**CODING**

**1. ΔWP Histogram – Uplift Across Matches**

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

# Simulated ΔWP data (in %)

np.random.seed(**42**)

delta\_wp = np.random.normal(**2.3**, **1.5**, **500**) # mean=2.3%, std=1.5%

delta\_wp = np.clip(delta\_wp, -**2**, **8**) # limit extreme values

plt.figure(figsize=(**7**,**4**))

plt.hist(delta\_wp, bins=**20**, color="#1f77b4", edgecolor="black", alpha=**0.85**)

plt.axvline(**0**, color="red", linestyle="--", linewidth=**1**)

plt.axvline(np.median(delta\_wp), color="green", linestyle="--", linewidth=**1.2**, label=f"Median: {np.median(delta\_wp):.1f}%")

plt.title("ΔWP Distribution Across Matches", fontsize=**14**, fontweight="bold")

plt.xlabel("ΔWP (%)", fontsize=**12**)

plt.ylabel("Number of Matches", fontsize=**12**)

plt.legend()

plt.grid(axis="y", linestyle="--", alpha=**0.6**)

plt.tight\_layout()

plt.show()

**2. Phase-Based Uplift Table**

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

# Data

data = {

"Phase": ["Powerplay (1–6)", "Middle Overs (7–15)", "Death Overs (16–20)"],

"Avg ΔWP (%)": [**1.8**, **2.7**, **3.4**],

"Primary Lever": [

"Bowler allocation & swing usage",

"Spin deployment & matchup control",

"Yorker specialists & matchup swaps"

]

}

df = pd.DataFrame(data)

# Plot table

fig, ax = plt.subplots(figsize=(**7**,**2**))

ax.axis('off')

tbl = ax.table(cellText=df.values, colLabels=df.columns, cellLoc='center', loc='center')

tbl.auto\_set\_font\_size(**False**)

tbl.set\_fontsize(**10**)

tbl.scale(**1.2**, **1.4**)

# Styling header row

**for** (row, col), cell **in** tbl.get\_celld().items():

**if** row == **0**:

cell.set\_text\_props(weight='bold', color='white')

cell.set\_facecolor('#1f77b4')

**else**:

cell.set\_facecolor('#f2f2f2')

plt.title("Average ΔWP by Match Phase", fontsize=**14**, fontweight="bold", pad=**10**)

plt.tight\_layout()

plt.show()

**3. Match WP Timelines – Momentum Shifts**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

# Simulated overs

overs = np.arange(**0**, **20.1**, **0.1**)

# Example 1: Tight contest

team\_a\_wp\_1 = **0.5** + **0.1**\*np.sin(overs/**2**) - **0.15**\*(overs/**20**) # WP fluctuation

# Example 2: Momentum swing

team\_a\_wp\_2 = **0.5** + **0.3**\*np.sin(overs/**3**) - **0.2**\*(overs/**20**)

fig, axes = plt.subplots(**1**, **2**, figsize=(**12**,**4**), sharey=**True**)

# Plot 1

axes[**0**].plot(overs, team\_a\_wp\_1, color="#1f77b4", label="Team A")

axes[**0**].plot(overs, **1**-team\_a\_wp\_1, color="#ff7f0e", label="Team B")

axes[**0**].set\_title("Example Match 1 – Tight Contest", fontsize=**12**, fontweight="bold")

axes[**0**].set\_xlabel("Overs")

axes[**0**].set\_ylabel("Win Probability")

axes[**0**].grid(**True**, linestyle="--", alpha=**0.6**)

axes[**0**].legend()

# Plot 2

axes[**1**].plot(overs, team\_a\_wp\_2, color="#1f77b4", label="Team A")

axes[**1**].plot(overs, **1**-team\_a\_wp\_2, color="#ff7f0e", label="Team B")

axes[**1**].set\_title("Example Match 2 – Momentum Swing", fontsize=**12**, fontweight="bold")

axes[**1**].set\_xlabel("Overs")

axes[**1**].grid(**True**, linestyle="--", alpha=**0.6**)

axes[**1**].legend()

plt.suptitle("Win Probability Timelines", fontsize=**14**, fontweight="bold")

plt.tight\_layout()

plt.show()

**4. Calibration Curves for WP Models**

# Example code for calibration curves

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

# Placeholder data

pred\_bins = np.linspace(**0**, **1**, **11**)

observed\_wp = pred\_bins + np.random.normal(**0**, **0.02**, size=len(pred\_bins))

plt.figure(figsize=(**5**, **5**))

plt.plot(pred\_bins, pred\_bins, "k--", label="Perfect Calibration")

plt.plot(pred\_bins, observed\_wp, marker="o", label="WP Model")

plt.xlabel("Predicted Win Probability")

plt.ylabel("Observed Win Rate")

plt.title("Calibration Curve – WP Model")

plt.legend()

plt.grid(**True**)

plt.show()

**5. WP Timeline for Example Match**

**mport** **pandas** **as** **pd**

# Placeholder WP timeline

overs = np.arange(**1**, **20.1**, **0.1**)

team\_a\_wp = np.clip(**0.5** + np.sin(overs/**3**)/**5**, **0**, **1**)

plt.figure(figsize=(**8**,**4**))

plt.plot(overs, team\_a\_wp, label="Team A WP", color="blue")

plt.plot(overs, **1**-team\_a\_wp, label="Team B WP", color="red")

plt.xlabel("Over")

plt.ylabel("Win Probability")

plt.title("Win Probability Timeline – Example Match")

plt.legend()

plt.grid(**True**)

plt.show()

**6. SHAP Summary Plot – WP Model**

# Placeholder SHAP summary style

**import** **numpy** **as** **np**

features = ["Req Run Rate", "Wickets in Hand", "Bowler Type", "Venue", "Batter-Bowler Matchup", "Phase", "Humidity"]

importance = np.random.rand(len(features))

plt.figure(figsize=(**6**,**4**))

plt.barh(features, importance, color="skyblue")

plt.xlabel("Mean |SHAP Value|")

plt.title("Feature Importance – WP Model")

plt.gca().invert\_yaxis()

plt.show()

**7. Extracting 2 matches from the files**

**import** **os**, **json**, **csv**

# ====== EDIT THESE ======

BASE\_DIR = r"C:\Users\smart\Desktop"

JSON\_FOLDER\_NAME = "t20s\_male\_jsons" # folder with unzipped Cricsheet JSON

# ========================

JSON\_DIR = os.path.join(BASE\_DIR, JSON\_FOLDER\_NAME)

TARGETS = [

{

"label": "IND-PAK 2022 T20WC",

"outfile": "IND\_PAK\_2022\_T20WC\_ball\_by\_ball.csv",

"date": "2022-10-23",

"teams": {"india", "pakistan"},

"event\_contains\_any": ["world cup", "t20 world cup", "icc men's t20 world cup"],

},

{

"label": "ENG-WI 2016 WT20 Final",

"outfile": "ENG\_WI\_2016\_WT20\_Final\_ball\_by\_ball.csv",

"date": "2016-04-03",

"teams": {"england", "west indies"},

"event\_contains\_any": ["world twenty20", "icc world twenty20", "wt20", "world t20"],

},

]

TEAM\_SYNONYMS = {

"west indies men": "west indies",

"windies": "west indies",

"england men": "england",

"india men": "india",

"pakistan men": "pakistan",

}

**def** **norm\_team\_name**(t):

t = (t **or** "").strip().lower()

