Imports

In [1]:

```
import pandas as pd
import os
import seaborn as sns
import plotly.graph_objects as go
import matplotlib_inline

import torch

import matplotlib.pyplot as plt
from matplotlib.pyplot import figure, show
from matplotlib.ticker import MaxNLocator

import numpy as np
from numpy.testing import assert_array_equal, assert_array_almost_equal
from numpy.testing import assert_almost_equal

from urllib.request import urlretrieve

from imblearn.over_sampling import SMOTE

from collections import Counter

from sklearn import svm
from sklearn.utils import shuffle
from sklearn.svm import LinearSVC
from sklearn.metrics import zero_one_loss, accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import MultiLabelBinarizer, normalize
from sklearn.naive_bayes import GaussianNB, CategoricalNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import *

from IPython.display import display, HTML, Image

from scipy.spatial.distance import pdist, cdist, squareform

pd.set_option('display.max_columns', None) # to display all columns

#matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
```

Preliminaries

Global variables

In [2]:

```
NR_BINS = 5 # discretization

font_labels = {'family': 'serif', 'color':  'black', 'weight': 'normal', 'size': 16}    # graphs' aesthetics

current_path = os.getcwd()
image_path = os.path.join(current_path, "images")
tuning_path = os.path.join(current_path, "tuning_hyperparams")
results_path = os.path.join(current_path, "results")
datasets_path = os.path.join(current_path, "datasets")
```

We set up the folders hierarchy.

In [3]:

```
if not os.path.exists(image_path):
    os.makedirs(image_path)
if not os.path.exists(tuning_path):
    os.makedirs(tuning_path)
if not os.path.exists(results_path):
    os.makedirs(results_path)
if not os.path.exists(datasets_path):
    os.makedirs(datasets_path)
```

Utility functions

In [4]:

```
def showImagesHorizontally(files):      # to plot images horizontally aligned
        n_files = len(files)
        fig, ax = plt.subplots(nrows=1, ncols=n_files)
        fig.set_size_inches(7 *n_files, 9)
```

```
        for i in range(n_files):
            image = plt.imread(files[i])

            ax[i].imshow(image, cmap='gray', aspect='equal', resample=False)
            ax[i].axis('off')


def showTablesHorizontally(dfs, captions, tablespacing=5):
    output = ""
    for (caption, df) in zip(captions, dfs):
        output += df.style.set_table_attributes("style='display:inline'").set_caption(caption)._repr_html_()
        output += tablespacing * "\xa0"
    display(HTML(output))


def name(dataset_name):
    return "_".join(['df', dataset_name])


def seed_everything(seed=42):
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

INTRODUCTION

The goal of this project is the design and the implementation of a classifier which manages to predict the score of a film starting from its features.

We build the data pipeline studied during the course:\ *Data acquisition* → *Data Pre-processing* → *Data Visualization* → *Modeling* → *Performance analysis* → *Data Visualization*

DATA ACQUISITION

We load the datasets in memory using pandas library

In [5]:
```
csv_names = ["movies.csv", "ratings.csv", "genome-scores.csv", "genome-tags.csv", "links.csv", "tags.csv"]
datasets_names = [i[:-4].replace("-", "_") for i in csv_names]

remote_csv_dir = "http://github.com/MickPerl/DataAnalyticsProject/releases/download/datasets/"
```

The following code tries to read the csv files given the local path: in case of failure, it downloads the file from the remote path (Github release) to the local path, so that it can successfully read them.

In any case we create the variabiles `df_[filename]` and load in memory the related DataFrame.

In [6]:
```
for i in range(len(csv_names)):
    local_csv_path = os.path.join(datasets_path, csv_names[i])

    try:
        globals()[name(datasets_names[i])] = pd.read_csv(local_csv_path)    # globals()[parametrized_name_variable] to crea

    except FileNotFoundError:
        print(f"Download in progress of {csv_names[i]}")
        remote_csv_path = os.path.join(remote_csv_dir, csv_names[i])
        file, _ = urlretrieve(url = remote_csv_path, filename=local_csv_path)
        globals()[name(datasets_names[i])] = pd.read_csv(file)
```

DATA PREPROCESSING

Data manipulation

Within this section, we study each DataFrame obtained by executing the previous data acquisition: we highlight their peculiarities and join them in order to end up getting the complete dataset.

df_movies

In [7]:
```
output = df_movies.head().style.set_caption("First 5 rows of movies.csv")._repr_html_()
display(HTML(output))
```

First 5 rows of movies.csv

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

In [8]:

```
print(*df_movies.columns)
```

```
movieId title genres
```

The `df_movies` DataFrame contains the above information with regard to 58 098 films.

In [9]:

```
len(df_movies)
```

Out[9]:

```
58098
```

We extract useful informations from `title`, namely the number of characters of the title as `title_lenght` and the year of the film as `year` since it is reported in the final part of the title.

Afterward, since the column `title` contains unique values, we consider it a negligible information so we drop it.

In [10]:

```
df_movies["title_length"] = df_movies['title'].str.len()

# regular expression and not a straightforward slicing in that the titles are quite noisy
exp = '\((18\d{2}|19\d{2}|20\d{2})(?!.*\1)'
df_movies["year"] = df_movies['title'].str.extract(pat=exp).astype(float)
# some titles have not the year, so there will be some NaN values which will be managed during the data cleaning process (w

df_movies.drop('title', axis=1, inplace=True)
```

For each film, the dataset `df_movies` stores its genres in a pipe-separated list:

In [11]:

```
df_movies.genres.head()
```

Out[11]:

```
0    Adventure|Animation|Children|Comedy|Fantasy
1                    Adventure|Children|Fantasy
2                                Comedy|Romance
3                          Comedy|Drama|Romance
4                                        Comedy
Name: genres, dtype: object
```

We choose to add one column for each possible genre: if the film belongs to a certain genre, the corresponding cell will contain `1`, `0` otherwise.

To do that, first of all we instantiate the class `MultiLabelBinarizer` from the `preprocessing` module of the `sklearn` library: its method `fit_transform` takes as input the set of labels for each samples and outputs a matrix of shape (# samples, # unique labels) such that each cell has `1` if the corresponding film belongs to the corresponding genre, `0` otherwise.

Then, we convert this matrix to a `DataFrame` object whose columns are the genres' name; eventually, the result is joined with `df_movies` DataFrame.

In [12]:

```
mlb = MultiLabelBinarizer()
df_movies = df_movies.join(pd.DataFrame(
                mlb.fit_transform(df_movies.pop('genres').str.split('|')),
                index=df_movies.index,
                columns=mlb.classes_))

df_movies.head()
```

Out[12]:

	movieId	title_length	year	(no genres listed)	Action	Adventure	Animation	Children	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horror
0	1	16	1995.0	0	0	1	1	1	1	0	0	0	1	0	0
1	2	14	1995.0	0	0	1	0	1	0	0	0	0	1	0	0
2	3	23	1995.0	0	0	0	0	0	1	0	0	0	0	0	0
3	4	24	1995.0	0	0	0	0	0	1	0	0	1	0	0	0
4	5	34	1995.0	0	0	0	0	0	1	0	0	0	0	0	0

df_genome_scores and df_genome_tags

`df_genome_scores` contains tag relevance scores for movies.\ The set of relevances regarding a film is its *genome*: thus, the genome encodes how strongly movies exhibit particular properties represented by tags (atmospheric, thought-provoking, realistic, etc.).

In [13]:
```
output = df_genome_scores.head().style.set_caption('First 5 rows of genome-scores.csv')._repr_html_()
display(HTML(output))
```

First 5 rows of genome-scores.csv

	movieId	tagId	relevance
0	1	1	0.029000
1	1	2	0.023750
2	1	3	0.054250
3	1	4	0.068750
4	1	5	0.160000

The `df_genome_scores` DataFrame contains $14\,862\,528$ relevance scores for movies.

In [14]:
```
len(df_genome_scores)
```

Out[14]: 14862528

In particular, only $13\,176$ out of $58\,098$ films have a genome: the remaining $48\,607$ are not characterized by any tag.

In [15]:
```
len(df_genome_scores.groupby("movieId"))
```

Out[15]: 13176

The structure of `df_genome_scores` is a dense matrix: each movie within it has a value for every tag in the genome as it is possible to prove through grouping by `movieId` value and seeing that the number of unique tags is the same over all films.

In [16]:
```
print(*df_genome_scores.groupby("movieId").count().tagId.unique())
```

1128

The `df_genome_tags` DataFrame contains $1\,128$ tags.

In [17]:
```
len(df_genome_tags)
```

Out[17]: 1128

We can continue merging `df_genome_scores` with `df_genome_tags` which explicits every tag.

In [18]:
```
showTablesHorizontally([df_genome_scores.head(), df_genome_tags.head()], ['First 5 rows of df_genome_scores', 'First 5 rows
```

First 5 rows of df_genome_scores

	movieId	tagId	relevance
0	1	1	0.029000
1	1	2	0.023750
2	1	3	0.054250
3	1	4	0.068750
4	1	5	0.160000

First 5 rows of df_genome_tags

	tagId	tag
0	1	007
1	2	007 (series)
2	3	18th century
3	4	1920s
4	5	1930s

In [19]:
```
df_genome = pd.merge(df_genome_scores, df_genome_tags, on="tagId", how="left")
```

After that, we want to manipulate the `df_genome` DataFrame in order to relate every film to its genome.

In [20]:
```
df_genome = df_genome.pivot(index='movieId', columns='tag', values="relevance")
df_genome.head()
```

Out[20]:

tag	007	007 (series)	18th century	1920s	1930s	1950s	1960s	1970s	1980s	19th century	3d	70mm	80s	9/11	aardman
movieId															
1	0.02900	0.02375	0.05425	0.06875	0.16000	0.19525	0.07600	0.25200	0.22750	0.02400	0.58700	0.09425	0.17800	0.00700	0.03525
2	0.03625	0.03625	0.08275	0.08175	0.10200	0.06900	0.05775	0.10100	0.08225	0.05250	0.08900	0.09800	0.16325	0.00650	0.00450
3	0.04150	0.04950	0.03000	0.09525	0.04525	0.05925	0.04000	0.14150	0.04075	0.03200	0.02850	0.05900	0.08550	0.00475	0.00525
4	0.03350	0.03675	0.04275	0.02625	0.05250	0.03025	0.02425	0.07475	0.03750	0.02400	0.02750	0.03375	0.07750	0.01075	0.00325
5	0.04050	0.05175	0.03600	0.04625	0.05500	0.08000	0.02150	0.07375	0.02825	0.02375	0.02825	0.03175	0.05675	0.00825	0.00300

We can merge the result with `df_movies`.

In [21]:
```
df = pd.merge(df_movies, df_genome, on="movieId", how = "left")
df.head()
```

Out[21]:

	movieId	title_length	year	(no genres listed)	Action	Adventure	Animation	Children	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horror
0	1	16	1995.0	0	0	1	1	1	1	0	0	0	1	0	0
1	2	14	1995.0	0	0	1	0	1	0	0	0	0	1	0	0
2	3	23	1995.0	0	0	0	0	0	1	0	0	0	0	0	0
3	4	24	1995.0	0	0	0	0	0	1	0	0	1	0	0	0
4	5	34	1995.0	0	0	0	0	0	1	0	0	0	0	0	0

df_ratings

Each row of `df_ratings` DataFrame represents one rating of one movie by one user, and has the following format:

In [22]:
```
output = df_ratings.head().style.set_caption("First 5 rows of ratings.csv")._repr_html_()
display(HTML(output))
```

First 5 rows of ratings.csv

	userId	movieId	rating	timestamp
0	1	307	3.500000	1256677221
1	1	481	3.500000	1256677456
2	1	1091	1.500000	1256677471
3	1	1257	4.500000	1256677460
4	1	1449	4.500000	1256677264

The `df_rating` DataFrame contains $27\,753\,444$ ratings scores for movies.

In [23]:
```
len(df_ratings)
```

Out[23]: 27753444

Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).\ *Timestamps* represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

We select only the `movieId` and `rating` columns and we average ratings relating each film and then we join the result with the complete DataFrame.

In [24]:
```
df_ratings = df_ratings.iloc[:,[1,2]]
```

```
df_ratings = df_ratings.groupby(by='movieId').rating.agg(rating_mean= 'mean', ratings_count= 'count')
df_ratings.head()
```

Out[24]:

movieId	rating_mean	ratings_count
1	3.886649	68469
2	3.246583	27143
3	3.173981	15585
4	2.874540	2989
5	3.077291	15474

Ratings are indicated for 53 889 out of 58 098 movies.

In [25]:
```
len(df_ratings)
```

Out[25]:
```
53889
```

In [26]:
```
df = pd.merge(df, df_ratings, on="movieId", how="left")
```

Result: Final Dataframe

In [27]:
```
df.head()
```

Out[27]:

	movieId	title_length	year	(no genres listed)	Action	Adventure	Animation	Children	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horro
0	1	16	1995.0	0	0	1	1	1	1	0	0	0	1	0	0
1	2	14	1995.0	0	0	1	0	1	0	0	0	0	1	0	0
2	3	23	1995.0	0	0	0	0	0	1	0	0	0	0	0	0
3	4	24	1995.0	0	0	0	0	0	1	0	0	1	0	0	0
4	5	34	1995.0	0	0	0	0	0	1	0	0	0	0	0	0

Data cleaning

Checking missing data

Films without rating

There are 4 209 films without a rating.

In [28]:
```
sum(df.rating_mean.isna())
```

Out[28]:
```
4209
```

Since the rating is the label and the cardinality of the complete dataset is big, we can drop these films.

In [29]:
```
df.dropna(subset=['rating_mean'], inplace=True)
```

The new cardinality is 53 889.

In [30]:
```
len(df)
```

Out[30]:
```
53889
```

We can downcast rating_count to int

In [31]:
```
df.ratings_count = df.ratings_count.astype('int')
```

Films without tags and/or genres

There are 40 713 films without a relevance for any tag.

In [32]:
```
sum((df.iloc[:, 23:-2]).isna().all(axis = 1))
```

Out[32]:
```
40713
```

We save these films in a temporary DataFrame.

In [33]:
```
df_without_tags = df[df.iloc[:, 23:-2].isna().all(axis = 1)]
```

Within this subset, we want to look for films without any genres.

In [34]:
```
df_without_tags_nor_genres = df_without_tags[df_without_tags['(no genres listed)'] == 1]
```

There are 3 703 films whithout tags nor genres.

In [35]:
```
len(df_without_tags_nor_genres)
```

Out[35]:
```
3703
```

We drop these samples.

In [36]:
```
index_rows_to_be_deleted = df.loc[df["movieId"].isin(df_without_tags_nor_genres["movieId"])].index
```

In [37]:
```
df.drop(index_rows_to_be_deleted, axis=0, inplace=True)
```

The new cardinality is 50 186.

In [38]:
```
len(df)
```

Out[38]:
```
50186
```

Regarding remaining films without tags (but with at least one genre), we set to `0` their relevance under the different tags.

In [39]:
```
df.iloc[:, 23:-2] = df.iloc[:, 23:-2].fillna(0)
```

After the dropping, we look for films whithout genres overall the new dataframe: it points out that there are still 29 films which do not belong to any genres.

In [40]:
```
sum(df['(no genres listed)'] == 1)
```

Out[40]:
```
29
```

We can prove that these films are those which have `0` under the columns of the genres.

In [41]:
```
sum((df.iloc[:, 4:23] == 0).all(axis = 1)) == sum(df['(no genres listed)'] == 1)
```

Out[41]:
```
True
```

Since the absence of genres is already stored in the genres columns, we drop the `(no genres listed)` column.

In [42]:
```
df.drop(['(no genres listed)'], inplace=True, axis=1)
```

Films without years

There are 100 films without an year.

In [43]:
```
sum(df.year.isna())
```

Out[43]:
```
100
```

To decide the value to substitute to `na`, we plot the distribution of the values.

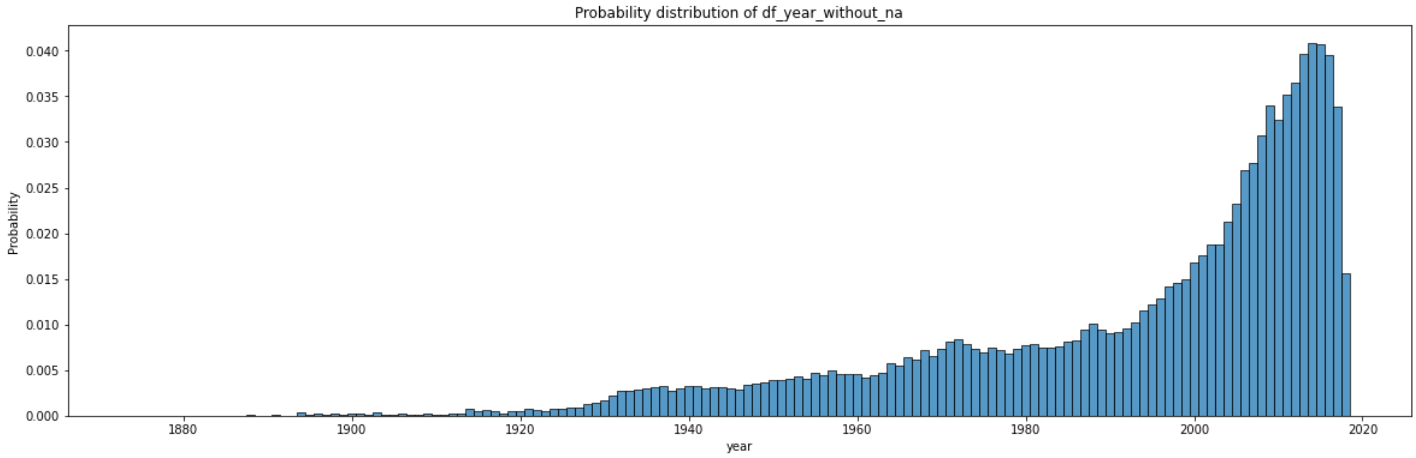
In [44]:
```
df_year_without_na = df.year[-pd.isna(df.year)]
```

Focus on probability and density plot

During the exploratory data analysis, we have to draw several histogram representing the probability distributions of discrete features: we use the function `histplot` of the `seaborn` library by specifying `probability` as `stat` value.

In [45]:

```
plt.figure(figsize=(20,6))
sns.histplot(df_year_without_na, stat='probability', discrete=True) # when discrete is True, the width of bins is 1 so tha
plt.title("Probability distribution of df_year_without_na")
plt.show()
```

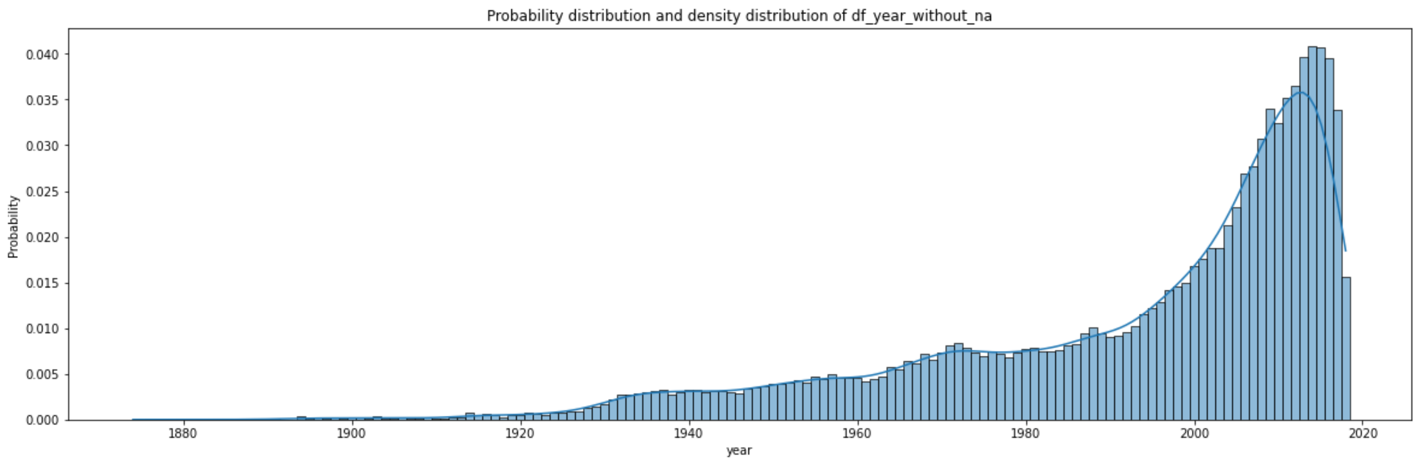


After some research, we discover the *probability density distribution* (aka *density plot*), which is a smoothed continuous version of the probability distribution and can be estimated through various methods. One of them is the *kernel density estimation* (kde) which draws a continuous curve (the kernel) at every individual data point and all of these curves are then added together to make a single smooth density estimation: the kernel most often used is a Gaussian (which produces a Gaussian bell curve at each data point).

Relative to a histogram, the density plot is less cluttered and more interpretable, especially when drawing multiple distributions: the function `histplot` lets to display on the same plot the histogram representing the probability distribution as well as the curve representing the density distribution.

In [46]:

```
plt.figure(figsize=(20,6))
sns.histplot(df_year_without_na, kde=True, stat='probability', discrete=True)
plt.title("Probability distribution and density distribution of df_year_without_na")
plt.show()
```

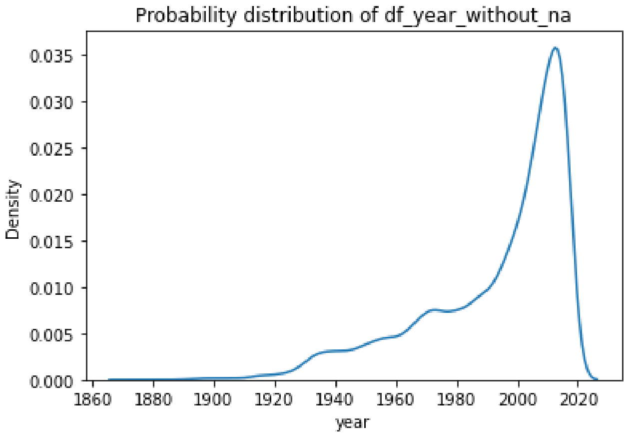


The *probability density* is the probability per unit on the x-axis: to convert to an actual probability, since the values are considered in a continuous fashion, we need to find the area under the curve for a specific interval on the x-axis.\ Since this is a probability density and not a probability, *the y-axis can take values greater than one*: the only requirement of the density plot is that the total area under the curve integrates to one.

Henceforth, to compare the distributions between different features, we are going to adopt the density plot thanks to its greater interpretability: we define a function which remove the bars from the above graphs in order to mantain only the density distribution.

In [47]:

```
sns.kdeplot(df_year_without_na)
plt.title("Probability distribution of df_year_without_na")
plt.show()
```



Since it's not simmetric, we decide to fill `na` values with the *median*.

In [48]:
```
df.year = df.loc[:, 'year'].fillna(np.median(df_year_without_na)).astype('int')
```

We can check the absence of `Null` or `Na` values.

In [49]:
```
sum(pd.isna(df).any(axis=1))
```

Out[49]:
```
0
```

After checking missing data and before checking duplicates, we drop the `movieId` column since contains unique values which are useful only to indexing purposes.

In [50]:
```
df.drop('movieId', inplace=True, axis=1)
```

Checking duplicates

There are 393 duplicated rows.

In [51]:
```
sum(df.duplicated())
```

Out[51]:
```
393
```

We have deleted them by keeping the first.

In [52]:
```
df.drop_duplicates(inplace=True)
```

So that, there are not duplicated rows.

In [53]:
```
sum(df.duplicated())
```

Out[53]:
```
0
```

As result of Data Cleaning we have the cardinality of $49\,793$ movies

In [54]:
```
len(df)
```

Out[54]:
```
49793
```

Data Trasformation

Continuous label discretization

The `ratings` column contains the labels which are continuous; we discretize them by binning into 5 intervals with the same length: since the original range is 4.5, the range of every bin/interval is 0.9.

In [55]:
```
df['bin_y'] = pd.cut(df['rating_mean'], bins=NR_BINS, labels=False)
```

Train/Test/Validation set splitting

In [56]:
```
df_train, df_test = train_test_split(df, test_size=0.2, random_state=13, stratify=df['bin_y'])
df_train, df_val = train_test_split(df_train, test_size=0.1, random_state=13, stratify=df_train['bin_y'])
```

After split we reset indexes

In [57]:
```
df_train.reset_index(drop=True, inplace=True)
df_test.reset_index(drop=True, inplace=True)
df_val.reset_index(drop=True, inplace=True)
```

Extracting X, y and weights from Training/Validation/Test Set

In [58]:
```
def split_XYweights(df):
        y_categorical = df['bin_y'].astype('int')
        y_continuous = df['rating_mean'].astype('int')
        weights = df['ratings_count']
        X = df.drop(columns=['bin_y', 'rating_mean', 'ratings_count'], axis=1)

        return X, weights, y_categorical, y_continuous
```

In [59]:
```
y_train_continuous = df_train['rating_mean']
X_train, train_ratings_count, y_train_categorical, y_train_continuous = split_XYweights(df_train)

y_val_continuous = df_val['rating_mean']
X_val, val_ratings_count, y_val_categorical, y_test_continuous = split_XYweights(df_val)

y_test_categorical = df_test['bin_y']
X_test, test_ratings_count, y_test_categorical, y_test_continuous = split_XYweights(df_test)
```

Evaluating Standardization or Min-Max Scaling

In [60]:
```
pd.set_option('display.max_rows', df.shape[0]+1)
X_train.describe().loc[['mean', 'min', 'max']]
```

Out[60]:

	title_length	year	Action	Adventure	Animation	Children	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horr
mean	26.203877	1992.935565	0.132469	0.075453	0.052915	0.054505	0.297685	0.094672	0.097183	0.44781	0.052413	0.007392	0.1042
min	3.000000	1888.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.0000
max	191.000000	2018.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	1.0000

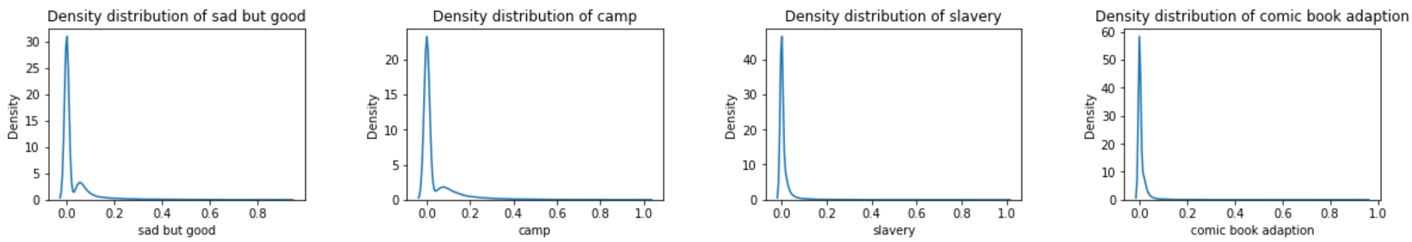
The genres columns have `0` or `1` as values.\ We evaluate the feasibility of the standardization over relevance tags by plotting their probability distribution.

Since the number of tags is too high ($1\,127$), we plot the distribution of n randomly sampled tags.

In [61]:
```
len(X_train.iloc[:,22:].columns)
```

Out[61]: 1127

In [62]:
```
n = 4
columns_sampled = np.random.choice(df.iloc[:,21:-2].columns, 12, replace=False)

fig = plt.figure(figsize = (20,10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in np.arange(1,n+1):
        ax = fig.add_subplot(3, 4, i)
        sns.kdeplot(X_train[columns_sampled[i-1]])
        plt.title(f"Density distribution of {columns_sampled[i-1]}")
plt.show()
```



It turns out that distributions are not gaussian, therefore we exclude the standardization: the min-max scaling is pointless since values are already scaled between 0 and 1.

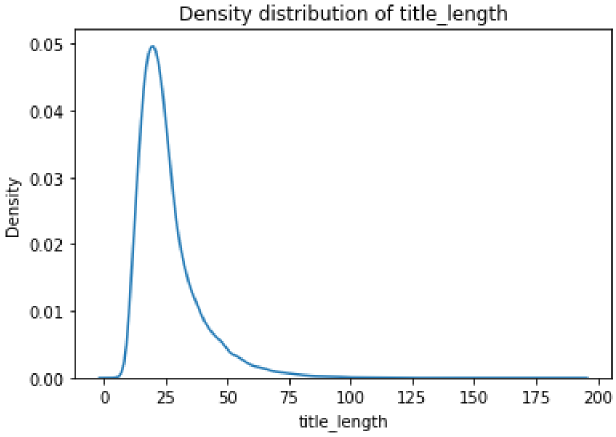
We define a function to min-max scale a specified column of the training set, validation set and test set: we evaluate it on remaining features.

In [63]:
```
def MinMaxScaling(X_train, X_val, X_test, cols):
        X_train_minmaxscaled = X_train.copy()
        X_val_minmaxscaled = X_val.copy()
        X_test_minmaxscaled = X_test.copy()

        for col in cols:
                min = np.min(X_train[col])
                max = np.max(X_train[col])
                range = max - min

                X_train_minmaxscaled[col] = (X_train[col] - min) / range
                X_val_minmaxscaled[col] = (X_val[col] - min) / range
                X_test_minmaxscaled[col] = (X_test[col] - min) / range

        return X_train_minmaxscaled, X_val_minmaxscaled, X_test_minmaxscaled
```

title_len

Regarding the `title_length` feature, we plot its distribution to evaluate the feasibility of the standardization.

In [64]:
```
sns.kdeplot(X_train['title_length'])
plt.title("Density distribution of title_length")
plt.savefig(os.path.join(image_path, "initial_title_lenght.png"),facecolor='white', transparent=False, bbox_inches='tight')
```

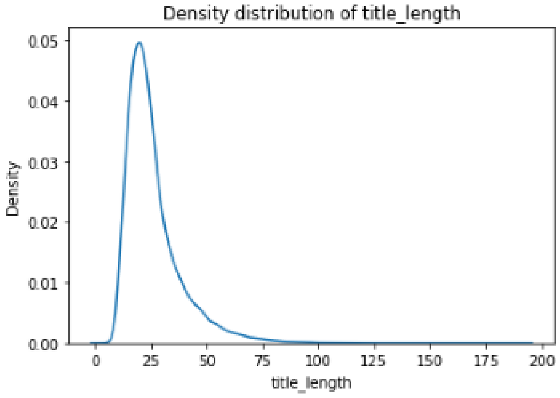


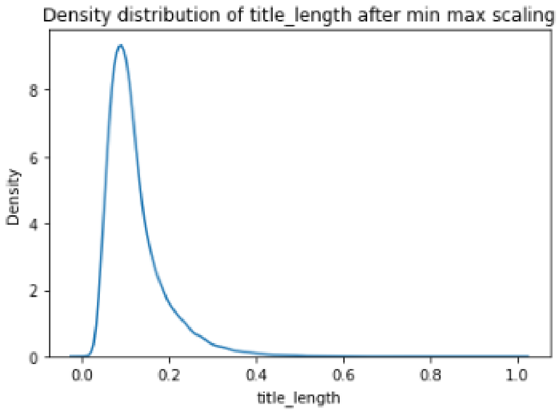
The distribution is not gaussian therefore we end up to apply *min-max scaling* in order to scale values in the range [0,1].

In [65]:
```
X_train_minmaxscaled, X_val_minmaxscaled, X_test_minmaxscaled = MinMaxScaling(X_train, X_val, X_test, ['title_length'])
```

In [66]:
```
sns.kdeplot(X_train_minmaxscaled.title_length)
plt.title("Density distribution of title_length after min max scaling")
plt.savefig(os.path.join(image_path, "after_minmaxscaled_title_lenght.png"),facecolor='white', transparent=False, bbox_incl
plt.close()
```

In [67]:
```
images = [os.path.join(image_path, "initial_title_lenght.png"), os.path.join(image_path, "after_minmaxscaled_title_lenght.p
showImagesHorizontally(images)
```



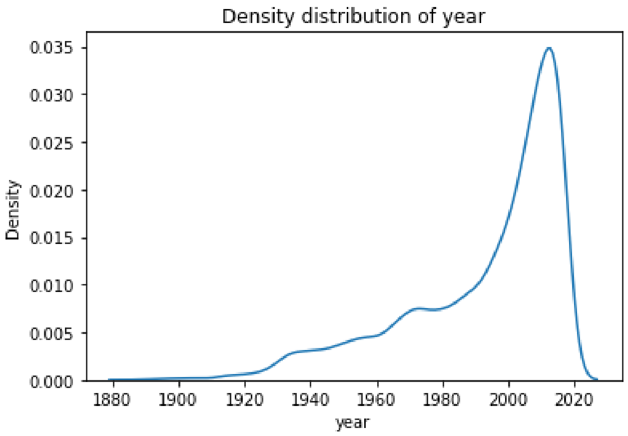


So the min max scaling changes only the scale of data but not their distribution.

year

In [68]:
```
sns.kdeplot(X_train['year'])
plt.title("Density distribution of year")
```

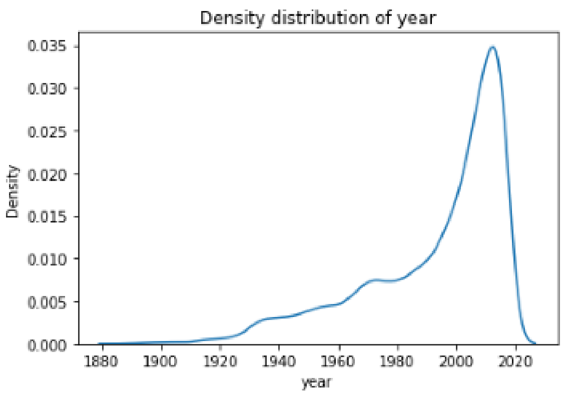
```
plt.savefig(os.path.join(image_path, "initial_year.png"), facecolor='white', transparent=False, bbox_inches='tight')
```

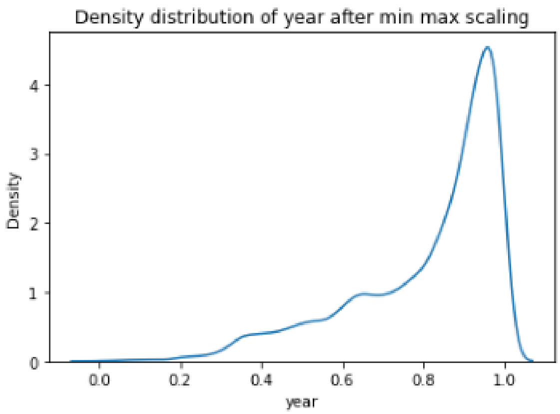


In [69]:
```
X_train_minmaxscaled, X_val_minmaxscaled, X_test_minmaxscaled = MinMaxScaling(X_train, X_val, X_test, ['year'])
```

In [70]:
```
sns.kdeplot(X_train_minmaxscaled['year'])
plt.title("Density distribution of year after min max scaling")
plt.savefig(os.path.join(image_path, "after_minmaxscaled_year.png"), facecolor='white', transparent=False, bbox_inches='ti
plt.close()
```

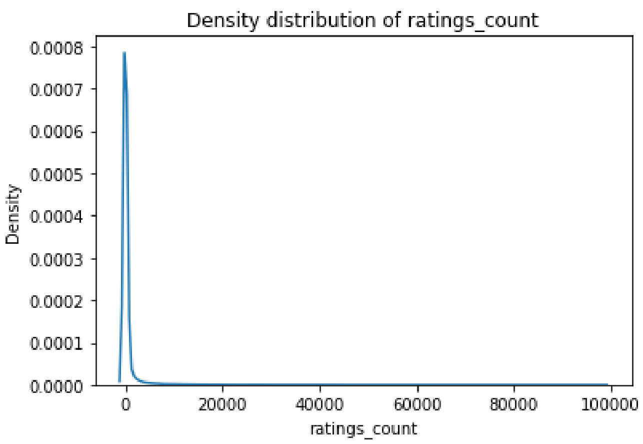
In [71]:
```
images = [os.path.join(image_path, "initial_year.png"), os.path.join(image_path, "after_minmaxscaled_year.png")]
showImagesHorizontally(images)
```





ratings_count

In [72]:
```
sns.kdeplot(train_ratings_count)
plt.title("Density distribution of ratings_count")
plt.savefig(os.path.join(image_path, "initial_ratings_count.png"), facecolor='white', transparent=False, bbox_inches='tight
```



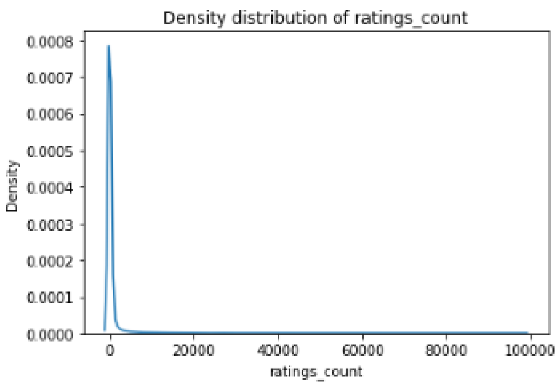
In [73]:
```
train_ratings_count_minmaxscaled, val_ratings_count_minmaxscaled, test_ratings_count_minmaxscaled = \
    MinMaxScaling(train_ratings_count.to_frame(), val_ratings_count.to_frame(), test_ratings_count.to_frame(), ['ratings_co
```

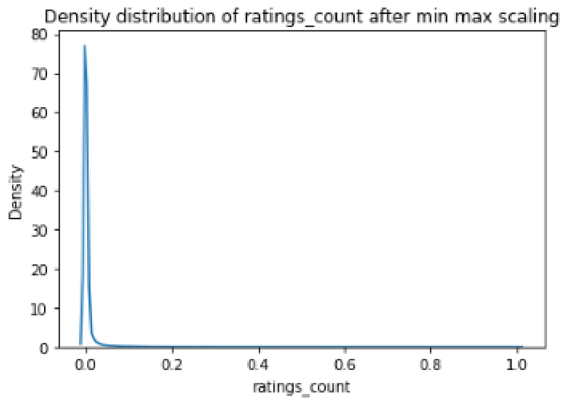
In [74]:
```
train_ratings_count_minmaxscaled = train_ratings_count_minmaxscaled.iloc[:,0]
sns.kdeplot(train_ratings_count_minmaxscaled)
plt.title("Density distribution of ratings_count after min max scaling")
```

```
plt.savefig(os.path.join(image_path, "after_minmaxscaled_ratings_count.png"), facecolor='white', transparent=False, bbox_i
plt.close()
```

In [75]:

```
images = [os.path.join(image_path, "initial_ratings_count.png"), os.path.join(image_path, "after_minmaxscaled_ratings_count
showImagesHorizontally(images)
```





Evaluating normalization

We define a function to normalize at a specified order the rows of a dataframe.

In [76]:

```
def normalization(X_train, ord):
        X_train_normalized = X_train.copy()
        X_train_normalized.iloc[:,:] = normalize(X_train, norm=ord)

        # alternative code in slides
        # x_norm2 = np.linalg.norm(x, ord=2)
        # x_normalized = x / x_norm2

        return X_train_normalized
```

L2 normalization is applied to each observation so that each row has a unit norm. Unit norm with L2 means that the sum of the squared elements is equal to 1.

In [77]:

```
X_train_normalized_l2 = normalization(X_train, 'l2')
```

In [78]:

```
X_train_normalized_l1 = normalization(X_train, 'l1')
```

In [79]:

```
X_train_normalized_lmax = normalization(X_train, 'max')
```

In [80]:

```
X_train_normalized_SSRN = X_train.copy()
X_train_normalized_SSRN.iloc[:,:] = np.sign(X_train)*np.sqrt(np.abs(X_train))
```

In [81]:

```
X_train_minmaxscaled_normalized_SSRN = X_train_minmaxscaled.copy()
X_train_minmaxscaled_normalized_SSRN.iloc[:,:] = np.sign(X_train_minmaxscaled)*np.sqrt(np.abs(X_train_minmaxscaled))
```

In [82]:

```
showTablesHorizontally(
    [
        X_train_normalized_l2.iloc[:,:4].describe().loc[['mean', 'min', 'max']],
        X_train_normalized_l1.iloc[:,:4].describe().loc[['mean', 'min', 'max']],
        X_train_normalized_lmax.iloc[:,:4].describe().loc[['mean', 'min', 'max']],
        X_train_minmaxscaled.iloc[:,:4].describe().loc[['mean', 'min', 'max']],
        X_train_normalized_SSRN.iloc[:,:4].describe().loc[['mean', 'min', 'max']],
        X_train_minmaxscaled_normalized_SSRN.iloc[:,:4].describe().loc[['mean', 'min', 'max']]
    ],
    [
        "X_train_normalized_l2",
        "X_train_normalized_l1",
        "X_train_normalized_lmax",
        "X_train_minmaxscaled",
        "X_train_normalized_SSRN",
        "X_train_minmaxscaled_normalized_SSRN"
    ])
```

X_train_normalized_l2

	title_length	year	Action	Adventure
mean	0.013152	0.999891	0.000066	0.000038
min	0.001498	0.995475	0.000000	0.000000
max	0.095020	0.999999	0.000527	0.000526

X_train_normalized_l1

	title_length	year	Action	Adventure
mean	0.012726	0.970397	0.000064	0.000036
min	0.001495	0.845371	0.000000	0.000000
max	0.087016	0.997509	0.000522	0.000517

X_train_normalized_lmax

	title_length	year	Action	Adventure
mean	0.013154	1.000000	0.000066	0.000038
min	0.001499	1.000000	0.000000	0.000000
max	0.095452	1.000000	0.000527	0.000526

X_train_minmaxscaled

	title_length	year	Action	Adventure
mean	26.203877	0.807197	0.132469	0.075453
min	3.000000	0.000000	0.000000	0.000000
max	191.000000	1.000000	1.000000	1.000000

X_train_normalized_SSRN

	title_length	year	Action	Adventure
mean	4.996935	44.641494	0.132469	0.075453
min	1.732051	43.451122	0.000000	0.000000
max	13.820275	44.922155	1.000000	1.000000

X_train_minmaxscaled_normalized_SSRN

	title_length	year	Action	Adventure
mean	4.996935	0.891019	0.132469	0.075453
min	1.732051	0.000000	0.000000	0.000000
max	13.820275	1.000000	1.000000	1.000000

Dimensionality reduction

In [83]:

```
lda = LinearDiscriminantAnalysis(solver='eigen')
lda.fit(X_train_minmaxscaled, y_train_categorical)
```

Out[83]:

```
LinearDiscriminantAnalysis(solver='eigen')
```

In [84]:

```
print(lda.explained_variance_ratio_)
```

```
[0.70935489 0.18105996 0.08829214 0.02129301]
```

Kaiser Method or Variance Explained Cumulative Plot

In [85]:

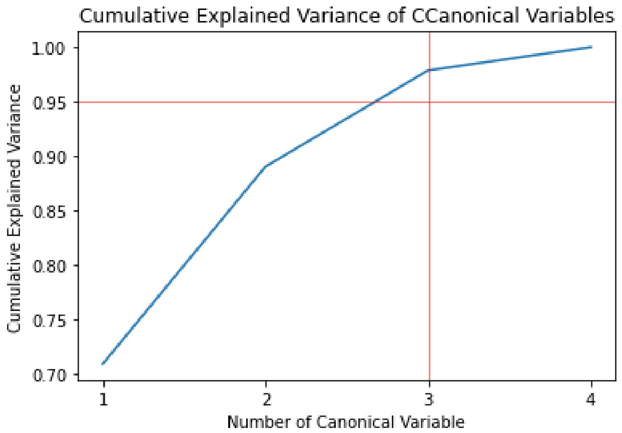
```
s = 0
nr_canonical_variables = 1
for comp in lda.explained_variance_ratio_:
        s += comp
        if s > 0.95:
                break
        nr_canonical_variables += 1

nr_canonical_variables
```

Out[85]:

```
3
```

In [86]:

```
ax = figure().gca()
ax.plot(range(1, len(lda.explained_variance_ratio_) + 1), np.cumsum(lda.explained_variance_ratio_))
ax.xaxis.set_major_locator(MaxNLocator(integer=True))

plt.xlabel('Number of Canonical Variable')
plt.ylabel('Cumulative Explained Variance')
plt.title('Cumulative Explained Variance of CCanonical Variables')

plt.axvline(x=3, linewidth=1, color='r', alpha=0.5)
plt.axhline(y=0.95, linewidth=1, color='r', alpha=0.5)
show()
```



In [87]:

```python
X_train_minmaxscaled_reduced = lda.transform(X_train_minmaxscaled)
```

In [88]:

```python
X_train_minmaxscaled_reduced = X_train_minmaxscaled_reduced[:,:nr_canonical_variables]
```

In [89]:

```python
X_train_minmaxscaled_reduced = pd.DataFrame(
    X_train_minmaxscaled_reduced,
    columns = [f"LD{i}" for i in range(1, X_train_minmaxscaled_reduced.shape[1] + 1)])
```

In [90]:

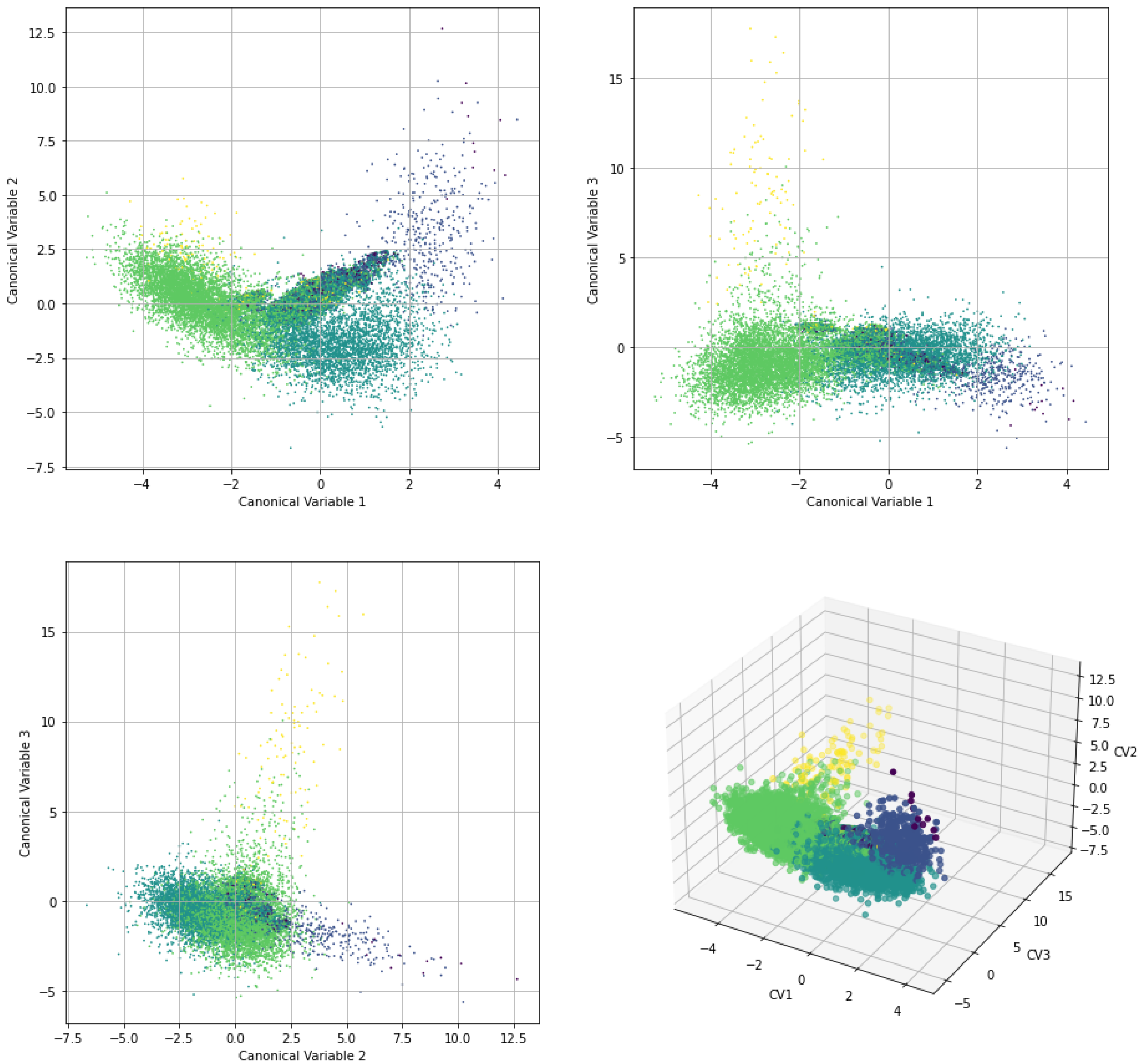
```python
X_val_minmaxscaled_reduced = pd.DataFrame(
    lda.transform(X_val_minmaxscaled)[:,:nr_canonical_variables],
    columns = [f"LD{i}" for i in range(1, X_train_minmaxscaled_reduced.shape[1] + 1)])
X_test_minmaxscaled_reduced = pd.DataFrame(
    lda.transform(X_test_minmaxscaled)[:,:nr_canonical_variables],
    columns = [f"LD{i}" for i in range(1, X_train_minmaxscaled_reduced.shape[1] + 1)])
```

In [91]:

```python
def myplot(n_cv1, n_cv2):
    cv1 = f"LD{n_cv1}"
    cv2 = f"LD{n_cv2}"
    cv1 = X_train_minmaxscaled_reduced[cv1]
    cv2 = X_train_minmaxscaled_reduced[cv2]

    plt.scatter(cv1 ,cv2, c = y_train_categorical, s = 0.5)

    plt.xlabel(f"Canonical Variable {n_cv1}")
    plt.ylabel(f"Canonical Variable {n_cv2}")
    plt.grid()

fig = plt.figure(figsize=(15,15))
plt.subplot(2, 2, 1)
myplot(1, 2)

plt.subplot(2, 2, 2)
myplot(1, 3)

plt.subplot(2, 2, 3)
myplot(2, 3)

ax = fig.add_subplot(2, 2, 4, projection='3d')
zdata = X_train_minmaxscaled_reduced.LD2
ydata = X_train_minmaxscaled_reduced.LD3
xdata = X_train_minmaxscaled_reduced.LD1
ax.set_xlabel('CV1')
ax.set_ylabel('CV3')
ax.set_zlabel('CV2')
ax.scatter3D(xdata, ydata, zdata, c=y_train_categorical)
```

Out[91]:

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x273809c5b70>
```



In [92]:

```
fig = go.Figure(data=[go.Scatter3d(
    x=X_train_minmaxscaled_reduced.LD1,
    y=X_train_minmaxscaled_reduced.LD3,
    z=X_train_minmaxscaled_reduced.LD2,
    mode='markers',
    marker=dict(
        size=2,
        color=y_train_categorical,                # set color to an array/list of desired values
        colorscale='Viridis',   # choose a colorscale
        opacity=0.8
    )
)])

fig.update_layout(
    margin=dict(l=0, r=0, b=0, t=0),
    scene = dict(
                xaxis_title='CV1',
                yaxis_title='CV3',
                zaxis_title='CV2'))
fig.show()
```

We define a function that implement all the above operations in order to call it during the next fine-tuning of hyperparameters.

In [ ]:

```
def LDA(X_train, X_val, X_test, y, solver="eigen", debug=False):
        lda = LinearDiscriminantAnalysis(solver)
        lda.fit(X_train, y)

        #Kaiser Method or Variance Explained Cumulative Plot
        s = 0
        nr_canonical_variables = 1
        for comp in lda.explained_variance_ratio_:
```

```
            s += comp
            if s < 0.95:
                nr_canonical_variables += 1

    if debug:
        print(lda.explained_variance_ratio_)
        print(f"get first {nr_canonical_variables} eigenvalues")

    X_train = lda.transform(X_train)
    X_train = X_train[:,:nr_canonical_variables]
    X_train = pd.DataFrame(X_train, columns = [f"LD{i}" for i in range(1, X_train.shape[1] + 1)])

    X_val = pd.DataFrame(lda.transform(X_val)[:,:nr_canonical_variables], columns = [f"LD{i}" for i in range(1, X_trai
    X_test = pd.DataFrame(lda.transform(X_test)[:,:nr_canonical_variables], columns = [f"LD{i}" for i in range(1, X_tra

    return X_train, X_val, X_test
```

Balancing Training Set

Evaluating imbalance

We evaluate to which extent bins are balanced.

In [77]:

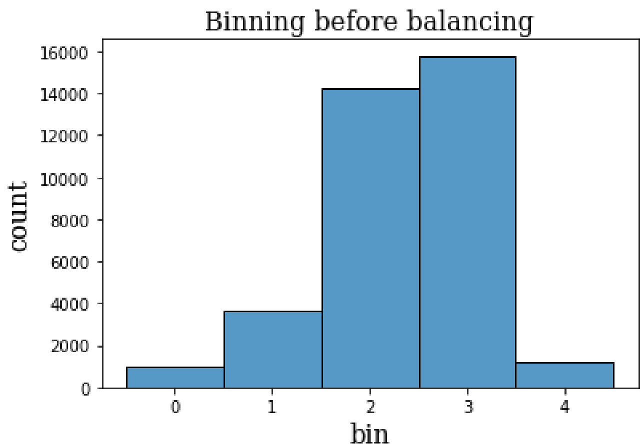
```
df_train.bin_y.value_counts()
```

Out[77]:

```
3    15797
2    14235
1     3672
4     1187
0      959
Name: bin_y, dtype: int64
```

In [78]:

```
sns.histplot(df_train.bin_y, discrete=True)
plt.xlabel('bin', fontdict=font_labels)
plt.ylabel('count', fontdict=font_labels)
plt.title("Binning before balancing", fontdict=font_labels)
plt.savefig(os.path.join(image_path, "initial_balance.png"), facecolor='white', transparent=False, bbox_inches='tight')
plt.show()
```



The bins are strongly unbalanced.

Managing imbalance

SMOTE

Since the training set is strongly unbalanced, the synthetic oversampling of the minority classes to the majority class is not recommended, therefore we decide to oversample until a specified lower bound through the following function.

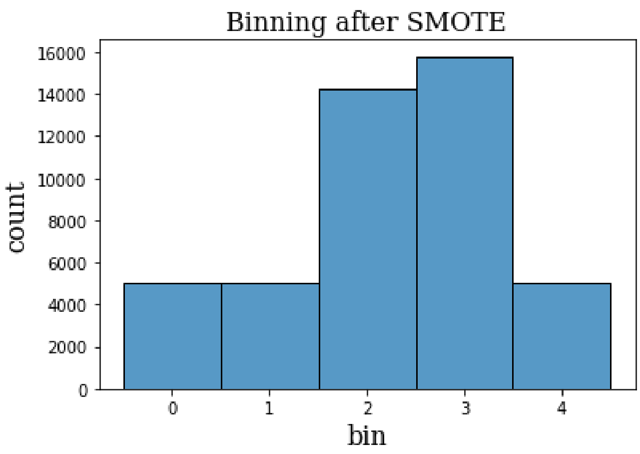
In [79]:

```
def balancing(df_train, lower_bound, remove_duplicates=True):

    bins_count = df_train.bin_y.value_counts()
    for i in range(len(bins_count)):
        if bins_count[i] <= lower_bound:
            bins_count[i] = lower_bound

    bin_sizes = bins_count.to_dict()

    sm = SMOTE(random_state=43, sampling_strategy=bin_sizes)
    df_train_balanced, _ = sm.fit_resample(df_train, df_train['bin_y'])

    if remove_duplicates:
        df_train_balanced.drop_duplicates(inplace=True)

    return df_train_balanced.reset_index(drop=True)
```

As an example, we decide to balance the training set by oversampling the minority classes as if the majority class has a cardinality of 5000 rows.

In [80]:
```
df_train_SMOTE = balancing(df_train, 5000, remove_duplicates=False)
```

In [81]:
```
sns.histplot(df_train_SMOTE.bin_y, discrete=True)
plt.xlabel('bin', fontdict=font_labels)
plt.ylabel('count', fontdict=font_labels)
plt.title("Binning after SMOTE", fontdict=font_labels)
plt.savefig(os.path.join(image_path, "after_SMOTE_balancing.png"),facecolor='white', transparent=False, bbox_inches='tight
plt.show()
```



After balancing, there could be new duplicates due to synthetic oversampling.

In [82]:
```
df_train_SMOTE.duplicated().sum()
```

Out[82]:
```
1323
```

In order to drop them, we call the `balancing` function by setting to True the `remove_duplicate` parameter.

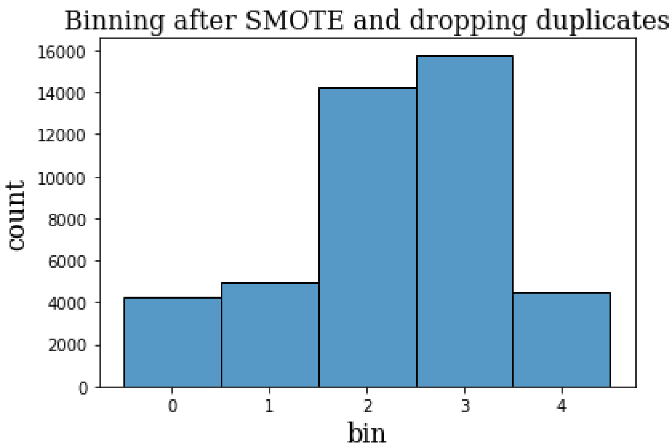
In [83]:
```
df_train_SMOTE = balancing(df_train, 5000, remove_duplicates=True)
```

As a matter of fact, there are no duplicates.

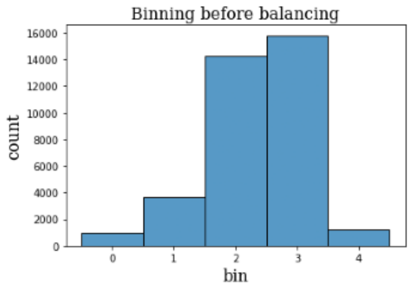
In [84]:
```
df_train_SMOTE.duplicated().sum()
```

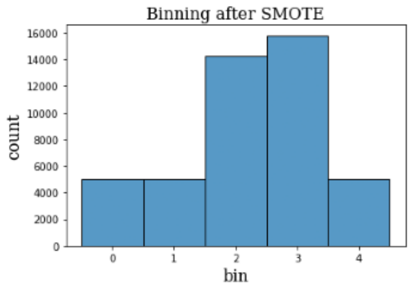
Out[84]:
```
0
```

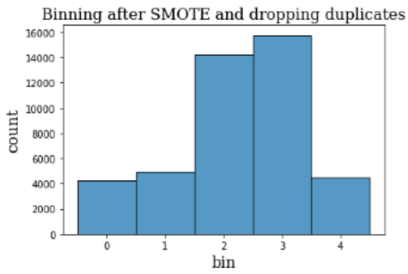
In [85]:
```
sns.histplot(df_train_SMOTE.bin_y, discrete=True)
plt.xlabel('bin', fontdict=font_labels)
plt.ylabel('count', fontdict=font_labels)
plt.title("Binning after SMOTE and dropping duplicates", fontdict=font_labels)
plt.savefig(os.path.join(image_path, "after_SMOTE_balancing_and_dropping_duplicates.png"),facecolor='white', transparent=Fa
plt.show()
```



In [86]:
```
images = [os.path.join(image_path, "initial_balance.png"), os.path.join(image_path, "after_SMOTE_balancing.png"), os.path.j
showImagesHorizontally(images)
```







Splitting into balanced subsets

The balancing realized through SMOTE ends up to thrown away a large amount of information: even in the case of setting a lower bound rather than oversampling until the majority class, the information loss is considerable.\ Thus, we decide to split the imbalanced training set into different balanced subsets through two functions:

- the first one, `generateSets`, takes as input `df_class_c`, namely the subset of the training set whose samples belong to the class `c` (i.e. the `bin_y == c`), and returns a list of `n_samples` subsets charachterized by `size` as cardinality:
  - in detail, if the cardinality of `df_class_c` is not sufficiently large to split it into `n_samples` of size `size`, we perform multiple random undersampling by `sample` function.

