2021

PROJECT REPORT

VIDEO GAMES HIT PREDICTOR

BORJA DOMÍNGUEZ NAKAMURA

Content

1. Introduction

1.1. Motivation

The video games industry has been in a constant evolution since its birth. Technological advancements helped this ever-shifting environment by accelerating the development of new consoles and video game products.

If there is anything characteristic in this industry apart from its fast-evolving nature, is **its uncertainty** tied to this aspect.

The development cost to pay up-front is very high (equipment, software licenses, etc.). Independent developers and small studios find it nearly impossible to pay this cost, as traditional lenders do not usually invest in these kinds of projects unless you already have a successful product. This sometimes leads them to:

- Work for free until they get funding.
- Get a loan to pay the costs.
- Get financial help from the so-called "family, friends and fools".
- Get another job to have an income.
- Etc.

Other non-traditional lenders in the industry are the publishers. Publishers can fund developers and can also do the marketing campaign for the product. However, to get funds from them you need to:

- **Prepare a working demo** featuring at least the most important elements and functionalities of the final product. Usually, it is required for developers to send it to the publishing company so that they test it themselves.

- **Send the** game's **GDD** for them to read and understand the game's concept and its design details.

    - A Game Design Document is a document featuring every single design aspect of the final game, from the concept to every mechanic. It is the "game's bible".

    - Studios develop their game based on it, which is not set in stone and can change if there is anything that is not working after some iterations. But any change must be reflected in this document.

- **Send the roadmap**.

- **Do a presentation** (live or via video call) so that the publisher can ask questions, make suggestions, share insights and opinions about the product, etc.

- **Make and send them a budget**. Here you should consider salaries, licenses, equipment cost, localization, marketing (if the studio is doing it), etc.

- **Negotiate the terms** of the funding contract (if it gets to that stage), in which publishers will agree to give the funds in exchange of a percentage of the sales made.
    - **Usually, this percentage is quite high** (70-80%) as it is the publisher who is risking the money with an unfinished product.
    - If the publisher is also helping with the promotion, this could be another reason for them to keep this percentage high.
- **Be patient**, as this is a process that **can take months**.

As we can see, the process of being in a conversation with a publisher can get very long. Studios also have to consider that publishers will not always be interested in their project (due to lack of polish; an unclear concept; a not good enough GDD; an unrealistic roadmap; etc.). This means that developers have to repeat this process n times until they get the fund, or they decide to either go for an alternative option (Kickstarter/ Indiegogo) or just cancel the project, losing the money put up-front.

This project, the **Video Games Hit Predictor** tries to help in this process so that it takes less time for them to get the funding or cancel the project (the longer it takes, the higher the amount of money lost).

1.2. Previous related work/ state of the art

There are many "Predict Video Games Hits" project on the internet, but they have something in common:

They are using the same dataset from VGChartz, scraped in different moments. The problem with using only this dataset is the following:

- VGChartz has **rough estimates of sales**, but it is not very accurate.
    - Many of them are missing.
    - They also do not support Steam games or games in online PC platforms.
- This dataset **lacked some information** that should be important when analyzing hits, such as:
    - Game genre
    - Additional information not tied to the genre (multiplayer or not; online or not; etc.)

- How well the game got received in the form of scores and positive/ negative feedback received.

  Now, VGChartz has scores indeed, but it is not a well-known web site and we do not know where these scores come from.

- The target user's age range (is it a Everyone 10+ game or an Adults 18+ game?)

This said, although VGChartz has rough estimates on sales, it is also true that **there is no other way of getting this information**. Thus, we are also going to use this dataset, but we are using two extra datasets for the information this dataset is lacking. We are going to get the datasets from:

- RAWG for the general information
- Metacritic for the scores and feedbacks

1.3. Objective

To be able to help developers know if their project can stand a chance, we have to decide what do we want out frontend to do.

Getting only a "Yes, your game will be a hit" after introducing the input data feels lackluster. Thus, we want the frontend to tell users the following:

- If the game will be a hit or not.
- How many units are expected to get sold in the first year.

To achieve the first one, we have to train a **classification model.** Furthermore, we must decide which one of **recall** and **precision** is more important.

In our case, precision is the most important metric, as false positives mean losing money and time for both publishers and developers, and even over-investing in the hopes of getting a return after.

That means that false negatives will be there. But we can also make the frontend to tell users the game's likelihood of becoming a hit, using *clf.predict_proba()* instead of just *clf.predict()*.

To achieve the second one, we have to train a regression model. As video games sales usually have a very wide range of sales (many outliers), we should consider **RMSE** as the go-to metric.

