A League of Legends Competitive Analysis

Name(s): Kevin Wong, Andrew Yang

Website Link: https://kev374k.github.io/league_prediction_models/

Notebook setup

In [1]:
```
import pandas as pd
import numpy as np
import os

import plotly.express as px
pd.options.plotting.backend = 'plotly'
```

In [2]:
```
import warnings
from sklearn.exceptions import ConvergenceWarning

# Suppress ConvergenceWarning
warnings.filterwarnings('ignore', category=ConvergenceWarning)
```

Introduction

League of Legends stands as a titan in esports. Arguably the most popular esport in the world, it requires skill, patient, and talent to master. Yet how do teams actually win?

Pro teams and players require a lot of studying in order to predict correctly. Identifying gaps in vision, making sure you kill all the minions, and practice all take a long time to understand and figure out — yet all are essential aspects for teams winning games in professional play. So, how can we make a model to predict what team or even player would win a game?

This prediction model delves into how we can better predict a League of Legends team or player winning each individual match. We got this data from the publicly available pro match data from Oracle's Elixir, and thank those who gave us the data to play with.

Framing the Problem

Problem Identification

One problem that capsulates the most important question in League of Legends is about which team is going to win? Each and every game has a singular answer, yet it is more complex than most people think.

There are a variety of factors that go into each and every game, which is why we analyzed a dataset of competitive games from the year 2022. In this data, each game was 12 lines long. 10 for the players (5 from each team), and 2 for the teams (where data was summarized).

Essentially, each line of data was an individual/team that was playing in a certain game, and each line had columns that gave stats for each player. This included important stats like Kills, Deaths, Assists, as well as others that only become more complicated as the game goes on.

In [3]:
```
# Using pd.read_csv() to parse the downloaded csv file
# Code reused from our League of Legends Competetive Analysis Project
df = pd.read_csv("2022_LoL_esports_match_data_from_OraclesElixir.csv")
df.head(12)
```

```
/Users/andrewyang/anaconda3/envs/dsc80/lib/python3.9/site-packages/IPython/core/interactiveshell.py:3550: DtypeWarning: Columns (2,76) have mixed types.Specify dtype option on import or set low_memory=False.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

Out[3]:

patch	...	opp_csat15	golddiffat15	xpdiffat15	csdiffat15	killsat15	assistsat15	deathsat15	opp_killsat15	opp_assistsat15	opp_deathsat15
12.01	...	121.0	391.0	345.0	14.0	0.0	1.0	0.0	0.0	1.0	0.0
12.01	...	100.0	541.0	-275.0	-11.0	2.0	3.0	2.0	0.0	5.0	1.0
12.01	...	119.0	-475.0	153.0	1.0	0.0	3.0	0.0	3.0	3.0	2.0
12.01	...	149.0	-793.0	-1343.0	-34.0	2.0	1.0	2.0	3.0	3.0	0.0
12.01	...	21.0	443.0	-497.0	7.0	1.0	2.0	2.0	0.0	6.0	2.0
12.01	...	135.0	-391.0	-345.0	-14.0	0.0	1.0	0.0	0.0	1.0	0.0
12.01	...	89.0	-541.0	275.0	11.0	0.0	5.0	1.0	2.0	3.0	2.0
12.01	...	120.0	475.0	-153.0	-1.0	3.0	3.0	2.0	0.0	3.0	0.0
12.01	...	115.0	793.0	1343.0	34.0	3.0	3.0	0.0	2.0	1.0	2.0
12.01	...	28.0	-443.0	497.0	-7.0	0.0	6.0	2.0	1.0	2.0	2.0
12.01	...	510.0	107.0	-1617.0	-23.0	5.0	10.0	6.0	6.0	18.0	5.0
12.01	...	487.0	-107.0	1617.0	23.0	6.0	18.0	5.0	5.0	10.0	6.0

To start, we did some data cleaning. We got rid of columns that we thought weren't incredibly important to our question (like the types of dragons, champion bans, game length, etc), and took columns that we thought would be important to deciding the winner of a specific match. Particularly, columns like `'teamname', 'position', 'kills', 'deaths', 'assists', 'firstblood'` were columns we thought we important for our overall prediction.

In [4]:
```
# Converts columns 'datacompleteness', 'playoffs', 'result' to Boolean types
# Filter columns that are needed for our hypothesis
# Code reused from our League of Legends Competetive Analysis Project

cleaned_data = df.assign(
    datacompleteness = df['datacompleteness'].apply(lambda x: True if x == "complete" else False),
    ).loc[
        :, ['result', 'datacompleteness', 'teamname', 'playoffs', 'side', 'position', 'kills', 'deaths', 'assists',
            'firstblood', 'doublekills', 'triplekills', 'quadrakills', 'pentakills', 'turretplates', 'towers',
            'dragons', 'barons', 'elders', 'heralds', 'inhibitors', 'dpm', 'cspm', 'wpm', 'vspm', 'earned gpm',
            'golddiffat15', 'xpdiffat15']
    ].fillna(0)
```

Additionally, in order to get more complete and overall better data for predictions, we trained our model only on completed data (where the column datacompleteness was True) because we observed that rows that weren't complete often had missing data.

In [5]:
```
# Filter for 'complete' rows
# Code reused from our League of Legends Competetive Analysis Project

cleaned_data = cleaned_data[cleaned_data['datacompleteness'] == True]
cleaned_data['teamname'] = cleaned_data['teamname'].astype(str)
cleaned_data.head(12)
```

Out[5]:

	result	datacompleteness	teamname	playoffs	side	position	kills	deaths	assists	firstblood	...	elders	heralds	inhibitors	dpm
0	0	True	Fredit BRION Challengers	0	Blue	top	2	3	2	0.0	...	0.0	0.0	0.0	552.2942
1	0	True	Fredit BRION Challengers	0	Blue	jng	2	5	6	1.0	...	0.0	0.0	0.0	412.0841
2	0	True	Fredit BRION Challengers	0	Blue	mid	2	2	3	0.0	...	0.0	0.0	0.0	499.4046
3	0	True	Fredit BRION Challengers	0	Blue	bot	2	4	2	1.0	...	0.0	0.0	0.0	389.0018
4	0	True	Fredit BRION Challengers	0	Blue	sup	1	5	6	1.0	...	0.0	0.0	0.0	128.3012
5	1	True	Nongshim RedForce Challengers	0	Red	top	1	1	12	0.0	...	0.0	0.0	0.0	611.3835
6	1	True	Nongshim RedForce Challengers	0	Red	jng	4	1	10	0.0	...	0.0	0.0	1.0	370.0175
7	1	True	Nongshim RedForce Challengers	0	Red	mid	6	3	12	0.0	...	0.0	0.0	0.0	724.6935
8	1	True	Nongshim RedForce Challengers	0	Red	bot	8	2	10	0.0	...	0.0	0.0	0.0	934.7461
9	1	True	Nongshim RedForce Challengers	0	Red	sup	0	2	18	0.0	...	0.0	0.0	0.0	158.1786
10	0	True	Fredit BRION Challengers	0	Blue	team	9	19	19	1.0	...	0.0	2.0	0.0	1981.0858
11	1	True	Nongshim RedForce Challengers	0	Red	team	19	9	62	0.0	...	0.0	0.0	1.0	2799.0193

12 rows × 28 columns

Since winning a game has a value of either True or False (1s and 0s), we decided that this was a classification problem, a binary classification. We are trying to predict the response variable, `result`, as either True or False scores, since it's the most important variable to success in League of Legends.

On the other hand, the metric we are using the evaluate our model is accuracy, because we want to determine how well our model can predict if a team wins or loses. We wanted to use accuracy rather than something like an F1-score because there wasn't really a larger significance to having false positives or false negatives as compared to the true positives/negatives. There's an equal amount of wins and losses (because a team either has to win or lose each game), and by treating both false negatives and false positives as equally important, we can make sure that accuracy makes the most sense for our predictin.

Finally, during our "time of prediction", we could only use data from the set given to us, so there were no outside sources/biases other than our downloaded data. This also means that our predictions have no influence from previous/future seasons, meaning this is an isolated dataset that we can use to predict on.

Baseline Model

Baseline Model Features

For our baseline model, we wanted to create something that would be able to predict wins based on only what we deemed were the most important individual player stats that happen to be available through almost every game that is played in League of Legends.

Quantitative Features: Kills, Deaths, and Assists (commonly known collectively as KDA)

Response Variable: Result (Win (1) / Lose (0))

For our encoding of these variables, since they were already discrete values (integers), they could easily be placed into a Pipeline and into our model to train and predict data.

In [6]:
```
# Importing necessary packages

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler, FunctionTransformer
from sklearn.compose import ColumnTransformer
from sklearn.tree import DecisionTreeClassifier
```

In [7]:
```
# Performing train test split to evaluate our model's performance on unseen data

train_X, test_X, train_y, test_y = train_test_split(cleaned_data, cleaned_data['result'])
```

In [8]:
```
# Creating the pipeline

# Helper function representing an identity function
def return_self(df):
    return df

# Column transformer to select only wanted features for the baseline model
select_ct = ColumnTransformer(
    transformers = [
        ('kills', FunctionTransformer(return_self), ['kills']),
        ('deaths', FunctionTransformer(return_self), ['deaths']),
        ('assists', FunctionTransformer(return_self), ['assists']),
    ],
    remainder='drop'
)

