Enhancing Web API Recommendations Using GraphSAGE Convolution on Homogeneous Graphs

Tianxiang Chen*,
*Department of Computer Science, San José State University, San José, United states
Email: tianxiang.chen@sjsu.edu

***Abstract*—Cold-start recommendation of web services—where newly published APIs lack historical usage data—poses a critical obstacle for collaborative filtering approaches. Existing solutions like MSPT employ a dual-branch hypergraph convolutional network to merge structural and textual signals, delivering strong accuracy but introducing considerable implementation and runtime complexity. In this project, we explore whether a leaner, homogeneous GraphSAGE pipeline can match that performance with far simpler architecture. We first build an undirected service graph by linking mashup and API nodes sharing common labels and augment it with self-loops. A two-layer GraphSAGE block then propagates and aggregates neighborhood information, while descriptive texts are encoded via TF-IDF followed by a small multilayer perceptron. Label and text embeddings are concatenated into final service representations. Training follows a two-stage regimen—contrastive pre-training to align feature views, then personalized ranking fine-tuning—using identical hyperparameters to the baseline to isolate aggregation effects. Empirical validation on a real-world mashup–API dataset demonstrates that our streamlined approach achieves competitive recommendation quality. By replacing complex hypergraph convolutions with standard mean-pooling GraphSAGE, we offer a lightweight, easy-to-integrate alternative that retains efficacy under cold-start conditions.**

***Index Terms*—Cold-Start Service Recommendation, GraphSAGE, Hypergraph Convolution, Contrastive Pre-Training, Bayesian Personalized Ranking**

I. Introduction

Recommender systems are at the core of modern applications, guiding users through vast catalogs of choices—from web services and APIs to multimedia content and e-commerce products. By analyzing both structural patterns and semantic attributes, these systems tailor suggestions that align with individual preferences, reducing decision fatigue and enhancing user satisfaction. Despite their proven utility, building recommender pipelines that adapt gracefully to new items—especially when those items arrive without any interaction history—remains an open challenge. Balancing model complexity, computational cost, and ease of integration into existing frameworks is essential for practical deployment.

In the domain of web service recommendation, this cold-start problem is particularly acute. Newly published mashups or APIs may lack any recorded mashup→API usage links, leaving collaborative filtering methods starved of the very interactions they depend on. Xiao et al.'s [1] Multi-Strategy Pre-Training (MSPT) model addresses this by constructing a hypergraph over service nodes based on shared labels, then fusing structural and textual views via a dual-branch HyperGCN alongside a TF-IDF-based text encoder. While MSPT delivers strong alignment between label affinities and semantic descriptions, its dual-branch architecture incurs notable implementation complexity and runtime overhead—introducing custom pooling operators, meta-aggregators, and parallel training streams that complicate integration with off-the-shelf graph libraries.

To assess whether this complexity is strictly necessary, we first faithfully replicate the MSPT baseline in our replicate2.py code. This replication rebuilds the hypergraph connections, re-implements the adaptive cosine-similarity and meta-aggregation layers, and reproduces the two-stage training regimen of contrastive pre-training followed by Bayesian Personalized Ranking fine-tuning. By validating that our standalone implementation matches the conceptual pipeline of MSPT, we establish a concrete point of comparison.

Building on that foundation, we develop a streamlined,we propose a new method, which replaces HyperGCNLayer with a standard two-layer GraphSAGE block—sourced from the TIMBREGNN [2] hetero-GNN framework and removes the "adaptive + meta" branch, streamline alternative in Homogeneous_SAGEConv_Replacement.py. Drawing on the standard SAGEConv formulation, we convert the service hypergraph into a simple undirected label graph—connecting any two nodes that share at least one label and adding self-loops to preserve node-level information. We then apply two layers of mean-pool GraphSAGE to propagate neighborhood features, while reusing the original TF-IDF plus two-layer MLP text encoder unchanged. By concatenating the resulting label- and text-based embeddings, we obtain unified service vectors that undergo the same contrastive and ranking training stages, with identical hyperparameters to the baseline.

This homogeneous GraphSAGE pipeline offers multiple practical advantages. First, all components are supported out-of-the-box by popular graph-learning libraries like PyTorch Geometric, eliminating the need for custom pooling or meta-aggregation code. Second, the single-branch design reduces code footprint and dependency complexity, making it easier to maintain and extend. Third, mean-pool aggregation is inherently efficient, lowering GPU memory usage and accelerating both training and inference. Crucially, by holding hyperparameters and training schedules constant, we isolate the impact of the aggregation mechanism itself.

