League of Legends Pro Play Analysis: Does Early Game Gold Lead Predict Victory?

Name(s): jul187

Website Link: https://jul187-creator.github.io/lol-analysis-dsc80/

In [1]:
```
import pandas as pd
import numpy as np
from pathlib import Path

import plotly.express as px

pd.options.plotting.backend = 'plotly'

# from dsc80_utils import * # Feel free to uncomment and use this.
```

Step 1: Introduction

Research Question

In professional League of Legends, teams often emphasize the importance of "early game" - the first 15 minutes of play where gaining advantages can snowball into victory. But just how predictive is an early gold lead?

My central question: Does having a gold advantage at 15 minutes significantly increase a team's chance of winning the game?

This question matters because it can help us understand whether professional teams should prioritize aggressive early-game strategies or if comebacks from early deficits are common enough that late-game scaling is equally viable. Understanding this relationship could inform coaching decisions and help teams develop better strategies based on their early-game performance.

In [2]:
```
# Load the data (suppress the dtype warning by setting low_memory=False)
df = pd.read_csv('2024_LoL_esports_match_data_from_OraclesElixir.csv', low_memor

# Display basic information
print(f"Total rows in dataset: {len(df)}")
print(f"\nFirst few rows:")
df.head()
```

```
Total rows in dataset: 117648

First few rows:
```

Out[2]:

	gameid	datacompleteness	url	league	year
0	10660-10660_game_1	partial	https://lpl.qq.com/es/stats.shtml?bmid=10660	DCup	2023
1	10660-10660_game_1	partial	https://lpl.qq.com/es/stats.shtml?bmid=10660	DCup	2023
2	10660-10660_game_1	partial	https://lpl.qq.com/es/stats.shtml?bmid=10660	DCup	2023
3	10660-10660_game_1	partial	https://lpl.qq.com/es/stats.shtml?bmid=10660	DCup	2023
4	10660-10660_game_1	partial	https://lpl.qq.com/es/stats.shtml?bmid=10660	DCup	2023

5 rows × 164 columns

Step 2: Data Cleaning and Exploratory Data Analysis

Data Cleaning

The raw dataset contains 12 rows per game - 10 for individual players and 2 for team summary statistics. Since my research question focuses on team-level gold advantages and win rates, I'll keep only the team summary rows and remove individual player data.

I'll also:

1. Keep only rows where position is 'team'
2. Convert boolean columns from 0/1 to True/False
3. Remove incomplete games (where datacompleteness is not 'complete')
4. Create a column for gold difference at 15 minutes
5. Focus on relevant columns for the analysis

In [3]:
```
# Data Cleaning

# Step 1: Keep only team rows (remove individual player data)
teams_df = df[df['position'] == 'team'].copy()

print(f"After keeping only team rows: {len(teams_df)} rows")
print(f"This represents {len(teams_df) // 2} games (2 teams per game)")

# Step 2: Keep only complete games
teams_df = teams_df[teams_df['datacompleteness'] == 'complete'].copy()

print(f"After filtering for complete games: {len(teams_df)} rows")
```

```
# Step 3: Convert result column to boolean
teams_df['result'] = teams_df['result'].astype(bool)

# Step 4: Handle missing values in gold diff at 15
# Some games might end before 15 minutes, so golddiffat15 could be NaN
print(f"\nMissing values in golddiffat15: {teams_df['golddiffat15'].isna().sum()}")

# Keep only games that lasted at least 15 minutes
teams_df = teams_df[teams_df['golddiffat15'].notna()].copy()

print(f"After removing games shorter than 15 min: {len(teams_df)} rows")

# Step 5: Create helpful columns
teams_df['had_gold_lead_at_15'] = teams_df['golddiffat15'] > 0
teams_df['gamelength_minutes'] = teams_df['gamelength'] / 60

# Display cleaned data
print(f"\n=== Cleaned Dataset Summary ===")
print(f"Total team records: {len(teams_df)}")
print(f"Total games analyzed: {len(teams_df) // 2}")
print(f"Columns of interest: result, golddiffat15, gamelength, league, kills, de

teams_df.head()
```

```
After keeping only team rows: 19608 rows
This represents 9804 games (2 teams per game)
After filtering for complete games: 16826 rows

Missing values in golddiffat15: 4
After removing games shorter than 15 min: 16822 rows

=== Cleaned Dataset Summary ===
Total team records: 16822
Total games analyzed: 8411
Columns of interest: result, golddiffat15, gamelength, league, kills, deaths
```

