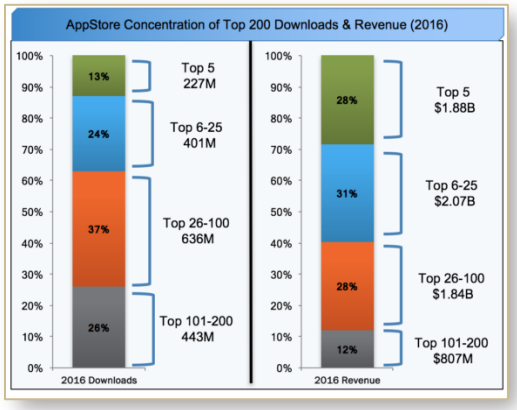
Predicting Paying Users in a Free-to-Play Game

by **Tiago Tex Pine**
https://www.linkedin.com/in/texpine/
https://twitter.com/TexPine

This is an attempt to produce a machine learning pipeline and a model that can predict which users are more likely to make purchases and spend money in a free-to-play mobile game after some time of play.

Introduction

The mobile games industry is dominated by the free-to-play model, but less and less companies are succeeding in making their business work. A huge amount of consolidation is happening as fewer and fewer games make money, and the ones in the 100 top charts respond for over 80% of all game revenue of the app stores.



Source:
https://www.gamasutra.com/blogs/JosephKim/20170427/296933/2016_Mobile_Gaming_Trends_Report_Top_8_Takeaways.php

What does the successful games do? Increasingly, it is a matter of **Live Ops** - keeping the game alive and relevant for its core base audience.

As the ecosystem has been flooded with new games over the last 5 years, the cost of acquisition of a single new user skyrocketed. The cost of acquisition of a single new user has reached thresholds between $7 to $10 *each*, and in the last few years companies realized it's much more expensive to acquire a new user through ad-buying than it is to *keep an existing user playing the game and spending money on it.*

Hence, Live Ops in a free-to-play game has two main goals:

1. *Keep Retention as high as possible*, so the game and its community remains alive and the company doesn't have to acquire more new users in the expensive ad market.
2. ***Convert players as much as possible - Paying Users*** are not only a source of revenue, but they are also the most enthusiastic users and create network effects that help keep other uses engaged, helping to stabilize retention too.

Therefore, the ability to scan every new user that downloads the app and predict which ones are more likely to become a Paying User can be a big benefit for keeping **both** retention and revenue in good levels.

With a more focused cohort to work on, a Live Ops teams can act more effectively on them - for example, by initiating direct marketing campaigns, a direct customer care contact, or a unique Starter Pack offer not available for others (working on our *exclusivity bias*).

Goal

The goal of our model is to **predict potential Paying Users with 3 days or less of data.**

In other words, predict if any new user who downloads the game and start playing will eventually become a Paying User in the future only by looking at his first 3 days after installation.

Game and Data

An undisclosed game company has agreed to provide us access to their data on a **Mobile Multiplayer Strategy Game**, with similar RTS-like mechanics to notable mobile titles such as *Clash of Clans, Castle Crash, Game of War, Rival Kingdoms* and *Siegefall*.

Source: *DomiNations* and *Clash of Clans*

Overview of Game Mechanics

Like the games of reference, our game is a strategy game where the flow is divided in basically two parts:

1. **Combat** another Player
2. **Build** your Base

The ***Combat phase*** is truly the core of the game. Players attack each other to loot resources and earn rankings in a competitive multiplayer League (a sort of leaderboard). As the player progresses in the ranks, she will need more and more powerful armies to *attack* successfully – and more and more powerful *defenses* to avoid being looted.

These upgrades are chosen in the ***Build phase***. Here, players construct their base and upgrade their troops to keep competitive. And although the core of the game is the Combat phase, the reality is players spend a *lot* more time in the Build phase. Just consider the amount and variety of options players have to choose from:

- which building to construct;
- which troop upgrade to invest;
- which troops to train and;
- which power-ups or in-app purchases to buy.

Moreover, the game design around these options is always built on top of complicated **dependency trees** that take a *lot* of time to build.

As an example, consider, these "tech trees" from the free-to-play game *Newerth* **in the Appendix**. It is a deep tree by itself – and in some of those games those trees can take over 5 months to be maxed, given how the game economy is balanced to manipulate upgrades cost in resources and real time to build. Consider also the fan-made table from *Clash of Clans* **in the Appendix** for a demonstration how fast those things scale over the game progression.

All of this is, of course, intentional: **free-to-play games monetize when players want to "speed-up" this process.**

Features of our Data

The data we have to work on is on the file *player_data.csv* in the capstone folder. Let's consider what kind of features we have available, and what they mean in the game mechanics:

- ***battles_...*** features contain information about the actual player-versus-player (PvP) interactions in the game. For example, how many attackshow many "revenge" counter-attacks, how many resources (Gold, Elixir and Trophies) were earned or lost, etc. The "start" sub-features are the level of success of an attack: players can earn 1, 2 or 3 stars for their performance.
- ***clan_...*** features are about the participation of each player in Clan activities. In our game, players can donate troops or request more troops to other players in the same Clan.
- ***connection_...*** features are about technical stability of the game.
- ***events_...*** features about participation in time-limited events in the game. Once in a while the game offers the opportunity to engage in special missions that are not normally available, and are only available for a few hours. Hence, this is, indirectly, a measurement of engagement with the game itself.
- ***friends_...*** are the overall information on how many other players each use has in their in-game friends list.
- ***gifts_...*** contains the amount of time each player received gifts from the Live Ops team. Developers can issue one-time gifts for one or more players for a variety of reasons, for example, if the game experienced lots of server instability.
- ***is_paying_user*** the label we are trying to predict.
- ***offers_...*** are time limited promotions on in-app purchases, such as Bundle Packs with lots of different resources, that the developer issue as a means to increase conversion and revenue temporarily.
- ***powerup_...*** are special boosts players can earn or purchase that affects their effectiveness in combat. These are not the troops themselves, but some kind of special protection or power. For example, in *Siegefall*, players can bring spell cards to battle and throw them at the enemy to help the troops win. This mechanics would be considered a "powerup" in our data.

- ***resource_op_...*** those are the operations in the game that consume resources. Any upgrade, troop creation, damage repair or building construction is considered a resource operation. There are many of them, since the game has several different options available to spend your currencies on.
- ***rewards_...*** are actually *special earnings* received outside the main combat->loot loop of the game. For example, when a time-limited event grants extra rewards.

Confidentiality Disclaimer

All player information has been anonymized - no information that could identify users, such as demographics or personal emails, are available. Only data of gameplay and some few other extraneous information like device/OS is present.

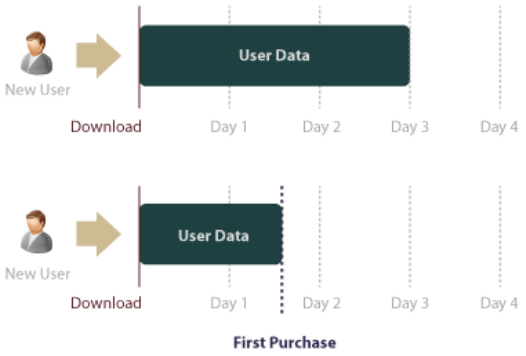
Also, the identity of the company, the name of the game and very specific information on how each feature works in the game (like the resource operations) will remain undisclosed - but it won't affect our capacity to develop machine learning algorithms.

Data Pre-Processing

Basic information and Splitting samples

The data at *player_data.csv* was collected from the game's server SQL database and compiled in a format friendly to machine learning algorithms. Here are some important traits of how it is formatted:

- Each row is a player.
- All data have been retrieved only from the **latest major version of the game**, all builds under the "1.3" major revision. We are users from older versions because between major updates mobile free-to-play games usually add or change features that may *significantly* change player behavior.
- The goal of our model is to predict potential Paying Users within their first 3 days of activities. Hence, **all data is an aggregation of *only* the first 3 days of play.**



- In case a player has become a Paying User *before* completing 3 days of activities, we only grab data *until the first purchase happens*. Because after an in-app purchases, combat activity and resource flows can be ***very*** different Regular Users, and we don't want to use them.

User Distribution

Let's start by looking into the data and how users distribute in the label we want to predict:

```
Total number of players: 310,298
Memory usage: 482,416 MB
Original features: 198
Number of Paying Users: 908
Number of Non-Paying Users: 309,390
```

We can see this is going to be a *challenging Machine Learning task* because:

1. The amount of Paying Users is very low compared to the amount of regular free-to-play users. With **such class imbalance**, we will need several methods and a complex pipeline to get the best possible classifier.
2. The main dataset is too big to be processed locally in a desktop machine after we do the *one-hot encoding on string fields*, the sheer size of the database in memory will more than double, which can lead to hangs and crashes.

The first problem will require a well-built pipeline to be addressed. The second one, we can solve by creating samples and training our model on them.

Samples

We *extract all 908 available Paying Users* and engineer samples in order to maximize our chances to train and generalize our classifier. These are the datasets we will construct:

1. **Training sample**, with 40k users, 538 Paying Users among them. This will be our main dataset to train and cross-validate classifier iterations.
2. **Validation sample**, with 20k users, 252 Paying Users among them. This will serve to validate our methods with the Training Sample, but never being used to train the classifier directly (otherwise it would be a huge methodological problem).
3. A **small Test sample of 100 users, with an even split** of 50% regular users and 50% paying users. We will use it to test how generalized the final classifier is.
4. A larger **Test sample with 20k users and a much smaller, "natural" proportion of Paying Users**, of about 0.2% of them, like in the original data. This will test how precise the classifier can be in a real-world context where Paying Users are very scarce.
5. An additional sample with all Paying Users, for data exploration.

	Users	Regular Users	Paying Users	% of Paying Users
Training	40000	39496	504	1.260%
Validation	20000	19748	252	1.260%
Even Split	200	100	100	50.000%
Natural Split	20052	20000	52	0.259%
Paying Users	908	0	908	100.000%

In the process of constructing these samples, we also *one-hot encode* all string columns (like "device_model") into binary features the learning algorithms can use.

