[redacted] Case Study

Alessio Miolla[1]

[1]Ph.D. candidate in Brain, Mind and Computer Science

June 10, 2021

1 Introduction

1.1 Case

World of Tanks is a massive multi-player online game. In the game, a player participates in virtual tank battles. All tanks are divided into 10 tiers (ranging from 1 to 10). When a player begins his journey in World of Tanks, only tier-1 tanks are available to him/her. To play a higher-tier tank, the player must unlock it first and then buy it. The unlock and purchase of tanks are made by virtual currency which is disseminated to the players based on battle results and players' individual performance. The more the player plays, and the more efficient he/she is, the more resources he/she receives after every battle. Purchase of a tier-10 tank is one of the key goals in the World of Tanks game. Players who own tier-10 tanks are the most engaged part of game audience.

1.2 Problem description

User Acquisition team needs a model which can help them to evaluate new players. Namely, the team asks you to create a model which is based on the first 30 days of players' history and makes a prediction if a given player will purchase a tier-10 tank in the future. To solve the problem, you should: 1. Develop a model 2. Evaluate the quality of the developed model 3. Provide a brief discussion on the data available, the performance of the model, and the possible next steps, assuming you were to continue improving it

2 Import

[4]:
```
import pandas as pd
import os
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from numpy.core.fromnumeric import mean, std
import pandas as pd
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.metrics import confusion_matrix
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, GroupKFold
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score, make_scorer
from sklearn.metrics import roc_auc_score
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.ensemble import VotingClassifier
from sklearn.svm import OneClassSVM
from sklearn import svm
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn.preprocessing import MinMaxScaler
import imblearn
from imblearn.over_sampling import SMOTE
from collections import Counter
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import roc_curve, roc_auc_score, precision_recall_curve,␣
 ↪precision_score, f1_score, recall_score, confusion_matrix, accuracy_score
```

3 Load Data

[5]:
```
data_dir = r'C:\Users\alessio\Desktop\wargaming_project\data'
file_name = 'battles.csv'
file_target = 'players.csv'
file_name_feather = 'battles.feather'
```

[4]:
```
#df.to_feather(os.path.join(data_dir, 'battles.feather'))
```

[6]:
```
#df = pd.read_csv(os.path.join(data_dir, file_name))
df = pd.read_feather(os.path.join(data_dir, file_name_feather))
df.head()
```

[6]:

	player_id	battle_num	vehicle_id	vehicle_level	team_id	battle_level	\
0	1	1	69	1	2	1	
1	1	2	13	1	1	1	
2	1	3	14	1	1	1	
3	1	4	81	2	2	2	
4	1	5	81	2	2	3	

```
   battle_type_id  winner_team_id  shots  hits  pierced  damage_dealt  \
0               1               1      1     1        1            38
1               1               1      2     2        2            58
2               1               2      2     1        1            15
3               1               1      7     5        4            91
4               1               2      8     6        5           252

   kills_made  shots_received  pierced_received  damage_received  is_survived
0           0               3                 3              110            0
1           0              17                11              105            0
2           1               5                 5              105            0
3           2              21                17              120            0
4           1               2                 2              120            0
```

4 Explorative data analysis

[75]:
```
#Shape of the df containing the original metrics
df.shape
```

[75]:
```
(32604038, 17)
```

[76]:
```
#checking miss values
df.isnull().sum() #no miss values detected
```

[76]:
```
player_id           0
battle_num          0
vehicle_id          0
vehicle_level       0
team_id             0
battle_level        0
battle_type_id      0
winner_team_id      0
shots               0
hits                0
pierced             0
damage_dealt        0
kills_made          0
shots_received      0
pierced_received    0
damage_received     0
is_survived         0
dtype: int64
```

[77]:
```
#checking IDs unique for each player
df['player_id'].nunique() #all the players has an unique ID
```

[77]: 96000

[15]:
```
#summary of each numerical attribute
df.describe()
```

[15]:

	player_id	battle_num	vehicle_id	vehicle_level	team_id
count	3.260404e+07	3.260404e+07	3.260404e+07	3.260404e+07	3.260404e+07
mean	4.787598e+04	4.299499e+02	1.057311e+02	3.277230e+00	1.500089e+00
std	2.762478e+04	4.461305e+02	8.222339e+01	1.552077e+00	5.000000e-01
min	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
25%	2.390100e+04	1.090000e+02	3.400000e+01	2.000000e+00	1.000000e+00
50%	4.766400e+04	2.890000e+02	7.600000e+01	3.000000e+00	2.000000e+00
75%	7.152100e+04	6.030000e+02	1.620000e+02	4.000000e+00	2.000000e+00
max	9.600000e+04	6.027000e+03	3.180000e+02	1.000000e+01	2.000000e+00

	battle_level	battle_type_id	winner_team_id	shots
count	3.260404e+07	3.260404e+07	3.260404e+07	3.260404e+07
mean	3.989793e+00	1.048217e+00	1.463420e+00	1.092118e+01
std	2.036252e+00	4.673202e-01	5.307613e-01	2.060828e+01
min	0.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00
25%	2.000000e+00	1.000000e+00	1.000000e+00	2.000000e+00
50%	4.000000e+00	1.000000e+00	1.000000e+00	5.000000e+00
75%	5.000000e+00	1.000000e+00	2.000000e+00	1.100000e+01
max	1.200000e+01	6.000000e+00	2.000000e+00	7.400000e+02

	hits	pierced	damage_dealt	kills_made	shots_received
count	3.260404e+07	3.260404e+07	3.260404e+07	3.260404e+07	3.260404e+07
mean	4.166025e+00	2.566584e+00	1.421129e+02	4.131252e-01	5.852370e+00
std	7.732037e+00	4.209242e+00	2.540897e+02	8.142049e-01	8.679900e+00
min	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	2.000000e+00
50%	2.000000e+00	1.000000e+00	5.500000e+01	0.000000e+00	4.000000e+00
75%	5.000000e+00	3.000000e+00	1.760000e+02	1.000000e+00	6.000000e+00
max	6.240000e+02	2.230000e+02	2.060400e+04	1.500000e+01	7.410000e+02

	pierced_received	damage_received	is_survived
count	3.260404e+07	3.260404e+07	3.260404e+07
mean	3.982253e+00	2.552924e+02	2.068069e-01
std	3.864928e+00	2.350070e+02	4.050158e-01
min	0.000000e+00	0.000000e+00	0.000000e+00
25%	2.000000e+00	1.200000e+02	0.000000e+00
50%	3.000000e+00	2.000000e+02	0.000000e+00
75%	5.000000e+00	3.400000e+02	0.000000e+00
max	1.190000e+02	3.000000e+03	1.000000e+00

[86]:
```
#finding number unique types for the following metrics
print('vehicle_id: ', '\n', df['vehicle_id'].value_counts())
```

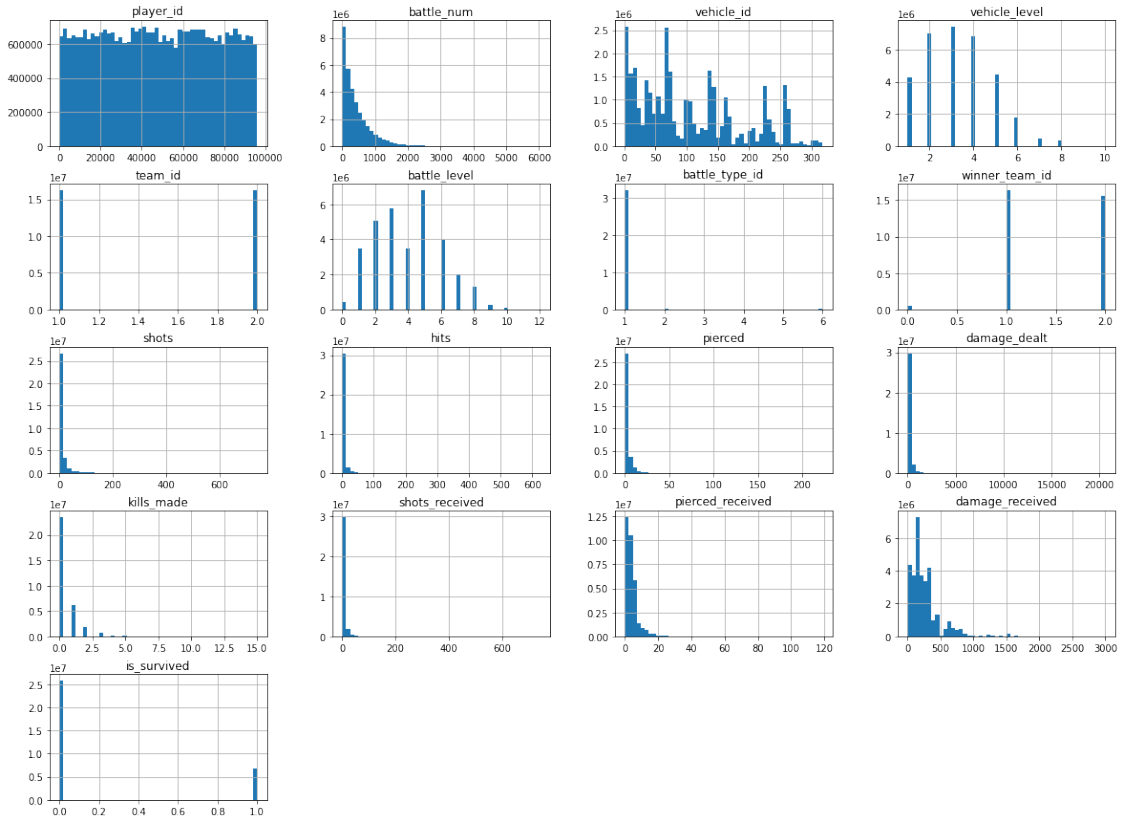
```
print('vehicle_level', '\n', df['vehicle_level'].value_counts())
print('battle_type_id', '\n', df['battle_type_id'].value_counts())
print('battle_level', '\n', df['battle_level'].value_counts())
```

```
vehicle_id:
 34     1295866
224     1252644
68      1195433
13      1029853
1        984063
         ...
227           3
127           1
43            1
173           1
273           1
Name: vehicle_id, Length: 317, dtype: int64
vehicle_level
 3      7448907
2      6987582
4      6845902
5      4452116
1      4303418
6      1761702
7       442572
8       346040
9        13090
10        2709
Name: vehicle_level, dtype: int64
battle_type_id
 1     32172591
6       278252
2       126874
3        25662
5          626
4           33
Name: battle_type_id, dtype: int64
battle_level
 5      6822531
3      5747018
2      5084242
6      3967533
1      3469253
4      3460736
7      1999213
8      1279801
0       429122
```

```
9       250707
10       79812
11       12454
12        1616
Name: battle_level, dtype: int64
```

