Session-Based Software Recommendation with Social and Dependency Graph

DENGCHENG YAN, Anhui University, China
TIANYI TANG, Anhui University, China
YIWEN ZHANG∗, Anhui University, China

Reusing mature software packages that have been developed repeatedly can greatly enhance the efficiency and quality of software development. However, with the rapidly growing number of software packages, developers are facing the challenge on technology choices. In this context, software recommendation plays a crucial role in software development. While conventional recommendation models can be applied to software recommendation, regrading to the unique characteristics of software development, there still remains three challenges: 1) developers’ interests are gradually evolving, 2) developer are influenced by their friends, and 3) software packages are influenced by their dependency. Notably, the social influences are dynamic and the dependency influences are attentive. That is, developers may trust different sets of friends at different times and different dependency exhibits different importance. In this paper, we propose a novel software recommendation model, named as Session-based Social and Dependence-aware Recommendation (SSDRec). It integrates recurrent neural network (RNN) and graph attention network (GAT) into a unified framework. This model employs RNN on short session-based data to model developers’ evolving interests. In addition, we extend GAT to Social-Dependency-GAT (SD-GAT) for modeling both dynamic social influences and attentive dependency influences. Extensive experiments are conducted on real-world datasets and the results demonstrate the advantages of our model over state-of-the-art methods for modeling developers’ evolving interests and the two influences.

CCS Concepts: • Computing methodologies → Neural networks; • Software and its engineering → Software libraries and repositories; • Information systems → Social recommendation.

Additional Key Words and Phrases: Software recommendation, Social network, Dependency network, Graph neural network

ACM Reference Format:
Dengcheng Yan, Tianyi Tang, and Yiwen Zhang. 2021. Session-Based Software Recommendation with Social and Dependency Graph. 1, 1 (March 2021), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The rapid development of the Internet has significantly fueled the prosperity of the software industry. According to AppBrain reports, in Q2, 2019 alone, over 2 million software was available on Google Play. Developers often need to develop high-quality software projects in a short time.

∗Corresponding author.

Authors’ addresses: Dengcheng Yan, yanzhou@ahu.edu.cn, Institutes of Physical Science and Information Technology, Anhui University, 111 Jiulong Rd, Hefei, Anhui, China; Tianyi Tang, tangtianyi202012@163.com, School of Computer Science and Technology, Anhui University, 111 Jiulong Rd, Hefei, Anhui, China; Yiwen Zhang, zhangyiwen@ahu.edu.cn, School of Computer Science and Technology, Anhui University, 111 Jiulong Rd, Hefei, Anhui, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

XXXX-XXXX/2021/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

arXiv:2103.06109v1 [cs.IR] 10 Mar 2021
When developing a software project, most developers prefer searching for mature software packages to coding from scratch. As software packages have been developed repeatedly, many bugs as well as deficiencies in the documentation have already been discovered. By reusing these packages, productivity is increased, time-to-market is reduced and overall project quality is improved.

Fortunately, with the wide adoption of open source movement and social collaborative coding paradigm, more and more enterprise-class software is open sourced and developers from all over the world are making contributions to the open source community, resulting in an explosive growth of open source software packages. But this, on the other hand, makes it difficult for developers' technology choices. Thus software recommender systems are urgently needed for the development efficiency and quality guarantee.

In recommender systems, collaborative filtering (e.g., Matrix Factorisation, MF) is a classic technique [1, 2]. Although also effective in software recommendation [3–5], some studies show it ignores two key elements. First, developers' interests are gradually evolving [6]. In the whole lifetime of a software project, new features and requirements emerge all the way and developers need to keep conducting technology choices. Moreover, developers often change to another software project frequently. A developer may focus on Angular one year ago, but is interested in Vue now due to work needs. Furthermore, it’s difficult to capture dynamic interests from long history behaviors [7, 8] (developers’ technology choices one year ago has a little influence now or even interferes with predictions). So we segment developers’ actions into short sessions in which developers’ interests are relatively stable and easy to model. Second, in GitHub, Stackoverflow and other communities, there are social influences among developers [9]. When developing projects, developers tend to choose packages recommended by their friends.

Although these studies can model developers’ dynamic interests or social influences, we argue they ignore that social influences are also dynamic. For example, when a developer develops Vue-based project, he may be influenced more by friends who are proficient in this field. While when developing another project, he will trust others. Further, there is not only the relationship between developers but also the dependency between packages in software recommendation. Although dependency influences have been neglected, it naturally exists in software recommendation and has a huge impact on developers’ technology choices. For example, when a developer has used iView, indicating he would like to develop a project based on iView, iView-admin needs to be recommended. What’s more, we notice that the dependency influences are attentive. In consideration of the requirements of a dependent software and the functions of software packages it depends on, different dependency exhibits different importance.

In this paper, we aim at predicting developers’ technology choices based on their behaviors history. To effectively capture developers’ dynamic interests, we introduce session mentioned above. What’s more, social network and dependency network are also considered to model both dynamic social influences and attentive dependency influences. In summary, we address the problem of session-based software recommendation with social and dependency graph.

