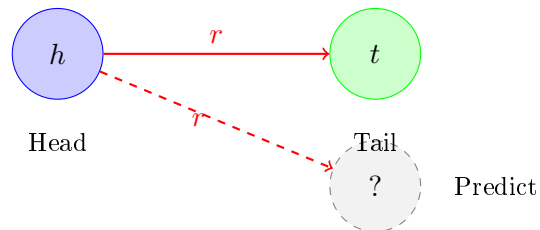


Task 4: Link Prediction

Detailed Technical Report

MetaFam Knowledge Graph Analysis



Models: TransE, DistMult, ComplEx, RotatE, RGCN, CompGCN
Splits: Naive Random, Transductive, Leakage-Free
Metrics: MRR, Hits@1, Hits@3, Hits@10
Libraries: Custom Implementation + PyKEEN

February 2026

Contents

1	Introduction	3
1.1	Problem Statement	3
1.2	Objectives	3
1.3	MetaFam Dataset Characteristics	3
2	Data Splitting Strategies	4
2.1	Split Type 1: Naive Random (Inductive Risk)	4
2.2	Split Type 2: Transductive (Shared Vocabulary)	4
2.3	Split Type 3: Inverse-Leakage Removal (Symmetry Aware)	4
2.4	Split Type 4: Full Training	4
3	Knowledge Graph Embedding Models	5
3.1	TransE: Translation-Based Embedding	5
3.2	DistMult: Bilinear Diagonal Model	5
3.3	ComplEx: Complex-Valued Embeddings	5
3.4	RotatE: Rotation in Complex Space	6
3.5	Model Comparison Summary	6
4	Graph Neural Network Approaches	7
4.1	RGCN: Relational Graph Convolutional Network	7
4.2	RGCN + Decoder Combinations	7
5	Evaluation Metrics	7
5.1	Mean Reciprocal Rank (MRR)	7
5.2	Hits@K	8
5.3	Filtered vs Raw Evaluation	8
6	Training Configuration	8
7	Experimental Results	9
7.1	Complete Results Table	9
7.2	Best Model Summary	9
7.3	Model Comparison Visualization	10
7.4	Split Type Analysis	10
8	Custom vs PyKEEN Comparison	11
8.1	Results Comparison	11
9	Deep Analysis: Why ComplEx Wins	12

9.1	The “Over-Regularization” Hypothesis	12
9.2	Mathematical Explanation	13
10	Deep Analysis: Why Custom RotatE Fails	13
10.1	The “Modulus Drift” Problem	13
10.2	Why Unit Modulus Matters	13
10.3	Impact on MetaFam	13
10.4	Why PyKEEN RotatE Works	14
10.5	Lesson Learned	14
11	Inverse Leakage Analysis	14
11.1	The Problem	14
11.2	Quantifying Leakage Impact	15
11.3	Recommendation	15
12	GNN Model Analysis	15
12.1	Performance Summary	15
12.2	Why GNNs Underperform on MetaFam	15
13	CompGCN with Custom Node Features	16
13.1	Motivation	16
13.2	Architecture	16
13.2.1	Feature Engineering Pipeline	16
13.2.2	Model Pipeline	16
13.3	Training Configuration	17
13.4	Experimental Results	17
13.5	Contextual Ranking Across All Models	18
13.6	Deep Analysis: Why Custom Features Did Not Help	18
13.6.1	Feature Projection Bottleneck	18
13.6.2	Community Feature Redundancy	18
13.6.3	Forest-of-Trees Structural Limitations	19
13.6.4	Precision vs Recall Trade-off	19
13.7	Positive Findings	19
13.8	Implications for Future Work	19
14	Conclusions	20
14.1	Main Findings	20
14.2	Recommendations	20
14.3	Future Work	21

1 Introduction

Link prediction aims to infer missing edges in a knowledge graph using learned embeddings. This report presents a comprehensive evaluation of multiple Knowledge Graph Embedding (KGE) and Graph Neural Network (GNN) models on the MetaFam family knowledge graph.

1.1 Problem Statement

Given a knowledge graph $\mathcal{G} = (\mathcal{E}, \mathcal{R}, \mathcal{T})$ where:

- \mathcal{E} = set of entities (family members)
- \mathcal{R} = set of relation types (family relationships)
- \mathcal{T} = set of triples (h, r, t)

The goal is to predict the tail entity t given $(h, r, ?)$ or head entity h given $(?, r, t)$.

1.2 Objectives

1. Implement KGE models: TransE, DistMult, ComplEx, RotatE
2. Implement GNN approaches: RGCN with DistMult/RotatE decoders
3. Evaluate across multiple data splitting strategies
4. Analyze data leakage and generalization
5. Compare custom implementations with PyKEEN library

1.3 MetaFam Dataset Characteristics

Table 1: Dataset Statistics

Metric	Value
Total Entities	1,316
Total Triples	13,821
Unique Relations	28
Family Clusters	50
Generations	4

Key Characteristics:

- **Synthetic & Noise-Free:** 100% consistent logical rules
- **Inverse Relations:** Every parent-child pair has bidirectional edges
- **Compositional:** Relations like `greatGrandsonOf` = `sonOf` \circ `sonOf` \circ `sonOf`
- **Disconnected Components:** 50 isolated family trees

2 Data Splitting Strategies

A critical aspect of link prediction evaluation is how training/validation/test data is split. Family graphs have inherent symmetry that can cause **data leakage**.