Out[3]:

	gameid	datacompleteness	url	league	year	split	playoffs
190	LOLTMNT99_132542	complete	NaN	TSC	2024	Winter	0
191	LOLTMNT99_132542	complete	NaN	TSC	2024	Winter	0
202	LOLTMNT99_132665	complete	NaN	TSC	2024	Winter	0
203	LOLTMNT99_132665	complete	NaN	TSC	2024	Winter	0
214	LOLTMNT99_132755	complete	NaN	TSC	2024	Winter	0

5 rows × 166 columns

Univariate Analysis

Let's explore the distribution of some key variables.

In [ ]:
```
# Univariate Analysis: Distribution of Gold Difference at 15 minutes

fig1 = px.histogram(
    teams_df,
    x='golddiffat15',
    nbins=60,
    title='Distribution of Gold Difference at 15 Minutes',
    labels={'golddiffat15': 'Gold Difference at 15 Minutes', 'count': 'Number of
    color_discrete_sequence=['#5DADE2']
)

fig1.update_layout(
    xaxis_title='Gold Difference at 15 Minutes',
    yaxis_title='Frequency',
    showlegend=False,
    template='plotly_white',
    font=dict(size=12),
    title_font_size=16
)

fig1.add_vline(x=0, line_dash="dash", line_color="red",
               annotation_text="Even Gold", annotation_position="top")

fig1.show()

# Save for GitHub Pages
fig1.write_html('fig1_gold_distribution.html')

print(f"Mean gold diff at 15: {teams_df['golddiffat15'].mean():.2f}")
print(f"Median gold diff at 15: {teams_df['golddiffat15'].median():.2f}")
print(f"Std deviation: {teams_df['golddiffat15'].std():.2f}")
```

Interesting Aggregates

Looking at how different professional leagues compare in terms of gameplay patterns.

In [ ]:
```
# Performance by League: Analyze top 10 most active leagues

league_stats = teams_df.groupby('league').agg({
    'golddiffat15': 'mean',
    'gamelength_minutes': 'mean',
    'result': 'mean',
    'gameid': 'count'
}).reset_index()

league_stats.columns = ['league', 'avg_gold_diff_15', 'avg_game_length', 'win_ra

# Get top 10 leagues by number of games
top_10_leagues = league_stats.nlargest(10, 'num_games')

# Round values for cleaner display
```

```
top_10_leagues['avg_gold_diff_15'] = top_10_leagues['avg_gold_diff_15'].round(0)
top_10_leagues['avg_game_length'] = top_10_leagues['avg_game_length'].round(2)
top_10_leagues['win_rate'] = top_10_leagues['win_rate'].round(2)

print("Performance by League (Top 10 Most Active):\n")
print(top_10_leagues.to_string(index=False))

print("\n\nKey Observation:")
print(f"The analysis shows that while gold leads matter across all leagues,")
print(f"some regions tend to have longer games or more volatile early games than
print(
    f"For example, {top_10_leagues.iloc[0]['league']} games average {top_10_leag
print(
    f"while {top_10_leagues.iloc[-1]['league']} games average {top_10_leagues.il
print(f"This could reflect different regional playstyles and meta preferences.")
```

