Executive Matching Ranker - Full Code Explanation

This script builds a **Learning-to-Rank model** to match executives to business opportunities using historical match data, executive attributes, and opportunity features. It leverages a **LightGBM Ranker** model trained with domain-specific features engineered from JSON-encoded and string-based metadata. Below is a detailed walkthrough of each section of the code.

---

1. Data Loading and Cleaning

We load three CSVs:

- `exec_roles.csv` : Contains executive-level metadata. Each row has an `exec_entity_id` , a `type` , and a value in either `json_value` or `string_value` .
- `match.csv` : Historical match labels for which executives (`exec_entity_id`) were matched to which assignments (`assignment_id`), along with a binary `outcome` (1 = match, 0 = no match).
- `opp.csv` : Metadata about each opportunity/assignment, including `assignment_id` , `industry` , `sectors` , `sub_sectors` , `scale` , and `country` .

All `assignment_id` and `exec_entity_id` columns are cast to `Int64` to ensure consistent merging and avoid mismatches due to NaNs or mixed types.

---

2. Feature Engineering

a. Exec Role Pivoting

We transform the `exec_roles` file from long to wide format using `.pivot_table()` . This creates one row per `exec_entity_id` and spreads the `type` column into multiple columns (e.g. `json_value_sectors` , `string_value_hq_address` , etc.). The `aggfunc="first"` ensures that for multiple rows with the same `(exec_entity_id, type)` , we take the first one.

b. Feature Matching Functions

The core function `build_features()` creates:

- **Exact match flags**:
  - `sector_match` : Whether the exec’s sectors match the opportunity’s.
  - `country_match` : Whether the exec’s HQ matches the opportunity's country.
  - `scale_match` : Whether the exec's business scale matches.
- **Jaccard similarity scores**:
  - `sector_jaccard` , `sub_sector_jaccard` , `industry_jaccard` : Compare exec and opportunity lists (e.g., sectors or industries) using the **Jaccard similarity**:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This quantifies overlap in sets (e.g. shared industry tags).

All comparisons use `ast.literal_eval` to parse JSON-like strings into Python lists.

---

3. Data Preparation for Learning to Rank

We filter down to:

- Only rows with no missing values in critical features and the outcome column.
- Only `assignment_id` s that have at least one positive match (`outcome == 1`) — otherwise the ranker can't learn.

We split the data into **training** and **testing** by randomly selecting ~10% of `assignment_id` s to be held out. We ensure no leakage by grouping the split by `assignment_id` — this is crucial since ranking is context-dependent per group.

The LightGBM ranker expects:

- `X_train` : Feature matrix.
- `y_train` : Binary outcomes per row.
- `group_train` : Number of rows per `assignment_id` — i.e., how many executives were evaluated for each opportunity.

---

4. LightGBM Ranker Training

We use `LGBMRanker` , a model designed for **Learning to Rank** (LTR) tasks using the **LambdaMART** algorithm. It optimizes ranking loss functions (e.g., NDCG) by learning which items (execs) should be ranked higher for each group (assignment).

We pass:

- `n_estimators=100` : Number of boosting rounds.
- `force_col_wise=True` : Forces column-wise tree building to avoid auto-detection overhead.
- `verbosity=-1` : Suppresses LightGBM's warnings.

---

5. Evaluation with Precision@5

We calculate **Precision@5**: for each `assignment_id` , we sort executives by predicted score and take the top 5. If *any* of them was actually a correct match (`outcome==1`), that counts as a hit. The mean hit rate over all groups gives Precision@5:

- High Precision@5 = the model often includes the correct exec in the top-5 predictions.

---

6. Prediction for a New Opportunity

The function `rank_execs_for_new_opp()` lets us simulate ranking executives for a new opportunity:

- For a given opportunity row, it pairs the opportunity with every exec.
- It constructs the same feature columns as used during training.
- It predicts a score for each exec-opportunity pair and returns the **top 10 ranked execs**.

This simulates the **real-world use case** of recommending executives for a live opportunity.