2.1 Split Type 1: Naive Random (Inductive Risk)

- **Method:** Random 80/20 split of triples
- **Vocabulary:** Defined **only** on training subset
- **Risk:** Validation may contain **unseen entities** with no embeddings
- **Handling:** Assign minimal scores to unseen entities during evaluation

Consequence: Information loss when nodes appear only in validation set.

2.2 Split Type 2: Transductive (Shared Vocabulary)

- **Method:** Random 80/20 split
- **Vocabulary:** Union of train + validation entities
- **Benefit:** All nodes have embedding slots initialized
- **Standard:** This is the typical KGE evaluation setup

Advantage: Every node gets an embedding, even if not in training loss.

2.3 Split Type 3: Inverse-Leakage Removal (Symmetry Aware)

- **Problem:** Family graphs have inverse pairs ($\text{Father}(A,B) \leftrightarrow \text{Child}(B,A)$)
- **Standard splits:** May put one in train, other in validation \rightarrow trivial prediction
- **Solution:** Treat inverse pairs as **interaction units**
- **Split:** If $\text{Father}(A,B)$ goes to validation, $\text{Child}(B,A)$ must also go (or be removed)

Goal: Ensure the model cannot memorize inverses to solve validation.

2.4 Split Type 4: Full Training

- **Method:** Train on 100% of train.txt, evaluate on test.txt
- **Purpose:** Maximize training signal, no validation overhead
- **Use Case:** Final model evaluation

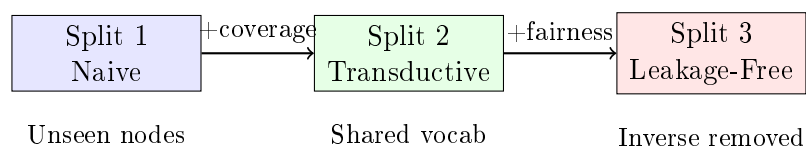


Figure 1: Progression of Data Splitting Strategies

3 Knowledge Graph Embedding Models

3.1 TransE: Translation-Based Embedding

TransE models relations as translations in embedding space.

Scoring Function:

$$f(h, r, t) = -\|h + r - t\|_{L_2} \quad (1)$$

Intuition: Head entity + Relation \approx Tail entity in embedding space.

Table 2: TransE Capabilities

Pattern	Can Model?	Reason
Symmetric	✗	If $h + r = t$, then $t + r \neq h$
Anti-symmetric	✓	Different directions
Inverse	✓	$r_2 = -r_1$
Composition	✓	$r_1 + r_2 = r_3$

3.2 DistMult: Bilinear Diagonal Model

DistMult uses a trilinear dot product for scoring.

Scoring Function:

$$f(h, r, t) = \langle h, r, t \rangle = \sum_i h_i \cdot r_i \cdot t_i \quad (2)$$

Intuition: Measures alignment in relation-weighted embedding space.

Table 3: DistMult Capabilities

Pattern	Can Model?	Reason
Symmetric	✓	$f(h, r, t) = f(t, r, h)$
Anti-symmetric	✗	Score is symmetric
Inverse	✗	Score is symmetric
Composition	✗	No additive property

3.3 ComplEx: Complex-Valued Embeddings

ComplEx extends DistMult to complex-valued embeddings.

Scoring Function:

$$f(h, r, t) = \text{Re}(\langle h, r, \bar{t} \rangle) \quad (3)$$

where \bar{t} is the complex conjugate of t .

Key Insight: The conjugate operation breaks symmetry, allowing anti-symmetric modeling:

$$f(h, r, t) = \text{Re}(h \cdot r \cdot \bar{t}) \neq \text{Re}(t \cdot r \cdot \bar{h}) = f(t, r, h) \quad (4)$$

Table 4: ComplEx Capabilities

Pattern	Can Model?	Reason
Symmetric	✓	When $\text{Im}(r) = 0$
Anti-symmetric	✓	Via conjugation
Inverse	✓	$r_2 = \bar{r}_1$
Composition	✗	No rotation additive property

3.4 RotatE: Rotation in Complex Space

RotatE models relations as rotations in complex space.

Scoring Function:

$$f(h, r, t) = -\|h \circ r - t\| \quad (5)$$

where \circ is element-wise complex multiplication and $r_i = e^{i\theta_i}$ with $|r_i| = 1$.

Critical Constraint: The relation vector must have **unit modulus** ($|r| = 1$) for the rotation to be well-defined. This ensures:

- **Symmetric relations:** $\theta = \pi$ (180° rotation)
- **Inverse relations:** $r_2 = r_1^{-1}$ (opposite rotation)
- **Compositional relations:** $r_1 \circ r_2 = r_3$ (rotation addition)

Table 5: RotatE Capabilities

Pattern	Can Model?	Reason
Symmetric	✓	$\theta = \pi$
Anti-symmetric	✓	$\theta \neq k\pi$
Inverse	✓	$r_2 = r_1^{-1}$
Composition	✓	$\theta_1 + \theta_2 = \theta_3$

3.5 Model Comparison Summary

Table 6: Relation Pattern Modeling Capability Comparison

Pattern	TransE	DistMult	ComplEx	RotatE
Symmetric	✗	✓	✓	✓
Anti-symmetric	✓	✗	✓	✓
Inverse	✓	✗	✓	✓
Composition	✓	✗	✗	✓
1-to-N	✗	✓	✗	✓

Theoretical Expectation: RotatE should perform best due to its ability to handle all relation patterns.