4.1 Frequency_distribution

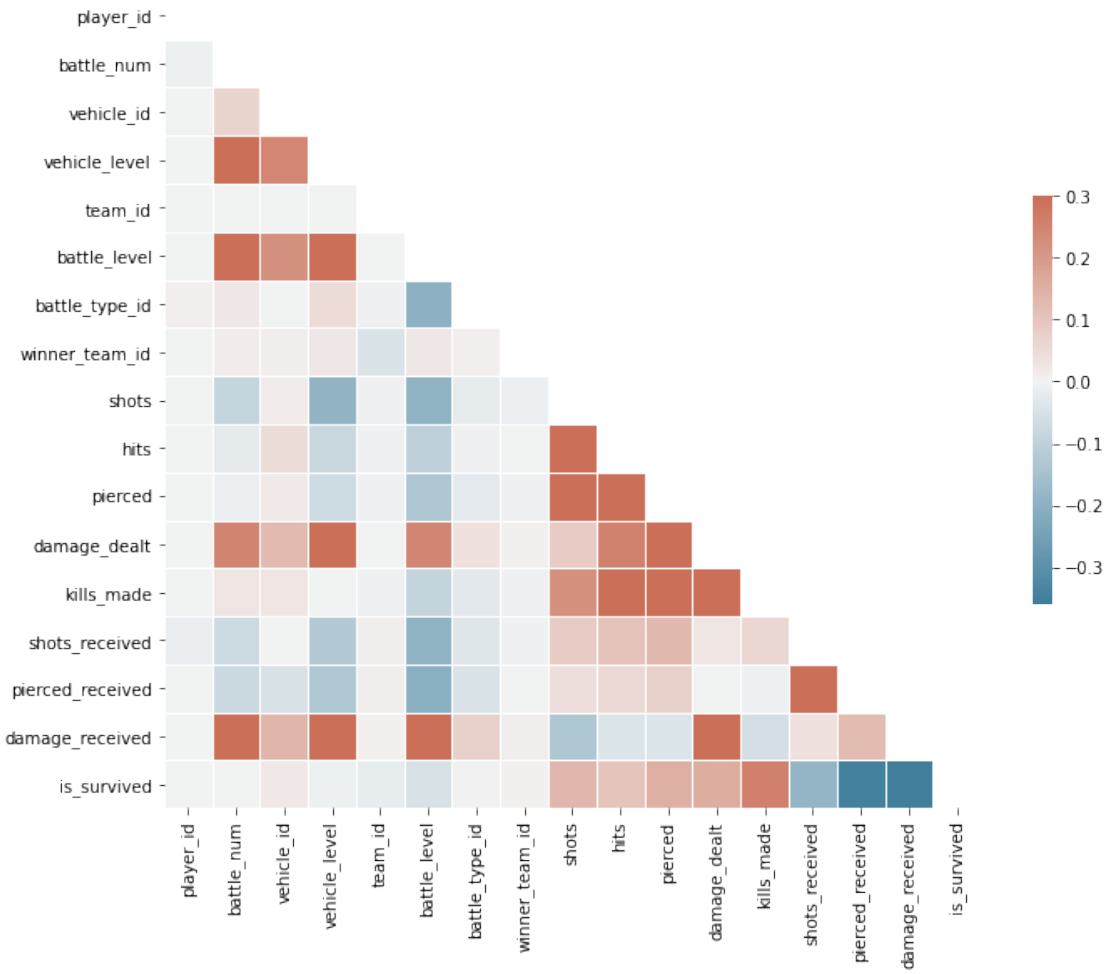
[45]:
```
#plot distribution of the metrics
df.hist(bins=50, figsize=(20, 15))
plt.show() #skewed distribution of the metrics
```



[6]:
```
#check potential correlation among the metrics

corr = df.corr()
# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))
```

```
# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

[6]: <AxesSubplot:>



Main considerations about the metrics correlation - Overall, the confusion matrix indicates a weak correlation among the metrics that does not exceed the Pearson correlation of +-=.3 - As expected, pierced and damage received negatively correlate with survival metrics (i.e., is_survived). In other words, the ratio of hits received (both pierced and damage) is minor if the players survive during the game than players that do not survive. - Conversely, there is a weak positive correlation between kills made and survival metrics. Thus, the higher the number of the kills, greater the chances of survival. - The number of battles is positively correlated with damage received/delt, the level of battles, and the vehicle level. In other words, a higher number of battles conduct to

higher metrics in the mentioned variables. - As expected, higher battle levels yield more damage dealt/received due to the more substantial power of hits of the tanks used in the game. - Naturally, the number of shots is positively correlated with hits, pierced, and kills. - Interestingly, damage dealt and received are positively correlated. These results may indicate that inflict more damages induce more damages received and vice-versa. Probably, the number of damage dealt depends on the risk that players use to take during the games, causing a major exposure to the enemy team, thus leading to more damage received.

5 Feature_extraction

AIM → Creation of a df with 1 row for player, and as cols as the number of features

[95]:
```
def feat(features_descriptors, df):

    list_id_players = list(range(1,96000+1))

    features_lists = {} # keys=column names, values= list of actual values of␣
↪column
    for colname in features_descriptors:
        features_lists[colname] = []

    for player_id in list_id_players:
        df_player = df[df['player_id'] == player_id]
        for colname, descriptor in features_descriptors.items():
            features_lists[colname].append(descriptor(df_player))

    return pd.DataFrame(features_lists)
```

Features extraction legend: 1) ***n_games*** = number of games played by the player 2) ***is_survived*** = % survival player for all the games played 3) ***avg_battle_lvl*** = average battle level played by the player for all the games 4) ***undefended_received_percent*** = ratio of the shots not blocked by the player for all the games played 5) ***precision*** = precision in shooting; ratio between hits and total shoots 6) ***hits_avg*** = mean hits for all the games played by the player 7) ***hits_sd*** = standard deviation for all the games played by the player 8) ***kills_avg*** = mean kills for all the games played by the player 9) ***kills_sd*** = standard deviation kills for all the games played by the player 10) ***vehicle_lev_avg*** = mean of the level of tanks used by the player during the games 11) ***vehicle_lev_sd*** = standard deviation of the level of tanks used by the player during the games 12) ***sniper***= sniper level (level of accuracy in hits) –> ratio between hits and kills made 13) ***damage_ratio*** = mean of the ratio between damage inflicted and suffered by the player for all the games played 14) ***win_rate*** = % wins for all the games played by the player 15) ***versatility*** = number of battle types different from 1 (the most battle type used in World of Tanks) played by the player 16) ***damage_level*** = mean of the damage dealt in relation with the level of the tier for all the games 17) ***damage_level_received*** = mean of the damage received in relation to the level of the tier for all the games

[96]:
```
features = {
        "n_games" : lambda df : len(df),
```

