AWS FOR GAMES

実践 ！ 機械学習でゲームの休眠ユーザーを予測してみよう

Sheng Hsia Leng
アマゾン ウェブ サービス ジャパン合同会社
ソリューションアーキテクト

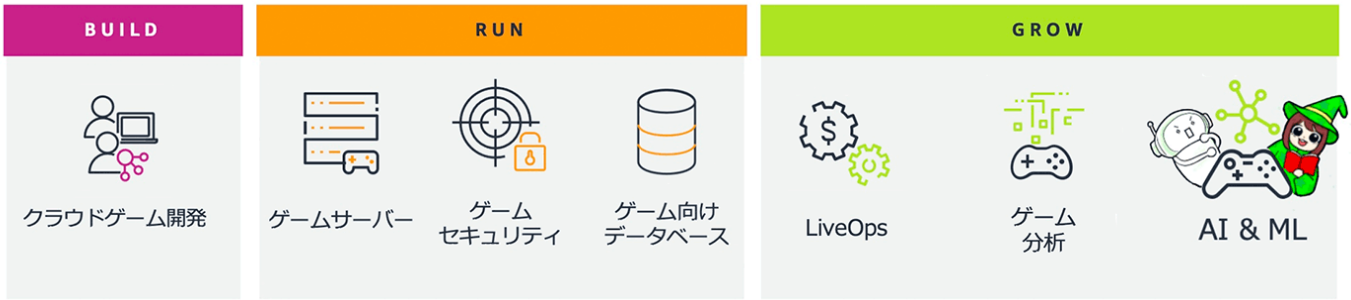
保里 善太
アマゾン ウェブ サービス ジャパン合同会社
シニアソリューションアーキテクト

ゲームをこよなく愛する皆さん、こんにちは！Game Solutions Architect の Leng (@msian.in.japan) と Partner Solutions Architect の Zen (@HoriZenta) です。

前々回、前回とゲーム分野に AI/ML を活用するための概要をお話ししました。特に前回は AI/ML の活用ステップを理解していたくだために「ユーザー離脱予測」をベースに「何を準備して」「どのように作っていくのか」を 3 つのステップでお伝えしました。今回は前回の記事をもとに実際に機械学習モデルを作ってユーザー離脱予測を実践してみましょう。また、ユーザー離脱予測モデルから、ユーザー離脱の要因分析をする方法にも触れます。

AWS FOR GAMES

AWS for Games はより早い開発、よりスマートな運営、そしてより楽しいゲームへの成長という Build、Run、Grow の 3 つの柱に沿ってサポートします。本日は Grow の柱、中でもゲームの運営に重要なゲーム分析と AI/ML に焦点を当てています。



ゲーム運営上、ユーザーのゲーム継続率を維持するために、ユーザーの休眠や離脱を事前に予測しそれに対して効果的な対策を行うことは重要です。例えば、事前に予測したゲームをやめそうなユーザーに対してはアイテムを付与したり特定のイベントに招待する通知を送り離脱を阻止する対策が考えられます。

また、後ほど述べるようにユーザー離脱の要因分析を行うことで、ユーザーが継続しやすいゲームシナリオへ改善する対策も取ることができます。

どうやるのか ？

ここでは、潜在的な休眠ユーザーの予測に Amazon SageMaker Studio と AutoGluon という機械学習ライブラリを利用します。AutoGluon は Amazon が提供する AutoML ライブラリであり、ここでは表形式のデータを推論するための AutoGluon-Tabular を利用します。Amazon SageMaker Studio の環境セットアップについては こちらのブログ を参考にしてください。

また、機械学習にはデータが必要です。今回は、とあるゲームデータをサンプルとして利用します。それぞれのデータはデータベースに保存されているものを CSV ファイルとして出力してあります。

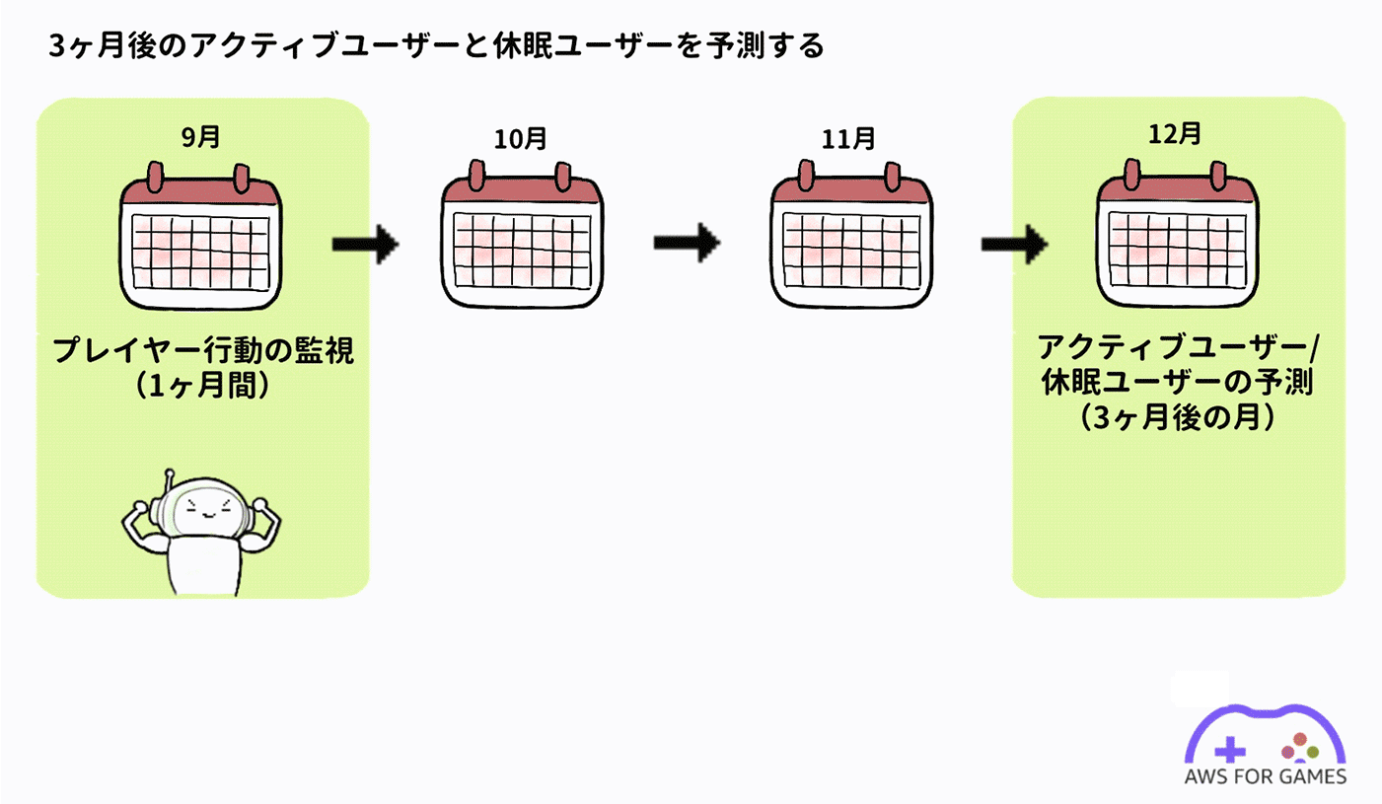
ここでは一般的なゲームプロダクトの運営でよく取得されている時系列ベースのユーザー行動を記録したゲームログ（プレイログ）とユーザーのサマリー情報を持つ二種類のデータを用います。ゲームログは目的に応じて、gameplay テーブルと sales テーブルと login テーブルに分かれています。ユーザーのサマリー情報は、user テーブルに保存されています。

