Data shuffle part

Steam Games Predictor by: *Kornel Zieliński, Krystian Rodzaj, Krystian Wojakiewicz*

Introduction

The topic of our project is centered around neural networks and machine learning algorithms. We decided to work in the *active* mode. After we had decided the basic topic of the project, the first thing that came to our mind was the **Kaggle** portal, which holds large amounts of datasets usable for data analysis and machine learning. The key was to pick such dataset, for which we would be able to accurately test our results, as well as keep the dataset relatively small, for practical purposes. We decided on a dataset containing video games details fetched from the **Steam** gaming service using their API. The dataset is just under one year old. This creates an opportunity for us to later test our results using the same **Steam API** to gather newer data on recently released games and treat them as input for our algorithms. The main goal and at the same time the output of our network is the total rating which is based on positive and negative ratings. So the task of our network is to predict how the game will be evaluated by people and how successful it will be in the gaming market. Those are the two main types of tasks that can be solved by the machine learning algorithms - linear regression and classification.

Data load and its structure

Firstly, we imported some helpful libraries and defined global variables. Variable names are pretty much self-explanatory, **Numpy** library was imported to help with mathematical problems and data preprocessing, an **Os** library provides ways of cooperating with the operating system.

In [1]:

```
from __future__ import division, print_function, unicode_literals
import numpy as np
import os
import matplotlib.pyplot as plt
import json
import pprint

plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

PROJECT_ROOT_DIR = "."
CHAPTER_ID = "preparing_dataset"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "pictures", CHAPTER_ID)
```

Then we defined some help functions. First one on the list is _save*fig* function responsible for saving generated during the excercise images.

In [2]:

```
def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving image", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

Next one is the _load*data* function that loads our csv file and saves it as *Pandas's DataFrame*. **Pandas** library is a tool that enables data analysis and manipulation in **Python**, it provides a lot of functions and objects such as the previously mentioned **DataFrame** that is basically a data structure built with labeled columns and rows (like an excel table or database).

In [3]:

```
import os
import pandas as pd

def load_data(steam_path, file):
    csv_path = os.path.join(steam_path, file)
    return pd.read_csv(csv_path, error_bad_lines=False)
```

Now, we can load our data and assign it to a variable creating a DataFrame object and see how it's structured. To do that we've used the prevoiusly declared load_data function and then **head()** function of the DataFrame object. The second one returns first few rows of the DataFrame, the default value of the rows is 5 but it can be also specified inside the brackets as a function parameter.

In [4]:

```
steam = load_data('../data', 'steam.csv')
steam.head()
```

Out[4]:

	appid	name	release_date	english	developer	publisher	platforms	required_age	categories	genres
0	10	Counter-Strike	2000-11-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Online Multi-Player;Local Multi-P...	Action
1	20	Team Fortress Classic	1999-04-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Online Multi-Player;Local Multi-P...	Action
2	30	Day of Defeat	2003-05-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Valve Anti-Cheat enabled	Action
3	40	Deathmatch Classic	2001-06-01	1	Valve	Valve	windows;mac;linux	0	Multi-player;Online Multi-Player;Local Multi-P...	Action
4	50	Half-Life: Opposing Force	1999-11-01	1	Gearbox Software	Valve	windows;mac;linux	0	Single-player;Multi-player;Valve Anti-Cheat en...	Action

Great, as you can see, our csv file has been successfully loaded into the DataFrame object, we can see it consists of 14 columns, from now on these columns will be called attributes. Now we want to find out how many rows our *steam* object has. To do that, we need use **info()** functions from the DataFrame object that does excatly what its name says - it prints information about the DataFrame object such as:

1. name of the column,
2. number of entries (rows),
3. number of columns (attributes),
4. number of non-null values,
5. types of the attributes.

In [5]:

```
steam.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27075 entries, 0 to 27074
Data columns (total 18 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   appid             27075 non-null  int64
 1   name              27075 non-null  object
 2   release_date      27075 non-null  object
 3   english           27075 non-null  int64
 4   developer         27075 non-null  object
 5   publisher         27075 non-null  object
 6   platforms         27075 non-null  object
 7   required_age      27075 non-null  int64
 8   categories        27075 non-null  object
 9   genres            27075 non-null  object
 10  steamspy_tags     27075 non-null  object
 11  achievements      27075 non-null  int64
 12  positive_ratings  27075 non-null  int64
 13  negative_ratings  27075 non-null  int64
 14  average_playtime  27075 non-null  int64
 15  median_playtime   27075 non-null  int64
 16  owners            27075 non-null  object
 17  price             27075 non-null  float64
dtypes: float64(1), int64(8), object(9)
memory usage: 3.7+ MB
```

Now we can see that, thankfully, our dataset does not contain any null values and we don't have to fill them in. Names of the attributes are pretty much self explanatory, but just to be precise, we need to clarify some of the most important to~us:

1. english - tells whether the given game supports english lanuague (binary attribute - 0/1),
2. platforms - tells which platform game is available (three main platforms, semicolon separated),
3. owners - tells how many gamers bought the game.

Now, seeing how the most important attributes for us are *owners* and to be made *rating* we can try to get some interesting information. There are some **object** types (basically string) in our dataset that will require encoding into numerical values before looking for correlation (only numerical attributes can be used in correlation searching functions) and preprocessing with some of the machine learning algorithms (same point). Firstly, we can check some attributes without encoding them, let's do exatcly that with our owners attribute then. For that we'll use the **value_counts()** function from the Pandas's **Series** object. Series is a one-dimensional array, it's what a DataFrame object is made of.

In [6]:

```
steam["owners"].value_counts()
```

Out[6]:

```
0-20000               18596
20000-50000            3059
50000-100000           1695
100000-200000          1386
200000-500000          1272
500000-1000000          513
1000000-2000000         288
2000000-5000000         193
5000000-10000000         46
10000000-20000000        21
20000000-50000000         3
50000000-100000000        2
100000000-200000000       1
Name: owners, dtype: int64
```

Now we can see how our owners attribute is classified, of course there are 18596 games owned by 0-20000 gamers, but there's only one game owned by 100000000-200000000, that being **Dota 2**.

Great, now let's check platforms attribute in the same way. We can also visualize this by building this attribute's histogram.

In [7]:

```
steam["platforms"].value_counts()
```

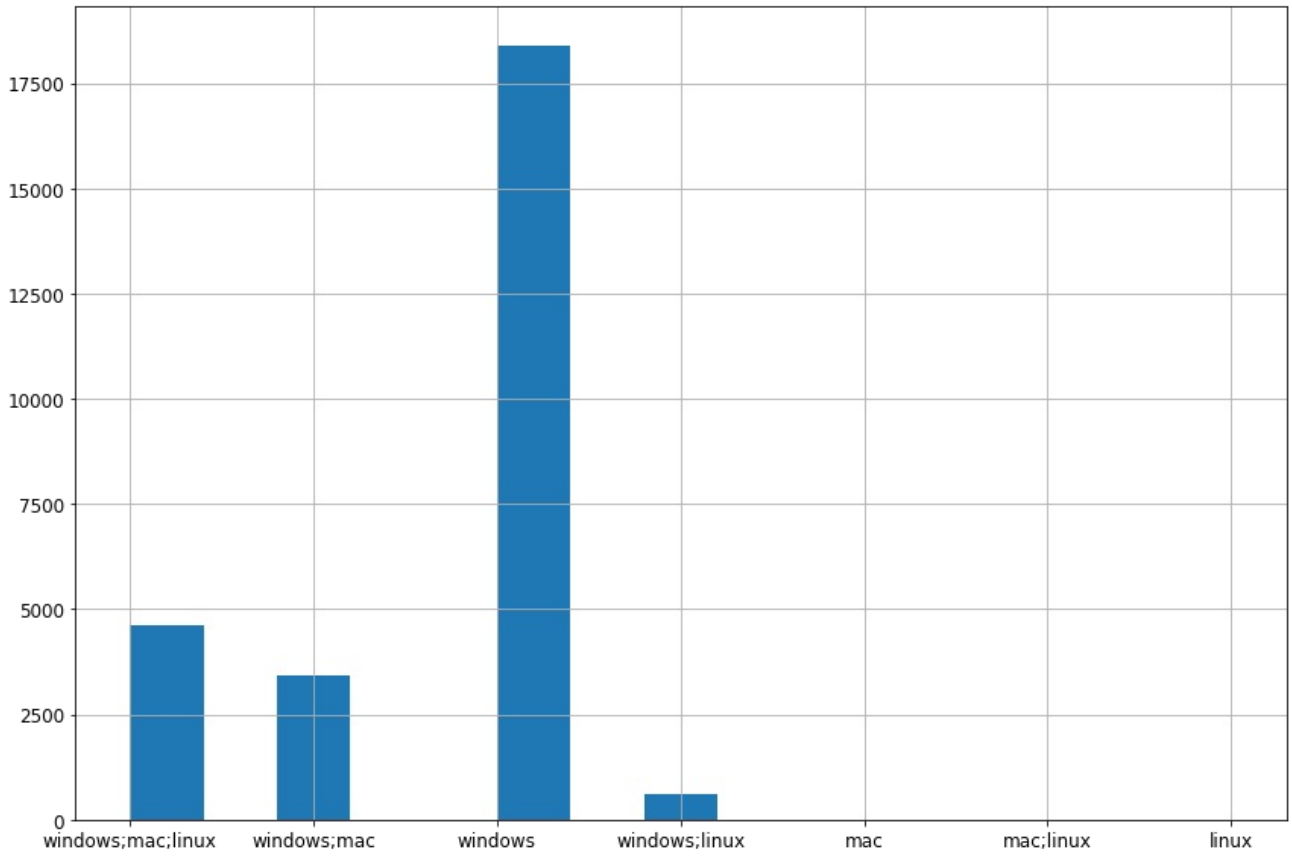
Out[7]:

```
windows              18398
windows;mac;linux     4623
windows;mac           3439
windows;linux          610
mac                      3
mac;linux                1
linux                    1
Name: platforms, dtype: int64
```

In [8]:

```
plt.figure(figsize = (12,8))
steam["platforms"].hist(bins=15)
save_fig("platforms_hist")
plt.show()
```

```
Saving image platforms_hist
```



Ok, by that we can see that this attribute consists of categorical values separated by semicolon, in the next steps we'll change it into indicator variables. Lastly, we can check the ratio between english and non-english games (english ones being 1, others 0).

In [9]:

```
steam["english"].value_counts()
```

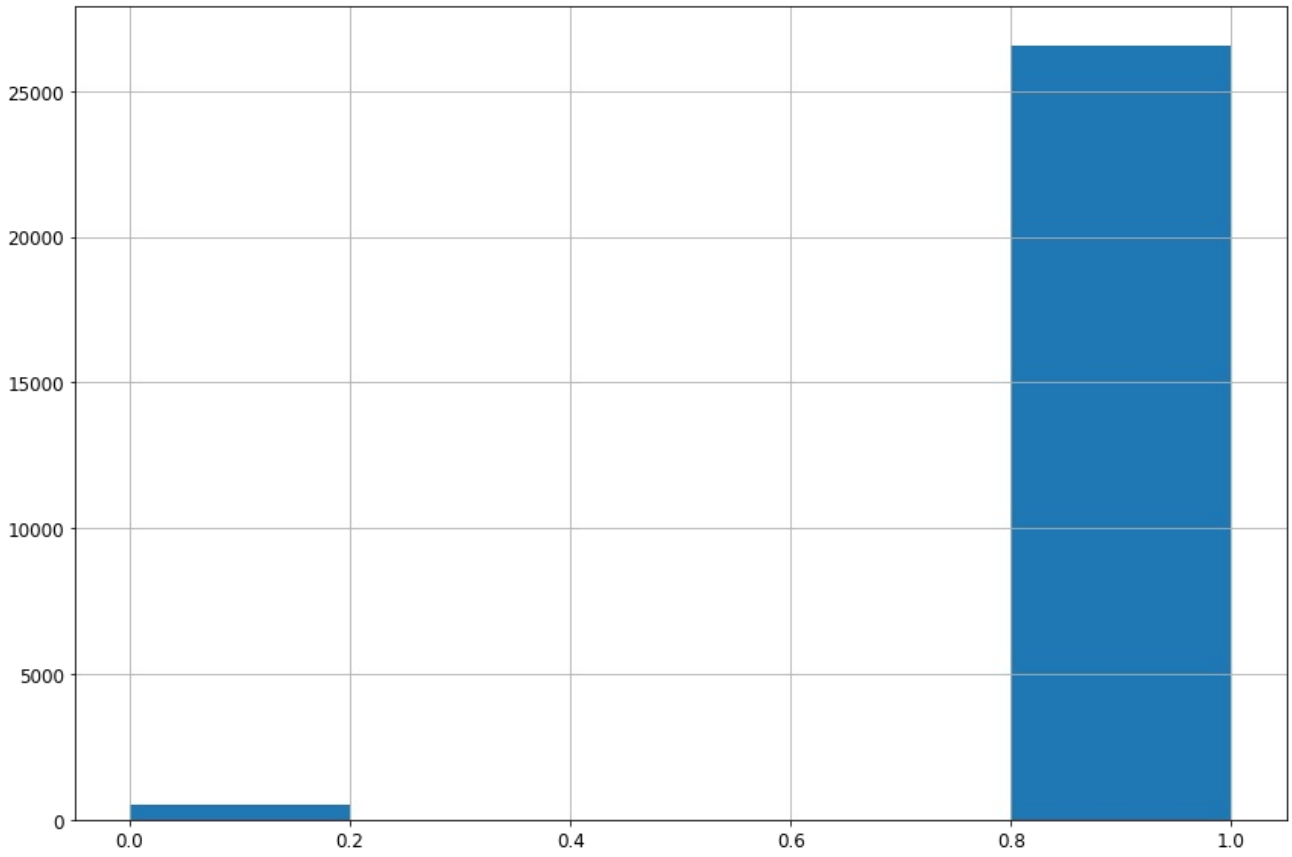
Out[9]:

```
1    26564
0      511
Name: english, dtype: int64
```

In [10]:

```
plt.figure(figsize = (12,8))
steam["english"].hist(bins=5)
save_fig("english_hist")
plt.show()
```

```
Saving image english_hist
```



We can see there's only 511 non-english games in our dataset. Let's end the first phase here and continue work on our dataset after encoding string values into numerical ones.

Encoding the data and searching for correlations

Seeing how now we start to manipulate the data best practice to do so is to work on the copy of the data, let's do exactly that and after that delete some of the attributes that won't be important for us. After a long discussion we decided that we won't need:

1. appid - it does not contain any relevant information from the machine learning point of view,
2. name - non-categorical value, it would be hard to extract some interesting information from it,
3. steamspy_tags - redundant information, same can be found in categories and genres attributes,
4. price - non-relevant information, especially at the date of the premier.

Then we used **drop()** function from Pandas, it deletes the column specified in its parameters, in our case:

In [11]:

```
steam_cp = steam.copy()
steam_cp = steam_cp.drop(columns=["appid", "name", "steamspy_tags", "price"])
```

The next step is to encode object attributes that are important to us, we could use objects provided by the Pandas library like **LabelEncoder**, but we decided to define our own function, that will encode the entire column and also create a dictionary where we will store our labels with their keys. The function is shown below.

In [12]:

```
def encode_series(series):
    col = pd.Series(steam_cp[series])
    dictionary = {}
    enc_series = []
    i = 0
    for c in col:
        if c not in dictionary.keys():
            dictionary[c] = (i, c)
            i = i + 1
        x = dictionary[c][0]
        enc_series.append(x)
    return enc_series, dictionary
```

And then with the help of the above function we encoded the following columns: developer, publisher and owners, along with assigning some keys to its lables. The purpose of this action is to be able to easily find values of these columns and its numerical equivalents.

In [13]:

```
enc_dev, dev_dict = encode_series('developer')
enc_publ, publ_dict = encode_series('publisher')
enc_own, own_dict = encode_series('owners')
```

For future references we might need to save the produced data in a file for further analysis in later stages of the project development. The method below saves the resulting dictionaries in a specified folder.

In [14]:

```
def save_encoded_dict_to_file(dest_filepath, data_to_write):
    with open(dest_filepath, 'w') as f:
        out = json.dumps(data_to_write)
        f.write(pp.pformat(out))
```

The next columns that had to be encoded were genres and categories. However, in this case we had to use another function to encode them. Because the data was saved in the following way: dataA;dataB;dateC... we had to separate this data and create columns from it. But before this process, we checked what new, potential columns are to be created when we did it.

In [15]:

```
def extract_values_in_columns(column):
    col = pd.Series(steam_cp[column])
    col_value_list = []
    col_value_set = set()

    for c in col:
        col_value_list.append(c.split(";"))

    for row in col_value_list:
        for r in row:
            col_value_set.add(r)

    return col_value_set
```

Then we used this function and created sets that consisted of categorical values of the *genres* and *categories* columns.

In [16]:

```
genres_set = extract_values_in_columns("genres")
cats_set = extract_values_in_columns("categories")
```

First let's check *genres* structure.

In [17]:

```
genres_set
```

Out[17]:

```
{'Accounting',
 'Action',
 'Adventure',
 'Animation & Modeling',
 'Audio Production',
 'Casual',
 'Design & Illustration',
 'Documentary',
 'Early Access',
 'Education',
 'Free to Play',
 'Game Development',
 'Gore',
 'Indie',
 'Massively Multiplayer',
 'Nudity',
 'Photo Editing',
 'RPG',
 'Racing',
 'Sexual Content',
 'Simulation',
 'Software Training',
 'Sports',
 'Strategy',
 'Tutorial',
 'Utilities',
 'Video Production',
 'Violent',
 'Web Publishing'}
```

And then *categories*.

In [18]:

```
cats_set
```

Out[18]:

```
{'Captions available',
 'Co-op',
 'Commentary available',
 'Cross-Platform Multiplayer',
 'Full controller support',
 'In-App Purchases',
 'Includes Source SDK',
 'Includes level editor',
 'Local Co-op',
 'Local Multi-Player',
 'MMO',
 'Mods',
 'Mods (require HL2)',
 'Multi-player',
 'Online Co-op',
 'Online Multi-Player',
 'Partial Controller Support',
 'Shared/Split Screen',
 'Single-player',
 'Stats',
 'Steam Achievements',
 'Steam Cloud',
 'Steam Leaderboards',
 'Steam Trading Cards',
 'Steam Turn Notifications',
 'Steam Workshop',
 'SteamVR Collectibles',
 'VR Support',
 'Valve Anti-Cheat enabled'}
```

After that we used the function **"get_dummies()"**, which splits strings in the Series and returns a DataFrame object of the "dummy" variables. We decided to test it on the **categories** attribute and use semicolon as the separator, the results are shown below.

In [19]:

```
steam_cp["categories"].str.get_dummies(sep=';')
```

Out[19]:

	Captions available	Co-op	Commentary available	Cross-Platform Multiplayer	Full controller support	In-App Purchases	Includes Source SDK	Includes level editor	Local Co-op	Local Multi-Player	...	Stats	Ach
0	0	0	0	0	0	0	0	0	0	1	...	0	
1	0	0	0	0	0	0	0	0	0	1	...	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	
3	0	0	0	0	0	0	0	0	0	1	...	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
27070	0	0	0	0	0	0	0	0	0	0	...	0	
27071	0	0	0	0	0	0	0	0	0	0	...	0	
27072	0	1	0	0	1	0	0	0	0	0	...	0	
27073	0	0	0	0	0	0	0	0	0	0	...	0	
27074	0	0	0	0	0	0	0	0	0	0	...	0	

27075 rows × 29 columns

Next step is to create a **Rating** attribute for our new dataset. To do that we used **positive_ratings** and **negative_ratings** and calculated from them the percentage of the positive ones from the total number of ratings. We decided that the ratings will resemble the Metacritic or RottenTomatoes ones (value from 0 to 100). The results of the equation were stored in the **ratings** list that was later assigned as the attribute of the new DataFrame object with the same name.

In [20]:

```
pos = pd.Series(steam_cp['positive_ratings'])
neg = pd.Series(steam_cp['negative_ratings'])
ratings = []
for p, n in zip(pos, neg):
    total = p + n
    i = np.round((p/total)*100)
    ratings.append(i)
```

Next step was to encode **release_date**. It could be problematic but we decided, to split this column into two separate ones: **Month** (month of release) and **Year** (year of release). To do this, first we changed the type of the release_date with the help of **to_datetime()** function with our Series as the first parameter and the format of the date as the second one. Then in the loop we extracted months and years values into separate lists, that were later assigned as the Series objects of the newly created DataFrame object.

In [21]:

```
dates = pd.Series(steam_cp['release_date'])
dates = pd.to_datetime(dates, format = '%Y-%m-%d')
months = []
years = []

for i in dates:
    d = i.month
    months.append(d)
    y = i.year
    years.append(y)
```

Ok, now we're ready to create a new DataFrame, without much explanation, because most of these operations have been already explained. Let's add them and check the structure of the object.

In [22]:

```
def produce_encoded_dataframe(relevant_attributes, attributes_to_concat, raw_data, value_sep=';'):
    encoded_data = pd.DataFrame()
    for attr in attributes_to_concat:
        encoded_data = pd.concat([encoded_data, raw_data[attr].str.get_dummies(sep=value_sep)], axis=1)

    for attr in relevant_attributes.keys():
        encoded_data[attr] = relevant_attributes[attr]
    return encoded_data
```

In [23]:

```
relevant_attributes = {'Month': months, 'Year': years,
                           'English': steam_cp['english'], 'Developer': enc_dev,
                           'Publisher': enc_publ, 'Required_Age': steam_cp['required_age'],
                           'Achievements': steam_cp['achievements'], 'Average_Playtime': steam_cp['average_playti
me'],
                           'Median_Playtime': steam_cp['median_playtime'], 'Rating': ratings, 'Owners': enc_own}
attributes_to_concat = ['genres', 'categories', 'platforms']
```

In [24]:

```
enc_data = produce_encoded_dataframe(relevant_attributes, attributes_to_concat, steam_cp)
enc_data.head(8)
```

Out[24]:

	Accounting	Action	Adventure	Animation & Modeling	Audio Production	Casual	Design & Illustration	Documentary	Early Access	Education	...	Y
0	0	1	0	0	0	0	0	0	0	0	...	2
1	0	1	0	0	0	0	0	0	0	0	...	1
2	0	1	0	0	0	0	0	0	0	0	...	2
3	0	1	0	0	0	0	0	0	0	0	...	2
4	0	1	0	0	0	0	0	0	0	0	...	1
5	0	1	0	0	0	0	0	0	0	0	...	2
6	0	1	0	0	0	0	0	0	0	0	...	1
7	0	1	0	0	0	0	0	0	0	0	...	2

8 rows × 72 columns

Having a DataFrame that only has numerical values, we were able to look for correlations. The first step is to create a correlation matrix. To do that we used the **corr()** function of the DataFrame object. We used it on our DataFrame and checked how it looked.

In [25]:

```
corr_matrix = enc_data.corr()
corr_matrix
```

Out[25]:

	Accounting	Action	Adventure	Animation & Modeling	Audio Production	Casual	Design & Illustration	Documentary	A
Accounting	1.000000	-0.013187	-0.006285	0.137206	0.227089	0.008895	0.174531	0.408211	0.0
Action	-0.013187	1.000000	0.017816	-0.047915	-0.029004	-0.167298	-0.050290	-0.005383	0.0
Adventure	-0.006285	0.017816	1.000000	-0.040086	-0.025123	-0.034569	-0.042209	-0.004663	-0.0
Animation & Modeling	0.137206	-0.047915	-0.040086	1.000000	0.144767	-0.030788	0.650291	-0.000329	0.0
Audio Production	0.227089	-0.029004	-0.025123	0.144767	1.000000	-0.016159	0.117849	-0.000199	0.0
...	...	...	...	...	...	...	...	...	
Achievements	-0.001896	0.016220	-0.029684	-0.006617	-0.003987	0.076451	-0.006906	-0.000763	-0.0
Average_Playtime	-0.001221	-0.002806	0.000778	-0.001826	-0.002645	-0.027567	-0.001168	-0.000498	-0.0
Median_Playtime	-0.000924	-0.010399	0.003395	-0.001599	-0.002001	-0.016380	-0.001078	-0.000377	-0.0
Rating	-0.009209	-0.017347	-0.002379	0.005604	-0.012715	-0.012720	0.009429	-0.005580	-0.0
Owners	0.002168	-0.022248	0.011626	0.004003	0.008701	0.117130	0.003155	0.003531	0.0

72 rows × 72 columns

Well, a correlation matrix has been created but it's not very readable, we checked how it looked for attributes that are the most interesting to us, Owners and Rating.

In [26]:

```
corr_matrix["Rating"].sort_values(ascending=False)
```

Out[26]:

```
Rating                     1.000000
Steam Cloud                0.182635
Steam Achievements         0.160252
Full controller support    0.140914
mac                        0.122155
                               ...
MMO                       -0.051557
In-App Purchases          -0.053645
Violent                   -0.063875
Massively Multiplayer     -0.072636
Simulation                -0.113912
Name: Rating, Length: 72, dtype: float64
```

In [27]:

```
corr_matrix["Owners"].sort_values(ascending=False)
```

Out[27]:

```
Owners                  1.000000
Year                    0.392090
Developer               0.385578
Publisher               0.362980
Indie                   0.127370
                             ...
Steam Cloud            -0.120236
In-App Purchases       -0.127851
Multi-player           -0.145272
Free to Play           -0.189014
Steam Trading Cards    -0.298738
Name: Owners, Length: 72, dtype: float64
```

Ok, we can see that for the Rating attribute, the strongest correlation was shown between genres and categories values. And for the owners it was the year of release, developers and publishers. Great, DataFrame prepared in that way is ready to be proccesed with the machine learning algorithms. The last step is to save this DataFrame into a new file that can be used in the future.

In [28]:

```
enc_data.to_csv('../data/enc_steam.csv')
```

Here's what the first phase of our project looks like. The processes and tests performed in it was necessary to create valid data which now are devoid of unnecessary and repetitive values. Data frame prepared in this way is able to be used in machine learning. In the next phase, we will try to find a model that will have the greatest learning predispositions and then we will begin to teach our network.