```
        "is_survived" : lambda df : len(df[df['is_survived'] == 1]) / len(df),
        "avg_battle_lvl" : lambda df : mean(df['battle_level']),
        "undefended_received_percent" : lambda df : mean(
                [ zero_dev(pierced, total) for pierced, total in␣
↪zip(df['pierced_received'], df['shots_received']) ]
            ),
        "precision" : lambda df : mean(
                [ zero_dev(hits, total) for hits, total in zip(df['hits'],␣
↪df['shots']) ]
            ),
        "hits_avg" : lambda df : mean(df['hits']),
        "hits_sd" : lambda df : np.std(df['hits']),
        "kills_avg" : lambda df : mean(df['kills_made']),
        "kills_sd" : lambda df : np.std(df['kills_made']),
        "vehicle_lev_avg" : lambda df : mean(df['vehicle_level']),
        "vehicle_lev_sd" : lambda df : np.std(df['vehicle_level']),
        "sniper" : lambda df : mean(
                [zero_dev(kills, total) for kills, total in␣
↪zip(df['kills_made'], df['hits']) ]
            ),
        "damage_ratio" : lambda df : mean(
                [zero_dev(damage_r, damage_d) for damage_r, damage_d in␣
↪zip(df['damage_received'], df['damage_dealt'])]
            ),
        "win_rate" : lambda df : len(df.query('team_id == winner_team_id'))/
↪len(df),
        "versatility" : lambda df : len(df[df['battle_type_id'] != 1]) / len(df),
        "damage_level" : lambda df : mean([zero_dev(damage_d, tier_lev) for␣
↪damage_d, tier_lev in zip(df['vehicle_level'], df['damage_dealt'])]),
        "damage_level_received" : lambda df : mean([zero_dev(damage_r, tier_lev)␣
↪for damage_r, tier_lev in zip(df['vehicle_level'], df['damage_received'])])

    }


df_ml_features = feat(features, df)
df_ml_features.to_csv('df_new_features.csv', index = False, header=True)
```

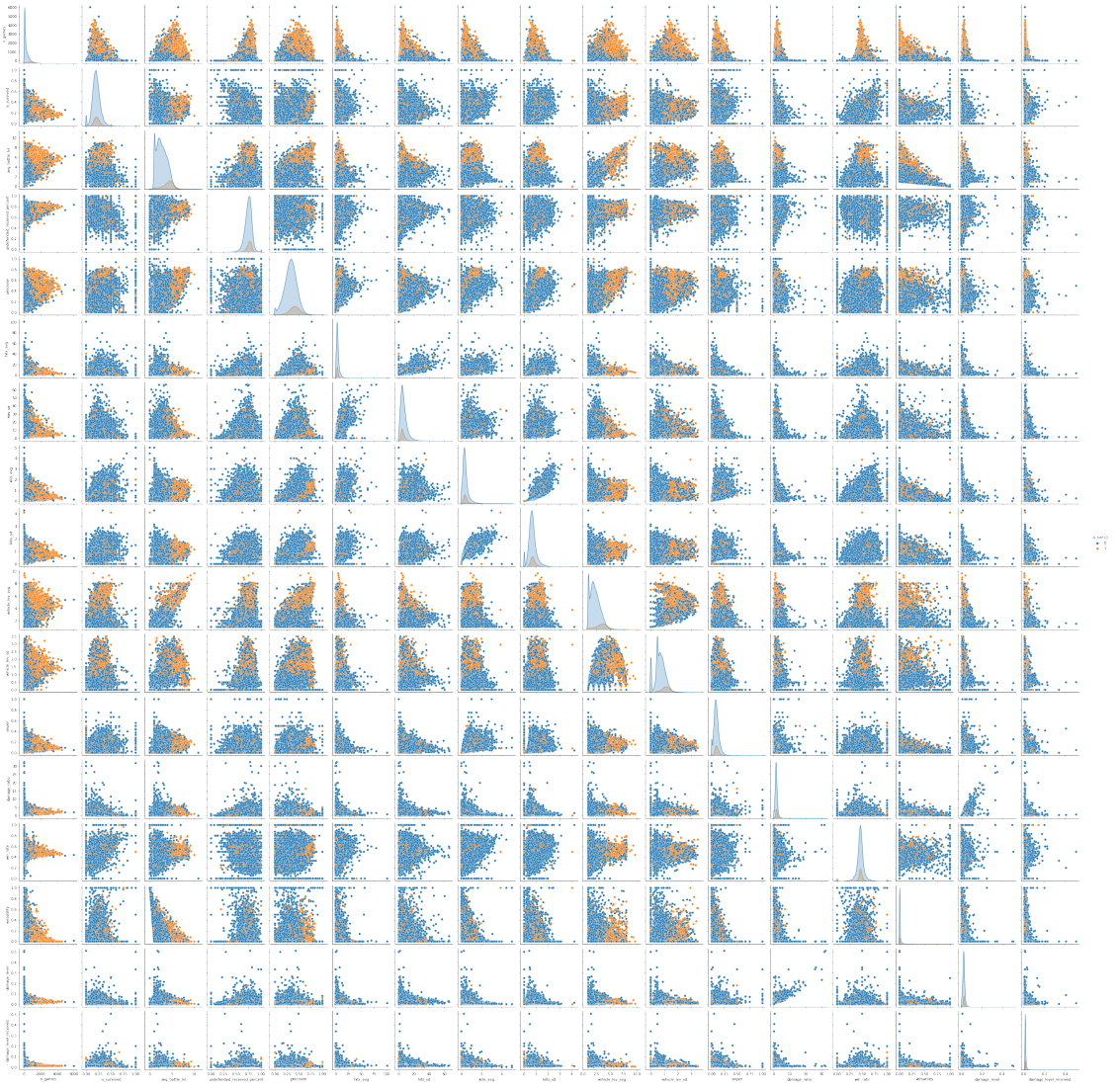
[8]:
```
#new file containing features
file_features = 'df_new_features.csv'
```

[9]:
```
#concatenate df containing features with the target variable included in players.
↪csv
target = pd.read_csv(os.path.join(data_dir, file_target))
df = pd.read_csv(os.path.join(data_dir, file_features))
df['is_tier10'] = target['is_tier10']
```

```
#print(df.head())
#df.to_csv('df_ML.csv', index = False, header=True) #df complete for the ML␣
↪analysis
```

[9]:
```
#plot of the distribution between features related to the target variable (e.g.,␣
↪0-1)
sns.pairplot(df, hue="is_tier10")
```

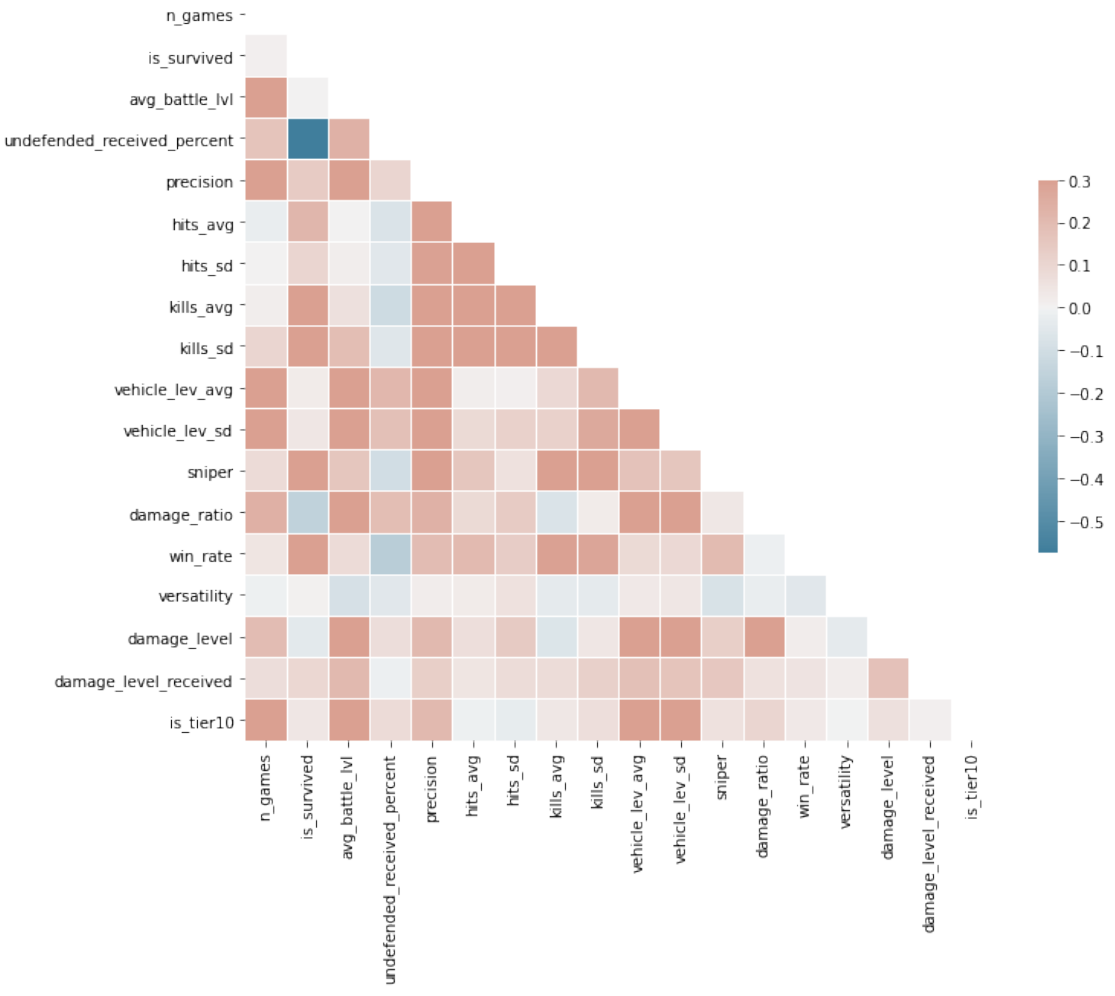
[9]: <seaborn.axisgrid.PairGrid at 0x1fd500bcd60>



[10]:
```
#correlation matrix of the features (to identify potential correlations among␣
↪them)
```