Also, prompting the estimated units sold to users may help them interpret the prediction even though the classification model said <50% chance of becoming a hit, as **we will define hit as a game selling more than 1,000,000 units its first year after the release**.

Another important thing to highlight is that **we will only focus on PC and console games, as mobile games and browser games work in a completely different way.**

2. Raw data description

We are going to explain the semantics used in our datasets separated in subsections.

2.1. Vgsales.csv

Column name	Description
Unnamed: 0	Index column.
Title	Title of the games.
Platform	Platform the game was released on.
Publisher	Name of publisher that published the game.
Developer	Developer of the game.
VGC_Score	A score unique from VGC.
Critic_Score	A score given by the critics.
User_Score	A score given by VGC users.
Total_Shipped	Total number of units shipped.
Total_Sales	Total number of units sold.
NA_Sales	Units sold in North America.
EU_Sales	Units sold in Europe.
JP_Sales	Units sold in Japan.
Other_Sales	Units sold in the rest of the world.
Release	Release date.
Last_Update	Last update on the game's data.

This is the scraped data from VGChartz after running ***01_VGC_Scraping***.

2.2. Rawg_data (both regular and small version)

Column name	Description
id	The game's id.
slug	The game's title in lower case and spaces replaced by '-'.
name	Game's title
released	Release date of the game.
tba	If the game is yet to be announced.
metacritic	Metacritic score.
suggestions_count	Number of suggestions for the game in RAWG.
updated	Last update date in the data.
platforms	Platforms the game is available in.
genres	Game genre.
stores	Digital stores the game can be purchased on.
tags	Extra information given by tags in RAWG.
esrb_rating	The ESRB rating for the game.

This is the scraped data from RAWG after running ***02_RAWG_API_Request*** and ***03_RAWG_Downscale***.

2.3. 01_vgc_clean.csv (also 01_vgc_ips.csv)

Column name	Description
Title	Title of the games..
Platform	Platform the game was released on.
Publisher	Name of publisher that published the game.
Developer	Developer of the game.
Release	Release date.
Sales	Total units sold.

This is one of the output datasets we get after running ***01_VGC_RAWG_Exploration***.

2.4. 01_rawg_clean.csv

Column name	Description
slug	The game's title in lower case and spaces replaced by '-'.
name	Game's title
released	Release date of the game.
suggestions_count	Number of suggestions for the game in RAWG.
platforms	Platforms the game is available in.
genres	Game genre.
stores	Digital stores the game can be purchased on.
tags	Extra information given by tags in RAWG.
esrb_rating	The ESRB rating for the game.

This is the other output dataset we get after running ***01_VGC_RAWG_Exploration***.

2.5. 02_rawg_metacritic_url.csv

Column name	Description
slug	The game's title in lower case and spaces replaced by '-'.
name	Game's title
released	Release date of the game.
metacritic	Metacritic score.
suggestions_count	Number of suggestions for the game in RAWG.
platform	Platform the game is available in.
genres	Game genre.
stores	Digital stores the game can be purchased on.
tags	Extra information given by tags in RAWG.
esrb	The ESRB rating for the game.
plat_mc	The platform column remapped to Metacritics' way of calling them.
url	URL to the game's page in Metacritic for the platform in plat_mc.

This is the dataset we get after running ***02 _RAWG_Processing***, prepared to web scrape Metacritic.

2.6. 03_metacritic.csv

Column name	Description
url	The URL of the game's page in Metacritic.
critic_score	Score given by the critics (out of 100).
critic_reviews	Number of reviews by critics.
critic_pos	Number of positive feedback from critics.
critic_mix	Number of mixed feedback from critics.
critic_neg	Number of negative feedback from critics.
user_score	Score given by the users (out of 10).
user_reviews	Number of reviews by the users.
user_pos	Number of positive feedback from the users.
user_mix	Number of mixed feedback from the users.
user_neg	Number of negative feedback from the users.

This is the scraped data from Metacritic after running ***03_Metacritic_Scraping***.

2.7. 04_merged_data.csv

Column name	Description
Title	Game title.
Platform	Platform the game is available in.
Publisher	Name of publisher that published the game.
Developer	Developer of the game.
Release	Release date.
Sales	Total units sold.
Suggest_count	Number of suggestions for the game in RAWG.
Genres	Game genre
Stores	Digital store the game can be purchased on.
Tags	Extra information given by tags in RAWG.
ESRB	The ESRB rating for the game.
C_Score	Score given by the critics (out of 100).
C_Reviews	Number of reviews by critics.
C_Positive	Number of positive feedback from critics.
C_Mixed	Number of mixed feedback from critics.
C_Negative	Number of negative feedback from critics.
U_Score	Score given by the users (out of 10).
U_Reviews	Number of reviews by the users.
U_Positive	Number of positive feedback from the users.
U_Mixed	Number of mixed feedback from the users.
U_Negative	Number of negative feedback from the users.