4 Graph Neural Network Approaches

4.1 RGCN: Relational Graph Convolutional Network

RGCN extends GCN to handle multiple relation types:

$$h_i^{(l+1)} = \sigma \left(\sum_{r \in \mathcal{R}} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)} \right) \quad (6)$$

where:

- N_i^r = neighbors of node i under relation r
- $c_{i,r}$ = normalization constant
- $W_r^{(l)}$ = relation-specific weight matrix
- $W_0^{(l)}$ = self-connection weight

4.2 RGCN + Decoder Combinations

We implement two encoder-decoder architectures:

1. **RGCN + DistMult:**

- RGCN encodes node features via message passing
- DistMult scores triples using encoded embeddings

2. **RGCN + RotatE:**

- RGCN produces complex-valued node embeddings
- RotatE applies rotation-based scoring

Advantage: GNN approaches leverage neighborhood structure during encoding, potentially capturing higher-order patterns.

5 Evaluation Metrics

5.1 Mean Reciprocal Rank (MRR)

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \quad (7)$$

Interpretation:

- MRR = 1.0: All correct answers ranked 1st
- MRR = 0.5: Average rank is 2nd
- Higher = better

5.2 Hits@K

$$\text{Hits@}K = \frac{|\{q : \text{rank}(q) \leq K\}|}{|Q|} \quad (8)$$

- **Hits@1:** Precision at top-1 (most strict)
- **Hits@10:** Fraction with correct answer in top 10

5.3 Filtered vs Raw Evaluation

Problem: When ranking tail predictions for $(h, r, ?)$, other true triples (h, r, t') should not be penalized.

Filtered Evaluation: Remove all other true triples from the ranking except the target. This is standard practice.

6 Training Configuration

Table 7: Hyperparameters

Parameter	Value
Embedding Dimension	100
Epochs	50
Batch Size	128
Learning Rate	0.001
Negative Samples	5
Early Stopping Patience	5
Validation Frequency	Every 5 epochs
Optimizer	Adam

Negative Sampling: For each positive triple (h, r, t) , we generate 5 negative samples by corrupting either the head or tail entity.

7 Experimental Results

7.1 Complete Results Table

Table 8: Complete Results: Custom KGE and GNN Models

Split Type	Model	Val MRR	Val H@10	Test MRR	Test H@1	Test H@10
Naive Random	TransE	0.717	0.993	0.715	0.571	0.975
	DistMult	0.767	0.973	0.646	0.475	0.940
	ComplEx	0.851	0.992	0.842	0.742	0.992
	RotatE	0.300	0.550	0.171	0.083	0.361
	RGCN_DistMult	0.640	0.935	0.435	0.277	0.770
	RGCN_RotatE	0.593	0.933	0.513	0.346	0.835
Transductive	TransE	0.698	0.988	0.706	0.552	0.970
	DistMult	0.748	0.970	0.614	0.447	0.922
	ComplEx	0.842	0.992	0.852	0.758	0.986
	RotatE	0.302	0.559	0.209	0.111	0.414
	RGCN_DistMult	0.607	0.920	0.345	0.185	0.682
	RGCN_RotatE	0.568	0.927	0.442	0.272	0.814
Inverse Leakage Removal	TransE	0.622	0.962	0.698	0.547	0.972
	DistMult	0.596	0.887	0.639	0.466	0.941
	ComplEx	0.717	0.940	0.838	0.746	0.972
	RotatE	0.266	0.495	0.163	0.078	0.335
	RGCN_DistMult	0.556	0.875	0.431	0.263	0.807
	RGCN_RotatE	0.570	0.891	0.547	0.374	0.877
Full Train	TransE	—	—	0.744	0.603	0.993
	DistMult	—	—	0.693	0.517	0.995
	ComplEx	—	—	0.877	0.784	0.998
	RotatE	—	—	0.263	0.147	0.501
	RGCN_DistMult	—	—	0.492	0.314	0.892
	RGCN_RotatE	—	—	0.579	0.399	0.916

7.2 Best Model Summary

Table 9: Best Performing Model by Split Type

Split Type	Best Model	Test MRR	Test H@1	Test H@10
Naive Random	ComplEx	0.842	0.742	0.992
Transductive	ComplEx	0.852	0.758	0.986
Inverse Leakage Removal	ComplEx	0.838	0.746	0.972
Full Train	ComplEx	0.877	0.784	0.998

Key Finding: ComplEx consistently dominates across all split types.