In [87]:

```
def generateSets(df_class_c, n_samples, size):
        samples = []
        df_class_c = shuffle(df_class_c, random_state=43).reset_index(drop=True)

        if(len(df_class_c) >= n_samples * size):
                for s in range(n_samples):
                        start = s * size
                        end = start + size
                        samples.append(df_class_c.iloc[start:end])
        else:
                for s in range(n_samples):
                        samples.append(df_class_c.sample(size, replace=False, ignore_index=True, random_state=43))

        return samples
```

- the second one, `RandomSubSets`, calls `generateSets` over each `df_class_c` obtained by passing over the various classes: its output is a list of `n_samples` subsets obtained by trasversely joining the subsets relative to the different classes.

In [88]:

```
def RandomSubSets(df, size, n_samples):
        df_samples = [pd.DataFrame(columns=df.columns) for _ in range(n_samples)]
        for c in df.bin_y.unique():
                df_class_c = df[df.bin_y == c]
                df_class_sets = generateSets(df_class_c, n_samples, size)

                for i in range(n_samples):
                        df_samples[i] = df_samples[i].append(df_class_sets[i], ignore_index=True)

        return df_samples
```

In this way, we can do the random undersampling until a specified size ($2\,000$, for instance) and generate a specified number of balanced subsets (7).

In [89]:

```
samples = RandomSubSets(df_train_SMOTE, 2000, 7)
```

Consequently, every balanced subset will have a cardinality of $size * n\_classes$: within the example, we have $2000 * 5 = 10\,000$.

In [90]:

```
[len(x) for x in samples]
```

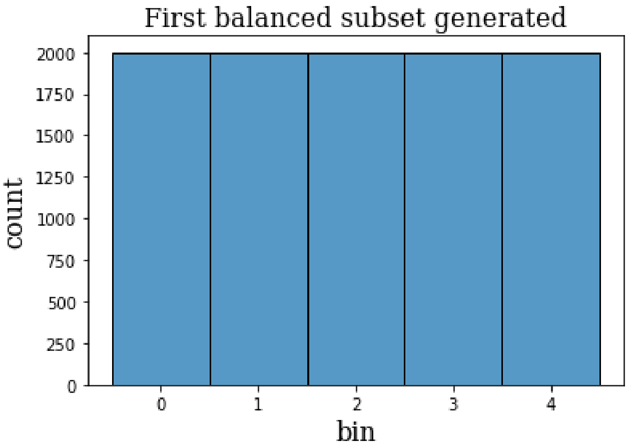
Out[90]:

```
[10000, 10000, 10000, 10000, 10000, 10000, 10000]
```

We plot the distribution over classes of the first subsets generated through the above technique.

In [91]:

```
sns.histplot(samples[0].bin_y, discrete=True)
plt.xlabel('bin', fontdict=font_labels)
plt.ylabel('count', fontdict=font_labels)
plt.title("First balanced subset generated", fontdict=font_labels)
plt.show()
```



DATA MODELING

We adopt the second mechanism to manage the imbalance of the training set: in particular, we implement the *bagging* method, namely we fit every technique on the different balanced subsets and then perform a majority voting over the various outputs. \ Depending on the number of balanced subsets, each of them has a certain size computed by dividing the cardinality of the majority class in the total training set by the number of desidered subsets: in this way, we assure that with regard to the majority class, the new subsets are obtained through slicing, namely without the creation of synthetic samples by SMOTE.

In [92]:

```
sizes = []
for i in range(9):
        size = df_train.bin_y.value_counts().max() // (i+1)
        print(f"{i+1} balanced subsets of cardinality equal to {size}")
        sizes.append(size)
```

```
1 balanced subsets of cardinality equal to 15797
2 balanced subsets of cardinality equal to 7898
3 balanced subsets of cardinality equal to 5265
4 balanced subsets of cardinality equal to 3949
5 balanced subsets of cardinality equal to 3159
6 balanced subsets of cardinality equal to 2632
7 balanced subsets of cardinality equal to 2256
8 balanced subsets of cardinality equal to 1974
9 balanced subsets of cardinality equal to 1755
```

SVM

Hyperparameters

In [93]:

```
nr_values = 4
C_range = np.geomspace(0.1, 100, nr_values)
gamma_range = np.geomspace(0.1, 100, nr_values)

nr_configurations = nr_values*nr_values*len(sizes_subsets)
```

In [94]:

```
print(*C_range, sep="\t")
```

```
0.1	1.0	10.0	100.0
```

In [95]:

```
print(*gamma_range, sep="\t")
```

```
0.1	1.0	10.0	100.0
```

RBF kernel

Hyperparameters optimization

In [ ]:

```
results = pd.DataFrame(columns=['size', 'samples', 'C', 'gamma', 'loss_ensemble'])
config = 0

for size in sizes:
        print("**********BALANCING DATASET**********")
        df_train_SMOTE = balancing(df_train, size)
        min_size = df_train_SMOTE.bin_y.value_counts().min()
        n_sample = int(df_train.bin_y.value_counts().max()/min_size)
        df_trains = RandomSubSets(df_train_SMOTE, min_size, n_samples=n_sample)

        for c in C_range:
                for gamma in gamma_range:
                        y_val_preds = []
```

```
                config += 1
                print(f"***************************************** {config} out of {nr_configurations} params' configu

                print("***********STARTING BAGGING***********")
                for n in range(len(df_trains)):
                        print(f"****{n+1}° FIT su {len(df_trains)}° sample del train****")

                        print("PRE-PROCESSING --> split_XYweights")
                        X_train, weights, y_train, _ = split_XYweights(df_trains[n])
                        X_val, _, y_val, _ = split_XYweights(df_val)
                        X_test, _, y_test, _ = split_XYweights(df_test)

                        print("PRE-PROCESSING --> MinMaxScaling")
                        X_train, X_val, X_test = MinMaxScaling(X_train, X_val, X_test, ['title_length','year'])

                        print("PRE-PROCESSING --> LDA")
                        X_train, X_val, X_test = LDA(X_train, X_val, X_test, y_train)

                        print("FITTING & PREDICTING")
                        svc = svm.SVC(kernel="rbf", C=c, gamma=gamma)
                        svc.fit(X_train, y_train)

                        y_val_pred = svc.predict(X_val).tolist()
                        y_val_preds.append(y_val_pred)

                        error = zero_one_loss(y_val, y_val_pred)
                        print(f"LOSS --> {error}")

                print("***********ENDING BAGGING***********")

                nr_predictions = len(y_val_preds[0])
                y_val_pred_voted = []
                print("VOTING")
                for prediction in range(nr_predictions):
                        y_val_pred_voted.append(Counter([item[prediction] for item in y_val_preds]).most_common(1)

                loss_ensemble = zero_one_loss(y_val, y_val_pred_voted)
                print(f"LOSS ENSEMBLE (C: {c}, gamma: {gamma}) --> {loss_ensemble}\n\n")

                results = results.append({
                        'size': size,
                        'samples': n_sample,
                        'C': c,
                        'gamma': gamma,
                        'loss_ensemble': loss_ensemble
                }, ignore_index=True)
```

We store the results.

In [ ]:

```
results.to_csv(os.path.join(current_path, tuning_path,'SVC_rbf.csv'), index=False)
```

We load the results.

In [100…

```
results = pd.read_csv(os.path.join(current_path, tuning_path,'SVC_rbf.csv'))
```

We select the best hyperparameters. It's that with a single balanced subset: we know that the voting with a single subset is pointless, however we mantain the same skeleton of code in order to a better readability and consistency.

In [101…

```
results.loc[results['loss_ensemble'] == results['loss_ensemble'].min()]
```

Out[101…

	size	samples	C	gamma	loss_ensemble
134	15797.0	1.0	1.0	10.0	0.551707

The resulting loss ensemble is 0.551707

Test with best hyperparameters

In [102…

```
size = 15797
c = 1
gamma = 1
```

In [103…

```
df_train_SMOTE = balancing(df_train, size)
min_size = df_train_SMOTE.bin_y.value_counts().min()
n_sample = int(df_train.bin_y.value_counts().max()/min_size)
df_trains = RandomSubSets(df_train_SMOTE, min_size, n_samples=n_sample)

y_test_preds = []
```

```
print("***********STARTING BAGGING***********")
for n in range(len(df_trains)):
        print(f"****{n+1}° FIT su {len(df_trains)}° sample del train****")

        print("PRE-PROCESSING --> split_XYweights")
        X_train, weights, y_train, _ = split_XYweights(df_trains[n])
        X_val, _, y_val, _ = split_XYweights(df_val)
        X_test, _, y_test, _ = split_XYweights(df_test)

        print("PRE-PROCESSING --> MinMaxScaling")
        X_train, X_val, X_test = MinMaxScaling(X_train, X_val, X_test, ['title_length','year'])

        print("PRE-PROCESSING --> LDA")
        X_train, X_val, X_test = LDA(X_train, X_val, X_test, y_train)

        print("FITTING & PREDICTING")
        svc = svm.SVC(kernel="rbf", C=c, gamma=gamma)
        svc.fit(X_train, y_train)

        y_test_pred = svc.predict(X_test).tolist()
        y_test_preds.append(y_test_pred)

        error = zero_one_loss(y_test, y_test_pred)
        print(f"LOSS --> {error}")

print("***********ENDING BAGGING***********")

nr_predictions = len(y_test_preds[0])
y_test_pred_voted = []
print("VOTING")
for prediction in range(nr_predictions):
        y_test_pred_voted.append(Counter([item[prediction] for item in y_test_preds]).most_common(1)[0][0])

loss_ensemble = zero_one_loss(y_test, y_test_pred_voted)
print(f"LOSS ENSEMBLE (C: {c}, gamma: {gamma}) --> {loss_ensemble}\n\n")
```

```
***********STARTING BAGGING***********
****1° FIT su 1° sample del train****
PRE-PROCESSING --> split_XYweights
PRE-PROCESSING --> MinMaxScaling
PRE-PROCESSING --> LDA
FITTING & PREDICTING
LOSS --> 0.5414198212671955
***********ENDING BAGGING***********
VOTING
LOSS ENSEMBLE (C: 1, gamma: 1) --> 0.5414198212671955
```

Results

The resulting loss ensemble is 0.5414198212671955.

In [104…
```
loss_ensemble
```

Out[104…
```
0.5414198212671955
```

Support vectors

In [313…
```
svc.support_vectors_.shape
```

Out[313…
```
(50920, 3)
```

In [263…
```
svc.support_vectors_
```

Out[263…
```
array([[-0.00372332,  0.72523788, -0.30330778],
       [ 0.56792718,  0.02382133,  0.18777365],
       [ 1.19442379, -0.85690044,  0.36897711],
       ...,
       [-0.03651834,  0.32903518,  0.04003358],
       [-0.02170164,  0.34863159, -0.11949132],
       [ 1.62559149,  0.10901064, -0.10343894]])
```

In [106…
```
fig = go.Figure(data=[go.Scatter3d(
    x=svc.support_vectors_[:,0],
    y=svc.support_vectors_[:,1],
    z=svc.support_vectors_[:,2],
    mode='markers',
    marker=dict(
        size=4,
        color=y_train_categorical,                # set color to an array/list of desired values
        colorscale='Viridis',   # choose a colorscale
```

```
        opacity=0.8
    )
)])

fig.update_layout(
    margin=dict(l=0, r=0, b=0, t=0),
    scene = dict(
                    xaxis_title='CV1',
                    yaxis_title='CV3',
                    zaxis_title='CV2'))
fig.show()
```

Custom kernel

Feature Engineering

We want to measure distances between samples by separately considering tags and genres: by the way, we create two new columns:

- `genres` which collapses in a list the values corresponding to genres' columns;
- `tags` which collapses in a list the values corresponding to tags' columns.

Regarding genres, we adopt the hamming distance: between two list of equal length, it's the the number of positions at which the corresponding values are different. Regarding tags, title_length and year, we adopt the root mean squared error.

In [ ]:

```
def distance(a,b):
        d = np.sqrt(np.square(np.subtract(a[0:2], b[0:2])).sum())     # distanze element wise tra year, title_length
        d += np.bitwise_xor(a[2], b[2]).sum()/len(a[2]) # hamming
        d += np.sqrt(np.square(np.subtract(a[3], b[3])).sum())
        return d
```

Hyperparameters optimization

We do two fine-tuning regarding SVC with custom kernel: however, the computing of the distance matrix is highly resource and time consuming, therefore we stop after 14 days the fine-tuning and gather the results obtained until that moment.\ The first one presents 7 balanced subsets of size equal to 2000.

In [ ]:

```
size_C_range = 8
size_gamma_range = 8
nr_configurations = size_gamma_range*size_C_range

C_range = np.logspace(-2, 5, size_C_range)
gamma_range = np.logspace(-5, 2, size_gamma_range)
```

In [ ]:

```
results = pd.DataFrame(columns=['C', 'gamma', 'loss_ensemble'])
config = 0

df_train = balancing(df_train, 2000)

df_train, df_val, df_test = MinMaxScaling(df_train, df_val, df_test, ["year", "title_length"])

print("CHANGING DATAFRAME FOR HAMMING")
df_train_ham = df_train.loc[:,["year", "title_length", "ratings_count", "bin_y", "rating_mean"]]
df_train_ham['genres'] = df_train.iloc[:,2:21].values.tolist()
df_train_ham['tags'] = df_train.iloc[:,22:-3].values.tolist()

df_val_ham = df_val.loc[:,["year", "title_length", "ratings_count", "bin_y", "rating_mean"]]
df_val_ham['genres'] = df_val.iloc[:,2:21].values.tolist()
df_val_ham['tags'] = df_val.iloc[:,22:-3].values.tolist()

df_test_ham = df_test.loc[:,["year", "title_length", "ratings_count", "bin_y", "rating_mean"]]
df_test_ham['genres'] = df_test.iloc[:,2:21].values.tolist()
df_test_ham['tags'] = df_test.iloc[:,22:-3].values.tolist()

min_size = df_train_ham.bin_y.value_counts().min()
n_sample = int(df_train_ham.bin_y.value_counts().max()/min_size)
df_trains = RandomSubSets(df_train_ham, min_size, n_samples=n_sample) # n_samples a 12 se size == min_bin_cardinality, 7 se

for c in C_range:
        for gamma in gamma_range:
                y_val_preds = []
                config += 1
                print(f"*************************************** {config} out of {nr_configurations} params' configurations 

                print("***********STARTING BAGGING***********")
                for n in range(len(df_trains)):
                        print(f"****{n+1}° FIT su {n+1}° sample del train****")

                        print("PRE-PROCESSING --> split_XYweights")
                        X_train, weights, y_train, _ = split_XYweights(df_trains[n])
```

```
            X_val, _, y_val, _ = split_XYweights(df_val_ham)
            X_test, _, y_test, _ = split_XYweights(df_test_ham)

            print("FITTING & PREDICTING")
            train_distances = cdist(X_train.values, X_train.values, lambda a,b: distance(a,b))
            svc = svm.SVC(kernel="precomputed", C=c, gamma=gamma)
            svc.fit(train_distances, y_train)

            val_distances = cdist(X_val.values, X_train.values, lambda a,b: distance(a,b))
            y_val_pred = svc.predict(val_distances).tolist()
            y_val_preds.append(y_val_pred)

            error = zero_one_loss(y_val, y_val_pred)
            print(f"LOSS --> {error}")
    print("***********ENDING BAGGING***********")

    nr_predictions = len(y_val_preds[0])
    y_val_pred_voted = []
    print("VOTING")
    for prediction in range(nr_predictions):
            y_val_pred_voted.append(Counter([item[prediction] for item in y_val_preds]).most_common(1)[0][0])

    loss_ensemble = zero_one_loss(y_val, y_val_pred_voted)
    print(f"LOSS ENSEMBLE (C: {c}, gamma: {gamma}) --> {loss_ensemble}\n\n")

    results = results.append({
            'C': c,
            'gamma': gamma,
            'loss_ensemble': loss_ensemble
    }, ignore_index=True)
```

In [132…

```
results.to_csv(os.path.join(current_path, tuning_path,'SVC_ou_custom_kernel.csv'), index=False)
```

In [136…

```
pd.read_csv(os.path.join(current_path, tuning_path,'SVC_ou_custom_kernel.csv'))
```

Out[136…

	C	gamma	loss_ensemble
0	0.01	0.00001	0.826054
1	0.01	0.00010	0.826054
2	0.01	0.00100	0.826054
3	0.01	0.01000	0.826054
4	0.01	0.10000	0.826054
5	0.01	1.00000	0.826054
6	0.01	10.00000	0.826054
7	0.01	100.00000	0.826054
8	0.10	0.00001	0.941265

Losses are generally high and most of them are equal to 0.826054: therefore, we skip this fine-tuning.

The second fine-tuning presents 12 balanced subsets of size equal to the cardinality of the minority class, thus it involves solely a random under sampling.

