore con tutte le distanze tra le osservazioni che hanno label l e il centroide di quelle
        # appena calcolato, ovvero center
        dist = cdist(cluster, center.reshape(1,-1))**2
        tot_distance += dist.sum()
        cluster_centers.append(center) # Mi salvo il centroide
    cluster_centers = np.array(cluster_centers)
    if return_centers:
        return tot_distance, cluster_centers
    return tot_distance

def bss_score(X, labels):
    cluster_centers = [] # mi creo la lista dei centri che vado piano a riempire
    cluster_cardinalities = [] # mi creo la lista della cardinalità di ogni cluster, da riempire

    for l in np.unique(labels):
        if l == -1: # Serve per gestire i NOISE del DBSCAN
            continue
        cluster = X[labels == l] # Solo osservazioni che hanno la label l
        cardinality = cluster.shape[0] # cardinalità del cluster
        center = cluster.mean(axis=0) # Medio il valore degli attributi di queste osservazioni ottenendo il CEN
        cluster_cardinalities.append(cardinality)
        cluster_centers.append(center) # Mi salvo il centroide

    # Normalizzo le cardinalità dei singoli clusters rispetto alla dimensione del dataset
    cluster_cardinalities = np.array(cluster_cardinalities) / len(X)
    cluster_centers = np.array(cluster_centers)

    data_center = X.mean(axis = 0) # centroide dell'unico grande cluster formato da tutte le osservazioni

    # Calcolo della distanza tra i centroidi dei singoli cluster e l'unico grande centroide di tutto il dataset
    dist = cluster_cardinalities.reshape(-1, 1) * cdist(cluster_centers, data_center.reshape(1,-1))**2
    return dist.sum()

def sil_score(X, labels):
    D = squareform(pdist(X))
    sil_samples = sorted(silhouette_samples(D, labels, metric = 'precomputed'))
    return sum(sil_samples) / len(sil_samples)
```

In [55]:

```
for alg in ['KM_clusters', 'DBScan_clusters', 'SingLink_clusters', 'CompLink_clusters', 'AvgLink_clusters', 'Ward']
    labels = df_cluster[alg]
    sse, cluster_centers = sse_score(X, labels, return_centers = True)
```

```

bss = bss_score(X, labels)
silhouette = sil_score(X, labels)
print('%s \t SILHOUETTE %.3f \t SSE %.2f \t BSS %.2f' % (alg[:10], silhouette, sse, bss))

KM_cluster      SILHOUETTE 0.156    SSE 30022.45    BSS 9.76
DBScan_clu     SILHOUETTE 0.389    SSE 46814.95    BSS 0.03
SingLink_c      SILHOUETTE 0.263    SSE 53297.42    BSS 0.26
Complink_c     SILHOUETTE 0.117    SSE 36149.75    BSS 7.26
AvgLink_cl     SILHOUETTE 0.176    SSE 45941.15    BSS 3.26
WardLink_c     SILHOUETTE 0.106    SSE 32652.54    BSS 8.68

```

In [56]: # Verifica che torni la SSE (inertia_) del kmeans
kmeans.inertia_

Out[56]: 30022.451383336873

In [57]: silhouette_score(D, kmeans.labels_, metric = 'precomputed')

Out[57]: 0.15564280685095738

Per la valutazione finale abbiamo deciso di analizzare anche la **silhouette per ogni osservazione**, per **ciascuno degli algoritmi** per avere conferma che i valori medi precedentemente osservati NON fossero frutto di una compensazione tra **osservazioni clusterizzate eccezionalmente bene (alta silhouette)** e **osservazioni clusterizzate eccezionalmente male (bassa silhouette)**.

```

In [58]: list_algo = ['KM_clusters', 'DBScan_clusters', 'SingLink_clusters', 'CompLink_clusters', 'AvgLink_clusters', 'WardLink_clusters']

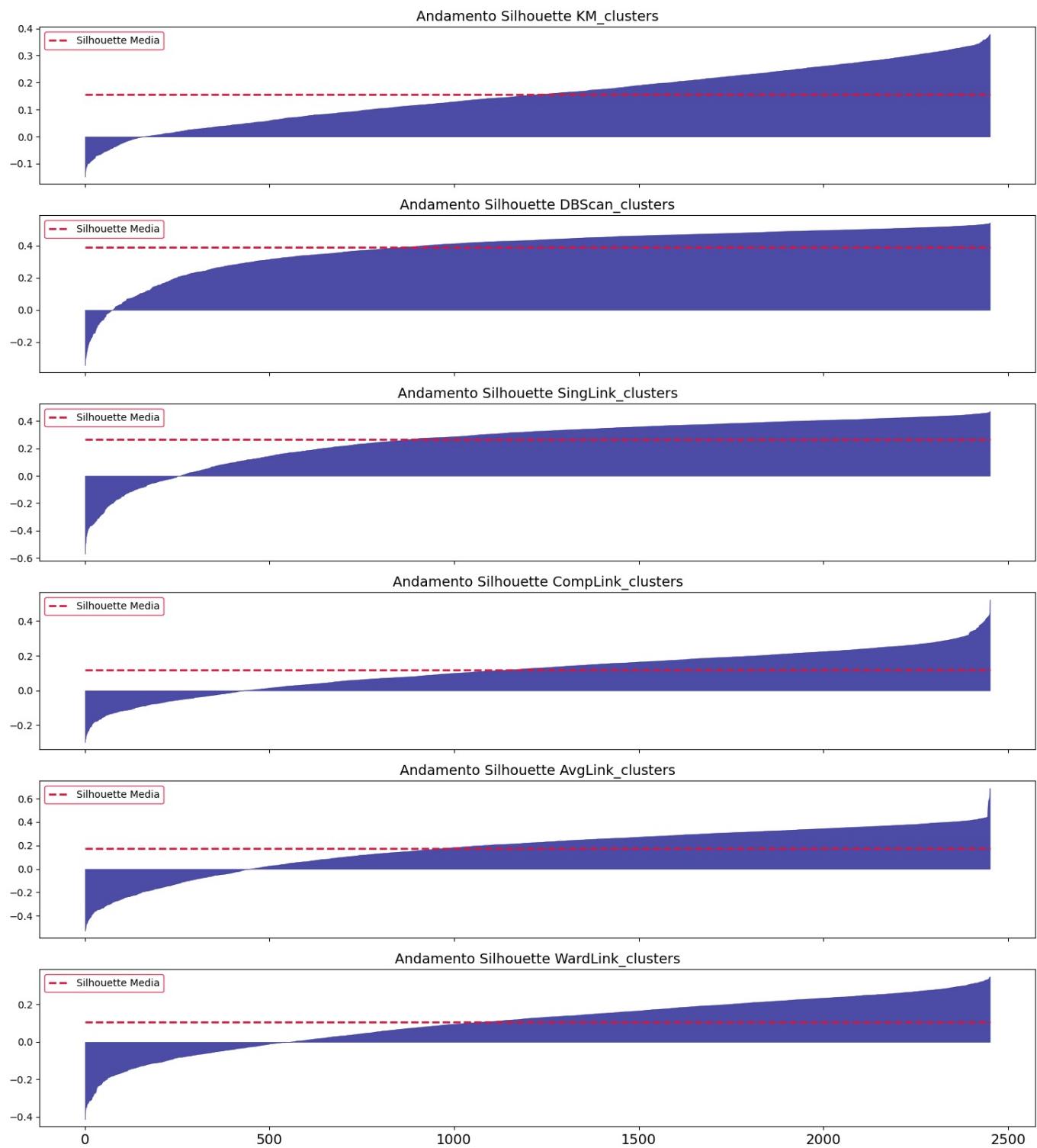
# Matrice delle distanze che uso per calcolare la SILHOUETTE
D = squareform(pdist(X))

fig, ax = plt.subplots(len(list_algo), 1, figsize = (18, 20), sharex = True)

for j in range(len(list_algo)):
    # Plotto i risultati di silhouette per ogni osservazione ordinati dai negativi ai positivi
    sil_samples = sorted(silhouette_samples(D, df_cluster[list_algo[j]], metric = 'precomputed'))
    ax[j].fill_between(range(len(sil_samples)), sil_samples, color = 'navy', alpha = 0.7, lw = 0.4)
    ax[j].plot([0, len(sil_samples)], np.mean(sil_samples)*np.ones(2), color = 'crimson', lw = 2, linestyle = 'solid')
    ax[j].set_title(f'Andamento Silhouette {list_algo[j]}', fontsize = 14)
    ax[j].legend(facecolor = 'white', edgecolor = 'crimson', fontsize = 10, loc = 'upper left')

plt.xticks(fontsize = 14)
plt.show()

```



Benché i numeri propendano per il **DBScan** e **Single Linkage** sappiamo che in realtà queste tecniche non consentono di ottenere dei risultati appaganti in quanto tendono a creare un unico grande cluster centrale, con eventuali altri punti rumore o singleton sulla periferia.

Possiamo concludere che quindi la **silhouette** ci ha tratti in inganno: le tecniche che mostrano una silhouette migliore sono proprio quelle che generano il risultato grafico meno appagante. Il fatto è che la propensione a generare un unico grande cluster centrale e piccoli cluster sulla periferia, **abbassa molto la BSS**, distorcendo il calcolo finale.

Cercando quindi di fare un bilanciamento tra metriche calcolate e risultato grafico, possiamo concludere che le migliori tecniche sono:

- **K-means**
- **Ward Linkage**

e un gradino sotto **Complete Linkage**.

Questo trova riscontro anche nel fatto che queste ultime tecniche menzionate sono anche quelle che permettono di ottenere un **SSE minore**.

In []:

Processing math: 100%

DMML 2024: Final Project

Francesco Peria, Maria Paola Sforza Fogliani, Andrea Vitali

```
In [33]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import seaborn as sns
```

```
In [35]: pd.set_option('display.max_columns', 100)
df = pd.read_csv('A Data Understanding & Preparation Output.csv', sep=',', skipinitialspace=True)
df.head()
```

```
Out[35]:   emotion  vocal_channel  emotional_intensity  statement  repetition  sex  length_ms  zero_crossings_sum  mfcc_mean  mfcc_st
0    fearful          0.0            0.0           0.0        1.0    0.0    3737.0      16995.0  -33.485947  134.65486
1     angry          0.0            0.0           0.0        0.0    0.0    3904.0      13906.0  -29.502108  130.48563
2    happy           0.0            1.0           0.0        1.0    0.0    4671.0      18723.0  -30.532463  126.57711
3  surprised          0.0            0.0           1.0        0.0    0.0    3637.0      11617.0  -36.059555  159.72516
4    happy           1.0            1.0           0.0        1.0    0.0    4404.0      15137.0  -31.405996  122.12582
```

```
In [37]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2452 entries, 0 to 2451
Data columns (total 28 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   emotion          2452 non-null   object  
 1   vocal_channel    2452 non-null   float64 
 2   emotional_intensity  2452 non-null   float64 
 3   statement         2452 non-null   float64 
 4   repetition        2452 non-null   float64 
 5   sex               2452 non-null   float64 
 6   length_ms         2452 non-null   float64 
 7   zero_crossings_sum  2452 non-null   float64 
 8   mfcc_mean         2452 non-null   float64 
 9   mfcc_std          2452 non-null   float64 
 10  mfcc_min          2452 non-null   float64 
 11  mfcc_max          2452 non-null   float64 
 12  sc_mean           2452 non-null   float64 
 13  sc_std            2452 non-null   float64 
 14  sc_min            2452 non-null   float64 
 15  sc_max            2452 non-null   float64 
 16  sc_kur            2452 non-null   float64 
 17  sc_skew           2452 non-null   float64 
 18  stft_mean         2452 non-null   float64 
 19  stft_std          2452 non-null   float64 
 20  stft_min          2452 non-null   float64 
 21  stft_kur          2452 non-null   float64 
 22  stft_skew         2452 non-null   float64 
 23  std               2452 non-null   float64 
 24  min               2452 non-null   float64 
 25  max               2452 non-null   float64 
 26  kur               2452 non-null   float64 
 27  skew              2452 non-null   float64 
dtypes: float64(27), object(1)
memory usage: 536.5+ KB
```

C) Classification

In questa sezione, esploreremo, analizzeremo e valuteremo diversi metodi di classificazione al fine di prevedere due variabili categoriche fondamentali nel nostro dataset: Emotion e Sex.

Sex

Decision Tree on Sex

Iniziamo con il modello chiamato **Decision Tree** che costruisce una struttura ad albero che simula una serie di split logici basati sulle

variabili usate per l'addestramento per arrivare a una conclusione sulla variabile target. Per utilizzare il **Decision Tree**, dobbiamo far sì che gli attributi categorici nel nostro dataframe siano trasformati in variabili binarie (0 e 1) per consentire al modello di apprendere da tali attributi. Questo passaggio, chiamato binarizzazione, è già stato fatto per tutti gli attributi categorici del dataset con solo due valori distinti nella sezione di clustering.

Tuttavia, per quanto riguarda l'attributo Emotion, poiché esso presenta più di due valori distinti, adotteremo la tecnica di **one-hot encoding**. Questa tecnica trasforma ciascun valore univoco dell'attributo Emotion in una variabile binaria separata creando una colonna per ogni emozione possibile, come "happy", "sad", "angry", ecc., con valori 1 o 0 a seconda della presenza o assenza di quella specifica emozione in ciascuna osservazione. Questa trasformazione preserva l'integrità delle informazioni categoriche senza introdurre un ordine fittizio tra le categorie.

```
In [49]: df_noobj = pd.get_dummies(df, columns=['emotion'])
```

Abbiamo quindi creato un dataframe chiamato **df_noobj** privo di oggetti

```
In [364]: df_noobj.dtypes
```

```
Out[364]: vocal_channel      float64
emotional_intensity     float64
statement                 float64
repetition                float64
sex                       float64
length_ms                  float64
zero_crossings_sum        float64
mfcc_mean                  float64
mfcc_std                   float64
mfcc_min                   float64
mfcc_max                   float64
sc_mean                     float64
sc_std                      float64
sc_min                      float64
sc_max                      float64
sc_kur                      float64
sc_skew                     float64
stft_mean                   float64
stft_std                    float64
stft_min                    float64
stft_kur                    float64
stft_skew                   float64
std                         float64
min                         float64
max                         float64
kur                          float64
skew                        float64
emotion_angry                bool
emotion_calm                 bool
emotion_disgust               bool
emotion_fearful                bool
emotion_happy                  bool
emotion_neutral                bool
emotion_sad                    bool
emotion_surprised               bool
dtype: object
```

```
In [366]: df_noobj
```

```
Out[366]:   vocal_channel  emotional_intensity  statement  repetition  sex  length_ms  zero_crossings_sum  mfcc_mean  mfcc_std  mfcc_min  mfcc_max  sc_mean  sc_std  sc_min  sc_max  sc_kur  sc_skew  stft_mean  stft_std  stft_min  stft_kur  stft_skew  std  min  max  kur  skew  emotion_angry  emotion_calm  emotion_disgust  emotion_fearful  emotion_happy  emotion_neutral  emotion_sad  emotion_surprised
0          0.0            0.0           0.0         0.0    1.0    0.0       3737.0      16995.0   -33.485947  134.654860  -755  ...
1          0.0            0.0           0.0         0.0    0.0    0.0       3904.0      13906.0   -29.502108  130.485630  -713  ...
2          0.0            1.0           0.0         0.0    1.0    0.0       4671.0      18723.0   -30.532463  126.577110  -726  ...
3          0.0            0.0           1.0         1.0    0.0    0.0       3637.0      11617.0   -36.059555  159.725160  -842  ...
4          1.0            1.0           0.0         0.0    1.0    0.0       4404.0      15137.0   -31.405996  122.125824  -700  ...
...          ...            ...           ...         ...    ...    ...       ...        ...        ...        ...        ...        ...
2447         0.0            1.0           1.0         1.0    0.0    1.0       4605.0      9871.0   -30.225578  158.845500  -855  ...
2448         0.0            0.0           0.0         0.0    0.0    1.0       4171.0      8963.0   -31.160332  157.499700  -825  ...
2449         1.0            1.0           0.0         1.0    1.0    0.0       5239.0      9765.0   -26.135280  138.133210  -768  ...
2450         0.0            0.0           1.0         0.0    0.0    1.0       3737.0      9716.0   -28.242815  159.943400  -868  ...
2451         1.0            0.0           0.0         0.0    1.0    1.0       3837.0      9427.0   -29.019236  149.188950  -799  ...

2452 rows × 35 columns
```

Sceglieremo come variabile target l'attributo **sex** e prendiamo tutte le altre colonne come quelle che useremo per allenare il modello predittivo del Decision Tree

```
In [369]: target = 'sex'  
columns = [c for c in df_noobj.columns if c not in target]
```

```
In [371]: columns
```

```
Out[371]: ['vocal_channel',  
          'emotional_intensity',  
          'statement',  
          'repetition',  
          'length_ms',  
          'zero_crossings_sum',  
          'mfcc_mean',  
          'mfcc_std',  
          'mfcc_min',  
          'mfcc_max',  
          'sc_mean',  
          'sc_std',  
          'sc_min',  
          'sc_max',  
          'sc_kur',  
          'sc_skew',  
          'stft_mean',  
          'stft_std',  
          'stft_min',  
          'stft_kur',  
          'stft_skew',  
          'std',  
          'min',  
          'max',  
          'kur',  
          'skew',  
          'emotion_angry',  
          'emotion_calm',  
          'emotion_disgust',  
          'emotion_fearful',  
          'emotion_happy',  
          'emotion_neutral',  
          'emotion_sad',  
          'emotion_surprised']
```

Splitteremo il dataset in due parti, una riservata al **Training & Validation Set** composta dall'80% dei dati e il **Test Set** composto dal 20% dei dati. All'interno della porzione Training & Validation Set, un 20% viene riservato al Validation Set (16% del totale) e il rimanente al Training Set (64%). Abbiamo scelto questa proporzione in quanto è una di quelle più comunemente usate nei modelli predittivi, ma successivamente andremo a controllare se esistono partizioni che migliorano il modello. La creazione dei due set è stratificata (con il parametro `stratify` settato a `y`) e riproducibile (con il parametro `random_state=0`). L'importanza di creare un Validation Set che sia indipendente permette di valutare e ottimizzare gli iperparametri del modello senza toccare il Test Set. Questo approccio può aiutare a prevenire l'overfitting sul Test Set.

```
In [374]: X = df_noobj[columns].values  
y = df_noobj[target].values
```

```
In [376]: from sklearn.model_selection import train_test_split
```

```
In [378]: X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
```

```
In [380]: X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val)
```

```
In [382]: from sklearn.tree import DecisionTreeClassifier
```

```
In [384]: from sklearn.metrics import accuracy_score, confusion_matrix
```

Dopo aver importato l'algoritmo `DecisionTreeClassifier`, lo inizializziamo usando il **gini index** per misurare la qualità degli split. In questo primo passaggio, in maniera arbitraria, decidiamo di non mettere un limite alla **profondità dell'albero**, di impostare come 2 il **numero minimo di campioni necessari per dividere un nodo**, e come 1 il **numero minimo di campioni che devono essere inclusi in una foglia**. L'albero viene prima addestrato sul Training set che abbiamo scelto e successivamente testato sul Validation set. L'accuracy ottenuta è 0.90 che indica che il modello ha predetto correttamente la classe di appartenenza nel 90% dei casi nel set di dati di validazione.

```
In [387]: clf = DecisionTreeClassifier(  
            criterion='gini',  
            max_depth=None,  
            min_samples_split=2,  
            min_samples_leaf=1,  
            ccp_alpha=0.0,
```

```

        random_state=0
    )
clf.fit(X_train, y_train)

y_pred = clf.predict(X_val)
accuracy_score(y_val, y_pred)

```

Out[387]: 0.9033078880407125

Visualizziamo la **confusion matrix**, dove:

- Nella prima cella (in alto a sinistra) abbiamo i True Positives (TP)
- Nella seconda cella (in alto a destra) abbiamo i False Negatives (FN)
- Nella terza cella (in basso a sinistra) abbiamo i False Positives (FP)
- Nella quarta cella (in basso a destra) abbiamo i True Negatives (TN)

Il numero di TP e TN rispetto al FP e FN rispecchia l'alta accuratezza del modello

In [76]: `confusion_matrix(y_val, y_pred)`

Out[76]: `array([[170, 23],
 [15, 185]])`

Con le seguenti due celle di codice valutiamo come varia l'accuratezza al variare della dimensione del Training Set. Dal grafico sembrerebbe che assegnare l'80% del dataset al Training Set dia i risultati migliori per l'accuratezza.

