GSL-Mash: Enhancing Mashup Creation Service Recommendations through Graph Structure Learning

Sihao Liu, Mingyi Liu(✉), Tianyu Jiang, Shuang Yu, Hanchuan Xu, and Zhongjie Wang(✉)

Faculty of Computing, Harbin Institute of Technology, Harbin, China
{liusihao, tyjiang}@stu.hit.edu.cn, {liumy, yushuang, xhc, rainy}@hit.edu.cn

Abstract. The proliferation of Web APIs has facilitated the creation of numerous software applications through the integration of diverse services, commonly referred to as mashups. However, the growing complexity and number of available Web APIs pose significant challenges in API services selection. Current service recommendation models, predominantly based on Graph Neural Networks (GNNs), often underperform due to the simplistic and overly complex APIs co-occurrence graphs they utilize, which impede both efficiency and performance. This paper introduces a novel model, GSL-Mash, which incorporates graph structure learning (GSL) to optimize graph data in service recommendations. By refining the graph structure to retain only pertinent connections, our model significantly reduces unnecessary complexity and noise, enhancing both the efficacy and accuracy of service recommendations. We validate GSL-Mash using real-world datasets from ProgrammableWeb, where it outperforms established baselines with up to 45.39% improvement on NDCG@10 metric. Additionally, we contribute to the academic and development communities by making our implementation publicly available. This study not only advances the technology of service recommendation systems but also sets a foundational approach for future research in optimizing graph-based service recommendation models.

Keywords: Service recommendation · Mashup creation · Graph Neural Network · Graph Structure Learning

1 Introduction

With the rapid advancement of the Web, mashups and Web APIs are increasingly capturing attention. The integration of Web API calls has become a fundamental element in software development, with a growing number of companies and organizations shaping their strategies to leverage these technologies to meet developmental needs. At the same time, the increasing number and diversity of available services introduce new challenges for service management and reuse. Consequently, the selection of suitable APIs from a vast array to create API

compositions, commonly referred to as mashups or workflows, remains a critical research area in both academic and industrial sectors, aiming to enhance the efficiency and flexibility of development.

Existing service recommendation models can be classified into three chronological categories: traditional[6, 18], deep learning-based[4, 9, 1] and graph network-based[10, 12]. This classification not only highlights the technological evolution but also underscores the success and necessity of utilizing graph structures to address service recommendation challenges. Despite these advancements, existing approaches often overlook the quality of the constructed graphs. Commonly, these models simplistically use API co-occurrence relationships to build graphs and then feed them into Graph Neural Networks (GNNs) to obtain service feature embeddings for downstream recommendation tasks. These approach fails to yield optimal graph structures for service recommendation. Specifically, this method results in a graph that is far more complex than the ideal "ground truth graph". For example, "Weatherbit" and "OpenWeatherMap" have similar functionalities, providing real-time weather data, weather forecasts, and historical weather data. When they appear together in an API co-occurrence graph and form an edge, it may lead to their simultaneous recommendation. This is something we want to avoid; we only want to recommend the API with the highest confidence among those with similar functionalities. This is considered noise in the API co-occurrence graph that we have defined. Such a graph primarily leads to two significant issues:

- **Efficiency Concern**: The excessive complexity of the API co-occurrence graph leads to a sharp increase in the number of edges; for instance, a mashup composed of 20 APIs could result in approximately 200 (undirected) edges. Since Graph Neural Networks (GNNs) are message-passing neural networks, their training efficiency is heavily dependent on the number of edges in the graph. Thus, this overly complex graph structure significantly reduces the model's efficiency.
- **Performance Degradation**: The surplus edges not only fail to provide beneficial information for the training of GNN-based service recommendation models but also act as noise, severely degrading the performance of GNN-based models. This phenomenon has been demonstrated in numerous studies, where simple models, through careful parameter tuning and tricks selection, have outperformed those service recommendation models based on GNNs.

In this paper, we introduce GSL-Mash, an innovative model that integrates graph structure learning to tackle the challenges associated with processing graph data in service recommendations(mashup creation). Graph structure learning (GSL) involves techniques to refine and optimize graph data, ensuring that only relevant connections are maintained. This is critical in addressing inefficiencies and performance degradation caused by excessive and noise edges in API co-occurrence graphs. GSL-Mash begin with a graph containing redundant edges. By applying graph structure learning, we evaluate and model the relationships between APIs, filtering out unnecessary edges. This process results in a streamlined graph that

feeds into the GNNs, containing only the essential edges required for robust service recommendation. This method significantly reduces redundancy and noise in the graph, which enhances the performance of our model. Additionally, this approach offers insights into optimizing graph structures for GNN-based service recommendations, showcasing the potential of graph structure learning in improving data handling and model efficacy.

