```
        ...


class ScorelineModel(BaseComponent):
    """
    Model that outputs a full scoreline PMF matrix for each sample.
    P[home_goals, away_goals].
    """
    max_goals: int = 8

    @abstractmethod
    def fit(self, matches: Any) -> None:
        """Matches = iterable of objects with (home_team, away_team, goals, date,...)."""
        ...

    @abstractmethod
    def predict_score_matrix(self, match_state: Any) -> np.ndarray:
        """
        match_state: object containing teams, league, maybe current score/state.
        Returns: (max_goals+1, max_goals+1) matrix that sums to 1.
        """
        ...

    def outcome_probs_from_matrix(self, matrix: np.ndarray) -> Dict[str, float]:
        home_win = matrix[np.tril_indices_from(matrix, -1)].sum()
        draw = matrix.diagonal().sum()
```

```python
        away_win = matrix[np.triu_indices_from(matrix, 1)].sum()
        return {"home_win": float(home_win),
                "draw": float(draw),
                "away_win": float(away_win)}


class TotalsModel(BaseComponent):
    """
    Models distribution of totals (goals, cards, corners) for a team or match.
    """
    @abstractmethod
    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        ...

    @abstractmethod
    def pmf(self, X: np.ndarray, max_total: int = 10) -> np.ndarray:
        """Return P(T = 0..max_total) for each row in X."""
        ...


class SurvivalModel(BaseComponent):
    """
    Hazard model for time-to-next-goal or similar.
    """
    @abstractmethod
    def fit(self, X: np.ndarray, durations: np.ndarray, events: np.ndarray) -> None:
```

```python
        ...

    @abstractmethod
    def hazard(self, X: np.ndarray, t: float) -> np.ndarray:
        """Hazard at time t (minutes remaining or elapsed)."""
        ...


class RatingModel(BaseComponent):
    """
    Systems like Elo, Glicko, TrueSkill, Bradley-Terry.
    """
    @abstractmethod
    def update_from_matches(self, matches: Any) -> None:
        ...

    @abstractmethod
    def get_rating(self, team_id: int, when: Optional[Any] = None) -> float:
        ...


class SimulationModel(BaseComponent):
    """
    Monte Carlo simulator that uses some base model(s).
    """
    @abstractmethod
```

```python
    def simulate(self, match_state: Any, n_sims: int = 10000) -> Dict[str, Any]:
        ...


class PreprocessorModel(BaseComponent):
    """
    PCA, KMeans, GMM, etc.
    """
    @abstractmethod
    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        ...

    @abstractmethod
    def transform(self, X: np.ndarray) -> np.ndarray:
        ...


class SequenceModel(BaseComponent):
    """
    LSTM/Transformer-style models on sequences of feature vectors.
    """
    @abstractmethod
    def fit_sequences(self, sequences: Any, targets: np.ndarray) -> None:
        ...

    @abstractmethod
```

```python
    def predict_sequence(self, sequence: Any) -> Any:
        ...
```

2.3. app/core/registry.py

```python
from typing import Dict, Type
from .base import BaseComponent


COMPONENT_REGISTRY: Dict[str, Type[BaseComponent]] = {}


def register_component(cls: Type[BaseComponent]):
    COMPONENT_REGISTRY[cls.__name__] = cls
    return cls


def get_component_class(name: str) -> Type[BaseComponent]:
    return COMPONENT_REGISTRY[name]


def list_components() -> Dict[str, Type[BaseComponent]]:
    return COMPONENT_REGISTRY.copy()
```

Now, each of your 36 models becomes a **concrete class** inheriting from one of these, registered with @register_component.