Key Concepts Explained

Concept	Explanation
`LGBMRanker`	A LightGBM model specialised for ranking tasks, optimised using pairwise ranking losses like LambdaRank.
Jaccard Similarity	Measures similarity between two sets. Important when comparing tag-like data (e.g., sectors or industries).
Group-based Ranking	Each opportunity is treated as a group. The model must learn to rank the correct execs highest within each group.
Precision@5	A ranking metric that checks if the correct result appears in the top 5 predictions per group.

Why This Approach?

This ranking framework is well-suited for this type of matching problem because:

- Traditional classifiers ignore the **contextual nature of matching** (which exec is best *for a specific assignment*).
- Learning-to-Rank frameworks like LightGBM Ranker consider *relative ordering* of candidates within each assignment.
- Feature engineering with **exact matches and fuzzy Jaccard scores** gives the model meaningful signals.
- It's scalable, explainable, and can easily be extended with embeddings or domain-specific metadata.

In [4]:

```
import pandas as pd
import numpy as np
import ast
from sklearn.model_selection import train_test_split
from lightgbm import LGBMRanker
from sklearn.metrics import precision_score

# ------------------------
# 1. Load and clean data
# ------------------------

# Load input CSVs
exec_roles = pd.read_csv("exec_roles.csv")  # Executive attributes
match = pd.read_csv("match.csv")            # Matches between execs and opportuniti
opp = pd.read_csv("opp.csv")                # Opportunities

# Ensure key IDs are properly typed
```

```
match["assignment_id"] = match["assignment_id"].astype("Int64")
match["exec_entity_id"] = match["exec_entity_id"].astype("Int64")
opp["assignment_id"] = opp["assignment_id"].astype("Int64")

# Pivot exec_roles to a wide format: one row per exec with structured features
exec_roles_wide = exec_roles.pivot_table(
    index="exec_entity_id",
    columns="type",
    values=["json_value", "string_value"],
    aggfunc="first"
)
exec_roles_wide.columns = [f"{a}_{b}" for a, b in exec_roles_wide.columns]
exec_roles_wide = exec_roles_wide.reset_index()
exec_roles_wide = exec_roles_wide.dropna(subset=["exec_entity_id"])
exec_roles_wide["exec_entity_id"] = exec_roles_wide["exec_entity_id"].astype("Int64

# Merge matches with opportunities and exec features
match_opp = pd.merge(match, opp, on="assignment_id", how="left")
data = pd.merge(match_opp, exec_roles_wide, on="exec_entity_id", how="left")


# ------------------------
# 2. Feature Engineering
# ------------------------

def jaccard_sim(list1, list2):
    """
    Compute Jaccard similarity between two stringified lists.
    Used for comparing sectors, sub-sectors, and industries.
    """
    if pd.isna(list1) or pd.isna(list2):
        return 0
    try:
        set1 = set(ast.literal_eval(list1))
        set2 = set(ast.literal_eval(list2))
        return len(set1 & set2) / len(set1 | set2) if (set1 | set2) else 0
    except:
        return 0

def build_features(df):
    """
    Generate binary and similarity-based features for each exec-opportunity pair.
    """
    df["sector_match"] = (df["json_value_sectors"] == df["sectors"]).astype(int)
    df["country_match"] = (df["string_value_hq_address"] == df["country"]).astype(i
    df["scale_match"] = (df["string_value_scale"] == df["scale"]).astype(int)
    df["sector_jaccard"] = df.apply(lambda r: jaccard_sim(r.get("json_value_sectors
    df["sub_sector_jaccard"] = df.apply(lambda r: jaccard_sim(r.get("json_value_sub
    df["industry_jaccard"] = df.apply(lambda r: jaccard_sim(r.get("json_value_indus
    return df

# Build feature columns
data = build_features(data)


# ------------------------
# 3. Prepare for ranking
# ------------------------

# Define features to use for ranking
features = [
    "sector_match", "country_match", "scale_match",
    "sector_jaccard", "sub_sector_jaccard", "industry_jaccard"
]
```

```
# Drop rows with missing critical fields
data = data.dropna(subset=features + ["outcome", "assignment_id"])
data["outcome"] = data["outcome"].astype(int)

# Keep only assignment_ids with at least one successful match
valid_assignments = data.groupby("assignment_id")["outcome"].sum()
valid_assignments = valid_assignments[valid_assignments > 0].index.tolist()