In [ ]:

```
# Bivariate Analysis: Gold Difference at 15 min vs Win Rate

# Create bins for gold difference
teams_df['gold_diff_bin'] = pd.cut(
    teams_df['golddiffat15'],
    bins=[-np.inf, -3000, -2000, -1000, 0, 1000, 2000, 3000, np.inf],
    labels=['< -3000', '-3000 to -2000', '-2000 to -1000', '-1000 to 0',
            '0 to 1000', '1000 to 2000', '2000 to 3000', '> 3000']
)

# Calculate win rate for each bin
win_rate_by_gold = teams_df.groupby('gold_diff_bin', observed=True).agg({
    'result': 'mean',
    'gameid': 'count'
}).reset_index()

win_rate_by_gold.columns = ['gold_diff_range', 'win_rate', 'num_games']
win_rate_by_gold['win_rate_pct'] = win_rate_by_gold['win_rate'] * 100

# Create the visualization
fig2 = px.bar(
    win_rate_by_gold,
    x='gold_diff_range',
    y='win_rate_pct',
    title='Win Rate by Gold Difference at 15 Minutes',
    labels={'gold_diff_range': 'Gold Difference Range at 15 Min',
            'win_rate_pct': 'Win Rate (%)'},
    color='win_rate_pct',
    color_continuous_scale=['#E74C3C', '#F39C12', '#F4D03F', '#52BE80', '#27AE60
    text='win_rate_pct'
)

fig2.update_traces(
    texttemplate='%{text:.1f}%',
    textposition='outside'
)

fig2.update_layout(
    xaxis_title='Gold Difference at 15 Minutes',
    yaxis_title='Win Rate (%)',
    template='plotly_white',
    font=dict(size=12),
    title_font_size=16,
    showlegend=False,
```

```
    yaxis_range=[0, 100]
)

fig2.add_hline(y=50, line_dash="dash", line_color="gray",
               annotation_text="50% (Expected if no advantage)",
               annotation_position="right")

fig2.show()

# Save for GitHub Pages
fig2.write_html('fig2_winrate_by_gold.html')

print("Win Rate by Gold Advantage:")
print(win_rate_by_gold[['gold_diff_range', 'win_rate_pct', 'num_games']])
```

Step 3: Assessment of Missingness

NMAR Analysis

Looking at the dataset, I think the `ban5` column (the 5th ban in draft phase) is a good candidate for NMAR (Not Missing At Random). This column is missing when teams choose not to use all 5 of their available bans.

The missingness here is likely related to the decision itself - teams might skip their 5th ban if they feel they've already banned the most threatening champions, or if they're confident enough in their strategy that an additional ban isn't necessary. In other words, the missing value tells us something about the team's strategic thinking, which makes it NMAR.

To make this MAR (Missing At Random) instead of NMAR, we would need additional information such as:

- The specific patch version (some patches have fewer viable champions to ban)
- Historical data on each team's ban strategy preferences
- The draft position when the ban would occur
- Team communication or interview data explaining their ban decisions

In [ ]:
```
# Permutation Test 1: Test if missingness of 'ban5' depends on 'league'
# (This should show dependency - some leagues might have different ban strategie

# Create missingness indicator
teams_df['ban5_missing'] = teams_df['ban5'].isnull()

# Get top leagues for clearer analysis
top_10_leagues = teams_df['league'].value_counts().head(10).index
test_df = teams_df[teams_df['league'].isin(top_10_leagues)].copy()
test_df = test_df.reset_index(drop=True)  # Reset index to avoid index mismatch

# Observed test statistic: TVD (Total Variation Distance) between distributions
observed_missing = test_df.groupby('league')['ban5_missing'].mean()
observed_not_missing = test_df.groupby('league')['ban5_missing'].apply(lambda x:
observed_tvd = np.sum(np.abs(observed_missing - observed_not_missing)) / 2
```

```
print(f"Test 1: Does missingness of ban5 depend on league?")
print(f"Observed TVD: {observed_tvd:.4f}")

# Perform permutation test
n_permutations = 1000
tvd_permuted = []

for _ in range(n_permutations):
    shuffled = test_df['ban5_missing'].sample(frac=1, replace=False).values
    perm_df = test_df.copy()
    perm_df['ban5_missing_shuffled'] = shuffled

    perm_missing = perm_df.groupby('league')['ban5_missing_shuffled'].mean()
    perm_not_missing = perm_df.groupby('league')['ban5_missing_shuffled'].apply(
    tvd_permuted.append(np.sum(np.abs(perm_missing - perm_not_missing)) / 2)

p_value_1 = np.mean(np.array(tvd_permuted) >= observed_tvd)
print(f"P-value: {p_value_1:.4f}")
print(
    f"Conclusion: {'Reject null' if p_value_1 < 0.05 else 'Fail to reject null'}
```