Exploration

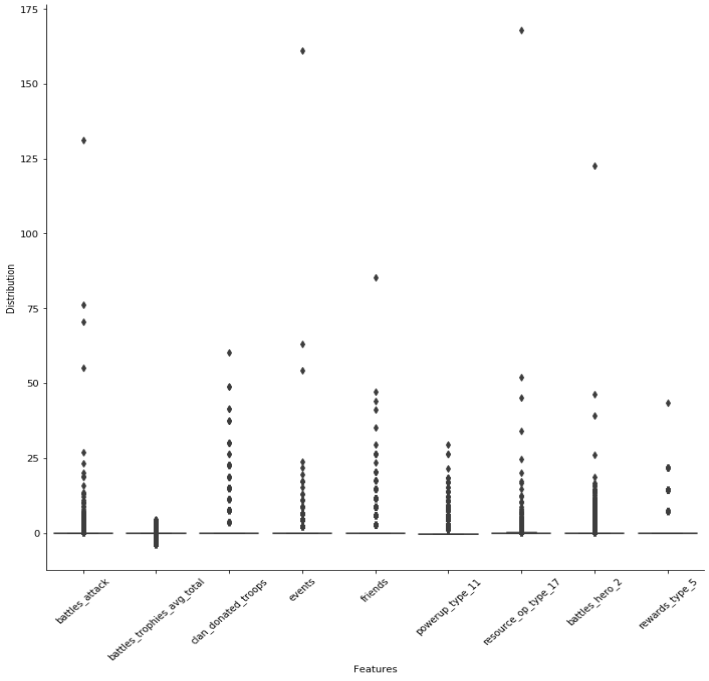
Plotting our Data

We'll look into some of the features that *should* be among the most important ones to classify and cluster users in the game:

- PvP attacks
- Trophies earned, which are an indication of PvP performance.
- Donations to the Clan, which indicates level of social activity .
- Events, which denotes how often the user comes back to the game.
- Friends, another social activity measure.
- Usage of a Power-up.
- Usage of a Resource Operation.
- Earnings of a Reward Type.
- Usage of the most popular Hero.

Boxplot

We can clearly see a big distance of outliers from the mean and median of the data. The inter-quartile range can't even be seen in the graphic.



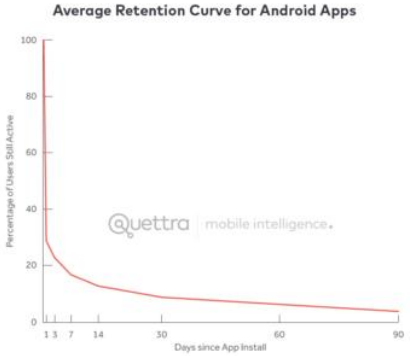
This indicates our data is probably shaped as exponential distributions, not normal distributions. We have the mean and median and IQR very close to zero, while outliers strongly deviate from the norm.

Scatter Matrix

A scatter matrix may provide us better information on how those features correlate, after normalizing all data to a space between 0 and 1 for consistency in the graphs.

The result **is in the Appendix**. The matrix point to another distinct problem, related to the shape of the boxplot before: the main issue is that a huge number of users have values very close to zero in all features.

This is actually ***expected in a free-to-play model***: usually, over 50% of all new users quit in minutes after downloading the application. Consider this graphic for an expected retention curve in a free-to-play game, on the side.



In a sense, any users who stick with the game over 3 days ***will be the minority*** when considering the total universe of all users who ever booted the game app.

Even among paying users we may have some of this problem. Some users will only really start playing the game after doing the first purchase in the first few days - the moment at which our trackers stop (all data is filtered to only have trackers *before* the first purchase).

Log-transforming

Since our data is heavily skewed and probably following an exponential or log-normal curve, we can try to log-transform it for visualization purposes.

In the Appendix, we have 2 other scatter matrices after log-transformation, one for **Regular Users** and another for **Paying Users**. Log-transformation helps to better visualize the data and draw some better conclusions:

Regular Users vs. Paying Users

The scatter matrices confirm the general behavior that is expected from Paying Users, as users who explore more the game even before making the first purchase.

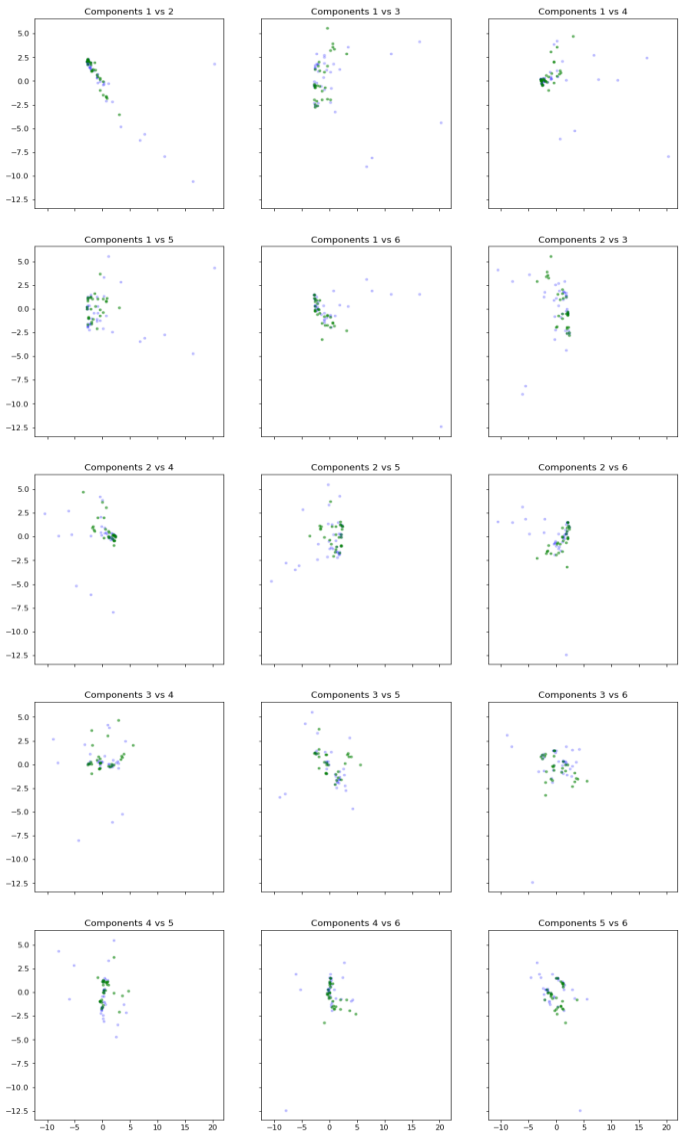
> 1. The usage of Hero 2 is heavily associated with non-paying users. Paying Users tend to use other Heroes after some time playing.
> 2. Paying Users commit more to Events.
> 3. Paying Users tend to attack more and win more Trophies - they perform better in combat.
> 4. Surprisingly, Paying Users are not more interested on making Friends in the game.

One additional insight: the correlations in these features, even if relatively weak (like resource operation 17 with participation in time-limited Events) hints many of our features are not conditionally independent from other features. This eliminate the possibility of models that assume independency, like Naive Bayes.

PCAs

We can also have a glimpse on how Paying Users and Regular Users distribute in the hyperspace by reducing dimensions with PCA, and the plotting the relationship on the components. (We'll log-transform and standardize the data, as this will result in more legible graphics across the charts.)

In the charts, Regular Users are blue dots and Paying Users are green dots. Clearly, Paying Users are very much mixed with the Regular Users in the latent factors of the dataset. That hints they will be hard to classify, so we will need very strong learners to split them apart.



Model Selection

Choosing the Algorithm

First, we need to complement our knowledge on the data with a quick classifier run just to assess how accuracy scores looks like. A Logistic Regression is a fast model to check.

	precision	recall	f1-score	support
0	0.99	1.00	1.00	19748
1	0.87	0.29	0.44	252
avg / total	0.99	0.99	0.99	20000

This classification report shows we have a big problem when it comes to classifying Paying Users correctly. While the average total F1 may look good, the numbers for Paying Users (class '1') are very low. Clearly, this issue is a by-product of the ***huge class imbalance*** between classes 0 and 1 in the dataset.

Choosing the Scorer

Recall?

Recall, in particular, was quite bad in the initial model. This is very concerning as in the free-to-play model, it's costlier to lose a good lead than to follow a bad lead.

Following a bad lead (*false positive*) results in linear costs for the Live Ops team. But losing a good lead (*false negative*) results in **exponential losses**: a single Paying User in the free-to-play model can represent hundreds, even *thousands of dollars* of revenue on her lifetime. One Paying User can be well worth $500 or more.

Hence, we need to prioritize Recall, very much *like machine learning in Medicine do*. (For those systems, is also far costlier to miss 1 diagnosis of cancer, for example, than to falsely diagnose 1 healthy patient.) But we also don't want to lose control on Precision, as too many false positives will also be very costly. So, we need a scorer that weights in both.

FBeta

The FBeta scorer allow us to use a Beta term that weights the F-score calculation more in favor of Precision or more in favor of Recall.

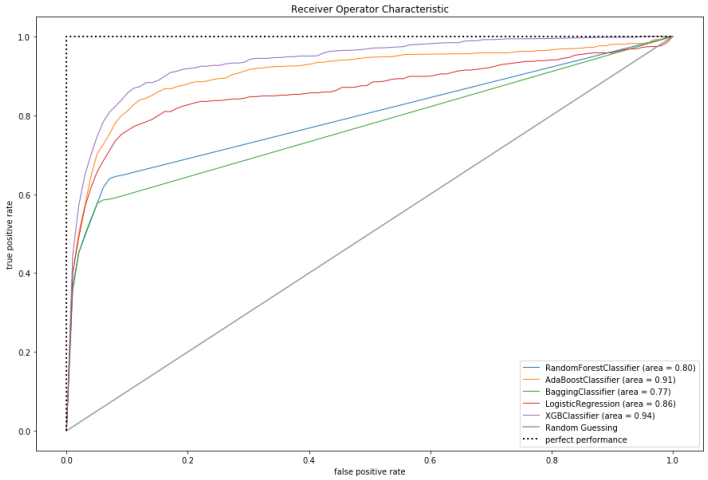
$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

When we plot in a table an FBeta score with *Beta=3*, we discover this **FBeta function can be perfect for us**: the score will initially increase much quicker as we improve Recall, but only achieve top performance if Precision starts to rise too.

See the Appendix for a simulation of this scorer.

ROC Curves

We can start by assessing the ROC curve of different classifiers, in order



to check which ones are better than pure chance and which ones are closer to a perfect fit. (We test with standardized data, as it is highly recommended for Logistic Regression and doesn't really affect the other ones which are more based in Decision Trees.)

The best algorithms are **the ones using boosting techniques**. Surprisingly, Logistic Regression seems to be better than the ones heavily reliant on Decision Trees, which may indicate a tendency in the data to overfit.

XGBoost

At this point, seems like the better idea to use the powerful **XGBoost** an implementation of the gradient boosting decision tree algorithm. This *ensemble method* is known to be flexible, fast and accurate, and it has won lots of competitions on Kaggle. (More information about the algorithm at http://xgboost.readthedocs.io/en/latest/model.html)

Given how hard it seems will be to split Paying Users from regular users, as we observed in the Exploration section, it seems fitting to try to use the best algorithm available for hard classification problems.