データの詳細は、「今回用いたデータの説明」を参照してください。

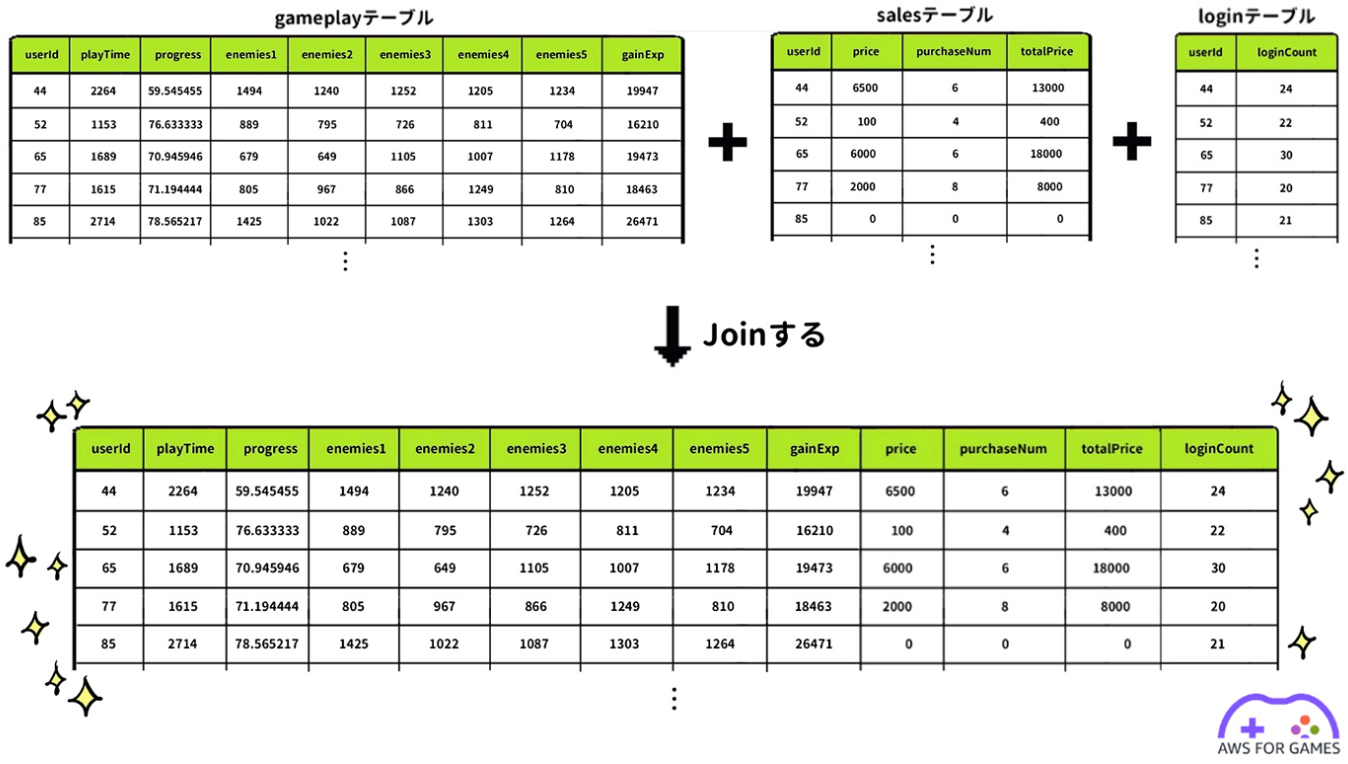
前提条件を決める

今回は一定期間プレイヤーの行動を監視して、そのユーザーが継続してゲームをプレイし続けるか否かを推定します。ユーザー行動の分析には、通常ゲームサーバーに保存されるゲームログ（プレイログ）を利用します。