```

In [401]: accuracy_mean_list = list()
accuracy_std_list = list()
for p in np.arange(0.1, 1.0, 0.1):
    accuracy_list_p = list()
    for i in range(0, 10):
        index = np.random.choice(np.arange(0, len(X_train)), int(len(X_train) * p), replace=False)
        clf = DecisionTreeClassifier(
            criterion='gini',
            max_depth=None,
            min_samples_split=2,
            min_samples_leaf=1,
            ccp_alpha=0.0,
            random_state=0)
        clf.fit(X_train[index], y_train[index])

        y_pred = clf.predict(X_val)
        accuracy_list_p.append(accuracy_score(y_val, y_pred))
    accuracy_mean_list.append(np.mean(accuracy_list_p))
    accuracy_std_list.append(np.std(accuracy_list_p))

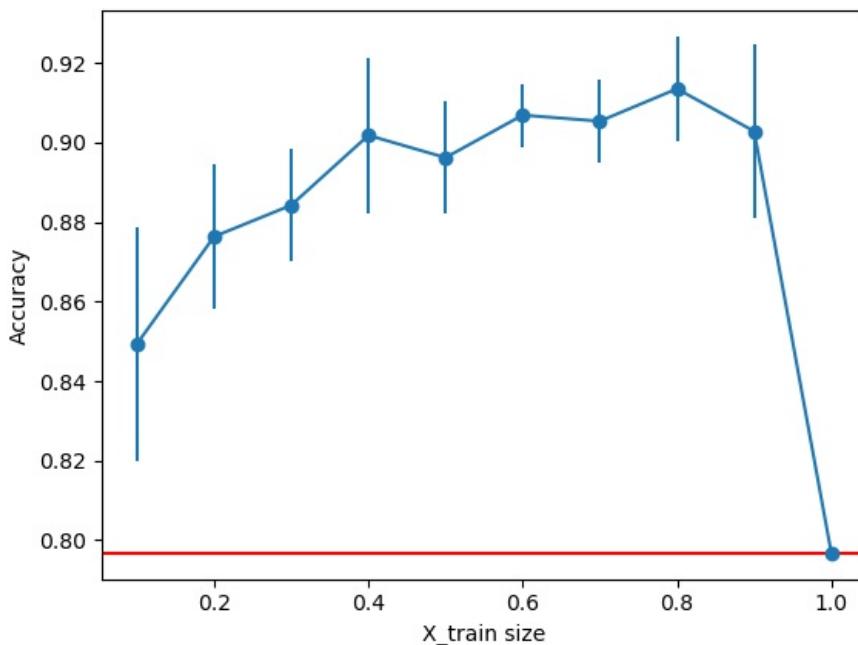
```

```

In [402]: accuracy_mean_list.append(0.7967914438502673)
accuracy_std_list.append(0.0)

plt.errorbar(x=np.arange(0.1, 1.1, 0.1), y=accuracy_mean_list,
             yerr=accuracy_std_list, marker='o')
plt.axhline(y=0.7967914438502673, color='r')
plt.ylabel('Accuracy')
plt.xlabel('X_train size')
plt.show()

```



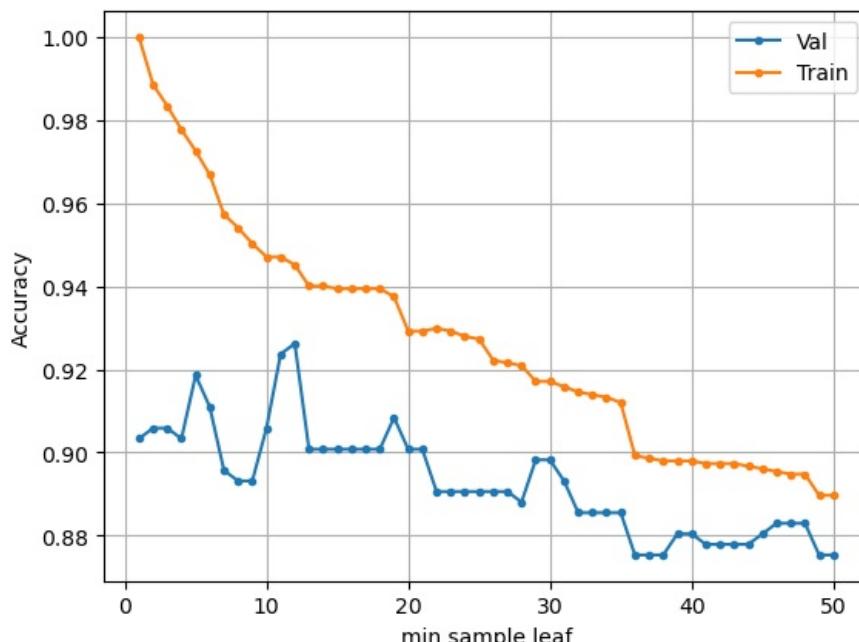
Nelle seguenti celle andiamo a esplorare come la selezione degli **iperparametri** per il Decision Tree vari l'accuratezza. Ne andremo ad analizzare 3:

1. *il numero minimo di campioni che deve avere ogni foglia dell'albero decisionale*
2. *il numero minimo di campioni necessari perché un nodo sia splittabile*
3. *la massima profondità dell'albero decisionale.*

Con le seguenti due celle di codice valutiamo come varia l'accuratezza con il **numero minimo di campioni che deve avere ogni foglia dell'albero decisionale**. Dal grafico sembrerebbe che i valori di 11 e 12 siano quelli che danno un'accuratezza migliore. Generalmente quando possibile, è preferibile scegliere numeri più alti per evitare overfitting quindi per le successive analisi imposteremo il parametro a 12.

```
In [405...]  
min_sample_leaf_list = np.arange(1, 50+1, 1)  
accuracy_val_list = list()  
accuracy_train_list = list()  
for min_sample_leaf in min_sample_leaf_list:  
    clf = DecisionTreeClassifier(  
        criterion='gini',  
        max_depth=None,  
        min_samples_split=2,  
        min_samples_leaf=min_sample_leaf,  
        ccp_alpha=0.0,  
        random_state=0  
    )  
    clf.fit(X_train, y_train)  
  
    y_pred = clf.predict(X_val)  
    accuracy_val_list.append(accuracy_score(y_val, y_pred))  
  
    y_pred_train = clf.predict(X_train)  
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

```
In [406...]  
plt.plot(min_sample_leaf_list, accuracy_val_list, label='Val', marker='.')  
plt.plot(min_sample_leaf_list, accuracy_train_list, label='Train', marker='.')  
plt.ylabel('Accuracy')  
plt.xlabel('min sample leaf')  
plt.grid('white')  
plt.legend()  
plt.show()
```



Con le seguenti due celle di codice andiamo a invece valutare come varia l'accuratezza con il **numero minimo di campioni necessari perché un nodo sia splittabile**. Dal grafico sembrerebbe che l'accuratezza sia ottimizzata con valori ≤ 25 . Per minimizzare il rischio di overfitting e avere un modello predittivo più robusto è generalmente consigliabile avere valori più alti e quindi scegliamo 25 per l'analisi successiva.

```
In [410...]  
min_sample_split_list = np.arange(2, 50+1, 1)  
min_sample_split_list  
  
accuracy_val_list = list()  
accuracy_train_list = list()  
for min_sample_split in min_sample_split_list:  
    clf = DecisionTreeClassifier(  
        criterion='gini',  
        max_depth=None,  
        min_samples_split=min_sample_split,  
        min_samples_leaf=12,  
        ccp_alpha=0.0,  
        random_state=0  
    )  
    clf.fit(X_train, y_train)  
  
    y_pred = clf.predict(X_val)  
    accuracy_val_list.append(accuracy_score(y_val, y_pred))  
  
    y_pred_train = clf.predict(X_train)  
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

```

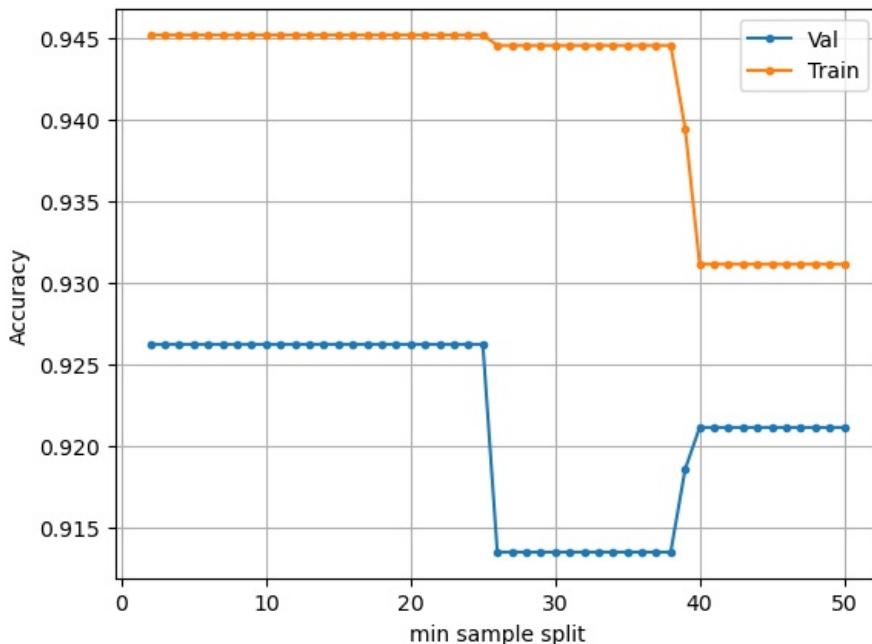
        criterion='gini',
        max_depth=None,
        min_samples_split=min_sample_split,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
clf.fit(X_train, y_train)

y_pred = clf.predict(X_val)
accuracy_val_list.append(accuracy_score(y_val, y_pred))

y_pred_train = clf.predict(X_train)
accuracy_train_list.append(accuracy_score(y_train, y_pred_train))

plt.plot(min_sample_split_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_split_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample split')
plt.grid('white')
plt.legend()
plt.show()

```



Per concludere, esamineremo come varia l'accuratezza al variare della **massima profondità** dell'albero decisionale. Dall'analisi grafica, sembra che l'accuratezza raggiunga il massimo con valore di profondità 7.

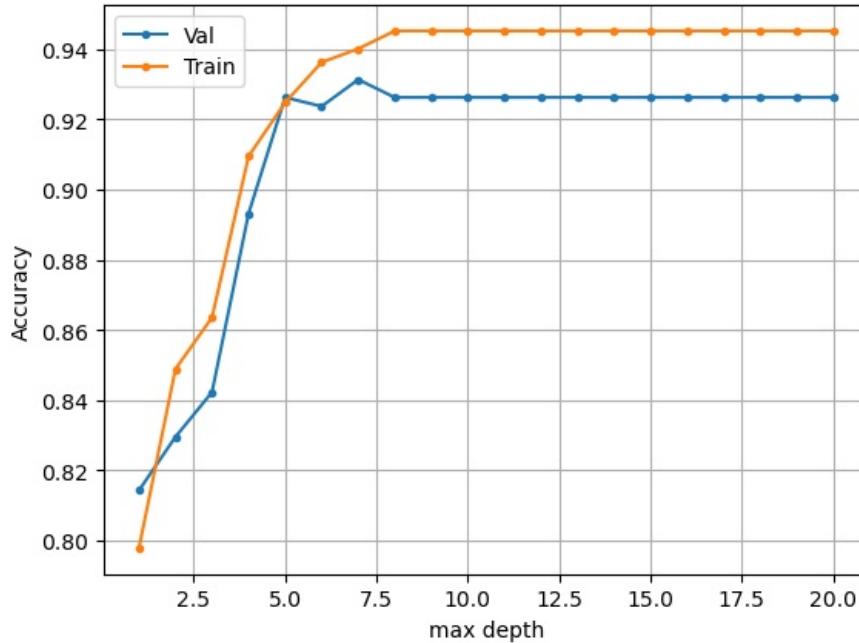
```
In [412]: max_depth_list = np.arange(1, 20+1, 1).tolist() + [None]

accuracy_val_list = list()
accuracy_train_list = list()
for max_depth in max_depth_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=max_depth,
        min_samples_split=25,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))

plt.plot(max_depth_list, accuracy_val_list, label='Val', marker='.')
plt.plot(max_depth_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('max depth')
plt.grid('white')
plt.legend()
plt.show()
```



Proviamo adesso ad usare **RandomizedSearch** che è una tecnica automatica di ottimizzazione della libreria **scikit-learn** per trovare i migliori parametri da impostare per il DecisionTree.

```
In [95]: from sklearn.model_selection import RandomizedSearchCV
```

```
In [99]: param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0, 1, 0.1)
}
```

```
In [101]: random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_
```

```
Out[101]: {'min_samples_split': 15,
           'min_samples_leaf': 7,
           'max_depth': 12,
           'ccp_alpha': 0.0}
```

```
In [102]: random.best_estimator_
```

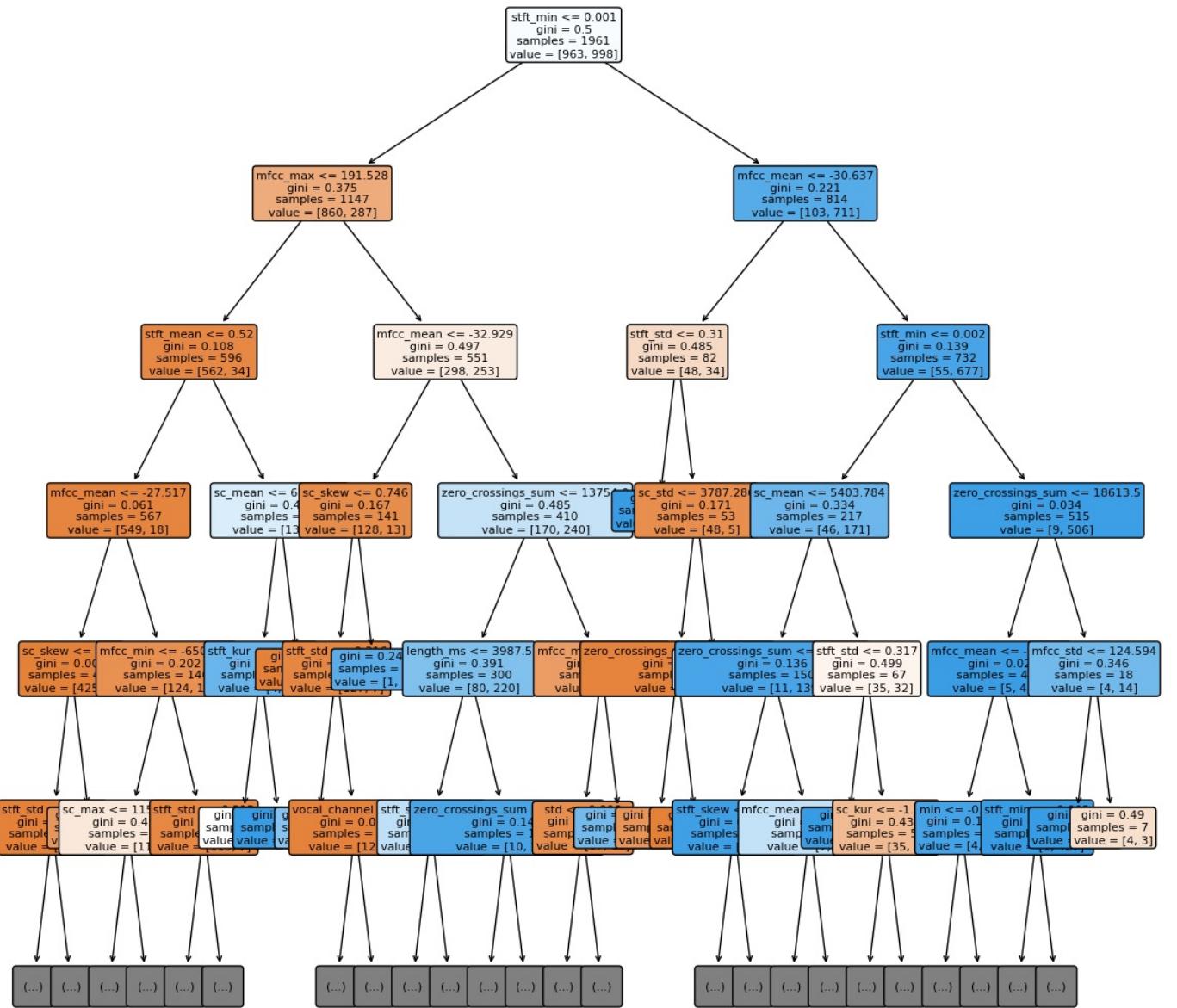
```
Out[102]: 
DecisionTreeClassifier(max_depth=12, min_samples_leaf=7, min_samples_split=15,
                       random_state=0)
```

Si può vedere che i valori trovati da **RandomizedSearch** non sono gli stessi di quelli che abbiamo precedentemente individuato cambiando un parametro alla volta. Abbiamo deciso di usare i parametri di RandomizedSearch visto che l'algoritmo è più efficiente nel valutare e ottimizzare le diverse combinazioni dei tre parametri.

```
In [104]: clf = random.best_estimator_
```

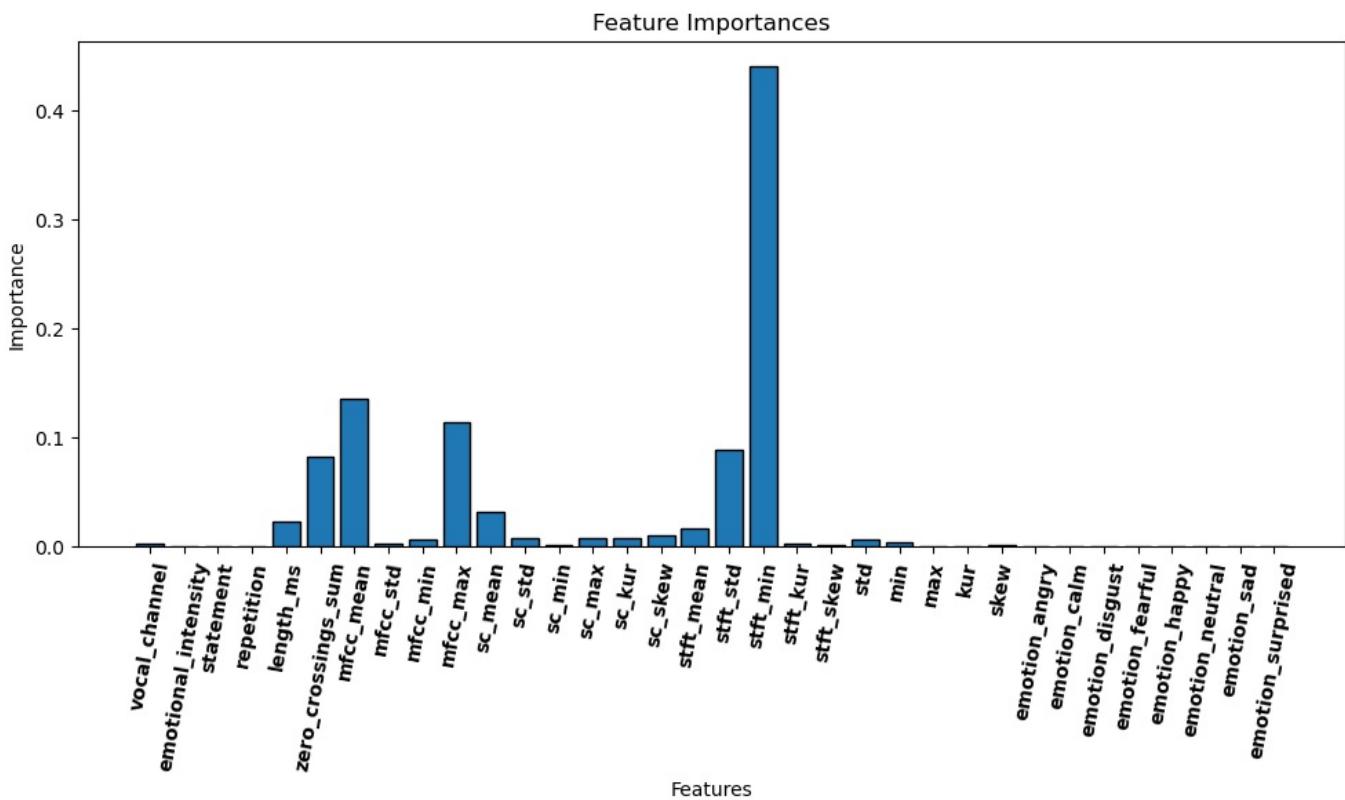
```
In [105]: from sklearn.tree import plot_tree, export_text
```

```
In [106]: plt.figure(figsize=(14, 14))
plot_tree(clf,
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=8,
          max_depth=5
         )
plt.show()
```



Con la seguente barchart visualizziamo quali sono le features del dataset più importanti per prevedere il valore "Sex" con il modello del DecisionTree. La feature più importante sembra essere "stft_min" seguita da "mfcc_max" e "mfcc_mean"

```
In [108]: plt.figure(figsize=(10, 6))
plt.bar(columns, clf.feature_importances_, edgecolor='k')
plt.xticks(rotation=80, fontweight='bold')
plt.title('Feature Importances')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.tight_layout()
plt.show()
```



Nel Classification Report stampato sotto, possiamo vedere che l'accuratezza del Decision Tree sul Test set è elevata (0.90), indicando una buona capacità di classificazione generale. Tuttavia, l'analisi di Precision, Recall, F1-Score e Confusion Matrix rivela un piccolo sbilanciamento nelle performance tra le classi. In particolare, il modello mostra una maggiore precisione nel prevedere la classe 0 (F), con meno falsi positivi, ma soffre di un numero più alto di falsi negativi per la stessa classe. Al contrario, per la classe 1 (M), si osserva un incremento dei falsi positivi rispetto ai falsi negativi.