```
corr = df.corr()
# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

[10]: <AxesSubplot:>



Main Considerations about the confusion matrix of the features - Overall, as in the previous confusion matrix, there is a weak correlation between the features except for a moderate negative correlation between the ratio of survival and the ratio of damage received on the total shots. In other words, as also the logic suggests, the percentages of survival increase with decreasing of the undefended damages during the games. - The survival rate is positively correlated with the win rate, the sniper level of the player, as well as the average and SD of the kills made. - The kills made are positively correlated with the level of precision and hits inflicted. - As expected, the number of games played positively correlates with increases in the average battle level, precision, and the target variable (i.e., the player will purchase a tier-10). - The target variable is also positively correlated with the average battle and the vehicle level (as well as the SD of the vehicle level) of the games.

6 Modeling

The machine learning models used in the current section have been chosen according to the preliminary results run by using the library Pycaret on Google Colab, which allowed me to compare the models that fit best at once (for additional information, please see in attach the file "pycaret_wg"). F1 score was adopted as the main criteria since the unbalanced between the two target variables. Indeed, the highly skewed class distribution (87.5% of class0 versus 12.5% of class1) can lead the classifier to get a low misclassification rate simply by choosing the majority class (i.e., class 0). Consequently, I decided to get the classifiers with high F1 scores in both classes instead of other criteria such as accuracy, precision, and recall of the majority class0.

6.1 Split training/set

[10]:
```
#split set of features (X) and target (y)
X = df.loc[:, df.columns != 'is_tier10']
y = df['is_tier10']

#split training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

6.2 LDA

[11]:
```
LDA = LinearDiscriminantAnalysis()

#Parameter tuning with GridSearchCV
parameters_2 = {
    'solver': ['svd', 'lsqr', 'eigen'],

    'store_covariance': [True, False],
                    }

grid_search_lda_B = GridSearchCV(
    estimator=LDA,
```

```
    param_grid=parameters_2,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#training
clf = grid_search_lda_B.fit(X_train, y_train)
clf.fit(X_train, y_train)

#test
test_results = []
y_test_pred = clf.predict(X_test)

#performances
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_test_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_test_pred)
print('accuracy: ', accuracy)
print('f1: ', f1_score(y_test, y_test_pred, average="macro"))
print(confusion_matrix(y_test, y_test_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_test_pred))
```

```
accuracy:  0.8832291666666666
f1:  0.6843698307112941
[[16099   713]
 [ 1529   859]]
auc:  [0.6586525298392989]
              precision    recall  f1-score   support

           0       0.91      0.96      0.93     16812
           1       0.55      0.36      0.43      2388

    accuracy                           0.88     19200
   macro avg       0.73      0.66      0.68     19200
weighted avg       0.87      0.88      0.87     19200
```

6.2.1 The current model got a poor performance, as demonstrated by the low level of AUC and F1 score. These results can be explained by the low percentages obtained in the prediction of the minority class 1.

6.3 QDA

[168]:
```
QDA = QuadraticDiscriminantAnalysis()

#Parameter tuning with GridSearchCV
parameters_3 = {
    'reg_param': [0.00001, 0.0001, 0.001,0.01, 0.1],
    'store_covariance': [True, False],
    'tol': [0.0001, 0.001,0.01, 0.1]}

grid_search_qda = GridSearchCV(
    estimator= QDA,
    param_grid=parameters_3,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#training
clf = grid_search_qda.fit(X_train, y_train)

#test
y_test_pred = clf.predict(X_test)

#performance
test_results = []
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_test_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_test_pred)
print('accuracy: ', accuracy)
print('f1: ', f1_score(y_test, y_test_pred, average="macro"))
print(confusion_matrix(y_test, y_test_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_test_pred))
```

```
accuracy:  0.8434895833333333
f1:  0.6919263807211015
[[14831  1981]
 [ 1024  1364]]
auc:  [0.7266783895685901]
              precision    recall  f1-score   support
```

```
              0       0.94      0.88      0.91     16812
              1       0.41      0.57      0.48      2388

    accuracy                            0.84     19200
   macro avg       0.67      0.73      0.69     19200
weighted avg       0.87      0.84      0.85     19200
```

6.3.1 The QDA obtained better performance compared to the LDA. The classification of class 1 improved both in precision and recall, increasing the AUC and the F1 score.

7 Oversample

The imbalance between the two target classes led to having poor performance in the minority class. In the current section, the oversampling of the minority class was used to addressing this problem. The Synthetic Minority Oversampling Technique (SMOTE) is a type of data augmentation that duplicates examples in the minority class, although these examples don't add any new information to the model. Doing so, I tested if the previous low predictions on class1 are not due to the relatively little amount of information given in the training set.

[169]:
```
# summarize class distribution
counter = Counter(y)
print(counter)

#split_training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

#compromise oversampling
over = SMOTE(sampling_strategy=0.5)
X_train, y_train = over.fit_resample(X_train, y_train)

# summarize the new class distribution
counter = Counter(y_train)
print(counter)
```

```
Counter({0: 84000, 1: 12000})
Counter({0: 67188, 1: 33594})
```

7.1 LDA

[170]:
```
LDA = LinearDiscriminantAnalysis()

#Parameter tuning with GridSearchCV
parameters_2 = {
    'solver': ['svd', 'lsqr', 'eigen'],

    'store_covariance': [True, False],
```

```
                  }

grid_search_lda_B = GridSearchCV(
    estimator=LDA,
    param_grid=parameters_2,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#training
clf = grid_search_lda_B.fit(X_train, y_train)
clf.fit(X_train, y_train)

#test
test_results = []
y_test_pred = clf.predict(X_test)

#performance
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_test_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_test_pred)
print('accuracy: ', accuracy)
print('f1: ', f1_score(y_test, y_test_pred, average="macro"))
print(confusion_matrix(y_test, y_test_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_test_pred))
```

```
accuracy:  0.8603125
f1:  0.7098608305157098
[[15172  1640]
 [ 1042  1346]]
auc:  [0.7330511108959022]
              precision    recall  f1-score   support

           0       0.94      0.90      0.92     16812
           1       0.45      0.56      0.50      2388

    accuracy                           0.86     19200
   macro avg       0.69      0.73      0.71     19200
weighted avg       0.88      0.86      0.87     19200
```

7.1.1 The oversampling of the LDA outperforms the LDA with unbalanced dataset. In addition, it performs slighty better than QDA, resulting in better AUC and F1 score.

7.2 QDA

[171]:
```
QDA = QuadraticDiscriminantAnalysis()

#Parameter tuning with GridSearchCV
parameters_3 = {
    'reg_param': [0.00001, 0.0001, 0.001,0.01, 0.1],
    'store_covariance': [True, False],
    'tol': [0.0001, 0.001,0.01, 0.1]}

grid_search_qda = GridSearchCV(
    estimator= QDA,
    param_grid=parameters_3,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#training
clf = grid_search_qda.fit(X_train, y_train)

#test
y_test_pred = clf.predict(X_test)

#performance
test_results = []
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_test_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_test_pred)
print('accuracy: ', accuracy)
print('f1: ', f1_score(y_test, y_test_pred, average="macro"))
print(confusion_matrix(y_test, y_test_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_test_pred))
```

```
accuracy:  0.8125
f1:  0.6743308908589902
[[14053  2759]
 [  841  1547]]
auc:  [0.7418567378888254]
              precision    recall  f1-score   support

           0       0.94      0.84      0.89     16812
```

```
           1       0.36      0.65      0.46      2388

    accuracy                           0.81     19200
   macro avg       0.65      0.74      0.67     19200
weighted avg       0.87      0.81      0.83     19200
```

7.2.1 The oversampling gets worse the prediction of the QDA.

8 PCA approach

Considering the weak correlation among the features, a PCA was used combined to the models used before to remove noise in data by reducing the number of features to a few of principal components.

[149]:
```
# summarize class distribution
counter = Counter(y)
print(counter)

#split_training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

#compromise oversampling
over = SMOTE(sampling_strategy=0.5)
X_train, y_train = over.fit_resample(X_train, y_train)