今回は 1 ヶ月間の行動ログを元に 3 ヶ月後（ここでは便宜上 90 日後）の月のアクティブユーザーと休眠ユーザーを予測することにします。



行動ログはデータベースに保存されており、それぞれ gameplay テーブルと sales テーブルと login テーブルに分かれています。これらは時系列のデータとして保存されているので、ユーザー毎に集計した後、テーブルを Join して一つの行動ログとして準備します。学習に用いるユーザー行動ログの期間は 2021/9/1 - 2021/9/30 の 30 日間とします。



gameplayテーブル

userId	playTime	progress	enemies1	enemies2	enemies3	enemies4	enemies5	gainExp
44	2264	59.545455	1494	1240	1252	1205	1234	19947
52	1153	76.633333	889	795	726	811	704	16210
65	1689	70.945946	679	649	1105	1007	1178	19473
77	1615	71.194444	805	967	866	1249	810	18463
85	2714	78.565217	1425	1022	1087	1303	1264	26471

salesテーブル

userId	price	purchaseNum	totalPrice
44	6500	6	13000
52	100	4	400
65	6000	6	18000
77	2000	8	8000
85	0	0	0

loginテーブル

userId	loginCount
44	24
52	22
65	30
77	20
85	21

Joinする

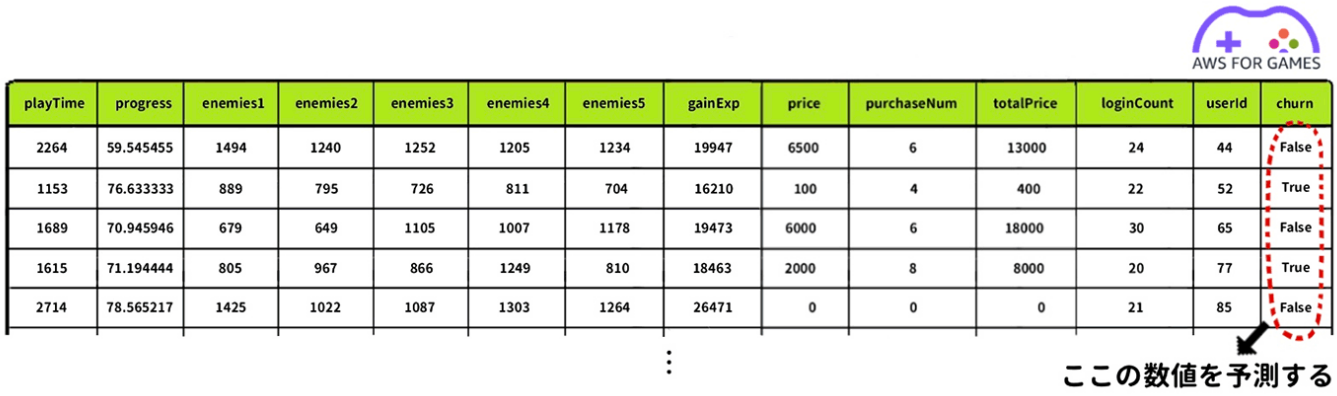
userId	playTime	progress	enemies1	enemies2	enemies3	enemies4	enemies5	gainExp	price	purchaseNum	totalPrice	loginCount
44	2264	59.545455	1494	1240	1252	1205	1234	19947	6500	6	13000	24
52	1153	76.633333	889	795	726	811	704	16210	100	4	400	22
65	1689	70.945946	679	649	1105	1007	1178	19473	6000	6	18000	30
77	1615	71.194444	805	967	866	1249	810	18463	2000	8	8000	20
85	2714	78.565217	1425	1022	1087	1303	1264	26471	0	0	0	21

ユーザー行動を監視してから 90 日後、つまりこの場合 2021/11/30 以降にユーザーがアクティブかどうかを分析してラベル付けを行い教師データとします。

アクティブなユーザーについてはサマリー情報を持つ user テーブルを使って分析します。user テーブルには通常 lastLoginTime のようなユーザーの最終ログイン日時が記録されたカラムが存在するからです。

これを用いて、 2021/11/30 以降でもログインしているユーザーを継続ユーザー（churn = False）と定義し、ログインしていないユーザーを離脱ユーザー（churn = True）と定義してフラグを追加します。そして上で用意した行動ログのユーザー ID と突き合わせ、各ユーザーの行動データに対して churn フラグ列をマージすることでラベル付けを行い、教師データを完成させます。

ここで完成した教師データの最後の列、churn フラグ列の True / False を予測することが今回の機械学習モデルの目的です。



なお、今回は休眠ユーザーの予測の大まかなプロセスを理解していただくことが目的ですので、予測の精度をチューニングして深く追い込んでいくようなことはしません。

セットアップ

さて、それでは Amazon SageMaker Studio を開いて、実際に手を動かしてユーザー離脱予測モデルを構築してみましょう。

こちらのチュートリアルでは、SageMaker Studio のインスタンスタイプとして ml.m5.xlarge を利用することをお勧めします。事前に ml.m5.xlarge が選択されているかご確認ください。

AutoGluon をインストールします。SageMaker Studio ノートブックで Python 3 (MXNet 1.8 Python 3.7 CPU Optimized) のカーネルが選択されていれば、以下のコマンドだけでインストールが可能です。インストールの詳細については こちら をご参照ください。Jupyter 環境では、冒頭に ！ を付けることでコマンドを実行できます。

```
!pip install autogluon
```

このノートブックで使用するライブラリをインポートします。

```
# AutoGluon をインポートする。

from autogluon.tabular import TabularDataset, TabularPredictor



# その他のライブラリをインポートする。

import pprint

import os

from IPython.display import display

from IPython.display import HTML

import pandas as pd

import numpy as np

import datetime
```

データのダウンロード

今回こちらのチュートリアルで使用するゲームのデータをダウンロードします。

```
!wget https://gametech-cfn-templates-public.s3.amazonaws.com/dataset/game_data_sample.zip
```

データは zip で圧縮されているので解凍します。

```
!unzip -qq -o "game_data_sample.zip"
```

解凍後に game_data_sample というディレクトリが作成され、その中に諸々の CSV データが保存されています。ファイルパスのベースディレクトリとして設定しておきます。

```
# ゲームデータ保存先のディレクトリ

basedir = 'game_data_sample'
```

前処理・データの準備

行動ログの分析と前処理

学習と予測に用いるプレイヤーの行動ログを用意します。先ほども述べたように学習と予測に用いるユーザー行動ログの期間は 2021/9/1 - 2021/9/30 の 30 日間ですので、データサンプリング期間（Time Window）を下記のように設定します。

```
# 今回の行動ログのサンプリング期間

window_start = '20210901'

window_end = '20210930'
```