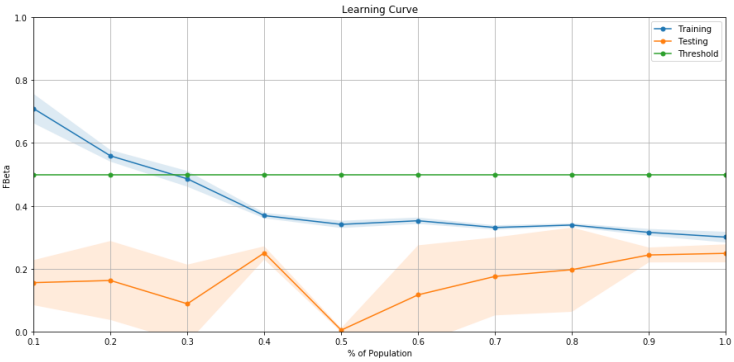
Learning Curve

This is how an *untuned* learning curve looks like under the XGBooster classifier.

```
Untunned classification report:
             precision    recall  f1-score   support

          0       0.99      1.00      1.00     19748
          1       0.95      0.35      0.51       252

avg / total       0.99      0.99      0.99     20000
```



From the shape of the learning curve, **our model is probably suffering from high Bias**. Our FBeta is quite poor and far away from a desirable threshold of at least 0.5

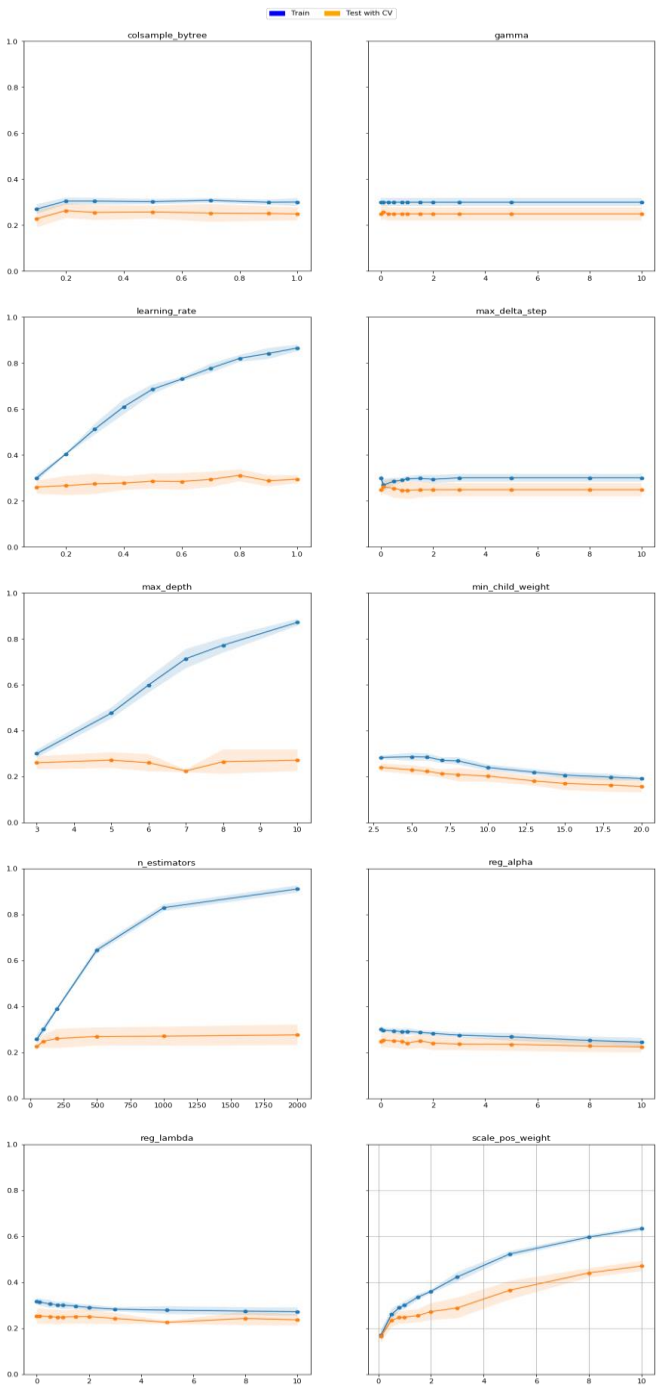
So we can tune it and increase Variance of the model, working to increase Recall, and keeping Precision in acceptable levels in the process.

Parameters and Validation Curves

In order to better explore the potential impact of tuning XGBoost's parameters, we can use Validation Curves to assess if each of them has a measurable impact on improving our Recall.

XGBoost has a lot of parameters, but for this analysis we can focus on the ones that can yield the best results:

- **max_depth**: the maximum depth in which the tree branches;
- **min_child_weights**: the minimum weight a leaf node must have;
- **gamma**: minimum loss reduction to branch a leaf;
- **colsample_bytree**: subsample ratio of columns when constructing each tree;
- **subsample**: how much data is collected to construct new trees;
- **reg_alpha**: the L1 regularization term;
- **reg_lambda**: the L2 regularization term;
- **learning_rate**: shrinks the feature weights to make the boosting process more conservative;
- **scale_pos_weight**: controls the balance of positive and negative weights, good to adjust our class imbalance;
- **max_delta_step**: constrain on the maximum weight a tree can have (the higher the more conservative the algorithm);
- **colsample_bylevel**: subsample ratio of columns for each split in each level;
- **n_estimators**: amount of trees;



Looks like we may have a problem: even with the best model, this data seems to be nearly untrainable. Increasing values doesn't seem to impact our ability to increase test values, except for one of the hyperparameters (scale_pos_weight that, not coincidentally, deals with class imbalance). In three of them - learning rate, max_depth and n_estimators - we actually have overfitting, as the score improves in the training set but remains flat in the test set.

So before tuning the XGBoost classifier, we can try to take a step back and do data processing techniques first.

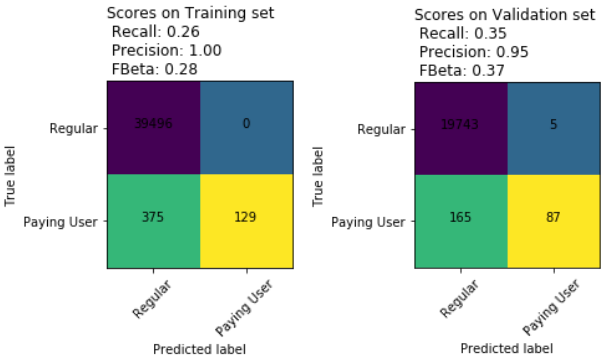
Data Processing

Selecting Data and Sampling Operations

We can attempt techniques to process our data to help the algorithm learn better:

1. Remove outliers.
2. Feature selection.
3. Undersample regular users.
4. Oversample synthetic Paying Users.

The assessment on how effectively these techniques progress will be made based on the score of an untuned XGBoost results:



Outlier Removal

Conceptually, the problem with outlier elimination is that**, by definition, free-to-play models rely on outliers**. Not only Paying Users are buying currency and accelerating much faster than regular users, but the very dedicated user - paying or not - is key to keep Guilds active and the metagame of a multiplayer game going.

By definition, the 2% of our users who are top players in the game also create a lot of **network effects** that keep the game community alive, and the game profitable month after month.

Therefore, we need not only strong techniques to find those outlier who really consistently skew training *across many features*, but also we need to be conservative in the tuning and careful in the hyperparameter search.

Outlier Detection through Dimensionality Reduction

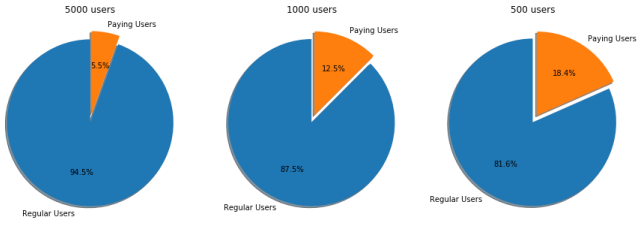
We are going a technique similar to the one used on this code for anomaly detection through deep neural networks: https://github.com/pedroconcejero/h2o_training/blob/master/2_h2o_anomaly_detection.r But instead of a network that constrains information through layers with less neurons, we will accomplish the same idea by using PCA.

The idea is to:

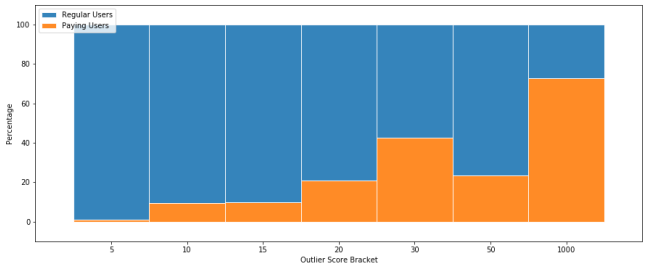
1. *Reduce the dimensionality of the data* by transforming it into PCA components.
2. Inverse-transform it, *extrapolating the reduced data back to the original feature space*
3. Run a **mean squared error** *for each player*, between the original data and the transformed one, to assess how much deviated from the original vector.
4. **Use this score as an "outlier score"** to decide which ones to remove.

In theory, this technique allows us to select outliers based on the latent factors of the data, and exclude only the ones that consistently deviates from the normal across many features.

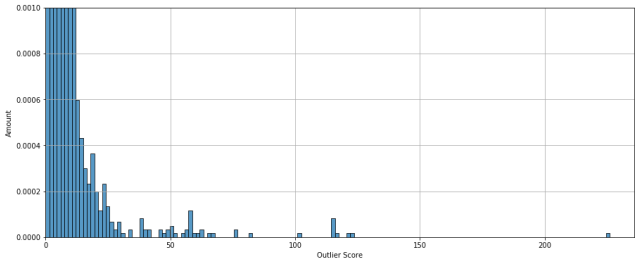
The results are as follows. Here's *the participation of Paying Users among the top X outliers:*



How Paying and Regular Users distribute across a 100% Stacked Histogram by the player's mean squared error:



An Histogram of the amount of players by thresholds of mean squared error:



Clearly, the high the outlier score, the greater the participation of Paying Users. This is expected, as Paying Users will tend to be statistical outliers a lot more than regular users.

Cutting Criteria

The goal is to cut outliers and re-train the classifier to acquire better scores. As a cutting criteria, we are going to use **the distance in standard deviations away from the mean of all "outlier scores".**

Also , we should *not cut* any Paying User because, as we've seen, they are disproportionally represented in the higher tiers. As a scarce class, excluding them may harm our ability to predict them in the first place.