data = data[data["assignment_id"].isin(valid_assignments)]

# Create holdout test set from a sample of assignment_ids
assignment_ids = data["assignment_id"].unique()
test_ids = np.random.choice(assignment_ids, size=max(3, int(0.1 * len(assignment_ic
train_ids = [aid for aid in assignment_ids if aid not in test_ids]

# Split the data
train_df = data[data["assignment_id"].isin(train_ids)].copy()
test_df = data[data["assignment_id"].isin(test_ids)].copy()

# Prepare data for LightGBM ranker
X_train = train_df[features]
y_train = train_df["outcome"]
group_train = train_df.groupby("assignment_id").size().values  # Number of items pe

X_test = test_df[features]
y_test = test_df["outcome"]
group_test = test_df.groupby("assignment_id").size().values


# ------------------------
# 4. Train LightGBM Ranker
# ------------------------

ranker = LGBMRanker(
    n_estimators=100,
    random_state=42,
    verbosity=-1,  # Suppresses most warnings
    force_col_wise=True  # Removes overhead message
)

ranker.fit(X_train, y_train, group=group_train)


# ------------------------
# 5. Evaluate Precision@5
# ------------------------

# Predict relevance scores for test set
test_df["score"] = ranker.predict(X_test)

# Get top 5 predictions per assignment_id
top_k = (
    test_df.groupby("assignment_id", group_keys=False)
    .apply(lambda g: g.sort_values("score", ascending=False).head(5))
    .reset_index(drop=True)
)

# Calculate average precision@5: proportion of groups where at least 1 top-5 item i
precision_at_5 = top_k.groupby("assignment_id")["outcome"].max().mean()
print(f"\nPrecision@5: {precision_at_5:.3f}")


# ------------------------
# 6. Predict top execs for new opportunity
```

```
# ------------------------

def rank_execs_for_new_opp(new_opp_row, exec_table, model, features):
    """
    Generate a ranked list of top executives for a given opportunity.

    Parameters:
        new_opp_row (Series): A single opportunity row.
        exec_table (DataFrame): Wide-format executive features.
        model: Trained ranking model.
        features (list): List of feature column names.

    Returns:
        DataFrame: Top 10 exec_entity_ids with predicted scores.
    """
    rows = []

    # For each exec, combine with new opportunity fields
    for _, exec_row in exec_table.iterrows():
        combined = new_opp_row.copy()
        for col in exec_row.index:
            combined[f"exec_{col}"] = exec_row[col]

        # Manually add needed fields
        combined["json_value_sectors"] = exec_row.get("json_value_sectors")
        combined["json_value_sub_sectors"] = exec_row.get("json_value_sub_sectors")
        combined["json_value_industry"] = exec_row.get("json_value_industry")
        combined["string_value_hq_address"] = exec_row.get("string_value_hq_address
        combined["string_value_scale"] = exec_row.get("string_value_scale")
        combined["exec_entity_id"] = exec_row.get("exec_entity_id")
        rows.append(combined)

    # Build feature matrix and predict scores
    pred_df = pd.DataFrame(rows)
    pred_df = build_features(pred_df)
    pred_df["score"] = model.predict(pred_df[features])

    return pred_df[["exec_entity_id", "score"]].sort_values("score", ascending=Fals
```

```
Precision@5: 0.333
```

```
C:\Users\DanielGodden\AppData\Local\Temp\ipykernel_14380\4278955408.py:135: Future
Warning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is
deprecated, and in a future version of pandas the grouping columns will be exclude
d from the operation. Either pass `include_groups=False` to exclude the groupings
or explicitly select the grouping columns after groupby to silence this warning.
  .apply(lambda g: g.sort_values("score", ascending=False).head(5))
```

Opportunities to Improve the Model

Better Models

- **CatBoostRanker**: Handles categorical features natively; often better on tabular data with high-cardinality strings.
- **XGBoostRanker**: Another strong baseline; can handle sparse features better in some cases.
- **Neural Ranking Models**: E.g., DSSM, BERT-based cross encoders, especially if exec bios or long descriptions are available.
- **Transformer Encoders**: Use textual similarity from `sentence-transformers` or `TF-IDF` embeddings for richer semantic similarity.

