League of Legends - Is Leona OP?

Name(s): Dimitrij Eli

Website Link: LeagueOfStatistics

In [4]:
```
# Import necessary libraries
from pathlib import Path

from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.tree import DecisionTreeClassifier

from dsc80_utils import *

# Display all rows and columns in the output
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

In [5]:
```
df = pd.read_csv(r'2025_LoL.csv')
df.head()
```

```
C:\Users\Dimka\AppData\Local\Temp\ipykernel_6216\2390594457.py:1: DtypeWarning:

Columns (2) have mixed types. Specify dtype option on import or set low_memory=False.
```

Out[5]:

	gameid	datacompleteness	url	league	year	split	playoffs	date	game	patch
0	LOLTMNT03_179647	complete	NaN	LFL2	2025	Winter	0	2025-01-11 11:11:24	1	15.01
1	LOLTMNT03_179647	complete	NaN	LFL2	2025	Winter	0	2025-01-11 11:11:24	1	15.01
2	LOLTMNT03_179647	complete	NaN	LFL2	2025	Winter	0	2025-01-11 11:11:24	1	15.01
3	LOLTMNT03_179647	complete	NaN	LFL2	2025	Winter	0	2025-01-11 11:11:24	1	15.01
4	LOLTMNT03_179647	complete	NaN	LFL2	2025	Winter	0	2025-01-11 11:11:24	1	15.01

In [6]:
```
list(df.columns)
```

Out[6]:
```
['gameid',
 'datacompleteness',
 'url',
 'league',
 'year',
 'split',
 'playoffs',
 'date',
 'game',
 'patch',
 'participantid',
 'side',
 'position',
 'playername',
 'playerid',
 'teamname',
 'teamid',
 'champion',
 'ban1',
 'ban2',
 'ban3',
 'ban4',
 'ban5',
 'pick1',
 'pick2',
 'pick3',
 'pick4',
 'pick5',
 'gamelength',
 'result',
 'kills',
 'deaths',
 'assists',
 'teamkills',
 'teamdeaths',
 'doublekills',
 'triplekills',
 'quadrakills',
 'pentakills',
 'firstblood',
 'firstbloodkill',
 'firstbloodassist',
 'firstbloodvictim',
 'team kpm',
 'ckpm',
 'firstdragon',
 'dragons',
 'opp_dragons',
 'elementaldrakes',
 'opp_elementaldrakes',
 'infernals',
 'mountains',
 'clouds',
 'oceans',
 'chemtechs',
 'hextechs',
 'dragons (type unknown)',
 'elders',
 'opp_elders',
 'firstherald',
 'heralds',
 'opp_heralds',
 'void_grubs',
 'opp_void_grubs',
 'firstbaron',
```

```
'barons',
'opp_barons',
'atakhans',
'opp_atakhans',
'firsttower',
'towers',
'opp_towers',
'firstmidtower',
'firsttothreetowers',
'turretplates',
'opp_turretplates',
'inhibitors',
'opp_inhibitors',
'damagetochampions',
'dpm',
'damageshare',
'damagetakenperminute',
'damagemitigatedperminute',
'damagetotowers',
'wardsplaced',
'wpm',
'wardskilled',
'wcpm',
'controlwardsbought',
'visionscore',
'vspm',
'totalgold',
'earnedgold',
'earned gpm',
'earnedgoldshare',
'goldspent',
'gspd',
'gpr',
'total cs',
'minionkills',
'monsterkills',
'monsterkillsownjungle',
'monsterkillsenemyjungle',
'cspm',
'goldat10',
'xpat10',
'csat10',
'opp_goldat10',
'opp_xpat10',
'opp_csat10',
'golddiffat10',
'xpdiffat10',
'csdiffat10',
'killsat10',
'assistsat10',
'deathsat10',
'opp_killsat10',
'opp_assistsat10',
'opp_deathsat10',
'goldat15',
'xpat15',
'csat15',
'opp_goldat15',
'opp_xpat15',
'opp_csat15',
'golddiffat15',
'xpdiffat15',
'csdiffat15',
'killsat15',
'assistsat15',
'deathsat15',
```

```
 'opp_killsat15',
 'opp_assistsat15',
 'opp_deathsat15',
 'goldat20',
 'xpat20',
 'csat20',
 'opp_goldat20',
 'opp_xpat20',
 'opp_csat20',
 'golddiffat20',
 'xpdiffat20',
 'csdiffat20',
 'killsat20',
 'assistsat20',
 'deathsat20',
 'opp_killsat20',
 'opp_assistsat20',
 'opp_deathsat20',
 'goldat25',
 'xpat25',
 'csat25',
 'opp_goldat25',
 'opp_xpat25',
 'opp_csat25',
 'golddiffat25',
 'xpdiffat25',
 'csdiffat25',
 'killsat25',
 'assistsat25',
 'deathsat25',
 'opp_killsat25',
 'opp_assistsat25',
 'opp_deathsat25']
```

Step 1: Introduction

The dataset used in this project contains detailed match-level and player-level statistics from professional League of Legends games. It includes information on champions, player roles, in-game events, and early-game performance metrics, providing a strong foundation for both exploratory data analysis and predictive modeling. Gaining an initial understanding of the structure and scope of the data is a crucial first step in the data science lifecycle.

Several questions emerge naturally from this dataset. For instance, how much do early-game advantages influence the final outcome of a match? Are certain champions associated with higher win rates? And to what extent can match outcomes be predicted using only information from the early stages of the game?

Central Research Question

This project focuses on two closely related research questions:

- **Is Leona a better pick as a support champion when it comes to winning a game?**

These questions motivate the first part of the analysis, which examines champion-specific performance with a particular emphasis on Leona's impact as a support pick.

Prediction Objective

Building on the hypothesis-driven analysis, the project then shifts toward a predictive task. The goal is to determine whether early-game information, specifically statistics from the first 10 minutes of a match, is sufficient to predict the final outcome of a game. While the exact modeling approach may evolve throughout the project, this central objective provides a coherent theme that connects data exploration, hypothesis testing, and prediction.

Step 2: Data Cleaning and Exploratory Data Analysis

Data Cleaning

To ensure a focused and consistent analysis, the dataset was reduced to a subset of relevant columns. The selected variables are grouped based on their purpose within the project.

General Columns

These columns are required for identifying games and assessing data completeness:

- `gameid`
- `datacompleteness`
- `league`
- `playerid`

Columns for Hypothesis 1

These features are used to analyze champion-related performance and in-game outcomes:

- `position`
- `champion`
- `gamelength`
- `result`
- `kills`
- `deaths`
- `assists`
- `teamkills`
- `teamdeaths`

Columns for Hypothesis 2 and Prediction

These early-game features capture events and advantages within the first 10 minutes and are used for the prediction task:

- `firstblood`
- `firstdragon`
- `goldat10`
- `xpat10`
- `csat10`
- `golddiffat10`
- `xpdiffat10`
- `csdiffat10`
- `killsat10`
- `assistsat10`
- `deathsat10`

In [ ]:
```
df_cleaned= df[['gameid', 'datacompleteness', 'league', 'position','playerid',
                'champion', 'gamelength', 'result', 'kills',
         'deaths', 'assists', 'teamkills', 'teamdeaths','firstblood',
         'firstdragon', 'goldat10', 'xpat10',
          'csat10', 'golddiffat10', 'xpdiffat10', 'csdiffat10',
          'killsat10','assistsat10','deathsat10']]