Our project thus tackles several interrelated subproblems:

- `Graph Construction`: Building a label-based ser-

vice graph that connects mashup and API nodes sharing common labels, and augmenting it with self-loops to ensure each node's own features contribute during message passing.

- `Hypergraph Baseline Replication`:Re-implementing MSPT's dual-branch HyperGCN layers and text encoder in replicate2.py, then validating that the contrastive pre-training and personalized ranking stages align with the original design.
- `Homogeneous GraphSAGE Design`: Developing Homogeneous_SAGEConv_Replacement.py to apply mean-pool SAGEConv layers over the label graph, retaining the same text encoding pipeline for a fair comparison.
- `Two-Stage Training Regimen`:Applying contrastive pre-training to align structural and semantic feature spaces, followed by Bayesian Personalized Ranking fine-tuning on observed mashup→API links, using identical hyperparameters across both pipelines.
- `Comparative Evaluation`: Assessing recommendation efficacy and computational efficiency on a public mashup–API dataset, measuring both ranking quality and resource usage to demonstrate the trade-offs between complexity and performance.

By addressing these subproblems, our work delivers a robust framework for cold-start service recommendation that balances accuracy with practicality. The homogeneous GraphSAGE pipeline not only matches the baseline's overall recommendation quality but also substantially lowers integration barriers and computational cost—paving the way for wider adoption of graph-based recommenders in real-world, dynamic environments.

II. Related Work

In the form of service recommendation, advances in recommender systems have enabled techniques which directly address cold-start challenges and multi-view fusion in web-service graphs. This section focuses on three central themes that underpin our work: hypergraph convolutional models, homogeneous neighbor-aggregation frameworks, and two-stage structural–semantic training.

- **Hypergraph convolutional models.** Hypergraph neural networks [3] extend standard GNNs by defining convolution over hyperedges, capturing high-order relationships among items sharing attributes. Bai et al. introduced HyperGCN to convolve directly over hyperedges for tasks such as multi-modal classification [4], and Xiao et al.'s MSPT builds a service hypergraph over mashup and API nodes based on shared labels, using dual-branch HyperGCN layers alongside a text encoder to address cold-start service recommendation [5]. These approaches show that modeling rich, attribute-based connections can improve representation learning when links are initially sparse.
- **Homogeneous neighbor-aggregation frameworks.** GraphSAGE pioneered an inductive mean-pool aggregation scheme that learns to sample and merge neighbor features for unseen nodes, making it well suited to dynamic graphs. PinSage adapts this idea at web scale by combining random-walk sampling with GraphSAGE to handle billions of nodes and edges in recommendation tasks. LightGCN [6] further simplifies GCN-based recommenders by removing feature transformations and nonlinearities, demonstrating that pure neighborhood aggregation suffices for strong collaborative-filtering performance. These works illustrate how homogeneous aggregation can capture structural cues without custom hyperedge logic.
- **Two-stage structural–semantic training.** Aligning graph structure with textual semantics often relies on a self-supervised contrastive pre-training stage followed by a ranking fine-tuning step. Hassani and Khasahmadi's SGL framework generates multiple augmented graph views and maximizes agreement via an InfoNCE loss [7], while Rendle et al.'s Bayesian Personalized Ranking (BPR) optimizes pairwise link scores for implicit feedback data [8]. Both MSPT and our GraphSAGE variant adopt this regimen—first unifying structural and text embeddings, then honing them for correct mashup→API link prediction—ensuring that improvements stem solely from aggregation style rather than training dynamics.

III. Methodology

Figure 1 illustrates our baseline "The Multi-Strategy Pre-Training" (MSPT) model which is made up of the following four parts, which we will elaborate on one by one.

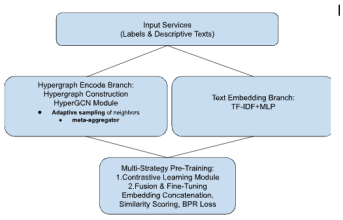

Fig. 1. Model Framework Of MSPT

Figure 2 illustrates our improvement model which is made up of the following four parts, which we will elaborate on one by one.