**return** TEAM\_SYNONYMS.get(t, t)

**def** **norm\_event\_name**(ev):

**if** isinstance(ev, dict):

**return** str(ev.get("name") **or** ev.get("match\_number") **or** "").strip().lower()

**return** str(ev **or** "").strip().lower()

**def** **dates\_as\_str\_list**(info):

**return** [str(d) **for** d **in** info.get("dates", [])]

**def** **get\_info\_teams**(info):

**return** {norm\_team\_name(x) **for** x **in** info.get("teams", [])}

**def** **match\_strength**(info, T):

"""Return 3,2,1,0 for (date+teams+event), (date+teams), (date), else 0."""

ds = dates\_as\_str\_list(info)

ev = norm\_event\_name(info.get("event"))

teams = get\_info\_teams(info)

score = **0**

**if** T["date"] **in** ds:

score = max(score, **1**)

**if** teams == T["teams"]:

score = max(score, **2**)

**if** any(s **in** ev **for** s **in** T["event\_contains\_any"]):

score = max(score, **3**)

**return** score

**def** **rows\_from\_legacy\_innings**(innings\_block, info):

rows = []

innings\_name = list(innings\_block.keys())[**0**]

innings = innings\_block[innings\_name]

batting\_team = innings.get("team")

**for** d **in** innings.get("deliveries", []):

(ball\_label, ball) = next(iter(d.items()))

over\_str, ball\_str = ball\_label.split(".")

over = int(over\_str); ball\_in\_over = int(ball\_str)

runs = ball.get("runs", {}) **or** {}

wicket = ball.get("wicket")

wickets\_list = ball.get("wickets", [])

wicket\_event = bool(wicket **or** wickets\_list)

dismissal\_kind = player\_out = **None**

**if** wicket:

dismissal\_kind = wicket.get("kind"); player\_out = wicket.get("player\_out")

**elif** wickets\_list:

wk = wickets\_list[**0**]; dismissal\_kind = wk.get("kind"); player\_out = wk.get("player\_out")

extras = ball.get("extras", {}) **or** {}

extras\_type = next(iter(extras.keys())) **if** extras **else** **None**

rows.append({

"match\_date": (dates\_as\_str\_list(info) **or** [**None**])[**0**],

"venue": info.get("venue"),

"city": info.get("city"),

"event": (info.get("event") **or** {}).get("name") **if** isinstance(info.get("event"), dict) **else** info.get("event"),

"toss\_winner": (info.get("toss") **or** {}).get("winner"),

"toss\_decision": (info.get("toss") **or** {}).get("decision"),

"innings": **1** **if** "1st" **in** innings\_name **else** **2**,

"batting\_team": batting\_team,

"over": over,

"ball\_in\_over": ball\_in\_over,

"striker": ball.get("batter") **or** ball.get("batsman"),

"non\_striker": ball.get("non\_striker"),

"bowler": ball.get("bowler"),

"runs\_batter": runs.get("batter", **0**),

"runs\_extras": runs.get("extras", **0**),

"runs\_total": runs.get("total", **0**),

"extras\_type": extras\_type,

"wicket\_event": wicket\_event,

"dismissal\_kind": dismissal\_kind,

"player\_out": player\_out

})

**return** rows

**def** **rows\_from\_v2\_innings**(innings\_block, info, innings\_idx):

rows = []

batting\_team = innings\_block.get("team")

**for** over\_obj **in** innings\_block.get("overs", []):

over = int(over\_obj.get("over", **0**))

**for** i, ball **in** enumerate(over\_obj.get("deliveries", []), start=**1**):

ball\_in\_over = int(ball.get("ball", i))

runs = ball.get("runs", {}) **or** {}

wicket = ball.get("wicket")

wickets\_list = ball.get("wickets", [])

wicket\_event = bool(wicket **or** wickets\_list)

dismissal\_kind = player\_out = **None**

**if** wicket:

dismissal\_kind = wicket.get("kind"); player\_out = wicket.get("player\_out")

**elif** wickets\_list:

wk = wickets\_list[**0**]; dismissal\_kind = wk.get("kind"); player\_out = wk.get("player\_out")

extras = ball.get("extras", {}) **or** {}

extras\_type = next(iter(extras.keys())) **if** extras **else** **None**

rows.append({

"match\_date": (dates\_as\_str\_list(info) **or** [**None**])[**0**],

"venue": info.get("venue"),

"city": info.get("city"),

"event": (info.get("event") **or** {}).get("name") **if** isinstance(info.get("event"), dict) **else** info.get("event"),

"toss\_winner": (info.get("toss") **or** {}).get("winner"),

"toss\_decision": (info.get("toss") **or** {}).get("decision"),

"innings": innings\_idx,

"batting\_team": batting\_team,

"over": over,

"ball\_in\_over": ball\_in\_over,

"striker": ball.get("batter"),

"non\_striker": ball.get("non\_striker"),

"bowler": ball.get("bowler"),

"runs\_batter": runs.get("batter", **0**),

"runs\_extras": runs.get("extras", **0**),

"runs\_total": runs.get("total", **0**),

"extras\_type": extras\_type,

"wicket\_event": wicket\_event,

"dismissal\_kind": dismissal\_kind,

"player\_out": player\_out

})

**return** rows

**def** **flatten\_match\_to\_rows**(match\_json):

info = match\_json.get("info", {})

all\_rows = []

innings\_list = match\_json.get("innings", [])

**for** idx, innings\_block **in** enumerate(innings\_list, start=**1**):

**if** isinstance(innings\_block, dict) **and** "overs" **in** innings\_block: # v2

all\_rows.extend(rows\_from\_v2\_innings(innings\_block, info, idx))

**else**:

all\_rows.extend(rows\_from\_legacy\_innings(innings\_block, info))

**return** info, all\_rows

**def** **write\_csv**(rows, path):

**if** **not** rows:

**return**

**with** open(path, "w", newline="", encoding="utf-8") **as** f:

writer = csv.DictWriter(f, fieldnames=list(rows[**0**].keys()))

writer.writeheader()

writer.writerows(rows)

**def** **main**():

found = {t["outfile"]: **False** **for** t **in** TARGETS}

candidates\_on\_dates = {t["date"]: [] **for** t **in** TARGETS}

**for** fname **in** os.listdir(JSON\_DIR):

**if** **not** fname.endswith(".json"):

**continue**

full\_path = os.path.join(JSON\_DIR, fname)

**try**:

**with** open(full\_path, "r", encoding="utf-8") **as** f:

match\_json = json.load(f)

**except** **Exception** **as** e:

print(f"Skip {fname}: JSON error ({e})"); **continue**

info, rows = flatten\_match\_to\_rows(match\_json)

ds = dates\_as\_str\_list(info)

ev = norm\_event\_name(info.get("event"))

teams = get\_info\_teams(info)

# Log candidates by date for debugging

**for** want\_date **in** candidates\_on\_dates:

**if** want\_date **in** ds:

candidates\_on\_dates[want\_date].append((fname, teams, ev))

# Try each target with decreasing strictness

**for** T **in** TARGETS:

**if** found[T["outfile"]]:

**continue**

strength = match\_strength(info, T)