# Pipeline that selects the wanted features and then fit them to a decision tree classifier
baseline_pl = Pipeline([
    ('select_columns', select_ct),
    ('decision_tree_classifier', DecisionTreeClassifier(max_depth = 3)),
])
```

In [9]:
```
# Fitting training data on the baseline model

baseline_pl.fit(train_X, train_y)
```

Out[9]:
```
Pipeline(steps=[('select_columns',
                 ColumnTransformer(transformers=[('kills',
                                                  FunctionTransformer(func=<function return_self at 0x16bd76280>),
                                                  ['kills']),
                                                 ('deaths',
                                                  FunctionTransformer(func=<function return_self at 0x16bd76280>),
                                                  ['deaths']),
                                                 ('assists',
                                                  FunctionTransformer(func=<function return_self at 0x16bd76280>),
                                                  ['assists'])])),
                ('decision_tree_classifier',
                 DecisionTreeClassifier(max_depth=3))])
```

Baseline Performance

For our model, we decided that a DecisionTreeClassifier would make the most sense for this simple prediction, while having a maximum depth of 3, because we presumed that a simpler model shouldn't be as complicated, only allowing for more simple and therefore straighter results.

In [10]:
```
# Examine score on training data

baseline_train_performance = baseline_pl.score(train_X, train_y)
baseline_train_performance
```

Out[10]:
```
0.8297282831145115
```

In [11]:
```
# Examine score on testing data

baseline_test_performance = baseline_pl.score(test_X, test_y)
baseline_test_performance
```

Out[11]:
```
0.8294134315670162
```

With this data in mind, we received an accuracy of approximately ~83% for both training data and testing data. Training and testing score being similar indicates that our model has not overfitted on the training data and can work just as well on unseen data. We think this is a good baseline model that is decently accurately — it also is simple and logical in the fact that it is more likely for a team to win when players individually do well in aspects like Kills and Assists (more kills and more assists equal more gold and time) while also lowering eaths (Deaths are bad, because they give the enemy team gold and time to get objectives). Overall, we were satisifed with this baseline models' performance because of how well it did with so little to train on.

Final Model

New Features

New Features Added to our Baseline Model: Position, Teamname, Dragons, Barons, Heralds, Turret Plates, Towers, Inhibitors, First Blood, Double Kills, Triple Kills, Earned GPM, golddiffat15, xpdiffat15, DPM, CSPM, and VSPM.

Why Position and Teamname Matter

As strong observes of esports ourselves, we obviously noticed a pattern as to how some teams did versus others in competitive matches. Instinctively, we understand that only one team will win Worlds, which also means that they are likely better than many other teams. For teams like T1 and DRX (both of which were in the Worlds Final), it would make sense that they would be predicted to win any given match.

On the other hand, we also postured that position matters a lot, because different positions prioritize different objectives. For instance, a position like a support is more likely to prioritize assists and helping their team with objectives, while a position like top more values kills, because for most of the early game, they are separated from the rest of their team, meaning we need to treat different positions in a different matter.

Why Objectives Matter

Objectives like Dragons, Barons, Heralds, Turret Plates, Towers, and Inhibitors are all objectives that give teams gold and/or buffs that either help in teamfights or by taking other objectives, both of which are primary objectives and goals in League of Legends. These provide champions gold (to buy items and empower themselves), while also potentially giving them boosts in stats (i.e. Killing the Infernal Drake gives players stacking attack damage/ability power), meaning they can scale themselves and be more likely to win in future teamfights.

Why Kills, Gold, and XP Matter

For features like First Blood, Double Kills, Triple Kills, Earned GPM, golddiffat15, and xpdiffat15, we wanted to emphasis why gold and kills were both important.

Killing also gives advantage to certain players and teams that gain the advantage. For instance, First Blood enables the killer to gain 100 gold, while the person who assisted them also gains 50 gold. Additionally, this allows the player who killed the other to gain an XP advantage, which is large, especially in the laning phase of League.

To add on, gold is one of the most important factors in winning in League of Legends. Gold enables the usage of the shop, as well as purchasing stronger items, vision-related items like wards, speed-related items like boots, and more. These are all large parts as to why gold is extremely important, because it allows champions to snowball. Similarly, XP also allows champions to gain levels and with that, new and stronger abilities that allow both solo fighting and team fighting to be stronger.

Why Damage, CS, and Vision Score Matter

Damage, CS and Vision Score are all amplifies that help us understand why certain teams do better. Damage, of course, is self-explanatory; the more damage you do, the more likely you are to win games. Similarly, CS (killing minions) allows you to generate more gold, so more CS theoretically means a larger gold advantage. Finally, for Vision Score, although it is not felt as much compared to a stat like Gold, it measures how well a team can see the map and identify their opponents within the Fog of War (areas where you can't normally see the opponent). Having a higher vision score naturally means that a team can see the other team and find weaknesses in pathing and ganking, allowing for stronger teamfights and objective taking, all of which are found earlier.

Hyperparameters

To test whether these features are important or redundant, we examine these features and make sure that these are the right ones to choose. To do so, we graphed them out and observed the difference between winning and losing teams/players. During our testing for hyperparamters, we also observed features that we originally thought we important to the model that were later excluded because we realized they provided little value. One example is the column `wpm`, or Wards Per Minute. Originally, we thought wards per minute would be important, because vision is such a high priority in pro play. However, when we observed the graph below, we realized that both teams that lost and won typically placed an equal amount of wards, and that it didn't ultimately play a significant role in our model.

In [12]:
```
# Here, we wanted to test columns that relied on team stats;
# things like dragons, heralds, barons, all that boost team stats and are typically team objectives
# so we seperated team rows from player rows in the data