This is the output dataset after running ***04_VGC_RAWG_Metacritic_Processing***, which combines all three scraped and cleaned data into one single dataset.

2.8. 05_encoded.csv

Column name	Description
Title	Game title.
Publisher	Name of publisher that published the game.
Developer	Developer of the game.
Hit	If the game sold more than 1M units in the first year or not.
Sales_total	Total units sold in the first year.
Suggest_count	Number of suggestions for the game in RAWG.
P_******	One-hot encoded platform columns.
C_Positive	Number of positive feedback from critics.
C_Mixed	Number of mixed feedback from critics.
C_Negative	Number of negative feedback from critics.
U_Score	Score given by the users (out of 10).
U_Reviews	Number of reviews by the users.
U_Positive	Number of positive feedback from the users.
U_Mixed	Number of mixed feedback from the users.
U_Negative	Number of negative feedback from the users.
G_******	One-hot encoded game genre columns.
S_******	One-hot encoded digital store columns.
ESRB_******	One-hot encoded ESRB rating columns.
T_******	One-hot encoded tag columns.
Release_Y	Release year
Release_M	Release month

This is the output dataset after running ***05_Encoding***, which applies several transformations to the previous dataset and one-hot encodes some of the columns.

2.9. 06_prepared_for_ml.csv

Column name	Description
Title	Game title.
Publisher	Name of publisher that published the game.
Developer	Developer of the game.
Hit	If the game sold more than 1M units in the first year or not.
Sales_total	Total units sold in the first year.
Suggest_count	Number of suggestions for the game in RAWG.
P_******	One-hot encoded platform columns.
G_******	One-hot encoded game genre columns.
S_******	One-hot encoded digital store columns.
ESRB_******	One-hot encoded ESRB rating columns.
T_******	One-hot encoded tag columns.
Release_Y	Release year
Release_M	Release month
Publisher_enc	Target-encoded top 30 publishers.
Developer_enc	Target-encoded top 30 developers.
Scores	Mean of the sum of critic and user scores.
Positives	Number of total positive feedbacks.
Negatives	Number of total negative feedbacks.

This is the output dataset we get from running ***06_ML***. The dataset has made further transformations and target encoded Publisher and Developer columns.

This is the dataset we are going to use to train the models with.

3. Important decisions made

3.1. In the 01_VGC_RAWG_Exploration notebook

3.1.1. For the VGC dataset

In this dataset we decided to drop *Last_Update* and *VGC_Score* as they are not going to use them, and *Critic_Score* and *User_Score*, as we are going to get the data from Metacritic (it is the most important web site whenever we are talking about video games scores).

We also transformed *Release* so that it kept the DD-MM-YYYY format.

The most important decision taken in this notebook is dropping all sales-related columns but *Total_Shipped* and *Total_Sales, which got combined*. We decided to do this because *Total_Sales* is the sum of the dropped columns (and we are only interested in the total units sold).

Furthermore, *Total_Sales* had many missing values, so we checked *Total_Shipped* if there was a relationship between them. The result is that whenever there is a value in *Total_Sales, Total_Shipped* had no values there and vice-versa.

We wanted to fill as many missing values with real values as possible, so we decided to sum both columns and namig the result column *Sales*.

Another important decision made is separating all data points containing "Series" in the *Platform* column from the dataset. "Series" meant that the data point correspond to the IP, the "brand" (Pokémon, Fallout, FIFA, etc.), so its sales was the accumulation of all game titles falling inside the same IP.

3.1.2. For the RAWG dataset

In this dataset we decided to drop *id*, *tba*, *updated* and *metacritic*. **We also dropped all data points whose release date is the same or after to the scraping date of VGChartz**, as they should not be in the VGC dataset anyway.

We kept *slug* because we realized metacritic urls had the following format:

https://www.metacritic.com/game/PLATFORM/SLUG

3.2. In the 02 _RAWG_Processing notebook

Here, we decided to drop rows meeting one of the following conditions:

- *platforms* contain a missing value.
- *platforms* is *Android* or *iOS*. -> Because we do not want mobile games.
- *stores* contains only *itch.io*. -> Because VGChartz does not have data for online platforms for PC.

The most important thing done in this notebook is the unpacking of the *platforms* column. We did this due to Metacritic having the scores for the same game separated by platforms. So, we created a new dataframe in which games have only one platform associated to them (and with as many copied rows as platform it had in the previous dataframe).

We used the newly created dataframe's *platform* column to create a new column with each game's URL to Metacritic.