```
In [110]: y_pred = clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

```
Out[110]: 0.8961303462321792
```

```
In [111]: from sklearn.metrics import precision_score, recall_score, f1_score, classification_report
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)
y_pred = clf.predict(X_test)
print(classification_report(y_pred, y_test))

y_pred_trainval = clf.predict(X_train_val)

print(classification_report(y_pred_trainval, y_train_val))
```

	precision	recall	f1-score	support
0.0	0.91	0.88	0.90	248
1.0	0.88	0.91	0.90	243
accuracy			0.90	491
macro avg	0.90	0.90	0.90	491
weighted avg	0.90	0.90	0.90	491
	precision	recall	f1-score	support
0.0	0.95	0.96	0.96	958
1.0	0.96	0.96	0.96	1003
accuracy			0.96	1961
macro avg	0.96	0.96	0.96	1961
weighted avg	0.96	0.96	0.96	1961

Plottiamo adesso la **ROC** che è uno strumento grafico usato per valutare le prestazioni di un modello predittivo su una variabile target binaria. La curva traccia il tasso di TP rispetto a FP. Calcoliamo l'area sotto alla curva che risulta essere 1 che indica un modello altamente performante. Questo valore è consistente con l'alta accuratezza calcolata in precedenza e implica che il modello predittivo è efficiente sulla variabile target Sex.

```
In [113]: from sklearn.metrics import roc_curve, auc
```

```

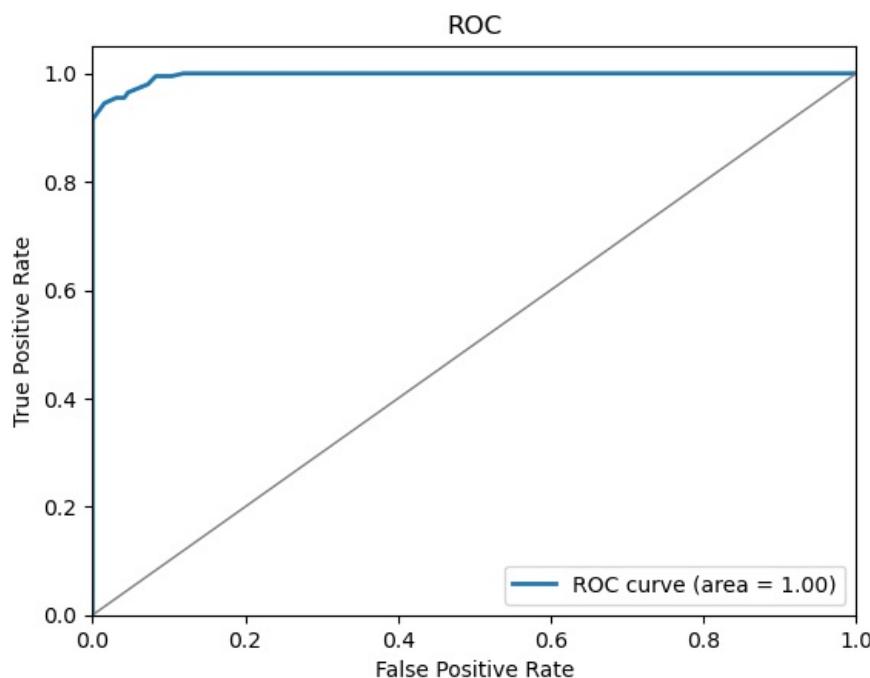
import matplotlib.pyplot as plt

y_pred = clf.predict_proba(X_val)[:, 1]

fpr, tpr, thresholds = roc_curve(y_val, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='gray', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend(loc="lower right")
plt.show()

```



KNN on Sex

Sebbene il Decision Tree abbia dato ottimi risultati sulla variabile target Sex, esploriamo ora un approccio differente adottando il **K-Nearest Neighbors (KNN)** come classificatore. Questo modello si basa sull'idea che le osservazioni vicine nello spazio dimensionale del dataset tendono ad appartenere alla stessa classe. In pratica, la classificazione di un nuovo campione viene determinata dalla maggioranza delle classi tra i suoi K vicini più prossimi.

Iniziamo con un numero di K=5 (il numero di *vicini* in base ai quali è fatta la classificazione). Il dataset è diviso in due parti: una riservata al **Training & Validation Set** composta dall'80% dei dati e il **Test Set** composto dal 20% dei dati. All'interno della porzione Training & Validation Set, un 20% viene riservato al Validation Set (16% del totale) e il rimanente al Training Set (64%). Abbiamo scelto questa proporzione in quanto è una di quelle più comunemente usate nei modelli predittivi, ma successivamente andremo a controllare se esistono partizioni che migliorano il modello. La creazione dei due set è stratificata (con il parametro `stratify` settato a `y`) e riproducibile (con il parametro `random_state=0`)

Adottando il KNN, vogliamo valutare come un approccio basato sulla prossimità possa confrontarsi con la metodologia di decisione gerarchica utilizzata dai Decision Tree.

```

In [125]: from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

target = 'sex'
columns = [c for c in df_noobj.columns if c not in target]

X = df_noobj[columns].values
Y = df_noobj[target].values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, Y, test_size=0.2, stratify=Y, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val)

```

```

print("Training set", X_train.shape, y_train.shape)
print("Validation set", X_val.shape, y_val.shape)
print("Test set", X_test.shape, y_test.shape)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_val)
y_pred_train = knn.predict(X_train)
y_pred_trainval=knn.predict(X_train_val)

print(accuracy_score(y_val, y_pred), accuracy_score(y_train, y_pred_train))
print(classification_report(y_val, y_pred))
print(classification_report(y_pred_trainval, y_train_val))

```

Training set (1568, 34) (1568,)
 Validation set (393, 34) (393,)
 Test set (491, 34) (491,)
 0.9338422391857506 0.9464285714285714

	precision	recall	f1-score	support
0.0	0.95	0.92	0.93	193
1.0	0.92	0.95	0.94	200
accuracy			0.93	393
macro avg	0.93	0.93	0.93	393
weighted avg	0.93	0.93	0.93	393

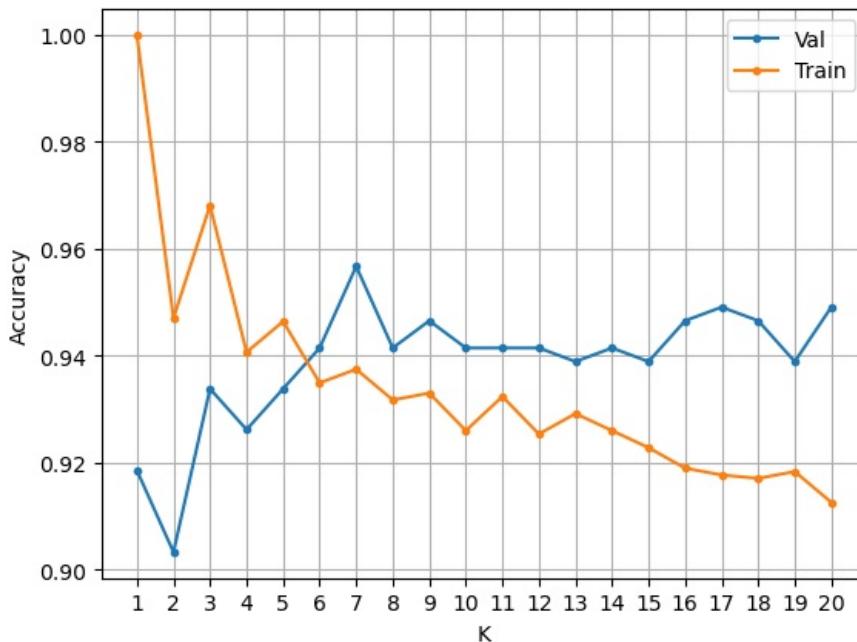
	precision	recall	f1-score	support
0.0	0.95	0.94	0.94	977
1.0	0.94	0.95	0.94	984
accuracy			0.94	1961
macro avg	0.94	0.94	0.94	1961
weighted avg	0.94	0.94	0.94	1961

L'accuratezza risulta già molto alta, ma comunque cerchiamo di ottimizzarla variando il parametro **K**.

```
In [128]: acc_val_list = list()
acc_train_list = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_val)
    y_pred_train = knn.predict(X_train)
    acc_val_list.append(accuracy_score(y_val, y_pred))
    acc_train_list.append(accuracy_score(y_train, y_pred_train))
```

Da questo plot sembrerebbe che il miglior numero di K sia 7 però questo potrebbe essere influenzato dalla scelta del training set e del test set che è randomica

```
In [131]: plt.plot(np.arange(1, 20+1, 1), acc_val_list, label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), acc_train_list, label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



Per ovviare al problema della scelta randomica del training set possiamo usare una tecnica chiamata **Repeated Holdout**. Questo metodo prevede la ripetizione della divisione del dataset in Training Set e Test Set per diverse iterazioni, consentendo di valutare la stabilità e l'affidabilità del modello su più suddivisioni dei dati. In questo caso, abbiamo eseguito la valutazione su 5 diverse combinazioni di Training Set e Test Set. Dall'analisi dei risultati, attraverso la visualizzazione, abbiamo identificato che K=7 offre le migliori prestazioni complessive.

In [134]...

```
nbr_holdout = 5
acc_val_list_all = list()
acc_train_list_all = list()
for i in range(nbr_holdout):
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                    test_size=0.2,
                                                    stratify=y_train_val,
                                                    random_state=i)

    acc_val_list = list()
    acc_train_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train)
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train, y_pred_train))

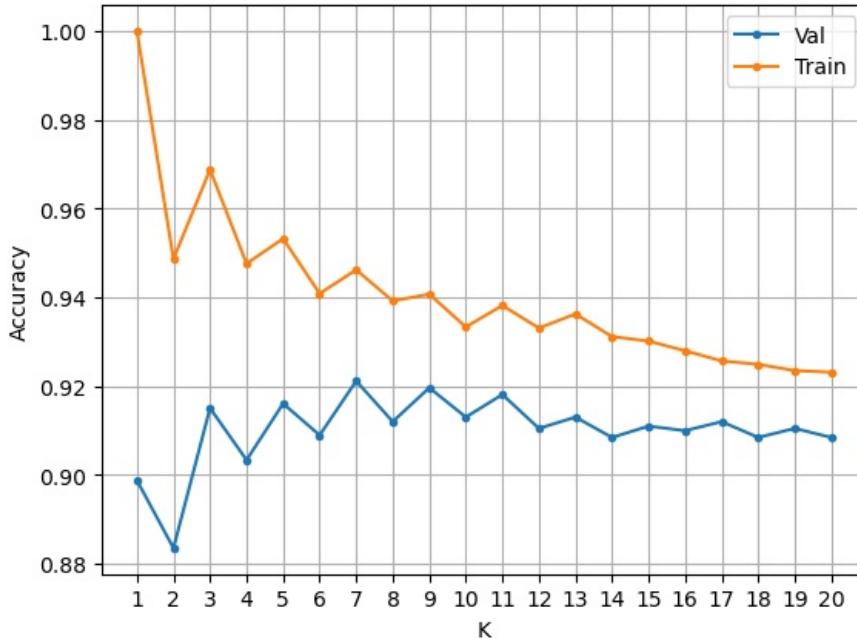
    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)
```

In [135]...

```
acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

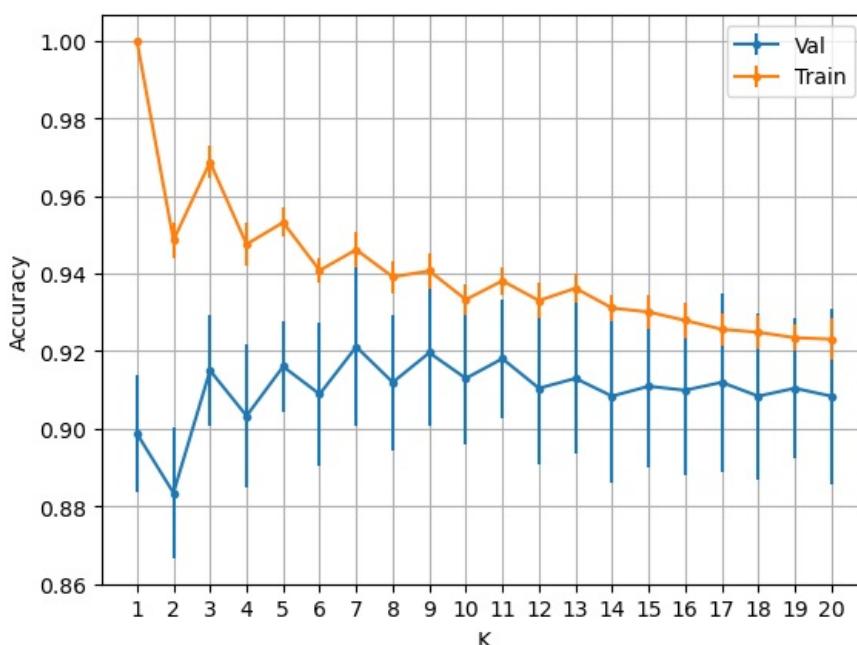
In [136]...

```
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_val_list_all, axis=0), label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_train_list_all, axis=0), label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



Facciamo lo stesso plot includendo la deviazione standard per ogni valore di K. Questo ci permette di valutare non solo la performance media ma anche la variabilità delle prestazioni. Dall'analisi, K=7 si conferma essere il valore che offre le migliori prestazioni complessive, combinando una buona accuratezza con una variabilità relativamente bassa.

```
In [141]: plt.errorbar(x=np.arange(1, 20+1, 1),
                   y=np.mean(acc_val_list_all, axis=0),
                   yerr=np.std(acc_val_list_all, axis=0),
                   label='Val', marker='.')
plt.errorbar(x=np.arange(1, 20+1, 1),
             y=np.mean(acc_train_list_all, axis=0),
             yerr=np.std(acc_train_list_all, axis=0),
             label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



Cross Validation

A questo punto è importante però sottolineare che il problema della scelta casuale dei Test Set e Training Set potrebbe persistere, dato che, in teoria, la distribuzione della variabile target potrebbe essere sbilanciata, influenzando così i risultati dei plot precedentemente analizzati. Per mitigare questo potenziale bias, possiamo ricorrere alla tecnica della **Cross Validation**. Questo metodo ci permette di valutare più accuratamente il nostro modello attraverso diverse iterazioni, utilizzando ogni volta diverse suddivisioni del dataset in Training Set e Test Set, assicurando così una più equa rappresentazione della variabile target in ogni fase di valutazione.

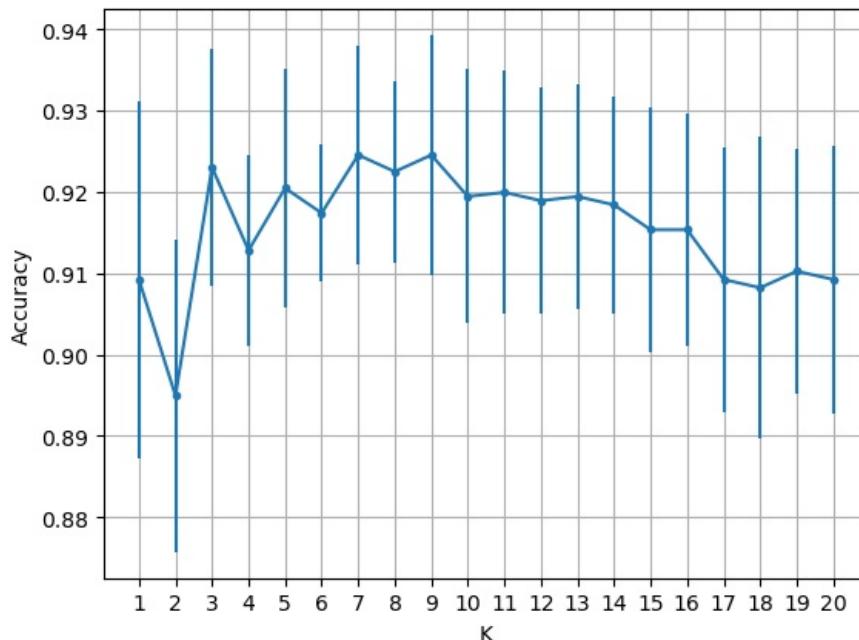
Utilizzando i parametri definiti sotto, la funzione cross_val_score organizza X_train_val in 10 sottoinsiemi. Per ogni iterazione di validazione incrociata, 9 di questi sottoinsiemi sono utilizzati per l'addestramento del modello, mentre quello rimanente serve come Test Set. In ogni iterazione, un diverso sottoinsieme è selezionato come test set, mentre i sottoinsiemi restanti costituiscono il training set. Questo processo garantisce che ogni sottoinsieme venga utilizzato almeno una volta come test set. Con k=7 per il nostro classificatore KNN, la funzione procede attraverso tutte le 10 iterazioni possibili, calcolando un valore di accuratezza per ciascuna.

```
In [146...]: from sklearn.model_selection import cross_val_score
knn = KNeighborsClassifier(n_neighbors=7)
scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
```

Realizzando un ciclo che varia il valore di K da 1 a 20 e utilizzando la funzione di validazione incrociata per ogni valore, possiamo plottare un grafico che mostra come l'accuratezza varia in funzione di K. Dai risultati ottenuti, K=9 emerge come il valore che massimizza l'accuratezza. Tuttavia, anche K=7 si conferma una scelta valida, mostrando elevate prestazioni.

```
In [149...]: acc_list_mean = list()
acc_list_std = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
    acc_list_mean.append(np.mean(scores))
    acc_list_std.append(np.std(scores))
```

```
In [150...]: plt.errorbar(x=np.arange(1, 20+1, 1),
                  y=acc_list_mean,
                  yerr=acc_list_std,
                  marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



Per minimizzare l'impatto della selezione casuale dei set di dati, intrinseco nella funzione cross_val_score, possiamo ripetere il processo di cross-validation più volte. In questo caso, sceglieremo di eseguire 5 ripetizioni facendo quindi 2 cicli for. Questo approccio aiuta a stabilizzare le stime di accuratezza. K=10 sembra offrire migliori prestazioni complessive, suggerendo che sia il valore ottimale per il nostro classificatore KNN in questo contesto.

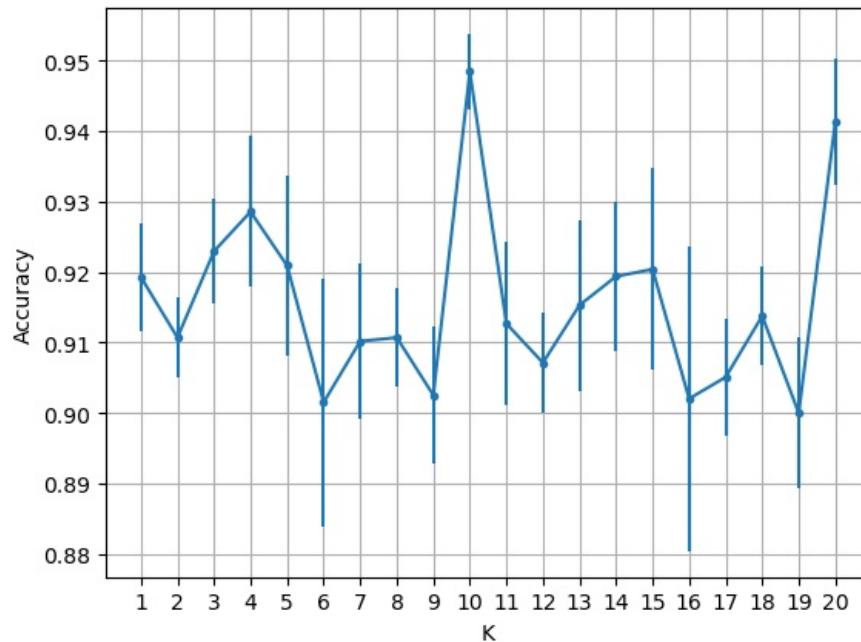
```
In [154...]: nbr_repetitions = 5
acc_list_all = list()
for i in range(nbr_repetitions):
    acc_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        scores = cross_val_score(knn, X_train_val, y_train_val, cv=10,
                                 scoring='accuracy')
        acc_list.append(scores)

    acc_list_all.append(acc_list)
```

```
In [155]: acc_list_all = np.array(acc_list_all)
acc_list_all.shape
acc_list_all.reshape(50, 20)
np.mean(acc_list_all.reshape(50, 20), axis=0)
```

```
Out[155]: array([0.91928934, 0.91071429, 0.92295918, 0.92857143, 0.92091837,
       0.90153061, 0.91020408, 0.91071429, 0.90255102, 0.94846939,
       0.91269036, 0.90714286, 0.91530612, 0.91938776, 0.92040816,
       0.90204082, 0.90510204, 0.91377551, 0.9           , 0.94132653])
```

```
In [156]: plt.errorbar(x=np.arange(1, 20+1, 1),
                    y=np.mean(acc_list_all.reshape(50, 20), axis=0),
                    yerr=np.std(acc_list_all.reshape(50, 20), axis=0),
                    marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