single_player_data = cleaned_data[cleaned_data['position'] != 'team']
team_data = cleaned_data[cleaned_data['position'] == 'team']
```

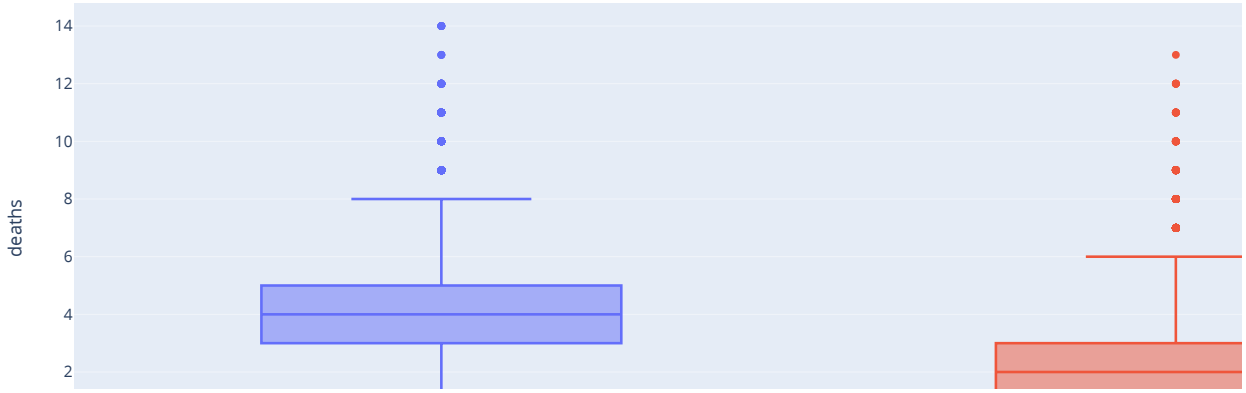
Player-Result Relationships

During our process, we identified what we thought were more solo relational features; these included things like CSPM, Kills, Deaths, Assists, and more, all of which we tested on filtered data that only included players (not teams' general stat summary)

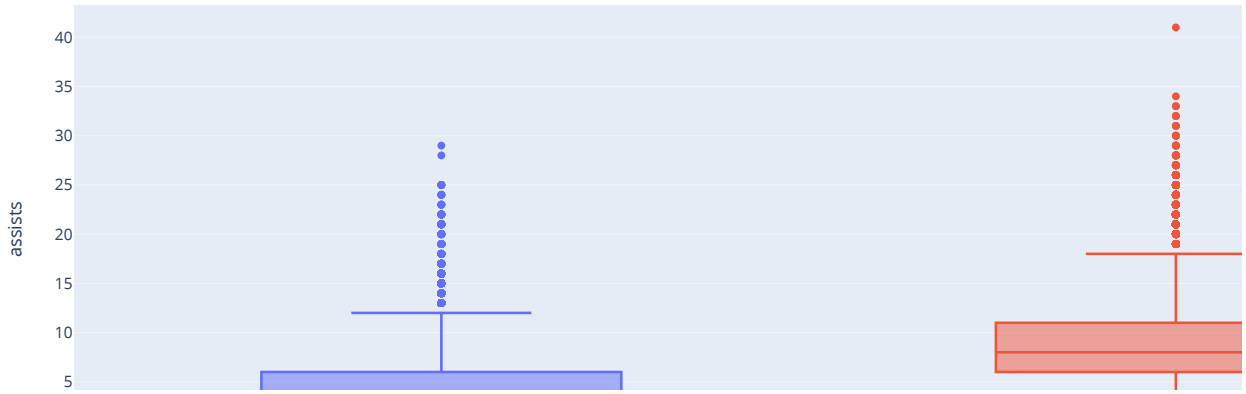
In [13]:
```
# Graphing single-player feature relationships
for col in ['kills', 'deaths', 'assists', 'doublekills', 'triplekills', 'firstblood', 'dpm', 'cspm', 'wpm', 'vspm',
            'earned gpm', 'golddiffat15', 'xpdiffat15']:
    fig = px.box(single_player_data[[col, 'result']], x = 'result', y = col, color = 'result',
                 title = f'Player Relationship Between {col} and Result')
    fig.update_layout(colorway = ['red', 'blue'])
    fig.show()
#     fig.write_html(f'{col}_result_player.html', include_plotlyjs = 'cdn')
```

Player Relationship Between kills and Result

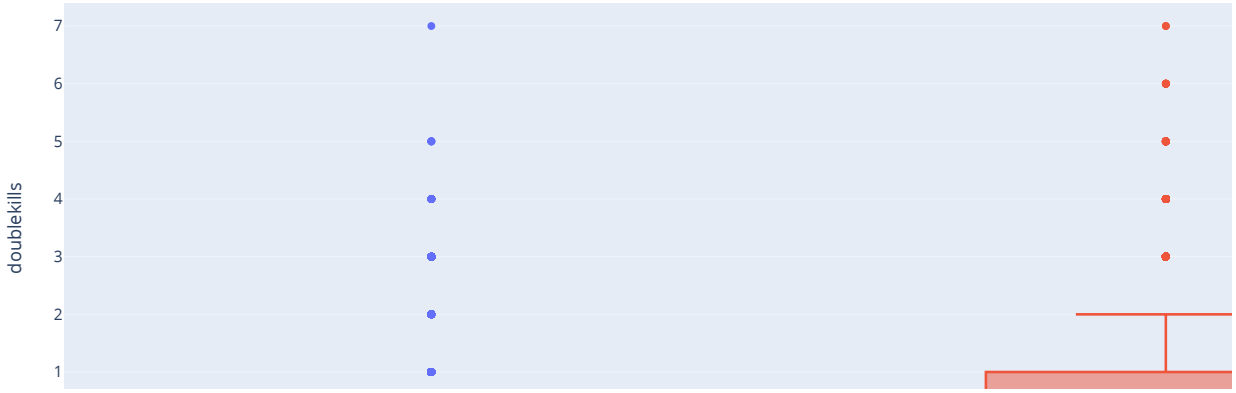
Player Relationship Between deaths and Result



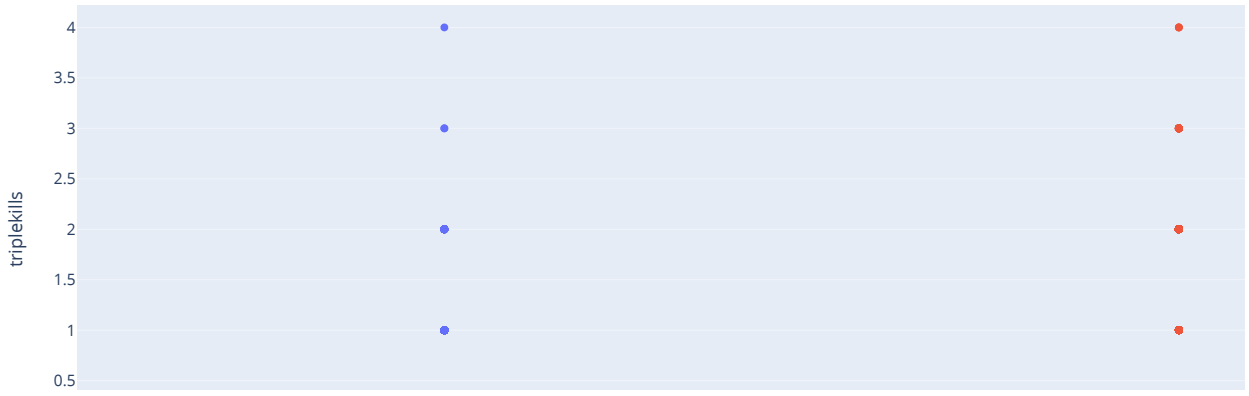
Player Relationship Between assists and Result



Player Relationship Between doublekills and Result

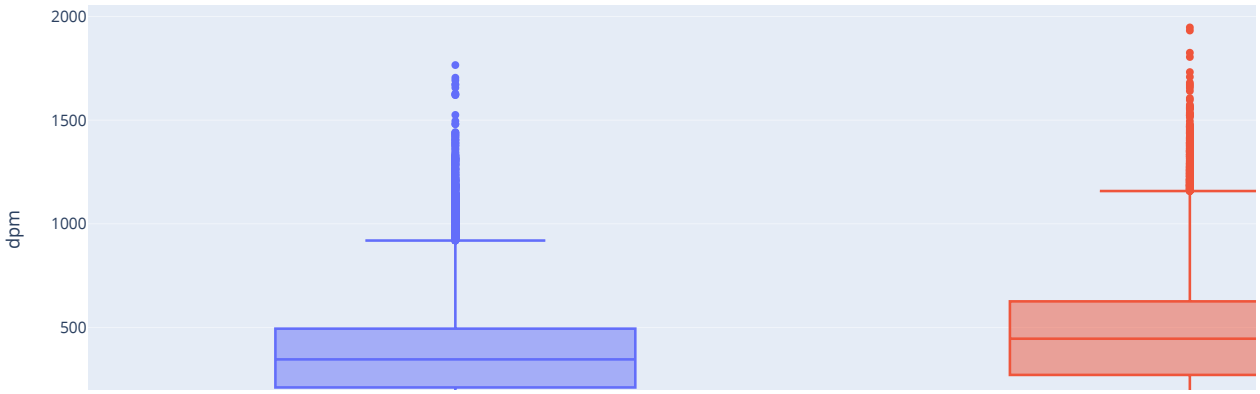


Player Relationship Between triplekills and Result

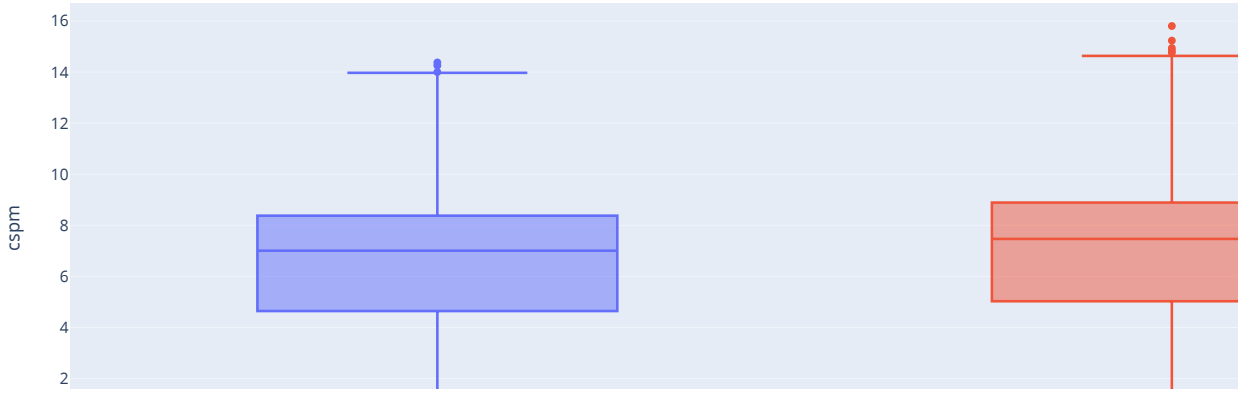


Player Relationship Between firstblood and Result

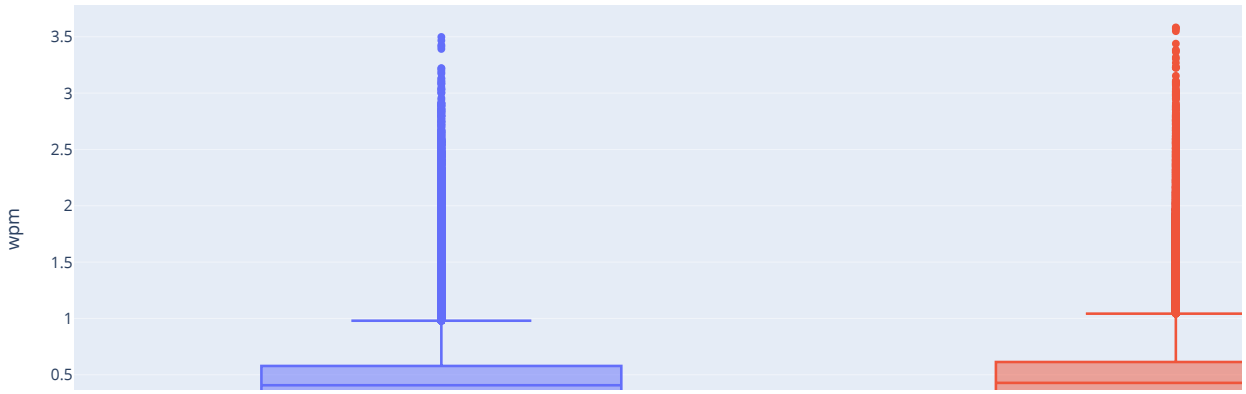
Player Relationship Between dpm and Result



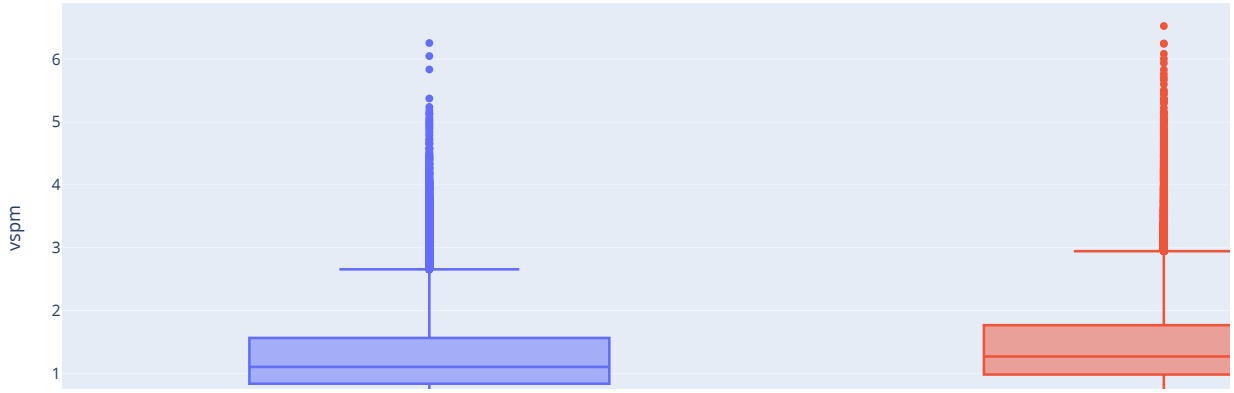
Player Relationship Between cspm and Result