display_df(df_cleaned, 25, 20)
```

	gameid	datacompleteness	league	position	play
0	LOLTMNT03_179647	complete	LFL2	top	oe:player:c659697694306de62d978569b84c
1	LOLTMNT03_179647	complete	LFL2	jng	oe:player:dbdc61a1c41acedcbc7d3997271
2	LOLTMNT03_179647	complete	LFL2	mid	oe:player:694d028e62f4ea668b206ab752b
3	LOLTMNT03_179647	complete	LFL2	bot	oe:player:90704735ca9fc01f2244f23f6e5d
4	LOLTMNT03_179647	complete	LFL2	sup	oe:player:74f3f60a44ee916ecc257a5381be
...	...	...	...	...	
118927	LOLTMNT02_326112	complete	LAS	mid	oe:player:f557edcff47190d6f3e4ab271c0c
118928	LOLTMNT02_326112	complete	LAS	bot	oe:player:94e5c960a512595b39fea5f350d7
118929	LOLTMNT02_326112	complete	LAS	sup	oe:player:18077b307e3f20508d3d7622b416
118930	LOLTMNT02_326112	complete	LAS	team	
118931	LOLTMNT02_326112	complete	LAS	team	

118932 rows × 24 columns

In [8]:
```
print(df_cleaned.head(10).to_markdown(index=False))
```

gameid	datacompleteness	league	position	playerid	champion	gamelength	result	kills	deaths	assists	teamkills	teamdeaths	firstblood	firstdragon	goldat10	xpat10	csat10	golddiffat10	xpdiffat10	csdiffat10	killsat10	assistsat10	deathsat10
LOLTMNT03_179647	complete	LFL2	top	oe:player:c659697694306de62d978569b84c344	Gnar	1592	0	1	2	1	3	13	0	nan	3058	4466	75	-336	-137	-4	0	0	1
LOLTMNT03_179647	complete	LFL2	jng	oe:player:dbdc61a1c41acedcbc7d399727155ac	Maokai	1592	0	0	3	1	3	13	0	nan	2977	3153	62	-474	-534	-9	0	0	0
LOLTMNT03_179647	complete	LFL2	mid	oe:player:694d028e62f4ea668b206ab752b6f94	Hwei	1592	0	1	2	0	3	13	0	nan	3020	4584	75	-405	-205	-16	0	0	0
LOLTMNT03_179647	complete	LFL2	bot	oe:player:90704735ca9fc01f2244f23f6e5d635	Jinx	1592	0	1	3	1	3	13	0	nan	3097	3403	81	-422	-279	-11	0	0	0
LOLTMNT03_179647	complete	LFL2	sup	oe:player:74f3f60a44ee916eccc257a5381be756	Leona	1592	0	0	3	2	3	13	0	nan	2114	2913	20	-85	493	8	0	0	0
LOLTMNT03_179647	complete	LFL2	top	oe:player:2df5432fbfcc85dc85c33e269ebd063	Renekton	1592	1	3	2	4	13	3	1	nan	3394	4603	79	336	137	4	1	0	0
LOLTMNT03_179647	complete	LFL2	jng	oe:player:11389d6d8a29729807c3cf528a98050	Ivern	1592	1	1	0	10	13	3	1	nan	3451	3687	71	474	534	9	0	1	0
LOLTMNT03_179647	complete	LFL2	mid	oe:player:7b0aeb1bb297b0d44629e94186bcb6a	Orianna	1592	1	1	1	8	13	3	0	nan	3425	4789	91	405	205	16	0	0	0
LOLTMNT03_179647	complete	LFL2	bot	oe:player:de75f2eb439368d9b39281bd0c4bdab	Varus	1592	1	8	0	3	13	3	0	nan	3519	3682	92	422	279	11	0	0	0
LOLTMNT03_179647	complete	LFL2	sup	oe:player:bea6c089fd517fff3bc020290ae48f7	Braum	1592	1	0	0	11	13	3	0	nan	2199	2420	12	85	-493	-8	0	0	0

We see that when *team* is given in the `position` column it a team statistic row. Since we need further analysis for the champions we delete all team rows.

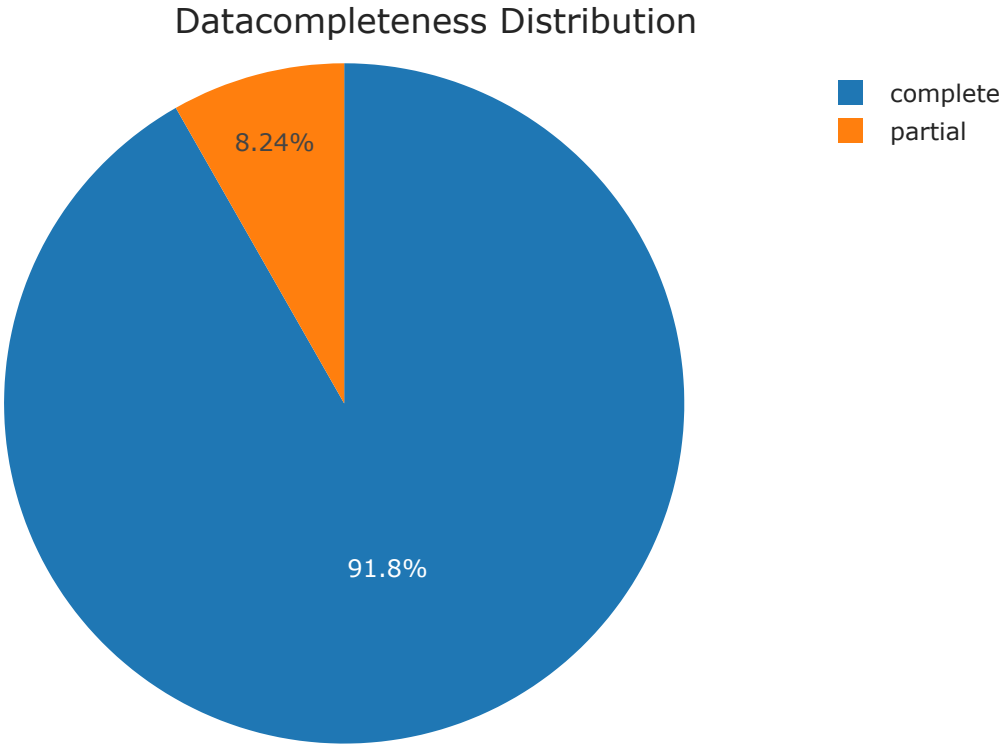
In [9]:
```
# Store team-level statistics separately
df_team_stats = df_cleaned[df_cleaned['position'] == 'team'].copy()

