Quantifying Professional Player Impact: A Comparative Analysis of Linear and Probit Models in Binary Outcome Metrics.

Since football's "statistical revolution" in the early to mid-2010s, casual football fans have been confronted with increasingly complex metrics trying to persuade them that certain players are worth getting excited about, or that the signing their club narrowly missed out on wasn't good enough anyway. Consequently, it's understandable that various organisations tasked with making football statistics accessible to the masses have cultivated their own iterations of player performance indices, which encompass all the numbers which are impossible to understand when viewed raw. The problem with this situation is that depending on what sports data platform you prefer, you are offered a very different perspective on what makes a player good. A study from the Journal of Sports Sciences showed that different apps (in their study it was WhoScored, FotMob and SofaScore) give different weightings to different aspects of player performances, and thus the ratings from those apps are not interchangeable (Ball et al., 2025). The very existence of a different rating depending on the app you're using proves that these ratings are not objective. Another issue with these ratings is that they are often centred around an arbitrary rating. For example, SofaScore ratings have a "starting point" of 6.5 and a minimum of 3.0 (SofaScore, 2026), both of which appear to have been selected without any particular justification. In this essay I will assess whether it is feasible to create an objective player rating using either a Linear Probability Model (LPM) or a Probit model, and analyse what metrics the models determine the most important.

When entering into the data collection process of this project, I knew the dependant variable needed to be some form of dummy variable. My initial idea was that for each data entry (each representing one player's unique match performance), I would have a dummy variable for whether that player's team won the match. This idea raised some issues, one of which is that the model would be biased towards players who play for teams looking to dominate play. To explain this problem, picture a defensive player who has played extremely well for their team, but has been very busy. The model might see that their team has needed to defend a lot and then predict that they either drew or lost that game, regardless of how effective that specific player was. This is problematic because I want a situation where the model can give any player a high score, no matter what team they play for. Having it based on the "won" dummy would also mean that the probabilities would average around 0.375, which is the general win rate of football games, which I deemed undesirable for my performance metric. My solution to both of these problems simultaneously is the "overperform" dummy. It's based on the idea that in every football game there is a favourite and that team "should" win the match in theory. The overperform dummy is a one for the favourite if they win the match, but zero if they draw or lose, and one for the underdog if they win or draw, but zero if they lose.

This means that in our earlier example, if the defender plays well the model might think they are the underdog, and that they have a higher chance of getting a draw because of the defender's performance. This did slightly complicate the data collection because I needed access to pre-game match odds, luckily my primary source for all the data used (SofaScore) also had that information available. My method to collect all that data from SofaScore was that I made a python script (listed in the appendix) to access SofaScore's API for all of the men's football games spanning from August 2022 – June 2025, in the top divisions of England, Spain, Italy, Germany and France, "commonly referred to as the 'big 5'" (Deloitte, 2024). The script adds in the relevant overperform value, then parses all of the player data from those matches to a .JSON file, which is then converted to a .csv format for use in Stata.

Once I had the data loaded into Stata, I ran a quick `describe` command to ensure all of the variables made sense from the perspective of what my aim was. It's important to mention that in my python script, there is a filter to only collect statistical categories for outfield players. This was so that there weren't any goalkeeper-only variables which would confuse the model. When I did a review of the categories, analysis revealed there was one labelled "goodHighClaim" and this sounded like something that only applied to goalkeepers. I performed an initial sort in the data editor and found that there was only one instance where anyone had a non-zero value for this category. The way that the data entries are labelled in my script is that the index is given a value like "845273-11392888", the first number being the SofaScore player ID and the second being the match ID. Using this I was able to locate the match where the high claim occurred, and I discovered it was in a game where the goalkeeper was shown a red card, meaning an outfield player was forced to go in goal for the remainder of the match. This is uncommon in games, so I identified this as an outlier and removed the variable from the selection.