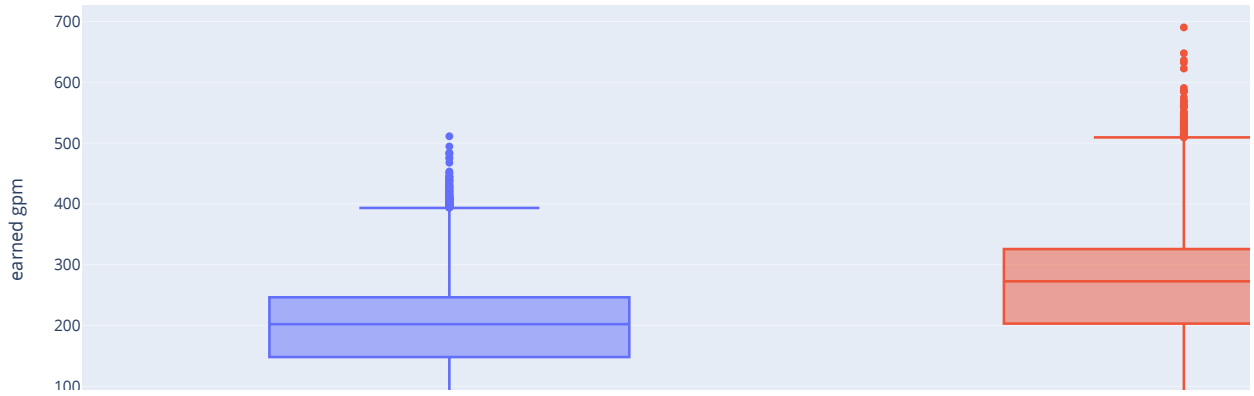
Player Relationship Between wpm and Result



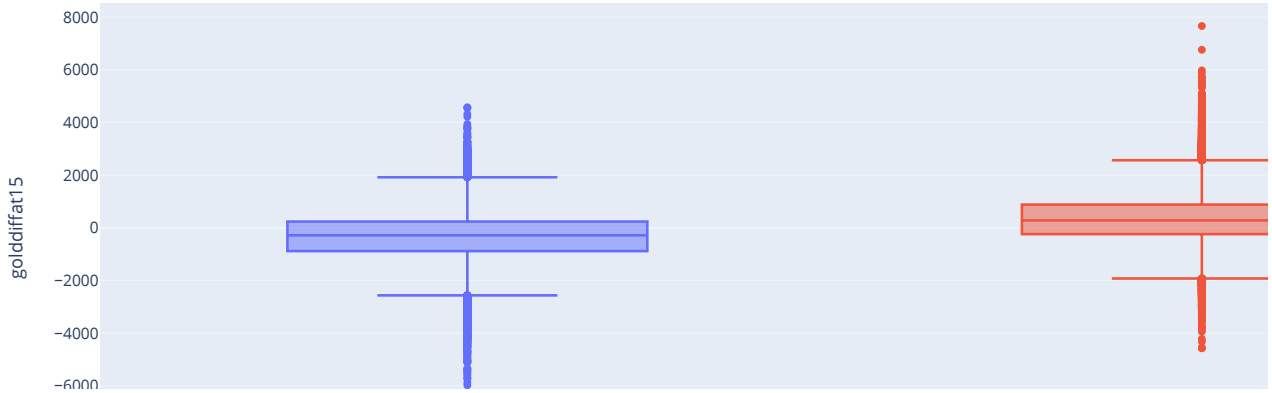
Player Relationship Between vspm and Result

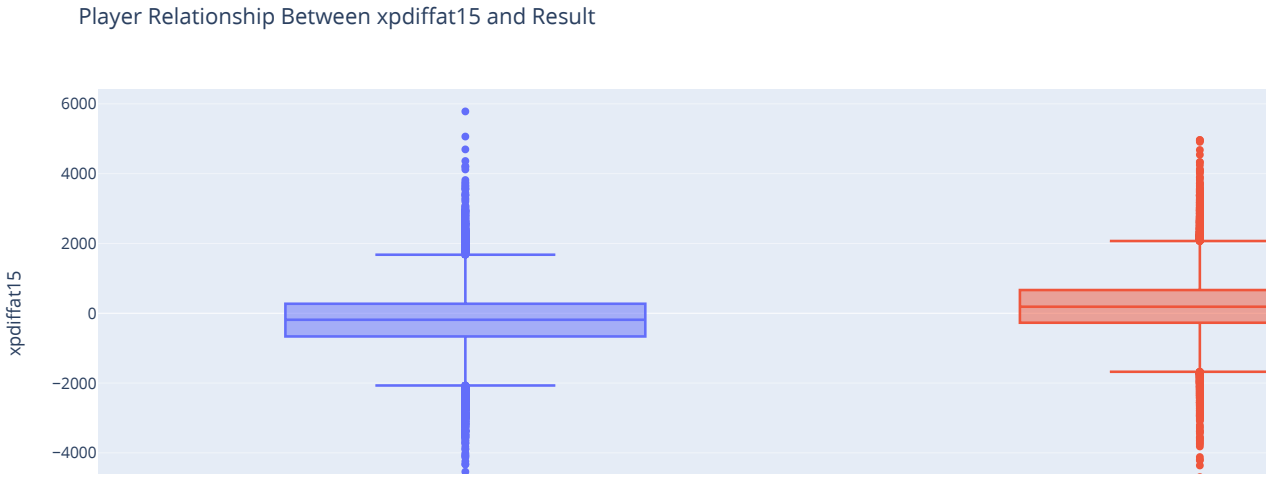


Player Relationship Between earned gpm and Result



Player Relationship Between golddiffat15 and Result

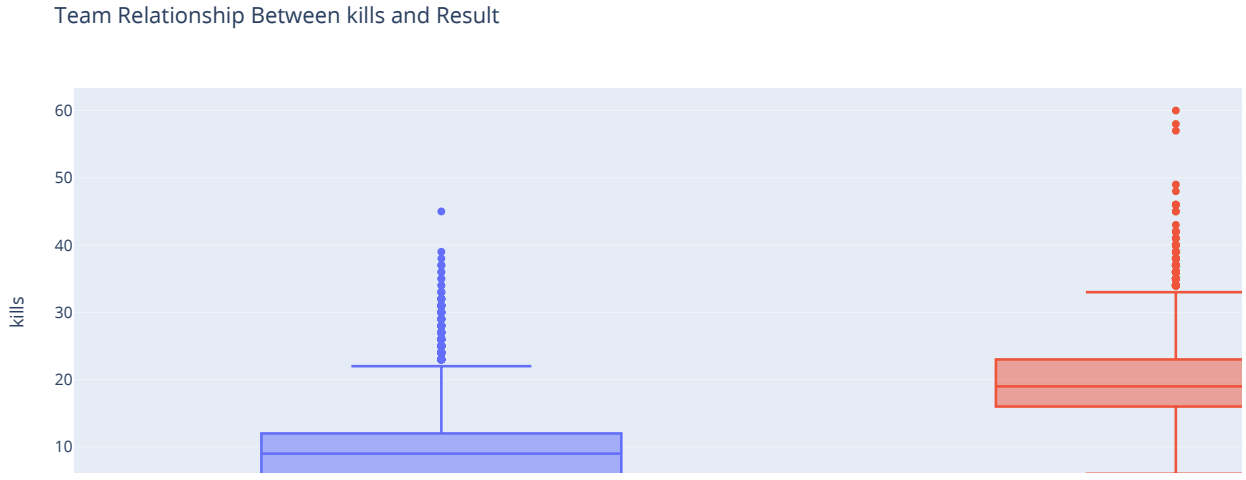




Team-Result Relationships

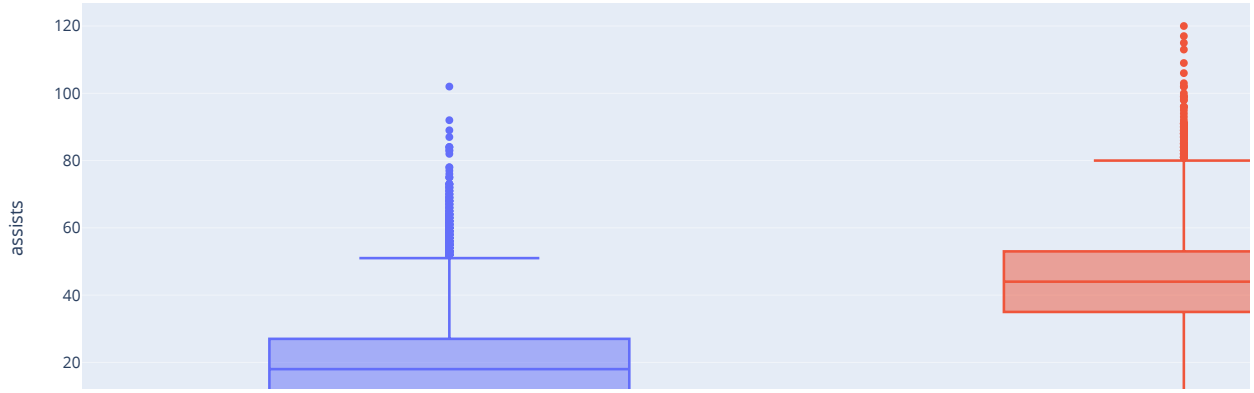
On the other hand, we then observed those that we observed were more team-reliant features, like barons, dragons, heralds, and more, which were then tested on a filtered dataframe that only had the teams' summary stats.

In [14]:
```
# Graphing team feature relationships
for col in ['kills', 'deaths', 'assists', 'doublekills', 'triplekills', 'quadrakills', 'pentakills',
            'turretplates', 'towers', 'inhibitors', 'dragons', 'barons', 'elders', 'heralds', 'cspm',
            'wpm', 'vspm']:
    fig = px.box(team_data[[col, 'result']], x = 'result', y = col, color = 'result',
                 title = f'Team Relationship Between {col} and Result')
    fig.show()
#     fig.write_html(f'{col}_result_relationship.html', include_plotlyjs = 'cdn')
```



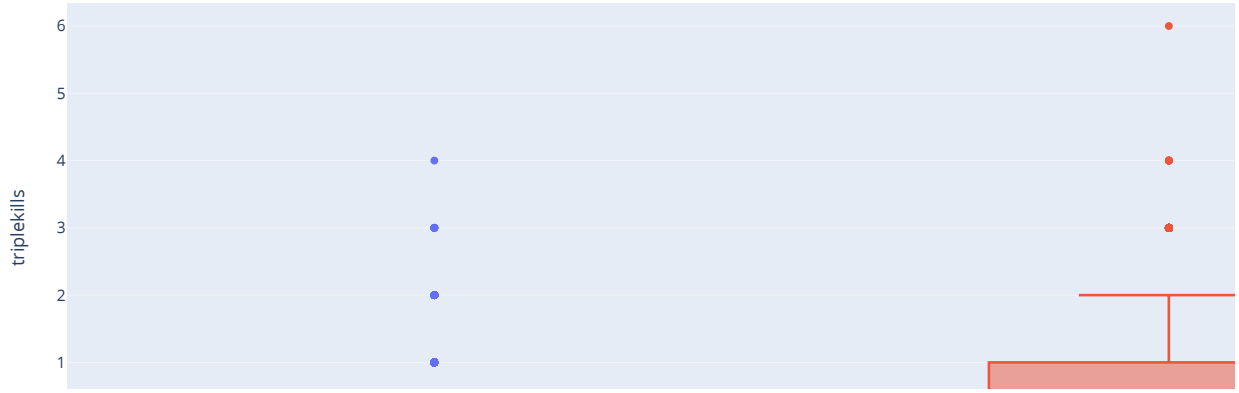
Team Relationship Between deaths and Result

Team Relationship Between assists and Result

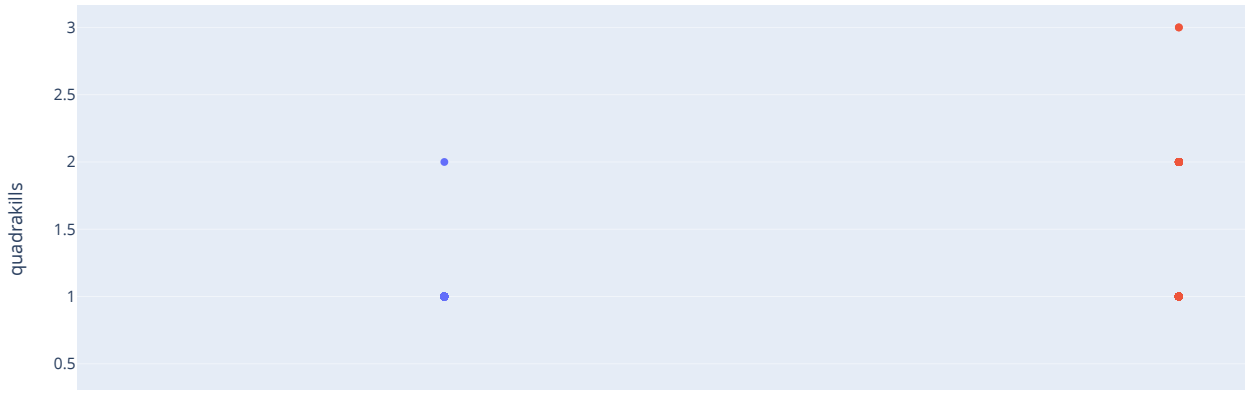


Team Relationship Between doublekills and Result

Team Relationship Between triplekills and Result

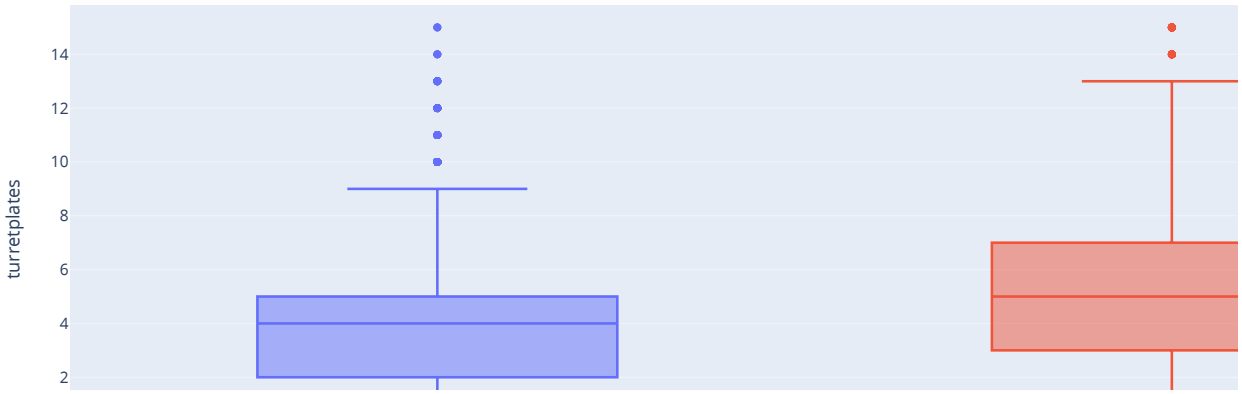


Team Relationship Between quadrakills and Result

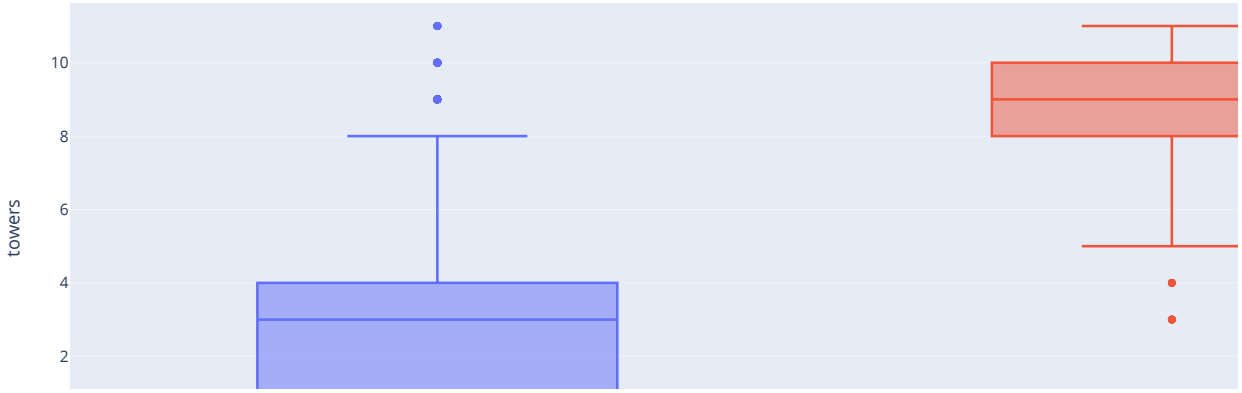


Team Relationship Between pentakills and Result