In [ ]:
```
# Permutation Test 2: Test if missingness of 'ban5' depends on 'gamelength'
# (This should show NO dependency - ban decisions happen before game length is d

# Observed difference in mean game length between missing and not missing
observed_diff = test_df[test_df['ban5_missing']]['gamelength_minutes'].mean() -
                test_df[~test_df['ban5_missing']]['gamelength_minutes'].mean()

print(f"Test 2: Does missingness of ban5 depend on game length?")
print(f"Observed difference in means: {observed_diff:.4f} minutes")

# Permutation test
diff_permuted = []
for _ in range(1000):
    shuffled = test_df['ban5_missing'].sample(frac=1, replace=False).values

    perm_diff = test_df.loc[shuffled == True, 'gamelength_minutes'].mean() - \
                test_df.loc[shuffled == False, 'gamelength_minutes'].mean()
    diff_permuted.append(perm_diff)

p_value_2 = np.mean(np.abs(diff_permuted) >= np.abs(observed_diff))
print(f"P-value: {p_value_2:.4f}")
print(
    f"Conclusion: {'Reject null' if p_value_2 < 0.05 else 'Fail to reject null'}
```

Step 4: Hypothesis Testing

I'll conduct two hypothesis tests to explore different aspects of the data:

1. Does having a gold lead at 15 minutes significantly increase win rate?
2. Do teams that secure first dragon have different average kill counts than those who don't?

In [9]:
```
### Hypothesis Test 2: First Dragon vs Kill Count

# Null Hypothesis: Teams that get first dragon have the same average kills as th
# Alternative Hypothesis: Teams that get first dragon have different average kil
```

```
# Calculate observed difference
kills_with_first_dragon = teams_df[teams_df['firstdragon'] == 1]['kills'].mean()
kills_without_first_dragon = teams_df[teams_df['firstdragon'] == 0]['kills'].mea
observed_diff_kills = kills_with_first_dragon - kills_without_first_dragon

print("=== Hypothesis Test 2: First Dragon vs Kill Count ===")
print(f"Average kills WITH first dragon: {kills_with_first_dragon:.2f}")
print(f"Average kills WITHOUT first dragon: {kills_without_first_dragon:.2f}")
print(f"Observed difference: {observed_diff_kills:.2f} kills\n")

# Permutation test
kill_differences = []

for _ in range(10000):
    # Shuffle the kills column
    shuffled_kills = teams_df['kills'].sample(frac=1, replace=False).values

    # Calculate difference under null
    with_dragon_mask = teams_df['firstdragon'].values == 1
    without_dragon_mask = teams_df['firstdragon'].values == 0
    with_dragon_mean = shuffled_kills[with_dragon_mask].mean()
    without_dragon_mean = shuffled_kills[without_dragon_mask].mean()
    kill_differences.append(with_dragon_mean - without_dragon_mean)

# Calculate p-value (two-tailed test)
p_value_test2 = np.mean(np.abs(kill_differences) >= np.abs(observed_diff_kills))

print(f"P-value: {p_value_test2:.6f}")
print(f"Conclusion: {'Reject null hypothesis' if p_value_test2 < 0.05 else 'Fail
print(
    f"Interpretation: Getting first dragon {'IS' if p_value_test2 < 0.05 else 'i
```

```
P-value: 0.000000
Conclusion: Reject null hypothesis
Interpretation: Getting first dragon IS significantly associated with different k
ill counts.
```

