PROJECT REPORT

Course : **Machine Learning**
Code : **AI511**

Project Topic : **PUBG-PREDICTION**

Faculty Incharge : Prof. Dinesh Babu
Prof. Neelam Sinha

Project TA : Preyas Garg

Team Name : MinusOne
Team Members : Narasimhan N(MT2022062)
Sreenidhi K S (MT2022115)

Problem Statement

In a PUBG game, up to 100 players start in each match. Players can be on teams which get ranked at the end of the game based on how many other teams are still alive when they are eliminated. In game, players can pick up different ammunition, revive knocked teammates, drive vehicles, swim, run, shoot, heal etc

We are provided with a large number of anonymized PUBG game stats, formatted so that each row contains one player's post-game stats. The data comes from matches of all types: solos, duos, squads, and custom; there is no guarantee of there being 100 players per match, nor at most 4 players per group.

We must create a model which predicts players' finishing placement based on their final stats, on a scale from 1 (first place) to 0 (last place).

Kaggle Link : *https://www.kaggle.com/competitions/pubg-version-3/overview*

Dataset Description

The PUBG dataset available on Kaggle has up to 100 players in each match which are uniquely identified based on their matchId.

The players can also form teams in a match by having the same groupId and the same final placement in the match.

The teams can be formed based on solo, duo, squad and customs.

There are around 3 Lakh train data points and a total of 1.3 Lakh test data points.

There are 29 features, many were Numeric and Object type

There was only 1 null value , which we dropped

ID	Description
DBNOs	Number of enemy players knocked.
assists	Number of enemy players this player damaged that were killed by teammates.
boosts	Number of boost items used.
damageDealt	Total damage dealt.
headshotKills	Number of enemy players killed with headshots.
heals	Number of healing items used.
Id	Player's Id
killPlace	Ranking in match of number of enemy players killed.
killPoints	Kills-based external ranking of player.
killStreaks	Max number of enemy players killed in a short amount of time.
kills	Number of enemy players killed.
longestKill	Longest distance between player and player killed at time of death.
matchDuration	Duration of match in seconds.
matchId	ID to identify match.
matchType	String identifying the game mode that the data comes from.
rankPoints	Elo-like ranking of player.
revives	Number of times this player revived teammates.
rideDistance	Total distance traveled in vehicles measured in meters.
roadKills	Number of kills while in a vehicle.
swimDistance	Total distance traveled by swimming measured in meters.
teamKills	Number of times this player killed a teammate.
vehicleDestroys	Number of vehicles destroyed.
walkDistance	Total distance traveled on foot measured in meters.
weaponsAcquired	Number of weapons picked up.
winPoints	Win-based external ranking of player.
groupId	ID to identify a group within a match.
numGroups	Number of groups we have data for in the match.
maxPlace	Worst placement we have data for in the match.
winPlacePerc	The target of prediction. This is a percentile winning placement where 1 corresponds to 1st place and 0 corresponds to last place in the match.

Approach Overview

PUBG consists of multiple match types , mainly solo ,duo and squad , in which duo and squad are played as a team game ,hence a team member's ranking is dependent upon the performance of other team members also.

The ranking is given as a team ranking rather than individual ranking , hence it is very possible that a team member gets killed very early in the game and if his team goes on to winning the match , the entire team will get first place So we decided to aggregate a team's performance and then predict the target based as a team.

Since we are aggregating based on match type ,predictions for may vary for different match types, for example in solo no of kills belongs to only one person , whereas in squad no of kills is sum of all kills by team members , hence it might give inaccurate predictions ,so we decided to split the data frame into 3 parts based on match type and train them separately.

DATA PREPROCESSING

OUTLIER REMOVAL :**Possible Hackers :**

1) Walk distance is zero but kills are very high
   Condition for Checking : Walk distance=0 and kills>0
   No of Outliers : 992
2) Walk distance is zero but weapons acquired are high
   Condition for Checking : Walk distance = 0 and weapons acquired > 0
   No of Outliers : 11736
3) Walk distance is zero but damageDelt are high
   Condition for Checking : Walk distance = 0 and Damage delt are > 0
   No of Outliers : 1190
4) Walk distance is zero but DBNO are high
   Condition for Checking : Walk distance = 0 and DBNO > 0
   No of Outliers : 44
5) Walk is zero but assist are high
   Condition for Checking : Walk distance = 0 and assists > 0
   No of Outliers : 5
6) When Walk is zero but Ride distance is greater than zero
   Condition for Checking : Walk distance = 0 and ride distance > 0
   No of Outliers : 86
7) Walk distance is zero and Heals/Boosts is greater than zero
   Condition for Checking : Walk distance = 0 and heals/boots> 0
   No of Outliers : 4
8) Very long kills
   Condition for Checking : Walk distance < 50 and have range > 900m
   No of Outliers : 4

9) Very high Headshots
   Condition for Checking : Headshots > 30
   No of Outliers : 8

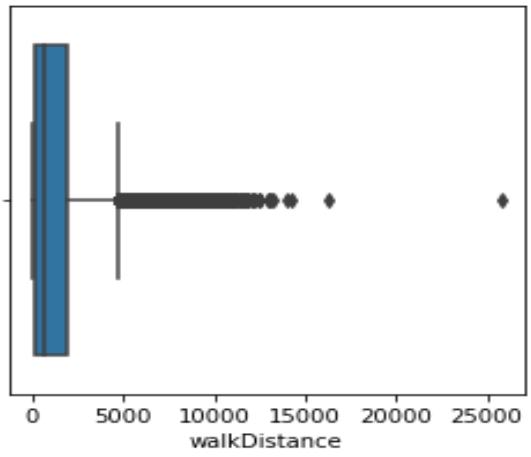
Other Outliers :

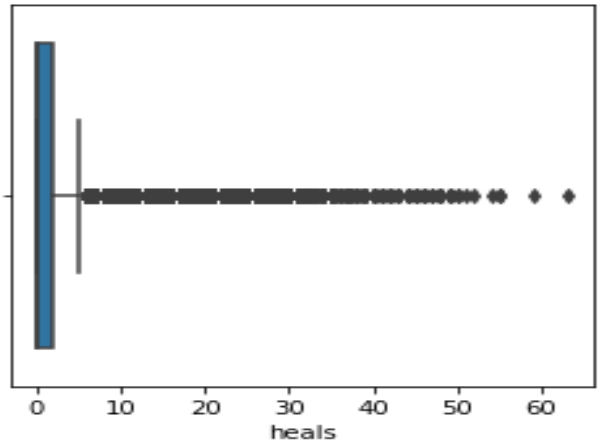
10) When walk distance is more but Win percentage = 0%
    Condition for Checking : Walk distance > 2 km and Win percentage = 0%
    No of Outliers : 58