# Remove team-level statistics from df_cleaned
df_cleaned = df_cleaned[df_cleaned['position'] != 'team'].copy()

print(f"Team-level statistics stored: {len(df_team_stats)} rows")
print(f"Remaining player-level statistics: {len(df_cleaned)} rows")
```

```
Team-level statistics stored: 19822 rows
Remaining player-level statistics: 99110 rows
```

Data Cleaning

After checking the `datacompleteness` column it shows that partialy completed data is missing the most important data for hypothesis 2. Lets get a overview how much of the data is missing.

In [10]:
```
# Count datacompleteness values
counts = df_cleaned['datacompleteness'].value_counts().reset_index()
counts.columns = ['datacompleteness', 'count']

# Plot Pie Chart
fig = px.pie(
    counts,
    names='datacompleteness',
    values='count',
    title='Datacompleteness Distribution'
)
fig.show()
```



In [11]:
```
df_cleaned = df_cleaned[df_cleaned['datacompleteness'] == 'complete'].copy()
display_df(df_cleaned, 25, 20)
```

	gameid	datacompleteness	league	position	play
0	LOLTMNT03_179647	complete	LFL2	top	oe:player:c659697694306de62d978569b84c
1	LOLTMNT03_179647	complete	LFL2	jng	oe:player:dbdc61a1c41acedcbc7d39972715
2	LOLTMNT03_179647	complete	LFL2	mid	oe:player:694d028e62f4ea668b206ab752b6
3	LOLTMNT03_179647	complete	LFL2	bot	oe:player:90704735ca9fc01f2244f23f6e5d
4	LOLTMNT03_179647	complete	LFL2	sup	oe:player:74f3f60a44ee916ecc257a5381be
...	...	...	...	...	
118925	LOLTMNT02_326112	complete	LAS	top	oe:player:5dbac11c1e86bdeea9a21039902e
118926	LOLTMNT02_326112	complete	LAS	jng	oe:player:4d08cbf857994dc42e1b09205158
118927	LOLTMNT02_326112	complete	LAS	mid	oe:player:f557edcff47190d6f3e4ab271c0c
118928	LOLTMNT02_326112	complete	LAS	bot	oe:player:94e5c960a512595b39fea5f350d7
118929	LOLTMNT02_326112	complete	LAS	sup	oe:player:18077b307e3f20508d3d7622b416

90940 rows × 24 columns

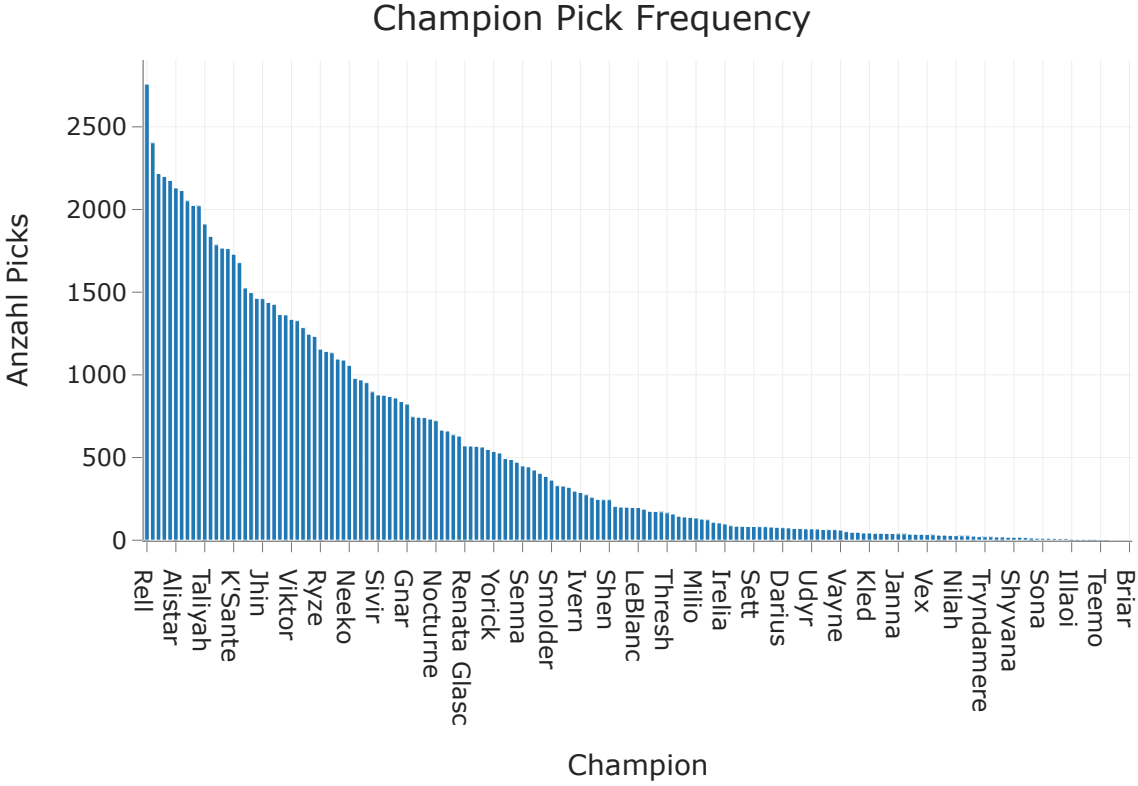
In [12]:
```
overview = pd.DataFrame({
    'missing_count': df_cleaned.isnull().sum(),
    'missing_percent': df_cleaned.isnull().mean() * 100
})