After that decision, I used the remaining variables and ran a simple regression, with the intention of performing a Variance Inflation Factor test afterwards, to detect occurrences of multicollinearity. Standard econometric literature generally states that a VIF threshold of ten or above was used to identify severe multicollinearity (Kutner et al., 2004). Table 1 shows the result of my VIF test, filtered by the categories that were near or above ten:

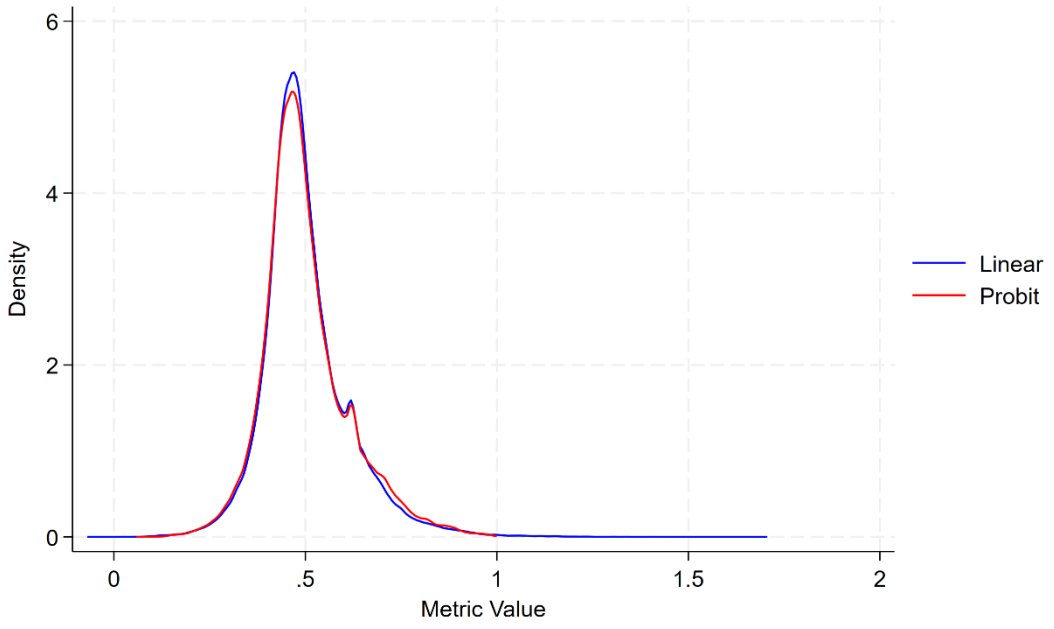
Table 1:

Variable	VIF	1/VIF
accuratepass	2663.18	0.000375
accurateownhalfpasses	1065.97	0.000938
accurateoppositionhalfpasses	777.92	0.001285
duelwon	510.21	0.001960
aerialwon	137.71	0.007262
duellost	136.45	0.007329
touches	111.12	0.008999
wasfouled	74.36	0.013447
wontackle	67.59	0.014795
woncontest	67.57	0.014799
possessionlostctrl	67.25	0.014869
losttackle	42.46	0.023553
aeriallost	33.41	0.029931
inaccurateoppositionhalfpasses	32.86	0.030436
dispossessed	19.50	0.051274
lostcontest	19.27	0.051883
fouls	17.91	0.055822
inaccuratepass	15.92	0.062799
challengelost	15.25	0.065555
minutesplayed	9.64	0.103755
inaccurateownhalfpasses	9.23	0.108343

After viewing these results, I deduced there were four distinct groups of variables that were likely to be correlated with each other: accurate passes, inaccurate passes, duel/challenge/contest/aerials won, and duel/challenge/contest/aerials lost. The variables I decided to keep were: accurateOwnHalfPasses, accurateOppositionHalfPasses, possessionLostControl, duelWon, duelLost, wonContest, lostContest, and minutesPlayed. Once I'd removed the others, I ran another VIF test and consequently, the only variable with a VIF above six was minutesPlayed (which stayed at approximately the same level) which made sense because that was likely to be correlated with every other variable. To me, these new results were enough to say I had removed the extreme multicollinearity that was present before.