Optimizing the Method

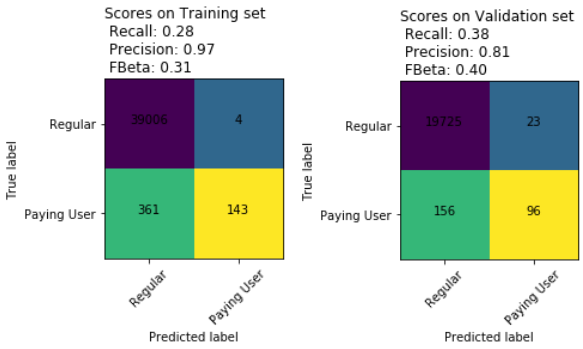
We can now **grid-search for the best possible FBeta** by varying out outlier cutting criteria by:

1. **The amount of dimensions** to reduce the data to.
2. **The amount of standard deviations** to use as cut criteria.

These are the top FBetas in our search:

	components	standard_deviations	train_cv_score	validation_score
365	141	2	0.289759	0.414226
335	135	2	0.289549	0.414226
405	149	2	0.289508	0.405858
325	133	2	0.287614	0.406028
400	148	2	0.287662	0.405858
410	150	2	0.281237	0.414399
240	116	2	0.283396	0.410729
445	157	2	0.283484	0.410042
420	152	2	0.285566	0.406198
205	109	2	0.285150	0.406709

By applying the optimal value of *141 components and 2 stds*, we get the following improvement in our confusion matrices:



The score in both sets improved a bit, so it seems we generalized the model a bit better. But a consequence in the Validation set is the increase of false positives, since we removed the most extreme regular users and the algorithm now classifiers more users with great activity as Paying Users.

Feature Selection

The next method we can try is to eliminate features that introduce more noise than information. We are going to use a **Recursive Feature Elimination** to rank the best features in terms of reducing the noise and optimizing the learning.

RFEs are pretty good to improve learners that **classify images** - and since we also have a large feature space, like images do, we can try to use it to really optimize our model.

RFE Results

Here's the top features selected by the RFE algorithm. Columns are:

- **is_feature_selected**: if the RFE selected the feature
- **feature_ranking**: the ranking as per the RFE algorithm. The lowest, the better.
- **cumulative_train**: our own cross-validation test in the Training set using the cumulative of all previous features in higher rankings.
- **cumulative_validation**: our own test on the Validation set using the cumulative of all previous features in higher rankings.

	is_feature_selected	feature_ranking	cumulative_train	cumulative_validation
resource_op_type_19	True	1	0.102508	0.180645
resource_op_type_24	True	1	0.174984	0.259574
resource_op_type_19_amount	True	1	0.189860	0.263605
resource_op_type_17_amount	True	1	0.189901	0.255428
resource_op_type_13	True	1	0.187864	0.004308
resource_op_type_1	True	1	0.278611	0.366470
has_clan	True	1	0.288955	0.366470
resource_op_type_13_amount	True	1	0.293111	0.382675
device_platform_Apple	True	1	0.301282	0.398825
player_language_English	False	2	0.309366	0.406879
resource_op_type_30_amount	False	3	0.309300	0.004246
clan_requested_troops	False	4	0.305164	0.394626
battles_defense_won	False	5	0.307561	0.394958
resource_op_type_24_amount	False	6	0.305295	0.398490
resource_op_type_17	False	7	0.305425	0.402854
resource_op_type_1004_amount	False	8	0.307495	0.394461
clan_features_joined_after_trackers	False	9	0.307364	0.414226
resource_op_type_5	False	10	0.307495	0.410042

Tuning the Results

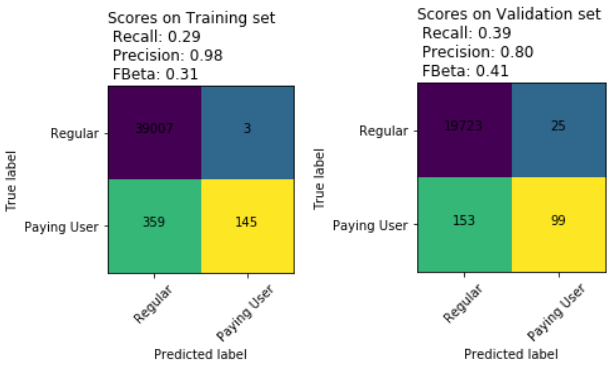
We created the *cumulative_train* and *cumulative_validation* columns in order to tune the results of RFE. While very good, RFE isn't perfect, and by *sorting with our own cumulative FBetas* we find out we can benefit from inserting more features:

	is_feature_selected	feature_ranking	cumulative_train	cumulative_validation
device_os_iPhone OS 8.3	False	58	0.309498	0.414226
clan_features_joined_after_trackers	False	9	0.307364	0.414226
device_model_TCL	False	59	0.313633	0.414226
events_joined_after_trackers	False	43	0.313767	0.414053
resource_op_type_34	False	26	0.309498	0.414053
resource_op_type_36_amount	False	54	0.311633	0.414053
resource_op_type_20	False	60	0.309498	0.413880

This sorting shows that the sweet spot is not only with rank 1 features, but with all features all the way to ranking 58. Hence, we can now decide in definitive to select only these features for our model:

resource_op_type_19	powerup_type_5
resource_op_type_24	connection_joined_after_trackers
resource_op_type_19_amount	resource_op_type_35_amount
resource_op_type_17_amount	resource_op_type_1004
resource_op_type_13	battles_attack_lost
resource_op_type_1	resource_op_type_21_amount
has_clan	device_os_Android OS 5.0.1
resource_op_type_13_amount	device_model_HUAWEI
device_platform_Apple	battles_trophies_total
player_language_English	device_model_iPad2,5
resource_op_type_30_amount	powerup_type_13
clan_requested_troops	device_model_HTC
battles_defense_won	resource_op_type_34_amount
resource_op_type_24_amount	resource_op_type_26
resource_op_type_17	powerup_type_7
resource_op_type_1004_amount	powerup_type_12
clan_features_joined_after_trackers	events_joined_after_trackers
resource_op_type_5	resource_op_type_3
resource_op_type_14	device_platform_WP8Store
device_model_samsung	battles_hero_2
device_os_Android OS 4.1.2	resource_op_type_12_amount
device_os_Android OS 4.2.2	battles_gold_lost
device_os_Android OS 4.4.2	battles_trophies_avg_lost
resource_op_type_14_amount	battles_elixir_lost
device_os_Android OS 6.0.1	resource_op_type_39_amount
resource_op_type_26_amount	battles_elixir_earned
battles_revengedefense	powerup_type_11
player_language_German	resource_op_type_36_amount
battles_attack	battles_revengeattack_won_star_3
resource_op_type_5_amount	battles_trophies_avg_earned
device_os_Android OS 5.0	device_os_iPhone OS 9.2
battles_defense	device_os_iPhone OS 8.3
resource_op_type_18_amount	
resource_op_type_34	

These are the results in the confusion matrices:



A slight improvement. On the other hand, we eliminated a lot of noise and complexity by getting rid of low-quality features, which can have very positive effects moving forward as we tune the model more and more.

Also, as a nice side effect, we can easily communicate to stakeholders which features seem to be causing the largest impact on the conversion of regular users to Paying Users. Game designers, for example, can maybe do something in games updates about the Resource Operation #19, increasing its need in the game progression at the player level-ups his base and his army.

Undersampling Regular Users

When dealing with highly imbalanced datasets, the idea of undersampling the majority class is to eliminate, move or replace data points too close to the boundaries with the minority classes.

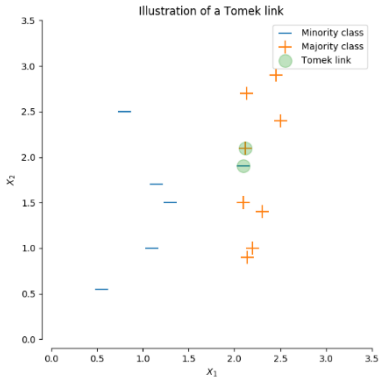
By manipulating the dataset with undersampling, we can improve the odds our classification learners will better learn to identify minority classes - in particular if our classifier uses Decision Trees as its underlaying engine, which is precisely the case for XGBoost.

Two techniques will be used: the removal of data point through the search for Tomek Links, and Instance Hardness Threshold to remove data points based on the difficulty to classify them.

We are going to use the **imbalanced-learn package**, a project that has integration with sklearn.

Tomek Links

The idea behind Tomek Links is to eliminate data points for which the nearest neighbor is a member of another class. The idea can be illustrated in the graphic.

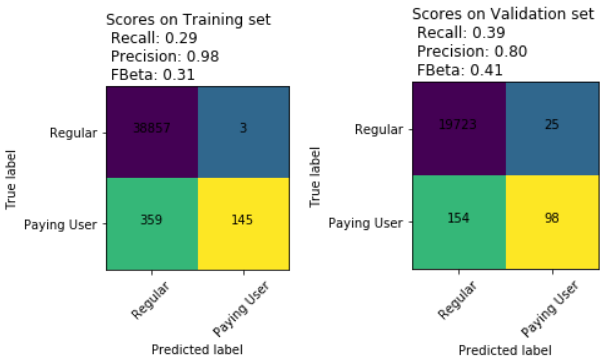


Tomek Links can be used to remove one or both of the data points above. In our case, as discussed, we will remove only regular users, leaving Paying Users in the dataset.

Here's the amount of reduction in the Training dataset:

	Before	After	Change
Users	39514	39364	-0.4%
Regular Users	39010	38860	-2.2%

And the scores result:



Practically no change, but at least we removed potentially problematic data points and that could improve our further efforts. Nonetheless, we need other methods.

Instance Hardness

Instance Hardness is a specific algorithm in which a classifier is trained on the data and the samples with lower probabilities are removed. The inner workings and the math of the algorithm is explained in this paper: http://axon.cs.byu.edu/papers/smith.ml2013.pdf .

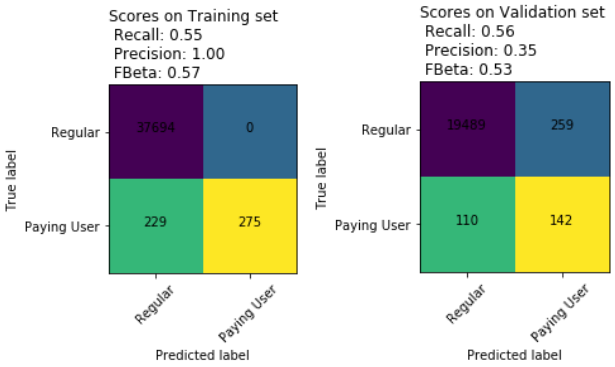
In other words, data points are removed based on their classification weakness or how close of being misclassified as Paying Users they are.