In summary, the key contributions of this paper are outlined as follows:

- **Innovative Model Design**: We introduce GSL-Mash, a novel service recommendation approach utilizing graph neural networks. This model employs advanced graph construction and graph structure learning techniques to effectively address issues of graph noise and incomplete data, enhancing the accuracy and reliability of service recommendations.
- **Pioneering Introduction of GSL**: To the best of our knowledge, we are the first to apply graph structure learning, an emerging method in graph optimization, to the field of service recommendation. This approach significantly improves the structural efficiency of graphs used in our models. The effectiveness and forward-thinking nature of this integration not only demonstrate its immediate benefits but also serve as a foundation for future research in optimizing graph structures for service recommendations.
- **Demonstrated Effectiveness**: Our model was rigorously tested on real-world datasets ProgrammableWeb[1], a platform widely recognized in the research community. The results are impressive, significantly surpassing established baselines. Additionally, we have made our code available to the public and the research community[2], fostering further innovation and validation of our approach.

2 Related work

2.1 Service recommendation algorithms

Traditional service recommendation algorithms are mainly divided into content-based filtering and collaborative filtering (CF). Content-based filtering uses mashup and API characteristics such as descriptions and tags to make recommendations. Instead, CF relies on historical interactions, treats mashups as users, and APIs as items, suggesting APIs based on mashup-like behavior. However, it faces challenges such as cold start and sparsity issues [9].

Deep learning-based service recommendation algorithms can identify complex non-linear mashup-API relationships to produce more accurate recommendations. For example, Neural Collaborative Filtering (NCF) [4] can set mashup-API relationships to user-project relationships to capture nonlinear relationships. The attention-based model also shows good results on some service recommendation tasks. Peng et al. [9] uses attention mechanism to assign different weights to

[1] https://www.programmableweb.com/
[2] https://github.com/MyToumaKazusa/GSL-Mash

adjacent different service nodes. Marwa et al. [1] built a service network that utilizes two attention mechanisms to consider functional-related features to learn to intelligently discover services.

The service recommendation method of constructing graph network also has great research value. Many recent approaches are based on graph neural networks, which use mashup and API call information to build a network of services and aggregate features of adjacent nodes to form a node representation. For example, Shi et al. [10] proposes a new probabilistic topic model to predict potential connections between mashups and APIs in a service network. Tang et al. [12] modeled API-related data into a heterogeneous information network, used the original path of APIs in the network to obtain contextual semantics, and combined with the attention mechanism to recommend services

However, in a nutshell, none of the existing approaches make good use of the graph structure to enhance the input mashup and service data, and inevitably face the problem of graph noise and missing graph information. This also leads to their aggregated features containing a lot of noise and harmful information, which is an important reason for us to introduce graph structure learning.

2.2 Graph structure learning

Graph neural networks (GNNs) are widely used for graph structure data analysis but are sensitive to the quality of graph structure. Poor results often arise from graphs with redundant or missing structures. To optimize graph structure for specific tasks, some works focus on graph structure learning (GSL), aiming to simultaneously refine the graph structure and its representation for better performance.

The general process of GSL is as follows: First, if the original graph structure is missing, it is necessary to construct the original graph structure from the input original data, and then learn an optimized graph structure by measuring functions such as Gaussian kernel, etc., or through neural networks. Finally, the graph data is input to GNN and transmitted to the downstream task and constantly adjusted according to the results. Fan et al. [3] uses the reparameterization technique to replace nondifferentiable nearest neighbor sampling with differentiable sampling, and constructs a difference GSL model. Zhang et al. [16] propose dynamic GSL, a novel supervisory signal that enables models to effectively combat noise in dynamic graphs. Zhao et al. [17] explores a new direction to learn a generic structural learning model that can generalize across graph datasets in the open world. In general, GSL can improve the performance and robustness of the model by solving problems such as graph noise and missing graph data. Our approach also integrates attention-based GSL, which brings great improvements.

3 Method

3.1 Framework of GSL-Mash

As illustrated in Fig. 1, GSL-Mash model consists of three main components: the *Graph Structure Learning Component*, the *Mashup Transformation Component*,

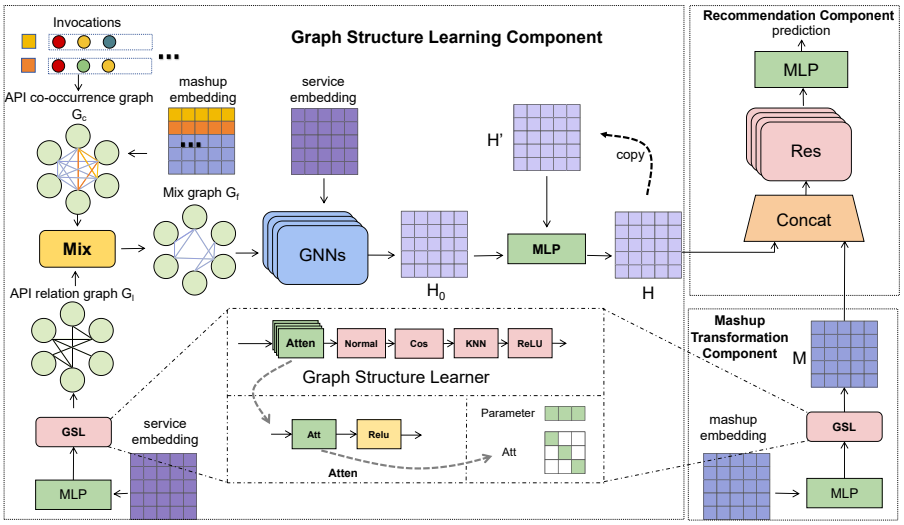


Fig. 1: Overview framework of GSL-Mash. Arrows denote the flow of data and grids represent feature vectors.