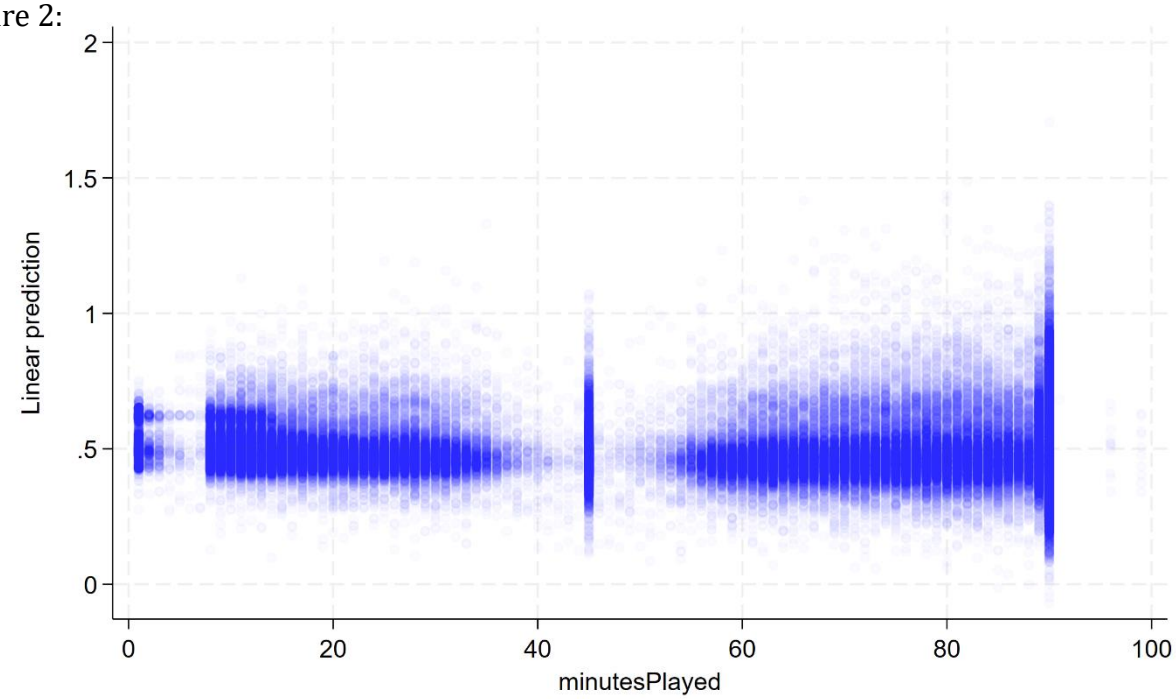
The final piece of data cleaning I did relates to something I spotted after running both a linear and a Probit regression on my selected variables. Before I started analysing the coefficient estimates and evaluating my findings, I elected to plot the distribution of values that the models were generating. Doing so yielded the graph in Figure 1:

Figure 1:



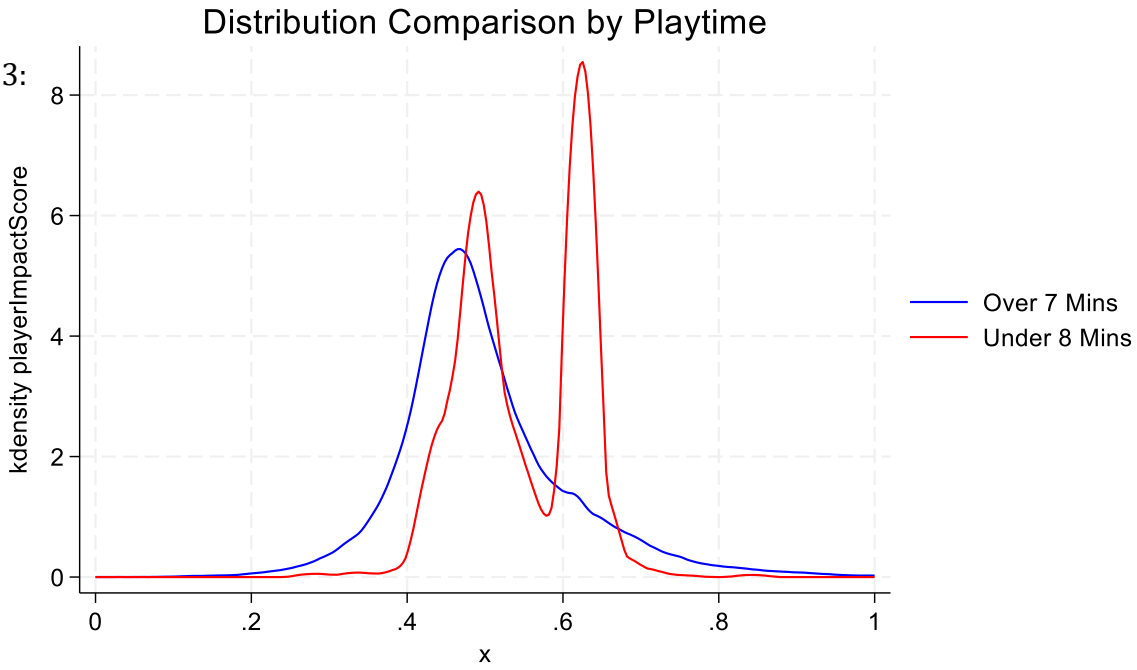
As is immediately evident, there is a "bump" around the 0.6 mark. Upon further investigation into the underlying causal factors, I noticed in the linear regression results that the intercept value was 0.637. I then hypothesised that there was a substantial group of players who weren't on the pitch for very long, and so didn't have large values in any of their categories, leading the model to predict their rating as very close to the intercept. To test this theory, I had the program display a scatter plot for minutesPlayed against predicted probability. The results of which are displayed in Figure 2:

Figure 2:



This graph seemed to suggest that there were irregularities with the ratings of players who had a minutesPlayed of less than eight. This was enough evidence for me to accept that my initial theory was promising, but I conducted an additional test to empirically demonstrate it further. I instructed Stata to compare the rating distribution of players with more and less than eight minutes played, only using one of the models. Figure 3 shows the outcome:

Figure 3:



The size of the spike around the intercept value and the fact that the blue line's distribution looked much more normally distributed confirmed my earlier suspicions. My solution to this problem was to remove all of the entries where minutesPlayed was less than eight.

The following section provides a comparison between Linear Prediction and Probit models, starting with the limitations of LPMs. The foremost problem from this project's perspective is that by definition, LPMs predict linearly. This means that on occasion, they return values that are above one or below zero due to the constant marginal effect of the coefficient estimates, leading to impossible probabilities. After review, I discovered that in my model's predictions, there were predicted overperform probabilities as high as 1.7, and as low as -0.07. The primary function of Probit models is to solve the constant marginal effect issue. It assumes that an additional unit of a given variable exerts a more significant influence on a player with an average probability of overperforming than on a player whose probability is already near certain. This means that the coefficient values in the regression results aren't as simple to interpret as they are with LPMs. The reason being that the raw coefficient values actually represent a change in Z-value not a change in total probability, this ensures that the effect magnitude varies depending on the "current" Z-value. This also remedies the impossible probabilities problem because the model maps the linear combination of predictors onto a cumulative standard normal distribution. Since the range of this distribution is bound between one and zero, it is mathematically impossible to predict a value beyond these points. It's now evident that in terms of generating a robust player rating which fulfils my initial goal, the Probit model's dynamics seem to make it the ideal candidate. To test this empirically, I used the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) to compare the two and determine which had the best fit. Below are the Stata results from running these tests:

Table 2:

Model	N	ll(null)	ll(model)	df	AIC	BIC
LPM	**150,015**	**-108874.8**	**-105025.8**	**47**	**210145.7**	**210611.8**

Model	N	ll(null)	ll(model)	df	AIC	BIC
Probit	**150,015**	**-103977.6**	**-100011.9**	**47**	**200117.9**	**200584**

The information criteria results provide compelling evidence that the Probit model is statistically superior to the LPM. Econometric literature suggests that a difference of roughly 10,000 in both AIC and BIC tests is more than enough to show the superiority of the Probit model in accuracy and reliability of fit (Burnham and Anderson, 2004). My final step before interpreting the results of the regressions was addressing

heteroskedacity. In LPMs, heteroskedacity is a mathematical certainty because the variance of the error term is tied directly to the value of the variables. In Probit models, it's not certain, but has larger ramifications as it affects the actual coefficient estimates not just the standard errors. To address this, I ran a Breusch-Pagan test in Stata, and the results led me to use robust standard errors for both of my models as a fix for heteroskedacity.