In [ ]:

```
min_bin_cardinality = df_train.bin_y.value_counts().min()
df_trains = RandomSubSampling(df_train, min_bin_cardinality, n_samples=12)

size_C_range = 8
size_gamma_range = 8
nr_configurations = size_gamma_range*size_C_range

C_range = np.logspace(-2, 5, size_C_range)
gamma_range = np.logspace(-5, 2, size_gamma_range)
```

In [ ]:

```
results = pd.DataFrame(columns=['C', 'gamma', 'loss_ensemble'])
config = 0

df_train = balancing(df_train, 2000)

df_train, df_val, df_test = MinMaxScaling(df_train, df_val, df_test, ["year", "title_length"])

print("CHANGING DATAFRAME FOR HAMMING")
df_train_ham = df_train.loc[:,["year", "title_length", "ratings_count", "bin_y", "rating_mean"]]
df_train_ham['genres'] = df_train.iloc[:,2:21].values.tolist()
df_train_ham['tags'] = df_train.iloc[:,22:-3].values.tolist()
```

```python
df_val_ham = df_val.loc[:,["year", "title_length", "ratings_count", "bin_y", "rating_mean"]]
df_val_ham['genres'] = df_val.iloc[:,2:21].values.tolist()
df_val_ham['tags'] = df_val.iloc[:,22:-3].values.tolist()

df_test_ham = df_test.loc[:,["year", "title_length", "ratings_count", "bin_y", "rating_mean"]]
df_test_ham['genres'] = df_test.iloc[:,2:21].values.tolist()
df_test_ham['tags'] = df_test.iloc[:,22:-3].values.tolist()

min_size = df_train_ham.bin_y.value_counts().min()
n_sample = int(df_train_ham.bin_y.value_counts().max()/min_size)
df_trains = RandomSubSets(df_train_ham, min_size, n_samples=n_sample) # n_samples a 12 se size == min_bin_cardinality, 7 se

for c in C_range:
        for gamma in gamma_range:
                y_val_preds = []
                config += 1
                print(f"************************************** {config} out of {nr_configurations} params' configurations 

                print("***********STARTING BAGGING***********")
                for n in range(len(df_trains)):
                        print(f"****{n+1}° FIT su {n+1}° sample del train****")

                        print("PRE-PROCESSING --> split_XYweights")
                        X_train, weights, y_train, _ = split_XYweights(df_trains[n])
                        X_val, _, y_val, _ = split_XYweights(df_val_ham)
                        X_test, _, y_test, _ = split_XYweights(df_test_ham)

                        print("FITTING & PREDICTING")
                        train_distances = cdist(X_train.values, X_train.values, lambda a,b: distance(a,b))
                        svc = svm.SVC(kernel="precomputed", C=c, gamma=gamma)
                        svc.fit(train_distances, y_train)

                        val_distances = cdist(X_val.values, X_train.values, lambda a,b: distance(a,b))
                        y_val_pred = svc.predict(val_distances).tolist()
                        y_val_preds.append(y_val_pred)

                        error = zero_one_loss(y_val, y_val_pred)
                        print(f"LOSS --> {error}")
                print("***********ENDING BAGGING***********")

                nr_predictions = len(y_val_preds[0])
                y_val_pred_voted = []
                print("VOTING")
                for prediction in range(nr_predictions):
                        y_val_pred_voted.append(Counter([item[prediction] for item in y_val_preds]).most_common(1)[0][0])

                loss_ensemble = zero_one_loss(y_val, y_val_pred_voted)
                print(f"LOSS ENSEMBLE (C: {c}, gamma: {gamma}) --> {loss_ensemble}\n\n")

                results = results.append({
                        'C': c,
                        'gamma': gamma,
                        'loss_ensemble': loss_ensemble
                }, ignore_index=True)
```

In [164…

```python
results.to_csv(os.path.join(current_path, tuning_path,'SVC_u_custom_kernel.csv'), index=False)
```

In [166…

```python
pd.read_csv(os.path.join(current_path, tuning_path,'SVC_u_custom_kernel.csv'))
```

Out[166…

	C	gamma	loss_ensemble
0	0.01	0.00001	0.736195
1	0.01	0.00010	0.736195
2	0.01	0.00100	0.736195
3	0.01	0.01000	0.736195
4	0.01	0.10000	0.736195
5	0.01	1.00000	0.736195
6	0.01	10.00000	0.736195
7	0.01	100.00000	0.736195
8	0.10	0.00001	0.927962
9	0.10	0.00010	0.927962
10	0.10	0.00100	0.927962
11	0.10	0.01000	0.927962
12	0.10	0.10000	0.927962

	C	gamma	loss_ensemble
13	0.10	1.00000	0.927962
14	0.10	10.00000	0.927962
15	0.10	100.00000	0.927962
16	1.00	0.00001	0.927460
17	1.00	0.00010	0.927460
18	1.00	0.00100	0.927460
19	1.00	0.01000	0.927460
20	1.00	0.10000	0.927460
21	1.00	1.00000	0.927460
22	1.00	10.00000	0.927460
23	1.00	100.00000	0.927460

It points out that the loss changes only when C changes: however, losses are still too high, so we skip this fine tuning.

Naive Bayes

Preprocessing: features discretization

In [ ]:
```
n_cats = 4
```

In [ ]:
```
df_val_discrete = df_val.copy()
df_val_discrete.iloc[:, 21:-3] = df_val_discrete.iloc[:, 21:-3].apply(lambda x: pd.cut(x, n_cats, labels=range(n_cats)), a
```

In [ ]:
```
df_test_discrete = df_test.copy()
df_test_discrete.iloc[:, 21:-3] = df_test_discrete.iloc[:, 21:-3].apply(lambda x: pd.cut(x, n_cats, labels=range(n_cats)),a
```

In [ ]:
```
df_train_discrete = df_train.copy()
df_train_discrete.iloc[:,21:-3] = df_train_discrete.iloc[:, 21:].apply(lambda x: pd.cut(x, n_cats, labels=range(n_cats)),a
```

Hyperparameters optimization

In [ ]:
```
sizes = [7898, 15797]
types = ["CategoricalNB", "GaussianNB", "QuadraticDiscriminantAnalysis"]
nr_configurations = len(sizes)*len(types)
```

In [ ]:
```
results = pd.DataFrame(columns=['size', 'sample', 'technique', 'loss_ensemble'])
config = 0

for size in sizes:
        print("***********BALANCING DATASET***********")
        df_train_SMOTE = balancing(df_train, size)
        min_size = df_train_SMOTE.bin_y.value_counts().min()
        n_sample = int(df_train.bin_y.value_counts().max()/min_size)
        df_trains = RandomSubSets(df_train_SMOTE, min_size, n_samples=n_sample) # n_samples a 12 se size == min_bin_cardin

        for t in types:
                y_val_preds = []
                config += 1
                print(f"**************************************** {config} out of {nr_configurations} params' configurations 

                print("***********STARTING BAGGING***********")
                for n in range(len(df_trains)):
                        print(f"****{n+1}° FIT su {len(df_trains)}° sample del train****")

                        print("PRE-PROCESSING --> split_XYweights")
                        X_train, weights, y_train, _ = split_XYweights(df_trains[n])
                        X_val, _, y_val, _ = split_XYweights(df_val)
                        X_test, _, y_test, _ = split_XYweights(df_test)

                        print("PRE-PROCESSING --> MinMaxScaling")
                        X_train, X_val, X_test = MinMaxScaling(X_train, X_val, X_test, ['title_length','year'])
                        print("PRE-PROCESSING --> LDA")
                        X_train, X_val, X_test = LDA(X_train, X_val, X_test, y_train)

                        if t == "CategoricalNB":
                                print("DISCRETIZATION for CategoricalNB")
```

```
                                n_cat_LD1 = int((X_train['LD1'].max() - X_train['LD1'].min()) * 2)
                                n_cat_LD2 = int((X_train['LD2'].max() - X_train['LD2'].min()) * 2)
                                n_cat_LD3 = int((X_train['LD3'].max() - X_train['LD3'].min()) * 2)
                                print(f"bins for LD1 are: {n_cat_LD1}")
                                print(f"bins for LD2 are: {n_cat_LD2}")
                                print(f"bins for LD3 are: {n_cat_LD3}")

                                X_train['LD1'] = pd.cut(X_train['LD1'], bins=n_cat_LD1, labels=range(n_cat_LD1))
                                X_train['LD2'] = pd.cut(X_train['LD2'], bins=n_cat_LD2, labels=range(n_cat_LD2))
                                X_train['LD3'] = pd.cut(X_train['LD3'], bins=n_cat_LD3, labels=range(n_cat_LD3))

                                X_val['LD1'] = pd.cut(X_val['LD1'], bins=n_cat_LD1, labels=range(n_cat_LD1))
                                X_val['LD2'] = pd.cut(X_val['LD2'], bins=n_cat_LD2, labels=range(n_cat_LD2))
                                X_val['LD3'] = pd.cut(X_val['LD3'], bins=n_cat_LD3, labels=range(n_cat_LD3))

                                X_test['LD1'] = pd.cut(X_test['LD1'], bins=n_cat_LD1, labels=range(n_cat_LD1))
                                X_test['LD2'] = pd.cut(X_test['LD2'], bins=n_cat_LD2, labels=range(n_cat_LD2))
                                X_test['LD3'] = pd.cut(X_test['LD3'], bins=n_cat_LD3, labels=range(n_cat_LD3))

                                clf = CategoricalNB()
                        elif t == "GaussianNB":
                                clf = GaussianNB()
                        else:
                                clf = QuadraticDiscriminantAnalysis()

                        print("FITTING & PREDICTING")
                        if t == "CategoricalNB":
                                #sample_weight is always the best choice
                                clf.fit(X_train, y_train, sample_weight=weights)
                        else:
                                clf.fit(X_train, y_train)

                        y_val_pred = clf.predict(X_val).tolist()
                        y_val_preds.append(y_val_pred)

                        error = zero_one_loss(y_val, y_val_pred)
                        print(f"LOSS --> {error}")
                print("***********ENDING BAGGING***********")

                nr_predictions = len(y_val_preds[0])
                y_val_pred_voted = []
                print("VOTING")
                for prediction in range(nr_predictions):
                        y_val_pred_voted.append(Counter([item[prediction] for item in y_val_preds]).most_common(1)[0][0])

                loss_ensemble = zero_one_loss(y_val, y_val_pred_voted)
                print(f"LOSS ENSEMBLE (size: {size}, samples: {n_sample} 'technique': {t}) --> {loss_ensemble}\n\n")

                results = results.append({
                        'size': size,
                        'sample': n_sample,
                        'technique': t,
                        'loss_ensemble': loss_ensemble
                }, ignore_index=True)
```

We store the results

In [ ]:

```
results.to_csv(os.path.join(current_path, tuning_path,'bayes.csv'), index=False)
```

We load the results

In [167…

```
results = pd.read_csv(os.path.join(current_path, tuning_path,'bayes.csv'))
```

we select the best hyperparametes

In [168…

```
results.loc[results['loss_ensemble'] == results['loss_ensemble'].min()]
```

Out[168…

	size	sample	technique	loss_ensemble
3	15797	1	CategoricalNB	0.475402

Test with best hyperparameters

In [169…

```
sizes = [15797]
types = ["CategoricalNB"]
nr_configurations = len(sizes)*len(types)
```

In [170…

```
results = pd.DataFrame(columns=['size', 'sample', 'technique', 'loss_ensemble'])
config = 0
```

```
for size in sizes:
        print("***********BALANCING DATASET***********")
        df_train_SMOTE = balancing(df_train, size)
        min_size = df_train_SMOTE.bin_y.value_counts().min()
        n_sample = int(df_train.bin_y.value_counts().max()/min_size)
        df_trains = RandomSubSets(df_train_SMOTE, min_size, n_samples=n_sample) # n_samples a 12 se size == min_bin_cardin

        for t in types:
                y_test_preds = []
                config += 1
                print(f"************************************** {config} out of {nr_configurations} params' configurations 

                print("***********STARTING BAGGING***********")
                for n in range(len(df_trains)):
                        print(f"****{n+1}° FIT su {len(df_trains)}° sample del train****")

                        print("PRE-PROCESSING --> split_XYweights")
                        X_train, weights, y_train, _ = split_XYweights(df_trains[n])
                        X_val, _, y_val, _ = split_XYweights(df_val)
                        X_test, _, y_test, _ = split_XYweights(df_test)

                        print("PRE-PROCESSING --> MinMaxScaling")
                        X_train, X_val, X_test = MinMaxScaling(X_train, X_val, X_test, ['title_length','year'])
                        print("PRE-PROCESSING --> LDA")
                        X_train, X_val, X_test = LDA(X_train, X_val, X_test, y_train)

                        if t == "CategoricalNB":
                                print("DISCRETIZATION for CategoricalNB")
                                n_cat_LD1 = int((X_train['LD1'].max() - X_train['LD1'].min()) * 2)
                                n_cat_LD2 = int((X_train['LD2'].max() - X_train['LD2'].min()) * 2)
                                n_cat_LD3 = int((X_train['LD3'].max() - X_train['LD3'].min()) * 2)
                                print(f"bins for LD1 are: {n_cat_LD1}")
                                print(f"bins for LD2 are: {n_cat_LD2}")
                                print(f"bins for LD3 are: {n_cat_LD3}")

                                X_train['LD1'] = pd.cut(X_train['LD1'], bins=n_cat_LD1, labels=range(n_cat_LD1))
                                X_train['LD2'] = pd.cut(X_train['LD2'], bins=n_cat_LD2, labels=range(n_cat_LD2))
                                X_train['LD3'] = pd.cut(X_train['LD3'], bins=n_cat_LD3, labels=range(n_cat_LD3))

                                X_val['LD1'] = pd.cut(X_val['LD1'], bins=n_cat_LD1, labels=range(n_cat_LD1))
                                X_val['LD2'] = pd.cut(X_val['LD2'], bins=n_cat_LD2, labels=range(n_cat_LD2))
                                X_val['LD3'] = pd.cut(X_val['LD3'], bins=n_cat_LD3, labels=range(n_cat_LD3))

                                X_test['LD1'] = pd.cut(X_test['LD1'], bins=n_cat_LD1, labels=range(n_cat_LD1))
                                X_test['LD2'] = pd.cut(X_test['LD2'], bins=n_cat_LD2, labels=range(n_cat_LD2))
                                X_test['LD3'] = pd.cut(X_test['LD3'], bins=n_cat_LD3, labels=range(n_cat_LD3))

                                clf = CategoricalNB()
                        elif t == "GaussianNB":
                                clf = GaussianNB()
                        else:
                                clf = QuadraticDiscriminantAnalysis()

                        print("FITTING & PREDICTING")
                        if t == "CategoricalNB":
                                #sample_weight is always the best choice
                                clf.fit(X_train, y_train, sample_weight=weights)
                        else:
                                clf.fit(X_train, y_train)

                        y_test_pred = clf.predict(X_test).tolist()
                        Z = clf.predict_proba(X_test)
                        y_test_preds.append(y_test_pred)

                        error = zero_one_loss(y_test, y_test_pred)
                        print(f"LOSS --> {error}")
                print("***********ENDING BAGGING***********")

                nr_predictions = len(y_test_preds[0])
                y_test_pred_voted = []
                print("VOTING")
                for prediction in range(nr_predictions):
                        y_test_pred_voted.append(Counter([item[prediction] for item in y_test_preds]).most_common(1)[0][0]

                loss_ensemble = zero_one_loss(y_test, y_test_pred_voted)
                print(f"LOSS ENSEMBLE (size: {size}, samples: {n_sample} 'technique': {t}) --> {loss_ensemble}\n\n")

                results = results.append({
                        'size': size,
                        'sample': n_sample,
                        'technique': t,
                        'loss_ensemble': loss_ensemble
                }, ignore_index=True)
```

```
***********BALANCING DATASET***********
```

```
*************************************** 1 out of 1 params' configurations --> size: 15797, samples: 1, technique Categorical
NB
***********STARTING BAGGING***********
****1° FIT su 1° sample del train****
PRE-PROCESSING --> split_XYweights
PRE-PROCESSING --> MinMaxScaling
PRE-PROCESSING --> LDA
DISCRETIZATION for CategoricalNB
bins for LD1 are: 19
bins for LD2 are: 24
bins for LD3 are: 26
FITTING & PREDICTING
LOSS --> 0.4824781604578773
***********ENDING BAGGING***********
VOTING
LOSS ENSEMBLE (size: 15797, samples: 1 'technique': CategoricalNB) --> 0.4824781604578773
```

Result

The best resulting loss ensamble is 0.482478 with CategoricalNB.

In [175…

```python
loss_ensemble
```

Out[175…

```
0.4824781604578773
```

We check for overfitting and underfitting.

In [171…

```python
print('Training set accuracy: {:.4f}'.format(clf.score(X_train, y_train)))

print('Test set accuracy: {:.4f}'.format(clf.score(X_test, y_test)))
```

```
Training set accuracy: 0.4002
Test set accuracy: 0.5175
```

By definition a confusion matrix C is such that $C_{i,j}$ is equal to the number of observations known to be in group i and predicted to be in group j.

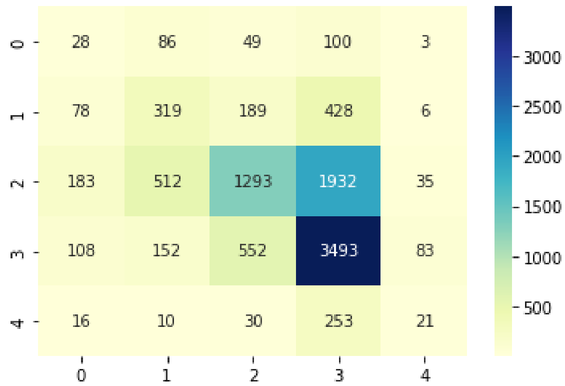
In [172…

```python
cm = confusion_matrix(y_test, y_test_pred_voted)

print('Confusion matrix\n\n', cm)
```

```
Confusion matrix

 [[  28   86   49  100    3]
 [  78  319  189  428    6]
 [ 183  512 1293 1932   35]
 [ 108  152  552 3493   83]
 [  16   10   30  253   21]]
```

In [173…

```python
cm_matrix = pd.DataFrame(data=cm)

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')
plt.show()
```



In [174…

```python
print(classification_report(y_test, y_test_pred_voted))
```

```
              precision    recall  f1-score   support

           0       0.07      0.11      0.08       266
           1       0.30      0.31      0.30      1020
           2       0.61      0.33      0.43      3955
           3       0.56      0.80      0.66      4388
           4       0.14      0.06      0.09       330
```

```
    accuracy                           0.52      9959
   macro avg       0.34      0.32      0.31      9959
weighted avg       0.53      0.52      0.50      9959
```

Random Forest

Hyperparameters optimization

In [306…

```
sizes = [1755, 1974, 2256, 2632, 3159, 3949, 5265, 7898, 15797]
n_estimators = [10, 40, 70, 90]
n_criterion = ["gini", "entropy"]
n_bootstrap = [True, False]
class_weight = ["balanced", "balanced_subsample"]

nr_configurations = len(sizes)*len(n_estimators)*len(n_criterion)*len(n_bootstrap)*len(class_weight)
```

In [ ]:

```
results = pd.DataFrame(columns=['size', 'samples', 'n_estimators', 'criterion', 'bootstrap', 'class_weight', 'loss_ensemble
config = 0

for size in sizes:
        print("***********BALANCING DATASET***********")
        df_train_SMOTE = balancing(df_train, size)
        min_size = df_train_SMOTE.bin_y.value_counts().min()
        n_sample = int(df_train.bin_y.value_counts().max()/min_size)
        df_trains = RandomSubSets(df_train_SMOTE, min_size, n_samples=n_sample) # n_samples a 12 se size == min_bin_cardin

        for estimator in n_estimators:
                for criterio in n_criterion:
                        for boot in n_bootstrap:
                                for weight in class_weight:
                                        y_val_preds = []
                                        config += 1
                                        print(f"************************************** {config} out of {nr_configurations}

                                        print("***********STARTING BAGGING***********")
                                        for n in range(len(df_trains)):
                                                print(f"****{n+1}° FIT su {len(df_trains)}° sample del train****")

                                                print("PRE-PROCESSING --> split_XYweights")
                                                X_train, weights, y_train, _, = split_XYweights(df_trains[n])
                                                X_val, _, y_val, _ = split_XYweights(df_val)
                                                X_test, _, y_test, _ = split_XYweights(df_test)

                                                print("PRE-PROCESSING --> MinMaxScaling")
                                                X_train, X_val, X_test = MinMaxScaling(X_train, X_val, X_test, ['title_leng

                                                #print("PRE-PROCESSING --> LDA")
                                                #X_train, X_val, X_test = LDA(X_train, X_val, X_test, y_train)

                                                print("FITTING & PREDICTING")
                                                rf = RandomForestClassifier(n_estimators=estimator, criterion=criterio, boo
                                                rf.fit(X_train, y_train)

                                                y_val_pred = rf.predict(X_val).tolist()
                                                y_val_preds.append(y_val_pred)

                                                error = zero_one_loss(y_val, y_val_pred)
                                                print(f"LOSS --> {error}")
                                        print("***********ENDING BAGGING***********")

                                        nr_predictions = len(y_val_preds[0])
                                        y_val_pred_voted = []
                                        print("VOTING")
                                        for prediction in range(nr_predictions):
                                                y_val_pred_voted.append(Counter([item[prediction] for item in y_val_preds]

                                        loss_ensemble = zero_one_loss(y_val, y_val_pred_voted)
                                        print(f"LOSS ENSEMBLE (n_estimators: {estimator}, criterion: {criterio}, bootstrap

                                        results = results.append({
                                                'size': size,
                                                'samples': n_sample,
                                                'n_estimators': estimator,
                                                'criterion': criterio,
                                                'bootstrap': boot,
                                                'class_weight': weight,
                                                'loss_ensemble': loss_ensemble
                                        }, ignore_index=True)
```