---

3. Mapping the 36 models to base types

Here's how each row in your table maps:

#	Model	Base type
1	Poisson goal	ScorelineModel
2	Dixon–Coles Poisson	ScorelineModel

#	Model	Base type
3	Bivariate Poisson	ScorelineModel
4	Skellam	SupervisedModel (OUTCOME on goal difference)
5	Negative Binomial	TotalsModel
6	Zero-inflated Poisson/NB	ScorelineModel + TotalsModel
7	Bayesian hier. Poisson	ScorelineModel
8	Ordered logit/probit	SupervisedModel (OUTCOME)
9	GAM	SupervisedModel (OUTCOME or REGRESSION)
10	Survival/time-to-event	SurvivalModel
11	Copula joint score	ScorelineModel
12	Markov chain (EPV/xT)	SimulationModel or bespoke xT component; used as feature provider
13	Hawkes process	SurvivalModel / event-intensity component
14	xG logistic	SupervisedModel on shot-level data
15	Elo	RatingModel
16	Glicko/TrueSkill	RatingModel
17	Bradley–Terry/Davidson	RatingModel
18	Monte-Carlo on score model	SimulationModel
19	Linear Regression	SupervisedModel (REGRESSION)
20	Logistic Regression	SupervisedModel (OUTCOME)
21	Decision Tree	SupervisedModel
22	Random Forest	SupervisedModel

#	Model	Base type
23	KNN	SupervisedModel
24	Naive Bayes	SupervisedModel
25	SVM	SupervisedModel
26	AdaBoost	SupervisedModel
27	XGBoost	SupervisedModel
28	K-Means	PreprocessorModel (clustering)
29	Hierarchical clustering	PreprocessorModel
30	DBSCAN	PreprocessorModel
31	PCA	PreprocessorModel (dim reduction)
32	MLP	SupervisedModel
33	CNN	SupervisedModel / SequenceModel (if using spatial/temporal encoding)
34	LSTM	SequenceModel
35	GMM	PreprocessorModel (density/clustering)
36	Transformer TS model	SequenceModel

---

4. Representative implementations for each family

I'll show code for a few key ones; the others follow exactly the same pattern.

4.1. #1 – Poisson goal model (independent Poisson)

app/models/statistical/poisson_goal.py

import numpy as np

from dataclasses import dataclass

from typing import Any

```
from app.core.base import ScorelineModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@dataclass
class MatchState:
    home_attack: float
    away_attack: float
    home_defense: float
    away_defense: float
    home_advantage: float = 0.0
    # You can add league, date, etc.


@register_component
class PoissonGoalModel(ScorelineModel):
    def __init__(self, max_goals: int = 8) -> None:
        self.name = "PoissonGoalModel"
        self.task_type = TaskType.SCORELINE
        self.mode = Mode.PREMATCH
        self.max_goals = max_goals
        # In a full implementation you'd learn attack/defence params per team.
        # Here we assume those are precomputed and passed via MatchState.

    def fit(self, matches: Any) -> None:
        """
        Implement maximum-likelihood fitting for team attack/defence strengths
```

```
        using match history (Poisson regression or log-linear model).
        For brevity: left as TODO.
        """
        raise NotImplementedError("Fit Poisson attack/defence per team here.")

    def _poisson_pmf(self, k: int, lam: float) -> float:
        from math import exp, factorial
        return exp(-lam) * lam**k / factorial(k)

    def predict_score_matrix(self, match_state: MatchState) -> np.ndarray:
        lam_home = np.exp(match_state.home_attack - match_state.away_defense +
                          match_state.home_advantage)
        lam_away = np.exp(match_state.away_attack - match_state.home_defense)

        g = self.max_goals
        pm = np.zeros((g+1, g+1))
        for h in range(g+1):
            for a in range(g+1):
                pm[h, a] = self._poisson_pmf(h, lam_home) * self._poisson_pmf(a, lam_away)
        pm /= pm.sum()
        return pm
```

You'll have similar classes for:

- DixonColesPoissonModel – same base, but apply DC correlation factor.
- BivariatePoissonModel – same base, with bivariate Poisson PMF.
- ZeroInflatedScoreModel – uses zero-inflation factor.