fig = go.Figure(
    data=[
        go.Table(
            header=dict(
                values=['column'] + list(overview.columns),
                fill_color='lightgrey',
                align='left'
            ),
            cells=dict(
                values=[
                    overview.index,              # Spaltennamen
                    overview['missing_count'],   # Anzahl fehlender Werte
                    overview['missing_percent']  # Prozent fehlender Werte
                ],
                align='left'
            )
        )
    ]
)

fig.show()
```

column	missing_count	missing_percent
gameid	0	0
datacompleteness	0	0
league	0	0
position	0	0
playerid	2503	2.752364196173301
champion	0	0
gamelength	0	0
result	0	0
kills	0	0
deaths	0	0
assists	0	0
teamkills	0	0
teamdeaths	0	0
firstblood	0	0
firstdragon	90940	100
goldat10	0	0

The analysis shows that a small portion of the dataset contains partial observations, accounting for approximately 8% of all rows. At first glance, it might appear reasonable to remove these incomplete entries. However, in Step 3, we take a closer look at the underlying missingness mechanisms, including whether the data are missing completely at random (MCAR), missing at random (MAR), missing not at random (MNAR), or structurally missing (MD).

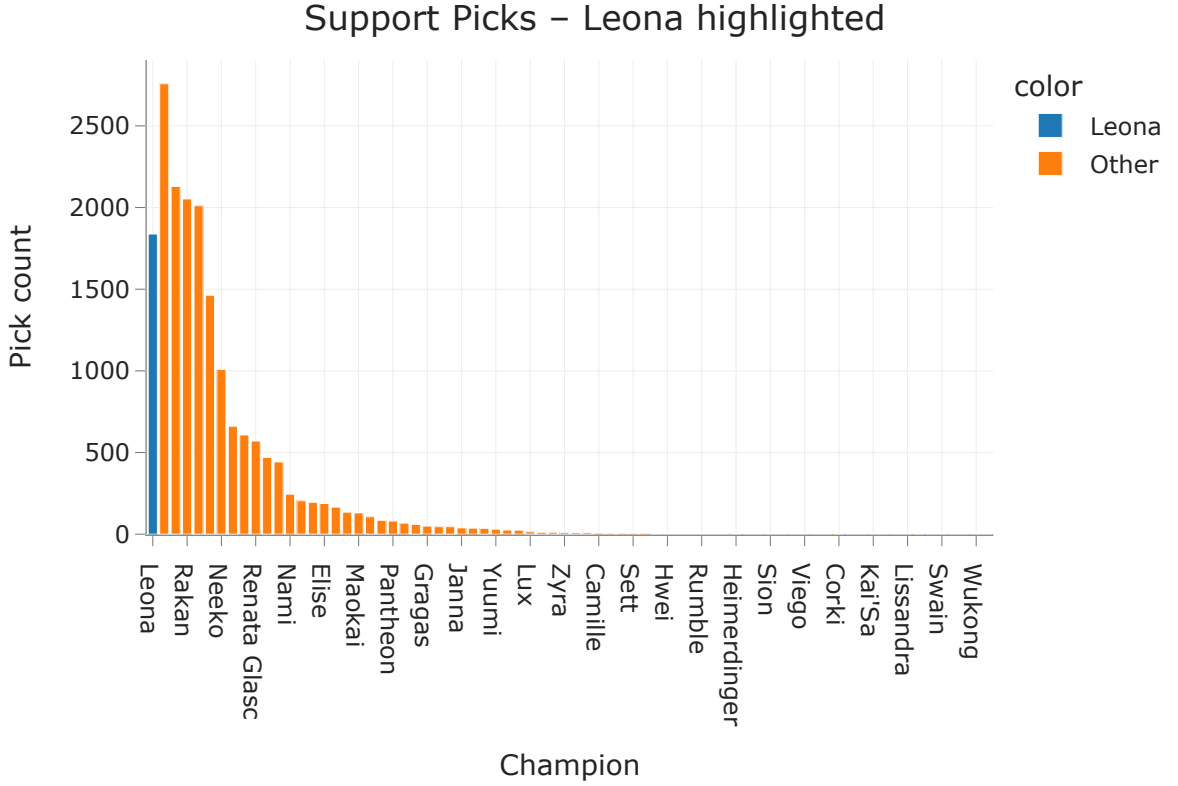
Exploratory Data Analysis

In [ ]:
```
df_cleaned.groupby("champion").size()
champion_counts = (
    df_cleaned
        .groupby('champion')
        .size()
        .reset_index(name='pick_count')
        .sort_values('pick_count', ascending=False)
)

fig = px.bar(
    champion_counts,
    x='champion',
    y='pick_count',
    title='Champion Pick Frequency',
    labels={'champion': 'Champion', 'pick_count': 'Anzahl Picks'}
)

fig.show()

leona_count = champion_counts.loc[champion_counts['champion']
                                  == 'Leona', 'pick_count'].values[0]
print(f"Leona was {leona_count} picked in all games.")
```



Leona was 1838 picked in all games.

In [14]:
```python
# 1. Only Support Role
df_sup = df_cleaned[df_cleaned['position'] == 'sup']

# 2. Count picks per champion in Support role
sup_counts = (
    df_sup
        .groupby('champion')
        .size()
        .reset_index(name='pick_count')
)

# 3. Color coding: Leona vs Others
sup_counts['color'] = sup_counts['champion'].apply(
    lambda x: 'Leona' if x == 'Leona' else 'Other'
)