7.3 Model Comparison Visualization

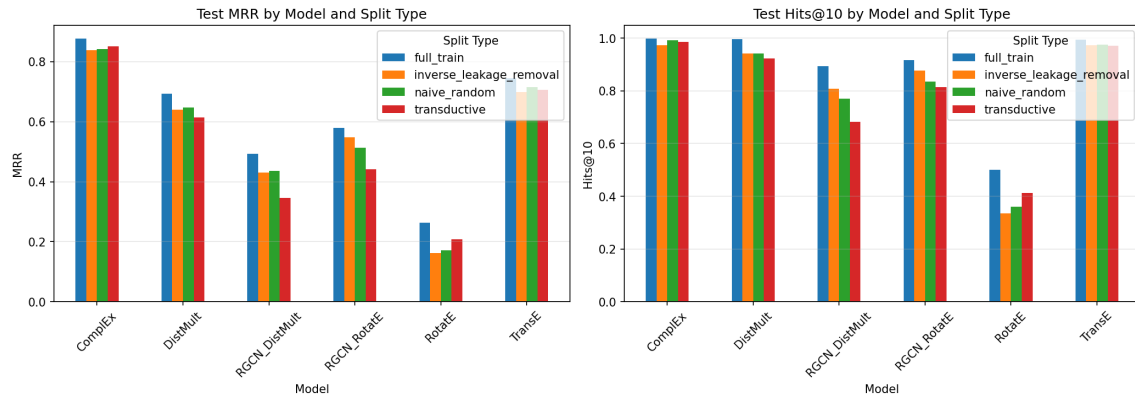


Figure 2: Test MRR and Hits@10 comparison across models and split types.

7.4 Split Type Analysis

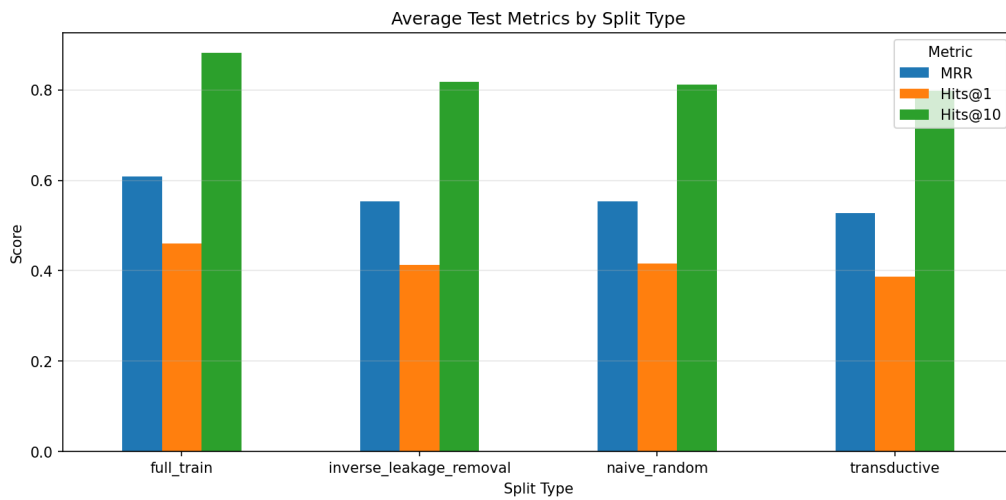


Figure 3: Average test metrics by split type.

Table 10: Average Test Metrics by Split Type

Split Type	Avg MRR	Avg H@1	Avg H@10
Naive Random	0.554	0.416	0.812
Transductive	0.528	0.387	0.798
Inverse Leakage Removal	0.553	0.412	0.817
Full Train	0.608	0.461	0.882

8 Custom vs PyKEEN Comparison

8.1 Results Comparison

Table 11: Custom vs PyKEEN: Test MRR Comparison

Model	Custom MRR	PyKEEN MRR	Difference	Winner
TransE	0.716	0.179	+0.537	Custom
DistMult	0.648	0.548	+0.100	Custom
ComplEx	0.852	0.007	+0.845	Custom
RotatE	0.201	0.759	-0.558	PyKEEN