In [10]:

```
### Hypothesis Test 1: Gold Lead at 15 Minutes vs Win Rate

# Null Hypothesis: Teams with gold lead at 15 min have the same win rate as team
# Alternative Hypothesis: Teams with gold lead at 15 min have higher win rate

# Create binary indicator for gold lead
teams_df['has_gold_lead_15'] = teams_df['golddiffat15'] > 0

# Calculate observed difference in win rates
win_rate_with_lead = teams_df[teams_df['has_gold_lead_15']]['result'].mean()
win_rate_without_lead = teams_df[~teams_df['has_gold_lead_15']]['result'].mean()
observed_diff_wr = win_rate_with_lead - win_rate_without_lead

print("=== Hypothesis Test 1: Gold Lead vs Win Rate ===")
print(f"Win rate WITH gold lead at 15: {win_rate_with_lead:.4f} ({win_rate_with_
print(f"Win rate WITHOUT gold lead at 15: {win_rate_without_lead:.4f} ({win_rate
print(f"Observed difference: {observed_diff_wr:.4f} ({observed_diff_wr * 100:.2f

# Permutation test
n_iterations = 10000
differences = []
```

```
for _ in range(n_iterations):
    # Shuffle the result column
    shuffled_results = teams_df['result'].sample(frac=1, replace=False).values

    # Calculate difference under null hypothesis
    with_lead_mean = shuffled_results[teams_df['has_gold_lead_15'].values].mean(
    without_lead_mean = shuffled_results[~teams_df['has_gold_lead_15'].values].m
    differences.append(with_lead_mean - without_lead_mean)

# Calculate p-value
p_value_test1 = np.mean(np.array(differences) >= observed_diff_wr)

print(f"P-value: {p_value_test1:.6f}")
print(f"Conclusion: {'Reject null hypothesis' if p_value_test1 < 0.05 else 'Fail
print(
    f"Interpretation: Having a gold lead at 15 minutes {'DOES' if p_value_test1

# Visualization of permutation distribution
fig_perm = px.histogram(x=differences, nbins=50,
                        title='Permutation Test: Distribution of Win Rate Differ
                        labels={'x': 'Difference in Win Rates', 'y': 'Frequency'
fig_perm.add_vline(x=observed_diff_wr, line_dash="dash", line_color="red",
                   annotation_text=f"Observed: {observed_diff_wr:.4f}")
fig_perm.update_layout(template='plotly_white')
fig_perm.show()

# Save for GitHub Pages
fig_perm.write_html('fig3_hypothesis_test.html')
```

```
=== Hypothesis Test 1: Gold Lead vs Win Rate ===
Win rate WITH gold lead at 15: 0.7284 (72.84%)
Win rate WITHOUT gold lead at 15: 0.2718 (27.18%)
Observed difference: 0.4567 (45.67 percentage points)
P-value: 0.000000
Conclusion: Reject null hypothesis
Interpretation: Having a gold lead at 15 minutes DOES significantly affect win ra
te.
```

Step 5: Framing a Prediction Problem

Problem Definition

Prediction Task: Predict whether a team will win or lose a game based on their performance at the 15-minute mark.

Type: Binary Classification

Response Variable: `result` (1 = Win, 0 = Loss)

Evaluation Metric: Accuracy (primary) and F1-Score (secondary)

- Accuracy is appropriate here since wins and losses are naturally balanced (every game has exactly one winner and one loser)
- F1-Score gives us additional insight into the precision-recall tradeoff

Time of Prediction: 15 minutes into the game

At this point in the match, we know:

- Gold differential, experience differential, and CS differential
- Kill, death, and assist counts

- Early objective control (first blood, first dragon, first herald, towers destroyed)

At this point, we do NOT know:

- Final game length
- Late-game objectives (baron, elder dragon)
- Final gold totals and kill counts

Why This Matters:

In professional League of Legends, coaches and analysts want to assess win probability mid-game to make crucial strategic decisions. Should the team play aggressively to close out the game early? Or should they focus on scaling for late-game team fights? This model helps answer these questions by quantifying how much early-game performance affects the final outcome.

In [11]:
```
# Prepare features that would be available at 15 minutes

# Features available at 15 minutes
features_at_15 = [
    'golddiffat15',
    'xpdiffat15',
    'csdiffat15',
    'killsat15',
    'deathsat15',
    'assistsat15',
    'firstblood',
    'firstdragon',
    'firstherald',
    'towers'  # towers destroyed by 15 minutes
]

# Create feature matrix X and target y
X = teams_df[features_at_15].copy()
y = teams_df['result'].copy()

print(f"Feature matrix shape: {X.shape}")
print(f"Target variable shape: {y.shape}")
print(f"\nClass distribution:")
print(y.value_counts(normalize=True))
```

```
Feature matrix shape: (16822, 10)
Target variable shape: (16822,)

Class distribution:
result
True     0.5
False    0.5
Name: proportion, dtype: float64
```

Step 6: Baseline Model

Model Description

For my baseline model, I'll use a simple **Logistic Regression** with just two features:

1. **golddiffat15** (quantitative) - The gold difference at 15 minutes
2. **firstblood** (nominal/binary) - Whether the team got first blood

These represent the most basic early-game metrics: economic advantage and first kill.

Model Assessment

The baseline model achieves solid performance with just two simple features. With around 73% accuracy, we're doing significantly better than random guessing (which would be 50%).