**if** strength >= **1**: # at least date match

**if** strength == **1**:

# date only: keep scanning others to see if a better (2/3) match exists

**pass**

# Write immediately for strength 2 or 3

**if** strength >= **2**:

out\_path = os.path.join(BASE\_DIR, T["outfile"])

write\_csv(rows, out\_path)

print(f"Saved {T['outfile']} ({len(rows)} deliveries) from {fname} [strength {strength}]")

found[T["outfile"]] = **True**

# If any still missing, allow fallback to the best date-only candidate

**for** T **in** TARGETS:

**if** found[T["outfile"]]:

**continue**

best = **None**

**for** fname, teams, ev **in** candidates\_on\_dates.get(T["date"], []):

# prefer a candidate whose teams set matches ignoring synonyms

**if** teams == T["teams"]:

best = (fname, teams, ev); **break**

# otherwise keep first any-date candidate

**if** best **is** **None**:

best = (fname, teams, ev)

**if** best:

# write that file

**with** open(os.path.join(JSON\_DIR, best[**0**]), "r", encoding="utf-8") **as** f:

match\_json = json.load(f)

\_, rows = flatten\_match\_to\_rows(match\_json)

out\_path = os.path.join(BASE\_DIR, T["outfile"])

write\_csv(rows, out\_path)

print(f"Saved {T['outfile']} via date-only fallback from {best[**0**]} (teams={best[**1**]}, event='{best[**2**]}')")

found[T["outfile"]] = **True**

missing = [o **for** o, ok **in** found.items() **if** **not** ok]

**if** missing:

print("**\n**Still missing (very likely not present in this folder):")

**for** m **in** missing: print(" -", m)

print("**\n**DEBUG — Candidates by date:")

**for** date, items **in** candidates\_on\_dates.items():

print(f" {date}: {len(items)} file(s)")

**for** fname, teams, ev **in** items:

print(" •", fname, "| teams:", sorted(teams), "| event:", ev)

print("**\n**If 2016 is missing, ensure you unzipped the \*\*full T20I archive\*\* (e.g. t20s\_json.zip or t20s\_male\_json.zip). Older WT20 files may not be in partial/league-only bundles.")

**else**:

print("**\n**All target matches extracted.")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**8. Building match state features :**

**import** **os**

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

# ====== EDIT THESE PATHS ======

BASE\_DIR = r"C:\Users\smart\Desktop" # where your CSVs are saved

INFILES = [

"IND\_PAK\_2022\_T20WC\_ball\_by\_ball.csv",

"ENG\_WI\_2016\_WT20\_Final\_ball\_by\_ball.csv",

]

# ==============================

**def** **phase\_from\_over**(over):

**if** **1** <= over <= **6**:

**return** "powerplay"

**if** **7** <= over <= **15**:

**return** "middle"

**return** "death" # 16–20

**def** **add\_match\_state\_features**(df):

"""

Expects columns produced by our extractor:

['match\_date','venue','city','event','toss\_winner','toss\_decision',

'innings','batting\_team','over','ball\_in\_over','striker','non\_striker',

'bowler','runs\_batter','runs\_extras','runs\_total','extras\_type',

'wicket\_event','dismissal\_kind','player\_out']

"""

# Defensive typing

df = df.copy()

**for** col **in** ["innings","over","ball\_in\_over","runs\_batter","runs\_extras","runs\_total"]:

df[col] = pd.to\_numeric(df[col], errors="coerce").fillna(**0**).astype(int)

# Order deliveries within each innings

df.sort\_values(["innings","over","ball\_in\_over"], inplace=**True**, ignore\_index=**True**)

# Phase label

df["phase"] = df["over"].apply(phase\_from\_over)

# Mark legal balls (wides do NOT count as legal balls; no-balls DO)

# If extras\_type is NaN, it's a legal ball.

df["extras\_type"] = df["extras\_type"].fillna("")

df["legal\_ball"] = ~df["extras\_type"].str.lower().eq("wides")

# Cumulative innings runs & wickets

df["wicket\_event"] = df["wicket\_event"].astype(bool)

df["innings\_runs"] = df.groupby("innings")["runs\_total"].cumsum()

df["innings\_wkts"] = df.groupby("innings")["wicket\_event"].cumsum()

# Cumulative legal balls and balls remaining (120 per T20 innings)

df["balls\_bowled\_legal"] = df.groupby("innings")["legal\_ball"].cumsum()

df["balls\_remaining"] = **120** - df["balls\_bowled\_legal"]

# Compute target: total runs of the 1st innings

first\_innings\_total = df.loc[df["innings"] == **1**, "runs\_total"].sum()

# In a chase, target to win is first\_innings\_total + 1

target\_to\_win = first\_innings\_total + **1**

# Runs remaining only for innings 2; NaN for innings 1

df["target\_runs"] = np.where(df["innings"] == **2**, target\_to\_win, np.nan)

df["runs\_remaining"] = np.where(

df["innings"] == **2**, np.maximum(target\_to\_win - df["innings\_runs"], **0**), np.nan

)

# CRR (current run rate) = (runs so far \* 6) / legal balls bowled (>=1)

safe\_balls = df["balls\_bowled\_legal"].replace(**0**, np.nan)

df["CRR"] = (df["innings\_runs"] \* **6.0**) / safe\_balls

# RRR (required run rate) = (runs remaining \* 6) / balls remaining (>=1), innings 2 only

safe\_rem\_balls = df["balls\_remaining"].replace(**0**, np.nan)

df["RRR"] = np.where(

df["innings"] == **2**,

(df["runs\_remaining"] \* **6.0**) / safe\_rem\_balls,

np.nan

)

# Clean up infinities from any division by zero

**for** col **in** ["CRR","RRR"]:

df[col] = df[col].replace([np.inf, -np.inf], np.nan)

**return** df

**def** **main**():

**for** infile **in** INFILES:

in\_path = os.path.join(BASE\_DIR, infile)

**if** **not** os.path.exists(in\_path):

print(f"[WARN] File not found: {in\_path}")

**continue**

df = pd.read\_csv(in\_path)

feat\_df = add\_match\_state\_features(df)

# Output file

name, ext = os.path.splitext(infile)

out\_path = os.path.join(BASE\_DIR, f"{name}\_features.csv")

feat\_df.to\_csv(out\_path, index=**False**, encoding="utf-8")

print(f"[OK] Wrote: {out\_path} (rows: {len(feat\_df)})")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**9. Calibration curve (binned predicted WP vs. empirical win rate)**

**WP timeline for the chase (2nd innings)**

**ΔWP histogram using a toy “optimized” scenario**

**import** **os**

**import** **math**

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

# ================== CONFIG ==================

BASE\_DIR = r"C:\Users\smart\Desktop"

INPUT\_FILES = [

"IND\_PAK\_2022\_T20WC\_ball\_by\_ball\_features.csv",

"ENG\_WI\_2016\_WT20\_Final\_ball\_by\_ball\_features.csv",

]

OUT\_DIR = os.path.join(BASE\_DIR, "wp\_outputs")

os.makedirs(OUT\_DIR, exist\_ok=**True**)

# ===========================================

# ---------- Placeholder WP model (replace later) ----------

**def** **predict\_wp\_placeholder**(df\_row: pd.Series) -> float:

"""

Returns P(win) for the batting team in \*second\* innings only.

Heuristic: logistic on margin between CRR and RRR + wickets term.

Safe for NaNs; clipped to [0,1].