Stampiamo i report di classificazione per il Test set e il Training set con K=10. Il metodo predittivo ha un'accuratezza molto alta per entrambi i sessi ma con uno sbilanciamento nella precisione e nella recall. Questo sbilanciamento implica che il modello è preciso nel prevedere 0 (Sex=F) con un basso numero di FP (falsi positivi), ma con un numero più alto di FN (falsi negativi) suggerendo una tendenza a sottovalutare questa classe . Per 'M' (Sex=1), si verifica la situazione opposta: ci sono più FP rispetto a FN, indicando una propensione a sovrastimare questa classe.

```
In [158]: knn = KNeighborsClassifier(n_neighbors=10)
knn.fit(X_train_val, y_train_val)

y_pred = knn.predict(X_test)
print(classification_report(y_pred, y_test))

y_pred_trainval = knn.predict(X_train_val)
print(classification_report(y_pred_trainval, y_train_val))
```

	precision	recall	f1-score	support
0.0	0.94	0.89	0.91	256
1.0	0.88	0.94	0.91	235
accuracy			0.91	491
macro avg	0.91	0.91	0.91	491
weighted avg	0.91	0.91	0.91	491
	precision	recall	f1-score	support
0.0	0.97	0.91	0.94	1028
1.0	0.91	0.97	0.94	933
accuracy			0.94	1961
macro avg	0.94	0.94	0.94	1961
weighted avg	0.94	0.94	0.94	1961

Plottiamo adesso la **ROC** che è uno strumento grafico usato per valutare le prestazioni di un modello predittivo su una variabile target

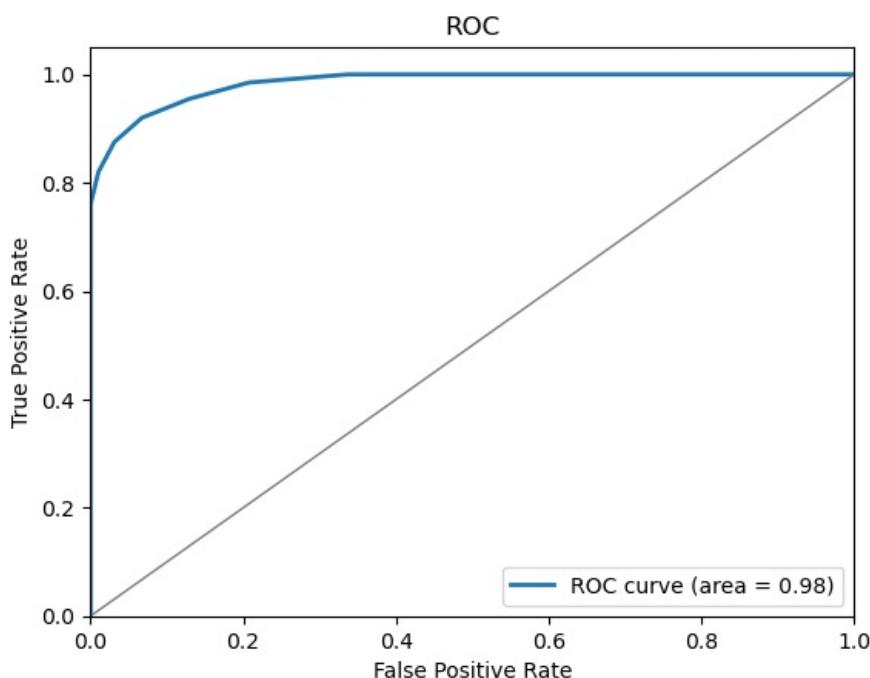
binaria. La curva traccia il tasso di TP rispetto a FP. Calcoliamo l'area sotto alla curva che risulta essere 0.98 che indica un modello altamente performante. Questo valore è consistente con l'alta accuratezza calcolata in precedenza e implica che il modello predittivo è efficiente sulla variabile target Sex.

```
In [166]: from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

y_pred = knn.predict_proba(X_val)[:, 1]

fpr, tpr, thresholds = roc_curve(y_val, y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='gray', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend(loc="lower right")
plt.show()
```



Per comprendere meglio come la dimensione del Training Set influisca sull'accuratezza del nostro modello, possiamo utilizzare quella che è nota come **Learning Curve**. Questa curva ci permette di visualizzare la relazione tra l'accuratezza del modello e la quantità di dati utilizzati per l'addestramento, il che è particolarmente rilevante quando si lavora con dataset di grandi dimensioni. Una riduzione mirata della dimensione del Training Set può, infatti, aumentare l'efficienza computazionale del modello senza sacrificare significativamente le prestazioni. Nel caso del nostro dataset, sembra che utilizzare l'80% dei dati come Training Set offra l'accuratezza migliore, anche se la variazione è minima tra il 60% e l'80%. Questo approccio potrebbe essere applicato anche focalizzandoci su altri indicatori di performance, come la precisione o la recall.

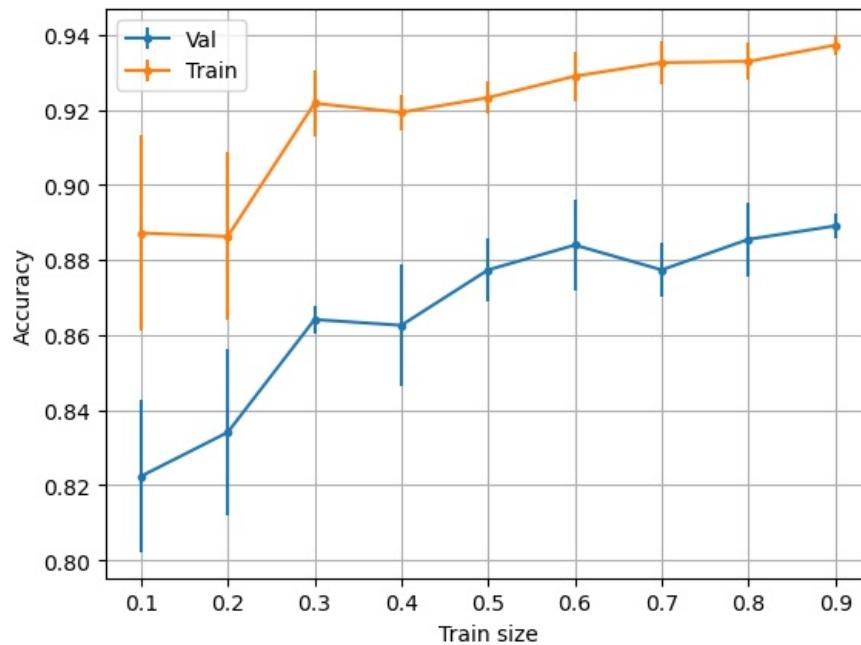
```
In [169]: nbr_repetitions = 5
acc_val_list_all = list()
acc_train_list_all = list()
for p in np.arange(0.1, 1.0, 0.1):
    acc_val_list = list()
    acc_train_list = list()
    for i in range(nbr_repetitions):
        index = np.random.choice(np.arange(len(X_train)), int(len(X_train) * p), replace=False)
        knn = KNeighborsClassifier(n_neighbors=10)
        knn.fit(X_train[index], y_train[index])
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train[index])
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train[index], y_pred_train))

    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)
```

```
In [171]: acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

In [173]

```
plt.errorbar(x=np.arange(0.1, 1.0, 0.1),
             y=np.mean(acc_val_list_all, axis=1),
             yerr=np.std(acc_val_list_all, axis=1),
             label='Val', marker='.')
plt.errorbar(x=np.arange(0.1, 1.0, 0.1),
             y=np.mean(acc_train_list_all, axis=1),
             yerr=np.std(acc_train_list_all, axis=1),
             label='Train', marker='.')
plt.xlabel('Train size')
plt.ylabel('Accuracy')
plt.xticks(np.arange(0.1, 1.0, 0.1))
plt.grid()
plt.legend()
plt.show()
```



Emotion

Decision Tree on Emotion

Per la classificazione della classe Emotion tramite Decision Tree, ripeteremo i passaggi già utilizzati per la classificazione della classe Sex. In questo scenario, tuttavia, per la preparazione dei dati, possiamo evitare l'one-hot encoding per la variabile target 'emotion', optando invece per convertirla direttamente in etichette numeriche da 0 a 7. Questa scelta si giustifica dal fatto che 'emotion' sarà la nostra variabile target in un contesto di classificazione multiclass, e l'uso di etichette numeriche semplifica il processo senza influire negativamente sull'apprendimento del modello. Le etichette saranno assegnate come segue: **0=angry, 1=calm, 2=disgust, 3=fearful, 4=happy, 5=neutral, 6=sad, 7=surprised.**

In [414]

```
dfe=pd.read_csv('A Data Understanding & Preparation Output.csv', sep=',', skipinitialspace=True)
dfe.head()
```

Out[414]

	emotion	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	zero_crossings_sum	mfcc_mean	mfcc_st
0	fearful	0.0	0.0	0.0	1.0	0.0	3737.0	16995.0	-33.485947	134.65486
1	angry	0.0	0.0	0.0	0.0	0.0	3904.0	13906.0	-29.502108	130.48563
2	happy	0.0	1.0	0.0	1.0	0.0	4671.0	18723.0	-30.532463	126.57711
3	surprised	0.0	0.0	1.0	0.0	0.0	3637.0	11617.0	-36.059555	159.72516
4	happy	1.0	1.0	0.0	1.0	0.0	4404.0	15137.0	-31.405996	122.12582

In [417]

```
emotion = sorted(dfe['emotion'].unique())
emotion_mapping = dict(zip(emotion, range(0, len(emotion) + 1)))
dfe['Emotion_val'] = dfe['emotion'].map(emotion_mapping).astype(int)
```

In [419]

```
dfe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2452 entries, 0 to 2451
Data columns (total 29 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   emotion          2452 non-null    object  
 1   vocal_channel    2452 non-null    float64 
 2   emotional_intensity 2452 non-null    float64 
 3   statement         2452 non-null    float64 
 4   repetition        2452 non-null    float64 
 5   sex               2452 non-null    float64 
 6   length_ms         2452 non-null    float64 
 7   zero_crossings_sum 2452 non-null    float64 
 8   mfcc_mean         2452 non-null    float64 
 9   mfcc_std          2452 non-null    float64 
 10  mfcc_min          2452 non-null    float64 
 11  mfcc_max          2452 non-null    float64 
 12  sc_mean           2452 non-null    float64 
 13  sc_std            2452 non-null    float64 
 14  sc_min            2452 non-null    float64 
 15  sc_max            2452 non-null    float64 
 16  sc_kur            2452 non-null    float64 
 17  sc_skew           2452 non-null    float64 
 18  stft_mean         2452 non-null    float64 
 19  stft_std          2452 non-null    float64 
 20  stft_min          2452 non-null    float64 
 21  stft_kur          2452 non-null    float64 
 22  stft_skew         2452 non-null    float64 
 23  std               2452 non-null    float64 
 24  min               2452 non-null    float64 
 25  max               2452 non-null    float64 
 26  kur               2452 non-null    float64 
 27  skew              2452 non-null    float64 
 28  Emotion_val       2452 non-null    int64  
dtypes: float64(27), int64(1), object(1)
memory usage: 555.7+ KB
```

```
In [421]: dfe.dtypes[dfe.dtypes.map(lambda x: x == 'object')]
```

```
Out[421]: emotion    object
          dtype: object
```

```
In [423]: dfe_noobj = dfe.drop(['emotion'], axis=1)
dfe_noobj.dtypes
```

```
vocal_channel      float64
emotional_intensity float64
statement          float64
repetition         float64
sex                float64
length_ms          float64
zero_crossings_sum float64
mfcc_mean          float64
mfcc_std           float64
mfcc_min           float64
mfcc_max           float64
sc_mean            float64
sc_std             float64
sc_min             float64
sc_max             float64
sc_kur             float64
sc_skew            float64
stft_mean          float64
stft_std           float64
stft_min           float64
stft_kur           float64
stft_skew          float64
std                float64
min                float64
max                float64
kur                float64
skew               float64
Emotion_val        int64  
dtype: object
```

```
In [426]: dfe_noobj
```

Out[426...]

	vocal_channel	emotional_intensity	statement	repetition	sex	length_ms	zero_crossings_sum	mfcc_mean	mfcc_std	mf...
0	0.0	0.0	0.0	1.0	0.0	3737.0	16995.0	-33.485947	134.654860	-755
1	0.0	0.0	0.0	0.0	0.0	3904.0	13906.0	-29.502108	130.485630	-713
2	0.0	1.0	0.0	1.0	0.0	4671.0	18723.0	-30.532463	126.577110	-726
3	0.0	0.0	1.0	0.0	0.0	3637.0	11617.0	-36.059555	159.725160	-842
4	1.0	1.0	0.0	1.0	0.0	4404.0	15137.0	-31.405996	122.125824	-700
...
2447	0.0	1.0	1.0	0.0	1.0	4605.0	9871.0	-30.225578	158.845500	-855
2448	0.0	0.0	0.0	0.0	1.0	4171.0	8963.0	-31.160332	157.499700	-825
2449	1.0	1.0	0.0	1.0	1.0	5239.0	9765.0	-26.135280	138.133210	-768
2450	0.0	0.0	1.0	0.0	1.0	3737.0	9716.0	-28.242815	159.943400	-868
2451	1.0	0.0	0.0	1.0	1.0	3837.0	9427.0	-29.019236	149.188950	-799

2452 rows × 28 columns

Scegliamo come variabile target l'attributo "Emotion_val" e prendiamo tutte le altre per addestrare il Decision Tree.

```
In [429...]: target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in target]
```

```
In [431...]: columns
```

```
Out[431...]: ['vocal_channel',
 'emotional_intensity',
 'statement',
 'repetition',
 'sex',
 'length_ms',
 'zero_crossings_sum',
 'mfcc_mean',
 'mfcc_std',
 'mfcc_min',
 'mfcc_max',
 'sc_mean',
 'sc_std',
 'sc_min',
 'sc_max',
 'sc_kur',
 'sc_skew',
 'stft_mean',
 'stft_std',
 'stft_min',
 'stft_kur',
 'stft_skew',
 'std',
 'min',
 'max',
 'kur',
 'skew']
```

```
In [433...]: X = dfe_noobj[columns].values
y = dfe_noobj[target].values
```

```
In [435...]: from sklearn.model_selection import train_test_split
```

```
In [437...]: X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
```

```
In [439...]: X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val)
```

```
In [441...]: from sklearn.tree import DecisionTreeClassifier
```

```
In [443...]: from sklearn.metrics import accuracy_score, confusion_matrix
```

```
In [445...]: clf = DecisionTreeClassifier(
    criterion='gini',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    ccp_alpha=0.0,
    random_state=0
)
clf.fit(X_train, y_train)
```

```
y_pred = clf.predict(X_val)
accuracy_score(y_val, y_pred)
```

```
Out[445]: 0.4071246819338422
```

```
In [447]: confusion_matrix(y_val, y_pred)
```

```
Out[447]: array([[36,  0,  3, 10,  2,  2,  5,  2],
   [ 2, 28,  5,  3,  6,  1, 14,  1],
   [ 2,  5,  9,  4,  0,  3,  6,  2],
   [ 9,  4,  1, 20, 11,  4,  2,  9],
   [10,  4,  3,  6, 19,  2, 12,  4],
   [ 1,  5,  0,  5,  3, 10,  6,  0],
   [ 0, 11, 11,  2,  3,  6, 27,  1],
   [ 0,  6,  2,  4,  3,  2,  3, 11]])
```

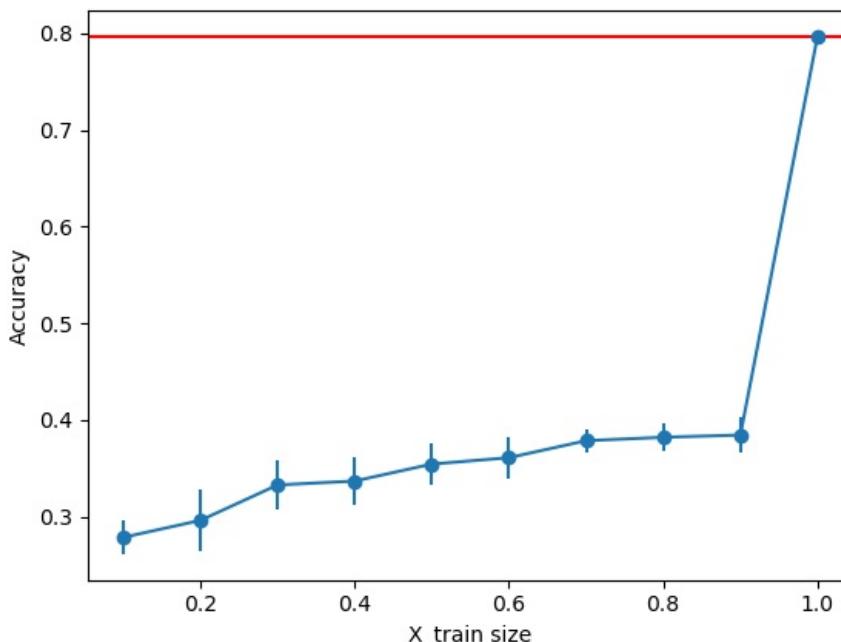
Per valutare l'impatto della dimensione del Test Set sull'accuratezza del nostro modello, abbiamo plottato l'accuratezza in funzione della percentuale del dataset assegnata al Test Set. Dai risultati emerge che assegnare l'80% dell'intero dataset al Training Set offre le prestazioni migliori, caratterizzate da un'accuratezza elevata e da una minore variazione della stessa.

```
In [211]: accuracy_mean_list = list()
accuracy_std_list = list()
for p in np.arange(0.1, 1.0, 0.1):
    accuracy_list_p = list()
    for i in range(0, 10):
        index = np.random.choice(np.arange(0, len(X_train)), int(len(X_train) * p), replace=False)
        clf = DecisionTreeClassifier(
            criterion='gini',
            max_depth=None,
            min_samples_split=2,
            min_samples_leaf=1,
            ccp_alpha=0.0,
            random_state=0
        )
        clf.fit(X_train[index], y_train[index])

        y_pred = clf.predict(X_val)
        accuracy_list_p.append(accuracy_score(y_val, y_pred))
    accuracy_mean_list.append(np.mean(accuracy_list_p))
    accuracy_std_list.append(np.std(accuracy_list_p))
```

```
In [212]: accuracy_mean_list.append(0.7967914438502673)
accuracy_std_list.append(0.0)
```

```
plt.errorbar(x=np.arange(0.1, 1.1, 0.1), y=accuracy_mean_list,
             yerr=accuracy_std_list, marker='o')
plt.axhline(y=0.7967914438502673, color='r')
plt.ylabel('Accuracy')
plt.xlabel('X_train size')
plt.show()
```



Come per l'analisi effettuata per la classificazione del sesso tramite Decision Tree, abbiamo proceduto a plottare l'effetto del parametro `min_samples_leaf` sull'accuratezza del modello. Questo parametro determina il numero minimo di campioni che devono essere presenti in una foglia. Dai nostri risultati, emerge che impostare `min_samples_leaf` a 12 ottimizza l'accuratezza del modello. Generalmente però

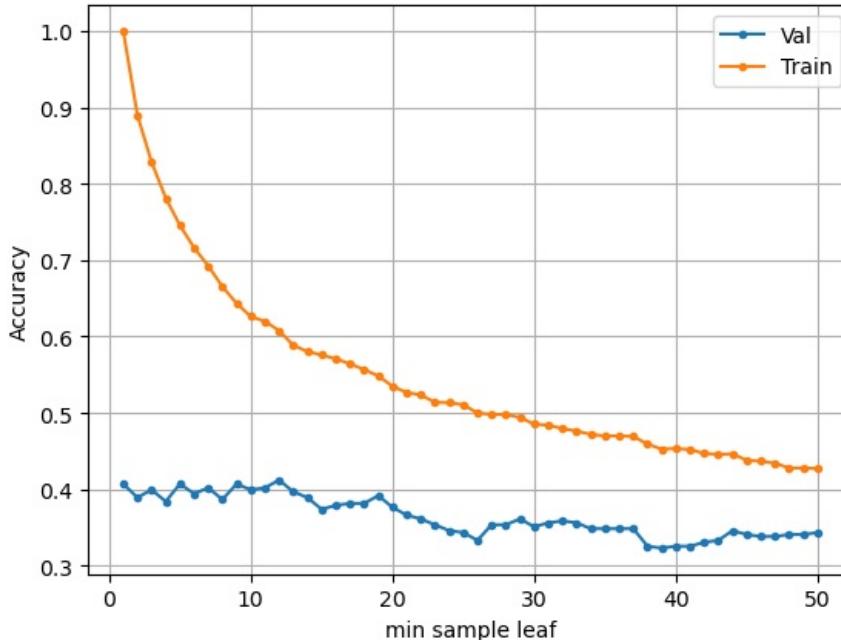
bisogna sottolineare che un numero più alto di questo valore produce un metodo più robusto.

