Laboratory Exercise 2. Unsupervised learning.

Content-based recommendation system

Semantic data visualization using SOM

Time scheduling

ANNOUNCEMENT: 29 November 2022

Introduction of the

Dataset

You will find this in jupyter notebook format as an attachment at the end of the pronunciation.

```
!pip install --upgrade pip
!pip install --upgrade numpy
!pip install --upgrade pandas
!pip install --upgrade nltk
!pip install --upgrade scikit-learn
!pip install --upgrade joblib
```

The dataset we will work with is based on the Carnegie Mellon Movie Summary Corpus. It is a dataset with 22,301 movie descriptions. Each movie description consists of its title, one or more tags that characterize the genre of the movie, and finally the summary of its hypothesis. First we import the dataset (use the code as is, you don't need the csv file) into the dataframe `df_data_1` :

```
import pandas as pd

dataset_url = "https://drive.google.com/uc?export=download&id=1zo13kUAf-
MDMPZmBDxq1FxWtZY01lsxD"
df_data_1 = pd.read_csv(dataset_url, sep='\t', header=None, quoting=3)
```

Each team will work on a unique subset of 5,000 films (different dataset for each team) as follows:

1. Each team in the neural lab has a number in helios. You will put that number in the team_seed_number variable in the next code cell.
2. The data frame `df_data_2` has as many rows as there are groups and 5,000 columns. In each group corresponds to the row of the table with `team_seed_number` its. This line will contains 5,000 different numbers corresponding to movies in the original dataset.

3. Run the code. This will produce the unique titles, categories, catbins, summaries and corpus for each group to work with.

```
import numpy as np

# On the next line put the number of the group in the neural lab
team_seed_number = 0

movie_seeds_url = "https://drive.google.com/uc?
export=download&id=1g6F4TCHrs2wgtdOk7D3gtONaeirNt_Vo"
df_data_2 = pd.read_csv(movie_seeds_url, header=None)

# is selected
my_index = df_data_2.iloc[team_seed_number,:].values

titles = df_data_1.iloc[:, [2]].values[my_index] # movie titles (string)
categories = df_data_1.iloc[:, [3]].values[my_index] # movie categories (string)
bins = df_data_1.iloc[:, [4]]
catbins = bins[4].str.split(',', expand=True).values.astype(float)[my_index] # movie
categories in binary form (1 feature per category)
summaries = df_data_1.iloc[:, [5]].values[my_index] # movie summaries (string)
corpus = summaries[:,0].tolist() # list form of summaries
corpus_df = pd.DataFrame(corpus) # dataframe version of corpus
```

- The titles table contains the titles of the films. Example: 'Sid and Nancy'.
- The categories table contains the categories (genres) of the movie in string format. Example: '"Tragedy", "Indie", "Punk rock", "Addiction Drama", "Cult", "Musical", "Drama", "Biopic [feature]", "Romantic drama", "Romance Film", "Biographical film"'. Notice that it's a comma separated list of strings, with each string being a category.
- The catbins table again contains the categories of the films but in binary format (one hot encoding). Its dimensions are 5,000 x 322 (as many different categories as there are different categories). If the film belongs to a particular genre the corresponding column is set to 1, otherwise it is set to
  0.
- The summaries table and the corpus list contain the summaries of the movies (the corpus is just the summaries in list form). Each summary is a (usually large) string.
  Example: *'The film is based on the real story of a Soviet Internal Troops soldier who killed his The plot unfolds mostly on board of the prisoner transport rail car guarded by a unit of paramilitary conscripts.'*
- the dataframe corpus_df which is simply the corpus in dataframe format. The summaries are on column 0. It is probably convenient to do some preprocessing with dataframes.