Figure 1 illustrates this in two sessions. Developer A’s interests gradually evolve from Field Y to Field X and his technology choice is influenced by both social relationships between developers and dependency relationships between software packages. In session 1, developer A has previously chosen software package 12 which is a dependent package for both package 11 and package 16. However, from the view of attentive software dependency, package 16 is coupled more tightly to package 12 than to package 11. On the other hand, from the view of dynamic social influence, developer A’s technology choice will be influenced more by similar developers in this session, that is developer D and E. Considering all these aspects, we will recommend package 16 to developer A in session 1. Then in session 2, developer A evolves his in interests slightly from Field Y to Field
X, and he will pay more attention to new social and dependency relationships and finally choose package 7.

In summary, the main contributions are listed as follows:

- We jointly combine developers’ evolving interests, attentive software dependency and dynamic social influence and propose a unified software recommendation framework.
- We extend graph attention network to multiple graphs and propose the SD-GAT for mutually learning developers’ and software packages’ representation.
- Extensive experiments are conducted on three real-world datasets and the results demonstrate the performance and effectiveness of our proposed model.

The rest of this survey is organized as follows. We first discuss some related works in §2. Then, we define some key concepts and formalize our problem in §3. Further, our proposed approach is described in §4. Lastly, we presents the experimental results in §5 and conclude the work in §6.

2 RELATED WORK

We will briefly review three aspects of previous researches related to our work: 1) dynamic recommendation that models developers’ dynamic interests; 2) social recommendation that considers influences in social network; 3) software recommendation that recommends software packages to developers.

2.1 Dynamic Recommendation

To model developers’ dynamic interests, earlier works in literature propose their approaches based on Markov Chains [10]. However, Markov Chains can only model developers’ latest behavior, discarding a lot of information. Therefore, recent works exploit recurrent neural network to model developers’ recent behaviors. For example, Manotumruksa et al. [11] considered time information on the basis of MF, and used RNN to model developers’ behaviors to enrich the representations.
the contrary, Dong et al. [12] performed joint optimization to train RNN and MF separately with shared model parameters in a multi-task learning framework. Furthermore, some studies apply RNN on session-based recommendation. For example, Hidasi et al. [8] first proposed a RNN-based approach for session-based recommendations to capture developers’ short-term interests within a session. Later, Wang et al. [13] assumed that a session often serves for different purposes and proposed a mixture-channel model with attention mechanism to detect the purposes of each item. In addition to interactions intra a session, information inter sessions also attracts some researchers. For example, Ruocco et al. [14] exploited two RNNs to process the developers’ current sessions and past sessions separately. Due to the limitation of computing resources, we only exploit RNN to model developers’ current sessions and reserve information inter sessions for future work.

2.2 Social Recommendation
Social network, which usually reveals and utilizes trust relationships [15], has been widely used in recommender systems. Ma et al. [16] constrain connected nodes’ latent factors similar to regularize developer representations. Zhao et al. [17] leveraged friends’ interaction as another positive feedback in Bayesian Personalized Ranking (BPR). Recently, neural network is widely used in various fields and some efforts have applied this technique to social recommendation. For example, Deng et al. [18] utilized Autoencoder to initialize vectors in MF and revamped them with both social trust ensemble and community effect. Further, with the success of Graph Neural Network (GNN) [19], Fan et al. [20] encoded developers on social network and developer-item network through GNN and Guo et al. [21] utilized GNN to model social relations and correlations among item features. As a special case of GNN, Song et al. [7] utilized GAT to model dynamic social influences. Although effective, they only focus on social network and ignore dependency network in software recommendation.

2.3 Software Recommendation
Due to the success of collaborative filtering in recommender systems [1, 2], many studies have introduced it into software recommendation. For instance, Ichii et al. [3] used collaborative filtering to recommend similar software packages to developers. Thung et al. [4] further combined association rule and collaborative filtering to mine deeper relationships between packages. He et al. [5] employed an adaptive weighting mechanism and neighborhood information to neutralize popularity bias in MF, significantly increasing diversity and accuracy of the recommendation results. What’s more, some studies model either developers’ evolving interests or social influences. For example, Guy et al. [9] aggregated developer’s familiarity network and similarity network to recommend social software. Jiang et al. [6] adopt time decay factor and operation behaviors weight to model developers’ dynamic interests in a popular programming platform, Scratch. However, ignoring the data characteristics in technology choice such as dynamic social influences and attentive dependency influences usually prevents the adoption of new methods in software recommendation. To model the two important influences, we propose SD-GAT built on social network and dependency network, where the attention in graphs is unequal and depends on current context.

3 PROBLEM DEFINITION
In this section, we first describe the concept of session and formalize the problem of session-based software recommendation. Then, social network and dependency network are introduced. Finally, we formalize our problem - session-based software recommendation with social and dependency graph. Main notations will be introduced are summarized in table 1.