# summarize the new class distribution
counter = Counter(y_train)
print(counter)
```

```
Counter({0: 84000, 1: 12000})
Counter({0: 67188, 1: 33594})
```

8.1 LDA

[42]:
```
#PCA transform
pca = PCA()
pca = pca.fit(X_train)
X_t_train = pca.transform(X_train)
X_t_test = pca.transform(X_test)

#model and tuning
clf = LinearDiscriminantAnalysis()
parameters_3 = {
    'reg_param': [0.00001, 0.0001, 0.001,0.01, 0.1],
    'store_covariance': [True, False],
    'tol': [0.0001, 0.001,0.01, 0.1]}
```

```
grid_search_qda = GridSearchCV(
    estimator=clf,
    param_grid=parameters_3,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#training
clf.fit(X_t_train, y_train)

#test
test_results = []
y_test_pred = clf.predict(X_test)

#performance
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_test_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_test_pred)
print('accuracy: ', accuracy)
print('f1: ', f1_score(y_test, y_test_pred, average="macro"))
print(confusion_matrix(y_test, y_test_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_test_pred))
```

```
accuracy:  0.8686979166666666
f1:  0.670364494403108
[[15786  1026]
 [ 1495   893]]
auc:  [0.656462630784185]
              precision    recall  f1-score   support

           0       0.91      0.94      0.93     16812
           1       0.47      0.37      0.41      2388

    accuracy                           0.87     19200
   macro avg       0.69      0.66      0.67     19200
weighted avg       0.86      0.87      0.86     19200
```

8.2 QDA

[43]:
```
#PCA transform
pca = PCA()
pca = pca.fit(X_train)
X_t_train = pca.transform(X_train)
X_t_test = pca.transform(X_test)

#model and tuning
clf = QuadraticDiscriminantAnalysis()
parameters_3 = {
    'reg_param': [0.00001, 0.0001, 0.001,0.01, 0.1],
    'store_covariance': [True, False],
    'tol': [0.0001, 0.001,0.01, 0.1]}

grid_search_qda = GridSearchCV(
    estimator=clf,
    param_grid=parameters_3,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#train
clf.fit(X_t_train, y_train)

#test
test_results = []
y_test_pred = clf.predict(X_test)

#performance
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,␣
 ↪y_test_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_test_pred)
print('accuracy: ', accuracy)
print('f1: ', f1_score(y_test, y_test_pred, average="macro"))
print(confusion_matrix(y_test, y_test_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_test_pred))
```

```
accuracy:  0.8734895833333334
f1:  0.467054966161782
[[16769    43]
 [ 2386     2]]
auc:  [0.49913991202742236]
              precision    recall  f1-score   support
```

```
              0       0.88      1.00      0.93     16812
              1       0.04      0.00      0.00      2388

    accuracy                            0.87     19200
   macro avg       0.46      0.50      0.47     19200
weighted avg       0.77      0.87      0.82     19200
```

8.2.1 The PCA clearly affects the minority class predictions. Balancing the class distribution (oversampling) was revealed to be more significant to increase the classifier's performance than PCA preprocessing like also suggested by T. Maruthi Padmaja, et al., 2014 [1]

9 Performances

In the current section the best model obtained in the section "Modeling" was used, namely the LDA with oversampling.

[15]:
```
# summarize class distribution
counter = Counter(y)
print(counter)

#split_training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

#compromise oversampling
over = SMOTE(sampling_strategy=0.5)
X_train, y_train = over.fit_resample(X_train, y_train)

# summarize the new class distribution
counter = Counter(y_train)
print(counter)
```

```
Counter({0: 84000, 1: 12000})
Counter({0: 67188, 1: 33594})
```

[16]:
```
LDA = LinearDiscriminantAnalysis()

#Parameter tuning with GridSearchCV
parameters_2 = {
    'solver': ['svd', 'lsqr', 'eigen'],

    'store_covariance': [True, False],
                    }
```

[1] https://dl.acm.org/doi/10.1504/IJKESDP.2014.064265

```
grid_search_lda_B = GridSearchCV(
    estimator=LDA,
    param_grid=parameters_2,
    scoring = 'f1',
    n_jobs = -1,
    cv = 5
)

#training
clf = grid_search_lda_B.fit(X_train, y_train)
clf.fit(X_train, y_train)

y_train_probs = clf.predict_proba(X_train)
y_training_pred = clf.predict(X_train)

#test
test_results = []
y_test_pred = clf.predict(X_test)
y_test_probs = clf.predict_proba(X_test)

#set threshold evaluation
threshold = np.arange(0.1, 0.6, 0.01)
for x in threshold:
    y_pred = (clf.predict_proba(X_test)[:,1] >= x).astype('int')
    print('test', "\n", x, "\n", classification_report(y_test, y_pred))
```

[176]:
```
#test with best threshold
test_results = []
y_pred = (clf.predict_proba(X_test)[:,1] >= 0.5).astype('int')

#performance
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
test_results.append(roc_auc)
accuracy = accuracy_score(y_test, y_pred)
print(confusion_matrix(y_test, y_pred))
print('auc: ', test_results)
print(classification_report(y_test, y_pred))
```

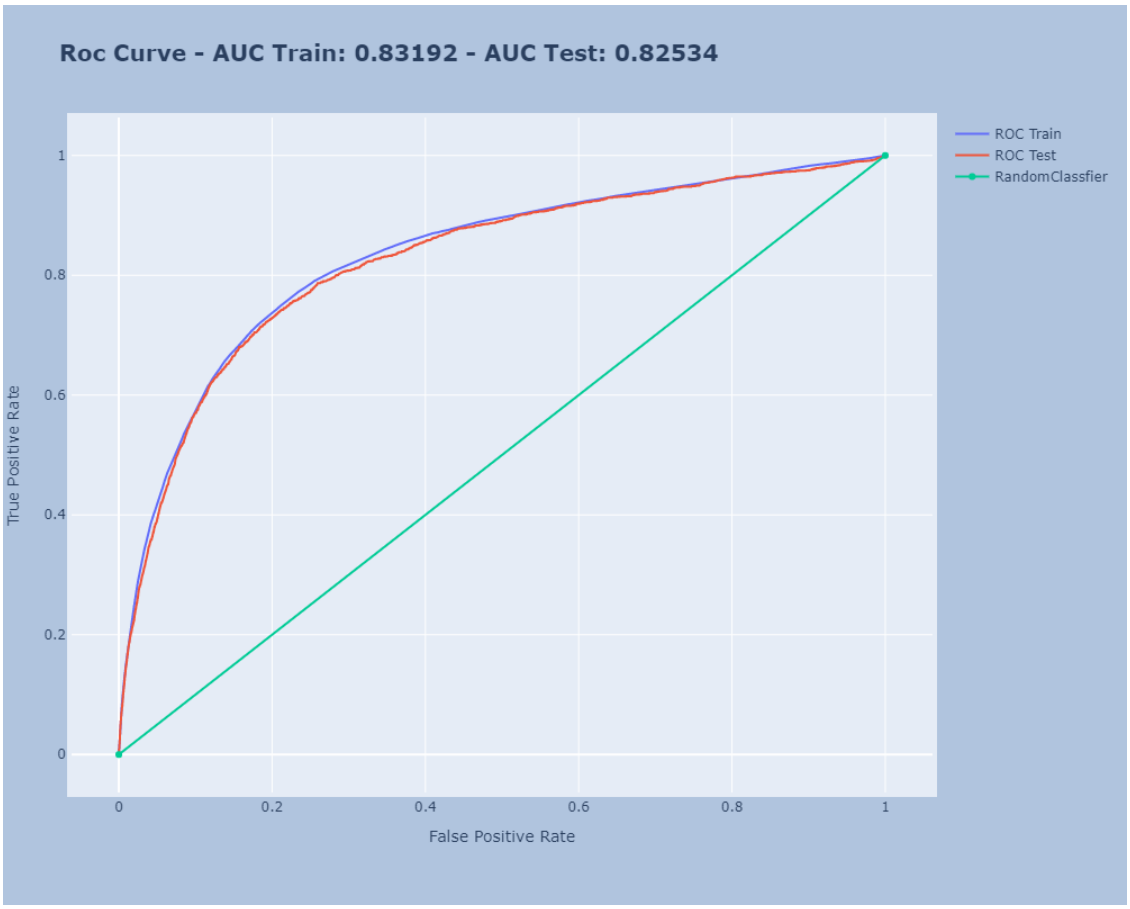
```
[[15172  1640]
 [ 1042  1346]]
auc:  [0.7330511108959022]
              precision    recall  f1-score   support