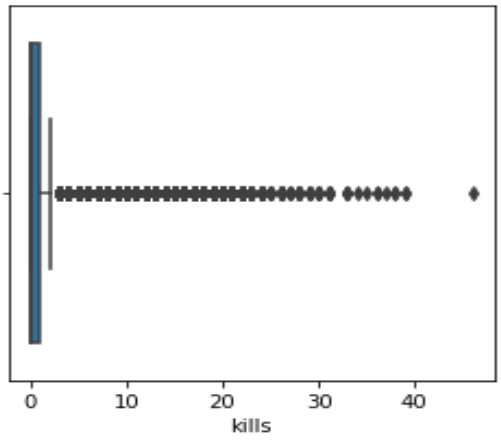
11) Walk distance is zero but Assist count is greater than zero
    Condition for Checking : Win percentage is 0% and assists is > 1
    No of Outliers : 212

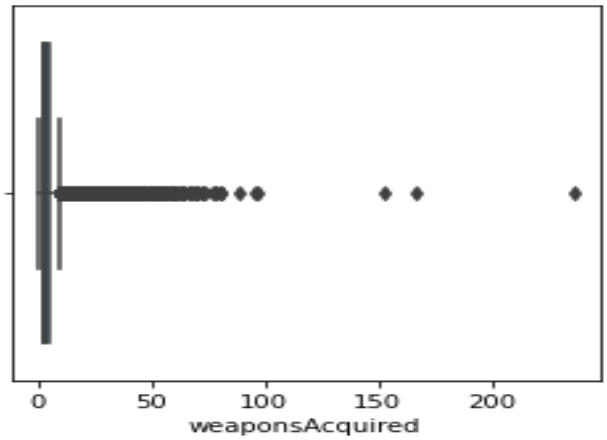
12) Win percentage is 0% but kills is a positive number
    Condition for Checking : Win percentage is 0% but kills is > 1
    No of Outliers : 1942

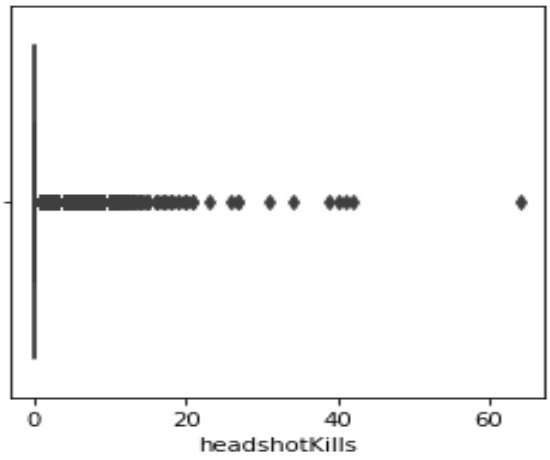
Few Visualizations of the above outliers :