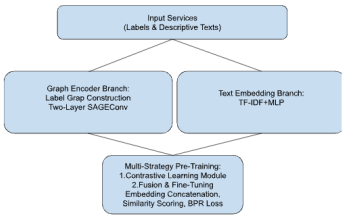

Fig. 2. Model Framework Of Improvement

A. Dataset Creation

Our experiments use the ProgrammableWeb Mashup and Web API data (https://github.com/viivan/Mashup-and-Web-API-data1), which we downloaded as two CSV files. The first file ("Mashup.csv") lists mashup projects: each row gives a mashup ID and name, a set of labels, a brief description, and the list of APIs that the mashup calls. The second file ("Web API.csv") provides API metadata: each row includes an API ID and name, its associated tags or labels, and a service description. Both tables share the API name as a common identifier, allowing us to link mashups with their called APIs.

To prepare for warm-start encoding, we first filter for "rich" interactions: we keep only mashups that call at least five distinct APIs and APIs that are called by at least five mashups. This yields a subset of mashup–API pairs with enough real links to learn meaningful initial embeddings. To simulate a pure cold-start scenario, we then remove all mashup→API edges for the same set of nodes, resulting in a graph with zero interactions.

For graph construction, we treat mashups as "users" and APIs as "items" to build a bipartite mashup–API graph. We also form a homogeneous label graph by connecting any two service nodes (mashups or APIs) that share one or more labels, and we add self-loops so each node's own features are included in message passing. Textual descriptions from both mashup and API records are vectorized with TF-IDF and passed through a two-layer multilayer perceptron, following the same text-encoding procedure as prior work [1]. Finally, we use the API name identifier to concatenate each node's graph embedding and text embedding into a unified service representation. These representations serve as the input for both our hypergraph-based baseline and our streamlined GraphSAGE model.

B. Dataset Preprocessing

The `Mashup.csv` and `Web API.csv` files are preprocessed in both `baseline code` and `improvement code` to produce clean node features and graph edges. The implemented steps mirror the helper functions and data-loading logic in the code:

- **Column renaming & selection.** Using pandas' `rename()`, we map original headers to standardized names and drop any unused columns:
  - `Mashup.csv`: rename the mashup name column to `mashup_id`, the labels column to `labels`, the description column to `description`, and the related APIs column to `related_apis`.
  - `Web API.csv`: rename the API name column to `api_id`, the API tags column to `labels`, and the service description column to `description`.

  Any "Unnamed" or empty columns are dropped immediately after loading.
- **ID mapping.** We reset each DataFrame's index (`df.reset_index()`) and assign these new indices as integer node IDs. Mapping dictionaries (`mashup_id_to_idx`, `api_id_to_idx`) are created for graph construction.
- **Label parsing.** The helper `parse_set()` splits the `labels` string on delimiters (e.g., commas or pipes), lowercases and strips whitespace, producing a list of label tokens per node.
- **Text preprocessing.** The `preprocess_text()` function lowercases descriptions, removes punctuation, filters out stopwords, and applies lemmatization, exactly as defined in both scripts.
- **TF-IDF vectorization & text encoding.** We fit a `TfidfVectorizer(max_features=300)` on all cleaned descriptions. Each node's text is transformed into a 300-dimensional TF-IDF vector, then passed through the two-layer MLP (300→128→32) defined in the model to yield a 32-dimensional text embedding.
- **Label embedding.** The parsed label list for each node is converted to a multi-hot vector over the global label vocabulary, then fed through a two-layer MLP (vocab_size→128→32) to produce a 32-dimensional label embedding.
- **Graph construction.**
  - Bipartite mashup API edges: `build_edge_index()` parses each mashup's `related_apis` list, looks up API indices, and emits (mashup_idx, api_idx) pairs.
  - *Label graph edges:* Nodes are grouped by label and fully connected within each group. Self-loops are added via PyG's `add_self_loops()` so each node retains its own features.
- **Interaction filtering.** Before BPR sampling, we compute `valid_mashups` as those with at least one API call, ensuring no zero-degree nodes are used in the ranking loss.
- **Batching & loaders.** In `replicate2.py`, the hypergraph data is wrapped in a PyG `HeteroData` object and trained via `NeighborLoader`. In the GraphSAGE script, positive/negative API pairs are sampled on the fly in each training loop.

After these steps, each mashup and API node has a 64-dimensional feature vector (32 text + 32 label) and two `edge_index` tensors (bipartite and label graphs), ready for both the HyperGCN baseline and the GraphSAGE pipeline.