           0       0.94      0.90      0.92     16812
           1       0.45      0.56      0.50      2388
```

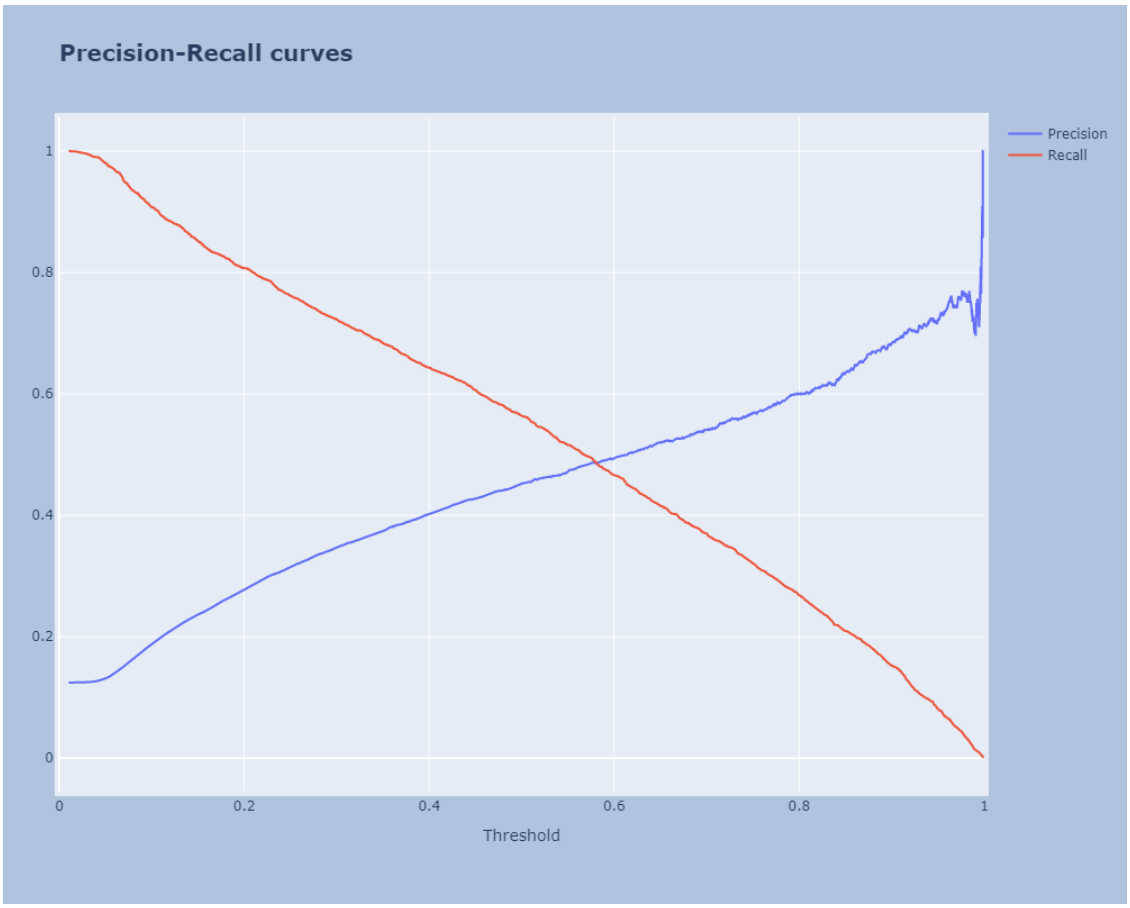
```
    accuracy                           0.86     19200
   macro avg       0.69      0.73      0.71     19200
weighted avg       0.88      0.86      0.87     19200
```

9.1 PLOTS

Please see the section "Appendix" below for the code of the plots

[177]:
```
plot_roc_auc(y_train_probs[:,1], y_test_probs[:,1], y_train, y_test,␣
↪show_AUC_plot=True, show_PREC_REC_plot=True, title_AUC=None,␣
↪title_PREC_REC=None, html_dir=data_dir, png_dir=None)
```





10 Save model

[19]:
```
import pickle
filename = 'finalized_model.sav'
pickle.dump(y_pred, open(filename, 'wb'))
```

11 Discussion

The final model was revealed to be the best model applied in the analysis, as demonstrated by the gold standard adopted for unbalanced set, such as AUC and F1 score. The AUC provides an aggregate measure of performance across all possible classification thresholds. The value of 0.82 suggests a good compromise between the True and False positive rates, with a global accuracy of 86%. In addition, the F1 score of 0.71 indicates low false positives and low false negatives. However, it is worth mentioning that the models still has poor performance on the minority class (i.e., people who purchase a 10 level tier), as suggested by the relatively low level of precision and recall. Nevertheless, in the deployment phase of the model, it is still possible to adjust the threshold of the model according to the use case and business needs. In other words, if we are interested in maximizing the level of precision of the minority class, we would increase the level of the threshold of the model. In a real-world scenario, for example, if we want to be totally sure about the player that will purchase (because the company is interested in contacting only the right

players), we will adopt a higher level of precision of the minority class, sacrificing its recall and global F1 score. To conclude, the final implementation of the model will be driven by the business and company needs.

12 Future studies

- ***Extract new metrics*** → (e.g., days played, tracking the duration of the games, and so on) to allow more combinations in the features extraction and enable time series analysis in the players' performance.
- ***Extract new features*** → the players' ability can be detected by using new or additional features that have been neglected in this report. New insights about the metrics should be taken into account for future studies
- ***Long Short-Term Memory network models (LSTM)*** → are a type of recurrent neural network that are able to learn and remember over long sequences of input data. The games played during the days can be used as time-series analyses to classify if a given player will purchase a tier-10 tank in the future.
- ***Removing outliers and implement anomaly detection for bots*** → Bot is a type of artificial intelligence that plays a video game in the place of a human. The presence of the bots may, in part, affect the model. For example, in the sample of data adopted in my analysis, a small sample of players played more than 3000 games (exceeding 6000 games in 30 days). Probably the unimaginable amount of games played had been performed by bots and not by humans. Future studies should combine prior bots detection to verify if the bots can affect the development of the models.
- ***Use different windows of interest in the players' prediction*** → The first thirty days of games were used as a fixed window of interest in the data analysis. Different windows of days could also be used in the classification (e.g., the first fifteen days as well as the first forty days) to provide more insights into the analysis and find new connections in the metrics.

13 Appendix

13.1 Comparison of the thresholds

```
test
 0.1
              precision    recall  f1-score   support

           0       0.97      0.44      0.60     16812
           1       0.19      0.91      0.31      2388

    accuracy                           0.50     19200
   macro avg       0.58      0.67      0.46     19200
weighted avg       0.87      0.50      0.57     19200

test
 0.11
              precision    recall  f1-score   support
```

```
           0       0.97      0.48      0.65     16812
           1       0.20      0.90      0.32      2388

    accuracy                           0.54     19200
   macro avg       0.58      0.69      0.49     19200
weighted avg       0.87      0.54      0.61     19200

test
 0.12
              precision    recall  f1-score   support

           0       0.97      0.52      0.68     16812
           1       0.21      0.88      0.34      2388

    accuracy                           0.57     19200
   macro avg       0.59      0.70      0.51     19200
weighted avg       0.87      0.57      0.64     19200

test
 0.13
              precision    recall  f1-score   support

           0       0.97      0.56      0.71     16812
           1       0.22      0.88      0.35      2388

    accuracy                           0.60     19200
   macro avg       0.59      0.72      0.53     19200
weighted avg       0.88      0.60      0.66     19200

test
 0.13999999999999999
              precision    recall  f1-score   support

           0       0.97      0.58      0.73     16812
           1       0.23      0.87      0.36      2388

    accuracy                           0.62     19200
   macro avg       0.60      0.72      0.54     19200
weighted avg       0.88      0.62      0.68     19200

test
 0.14999999999999997
              precision    recall  f1-score   support

           0       0.97      0.61      0.74     16812
           1       0.24      0.85      0.37      2388
```

```
    accuracy                           0.64     19200
   macro avg       0.60      0.73      0.56     19200
weighted avg       0.88      0.64      0.70     19200

test
 0.15999999999999998
              precision    recall  f1-score   support

           0       0.97      0.63      0.76     16812
           1       0.24      0.84      0.38      2388

    accuracy                           0.65     19200
   macro avg       0.60      0.73      0.57     19200
weighted avg       0.88      0.65      0.71     19200

test
 0.16999999999999998
              precision    recall  f1-score   support

           0       0.96      0.65      0.78     16812
           1       0.25      0.83      0.39      2388

    accuracy                           0.67     19200
   macro avg       0.61      0.74      0.58     19200
weighted avg       0.88      0.67      0.73     19200

test
 0.17999999999999997
              precision    recall  f1-score   support

           0       0.96      0.67      0.79     16812
           1       0.26      0.83      0.40      2388

    accuracy                           0.69     19200
   macro avg       0.61      0.75      0.59     19200
weighted avg       0.88      0.69      0.74     19200

test
 0.18999999999999995
              precision    recall  f1-score   support

           0       0.96      0.68      0.80     16812
           1       0.27      0.82      0.40      2388

    accuracy                           0.70     19200
   macro avg       0.62      0.75      0.60     19200
weighted avg       0.88      0.70      0.75     19200
```