Now that the model is complete and I know that the Probit model has a better fit than the LPM, I am prepared to interpret the Stata regression output. As discussed earlier, the pure coefficient values that Stata returns when regressing Probit models cannot be evaluated in the same way as linear regression results. The solution is to run the `margins` command in Stata, which converts the regression results into the Average Marginal Effect (AME) of each feature in order to see the likely effect of an extra unit on the typical player's predicted probability. The model uses approximately 45 independent variables, so it would be unreasonable to discuss all of them in this essay. I have isolated the ones I believe to be the most interesting and which the model gave the most weight to in Table 3 below: (I have also included the mean values for context)

Table 3:

Feature	AME	Z-value	Dataset Average
Goals	0.2395	35.07	0.097
Expected Assists	0.0819	5.16	0.067
Big Chances Created	0.1055	22.97	0.11
Key Passes	0.0178	9.77	0.67
Accurate Passes (Own Half)	-0.0004897	-2.92	12.4
Accurate Passes (Opposing Half)	0.001547	8.33	12.44
Accurate Long Balls	-0.005114	-4.48	1.3
Inaccurate Long Balls	0.0017	1.46	1.3
Pass Success Rate	-0.0803	-5.4	0.79
Total Clearances	0.0259	30.08	1.33
Ball Recovery	0.00812	12.65	3.29
Own Goals	-0.2559	-9.98	0.0028
Error Led to Shot	-0.108	-13.27	0.024
Red Card	-0.157	-9.53	0.0064
Big Chances Missed	0.0269	5.21	0.094

The logical place to begin this dissection is the features with the highest AME. As expected, Goals and Own Goals lead the way in terms of magnitude of marginal effect. Goals are given an AME of ~0.24, a value which is similar to the LPM coefficient. This exemplifies why LPMs are inadequate for this purpose because if a player scores four goals in a game (which is unlikely but happens occasionally) that effectively adds an entire 100% to the player's existing probability. Another interesting characteristic is that the AME magnitude for Own Goals is slightly higher than it is for goals in the opposition net, even though they have the same effect on the scoreline. It makes sense that worse teams would score more own goals in theory, so it's possible this is the model accounting for that. A further notable finding is that the Z-value for Total Clearances is the second highest in the entire model, meaning it's near-certain that clearances have a positive impact on match result. This demonstrates that the factor I raised earlier about the model giving underdogs a fair rating is in effect. Another point of interest is that the coefficient for Big Chances Missed is positive. This suggests that having the chance in the first place, even if you miss it, is an indication that the team as a whole are creating opportunities. Alternatively, it could imply that the player in question has good positioning and is likely to get another chance. Finally, I want to examine what in my assessment is the most intriguing section of the AME results, which is the interaction between different aspects of passing statistics. There are three variables which combine to make up what I will be referring to as "creative output" (Big Chances Created, Expected Assists and Key Passes). These all have positive and relatively large marginal effects, but what I think is interesting is that many of the statistics I would associate with slow, calculated build up (Own Half Accurate Passes, Accurate Long Balls, Pass Success Rate) all have negative AMEs. Perhaps what the model has identified is that passing the ball around without producing any creative output has a negative impact on your chances of getting a positive result. Further evidence for this is that the marginal effect for Inaccurate Long Balls is positive, which seems perplexing until you consider it in the context of the creative output hypothesis. It's likely that the model partially uses Inaccurate Long Balls to determine which players are taking more risks with their passing, in an attempt to unlock the opposition defence. Evidently generating creative output is better than a misplaced pass, but it's interesting that the model still identifies it as encouraging.

In conclusion, I believe it's absolutely feasible to use an LPM, but even better to use a Probit model, to develop an objective, understandable player performance metric. Analysis of AME values suggests the model is effective at identifying "underrated" players at underdog teams and "star players" at established teams, from both attacking and defending perspectives, something that mainstream performance metrics struggle with. Although the model likely suffers from the common problem of omitted variables due to imperfect information (e.g. managerial instructions, ability), it could possibly be used as an efficient way to scout players using easily accessible data.

Bibliography:

Ball, J., Huynh, M. and Varley, M.C. (2025) 'Comparing player rating systems as a metric for assessing individual performance in soccer', *Journal of Sports Sciences*, 43(7), pp. 676-686.

Burnham, K.P. and Anderson, D.R. (2004) 'Multimodel inference: understanding AIC and BIC in model selection', *Sociological Methods & Research*, 33(2), pp. 261-304.

Deloitte (2024) *Annual Review of Football Finance 2024*. London: Deloitte LLP.

Kutner, M.H., Nachtsheim, C.J., Neter, J. and Li, W. (2004) *Applied Linear Statistical Models*. 5th edn. Boston: McGraw-Hill Irwin.

SofaScore (2026) *Professional Football Performance Data (August 2022 – June 2025)* [Data set]. Available via SofaScore API (Accessed: 3 January 2026).