We configure this algorithm to try to reduce 3% of all regular users, (although the calculation of this particular algorithm cannot always guarantee this number is going to happen exactly).

The resulting reduction is as follows:

	Before	After	Change
Users	39364	38198	-3.0%
Regular Users	38860	37694	-3.0%

And the score results in the Training and Validation sets:



We manage to improve our scores substantially, but **we need to be careful with overfitting** at this point. If the score in the Training set starts to improve much faster than in the Validation set, this is a sign our model is starting to overfit. Moreover, the sharp increase of false positives in the Validation set is also an extra indication of overfitting.

For now, we can stick to these results as the Precision in the Validation set is still very much acceptable from a business perspective.

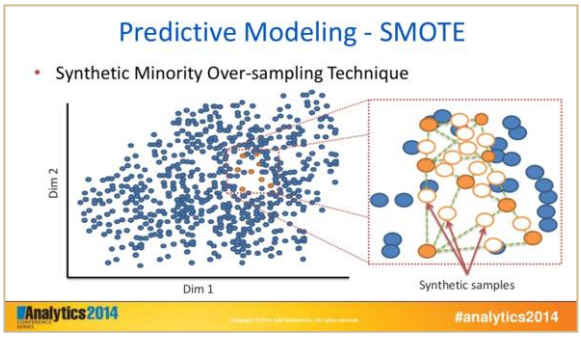
Oversampling Paying Users

This is the reverse operation from before: the idea is to oversample the minority class of Paying Users and *create synthetic datapoints that add more information* to the model, so the learner can have a better grasp of what those inviduals look like.

Oversampling is frequently used in domains like Medicine, where, for example, a given cancer detection algorithm may have very few images to learn how to classify a tumor.

SMOTE

We will use the well-know **SMOTE** algorithm to create these synthetic Paying Users in the Euclidean space between real Paying Users.



SMOTE is reasonably customizable, so we can actually make a **grid-search looking for the best possible hyperparameters** that maximizes our FBeta scores.

Here's the best results:

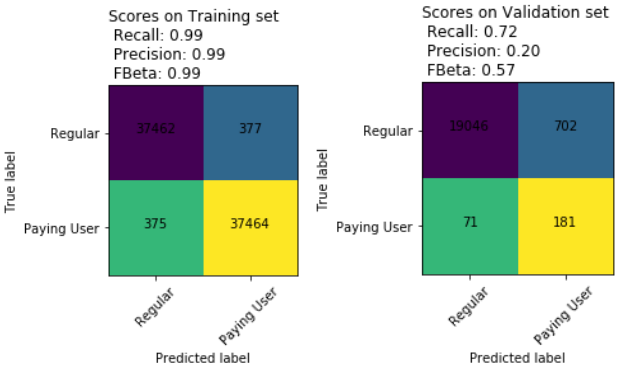
- **kind**: the type of SMOTE algorithm, we will use *borderline1*
- **ratio**: how many synthetic users will be created on each class. For us, we will create as many Paying Users as we have Regular Users.
- **k_neighbors**: number of nearest neighbours to be used to interpolate date and construct synthetic samples between them. Our value will be 8.
- **m_neighbors**: number of nearest neighbours to determine if a minority sample is in danger. This value usually has a bigger impact in our results than k_neighbors. Our value will be 20.

Results

The resulting oversampling looks like this:

	Before	After	Change
Users	38198	75678	98.1%
Regular Users	37694	37839	0.4%
Paying Users	504	37839	7407.7%

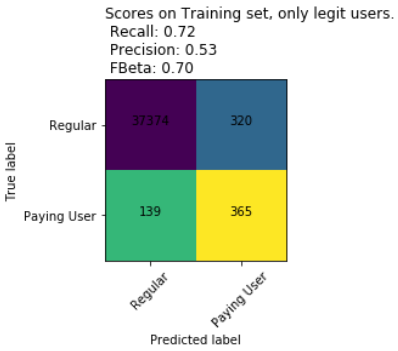
We doubled the number of total users and dramatically increased the amount of Paying Users. By fitting the XGBoost with this data, we have the following scores in our sets:



The FBeta and Recall in the Validation set have improved considerably, with the downside of getting a lot more false positives - but the **proportion of roughly 1 out of 4 leads for potential Paying Users is still very good** and would optimize tremendously the efforts of Live Ops teams.

Note, however, that our FBeta scores in the Training set are looking super-high, but the presence of **synthetic users** is now **skewing the scores** very much.

It is important, thus, to ***filter synthetic users out*** when evaluating our confusion matrices from now on. They will still be used in all fitting, but not on *testing*. Here's the result without on the Training set without them:



Without synthetic users, it's clear the results for the legitimate Paying Users are much better than the results of before. Looking at a previous confusion matrices on the Training set before the SMOTE, we can see that the fitting of XGBoost with synthetic users ***did improve considerably*** the capacity of the classifier to learn to classify the true users as well.

This is a pretty good result. but we will continue to iterate on it during the tuning phase.

Tuning

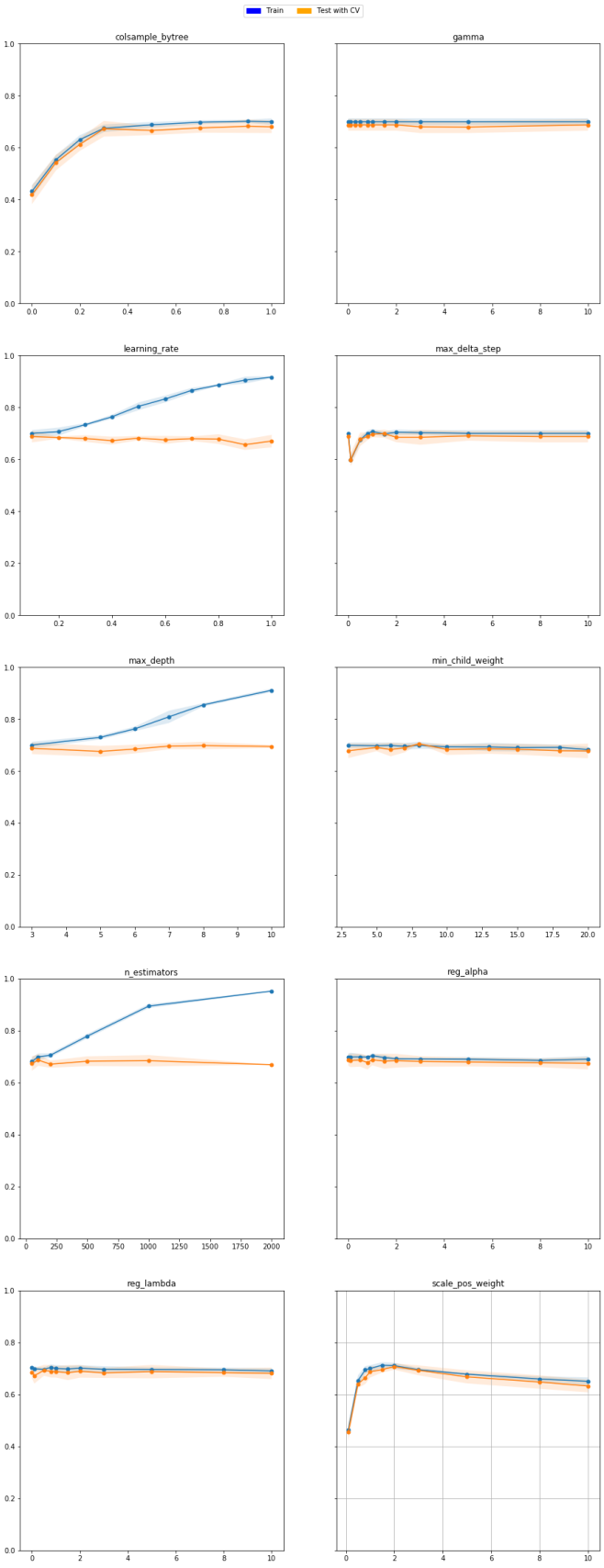
Fine-tuning of XGBoost

We are now ready to grid-search for the best hyperparameters possible on this XGBoost model. But first, let's update our Validation Curves to narrow down the scope in which the grid-search will be performed.

Updated Validation Curve

(Note that after data processing, the general FBeta score skyrocketed due to the presence of synthetic users, and the general validation curve function from *sklearn* would not work very well.

Thus, we developed our own custom Validation Curve function is also configured to output **scores ignoring the synthetic users)**



Although the scenario still looks hard to optimize, we can still do *some* tuning. In our Grid Search, we will look for:

- **max_depth**: between 5 and 8.
- **min_child_weights**: between 7.5 and 10
- **gamma**: between 0 and 2
- **colsample_bytree:** between 0.2 and 0.9
- **subsample**: between 0.1 and 1
- **reg_alpha**: between 0.0 (no L1 regularization) and 2

- **reg_lambda**: between 0.0 (no L2 regularization) and 2.5
- **learning_rate**: between 0.01 and 0.15
- **scale_pos_weight**: between 1 and 3
- **max_delta_step**: between 0.5 and 2
- **colsample_bylevel**: between 0.1 and 1.0
- **n_estimators**: between 10 and 200

Grid-Search

In our grid search, we will evaluate results based on both the FBeta scores in both the Training and Validation sets. We want to maximize both, but we don't want to get trapped overfitting in the Training, so the distance between these two scores should also be minimized.

Hence, we will use this **meta-score as evaluation metric of grid-search outputs** to which hyperparameter combination is best:

$$\frac{fbeta_{training} \times fbeta_{validation}}{\sqrt{fbeta_{training} - fbeta_{validation}}}$$

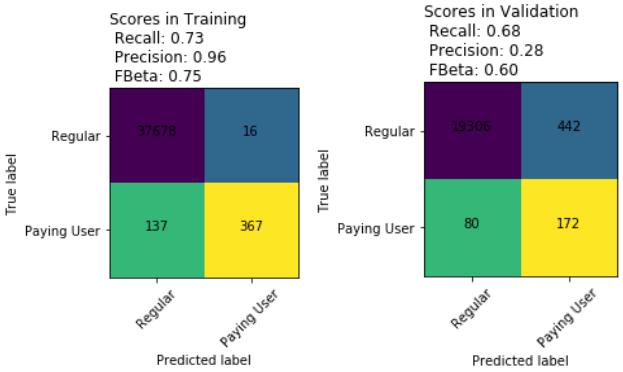
Also, as before, we need to remove synthetic users from the scoring evaluation. These two things are possible by modifying our own grid-search function, instead of using the one from *sklearn*.