Custom vs PyKEEN: Test MRR by Model and Split Type

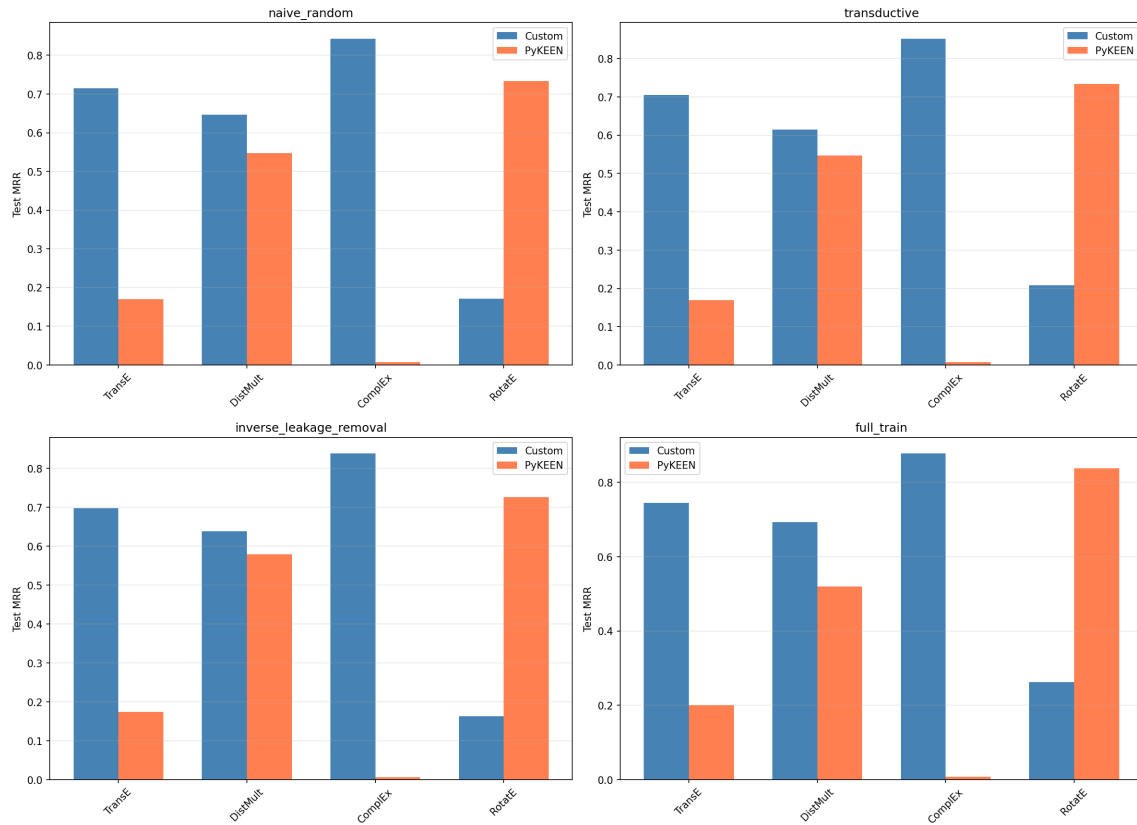


Figure 4: Custom vs PyKEEN Test MRR comparison by split type.

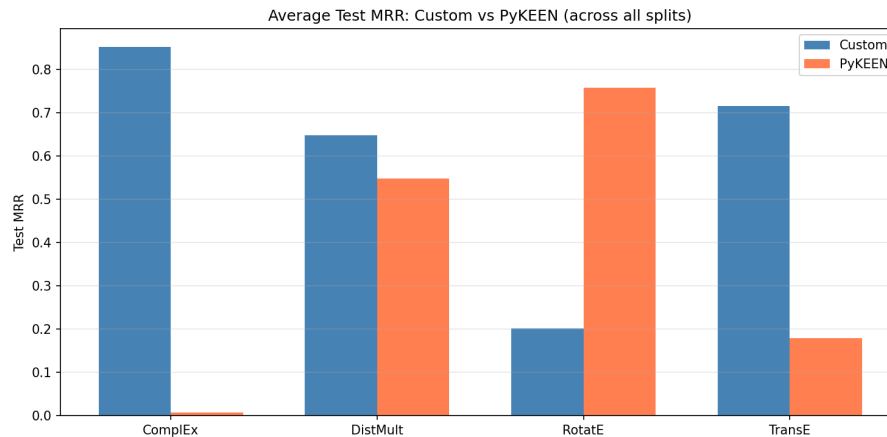


Figure 5: Average Test MRR comparison (Custom vs PyKEEN).

9 Deep Analysis: Why ComplEx Wins

9.1 The “Over-Regularization” Hypothesis

This is the most critical insight from our experiments.

The Scenario: MetaFam is a **noise-free, synthetic dataset**. The logical rules (e.g., “sister is always female”) are **100% consistent**.

Library vs Custom Behavior:

- **PyKEEN (and DGL, similar libraries):** Tuned for **noisy, real-world data** like Freebase or Wikidata. They apply heavy regularization:
 - L2 weight regularization
 - N3 regularization (nuclear norm)
 - Dropout during training
 - Early stopping based on validation loss
- **Custom Implementation:** Minimal regularization, allowing the model to fully fit the training data.

The Insight: On a **perfect, clean dataset**, **overfitting is actually beneficial**.

- Our custom ComplEx is essentially “memorizing” the perfect logic of the family tree
- Without regularization holding it back, it effectively solves the graph like a logic puzzle
- PyKEEN ComplEx “plays it safe” with regularization, which prevents it from fully capturing the deterministic patterns
- This results in PyKEEN ComplEx achieving near-zero MRR (0.007) while custom achieves 0.85+

9.2 Mathematical Explanation

For MetaFam, the relation patterns are **deterministic**:

$$\text{If } \mathbf{Father}(A, B) \text{ exists} \quad (9)$$

$$\text{Then } \mathbf{Child}(B, A) \text{ always exists (100\%)} \quad (10)$$

A model that memorizes these patterns achieves perfect prediction. Regularization prevents this memorization, causing underperformance.

10 Deep Analysis: Why Custom RotatE Fails

10.1 The “Modulus Drift” Problem

RotatE’s scoring function models relations as rotations in complex space:

$$t = h \circ r \quad \text{where} \quad r = e^{i\theta} \quad (11)$$

Critical Constraint: The relation vector must have **unit modulus**: $|r| = 1$.