```
In [451]: accuracy_val_list = list()
accuracy_train_list = list()
for min_sample_leaf in range(1,51):
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=None,
        min_samples_split=2,
        min_samples_leaf=min_sample_leaf,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

```
In [453]: plt.plot(min_sample_leaf_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_leaf_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample leaf')
plt.grid('white')
plt.legend()
plt.show()
```



Abbiamo condotto un'ulteriore analisi per determinare come il parametro **min_samples_split** (il numero minimo di campioni che un nodo deve avere per essere considerato per la divisione) influisca sull'accuratezza del nostro modello. Dalla visualizzazione, possiamo osservare che i valori ≤ 24 offrono le migliori prestazioni in termini di accuratezza.

```
In [456]: min_sample_split_list = np.arange(2, 50+1, 1)
min_sample_split_list

accuracy_val_list = list()
accuracy_train_list = list()
for min_sample_split in min_sample_split_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=None,
        min_samples_split=min_sample_split,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

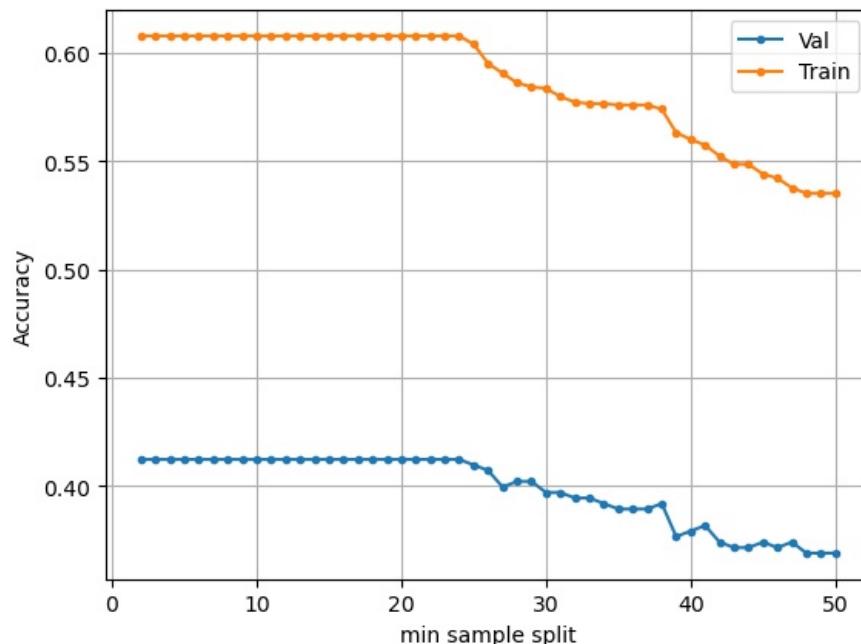
    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))
```

```

plt.plot(min_sample_split_list, accuracy_val_list, label='Val', marker='.')
plt.plot(min_sample_split_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('min sample split')
plt.grid('white')
plt.legend()
plt.show()

```



Abbiamo ripetuto l'analisi precedente per valutare come la massima profondità dell'albero (**max_depth**) influisca sull'accuratezza del nostro modello di classificazione. Plotando l'accuratezza del modello in funzione di diverse profondità massime, abbiamo identificato che il valore di **max_depth** pari a 10 offre il miglior risultato in termini di accuratezza. Questo suggerisce che un albero con una profondità massima di 10 è sufficientemente complesso da catturare le relazioni significative nei dati senza andare in overfitting.

In [462]...

```

max_depth_list = np.arange(1, 20+1, 1).tolist() + [None]

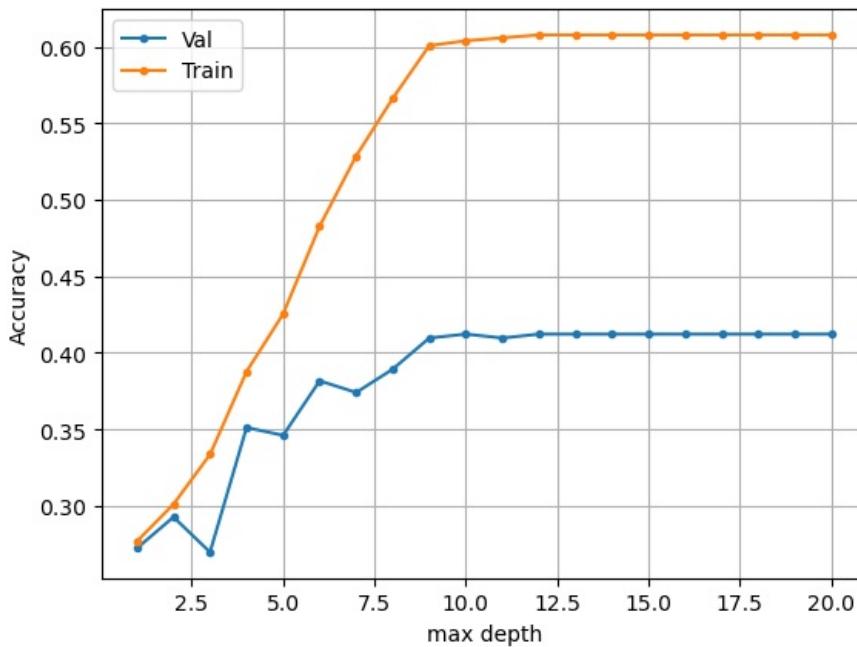
accuracy_val_list = list()
accuracy_train_list = list()
for max_depth in max_depth_list:
    clf = DecisionTreeClassifier(
        criterion='gini',
        max_depth=max_depth,
        min_samples_split=24,
        min_samples_leaf=12,
        ccp_alpha=0.0,
        random_state=0
    )
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_val)
    accuracy_val_list.append(accuracy_score(y_val, y_pred))

    y_pred_train = clf.predict(X_train)
    accuracy_train_list.append(accuracy_score(y_train, y_pred_train))

plt.plot(max_depth_list, accuracy_val_list, label='Val', marker='.')
plt.plot(max_depth_list, accuracy_train_list, label='Train', marker='.')
plt.ylabel('Accuracy')
plt.xlabel('max depth')
plt.grid('white')
plt.legend()
plt.show()

```



Procederemo ora all'utilizzo di **RandomizedSearchCV**, una tecnica di ottimizzazione automatica fornita da scikit-learn, per identificare la migliore combinazione di parametri per il nostro modello Decision Tree. Diversamente dalla ricerca esaustiva su griglia (GridSearchCV), RandomizedSearchCV riduce il tempo computazionale, ma permette comunque di esplorare un ampio spazio dei parametri. Questo approccio è particolarmente utile per ottimizzare modelli complessi su grandi set di dati, poiché consente di trovare una buona combinazione di parametri in modo più efficiente. Imposteremo un range di valori per ciascun parametro di interesse e lasceremo che RandomizedSearchCV determini la combinazione ottimale attraverso validazione incrociata.

```
In [227]: from sklearn.model_selection import RandomizedSearchCV
```

```
In [229]: clf = DecisionTreeClassifier(
    criterion='gini',
    max_depth=10,
    min_samples_split=10,
    min_samples_leaf=11,
    ccp_alpha=0.0,
    random_state=0)
```

```
In [232]: param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0, 0.1)}
```

```
In [234]: random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_
```

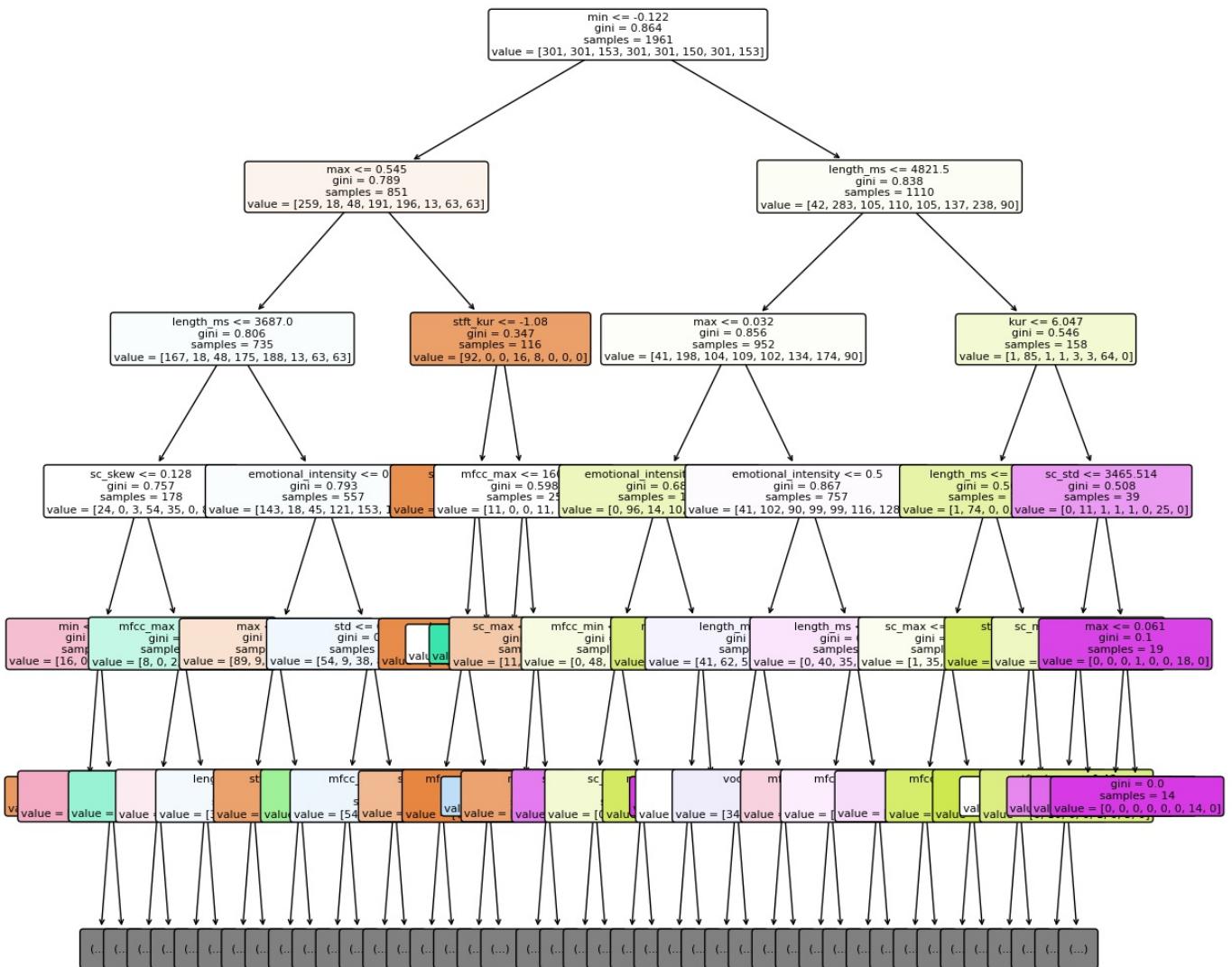
```
Out[234]: {'min_samples_split': 6,
 'min_samples_leaf': 5,
 'max_depth': 16,
 'ccp_alpha': 0.0}
```

Possiamo vedere che i valori dei parametri individuati tramite **RandomizedSearchCV** non corrispondono a quelli precedentemente identificati variando un parametro alla volta. Questa differenza sottolinea l'efficacia di RandomizedSearchCV nell'esplorare le varie combinazioni di parametri in modo più ampio e sistematico, considerando le interazioni tra più parametri contemporaneamente. Di conseguenza, abbiamo deciso di adottare i parametri suggeriti da RandomizedSearchCV per il nostro modello Decision Tree.

```
In [236]: clf = random.best_estimator_
```

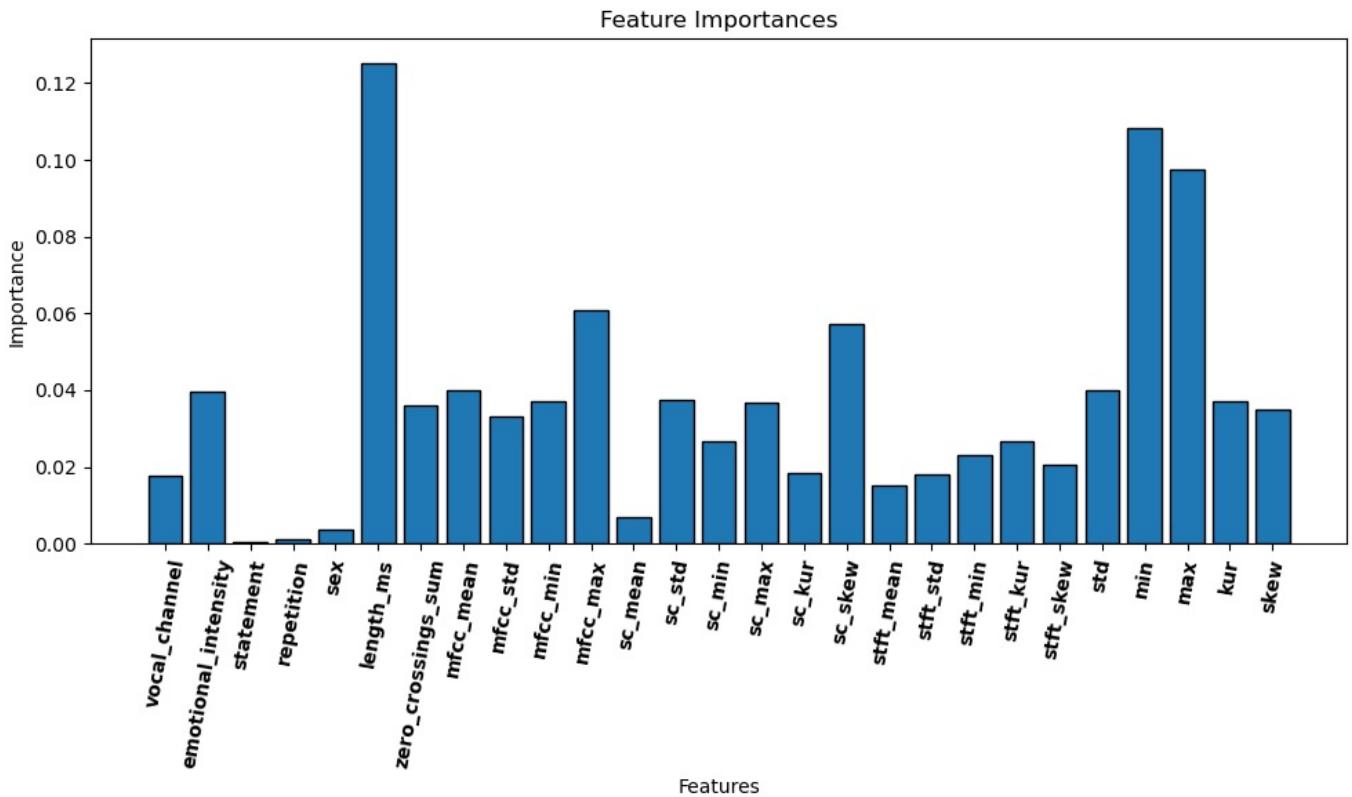
```
In [237]: from sklearn.tree import plot_tree, export_text
```

```
In [238]: plt.figure(figsize=(14, 14))
plot_tree(clf,
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=8,
          max_depth=5
         )
plt.show()
```



Utilizzando una barchart, abbiamo visualizzato **l'importanza delle diverse features** del nostro dataset nell'influenzare la previsione della variabile 'Emotion' con il modello di Decision Tree. L'analisi rivela che la feature 'length_ms' è la più influente, seguita da 'min' e 'max'. Queste informazioni sono state ottenute analizzando quanto ogni feature sia efficace nel separare le diverse classi di 'Emotion' nel dataset.

```
In [240]: plt.figure(figsize=(10, 6))
plt.bar(columns, clf.feature_importances_, edgecolor='k')
plt.xticks(rotation=80, fontweight='bold')
plt.title('Feature Importances')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.tight_layout()
plt.show()
```



Abbiamo valutato l'accuratezza del nostro modello di Decision Tree sul Test Set, ottenendo un risultato di 0.43. Oltre all'accuratezza, abbiamo calcolato anche Precision, Recall, F1-Score e generato la Confusion Matrix per avere una visione più completa delle prestazioni del modello. Per quanto riguarda il bilanciamento tra le classi predette correttamente e quelle erroneamente interpretate, è interessante notare che il modello preveda con maggiore accuratezza le emozioni 'angry' (0) e 'calm' (1), in linea con quanto osservato nella sezione di data understanding, dove queste emozioni risultavano essere ben separate dalle altre. Al contrario, le emozioni 'surprised' (7), 'fearful' (3), 'disgust' (2) e 'sad' (6) sono quelle che il modello trova più difficili da prevedere.

```
In [242]: y_pred = clf.predict(X_test)
y_pred_trainval=clf.predict(X_train_val)

accuracy_score(y_test, y_pred)
```

```
Out[242]: 0.42769857433808556
```

```
In [243]: from sklearn.metrics import precision_score, recall_score, f1_score
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)

print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("Confusion Matrix:", conf_matrix)
print(classification_report(y_pred, y_test))

print(classification_report(y_pred_trainval, y_train_val))
```

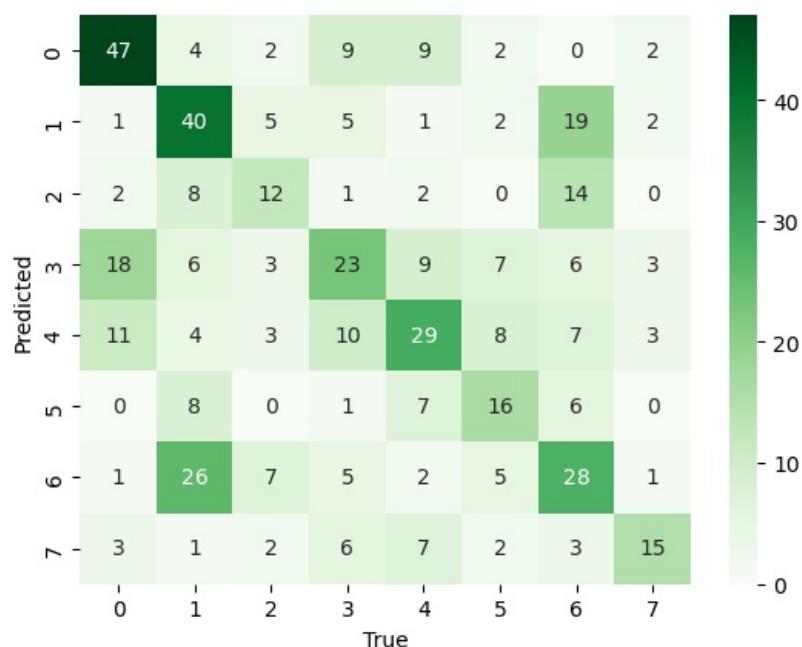
```
Precision: 0.43002924888387906
Recall: 0.42769857433808556
F1 Score: 0.4246737062512508
Confusion Matrix: [[47  4  2  9  9  2  0  2]
 [ 1 40  5  5  1  2 19  2]
 [ 2  8 12  1  2  0 14  0]
 [18  6  3 23  9  7  6  3]
 [11  4  3 10 29  8  7  3]
 [ 0  8  0  1  7 16  6  0]
 [ 1 26  7  5  2  5 28  1]
 [ 3  1  2  6  7  2  3 15]]
```

	precision	recall	f1-score	support
0	0.63	0.57	0.59	83
1	0.53	0.41	0.47	97
2	0.31	0.35	0.33	34
3	0.31	0.38	0.34	60
4	0.39	0.44	0.41	66
5	0.42	0.38	0.40	42
6	0.37	0.34	0.35	83
7	0.38	0.58	0.46	26
accuracy			0.43	491
macro avg	0.42	0.43	0.42	491
weighted avg	0.44	0.43	0.43	491
	precision	recall	f1-score	support
0	0.84	0.79	0.82	317
1	0.86	0.75	0.80	347
2	0.69	0.70	0.69	150
3	0.71	0.77	0.74	279
4	0.72	0.74	0.73	293
5	0.70	0.75	0.72	140
6	0.72	0.74	0.73	292
7	0.70	0.75	0.72	143
accuracy			0.75	1961
macro avg	0.74	0.75	0.74	1961
weighted avg	0.76	0.75	0.75	1961

Visualizziamo anche la Confusion Matrix e plottiamo la ROC curve per questo modello predittivo

In [245...]

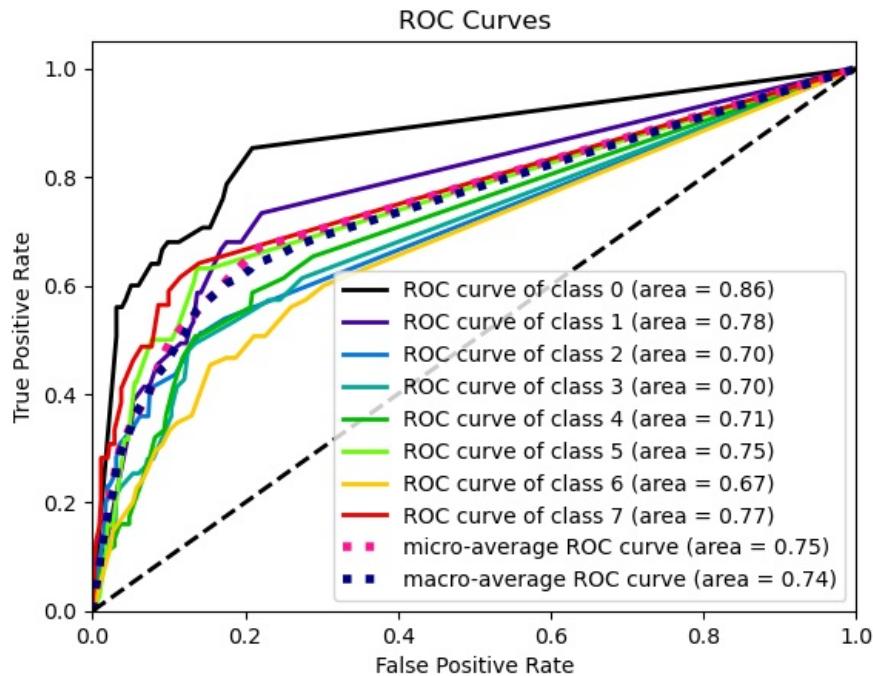
```
import seaborn as sns
cf = confusion_matrix(y_test, y_pred)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



In [247...]