and the *Recommendation Component*. The core idea of GSL-Mash is to use the graph structure learner to learn the similarity between APIs and between mashups, and then use these similarity to optimize the input data. Specifically, we treat the edges constructed by similar APIs in the API co-occurrence graph as noise and use the similarity between APIs to denoise the API co-occurrence graph, thereby obtaining a more concise and better-performing graph representation. On the other hand, we utilize the similarity between mashups to ensure that similar mashups have more similar vector representations, making them more likely to call commonly used APIs. This enhances the stability of the test results. The *Graph Structure Learning Component* is tasked with constructing an optimized API invocation graph G_t and use it to learn an aggregated vector representation of the APIs, denoted as $\mathbf{H}$. Specifically, it uses the original invocation data to build the API co-occurrence graph G_c, and then learns the similarity between APIs, called G_l through the graph structure learner depicted in Fig. 1. The results are then merged using the Mix function to obtain the optimized graph G_t. Subsequently, we aggregate and refine the optimized graph G_t through a GNN to produce the enhanced aggregated API vector representation $\mathbf{H}$. Similarly, the *Mashup Transformation Component* aims to learn the aggregated vector representation of mashups $\mathbf{M}$. In this part, we first compress the dimensional space of the mashup data using an multi-layer perceptron (MLP). Then, we learn the similarity of different mashup vectors through the graph structure learner, and use this similarity to enhance the mashup vectors. Finally, the *Recommendation Component* integrates the outputs from the previous components to recommend pertinent services, as depicted in Fig. 1.

3.2 Graph Structure Learning Component

Word Vectorization Given that the descriptions of services are in textual format, we employ natural language processing (NLP) techniques to convert them into computable vectors. A prevalent method involves the use of word embeddings[8, 5, 7], which transform words from natural language into spatial vectors. Specifically, for the i-th service s_i with description $D_{s_i} = \{w_1, w_2, \cdots, w_k, \cdots\}$, the initial service embedding can be represented as

$$\mathbf{s}_i^0 = \frac{1}{|D_s|} \sum_{\mathbf{v}_k \in D_s} \mathbf{v}_k \tag{1}$$

where $\mathbf{v}_k$ represents the embedding of the word w_k. We then employ an MLP to enhance embeddings' expressiveness and reduce its dimension, i.e.

$$\mathbf{s}_i = \mathrm{MLP}(\mathbf{s}_i^0) \tag{2}$$

Similarly, we apply the above approach to each mashup. After processing, we have the embedding $\mathcal{X}_s = [\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_n]^\top$ of the services and the embedding $\mathcal{X}_m = [\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_n]^\top$ of the mashup.

API Co-occurrence Graph On the condition that the embeddings are already available, we employ a mashup embedding alongside the invocation relationship between mashups and services to construct an API co-occurrence graph, as is common in existing studies [7, 2]. For a mashup m and its corresponding services $S = \{s_0, s_1, s_2, \ldots, s_n\}$ that it invokes, we form a **fully connected graph** $\hat{G}_c = (\hat{\mathcal{V}}_c, \hat{\mathcal{E}}_c, \hat{\mathcal{X}}_c^e)$ among services S, where $\mathcal{V}_c$ is the node set with size n_v, $\mathcal{E}_c = \{(u, v) | \forall u, v \in \hat{\mathcal{V}}_c\}$ with size n_v^2 is the set of edges, and $\hat{\mathcal{X}}_c^e = [\mathbf{m}, \cdots, \mathbf{m}]^\top$ with size n_v^2 is the feature matrix of edges. Thus, if two services are connected, it implies that they are both invoked by the same mashup. We intentionally designed it to be simple and straightforward, ensuring that it does not require extensive computational resources while still embedding as much information as possible. Since inputs in actual processes are typically processed in batches, the input for each batch can be depicted as a collection of mashups $M = \{m_0, m_1, m_2, \ldots, m_n\}$. For each batch M, every mashup $m \in M$ undergoes the aforementioned subprocess and subsequently merges. The merged graph incorporates all nodes and edges from the subgraphs. For duplicate edges, the edge attributes are averaged pooling. The batch size significantly influences how many mashups are merged concurrently, which in turn substantially affects the final test outcomes. In general, for a batch with n samples, the API co-occurrence graph can be expressed as $G_c = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{X}_c)$, where $\mathcal{V}_c = \hat{\mathcal{V}}_{c_1} \cup \hat{\mathcal{V}}_{c_2} \cup \cdots \cup \hat{\mathcal{V}}_{c_n}$ is the set of service nodes, $\mathcal{E}_c = \hat{\mathcal{E}}_{c_1} \cup \hat{\mathcal{E}}_{c_2} \cup \cdots \cup \hat{\mathcal{E}}_{c_n}$ is the set of edges, and $\mathcal{X}_c^e = [\hat{\mathcal{X}}_{c_1}^{e\top} | \hat{\mathcal{X}}_{c_2}^{e\top} | \cdots | \hat{\mathcal{X}}_{c_n}^{e\top}]^\top$ representing the edge feature matrix.