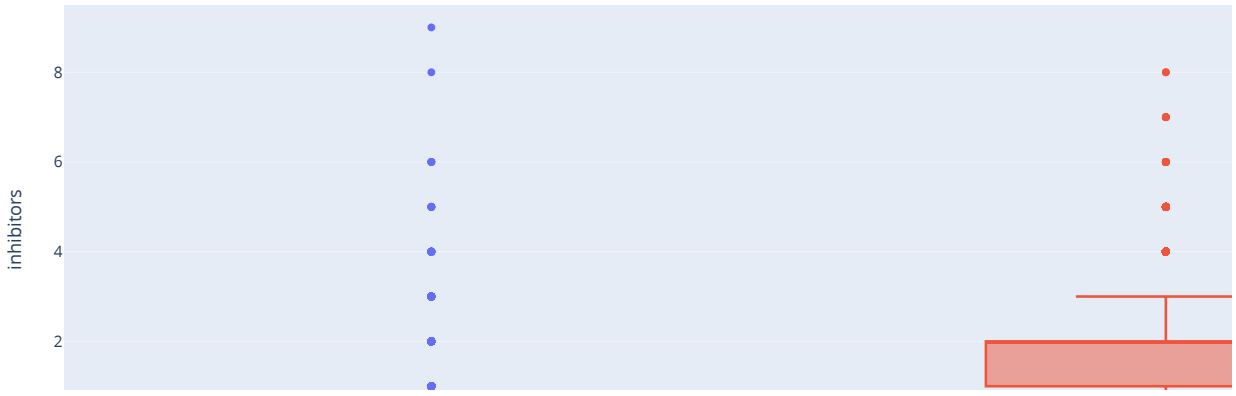
Team Relationship Between turretplates and Result



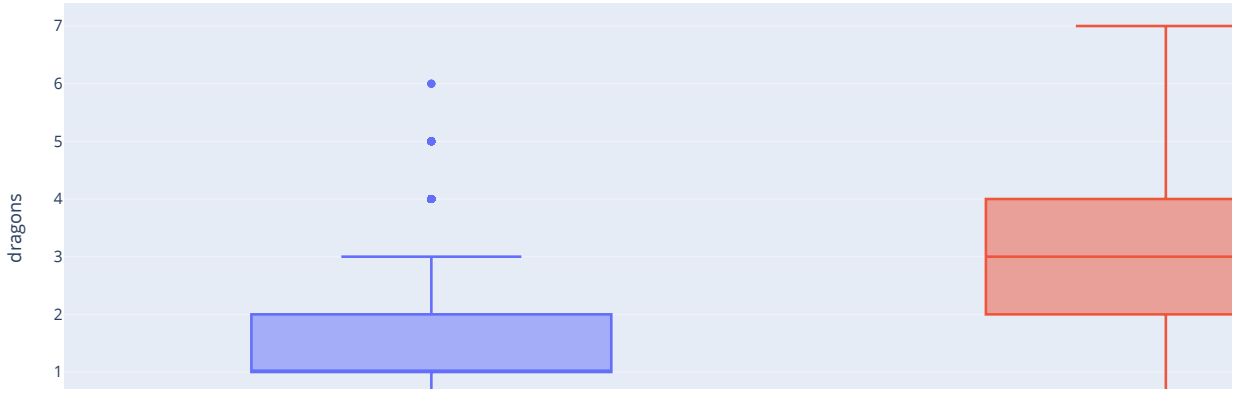
Team Relationship Between towers and Result



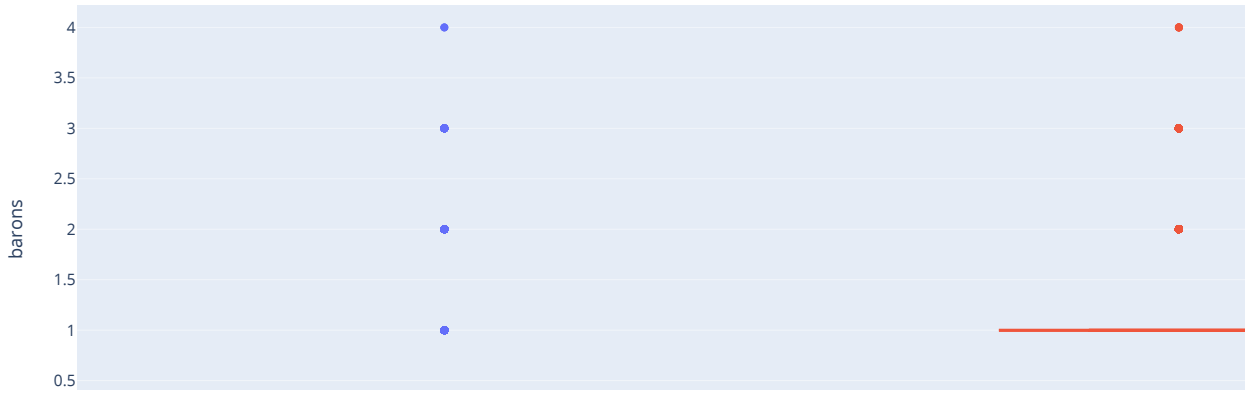
Team Relationship Between inhibitors and Result



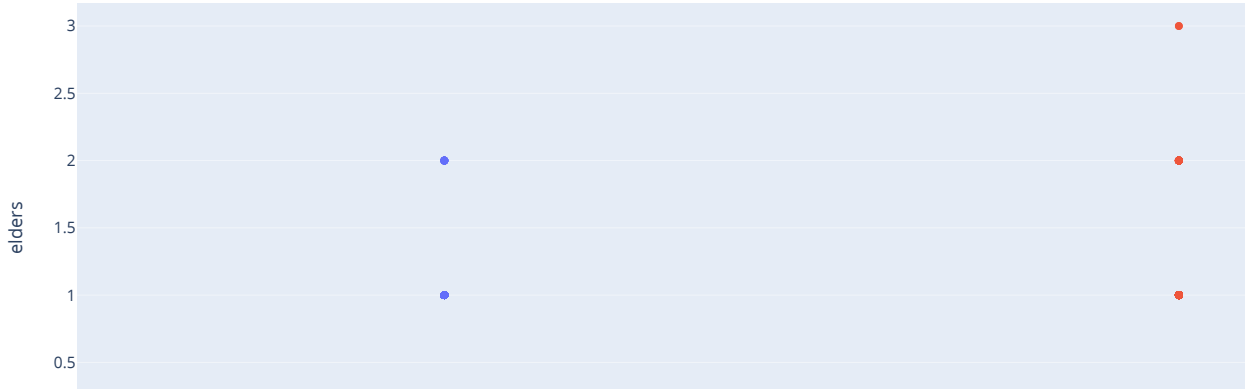
Team Relationship Between dragons and Result



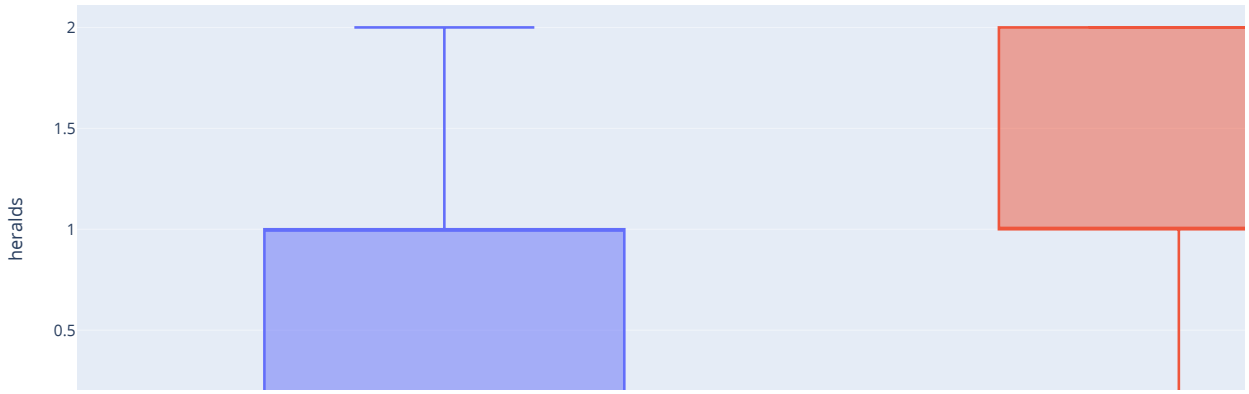
Team Relationship Between barons and Result



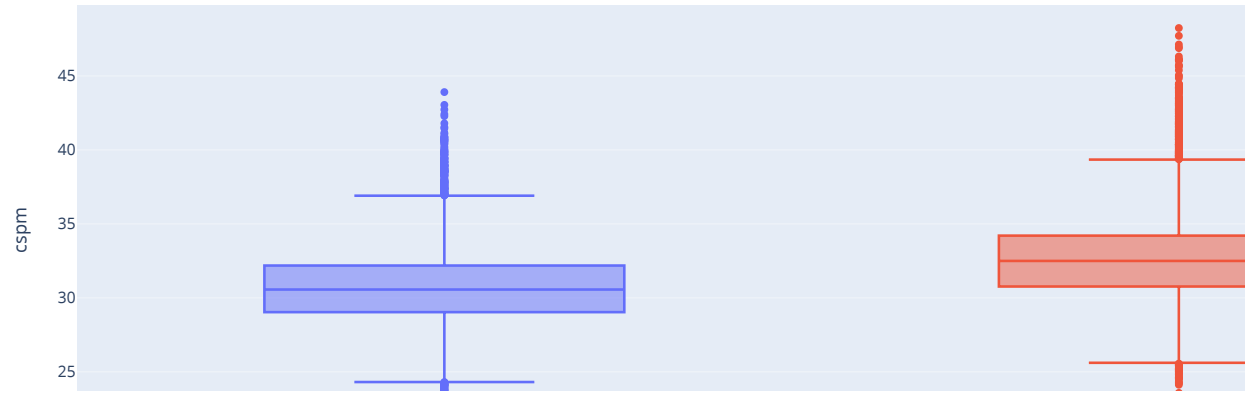
Team Relationship Between elders and Result

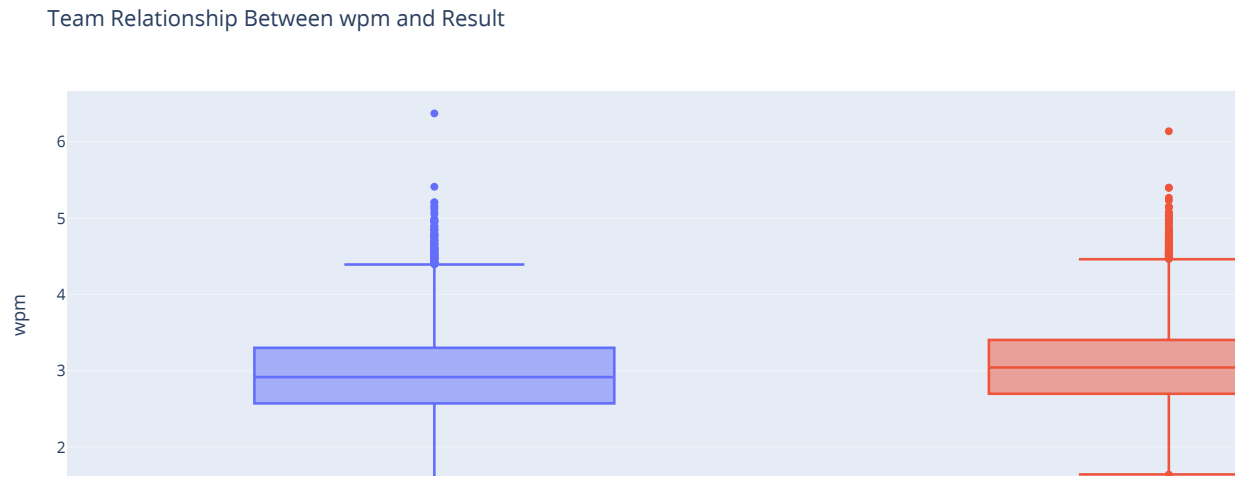


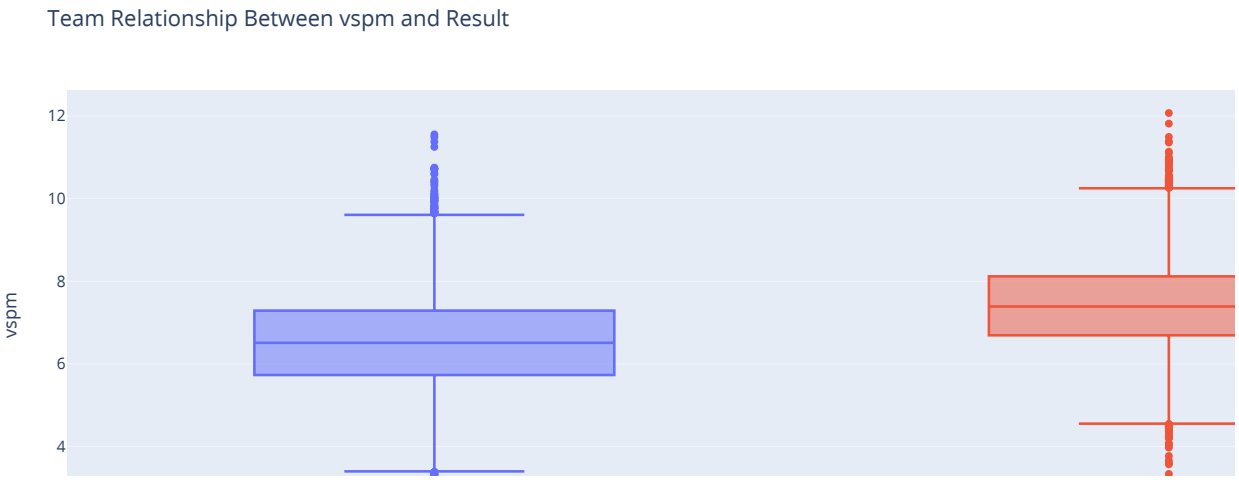
Team Relationship Between heralds and Result



Team Relationship Between cspm and Result







Summary of feature analysis

From those graphs, we can infer what are likely important hyperparamters for this model:

Player Relationships - Kills, Deaths, Assists, Doublekills, FirstBloodKill, dpm, cspm, vspm, gpm, golddiffat15, xpdiffat15

Team Relationships - Kills, Deaths, Assists, Doublekills, Triplekills, Turretplates, Towers, Inhibitors, Dragons, Barons, Heralds, cspm, vspm,

Model Description

Ultimately, we decided to make a model based on either the DecisionTreeClassifier or LogisticRegression. First, we tested different LogisticRegression models and different DecisionTreeClassifier models by hand. The best LogisticRegression had an accuracy of ~91.5%. The best DecisionTreeClassifier had an accuracy of ~90.5%. Using this, we decided that our final model would be a LogisticRegression. We used a GridSearchCV with cross-validation to determine the best 'max_iter' and 'penalty' to use. This turned out to be a max iteration of 300 and using the 