SofaScore (2026) *SofaScore Statistical Ratings Explained*. Available at: https://corporate.sofascore.com/about/rating (Accessed: 9 January 2026).

Appendix:

Python Script:

```
import os
import json
import time
import random
import hashlib
import requests
import pandas as pd
from pathlib import Path
from threading import Lock

CACHE_DIR = Path("./cache")
CACHE_TTL = 60 * 60 * 6
REQUESTS_PER_SECOND = 1
MAX_RETRIES = 5
BACKOFF_BASE = 2.0
USER_AGENT = (
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
    "AppleWebKit/537.36 (KHTML, like Gecko) "
    "Chrome/121.0 Safari/537.36"
)

_lock = Lock()
_last_request_time = 0


def _hash_url(url: str) -> str:
    cleaned = url.replace("https://", "").replace("http://", "")
    safe = cleaned.replace("/", "_").replace("?", "_").replace("&", "_").replace(":", "_").replace("=", "_")
    h = hashlib.sha256(url.encode("utf-8")).hexdigest()[:8]
    return f"{safe}_{h}"


def _cache_path(url: str) -> Path:
    CACHE_DIR.mkdir(exist_ok=True)
    return CACHE_DIR / f"{_hash_url(url)}.json"


def _load_from_cache(url: str, cache_ttl: float):
    path = _cache_path(url)
    if not path.exists():
        return None
    try:
        with open(path, "r", encoding="utf-8") as f:
```

```
            cached = json.load(f)
            age = time.time() - cached["timestamp"]
            if cache_ttl == 0 or age < cache_ttl:
                return cached["status"], cached["headers"], cached["text"]
    except Exception:
        return None
    return None


def _save_to_cache(url: str, status, headers, text):
    path = _cache_path(url)
    tmp_path = path.with_suffix(".tmp")
    try:
        with open(tmp_path, "w", encoding="utf-8") as f:
            json.dump(
                {
                    "timestamp": time.time(),
                    "status": status,
                    "headers": headers,
                    "text": text,
                },
                f,
            )
        os.replace(tmp_path, path)
    except Exception as e:
        print(f"[safe_requests] Cache save failed for {url}: {e}")


def _rate_limit():
    global _last_request_time
    with _lock:
        now = time.time()
        elapsed = now - _last_request_time
        delay_needed = max(0, (1.0 / REQUESTS_PER_SECOND) - elapsed)
        if delay_needed > 0:
            time.sleep(delay_needed + random.uniform(0, delay_needed * 0.3))
        _last_request_time = time.time()


def safe_request(url: str, cache_ttl: float = CACHE_TTL, defaultSession=None, **kwargs):
    cached = _load_from_cache(url, cache_ttl)
    if cached:
        status, headers, text = cached
        resp = requests.Response()
        resp.status_code = status
        resp._content = text.encode("utf-8")
        resp.headers = headers
        resp.url = url
        resp._from_cache = True
        print(f"[safe_requests] Cache hit: {url}")
        return resp

    session = requests.Session() if defaultSession is None else defaultSession
    session.headers.update({"User-Agent": USER_AGENT})
    attempt = 0

    while attempt < MAX_RETRIES:
        _rate_limit()
        try:
            resp = session.get(url, **kwargs)
            if resp.status_code == 200:
                _save_to_cache(url, resp.status_code, dict(resp.headers), resp.text)
                print(f"[safe_requests] Fetched OK: {url}")
                return resp
            elif resp.status_code in (403, 429) or 500 <= resp.status_code < 600:
                delay = BACKOFF_BASE ** attempt + random.uniform(0, 1)
                print(
                    f"[safe_requests] Got {resp.status_code}, retrying in {delay:.1f}s..."
                )
                time.sleep(delay)
                attempt += 1
            else:
                print(f"[safe_requests] Non-retryable {resp.status_code} for {url}")
                return resp
        except requests.RequestException as e:
            delay = BACKOFF_BASE ** attempt + random.uniform(0, 1)
            print(f"[safe_requests] Exception {e}, retrying in {delay:.1f}s...")
            time.sleep(delay)
            attempt += 1
    raise RuntimeError(f"[safe_requests] Failed after {MAX_RETRIES} attempts: {url}")


def jsonRequest(url, cache_ttl=None, defaultSession=None):
    if cache_ttl is None:
        resp = safe_request(url, defaultSession=defaultSession)
    else:
        cache_ttl = cache_ttl * 3600