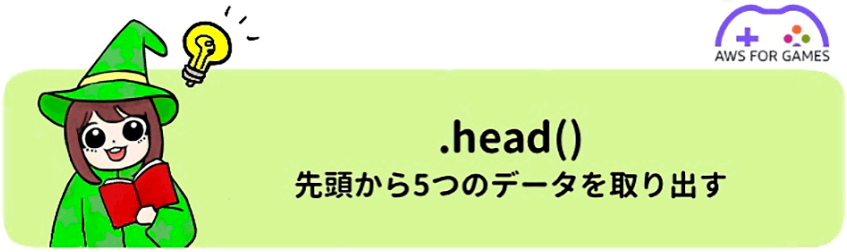
行動ログは時系列のログとして login テーブル、gameplay テーブル、そして sales テーブルに分かれてデータベースに保存されていますので、まずはそれぞれのテーブルに対してユーザー毎の集計を行います。

データサンプリング期間をユーザー単位でグルーピングした後、各ユーザーのログイン回数やシナリオの進捗状況、アイテム課金額などを集計します。最後に集計した 3 つのテーブルを Join (結合) して一つの行動ログを作ります。

次の処理は login テーブルを CSV ファイルから読み込んで、データサンプリング期間内での各ユーザー毎のログイン回数を集計しています。

```
login_data = pd.read_csv(os.path.join(basedir, 'login.csv'))

login_data['loginTime'] = pd.to_datetime(login_data['loginTime'], format='%Y/%m/%d %H:%M:%S')

login_data = login_data[(pd.to_datetime(window_start, format='%Y/%m/%d', utc=True) <= login_data['loginTime']) &
(pd.to_datetime(window_end, format='%Y/%m/%d', utc=True) >= login_data['loginTime'])]

login_data = login_data.groupby('userId').count()

login_data = login_data.rename(columns={'loginTime':'loginCount'})

login_data.head()
```

userId	loginCount
44	24
52	22
65	30
77	20
85	21



次の処理では gameplay テーブルを CSV ファイルから読み込んで、データサンプリング期間内での各ユーザー毎のゲームプレイ結果を集計しています。集計前に id カラムやアイテム名など休眠ユーザーの予測に寄与しないカラムは予め削除しています。

```
gameplay_data = pd.read_csv(os.path.join(basedir, 'gameplay.csv'))

gameplay_data['startTime'] = pd.to_datetime(gameplay_data['startTime'], format='%Y/%m/%d %H:%M:%S')
```

```
gameplay_data = gameplay_data[(pd.to_datetime(window_start, format='%Y/%m/%d', utc=True) <=
gameplay_data['startTime']) & (pd.to_datetime(window_end, format='%Y/%m/%d', utc=True) >=
gameplay_data['startTime'])]

gameplay_data = gameplay_data.drop(['id', 'startTime', 'dungeonId', 'result', 'friendId', 'userCharacterId',
'userCharacterName', 'userWeaponId', 'userWeaponName', 'userSkinId', 'userSkinName'], axis=1)

gameplay_data = gameplay_data.groupby('userId').agg({'playTime': np.sum, 'progress': np.mean, 'enemies1': np.sum,
'enemies2': np.sum, 'enemies3': np.sum, 'enemies4': np.sum, 'enemies5': np.sum, 'gainExp': np.sum})

gameplay_data.head()
```

userId	playTime	progress	enemies1	enemies2	enemies3	enemies4	enemies5	gainExp
44	2264	59.545455	1494	1240	1252	1205	1234	19947
52	1153	76.633333	889	795	726	811	704	16210
65	1689	70.945946	679	649	1105	1007	1178	19473
77	1615	71.194444	805	967	866	1249	810	18463
85	2714	78.565217	1425	1022	1087	1303	1264	26471

次の処理では sales テーブルを CSV ファイルから読み込んで、データサンプリング期間内での各ユーザーのアイテム購入数や課金額を集計しています。こちらも集計前に id カラムやアイテム名など休眠ユーザーの予測に寄与しないカラムは予め削除しています。

```
sales_data = pd.read_csv(os.path.join(basedir, 'sales.csv'))

sales_data['purchasedTime'] = pd.to_datetime(sales_data['purchasedTime'], format='%Y/%m/%d %H:%M:%S')

sales_data = sales_data[(pd.to_datetime(window_start, format='%Y/%m/%d', utc=True) <= sales_data['purchasedTime'])
& (pd.to_datetime(window_end, format='%Y/%m/%d', utc=True) >= sales_data['purchasedTime'])]

sales_data = sales_data.drop(['id', 'purchasedTime', 'store', 'itemId', 'itemName'], axis=1)

sales_data = sales_data.groupby('userId').agg({'price': np.sum, 'purchaseNum': np.sum, 'totalPrice': np.sum})

sales_data.head()
```

userId	price	purchaseNum	totalPrice
44	6500	6	13000
52	100	4	400
65	6000	6	18000
77	2000	8	8000
91	5000	3	15000

最後に集計した gameplay テーブルと login テーブルと sales テーブルを結合して一つのデータとしてまとめます。

```
data = pd.merge(gameplay_data, sales_data, on='userId', how='left')

data = pd.merge(data, login_data, on='userId', how='left')

data = data.fillna(0)

data.head()
```

userId	playTime	progress	enemies1	enemies2	enemies3	enemies4	enemies5	gainExp	price	purchaseNum	totalPrice	loginCount
44	2264	59.545455	1494	1240	1252	1205	1234	19947	6500.0	6.0	13000.0	24.0
52	1153	76.633333	889	795	726	811	704	16210	100.0	4.0	400.0	22.0
65	1689	70.945946	679	649	1105	1007	1178	19473	6000.0	6.0	18000.0	30.0
77	1615	71.194444	805	967	866	1249	810	18463	2000.0	8.0	8000.0	20.0
85	2714	78.565217	1425	1022	1087	1303	1264	26471	0.0	0.0	0.0	21.0

これでデータサンプリング期間内におけるユーザーの行動ベースのログを作成することができました。このデータを学習させてモデルを作ります。

ただ、この時点では、各ユーザーの行動結果に対してそれが離脱につながるのか、継続につながるのかを判別する正解ラベルが付与されていません。そこで次に、教師あり学習の学習用データを作成するために、得られた行動ログに正解ラベルをラベリングしていきます。