Results

Here's the best tuning found by this grid-search:

```
== Grid Search best results: ==
Training: 0.7423225544, Validation: 0.6040731792
Parameters:
- subsample = 1.0
- colsample_bytree = 0.99
- gamma = 0.0
- min_child_weight = 8.0
- max_depth = 7
```

…leaving the remaining hyperparameters as default values. With this profile, we fit again the XGBoost model for these results:



Pretty good! Recall reduced a bit, but we had a larger increase on Precision, and overall the model is working quite well. In the Validation set, we are able to predict 68% of all users with potential of becoming Paying Users, with a 28% chance of a good lead for all players marked as candidates.

Not only that, but this pool of false positives in Validation may actually be a bonus as, with them, we **can actually increase the conversion rate of the game**. With a focused direct marketing, offers of great value, exclusive content and pre-emptive customer support on players our algorithm learned are showing signs very similar of future Paying Users, it is not unreasonable to think *we can turn at least 1/4 of those false positives in new Paying Users!*

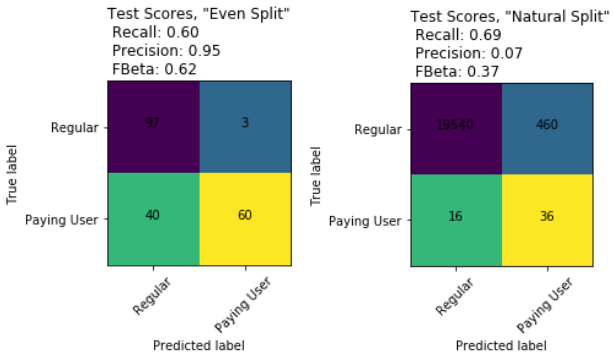
Final Tests

Predicting in Test Sets

The final tests of the algorithm will be done with the 2 Test samples we constructed in the beginning:

1. The small test set with 50%/50% Paying Users and regular users, with 200 users total;
2. The large test set with a "natural" distribution where Paying Users are 0.2% of the population, with nearly 20k users total.

Thus, the final results!



Conclusions

While far from perfect, the final Tests were still interesting: we were able to identify more than half of the Paying Users in the "even split" sample, with only 3 false positives, and 69% of Paying Users in the "natural split".

The main problem is that, in the later, we had a **very high ratio of false positives**. While some of them can be converted to Paying Users with better and more focused incentives, this is a problem because this Test set is the one *closest to real-life operations*.

Hence, such a low Precision might make the model non-viable for Production, *in its current state*. It could, in fact, still be operated, but we would need to scale down the costs-per-user of our direct marketing actions by simply creating a cluster with these Positives that will receive better "Starter Pack" and "Beginners Pack" kind of offers. More personalized one-to-one contact from an outbound team would be probably too expensive for this kind of accuracy.

Further Improvements

Upgrading the model in future versions

This model can still be improved a lot in future versions. We can already identify a few ways how this pipeline could be upgraded in future versions:

1. More Extraneous Information

Extraneous information is, in this case**, information outside the context of the gameplay and game mechanics**. Among the features selected by the RFE we can already the information of which Device and which operation system the player uses makes a difference.

This is a strong indication that we should add to the model not only demographics such as age, gender, geolocation, etc.

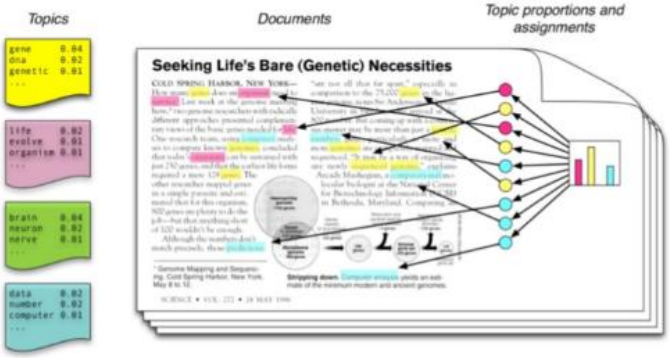
So, while *our initial modeling focused on what players are doing inside the game*, a second version of the model should account more for *who players are and what they do outside the game.*

Apps Used

One of the possibilities the company can immediately start to explore is to add to learning which other Mobile Apps the users have installed in the same device, besides our game. This information is already available in the original database of this game.

The issue with this approach is that, because there are so ***many*** different apps in the world, if we try to one-hot encode the label of these apps, we can end up with literally over 100,000 new dimensions in the dataset!

To work around this problem we could, instead of one-hot encoding every single individual app, we can use **Latent Dirichlet Allocation (LDA) Topic Modelling**, a technique common for text documents. We would treat the collection of apps' names of each user as a "document", and use LDA Topic Modelling to create 10 or so probabilistic categories



that we would add to the dataset for the learning algorithm to use.

Demography Info

Another possibility is to use demographics. The company has some of this information, but it could not be disclosed for this work due to the privacy policy with their users. The data scientists internally could add this information, and it would go a long way towards improving accuracy.

2. Observation Weighting with Age

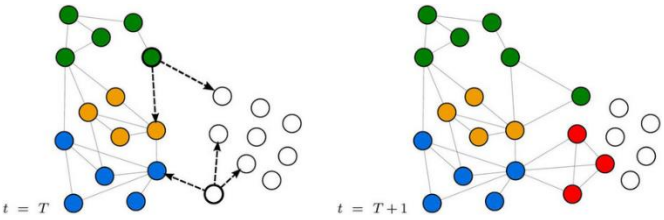
The goal is to weight every single new user according to some constant criteria, so users would "worth more or less" for the learner to use in the learning process. In this case, the idea to use the **age in the game**: the difference in days between today and the date a user first installed the game.

The idea is that a *Live game is a fluid thing*. It evolves and changes - for example, as certain Clans become more or less powerful, as new users join and old users retire, and - above all - as users buy new devices and as the competition in the market changes over time.

Our selected features are already pointing this is a direction worth exploring. **The selected features with the suffix "joined_after_trackers" are temporal ones,** indicating when an user joined the game after a certain feature started to be tracked at all across the many minor updates of version 4. The fact those feature swere chosen is a strong indication that **the game became better at converting users over time,** and it makes a lot of sense to factor this in the model.

3. Label Propagation / Spreading

This is a social game. Not only all progression systems are based on player vs. player, adversarial gameplay, but players also congregate in clans in order to beat other clans. And in clan-based strategy games, it is well-known in the mobile games industry the tendency of come of these games to stimulate "social spending" - features that enable certain "major" players to stimulate the rest of the Clan to purchase additional resources, and to continue to purchase again and again as new Clan Wars come buy.



Thus, it's possible we could use Label Propagation or Lable Spreading techniques to add information about potential Paying Users to the model. By wrangling other types of social interaction data from the original database into a "learnable" format like *player_data.csv*, we could **train a separate Label Spreading and add the class labeling as an extra feature of our model.**

4. Ensemble

XGBoost is already an ensemble algorithm. Akin to a Random Forest, it trains lots of "weak" Decision Trees, then combining the results with boosting.

However, it is feasible different approaches to machine learning could yield different results in terms of classification of who will be a Paying User. A Neural Network or even a Logistic Regression can have different decision boundaries. It is feasible classification could improve by ensembling these algorithms together into a higher-level meta-learner, together with XGBoost.

It's also possible XGBoost itself could be configured in a very different manner, because the algorithm has a fairly big amount of hyperparameters and "ways to do things", thus being ensembled with our main model in this project.

5. Less than 3 Days

Once the model improves on its scores on the test sets, we could attempt to reduce the amount of days since the app install that we are using as a threshold to accumulate data and feed the learner.

Ideally, we would be able to reduce this number all the way to 16 hours or less. The sooner we can identify these users, the faster Live Ops teams can act. The ability to identify a list of potential users overnight could significantly improve the ratio of daily conversions and first purchases in the game.

6. Online Learning

Finally, a crucial development to make this model available for Production would be to transform it into an online learner, gradually improving predictions as new users come by and behaviors change over time (which would make the Observation Weighting with Age even more important).

Fortunately, XGBoost is capable of partial fits, and very much compatible with online learning.

Final Words

While still not perfect, this model can be the beginning of a very interesting new Business Intelligence service for Live Ops teams trying to improve their revenue per user in free-to-play games.

The ability to predict which users have potential to become Paying Users can make a huge difference in the bottom line of a development team wondering what features to develop, what power-ups to give, what offers to make, to which users.

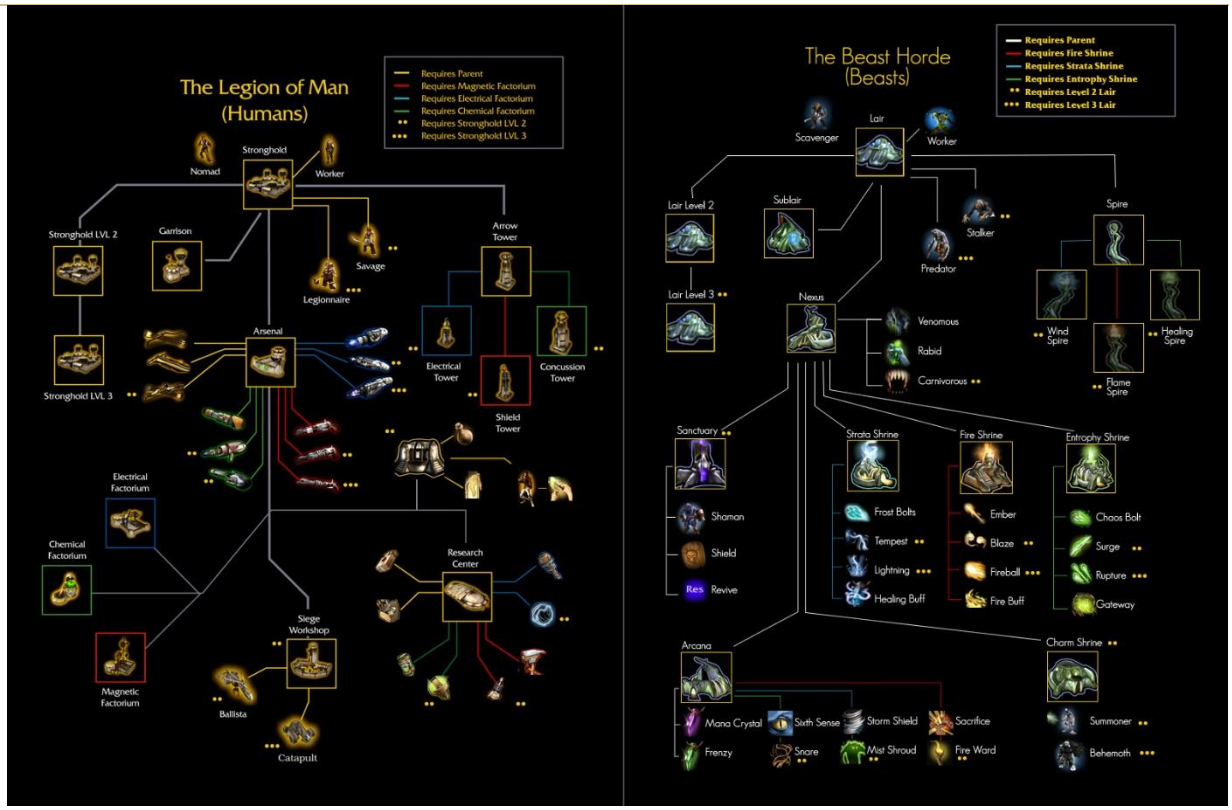
As the free-to-play model continues to dominate the mobile app stores, the consolidation of the market around fewer and fewer games is forcing companies to think out of the box and look for solutions.