4.2. #4 – Skellam (goal difference) model as outcome

app/models/statistical/skellam_diff.py

```python
import numpy as np
from scipy.stats import skellam
from app.core.base import SupervisedModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class SkellamOutcomeModel(SupervisedModel):
    """
    Uses lambda_home and lambda_away (from Poisson) and models goal difference.
    """
    def __init__(self) -> None:
        self.name = "SkellamOutcomeModel"
        self.task_type = TaskType.OUTCOME
        self.mode = Mode.PREMATCH

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """
        Typically you don't 'fit' Skellam directly; you fit lambdas via Poisson model.
        Here we treat X as [lambda_home, lambda_away] so there is no training.
        """
        # No training required if lambdas come from another model.
        pass

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
```

```
    """
    X[:,0] = lambda_home, X[:,1] = lambda_away
    Outputs [P(home_win), P(draw), P(away_win)].
    """
    probs = []
    for lam_home, lam_away in X:
        # Skellam distribution for D = home_goals - away_goals
        # We'll approximate by summing D from -10..10
        d_vals = np.arange(-10, 11)
        pmf = skellam.pmf(d_vals, lam_home, lam_away)
        p_home = pmf[d_vals > 0].sum()
        p_draw = pmf[d_vals == 0].sum()
        p_away = pmf[d_vals < 0].sum()
        probs.append([p_home, p_draw, p_away])
    return np.array(probs)
```

4.3. #5 – Negative Binomial totals model

app/models/statistical/negbin_totals.py

```
import numpy as np
from statsmodels.discrete.discrete_model import NegativeBinomial
from app.core.base import TotalsModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class NegBinTotalsModel(TotalsModel):
    """
```

```
Models total goals / cards / corners with Negative Binomial regression.
"""
def __init__(self) -> None:
    self.name = "NegBinTotalsModel"
    self.task_type = TaskType.TOTALS
    self.mode = Mode.PREMATCH
    self.model = None

def fit(self, X: np.ndarray, y: np.ndarray) -> None:
    # Add intercept
    import statsmodels.api as sm
    X_ = sm.add_constant(X)
    self.model = NegativeBinomial(y, X_).fit(disp=0)

def pmf(self, X: np.ndarray, max_total: int = 10) -> np.ndarray:
    import statsmodels.api as sm
    if self.model is None:
        raise RuntimeError("Model not fit")
    X_ = sm.add_constant(X)
    mu = self.model.predict(X_)
    # Convert NB parameters to pmf for each integer 0..max_total
    # Here we use a simple approximation: treat as Poisson with mean mu
    # In full implementation, you'd use NB pmf with alpha from model.
    probs = []
    from math import exp, factorial
    for m in mu:
```

```python
        p = np.array([exp(-m) * m**k / factorial(k) for k in range(max_total+1)])
        p /= p.sum()
        probs.append(p)
    return np.vstack(probs)
```

(Real NB pmf would use model's dispersion; kept short here.)

4.4. #10 – Survival / time-to-event model

We'll sketch with lifelines (you can add to requirements):

app/models/statistical/survival_goal.py

```python
import numpy as np
from lifelines import CoxPHFitter
from app.core.base import SurvivalModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class CoxGoalSurvivalModel(SurvivalModel):
    """
    Cox proportional hazards model for time-to-next-goal.
    X contains in-play state at the start of an interval; durations are minutes
    until next goal (or censoring).
    """
    def __init__(self) -> None:
        self.name = "CoxGoalSurvivalModel"
        self.task_type = TaskType.SURVIVAL
        self.mode = Mode.INPLAY
        self.cph = CoxPHFitter()
```

```
    def fit(self, X: np.ndarray, durations: np.ndarray, events: np.ndarray) -> None:
        import pandas as pd
        df = pd.DataFrame(X)
        df["T"] = durations
        df["E"] = events
        self.cph.fit(df, duration_col="T", event_col="E")

    def hazard(self, X: np.ndarray, t: float) -> np.ndarray:
        import pandas as pd
        df = pd.DataFrame(X)
        # Partial estimation: we use baseline hazard from fitted CPH
        base_cum_haz = self.cph.baseline_cumulative_hazard_.iloc[:, 0]
        # pick closest time to t
        idx = (base_cum_haz.index - t).abs().argmin()
        base_h = base_cum_haz.iloc[idx]
        # hazard ~ base_h * exp(beta^T x)
        lin_pred = self.cph.predict_partial_hazard(df).values.flatten()
        return base_h * lin_pred
```