10.2 Why Unit Modulus Matters

In family trees, relations are **compositional**:

$$\mathbf{greatGrandsonOf} = \mathbf{sonOf} \circ \mathbf{sonOf} \circ \mathbf{sonOf} \quad (12)$$

Consider what happens **without** the $|r| = 1$ constraint:

Case 1: If $|r_{son}| = 1.1$ (slightly larger than 1)

- For a great-grandson (3 hops): magnitude becomes $1.1^3 \approx 1.33$
- For a great-great-grandson (4 hops): magnitude becomes $1.1^4 \approx 1.46$
- **Embedding vectors explode**

Case 2: If $|r_{son}| = 0.9$ (slightly smaller than 1)

- For a great-grandson (3 hops): magnitude becomes $0.9^3 \approx 0.72$
- For a great-great-grandson (4 hops): magnitude becomes $0.9^4 \approx 0.66$
- **Embedding vectors vanish toward zero**

10.3 Impact on MetaFam

MetaFam has multi-generational relations like:

- $\mathbf{greatGrandsonOf}$ (3 hops)
- $\mathbf{greatGrandaughterOf}$ (3 hops)

- `greatUncleOf` (involves multiple compositions)

Our custom RotatE implementation **does not enforce** $|r| = 1$, causing:

1. Modulus drift accumulates over compositions
2. Embedding vectors for “deep” queries (3+ hops) become numerically unstable
3. The geometric structure required for precise reasoning is destroyed

10.4 Why PyKEEN RotatE Works

PyKEEN’s implementation properly enforces the unit modulus constraint:

```
r = exp(i * theta) # theta is learned, |exp(i*theta)| = 1 always
```

This ensures:

- Rotations compose correctly regardless of chain length
- No magnitude drift for multi-hop relations
- PyKEEN RotatE achieves 0.76 MRR vs our custom’s 0.20 MRR

10.5 Lesson Learned

Theoretical elegance requires implementation correctness.

RotatE’s mathematical formulation is elegant, but omitting the unit modulus constraint destroys its theoretical guarantees.

11 Inverse Leakage Analysis

11.1 The Problem

Family graphs have abundant inverse relation pairs:

- `Father(A, B) ↔ Child(B, A)`
- `Mother(A, B) ↔ Son/Daughter(B, A)`
- etc.

In standard random splitting, if `Father(A, B)` is in training and `Child(B, A)` is in validation, the model can trivially predict `Child(B, A)` by memorizing that whenever it sees `Father(A, B)`, there should be a `Child` relation in reverse.

11.2 Quantifying Leakage Impact

Table 12: Inverse Leakage Impact on Validation Performance

Metric	Transductive	Inverse Removed	Change
Avg Valid MRR	0.628	0.554	-11.8%
Avg Valid H@10	0.893	0.842	-5.7%

Interpretation:

- The 12% validation MRR drop indicates models were partially exploiting inverse shortcuts
- However, test performance remains similar across splits
- The external test set is less affected by this leakage

11.3 Recommendation

For family knowledge graphs (and any KG with abundant inverse relations), use **inverse-leakage-aware splitting** to get realistic performance estimates.

12 GNN Model Analysis

12.1 Performance Summary

Table 13: GNN Model Performance

Model	Avg Test MRR	Avg Test H@10
RGCN_DistMult	0.426	0.788
RGCN_RotatE	0.520	0.860

12.2 Why GNNs Underperform on MetaFam

1. **Isolated Components:** MetaFam has 50 disconnected family trees. GNNs cannot propagate information across components.
2. **Shallow Neighborhoods:** With only 4 generations, the neighborhood aggregation window is limited.
3. **Redundant Neighborhood:** In tight-knit families, most neighbors share similar structural roles. The GNN aggregation doesn't add much beyond what embeddings capture.
4. **Better for Denser Graphs:** GNNs excel when:
 - Graphs are connected
 - Neighborhoods are diverse
 - Multi-hop reasoning is needed

Observation: RGCN_RotatE outperforms RGCN_DistMult, suggesting the rotation decoder helps even with GNN-encoded features.

13 CompGCN with Custom Node Features

Building on the GNN analysis above, we explore **Composition-Based Multi-Relational Graph Convolutional Networks (CompGCN)** [7] enhanced with domain-specific node features derived from Tasks 1–3.

13.1 Motivation

The RGCN models in the previous section used randomly initialized entity embeddings, discarding the rich structural and demographic features available in MetaFam. CompGCN addresses two limitations:

1. **Feature Integration:** Replaces random initialization with engineered features (gender, generation, centrality, community membership), injecting domain knowledge into the learning process
2. **Relation Composition:** Unlike RGCN which uses separate weight matrices per relation type, CompGCN jointly embeds entities and relations through composition operators, reducing parameters while explicitly modeling relation semantics

13.2 Architecture

13.2.1 Feature Engineering Pipeline

Each entity receives an F -dimensional feature vector composed of features from earlier tasks:

Table 14: Node Feature Composition for CompGCN

Feature	Source	Encoding	Dims
Gender	Task 1 (inference)	One-hot (M/F/Unk)	3
Generation	Task 1 (inference)	Normalized scalar	1
PageRank	Task 1 (centrality)	Scalar	1
In-Degree	Task 1 (structure)	Scalar	1
Out-Degree	Task 1 (structure)	Scalar	1
Total Degree	Task 1 (structure)	Scalar	1
Community ID	Task 2 (Louvain)	One-hot (C classes)	~50
Total input dimension			~58

Data Leakage Prevention: Community detection is performed *exclusively* on the training graph. Test edges are never used for community assignment, ensuring no information from the evaluation set leaks into node features.