Definition 1 (Session). A session is a set of items (i.e., software packages in our problem) that are interacted in a certain period (e.g., one week or one month). Let $U$ and $I$ denote the set of developers and software packages, respectively. Assume developer $u$ chooses $n$ packages
Table 1. Summary of Main Notations

| Symbol | Description |
|--------|-------------|
| U      | the set of developers |
| I      | the set of software packages |
| $S^u_T$ | $u$’s session in time step $T$ |
| $N_{u,T}$ | the amount of software packages in $S^u_T$ |
| $N^l(i)$ | $i$’s $l$-hop neighbor in dependency network |
| $N^l(u)$ | $u$’s $l$-hop neighbor in social network |
| $e_i^{(l)}, h_u^{(l)}$ | the $l$-layer representation of $i$ and $u$ |
| $W_D^{(l)}, W_S^{(l)}$ | aggregation weight matrices for dependency and social network in $l$-layer |
| $W_{TD}, W_{TS}$ | transformation matrix in dependency and social network |

$<i_1^u, i_2^u, i_3^u, ..., i_n^u>$ sequentially in a period of time, the $n$ packages compose a session $S^u$, i.e., $S^u = <i_1^u, i_2^u, i_3^u, ..., i_n^u>$. When developing projects, interests of developers change rapidly, resulting in modeling developers’ dynamic interests from all historical behaviors is difficult. Taking this into consideration, we segment whole interaction history into several sessions. In this way, each developer’s interests are relatively stable and easy to model.

**Problem 1 (Session-based software recommendation).** There are two research directions in this problem: next-package(s) recommendation that recommends next one(some) software package(s) in current session and next-session recommendation that predicts information for the next session. In this paper, session-based software recommendation refers to next-package recommendation and others are remained for future. Given a session $S = <i_1, i_2, i_3, ..., i_n>$, the goal of it is to recommend the next software package $i_{n+1}$.

**Definition 2 (Social network and dependency network).** In software recommendation, developers are influenced by their friends. In addition, packages are influenced by their dependency. To take advantage of these two influences, we first assume developer $u$’s friends $N(u)$ to construct social network. Notice that this network is dynamic and each node consists of a session (we can generalize it to multiple sessions but this is very resource intensive and single session is effective enough). For a target node $u$ and session $S^u_T$, each neighbor node $f$ in $N(u)$ uses his session right before time step $T$, i.e., $S^f_{T-1}$. Then we assume software package $i$’s dependency $N(i)$ to construct dependency network. In fact, for each developer $u$, we obtain all packages’ dependency, i.e., construct complete dependency network.

**Problem 2 (Session-based software recommendation with social and dependency network).** In our problem, each developer $u$ interacts with a temporally ordered list of sessions $L^u = <S^u_1, S^u_2, S^u_3, ..., S^u_{|L^u|}>$, where $S^u_T$ is the $t$th session of developer $u$ and $|L^u|$ is the length of list. In time step $T$, we set $S^u_T = <i_1^u_{T,1}, i_2^u_{T,2}, i_3^u_{T,3}, ..., i_{N_{u,T}}^u_{T,N_{u,T}}>$ as an input, where $N_{u,T}$ is the length of this session. Note that both $|L|$ and $N_{u,T}$ are variable, and $T$ denotes arbitrary time step. To model dynamic social influences and attentive dependency influence, as another input, we utilize $u$’s social network and complete dependency network. As output, we predict the next package that $u$ is most likely to choose.
4 METHODOLOGIES

In this section, we propose a novel multi-graph model Session-based Social and Dependency-aware Recommendation, which jointly models developers’ evolving interests, attentive software dependency and dynamic social influences.

SSDRec outlined in Figure 2 is composed of three layers, namely, input layer, embedding propagation layer and prediction layer. First (§4.1), the input layer produces the representation of the nodes in social network and dependency network. On the one hand, this layer utilizes embedding to obtain representations of software packages and static interests of friends. On the other hand, it uses RNN to capture the dynamic interests of developers and friends from their sessions. Then (§4.2), the propagation layer propagates influences, i.e., representations in the two networks. Specially, a SD-GAT is used to weight the dynamic social influences and attentive dependency influences. At the final step (§4.3), the prediction layer combines the two influences and produces recommendation.

4.1 Input Layer

4.1.1 Represent software package and Its dependency. Following MF, we utilize embedding to map the software package $i$’s ID to a $E$-dimension dense real-value vector $e_i^{(0)}$ and obtain the representation of $i$’s dependency $e_1^{(0)}, e_2^{(0)}, ..., e_{|\mathcal{N}(i)|}^{(0)}$ the same way.
4.1.2 Represent Developer. To model the developer $u$’s dynamic preferences, we employ a RNN to capture the sequential associations in current session. Given a session $S^u_T = \langle i^u_{T,1}, i^u_{T,2}, i^u_{T,3}, ..., i^u_{T,N_u,T} \rangle$, the RNN recursively combines the representations of all tokens as follow:

$$h_t = \text{tanh}(W_i h^u_{T,t} + W_h h_{t-1} + b)$$

where $h_t \in \mathbb{R}^H$ is the $H$-dimension recurrent hidden output at time $t$ and $W_i \in \mathbb{R}^{H \times E}, W_h \in \mathbb{R}^{H \times H}, b \in \mathbb{R}^H$ represent weight and bias parameters to be learnt. As shown in Figure 2, we represent developer’s interests with the final output of RNN $h_n$. While SSDRec adopts LSTM units in practice, it is flexible to plug in other recurrent units, e.g., GRU.

4.1.3 Represent Friends. We consider friends’ two interests: dynamic interests that represent recent interests and static interests that represent long-term and average interests.