API Relation Graph We use **graph structure learning** to build another graph and use it to optimize the API co-occurrence graph $G_c = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{X}_c)$

obtained in the previous paragraph. The input of the graph structure learner is the service embedding $\mathcal{X}_s = [\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_n]^\top$.

In the graph structure learner, we construct a forward propagation network that integrates an attention mechanism with learnable weights $\mathbf{w}$. Each layer of the attention mechanism, denoted as *Atten*, can be formulated as follows:

$$\mathbf{s}_i' = \mathrm{ReLU}(\mathbf{s}_i \cdot diag(\mathbf{w})) \tag{3}$$

where $\mathbf{s_i}$ is the embedding of service s_i, $\mathbf{w}$ is the learnable weight vector, and $diag(\mathbf{w})$ represents the diagonal matrix that places $\mathbf{w}$ on the diagonal. ReLU is used to ensure that the output is non-negative. The purpose of this design is to enable the API feature vectors to learn an attention weight that represents each of their dimensions. By adjusting these attention weights, we can obtain a feature vector representation that better aligns with the downstream task.

Then, we calculate the similarity between different API services using a similarity algorithm. There are many options for this, and for convenience, we chose cosine similarity. The formula is as follows:

$$\cos(\mathbf{s}_i', \mathbf{s}_j') = \frac{\mathbf{s}_i' \odot \mathbf{s}_j'}{\|\mathbf{s}_i'\| \|\mathbf{s}_j'\|} \tag{4}$$

Here, $\odot$ represents the dot product of two vectors, while$\|\cdot\|$denote the Euclidean norms (magnitudes). Next, we apply the K-Nearest Neighbors (KNN) algorithm to identify the top-k APIs that are most similar to each API. By doing this, we obtain the API relation graph $G_l = (\mathcal{V}_l, \mathcal{E}_l)$ that illustrates the quantified similarity between APIs. The $\mathcal{V}_l$ is the service nodes and the weights of edges $\mathcal{E}_l$ in this graph represent the degree of similarity between the APIs at the edge endpoints. It is important to note that the graph G_l generated here contains no feature.

Mix Graph To integrate the API co-occurrence graph G_c and the API relation graph G_l, we developed a Mix function. This function is designed based on the intuition that: For mashup creation, we should recommend the API with the highest predicted probability from each category, rather than recommending multiple APIs from the same category even if their predicted probabilities are very high.

As detailed in Fig.2, different colored spheres represent different APIs, and the numbers on the spheres represent their categories. We abstractly consider that APIs with high similarity belong to the same category. APIs within the same category can perform similar functions. We constructed the category information of APIs through graph structure learning and removed the call information (edges) of APIs within the same category in the API co-occurrence graph to achieve "denoising". Given $G_c = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{X}_c)$ and $G_l = (\mathcal{V}_l, \mathcal{E}_l)$, the mix graph $G_f = (\mathcal{V}_f, \mathcal{E}_f, \mathcal{X}_f)$ is obtained as follows:

$$\mathcal{E}_f = \mathcal{E}_c - (\mathcal{E}_c \cap \mathcal{E}_l) \tag{5}$$

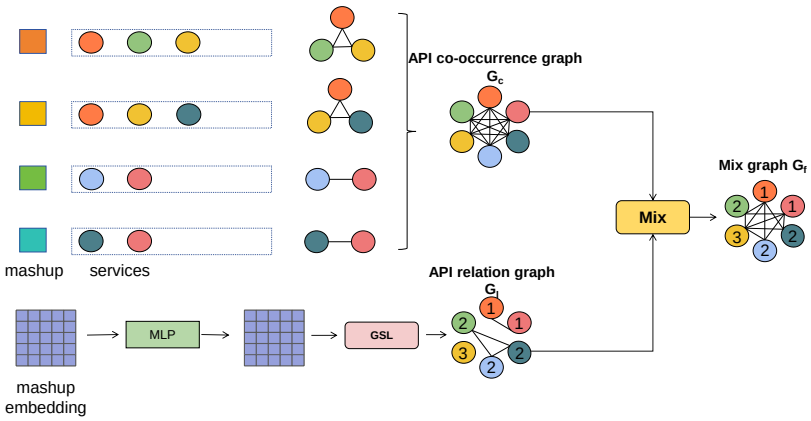


Fig. 2: Illustration of the Mix functions. The numbers on the APIs represent categories.

$$\mathcal{V}_f = \mathcal{V}_c \tag{6}$$

and $\mathcal{X}_f$ is the corresponding edge feature of the remaining edges.

Service Embeddings We assign the service embeddings $\mathcal{X}_s$ to the mix graph G_f as the node feature matrix. So far, we have obtained an optimized graph that contains both node features and edge features. Since we cannot directly process graph data, we need a component to aggregate these features. The GNNs (Graph Neural Networks) can be used to aggregate feature vectors and enhance node representations. It should be noted that there are many options for GNNs here, such as GCN [15] , GAT [13] , etc. We used GraphTranformer [11], as it can be easily extended to support edge features (mashup embeddings) as additional input. For a service $\mathbf{s}_i \in \mathcal{V}_f$, it goes through the following process to update.

$$\mathbf{s}_i = \mathbf{W}_s \mathbf{s}_i + \sum_{s_j \in \mathcal{N}_i} \alpha_{i,j} \mathbf{W}_v \mathbf{s}_j \tag{7}$$

where $\mathbf{W}_s$ and $\mathbf{W}_v$ are learnable parameter weights, $\mathcal{N}_i$ is the neighbor set of s_i, and $a_{i,j}$ is the attention coefficient. Here, the output of GNNs is a series of aggregated embeddings, and we represent these as $\mathbf{H}_0$.