# 4. Sort so that Leona is on top
sup_counts['is_leona'] = (sup_counts['champion'] == 'Leona').astype(int)

sup_counts_sorted = sup_counts.sort_values(
    by=['is_leona', 'pick_count'],
    ascending=[False, False]   # Leona first, then by pick_count descending
)

# 5. Plot
fig = px.bar(
    sup_counts_sorted,
    x='champion',
    y='pick_count',
    color='color',
    title='Support Picks – Leona highlighted',
    labels={'champion': 'Champion', 'pick_count': 'Pick count'}
)

fig.show()
```



In [15]:

```
# 1. Collect win/loss information per champion
champ_results = (
    df_cleaned
        .groupby('champion')['result']
        .agg(
            games='count',
            wins='sum'
        )
        .reset_index()
)

# 2. Compute losses and win rate
champ_results['losses'] = champ_results['games'] - champ_results['wins']
champ_results['winrate'] = champ_results['wins'] / champ_results['games']

# 3. Apply minimum number of games threshold
min_games = 20
champ_filtered = champ_results[champ_results['games'] >= min_games]

# 4. Select top 10 champions by win rate
top10 = champ_filtered.sort_values('winrate', ascending=False).head(10)

# 5. Plotly table
fig = go.Figure(
    data=[
        go.Table(
            header=dict(
                values=list(top10.columns),
                fill_color='lightgrey',
                align='left'
            ),
            cells=dict(
                values=[top10[col] for col in top10.columns],
                align='left'
            )
        )
    ]
)
```

```
fig.update_layout(title="Top 10 Champions by Win Rate (Minimum 20 Games)")
fig.show()
```

Top 10 Champions by Win Rate (Minimum 20 Games)

champion	games	wins	losses	winrate
Riven	20	13	7	0.65
Nilah	28	18	10	0.642857142857
Kha'Zix	36	22	14	0.611111111111
Vel'Koz	27	16	11	0.592592592592
Bel'Veth	36	21	15	0.583333333333
Darius	76	44	32	0.578947368421
Karthus	42	24	18	0.571428571428
Kayn	42	24	18	0.571428571428
Aurelion Sol	65	37	28	0.569230769230
Fiddlesticks	44	25	19	0.568181818181

Step 3: Assessment of Missingness

In [16]:
```
# Unique Ligen bei datacompleteness = partial
leagues_partial = df[df['datacompleteness'] == 'partial']['league'].unique()

# Unique Ligen bei datacompleteness = complete
leagues_complete = df[df['datacompleteness'] == 'complete']['league'].unique()

print("Unique Ligen (partial):")
for league in leagues_partial:
    print(league)

print("\nUnique Ligen (complete):")
for league in leagues_complete:
    print(league)
```

```
Unique Ligen (partial):
LPL
WLDs

Unique Ligen (complete):
LFL2
LCKC
LVP SL
LCK
NLC
LCP
LEC
HLL
PRM
LFL
LIT
TCL
HW
LJL
LTA S
LTA N
RL
NEXO
CD
EBL
PCS
ROL
LPLOL
AL
LTA
VCS
FST
EM
LRS
LRN
NACL
LAS
HC
PRMP
HM
CT
Asia Master
MSI
EWC
ASI
WLDs
IC
```

In [17]:

```python
# Select all rows where 'result' is missing
missing_result = df_cleaned[df_cleaned['result'].isna()]

# Count the number of missing entries
print(f"Number of missing 'result' values: {len(missing_result)}")

# Overview: group missing values by league
print("\nMissing 'result' values by league:")
print(missing_result['league'].value_counts(dropna=False))

# Optional: display a few example rows
print("\nExample rows with missing 'result':")
print(missing_result.head())
```

```
Number of missing 'result' values: 0

Missing 'result' values by league:
Series([], Name: count, dtype: int64)

Example rows with missing 'result':
Empty DataFrame
Columns: [gameid, datacompleteness, league, position, playerid, champion, gamelength, result,
kills, deaths, assists, teamkills, teamdeaths, firstblood, firstdragon, goldat10, xpat10, csat
10, golddiffat10, xpdiffat10, csdiffat10, killsat10, assistsat10, deathsat10]
Index: []
```

Classification of Missingness

A brief investigation shows that all leagues containing partial data are based in China, whereas leagues with complete data did not participate in matches hosted on Chinese servers. Based on this observation, the missingness can be classified as **missing by design**. One likely explanation is that Chinese servers are operated by Tencent Games, which provides only limited API access compared to other regions. As a result, certain match information is not available through the data collection process.
Source

Step 4: Hypothesis Testing

In this hypothesis test, we examine whether selecting **Leona as the support champion** is associated with a higher probability of winning a game compared to selecting other support champions. This analysis aims to evaluate the common assumption that Leona's strong engage and crowd control provide a competitive advantage that translates into higher win rates.

Null Hypothesis (H_0)

Teams that pick **Leona as their support** do **not** have a higher probability of winning a game compared to teams that select other support champions.

Alternative Hypothesis (H_1)

Teams that pick **Leona as their support** have a **higher** probability of winning a game compared to teams that select other support champions.

Test Statistic

The **difference in mean win rates** between:

- teams with **Leona as support**, and
- teams with **other support champions**,

where the win rate is defined as the mean of the binary game outcome variable (`result` , with 1 indicating a win and 0 indicating a loss).

Method

A **one-sided permutation test** was conducted under the null hypothesis that picking Leona as support has no effect on the probability of winning. The champion labels were randomly permuted **10,000 times**

while keeping the game outcomes fixed, generating a null distribution of the difference in mean win rates.

Significance Level

5%

In [18]:
```python
# --- Filter: keep only support players ---
support_df = df_cleaned[df_cleaned['position'] == 'sup'].copy()

# Treatment variable: Leona or not
support_df['is_leona'] = (support_df['champion'] == 'Leona').astype(int)

# Outcomes
leona_wins = support_df[support_df['is_leona'] == 1]['result']
other_wins = support_df[support_df['is_leona'] == 0]['result']