```
test
 0.19999999999999996
              precision    recall  f1-score   support

           0       0.96      0.70      0.81     16812
           1       0.28      0.81      0.41      2388

    accuracy                           0.71     19200
   macro avg       0.62      0.75      0.61     19200
weighted avg       0.88      0.71      0.76     19200

test
 0.20999999999999996
              precision    recall  f1-score   support

           0       0.96      0.71      0.82     16812
           1       0.28      0.80      0.42      2388

    accuracy                           0.72     19200
   macro avg       0.62      0.76      0.62     19200
weighted avg       0.88      0.72      0.77     19200

test
 0.21999999999999995
              precision    recall  f1-score   support

           0       0.96      0.73      0.83     16812
           1       0.29      0.79      0.43      2388

    accuracy                           0.74     19200
   macro avg       0.63      0.76      0.63     19200
weighted avg       0.88      0.74      0.78     19200

test
 0.22999999999999995
              precision    recall  f1-score   support

           0       0.96      0.74      0.84     16812
           1       0.30      0.79      0.44      2388

    accuracy                           0.75     19200
   macro avg       0.63      0.76      0.64     19200
weighted avg       0.88      0.75      0.79     19200

test
 0.23999999999999994
              precision    recall  f1-score   support
```

```
              0       0.96      0.75      0.84     16812
              1       0.31      0.77      0.44      2388

    accuracy                            0.75     19200
   macro avg       0.63      0.76      0.64     19200
weighted avg       0.88      0.75      0.79     19200

test
 0.24999999999999992
              precision    recall  f1-score   support

              0       0.96      0.76      0.85     16812
              1       0.31      0.76      0.44      2388

    accuracy                            0.76     19200
   macro avg       0.64      0.76      0.65     19200
weighted avg       0.88      0.76      0.80     19200

test
 0.2599999999999999
              precision    recall  f1-score   support

              0       0.96      0.77      0.86     16812
              1       0.32      0.76      0.45      2388

    accuracy                            0.77     19200
   macro avg       0.64      0.76      0.65     19200
weighted avg       0.88      0.77      0.80     19200

test
 0.2699999999999999
              precision    recall  f1-score   support

              0       0.96      0.78      0.86     16812
              1       0.33      0.75      0.46      2388

    accuracy                            0.78     19200
   macro avg       0.64      0.76      0.66     19200
weighted avg       0.88      0.78      0.81     19200

test
 0.2799999999999999
              precision    recall  f1-score   support

              0       0.96      0.79      0.87     16812
              1       0.33      0.74      0.46      2388

    accuracy                            0.78     19200
```

```
   macro avg       0.64      0.77      0.66     19200
weighted avg       0.88      0.78      0.81     19200

test
 0.2899999999999999
              precision    recall  f1-score   support

           0       0.95      0.80      0.87     16812
           1       0.34      0.73      0.46      2388

    accuracy                           0.79     19200
   macro avg       0.65      0.76      0.67     19200
weighted avg       0.88      0.79      0.82     19200

test
 0.29999999999999993
              precision    recall  f1-score   support

           0       0.95      0.81      0.87     16812
           1       0.35      0.72      0.47      2388

    accuracy                           0.80     19200
   macro avg       0.65      0.76      0.67     19200
weighted avg       0.88      0.80      0.82     19200

test
 0.30999999999999994
              precision    recall  f1-score   support

           0       0.95      0.81      0.88     16812
           1       0.35      0.72      0.47      2388

    accuracy                           0.80     19200
   macro avg       0.65      0.76      0.67     19200
weighted avg       0.88      0.80      0.83     19200

test
 0.3199999999999999
              precision    recall  f1-score   support

           0       0.95      0.82      0.88     16812
           1       0.36      0.71      0.47      2388

    accuracy                           0.80     19200
   macro avg       0.65      0.76      0.68     19200
weighted avg       0.88      0.80      0.83     19200

test
```

```
 0.32999999999999985
              precision    recall  f1-score   support

           0       0.95      0.82      0.88     16812
           1       0.36      0.70      0.48      2388

    accuracy                           0.81     19200
   macro avg       0.66      0.76      0.68     19200
weighted avg       0.88      0.81      0.83     19200

test
 0.33999999999999986
              precision    recall  f1-score   support

           0       0.95      0.83      0.89     16812
           1       0.37      0.69      0.48      2388

    accuracy                           0.81     19200
   macro avg       0.66      0.76      0.68     19200
weighted avg       0.88      0.81      0.84     19200

test
 0.34999999999999987
              precision    recall  f1-score   support

           0       0.95      0.84      0.89     16812
           1       0.37      0.69      0.48      2388

    accuracy                           0.82     19200
   macro avg       0.66      0.76      0.69     19200
weighted avg       0.88      0.82      0.84     19200

test
 0.3599999999999999
              precision    recall  f1-score   support

           0       0.95      0.84      0.89     16812
           1       0.38      0.68      0.49      2388

    accuracy                           0.82     19200
   macro avg       0.66      0.76      0.69     19200
weighted avg       0.88      0.82      0.84     19200

test
 0.3699999999999999
              precision    recall  f1-score   support

           0       0.95      0.85      0.90     16812
```

```
           1       0.39      0.67      0.49      2388

    accuracy                           0.83     19200
   macro avg       0.67      0.76      0.69     19200
weighted avg       0.88      0.83      0.84     19200

test
 0.3799999999999999
              precision    recall  f1-score   support

           0       0.95      0.85      0.90     16812
           1       0.39      0.66      0.49      2388

    accuracy                           0.83     19200
   macro avg       0.67      0.76      0.69     19200
weighted avg       0.88      0.83      0.85     19200

test
 0.3899999999999999
              precision    recall  f1-score   support

           0       0.95      0.86      0.90     16812
           1       0.40      0.65      0.49      2388

    accuracy                           0.83     19200
   macro avg       0.67      0.75      0.70     19200
weighted avg       0.88      0.83      0.85     19200

test
 0.3999999999999998
              precision    recall  f1-score   support

           0       0.94      0.86      0.90     16812
           1       0.40      0.64      0.49      2388

    accuracy                           0.84     19200
   macro avg       0.67      0.75      0.70     19200
weighted avg       0.88      0.84      0.85     19200

test
 0.4099999999999998
              precision    recall  f1-score   support

           0       0.94      0.87      0.90     16812
           1       0.41      0.64      0.50      2388

    accuracy                           0.84     19200
   macro avg       0.68      0.75      0.70     19200
```

```
weighted avg       0.88      0.84      0.85     19200

test
 0.4199999999999998
              precision    recall  f1-score   support

           0       0.94      0.87      0.91     16812
           1       0.41      0.63      0.50      2388

    accuracy                           0.84     19200
   macro avg       0.68      0.75      0.70     19200
weighted avg       0.88      0.84      0.86     19200

test
 0.4299999999999998
              precision    recall  f1-score   support

           0       0.94      0.88      0.91     16812
           1       0.42      0.63      0.50      2388

    accuracy                           0.84     19200
   macro avg       0.68      0.75      0.70     19200
weighted avg       0.88      0.84      0.86     19200

test
 0.43999999999999984
              precision    recall  f1-score   support

           0       0.94      0.88      0.91     16812
           1       0.42      0.62      0.50      2388

    accuracy                           0.85     19200
   macro avg       0.68      0.75      0.71     19200
weighted avg       0.88      0.85      0.86     19200

test
 0.44999999999999984
              precision    recall  f1-score   support

           0       0.94      0.88      0.91     16812
           1       0.43      0.61      0.50      2388

    accuracy                           0.85     19200
   macro avg       0.68      0.75      0.71     19200
weighted avg       0.88      0.85      0.86     19200

test
 0.45999999999999985
```

```
              precision    recall  f1-score   support

           0       0.94      0.89      0.91     16812
           1       0.43      0.60      0.50      2388

    accuracy                           0.85     19200
   macro avg       0.69      0.74      0.71     19200
weighted avg       0.88      0.85      0.86     19200

test
 0.46999999999999986
              precision    recall  f1-score   support

           0       0.94      0.89      0.91     16812
           1       0.44      0.59      0.50      2388

    accuracy                           0.85     19200
   macro avg       0.69      0.74      0.71     19200
weighted avg       0.88      0.85      0.86     19200

test
 0.47999999999999976
              precision    recall  f1-score   support

           0       0.94      0.89      0.92     16812
           1       0.44      0.58      0.50      2388

    accuracy                           0.86     19200
   macro avg       0.69      0.74      0.71     19200
weighted avg       0.88      0.86      0.86     19200

test
 0.48999999999999977
              precision    recall  f1-score   support

           0       0.94      0.90      0.92     16812
           1       0.44      0.57      0.50      2388

    accuracy                           0.86     19200
   macro avg       0.69      0.73      0.71     19200
weighted avg       0.88      0.86      0.87     19200

test
 0.4999999999999998
              precision    recall  f1-score   support

           0       0.94      0.90      0.92     16812
           1       0.45      0.56      0.50      2388
```