Dynamic interests: For a target developer $u$, given his session $S^u_T$ in time step $T$, his friends’ sessions in $T-1$ are selected as their visible actions. After adopting friend $f$’s session $S^f_{T-1}$, we use RNN to model $f$’s dynamic interest $s^d_f$:

$$s^d_f = \text{RNN}(S^f_{T-1})$$

Notice that we reuse RNN representing developers to represent friends by enforcing the two RNNs share the same weight parameters.

Static interests: Similar to §4.1.1, for each friend $f$, we use embedding to model the static interests $s^s_f \in \mathbb{R}^E$. 1.

Finally, we represent friends by concatenating friends’ dynamic and static interests with a non-linear transformation:

$$h_f = \text{ReLU}(W_f(s^d_f \parallel s^s_f))$$

where $\parallel$ is the concatenation operation of two vectors and $W_f$ is the transformation matrix.

4.2 Propagation Layer

Recent works like [19–21] have demonstrated the convincing performance of performing GNN on recommender systems. Despite their effectiveness, we argue that they can’t model dynamic social influences in social network and attentive dependency influences in dependency network. To propagate the influences in the two networks, we extend graph attention network to multiple graphs and propose SD-GAT, which is shown in Fig. 2. When the type of the target node is "package", the left dependency-oriented GAT will be used to propagate attentive dependency influences. Otherwise, the right social-oriented GAT is adopted to propagate dynamic social influences. These two parts work on two network and adopt different transformation functions. In this way, we can model both dynamic social influences and attentive dependency influences.

4.2.1 Graph construction. For each software package, we build a graph that includes $|N(i)|+1$ nodes, where nodes correspond to that software package and its dependency. In this graph, software package node $i$ and its neighbors $N(i)$ is directly set to $e^{(0)}_i$ and $e^{(0)}_1, e^{(0)}_2, ..., e^{(0)}_{|N(i)|}$. For each developer, we also build a graph that includes $|N(u)|+1$ nodes. Similarly, we use developer $u$’s dynamic preferences $h_u$ and his friend $f$’s preference $h_f$ to represent node $u$ and its neighbor $f$, i.e., $h^{(0)}_u = h_u, h^{(0)}_f = h_f, f \in N(u)$. It should be noted that, unlike the software package node representation remaining unchanged in $S^u_T$, the representation of developer node will be updated whenever he chooses a new software package during time step $T$.

1We simply set the dimension of developer representations equal to package representations, but our model can be modified to distinguish the dimensions easily.
4.2.2 Message Propagation. Intuitively, the dependency that is dependent on a software package can be treated as the package’s features; analogously, friends provide evidence of developer’s interest. We build upon this basis to propagate item representations between software package and its dependency as well as developer representations between developer and friends. Although graph convolutional networks [22] is widely used to propagate information in graph, it treats all neighbors equally using method like symmetric normalized Laplacian. Instead, we employ graph attention network [23], which use attention mechanism to distinguish the influences in the two graphs. There are two major operations in graph attention network: message construction and message aggregation.

Message construction: For software package \(i\), we use attention mechanism to measure the influences of its dependency. Firstly, we calculate the attention score \(a_{ij}^{(0)}\) between the target node representation \(e_i^{(0)}\) and all of its one-hop neighbors’ representations \(e_1^{(0)}, e_2^{(0)}, \ldots, e_{|N(i)|}^{(0)}\):

\[
a_{ij}^{(0)} = f(e_i^{(0)}, e_j^{(0)}), j \in N(i) \cup i
\]

(4)

In particular, \(f(e_i^{(0)}, e_j^{(0)}) = e_i^{(0)\top} e_j^{(0)}\) represents inner production, a similarity function between two elements. Notice that we also use self-attention to preserve \(i'\) representation here. After adopt attention score, we calculate attention weight with softmax function:

\[
a_{ij}^{(0)} = \frac{\exp(a_{ij}^{(0)})}{\sum_{k \in N(i) \cup i} \exp(a_{ik}^{(0)})}, j \in N(i)
\]

(5)

intuitively, \(a_{ij}^{(0)}\) is the level of influence of dependency \(j\) on software package \(i\) (conditioned on layer 0). Finally, we define message from \(j\) to \(i\) as:

\[
m_{i \rightarrow j}^{(0)} = a_{ij}^{(0)} e_j^{(0)}, j \in N(i) \cup i
\]

(6)

Analogously, for developer \(u\), the message that one-hop neighbors propagate can be represented by:

\[
m_{u \rightarrow f}^{(0)} = a_{uf}^{(0)} h_f^{(0)}, f \in N(u) \cup u
\]

(7)

where \(a_{uf}^{(0)} = \text{Softmax}(f(h_u^{(0)}, h_f^{(0)}))\).