        resp = safe_request(url, cache_ttl=cache_ttl, defaultSession=defaultSession)
```

```
    return resp.json()


def zeroDivide(numerator, denominator):
    if denominator == 0:
        return 0
    return numerator / denominator


def isOverperform(oddsData, home=True):
    data = oddsData
    for odds in data:
        if odds['marketName'] == "Full time":
            matchOdds = odds['choices']
            break
    total = 0
    for odd in matchOdds:
        odd['percentageValue'] = 1 / (
                1 + int(odd['fractionalValue'].split("/")[0]) / int(odd['fractionalValue'].split("/")[1]))
        total += odd['percentageValue']
    for odd in matchOdds:
        odd['percentageValue'] /= total
    homeOdds = matchOdds[0]['percentageValue']
    drawOdds = matchOdds[1]['percentageValue']
    awayOdds = matchOdds[2]['percentageValue']
    if home:
        if homeOdds > awayOdds:
            favourite = True
        else:
            favourite = False
        if (matchOdds[0]['winning'] and favourite == False) or (matchOdds[1]['winning'] and favourite == False):
            return 1
        elif matchOdds[0]['winning'] and favourite:
            return 1
        else:
            return 0
    else:
        if homeOdds < awayOdds:
            favourite = True
        else:
            favourite = False
        if (matchOdds[2]['winning'] and favourite == False) or (matchOdds[1]['winning'] and favourite == False):
            return 1
        elif matchOdds[2]['winning'] and favourite:
            return 1
        else:
            return 0


def isSubYellowRedPenalty(playerId, incidents):
    isSub = 0
    isYellow = 0
    isRed = 0
    penaltyScored = 0
    for incident in incidents:
        if incident['incidentType'] == 'substitution':
            if 'playerIn' in incident:
                if incident['playerIn']['id'] == playerId:
                    isSub = 1
        elif incident['incidentType'] == 'card' and 'player' in incident:
            if incident['incidentClass'] == 'yellow':
                if incident['player']['id'] == playerId:
                    isYellow = 1
            elif (incident['incidentClass'] == 'yellowRed' or incident[
                'incidentClass'] == 'red') and 'player' in incident:
                if incident['player']['id'] == playerId:
                    isRed = 1
        if incident['incidentType'] == 'goal':
            if incident['incidentClass'] == 'penalty':
                if incident['player']['id'] == playerId:
                    penaltyScored += 1
    return isSub, isYellow, isRed, penaltyScored


def convertStats(stats):
    forbiddenColumns = ['rating', 'ratingVersions', 'statisticsType', 'goalAssist']
    columns = ['totalPass', 'totalLongBalls', 'totalOwnHalfPasses', 'totalOppositionHalfPasses', 'totalCross',
               'totalShots', 'totalTackle', 'totalContest']
    columnSuccess = ['accuratePass', 'accurateLongBalls', 'accurateOwnHalfPasses', 'accurateOppositionHalfPasses',
                     'accurateCross', 'onTargetScoringAttempt', 'wonTackle', 'wonContest']
    columnFail = ['inaccuratePass', 'inaccurateLongBalls', 'inaccurateOwnHalfPasses', 'inaccurateOppositionHalfPasses',
                  'inaccurateCross', 'offTargetScoringAttempt', 'lostTackle', 'lostContest']
    successRates = ['passSuccess', 'longBallSuccess', 'ownHalfPassSuccess', 'oppositionHalfPassSuccess', 'crossSuccess',
                    'onTargetSuccess', 'tackleSuccess', 'contestSuccess']
    keys = list(stats.keys())
    for key in keys:
        if key in forbiddenColumns:
            stats.pop(key)
```

```
    for x in range(len(columns)):
        if columns[x] in stats:
            if columnSuccess[x] in stats:
                stats[columnFail[x]] = stats[columns[x]] - stats[columnSuccess[x]]
                stats[successRates[x]] = zeroDivide(stats[columnSuccess[x]], stats[columns[x]])
            else:
                stats[columnFail[x]] = stats[columns[x]]
                stats[successRates[x]] = 0
            stats.pop(columns[x])
    if 'errorLeadToAGoal' in stats:
        if 'errorLeadToAShot' in stats:
            stats['errorLeadToAShot'] += stats['errorLeadToAGoal']
        else:
            stats['errorLeadToAShot'] = stats['errorLeadToAGoal']
        stats.pop('errorLeadToAGoal')
    return stats


def getSeasonStatList(leagueId, seasonId):
    with requests.Session() as session:
        gamesPlayed = []