We store the results

In [ ]:

```
results.to_csv(os.path.join(current_path, tuning_path,'random_forest.csv'), index=False)
```

We load the results

In [540…

```
results = pd.read_csv(os.path.join(current_path, tuning_path,'random_forest.csv'))
```

We select the best hyperparametes

In [541…

```
results.loc[results['loss_ensemble'] == results['loss_ensemble'].min()]
```

Out[541…

	size	samples	n_estimators	criterion	bootstrap	class_weight	loss_ensemble
285	15797	1	90	entropy	True	balanced_subsample	0.502008

Test with best hyperparameters

In [ ]:

```
size = 15797
estimator = 90
criterio = "entropy"
boot = True
weight = "balanced_subsample"

df_train_SMOTE = balancing(df_train, size)
n_sample = int(df_train.bin_y.value_counts().max()/size)
min_size = df_train_SMOTE.bin_y.value_counts().min()
df_trains = RandomSubSets(df_train_SMOTE, min_size, n_samples=n_sample) # n_samples a 12 se size == min_bin_cardinality, 7
```

In [211…

```
y_test_preds = []

print(f"params' configurations --> size: {size}, samples: {n_sample}, n_estimators: {estimator}, criterion: {criterio}, boo

print("***********STARTING BAGGING***********")
for n in range(len(df_trains)):
        print(f"****{n+1}° FIT su {len(df_trains)}° sample del train****")

        print("PRE-PROCESSING --> split_XYweights")
        X_train, weights, y_train, weights = split_XYweights(df_trains[n])
        X_val, _, y_val, _ = split_XYweights(df_val)
        X_test, _, y_test, _ = split_XYweights(df_test)

        print("PRE-PROCESSING --> MinMaxScaling")
        X_train, X_val, X_test = MinMaxScaling(X_train, X_val, X_test, ['title_length','year'])

        print("FITTING & PREDICTING")
        rf = RandomForestClassifier(n_estimators=estimator, criterion=criterio, bootstrap=boot, class_weight=weight, randor
        rf.fit(X_train, y_train)

        y_test_pred = rf.predict(X_test).tolist()
        y_test_preds.append(y_test_pred)

        error = zero_one_loss(y_test, y_test_pred)
        print(f"LOSS --> {error}")
print("***********ENDING BAGGING***********")

nr_predictions = len(y_test_preds[0])
y_test_pred_voted = []
print("VOTING")
for prediction in range(nr_predictions):
        y_test_pred_voted.append(Counter([item[prediction] for item in y_test_preds]).most_common(1)[0][0])

loss_ensemble = zero_one_loss(y_test, y_test_pred_voted)
print(f"LOSS ENSEMBLE (n_estimators: {estimator}, criterion: {criterio}, bootstrap: {boot}, class_weight: {weight}) --> {lo
```

```
params' configurations --> size: 15797, samples: 1, n_estimators: 90, criterion: entropy, bootstrap: True, class_weight: ba
lanced_subsample
***********STARTING BAGGING***********
****1° FIT su 1° sample del train****
PRE-PROCESSING --> split_XYweights
PRE-PROCESSING --> MinMaxScaling
FITTING & PREDICTING
LOSS --> 0.5118987850185761
***********ENDING BAGGING***********
VOTING
LOSS ENSEMBLE (n_estimators: 90, criterion: entropy, bootstrap: True, class_weight: balanced_subsample) --> 0.5118987850185
761
```

Store columns relevance

In [ ]:
```
df_importance = pd.DataFrame([rf.feature_importances_], columns=X_train.columns)
df_importance.to_csv(os.path.join(current_path, results_path,'random_forest_feature_importances.csv'), index=False)
```

Result

The resulting loss ensamble is 0.511899.

In [213…
```
loss_ensemble
```

Out[213…
```
0.5118987850185761
```

Column relevance

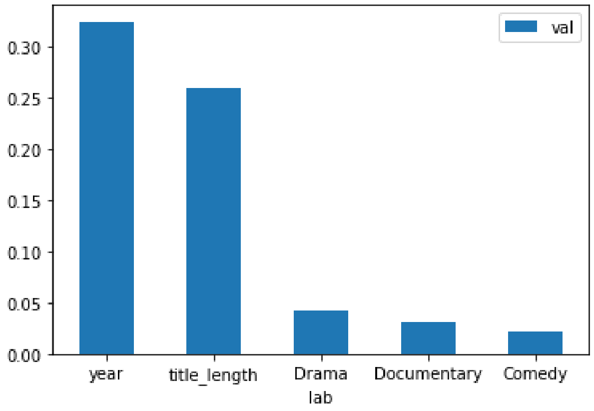
We load columns relevance.

In [216…
```
df_importance = pd.read_csv(os.path.join(current_path, results_path,'random_forest_feature_importances.csv'))
```

In [217…
```
features_importance = pd.DataFrame([df_importance.iloc[0]], columns=X_train.columns)
```

We plot first 5 columns relevance.

In [219…
```
first = 5
features_importance = df_importance.sum().sort_values(ascending=False)
df = pd.DataFrame({'lab':features_importance.iloc[:first].index, 'val':features_importance.iloc[:first]})
ax = df.plot.bar(x='lab', y='val', rot=0)
```



We try to print graphs of trees in PNG/SVG but they are too much large to display.

In [ ]:
```
from sklearn.tree import export_graphviz

for n in range(len(rf.estimators_)):
        export_graphviz(rf.estimators_[n],
                        out_file=os.path.join(results_path, f'tree_{n}.dot'),
                        feature_names = X_train.columns,
                        rounded = True, proportion = False,
                        precision = 2, filled = True)

        dot = os.path.join(results_path, f'tree_{n}.dot')
        png = os.path.join(results_path, f'tree_{n}.png')
        dot = f'tree_{n}.dot'
        png = f'tree_{n}.png'
        command = f"dot -Tsvg \"{dot}\" -o \"{png}\""
        print(command)
        #os.system(f"dot -Tpng {dot} -o {png} - Gdpi=600")
```

Neural Network

Preliminaries

Imports

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import classification_report
import itertools
import torch
from torch import nn
torch.backends.cudnn.benchmark = False
from torch.utils.data.sampler import WeightedRandomSampler
from torch.utils.data import Dataset, DataLoader, Subset, TensorDataset
from torch.utils.tensorboard import SummaryWriter
from torch.utils.tensorboard.summary import hparams
from pprint import pprint
from sklearn.metrics import precision_recall_fscore_support as score

from torchinfo import summary
from textwrap import dedent

from urllib.request import urlretrieve

import os
```

We want to exploit the parallel computing offered by CUDA on the GPU, if available.

In [ ]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Device: {}".format(device))
```

```
Device: cuda
```

The following function assures the reproducibility of experiments.

In [ ]:

```
def set_reproducibility(seed = 42):
        torch.manual_seed(seed)
        np.random.seed(seed)
        os.environ["CUBLAS_WORKSPACE_CONFIG"] = ":4096:8"
        torch.use_deterministic_algorithms(True)
```

Design

Class for data loading and pre-processing

By instatiating this class, we load the dataset in output by the *Data Manipulation* section of the notebook, we clean it with the same operations done in the *Data Cleaning section*, we split it into `X`, `y` and `ratings_count` as `weights` and, eventually, we discretize the continuous label into 5 discrete classes.

In [ ]:

```
class MoviesDataset(Dataset):
        def __init__(self):
                try:
                        df = pd.read_csv("datasets/df.csv")

                except FileNotFoundError:
                        print(f"Download in progress of df.csv")
                        file, _ = urlretrieve(url = "http://github.com/MickPerl/DataAnalyticsProject/releases/download/data
                        df = pd.read_csv(file)

                df = pd.read_csv("datasets/df.csv")
                df = self.cleaning(df)

                X, y, weights = self.split_XYweights(df)

                y = self.discretization(y)

                self.num_classes = y.nunique()
                self.X = torch.FloatTensor(X.values)
                self.y = torch.LongTensor(y)
                self.weights = torch.FloatTensor(weights)

        def __len__(self):
                return self.X.shape[0]

        def __getitem__(self, idx):
                return self.X[idx, :], self.y[idx], self.weights[idx]

        def split_XYweights(self, df):
                y = df['rating_mean']
                weights = df['ratings_count']
                X = df.drop(columns=['ratings_count', 'rating_mean'], axis=1)
                return X, y, weights

        def cleaning(self, df):
                df.dropna(subset = ['rating_mean'], inplace=True)
                df_without_tags = df[df.iloc[:, 23:-2].isna().all(axis=1)]
                df_without_tags_nor_genres = df_without_tags[df_without_tags['(no genres listed)'] == 1]
                rows_to_be_deleted = df.loc[df["movieId"].isin(df_without_tags_nor_genres["movieId"])].index
```

```
            df.drop(rows_to_be_deleted, axis=0, inplace=True)
            df.iloc[:, 23:-2] = df.iloc[:, 23:-2].fillna(0)
            df.drop(['(no genres listed)'], inplace=True, axis=1)
            df_year_without_na = df.year[-pd.isna(df.year)]
            df.year = df.loc[:, 'year'].fillna(np.median(df_year_without_na)).astype('int')
            df.drop('movieId', inplace=True, axis=1)
            df.drop_duplicates(inplace=True)
            return df

        def discretization(self, series):
            return pd.cut(series, bins=5, labels=False)
```

Class for the network architecture

By instantiating this class, we build the network architecture.\ The architecture is highly parametrized: in particular, some of the parameters that it is possible to specify are the activation functions of the first layer, that of the hidden layers and that of the output layer as well as the number of hidden layers, the probability of dropout and batch normalization.

In [ ]:

```
class Feedforward(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes, af_first_layer, af_hidden_layers, af_output_layer, num_hidden_
        super(Feedforward, self).__init__()

        model = [nn.Linear(input_size, hidden_size), af_first_layer]

        for i in range(num_hidden_layers):
            model.append(nn.Linear(hidden_size, hidden_size))

            if batch_norm:
                model.append(nn.BatchNorm1d(hidden_size))

            model.append(af_hidden_layers)

            if dropout != 0:
                model.append(nn.Dropout(dropout))


        model.append(nn.Linear(hidden_size, num_classes))

        if af_output_layer :
            model.append(af_output_layer)

        self.model = nn.Sequential(*model)


    def forward(self, x):
        return self.model(x)
```

Training function

We implement by hand the **early stopping** mechanism; in detail, we trigger it after the fifth epoch and we set to 3 the number of consecutive epochs we tolerate an increase of the loss (`n_bad_epochs`): every time the loss decreases with respect to the last min value, the counter of bad epochs is reset.

We log on **TensorBoard** some values such as the loss and the accuracy every batch, the loss and the accuracy every epoch as well as the weights and the bias every batch.

Furthermore, we check for the **vanishing and exploding gradient phenomenon**; even thought the architecture is well designed, there could be some batch containing bad examples which cause a na or inf gradient: ideally, these samples should be removed, but we solely skip them and continue training.

In [ ]:

```
def get_num_correct(preds, labels):
    return preds.argmax(dim=1).eq(labels).sum().item()

def train_model(model, criterion, optimizer, data_loader, epochs, n_bad_epochs, device, tb, cardinality_training_set):
        model.train()

        loss_values = []        # to store loss values over all batches regardless distinct epochs: it's the list we retur

        n_bad_epochs = n_bad_epochs
        patience = 0
        min_loss = np.Inf


        for epoch in range(epochs):
                losses_batches_current_epoch = []       # to store loss values over all batches with regard to a single epo
                correct_batches_current_epoch = []

                for batch_idx, samples in enumerate(data_loader):
                        data, targets = samples[0].to(device), samples[1].to(device)
```

```
            optimizer.zero_grad()

            y_pred = model(data)

            if str(criterion) == "CrossEntropyLoss()":
                    loss = criterion(y_pred, targets)
            else:   # "KLDivLoss()"
                    targets_one_hot_encoded = torch.nn.functional.one_hot(targets, num_classes=5).float()
                    loss = criterion(y_pred, targets_one_hot_encoded)

            correct = get_num_correct(y_pred, targets)

            tb.add_scalar("Loss every batch", loss, epoch * len(data_loader) + batch_idx + 1)
            tb.add_scalar("Correct every batch", correct, epoch * len(data_loader) + batch_idx + 1)
            tb.add_scalar("Accuracy every batch", correct / len(data), epoch * len(data_loader) + batch_idx + 1

            loss_values.append(loss.item())
            losses_batches_current_epoch.append(loss.item())
            correct_batches_current_epoch.append(correct)

            # Backward pass
            loss.backward()

            valid_gradients = True
            for name, param in model.named_parameters():
                    if param.grad is not None:
                            if torch.isnan(param.grad).any():
                                    print(f"{name} is nan, so model parameters are not going to be updated: thi
                                    optimizer.zero_grad()
                                    valid_gradients = False
                            if torch.isinf(param.grad).any():
                                    print(f"{name} is inf, so model parameters are not going to be updated: thi
                                    optimizer.zero_grad()
                                    valid_gradients = False
            if not valid_gradients :
                    continue

            optimizer.step()

            for name, value in model.named_parameters():
                    name = name.replace('.', '/')
                    tb.add_histogram('every batch_' + name, param.data.cpu().detach().numpy(), batch_idx + 1)
                    tb.add_histogram('every batch_' + name + '/grad', param.grad.data.cpu().numpy(), batch_idx

    total_correct_current_epoch = np.sum(correct_batches_current_epoch)
    tb.add_scalar("Correct every epoch", total_correct_current_epoch, epoch)

    accuracy_current_epoch = total_correct_current_epoch / cardinality_training_set
    tb.add_scalar("Accuracy every epoch", accuracy_current_epoch, epoch)

    for name, param in model.named_parameters():
            name = name.replace('.', '/')
            tb.add_histogram('every epoch_' + name, param.data.cpu().detach().numpy(), epoch)
            tb.add_histogram('every epoch_' + name + '/grad', param.grad.data.cpu().numpy(), epoch)

    mean_loss_current_epoch = np.mean(losses_batches_current_epoch)
    tb.add_scalar("Loss every epoch", mean_loss_current_epoch, epoch)

    if epoch < 5 :
            print(f"Epoch: {epoch}\t Mean Loss: {mean_loss_current_epoch}")
            continue

    if epoch == 5 :
            print("Waiting for three consecutive epochs during which the mean loss over batches does not decre

    if mean_loss_current_epoch < min_loss:
            # Save the model
            # torch.save(model)
            patience = 0
            min_loss = mean_loss_current_epoch
    else:
            patience += 1

    print(f"Epoch: {epoch}\t Mean Loss: {mean_loss_current_epoch}\t Current min mean loss: {min_loss}")

    if patience == n_bad_epochs:
            print(f"Early stopped at {epoch}-th epoch, since the mean loss over batches didn't decrease during
            return model, loss_values, epoch, mean_loss_current_epoch, accuracy_current_epoch


return model, loss_values, epoch, mean_loss_current_epoch, accuracy_current_epoch
```

Testing function

In [ ]:
```
def test_model(model, data_loader, device, output_dict = False):
        model.eval()
        y_pred = []
        y_test = []

        for batch_idx, samples in enumerate(data_loader):
            data, targets = samples[0].to(device), samples[1].to(device)
            y_pred.append(model(data))
            y_test.append(targets)
        y_pred = torch.stack(y_pred).squeeze()
        y_test = torch.stack(y_test).squeeze()
        y_pred = y_pred.argmax(dim=1, keepdim=True).squeeze()
        return classification_report(y_test.cpu(), y_pred.cpu(), zero_division=0, output_dict=output_dict)
```

Utilities

The following utility function lets us to obtain the samples' weights from the classes' weights: this output are going to be used in the sampler of the `DataLoader` object in order to manage the data imbalance.

In [ ]:
```
def class_weights(y):
    class_count = torch.bincount(y)
    class_weighting = 1. / class_count
    sample_weights = class_weighting[y]    # np.array([weighting[t] for t in y_train])
    return sample_weights
```

Due to a bug in the TensorBoard porting to PyTorch, we inherit the `SummaryWriter` class and overwrite the `add_hparams` function with some modifications.

In [ ]:
```
class SummaryWriter(SummaryWriter):

        def add_hparams(self, hparam_dict, metric_dict):
                torch._C._log_api_usage_once("tensorboard.logging.add_hparams")
                if type(hparam_dict) is not dict or type(metric_dict) is not dict:
                        raise TypeError('hparam_dict and metric_dict should be dictionary.')
                exp, ssi, sei = hparams(hparam_dict, metric_dict)

                self.file_writer.add_summary(exp)
                self.file_writer.add_summary(ssi)
                self.file_writer.add_summary(sei)
                for k, v in metric_dict.items():
                        if v is not None:
                                self.add_scalar(k, v)
```

We define a function to extract dictionaries containing a hyperparameters' configuration from the cartesian product of values of the hyperparameters; in detail, before creating a dictionary we check some condition in order to skip pointless or incorrect configurations.\
Examples of skipped configurations are those with:

- batch_size < 32 and batch norm, since batches aren't statistically significant;
- CrossEntropy as loss function and whichever activation function in the output layer, since CrossEntropy always contains SoftMax as activation function of output layer;
- Kullback-Leibler divergence as loss function and whichever activation function in the output layer other than SoftMax: since Kullback-Leibler divergence works with probability distributions, the SoftMax as the activation function of the output layer is a suitable choice in that it returns a probability distribution over classes for each feature vector in input.
- high probability of dropout (0.5) and a hidden layer sizes less than 64;
- low probability of dropout (0.2) and hidden layer size greater than 32;

In [ ]:
```
def dict_configs_from_params_cartesian_product(hyperparams) :
        name_params = list(hyperparams.keys())
        cartesian_product_filtered = []
        cartesian_product_config_params = itertools.product(*hyperparams.values())

        for conf_params in cartesian_product_config_params:
                conf_params_dict = {name_params[i]: conf_params[i] for i in range(len(hyperparams))}

                if conf_params_dict['batch_norm'] and conf_params_dict['batch_size'] < 32 :
                        continue

                if str(conf_params_dict['loss_function']) == "CrossEntropyLoss()" and conf_params_dict['af_output_layer']
                        continue

                if str(conf_params_dict['loss_function']) == "KLDivLoss()" and str(conf_params_dict['af_output_layer']) !=
                        continue

                if conf_params_dict['dropout'] == 0.5 and conf_params_dict['hidden_size'] < 64 :
                        continue

                if conf_params_dict['dropout'] == 0.2 and conf_params_dict['hidden_size'] > 32 :
```

```
                continue

            cartesian_product_filtered.append(conf_params_dict)

    return cartesian_product_filtered
```

Since the number of parameters' configurations are really high (~ 6000), we implement a function to split them into `nr_sets` subsets: so that, we are able to execute the hyperparameters optimization in parallel.

In [ ]:

```
def split_configs_params(dict_configs, nr_sets = 4):
        assert len(dict_configs) % nr_sets == 0,  "The number of configs params sets have to be a dividend of the cardinal
        print(f"Newly created sets (ratio {nr_sets}:1 to all {len(dict_configs)} configs):")

        for i in range(nr_sets):
                globals()[f"configs_set{i}"] = np.array_split(dict_configs, nr_sets)[i]
                print(f"configs_set{i}")
```

Neural Network in action

Creation training, validation and test set

In [ ]:

```
dataset = MoviesDataset()
train_idx, test_idx = train_test_split(np.arange(len(dataset)), test_size=0.2, stratify=dataset.y, random_state=42)
train_idx, val_idx = train_test_split(train_idx, test_size=0.1, stratify=dataset.y[train_idx], random_state=42)

X_train = dataset.X[train_idx]
X_val = dataset.X[val_idx]
X_test = dataset.X[test_idx]
```

We min-max scale `year` e `title_length` on training, validation and testing set.