```
import scikitplot
y_pred_proba = clf.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

Out[247...]: <Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>



KNN on Emotion

Per l'analisi delle 'Emotion' con il modello KNN, applicheremo la stessa procedura e gli stessi passi precedentemente discussi e analizzati nella classificazione della variabile Sex.

```
In [473]: from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in target]

X = dfe_noobj[columns].values
Y = dfe_noobj[target].values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, Y, test_size=0.3, stratify=Y, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.3, stratify=y_train_val)
print("Training set", X_train.shape, y_train.shape)
print("Validation set", X_val.shape, y_val.shape)
print("Test set", X_test.shape, y_test.shape)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_val = knn.predict(X_val)
y_pred_train = knn.predict(X_train)
y_pred = knn.predict(X_test)

print("Test set:\n", classification_report(y_pred, y_test))
print(accuracy_score(y_val, y_pred_val), accuracy_score(y_train, y_pred_train))
print(classification_report(y_val, y_pred_val))
```

```
Training set (1201, 27) (1201,)
```

```
Validation set (515, 27) (515,)
```

```
Test set (736, 27) (736,)
```

```
Test set:
```

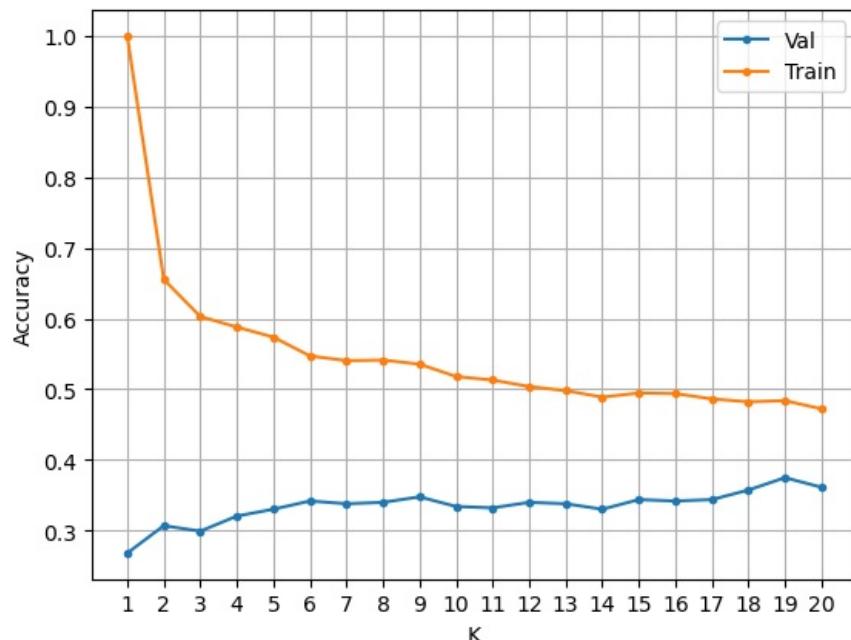
	precision	recall	f1-score	support
0	0.58	0.40	0.47	167
1	0.54	0.39	0.45	157
2	0.22	0.25	0.24	52
3	0.27	0.33	0.30	93
4	0.25	0.26	0.25	108
5	0.25	0.24	0.25	58
6	0.17	0.35	0.23	54
7	0.33	0.40	0.37	47
accuracy			0.34	736
macro avg	0.33	0.33	0.32	736
weighted avg	0.39	0.34	0.35	736

```
0.3300970873786408 0.5736885928393006
```

	precision	recall	f1-score	support
0	0.41	0.61	0.49	79
1	0.41	0.48	0.44	79
2	0.34	0.28	0.31	40
3	0.25	0.23	0.24	79
4	0.24	0.27	0.25	79
5	0.34	0.25	0.29	40
6	0.31	0.22	0.26	79
7	0.24	0.17	0.20	40
accuracy			0.33	515
macro avg	0.32	0.31	0.31	515
weighted avg	0.32	0.33	0.32	515

```
In [475]: acc_val_list = list()
acc_train_list = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_val)
    y_pred_train = knn.predict(X_train)
    acc_val_list.append(accuracy_score(y_val, y_pred))
    acc_train_list.append(accuracy_score(y_train, y_pred_train))
```

```
In [477]: plt.plot(np.arange(1, 20+1, 1), acc_val_list, label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), acc_train_list, label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



```
In [479]: nbr_holdout = 5
acc_val_list_all = list()
```

```

acc_train_list_all = list()
for i in range(nbr_holdout):

    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                    test_size=0.3,
                                                    stratify=y_train_val,
                                                    random_state=i)

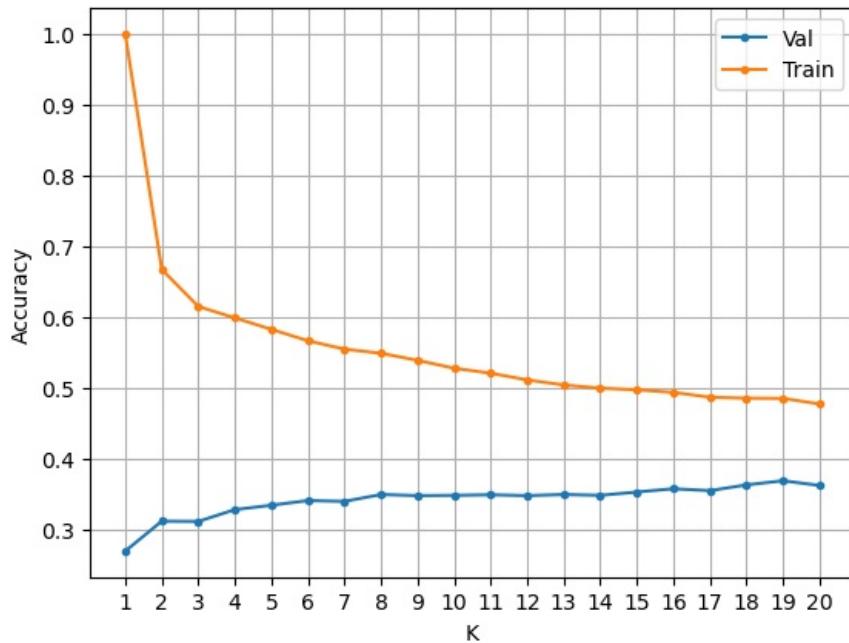
    acc_val_list = list()
    acc_train_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_val)
        y_pred_train = knn.predict(X_train)
        acc_val_list.append(accuracy_score(y_val, y_pred))
        acc_train_list.append(accuracy_score(y_train, y_pred_train))

    acc_val_list_all.append(acc_val_list)
    acc_train_list_all.append(acc_train_list)

```

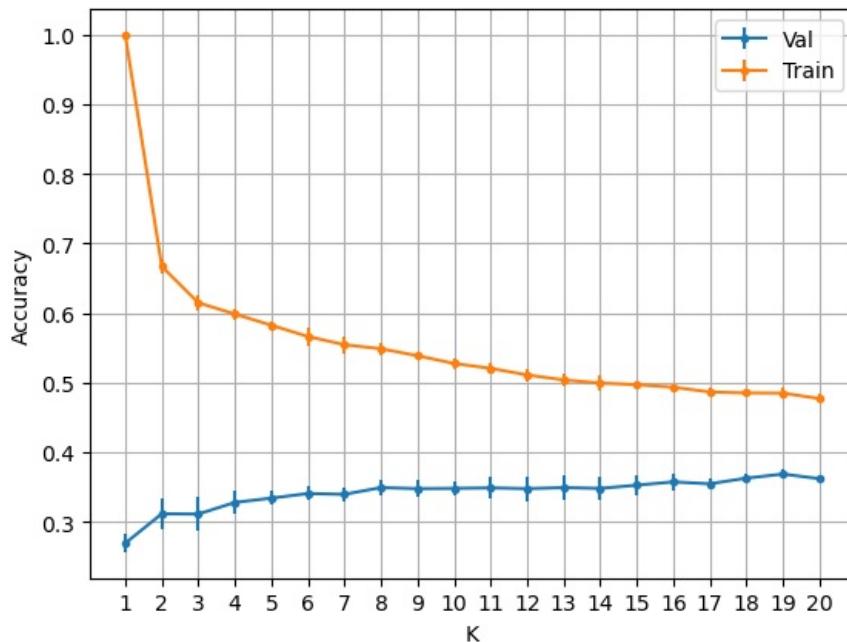
```
In [480]: acc_val_list_all = np.array(acc_val_list_all)
acc_train_list_all = np.array(acc_train_list_all)
```

```
In [483]: plt.plot(np.arange(1, 20+1, 1), np.mean(acc_val_list_all, axis=0), label='Val', marker='.')
plt.plot(np.arange(1, 20+1, 1), np.mean(acc_train_list_all, axis=0), label='Train', marker='.')
plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



```
In [485]: plt.errorbar(x=np.arange(1, 20+1, 1),
                    y=np.mean(acc_val_list_all, axis=0),
                    yerr=np.std(acc_val_list_all, axis=0),
                    label='Val', marker='.')
plt.errorbar(x=np.arange(1, 20+1, 1),
                    y=np.mean(acc_train_list_all, axis=0),
                    yerr=np.std(acc_train_list_all, axis=0),
                    label='Train', marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.legend()
plt.show()
```



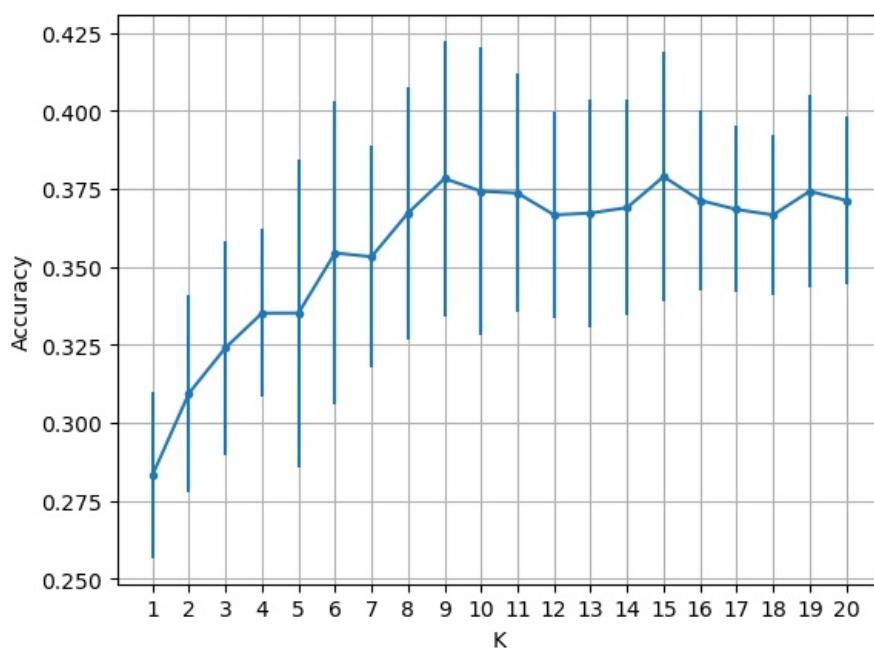
Cross Validation

```
In [488]: from sklearn.model_selection import cross_val_score
knn = KNeighborsClassifier(n_neighbors=5)
scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
```

```
In [490]: acc_list_mean = list()
acc_list_std = list()
for k in np.arange(1, 20+1, 1):
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train_val, y_train_val, cv=10)
    acc_list_mean.append(np.mean(scores))
    acc_list_std.append(np.std(scores))
```

```
In [491]: plt.errorbar(x=np.arange(1, 20+1, 1),
                  y=acc_list_mean,
                  yerr=acc_list_std,
                  marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



Dall'analisi del grafico plottato sotto, il numero migliore di K per il modello KNN su Emotion sembrano essere 8 e 18. Abbiamo deciso di scegliere K=18 per il nostro modello predittivo perché dal grafico sembra avere una variazione più bassa.

```
In [495...]
nbr_repetitions = 5
acc_list_all = list()
for i in range(nbr_repetitions):
    acc_list = list()
    for k in np.arange(1, 20+1, 1):
        knn = KNeighborsClassifier(n_neighbors=k)
        scores = cross_val_score(knn, X_train_val, y_train_val, cv=10, #k-fold
                                  scoring='accuracy')
        acc_list.append(scores)

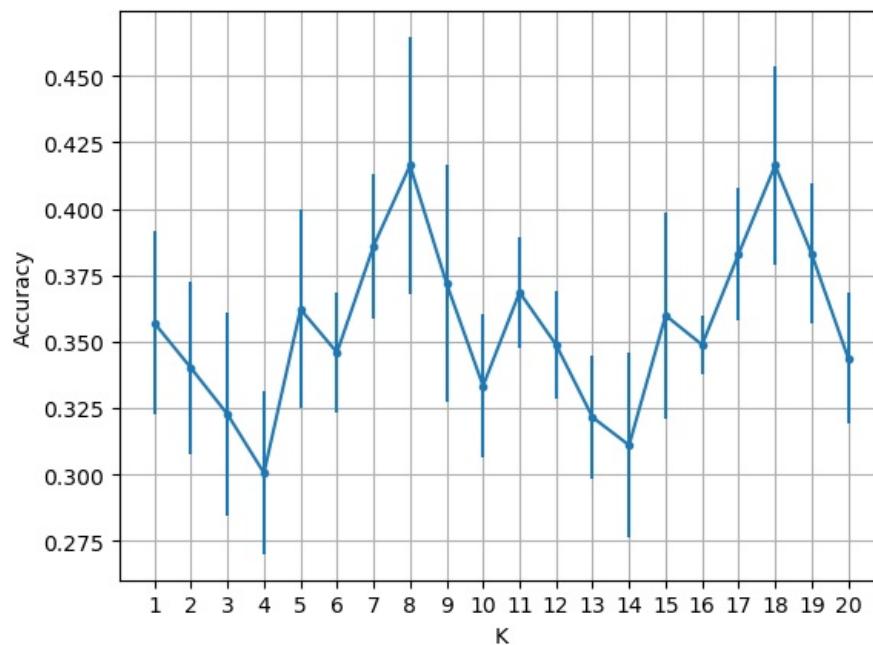
    acc_list_all.append(acc_list)
```

```
In [496...]
acc_list_all = np.array(acc_list_all)
acc_list_all.shape
acc_list_all.reshape(50, 20)
np.mean(acc_list_all.reshape(50, 20), axis=0)
```

```
Out[496...]
array([0.35697674, 0.34011628, 0.32267442, 0.3005814 , 0.3622093 ,
       0.34593023, 0.38596491, 0.41637427, 0.37192982, 0.33333333,
       0.36860465, 0.34883721, 0.32151163, 0.31104651, 0.35988372,
       0.34883721, 0.38304094, 0.41637427, 0.38304094, 0.34385965])
```

```
In [497...]
plt.errorbar(x=np.arange(1, 20+1, 1),
              y=np.mean(acc_list_all.reshape(50, 20), axis=0),
              yerr=np.std(acc_list_all.reshape(50, 20), axis=0),
              marker='.')

plt.xlabel('K')
plt.ylabel('Accuracy')
plt.xticks(np.arange(1, 20+1, 1))
plt.grid()
plt.show()
```



Dall'analisi del report generato, osserviamo che il modello KNN ha un'accuratezza simile ma leggermente più bassa rispetto al Decision Tree. Inoltre, è possibile notare come anche per il modello KNN, le emozioni 'angry' e 'calm' sono quelle che vengono previste meglio. Notiamo invece che le precision per le emozioni 'sad', 'happy' sono basse e addirittura peggiori che nel caso del Decision Tree. Questo riflette l'analisi fatta nella data preparation che indicava che queste emozioni condividono caratteristiche simili e ciò rende più complessa la loro distinzione da parte dei modelli utilizzati. Un'altra cosa molto interessante da evidenziare per questo modello è come la precision per Surprised (class=7) sia significativamente più alta rispetto al Decision Tree.

```
In [501...]
knn = KNeighborsClassifier(n_neighbors=18)
knn.fit(X_train_val, y_train_val)

y_pred = knn.predict(X_test)
print("Test set:\n", classification_report(y_pred, y_test))

y_pred_trainval = knn.predict(X_train_val)
print(classification_report(y_pred_trainval, y_train_val))
```

Test set:

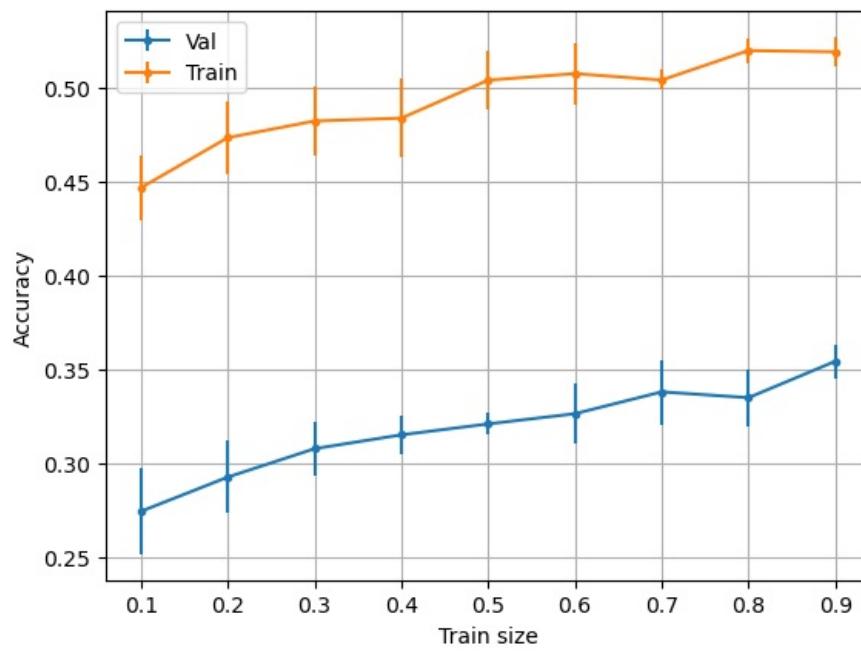
	precision	recall	f1-score	support
0	0.65	0.61	0.63	122
1	0.50	0.41	0.45	135
2	0.36	0.38	0.37	55
3	0.38	0.42	0.40	102
4	0.23	0.23	0.23	112
5	0.34	0.28	0.31	68
6	0.19	0.30	0.23	70
7	0.54	0.43	0.48	72
accuracy			0.40	736
macro avg	0.40	0.38	0.39	736
weighted avg	0.42	0.40	0.40	736
	precision	recall	f1-score	support
0	0.74	0.63	0.68	308
1	0.62	0.54	0.58	306
2	0.35	0.42	0.38	112
3	0.49	0.47	0.48	274
4	0.47	0.49	0.48	255
5	0.45	0.40	0.43	149
6	0.31	0.46	0.37	176
7	0.40	0.40	0.40	136
accuracy			0.50	1716
macro avg	0.48	0.48	0.47	1716
weighted avg	0.52	0.50	0.50	1716

Attraverso l'analisi delle **Learning Curves**, abbiamo valutato come l'accuratezza del nostro modello varia al cambiare della dimensione del Training Set. Dai risultati ottenuti con il DataSet in esame, emerge che destinare l'70% dei dati al Training Set offre le migliori prestazioni in termini di accuratezza. Questo ci fornisce una linea guida su quale proporzione di dati utilizzare per massimizzare l'efficacia del modello. Lo stesso tipo di analisi potrebbe essere fatta per calcolare la migliore dimensione del Training Set riguardo agli altri parametri di valutazione come la precision o la recall.