行動ログにラベル付を行い教師データの完成

ここでは先ほど得られた行動ログに対して、実際に 90 日後にログインしているユーザーとそうでないユーザーに対して正解のラベル付けを行います。

ここで休眠ユーザー予測の目的をもう一度確認しておきましょう。今回の目的は 1 ヶ月間の行動ログを元に 3 ヶ月後（便宜上 90 日後）の月のアクティブユーザーと休眠ユーザーを予測することでした。

ユーザーがアクティブか非アクティブかはユーザーのログイン履歴から確認します。各ユーザーの最終ログイン履歴を確認するには user テーブルを見ると分かります。user テーブルを俯瞰してみましょう。

```
user_data = pd.read_csv(os.path.join(basedir, 'user.csv'))

user_data.head()
```

最後の列に lastLoginTime というカラムがあり、これが各ユーザーの最終ログイン日時を表しています。先頭の id カラムがユーザーID を表しています。このテーブルにユーザー離脱の有無を表す churn 列を追加します。

	id	userName	totalTime	releasedDungeonId	level	clearTotalNum	coin	totalExp	friendsNum	startTime	lastLoginTime
0	1	user000001	68	5	4	11	255	935	4	2021-12-27T18:27:29.000Z	2021-12-31T23:07:00.000Z
1	2	user000002	2522	33	31	44	7041	3520	19	2021-11-12T21:22:02.000Z	2021-12-03T22:34:51.000Z
2	3	user000003	1266	19	25	32	3442	2720	18	2021-12-07T06:37:38.000Z	2021-12-25T20:04:07.000Z
3	4	user000004	757	15	16	21	3158	1869	6	2021-12-02T23:35:46.000Z	2021-12-31T18:05:23.000Z
4	5	user000005	1271	25	14	36	1905	3492	10	2021-12-11T14:04:52.000Z	2021-12-28T17:41:28.000Z

ユーザー休眠の定義に基づいて 2021/11/30 以降のログインが確認できないユーザーを休眠ユーザー（離脱ユーザー）として churn フラグを True に設定します。それ以外のユーザーは 2021/11/30 以降のログインが確認できるのでアクティブユーザーとして churn フラグを False に設定しています。

```
# 行動ログの観測から 90 日後に Active/Inactive ユーザーをチェック

activity_check_interval = 90



# timestamp を datetime 型に変換

user_data['startTime'] = pd.to_datetime(user_data['startTime'], format='%Y/%m/%d %H:%M:%S')

user_data['lastLoginTime'] = pd.to_datetime(user_data['lastLoginTime'], format='%Y/%m/%d %H:%M:%S')



# 90 日以降にアクティブでないユーザーの Churn フラグを True とする
```

```
user_data.loc[pd.to_datetime(window_start, format='%Y/%m/%d', utc=True) +
datetime.timedelta(days=activity_check_interval) > user_data['lastLoginTime'], 'churn'] = True



# それ以外のユーザーは１ヶ月後でもアクティブにログインしているので Churn は False

user_data['churn'] = user_data['churn'].fillna(False)

user_data.head()
```

実行すると先ほどの user テーブル に対して churn カラムが追加され、ユーザーの最終ログイン日時に応じて True/False フラグが追加されています。

	id	userName	totalTime	releasedDungeonId	level	clearTotalNum	coin	totalExp	friendsNum	startTime	lastLoginTime	churn
0	1	user000001	68	5	4	11	255	935	4	2021-12-27 18:27:29+00:00	2021-12-31 23:07:00+00:00	False
1	2	user000002	2522	33	31	44	7041	3520	19	2021-11-12 21:22:02+00:00	2021-12-03 22:34:51+00:00	False
2	3	user000003	1266	19	25	32	3442	2720	18	2021-12-07 06:37:38+00:00	2021-12-25 20:04:07+00:00	False
3	4	user000004	757	15	16	21	3158	1869	6	2021-12-02 23:35:46+00:00	2021-12-31 18:05:23+00:00	False
4	5	user000005	1271	25	14	36	1905	3492	10	2021-12-11 14:04:52+00:00	2021-12-28 17:41:28+00:00	False

ここまで 2021/9/1 - 2021/9/30 の 30 日間のユーザーの行動データの集計結果（data）と各ユーザーが 2021/11/30 以降に休眠しているかアクティブなのかのデータ（user_data）が出揃いました。これらは別々のテーブルに存在しているので、最後にユーザー ID を突き合わせて各ユーザーの行動データに対して churn フラグを正解データとしてラベリングします。

```
data = pd.merge(data, user_data[['id', 'churn']], left_on='userId', right_on='id', how='left')

data = data.rename(columns={'id':'userId'})

data.head()
```

ユーザーの行動ログに対して、一番右側に churn カラムが追加され、正解ラベルつきの行動ログが完成したことがわかります。

なお、右から二番目にある userId カラムは単なるユーザー ID であり、特徴量としては休眠ユーザーの予測に寄与しませんが、どのユーザーが離脱するかを予測した結果を分析する際にユーザー ID があった方が運用上便利ですのでデータの中に残してあります。

	playTime	progress	enemies1	enemies2	enemies3	enemies4	enemies5	gainExp	price	purchaseNum	totalPrice	loginCount	userId	churn
0	2264	59.545455	1494	1240	1252	1205	1234	19947	6500.0	6.0	13000.0	24.0	44	False
1	1153	76.633333	889	795	726	811	704	16210	100.0	4.0	400.0	22.0	52	True
2	1689	70.945946	679	649	1105	1007	1178	19473	6000.0	6.0	18000.0	30.0	65	False
3	1615	71.194444	805	967	866	1249	810	18463	2000.0	8.0	8000.0	20.0	77	True
4	2714	78.565217	1425	1022	1087	1303	1264	26471	0.0	0.0	0.0	21.0	85	False