4.5. #14 – Expected Goals (xG) logistic

app/models/statistical/xg_logistic.py

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from app.core.base import SupervisedModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode
```

```python
@register_component
class XGShotLogisticModel(SupervisedModel):
    """
    Shot-level xG model: predicts probability that a shot is a goal.
    """
    def __init__(self) -> None:
        self.name = "XGShotLogisticModel"
        self.task_type = TaskType.OUTCOME  # binary (goal vs no-goal)
        self.mode = Mode.BOTH
        self.model = LogisticRegression(max_iter=500)

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        self.model.fit(X, y)

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        """
        Returns [P(no-goal), P(goal)] per shot.
        """
        return self.model.predict_proba(X)
```

Your **shot-level feature builder** (distance, angle, body part, etc.) plugs into this.

4.6. #15–17 – rating systems

Example: Elo.

app/models/ratings/elo.py (refactoring previous version):

```python
from dataclasses import dataclass
from typing import Any, Optional
```

```
from sqlalchemy.orm import Session
from datetime import date
from app.core.base import RatingModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode
from app.db.models import Match, RatingHistory


def expected_score(r_a: float, r_b: float) -> float:
    return 1.0 / (1.0 + 10 ** ((r_b - r_a) / 400))


@dataclass
class EloConfig:
    k: float = 20.0
    initial_rating: float = 1500.0


@register_component
class EloRatingSystem(RatingModel):
    def __init__(self, config: EloConfig | None = None) -> None:
        self.name = "EloRatingSystem"
        self.task_type = TaskType.RATING
        self.mode = Mode.BOTH
        self.config = config or EloConfig()
        self._cache: dict[int, float] = {}

    def update_from_matches(self, matches: list[Match], session: Session) -> None:
        for m in sorted(matches, key=lambda x: x.date):
```

```
            h = self.get_rating(m.home_team_id, m.date)
            a = self.get_rating(m.away_team_id, m.date)
            exp_h = expected_score(h, a)
            exp_a = 1 - exp_h

            hg = m.home_ft_goals or 0
            ag = m.away_ft_goals or 0
            if hg > ag:
                act_h, act_a = 1.0, 0.0
            elif hg == ag:
                act_h, act_a = 0.5, 0.5
            else:
                act_h, act_a = 0.0, 1.0

            new_h = h + self.config.k * (act_h - exp_h)
            new_a = a + self.config.k * (act_a - exp_a)

            session.add(RatingHistory(team_id=m.home_team_id, date=m.date,
                                      rating_type="elo", rating=new_h))
            session.add(RatingHistory(team_id=m.away_team_id, date=m.date,
                                      rating_type="elo", rating=new_a))
            self._cache[m.home_team_id] = new_h
            self._cache[m.away_team_id] = new_a
        session.commit()

    def get_rating(self, team_id: int, when: Optional[date] = None) -> float:
```

```
        # In memory cache fallback; in production, read from DB for date <= when
        return self._cache.get(team_id, self.config.initial_rating)
```

Similar pattern for:

- **Glicko/TrueSkill** → GlickoRatingSystem, TrueSkillRatingSystem.
- **Bradley–Terry** → BradleyTerrySystem (fit strengths via MLE).

4.7. #18 – Monte Carlo on top of score model

app/models/simulation/score_monte_carlo.py

```
import numpy as np
from dataclasses import dataclass
from typing import Any, Dict
from app.core.base import SimulationModel, ScorelineModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@dataclass
class MCConfig:
    n_sims: int = 20000
    max_goals: int = 8


@register_component
class ScoreMonteCarloSimulator(SimulationModel):
    """
    Wraps any ScorelineModel (Poisson, DC, Bivariate, Copula etc.)
    and simulates full match or remaining time (if you adjust lambdas).
    """
    def __init__(self, base_model: ScorelineModel, config: MCConfig | None = None):
```