We maintain an aggregate feature $\mathbf{H}'$ with the same dimensions as $\mathbf{H}_0$ continuously during training. For every training step, we combine the features $\mathbf{H}_0$ and $\mathbf{H}'$ by an MLP. $\mathbf{H}'$ is initialed to zero and updated by assignment of $\mathbf{H}$. This process can be represented as follows:

$$\mathbf{H} = \text{MLP}(\mathbf{H}_0, \mathbf{H}') \tag{8}$$

$$\mathbf{H}' \leftarrow \mathbf{H} \tag{9}$$

where $\leftarrow$ means assignment.

3.3 Mashup Transformation Component

As the mashup embedding $\mathcal{X}_m = [\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_n]^\top$ in Section 3.2 is available, we also need to enhance it like that of services. We input the mashup embedding into the graph structure learner, which is detailed in Section 3.2. In this module, we learn the relation graph of mashups, which can be represented as $G_m = (\mathcal{V}_m, \mathcal{E}_m, \mathcal{X}_m)$, where $\mathcal{V}_m$ is the set of mashup nodes, $\mathcal{E}_m$ is the set of edge, $\mathcal{X}_m = [\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_n]^\top$ represents the node feature matrix and $\mathbf{m}_i$ is the feature vector of node m_i. In this graph, mashups with similar embeddings are connected by edges. Leveraging the principles of message passing and aggregation inherent to Graph Neural Networks (GNNs), we dynamically adjust the embedding of each mashup by aggregating the embeddings of its neighboring mashups as follows,

$$\hat{\mathbf{m}}_i = \alpha \cdot \mathbf{m}_i + \frac{1-\alpha}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \mathbf{m}_j \tag{10}$$

where α is a hyperparameter , and $\mathcal{N}_i$ indicates the set of the neighbors of the i-th mashup. For ease of representation, we name the set of mashup embeddings obtained here as $\mathbf{M}$. The intuition behind this component is that a mashup tends to call the same APIs as the mashups that are highly similar. We manually enhance this "similarity," making mashups with initially high similarity even more similar, and those with low similarity even more dissimilar. We demonstrated the improvement brought by this component in Section 5.3.

3.4 Recommendation Component

At this point, we have the aggregate feature vector $\mathbf{H}$ formed in Section 3.2 for the APIs and the aggregate feature vector $\mathbf{M}$ formed in Section 3.3 for the mashup. We concatenate them into vector $\mathbf{E}$ and we feed vector $\mathbf{E}$ into a final recommendation neural network. Our design here is very intuitive and straightforward.

We use a residual network as the decoder to address the issue of high level of aggregation in vector $\mathbf{E}$. The recommendation network consists of three Residual modules plus an MLP for the final output prediction and in our experiments, we found that using three layers achieves good performance without significantly increasing the model's complexity. The Residual module is very simple, essentially a stack of MLPs, with ReLU function, batch normalization, and residual connection. We make a residual connection for each two-layer MLP, and then control the dimension of the feature through an additional MLP. Finally, the prediction output is aggregated by an MLP of the final result.

We take binary cross entropy as the loss function of GSL-Mash. Here the prediction can be represented as $p_i \in [0, 1]$, the label can be represented as $y_i \in 0, 1$, and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the Sigmoid function, then if N represents the number of mashups in a batch, the loss function of GSL-Mash is:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (y_i \cdot \log(\sigma(p_i)) + (1 - y_i) \cdot \log(1 - \sigma(p_i))) \tag{11}$$

4 Experiment Settings

4.1 Dataset

We evaluate the proposed GSL-Mash model on a real-world ProgrammableWeb dataset, which is also the dataset used in existing service recommendation studies. Specifically, we use the method for processing the data set provided by SMRB [5], with 6,423 mashups and 21,633 APIs remaining. We then removed mashups that contained uncrawled APIs, yielding 4557 mashups, and removed APIs without text description, resulting in 21,495 APIs. Since some APIs have never been used, our experimental tests remove them, with 932 APIs left, all of which have been used at least once.

In the raw data collected, each mashup is characterized by multiple attributes including name, description, related APIs, among others. Similarly, each WebAPI is described by attributes such as name and description. We selected descriptions as the sole metadata to represent both mashups and WebAPIs in our dataset. Regarding dataset splitting, we divided the data into three subsets: a training set, a validation set, and a test set, using a ratio of 7:2:1, respectively.

4.2 Metrics

We choose metrics widely used in existing work to measure the model: precision (P), recall (R), F1 score (F1), Normalized Discounted Cumulative Gain (NDCG).