学習用データと検証用データの用意

正解ラベルつきの行動ログを学習用データと学習後のモデルの精度を検証するための検証用データに分割します。ここでは、学習用データ：検証用データを 7 : 3 の割合で分割しています。

```
# 学習とテストデータに分割する。

train_data = data.sample(frac=0.7, random_state=42)

test_data = data.drop(train_data.index)



# テストデータの入力と出力を分割する。

label_column = "churn"

y_test = test_data[label_column]

X_test = test_data.drop(columns=[label_column])
```

今回予測するターゲットは、潜在的な休眠ユーザーで行動ログの一番右列の churn カラムです。このカラムの頻度を確認しておきましょう。

```
print("ターゲット変数の値と頻度: ¥n", train_data[label_column].value_counts().to_string())
```

churn カラムの頻度が True : False = 5750 : 5129 となっており、どちらかの値に偏っていたり数が少なすぎるわけでもなく、学習に適しているデータであることがわかります。

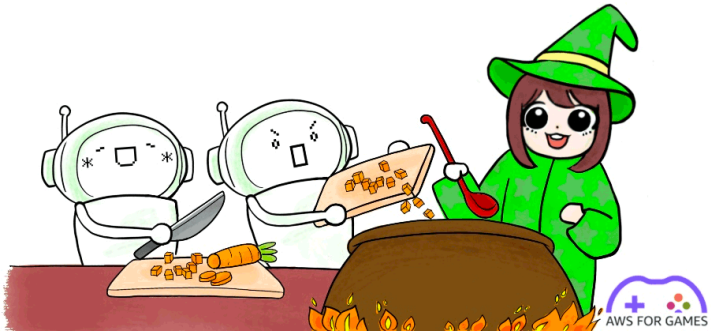
```
print("ターゲット変数の値と頻度: \n", train_data[label_column].value_counts().to_string())
```

```
ターゲット変数の値と頻度:
 True     5750
False    5129
```

AutoGluon を使って学習を開始

さて、いよいよ学習プロセスになります。ここで初めて AutoML である AutoGluon-Tabular の威力が発揮されます。

料理で言えば実際に火を通して調理するプロセスになります。それまではいわばデータの下拵えでした。



データと予測したいカラム名、モデルを保存するパスを指定して fit() メソッドを実行すると学習を開始します。

```
# 学習したモデルを保存するディレクトリを指定する。

dir_base_name = "churnModels"

dir_best = f"{dir_base_name}_best"
```

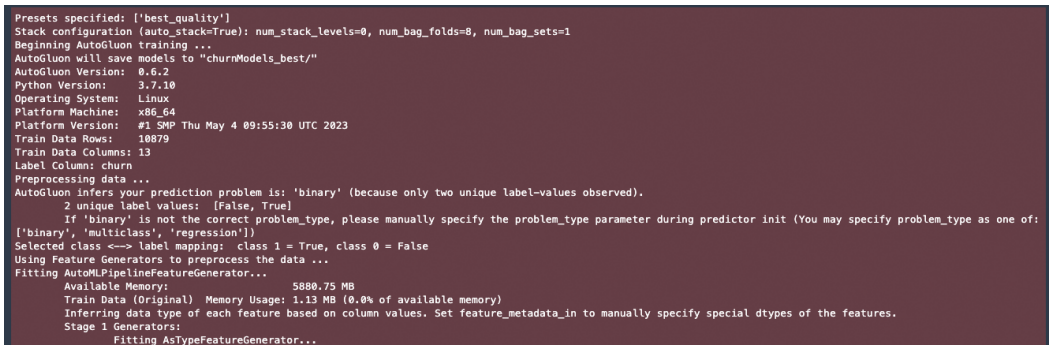
デフォルトでは精度とコスト（メモリ使用量、推論速度、ディスク容量等）においてバランスが取られた設定となっていますが、今回は精度を優先してモデル構築をしたいので、 fit() メソッドの引数で presets='best_quality' を指定します。

```
%%time

predictor = TabularPredictor(label=label_column, path=dir_best).fit(train_data=train_data, presets='best_quality')
```

実行を開始すると学習状態が色々と表示されます。その中で

AutoGluon infers your prediction problem is: 'binary' (because only two unique label-values observed).

というような記述があり AutoGluon が自動で今回のタスクは True/False の二値分類問題であると推論したことがわかります。私の環境では 32 秒ほどで学習が終了しました。



モデルの性能評価

晴れて学習が終わりモデルが作成されました。ここでは先ほど作成した検証用データを用いて、このモデルの精度を確認してみましょう。

先ほどの fit() メソッドの実行結果とモデルを predictor という変数に格納しました。これは AutoGluon の TabularPredictor オブジェクトになっていて、predict() というメソッドで予測をしたり、evaluate_predictions() というメソッドで性能を確認することができます。

```
y_pred = predictor.predict(X_test)

perf = predictor.evaluate_predictions(y_true=y_test, y_pred=y_pred, auxiliary_metrics=True)

print("Predictions:¥n", y_pred)
```

実行するとモデルの精度と検証データに対して実際に True/False を予測した結果が表示されます。

```
Evaluation: accuracy on test data: 0.6522951522951523
Evaluations on test data:
{
    "accuracy": 0.6522951522951523,
    "balanced_accuracy": 0.6535488801754503,
    "mcc": 0.30675011897592996,
    "f1": 0.6569312169312169,
    "precision": 0.6852097130242826,
    "recall": 0.6308943089430894
}
Predictions:
 1         True
4          True
5          True
6          True
7         False
          ...
15521      True
15524      True
15527      True
15529      True
15538      True
Name: churn, Length: 4662, dtype: bool
```

機械学習ライブラリの scikit-learn を使って、検証データの予測結果を混同行列（confusion matrix）で表示してみましょう。

```
from sklearn.metrics import confusion_matrix

# ラベルの順序を指定

labels = [1, 0]

# 混同行列の取得&ラベル順序指定

cm = confusion_matrix(y_test, y_pred, labels=labels)

# 出力を整形

columns_labels = ["pred_" + str(l) for l in labels]

index_labels = ["act_" + str(l) for l in labels]

cm = pd.DataFrame(cm, columns=columns_labels, index=index_labels)

print(cm.to_markdown())
```

```
|       |   pred_1 |   pred_0 |
|:------|---------:|---------:|
| act_1 |     1552 |      908 |
| act_0 |      713 |     1489 |
```