'l2' penalty. After training with the entire training data, we arrived at the final model with an **~91.5% accuracy**. Overall, we were satisfied with this accuracy while adding what we deemed as good features to our model. Our optimization of the hyperparameters and features helped us create a model that can predict a match a very high percentage of the time.

Quantitative Varaibles Our discrete variables included `'kills', 'deaths', 'assists', 'dragons', 'barons', 'heralds', 'firstblood', 'doublekills', 'triplekills', 'dpm', 'cspm', 'vspm', 'earned gpm', 'turretplates', 'towers', and 'inhibitors'`. For these, we standardized `'dpm', 'cspm', 'vspm', 'golddiffat15', and 'xpdiffat15'`, because we deemed them to show a fluctuation and difference especially in regards to explaining how much better a player/team does compared to others.

Qualitative Variables For our qualitative varibales, we chose `'position', and 'teamname'`. We then used the OneHotEncoder in order to convert them into quantitative columns, because they were both ordinal data types.

In [15]:
```
# Testing Logistic Regression
column_transformer = ColumnTransformer([
    ('kda', FunctionTransformer(), ['kills', 'deaths', 'assists']),
    ('team_position', OneHotEncoder(), ['position', 'teamname']),
    ('big_monsters', FunctionTransformer(), ['dragons', 'barons', 'heralds']),
    ('multikills', FunctionTransformer(), ['firstblood', 'doublekills', 'triplekills']),
    ('pm_stats', StandardScaler(), ['dpm', 'cspm', 'vspm', 'earned gpm']),
    ('gold_xp_diff', StandardScaler(), ['golddiffat15', 'xpdiffat15']),
    ('tower_related', FunctionTransformer(), ['turretplates', 'towers', 'inhibitors'])
])