Message Aggregation: With the message defined as above, we then combine the target node and its neighbors to update its representation. The first-layer software package representation \(e_i^{(1)}\) and developer representation \(h_u^{(1)}\) are defined as follows:

\[
e_i^{(1)} = \text{ReLU}(W_D^{(1)} \cdot \sum_{j \in N(i) \cup i} m_{i \rightarrow j}^{(0)})
\]

(8)

\[
h_u^{(1)} = \text{ReLU}(W_S^{(1)} \cdot \sum_{f \in N(u) \cup u} m_{u \rightarrow f}^{(0)})
\]

(9)

where \(W_D^{(1)} \in \mathbb{R}^{E \times E}\) and \(W_S^{(1)} \in \mathbb{R}^{H \times H}\) denote the aggregation weight matrices for dependency network and social network, respectively.

4.2.3 High-order Propagation. We can further obtain the high-order connectivity information of each node by stacking this attention layer. Specifically, when we stack \(l\) layers, the representation of software package \(i\) and developer \(u\) are recursively formulated as:

\[
e_i^{(l)} = \text{ReLU}(W_D^{(l)} \cdot \sum_{j \in N(i) \cup i} a_{ij}^{(l-1)} e_j^{(l-1)})
\]

(10)
\[ h_u^{(l)} = \text{ReLU}(W_S^{(l)} \cdot \sum_{f \in N(u) \cup u} \alpha_f^{(l-1)} h_f^{(l-1)}) \]  

(11) 

where \( W_D \in \mathbb{R}^{E \times E} \) and \( W_S \in \mathbb{R}^{H \times H} \) are the trainable transformation matrices in layer \( l \), the \( \alpha_f^{(l-1)} \) guiding message propagation in the \( l - 1 \) layer is defined as follows:

\[ \alpha_f^{(l-1)} = \text{Softmax} (f(e_i^{(l-1)}, e_j^{(l-1)})), j \in N(i) \cup i \]  

(12) 

\[ \alpha_f^{(l-1)} = \text{Softmax} (f(h_u^{(l-1)}, h_f^{(l-1)})), f \in N(u) \cup u \]  

(13) 

4.3 Prediction Layer

After propagating with \( L \) layers, we obtain the merged representation of software package \( i \), namely \( e_i^{(L)} \). Since a package’s features depends on both its origin function and attentive dependency influences, we combine them with a fully-connected layer to constitute the final representation \( e_i \in \mathbb{R}^E \):

\[ e_i = W_{TD} \cdot (e_i^{(0)} \parallel e_i^{(L)}) \]  

(14) 

where \( W_{TD} \in \mathbb{R}^{E \times 2E} \) is a linear transformation matrix in dependency network. For developer \( u \), the formulations are similar for combine developers’ evolving interests and dynamic social influences:

\[ h_u = W_{TS} \cdot (h_u^{(0)} \parallel h_u^{(L)}) \]  

(15) 

where \( W_{TS} \in \mathbb{R}^{E \times 2H} \) is transformation matrix in social network, and \( h_u \in \mathbb{R}^E \) indicates developer \( u \)’s current interest.

Finally, we conduct the softmax to estimate the probability that developer \( u \) will choose software package \( i \):

\[ p(i | S^u_T) = \frac{\exp(h_u^T e_i)}{\sum_{z=1}^{|l|} \exp(h_u^T e_z)} \]  

(16) 

where \(|l|\) is total number of packages and \( T \) represent transposition.

4.4 Train

To learn model parameters, we maximize the log-likelihood of the observed packages in all sessions:

\[ \sum_{u \in U} \sum_{t=2}^{T-1} \sum_{n=1}^{N_{u,t-1}} \log(p(i^u_t | i^u_{t,1}, \ldots, i^u_{t,n})) \]  

(17) 

where \( U \) denotes all developers. Notice that SSDRec model utilizes an iterative method to update developer representations, which can increase the size of training set and reduce the cost of updating (some state-of-the-art approaches, such as NCF [2], need to build a new developer-package matrix for any new interaction).

4.5 Time Complexity Analysis

Assume batch size \( B \) and we calculate the time complexity of a batch. In input layer, embedding and RNN is the main operation. According to [24], the complexity of an LSTM unit is \( O(H^2 + H \times E) \). Thus, input layer has computational complexity \( O(B \times N^2 \times (H^2 + H \times E)) \), where \( N \) denotes the length of a session. In propagation layer, we first compute the complexity of each layer for a session. For dependency network, complexity of \( l \)-th layer is \( O(\sum_{i=1}^{|l|} |N^l(i)| \times E^2) \). For social network, the complexity is \( O(\sum_{r=1}^L (\sum_{i=1}^{|r|} |N^r(i)| \times E^2 + B \times N \times \sum_{r=1}^L |N^r(i)| \times H^2) \). As we only use the dependency network once in a batch, the overall complexity for propagation layer is \( O(\sum_{i=1}^L (\sum_{i=1}^{|l|} |N^l(i)| \times E^2 + B \times N \times \sum_{r=1}^L |N^r(i)| \times H^2)) \). In prediction layer, the time complexity of transformation and softmax is
Therefore, the overall complexity of SSDRec can be simplified to
\[ O(B \times N^2 \times (H^2 + H \times E) + \sum_{i=1}^{L} \sum_{l=1}^{|I(i)|} N_l(i) \times E^2 + B \times N \times \sum_{r=1}^{L} \sum_{i=1}^{|N^r(i)|} |N^r(i)| \times H^2)) \].