We take the ID of each movie as its line number or the corresponding item in the list. Example: to print the summary of the movie with `ID=999` (the thousandth) we will write `print(corpus[999])`.

```
ID = 999
print(titles[ID])
print(categories[ID])
print(catbins[ID])
print(corpus[ID])
```

Application 1. Implementation of a content-based movie recommendation system

The first application you will develop will be a content based recommender system. Recommendation systems aim to automatically recommend to the user items from a collection that we ideally want the user to find interesting. The categorisation of recommender systems is based on how the recommended objects are filtered. The two main categories are collaborative filtering, where the system recommends to the user objects that have been positively rated by users who have a similar rating history to the user, and content based filtering, where objects with similar content (based on some characteristics) to those that the user has previously rated positively are suggested to the user.

The system of recommendations you will develop will be based on the content and specifically on the corpus of films.

Preview

The first step in our processing is to clean up the descriptions of the movies.

Print (several) different movie descriptions to see possible problems that need to be addressed.

The (minimum) cleaning steps we recommend are:

- convert all characters to lowercase,

- removal of stopwords. Note here that for the given task of the recommendation system, which is to suggest movies, it might be interesting to have stopword lists in addition to those of the common language.
- removing punctuation and special characters. This is not only done with NLTK. You could rely on regular expressions, and
- removal of very short strings.

Warning: the corpus and the final tokens that make it up will then be used as keys to find implants. For this reason, you should be careful about applying text normalization methods such as stemming and lemmatization.

Conversion in TFIDF

So the first step is to convert the corpus to a tf-idf representation:

```
from sklearn.feature_extraction.text import TfidfVectorizer
# create sparse tf_idf representation
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
corpus_tf_idf_plain = vectorizer.transform(corpus)
```

The TfidfVectorizer function as called here is not optimized. Its choice of methods and parameters can have a dramatic effect on the quality of the recommendations and is different for each dataset. Also, these choices also have a very large effect on the dimensionality and volume of the data. The dimensionality of the data in turn will have a very large impact on training times, especially in the second application of the exercise.

Warning: the TfidfVectorizer has some preprocessing features similar to those mentioned in the previous section. Whatever preprocessing you can do that only needs each document individually as input, do it in the first preprocessing step. If you need knowledge of the overall statistics of the collection, do it with TfidfVectorizer.

```
print(corpus_tf_idf_plain.shape)
```

Implementation of the recommendation system

The recommendation system you will implement will be a function `content_recommender` with three `max_recommendations` `target_movie` arguments: `target_movie` and `corpus_type` . In we pass the ,
ID of a target movie for which we are interested in finding similar content
(the synopsis) movies `max_recommendations` in the crowd.
Implement the function as follows:

- for the target movie, you will calculate its cosine similarity to all movies in your collection as represented in `corpus_type` .
- based on the cosine similarity calculated, create a sorted table from largest to smallest, with the indices (`ID`) of the movies. Example: if the movie with

index 1 has cosine similarity to 3 bands [0.2 1 0.6] (it has similarity 1 to itself) this ordered table of indices will be [1 2 0].

- For the target film print: id, title, synopsis, categories
- For the `max_recommendations` films (other than the target film itself which has a cosine similarity 1 to itself) with the highest cosine similarity (in descending order), print order of recommendation (1 closest, 2 second closest etc.), cosine similarity, id, title, abstract, and categories

Optimization of TfidfVectorizer

After you have implemented the `content_recommender` function, use it to optimize the `TfidfVectorizer` . In particular, first you can see what the `TfidfVectorizer` returns. system for random target strips and for a small `max_recommendations` (2 ή 3). If in some movies the system seems to return semantically close movies note the `ID` them. Then try to optimize the `TfidfVectorizer` for the specific `ID` Listing return semantically close movies for a larger number of `max_recommendations` . At the same time, as you optimize `TfidfVectorizer` , you should get good recommendations for a larger number of random movies.

At the same time, a somewhat countervailing direction of optimization is to use `TfidfVectorizer` its parameters in such a way that the dimensions of the
Vector Space Model up to the point where it starts to show effects on the quality of the recommendations.