3.3. In the 04_VGC_RAWG_Metacritic_Processing notebook

In this notebook we merged all three previous datasets into a single dataset so that we can work on it for preprocessing and training.

First, we merged *01_rawg_clean* with *03_metacritic* as they shared the columns with the URLs.

Second, we merged the result of the prior step with *01_vgc_clean* by using the *platform* column. To do this, we had first to remap the platforms so that they had the same names to avoid creating two or more instances of the same game with the same platform in different format.

3.4. In the 05_Encoding notebook

Many important decisions have been taken in this notebook, the first one of them being how to one-hot encode *Genres*, *Stores* and *ESRB*, as they do not contain that many values.

Genres and *Stores* have lists in them, so we cannot just apply *pd.get_dummy()* to achieve our purpose. *ESRB*, on the contrary, has single values, thus applying *pd.get_dummy()* is what we will do.

To one-hot encode columns containing lists, we have to use ***MultiLabelBinarizer*** from the *preprocessing* module from the *sklearn* library.

The next important decision taken is: **what to do with *Platforms*?**

There are too many consoles to just one-hot encode, but we need to convert them into numeric somehow. We could also consider target encoding, but a game can be available in multiple platforms so it would not work.

What we decided to do is simple: **we converted the platforms having the same manufacturer to one and saved them into a new column called *Platform_Group*.**

Of course, it would also cause problems due to old consoles having many different manufacturers. Thus, we only considered All, PC, Sony, Nintendo, Microsoft and Other (includes the rest).

All was neither dropped nor merged into Other because there are some games having only this platform and we should study what to do with them. There are also games with missing values in *Platforms*. Whatever transformation we would do to *All* could also be applied to these values.

After checking values**, we decided to fill all missing values and replace *All* in *Platforms* with different platform group depending on their release year**.

To do this, **we separated release years by periods in the video games history**.

- ~1972 : No main consoles existed.
- 1973-1983 : Corresponds to the 1st and 2nd generations of consoles.
- 1983-1993 : Corresponds to the 3rd and 4th generations of consoles.
- 1993-1999 : Corresponds to the 5th generation of consoles.
- 1999-2005 : Corresponds to the 6th generation of consoles.
- 2005-2012 : Corresponds to the 7th generation of consoles.
- 2012-2020 : Corresponds to the 8th generation of consoles.
- 2020~ : Corresponds to the 9th generation of consoles.

Then we checked the top 2 values from every period and if the top value is more than twice as big than the second value (if it is true, we only keep the top value). These are the values we are going to fill missing values and replacing *All*.

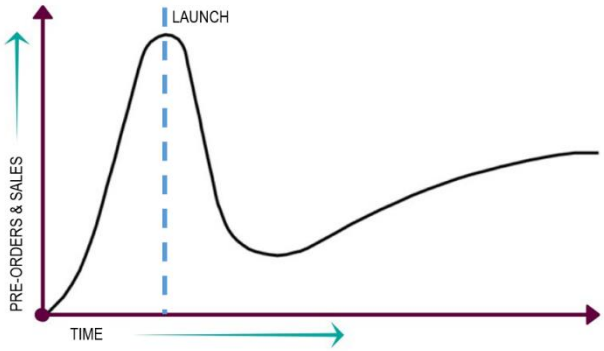
And now, **what to do with *Tags*?**

Tags have the same problem Platforms had: too many values to one-hot encode and we cannot consider target encoding either because a game can have multiple tags.

Then, the decision we took is **to keep only the top 20 tags as our one-hot encoded columns**. In the process we replaced some tags with other similar ones to get as many useful tags as possible.

After that**, we encoded *Release* by separating the year and the month and storing them into two separate columns as integers**.

Now, **we decided to transform *Sales* because this column contained the accumulated sales over the years**. Sales in video games have approximately this shape:



Thus, we are going to create a function to estimate the sales in the first year, as what we have is the accumulative sales over the years.

The sales in the first year have a certain area below the curve, which means that at some point, the "tail" will accumulate enough sales to be comparable to the first year (and may happen more than once).