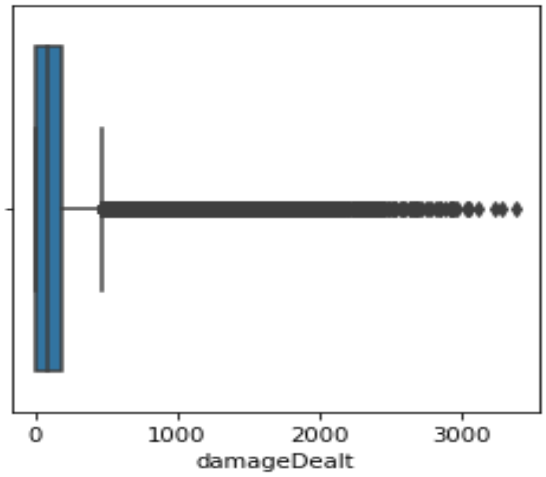




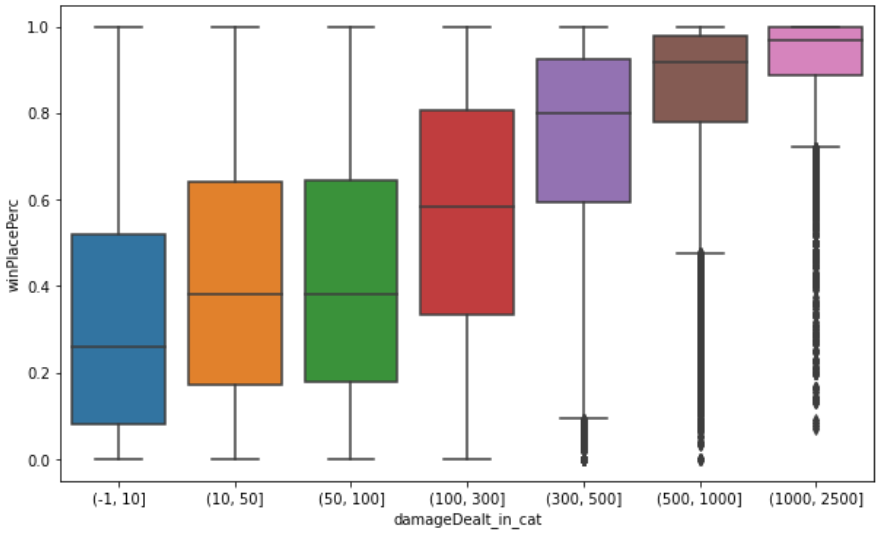




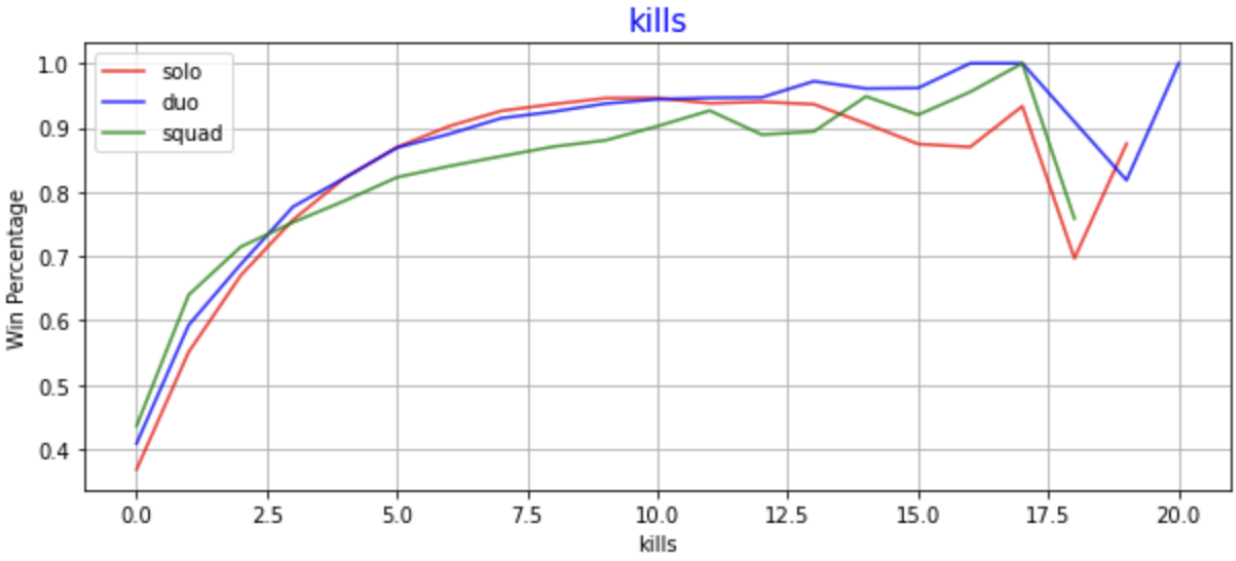




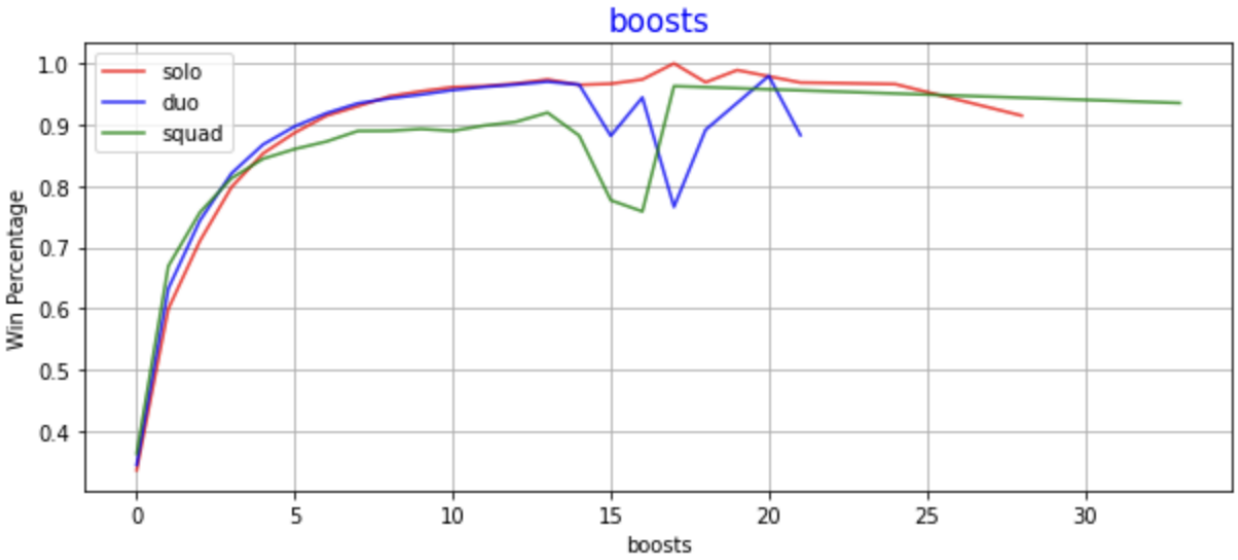
EXPLORATORY DATA ANALYSIS :



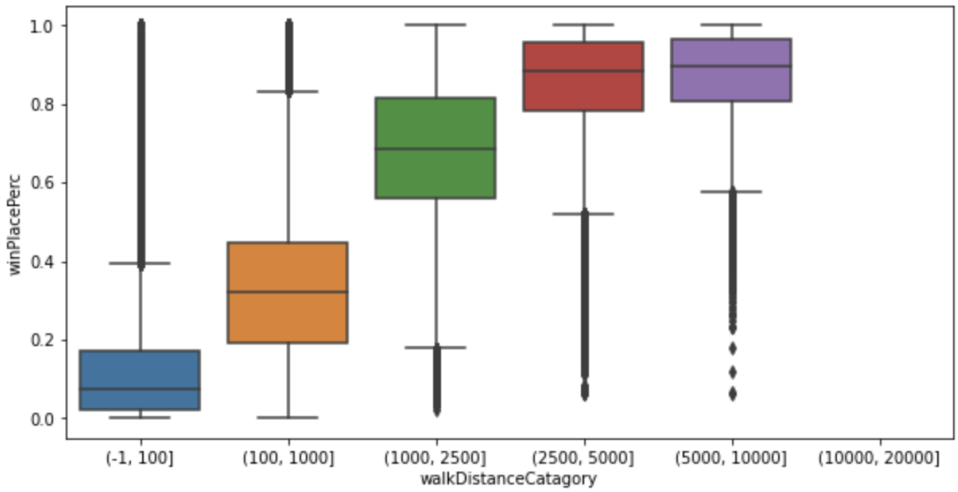
From the above plot, as Damage Dealt (damage caused to the opponent) increases, the chance of winning also increases.. Here damage dealt is grouped into ranges such as 0-10, 10-50, 50-100 etc.. and win percentage is calculated for the particular range.



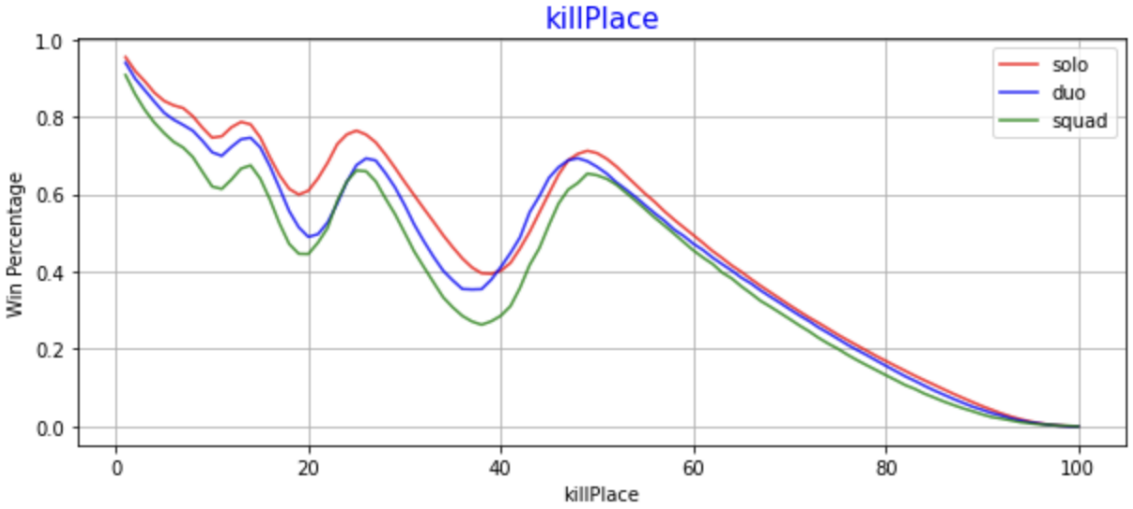
From the above plot, as kills increase, the chance of winning also increases. This is due to the fact that the more kills are made, lesser the number of opponents and the better is the ranking, hence Win percentage is high..



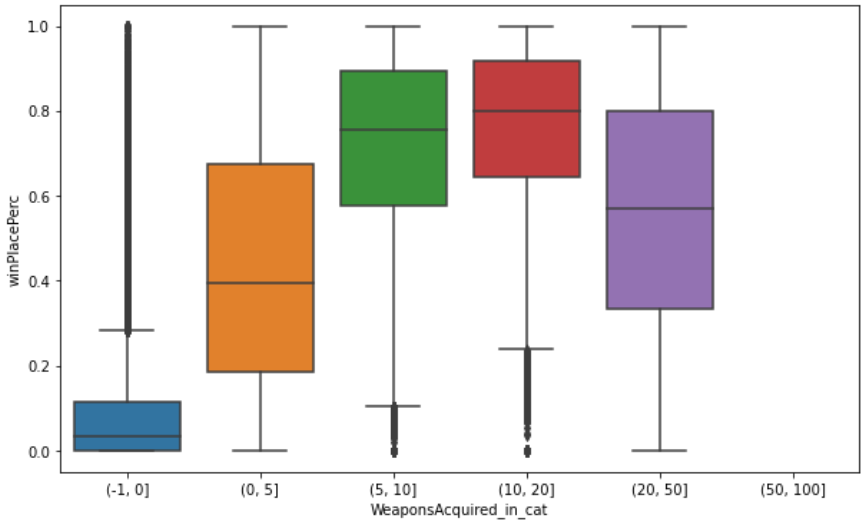
From the above plot, as boosts increase, the chance of winning also increases. This is due to the fact that the more boosts taken increases the chance of survival and hence has a better chance of winning.