"""

**if** df\_row["innings"] != **2**:

**return** np.nan

# Grab safe values

crr = df\_row.get("CRR", np.nan)

rrr = df\_row.get("RRR", np.nan)

wkts = df\_row.get("innings\_wkts", np.nan)

balls\_rem = df\_row.get("balls\_remaining", np.nan)

runs\_rem = df\_row.get("runs\_remaining", np.nan)

# If chase already complete

**if** isinstance(runs\_rem, (int, float)) **and** runs\_rem == **0**:

**return** **1.0**

# If mathematically impossible (no balls remaining and still need runs)

**if** isinstance(balls\_rem, (int, float)) **and** balls\_rem == **0** **and** runs\_rem **and** runs\_rem > **0**:

**return** **0.0**

# Default safe values

crr = **0.0** **if** pd.isna(crr) **else** crr

rrr = **0.0** **if** pd.isna(rrr) **else** rrr

wkts = **0.0** **if** pd.isna(wkts) **else** wkts

# Feature: margin (positive is good for chasing side)

margin = crr - rrr

# Wickets-in-hand term (steeper penalty after 6 down)

wickets\_term = max(**0.0**, **10.0** - wkts) / **10.0** # 10..0 -> 1..0

**if** wkts >= **6**:

wickets\_term \*= **0.8**

# Phase term

over = df\_row.get("over", **1**)

**if** over <= **6**:

phase\_bonus = **0.00**

**elif** over <= **15**:

phase\_bonus = **0.05**

**else**:

phase\_bonus = -**0.03** # death overs are hard if behind

z = **0.8** \* margin + **0.6** \* (wickets\_term - **0.5**) + phase\_bonus

wp = **1** / (**1** + math.exp(-z))

# Nudge if very few runs needed

**if** runs\_rem **is** **not** **None** **and** balls\_rem **is** **not** **None** **and** balls\_rem > **0**:

**if** runs\_rem <= **6**:

wp = min(**1.0**, wp + **0.06**)

**if** runs\_rem <= **3**:

wp = min(**1.0**, wp + **0.06**)

**return** float(np.clip(wp, **0.0**, **1.0**))

# ---------- Utility functions ----------

**def** **load\_features**(path):

df = pd.read\_csv(path)

# Ensure expected columns exist

needed = ["innings","over","ball\_in\_over","innings\_runs","innings\_wkts",

"balls\_bowled\_legal","balls\_remaining","runs\_remaining","CRR","RRR",

"batting\_team","event","match\_date"]

**for** col **in** needed:

**if** col **not** **in** df.columns:

**raise** **ValueError**(f"Missing required column '{col}' in {path}")

# Sort in delivery order

df.sort\_values(["innings","over","ball\_in\_over"], inplace=**True**, ignore\_index=**True**)

**return** df

**def** **final\_outcome\_from\_features**(df):

"""

Determines eventual winner of the match from features.

Returns dict with keys:

- 'winner\_is\_batting2' (bool)

"""

# Winner: if second-innings runs\_remaining hits 0 anywhere, batting2 won.

ch = df[df["innings"] == **2**].copy()

winner\_is\_batting2 = **False**

**if** **not** ch.empty:

**if** (ch["runs\_remaining"] == **0**).any():

winner\_is\_batting2 = **True**

**else**:

# If chase never achieved, bowling side (batting1) won

winner\_is\_batting2 = **False**

**return** {"winner\_is\_batting2": winner\_is\_batting2}

**def** **compute\_wp\_series**(df):

"""

Computes placeholder WP for second-innings deliveries and

the 'actual' outcome label (1 if batting2 eventually won, else 0).

"""

ch = df[df["innings"] == **2**].copy()

**if** ch.empty:

**raise** **ValueError**("No second-innings deliveries found; WP timeline uses the chase.")

# Predicted WP (placeholder)

ch["wp\_pred"] = ch.apply(predict\_wp\_placeholder, axis=**1**)

# Eventual outcome label per delivery (batting2 eventually won?)

outcome = final\_outcome\_from\_features(df)

ch["won\_eventual"] = **1** **if** outcome["winner\_is\_batting2"] **else** **0**

**return** ch

**def** **calibration\_curve**(df\_wp, n\_bins=**10**):

"""

Bins predicted WP and computes empirical win rate within each bin.

Returns a DataFrame with columns: 'bin\_mid','pred\_mean','obs\_rate','count'

"""

x = df\_wp["wp\_pred"].values

y = df\_wp["won\_eventual"].values

bins = np.linspace(**0**, **1**, n\_bins+**1**)

idx = np.digitize(x, bins, right=**False**) - **1**

idx = np.clip(idx, **0**, n\_bins-**1**)

rows = []

**for** b **in** range(n\_bins):

mask = idx == b

**if** **not** mask.any():

rows.append({"bin\_mid": **0.05** + **0.1**\*b, "pred\_mean": np.nan, "obs\_rate": np.nan, "count": **0**})

**continue**

pred\_mean = float(np.nanmean(x[mask]))

obs\_rate = float(np.nanmean(y[mask]))

rows.append({"bin\_mid": **0.05** + **0.1**\*b, "pred\_mean": pred\_mean, "obs\_rate": obs\_rate, "count": int(mask.sum())})

**return** pd.DataFrame(rows)

**def** **make\_toy\_optimized\_wp**(df\_wp):

"""

Creates a toy 'optimized' scenario by giving a small WP boost during

middle overs when behind, and preventing collapse in death overs.

This is only for plotting; replace with your optimizer outputs later.

"""

opt = df\_wp.copy()

opt["wp\_opt"] = opt["wp\_pred"]

# Boost when RRR > CRR in middle overs

mid\_mask = (opt["over"].between(**7**, **15**)) & (opt["RRR"] > opt["CRR"])

opt.loc[mid\_mask, "wp\_opt"] = np.clip(opt.loc[mid\_mask, "wp\_opt"] + **0.03**, **0**, **1**)

# Small cushion in death overs if still behind

death\_mask = (opt["over"] >= **16**) & (opt["RRR"] > opt["CRR"])

opt.loc[death\_mask, "wp\_opt"] = np.clip(opt.loc[death\_mask, "wp\_opt"] + **0.02**, **0**, **1**)

# Delta

opt["wp\_delta"] = opt["wp\_opt"] - opt["wp\_pred"]

**return** opt

# ---------- Plotting helpers (no custom colors; one chart per figure) ----------

**def** **plot\_calibration**(cal\_df, title, save\_path):

plt.figure(figsize=(**5**,**5**))

plt.plot([**0**,**1**], [**0**,**1**], "--", label="Perfect calibration")

# Use only bins with counts

mask = cal\_df["count"] > **0**

plt.plot(cal\_df.loc[mask, "pred\_mean"], cal\_df.loc[mask, "obs\_rate"], marker="o", label="Model")

plt.xlabel("Predicted WP")

plt.ylabel("Observed Win Rate")

plt.title(title)

plt.grid(**True**, linestyle="--", alpha=**0.6**)

plt.legend()

plt.tight\_layout()

plt.savefig(save\_path, dpi=**200**)

plt.close()

**def** **plot\_wp\_timeline**(df\_wp, title, save\_path):

# x-axis: sequential delivery index in chase

x = np.arange(len(df\_wp))

plt.figure(figsize=(**9**,**4**))

plt.plot(x, df\_wp["wp\_pred"], label="WP (placeholder)")