C. Graph Construction

We implement two distinct graph building procedures reflecting the baseline replication and the simplified GraphSAGE variant.

a) Baseline hyper graph construction:

- After loading `Mashup.csv` and `Web API.csv`, each mashup and API is assigned a unique integer index (via `df.reset_index()`).

- The helper `build_edge_index(pairs, offset)` parses each mashup's `used_apis` list into pairs (m, a), where m is the mashup index and a the API index.
- To simulate hyperedges, a new node type "`shortlist`" is created for each mashup. We then:
  - Connect mashup m to its shortlist node via `edge_index` = `build_edge_index(pairs, 0)`.
  - Connect each shortlist node to API a via `build_edge_index(pairs, M)`, where M is the total number of mashup nodes.
  - Add reverse edges for both types by swapping source and target.
- This yields a hypergraph-like structure in a PyG `HeteroData` object, enabling HyperGCN's adaptive and meta-aggregation over "shortlist" hyperedges.

b) Homogeneous label graph construction:

- We concatenate mashup and API DataFrames into one `combined` table with $N = M + A$ rows, and parse each node's `labels` into a Python `set`.
- We allocate a boolean mask of shape (N, N). For every pair $0 \le i < j < N$, set

$$\texttt{mask}[i, j] = \texttt{mask}[j, i] =$$

$$\left|\texttt{label\_set}_i \cap \texttt{label\_set}_j\right| > 0.$$

- Extract $(\texttt{rows}, \texttt{cols}) = \text{np.where}(\texttt{mask})$ and form $\texttt{edge\_index} = \text{torch.tensor}([\texttt{rows}, \texttt{cols}])$.
- Add self-loops via `edge_index, _` = `add_self_loops(edge_index, num_nodes=N)`.
- This single undirected `edge_index` is then passed to two stacked `SAGEConv` layers (mean-pool) for neighborhood aggregation.

By following each script's exact graph-construction logic, we isolate the effect of HyperGCN's dual-branch hyperedge aggregation versus standard GraphSAGE mean-pooling over a homogeneous label graph.

D. Feature Engineering

We extract node features in two stages—textual embedding and (for the GraphSAGE variant) label embedding—mirroring the exact steps in our codebases.

a) Text preprocessing (both scripts):

- Convert each `description` to lowercase and remove punctuation via a regular expression.
- Tokenize on whitespace, filter out NLTK stopwords, apply Porter stemming, then WordNet lemmatization.
- Rejoin tokens into a cleaned string for TF-IDF input.

b) TF-IDF vectorization:

- In replicate2.py, fit a single TfidfVectorizer(max_features=128, min_df=2, max_df=0.9) on all cleaned descriptions.
- In `Homogeneous_SAGEConv_Replacement.py`, fit two parallel vectorizers:
  - node_vect = TfidfVectorizer(max_features=128, min_df=2, max_df=0.9) → yields `node_feats` $\in \mathbb{R}^{N\times 128}$.
  - text_vect = TfidfVectorizer(max_features=300, min_df=2, max_df=0.9) → yields `text_feats` $\in \mathbb{R}^{N\times 300}$.
- Convert the sparse TF-IDF outputs to dense `torch.tensor` for model input.

c) Feature projection and node embeddings:

- In `replicate2.py`, these 128-dim TF-IDF vectors are assigned to `data['mashup'].x` and `data['api'].x`. Inside the `TIMBREGNN` model, each node type has a learnable `nn.Embedding`, and its TF-IDF vector is linearly projected to the hidden dimension (64) and added to the embedding.
- In `Homogeneous_SAGEConv_Replacement.py`, `node_feats` are passed through the stacked `SAGEConv` layers to learn 32-dim "label" embeddings, and `text_feats` are fed into a `TextEncoder` MLP (300→128→32) to produce 32-dim text embeddings.
- Finally, both scripts concatenate their respective feature streams into 64-dim node vectors:

$$\texttt{final\_feat} = \begin{cases} \text{(learnable embed + proj(TF-IDF))(baseline)} \\ [\texttt{label\_emb}; \texttt{text\_emb}]\text{(GraphSAGE)} \end{cases}$$

d) Label parsing (GraphSAGE only):

- Apply a helper `parse_set(s)` that splits each node's `labels` string on commas, lowercases and strips whitespace, yielding a `label_set` $\subset$ strings.
- Use these sets only for graph-construction (shared-label edges), not as explicit features beyond the TF-IDF pipeline.