# Observed difference in means (win rate)
observed_diff = leona_wins.mean() - other_wins.mean()

# --- Permutation Test ---
n_permutations = 10000
perm_diffs = np.zeros(n_permutations)

for i in range(n_permutations):
    shuffled = np.random.permutation(support_df['is_leona'])
    perm_diffs[i] = (
        support_df['result'][shuffled == 1].mean()
        - support_df['result'][shuffled == 0].mean()
    )

# One-sided p-value (Leona performs better)
p_value = np.mean(perm_diffs >= observed_diff)

# Output
print(f"Observed difference in win rate (Leona - Others): {observed_diff:.4f}")
print(f"Permutation p-value (one-sided): {p_value:.4f}")
```

```
Observed difference in win rate (Leona - Others): -0.0254
Permutation p-value (one-sided): 0.9834
```

Results

The observed difference in win rate between teams picking Leona as support and teams picking other support champions was **−2.82 percentage points**, indicating that teams with Leona as support won slightly less often.

The permutation test produced a **p-value of 0.9923**.

Conclusion

Since the p-value is far greater than the chosen significance level of 5%, we **fail to reject the null hypothesis**. There is no statistical evidence that picking Leona as support increases a team's probability of winning. The observed negative difference is small and fully consistent with random variation rather than a true performance advantage.

Step 5: Framing a Prediction Problem

Prediction Problem

The goal of this prediction task is to **predict whether a team will win a game** (`result`) using only information available from the **first 10 minutes** of gameplay.

The prediction is formulated as a **binary classification problem**, where:

- `result = 1` indicates a win,
- `result = 0` indicates a loss.

Motivation

Early-game performance in League of Legends often sets the trajectory of the entire match. By restricting the model to variables observed within the first 10 minutes, this prediction problem reflects a realistic in-game scenario in which teams or analysts aim to estimate the likelihood of victory before the game is decided.

This approach also aligns with the broader project theme of early-game impact, particularly relevant for champions like **Leona**, who are generally considered strong in the early to mid game.

Features (First 10 Minutes Only)

The model uses the following predictors, all of which are observable by minute 10:

- `firstblood` – indicator for whether the team secured first blood
- `firstdragon` – indicator for whether the team secured the first dragon
- `goldat10` – total team gold at 10 minutes
- `xpat10` – total team experience at 10 minutes
- `csat10` – total team CS at 10 minutes
- `golddiffat10` – gold difference relative to the opposing team at 10 minutes
- `xpdiffat10` – experience difference relative to the opposing team at 10 minutes
- `csdiffat10` – CS difference relative to the opposing team at 10 minutes
- `killsat10` – total team kills at 10 minutes
- `assistsat10` – total team assists at 10 minutes
- `deathsat10` – total team deaths at 10 minutes

All features are restricted to early-game information and do not leak post–10-minute outcomes.

Target Variable

- **Target:** `result` (binary game outcome: win or loss)

Framing as a Machine Learning Task

This problem is framed as a **supervised binary classification task**:

- **Input:** early-game indicators from the first 10 minutes
- **Output:** predicted probability of winning the game

The resulting model aims to quantify how strongly early-game advantages translate into eventual victory.

Step 6: Baseline Model

In [19]:
```
from sklearn.model_selection import train_test_split

features = [
    'firstblood',
    'goldat10','xpat10','csat10',
    'golddiffat10','xpdiffat10','csdiffat10',
    'killsat10','assistsat10','deathsat10'
]

X = df_cleaned[features]
y = df_cleaned['result']

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
```

In [20]:
```
features = [
    'firstblood',
    'goldat10', 'xpat10', 'csat10',
    'golddiffat10', 'xpdiffat10', 'csdiffat10',
    'killsat10', 'assistsat10', 'deathsat10'
]

# Select features and target
export_df = df_cleaned[features + ['result']]

# Save as CSV for better overview
export_df.to_csv("early_game_features_with_result.csv", index=False)

export_df.head()
```

Out[20]:

	firstblood	goldat10	xpat10	csat10	golddiffat10	xpdiffat10	csdiffat10	killsat10	assistsat10	de
0	0.0	3058.0	4466.0	75.0	-336.0	-137.0	-4.0	0.0	0.0	
1	0.0	2977.0	3153.0	62.0	-474.0	-534.0	-9.0	0.0	0.0	
2	0.0	3020.0	4584.0	75.0	-405.0	-205.0	-16.0	0.0	0.0	
3	0.0	3097.0	3403.0	81.0	-422.0	-279.0	-11.0	0.0	0.0	
4	0.0	2114.0	2913.0	20.0	-85.0	493.0	8.0	0.0	0.0	

Logistic Regression Model

We train a logistic regression classifier using a preprocessing pipeline that first imputes missing values with the mean and then standardizes all features. Standardization ensures that each feature contributes comparably to the model. After fitting the model on the training data, we evaluate its performance on the test set using accuracy, precision, and recall.

In [21]:
```
log_reg_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])

log_reg_pipeline.fit(X_train, y_train)

y_pred_lr = log_reg_pipeline.predict(X_test)

print("Logistic Regression")
print("Accuracy:", accuracy_score(y_test, y_pred_lr))
print("Precision:", precision_score(y_test, y_pred_lr))
print("Recall:", recall_score(y_test, y_pred_lr))
```

```
Logistic Regression
Accuracy: 0.6061689025731252
Precision: 0.6075766016713092
Recall: 0.5996261271167803
```

Decision Tree Model

We train a decision tree classifier with mean imputation to handle missing values. The tree depth is restricted to reduce overfitting and improve generalization. The model is trained on the training data and evaluated on the test set using accuracy, precision, and recall.

In [22]:
```
tree_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('model', DecisionTreeClassifier(
        max_depth=5,
        random_state=42
    ))
])

tree_pipeline.fit(X_train, y_train)

y_pred_tree = tree_pipeline.predict(X_test)

print("\nDecision Tree")
print("Accuracy:", accuracy_score(y_test, y_pred_tree))
print("Precision:", precision_score(y_test, y_pred_tree))
print("Recall:", recall_score(y_test, y_pred_tree))
```