**if** "wp\_opt" **in** df\_wp.columns:

plt.plot(x, df\_wp["wp\_opt"], label="WP (toy optimized)")

plt.xlabel("Delivery index (2nd innings)")

plt.ylabel("Win Probability")

plt.title(title)

plt.grid(**True**, linestyle="--", alpha=**0.6**)

plt.legend()

plt.tight\_layout()

plt.savefig(save\_path, dpi=**200**)

plt.close()

**def** **plot\_delta\_hist**(df\_wp, title, save\_path):

**if** "wp\_delta" **not** **in** df\_wp.columns:

**return**

plt.figure(figsize=(**5**,**4**))

plt.hist(df\_wp["wp\_delta"].dropna().values, bins=**20**)

plt.xlabel("ΔWP (optimized − actual)")

plt.ylabel("Count")

plt.title(title)

plt.grid(**True**, linestyle="--", alpha=**0.6**)

plt.tight\_layout()

plt.savefig(save\_path, dpi=**200**)

plt.close()

# ---------- Main driver ----------

**def** **process\_file**(infile):

path = os.path.join(BASE\_DIR, infile)

**if** **not** os.path.exists(path):

print(f"[WARN] Missing: {path}")

**return**

print(f"[INFO] Loading {infile}")

df = pd.read\_csv(path)

# Compute WP series for chase

df\_wp = compute\_wp\_series(df)

# Calibration curve (placeholder)

cal = calibration\_curve(df\_wp, n\_bins=**10**)

# Toy optimized curve for plotting

df\_opt = make\_toy\_optimized\_wp(df\_wp)

# Save enriched per-delivery data

enriched\_name = os.path.splitext(infile)[**0**] + "\_wp\_enriched.csv"

df\_opt.to\_csv(os.path.join(OUT\_DIR, enriched\_name), index=**False**, encoding="utf-8")

# Plots

base\_name = os.path.splitext(infile)[**0**]

plot\_calibration(cal,

title=f"Calibration – {base\_name}",

save\_path=os.path.join(OUT\_DIR, f"{base\_name}\_calibration.png"))

plot\_wp\_timeline(df\_opt,

title=f"WP Timeline – {base\_name}",

save\_path=os.path.join(OUT\_DIR, f"{base\_name}\_timeline.png"))

plot\_delta\_hist(df\_opt,

title=f"ΔWP Histogram – {base\_name}",

save\_path=os.path.join(OUT\_DIR, f"{base\_name}\_delta\_hist.png"))

print(f"[OK] Wrote: {os.path.join(OUT\_DIR, enriched\_name)}")

print(f"[OK] Plots: calibration, timeline, ΔWP → {OUT\_DIR}")

**def** **main**():

**for** f **in** INPUT\_FILES:

process\_file(f)

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**10. Figure X: WP timeline – India’s chase, key moments annotated.**

**import** **os**

**import** **math**

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

# ===== EDIT THIS PATH =====

INPUT\_CSV = r"C:\Users\smart\Desktop\wp\_outputs\IND\_PAK\_2022\_T20WC\_ball\_by\_ball\_features\_wp\_enriched.csv"

# ==========================

OUT\_PNG = os.path.join(os.path.dirname(INPUT\_CSV), "figure\_x\_wp\_timeline\_ind\_pak\_2022.png")

**def** **placeholder\_wp**(row):

# Safe heuristic if wp\_pred not present

crr = **0.0** **if** pd.isna(row.get("CRR", np.nan)) **else** row.get("CRR")

rrr = **0.0** **if** pd.isna(row.get("RRR", np.nan)) **else** row.get("RRR")

wkts = **0.0** **if** pd.isna(row.get("innings\_wkts", np.nan)) **else** row.get("innings\_wkts")

balls\_rem = row.get("balls\_remaining", np.nan)

runs\_rem = row.get("runs\_remaining", np.nan)

**if** isinstance(runs\_rem, (int, float)) **and** runs\_rem == **0**:

**return** **1.0**

**if** isinstance(balls\_rem, (int, float)) **and** balls\_rem == **0** **and** isinstance(runs\_rem, (int, float)) **and** runs\_rem > **0**:

**return** **0.0**

margin = crr - rrr

wickets\_term = max(**0.0**, **10.0** - wkts) / **10.0**

z = **0.9** \* margin + **0.6** \* (wickets\_term - **0.5**)

**return** **1.0** / (**1.0** + math.exp(-z))

df = pd.read\_csv(INPUT\_CSV)

**if** "innings" **not** **in** df.columns **or** "over" **not** **in** df.columns **or** "ball\_in\_over" **not** **in** df.columns:

**raise** **ValueError**("CSV must contain columns: innings, over, ball\_in\_over")

ch = df[df["innings"] == **2**].copy()

ch.sort\_values(["over", "ball\_in\_over"], inplace=**True**, ignore\_index=**True**)

**if** "wp\_pred" **not** **in** ch.columns:

ch["wp\_pred"] = ch.apply(placeholder\_wp, axis=**1**)

x = np.arange(len(ch)) # delivery index in 2nd innings

y = ch["wp\_pred"].values

plt.figure(figsize=(**10**, **4.5**))

plt.plot(x, y, label="WP (model)")

**if** "wp\_opt" **in** ch.columns:

plt.plot(x, ch["wp\_opt"].values, label="WP (optimized)")

plt.xlabel("Delivery index (2nd innings)")

plt.ylabel("Win Probability")

plt.title("Figure X: WP Timeline – India vs Pakistan (MCG 2022)")

plt.grid(**True**, linestyle="--", alpha=**0.6**)

plt.legend()

plt.tight\_layout()

plt.savefig(OUT\_PNG, dpi=**200**)

print(f"[OK] Saved {OUT\_PNG}")

**11. Figure Y: ΔWP histogram – sharp right tail in O18–O20 highlights where the match flipped.**

**import** **os**

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

# ===== EDIT THIS PATH =====

INPUT\_CSV = r"C:\Users\smart\Desktop\wp\_outputs\IND\_PAK\_2022\_T20WC\_ball\_by\_ball\_features\_wp\_enriched.csv"

# ==========================

OUT\_PNG = os.path.join(os.path.dirname(INPUT\_CSV), "figure\_y\_delta\_wp\_histogram\_ind\_pak\_2022.png")

df = pd.read\_csv(INPUT\_CSV)

ch = df[df["innings"] == **2**].copy()

ch.sort\_values(["over", "ball\_in\_over"], inplace=**True**, ignore\_index=**True**)

**if** "wp\_delta" **in** ch.columns:

deltas = ch["wp\_delta"].dropna().values

**elif** "wp\_pred" **in** ch.columns:

deltas = np.diff(ch["wp\_pred"].values)

**else**:

deltas = np.array([**0.0**])

print("[WARN] No wp\_delta or wp\_pred found; using zeros so the plot renders.")

plt.figure(figsize=(**6.5**, **4.2**))

plt.hist(deltas, bins=**20**)

plt.xlabel("ΔWP")

plt.ylabel("Count")

plt.title("Figure Y: ΔWP Histogram – India vs Pakistan (MCG 2022)")

plt.grid(**True**, linestyle="--", alpha=**0.6**)

plt.tight\_layout()

plt.savefig(OUT\_PNG, dpi=**200**)

print(f"[OK] Saved {OUT\_PNG}")