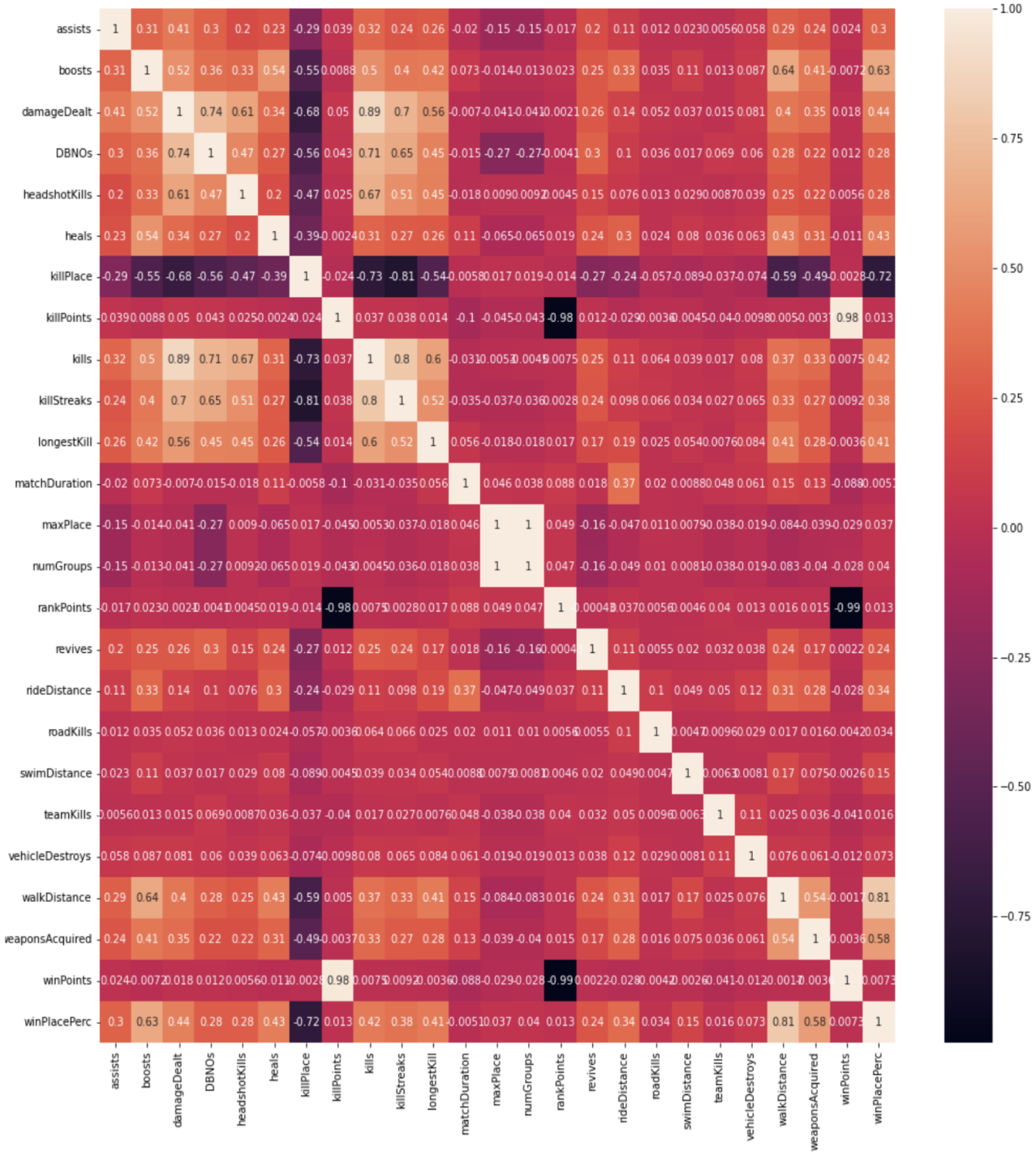
From the above plot, as Walk distance increases, the chance of winning also increases. Here walk distance is grouped into ranges of 0-100m, 101-1000m, etc and win percentage is calculated for the particular range.



From the above plot, as Kill place increase, the chance of winning is low. This is due to the fact that the more kills means the lesser the rank. The one with highest kills will have kill place as one



From the above plot, as Weapons Acquired increases as the chances of winning also increases. Here weapons are grouped into ranges like 0,1-5, 5-10,etc.. and win percentage is calculated for the given range.



The above heatmap represents the correlation matrix of different attributes. The lighter the values implies high positive correlation and the darker value implies high negative correlation.

It is evident that walk distance, kill place, weapons acquired have high correlation with the target attribute (Win place percentage).

Also there are attributes such as killPoints and winPoints that are highly correlated, and hence can be dropped during data cleaning.

Also killPlace is highly correlated with kills and killStreak so any two of them can be dropped during data cleaning,

FEATURE ENGINEERING

The attributes with high correlation with target can be featured engineered to get proper insights on the given data.

New Column	Formula	Infernce
walkDistancePerMatchDuration	walkDistance /matchDuration	Useful
weaponsAcquiredPerWalkDistance	weaponsAcquired / walkDistance	Useful
killsPerWalkDistance	kills / walkDistance	Useful
damageDeltPerWalkDistance	damageDealt / walkDistance	Useful
boostsPerWalkDistance	boosts / walkDistance	Useful
killPlacePerMaxPlace	killPlace /maxPlace	Useful
killPlacePerNumGroups	killPlace / numGroups	Useful
healthItems	heals +boosts	NOT USEFUL
totalDistance	rideDistance + walkDistance + swimDistance	NOT USEFUL

walkDistancePerMatchDuration is feature engineered into walkDistance/matchDistance. It is to find how much he walks on average per minute. This is to normalize to find out new patterns.

weaponsAcquiredPerWalkDistance is feature engineered into weaponsAcquired/walkDistance. It is to find how many weapons are acquired in unit distance. This is to normalize to find out new patterns.

killsPerWalkDistance is feature engineered into no of kills / walkDistance. It is to find how many people were killed in unit distance. This is to normalize to find out new patterns.

damageDelthPerWalkDistance is feature engineered into damageDealt/ walkDistance. It is to find how much damage was dealt in unit distance. This is to normalize to find out new patterns.

boostaPerWalkDistance is feature engineered into boosts/ walkDistance. It is to find how many health boosts were taken in unit distance. This is to normalize to find out new patterns.

killPlacePerMaxPlace is feature engineered into killPlace/maxPlace. maxPlace is the last person rank and kill place gives the rank of the particular player. The ratio adds weight to rank along with respect to number of players. This is due to the number of players as getting rank 1 with 4 people is different from getting rank 1 with 16 players. This is to normalize to find out new patterns. The same is for killPlacePerNumGroups where killPlace is divided by the number of groups.

healthItems is featured engineered into heals + boosts. It was under the intuition that both belonged to the same category, but this was not considered as the Information gain is very less.

totalDistance is featured engineered into a sum of rideDistance, walkDistance and swimDistance. It was under the intuition that all distance would be same with respect to the correlation with the target, but this was not considered as the Information gain is very less.