Deep learning: creating corpora using word emmbeddings

The approach of building only through tfidf of the recommendation system has several disadvantages. We would therefore be interested to see if we can use for words embeddings, i.e. the dense vector representations for words given by the Word2Vec

However, the dataset of each group is too small to extract our own word embeddings (and well). For this reason we will use the Deep Learning methodology which is Transfer Learning..

In learning transfer we are essentially transferring the knowledge acquired by an already trained (and usually very large) system. The transfer is done through the values of the weights that it has determined after the training.

In our case, we are not so interested in the weights of the models themselves from which we will transfer learning. We would be interested in this if, for example, we wanted to continue training on our own texts. But we are interested in the implants themselves, i.e. the embeddings (dimension vectors m) that the neural has learned for its vocabulary. The vocabulary in such large neural will likely be a superset of ours.

Implant learning transfer

Implants of Gensim-data

Gensim includes several pre-trained Word2Vec implant models. With the next cell we get their list.

```
!pip install -U gensim
import gensim.downloader
print(list(gensim.downloader.info()['models'].keys()))
```

These models can be found in the Gensim-data repository where you can find their documentation. The loading of these models is done with the function `gensim.downloader.load`.

Other implants

You can find pre-trained implants from sources other than Gensim. For example:

- Google News dataset. These are pre-trained vectors trained on part of the Google News dataset (about 100 billion words). The model contains 300 dimensional vectors for 3 million words and phrases.
- Amazon BlazingText. BlazingText is not only pre-trained implants but also optimized implementations of Word2vec algorithms for text editing. A prerequisite is to work in SageMaker.

The procedures for loading embeddings from external data may be slightly different from that of Gensim.

Comments

- We repeat that in this paper we are not interested in the models themselves but in being able to find the embedding (vector) for a word in our vocabulary that corresponds to it in the pre-trained model.
- Also, we will not use `Phrases` it to find bigrams in our dataset as would be the most correct, as this would require continuing to train the model on a new vocabulary with very little new data.

Creating corpora based on the implants

In order to be able to incorporate the knowledge available in the pre-trained implants into our own corpus we will proceed as described below.

For each movie description d, which consists of N_d words w_i, the $tfidf$ of each word w_i is given by:

$$ tfidf(w_i) = tf(w_i,d) \cdot idf(w_i)$$

At the same time, to each word w_i corresponds a vector $W2V(w_i)$ from the implant model we have introduced. The implant vectors $W2V$ will have dimension m, depending on the model.

For each film d, we can create a vector representation $W2V(d)$ of dimension m using $tfidf(w_i)$ as the gravity coefficient for each implant
$W2V(w_i)$:

$$ W2V(d) = \frac{tfidf(w_1)\cdot W2V(w_i) + tfidf(w_2)\cdot W2V(w_2) + \dotsc + tfidf(*w{N{d}}*)\cdot W2V(*w{N{d}}*)}{tfidf(w_1)+tfidf(w_2)+ \dotsc + tfidf(*w{N{d}}*)}$$

build_tfw2v

Implement a function `build_tfw2v` with arguments:

- `corpus` which will be your preprocessed dataset,
- `vectors` which will be the model that will give you the vectors of the implantation vectors, an
  `embeddings_size` which will be the dimension of the implants m.
-

This function will return a new corpus which will be a matrix of 5000 (as many movies as you have) x m (the dimension of the implants). Depending on which model you use for transfer learning this table will be different.

You can now call it `content_recommender` with different corpora in the argument `corpus_type`. Note that in TFidfVectorizer we use the serial form of numpy arrays and you might find it useful `sparse.csr_matrix()` from Scipy.

Analysis of results

Recommendation system based only on tfidf

- In markdown, describe what pre-processing you do to the texts and why.
- Describe how you went about your choices for optimizing `TfidfVectorizer`. Cherry-picking:
- give examples (IDs) from your collection that return good
  results up to `max_recommendations` (at least 5) and comment.
- Nit-picking: give examples (IDs) from your collection that return bad results and comment.
- What are the overall advantages and disadvantages of a tfidf-based recommender?