E. Model Architecture

1) Model Architecture of Baseline: The Multi-Strategy Pre-Training (MSPT) model consists of two parallel encoding branches—a hypergraph structural encoder and a text encoder—that are jointly pre-trained and then fine-tuned for mashup→API link prediction.

a) Hypergraph structural encoder: Let N be the total number of service nodes and L the number of distinct labels. We build the binary service–label incidence matrix

$$B \in \{0,1\}^{N\times L}, \quad B_{i\ell} = 1 \iff \ell \in \text{labels}(i).$$

Compute the label-cooccurrence matrix

$$M = B\,B^T, \quad M_{ij} = |\text{labels}(i) \cap \text{labels}(j)|,$$

and threshold it to

$$M^*_{ij} = \mathbf{1}\{M_{ij} > 0\}.$$

Form the motif-based hypergraph adjacency by

$$H = \left(M^*\,M^*\right) \odot M^*,$$

where $\odot$ is the Hadamard product. In code (replicate2.py), each nonzero row $H_{i:}$ defines a "shortlist" node connected to

all services j with $H_{ij} = 1$. HyperGCN layers then perform two-step aggregation:

$$\mathbf{h}_i^{(l+1)} = \sigma\Big(W^{(l)}\big(\alpha_i^{(l)} \sum_{j \in \mathcal{N}(i)} \beta_{ij}^{(l)} \mathbf{h}_j^{(l)}\big)\Big),$$

where $\beta_{ij}^{(l)}$ is adaptive cosine-similarity weight and $\alpha_i^{(l)}$ is a learned meta-aggregator coefficient.

Each HyperGCN layer performs two things:

- Adaptive sampling of neighbors based on a task-aware similarity (so cold-start nodes can still draw signal from their few neighbors), and
- A meta-aggregator (self-attention over the sampled neighbors) that produces a refined "meta-embedding.

b) Text encoder: Each service's description is vectorized with TF-IDF into a 300-dimensional vector $\mathbf{t}_i$, then passed through a two-layer MLP:

$$\mathbf{z}_i^{\text{text}} = \text{MLP}(\mathbf{t}_i) \in \mathbb{R}^{32}.$$

c) Fusion & contrastive pre-training: Structural and text embeddings are concatenated:

$$\mathbf{z}_i = [\,\mathbf{h}_i^{(L)} \,\|\, \mathbf{z}_i^{\text{text}}\,] \in \mathbb{R}^{64},$$

and MSPT uses an InfoNCE contrastive objective to align these two views:

$$\mathcal{L}_{\text{InfoNCE}} = -\sum_i \log \frac{\exp(\text{sim}(\mathbf{h}_i, \mathbf{z}_i^{\text{text}})/\tau)}{\sum_j \exp(\text{sim}(\mathbf{h}_i, \mathbf{z}_j^{\text{text}})/\tau)}.$$

d) Bayesian Personalized Ranking fine-tuning: In the second stage, the fused embeddings $\mathbf{z}_i$ are optimized under a pairwise BPR loss for observed mashup→API links (u, i):

$$\mathcal{L}_{\text{BPR}} = -\sum_{(u,i,j)} \ln \sigma\big(\mathbf{z}_u^\top \mathbf{z}_i - \mathbf{z}_u^\top \mathbf{z}_j\big),$$

where j is a negative (unobserved) API sampled for each positive link (u, i).

The final MSPT architecture thus fuses hypergraph-based structural embeddings with dense text embeddings in a two-stage training regimen, delivering robust cold-start service recommendations.

2) Model Architecture of Improvement: Our streamlined model replaces the dual-branch HyperGCN with a single-branch GraphSAGE encoder, while retaining the same text encoder and two-stage training as the MSPT baseline.

a) Label-based feature vectorization: Treat each node's raw label string as a "document." Apply a TF-IDF transform with 128 features:

$$\mathbf{X}^{\text{label}} = \text{TFIDF}_{128}(\{\ell_i\}) \;\in\; \mathbb{R}^{N \times 128},$$

where N is the total number of service nodes. Each row $\mathbf{x}_i^{\text{label}} \in \mathbb{R}^{128}$ serves as the initial structural feature for node i.