The inference is derived by comparing the Information gain before and after feature engineering and hence attributes like healthItems and totalDistance had been dropped.

Attributes Dropped :

Column Deleted	Reason
rankPoints **winPoints** **killPoints**	Incomplete with many missing data, and less correlation
teamKills **vehicleDestroys** **roadKills** **swimDistance**	less correlation with target
matchId	not required for training

Top 10 Features BEFORE Feature Engineering :

COLUMN	Information Gain
maxPlace	2.378594
numGroups	1.181999
killPlace	0.948081
walkDistance	0.733891
weaponsAcquired	0.326114
boosts	0.322689
heals	0.203367
damageDealt	0.162876
longestKill	0.144017
kills	0.128810

COLUMN	Correlation Score
walkDistance	0.810920
killPlace	-0.719009
boosts	0.634121
weaponsAcquired	0.582801
damageDealt	0.440601
longestKill	0.409985
kills	0.419801
heals	0.427901
rideDistance	0.342521
killStreaks	0.377600

Top 10 Features AFTER Feature Engineering :

COLUMN	Information Gain
killPlacePerMaxPlace	2.499106
maxPlace	2.367130
killPlacePerNumGroups	1.493418
numGroups	1.164521
killPlace	0.946137
walkDistancePerMatchDuration	0.782216
walkDistance	0.740667
weaponsAcquiredPerWalkDistance	0.495817
weaponsAcquired	0.337588
boosts	0.325896

COLUMN	Correlation Score
killPlace	-0.725854
killPlacePerNumGroups	-0.609368
killPlacePerMaxPlace	-0.608327
heals	0.432353
kills	0.434179
damageDealt	0.456147
weaponsAcquired	0.614288
boosts	0.637994
walkDistance	0.817985
walkDistancePerMatchDuration	0.831719

Aggregating based on Group ID

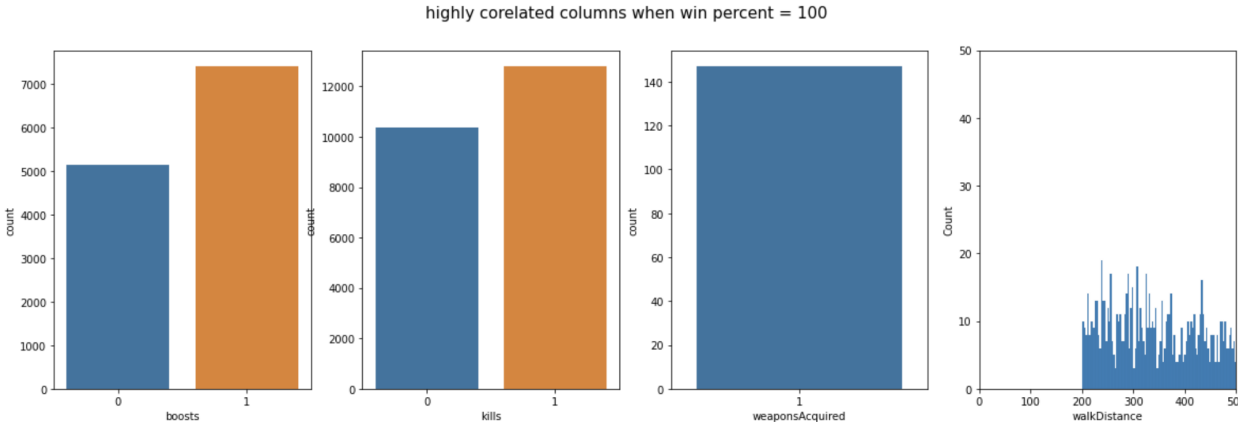
Since PUBG is mostly a group game (except in solo) the ranking of a player depends on the ranking of the other players in the group

for example : even if a player in a group gets out early , where as his team-mates go on to win the match , the player who got out early still gets 1st place ,so if we make predictions just by seeing a player , it will lead to many inaccuracies , therefore we group the data based on group id and collectively judge the groups performance.

Even the testing data would be grouped by the group ID on which the model will predict.

From EDA, it is evident that walkDistance, killPlace, boosts and weaponsAcquired have high correlation with target attribute

Plotting data values with high high correlated attributes with winPlacePerc (Target attribute)



From the above plot, it can be seen that even when boosts and kills are zero, the winning percentage given is 100%. Also when weapons acquired is 1 the winning percentage is 100%. It is also evident that even with a walk distance around 200m, still the win percentage is 100%.

Hence grouping is required to make sure there is no inconsistency in the training,

Attribute used to form Different groups : Match Type

Attribute on which Grouping is performed : Group ID

Number of Groups : 3:

List of Groups :

Name of the Group	Match Type values
Solo	Solo-fpp, solo, normal-solo, normal-solo-fpp
Dual	Duo-fpp, duo, normal-duo-fpp, normal-duo
Squad	squad-fpp, squad, normal-squad-fpp, normal-squad

There would be 3 groups formed and one model for each group where each model would be trained on a different dataset based on which group it belongs to.

Columns Aggregated By :

Column	Aggregate By
winPlacePerc	max
killStreaks	max
longestKill	max
walkDistancePerMatchDuration	sum,mean,min,max
weaponsAcquiredPerWalkDistance	sum,mean,min,max
killsPerWalkDistance	sum,mean,min,max
damageDeltPerWalkDistance	sum,mean,min,max
rideDistance	sum
DBNOs	sum
assists	sum
numGroups	first
maxPlace	first
damageDealt	sum,mean,min,max
revives	sum
headshotKills	sum
kills	sum,mean,min,max
weaponsAcquired	sum,mean,min,max
boosts	sum,mean,min,max
heals	sum,mean,min,max
matchDuration	first
walkDistance	sum,mean,min,max
killPlace	sum,mean,min,max
killPlacePerMaxPlace	sum,mean,min,max
killPlacePerNumGroups	sum,mean,min,max

Top 10 Features after Grouping :