In [ ]:

```
train_year_max = torch.max(X_train[:,0])
train_year_min = torch.min(X_train[:,0])
dataset.X[train_idx, 0] = (X_train[:,0] - train_year_min)/(train_year_max - train_year_min)
dataset.X[val_idx, 0] = (X_val[:,0] - train_year_min)/(train_year_max - train_year_min)
dataset.X[test_idx, 0] = (X_test[:,0] - train_year_min)/(train_year_max - train_year_min)

train_title_length_max = torch.max(X_train[:,1])
train_title_length_min = torch.min(X_train[:,1])
dataset.X[train_idx, 1] = (X_train[:,1] - train_title_length_min)/(train_title_length_max - train_title_length_min)
dataset.X[val_idx, 1] = (X_val[:,1] - train_title_length_min)/(train_title_length_max - train_title_length_min)
dataset.X[test_idx, 1] = (X_test[:,1] - train_title_length_min)/(train_title_length_max - train_title_length_min)
```

Managing imbalance

We create two samplers which we are going to pass to the `DataLoader` object in order to manage the data imbalance:

- `sampler_class_frequency` which, as its name reveals, weights each sample depending on the frequency of the class it belongs to.

In [ ]:

```
y_train = dataset.y[train_idx]

sample_weights = class_weights(y_train)
sampler_class_frequency = WeightedRandomSampler(sample_weights, len(train_idx))
```

The following code shows the classes' distribution over a subsets of batches.

In [ ]:

```
train_subset = Subset(dataset, train_idx)
print(type(train_subset))
train_loader=DataLoader(train_subset, batch_size=128, shuffle=False, sampler=sampler_class_frequency, drop_last=True)

for i, samples in enumerate(train_loader):
        if not i%10:
                print(len(np.where(samples[1].numpy() == 0)[0]),
        len(np.where(samples[1].numpy() == 1)[0]),
        len(np.where(samples[1].numpy() == 2)[0]),
        len(np.where(samples[1].numpy() == 3)[0]),
        len(np.where(samples[1].numpy() == 4)[0]), sep = "\t"
    )
```

```
<class 'torch.utils.data.dataset.Subset'>
26	22	25	28	27
35	18	28	26	21
29	24	27	27	21
23	28	28	27	22
28	22	23	23	32
28	30	18	31	21
31	21	18	29	29
```

20	28	36	13	31
22	27	29	26	24
17	23	29	30	29
31	25	22	28	22
39	19	19	24	27
30	27	25	28	18
31	17	28	20	32
27	22	29	29	21
23	29	35	22	19
28	30	23	24	23
22	29	30	23	24
21	28	25	32	22
31	23	25	22	27
22	35	23	18	30
24	28	16	31	29
25	30	24	28	21

- `sampler_ratings_count` which weights each sample depending on the `ratings_count` values.

In [ ]:

```
# MinMaxScaling ratings_count
weights_train = dataset.weights[train_idx]
weights_val = dataset.weights[val_idx]
weights_test = dataset.weights[test_idx]

weights_train_max = torch.max(weights_train)
weights_train_min = torch.min(weights_train)
dataset.weights[train_idx]  = (weights_train - weights_train_min) / (weights_train_max - weights_train_min)
dataset.weights[val_idx] = (weights_val - weights_train_min) / (weights_train_max - weights_train_min)
dataset.weights[test_idx] = (weights_test - weights_train_min) / (weights_train_max - weights_train_min)

sampler_ratings_count = WeightedRandomSampler(dataset.weights[train_idx], len(train_idx))
```

We have conducted some experiments with both and the performance have consistently been better with `sampler_class_frequency`, therefore we have ever adopted it during the following fine tuning.

Defining first hyperparameters space

Within the first hyperparameters optimization, we set the number of epochs to a very high value (500) as the early stopping assures that the training continues as long as the loss decreases and no further (in detail, the patience is set to 3). For remaining hyperparameters we define a wide space.

In [ ]:

```
first_hyperparams = {
        'num_epochs' : [500],
        'n_bad_epochs': [3],
        'num_hidden_layers' : [1, 3, 5, 7],
        'hidden_size' : [8, 16, 32, 64, 128],
        'batch_size' : [16, 32, 64, 128, 256],
        'af_first_layer' : [nn.Tanh(), nn.LeakyReLU()],
        'af_hidden_layers' : [nn.LeakyReLU()],
        'af_output_layer' : [None, nn.LogSoftmax(dim=1)],
        'loss_function' : [nn.CrossEntropyLoss(), nn.KLDivLoss(reduction = 'batchmean')],
        'dropout' : [0, 0.2, 0.5],
        'batch_norm' : [False, True],
        'learning_rate' : [0.01, 0.001],
        'optimizer': ["torch.optim.SGD", "torch.optim.Adam"],
        'weight_decay': [1e-4]
}
```

First training

We split the parameters' configurations into 6 sets and then we execute scripts specifying the index of the sets we want to consider.

In [ ]:

```
first_configs = dict_configs_from_params_cartesian_product(first_hyperparams)
nr_sets = 6
split_configs_params(first_configs, nr_sets)

idx_set = 1
assert idx_set < nr_sets, f"You can specify a set with an index until {nr_sets-1}"
config_set = eval(f"configs_set{idx_set}")

if config_set == first_configs:
        nr_train = 0
else :
        nr_train = len(configs_set0) * idx_set
```

We log to TensorBoard the architecture of the network and the various hyperparameters' configurations.

In [ ]:

```
set_reproducibility()
```

```
columns = ["nr_train"] + list(first_configs[0].keys()) + ["epoch_stopped", "loss", "accuracy", "precision", "precision_tota
results_first_ft = pd.DataFrame(columns=columns)

for config_params in config_set:
        nr_train += 1
        print(f"{nr_train}° training with params:")
        pprint(config_params)

        list_params_config = list(map(str, list(config_params.values())))
        name_run = '__'.join(list_params_config)
        with SummaryWriter(log_dir=os.path.join('tensorboard_logs', f"{idx_set}_out_of_{nr_sets - 1}", 'Train_' + str(nr_tr
        # tb = SummaryWriter(log_dir=os.path.join('tensorboard_logs', f"{idx_set}_out_of_{nr_sets - 1}", 'Train_' + str(nr_

                train_subset = Subset(dataset, train_idx)
                val_subset=Subset(dataset, val_idx)
                test_subset=Subset(dataset, test_idx)
                train_loader=DataLoader(train_subset, batch_size=config_params['batch_size'], shuffle=False, sampler=sample
                val_loader=DataLoader(val_subset, batch_size=1, shuffle=False, drop_last=True)
                test_loader=DataLoader(test_subset, batch_size=1, shuffle=False, drop_last=True)

                model = Feedforward(
                        dataset.X.shape[1],
                        config_params['hidden_size'],
                        dataset.num_classes,
                        config_params['af_first_layer'],
                        config_params['af_hidden_layers'],
                        config_params['af_output_layer'],
                        config_params['num_hidden_layers'],
                        config_params['dropout'],
                        config_params['batch_norm'])

                model.to(device)
                input_model = dataset.X[train_idx][:config_params['batch_size']].to(device)
                tb.add_graph(model, input_model)

                summary(model, input_size=(config_params['batch_size'], int(35850 // config_params['batch_size']), 1149), c

                loss_func = config_params['loss_function']

                optim = eval(config_params['optimizer'] + "(model.parameters(), lr=config_params['learning_rate'])")

                cardinality_training_set = len(X_train)
                model, loss_values, epoch_stopped, loss_value_last_epoch, accuracy_last_epoch = train_model(model, loss_fun

                print(f"Loss: {loss_value_last_epoch}", end="\n\n")

                report = test_model(model, val_loader, device, True)
                index_classes = len(report) - 3
                f1_score = [float(report[str(i)]['f1-score']) for i in range(index_classes)]
                f1_score_total = np.sum(f1_score)
                precision = [float(report[str(i)]['precision']) for i in range(index_classes)]
                precision_total = np.sum(precision)
                recall = [float(report[str(i)]['recall']) for i in range(index_classes)]
                recall_total = np.sum(recall)
                support = [int(report[str(i)]['support']) for i in range(index_classes)]
                accuracy = report['accuracy']


                row_values= [nr_train] + list_params_config + [epoch_stopped, loss_value_last_epoch, accuracy, precision, p
                results_first_ft=results_first_ft.append(pd.Series(row_values, index=columns), ignore_index=True)

                dict_params_config = {list(config_params.keys())[z]: list_params_config[z] for z in range(len(config_params
                tb.add_hparams(hparam_dict = dict_params_config, metric_dict = {"Accuracy every epoch": None, "Loss every e
                tb.flush()
                tb.close()
        del model, optim, train_loader, val_loader
```

In [ ]:

```
if config_set == first_configs:
        results_first_ft.to_csv("tuning_hyperparams/results_first_ft.csv", index=False)
else :
        results_first_ft.to_csv(f"tuning_hyperparams/results_nrSets{nr_sets}_idxSet{idx_set}.csv", index=False)
```

In [ ]:

```
results_first_ft = pd.concat([pd.read_csv(f"results_hyperparams_optimization/NN/results_nrSets6_idxSet{i}.csv") for i in ra
```

In [ ]:

```
results_first_ft.to_csv("tuning_hyperparams/results_first_ft.csv", index=False)
```

In [ ]:

```
results_first_ft = pd.read_csv("tuning_hyperparams/results_first_ft.csv")
```

We display the first 10 trainings sorted in descending order by accuracy: we note null precisions and recalls regarding class with lower frequency.

In [ ]:
```
results_first_ft.sort_values(by=['accuracy'], ascending=False).iloc[:10, -6:]
```

	loss	accuracy	precision	recall	f1_score	support
4041	1.715587	0.523795	[0.0, 0.0, 0.74194, 0.49482, 0.0]	[0.0, 0.0, 0.24147, 0.97881, 0.0]	[0.0, 0.0, 0.36436, 0.65734, 0.0]	[122, 335, 1143, 1416, 157]
4248	1.609664	0.475260	[0.0, 0.0, 0.39234, 0.68404, 0.0]	[0.0, 0.0, 0.77953, 0.43573, 0.0]	[0.0, 0.0, 0.52197, 0.53236, 0.0]	[122, 335, 1143, 1416, 157]
5489	32.183064	0.453829	[0.07336, 0.0, 0.41394, 0.72903, 0.07955]	[0.15574, 0.0, 0.74278, 0.39901, 0.04459]	[0.09974, 0.0, 0.53162, 0.51575, 0.05714]	[122, 335, 1143, 1416, 157]
3000	1.609313	0.452884	[0.0, 0.30435, 0.0, 0.45847, 0.0]	[0.0, 0.10448, 0.0, 0.99011, 0.0]	[0.0, 0.15556, 0.0, 0.62673, 0.0]	[122, 335, 1143, 1416, 157]
2442	1.609575	0.446896	[0.0, 0.0, 0.37011, 0.60171, 0.0]	[0.0, 0.0, 0.68679, 0.44703, 0.0]	[0.0, 0.0, 0.481, 0.51297, 0.0]	[122, 335, 1143, 1416, 157]
3016	1.609671	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	[0.0, 0.0, 0.0, 1.0, 0.0]	[0.0, 0.0, 0.0, 0.61713, 0.0]	[122, 335, 1143, 1416, 157]
3498	1.609638	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	[0.0, 0.0, 0.0, 1.0, 0.0]	[0.0, 0.0, 0.0, 0.61713, 0.0]	[122, 335, 1143, 1416, 157]
3034	1.612220	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	[0.0, 0.0, 0.0, 1.0, 0.0]	[0.0, 0.0, 0.0, 0.61713, 0.0]	[122, 335, 1143, 1416, 157]
5282	1.609482	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	[0.0, 0.0, 0.0, 1.0, 0.0]	[0.0, 0.0, 0.0, 0.61713, 0.0]	[122, 335, 1143, 1416, 157]
4626	1.609466	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	[0.0, 0.0, 0.0, 1.0, 0.0]	[0.0, 0.0, 0.0, 0.61713, 0.0]	[122, 335, 1143, 1416, 157]

We display the first 10 trainings sorted in ascending order by loss: the precision and recall regarding class with lower frequency are still quite imbalanced with respect to class with higher frequency but to a lesser extent.

In [ ]:
```
results_first_ft.sort_values(by=['loss']).iloc[:10, -6:]
```

	loss	accuracy	precision	recall	f1_score	support
5587	0.859668	0.410652	[0.06918, 0.22021, 0.6506, 0.80118, 0.09045]	[0.27049, 0.25373, 0.37795, 0.48093, 0.4586]	[0.11018, 0.23578, 0.47814, 0.60106, 0.1511]	[122, 335, 1143, 1416, 157]
2627	0.889824	0.414119	[0.07783, 0.21042, 0.61134, 0.80196, 0.11001]	[0.27049, 0.31343, 0.3867, 0.46328, 0.49682]	[0.12088, 0.2518, 0.47374, 0.58729, 0.18014]	[122, 335, 1143, 1416, 157]
5751	0.899446	0.414434	[0.07598, 0.2367, 0.61605, 0.78498, 0.09413]	[0.30328, 0.26567, 0.3762, 0.48729, 0.43949]	[0.12151, 0.25035, 0.46714, 0.60131, 0.15506]	[122, 335, 1143, 1416, 157]
5735	0.901097	0.414434	[0.07246, 0.22685, 0.64984, 0.81316, 0.09857]	[0.2459, 0.29254, 0.36045, 0.4887, 0.52866]	[0.11194, 0.25554, 0.4637, 0.6105, 0.16617]	[122, 335, 1143, 1416, 157]
4115	0.901097	0.409392	[0.0751, 0.24148, 0.65263, 0.82911, 0.10116]	[0.31148, 0.25373, 0.3797, 0.46257, 0.55414]	[0.12102, 0.24745, 0.48009, 0.59383, 0.17109]	[122, 335, 1143, 1416, 157]
2691	0.907835	0.424519	[0.08578, 0.21218, 0.6767, 0.80765, 0.11053]	[0.31148, 0.30149, 0.37358, 0.49223, 0.53503]	[0.13451, 0.24908, 0.4814, 0.61167, 0.18321]	[122, 335, 1143, 1416, 157]
4067	0.912362	0.414434	[0.08416, 0.23211, 0.63663, 0.83014, 0.10101]	[0.27869, 0.35821, 0.37708, 0.45904, 0.50955]	[0.12928, 0.28169, 0.47363, 0.59118, 0.1686]	[122, 335, 1143, 1416, 157]
4163	0.912675	0.417271	[0.07919, 0.22922, 0.63526, 0.80392, 0.1026]	[0.28689, 0.27164, 0.3657, 0.49223, 0.52866]	[0.12411, 0.24863, 0.46419, 0.6106, 0.17184]	[122, 335, 1143, 1416, 157]
2659	0.918037	0.402143	[0.07795, 0.20388, 0.69039, 0.79138, 0.10307]	[0.33607, 0.25075, 0.33946, 0.47952, 0.53503]	[0.12654, 0.2249, 0.45513, 0.59719, 0.17284]	[122, 335, 1143, 1416, 157]
4211	0.920030	0.415380	[0.07743, 0.22654, 0.67213, 0.79472, 0.101]	[0.28689, 0.29552, 0.35871, 0.48941, 0.51592]	[0.12195, 0.25648, 0.46777, 0.60577, 0.16893]	[122, 335, 1143, 1416, 157]

By analysing the corresponding hyperparameters, we understand that the best performance are obtained with hidden sizes greater than 16: therefore we extend the space of hidden sizes values. Moreover, we want to make experiment with greater value of batch_size (512) and a lesser learning rate.

Defining second hyperparameters space

In [ ]:
```
second_hyperparams = {
        'num_epochs' : [500],
        'n_bad_epochs': [3],
        'num_hidden_layers' : [3, 5, 7, 10],
        'hidden_size' : [16, 64, 128, 256],
        'batch_size' : [16, 64, 256, 512],
```

```
        'af_first_layer' : [nn.Tanh(), nn.LeakyReLU()],
        'af_hidden_layers' : [nn.LeakyReLU()],
        'af_output_layer' : [None, nn.LogSoftmax(dim=1)],
        'loss_function' : [nn.CrossEntropyLoss(), nn.KLDivLoss(reduction = 'batchmean')],
        'dropout' : [0, 0.5],
        'batch_norm' : [False, True],
        'learning_rate' : [0.01, 1e-5],
        'optimizer': ["torch.optim.SGD", "torch.optim.Adam"],
        'weight_decay': [1e-4]
}
```

For a better readability, we do not present once again the code implementing the training and the testing.

Within this second fine tuning, we enhance the performance analysis by computing also the sums of precision, recall and f1_score which are conditional to single classes.

In [ ]:

```
results_second_ft.to_csv("tuning_hyperparams/results_second_ft.csv", index=False)
```

In [ ]:

```
results_second_ft = pd.read_csv("tuning_hyperparams/results_second_ft.csv")
```

We display the first 10 trainings sorted in descending order by accuracy: we note null precisions and recalls regarding class with lower frequency.

In [ ]:

```
results_second_ft.sort_values(by=['accuracy'], ascending=False).iloc[:10, -9:]
```

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
41	1.316624	0.448471	[0.11633, 0.20382, 0.48902, 0.93333, 0.15126]	1.893765	[0.46721, 0.38209, 0.56518, 0.40537, 0.11465]	1.934499	[0.18627, 0.26584, 0.52435, 0.56524, 0.13043]	1.672135	[122, 335, 1143, 1416, 157]
56	1.611139	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
9	1.626771	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
101	1.609784	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
104	1.611960	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
48	1.612005	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
29	1.612719	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
53	1.610725	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
12	1.623199	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
64	1.611939	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.000000	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]

We display the first 10 trainings sorted in ascending order by loss: the precision and recall regarding class with lower frequency are still quite imbalanced with respect to class with higher frequency but to a lesser extent. However, the second fine tuning generally leads to a lower accuracy and a higher loss.

In [ ]:
```
results_second_ft.sort_values(by=['loss']).iloc[:10, -9:]
```

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
91	1.080970	0.386070	[0.08269, 0.22753, 0.64634, 0.80328, 0.1]	1.859841	[0.35246, 0.24179, 0.32458, 0.44986, 0.59236]	1.961050	[0.13396, 0.23444, 0.43215, 0.57673, 0.17111]	1.548393	[122, 335, 1143, 1416, 157]
83	1.098394	0.381658	[0.08611, 0.22701, 0.63393, 0.83508, 0.09596]	1.878084	[0.36066, 0.23582, 0.31059, 0.45056, 0.6051]	1.962723	[0.13902, 0.23133, 0.41691, 0.58532, 0.16565]	1.538235	[122, 335, 1143, 1416, 157]
107	1.103853	0.377246	[0.08961, 0.2125, 0.63194, 0.8099, 0.09779]	1.841738	[0.40984, 0.20299, 0.31846, 0.43927, 0.59236]	1.962904	[0.14706, 0.20763, 0.4235, 0.5696, 0.16787]	1.515662	[122, 335, 1143, 1416, 157]
75	1.155672	0.367160	[0.08722, 0.23214, 0.6568, 0.84637, 0.09555]	1.918089	[0.35246, 0.27164, 0.29134, 0.4202, 0.65605]	1.991688	[0.13984, 0.25034, 0.40364, 0.56159, 0.1668]	1.522205	[122, 335, 1143, 1416, 157]
59	1.160277	0.364324	[0.0775, 0.23843, 0.64272, 0.8209, 0.09389]	1.873449	[0.33607, 0.2, 0.29746, 0.42726, 0.65605]	1.916839	[0.12596, 0.21753, 0.4067, 0.56201, 0.16427]	1.476472	[122, 335, 1143, 1416, 157]
99	1.160518	0.364324	[0.07356, 0.2153, 0.65577, 0.80345, 0.09291]	1.840990	[0.30328, 0.22687, 0.29834, 0.42726, 0.61783]	1.873576	[0.1184, 0.22093, 0.4101, 0.55786, 0.16153]	1.468825	[122, 335, 1143, 1416, 157]
51	1.163139	0.375355	[0.09091, 0.24501, 0.68952, 0.84203, 0.09242]	1.959900	[0.37705, 0.25672, 0.31671, 0.42161, 0.63694]	2.009029	[0.1465, 0.25073, 0.43405, 0.56188, 0.16142]	1.554581	[122, 335, 1143, 1416, 157]
67	1.165624	0.365585	[0.09091, 0.2112, 0.64245, 0.82639, 0.09657]	1.867512	[0.31967, 0.24776, 0.29396, 0.4202, 0.68153]	1.963123	[0.14156, 0.22802, 0.40336, 0.55712, 0.16917]	1.499230	[122, 335, 1143, 1416, 157]
19	1.251483	0.352978	[0.08277, 0.25455, 0.64646, 0.83843, 0.09342]	1.915625	[0.40164, 0.20896, 0.27997, 0.40678, 0.66879]	1.966129	[0.13725, 0.22951, 0.39072, 0.54779, 0.16393]	1.469207	[122, 335, 1143, 1416, 157]
57	1.277507	0.363694	[0.08945, 0.31383, 0.78864, 0.93596, 0.09351]	2.221392	[0.54918, 0.17612, 0.30359, 0.40254, 0.70701]	2.138436	[0.15385, 0.22562, 0.43841, 0.56296, 0.16518]	1.546017	[122, 335, 1143, 1416, 157]

We display the first 10 trainings sorted in descending order by f1_score, which synthesize the precision and the recall.