What's interesting is that the gold difference feature carries most of the predictive power - the positive coefficient shows that teams ahead in gold are much more likely to win. First blood also helps, but its effect is smaller compared to the overall economic advantage by 15 minutes.

This makes intuitive sense: gold translates directly into item advantages, which affect fights throughout the game. While first blood gives an early lead, it's the sustained gold advantage over 15 minutes that really matters.

However, there's definitely room for improvement. We're only using two features when we have access to much richer information at the 15-minute mark.

In [12]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_mat

# Baseline model: Use only 2 features
baseline_features = ['golddiffat15', 'firstblood']
X_baseline = teams_df[baseline_features].copy()
y_baseline = teams_df['result'].copy()

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_baseline, y_baseline, test_size=0.25, random_state=42
)

print(f"Training set size: {len(X_train)}")
print(f"Test set size: {len(X_test)}\n")

# Train baseline model
baseline_model = LogisticRegression(max_iter=1000, random_state=42)
baseline_model.fit(X_train, y_train)

# Make predictions
y_train_pred = baseline_model.predict(X_train)
y_test_pred = baseline_model.predict(X_test)

# Evaluate
train_accuracy = accuracy_score(y_train, y_train_pred)
```

```
test_accuracy = accuracy_score(y_test, y_test_pred)

print("=== Baseline Model Performance ===")
print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"\nTest Set Classification Report:")
print(classification_report(y_test, y_test_pred, target_names=['Loss', 'Win']))

# Feature importance (coefficients)
print("\nFeature Coefficients:")
for feature, coef in zip(baseline_features, baseline_model.coef_[0]):
    print(f"  {feature}: {coef:.6f}")
```

```
Training set size: 12616
Test set size: 4206

=== Baseline Model Performance ===
Training Accuracy: 0.7284
Test Accuracy: 0.7306

Test Set Classification Report:
              precision    recall  f1-score   support

        Loss       0.72      0.74      0.73      2073
         Win       0.74      0.72      0.73      2133

    accuracy                           0.73      4206
   macro avg       0.73      0.73      0.73      4206
weighted avg       0.73      0.73      0.73      4206