混同行列とモデルの精度の意味は、下の図をご覧ください。

これによると予測の精度としては、65 % の Accuracy (正解率) であることがわかります。ここの数値は学習の過程で場合によって微妙に変わってくると思いますが、65 % から大きく外れた数値になることはないと思います。65 % の正解率が要件に合わない場合さらにチューニングすることもできますが、これ以上の精度を求めるには追加の特徴量が必要になってくるでしょう。また Precision (適合率) が 70 % 近くと比較的高いので False Positive は抑えられていると考えられます。

Confusion Matrix		モデルの予測ラベル	
		True	False
離脱の正解ラベル	True	1552 (True Positive)	908 (False Negative)
	False	713 (False Positive)	1489 (True Negative)

65%の推論精度で予測が可能

メトリクス	結果	説明
Accuracy (正解率)	0.65	離脱あり、なしと予測したデータのうち、実際に正解したものの割合: accuracy = (tp+tn)/(tp+tn+fp+fn)
Precision (適合率)	0.69	離脱ありと予測したデータのうち実際に離脱であるものの割合: precision = tp/(tp+fp)
Recall (再現性)	0.63	実際の離脱データのうち、離脱と予測されたものの割合: recall = tp/(tp+fn)
F1スコア	0.66	適合率と再現率の調和平均: f1 = (2*precision*recall)/(precision+recall)

モデルの学習中に何が起きていたのかを確認してみましょう。

fit_summary() メソッドを実行すると、学習の過程においてどのような探索を行ったのか確認できます。

```
results = predictor.fit_summary()
```

学習したそれぞれのモデルについて検証データにおける性能、学習にかかった時間などが表示されています。7 種類のモデル名が表示されており、AutoGluon が様々なモデルを学習したことが分かります。

Types of models trained というところを確認すると、StackerEnsembleModel_KNN など複数のベースとなるモデルと、それらをアンサンブルしたモデルになっています。

```
results = predictor.fit_summary()
*** Summary of fit() ***
Estimated performance of each model:
                     model  score_val  pred_time_val   fit_time  pred_time_val_marginal  fit_time_marginal  stack_level  can_infer  fit_order
0       WeightedEnsemble_L2   0.661458       2.390724  22.935209                0.018216           2.166535            2       True          7
1  RandomForestGini_BAG_L1   0.655207       0.511417   5.634125                0.511417           5.634125            1       True          3
2    ExtraTreesEntr_BAG_L1   0.654932       0.572796   2.527168                0.572796           2.527168            1       True          6
3    ExtraTreesGini_BAG_L1   0.654380       0.560919   2.226732                0.560919           2.226732            1       True          5
4  RandomForestEntr_BAG_L1   0.652725       0.517618  10.339825                0.517618          10.339825            1       True          4
5    KNeighborsUnif_BAG_L1   0.598584       0.103578   0.020335                0.103578           0.020335            1       True          1
6    KNeighborsDist_BAG_L1   0.595459       0.106179   0.020489                0.106179           0.020489            1       True          2
Number of models trained: 7
Types of models trained:
{'StackerEnsembleModel_KNN', 'WeightedEnsembleModel', 'StackerEnsembleModel_RF', 'StackerEnsembleModel_XT'}
Bagging used: True  (with 8 folds)
Multi-layer stack-ensembling used: False
Feature Metadata (Processed):
(raw dtype, special dtypes):
('float', []) : 5 | ['progress', 'price', 'purchaseNum', 'totalPrice', 'loginCount']
('int', [])   : 8 | ['playTime', 'enemies1', 'enemies2', 'enemies3', 'enemies4', ...]
Plot summary of models saved to file: churnModels_best/SummaryOfModels.html
*** End of fit() summary ***
```

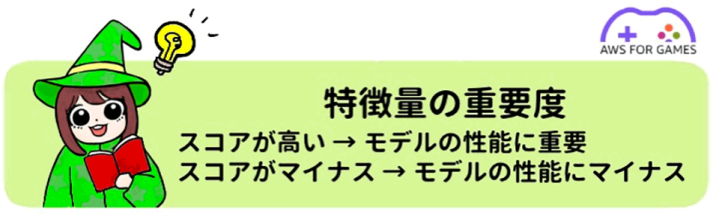
モデル予測の要因分析

ここまででユーザー休眠予測モデルを作ることができました。このモデルを使えば、季節性を無視した場合、年間のどの月でも 30 日間の行動ログを元に 90 日後（約 3 ヶ月後）のアクティブユーザーと休眠ユーザーを予測することができます。

では、実際に休眠する可能性のあるユーザーを予測した場合、どのようなアクションが取れるでしょうか？

一つは、モデルの予測に寄与した特徴量を分析することで、どのような要因でユーザーが離脱に至っているのかを推測することができます。離脱の要因になっているパラメータを調整することでゲーム離脱を阻止するゲームシナリオへと改善できるかもしれません。
下記のメソッドで計算される特徴量の重要度スコアは、スコアが高いほどその特徴量がモデルの性能に重要であると考えられます。スコアがマイナスの場合、その特徴量がモデルの性能にとってマイナスである可能性があります。詳しくは こちら をご確認ください。

```
feature_importances = predictor.feature_importance(test_data)

print("特徴量の重要度:")

display(feature_importances)
```



importance の数値を見ると、 gainExp という特徴量が一番モデルの性能に寄与していることがわかります。
gainExp は獲得経験値を表すパラメータです。獲得経験値が高ければ高いほど、ゲームへ没頭し依存していることが窺えるので、ユーザーの継続性の予測に寄与していることがなんとなく理解できますね。

次に数値が高いのが、 progress というパラメータです。progress はプレイ進捗状況（100 % 中どこまでクリアできたか）を表すので、この数値が高いユーザーの継続率が高くなることもなんとなく予想ができます。

```
feature_importances = predictor.feature_importance(test_data)
print("特徴量の重要度:")
display(feature_importances)
```

```
Computing feature importance via permutation shuffling for 13 features using 4662 rows with 5 shuffle sets...
        125.39s = Expected runtime (25.08s per shuffle set)
        52.34s  = Actual runtime (Completed 5 of 5 shuffle sets)
```