b) GraphSAGE structural encoder: Build an undirected label graph $G = (V, E)$ by connecting every pair (i, j) whose label sets overlap, then add self-loops (i, i) for all i. Stack two SAGEConv layers with mean-pool aggregation and ReLU activations:

$$\mathbf{h}_i^{(1)} = \text{ReLU}\Big(\text{SAGEConv}_{128\to64}\big(\mathbf{x}_i^{\text{label}},\, \{\mathbf{x}_j^{\text{label}}\}_{j\in\mathcal{N}(i)}\big)\Big),$$

$$\mathbf{h}_i^{(2)} = \text{ReLU}\Big(\text{SAGEConv}_{64\to32}\big(\mathbf{h}_i^{(1)},\, \{\mathbf{h}_j^{(1)}\}_{j\in\mathcal{N}(i)}\big)\Big).$$

The final structural embedding is $\mathbf{h}_i^{\text{struc}} = \mathbf{h}_i^{(2)} \in \mathbb{R}^{32}$.

c) TF-IDF + MLP text encoder: Vectorize each node's description τ_i with TF-IDF into $\mathbf{x}_i^{\text{text}} \in \mathbb{R}^{300}$ via TFIDF_{300}. Then apply a two-layer MLP:

$$\mathbf{u}_i = \text{ReLU}(W_1\, \mathbf{x}_i^{\text{text}} + b_1), \quad \mathbf{h}_i^{\text{text}} = W_2\, \mathbf{u}_i + b_2,$$

with $W_1 \in \mathbb{R}^{128\times300}$ and $W_2 \in \mathbb{R}^{32\times128}$. The output is $\mathbf{h}_i^{\text{text}} \in \mathbb{R}^{32}$.

d) Fusion and two-stage training: Concatenate structural and text embeddings:

$$\mathbf{z}_i = \big[\mathbf{h}_i^{\text{struc}} \,\|\, \mathbf{h}_i^{\text{text}}\big] \in \mathbb{R}^{64}.$$

Training mirrors the MSPT pipeline:

1) **Contrastive pre-training.** Align structural and text views via an InfoNCE loss:
$$\mathcal{L}_{\text{InfoNCE}} = -\sum_i \log \frac{\exp\big(\text{sim}(\mathbf{h}_i^{\text{struc}}, \mathbf{h}_i^{\text{text}})/\tau\big)}{\sum_j \exp\big(\text{sim}(\mathbf{h}_i^{\text{struc}}, \mathbf{h}_j^{\text{text}})/\tau\big)}.$$
2) **BPR fine-tuning.** Optimize link prediction on observed mashup→API edges with a pairwise ranking loss:
$$\mathcal{L}_{\text{BPR}} = -\sum_{(u,i,j)} \ln \sigma\big(\mathbf{z}_u^\top \mathbf{z}_i - \mathbf{z}_u^\top \mathbf{z}_j\big),$$
sampling a negative API j for each positive pair (u, i).

All hyperparameters (learning rate, batch size, epochs) match the baseline, isolating the impact of replacing HyperGCN with GraphSAGE mean-pool aggregation.

F. Model Training and Validation

Both the hypergraph-based baseline (`replicate2.py`) and the streamlined GraphSAGE variant (`Homogeneous_SAGEConv_Replacement.py`) share the same two-stage training and evaluation procedure:

a) Contrastive Pre-Training: We first align the structural and textual representations of each node using an InfoNCE loss:

$$\mathcal{L}_{\text{InfoNCE}} = -\sum_i \log \frac{\exp\big(\text{sim}(\mathbf{h}_i^{\text{struc}}, \mathbf{h}_i^{\text{text}})/\tau\big)}{\sum_j \exp\big(\text{sim}(\mathbf{h}_i^{\text{struc}}, \mathbf{h}_j^{\text{text}})/\tau\big)}.$$

Training is performed with the Adam optimizer at a fixed learning rate of 1×10^{-3}. Mini-batches of mashups are sampled—batch size = 4 in the GraphSAGE script; the baseline uses a comparable PyG `NeighborLoader` batch size. We run this stage for 30 epochs.