Thanks for reading!

- Tiago

Appendix for "Predicting Paying Users in a Free-to-Play Game"

"Tech trees" from the free-to-play game *Newerth*:

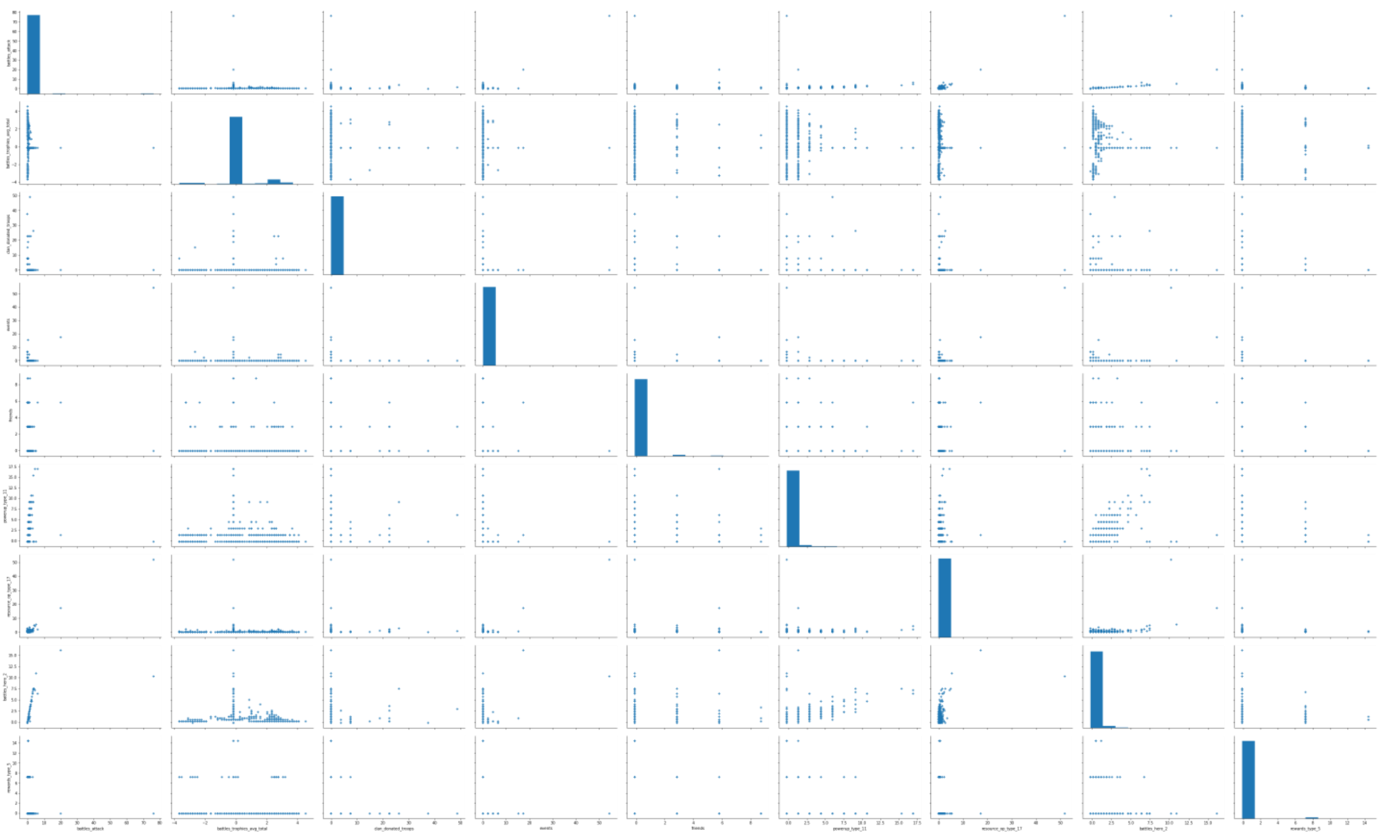


Fan-made table of costs of resources and time to build in *Clash of Clans*

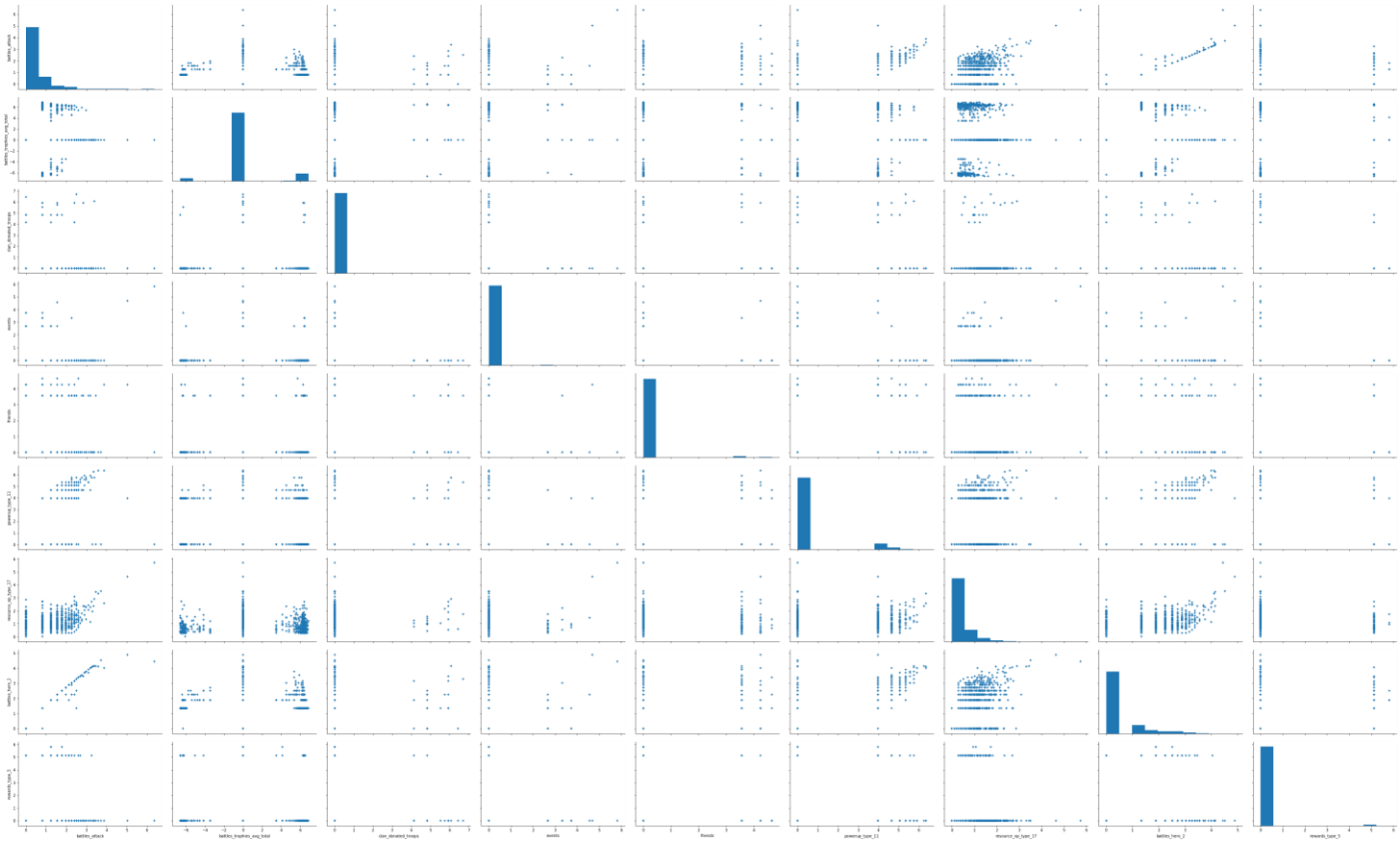
	Resource	1		2		3		4		5		6		7		8		9		10		11		12	
Major Infrastructure																									
Town Hall	Gold	-	N/A	1,000	1min	4,000	3hr	25,000	1 day	150,000	2 days	750,000	4 days	1,200,000	6 days	2,000,000	8 days	3,000,000	10 days	5,000,000	12 days	7,000,000	14 days		
Clan Castle	Gold	10,000	N/A	100,000	6hr	800,000	1 day	1,800,000	2 days	5,000,000	7 days	7,000,000	14 days												
Laboratory	Elixir	25,000	30min	50,000	5hr	90,000	12hr	270,000	1 day	500,000	2 days	1,000,000	4 days	2,500,000	5 days	4,000,000	6 days	6,000,000	7 days						
Spell Factory	Elixir	200,000	1 day	400,000	2 days	800,000	4 days	1,600,000	5 days	3,200,000	6 days														
Dark Spell Factory	Elixir	1,500,000	4 days	2,500,000	6 days	3,500,000	8 days																		
Heroes																									
Barbarian King	Dark Elixir	10,000	N/A	12,500	12hr	15,000	1 day	17,500	1.5 days	20,000	2 days	22,500	2.5 days	25,000	3 days	30,000	3.5 days	35,000	4 days	40,000	4.5 days	45,000	5 days	+ 5,000 | .5	Level 31:
Archer Queen	Dark Elixir	40,000	N/A	22,500	12hr	25,000	1 day	27,500	1.5 days	30,000	2 days	32,500	2.5 days	35,000	3 days	40,000	3.5 days	45,000	4 days	50,000	4.5 days	55,000	5 days	+ 5,000 | .5	Level 31:
Grand Warden	Elixir	6,000,000	N/A	2,500,000	12hr	3,000,000	1 day	3,500,000	1.5 days	4,000,000	2 days	4,500,000	2.5 days	+ 500,000 | .5	Level 13:	8,000,000	6 days	8,400,000	6.5 days	8,800,000	7 days	9,100,000	7 days	9,400,000	7 days
Troops																									
Barbarian	Elixir	-	N/A	50,000	6hr	150,000	1 day	500,000	3 days	1,500,000	5 days	4,500,000	10 days	6,000,000	14 days										
Archer	Elixir	-	N/A	50,000	12hr	250,000	2 days	750,000	3 days	2,250,000	5 days	6,000,000	10 days	7,500,000	14 days										
Giant	Elixir	-	N/A	100,000	1 day	250,000	2 days	750,000	3 days	2,250,000	5 days	6,000,000	10 days	7,000,000	14 days										
Goblin	Elixir	-	N/A	50,000	12hr	250,000	2 days	750,000	3 days	2,250,000	5 days	4,500,000	10 days												
Vall Breaker	Elixir	-	N/A	100,000	1 day	250,000	2 days	750,000	3 days	2,250,000	5 days	6,750,000	10 days												
Balloon	Elixir	-	N/A	150,000	1 day	450,000	2 days	1,350,000	3 days	2,500,000	5 days	6,000,000	10 days												
Vizard	Elixir	-	N/A	150,000	1 day	450,000	2 days	1,350,000	3 days	2,500,000	5 days	7,500,000	14 days												
Healer	Elixir	-	N/A	750,000	3 days	1,500,000	5 days	3,000,000	7 days																
Dragon	Elixir	-	N/A	2,000,000	7 days	3,000,000	10 days	7,000,000	14 days	8,000,000	14 days														
P.E.K.K.A.	Elixir	-	N/A	3,000,000	10 days	6,000,000	12 days	7,000,000	14 days	8,000,000	14 days														
Minion	Dark Elixir	-	N/A	10,000	5 days	20,000	6 days	30,000	7 days	40,000	10 days	100,000	14 days	140,000	14 days										
Hog Rider	Dark Elixir	-	N/A	20,000	8 days	30,000	10 days	40,000	12 days	50,000	14 days														
Valkyrie	Dark Elixir	-	N/A	50,000	10 days	60,000	12 days	70,000	14 days																
Golem	Dark Elixir	-	N/A	60,000	10 days	70,000	12 days	80,000	14 days	90,000	14 days														
Vitch	Dark Elixir	-	N/A	75,000	10 days	160,000	14 days																		
Lava Hound	Dark Elixir	-	N/A	60,000	10 days	70,000	12 days																		
Spells																									
Lightning	Elixir	-	N/A	200,000	1 day	500,000	2 days	1,000,000	3 days	2,000,000	4 days	6,000,000	10 days	8,000,000	14 days										
Healing	Elixir	-	N/A	300,000	1 day	600,000	2 days	1,200,000	3 days	2,400,000	5 days	4,800,000	7 days												
Rage	Elixir	-	N/A	450,000	2 days	900,000	3 days	1,800,000	5 days	3,000,000	7 days														
Jump	Elixir	-	N/A	3,000,000	5 days	6,000,000	7 days																		
Freeze	Elixir	-	N/A	4,000,000	5 days	5,000,000	7 days	6,500,000	10 days	8,000,000	14 days														
Poison	Dark Elixir	-	N/A	25,000	4 days	50,000	6 days	75,000	10 days																
Earthquake	Dark Elixir	-	N/A	30,000	6 days	60,000	8 days	90,000	12 days																
Haste	Dark Elixir	-	N/A	40,000	6 days	80,000	8 days	120,000	10 days																
Resources																									
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Mine	Elixir	150	1min	300	5min	700	15min	1,400	1hr	3,000	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Gold Storage	Elixir	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Gold Storage	Elixir	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Gold Storage	Elixir	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Gold Storage	Elixir	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Collector	Gold	150	1min	300	5min	700	15min	1,400	1hr	3,500	2hr	7,000	6hr	14,000	12hr	28,000	1 day	56,000	2 days	84,000	3 days	168,000	4 days	336,000	5 days
Elixir Storage	Gold	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Elixir Storage	Gold	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Elixir Storage	Gold	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days
Elixir Storage	Gold	300	1min	750	30min	1,500	1hr	3,000	2hr	6,000	3hr	12,000	4hr	25,000	6hr	50,000	8hr	100,000	12hr	250,000	1 day	500,000	2 days	2,500,000	7 days