log_pipeline = Pipeline([
    ('columns', column_transformer),
    ('log-reg', LogisticRegression(max_iter=1000)),
])
```

In [16]:
```
# Finding testing score for the LogisticalRegression model

log_pipeline.fit(train_X, train_y)
log_pipeline.score(test_X, test_y)
```

Out[16]:
```
0.9146122603192595
```

In [17]:
```
# Testing RandomForest
from sklearn.ensemble import RandomForestClassifier

column_transformer = ColumnTransformer([
    ('kda', FunctionTransformer(return_self), ['kills', 'deaths', 'assists']),
    ('team_position', OneHotEncoder(), ['position', 'teamname']),
    ('big_monsters', FunctionTransformer(return_self), ['dragons', 'barons', 'heralds']),
    ('multikills', FunctionTransformer(return_self), ['doublekills', 'triplekills']),
    ('pm_stats', StandardScaler(), ['dpm', 'cspm', 'vspm', 'earned gpm']),
    ('gold_xp_diff', StandardScaler(), ['golddiffat15', 'xpdiffat15']),
    ('tower_related', FunctionTransformer(return_self), ['turretplates', 'towers', 'inhibitors'])
])

decision_tree_pipeline = Pipeline([
    ('columns', column_transformer),
    ('decision_tree', DecisionTreeClassifier(max_depth = 10, min_samples_leaf = 1))
])
```

In [18]:
```
# Finding testing score for the DecisionTree model

decision_tree_pipeline.fit(train_X, train_y)
decision_tree_pipeline.score(test_X, test_y)
```

Out[18]:
```
0.9003494852177198
```

Our logistical regression model performed better from initial testing (0.915 compared to 0.900). So for the final model we will perform a GridSearchCV to determine the best 'max_iter' and 'penalty' to use.

In [19]:
```
# GridSearchCV using combinations of penalty and max_iter, using the accuracy as the scoring metric
# while incorporating a 5-fold cross-validation

from sklearn.model_selection import GridSearchCV

searcher = GridSearchCV(
    log_pipeline,
    param_grid = {
        'log-reg__penalty': ['l2', 'none'],
        'log-reg__max_iter': [100, 200, 300]
    },
    scoring = 'accuracy',
    cv = 5,
)
```

In [20]:
```
# Fitting the GridSearchCV on training data

searcher.fit(train_X, train_y)
```

Out[20]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('columns',
                                        ColumnTransformer(transformers=[('kda',
                                                                         FunctionTransformer(),
                                                                         ['kills',
                                                                          'deaths',
                                                                          'assists']),
                                                                        ('team_position',
                                                                         OneHotEncoder(),
                                                                         ['position',
                                                                          'teamname']),
                                                                        ('big_monsters',
                                                                         FunctionTransformer(),
                                                                         ['dragons',
                                                                          'barons',
                                                                          'heralds']),
                                                                        ('multikills',
                                                                         FunctionTransformer(),
                                                                         ['firstblood',
                                                                          'doublekills',
                                                                          'triplekills...
                                                                        ('pm_stats',
                                                                         StandardScaler(),
                                                                         ['dpm',
                                                                          'cspm',
                                                                          'vspm',
                                                                          'earned '
                                                                          'gpm']),
                                                                        ('gold_xp_diff',
                                                                         StandardScaler(),
                                                                         ['golddiffat15',
                                                                          'xpdiffat15']),
                                                                        ('tower_related',
                                                                         FunctionTransformer(),
                                                                         ['turretplates',
                                                                          'towers',
                                                                          'inhibitors'])])),
                                       ('log-reg',
                                        LogisticRegression(max_iter=1000))]),
             param_grid={'log-reg__max_iter': [100, 200, 300],
                         'log-reg__penalty': ['l2', 'none']},
             scoring='accuracy')
```

In [21]:
```
# Storing the best parameters to train the final model

best_params = searcher.best_params_
best_params
```

Out[21]:
```
{'log-reg__max_iter': 300, 'log-reg__penalty': 'l2'}
```

In [22]:
```
# Fitting the final model using the best parameters

log_pipeline.set_params(**best_params)
log_pipeline.fit(train_X, train_y)
```

Out[22]:
```
Pipeline(steps=[('columns',
                 ColumnTransformer(transformers=[('kda', FunctionTransformer(),
                                                  ['kills', 'deaths',
                                                   'assists']),
                                                 ('team_position',
                                                  OneHotEncoder(),
                                                  ['position', 'teamname']),
                                                 ('big_monsters',
                                                  FunctionTransformer(),
                                                  ['dragons', 'barons',
                                                   'heralds']),
                                                 ('multikills',
                                                  FunctionTransformer(),
                                                  ['firstblood', 'doublekills',
                                                   'triplekills']),
                                                 ('pm_stats', StandardScaler(),
                                                  ['dpm', 'cspm', 'vspm',
                                                   'earned gpm']),
                                                 ('gold_xp_diff',
                                                  StandardScaler(),
                                                  ['golddiffat15',
                                                   'xpdiffat15']),
                                                 ('tower_related',
                                                  FunctionTransformer(),
                                                  ['turretplates', 'towers',
                                                   'inhibitors'])])),
                ('log-reg', LogisticRegression(max_iter=300))])
```

In [23]:
```
# Finding training score for the final model

log_pipeline.score(train_X, train_y)
```

Out[23]:
```
0.9198807762140151
```

In [24]:
```
# Finding testing score for the final model

log_pipeline.score(test_X, test_y)
```

Out[24]:
```
0.9143918642360127
```

Training and testing score are similar, indicating our final model is not overfit and can generalize to unseen data.

Fairness Analysis

Choice of group for fairness analysis

Do games with a high number of kills have different predictions from the games with less kills?

In [25]:
```
# To be able to split the data into high number of kills vs low number of kills,
# we need to be able to group by individual games
# Thus we need to clean the original data again, this time keeping the 'gameid' column

# Same code from the Framing the Problem section, adding 'gameid' to the list of desired columns

fairness_data = df.assign(
    datacompleteness = df['datacompleteness'].apply(lambda x: True if x == "complete" else False),
    ).loc[
        :, ['gameid', 'result', 'datacompleteness', 'teamname', 'playoffs', 'side', 'position', 'kills',
            'deaths', 'assists', 'firstblood', 'doublekills', 'triplekills', 'quadrakills', 'pentakills',
            'turretplates', 'towers', 'dragons', 'barons', 'elders', 'heralds', 'inhibitors', 'dpm', 'cspm',
            'wpm', 'vspm', 'earned gpm', 'golddiffat15', 'xpdiffat15']
    ].fillna(0)

fairness_data = fairness_data[fairness_data['datacompleteness'] == True]
fairness_data['teamname'] = fairness_data['teamname'].astype(str)
fairness_data.head(12)
```

Out[25]:

	gameid	result	datacompleteness	teamname	playoffs	side	position	kills	deaths	assists	...	elders	heralds	inhib
0	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	top	2	3	2	...	0.0	0.0	
1	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	jng	2	5	6	...	0.0	0.0	
2	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	mid	2	2	3	...	0.0	0.0	
3	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	bot	2	4	2	...	0.0	0.0	
4	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	sup	1	5	6	...	0.0	0.0	
5	ESPORTSTMNT01_2690210	1	True	Nongshim RedForce Challengers	0	Red	top	1	1	12	...	0.0	0.0	
6	ESPORTSTMNT01_2690210	1	True	Nongshim RedForce Challengers	0	Red	jng	4	1	10	...	0.0	0.0	
7	ESPORTSTMNT01_2690210	1	True	Nongshim RedForce Challengers	0	Red	mid	6	3	12	...	0.0	0.0	
8	ESPORTSTMNT01_2690210	1	True	Nongshim RedForce Challengers	0	Red	bot	8	2	10	...	0.0	0.0	
9	ESPORTSTMNT01_2690210	1	True	Nongshim RedForce Challengers	0	Red	sup	0	2	18	...	0.0	0.0	
10	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	team	9	19	19	...	0.0	2.0	
11	ESPORTSTMNT01_2690210	1	True	Nongshim RedForce Challengers	0	Red	team	19	9	62	...	0.0	0.0	

12 rows × 29 columns

Operationalized groups

The mean number of kills per game is around 29. Thus, we define high number of kills as greater than 29 and low number of kills as smaller or equals to 29.