```
In [505...]  
nbr_repetitions = 5  
acc_val_list_all = list()  
acc_train_list_all = list()  
for p in np.arange(0.1, 1.0, 0.1):  
    acc_val_list = list()  
    acc_train_list = list()  
    for i in range(nbr_repetitions):  
        index = np.random.choice(np.arange(len(X_train)), int(len(X_train) * p), replace=False)  
        knn = KNeighborsClassifier(n_neighbors=10)  
        knn.fit(X_train[index], y_train[index])  
        y_pred = knn.predict(X_val)  
        y_pred_train = knn.predict(X_train[index])  
        acc_val_list.append(accuracy_score(y_val, y_pred))  
        acc_train_list.append(accuracy_score(y_train[index], y_pred_train))  
  
    acc_val_list_all.append(acc_val_list)  
    acc_train_list_all.append(acc_train_list)
```

```
In [507...]  
acc_val_list_all = np.array(acc_val_list_all)  
acc_train_list_all = np.array(acc_train_list_all)
```

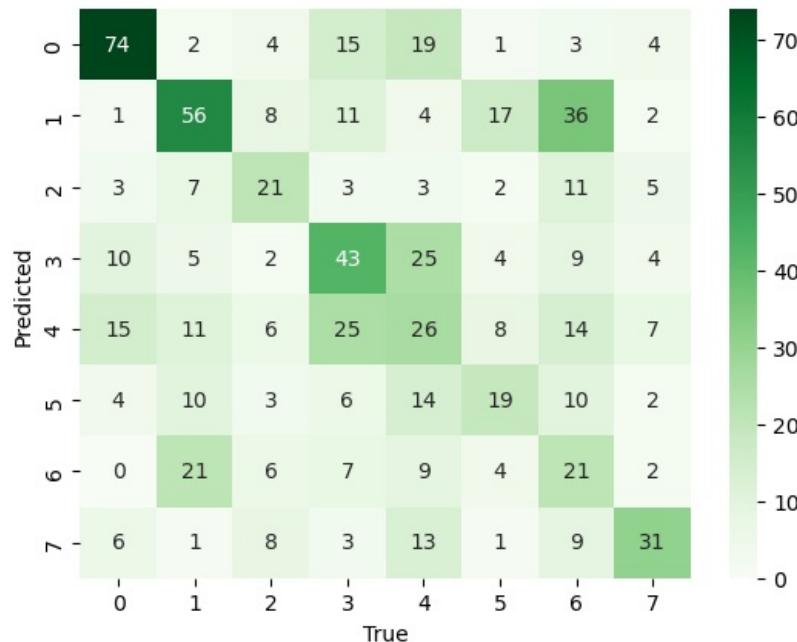
```
In [509...]  
plt.errorbar(x=np.arange(0.1, 1.0, 0.1),  
            y=np.mean(acc_val_list_all, axis=1),  
            yerr=np.std(acc_val_list_all, axis=1),  
            label='Val', marker='.')  
  
plt.errorbar(x=np.arange(0.1, 1.0, 0.1),  
            y=np.mean(acc_train_list_all, axis=1),  
            yerr=np.std(acc_train_list_all, axis=1),  
            label='Train', marker='.')  
  
plt.xlabel('Train size')  
plt.ylabel('Accuracy')  
plt.xticks(np.arange(0.1, 1.0, 0.1))  
plt.grid()  
plt.legend()  
plt.show()
```



Nelle celle sotto abbiamo disegnato la confusion matrix per questo modello e la ROC curve

In [303]:

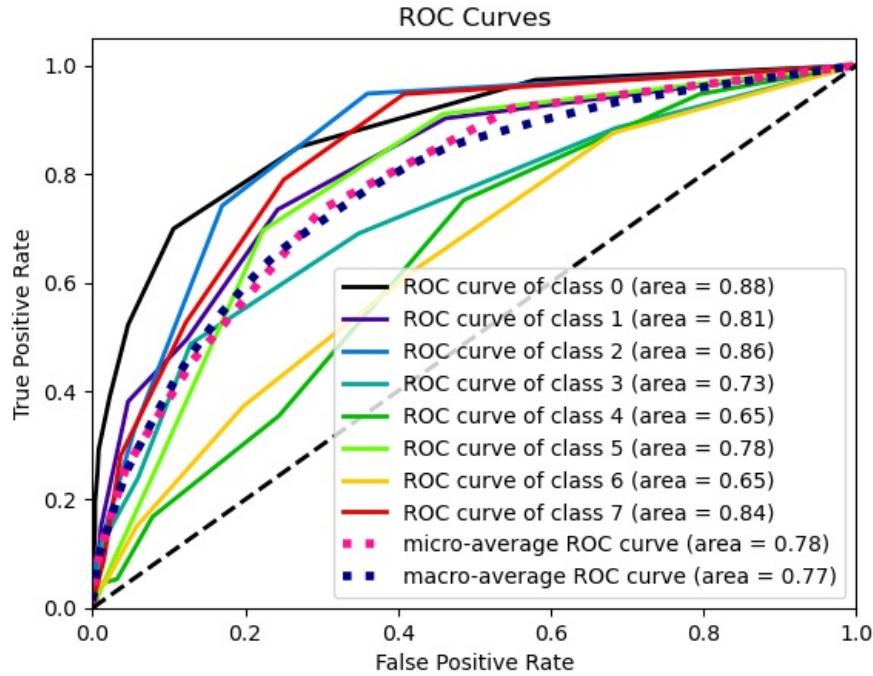
```
import seaborn as sns
cf = confusion_matrix(y_pred, y_test)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



In [299]:

```
import scikitplot
y_pred_proba = knn.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

Out[299]: <Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>



Random Forest Emotion

In questa sezione, abbiamo esplorato l'utilizzo dei **metodi ensemble**, che migliorano l'accuratezza delle predizioni aggregando i risultati di più classificatori, basandosi sul concetto di "**wisdom of the crowd**". Uno di questi metodi ensemble è il **RandomForest**, che combina le previsioni di numerosi Decision Tree. Per ogni attributo, il RandomForest restituisce la moda dei risultati ottenuti dai vari alberi di decisione e questo tende a migliorare l'accuratezza e la robustezza del modello riducendo il rischio di overfitting. Questo metodo è particolarmente efficace perché sfrutta la diversità tra gli alberi di decisione, ognuno dei quali è addestrato su un sottoinsieme casuale dei dati (**bootstrap sampling**) e caratteristiche, selezionate anche loro in maniera casuale. Applicheremo lo stesso approccio che abbiamo utilizzato con il Decision Tree per prevedere 'Emotion'.

```
In [305]: from sklearn.ensemble import RandomForestClassifier
target = 'Emotion_val'
columns = [c for c in dfe_noobj.columns if c not in target]

In [307]: X = dfe_noobj[columns].values
Y = dfe_noobj[target].values

In [309]: X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val)

In [311]: clf = RandomForestClassifier (criterion='gini',
                                 max_depth=11,
                                 min_samples_split=11,
                                 min_samples_leaf=3,
                                 ccp_alpha=0.0,
                                 random_state=0
                               )

param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
    'min_samples_leaf': np.arange(1, 50+1, 1),
    'ccp_alpha': np.arange(0.0, 1, 0.1)
}
```

```

random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_

best_params = random.best_params_

clf = random.best_estimator_

clf.fit(X_train, y_train)

```

Out[311]:

```

▼ RandomForestClassifier
RandomForestClassifier(max_depth=16, min_samples_leaf=5, min_samples_split=6,
                      random_state=0)

```

L'accuratezza calcolata per il modello RandomForest è **0.49** e quindi superiore rispetto ai modelli Decision Tree e KNN. Questo evidenzia la potenza dell'approccio ensemble, in particolare la capacità del RandomForest di aggregare le previsioni di molti alberi decisionali per ridurre il rischio di overfitting e gestire efficacemente la varianza. Possiamo notare che questo modello mostra un'alta accuratezza nella previsione delle emozioni 'angry' e 'calm' (similmente ai modelli precedenti ma con un aumento significativo), e dimostra anche un notevole miglioramento per le emozioni 'fearful' e 'surprised' rispetto al KNN e al Decision Tree. Questo sembra dimostrare la robustezza del RandomForest nel catturare le differenze di dati che caratterizzano queste emozioni specifiche. Al contrario, le emozioni 'disgust', 'sad' e 'neutral' registrano una precisione più bassa.

In [313]:

```

y_pred = clf.predict(X_test)
y_pred_trainval = clf.predict(X_train_val)

accuracy = accuracy_score(y_test, y_pred)
print(accuracy)

print(classification_report(y_pred, y_test))
print(classification_report(y_pred_trainval, y_train_val))

```

0.48676171079429736

	precision	recall	f1-score	support
0	0.80	0.66	0.72	91
1	0.83	0.50	0.62	124
2	0.26	0.53	0.34	19
3	0.32	0.56	0.41	43
4	0.36	0.39	0.37	70
5	0.29	0.23	0.26	48
6	0.27	0.40	0.32	50
7	0.64	0.54	0.59	46
accuracy			0.49	491
macro avg	0.47	0.48	0.45	491
weighted avg	0.56	0.49	0.51	491

	precision	recall	f1-score	support
0	0.93	0.87	0.90	319
1	0.92	0.76	0.83	365
2	0.75	0.89	0.82	129
3	0.81	0.86	0.83	284
4	0.82	0.80	0.81	310
5	0.80	0.84	0.82	143
6	0.76	0.87	0.81	262
7	0.79	0.81	0.80	149
accuracy			0.83	1961
macro avg	0.82	0.84	0.83	1961
weighted avg	0.84	0.83	0.83	1961

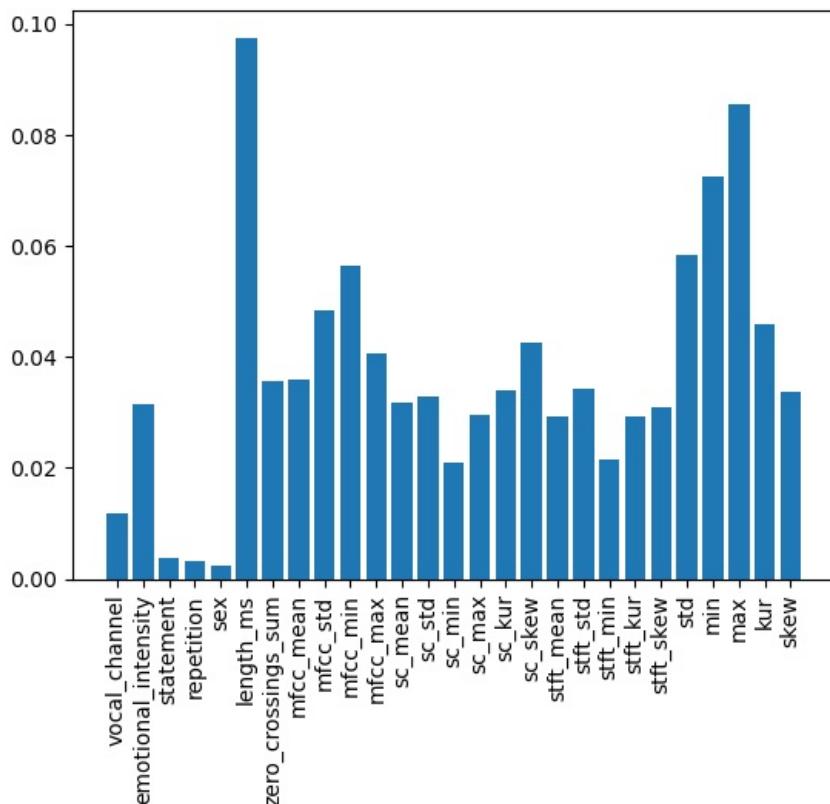
Dall'analisi della **Feature Importance** del Random Forest si può notare che le variabili *min,max,length_ms* sono le più importanti per prevedere le Emozioni.

In [315]:

```

plt.bar(columns, clf.feature_importances_)
plt.xticks(rotation=90)
plt.show()

```

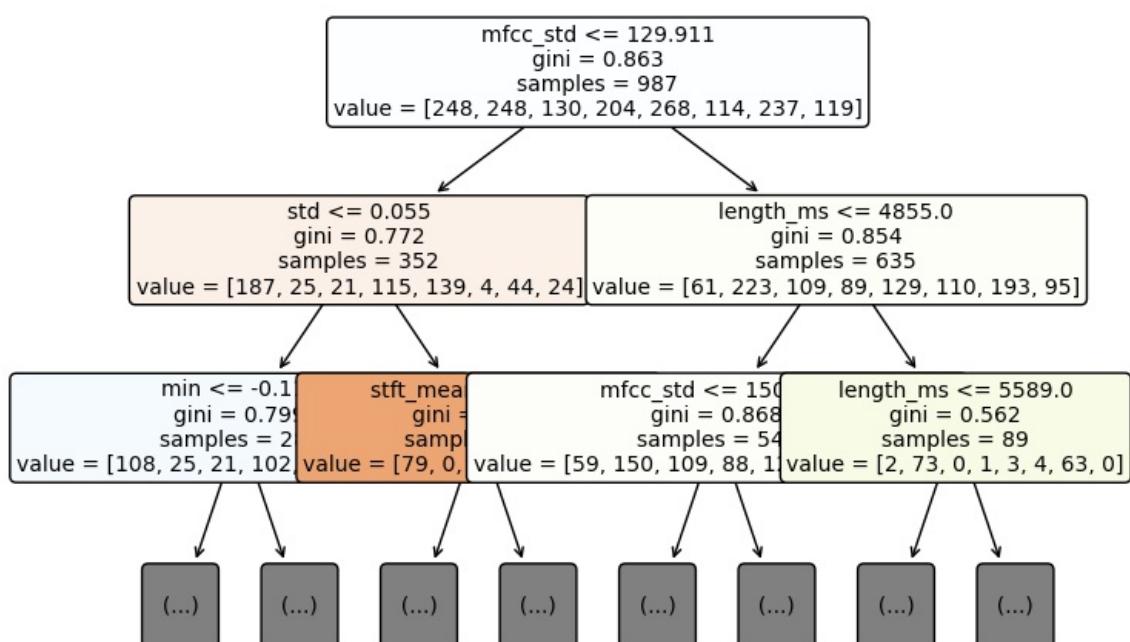


Vediamo adesso quanti sono i DecisionTrees usati per creare la RandomForest (100) e ne visualizziamo due (il primo e il secondo)

```
In [317]: len(clf.estimators_)
```

```
Out[317]: 100
```

```
In [318]: plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[0],
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=10,
          max_depth=2
         )
plt.show()
```

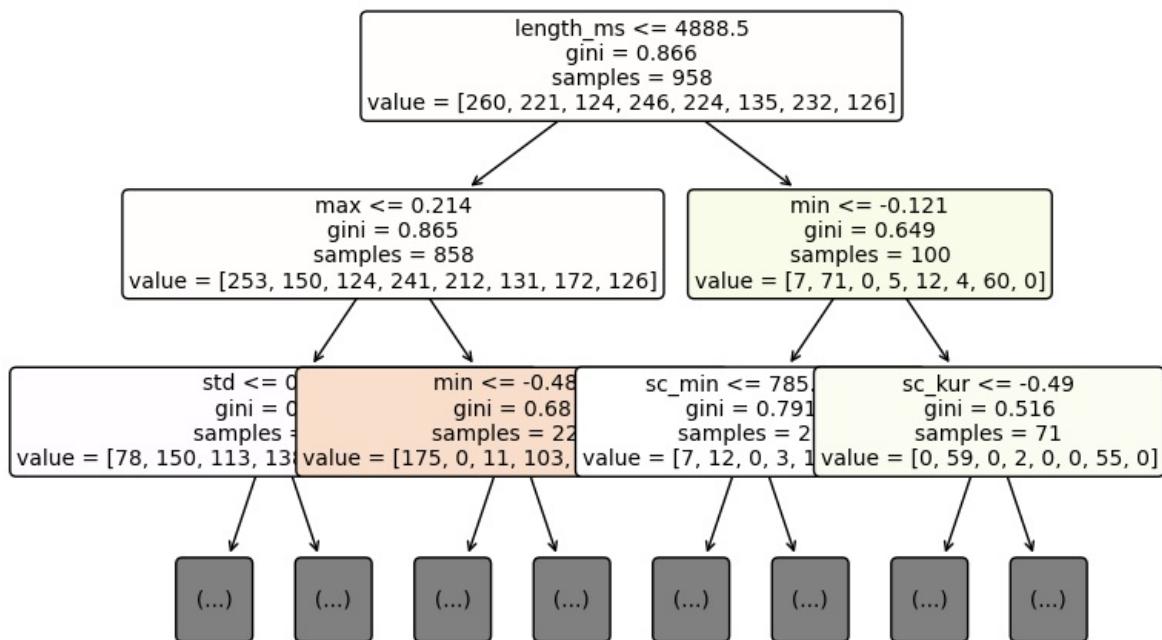


```
In [319]: plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[1],
          feature_names=columns,
          #class_names=clf.classes_,
          filled=True,
```

```

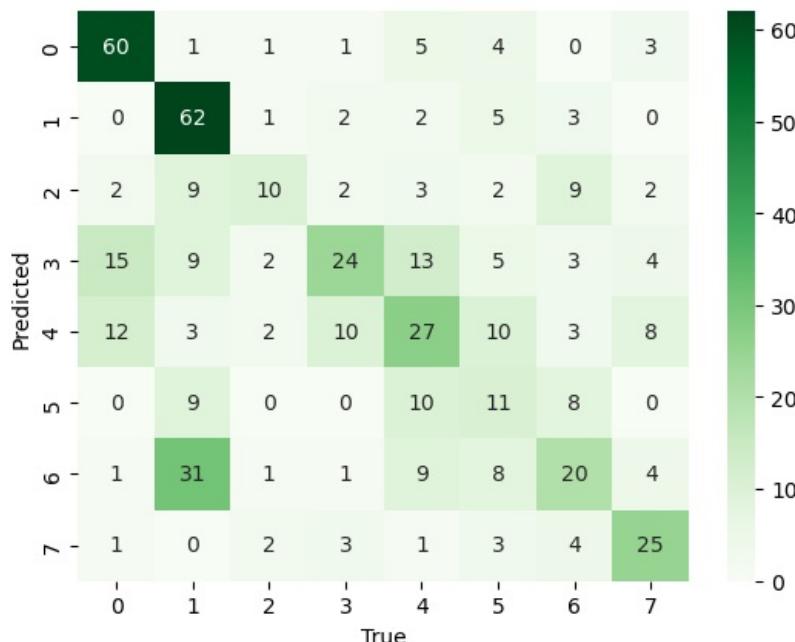
        rounded=True,
        fontsize=10,
        max_depth=2
    )
plt.show()

```



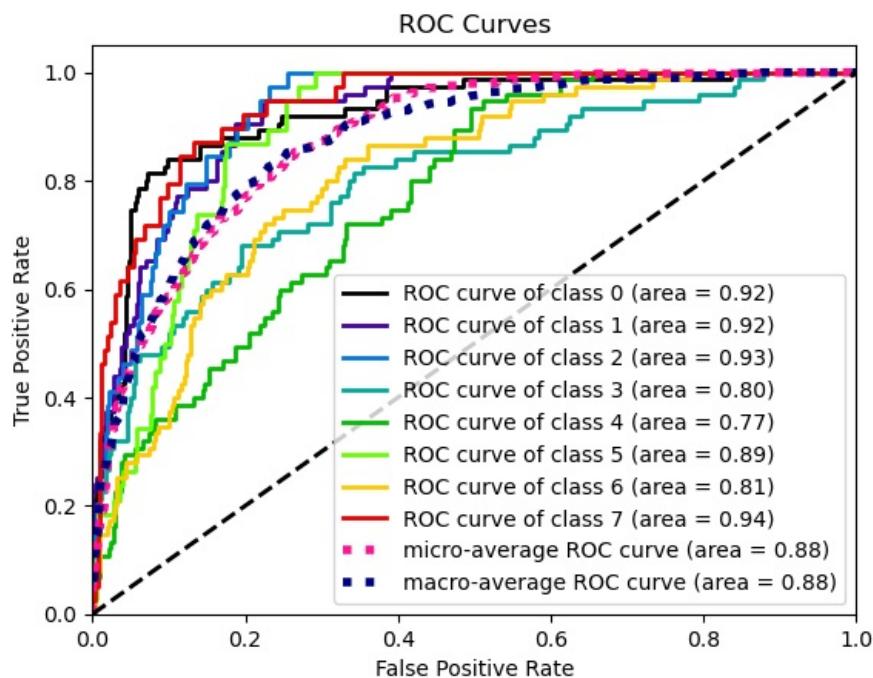
Nelle celle sotto abbiamo stampato la confusion matrix e plottato la ROC curve

```
In [321]: import seaborn as sns
cf = confusion_matrix(y_test, y_pred)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



```
In [322]: import scikitplot
y_pred_proba = clf.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

```
Out[322]: <Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>
```



Random Forest con class_weight

La variabile target *emotion* è sbilanciata nel nostro dataset nel senso che i campioni dove la emotion è surprised, disgust e neutral sono sotto rappresentati:

Emotion	Samples
fearful	376
angry	376
happy	376
calm	376
sad	376
surprised	192
disgust	192
neutral	188

0=angry, 1=calm, 2=disgust, 3=Fearful, 4=happy, 5=Neutral, 6=sad, 7=surprised.

Questo potrebbe portare il modello predittivo a essere sbilanciato verso la previsione delle classi più frequenti, potenzialmente ignorando le classi minoritarie. Ciò può causare scarse prestazioni, specialmente in termini della capacità del modello di identificare correttamente le istanze delle classi meno rappresentate. Mentre nel caso di surprised (7) il problema non sembra sussistere usando il Random Forest, le altre due classi minoritarie (disgust e neutral) sono effettivamente quelle dove il Random Forest ha la precisione più bassa. Per affrontare questo problema, aggiungiamo quindi il parametro **class_weight** alla Randomized Search lo settiamo a *balanced* che imposta il peso di ogni classe in maniera inversamente proporzionale alla frequenza di quella classe.