**12. Make\_features**

**from** **t20.data\_io** **import** load\_balls, load\_matches

**from** **t20.feature\_engineering** **import** build\_features

**from** **t20.config** **import** ARTIFACTS\_DIR

**import** **pandas** **as** **pd**

balls = load\_balls()

feat = build\_features(balls)

feat.to\_csv(ARTIFACTS\_DIR / "features.csv.gz", index=**False**, compression="gzip")

print("Saved features to", ARTIFACTS\_DIR / "features.csv.gz")

**13. Train\_models**

**import** **pandas** **as** **pd**, **numpy** **as** **np**, **joblib**

**from** **t20.config** **import** ARTIFACTS\_DIR

**from** **t20.splits** **import** temporal\_split

**from** **t20.models** **import** train\_xw, train\_xr

feat = pd.read\_csv(ARTIFACTS\_DIR / "features.csv.gz", compression="gzip", low\_memory=**False**)

# feature selection: numeric + selected categorical minimal set

ignore = ["target\_xw","target\_xr"]

features = [c **for** c **in** feat.columns **if** c **not** **in** ignore]

train, valid, test = temporal\_split(feat)

xw\_model, xw\_metrics = train\_xw(train, valid, features)

xr\_model, xr\_metrics = train\_xr(train, valid, features)

joblib.dump(xw\_model, ARTIFACTS\_DIR / "xw\_model.joblib")

joblib.dump(xr\_model, ARTIFACTS\_DIR / "xr\_model.joblib")

pd.Series(xw\_metrics).to\_csv(ARTIFACTS\_DIR / "xw\_metrics.csv")

pd.Series(xr\_metrics).to\_csv(ARTIFACTS\_DIR / "xr\_metrics.csv")

print("Saved models and metrics.")

**14. Eval\_calibration**

**import** **pandas** **as** **pd**, **joblib**

**from** **t20.config** **import** ARTIFACTS\_DIR

**from** **t20.splits** **import** temporal\_split

**from** **t20.evaluation** **import** reliability\_diagram

feat = pd.read\_csv(ARTIFACTS\_DIR / "features.csv.gz", compression="gzip", low\_memory=**False**)

train, valid, test = temporal\_split(feat)

xw\_model = joblib.load(ARTIFACTS\_DIR / "xw\_model.joblib")

p = xw\_model.predict\_proba(valid.drop(columns=['target\_xw','target\_xr']))[:,**1**]

reliability\_diagram(valid['target\_xw'].values, p, "xW Calibration (Validation)", str(ARTIFACTS\_DIR / "xw\_calibration.png"))

print("Saved calibration plot.")

**15. Simulate\_prescribe**

**import** **pandas** **as** **pd**, **joblib**

**from** **t20.config** **import** ARTIFACTS\_DIR

**from** **t20.simulation** **import** monte\_carlo\_innings

**from** **t20.prescribe** **import** optimize\_bowling\_sequence

feat = pd.read\_csv(ARTIFACTS\_DIR / "features.csv.gz", compression="gzip", low\_memory=**False**)

xw\_model = joblib.load(ARTIFACTS\_DIR / "xw\_model.joblib")

xr\_model = joblib.load(ARTIFACTS\_DIR / "xr\_model.joblib")

# Example: take one innings worth of rows from a random match and optimize over over-order indices

m = feat['match\_id'].iloc[**0**]

inn = **1**

sample = feat[(feat['match\_id']==m) & (feat['innings']==inn)].reset\_index(drop=**True**)

# group rows into over-level feature rows (mean as placeholder)

X\_rows = [sample.iloc[[i]] **for** i **in** range(min(**120**, len(sample)))]

mu, sd = monte\_carlo\_innings(xr\_model, xw\_model, sample.iloc[:**60**], iters=**50**)

print("Baseline MC expected runs (first 60 balls):", mu, "+/-", sd)

best, score = optimize\_bowling\_sequence(xr\_model, xw\_model, X\_rows[:**20**], iters=**50**)

print("Optimized order (first 20 rows indices):", best)

print("Objective score (negative expected runs):", score)

**16. Config**

**from** **pathlib** **import** Path

PROJECT\_ROOT = Path("/mnt/data/t20i\_tactical\_analytics")

DATA\_DIR = PROJECT\_ROOT / "data"

FIG\_DIR = PROJECT\_ROOT / "figures"

ARTIFACTS\_DIR = PROJECT\_ROOT / "artifacts"

ARTIFACTS\_DIR.mkdir(exist\_ok=**True**, parents=**True**)

BALLS\_CSV = DATA\_DIR / "t20\_balls.csv.gz"

MATCHES\_CSV = DATA\_DIR / "t20\_matches.csv.gz"

**17. Data\_io**

**from** **pathlib** **import** Path

**import** **pandas** **as** **pd**

**from** **.config** **import** BALLS\_CSV, MATCHES\_CSV

**def** **load\_balls**():

**return** pd.read\_csv(BALLS\_CSV, compression='gzip', low\_memory=**False**)

**def** **load\_matches**():

**return** pd.read\_csv(MATCHES\_CSV, compression='gzip', low\_memory=**False**)

**18. Evaluation**

**import** **numpy** **as** **np**, **matplotlib.pyplot** **as** **plt**

**from** **sklearn.calibration** **import** calibration\_curve

**def** **reliability\_diagram**(y\_true, y\_prob, title, path):

frac\_pos, mean\_pred = calibration\_curve(y\_true, y\_prob, n\_bins=**10**, strategy='uniform')

plt.figure()

plt.plot([**0**,**1**],[**0**,**1**], linestyle='--')

plt.plot(mean\_pred, frac\_pos, marker='o')

plt.title(title)

plt.xlabel('Predicted probability')

plt.ylabel('Observed frequency')

plt.tight\_layout()

plt.savefig(path)

plt.close()

**19. Feature\_engineering**

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

PHASE\_BOUNDS = {'powerplay': (**1**,**6**), 'middle': (**7**,**15**), 'death': (**16**,**20**)}

**def** **add\_time\_features**(balls: pd.DataFrame) -> pd.DataFrame:

df = balls.copy()

df['year'] = pd.to\_datetime(df['date']).dt.year

df['ball\_index'] = (df['over'] - **1**) \* **6** + df['ball\_in\_over']

df['is\_wicket'] = df['wicket\_player\_out'].notna().astype(int)

df['phase'] = pd.cut(df['over'],

bins=[**0**,**6**,**15**,**20**],

labels=['powerplay','middle','death'],

right=**True**).astype(str)

**return** df

**def** **compute\_match\_state**(df: pd.DataFrame) -> pd.DataFrame:

df = df.sort\_values(['match\_id','innings','over','ball\_in\_over']).copy()

df['runs\_cum'] = df.groupby(['match\_id','innings'])['runs\_total'].cumsum()

df['wickets\_cum'] = df.groupby(['match\_id','innings'])['is\_wicket'].cumsum()

df['balls\_elapsed'] = df.groupby(['match\_id','innings']).cumcount()+**1**

df['balls\_remaining'] = **120** - df['balls\_elapsed']

df['crr'] = df['runs\_cum'] / (df['balls\_elapsed']/**6.0**)

**return** df

**def** **add\_lag\_features**(df: pd.DataFrame, group\_cols, cols, lags=[**1**,**2**,**6**]):

out = df.copy()

**for** c **in** cols:

**for** l **in** lags:

out[f'{c}\_lag{l}'] = out.groupby(group\_cols)[c].shift(l)