In [ ]:
```
results_second_ft.sort_values(by=['f1_score_total'], ascending=False).iloc[:10, -9:]
```

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
41	1.316624	0.448471	[0.11633, 0.20382, 0.48902, 0.93333, 0.15126]	1.893765	[0.46721, 0.38209, 0.56518, 0.40537, 0.11465]	1.934499	[0.18627, 0.26584, 0.52435, 0.56524, 0.13043]	1.672135	[122, 335, 1143, 1416, 157]
105	1.291049	0.379136	[0.08971, 0.34146, 0.72505, 0.94617, 0.09413]	2.196519	[0.61475, 0.20896, 0.33683, 0.4096, 0.59236]	2.162503	[0.15658, 0.25926, 0.45998, 0.57171, 0.16245]	1.609967	[122, 335, 1143, 1416, 157]
73	1.298850	0.364639	[0.09821, 0.36111, 0.78005, 0.9479, 0.09183]	2.279099	[0.54098, 0.19403, 0.30096, 0.39831, 0.75159]	2.185873	[0.16625, 0.25243, 0.43434, 0.56091, 0.16366]	1.577594	[122, 335, 1143, 1416, 157]
51	1.163139	0.375355	[0.09091, 0.24501, 0.68952, 0.84203, 0.09242]	1.959900	[0.37705, 0.25672, 0.31671, 0.42161, 0.63694]	2.009029	[0.1465, 0.25073, 0.43405, 0.56188, 0.16142]	1.554581	[122, 335, 1143, 1416, 157]
91	1.080970	0.386070	[0.08269, 0.22753, 0.64634, 0.80328, 0.1]	1.859841	[0.35246, 0.24179, 0.32458, 0.44986, 0.59236]	1.961050	[0.13396, 0.23444, 0.43215, 0.57673, 0.17111]	1.548393	[122, 335, 1143, 1416, 157]
57	1.277507	0.363694	[0.08945, 0.31383, 0.78864, 0.93596, 0.09351]	2.221392	[0.54918, 0.17612, 0.30359, 0.40254, 0.70701]	2.138436	[0.15385, 0.22562, 0.43841, 0.56296, 0.16518]	1.546017	[122, 335, 1143, 1416, 157]
89	1.285883	0.355184	[0.10094, 0.70312, 0.80535, 0.94359, 0.09051]	2.643521	[0.61475, 0.13433, 0.28959, 0.38983, 0.78981]	2.218311	[0.17341, 0.22556, 0.426, 0.55172, 0.16241]	1.539106	[122, 335, 1143, 1416, 157]
83	1.098394	0.381658	[0.08611, 0.22701, 0.63393, 0.83508, 0.09596]	1.878084	[0.36066, 0.23582, 0.31059, 0.45056, 0.6051]	1.962723	[0.13902, 0.23133, 0.41691, 0.58532, 0.16565]	1.538235	[122, 335, 1143, 1416, 157]
97	1.290532	0.354554	[0.09446, 0.42424, 0.79268, 0.94898, 0.08887]	2.349234	[0.61475, 0.16716, 0.28434, 0.39407, 0.70701]	2.167332	[0.16376, 0.23983, 0.41854, 0.55689, 0.15789]	1.536910	[122, 335, 1143, 1416, 157]
81	1.280429	0.360227	[0.09852, 0.75472, 0.80048, 0.94676, 0.09138]	2.691841	[0.59836, 0.1194, 0.29484, 0.40184, 0.78981]	2.204247	[0.16918, 0.20619, 0.43095, 0.5642, 0.1638]	1.534318	[122, 335, 1143, 1416, 157]

The first trainings seems to be a good candidate as the best hyperparameters' configuration, since it presents an acceptable accuracy (0.44, given the fact that a random classifier over 5 classes presents an accuracy equal to 0.2) and low loss (1.31).

In [ ]:

```
results_second_ft.sort_values(by=['f1_score_total'], ascending=False).iloc[0,:]
```

```
nr_train                                                    5803
num_epochs                                                   500
n_bad_epochs                                                   3
num_hidden_layers                                             10
hidden_size                                                   64
batch_size                                                   512
af_first_layer                        LeakyReLU(negative_slope=0.01)
af_hidden_layers                      LeakyReLU(negative_slope=0.01)
af_output_layer                                  LogSoftmax(dim=1)
loss_function                                         KLDivLoss()
dropout                                                      0.0
batch_norm                                                 False
learning_rate                                            0.00001
optimizer                                       torch.optim.Adam
weight_decay                                              0.0001
epoch_stopped                                                 66
loss                                                    1.316624
accuracy                                                0.448471
precision            [0.11633, 0.20382, 0.48902, 0.93333, 0.15126]
precision_total                                         1.893765
```

```
recall                [0.46721, 0.38209, 0.56518, 0.40537, 0.11465]
recall_total                                                1.934499
f1_score              [0.18627, 0.26584, 0.52435, 0.56524, 0.13043]
f1_score_total                                              1.672135
support                                [122, 335, 1143, 1416, 157]
Name: 41, dtype: object
```

Defining third hyperparameters space

By analysing the corresponding hyperparameters, we understand that the best performance are obtained with higher number of hidden layers and bigger batches, so we extend their space: moreover, we delete 0.01 as learning rate and 0.2 as dropout probability since they do not lead to good performances.

In [ ]:

```
second_hyperparams = {
        'num_epochs' : [500],
        'n_bad_epochs': [3],
        'num_hidden_layers' : [3, 5, 7, 10],
        'hidden_size' : [16, 64, 128, 256],
        'batch_size' : [16, 64, 256, 512],
        'af_first_layer' : [nn.Tanh(), nn.LeakyReLU()],
        'af_hidden_layers' : [nn.LeakyReLU()],
        'af_output_layer' : [None, nn.LogSoftmax(dim=1)],
        'loss_function' : [nn.CrossEntropyLoss(), nn.KLDivLoss(reduction = 'batchmean')],
        'dropout' : [0, 0.5],
        'batch_norm' : [False, True],
        'learning_rate' : [0.01, 1e-5],
        'optimizer': ["torch.optim.SGD", "torch.optim.Adam"],
        'weight_decay': [1e-4]
}
```

In [ ]:

```
new_new_hyperparams = {
        'num_epochs' : [500],
        'n_bad_epochs': [3],
        'num_hidden_layers' : [12, 15, 18],
        'hidden_size' : [64, 128, 256],
        'batch_size' : [256, 512, 1024, 2048],
        'af_first_layer' : [nn.LeakyReLU()],
        'af_hidden_layers' : [nn.LeakyReLU()],
        'af_output_layer' : [None, nn.LogSoftmax(dim=1)],
        'loss_function' : [nn.CrossEntropyLoss(), nn.KLDivLoss(reduction = 'batchmean')],
        'dropout' : [0, 0.5],
        'batch_norm' : [False, True],
        'learning_rate' : [1e-5],
        'optimizer': ["torch.optim.Adam"],
        'weight_decay': [1e-4]
}
```

In [ ]:

```
results_third_ft.to_csv("tuning_hyperparams/results_third_ft.csv", index=False)
```

In [ ]:

```
results_third_ft = pd.read_csv("tuning_hyperparams/results_third_ft.csv")
```

We display the first 10 trainings sorted in descending order by accuracy: we note null precisions and recalls regarding class with lower frequency.

In [ ]:

```
results_third_ft.sort_values(by=['accuracy'], ascending=False).iloc[:10, -9:]
```

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
0	1.610409	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
244	1.609764	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
240	1.609740	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
28	1.613400	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
262	1.609661	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
264	1.609421	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
56	1.611898	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
232	1.609957	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
60	1.609902	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]
230	1.610333	0.446265	[0.0, 0.0, 0.0, 0.44627, 0.0]	0.446265	[0.0, 0.0, 0.0, 1.0, 0.0]	1.0	[0.0, 0.0, 0.0, 0.61713, 0.0]	0.617128	[122, 335, 1143, 1416, 157]

We display the first 10 trainings sorted in ascending order by loss: the precision and recall regarding class with lower frequency are still quite imbalanced with respect to class with higher frequency but to a lesser extent. However, the second fine tuning generally leads to a lower accuracy and a higher loss.

In [ ]:

```
results_third_ft.sort_values(by=['loss']).iloc[:10, -9:]
```

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
189	1.063103	0.345730	[0.08015, 0.20163, 0.55172, 0.73881, 0.0898]	1.662111	[0.36066, 0.2209, 0.26597, 0.41949, 0.51592]	1.782933	[0.13115, 0.21083, 0.35891, 0.53514, 0.15297]	1.388997	[122, 335, 1143, 1416, 157]
177	1.070294	0.340372	[0.0748, 0.17344, 0.54919, 0.72795, 0.0912]	1.616592	[0.31148, 0.19104, 0.26859, 0.41384, 0.5414]	1.726355	[0.12063, 0.18182, 0.36075, 0.52769, 0.15611]	1.347002	[122, 335, 1143, 1416, 157]
285	1.074537	0.328396	[0.08368, 0.1687, 0.50333, 0.72118, 0.09894]	1.575835	[0.32787, 0.20597, 0.26422, 0.37994, 0.59236]	1.770356	[0.13333, 0.18548, 0.34653, 0.49769, 0.16955]	1.332587	[122, 335, 1143, 1416, 157]
77	1.079831	0.371257	[0.08268, 0.22781, 0.62101, 0.78315, 0.09726]	1.811913	[0.34426, 0.22985, 0.28959, 0.44633, 0.61146]	1.921494	[0.13333, 0.22883, 0.39499, 0.5686, 0.16783]	1.493581	[122, 335, 1143, 1416, 157]
93	1.094534	0.344469	[0.07773, 0.19841, 0.56228, 0.78219, 0.09153]	1.712142	[0.30328, 0.22388, 0.27647, 0.40325, 0.59873]	1.805599	[0.12375, 0.21038, 0.37067, 0.53215, 0.15878]	1.395736	[122, 335, 1143, 1416, 157]
85	1.098131	0.356760	[0.07629, 0.18378, 0.57841, 0.80518, 0.09384]	1.737509	[0.30328, 0.20299, 0.30009, 0.41737, 0.59236]	1.816081	[0.12191, 0.19291, 0.39516, 0.54977, 0.16202]	1.421768	[122, 335, 1143, 1416, 157]
73	1.103233	0.357706	[0.08299, 0.21902, 0.60256, 0.78129, 0.09336]	1.779222	[0.32787, 0.22687, 0.28784, 0.41879, 0.61783]	1.879193	[0.13245, 0.22287, 0.38958, 0.54529, 0.16221]	1.452399	[122, 335, 1143, 1416, 157]
273	1.104342	0.326505	[0.07963, 0.16316, 0.51646, 0.73978, 0.09554]	1.594575	[0.35246, 0.18507, 0.26072, 0.38347, 0.57325]	1.754974	[0.12991, 0.17343, 0.34651, 0.50512, 0.16379]	1.318749	[122, 335, 1143, 1416, 157]
165	1.106367	0.364639	[0.08408, 0.22164, 0.64466, 0.82434, 0.09829]	1.873013	[0.38525, 0.25075, 0.29046, 0.4209, 0.6242]	1.971564	[0.13803, 0.23529, 0.40048, 0.55727, 0.16984]	1.500923	[122, 335, 1143, 1416, 157]
81	1.107266	0.348566	[0.08485, 0.21833, 0.57268, 0.77217, 0.08973]	1.737757	[0.34426, 0.24179, 0.28609, 0.39972, 0.57325]	1.845109	[0.13614, 0.22946, 0.38156, 0.52676, 0.15517]	1.429097	[122, 335, 1143, 1416, 157]

We display the first 10 trainings sorted in descending order by f1_score, however the first ones present accuracies lower than that of the beforementioned good configuration.

In [ ]:

```
results_third_ft.sort_values(by=['f1_score_total'], ascending=False).iloc[:10, -9:]
```

	loss	accuracy	precision	precision_total	recall	recall_total	f1_score	f1_score_total	support
76	1.283165	0.376930	[0.13428, 0.22107, 0.78251, 0.93677, 0.09335]	2.167984	[0.31148, 0.38209, 0.30534, 0.3976, 0.75159]	2.148093	[0.18765, 0.28009, 0.43927, 0.55825, 0.16608]	1.631347	[122, 335, 1143, 1416, 157]
4	1.303822	0.370942	[0.12058, 0.26269, 0.77427, 0.94676, 0.09063]	2.194923	[0.47541, 0.26269, 0.30009, 0.40184, 0.75796]	2.197982	[0.19237, 0.26269, 0.43253, 0.5642, 0.1619]	1.613702	[122, 335, 1143, 1416, 157]
36	1.295256	0.363063	[0.10375, 0.35417, 0.80145, 0.96055, 0.09373]	2.313641	[0.59016, 0.20299, 0.28959, 0.39548, 0.7707]	2.248919	[0.17647, 0.25806, 0.42545, 0.56028, 0.16713]	1.587392	[122, 335, 1143, 1416, 157]
68	1.254475	0.357075	[0.08079, 0.74667, 0.82915, 0.9527, 0.09144]	2.700744	[0.60656, 0.16716, 0.28871, 0.39831, 0.69427]	2.155008	[0.14258, 0.27317, 0.42829, 0.56175, 0.1616]	1.567400	[122, 335, 1143, 1416, 157]
72	1.286500	0.368736	[0.0922, 0.49485, 0.7584, 0.91733, 0.09422]	2.356999	[0.63934, 0.14328, 0.31584, 0.40749, 0.67516]	2.181108	[0.16116, 0.22222, 0.44595, 0.5643, 0.16537]	1.559003	[122, 335, 1143, 1416, 157]
160	1.265158	0.363694	[0.09439, 0.66154, 0.76068, 0.94684, 0.08852]	2.551971	[0.60656, 0.12836, 0.31146, 0.40254, 0.70701]	2.155925	[0.16336, 0.215, 0.44196, 0.56492, 0.15734]	1.542568	[122, 335, 1143, 1416, 157]
100	1.385441	0.363063	[0.07456, 0.30423, 0.74121, 0.83062, 0.0832]	2.033813	[0.27869, 0.32239, 0.25809, 0.43291, 0.64968]	1.941760	[0.11765, 0.31304, 0.38287, 0.56917, 0.14751]	1.530238	[122, 335, 1143, 1416, 157]
64	1.261881	0.354869	[0.09517, 0.44348, 0.83721, 0.94463, 0.08836]	2.408849	[0.53279, 0.15224, 0.28346, 0.3976, 0.78344]	2.149529	[0.16149, 0.22667, 0.42353, 0.55964, 0.15881]	1.530141	[122, 335, 1143, 1416, 157]
32	1.318057	0.350772	[0.09498, 0.72464, 0.78325, 0.92916, 0.08198]	2.614008	[0.57377, 0.14925, 0.27822, 0.39831, 0.70701]	2.106551	[0.16298, 0.24752, 0.41059, 0.55759, 0.14692]	1.525603	[122, 335, 1143, 1416, 157]
65	1.133167	0.366215	[0.08863, 0.21727, 0.66102, 0.82633, 0.09238]	1.885631	[0.37705, 0.23284, 0.30709, 0.41667, 0.61783]	1.951473	[0.14353, 0.22478, 0.41935, 0.55399, 0.16073]	1.502384	[122, 335, 1143, 1416, 157]

So the best model is the following:

In [ ]:

```
results_second_ft.sort_values(by=['f1_score_total'], ascending=False).iloc[0,:]
```

```
nr_train                                                      5803
num_epochs                                                     500
n_bad_epochs                                                     3
num_hidden_layers                                               10
hidden_size                                                     64
batch_size                                                     512
af_first_layer                           LeakyReLU(negative_slope=0.01)
af_hidden_layers                         LeakyReLU(negative_slope=0.01)
af_output_layer                                    LogSoftmax(dim=1)
loss_function                                            KLDivLoss()
dropout                                                        0.0
batch_norm                                                   False
learning_rate                                              0.00001
optimizer                                         torch.optim.Adam
weight_decay                                                0.0001
epoch_stopped                                                   66
loss                                                      1.316624
accuracy                                                  0.448471
precision            [0.11633, 0.20382, 0.48902, 0.93333, 0.15126]
precision_total                                           1.893765
recall               [0.46721, 0.38209, 0.56518, 0.40537, 0.11465]
```

```
recall_total                                              1.934499
f1_score            [0.18627, 0.26584, 0.52435, 0.56524, 0.13043]
f1_score_total                                            1.672135
support                          [122, 335, 1143, 1416, 157]
Name: 41, dtype: object
```

Save model

In [ ]:
```
torch.save(model.state_dict(), "best_model.pth")
```

CONCLUSION

In order to predict a movie rating we build or data pipeline:

Data acquisition -> Data Pre-processing + Visualization -> Modeling -> Performance analysis + Visualization

We find the best hyperparameters foreach method and we test the results.

Interesting note is that the best results are with a data balancing to the high class cardinality with SMOTE function.

Tests comparison

Random Forest loss ensemble is 0.5118987850185761 with params' configurations --> size: 15797, samples: 1, n_estimators: 90, criterion: entropy, bootstrap: True, class_weight: balanced_subsample

CategoricalNB loss ensemble is 0.482478 with params' configurations --> size: 15797, samples: 1

SVC RBF loss ensemble is 0.5414198212671955 with params' configurations --> size: 15797, samples: 1

Neural Network presents loss equal to 1.316624 and accuracy equal to 0.448471.

FUTURE WORKS

Parallel computing in preprocessing

In [ ]:
```
'''
    alternative code in case .fillna is too computationally intensive
'''

# df.to_csv("df_per_fillna.csv", index=False)
#
# #os.environ["MODIN_ENGINE"] = "ray"  # Modin will use Ray
# os.environ["MODIN_ENGINE"] = "dask"  # Modin will use Dask
#
# import dask
# import modin.pandas as pd_mod
# df_temp = pd_mod.read_csv("df_per_fillna.csv")
# df_temp.fillna(value=0)
# df_temp.to_csv("df_without_na.csv")
# df = pd.read_csv("df_without_na.csv")
```

Neural network regression task

Ratings_count as weight in bagging

Within the majority voting of bagging, we would try to exploit the feature `ratings_count` to break eventual ties.