In [26]:
```
# Calculating the number of kills per game by groupby and aggregating

kills_per_game = fairness_data.groupby('gameid')['kills'].agg(lambda ser: int(ser.sum() / 2))
kills_per_game
```

Out[26]:
```
gameid
ESPORTSTMNT01_2690210    28
ESPORTSTMNT01_2690219    19
ESPORTSTMNT01_2690227    19
ESPORTSTMNT01_2690255    29
ESPORTSTMNT01_2690264    21
                         ..
NA1_4493482900           43
NA1_4493540863           42
NA1_4493591166           45
NA1_4493652706           29
NA1_4493722220           41
Name: kills, Length: 10587, dtype: int64
```

In [27]:
```
# Calculating the mean number of kills per game

mean_kills_per_game = kills_per_game.mean()
mean_kills_per_game
```

Out[27]:
```
29.198262019457825
```

Setting up the permutation test

Spliting the data by number of kills in the game

In [28]:
```
# Helper function that returns 1 if the number of kills in the game is higher than the threshold, else 0
def kills_threshold(ser):
    num_kills = ser.sum() / 2
    if num_kills > 29:
        return pd.Series([1] * ser.shape[0])
    return pd.Series([0] * ser.shape[0])

kill_threshold_df = fairness_data.assign(high_kills = fairness_data.groupby('gameid')[['kills']]
                                         .transform(kills_threshold))
kill_threshold_df
```

Out[28]:

	gameid	result	datacompleteness	teamname	playoffs	side	position	kills	deaths	assists	...	heralds	inhibitor
0	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	top	2	3	2	...	0.0	0.
1	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	jng	2	5	6	...	0.0	0.
2	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	mid	2	2	3	...	0.0	0.
3	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	bot	2	4	2	...	0.0	0.
4	ESPORTSTMNT01_2690210	0	True	Fredit BRION Challengers	0	Blue	sup	1	5	6	...	0.0	0.
...	...	...	...	...	...	...	...	...	...	...	...	...	
149131	ESPORTSTMNT01_3268705	0	True	Córdoba Patrimonio eSports	0	Red	mid	2	3	3	...	0.0	0.
149132	ESPORTSTMNT01_3268705	0	True	Córdoba Patrimonio eSports	0	Red	bot	2	1	3	...	0.0	0.
149133	ESPORTSTMNT01_3268705	0	True	Córdoba Patrimonio eSports	0	Red	sup	0	5	4	...	0.0	0.
149134	ESPORTSTMNT01_3268705	1	True	unknown team	0	Blue	team	22	9	50	...	2.0	2.
149135	ESPORTSTMNT01_3268705	0	True	Córdoba Patrimonio eSports	0	Red	team	9	22	14	...	0.0	0.

127044 rows × 30 columns

In [29]:
```
# Splitting the data into more kills and less kills while dropping columns that our final model cannot use

# Helper function to split the data into high number of kills games and low number of kills games
def split_data(df):
    more_kills_data = df[df['high_kills'] == 1].drop(columns=['gameid', 'high_kills'])
    less_kills_data = df[df['high_kills'] == 0].drop(columns=['gameid', 'high_kills'])

    return more_kills_data, less_kills_data

split_data(kill_threshold_df)[0]
```

Out[29]:

	result	datacompleteness	teamname	playoffs	side	position	kills	deaths	assists	firstblood	...	elders	heralds	inhibitors	
120	1	True	Vanir	0	Blue	top	2	6	6	0.0	...	0.0	0.0	0.0	335.9
121	1	True	Vanir	0	Blue	jng	4	7	11	1.0	...	0.0	0.0	0.0	506.5
122	1	True	Vanir	0	Blue	mid	8	5	5	0.0	...	0.0	0.0	1.0	582.2
123	1	True	Vanir	0	Blue	bot	6	2	10	0.0	...	0.0	0.0	2.0	920.5
124	1	True	Vanir	0	Blue	sup	0	3	18	0.0	...	0.0	0.0	0.0	60.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
149131	0	True	Córdoba Patrimonio eSports	0	Red	mid	2	3	3	0.0	...	0.0	0.0	0.0	424.
149132	0	True	Córdoba Patrimonio eSports	0	Red	bot	2	1	3	0.0	...	0.0	0.0	0.0	386.3
149133	0	True	Córdoba Patrimonio eSports	0	Red	sup	0	5	4	0.0	...	0.0	0.0	0.0	243.9
149134	1	True	unknown team	0	Blue	team	22	9	50	1.0	...	0.0	2.0	2.0	2771.
149135	0	True	Córdoba Patrimonio eSports	0	Red	team	9	22	14	0.0	...	0.0	0.0	0.0	1588.2

55536 rows × 28 columns

Performing the Permutations Test

Evaluation metric

The most appropriate metric to use for this prediction task is accuracy. This is because if one team wins, the other team loses. Thus, false positives and false negatives are both irrelevant.

Null Hypothesis

The classifier's accuracy is the same for games with high number of kills and games with low number of kills, any differences are due to chance.

Alternative Hypothesis

The classifier's accuracy is higher for games with less number of kills.

Test statistic

Difference in accuracy (less number of kills - high number of kills).

Significance Level

0.01

In [30]:
```
# Helper functiont to calculate the test statistic given data

def test_statistic(df):
    more, less = split_data(df)
    score_more = log_pipeline.score(more.drop(columns=['result']), more['result'])
    score_less = log_pipeline.score(less.drop(columns=['result']), less['result'])
    return score_less - score_more

# Calculating the observed statistic using the helper function on the not shuffled dataframe

observed_stat = test_statistic(kill_threshold_df)
observed_stat
```

Out[30]:
```
0.05551331716020291
```

In [31]:
```
# Function to shuffle the dataframe.
# Because games are in 'bundles' of 12 rows, we shuffled rows in groups of 12

def shuffle_df(df):
    # Getting the label for each game
    kills = df.groupby('gameid')['high_kills'].agg(lambda x: int(x.mean()))
    # Shuffling the labels and then repeating each row 12 times so that it can be added back to the original
    shuffled_kills = kills.sample(frac=1.0).repeat(12).reset_index(drop=True)
    # Reassigning the shuffled column to create the shuffled dataframe
    out_df = df.reset_index().assign(high_kills = shuffled_kills)
```

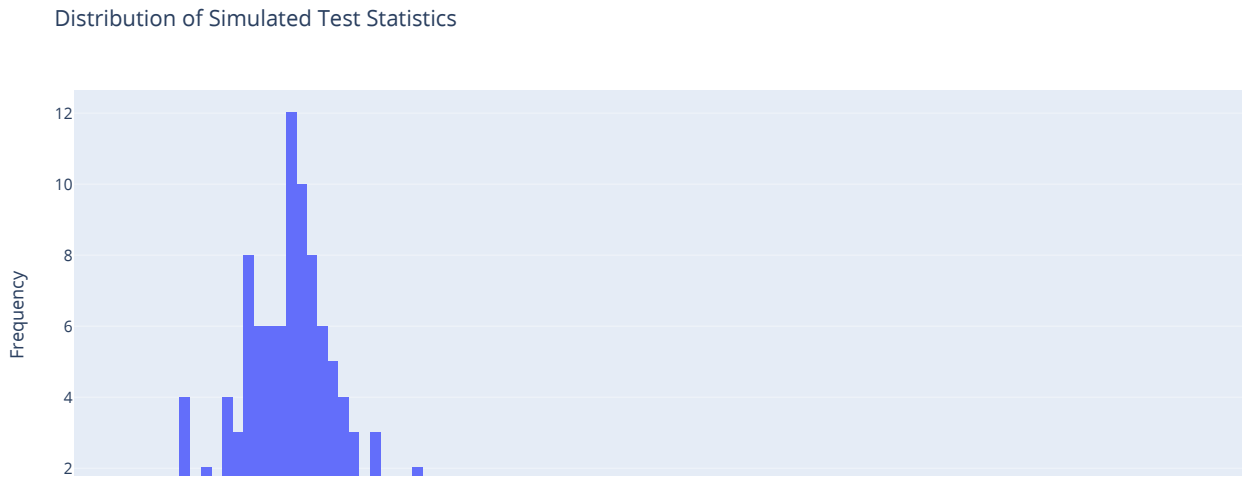
```
    return out_df.drop(columns=['index'])

# Performing the permutations test by simulating 100 test statistics

test_stats = []
for i in range(100):
    shuffled = shuffle_df(kill_threshold_df)
    test_stats.append(test_statistic(shuffled))
```

In [32]:
```
# Plot the distribution of simulated statistics

fig = px.histogram(test_stats, nbins=50, title='Distribution of Simulated Test Statistics')
fig.update_layout(showlegend=False)
fig.add_vline(observed_stat)
fig.add_annotation(text='observed statistic', x=0.05, y=7, showarrow=False)
fig.update_xaxes(range=[-0.01, 0.06],
                 title_text='Difference in accuracy (less kills games - high kills games)')
fig.update_yaxes(title_text='Frequency')
fig.show()
```



In [33]:
```
# Calculate p-value

pval = np.mean(np.array(test_stats) > observed_stat)
print(f'p-value: {pval}')
```

```
p-value: 0.0
```

Under the null hypothesis, we rarely see differences in probability as large as the observed statistic (0.0 < 0.01).

Thus, we can reject the null hypothesis and have sufficient evidence to support the alternate hypothesis: suggesting that our model works significantly better for games with lower total kills compared to games with higher total kills.