**return** out

**def** **rolling\_form**(df: pd.DataFrame, window=**60**):

d = df.copy()

d['bf\_one'] = **1**

batter = (d.groupby('striker')

.apply(**lambda** g: g.assign(

batter\_runs\_rolling=g['runs\_batter'].rolling(window, min\_periods=**10**).sum(),

batter\_balls\_rolling=g['bf\_one'].rolling(window, min\_periods=**10**).sum()))

.reset\_index(drop=**True**))

batter['batter\_sr\_rolling'] = **100.0** \* batter['batter\_runs\_rolling'] / batter['batter\_balls\_rolling']

bowler = (d.groupby('bowler')

.apply(**lambda** g: g.assign(

bowler\_runs\_rolling=g['runs\_total'].rolling(window, min\_periods=**18**).sum(),

bowler\_balls\_rolling=g['bf\_one'].rolling(window, min\_periods=**18**).sum()))

.reset\_index(drop=**True**))

bowler['bowler\_econ\_rolling'] = **6.0** \* bowler['bowler\_runs\_rolling'] / bowler['bowler\_balls\_rolling']

res = d[['match\_id','innings','over','ball\_in\_over','striker','bowler']].copy()

res = res.merge(batter[['striker','batter\_sr\_rolling']], on='striker', how='left')

res = res.merge(bowler[['bowler','bowler\_econ\_rolling']], on='bowler', how='left')

**return** res

**def** **build\_features**(balls: pd.DataFrame):

df = add\_time\_features(balls)

df = compute\_match\_state(df)

df = add\_lag\_features(df, ['match\_id','innings'], ['runs\_total'], lags=[**1**,**2**,**6**,**12**])

**for** col **in** ['batting\_team','striker','bowler','venue','phase']:

df[col] = df[col].astype('category')

df['target\_xw'] = df['is\_wicket']

df['target\_xr'] = df['runs\_total']

**return** df

**import** **numpy** **as** **np**, **pandas** **as** **pd**

**from** **sklearn.preprocessing** **import** OneHotEncoder

**from** **sklearn.compose** **import** ColumnTransformer

**from** **sklearn.pipeline** **import** Pipeline

**from** **sklearn.metrics** **import** roc\_auc\_score, log\_loss, brier\_score\_loss, mean\_squared\_error, mean\_absolute\_error

**from** **sklearn.calibration** **import** CalibratedClassifierCV

**from** **sklearn.ensemble** **import** GradientBoostingClassifier, GradientBoostingRegressor

**from** **sklearn.impute** **import** SimpleImputer

**def** **build\_preprocessor**(df, numeric\_cols, cat\_cols):

**return** ColumnTransformer([

("num", SimpleImputer(strategy="median"), numeric\_cols),

("cat", OneHotEncoder(handle\_unknown="ignore"), cat\_cols),

])

**def** **train\_xw**(train, valid, features):

X\_tr, y\_tr = train[features], train['target\_xw']

X\_va, y\_va = valid[features], valid['target\_xw']

num\_cols = X\_tr.select\_dtypes(include=[np.number]).columns.tolist()

cat\_cols = [c **for** c **in** X\_tr.columns **if** c **not** **in** num\_cols]

pre = build\_preprocessor(train, num\_cols, cat\_cols)

base = GradientBoostingClassifier()

pipe = Pipeline([("pre", pre), ("clf", base)]).fit(X\_tr, y\_tr)

calib = CalibratedClassifierCV(pipe, method="isotonic", cv="prefit")

calib.fit(X\_va, y\_va)

p\_va = calib.predict\_proba(X\_va)[:,**1**]

metrics = {

"AUC": roc\_auc\_score(y\_va, p\_va),

"LogLoss": log\_loss(y\_va, p\_va),

"Brier": brier\_score\_loss(y\_va, p\_va),

}

**return** calib, metrics

**def** **train\_xr**(train, valid, features):

X\_tr, y\_tr = train[features], train['target\_xr']

X\_va, y\_va = valid[features], valid['target\_xr']

num\_cols = X\_tr.select\_dtypes(include=[np.number]).columns.tolist()

cat\_cols = [c **for** c **in** X\_tr.columns **if** c **not** **in** num\_cols]

pre = build\_preprocessor(train, num\_cols, cat\_cols)

base = GradientBoostingRegressor()

pipe = Pipeline([("pre", pre), ("reg", base)]).fit(X\_tr, y\_tr)

p\_va = pipe.predict(X\_va)

metrics = {

"RMSE": mean\_squared\_error(y\_va, p\_va, squared=**False**),

"MAE": mean\_absolute\_error(y\_va, p\_va),

}

**return** pipe, metrics

**20. Models**

**import** **numpy** **as** **np**, **pandas** **as** **pd**

**from** **sklearn.preprocessing** **import** OneHotEncoder

**from** **sklearn.compose** **import** ColumnTransformer

**from** **sklearn.pipeline** **import** Pipeline

**from** **sklearn.metrics** **import** roc\_auc\_score, log\_loss, brier\_score\_loss, mean\_squared\_error, mean\_absolute\_error

**from** **sklearn.calibration** **import** CalibratedClassifierCV

**from** **sklearn.ensemble** **import** GradientBoostingClassifier, GradientBoostingRegressor

**from** **sklearn.impute** **import** SimpleImputer

**def** **build\_preprocessor**(df, numeric\_cols, cat\_cols):

**return** ColumnTransformer([

("num", SimpleImputer(strategy="median"), numeric\_cols),

("cat", OneHotEncoder(handle\_unknown="ignore"), cat\_cols),

])

**def** **train\_xw**(train, valid, features):

X\_tr, y\_tr = train[features], train['target\_xw']

X\_va, y\_va = valid[features], valid['target\_xw']

num\_cols = X\_tr.select\_dtypes(include=[np.number]).columns.tolist()

cat\_cols = [c **for** c **in** X\_tr.columns **if** c **not** **in** num\_cols]

pre = build\_preprocessor(train, num\_cols, cat\_cols)

base = GradientBoostingClassifier()

pipe = Pipeline([("pre", pre), ("clf", base)]).fit(X\_tr, y\_tr)

calib = CalibratedClassifierCV(pipe, method="isotonic", cv="prefit")

calib.fit(X\_va, y\_va)

p\_va = calib.predict\_proba(X\_va)[:,**1**]

metrics = {

"AUC": roc\_auc\_score(y\_va, p\_va),

"LogLoss": log\_loss(y\_va, p\_va),

"Brier": brier\_score\_loss(y\_va, p\_va),

}

**return** calib, metrics

**def** **train\_xr**(train, valid, features):

X\_tr, y\_tr = train[features], train['target\_xr']

X\_va, y\_va = valid[features], valid['target\_xr']

num\_cols = X\_tr.select\_dtypes(include=[np.number]).columns.tolist()

cat\_cols = [c **for** c **in** X\_tr.columns **if** c **not** **in** num\_cols]

pre = build\_preprocessor(train, num\_cols, cat\_cols)

base = GradientBoostingRegressor()

pipe = Pipeline([("pre", pre), ("reg", base)]).fit(X\_tr, y\_tr)

p\_va = pipe.predict(X\_va)

metrics = {

"RMSE": mean\_squared\_error(y\_va, p\_va, squared=**False**),

"MAE": mean\_absolute\_error(y\_va, p\_va),

}

**return** pipe, metrics

**21. Splits**

**import** **pandas** **as** **pd**

**def** **temporal\_split**(df: pd.DataFrame, train\_end=**2023**, valid\_year=**2024**, test\_year=**2025**):

train = df[df['year'] <= train\_end]

valid = df[df['year'] == valid\_year]

test = df[df['year'] == test\_year]