Scatter Matrices

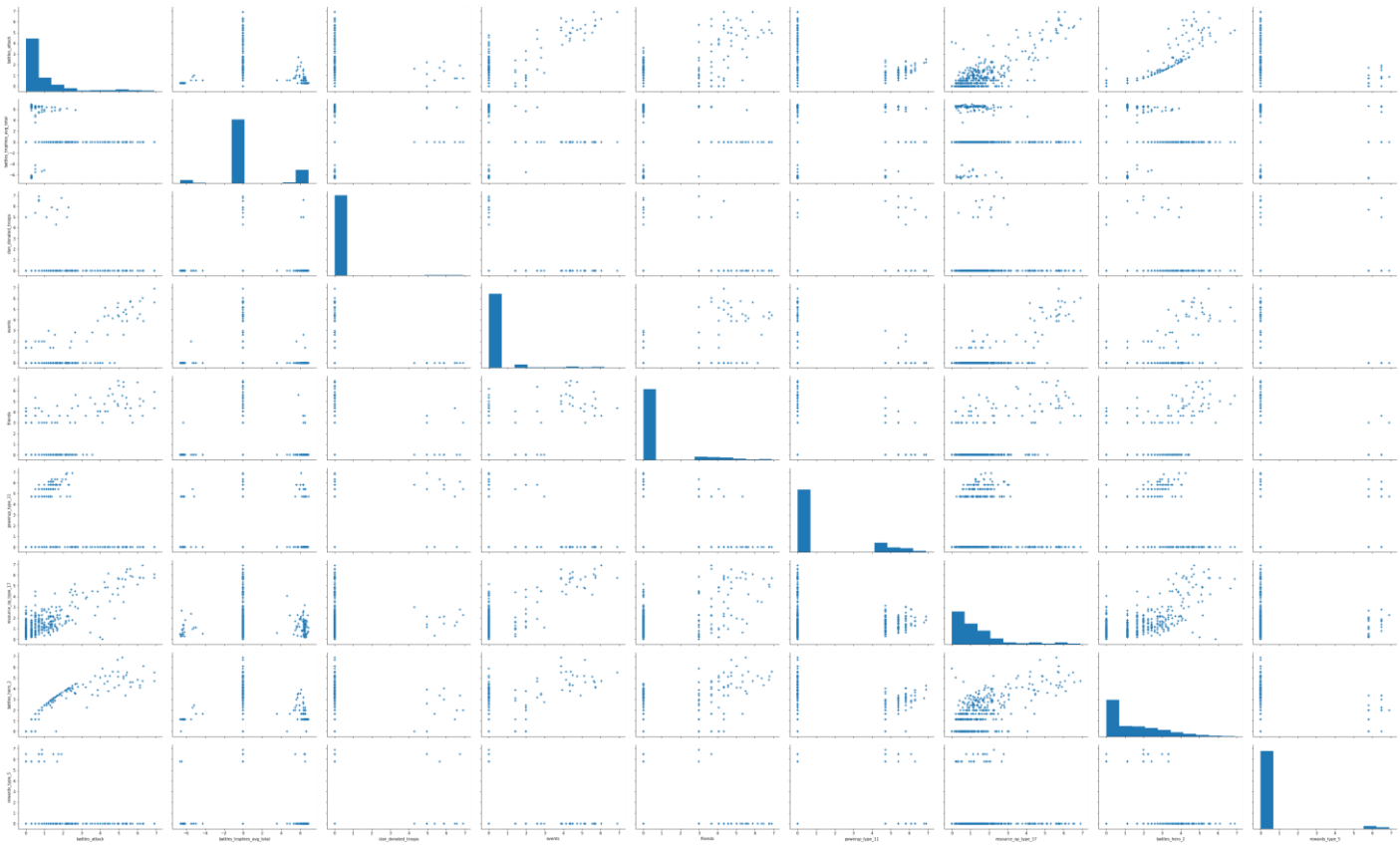
Initial scatter matrix on player data (features selected in the text).



Log-transformed matrix for ***Regular Users***:



Log-transformed matrix for ***Paying Users***:



FBeta demonstration

Equilibrium

Beta	1

Recall \ Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1	0.10	0.13	0.15	0.16	0.17	0.17	0.18	0.18	0.18	0.18
0.2	0.13	0.20	0.24	0.27	0.29	0.30	0.31	0.32	0.33	0.33
0.3	0.15	0.24	0.30	0.34	0.38	0.40	0.42	0.44	0.45	0.46
0.4	0.16	0.27	0.34	0.40	0.44	0.48	0.51	0.53	0.55	0.57
0.5	0.17	0.29	0.38	0.44	0.50	0.55	0.58	0.62	0.64	0.67
0.6	0.17	0.30	0.40	0.48	0.55	0.60	0.65	0.69	0.72	0.75
0.7	0.18	0.31	0.42	0.51	0.58	0.65	0.70	0.75	0.79	0.82
0.8	0.18	0.32	0.44	0.53	0.62	0.69	0.75	0.80	0.85	0.89
0.9	0.18	0.33	0.45	0.55	0.64	0.72	0.79	0.85	0.90	0.95
1.0	0.18	0.33	0.46	0.57	0.67	0.75	0.82	0.89	0.95	1.00

Skewed to Recall

Beta	3

Recall \ Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1	0.10	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
0.2	0.18	0.20	0.21	0.21	0.21	0.21	0.22	0.22	0.22	0.22
0.3	0.25	0.29	0.30	0.31	0.31	0.32	0.32	0.32	0.32	0.32
0.4	0.31	0.36	0.39	0.40	0.41	0.41	0.42	0.42	0.42	0.43
0.5	0.36	0.43	0.47	0.49	0.50	0.51	0.51	0.52	0.52	0.53
0.6	0.40	0.50	0.55	0.57	0.59	0.60	0.61	0.62	0.62	0.63
0.7	0.44	0.56	0.62	0.65	0.67	0.69	0.70	0.71	0.72	0.72
0.8	0.47	0.62	0.69	0.73	0.75	0.77	0.79	0.80	0.81	0.82
0.9	0.50	0.67	0.75	0.80	0.83	0.86	0.88	0.89	0.90	0.91
1.0	0.53	0.71	0.81	0.87	0.91	0.94	0.96	0.98	0.99	1.00