Comparison and annotation with Word2Vec based recommenders

- Implement recommenders based on learning transfer and implants. Use examples to point out their strengths and weaknesses.
- You can comment on Word2Vec-based recommenders versus the simple tfidf model by looking at recommendations for the same ID.
- You can also compare Word2Vec recommenders against each other, again based on examples.
- Your comments will be based on the analysis of the qualitative characteristics which are the order and the set of recommendations. However, you can also include quantitative

characteristics such as the loading and assembly times of the corpus and the dimensionality m.

Use whatever reporting format you consider most appropriate: text, tables, charts.

Practical tip - object persistence with joblib.dump

As the second task requires you to create several corpora which take time to create, there is an easy way to store variables in dump files and read them directly.

The Python joblib library provides some extremely useful properties in code development: pipelining, parallelism, caching and variable persistence. We saw the first three properties in the first exercise. In this exercise we will find the fourth one, object persistence, useful. In particular we can with:

```
joblib.dump(my_object, 'my_object.pkl')
```

store any variable object (here `my_object`) directly on the filesystem as a file, which can then be recalled as follows:

```
my_object = joblib.load('my_object.pkl')
```

We can thus recall variables even after closing and reopening the notebook, without having to go through all the steps again one by one to generate them, which is especially useful if this process is time-consuming.

Let's save it `corpus_tf_idf` and then recall it.

```
import joblib

joblib.dump(corpus_tf_idf, 'corpus_tf_idf.pkl')
```

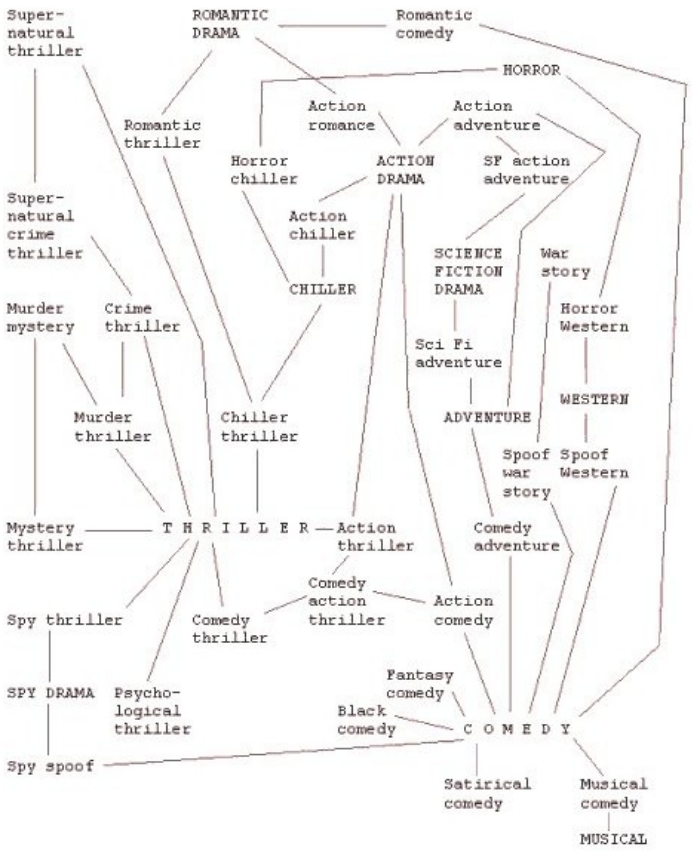
You can with a simple `!ls` to see that the file `corpus_tf_idf.pkl` exists in your filesystem (== persistence):

```
!ls -lh
```

and we can read them with `joblib.load`

```
corpus_tf_idf = joblib.load('corpus_tf_idf.pkl')
```

Application 2. Topological and semantic visualization of the film using SOM



Create dataset

In the second application we will rely on the topological properties of Self Organizing Maps (SOM) to create a two-dimensional grid map where all the movies in the collection of the group will be represented in a spatially coherent way in terms of their content and especially their type (the above map is indicative, it does not correspond to our dataset).