So, taking this into account, the estimator function could look like something like this:

$$Sales_{1st_{year}} = \begin{cases} \dfrac{Sales_{total}}{1 + \dfrac{(\Delta Release - 1)}{\gamma}} & if\ \Delta Release < 10 \\ \dfrac{Sales_{total}}{\dfrac{\gamma + 9}{\gamma}} & if\ \Delta Release \geq 10 \end{cases} -$$

- $Sales_{total}$: It is the current *Sales* column.
- $\Delta Release$: It is the difference between **current year** and *Release_Y*.
    - The -1 is to filter out the first year, which is already considered in the 1+ side of the denominator.
    - If the delta is greater than 10 years, we will fix delta's value to 10. This is because it is highly unlikely a product accumulates enough sales after 10 years to be relevant.
    - The scraping process needs to be improved further as there are still problems to automate the scraping process due to issues such as error 503, taking too long to gather some data (4 days to scrape Metacritic on 2021). We will consider 2021 as the upper limit for now.
- γ: It is the number of years we estimate for accumulative sales to be comparable to the first year. **It is totally arbitrary** as we don't know what the correct value is but we assume that 5 is the value.

Once we performed all the transformations, **now we have to try converting rows with the same *Title* to a single row**. To do so, we will use the *groupby()* and *agg()* methods.

We are going to group by *Title* first:

- ***Publisher* and *Developer*** should only keep one value, so we are getting the mode among all the values they have and save the first position in case there were more than one mode.

Then we are going to group by *Title* and *Platform_Group*:

- ***Sales*** within the same *Platform_Group* should get added.
- ***Suggest_count*** is the same for every row with the same *Title*, so we can get any value (in this case we use max for aggfunc)
- **After this, we are one-hot encoding *Platform_Group*.**

Then we group this dataframe by *Title*:

- We define a *Hit* column where we have 1 if it is a hit (Sales_total >= 1)and 0 otherwise.
- *Sales_total* will sum the sales within the games with same title.
- Suggest_count : we are performing the same operation as before.
- And create renamed columns for the one-hot encoded *Platform_Group* columns.

Now we group the rest of the columns by *Title*:

- *Scores* (for both *C_* and *U_*): we will keep the max value.
- *Positive*, *Mixed* and *Negative* (for both *C_* and *U_*): we will get the total values.
- One-hot encoded columns: we only need to keep the max value as they only store 0s and 1s.
- *Release_Y* and *Release_M*: we will keep the min value, as it would be the earliest release.

And finally merge the three dataframes into one.

3.5. In the 06_ML notebook

In this notebook we have taken the following important decisions:

- Before target encoding the top 30 *Publisher* and *Developer* values, we corrected redundancies by converting them into one single label.
    - We have also kept the top 29 values and combined the rest under *Other* for both *Publisher* and *Developer*.
- **For missing values found in Suggestion_count, C_Score and U_Score, we decided to check its *Sales_total* column and filling them with either of among their minimum, quantile 25%, quantile 50%, quantile 75% or maximum value**.

- We also studied the correlation between the target variables and decided to make the following transformations based on it:

    - **We are going to combine both Scores, both Positives and both Negatives and save them as *Scores*, *Positives* and *Negatives***.

    - We are going to drop C_Mixed and U_Mixed.

    - **We are going to drop all *S_* columns** as the most important of them (*S_Steam*, as it is the major online platform for PC) has high correlation with a major tag and the rest of options in S_ do not seem to provide much information.

- After the feature selection, we started to test different models for both classification and regression. To do so, we did the *train_test_split* and applied standardization on them. We did this as there are estimators based on distances among the candidates and it affects their output.

- After checking our candidates' metrics (individual and k-fold cross validated), **we decided to use XGBoost for both classification and regression** due to having an overall good performance.

- After deciding which models to use, **we have begun with the hyperparameter optimization using GridSearchCV** to find the optimal hyperparameters **to maximize precision** (we also included f1 but the dominant metric is precision) **for classification and to minimize MRSE for regression**.

3.6. In the frontend

For the frontend, there are some features that are hard or nearly impossible to get by the user such as *Scores*, *Positives*, *Negatives* and *Suggest_count* because those are properties only published games have due to the need of evaluation by the critics and users. However, if we can abstract the ideas these features represent we could let them introduce values so that they do not remain as 0, which can negatively impact the result.

- *Scores*: This is hard to abstract. As we do not know how good the game will be, we can either set the average/median value as default (after filtering the 0s out).

- *Positives/ Negatives*: The idea behind these features is the **amount of feedback received**, thus we can make the user introduce the amount of feedback they received for the game.

- *Suggest_count*: Suggestions affect sales as they increase the interest on the product. Thus, if we consider this feature a measure of interest, we could also consider followers the studio/ game has in social media, as they are usually interested.