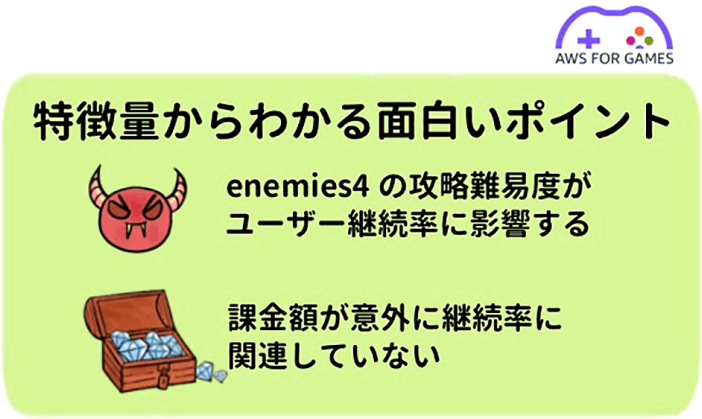
特徴量の重要度:

	importance	stddev	p_value	n	p99_high	p99_low
gainExp	0.036122	0.007953	0.000265	5	0.052498	0.019746
progress	0.016345	0.001418	0.000007	5	0.019265	0.013425
enemies4	0.016345	0.003962	0.000384	5	0.024503	0.008187
enemies5	0.015616	0.002553	0.000083	5	0.020873	0.010358
enemies1	0.012227	0.002776	0.000298	5	0.017943	0.006510
loginCount	0.010725	0.002709	0.000450	5	0.016303	0.005147
playTime	0.010039	0.006740	0.014544	5	0.023915	-0.003838
enemies3	0.007550	0.003450	0.004041	5	0.014654	0.000447
enemies2	0.006221	0.002497	0.002544	5	0.011362	0.001079
userId	0.001802	0.001256	0.016339	5	0.004388	-0.000785
price	0.000257	0.000749	0.242599	5	0.001800	-0.001285
totalPrice	-0.000815	0.003244	0.697902	5	0.005864	-0.007494
purchaseNum	-0.001673	0.000465	0.999352	5	-0.000716	-0.002631

面白い特徴としては、enemies4、enemies5、enemies1 などの数値が特徴量重要度の上位に来ている点です。enemies4 は 4 という名前が付けられた種類の敵をプレイヤーが倒した数を表します。
さらなる分析が必要ですが、例えば enemies4 を多く倒したユーザーが継続しているのに対して、 enemies4 を倒せていないユーザーが多く離脱してしまっているのだとすると、enemies4 の攻略がゲームバランス的に難しいのかもしれません。enemies4 の難易度を下げることによってユーザーの継続率が高まるかは試してみる価値はありそうです。

また、もう一つ面白い特徴としては、totalPrice や purchaseNum のスコアがマイナスになっている点です。totalPrice や purchaseNum はユーザーのゲーム内アイテムの課金額と購入数を表していますが、モデルの性能に寄与していないどころかマイナス要因になっていることがわかります。

ゲームの継続性を予測する際にアイテムの課金額と購入数が意外にも関連していないことも見えてきます。課金しているユーザーだからといってゲームの継続性を予測できないのだとすると、もしかしたらアイテムを購入してもゲームの継続に結びつかないゲームシナリオになっているのかもしれません。



まとめ

ここまで、とあるゲームログを参考にして、将来の休眠ユーザーとアクティブユーザーを予測する方法を紹介しました。ここに書いた内容はほんの入り口に過ぎませんが、少しでもユーザー離脱予測の敷居が下がれば、本記事が皆様のお役に立てたかと思います。

なお、ここで使った SageMaker Studio は停止しない限り課金が発生してしまうので、作業が終わりましたら、こちら を参考に Studio ノートブックを停止しておきましょう。

(補足) 今回用いたデータの説明

ユーザーのゲームログ

データベースには時系列ベースのユーザー行動履歴が保存されており、それぞれ記録の目的に応じて gameplay テーブルと sales テーブルと login テーブルに分かれて保存されています。

gameplay : ダンジョンでのゲームのプレイ履歴情報

カラム名	説明
id	ゲームごとの一意の UUID
startTime	開始日時
dungeonId	ダンジョン ID

userId	ユーザー ID
result	プレイ結果（Clear or Fail）
playTime	プレイ時間（分）
progress	プレイ進捗状況（100 % 中どこまでクリアできたか）
friendId	フレンドのユーザー ID
userCharacterID	ユーザーの利用キャラクター ID
userCharacterName	ユーザーの利用キャラクター名
userWeaponId	ユーザーの利用武器 ID
userWeaponName	ユーザーの利用武器名
userSkinId	ユーザーの利用スキン ID
userSkinName	ユーザーの利用スキン名
enemies1	倒した敵 1 の数
enemies2	倒した敵 2 の数
enemies3	倒した敵 3 の数
enemies4	倒した敵 4 の数
enemies5	倒した敵 5 の数
gainExp	獲得経験値

sales : ユーザーが購入したアイテムの売上履歴情報

カラム名	説明
id	売上 ID
purchasedTime	購入日時
store	購入ストア

userId	ユーザー ID
itemId	アイテム ID
itemName	購入アイテム名
price	商品単価
purchaseNum	購入個数
totalPrice	合計金額

login : ゲームユーザーのログイン履歴情報

カラム名	説明
userId	ユーザー ID
loginTime	ログイン日時

ユーザーのサマリー情報

各ユーザーの最新のサマリーデータがデータベース内の user テーブルに保存されています。

user : ゲームユーザーの情報

カラム名	説明
id	ユーザー ID
userName	ユーザー名
totalTime	総プレイ時間（分）
releasedDungeonId	開放済みダンジョン ID
level	レベル
clearTotalNum	ダンジョンクリア回数
coin	所持コイン（ゲーム内通貨）

totalExp	総獲得経験値
friendsNum	フレンド数
startTime	ゲーム開始日時
lastLoginTime	最終ログイン日時