SQUAD

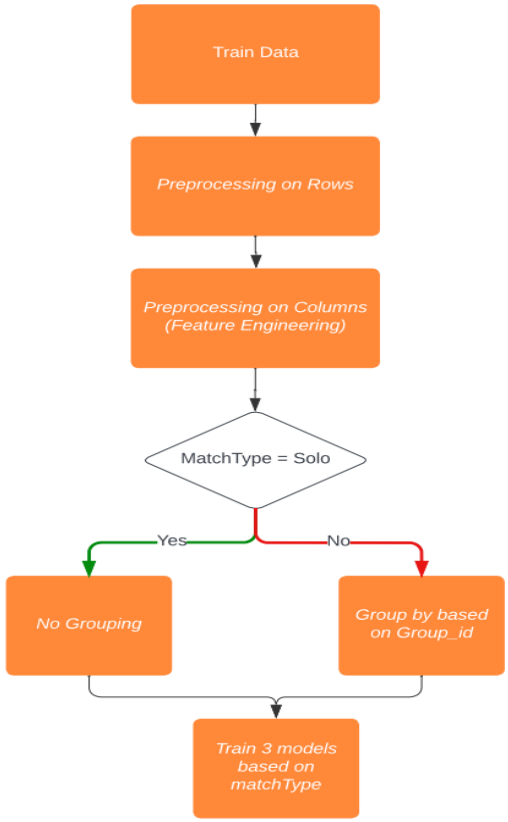
COLUMN	Information Gain
killPlacePerMaxPlacemax	2.413984
killPlacePerMaxPlacemin	2.040084
maxPlacefirst	1.687704
killPlacePerMaxPlacemean	1.555259
killPlacePerNumGroupsmax	1.483027
killPlacePerMaxPlacesum	1.413369
killPlacePerNumGroupsmin	1.099005
killPlacemax	0.993555
killPlacePerNumGroupsmean	0.854838
walkDistancePerMatchDurationmean	0.825030

SOLO

COLUMN	Information Gain
killPlacePerMaxPlace	2.972752
maxPlace	2.201047
killPlacePerNumGroups	1.677426
killPlace	1.296133
walkDistancePerMatchDuration	0.981246
walkDistance	0.910917
weaponsAcquiredPerWalkDistance	0.608609
numGroups	0.556885
weaponsAcquired	0.421503
boosts	0.360736

TRAINING :

The training data is grouped into 3 categories based on the match type. The data within each category is grouped on the attribute GroupID and remaining attributes are aggregate respectively.Each category has its own model namely Solo, Dual and Squad.



BENCHMARKING :

For different types of Regression models with their best parameters and best score.

1) LINEAR REGRESSION :

For Solo Grouped Data :

```
model = Ridge() # For Linear Regression

cv = RepeatedKFold(n_splits=5, n_repeats=10, random_state=10)
param = {
    'alpha': [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100]
}

# Grid Search : Linear Regression for Solo Data
search = GridSearchCV(model, param, cv=cv, verbose=3)
resSolo = search.fit(solo_df.drop(['winPlacePerc'], axis = 1), solo_df['winPlacePerc'])

# Result : Linear Regression for Solo Data
print('Best Score: %s' % resSolo.best_score_)
print('Best Hyperparameters: %s' % resSolo.best_params_)
```

```
[CV 38/50] END .....................alpha=100;, score=0.905 total time=   0.2s
[CV 39/50] END .....................alpha=100;, score=0.904 total time=   0.2s
[CV 40/50] END .....................alpha=100;, score=0.904 total time=   0.2s
[CV 41/50] END .....................alpha=100;, score=0.904 total time=   0.2s
[CV 42/50] END .....................alpha=100;, score=0.905 total time=   0.2s
[CV 43/50] END .....................alpha=100;, score=0.904 total time=   0.2s
[CV 44/50] END .....................alpha=100;, score=0.905 total time=   0.2s
[CV 45/50] END .....................alpha=100;, score=0.897 total time=   0.2s
[CV 46/50] END .....................alpha=100;, score=0.904 total time=   0.2s
[CV 47/50] END .....................alpha=100;, score=0.904 total time=   0.2s
[CV 48/50] END .....................alpha=100;, score=0.902 total time=   0.2s
[CV 49/50] END .....................alpha=100;, score=0.905 total time=   0.2s
[CV 50/50] END .....................alpha=100;, score=0.902 total time=   0.2s
Best Score: 0.9030150226209237
Best Hyperparameters: {'alpha': 1e-05}
```

For Dual Grouped Data :

```
# Grid Search : Linear Regression for Dual
search = GridSearchCV(model, param, cv=cv, verbose=3)
resDuo = search.fit(duo_df.drop(['winPlacePerc'], axis = 1), duo_df['winPlacePerc'])

# Result : Linear Regression for Dual Data
print('Best Score: %s' % resDuo.best_score_)
print('Best Hyperparameters: %s' % resDuo.best_params_)
```

```
[CV 34/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 35/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 36/50] END .....................alpha=100;, score=0.886 total time=   0.9s
[CV 37/50] END .....................alpha=100;, score=0.894 total time=   0.9s
[CV 38/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 39/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 40/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 41/50] END .....................alpha=100;, score=0.889 total time=   0.9s
[CV 42/50] END .....................alpha=100;, score=0.892 total time=   1.0s
[CV 43/50] END .....................alpha=100;, score=0.892 total time=   1.0s
[CV 44/50] END .....................alpha=100;, score=0.894 total time=   1.0s
[CV 45/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 46/50] END .....................alpha=100;, score=0.887 total time=   1.0s
[CV 47/50] END .....................alpha=100;, score=0.893 total time=   1.0s
[CV 48/50] END .....................alpha=100;, score=0.894 total time=   1.0s
[CV 49/50] END .....................alpha=100;, score=0.894 total time=   1.1s
[CV 50/50] END .....................alpha=100;, score=0.892 total time=   1.0s
Best Score: 0.8918133584503248
Best Hyperparameters: {'alpha': 100}
```

For Squad Grouped Data :

```
# Grid Search : Linear Regression for Squad
search = GridSearchCV(model, param, cv=cv, verbose=3)
resSquad = search.fit(squad_df.drop(['winPlacePerc'], axis = 1), squad_df['winPlacePerc'])

# Result : Linear Regression for Squad Data
print('Best Score: %s' % resSquad.best_score_)
print('Best Hyperparameters: %s' % resSquad.best_params_)
```

```
[CV 34/50] END .....................alpha=100;, score=0.868 total time=   1.4s
[CV 35/50] END .....................alpha=100;, score=0.866 total time=   1.2s
[CV 36/50] END .....................alpha=100;, score=0.866 total time=   1.3s
[CV 37/50] END .....................alpha=100;, score=0.869 total time=   1.2s
[CV 38/50] END .....................alpha=100;, score=0.868 total time=   1.3s
[CV 39/50] END .....................alpha=100;, score=0.866 total time=   1.1s
[CV 40/50] END .....................alpha=100;, score=0.867 total time=   1.2s
[CV 41/50] END .....................alpha=100;, score=0.868 total time=   1.2s
[CV 42/50] END .....................alpha=100;, score=0.868 total time=   1.2s
[CV 43/50] END .....................alpha=100;, score=0.864 total time=   1.3s
[CV 44/50] END .....................alpha=100;, score=0.868 total time=   1.3s
[CV 45/50] END .....................alpha=100;, score=0.867 total time=   1.2s
[CV 46/50] END .....................alpha=100;, score=0.868 total time=   1.1s
[CV 47/50] END .....................alpha=100;, score=0.866 total time=   1.3s
[CV 48/50] END .....................alpha=100;, score=0.868 total time=   1.3s
[CV 49/50] END .....................alpha=100;, score=0.867 total time=   1.3s
[CV 50/50] END .....................alpha=100;, score=0.867 total time=   1.2s
Best Score: 0.8670896342291694
Best Hyperparameters: {'alpha': 100}
```