4. Summary of main results

In this section, we are going to share screenshots of the metrics we have gotten in the model selection and hyperparameter optimization. **For the sake of replicability, we used *random_state = 27* in the train_test_split.**

4.1. Classification

XGBoost Classifier metrics prior to hyperparameter tuning using GridSearchCV():

```
================================================================================
XGBClassifier
================================================================================
    Accuracy : 0.963363   |  CV : 0.962345 (avg) [0.957671, 0.966084]
   Precision : 0.735385   |  CV : 0.695108 (avg) [0.628099, 0.743191]
      Recall : 0.481855   |  CV : 0.456744 (avg) [0.401055, 0.503958]
    F1 Score : 0.582217   |  CV : 0.550971 (avg) [0.489533, 0.600629]
     ROC AUC : 0.736077   |  CV : 0.956500 (avg) [0.952442, 0.961517]

[[8780   86]
 [ 257  239]]
```

XGBoost Classifier metrics after hyperparameter tuning:

```
================================================================================
XGBClassifier
================================================================================
    Accuracy : 0.956633   |  CV : 0.958179 (avg) [0.955401, 0.959808]
   Precision : 0.812500   |  CV : 0.773391 (avg) [0.682927, 0.861111]
      Recall : 0.235887   |  CV : 0.249482 (avg) [0.221636, 0.306069]
    F1 Score : 0.365625   |  CV : 0.375768 (avg) [0.334661, 0.431227]
     ROC AUC : 0.616421   |  CV : 0.935852 (avg) [0.926240, 0.948073]

[[8839   27]
 [ 379  117]]
```

Comparing the same classifier before and after the tuning, we see that our average precision got increased by 8% at the cost of decreasing the rest of metrics.

We may want to use the untuned version of the model rather than the tuned version.

We can leave the option to change the estimator in the frontend app.

Let us also compare metrics for regression.

4.2. Regression

XGBoost Regressor metrics prior to hyperparameter tuning using GridSearchCV():

```
========================================================================
XGBRegressor
========================================================================
     MSE : 0.902618   |  CV : 0.845306 (avg) [0.411064, 2.045943]
    RMSE : 0.950062   |  CV : 0.874248 (avg) [0.641143, 1.430365]
     MAE : 0.166277   |  CV : 0.160356 (avg) [0.138990, 0.188633]
    MAPE : 0.465886   |----------------------------------------
      R2 : 0.784669   |  CV : 0.608575 (avg) [0.397289, 0.794949]
      EV : 0.784671   |  CV : 0.608623 (avg) [0.397358, 0.794976]
```

XGBoost Regressor metrics after hyperparameter tuning:

```
========================================================================
XGBRegressor
========================================================================
     MSE : 1.094383   |  CV : 0.736004 (avg) [0.416865, 1.572083]
    RMSE : 1.046128   |  CV : 0.828633 (avg) [0.645651, 1.253827]
     MAE : 0.179758   |  CV : 0.165879 (avg) [0.144873, 0.187333]
    MAPE : 0.504337   |----------------------------------------
      R2 : 0.738921   |  CV : 0.644251 (avg) [0.446196, 0.789491]
      EV : 0.738973   |  CV : 0.644411 (avg) [0.446566, 0.789557]
```

In this case, our average RMSE got decreased and both average R^2 and explained variance values got increased, so we could use the tuned regressor over the untuned version.

5. Conclusions

Our conclusion is that **our model is not good enough**:

- After exporting the model and doing some testing using the frontend application putting combinations that should have gotten >50% chance of becoming a hit (because it was), they failed.

- We may want to use the untuned classificator as increasing 7% of precision at the cost of losing 20% of recall makes these things to keep happening. But as we defined at the beginning, false negatives cost less than false positives.

- The regressor seems still weak as the RMSE, even after having gotten decreased, is 83,2025 units sold. But on the other hand, **on average, almost 64% of the variability is explained with the regressor**, so this could be happening due to the numerous outliers represented by the greatest hits.

- Apart from the cumulative effect of the errors we have made along the entirety of this process by substituting missing/ redundant/ etc. values, dropping columns, deciding not to scrape a column, combining sold and shipped amounts, etc., we can also consider the effects of irreducible error due to things such as:

    - Relevant data/features not being available for us, such as money spend on marketing or digital PC games sold via Steam and other platforms.

    - Errors in the data itself, most probably from the VGChartz, as we said at the beginning, it is a rough estimate of sales.

- Maybe we have chosen the wrong estimator or chosen the wrong parameters to tune them. Or we just needed more data but getting them is nearly impossible as companies usually keep them as a secret.

- After trying to drop near 10 least significant features, the metrics did not change by much. Maybe performing a **PCA would have improved the performance**.