        matchIds = []
        stats = {}
        standings = \
            jsonRequest(
                f"http://www.sofascore.com/api/v1/unique-tournament/{leagueId}/season/{seasonId}/standings/total",
                defaultSession=session)[
                'standings'][0]['rows']
        for x in standings:
            gamesPlayed.append(x['matches'])
        rounds = max(gamesPlayed)
        for round in range(rounds):
            roundEvents = jsonRequest(
                f"http://www.sofascore.com/api/v1/unique-tournament/{leagueId}/season/{seasonId}/events/round/{round + 1}",
                cache_ttl=24 * 14, defaultSession=session)[
                'events']
            for event in roundEvents:
                if safe_request(f"http://www.sofascore.com/api/v1/event/{event['id']}/lineups",
                                cache_ttl=0, defaultSession=session).status_code == 200 and safe_request(
                    f"http://www.sofascore.com/api/v1/event/{event['id']}/odds/1/all", cache_ttl=0,
                    defaultSession=session).status_code == 200 and \
                        event['status']['code'] == 100:
                    matchIds.append(event['id'])
        for matchId in matchIds:
            lineups = jsonRequest(f"http://www.sofascore.com/api/v1/event/{matchId}/lineups", cache_ttl=0,
                                  defaultSession=session)
            incidents = jsonRequest(f"http://www.sofascore.com/api/v1/event/{matchId}/incidents", cache_ttl=0,
                                    defaultSession=session)['incidents']
            odds = jsonRequest(f"http://www.sofascore.com/api/v1/event/{matchId}/odds/1/all", cache_ttl=0,
                               defaultSession=session)['markets']
            homeOverperform = isOverperform(odds)
            awayOverperform = isOverperform(odds, home=False)
            for x in lineups['home']['players']:
                x = lineups['home']['players'].index(x)
                player = lineups['home']['players'][x]
                playerId = player['player']['id']
                player['statistics']['isSub'], player['statistics']['yellowCard'], player['statistics']['redCard'], \
                    player['statistics']['penaltyScored'] = isSubYellowRedPenalty(playerId, incidents)
                player['statistics']['overperform'] = homeOverperform
                player['statistics'] = convertStats(player['statistics'])
                playerMatchId = f"{playerId}-{matchId}"
                position = player['position']
                if position != 'G' and 'minutesPlayed' in player['statistics']:
                    player['statistics']['position'] = position
                    stats[playerMatchId] = player['statistics']
            for x in lineups['away']['players']:
                x = lineups['away']['players'].index(x)
                player = lineups['away']['players'][x]
                playerId = player['player']['id']
                player['statistics']['isSub'], player['statistics']['yellowCard'], player['statistics']['redCard'], \
                    player['statistics']['penaltyScored'] = isSubYellowRedPenalty(playerId, incidents)
                player['statistics']['overperform'] = awayOverperform
                player['statistics'] = convertStats(player['statistics'])
                playerMatchId = f"{playerId}-{matchId}"
                position = player['position']
                if position != 'G' and 'minutesPlayed' in player['statistics']:
                    player['statistics']['position'] = position
                    stats[playerMatchId] = player['statistics']
        return stats


leagueIds = [17, 8, 23, 35, 34]
seasonIdTuples = [(41886, 52186, 61627), (42409, 52376, 61643), (42415, 52760, 63515), (42268, 52608, 63516),
                  (42273, 52571, 61736)]

filepath = "data/playerStats.json"
csvFilepath = "data/playerStatsCsv.csv"
```

```
with open(filepath, "w", encoding="utf-8") as f:
    data = {}
    for x in range(len(leagueIds)):
        leagueId = leagueIds[x]
        for seasonId in seasonIdTuples[x]:
            data.update(getSeasonStatList(leagueId, seasonId))
    json.dump(data, f, indent=4)

with open(filepath, 'r') as f:
    data = json.load(f)
    df = pd.DataFrame(data).T
    df = df.reset_index()
    df.rename(columns={'index': 'playerMatchId'}, inplace=True)
    df = df.fillna(0)
    df.to_csv(csvFilepath, index=False)
```