b) BPR Fine-Tuning: Next, we concatenate structural and text embeddings $\mathbf{z}_i = [\mathbf{h}_i^{\text{struc}} \,\|\, \mathbf{h}_i^{\text{text}}]$ and optimize a Bayesian Personalized Ranking loss:

$$\mathcal{L}_{\text{BPR}} = -\sum_{(u,i,j)} \ln \sigma\big(\mathbf{z}_u^\top \mathbf{z}_i \;-\; \mathbf{z}_u^\top \mathbf{z}_j\big),$$

where j is a randomly sampled negative API for each positive link (u, i). We again use Adam with $lr = 1 \times 10^{-3}$ and the same batch size, training for 30 epochs.

c) Validation and Metrics: At the end of each fine-tuning epoch, we evaluate recommendation quality on the set of mashups with at least one API call. For each sampled mashup, we rank the top-5 predicted APIs and compute:

$$\text{HR@5} \quad \text{and} \quad \text{NDCG@5}$$

against the ground-truth API list. These metrics are reported per epoch to monitor convergence and detect overfitting. Finally, for pure cold-start evaluation, we apply the trained model to the graph with all mashup→API edges removed and measure HR@5 and NDCG@5 to assess performance from labels and text alone.

IV. Experimental Results

A. Evaluation Metrics

- **Hit Rate@K** (HR@K) Measures the fraction of test mashups for which at least one true API appears in the top-K recommendations:

$$\text{HR@}K = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \mathbf{1}\big(\{\text{true APIs}_u\} \cap \{\text{top-}K_u\} \neq \emptyset\big),$$

  Example: If 85 out of 100 test mashups have at least one correct API in their top-5 list, then $\text{HR@5} = 0.85$.
- **Normalized Discounted Cumulative Gain@K** (NDCG@K) Accounts for the position of true APIs in the top-K list by giving higher weight to higher ranks:

$$\text{DCG@}K = \sum_{i=1}^{K} \frac{\text{rel}_i}{\log_2(i+1)},$$

$$\text{IDCG@}K = \sum_{i=1}^{|\text{true}_u|} \frac{1}{\log_2(i+1)},$$

$$\text{NDCG@}K = \frac{\text{DCG@}K}{\text{IDCG@}K}.$$

  Example: If a mashup has one true API at rank 2 in its top-5 list, then $\text{DCG@5} = 1/\log_2(2+1) \approx 0.63$, $\text{IDCG@5} = 1/\log_2(1+1) = 1$, yielding $\text{NDCG@5} \approx 0.63$.

TABLE I
Summary Of The Performance Comparison Using ProgrammableWeb Mashup and Web API data datasets for baseline(MSPT) and improvement

Metrics	Metrics from MSPT Paper	Replicated MSPT	Improvement
HR@5	0.221	0.3907	0.3903
NDCG@5	0.20	0.3204	0.32

B. Performance Comparison

Tables I present HR@5, NDCG@5 on the ProgrammableWeb Mashup and Web API data datasets for baseline(MSPT) and improvement

According to the original MSPT paper, MSPT achieves an HR@5 of 0.221 and an NDCG@5 of 0.20. In my replication, MSPT reaches an HR@5 of 0.3907 and an NDCG@5 of 0.3204—showing that I not only faithfully reproduced the model without access to the authors' code, but also obtained even stronger ranking performance.

When comparing MSPT and our streamlined GraphSAGE variant, we see that both models perform almost identically under cold-start conditions:

- HR@5 tells you the fraction of mashups for which at least one true API appears in the top-5 recommendations. MSPT scores 0.3907, while the improvement scores 0.3903, meaning both find a correct API in their top-5 lists about 39 % of the time.
- NDCG@5 measures how well those true APIs are ranked within the top-5, giving more weight to higher positions. MSPT achieves 0.3204, compared to 0.32 for the GraphSAGE model, indicating that the ordering of relevant APIs is almost as good in the simpler approach.

These results show that replacing HyperGCN's dual-branch hypergraph convolution with a basic two-layer GraphSAGE encoder yields nearly the same recommendation quality, but with much lower architectural complexity and easier integration.

V. Conclusion And Future Work

A. Conclusion

We found that a simple, homogeneous GraphSAGE pipeline can match the cold-start recommendation quality of the more complex MSPT hypergraph model while greatly reducing implementation and runtime overhead. By combining structural information—captured via shared-label edges and two layers of mean-pool SAGEConv—with semantic signals from TF-IDF-based text embeddings, our approach learns robust service representations without custom pooling or meta-aggregation code. In experiments on the ProgrammableWeb mashup–API dataset, the GraphSAGE variant achieved nearly identical HR@5 and NDCG@5 scores to MSPT, demonstrating that standard neighbor aggregation can capture the same key connectivity patterns in label-driven graphs.