5 EXPERIMENTS

In this section, we conduct experiments to answer the following research questions:

- RQ1 Does SSDRec outperform state-of-the-art methods on all experimental settings?
- RQ2 Do the components of the SSDRec enhance the effectiveness by (a) modeling dynamic social influences and attentive dependency influences, (b) capturing friends’ dynamic and static interests.
- RQ3 How do some hyper-parameters (e.g., neighborhood sample sizes, segmentation strategies of session) affect the performance of SSDRec?

In the remainder of this section, we will first describe the experimental settings (§5.1) and then answer the above three research questions(§5.2, §5.3, §5.4). Final, some illustrative examples is given (§5.5).

5.1 Experimental Setup

5.1.1 Datasets. We conduct experiments mainly on GitHub [25], a software hosting platform and open source cooperative community specially designed for programmers. In GitHub, developers can watch software packages and the behaviors are segmented into sessions. We also obtain follow relationship as social network. However, there is no obvious dependency between packages on GitHub. Thus, we combine GitHub and Libraries, an online search repository that contains dependency between software packages, to construct a new dataset. Then, to make the experimental results more accurate and reliable, we divide the new dataset according to language. Finally, we choose three biggest datasets as follows: 1) PHP, a server-side scripting language, is commonly used for website programming; 2) Ruby, a completely object-oriented language, is suitable for rapid development; 3) JavaScript, a very popular language with cross-platform features.

5.1.2 Preprocessing. To ensure that each developer in datasets focuses on social network, we filter out developers with fewer than \( m \) friends. Then, to cater sufficient information about each friend and software package, we ensure that each developer is followed by at least \( n \) times and each package is watched by at least \( k \) developers. Due to different size of datasets, we set \( m = 20, 50, 100; n = 50, 100, 200; k = 20, 50, 50 \) for PHP, Ruby and JavaScript respectively. Additionally, we segment all datasets into week-long sessions and filter out sessions with fewer than 2 packages or more than 30 packages following [7]. To split datasets for train, validation and test, for each developer, we reserve about two years (104 weeks)’s sessions and randomly split them into validation/test sets. Note that when splitting dataset, we ensure all packages in validation/test sets appear in train set. Descriptive statistics for all datasets after preprocessed are shown in Table 2.

5.1.3 Parameter Settings. We implement our model based on TensorFlow [26]. Adam [27] is chosen as optimizer and optimization parameters are set as suggested in [26]. In this environment, our model is trained in 20 epochs of batch size 200 and dropout [28] with 0.2 is applied. For all models, the dimensions of package representations and hidden vectors are fixed to 100. Both in social and dependency networks, we set number of layers \( L = 2 \) and sample 10 one-hop neighbors and 5 two-hop neighbors as recommended in [29].

5.1.4 Baselines. We compare SSDRec to the following baselines:

- BPR [1]: a classical MF-based method optimized with a ranking objective.
Table 2. Descriptive statistics of our three data sets.

| Dataset | PHP    | Ruby   | JavaScript |
|---------|--------|--------|------------|
| # Developers | 7,095  | 5,935  | 8,144      |
| # Software packages | 1,402  | 1,487  | 2,282      |
| # Events | 118,256 | 116,832 | 436,409    |
| Avg. friends/developer | 8.59   | 13.05  | 16.24      |
| Avg. dependency/packages | 4.56   | 5.09   | 10.09      |
| Avg. session length | 2.95   | 3.13   | 3.25       |

- NCF [2]: uses neural network instead of inner product to model relationship between developers and items of MF.
- RNN [8]: captures developers’ session-level dynamic interests with RNN.
- DGRec [7]: a state-of-the-art model for session-based social recommendation. It utilizes RNN and GAT to model dynamic interests and dynamic social influences.

5.1.5 Evaluation Metrics. For all models, their recommendation effectiveness is evaluated by two well-known metrics: HitRate(HR)@K and NormalizedDiscountedCumulativeGain(NDCG)@K. HitRate@K emphasizes the accuracy of prediction, measures the proportion of watched packages in K-length truncated ranked list. It can be formulated as:

$$HR@K = \frac{\sum_{u \in U} |R_K(u) \cap T(u)|}{|U|}$$

(18)  

where $R_K(u)$ is top-K recommended packages, $T(u)$ is the software packages developers actually watch.

NDCG@K emphasizes the sequence of recommendation, measures the position watched packages are placed in. It can be formulated as:

$$NDCG@K = \frac{1}{|U|} \sum_{u \in U} \frac{1}{\log_2(1 + pos(T(u)))}$$

(19)

where $pos(T(u))$ represents the rank of watched packages. It’s obvious that the higher position that positive packages are placed in, the higher NDCG@K is. Note that because developers don’t have the patience to flip through the entire list, NDCG@K is equal to 0 when $pos(T(u)) > K$.