```
    accuracy                           0.86     19200
   macro avg       0.69      0.73      0.71     19200
weighted avg       0.88      0.86      0.87     19200

test
 0.5099999999999998
              precision    recall  f1-score   support

           0       0.94      0.91      0.92     16812
           1       0.46      0.56      0.50      2388

    accuracy                           0.86     19200
   macro avg       0.70      0.73      0.71     19200
weighted avg       0.88      0.86      0.87     19200

test
 0.5199999999999998
              precision    recall  f1-score   support

           0       0.93      0.91      0.92     16812
           1       0.46      0.55      0.50      2388

    accuracy                           0.86     19200
   macro avg       0.70      0.73      0.71     19200
weighted avg       0.88      0.86      0.87     19200

test
 0.5299999999999998
              precision    recall  f1-score   support

           0       0.93      0.91      0.92     16812
           1       0.46      0.54      0.50      2388

    accuracy                           0.86     19200
   macro avg       0.70      0.72      0.71     19200
weighted avg       0.87      0.86      0.87     19200

test
 0.5399999999999998
              precision    recall  f1-score   support

           0       0.93      0.91      0.92     16812
           1       0.47      0.53      0.49      2388

    accuracy                           0.87     19200
   macro avg       0.70      0.72      0.71     19200
weighted avg       0.87      0.87      0.87     19200
```

```
test
 0.5499999999999998
              precision    recall  f1-score   support

           0       0.93      0.92      0.92     16812
           1       0.47      0.52      0.49      2388

    accuracy                           0.87     19200
   macro avg       0.70      0.72      0.71     19200
weighted avg       0.87      0.87      0.87     19200

test
 0.5599999999999997
              precision    recall  f1-score   support

           0       0.93      0.92      0.93     16812
           1       0.48      0.51      0.49      2388

    accuracy                           0.87     19200
   macro avg       0.70      0.72      0.71     19200
weighted avg       0.87      0.87      0.87     19200

test
 0.5699999999999997
              precision    recall  f1-score   support

           0       0.93      0.92      0.93     16812
           1       0.48      0.50      0.49      2388

    accuracy                           0.87     19200
   macro avg       0.71      0.71      0.71     19200
weighted avg       0.87      0.87      0.87     19200

test
 0.5799999999999997
              precision    recall  f1-score   support

           0       0.93      0.93      0.93     16812
           1       0.49      0.49      0.49      2388

    accuracy                           0.87     19200
   macro avg       0.71      0.71      0.71     19200
weighted avg       0.87      0.87      0.87     19200

test
 0.5899999999999997
              precision    recall  f1-score   support
```

```
              0       0.93      0.93      0.93     16812
              1       0.49      0.48      0.49      2388

       accuracy                           0.87     19200
      macro avg       0.71      0.70      0.71     19200
   weighted avg       0.87      0.87      0.87     19200
```

13.2 Code of the plots

[132]:
```
def _check_value_outside_range(x, lower, upper):
    value_out = (x < lower) | (x > upper)
    return value_out


def _check_within_range(sequence, lower, upper):
    is_bad = reduce((lambda x, y: x | y), (_check_value_outside_range(x, lower,␣
 ↪upper) for x in sequence))
    return is_bad


def _check_in_possible_values(sequence, possible_values: set):
    is_bad = len(set(sequence).difference(possible_values)) > 0
    return is_bad
```

[133]:
```
from functools import reduce
import os
import plotly.graph_objects as go
import plotly.offline as poff

def _output_plot(fig, file_name, html_dir, png_dir, show_plot):
    if show_plot:
        fig.show()
    if png_dir:
        file_path = os.path.join(png_dir, file_name + r'.png')
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        fig.write_image(file_path)
    if html_dir:
        file_path = os.path.join(html_dir, file_name + r'.html')
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        poff.plot(fig, filename=file_path, auto_open=False)


def _update_layout(fig, height=800, title=""):
    fig.update_layout(
    autosize=False,
```

```
    width=1000,
    height=height,
    margin=go.layout.Margin(
        l=50,
        r=50,
        b=100,
        t=100,
        pad=4
    ),
    paper_bgcolor='lightsteelblue',
    title = dict(text="<b>{}</b>".format(title),
                 font=dict(size=20)
                 )
    )
```

[134]:
```
# ROC CURVE ON TRAIN AND TEST AND PRECISION-RECALL DISTRIBUTION OVER DIFFERENT␣
↪THRESHOLDS
def plot_roc_auc(probs_train, probs_test, y_train, y_test, show_AUC_plot=False,␣
↪show_PREC_REC_plot=False, title_AUC=None, title_PREC_REC=None, html_dir=None,␣
↪png_dir=None):

    """Plots ROC curve against straight line referred to random model, both on␣
↪train and test set.
        It also plots precision and recall values on different levels of␣
↪threshold.

    Arguments:
        probs_train {1D array, list or tuple, or a pandas Series} -- array of␣
↪predicted probabilities on train set. Values must be within the continuous␣
↪interval [0,1]
        probs_test {1D array, list or tuple, or a pandas Series} -- array of␣
↪predicted probabilities on test set. Values must be within the continuous␣
↪interval [0,1]
        y_train {1D array, list or tuple, or a pandas Series} -- groundtruth on␣
↪train set. Values must be in {0,1}
        y_test {1D array, list or tuple, or a pandas Series} -- groundtruth on␣
↪test set. Values must be in {0,1}

    Keyword Arguments:
        show_AUC_plot {bool} -- whether to show AUC plot (default: {False})
        show_PREC_REC_plot {bool} -- whether to show PRECISION-RECALL plot␣
↪(default: {False})
        html_dir {str} -- path where interactive html file will be saved; if␣
↪None, nothing will be saved (default: {None})
        png_dir {str} -- path where static png file will be saved; if None,␣
↪nothing will be saved (default: {None})
```

```
    Return:
        None
    """

    # check input params
    binary_values = set([0, 1])
    if (len(probs_train) != len(y_train)) | (len(probs_test) != len(y_test)):
        raise ValueError("Check input dimensions! Train arrays must have same␣
↪length, as well as test arrays")

    if _check_in_possible_values(y_train, binary_values) |␣
↪_check_in_possible_values(y_test, binary_values):
        raise ValueError("`y_train` and `y_test` must be arrays with values 0 or␣
↪1")

    if _check_within_range(probs_train, 0, 1) | _check_within_range(probs_test,␣
↪0, 1):
        raise ValueError("Values of `probs_train` and `probs_test` must be␣
↪within the continuous interval [0,1]")

    ### FIRST PLOT: ROC CURVE ###
    # auc_train
    fpr_train, tpr_train, thresholds_train = roc_curve(y_train, probs_train)
    roc_auc_train = roc_auc_score(y_train, probs_train)
    # auc_test
    fpr_test, tpr_test, thresholds_test = roc_curve(y_test, probs_test)
    roc_auc_test = roc_auc_score(y_test, probs_test)

    trace0 = go.Scatter(x=fpr_train, y=tpr_train, name='ROC Train')
    trace1 = go.Scatter(x=fpr_test, y=tpr_test, name='ROC Test')
    trace2 = go.Scatter(x=[0, 1], y=[0, 1], name='RandomClassfier')
    data = [trace0, trace1, trace2]
    layout = {
        'xaxis': {'title': 'False Positive Rate'},
        'yaxis': {'title': 'True Positive Rate'}
    }

    fig = go.Figure({'data': data, 'layout': layout})
    if title_AUC is None:
        title_AUC = 'Roc Curve - AUC Train: {:.5f} - AUC Test: {:.5f}'.
↪format(roc_auc_train, roc_auc_test)
    _update_layout(fig, height=800, title=title_AUC)
    _output_plot(fig, 'AUC_ROC', html_dir, png_dir, show_AUC_plot)

    ### SECOND PLOT: PRECISION-RECALL ###
```

```
    # precision recall on test
    precision, recall, thresholds = precision_recall_curve(y_test, probs_test)

    trace0 = go.Scatter(x=thresholds, y=precision[:-1], name='Precision')
    trace1 = go.Scatter(x=thresholds, y=recall[:-1], name='Recall')
    data = [trace0, trace1]
    layout = {
        'title': 'Precision-Recall vs Threshold',
        'xaxis': {'title': 'Threshold', 'range': [0, 1]}
    }

    fig = go.Figure({'data': data, 'layout': layout})
    if title_PREC_REC is None:
        title_PREC_REC = 'Precision-Recall curves'
    _update_layout(fig, height=800, title=title_PREC_REC)
    _output_plot(fig, 'Prec_Recall', html_dir, png_dir, show_PREC_REC_plot)
```

[ ]:
```
plot_roc_auc(probs_train, probs_test, y_train, y_test, show_AUC_plot=True,␣
↪show_PREC_REC_plot=True, title_AUC=None, title_PREC_REC=None, html_dir=None,␣
↪png_dir=None):
```