4.3 Baselines

To evaluate the effectiveness of the model, we selected one heuristic and eight state-of-theart service recommendation methods:

- *Frequent* [5]: This approach always recommends the top k frequently invoked services. Frequent is a heuristic approach.
- *T2L2* [8]: This approach enhance service recommendations by addressing representation heterogeneity.
- *T2L2-W/O-Propagation* [5]:Due to the requirement of T2L2 for chronologically ordered training data, which contrasts with the disordered nature of our dataset. We eliminated the propagation component to prevent the possibility of transmitting erroneous messages.
- *MLP* [5]: It uses two linear layers to handle mashups and features of MLP-based Web APIs to predict the score of matches.
- *MTFM* [14]: This approach presents a multi-model fusion and multi-task learning approach for recommending mashup-oriented web APIs.
- *HCF* [2]: This model based on hypergraph and high order Cooperative Filtering , which captures neighborhood information by neural networks.
- *FREEDOM* [19] : A graph-based recommendation algorithm aggregates multimodal features and uses degree-sensitive edge pruning to reject noisy edges during graph sampling.

Table 1: Performance Comparison of Different Approaches (Mean ± std. Best result are colored in **first**, **second**, **third**)

Model	precision@k		recall@N		F1@N		NDCG@N	
	@5	@10	@5	@10	@5	@10	@5	@10
Frequent	13.36±0.08	8.23±0.13	42.90±1.21	52.85±1.11	20.38±0.14	14.25±0.19	35.38±2.99	32.66±2.38
T2L2	13.26±1.04	8.13±0.38	43.23±2.22	53.08±2.07	20.29±1.42	14.10±0.60	35.51±11.19	32.69±4.28
T2L2-wo	**16.92±0.56**	**9.86±0.34**	**54.43±2.41**	**63.40±2.94**	**25.81±0.79**	**17.06±0.55**	**59.84±4.13**	**49.74±3.16**
MLP	**15.18±1.29**	**8.86±0.90**	**48.35±3.89**	**56.39±4.12**	**23.10±1.72**	**15.31±1.43**	**65.25±1.72**	**51.10±3.64**
MTFM	13.64±0.67	8.28±0.73	42.99±0.71	52.19±3.11	20.70±0.84	14.29±1.20	53.08±3.63	39.96±1.95
HCF	6.43±0.33	3.77±0.09	32.14±1.65	37.69±0.94	10.71±0.55	6.86±0.16	24.69±2.03	26.46±1.69
FREEDOM	9.76±0.39	8.62±0.27	41.97±1.74	45.01±1.65	15.84±0.64	9.84±0.33	34.51±0.92	33.95±0.61
GSL-Mash	**23.72±0.45**	**12.55±0.23**	**74.41±3.61**	**78.70±3.05**	**35.97±0.85**	**21.64±0.37**	**97.30±2.37**	**96.49±2.21**
improv.	**+6.80%** ↑	**+2.69%** ↑	**+19.98%** ↑	**+15.30%**↑	**+10.16%**↑	**+4.58%**↑	**+32.05%**↑	**+45.39%**↑

4.4 Implement Details

We initialize mashup embeddings and service embeddings with BERT pre-trained word embeddings, both of which start with 768 dimensions and become 128 after each passing through an MLP. N in the evaluation metrics is set to 5 and 10. To pursue stable experimental results, we run each model five times, all being equal except for the random seeds, and average the results. All experiments were performed on a GeForce RTX 3090.

5 Experiment Results

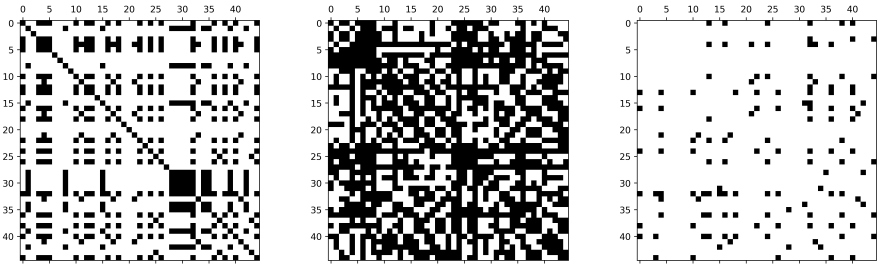
5.1 Overview

In this section, we outline the performance comparison between our proposed GSL-Mash and the baselines method.

The test results presented in the Table 1 indicate that some baseline models do not perform as well as expected. Among baselines, it is noteworthy that the results of HCF and FREEDOM on all four tests are relatively low. Although HCF is also based on graph construction, the collaborative filtering method it uses cannot handle situations with sparse data and limited descriptive information. The FREEDOM algorithm, on the other hand, might be more suitable for handling multimodal data. Another important observation is that many models perform worse than MLP on all four metrics. This, to some extent, implies that much of the current work might be overly complicating network construction. This complexity could result in these models being less effective at extracting features when dealing with real-world service recommendation datasets that involve only brief textual descriptions of mashups and APIs' core functionalities.