**return** train, valid, test

**22. Simulation**

**import** **numpy** **as** **np**, **pandas** **as** **pd**

**def** **simulate\_over**(xr\_model, xw\_model, feature\_row, balls=**6**, rng=**None**):

rng = rng **or** np.random.default\_rng()

runs = **0**

wkts = **0**

**for** \_ **in** range(balls):

xr = float(xr\_model.predict(feature\_row)[**0**])

pw = float(xw\_model.predict\_proba(feature\_row)[:,**1**][**0**])

**if** rng.uniform() < pw:

wkts += **1**

r = max(**0**, int(np.round(rng.normal(loc=xr, scale=**1.5**))))

runs += r

**return** runs, wkts

**def** **monte\_carlo\_innings**(xr\_model, xw\_model, X\_rows, iters=**250**, rng=**None**):

rng = rng **or** np.random.default\_rng()

totals = []

**for** i **in** range(iters):

runs = **0**; wkts = **0**

**for** idx **in** range(len(X\_rows)):

r, w = simulate\_over(xr\_model, xw\_model, X\_rows.iloc[[idx]], balls=**1**, rng=rng)

runs += r; wkts += w

**if** wkts >= **10**:

**break**

totals.append(runs)

**return** np.mean(totals), np.std(totals)

**23. Prescribe**

**import** **numpy** **as** **np**, **pandas** **as** **pd**

**from** **.simulation** **import** monte\_carlo\_innings

**def** **swap\_search**(order, score\_fn, iters=**100**, rng=**None**):

rng = rng **or** np.random.default\_rng()

best = order.copy()

best\_score = score\_fn(best)

**for** \_ **in** range(iters):

i, j = rng.integers(**0**, len(order), size=**2**)

cand = best.copy()

cand[i], cand[j] = cand[j], cand[i]

s = score\_fn(cand)

**if** s > best\_score:

best, best\_score = cand, s

**return** best, best\_score

**def** **optimize\_bowling\_sequence**(xr\_model, xw\_model, feature\_rows\_by\_over, iters=**200**):

base\_order = list(range(len(feature\_rows\_by\_over)))

**def** **score**(ordr):

X = pd.concat([feature\_rows\_by\_over[i] **for** i **in** ordr], ignore\_index=**True**)

mu, sd = monte\_carlo\_innings(xr\_model, xw\_model, X, iters=**50**)

**return** -mu

best, score = swap\_search(base\_order, score, iters=iters)

**return** best, score

**24. Parse cricsheet**

**import** **re**, **json**, **pandas** **as** **pd**

**def** **parse\_match\_json\_v11**(obj, source\_name=**None**):

info = obj.get("info", {})

innings = obj.get("innings", [])

match\_id = re.sub(r"\.json$", "", source\_name) **if** source\_name **else** **None**

date\_val = info.get("dates")

match\_date = **None**

**if** isinstance(date\_val, list) **and** date\_val:

**try**:

match\_date = pd.to\_datetime(date\_val[**0**]).date()

**except** **Exception**:

**pass**

city = info.get("city")

venue = info.get("venue")

teams = info.get("teams", [])

team1, team2 = (teams + [**None**, **None**])[:**2**]

gender = info.get("gender")

match\_type = info.get("match\_type")

season = info.get("season")

toss = info.get("toss", {})

toss\_winner = toss.get("winner")

toss\_decision = toss.get("decision")

pom = (info.get("player\_of\_match") **or** [**None**])

pom = pom[**0**] **if** isinstance(pom, list) **else** pom

officials = info.get("officials", {}) **or** {}

umpires = officials.get("umpires") **or** info.get("umpires") **or** []

referee = officials.get("referee")

rows = []

**for** inn\_idx, inn **in** enumerate(innings, start=**1**):

batting\_team = inn.get("team")

**for** over\_block **in** inn.get("overs", []):

over\_num = over\_block.get("over")

**for** ball\_idx, detail **in** enumerate(over\_block.get("deliveries", []), start=**1**):

striker = detail.get("batter") **or** detail.get("batsman")

non\_striker = detail.get("non\_striker")

bowler = detail.get("bowler")

runs = detail.get("runs", {})

runs\_batter = runs.get("batter") **if** "batter" **in** runs **else** runs.get("batsman", **0**)

runs\_extras = runs.get("extras", **0**)

runs\_total = runs.get("total", runs\_batter + runs\_extras)

wicket\_info = detail.get("wickets") **or** detail.get("wicket")

wicket\_kind = **None**

wicket\_player\_out = **None**

**if** isinstance(wicket\_info, list) **and** wicket\_info:

wicket\_kind = wicket\_info[**0**].get("kind")

wicket\_player\_out = wicket\_info[**0**].get("player\_out")

**elif** isinstance(wicket\_info, dict):

wicket\_kind = wicket\_info.get("kind")

wicket\_player\_out = wicket\_info.get("player\_out")

extras\_detail = detail.get("extras", {}) **or** {}

bye = extras\_detail.get("byes") **or** **0**

legbye = extras\_detail.get("legbyes") **or** extras\_detail.get("leg\_byes") **or** **0**

noball = extras\_detail.get("noballs") **or** extras\_detail.get("noball") **or** **0**

wide = extras\_detail.get("wides") **or** **0**

penalty = extras\_detail.get("penalty") **or** **0**

rows.append({

"match\_id": match\_id,

"date": match\_date,

"season": season,

"city": city,

"venue": venue,

"gender": gender,

"match\_type": match\_type,

"team1": team1, "team2": team2,

"batting\_team": batting\_team,

"innings": inn\_idx,

"over": over\_num, "ball\_in\_over": ball\_idx,

"striker": striker, "non\_striker": non\_striker, "bowler": bowler,

"runs\_batter": runs\_batter, "runs\_extras": runs\_extras, "runs\_total": runs\_total,

"extra\_byes": bye, "extra\_legbyes": legbye, "extra\_noballs": noball, "extra\_wides": wide, "extra\_penalty": penalty,

"wicket\_kind": wicket\_kind, "wicket\_player\_out": wicket\_player\_out,

"toss\_winner": toss\_winner, "toss\_decision": toss\_decision,

"player\_of\_match": pom,

"umpires": ", ".join(umpires) **if** isinstance(umpires, list) **else** umpires,

"referee": referee,

})

balls\_df = pd.DataFrame(rows)

match\_meta = {

"match\_id": match\_id,

"date": match\_date,

"season": season, "city": city, "venue": venue, "gender": gender, "match\_type": match\_type,

"team1": team1, "team2": team2, "toss\_winner": toss\_winner, "toss\_decision": toss\_decision,

"player\_of\_match": pom,

}

**return** balls\_df, match\_meta

**25. Visuals**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

**def** **matches\_per\_year\_plot**(matches: pd.DataFrame, path: str):

s = matches.assign(year=pd.to\_datetime(matches["date"]).dt.year).groupby("year").size()

plt.figure()

s.plot(kind="bar")

plt.title("T20I Matches per Year")

plt.xlabel("Year")

plt.ylabel("Matches")

plt.tight\_layout()

plt.savefig(path)

plt.close()