For the representation of the documents, choose the one that you think worked best in the first part of the exercise. Let's say this is `my_best_corpus .`

The ready function `build_final_set` will join the representation you give it as an argument to the `mycorpus` with binarized categories `catbins` movies as extra columns (attributes). Therefore, each movie is represented in the Vector Space Model by its features reproduction `mycorpus` and its categories.

Finally, the function takes a definition of how many movies to return, with a default value of all movies (5000). This is useful if you want to build smaller datasets to train the SOM faster.

We will run the function with `final_set = build_final_set(my_best_corpus) .`

```
def build_final_set(mycorpus, doc_limit = 5000, tf_idf_only=False):
    # convert sparse tf_idf to dense tf_idf representation
    dense_tf_idf = mycorpus.toarray()[0:doc_limit,:]
    if tf_idf_only:
        # use only tf_idf
        final_set = dense_tf_idf
    else:
        # append the binary categories features horizontally to the (dense) tf_idf
features
        final_set = np.hstack((dense_tf_idf, catbins[0:doc_limit,:]))
    # somoclu wants data in float32 return
    np.array(final_set, dtype=np.float32)
```

In the next cell, we print the dimensions of our final dataset. Without TFIDF optimization we will have about 50,000 features and the it will be impractical to proceed with SOM training.

```
final_set.shape
```

SOM map training

We will work with the SOM library "Somoclu". We import somoclu and matplotlib and tell matplotlib to print within the notebook (not in a pop up window).

```
# install somoclu
!pip install --upgrade somoclu
# import sompoclu, matplotlib
import somoclu
import matplotlib
# we will plot inside the notebook and not in separate window
%matplotlib inline
```

First read the function reference of somoclu. We will work with a planar, parallelogram-shaped neuron map with random initialization (all these are default).
You can try different map sizes, but as the number of neurons increases, the training time increases. For training you do not need to exceed 100 epochs. In general we can rely on the default parameters until we are able to visualize and analyze the results qualitatively. Start with a 10 x 10 map, 100 training epochs and a subset of the movies (e.g. 2000). Use the `time` to you have a picture of the training times.

Best matching units

After each training, store in a variable the best matching units (bmus) for each movie. The bmus show us which neuron each strip belongs to. Attention: the convention of neuron coordinates in Somoclu is (column, row) i.e. the reverse of Python. Using np.unique (a very useful function in the exercise) store the unique best matching units and their indices to the bands.

Note that you may have fewer unique bmus than number of neurons because some neurons may not have assigned bands. We will consider the row number in the unique bmus table as the neuron number.

Grouping (clustering)

Typically, the clustering in a SOM map is derived from the unified distance matrix (U-matrix): for each node the average distance from its neighbours is calculated. If blue is used in the areas of the map where this value is low (short distance) and red where the value is high (long distance), then we can say that the blue areas are clusters and the red areas are boundaries between clusters.

Somoclu gives the additional possibility to cluster the neurons using any scikit-learn clustering algorithm. In this exercise we will use k-Means. For your initial map try a k=20 or 25. The two clustering approaches are different, so expect the results to be close but not the same.

Storage of the SOM

Because SOM initialization is random and clustering is also a stochastic process, the locations and labels of neurons and clusters will be different every time you run the map, even with the same parameters. To save a specific som and clustering again use `joblib` . After recalling a SOM remember to follow the procedure for bmus.