```python
    self.name = "ScoreMonteCarloSimulator"
    self.task_type = TaskType.SIMULATION
    self.mode = Mode.BOTH
    self.base_model = base_model
    self.config = config or MCConfig()

def simulate(self, match_state: Any, n_sims: int | None = None) -> Dict[str, Any]:
    n = n_sims or self.config.n_sims
    matrix = self.base_model.predict_score_matrix(match_state)
    # Flatten PMF for sampling
    pmf_flat = matrix.flatten()
    outcomes = np.random.choice(
        matrix.size,
        size=n,
        p=pmf_flat
    )
    hg, ag = np.divmod(outcomes, matrix.shape[1])
    home_win = (hg > ag).mean()
    draw = (hg == ag).mean()
    away_win = (hg < ag).mean()
    return {
        "home_win": float(home_win),
        "draw": float(draw),
        "away_win": float(away_win),
        "sampled_scores": list(zip(hg.tolist(), ag.tolist()))
    }
```

4.8. #19–27 – classical ML models

Refactor earlier LogisticOutcome etc. to inherit from SupervisedModel.

Example: app/models/ml/classical/logistic_cls.py for #20.

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from app.core.base import SupervisedModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class LogisticOutcomeModel(SupervisedModel):
    def __init__(self) -> None:
        self.name = "LogisticOutcomeModel"
        self.task_type = TaskType.OUTCOME
        self.mode = Mode.PREMATCH
        self.model = LogisticRegression(max_iter=500, multi_class="multinomial")

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        self.model.fit(X, y)

    def predict_proba(self, X: np.ndarray) -> np.ndarray:
        return self.model.predict_proba(X)
```

RandomForest, SVM, XGBoost, etc. are analogous—same base, different underlying estimator.

4.9. #28–31, 35 – clustering & PCA

app/models/ml/unsupervised/kmeans_clusters.py:

```
import numpy as np
```

```
from sklearn.cluster import KMeans
from app.core.base import PreprocessorModel
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class KMeansTeamClusterer(PreprocessorModel):
    """
    Clusters teams/matches into style clusters (attack-heavy, defense-heavy, etc.).
    Used mostly offline or as feature transformer: cluster id as feature.
    """
    def __init__(self, n_clusters: int = 6) -> None:
        self.name = "KMeansTeamClusterer"
        self.task_type = TaskType.PREPROCESSOR
        self.mode = Mode.BOTH
        self.model = KMeans(n_clusters=n_clusters, random_state=42)

    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.fit_predict(X).reshape(-1, 1)

    def transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.predict(X).reshape(-1, 1)
```

app/models/ml/dim_reduction/pca_reducer.py:

```
import numpy as np
from sklearn.decomposition import PCA
from app.core.base import PreprocessorModel
```

```
from app.core.registry import register_component
from app.core.tasks import TaskType, Mode


@register_component
class PCAMatchReducer(PreprocessorModel):
    def __init__(self, n_components: int = 10) -> None:
        self.name = "PCAMatchReducer"
        self.task_type = TaskType.PREPROCESSOR
        self.mode = Mode.BOTH
        self.model = PCA(n_components=n_components)

    def fit_transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.fit_transform(X)

    def transform(self, X: np.ndarray) -> np.ndarray:
        return self.model.transform(X)
```

GMM / DBSCAN / Hierarchical clustering implement the same interface.

4.10. #32–34 & #36 – deep & sequence models (LSTM/Transformer)

We'll sketch an LSTM model using PyTorch (you can swap for Keras if you prefer):