```
In [338]: from sklearn.metrics import accuracy_score
from sklearn.model_selection import RandomizedSearchCV
```

```
In [340]: clf = RandomForestClassifier (criterion='gini',
                                 max_depth=11,
                                 min_samples_split=11,
                                 min_samples_leaf=3,
                                 ccp_alpha=0.0,
                                 random_state=0
)
param_dict = {
    'max_depth': np.arange(1, 20+1, 1).tolist(),
    'min_samples_split': np.arange(2, 50+1, 1),
}
```

```

'min_samples_leaf': np.arange(1, 50+1, 1),
'ccp_alpha': np.arange(0.0,1, 0.1),
'class_weight':['balanced']
}

random = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=500, random_state=0)
random.fit(X_train_val, y_train_val)
random.best_params_

best_params = random.best_params_

clf = random.best_estimator_

clf.fit(X_train, y_train)

```

Out[340]: RandomForestClassifier

```
RandomForestClassifier(class_weight='balanced', max_depth=16,
                      min_samples_leaf=5, min_samples_split=6, random_state=0)
```

Applicando il parametro `class_weight` al nostro classificatore Random Forest, abbiamo osservato un'accuratezza complessiva del 0.49, un risultato in linea con le prestazioni del modello precedente, che non teneva conto del peso delle classi. Tuttavia, l'aspetto più rilevante di questa implementazione si nota nell'analisi dettagliata della precisione relativa alle singole classi di emozioni. Abbiamo registrato un netto miglioramento nella capacità del modello di identificare con precisione le emozioni 'neutral', 'disgust', e 'surprised'. Queste classi, tra le meno rappresentate nel nostro dataset, hanno beneficiato in modo evidente dell'adozione di `class_weight`, che ha equilibrato efficacemente la loro rappresentazione nell'addestramento del modello. Questo approccio ha dimostrato la sua validità non solo nel migliorare la precisione di classi tradizionalmente problematiche o sotto-rappresentate ma ha anche confermato la capacità del parametro di ottimizzare le prestazioni complessive del modello, rendendolo più accurato e bilanciato nel riconoscimento delle diverse emozioni.

```
In [342]: from sklearn.metrics import classification_report
y_pred = clf.predict(X_test)
y_pred_trainval = clf.predict(X_train_val)

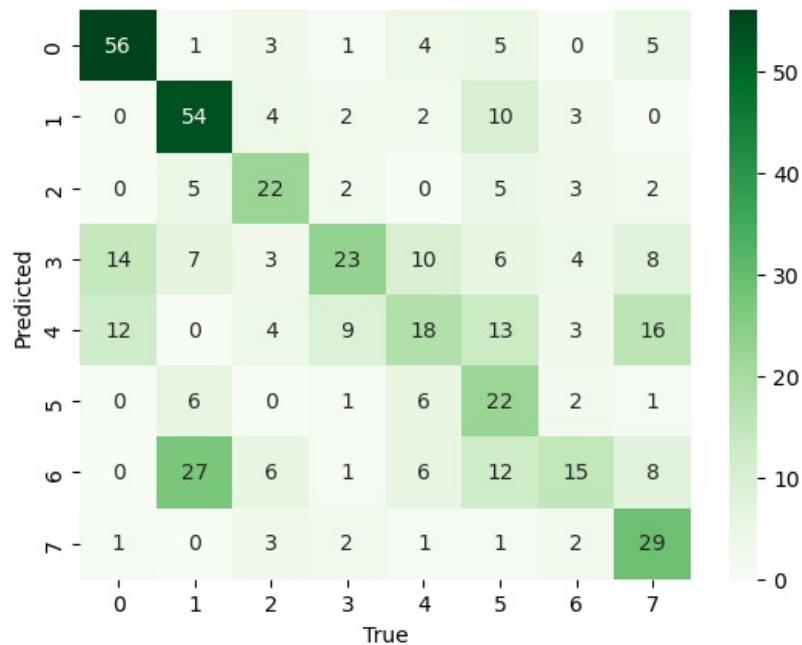
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)

print(classification_report(y_pred, y_test))
print(classification_report(y_pred_trainval, y_train_val))
```

	precision	recall	f1-score	support
0	0.75	0.67	0.71	83
1	0.72	0.54	0.62	100
2	0.56	0.49	0.52	45
3	0.31	0.56	0.40	41
4	0.24	0.38	0.30	47
5	0.58	0.30	0.39	74
6	0.20	0.47	0.28	32
7	0.74	0.42	0.54	69
accuracy			0.49	491
macro avg	0.51	0.48	0.47	491
weighted avg	0.58	0.49	0.51	491
	precision	recall	f1-score	support
0	0.83	0.88	0.85	285
1	0.89	0.84	0.87	318
2	0.90	0.75	0.82	182
3	0.75	0.86	0.80	263
4	0.71	0.82	0.76	261
5	0.93	0.65	0.76	214
6	0.68	0.89	0.77	232
7	0.88	0.65	0.75	206
accuracy			0.80	1961
macro avg	0.82	0.79	0.80	1961
weighted avg	0.82	0.80	0.80	1961

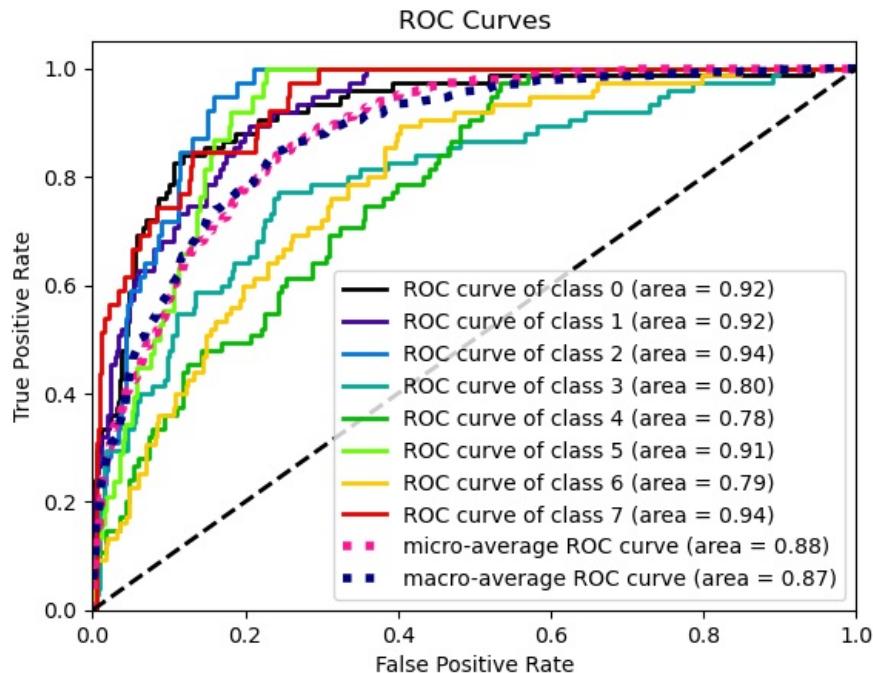
Nelle celle seguenti abbiamo disegnato la confusion matrix e plottato la ROC curve per questo modello predittivo.

```
In [344]: import seaborn as sns
cf = confusion_matrix(y_test, y_pred)
sns.heatmap(cf, annot=True, cmap="Greens")
plt.xlabel("True")
plt.ylabel("Predicted")
plt.show()
```



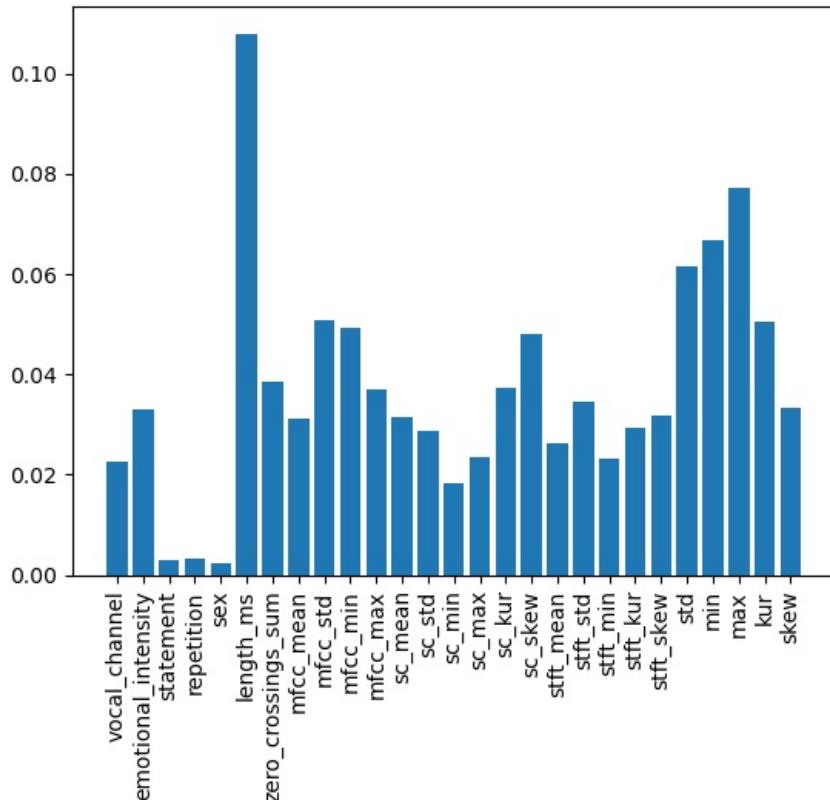
```
In [345]: import scikitplot
y_pred_proba = clf.predict_proba(X_test)
scikitplot.metrics.plot_roc(y_test,y_pred_proba)
```

```
Out[345]: <Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate', ylabel='True Positive Rate'>
```



Determiniamo le Feature Importance del **Random Forest**, in cui le più influenti risultano essere ancora *min,max,length_ms* anche se per questo modello *length_ms* diventa la feature più importante. Questo risultato suggerisce che probabilmente *length_ms* riveste un'importanza particolare nella predizione delle classi meno rappresentate, il cui equilibrio è stato ottimizzato attraverso l'applicazione del parametro *class_weight*.

```
In [347]: plt.bar(columns, clf.feature_importances_)
plt.xticks(rotation=90)
plt.show()
```

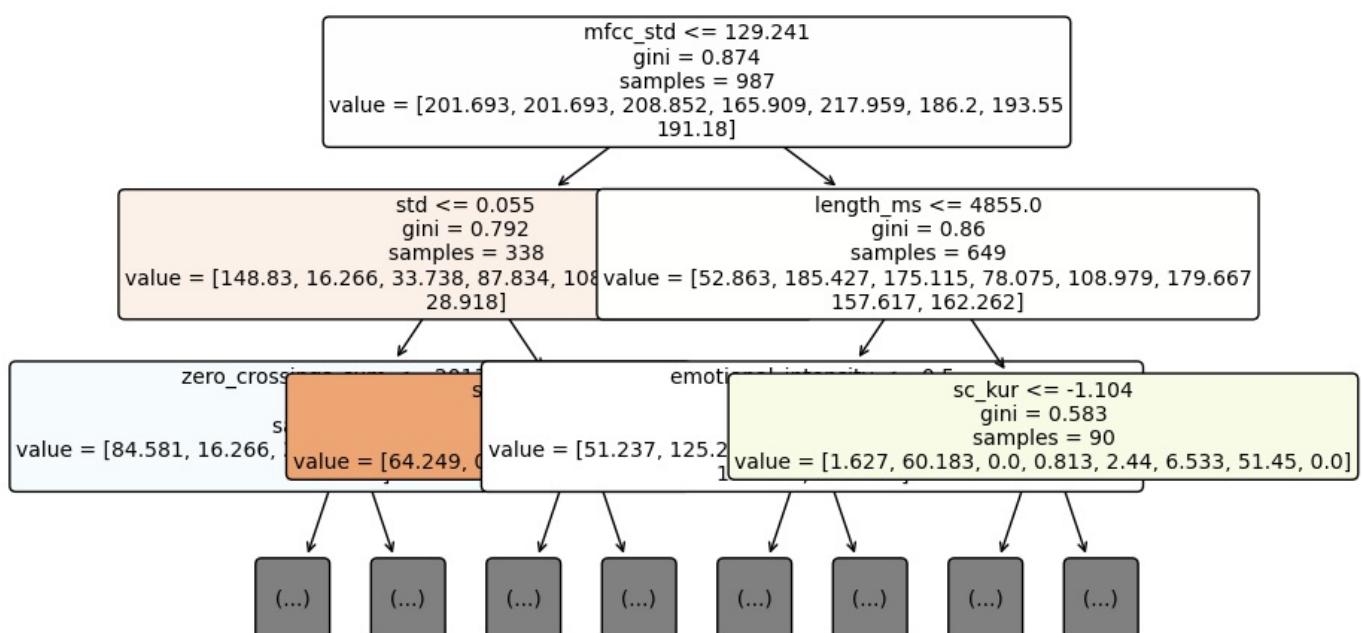


Vediamo adesso quanti DecisionTrees sono stati usati per creare la RandomForest (100) e ne visualizziamo due (il primo e il secondo)

```
In [349]: len(clf.estimators_)
```

```
Out[349]: 100
```

```
In [350]: plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[0],
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=10,
          max_depth=2
         )
plt.show()
```

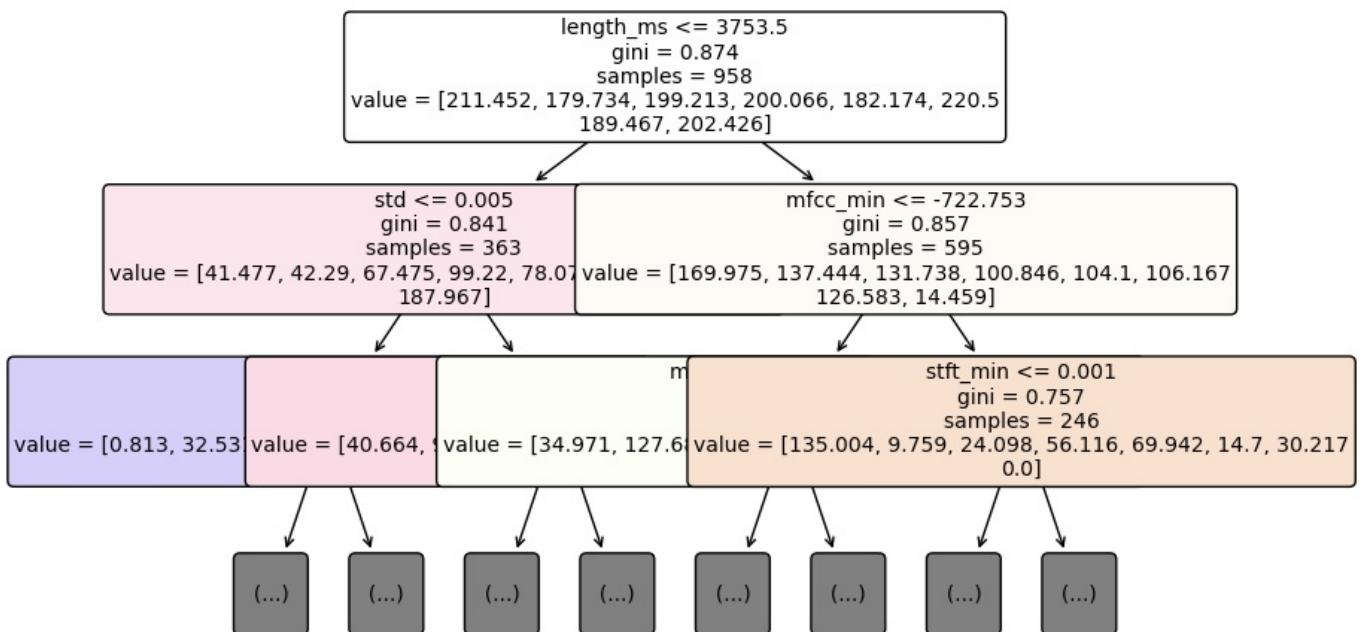


```
In [351]: plt.figure(figsize=(8, 6))
plot_tree(clf.estimators_[1],
          feature_names=columns,
          filled=True,
          rounded=True,
          fontsize=10,
```

```

        max_depth=2
    )
plt.show()

```



SVM

Procediamo ora ad utilizzare la Support Vector Machine (SVM) sul nostro dataset. La SVM è un metodo predittivo che cerca un iper piano per separare i dati nello spazio delle feature. Per ridurre il carico computazionale, abbiamo deciso di ridurre la dimensionalità del Training Set.

Importiamo sia LinearSVC, che effettua una separazione lineare, sia SVC, che consente l'utilizzo del kernel trick per gestire dati non linearmente separabili.

```

In [262...]: from sklearn.model_selection import train_test_split
          from sklearn.svm import LinearSVC, SVC
          from sklearn.metrics import accuracy_score
          from sklearn.model_selection import RandomizedSearchCV

In [264...]: target = 'Emotion_val'
            columns = [c for c in dfe_noobj.columns if c not in ['Emotion_val','sex','emotional_intensity','statement','repo...']

In [266...]: X = dfe_noobj[columns].values
            y = dfe_noobj[target].values

In [268...]: X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=0)
            X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.2, stratify=y_train_val)

In [284...]: lin_svm = LinearSVC(C=1,max_iter=5000, dual=False)
            lin_svm.fit(X_train_val, y_train_val)
            y_pred = lin_svm.predict(X_test)
            accuracy_score(y_test, y_pred)

Out[284...]: 0.37067209775967414

In [294...]: svm = SVC(C=100, kernel='poly')
            svm.fit(X_train_val, y_train_val)
            y_pred = svm.predict(X_test)
            accuracy_score(y_test, y_pred)

Out[294...]: 0.3258655804480652

```

Possiamo valutare la tipologia di Kernel e il parametro C attraverso l'**HYPERTPARAMETER TUNING** durante la fase di validazione con una RandomizedSearch. Rispetto a un tentativo iniziale abbiamo ristretto il range dei parametri e il numero di iterazioni per ridurre il carico computazionale. Abbiamo anche settato il parametro n_jobs=-1 che permette di fare più calcoli in parallelo su più core potenzialmente aumentando la velocità di calcolo.

Purtroppo, nonostante i tentativi per ridurre i parametri, il numero di iterazioni e la dimensionalità, il sistema non è stato in grado di completare i calcoli entro un tempo accettabile, e abbiamo deciso di interrompere l'esecuzione. Non è chiaro se sussiste un problema di velocità computazionale dei calcoli oppure se una delle combinazioni dei vari parametri crei un problema al sistema.

```
In [ ]: param_dict = {
    'C': [0.1, 10, 100],
    'kernel': ['linear', 'poly', 'rbf'],
    'degree': [1, 2, 4, 5],
    'gamma': ['scale', 'auto'],
    'coef0': [0.0, 1, 10],
}
clfSCV = SVC()

rands = RandomizedSearchCV(clf, param_dict, cv=5, scoring='accuracy', refit=True, n_iter=50, random_state=0, n_jobs=-1)
rands.fit(X_train_val, y_train_val)

print("Best parameters:", rands.best_params_)

clf = rands.best_estimator_

clf.fit(X_train_val, y_train_val)

y_pred = clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.4f}')
```

Classification - Discussion and Conclusions

In questa sezione abbiamo esplorato l'utilizzo di alcuni modelli di classificazione per prevedere due specifiche variabili del nostro dataset: "Sex" ed "Emotion". Ogni modello ha offerto spunti di riflessione e in particolare sulla performance in termini di accuratezza e precisione.

Nel caso della variabile target Sex, sia il Decision Tree che il KNN hanno dato ottimi risultati con accuratezze intorno al 90%. Nonostante questa similarità, il KNN sembra essere migliore del Decision Tree non solo perché la sua accuratezza è leggermente più alta, ma anche perché la precision sulle due classi è meno sbilanciata. Questo può essere attribuito alle sue caratteristiche intrinseche. Il KNN, essendo un algoritmo basato sulla distanza, tende a essere più flessibile e meno incline all'overfitting rispetto ai Decision Trees. Inoltre, la capacità del KNN di mantenere una precisione equilibrata tra le classi suggerisce che gestisce meglio il problema dello sbilanciamento delle classi senza necessità di tecniche aggiuntive.

Per quel che riguarda la classe Emotion, il modello che si è dimostrato migliore è stato il Random Forest. Questo non è sorprendente visto che è un modello ensemble che combina le previsioni di diversi alberi decisionali e quindi generalmente più robusto e accurato. Andando invece ad analizzare la precisione dei vari modelli per le singole emozioni, notiamo che questa varia in maniera evidente in tutti i modelli esaminati, con le emozioni come *angry* e *calm* che consistentemente mostrano una precision più alta per tutti i modelli esaminati e altre come *surprised*, *sad*, *disgust* e *neutral* con precision tendenzialmente più basse (anche se queste variano al variare del modello). La conclusione che possiamo trarre è quella che anche se il modello di Random Forest è quello migliore, questo non vale per tutte le classi della variabile target. Per ovviare a questo, abbiamo provato a introdurre il parametro `class_weight` durante l'addestramento del Random Forest e dato un peso alle varie classi inversamente proporziale alla loro rappresentazione nel dataset. La precision per alcune classi è migliorata anche se per altre è peggiorata, così come è peggiorata l'accuracy generale del modello.

Se volessimo prevedere una classe specifica di Emotion, potremmo usare le seguenti strategie:

1. Scegliere il metodo con la migliore precisione per la classe d'interesse (o altre metriche di valutazione a seconda di quello su cui vogliamo focalizzarci).
2. Lavorare sulla `class_weight` delle classi durante la fase di allenamento del modello
3. Identificare e utilizzare le feature che esercitano maggiore influenza sulla predizione della classe di interesse, ottimizzando così le prestazioni del modello su quella specifica categoria.
4. Validazione incrociata e tuning degli iperparametri focalizzandoci su una classe specifica

In conclusione, la scelta del modello migliore dipende dall'obiettivo specifico, dalla natura dei dati e dalla distribuzione delle classi, richiedendo una valutazione per bilanciare accuratezza generale e distribuzione tra le classi.

In []:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js