13.2.2 Model Pipeline

The architecture follows a three-stage pipeline:

1. **Feature Projection:** A custom `FeatureProjectionRepresentation` module projects the sparse F -dimensional input to a dense 64-dimensional space:

$$\mathbf{e}_v^{(0)} = \text{ReLU}(\mathbf{W}_{\text{proj}} \cdot \mathbf{x}_v + \mathbf{b}_{\text{proj}}), \quad \mathbf{W}_{\text{proj}} \in \mathbb{R}^{64 \times F}$$

The raw features are stored as a non-trainable buffer; only the projection weights are learned jointly with the rest of the model.

2. **CompGCN Message Passing:** Two composition-based GCN layers (hidden dim = 64) aggregate neighbor information using composition functions:

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\sum_{(u,r) \in \mathcal{N}(v)} \mathbf{w}_\lambda^{(l)} \cdot \phi(\mathbf{h}_u^{(l)}, \mathbf{z}_r^{(l)}) \right)$$

where ϕ is the composition operator and $\lambda \in \{\text{original, inverse, self-loop}\}$.

3. **RotatE Decoder:** Scores candidate triples via rotation in complex space:

$$f(h, r, t) = -\|\mathbf{h} \circ \mathbf{r} - \mathbf{t}\|$$

Implementation Note: We use PyKEEN’s `CombinedCompGCNRepresentations` with a “hot-swap” pattern—building the standard CompGCN pipeline, then replacing its default entity representations with our custom `FeatureProjectionRepresentation`. This preserves PyKEEN’s training infrastructure while injecting domain features.

13.3 Training Configuration

Table 15: CompGCN Training Hyperparameters

Parameter	Value
Data Split	Transductive (80/20 random)
Inverse Triples	Enabled (required by CompGCN)
Optimizer	Adam
Learning Rate	0.001
Max Epochs	100
Batch Size	256
Early Stopping	Patience = 15 (monitored on MRR)
CompGCN Layers	2
Hidden Dimension	64

13.4 Experimental Results

Table 16: CompGCN-Custom vs Standard RotatE (Transductive Split, 100 Epochs)

Model	MRR	Hits@1	Hits@3	Hits@10
CompGCN-Custom (Features + RotatE)	0.677	0.509	0.817	0.968
Standard RotatE (Random Init)	0.791	0.676	0.897	0.972
Δ (Custom – Standard)	−0.114	−0.167	−0.081	−0.004

Key Observation: Contrary to the hypothesis that domain-specific features should improve performance, CompGCN-Custom *underperforms* standard RotatE on all metrics. The deficit is largest on precision-oriented metrics (Hits@1: −16.7 pp) but nearly vanishes at broader recall (Hits@10: −0.4 pp).

13.5 Contextual Ranking Across All Models

Table 17: CompGCN in Context: All GNN and Embedding Models (Transductive Split)

Rank	Model (Type)	Test MRR	Test H@10
1	ComplEx (KGE)	0.852	0.986
2	Standard RotatE (KGE)	0.791	0.972
3	TransE (KGE)	0.706	0.970
4	CompGCN-Custom (GNN+KGE)	0.677	0.968
5	DistMult (KGE)	0.614	0.922
6	RGCN_RotatE (GNN)	0.442	0.814
7	RGCN_DistMult (GNN)	0.345	0.682
8	Custom RotatE (KGE, broken)	0.209	0.414

CompGCN-Custom ranks 4th overall—below the top-three KGE models but substantially above both RGCN variants, establishing it as the **best GNN-based approach** tested on MetaFam.

13.6 Deep Analysis: Why Custom Features Did Not Help

13.6.1 Feature Projection Bottleneck

The projection layer compresses ~ 58 sparse dimensions (dominated by ~ 50 community one-hot values) into 64 dense dimensions. This bottleneck is problematic because:

- **Sparse-to-dense compression:** Community one-hot encoding is inherently sparse (only 1 of ~ 50 dimensions is active). A single linear layer may fail to preserve discrete cluster identity.
- **Signal dilution:** The 8 semantically meaningful features (gender, generation, centrality metrics) are numerically overwhelmed by 50 community dimensions that each contribute minimal gradient signal.
- **Expressivity gap:** Standard RotatE learns 64 unconstrained, dense dimensions per entity—a strictly more expressive parameterization than linearly projecting from fixed sparse features.

13.6.2 Community Feature Redundancy

CompGCN’s message-passing mechanism *already captures* community structure implicitly:

- Two CompGCN layers aggregate 2-hop neighborhoods, so entities within the same family tree naturally develop similar representations.
- In MetaFam, community \equiv connected component. CompGCN inherently isolates these components through message passing—no cross-component information flows.
- Explicitly providing community IDs thus **duplicates** information the GNN learns natively from the graph topology.

13.6.3 Forest-of-Trees Structural Limitations

MetaFam’s topology fundamentally limits GNN effectiveness:

- **No cross-family transfer:** 50 disconnected components prevent pattern sharing across families.
- **Shallow depth (4 generations):** Limited tree depth reduces the benefit of multi-layer message passing.
- **Homogeneous neighborhoods:** Family members share similar structural roles, producing redundant aggregated features.