app/models/ml/deep/lstm_seq.py

```
import numpy as np
from typing import Any
import torch
from torch import nn
from app.core.base import SequenceModel
from app.core.registry import register_component
```

```
from app.core.tasks import TaskType, Mode


class _LSTMNet(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int = 64, num_layers: int = 1, num_classes: int =
3):
        super().__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_classes)


    def forward(self, x):
        # x: (batch, seq_len, input_dim)
        out, _ = self.lstm(x)
        # take last time step
        out = out[:, -1, :]
        out = self.fc(out)
        return out


@register_component
class LSTMOutcomeSequenceModel(SequenceModel):
    """
    Sequence model for outcome prediction based on last N matches or in-play event streams.
    """
    def __init__(self, input_dim: int = 16, hidden_dim: int = 64) -> None:
        self.name = "LSTMOutcomeSequenceModel"
        self.task_type = TaskType.SEQUENCE
        self.mode = Mode.BOTH
```

```
        self.input_dim = input_dim
        self.net = _LSTMNet(input_dim, hidden_dim)
        self.loss_fn = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=1e-3)

    def fit_sequences(self, sequences: Any, targets: np.ndarray, epochs: int = 10) -> None:
        """
        sequences: list/array of shape (n_samples, seq_len, input_dim)
        targets: class labels (0=home,1=draw,2=away)
        """
        X = torch.tensor(np.array(sequences), dtype=torch.float32)
        y = torch.tensor(targets, dtype=torch.long)
        dataset = torch.utils.data.TensorDataset(X, y)
        loader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=True)

        self.net.train()
        for _ in range(epochs):
            for xb, yb in loader:
                self.optimizer.zero_grad()
                logits = self.net(xb)
                loss = self.loss_fn(logits, yb)
                loss.backward()
                self.optimizer.step()

    def predict_sequence(self, sequence: Any) -> np.ndarray:
        self.net.eval()
```

```
    x = torch.tensor(np.array(sequence)[None, ...], dtype=torch.float32)
    with torch.no_grad():
        logits = self.net(x)
        probs = torch.softmax(logits, dim=1).cpu().numpy()[0]
    return probs
```

MLP and CNN can be implemented with either scikit-learn MLP or PyTorch, but they still implement either SupervisedModel or SequenceModel.

Transformer-based models would have a similar pattern, just with a different network architecture.

---

5. Training orchestrator (works for any model)

We refactor training so it can handle *any* component type, based on its task_type and mode.

app/services/training.py

```
import os
import numpy as np
from typing import List, Any
from sqlalchemy.orm import Session
from app.core.registry import get_component_class
from app.core.base import (
    SupervisedModel,
    ScorelineModel,
    TotalsModel,
    SurvivalModel,
    RatingModel,
    SequenceModel,
    PreprocessorModel,
)
```

```
from app.features import prematch, shot_level, inplay, sequences
from app.config import MODEL_STORE_DIR, DEFAULT_LEAGUE_ID

os.makedirs(MODEL_STORE_DIR, exist_ok=True)

class TrainingService:
    def __init__(self, session: Session, league_id: int = DEFAULT_LEAGUE_ID):
        self.session = session
        self.league_id = league_id

    # --- Main entry point ---

    def train_component(self, component_name: str) -> None:
        cls = get_component_class(component_name)
        comp = cls()  # type: ignore
        print(f"Training {component_name} for league {self.league_id}")

        if isinstance(comp, SupervisedModel):
            X, y = prematch.build_outcome_dataset(self.session, self.league_id)
            comp.fit(X, y)

        elif isinstance(comp, ScorelineModel):
            matches = prematch.get_match_objects(self.session, self.league_id)
            comp.fit(matches)

        elif isinstance(comp, TotalsModel):
```

```
            X, y = prematch.build_totals_dataset(self.session, self.league_id, target="goals")
            comp.fit(X, y)

        elif isinstance(comp, SurvivalModel):
            X, durations, events = inplay.build_survival_dataset(self.session, self.league_id)
            comp.fit(X, durations, events)

        elif isinstance(comp, RatingModel):
            matches = prematch.get_match_objects(self.session, self.league_id)
            comp.update_from_matches(matches, self.session)

        elif isinstance(comp, SequenceModel):
            seqs, targets = sequences.build_outcome_sequences(self.session, self.league_id)
            comp.fit_sequences(seqs, targets)

        elif isinstance(comp, PreprocessorModel):
            X = prematch.build_feature_matrix(self.session, self.league_id)
            comp.fit_transform(X)

        else:
            raise NotImplementedError(f"No training path for {component_name}")

        out_path = os.path.join(MODEL_STORE_DIR,
f"{component_name}_L{self.league_id}.joblib")
        comp.save(out_path)
        print(f"Saved {component_name} to {out_path}")
```