Benchmarking : For Ridge Regression Models with different parameters

Parameter - Alpha	Score : Solo Model	Score : Dual Model	Score : Squad Model
1e-07	0.9030150226209629	0.8918107597551744	0.8670887324596849
1e-06	0.9030150226209629	0.8918107597553199	0.8670887324596939
1e-05	0.9030150226209237	0.8918107597567841	0.8670887324597838
1e-04	0.9030150226205675	0.891810759771423	0.8670887324606825
1e-03	0.9030150226170067	0.8918107599178035	0.8670887324696693
0.01	0.9030150225813918	0.891810761381029	0.8670887325595352
0.1	0.9030150222244786	0.891810775954864	0.8670887334581046
1	0.9030150185791057	0.8918109160510385	0.8670887424348155
10	0.9030149745093616	0.8918119034357299	0.8670888313072785
100	0.9030137804241819	0.8918133584503248	0.8670896342291694

2) XG BOOST :

For Solo Grouped Data:

```
# Grid Search for Solo Data on XG Boost
xgb1 = XGBRegressor()
parameters = {'nthread':[4], #when use hyperthread, xgboost may become slower
              'objective':['reg:linear'],
              'learning_rate': [.03, 0.05, .07], #so called `eta` value
              'max_depth': [5, 6, 7],
              'min_child_weight': [4],
              'silent': [1],
              'subsample': [0.7],
              'colsample_bytree': [0.7],
              'n_estimators': [500],
             'tree_method':['gpu_hist']}

xgb_grid = GridSearchCV(xgb1,
                        parameters,
                        cv = 2,
                        n_jobs = 5,
                        verbose=True)

xgb_grid.fit(solo_df.drop(columns='winPlacePerc',axis=1), solo_df['winPlacePerc'])

print(xgb_grid.best_score_)
print(xgb_grid.best_params_)
```

```
Fitting 2 folds for each of 9 candidates, totalling 18 fits
[01:30:13] WARNING: C:/buildkite-agent/builds/buildkite-windows-cpu-autoscaling-group-i-03de431ba26204c4d-1/xgboost/xgboost-ci-
windows/src/objective/regression_obj.cu:213: reg:linear is now deprecated in favor of reg:squarederror.
[01:30:13] WARNING: C:/buildkite-agent/builds/buildkite-windows-cpu-autoscaling-group-i-03de431ba26204c4d-1/xgboost/xgboost-ci-
windows/src/learner.cc:767:
Parameters: { "silent" } are not used.

0.9620177534505304
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'obj
ective': 'reg:linear', 'silent': 1, 'subsample': 0.7, 'tree_method': 'gpu_hist'}
```

For Dual Grouped Data:

```
# Grid Search for Duo grouped Data on XG Boost
xgb1 = XGBRegressor()
parameters = {'nthread':[4], #when use hyperthread, xgboost may become slower
              'objective':['reg:linear'],
              'learning_rate': [.03, 0.05, .07], #so called `eta` value
              'max_depth': [5, 6, 7],
              'min_child_weight': [4],
              'silent': [1],
              'subsample': [0.7],
              'colsample_bytree': [0.7],
              'n_estimators': [500],
             'tree_method':['gpu_hist']}

xgb_grid = GridSearchCV(xgb1,
                        parameters,
                        cv = 2,
                        n_jobs = 5,
                        verbose=True)

xgb_grid.fit(duo_df.drop(columns='winPlacePerc',axis=1), duo_df['winPlacePerc'])

print(xgb_grid.best_score_)
print(xgb_grid.best_params_)
```

```
Fitting 2 folds for each of 9 candidates, totalling 18 fits
[11:59:40] WARNING: C:/buildkite-agent/builds/buildkite-windows-cpu-autoscaling-group-i-03de431ba26204c4d-1/xgboost/xgboost-ci-
windows/src/objective/regression_obj.cu:213: reg:linear is now deprecated in favor of reg:squarederror.
[11:59:41] WARNING: C:/buildkite-agent/builds/buildkite-windows-cpu-autoscaling-group-i-03de431ba26204c4d-1/xgboost/xgboost-ci-
windows/src/learner.cc:767:
Parameters: { "silent" } are not used.

0.9546865985514996
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'obj
ective': 'reg:linear', 'silent': 1, 'subsample': 0.7, 'tree_method': 'gpu_hist'}
```

For Squad Grouped Data:

```
# Grid Search for Squad Data on XG Boost
xgb1 = XGBRegressor()
parameters = {'nthread':[4], #when use hyperthread, xgboost may become slower
              'objective':['reg:linear'],
              'learning_rate': [.03, 0.05, .07], #so called `eta` value
              'max_depth': [5, 6, 7],
              'min_child_weight': [4],
              'silent': [1],
              'subsample': [0.7],
              'colsample_bytree': [0.7],
              'n_estimators': [500],
             'tree_method':['gpu_hist']}

xgb_grid = GridSearchCV(xgb1,
                        parameters,
                        cv = 2,
                        n_jobs = 5,
                        verbose=True)

xgb_grid.fit(squad_df.drop(columns='winPlacePerc',axis=1), squad_df['winPlacePerc'])

print(xgb_grid.best_score_)
print(xgb_grid.best_params_)
```

```
Fitting 2 folds for each of 9 candidates, totalling 18 fits
[09:47:28] WARNING: C:/buildkite-agent/builds/buildkite-windows-cpu-autoscaling-group-i-03de431ba26204c4d-1/xgboost/xgboost
windows/src/objective/regression_obj.cu:213: reg:linear is now deprecated in favor of reg:squarederror.
[09:47:28] WARNING: C:/buildkite-agent/builds/buildkite-windows-cpu-autoscaling-group-i-03de431ba26204c4d-1/xgboost/xgboost
windows/src/learner.cc:767:
Parameters: { "silent" } are not used.

0.9315941745787972
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4,
ective': 'reg:linear', 'silent': 1, 'subsample': 0.7, 'tree_method': 'gpu_hist'}
```

Benchmarking : For XG Boost Models with different parameters

Squad Model

PARAMETERS	Mean Absolute Error For Solo Model
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04277823164065137
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.042693481510916204
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04253511252141473
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04250890869370812
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04281594789905563
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.042718450311204303
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04258836995790318
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.042534955041867294
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.042708741885590595
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.042653739169740836
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04250346078504994
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04246925276328048

Duo Model

PARAMETERS	Mean Absolute Error For Duo Model
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04712668620785963
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04707425384270283
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.046984259960902275
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.0469556356273892
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.047186249716432434
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04712572601281344
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04694967451225951
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.046923191138998045
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04710603799135232
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04705759494241881
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04703563942424392
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.04702919172339137