Enhanced Features

- **Embeddings**:
  - Convert `sector`, `industry`, and `country` fields to dense vectors.
  - Use cosine similarity instead of Jaccard.
- **Temporal features**:
  - Time since last successful match for execs.
  - Recency of opportunity posting.
- **Exec activity rate**:
  - How many times the exec has been selected.
  - Success ratio per sector/industry.
- **Interaction features**:
  - Historical co-occurrence of sector + country + scale for success.

Data Enrichment

- Incorporate **textual bios**, **past job history**, or **skills tags** for execs.
- Add **company descriptions** or **financial metrics** for opportunities.
- Resolve mismatches in naming conventions (e.g., sector tags) with standardised vocab or NLP similarity.

Model Performance & Data Issues

- If you're seeing `best gain: -inf`, it means:
  - Features may not have enough variance to create informative splits.
  - Consider binning or embedding categorical values for better signal.
- Small sample size or unbalanced `outcome` labels can reduce model learning.

Productionising the Model (AWS + PostgreSQL stack)

Let’s assume:

- AWS is used for compute, deployment, monitoring
- PostgreSQL stores execs, opportunities, and predictions

System Architecture

```text +-----------------+ +----------------+ +--------------------+ | PostgreSQL DB | <---> | ETL/Feature Job| ---> | Model Training | | (execs, opps) | | (Lambda/Airflow)| | (SageMaker/EC2) | +-----------------+ +----------------+ +--------------------+ | | | v --------------+ +---------------------+ | Model API | <-- | Model Registry | | (FastAPI ECS)| | (MLflow/S3/DVC) | +--------------+ +---------------------+

# Steps to Productionise
```

1. ETL Pipeline

- Extract new execs, opportunities, and outcomes from **PostgreSQL** or **S3**
- Compute features via **Airflow** or **Lambda** job
- Save processed data to **S3** or a staging table

---

2. Model Training

- Run **nightly/weekly batch jobs** on **EC2** or **SageMaker**
- Use **MLflow** for model versioning and hyperparameter tracking

---

3. Model API

- Deploy via **FastAPI** app on **AWS ECS** or **Lambda**
- Expose `/predict` endpoint taking `assignment_id` and returning **top 10 `exec_entity_id` s**

---

4. Predictions Table

- Store `assignment_id` , `exec_entity_id` , `score` , `prediction_time` into **PostgreSQL**
- Use for validation or feedback loop later

---

5. Monitoring & Retraining

- Use **CloudWatch** + custom metrics to track:
  - Prediction volume
  - Latency
  - Precision@5 drift
- Auto-trigger retraining when:
  - New batch of labelled data arrives
  - **Precision@5** degrades over time

---

6. Model Retraining Workflow

- Re-train on new labels **weekly or monthly**
- Compare new vs old model performance
- If better, **promote to production** (via MLflow or internal registry)

Evaluation Metrics To Track

Metric	Purpose
Precision@5	Measures top-k match success

Metric	Purpose
MRR	Mean Reciprocal Rank – position of first relevant match
NDCG	Normalised Discounted Cumulative Gain – ranks correct results higher
Recall@K	Measures recall within top-k predictions

Use these metrics **over rolling windows (e.g. last 7 days)** to track live model performance.

Final Thoughts

This ranking system is:

- Simple to implement
- Scalable with minimal infrastructure
- Already delivers business insight via **Precision@5**

With improvements such as:

- More data
- Embedding-based similarity
- Regular retraining
- Full MLOps integration

...it can evolve into a production-grade, self-improving exec recommender system.