5.2 Performance Comparison(RQ1)

Table 3 shows the comparison between five recommendation methods when generating top-K recommendation. It can be seen from the table that: 1) SSDRec achieves the best performance on three datasets by considering developers’ evolving interests, dynamic social influences and attentive dependency influences in software recommendation. 2) The models based on RNN have much higher HR and NDCG than other methods(on average, the worst RNN-based model RNN outperforms the best conventional model NCF by 97.17%). The reason may be that the interests of developers in the computer industry change frequently, and the dynamic interests of developers have much impact on their behaviors. 3) The prediction accuracy of DGRec and SSDRec that consider dynamic social influences are generally high, which proves that modeling such influences is necessary.
Table 3. Performance in terms of HR and NDCG between various approaches. The best performing result is highlighted in bold and the improve rate of best result over second best result is also be shown.

| Dataset | Model | HR@K(%) | NDCG@K(%) |
|---------|-------|---------|-----------|
| PHP     |       |         |           |
|         | BPR   | 3.33    | 6.68      | 15.33     | 1.36 | 2.20 | 3.90 |
|         | NCF   | 4.59    | 8.09      | 17.77     | 2.10 | 2.98 | 4.88 |
|         | RNN   | 9.72    | 16.14     | 28.80     | 4.64 | 6.31 | 8.85 |
|         | DGRec | 11.07   | 17.29     | 29.64     | 5.37 | 7.04 | 9.55 |
|         | SSDRec | **11.95** | **18.93** | **31.13** | **5.88** | **7.74** | **10.22** |
| Ruby    |       |         |           |
|         | BPR   | 2.07    | 3.81      | 9.74      | 0.93 | 1.36 | 2.52 |
|         | NCF   | 2.91    | 5.39      | 12.61     | 1.37 | 1.98 | 3.39 |
|         | RNN   | 6.24    | 9.79      | 19.04     | 3.23 | 4.17 | 6.01 |
|         | DGRec | 7.00    | 10.86     | 20.73     | 3.68 | 4.70 | 6.68 |
|         | SSDRec | **7.63** | **12.05** | **21.98** | **4.01** | **5.19** | **7.17** |
| JavaScript |       |         |           |
|         | BPR   | 2.12    | 3.86      | 8.23      | 0.95 | 1.38 | 2.24 |
|         | NCF   | 2.33    | 3.98      | 8.97      | 1.08 | 1.49 | 2.47 |
|         | RNN   | 4.57    | 7.72      | 15.00     | 2.17 | 2.99 | 4.45 |
|         | DGRec | 5.26    | 8.87      | 17.58     | 2.53 | 3.47 | 5.22 |
|         | SSDRec | **5.70** | **9.73** | **18.47** | **2.65** | **3.73** | **5.48** |

Table 4. Variants of SSDRec that only consider one network. SSDRec-social ignores the dependency network and SSDRec-dependency ignores the social network. Note that when ignoring the dependency network, SSDRec-social is identical to DGRec.(Here the function of $W_T$ transforms the dimension of developer from $H$ to $E$)

| Variants         | Modification                        |
|------------------|-------------------------------------|
| SSDRec-social    | Eq.(14) → $e_i = e_i^{(0)}$         |
| SSDRec-dependency| Eq.(15) → $h_u = W_T \cdot h_u^{(0)}$|

5.3 Ablation Studies(RQ2)

To demonstrate the usefulness of different components, we compare the performance of variants of SSDRec.

5.3.1 Effect of social network and dependency network(RQ2(a)). SSDRec utilizes SD-GAT to model dynamic social influences and attentive dependency influences. To study the contribution of each GAT, we modify SSDRec by masking corresponding network. Table 4 shows the details of variants in this part. We compare the performance of the two variants with SSDRec and report results in Table 5. It’s obvious that SSDRec significantly beats SSDRec-social and SSDRec-dependency, proving that our model benefits from both social network and dependency network.

5.3.2 Effect of dynamic interests and static interests(RQ2(b)). Dynamic and static interests are two main features of developers. Table 6 shows the effect of two interests respectively. We can observe
Table 5. Performance of SSDRec and its variants ignoring social or dependency network.

| Dataset | Model            | HR@K(%) 10 | 20 | 50 | NDCG@K(%) 10 | 20 | 50 |
|---------|------------------|------------|----|----|---------------|----|----|
| PHP     | SSDRec-social    | 11.07      | 17.29 | 29.64 | 5.37         | 7.04 | 9.55 |
|         | SSDRec-depndence | 11.07      | 17.70 | 29.89 | 5.30         | 7.05 | 9.53 |
|         | SSDRec           | **11.95**  | **18.93** | **31.13** | **5.88** | **7.74** | **10.22** |
| Ruby    | SSDRec-social    | 7.00       | 10.86 | 20.73 | 3.68         | 4.70 | 6.68 |
|         | SSDRec-depndence | 7.25       | 10.69 | 19.76 | 3.74         | 4.65 | 6.44 |
|         | SSDRec           | **7.63**   | **12.05** | **21.98** | **4.01** | **5.19** | **7.17** |
| JavaScript | SSDRec-social    | 5.26       | 8.87 | 17.58 | 2.53         | 3.47 | 5.22 |
|         | SSDRec-depndence | 4.68       | 8.04 | 15.73 | 2.28         | 3.14 | 4.69 |
|         | SSDRec           | **5.70**   | **9.73** | **18.47** | **2.65** | **3.73** | **5.48** |

Table 6. Performance of SSDRec and its variants only with dynamic interests (SSDRec-dynamic) or static interests (SSDRec-static).