Squad Model

PARAMETERS	Mean Absolute Error For Squad Model
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06361005108499965
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06354192555817892
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06346587313767461
{'colsample_bytree': 0.7, 'learning_rate': 0.05, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06345825547122182
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06368341112786136
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06358561493414053
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06341990514865593
{'colsample_bytree': 0.7, 'learning_rate': 0.04, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06338337728980284
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06359596562743011
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 7, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06354468636892242
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 500, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06350351184681771
{'colsample_bytree': 0.7, 'learning_rate': 0.06, 'max_depth': 9, 'min_child_weight': 4, 'n_estimators': 600, 'nthread': 4, 'objective': 'reg:linear', 'subsample': 0.7, 'tree_method': 'gpu_hist'}	0.06351258369358434

3) LIGHT BGM :

```
def find_best_hyperparameters(df,model):
    # Grid parameters for using in Gridsearch while tuning
    y_train = pd.DataFrame(df,columns = ['winPlacePerc'])
    X_train = df.drop(['winPlacePerc'], axis = 1)
    y_train = y_train.fillna(0)
    gridParams = {
        'learning_rate'         : [0.1 , 0.05 , 0.75],
        'n_estimators '         : [ 1200, 1700,2000,2400],
        'bagging_fraction'      : [0.7 , 0.8],
        'feature_fraction'      : [0.8,0.7],
        'num_leaves'            : [60,85, 110,140],
        "colsample_bytree"    :   [0.7,0.8],
        "objective" : ["regression"],
        "metric" : ["mae"],
        "num_threads" : [4],
        'device' : ['gpu'],
    }
    # Create the grid
    grid = GridSearchCV(model,
                        gridParams,
                        verbose=1,
                        cv=3)
    # Run the grid
    grid.fit(X_train, y_train)
    print('Best parameters: %s' % grid.best_params_)
    print('Accuracy: %.2f' % grid.best_score_)
    return
```

```
Best parameters: {'bagging_fraction': 0.8, 'colsample_bytree': 0.8, 'device': 'gpu','feature_fraction': 0.7,
'learning_rate': 0.1, 'metric': 'mae', 'n_estimators ': 2000, 'num_leaves': 140, 'num_threads': 4, 'objective': 'regression'}


Accuracy: 0.96
```

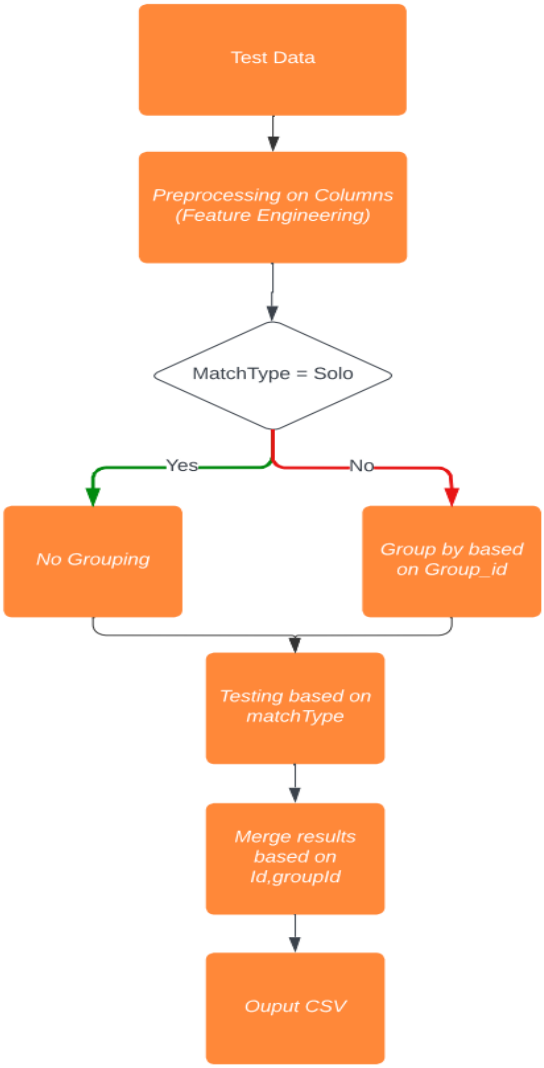
Benchmarking : For Light GBM Model with different parameters

Solo Model :

Parameters	Score
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'device': 'gpu', 'feature_fraction': 0.7, 'learning_rate': 0.05, 'metric': 'mae', 'n_estimators ': 2100, 'num_leaves': 130,}	0.043603202682344556
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'feature_fraction': 0.7, 'learning_rate': 0.05, 'metric': 'mae', 'n_estimators ': 2000, 'num_leaves': 120}	0.04365426207471159
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'device': 'gpu', 'feature_fraction': 0.7, 'learning_rate': 0.05, 'metric': 'mae', 'n_estimators ': 1900, 'num_leaves': 130,}	0.04359696933941155
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'device': 'gpu', 'feature_fraction': 0.7, 'learning_rate': 0.05, 'metric': 'mae', 'n_estimators ': 2100, 'num_leaves': 85}	0.04390603684533602
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'device': 'gpu', 'feature_fraction': 0.7, 'learning_rate': 0.08, 'metric': 'mae', 'n_estimators ': 2100, 'num_leaves': 130}	0.04304224999997862
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'device': 'gpu', 'feature_fraction': 0.7, 'learning_rate': 0.08, 'metric': 'mae', 'n_estimators ': 2100, 'num_leaves': 120,'}	0.0429668394808967
{'bagging_fraction': 0.7, 'colsample_bytree': 0.8, 'device': 'gpu', 'feature_fraction': 0.7, 'learning_rate': 0.08, 'metric': 'mae', 'n_estimators ': 2100, 'num_leaves': 85,'}	0.04317670666438045
{'n_estimators':2000,"num_leaves" : 85, "learning_rate" : 0.05, "bagging_fraction" : 0.7, "bagging_seed" : 0, "colsample_bytree" : 0.8,}	0.04211989189891919

TESTING :

We follow a similar flow even for testing , we perform feature engineering and then split the dataset into 3 parts based on match type and then we forward the dataset into the respective model.
Since we are predicting based on group id we will have to merge the predicted outcomes i.e (group_id , winPlacePerc) back to the format required i.e (Id , winPlacePerc)



RESULT

Below displays the score of the 3 grouped models using the Model Light BGM, XG Boost and Linear Regression along with using the same models without grouping. The percentage of increase in score is noted in the last column.

MODEL	SOLO	DUO	SQUAD	AVERAGE SCORE AFTER GROUPING AND SPLITTING	SCORE WITHOUT GROUPING AND SPLITTING	% INCREASE IN SCORE
light bgm	0.0421	0.0467	0.0633	0.0507	0.0573	**13.01%**
xg_boost	0.0423	0.0469	0.0634	0.0508	0.0574	**12.99%**
linear regression	0.0659	0.0924	0.1198	0.0927	0.0961	**3.66%**

It can be seen that XG Boost and Light-bgm almost perform similarly with a good score compared to Linear regression.

It can be Inferred that grouping based on Match ID values on the attribute GroupID has yielded a better result than training on a single model. There is a **13% increase** in score in XGBoost when training on different categories and different models rather than a single model.

Kaggle Contest link : *https://www.kaggle.com/competitions/pubg-version-3/overview*

LeaderBoard position on Kaggle: **1**
Score of Submitted Model : **0.00534**