Feature Coefficients:
  golddiffat15: 0.000536
  firstblood: -0.117870
```

Step 7: Final Model

Improvements and Rationale

To improve the baseline, I'm making several enhancements:

Added Features:

1. `xpdiffat15` - Experience differential (helps with level advantages)
2. `csdiffat15` - CS differential (farming efficiency)
3. `killsat15`, `deathsat15`, `assistsat15` - Combat statistics
4. `firstdragon` - Early objective control
5. `firstherald` - Another key early objective
6. `towers` - Tower control (map pressure)

Why These Features Help:

- XP and CS differentials capture farming and level advantages beyond just gold
- Kill/death/assist stats show early game combat success

- Objective control (dragon, herald, towers) indicates map control and team coordination
- These features together paint a complete picture of early-game dominance

Model Selection: Random Forest

- Can capture non-linear relationships between features
- Handles feature interactions automatically
- Robust to outliers and doesn't require feature scaling

Hyperparameter Tuning: Using GridSearchCV with 5-fold cross-validation to find optimal parameters.

In [13]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

### Improvements and Rationale

# For the final model, I'm adding several enhancements:
#
# **Added Features:**
# 1. xpdiffat15 - Experience differential (level advantages)
# 2. csdiffat15 - CS differential (farming efficiency)
# 3. killsat15, deathsat15, assistsat15 - Combat statistics
# 4. firstdragon - Early objective control
# 5. firstherald - Another key early objective
# 6. towers - Tower control (map pressure)
#
# **Model Selection:** Random Forest
# - Can capture non-linear relationships between features
# - Handles feature interactions automatically
# - Robust to outliers and doesn't require feature scaling

# Prepare full feature set
final_features = [
    'golddiffat15',
    'xpdiffat15',
    'csdiffat15',
    'killsat15',
    'deathsat15',
    'assistsat15',
    'firstblood',
    'firstdragon',
    'firstherald',
    'towers'
]

X_final = teams_df[final_features].copy()
y_final = teams_df['result'].copy()

# Split data
X_train_final, X_test_final, y_train_final, y_test_final = train_test_split(
    X_final, y_final, test_size=0.25, random_state=42
)

print(f"Training set: {len(X_train_final)} games")
print(f"Test set: {len(X_test_final)} games")
```

```
print(f"Number of features: {len(final_features)}\n")

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10]
}

print("Performing GridSearchCV (this may take a few minutes)...")
rf = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(rf, param_grid, cv=5, scoring='accuracy', n_jobs=-1,
grid_search.fit(X_train_final, y_train_final)

print(f"\nBest hyperparameters found:")
print(f"  n_estimators: {grid_search.best_params_['n_estimators']}")
print(f"  max_depth: {grid_search.best_params_['max_depth']}")
print(f"  min_samples_split: {grid_search.best_params_['min_samples_split']}")
print(f"\nBest cross-validation score: {grid_search.best_score_:.4f}")

# Use best model
final_model = grid_search.best_estimator_

# Make predictions
y_train_pred_final = final_model.predict(X_train_final)
y_test_pred_final = final_model.predict(X_test_final)

# Evaluate
train_accuracy_final = accuracy_score(y_train_final, y_train_pred_final)
test_accuracy_final = accuracy_score(y_test_final, y_test_pred_final)

print("\n=== Final Model Performance ===")
print(f"Training Accuracy: {train_accuracy_final:.4f}")
print(f"Test Accuracy: {test_accuracy_final:.4f}")
print(
    f"Improvement over baseline: {(test_accuracy_final - test_accuracy):.4f} (+{

print("Test Set Classification Report:")
print(classification_report(y_test_final, y_test_pred_final, target_names=['Loss

# Feature importance
feature_importance = pd.DataFrame({
    'feature': final_features,
    'importance': final_model.feature_importances_
}).sort_values('importance', ascending=False)

print("\nFeature Importance (Top 5):")
for idx, row in feature_importance.head(5).iterrows():
    print(f"  {row['feature']}: {row['importance'] * 100:.1f}%")
```

```
Training set: 12616 games
Test set: 4206 games
Number of features: 10

Performing GridSearchCV (this may take a few minutes)...

Best hyperparameters found:
  n_estimators: 200
  max_depth: 20
  min_samples_split: 2

Best cross-validation score: 0.9649

=== Final Model Performance ===
Training Accuracy: 1.0000
Test Accuracy: 0.9662
Improvement over baseline: 0.2356 (+23.56%)

Test Set Classification Report:
              precision    recall  f1-score   support

        Loss       0.99      0.95      0.97      2073
         Win       0.95      0.99      0.97      2133

    accuracy                           0.97      4206
   macro avg       0.97      0.97      0.97      4206
weighted avg       0.97      0.97      0.97      4206


Feature Importance (Top 5):
  towers: 71.9%
  golddiffat15: 10.6%
  xpdiffat15: 6.4%
  csdiffat15: 4.8%
  assistsat15: 1.6%
```

Step 8: Fairness Analysis

Research Question

Does the model perform fairly across different types of games?

Specifically, I'll examine: **Does the model perform differently for short games (< 30 minutes) vs. long games (≥ 30 minutes)?**

This matters because short games might have more extreme early-game statistics, while long games could have more comebacks and less predictable outcomes.

Test Setup

Groups:

- Group X: Short games (< 30 minutes)
- Group Y: Long games (≥ 30 minutes)

Hypotheses:

- Null Hypothesis (H_0): The model's accuracy is the same for short and long games
- Alternative Hypothesis (H_1): The model's accuracy differs between short and long games

Method:

- Evaluation Metric: Accuracy
- Test Statistic: Difference in accuracy between groups
- Permutation test with 1,000 iterations
- Significance Level: $\alpha = 0.05$