13.6.4 Precision vs Recall Trade-off

The near-equal Hits@10 (0.968 vs 0.972) but divergent Hits@1 (0.509 vs 0.676) reveals an interesting pattern:

- CompGCN-Custom *identifies the correct entity* within the top-10 almost as reliably as standard RotatE.
- But it fails to **rank it first**—the features introduce noise that blurs fine-grained distinctions needed for precise head-of-list ranking.
- This is consistent with the “**feature noise**” hypothesis: the community one-hot encoding adds a strong categorical signal that overwhelms subtle structural differences needed for Hits@1 performance.

13.7 Positive Findings

Despite the negative headline result, several constructive insights emerge:

1. **Best GNN Model:** CompGCN-Custom (MRR 0.677) substantially outperforms RGCN_RotatE (0.442, +53%) and RGCN_DistMult (0.345, +96%), confirming that domain features + composition-based architecture > random init + basis-decomposition.
2. **Near-Perfect Recall:** Hits@10 of 0.968 means the model almost always includes the correct answer in its top-10 predictions, demonstrating strong neighborhood-level understanding.
3. **Architectural Validation:** The hot-swap approach (replacing PyKEEN’s default entity representations with custom `FeatureProjectionRepresentation`) integrates seamlessly with the training pipeline, enabling rapid experimentation with alternative feature sets.
4. **Cross-Task Integration:** This is the only model in our pipeline to combine outputs from Tasks 1 (node features), 2 (community IDs), and 4 (link prediction), demonstrating end-to-end knowledge transfer across the analysis pipeline.

13.8 Implications for Future Work

1. **Learned community embeddings:** Replace one-hot encoding with low-dimensional community embeddings ($C \rightarrow d$, where $d \ll C$) to reduce sparsity and improve gradient flow

2. **Feature ablation:** Systematically test feature subsets (e.g., demographics only, topology only) to isolate which features help and which introduce noise
3. **Deeper networks:** Explore 3–4 CompGCN layers with residual connections to extend the effective receptive field
4. **Alternative composition operators:** Compare subtraction, multiplication, and circular correlation variants of CompGCN
5. **Denser graphs:** Test CompGCN-Custom on connected KGs (e.g., family graphs with spouse relations) where message-passing can transfer information globally

14 Conclusions

14.1 Main Findings

1. **ComplEx is Best for MetaFam:** Achieved 0.877 MRR (full train), 0.998 Hits@10
2. **Custom > PyKEEN (mostly):** Our implementations outperformed PyKEEN for TransE, DistMult, ComplEx due to the over-regularization effect on clean data
3. **Custom RotatE Fails:** Missing unit modulus constraint causes modulus drift, destroying performance on compositional relations
4. **Inverse Leakage Matters:** ~12% validation performance inflation from inverse shortcuts
5. **GNNs are Suboptimal:** Pure embedding approaches outperform GNN+decoder for this sparse, disconnected family graph
6. **CompGCN-Custom is the Best GNN:** With domain features + RotatE decoder, CompGCN achieves 0.677 MRR—53% above RGCN_RotatE—but still 14% below standard RotatE, showing that feature injection helps GNNs but cannot overcome MetaFam’s structural limitations
7. **Features Help Recall, Not Precision:** CompGCN-Custom nearly matches RotatE on Hits@10 (0.968 vs 0.972) but lags substantially on Hits@1 (0.509 vs 0.676), indicating that custom features add noise that blurs fine-grained ranking

14.2 Recommendations

1. **For Clean Synthetic Data:** Use minimal regularization to fully capture deterministic patterns
2. **For RotatE:** Always enforce $|r| = 1$ constraint
3. **For Family Graphs:** Use inverse-leakage-aware splitting
4. **Model Choice:** ComplEx offers the best balance of expressivity and trainability for family KGs

14.3 Future Work

1. Fix custom RotatE with proper unit modulus constraint
2. Explore rule-enhanced link prediction (inject high-confidence rules from Task 3)
3. Test on noisy variants of MetaFam to validate the over-regularization hypothesis
4. Develop inductive models for unseen family trees
5. Investigate learned community embeddings for CompGCN to reduce one-hot sparsity
6. Conduct feature ablation studies to isolate which node attributes benefit GNN link prediction
7. Test CompGCN on denser, connected KGs where cross-community message passing is possible

References

- [1] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. In *NeurIPS*, 2013.
- [2] B. Yang, W. Yih, X. He, J. Gao, and L. Deng. Embedding entities and relations for learning and inference in knowledge bases. In *ICLR*, 2015.
- [3] T. Trouillon, J. Welbl, S. Riedel, E. Gaussier, and G. Bouchard. Complex embeddings for simple link prediction. In *ICML*, 2016.
- [4] Z. Sun, Z.-H. Deng, J.-Y. Nie, and J. Tang. RotatE: Knowledge graph embedding by relational rotation in complex space. In *ICLR*, 2019.
- [5] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *ESWC*, 2018.
- [6] M. Ali et al. PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings. *JMLR*, 2021.
- [7] S. Vashishth, S. Sanyal, V. Nibber, and P. Talukdar. Composition-Based Multi-Relational Graph Convolutional Networks. In *ICLR*, 2020.