Among all the models, the GSL-Mash model we proposed has achieved the best test results. It shows a significant improvement of 10.10% on F1@5 and 4.50% on F1@10 over the second-ranked T2L2-wo. In terms of NDCG@5 and NDCG@10, it scored an impressive 97.30% and 96.49%, far surpassing the second-best scores of 59.83% and 51.10%. To explain the near-perfect NDCG and very high Recall metrics in our experimental results, we randomly printed some prediction results along with their corresponding labels. We found that

in most real-world scenarios, mashups only call 2 to 3 APIs, and GSL-Mash almost always predicts the correct APIs in the top ranks. The Precision metric is not particularly high because the number of correct APIs in the labels is small, and when the number of predicted APIs exceeds the actual number, this metric inevitably decreases. Even so, our Precision metric is still far ahead of other models. This also implies that GSL-Mash has significant untapped potential. It is evident that GSL-Mash significantly outperforms the baseline models, which can be attributed to our advanced combination of graph construction methods and graph structure learning methods. Generally, fully connected graph construction methods can encapsulate rich information, but they also include a lot of noise. However, by simultaneously using graph structure learning to denoise the API co-occurrence graph and enhancing the feature vector representation of mashups, we successfully eliminated the negative impact of fully connected graph construction, achieving outstanding performance.



(a) Adjacency matrix of G_c (b) Adjacency matrix of G_l (c) Adjacency matrix of G_f

Fig. 3: Adjacency matrix during model operation. For readability, we randomly select an adjacency matrix of the APIs that are called in a training step

5.2 Example and Batch Experiment

In this part, we present the adjacency matrix graph of the model in operation (Fig. 3). The graphs are displayed from left to right: the original API co-occurrence graph G_c generated in one step; the API relation graph G_l produced by the graph structure learner in a single step; and the mix graph G_f optimized by the graph structure learner for the final input to the GNN in one step.

From the figure, it is evident that the initial API co-occurrence graph $G_c = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{X}_c)$ is relatively dense, containing numerous noise edges connected by APIs of the same type. Such a complex graph structure can significantly degrade the model's efficiency and performance, a point we will demonstrate in the following subsection. In our model, we employ a graph structure learner to explicitly connect edges between APIs with similar functions. This learner filters out the original API co-occurrence graph G_c, substantially reducing noise and redundant

edges in the graph structure, thereby enhancing the model's performance and efficiency.

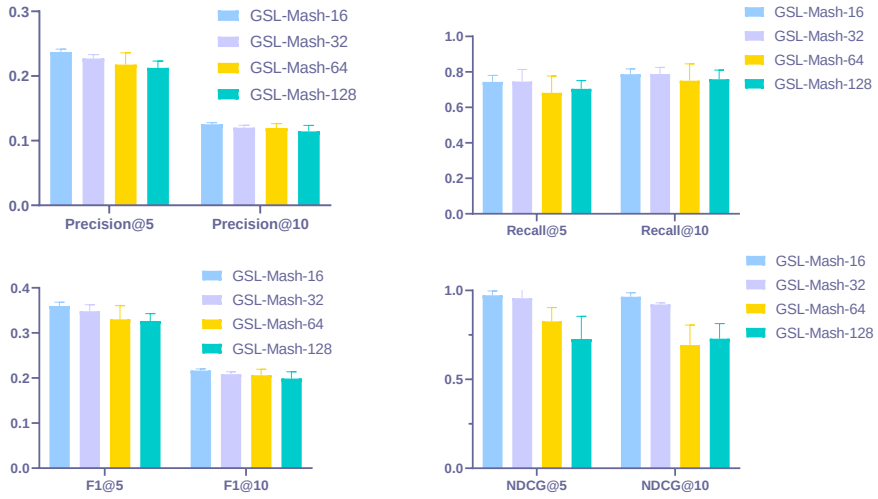


Fig. 4: The performance difference of batch experiments, in which the number indicates the value of batch.

The Fig. 4 illustrates the impact of the hyperparameter batch size on experimental outcomes. The results indicate that a larger batch size leads to increased noise in the API co-occurrence graph $G_c = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{X}_c)$ constructed in Section 3.2, which consequently diminishes the experimental performance.

5.3 Ablation Experiments

We have designed three ablation experiments to validate the contribution of our proposed Graph Structure Learner (GSL) to the experimental outcomes:

- GSL-Mash-m: This variant removes the Mashup Transformation Component that enhance the embedding input by using graph structure learning to learn mashup similarity. That is, Section 3.3 is removed.
- GSL-Mash-s: This variant removes components that learn constraints between services of the same type through the graph structure. That is, (3), (4), (5) and (6) are deleted from Section 3.2.
- GSL-Mash-b: All modules using graph structure learning are removed, which is the synthesis of the removed parts in GSL-Mash-m and GSL-Mash-s.

The performance differences are depicted in Fig. 5. The experimental results reveal that the index of GSL-Mash-m on F1@5 is 4% lower than that of GSL-Mash. This indicates that using graph structure learning to construct similarity

and achieve a weighted average of mashup embeddings essentially allows similar mashups to share API call information, making similar mashups more alike and dissimilar mashups more different, thereby achieving data augmentation. GSL-Mash-s omits the module that utilizes graph structure learning to optimize the API co-occurrence graph $G_c = (\mathcal{V}_c, \mathcal{E}_c, \mathcal{X}_c)$. This resulted in a reduction in the indices of GSL-Mash-s on F1@5 and NDCG@5 by 13.8% and 17.5%, respectively, compared with GSL-Mash, which demonstrates that GSL can reduce noise in the API co-occurrence graph G_c and learn a more precise graph representation that is suitable for downstream recommendation tasks, significantly enhancing service recommendation. The experimental results of GSL-Mash-b and GSL-Mash-s are very similar, and they are also very close to the test results of the MLP. This suggests that simply using a fully connected graph construction method causes the MLP to struggle with the inherent noise. However, by incorporating graph structure learning for denoising and data augmentation, we can not only streamline the amount of graph data obtained to improve efficiency but also make better use of the data to significantly enhance our performance.