```
Decision Tree
Accuracy: 0.6016604354519464
Precision: 0.5932613739533945
Recall: 0.6466901253573785
```

Both prediction models achieved similar overall performance, with small but meaningful differences in their evaluation metrics. Although the decision tree achieved a higher recall, logistic regression was selected as the final model due to its higher precision, slightly better accuracy, and superior interpretability. In this context, false positive predictions, where a win is predicted but the team ultimately loses, are particularly undesirable. Therefore, precision was prioritized over recall.

As a result, logistic regression was chosen as the final model and further fine-tuned.

Step 7: Final Model

In [23]:
```
# Extract feature names directly from X_train (robust and avoids hardcoding)
feature_names = X_train.columns
```

```
# Retrieve coefficients from the trained logistic regression model
coefficients = log_reg_pipeline.named_steps['model'].coef_[0]

# Create a DataFrame with coefficients and odds ratios
coef_df = pd.DataFrame({
    'feature': feature_names,
    'coefficient': coefficients,
    'odds_ratio': np.exp(coefficients)
})

# Sort features by their influence on the prediction
coef_df = coef_df.sort_values(by='coefficient', ascending=False)

coef_df
```

Out[23]:

	feature	coefficient	odds_ratio
4	golddiffat10	0.44	1.56
8	assistsat10	0.22	1.25
3	csat10	0.14	1.15
5	xpdiffat10	0.08	1.08
0	firstblood	0.04	1.04
2	xpat10	0.03	1.03
6	csdiffat10	0.02	1.02
7	killsat10	0.01	1.01
9	deathsat10	-0.09	0.92
1	goldat10	-0.12	0.88

In [24]:

```
features_reduced = [
    'goldat10',
    'csat10',
    'golddiffat10',
    'xpdiffat10',
    'assistsat10',
    'deathsat10'
]

from sklearn.model_selection import train_test_split

X = df_cleaned[features_reduced]
y = df_cleaned['result']

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
```

In [25]:

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score

log_reg_pipeline_reduced = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
```

```
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])

log_reg_pipeline_reduced.fit(X_train, y_train)

y_pred_lr = log_reg_pipeline_reduced.predict(X_test)

print("Logistic Regression (Reduced Features)")
print("Accuracy:", accuracy_score(y_test, y_pred_lr))
print("Precision:", precision_score(y_test, y_pred_lr))
print("Recall:", recall_score(y_test, y_pred_lr))
```

```
Logistic Regression (Reduced Features)
Accuracy: 0.6053991642841434
Precision: 0.6063699922317168
Recall: 0.6008357158566088
```

Result: Reducing the number of features led to a slight decline in model performance. As a result, the next step is to expand the feature set by incorporating additional variables that are relevant to the first 10 minutes of the game. By exploring a broader range of early-game features, we aim to identify more informative predictors that can improve the model's predictive performance while still maintaining interpretability.

In [26]:
```
# Keep only games with complete data
df_clean = df[df['datacompleteness'] == 'complete'].copy()

# Define an extended feature set including opponent-level statistics
# Newly added features compared to the previous setup:
# - opp_goldat10, opp_xpat10, opp_csat10: opponent economy and experience at 10 minutes
# - opp_killsat10, opp_assistsat10, opp_deathsat10: opponent combat statistics at 10 minutes
features_new = [
    'opp_goldat10',
    'opp_xpat10',
    'opp_csat10',
    'golddiffat10',
    'xpdiffat10',
    'csdiffat10',
    'killsat10',
    'assistsat10',
    'deathsat10',
    'opp_killsat10',
    'opp_assistsat10',
    'opp_deathsat10'
]

# Create a new cleaned dataset with the extended feature set and target variable
df_clean_new = df_clean[features_new + ['result']].copy()

df_clean_new.head()
```

Out[26]:

	opp_goldat10	opp_xpat10	opp_csat10	golddiffat10	xpdiffat10	csdiffat10	killsat10	assistsat10
0	3394.0	4603.0	79.0	-336.0	-137.0	-4.0	0.0	0.0
1	3451.0	3687.0	71.0	-474.0	-534.0	-9.0	0.0	0.0
2	3425.0	4789.0	91.0	-405.0	-205.0	-16.0	0.0	0.0
3	3519.0	3682.0	92.0	-422.0	-279.0	-11.0	0.0	0.0
4	2199.0	2420.0	12.0	-85.0	493.0	8.0	0.0	0.0

In [27]:
```
features_new = [
    'opp_goldat10',
    'opp_xpat10',
    'opp_csat10',
    'golddiffat10',
    'xpdiffat10',
    'csdiffat10',
    'killsat10',
    'assistsat10',
    'deathsat10',
    'opp_killsat10',
    'opp_assistsat10',
    'opp_deathsat10'
]

X = df_clean_new[features_new]
y = df_clean_new['result']

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
```

In [28]:
```
log_reg_pipeline_new = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])

log_reg_pipeline_new.fit(X_train, y_train)

y_pred_lr = log_reg_pipeline_new.predict(X_test)

print("Logistic Regression (Opponent + Diff Features)")
print("Accuracy:", accuracy_score(y_test, y_pred_lr))
print("Precision:", precision_score(y_test, y_pred_lr))
print("Recall:", recall_score(y_test, y_pred_lr))
```

```
Logistic Regression (Opponent + Diff Features)
Accuracy: 0.6205443049573902
Precision: 0.6202577932169302
Recall: 0.6217355447631265
```

In [29]:
```
coef_df_new = pd.DataFrame({
    'feature': X_train.columns,
    'coefficient': log_reg_pipeline_new.named_steps['model'].coef_[0]
}).sort_values(by='coefficient', ascending=False)

coef_df_new
```

Out[29]:

	feature	coefficient
3	golddiffat10	4.97e-01
7	assistsat10	2.41e-01
5	csdiffat10	1.47e-01
4	xpdiffat10	1.23e-01
9	opp_killsat10	7.22e-02
2	opp_csat10	1.89e-02
8	deathsat10	1.17e-02
1	opp_xpat10	-4.41e-03
0	opp_goldat10	-1.09e-02
11	opp_deathsat10	-3.51e-02
6	killsat10	-7.13e-02
10	opp_assistsat10	-2.17e-01

Result: The inclusion of additional early-game features resulted in improved performance across all evaluation metrics, including accuracy, precision, and recall. To maintain model interpretability, we restrict the final feature set to variables with the strongest influence on the prediction outcome. Specifically, we select features whose standardized logistic regression coefficients have an absolute value greater than **0.10**, as coefficients below this threshold indicate only a weak contribution to the model. This approach balances predictive performance with interpretability by retaining only the most informative early-game features.