U-matrix visualization, clustering and cluster size

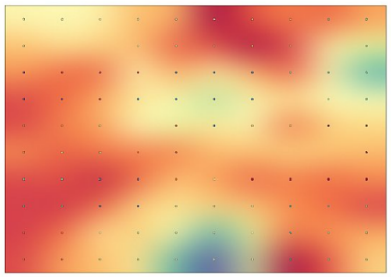
To print the U-matrix use the `view_umatrix` with argume`bestmatches=True` and `figsize=(15, 15)` or `figsize=(20, 20)` . The different colors shown in the nodes represent the different clusters resulting from the k-Means. You can display the caption of the U-matrix with the `colorbar` argument.

samples, the number is very large.

For a second clearer visualization of clustering, print directly the variable `clusters`

Finally, again using the `np.unique` (with a different argument) and `np.argsort` (there are other ways of implementation) print the labels of the clusters (numbers from 0 to k- 1) and the number of neurons in each cluster, in descending or ascending order with respect to the number of neurons. It is essentially a tool to easily find the large and small clusters.

The following is a non-optimized example for the three previous outputs:



```
[[11 11 11  9  9  9 13 13  5  5]
 [11 11 17  9  9 13 13 13  5  5]
 [ 0 17 17 17  2  2  2  5  5  5]
 [ 0  0 17  2  2  2  2 12  5  5]
 [14 14  8  2 16  2 12 12 19 19]
 [14  8  8 16 16 16 12 12 19 19]
 [ 4  4  1  1 16 10 18 18 19]
 [ 4  4  1  1 10 10 10  3  3  6]
 [15 15  7  7 10 10 10  3  6  6]
 [15 15  7  7 10 10 10  3  6  6]]

Clusters sorted by increasing number of neurons:
Cluster index
Number of neurons
[[ 0  8 18 14  1  3  4 15  7 17 16 13  9 11  6 12 19  5  2 10]
 [ 3  3  3  3  4  4  4  4  4  5  5  5  5  5  5  5  5  9  9 10]]
```

Semantic interpretation of clusters

In order to study the topological properties of the SOM and whether they have incorporated semantic information about the movies through the vector representation of tf-idf, implants and categories, we need a qualitative review criterion for clusters.

We will implement the following criterion: We take as argument a cluster number (label). For this cluster we find all the neurons assigned to it by k-Means. For all these neurons we find all the bands assigned to them (for which they are bmus). For all these bands we print sorted total statistics of all species (categories) and their frequencies. If the cluster has good consistency and specificity, some categories should have a significantly higher frequency than the others. We can then assign this/these category(ies) as movie genre tags to the cluster.

You can implement this function however you want. A possible procedure could be the following:

1. We define a function `print_categories_stats` that takes as input a list of movie ids. We create an empty list of global categories. Then, for each movie process the string string `categories` as follows: create a list by separating it and remove appropriately with the `split` the whitespaces between tags with `strip` . We add this list to the total list of categories with the `extend` . Finally we use `np.unique` it again to measure the frequency of unique tags categories and sort with `np.argsort` . We print the categories and frequencies appearance sorted. You may also find `np.ravel` , `np.nditer` useful, `np.array2string` and `zip`
2. We define our basic function `print_cluster_neurons_movies_report` which accepts as argument the number of a cluster. Using the `np.where` we can find the coordinates of the bmus corresponding to the cluster and with the `column_stack` make a bmus table for the cluster. Pay attention to the order (column - row) in the bmus table. For each bmu of this table we check if it exists in the unique bmus table we have

compute at the beginning in total and if so add the corresponding index of the neuron to a list. Useful can also be `np.rollaxis` , `np.append` , `np.asscalar` . You will also probably need to implement a similarity criterion between a bmu and a unique bmu from the original bmus table.

3. We implement a helper function `neuron_movies_report` . It takes a set of neurons from the `print_cluster_neurons_movies_report` and through `indices` makes the a list of the set of films belonging to these neurons. At the end it calls with this the list the `print_categories_stats` which prints the statistics of the categories.

You can of course add any additional output that helps you. A useful output is how many neurons belong to the cluster and how many and which of them have been assigned tapes.