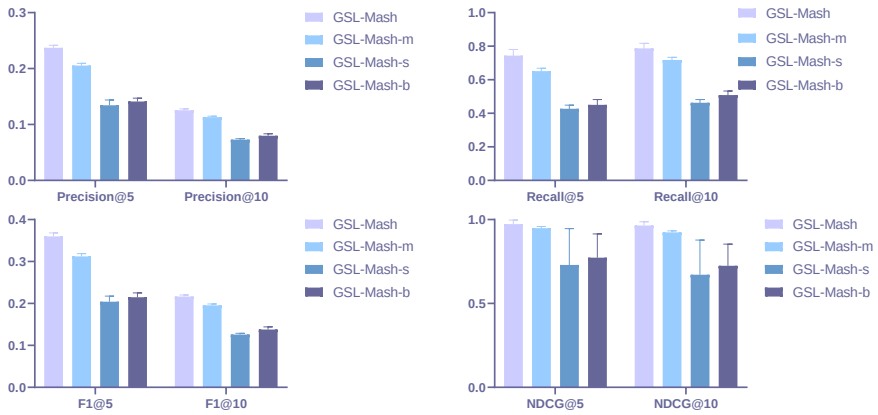


Fig. 5: The performance difference of ablation experiments

6 Conclusion

In this paper, we introduce a deep learning model named GSL-Mash. This model features a module designed to extract an information-rich graph structure from historical interactions between mashups and services. It utilizes graph structure learning to de-noise the API co-occurrence graph and inputs the enhanced features into an MLP-based residual neural network to provide accurate service recommendations. The experimental results demonstrate that the proposed model outperforms several recent service recommendation models. Moving forward, we

plan to further explore the application of graph structure learning in service recommendation and conduct additional experiments to evaluate its effectiveness.

Acknowledgements

The research in this paper is partially supported by the National Key Research and Development Program of China (No.2021YFF0900900), Key Research and Development Program of Heilongjiang Providence (2022ZX01A28), the Postdoctoral Fellowship Program of CPSF (GZC20242204), and the Postdoctoral Science Foundation of Heilongjiang Province, China (LBH-Z23161).

References

1. Boulakbech, M., Messai, N., Sam, Y.: Deep learning model for personalized web service recommendations using attention mechanism. In: ICSOC (2023)
2. Chen, L., Mei, N., He, Y.: High-order collaborative filtering for third-party library recommendation. In: ICWS (2023)
3. Fan, X., Gong, M., Wu, Y.: Neural gaussian similarity modeling for differential graph structure learning. In: AAAI (2024)
4. He, X., Liao, L., Zhang, H.: Neural collaborative filtering. WWW (2017)
5. Jiang, T., Liu, M., Tu, Z.: Identifying and removing the ghosts of reproducibility in service recommendation research. In: CAiSE (2023)
6. Linden, G., Smith, B., York, J.: Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Computing (2003)
7. Liu, M., Tu, Z., Xu, H.: Dysr: A dynamic graph neural network based service bundle recommendation model for mashup creation. TSC (2023)
8. Liu, M., Zhu, Y., Xu, H.: T2l2: A tiny three linear layers model for service mashup creation. In: ICSOC (2021)
9. Peng, M., Cao, B.Q., Chen, J.: SC-GAT: Web Services Classification Based on Graph Attention Network (2021)
10. Shi, M., Zhuang, Y., Tang, Y.: Web service network embedding based on link prediction and convolutional learning. TSC (2022)
11. Shi, Y., Huang, Z., Wang, W.: Masked label prediction: Unified massage passing model for semi-supervised classification. ArXiv (2020)
12. Tang, M., Xie, F., Lian, S.: Mashup-oriented api recommendation via pre-trained heterogeneous information networks. Information and Software Technology (2024)
13. Velickovic, P., Cucurull, G., Casanova, A.: Graph attention networks. ArXiv (2017)
14. Wu, H., Duan, Y., Yue, K.: Mashup-oriented web api recommendation via multi-model fusion and multi-task learning. TSC (2021)
15. Zhang, S., Tong, H., Xu, J.: Graph convolutional networks: a comprehensive review. Computational Social Networks (2019)
16. Zhang, S., Xiong, Y., Zhang, Y.: Rdgsl: Dynamic graph representation learning with structure learning. In: CIKM (2023)
17. Zhao, W., Wu, Q., Yang, C.: Graphglow: Universal and generalizable structure learning for graph neural networks. In: KDD (2023)
18. Zheng, Z., Ma, H., Lyu, M.R.: Qos-aware web service recommendation by collaborative filtering. TSC (2011)
19. Zhou, X., Shen, Z.: A tale of two graphs: Freezing and denoising graph structures for multimodal recommendation. In: MM (2023)