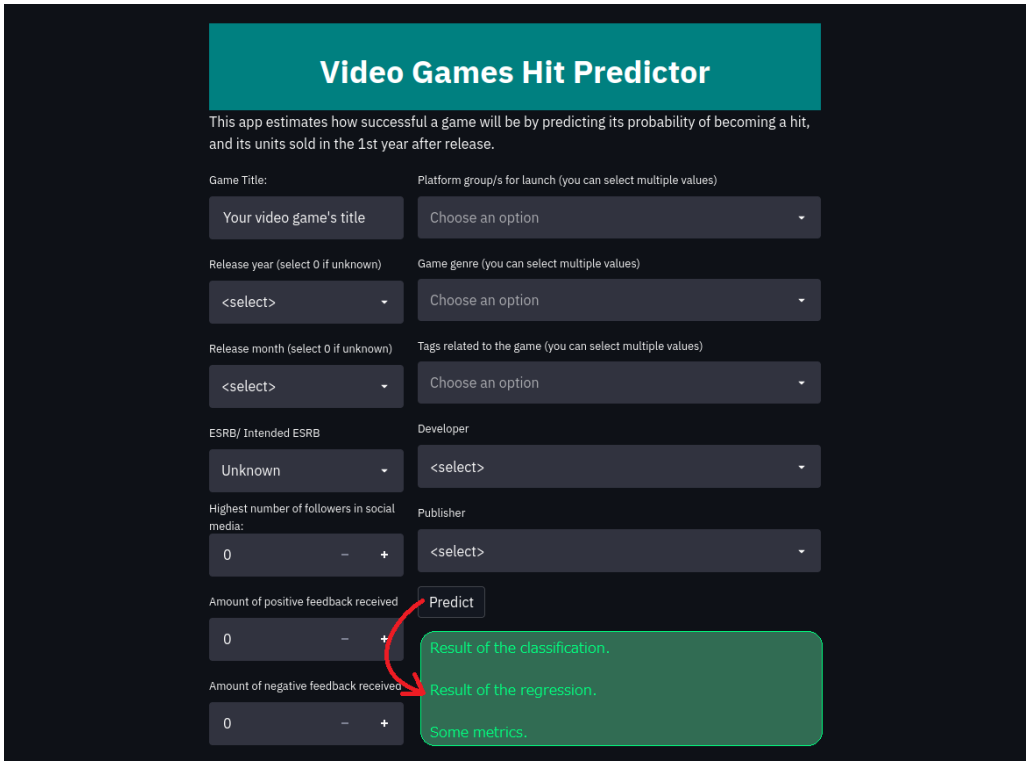
- Even though we consider the model to not be good enough, **metrics did improve indeed after performing transformations** such as applying different periods in video games history to fill missing values and *All* values in the *Platform* column or filling missing values in *Scores* and *Suggest_count* with different values depending on the *Sales_total* column.

- At least, even though the likelihood of a game becoming a hit is less than 50%, users can check the regression counterpart and analyze it to decide if it is worth to continue with their project or not.

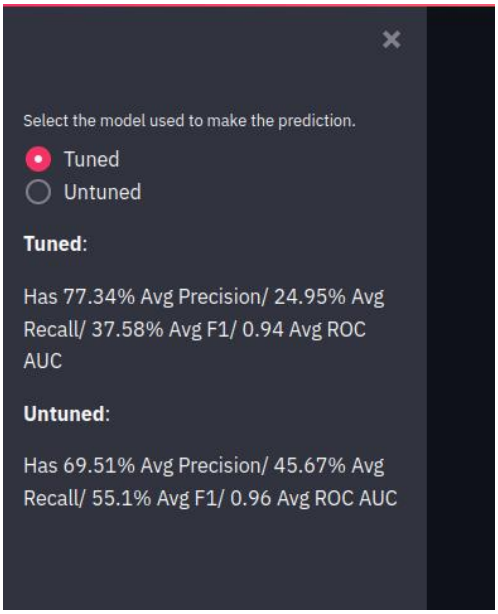
6. User manual for the frontend

The app is simple: after opening the web app, users only have to fill the values and click on [**Predict].**

Then the web app will show the result of the predictions and some metrics.



Users can also select which classification model they want to use from the sidebar. The model used by default is the tuned one.



7. Errata

There was an erratum in one of the functions in the 06_ML notebook, causing more than 1000 data points to have scores with incorrect values. After fixing the code of said function and undergoing the training process again, the metrics are the following:

7.1. Classificator (top one = before correction)

7.1.1. Untuned

```
========================================================================
XGBClassifier
========================================================================
    Accuracy : 0.963363   |  CV : 0.962345 (avg) [0.957671, 0.966084]
   Precision : 0.735385   |  CV : 0.695108 (avg) [0.628099, 0.743191]
      Recall : 0.481855   |  CV : 0.456744 (avg) [0.401055, 0.503958]
    F1 Score : 0.582217   |  CV : 0.550971 (avg) [0.489533, 0.600629]
     ROC AUC : 0.736077   |  CV : 0.956500 (avg) [0.952442, 0.961517]

[[8780   86]
 [ 257  239]]
========================================================================
XGBClassifier
========================================================================
    Accuracy : 0.961440   |  CV : 0.961490 (avg) [0.958339, 0.965416]
   Precision : 0.699115   |  CV : 0.682394 (avg) [0.642308, 0.727273]
      Recall : 0.477823   |  CV : 0.447774 (avg) [0.379947, 0.506596]
    F1 Score : 0.567665   |  CV : 0.540008 (avg) [0.480000, 0.597201]
     ROC AUC : 0.733159   |  CV : 0.961349 (avg) [0.959637, 0.964391]

[[8764  102]
 [ 259  237]]
```

7.1.2. Tuned

```
========================================================================
XGBClassifier
========================================================================
    Accuracy : 0.956633   |  CV : 0.958179 (avg) [0.955401, 0.959808]
   Precision : 0.812500   |  CV : 0.773391 (avg) [0.682927, 0.861111]
      Recall : 0.235887   |  CV : 0.249482 (avg) [0.221636, 0.306069]
    F1 Score : 0.365625   |  CV : 0.375768 (avg) [0.334661, 0.431227]
     ROC AUC : 0.616421   |  CV : 0.935852 (avg) [0.926240, 0.948073]

[[8839   27]
 [ 379  117]]
------------------------------------------------------------------------
XGBClassifier
========================================================================
    Accuracy : 0.958449   |  CV : 0.959300 (avg) [0.956202, 0.960742]
   Precision : 0.828221   |  CV : 0.777766 (avg) [0.700787, 0.846154]
      Recall : 0.272177   |  CV : 0.276892 (avg) [0.234828, 0.321900]
    F1 Score : 0.409712   |  CV : 0.407215 (avg) [0.351779, 0.450185]
     ROC AUC : 0.634510   |  CV : 0.956842 (avg) [0.953100, 0.963281]

[[8838   28]
 [ 361  135]]
```

In this case, the tuned version of the model performed better, but performances do not seem to change that much.

7.2. Regressor (top one = before correction)

7.2.1. Untuned

```
================================================================
XGBRegressor
================================================================
     MSE : 0.902618   |  CV : 0.845306 (avg) [0.411064, 2.045943]
    RMSE : 0.950062   |  CV : 0.874248 (avg) [0.641143, 1.430365]
     MAE : 0.166277   |  CV : 0.160356 (avg) [0.138990, 0.188633]
    MAPE : 0.465886   |---------------------------------------
      R2 : 0.784669   |  CV : 0.608575 (avg) [0.397289, 0.794949]
      EV : 0.784671   |  CV : 0.608623 (avg) [0.397358, 0.794976]
================================================================
XGBRegressor
================================================================
     MSE : 1.075928   |  CV : 0.687776 (avg) [0.289438, 1.573178]
    RMSE : 1.037269   |  CV : 0.790077 (avg) [0.537995, 1.254264]
     MAE : 0.161880   |  CV : 0.149299 (avg) [0.134558, 0.168583]
    MAPE : 0.421134   |---------------------------------------
      R2 : 0.743323   |  CV : 0.688127 (avg) [0.540789, 0.783858]
      EV : 0.743391   |  CV : 0.688266 (avg) [0.540900, 0.783941]
```

7.2.2. Tuned

```
================================================================
XGBRegressor
================================================================
     MSE : 1.094383   |  CV : 0.736004 (avg) [0.416865, 1.572083]
    RMSE : 1.046128   |  CV : 0.828633 (avg) [0.645651, 1.253827]
     MAE : 0.179758   |  CV : 0.165879 (avg) [0.144873, 0.187333]
    MAPE : 0.504337   |---------------------------------------
      R2 : 0.738921   |  CV : 0.644251 (avg) [0.446196, 0.789491]
      EV : 0.738973   |  CV : 0.644411 (avg) [0.446566, 0.789557]
================================================================
XGBRegressor
================================================================
     MSE : 0.879564   |  CV : 0.789924 (avg) [0.341772, 1.976586]
    RMSE : 0.937851   |  CV : 0.839242 (avg) [0.584613, 1.405911]
     MAE : 0.156548   |  CV : 0.150142 (avg) [0.129665, 0.178622]
    MAPE : 0.440586   |---------------------------------------
      R2 : 0.790169   |  CV : 0.648255 (avg) [0.514293, 0.798451]
      EV : 0.790187   |  CV : 0.648377 (avg) [0.514370, 0.798508]
```

In this case, happens the opposite: the tuned regressor seems to work worse than the tuned estimator.