In [14]:
```
### Research Question

# Does the model perform fairly across different types of games?
# Specifically: Does the model perform differently for short games (< 30 minutes
# vs. long games (≥ 30 minutes)?
#
# This matters because short games might have more extreme early-game statistics
# while long games could have more comebacks and less predictable outcomes.

### Fairness Test Setup

# Create game length groups
test_df = pd.DataFrame({
    'gamelength_minutes': teams_df.loc[X_test_final.index, 'gamelength_minutes']
    'prediction': y_test_pred_final,
    'actual': y_test_final
})

test_df['game_type'] = test_df['gamelength_minutes'].apply(
    lambda x: 'short' if x < 30 else 'long'
)

# Calculate accuracy for each group
short_games = test_df[test_df['game_type'] == 'short']
long_games = test_df[test_df['game_type'] == 'long']

accuracy_short = (short_games['prediction'] == short_games['actual']).mean()
accuracy_long = (long_games['prediction'] == long_games['actual']).mean()
observed_diff = accuracy_short - accuracy_long

print("=== Fairness Analysis ===")
print(f"\nGroup X (Short Games, < 30 min): {len(short_games)} games")
print(f"Group Y (Long Games, ≥ 30 min): {len(long_games)} games")
print(f"\nAccuracy for short games: {accuracy_short:.4f} ({accuracy_short * 100:
print(f"Accuracy for long games: {accuracy_long:.4f} ({accuracy_long * 100:.1f}%
print(f"Observed difference: {observed_diff:.4f} ({observed_diff * 100:.1f}%)\n"

# Permutation test
# Null Hypothesis: The model's accuracy is the same for short and long games
# Alternative Hypothesis: The model's accuracy differs between short and long ga

n_permutations = 1000
differences = []

for _ in range(n_permutations):
```

```
    # Shuffle game type labels
    shuffled_types = test_df['game_type'].sample(frac=1, replace=False).values

    # Calculate accuracy difference under null
    short_mask = shuffled_types == 'short'
    long_mask = shuffled_types == 'long'

    short_accuracy = (test_df['prediction'].values[short_mask] == test_df['actua
    long_accuracy = (test_df['prediction'].values[long_mask] == test_df['actual'

    differences.append(short_accuracy - long_accuracy)

# Calculate p-value (two-tailed test)
p_value_fairness = np.mean(np.abs(differences) >= np.abs(observed_diff))

print("Hypothesis Test Results:")
print(f"Null Hypothesis: Model accuracy is the same for both groups")
print(f"Alternative Hypothesis: Model accuracy differs between groups")
print(f"Significance Level: α = 0.05")
print(f"\nP-value: {p_value_fairness:.4f}")
print(f"\nConclusion: {'Reject' if p_value_fairness < 0.05 else 'Fail to reject'

if p_value_fairness < 0.05:
    print(f"The model performs DIFFERENTLY for short vs long games (statisticall
else:
    print(f"The model performs FAIRLY across both game lengths (no significant d

print(f"\nInterpretation: The {abs(observed_diff) * 100:.1f}% difference in accu
print(f"short and long games is {'NOT ' if p_value_fairness >= 0.05 else ''}stat
print(f"The model's predictions are {'equally reliable' if p_value_fairness >= 0
print(f"whether you're looking at a quick 25-minute stomp or a 40-minute slugfes

# Visualization of permutation distribution
fig_fairness = px.histogram(
    x=differences,
    nbins=50,
    title='Fairness Analysis: Permutation Test Distribution',
    labels={'x': 'Difference in Accuracy (Short - Long)', 'y': 'Frequency'}
)
fig_fairness.add_vline(
    x=observed_diff,
    line_dash="dash",
    line_color="red",
    annotation_text=f"Observed: {observed_diff:.4f}"
)
fig_fairness.update_layout(template='plotly_white')
fig_fairness.show()

# Save for GitHub Pages (optional - can be used for documentation)
# fig_fairness.write_html('fig_fairness.html')
```

```
=== Fairness Analysis ===

Group X (Short Games, < 30 min): 1833 games
Group Y (Long Games, ≥ 30 min): 2373 games

Accuracy for short games: 0.9924 (99.2%)
Accuracy for long games: 0.9461 (94.6%)
Observed difference: 0.0463 (4.6%)
Hypothesis Test Results:
Null Hypothesis: Model accuracy is the same for both groups
Alternative Hypothesis: Model accuracy differs between groups
Significance Level: α = 0.05

P-value: 0.0000

Conclusion: Reject the null hypothesis
The model performs DIFFERENTLY for short vs long games (statistically significan
t)

Interpretation: The 4.6% difference in accuracy between
short and long games is statistically significant.
The model's predictions are NOT equally reliable
whether you're looking at a quick 25-minute stomp or a 40-minute slugfest.
```