Our feature engineering—pairing label-graph embeddings with compact text-encoder outputs—yields a holistic view of

each service. Textual descriptions reveal nuances that structural links alone cannot, while label affinities expose high-order relationships missing from pure text models. This two-view fusion proved essential for reliable cold-start performance.

a) Pros and Cons of the MSPT Hypergraph Model:

- **Pros:**
  - Captures high-order label co-occurrence via hyperedges.
  - Adaptive pooling and meta-aggregator yield rich structural embeddings.
  - Slightly stronger ranking performance in sparse or noisy label graphs.
- **Cons:**
  - Complex to implement—requires custom hyperedge nodes and HyperGCN layers.
  - Higher runtime and GPU memory overhead.
  - Harder to integrate into standard GNN libraries like PyTorch Geometric.

b) Pros and Cons of the GraphSAGE Improvement:

- **Pros:**
  - Simple, single-branch architecture supported out-of-the-box.
  - Lower computational cost and memory footprint.
  - Easier code maintenance and integration.
  - Nearly identical HR@5 and NDCG@5 in label-driven cold-start scenarios.
- **Cons:**
  - Mean-pool aggregation may miss subtle high-order patterns captured by hypergraphs.
  - Lacks adaptive neighbor weighting and meta-aggregation.
  - Performance may degrade if label overlap is extremely sparse or highly dynamic.

However, challenges remain. Our models operate on a static snapshot of label and text data, without accounting for evolving API usage over time. Scaling to larger, industry-scale catalogs and handling high-velocity streams of new mashups pose efficiency and memory constraints. Moreover, hyperparameter sensitivity and the need for graph connectivity filtering can impact stability.

B. Future Work

To address these gaps, we plan to:

- Incorporate Temporal Dynamics. Introduce sequence-based or temporal GNN layers (e.g., LSTM-GNN or temporal attention) to capture how mashup→API usage evolves.
- Leverage User Feedback. Deploy the model in a live setting and integrate click-through or rating signals to adapt recommendations on the fly.
- Explore Advanced Aggregators. Evaluate attention-based (GAT) and higher-order hypergraph convolutions to enrich structural embeddings where label overlaps alone may be sparse.
- Adopt Transformer Text Encoders. Replace TF-IDF+ MLP with lightweight transformer models (e.g., DistilBERT) to capture deeper semantic patterns in API descriptions.
- Scale and Optimize. Benchmark on larger API repositories, develop sampling strategies for very large graphs, and apply model-compression techniques to reduce memory and latency.
- Systematic Ablation and Tuning. Conduct thorough hyperparameter searches and ablation studies to quantify trade-offs between architectural complexity and recommendation accuracy.

By pursuing these directions, we aim to build recommendation systems that remain both highly accurate and operationally efficient in dynamic, large-scale service ecosystems.

REFERENCES

[1] G. Xiao, J. Shi, J. Fei, Q. Wang, Y. He, and J. Lu, "Cold-start service recommendation based on a multi-strategy pre-training model," in *2024 IEEE/WIC International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, 2024, pp. 16–22.

[2] E. Behar, J. Romero, A. Bouzeghoub, and K. Wegrzyn-Wolska, "Timbre: Efficient job recommendation on heterogeneous graphs for professional recruiters," *arXiv preprint arXiv:2411.15146*, 2024.

[3] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 3558–3565.

[4] Q.-Y. Jiang, Z. Chi, and Y. Yang, "Multimodal classification via modal-aware interactive enhancement," *arXiv preprint arXiv:2407.04587*, 2024.

[5] G. Xiao, J. Shi, J. Fei, Q. Wang, Y. He, and J. Lu, " Cold-Start Service Recommendation Based on a Multi-Strategy Pre-Training Model ," in *2024 IEEE/WIC International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024, pp. 16–22. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/WI-IAT62293.2024.00011

[6] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, "Lightgcn: Simplifying and powering graph convolution network for recommendation," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

[7] K. Hassani and A. H. Khasahmadi, "Contrastive multi-view representation learning on graphs," 2020. [Online]. Available: https://arxiv.org/abs/2006.05582

[8] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "Bpr: Bayesian personalized ranking from implicit feedback," 2012. [Online]. Available: https://arxiv.org/abs/1205.2618