You'd implement the helper functions in app/features/prematch.py, inplay.py, etc., to return appropriate X,y / sequences / durations.

train_all_models.py can just loop over a list of component names:

```
# scripts/train_all_models.py

from app.db.session import SessionLocal

from app.services.training import TrainingService

from app.core.registry import list_components


if __name__ == "__main__":

    session = SessionLocal()

    svc = TrainingService(session)

    for name in list_components().keys():

        # You might filter which ones to train for now:

        if name.endswith("OutcomeModel") or name.endswith("GoalModel"):

            svc.train_component(name)

    session.close()
```

---

6. Prediction service + FastAPI endpoints (unchanged on the surface)

From the API's point of view, the user still wants:

- /predictions/prematch – W/D/L, scoreline, totals, etc.
- /predictions/inplay – live W/D/L, next goal, etc.

The prediction service just has to know *which* components to load and which feature builder to call.

app/services/prediction.py (simplified for OUTCOME & SCORELINE only):

```
import os

import numpy as np

from typing import Dict, Any, List
```

```
from sqlalchemy.orm import Session
from app.core.registry import get_component_class
from app.core.base import SupervisedModel, ScorelineModel
from app.features import prematch, inplay
from app.config import MODEL_STORE_DIR, DEFAULT_LEAGUE_ID


class PredictionService:
    def __init__(self, session: Session, league_id: int = DEFAULT_LEAGUE_ID):
        self.session = session
        self.league_id = league_id


    def _load(self, name: str):
        cls = get_component_class(name)
        path = os.path.join(MODEL_STORE_DIR, f"{name}_L{self.league_id}.joblib")
        return cls.load(path)


    def prematch_outcome(self, match_id: int, model_names: List[str]) -> Dict[str, Any]:
        X_match = prematch.build_single_match_features(self.session, match_id, self.league_id)
        X = X_match.reshape(1, -1)
        result: Dict[str, Any] = {}
        for name in model_names:
            model = self._load(name)
            if isinstance(model, SupervisedModel):
                probs = model.predict_proba(X)[0]
                result[name] = {
                    "home_win": float(probs[0]),
```

```python
            "draw": float(probs[1]),
            "away_win": float(probs[2]),
        }
    elif isinstance(model, ScorelineModel):
        state = prematch.build_match_state(self.session, match_id, self.league_id)
        matrix = model.predict_score_matrix(state)
        odds = model.outcome_probs_from_matrix(matrix)
        result[name] = odds
    else:
        # could add support for SimulationModel or others
        continue
return result

def inplay_outcome(self, live_event_json: dict, model_names: List[str]) -> Dict[str, Any]:
    snap = inplay.build_snapshot_from_event(live_event_json)
    X = snap.X.reshape(1, -1)
    result: Dict[str, Any] = {"minute": snap.minute}
    for name in model_names:
        model = self._load(name)
        if isinstance(model, SupervisedModel):
            probs = model.predict_proba(X)[0]
            result[name] = {
                "home_win": float(probs[0]),
                "draw": float(probs[1]),
                "away_win": float(probs[2]),
            }
```

```
    return result
```

FastAPI endpoints from the previous answer remain very similar; they only call PredictionService with a list of component names.

---

7. How this refactor supports *all 36 models*

- Every row in your table maps to a **concrete class** inheriting from one of the base types.
- Training is **unified** via TrainingService.train_component, which chooses the right data based on the component type.
- Prediction is **unified** via PredictionService, which loads components and calls the correct method.
- Unsupervised models (KMeans, PCA, GMM, DBSCAN) are used as **preprocessors** or for **analysis endpoints** (e.g. “cluster my teams” in /analysis routes).
- Sequence models (LSTM, Transformer) have their own SequenceModel path, trained on sequences of matches or in-play events built by features/sequences.py.
- Rating systems (Elo, Glicko, TrueSkill, Bradley–Terry) are all RatingModels but can also feed into feature builders (as rating features) and baseline outcome predictions.
- Monte Carlo simulator wraps *any* ScorelineModel (Poisson, DC, Bivariate, Copula, Zero-inflated) and provides richer simulation-based forecasts for both pre-match and in-play.