We will perform the semantic interpretation of the map by calling the `print_cluster_neurons_movies_report` with the number of a cluster of interest.

Example output for a cluster (probably not a globally optimized map, but you can see that the major categories have semantic relevance):

```
Overall Cluster Genres stats: [("Horror", 86), ("Science Fiction", 24), ("B-
16), ("Monster movie", 10), ("Creature Film", 10), ("Indie", 9), ("Zombie Film",
("Slasher", 8), ("World cinema", 8), ("Sci-Fi Horror", 7), ("Natural horror films",
("Supernatural", 6), ("Thriller", 6), ("Cult", 5), ("Black-and-white", 5), ("Japanese
Movies", 4), ("Short Film", 3), ("Drama", 3), ("Psychological thriller", 3),
Fiction", 3), ("Monster", 3), ("Comedy", 2), ("Western", 2), ("Horror Comedy", 2),
("Archaeology", 2), ("Alien Film", 2), ("Teen", 2), ("Mystery", 2), ("Adventure", 2),
("Comedy film", 2), ("Combat Films", 1), ("Chinese Movies", 1), ("Action/Adventure",
1), ("Gothic film", 1), ("Costume drama", 1), ("Disaster", 1), ("Docudrama", 1), ("Film
adaptation", 1), ("Film noir", 1), ("Parody", 1), ("Period piece", 1), ("Action",
```

Tips for SOM and clustering

- For clustering a U-matrix it is good to show both blue-green regions (clusters) and red regions (boundaries). Notice what relationship exists between number of bands in the final set, grid size and U-matrix quality.
- For the k of k-Means try to approximate relatively the clusters of the U-matrix (as we said they are different clustering methods). Small number of k will not respect the bounds. Large number will create sub-clusters within the clusters shown in the U-matrix. The latter is not necessarily bad, but it increases the number of clusters that need to be semantically analyzed.
- On small maps and with small final sets try different parameters for SOM training. Note any parameters that affect the quality of clustering for your dataset so that you can apply them to large maps.
- Some topological features already appear on small maps. Others need larger maps. Try 20x20, 25x25 or even 30x30 sizes and adjust the k accordingly.

Analysis of topological properties of SOM map

At the end of the training and clustering you will have a map with topological properties for the types of taines in your collection, similar to the image at the beginning of Application 2 of this notebook. This image is for illustration purposes only, it is not a SOM map andhas no relation to our data collection and categories.

For the final SOM map you will produce for your collection, analyze in markdown with specific reference to cluster numbers and their semantic interpretation the following three topological properties of the SOM:

1. Data that have a higher probability density in the input space tend to be represented by more neurons in the reduced dimensionality space. Give examples of frequent and less frequent classes of bands. Use the statistics of the categories in your collection and the number of nodes they characterize.
2. Distant input patterns tend to be displayed remotely on the map. There are typical categories of movies that already from small maps tend to be placed in different or remote parts of the map.
3. Nearby input patterns tend to be plotted close to the map. On large maps identify film types and their nearby subspecies.

Obviously fitting in 2 dimensions that respects an absolute topology is not possible, firstly because there is no absolute by definition for cinematic genres even in many dimensions, and secondly because we are realizing dimensionality reduction.

Identify large clusters and small clusters that do not have clear characteristics. Identify clusters of specific species that appear to lack topological relevance to surrounding areas. Suggest possible interpretations.

Finally, identify clusters that in your view are of particular interest in your team's collection (data exploration / discovery value) and comment.

Final exercise delivery

- You will deliver to helios this notebook edited or one or two new zip files with your answers to the questions of both applications.
- Remember that the markdown map analysis with reference to cluster numbers must refer to the final map with the cells visible that you deliver or the resulting map will be different and the cluster labels will not correspond to your analysis.
- Don't forget a markdown cell with your team details at the beginning.

Please go through the notebook step-by-step to avoid forgetting deliverables