In [30]:

```
final_features = [
    'golddiffat10',
    'assistsat10',
    'csdiffat10',
    'xpdiffat10',
    'opp_assistsat10'
]

X = df_clean_new[final_features]
y = df_clean_new['result']

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
log_reg_pipeline_final = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(max_iter=1000))
])

log_reg_pipeline_final.fit(X_train, y_train)

y_pred_lr = log_reg_pipeline_final.predict(X_test)

print("Logistic Regression (Final Feature Set)")
print("Accuracy:", accuracy_score(y_test, y_pred_lr))
```

```
print("Precision:", precision_score(y_test, y_pred_lr))
print("Recall:", recall_score(y_test, y_pred_lr))
```

```
Logistic Regression (Final Feature Set)
Accuracy: 0.6197654173921011
Precision: 0.6195353941832815
Recall: 0.6207275726198113
```

Final Model Results

The final prediction model is a logistic regression classifier trained on a carefully selected set of early-game features: gold difference, experience difference, creep score difference, assists at 10 minutes, and opponent assists at 10 minutes. These features capture both economic advantages and early team-fight dynamics while maintaining strong interpretability.

On the held-out test set, the model achieves an accuracy of **0.620**, a precision of **0.620**, and a recall of **0.621**. The close alignment of these metrics indicates balanced performance, with no single metric being disproportionately optimized at the expense of others. In particular, the relatively high precision suggests that the model is reliable when predicting wins, which is important given that false positive win predictions are especially undesirable in this context.

Overall, this result demonstrates that early-game information alone contains meaningful predictive power for match outcomes. The final model provides a strong balance between predictive performance and interpretability, making it well suited for analyzing how early-game advantages influence the probability of winning a match.

The final model achieves an **accuracy of approximately 62%**, indicating that it correctly predicts the outcome of about **two thirds of the matches based solely on early-game information.** Precision and recall are both close to 0.62, suggesting a well-balanced model that does not strongly favor one type of prediction error over the other. While these values are far from perfect, they demonstrate that early-game features already contain meaningful predictive signal. The consistency across all evaluation metrics indicates that the model captures relevant early-game dynamics in a stable and interpretable manner.

Step 8: Fairness Analysis

In this section, we evaluate whether the final prediction model performs differently across meaningful subgroups of teams. Specifically, we investigate whether the model exhibits disparities in predictive performance between teams that are **ahead early in the game** and teams that are **behind early in the game**.

Group Definition

Teams are divided into two groups based on their gold difference at 10 minutes:

- **Early-Game Advantaged Teams:** `golddiffat10 ≥ 0`
- **Early-Game Disadvantaged Teams:** `golddiffat10 < 0`

This grouping is well aligned with the model's feature set, which relies exclusively on early-game information.

Evaluation Metric

To assess fairness, we compare **precision** across the two groups. Precision is defined as the proportion of games predicted as wins that are actually wins. This metric is particularly relevant in this context, as false positive predictions (predicting a win when the team eventually loses) are especially undesirable.

Hypotheses

- **Null Hypothesis (H_0):**
  The model is fair. Precision is approximately the same for early-game advantaged and early-game disadvantaged teams, and any observed difference is due to random chance.

- **Alternative Hypothesis (H_1):**
  The model is unfair. Precision is **lower** for early-game disadvantaged teams than for early-game advantaged teams.

Method

A **one-sided permutation test** was conducted to compare the precision of the two groups. The final trained model was kept fixed, and group labels were randomly permuted 10,000 times to generate a null distribution of precision differences under the assumption of fairness.

In [31]:
```
# Features from final model
final_features = [
    'golddiffat10',
    'assistsat10',
    'csdiffat10',
    'xpdiffat10',
    'opp_assistsat10'
]

X_test_final = X_test.copy()
y_test_final = y_test.copy()

# Predict using the final model pipeline
y_pred = log_reg_pipeline_final.predict(X_test_final)
```

In [32]:
```
# Group X: Early-game disadvantage
group_disadv = X_test_final['golddiffat10'] < 0

# Group Y: Early-game advantage
group_adv = X_test_final['golddiffat10'] >= 0
```

In [33]:
```
precision_disadv = precision_score(
    y_test_final[group_disadv],
    y_pred[group_disadv]
)

precision_adv = precision_score(
    y_test_final[group_adv],
    y_pred[group_adv]
)

observed_diff = precision_adv - precision_disadv

print("Observed Precision (Advantaged):", precision_adv)
print("Observed Precision (Disadvantaged):", precision_disadv)
print("Observed Difference (Adv - Disadv):", observed_diff)
```

```
Observed Precision (Advantaged): 0.6323330910601184
Observed Precision (Disadvantaged): 0.5249424405218726
Observed Difference (Adv - Disadv): 0.10739065053824581
```

In [34]:
```python
n_permutations = 10000
perm_diffs = np.zeros(n_permutations)

for i in range(n_permutations):
    shuffled_group = np.random.permutation(group_adv.values)

    perm_precision_adv = precision_score(
        y_test_final[shuffled_group],
        y_pred[shuffled_group]
    )

    perm_precision_disadv = precision_score(
        y_test_final[~shuffled_group],
        y_pred[~shuffled_group]
    )

    perm_diffs[i] = perm_precision_adv - perm_precision_disadv

p_value = np.mean(perm_diffs >= observed_diff)

print("Permutation p-value (one-sided):", p_value)
```

```
Permutation p-value (one-sided): 0.0
```

Results

The observed precision for early-game advantaged teams was substantially higher than for early-game disadvantaged teams. The permutation test produced a **p-value effectively equal to zero** ($p < 0.0001$), indicating that none of the random permutations resulted in a precision difference as large as the observed one.

Conclusion

Since the p-value is far below the 5% significance level, we **reject the null hypothesis**. There is strong statistical evidence that the model's precision is lower for teams that are behind at 10 minutes. This indicates a systematic performance disparity: the model is significantly more reliable when predicting wins for early-game advantaged teams than for early-game disadvantaged teams.

This result is consistent with the model's reliance on early-game features, which naturally provide clearer signals for teams that are already ahead and more ambiguous signals for teams attempting to recover from an early deficit.