| Dataset | Model            | HR@K(%) 10 | 20 | 50 | NDCG@K(%) 10 | 20 | 50 |
|---------|------------------|------------|----|----|---------------|----|----|
| PHP     | SSDRec-dynamic   | 11.37      | 18.04 | 30.83 | 5.68         | 7.46 | 10.05 |
|         | SSDRec-static    | 11.03      | 17.62 | 30.00 | 5.44         | 7.23 | 9.73 |
|         | SSDRec           | **11.95**  | **18.93** | **31.13** | **5.88** | **7.74** | **10.22** |
| Ruby    | SSDRec-dynamic   | 7.30       | 11.34 | 21.29 | 3.77         | 4.81 | 6.82 |
|         | SSDRec-static    | 7.39       | 11.36 | 21.01 | 3.84         | 4.88 | 6.81 |
|         | SSDRec           | **7.63**   | **12.05** | **21.98** | **4.01** | **5.19** | **7.17** |
| JavaScript | SSDRec-dynamic   | 5.54       | 9.50 | 18.23 | 2.58         | 3.62 | 5.37 |
|         | SSDRec-static    | 5.10       | 8.67 | 16.76 | 2.36         | 3.30 | 4.92 |
|         | SSDRec           | **5.70**   | **9.73** | **18.47** | **2.65** | **3.73** | **5.48** |

that SSDRec-dynamic outperforms SSDRec-static in most cases and is completely defeated by SSDRec. The reasons are as follows.

1. With the rapid development of computer field, both software and hardware are constantly updated, which leads to the rapid change of developers’ interests. A developer may have been using Angular before, but now he is learning VUE following the trend. Compared with other fields, the dynamic interests of developers in computer field is particularly important. Thus, SSDRec-dynamic beats SSDRec-static in most cases.

2. Although most developers’ interests change frequently, some of their characteristics will not change easily, such as learning speed, coding ability and so on. Thus, SSDRec considering two interests achieves highest HR and NDCG.

5.4 Exploring Hyper-parameters(RQ3)

In this part, we conduct experiments to show how hyper-parameters affect our model.

5.4.1 Neighborhood sample sizes. Due to the large range of node degrees in social and dependency networks, our model uses a sampling technique proposed in [29]. To study the influence of sampling size on aggregation ability, we set $\beta$ and $\gamma$ as sampling sizes of first layer in the two networks
(number of neighbors in every layer is half of previous layer, i.e. sampling sizes of second layer are $\beta/2$ and $\gamma/2$). Figure 3 illustrates the performance of SSDRec when changing $\beta$ and $\gamma$. Note that when $\beta = 0$, the model is equal to SSDRec-dependency proposed in the previous subsection. Similarly, when $\gamma = 0$, our model is equal to SSDRec-social.

As shown in Figure 3, for both $HR@10$ and $NDCG@10$ in PHP, the optimal $\gamma$ and $\beta$ are 10. What’s more, the optimal $\gamma$ and $\beta$ are 8 and 14 respectively in Ruby.

5.4.2 Segmentation strategies of session. Segmentation strategies of session is an important factor that controls the lifespan of sessions. In this part, we compare the performance of SSDRec with different segmentation strategies $^2$ and comparison results on three datasets are shown in Figure 4. From the results, we observe that the performance of SSDRec is dropped with the increase of lifespan in most cases. The possible reason is that one week is enough for the developer to complete a feature of the project, which means the developers’ interests may change in just two weeks, i.e., it will become more and more difficult to model interests from sessions with the lifespan increased.

5.5 Attention Visualization

We hypothesized that developers’ interests rapidly change inter sessions and are relatively stable intra a session. As developers prefer to focus on friends with similar interests, the influences of their friends should follow the same rules. To reflect the change of influences of friends across sessions, we conduct the experiments from both the whole and specific individual.

$^2$To ensure the consistency of data for each segmentation strategy, we do not reprocess the data, but merge adjacent sessions to increase the lifespan of sessions.
5.5.1 developers as a whole. Figure 5 illustrates the empirical distribution of inter-session, intra-session and across-friend attention variance on three datasets. From Figure 5, we can observe:

![Attention variance distribution](image)

Fig. 5. Attention variance distribution of SSDRec for inter-session, intra-session and across friends.

1. As a whole, the inter-session variance is much larger than intra-session variance, indicating attention changes rapidly inter sessions. This confirms that developers’ interests are relatively stable.

2. The attention variance across friends is highest on average, which indicates that although interests of developers are dynamic, they rarely changes dramatically. This is reasonable. For example, a developer may learn angular and Vue sequentially and rarely uses front-end development after learning back-end development. But some of his friends may specialize in front-end development and some specialize in back-end development.

5.5.2 Individual case studies. To show individual developer behavior within and across session(s), we random select a developer $u_{1039}$ who have 8 test sessions and at least 5 friends in PHP. Figure 6 shows the attention visualization inter sessions and intra a session. From Figure 6, we can observe:

![Attention visualization](image)

Fig. 6. Inter(left) and intra-session(right) attention of $u_{1039}$. For both plot, y-axis represents $u_{1039}$’s friends and the color of attention is consistent. In the left plot, x-axis represents sessions and the average attention weight within a session is plotted. In the right plot, to obtain more convincing results, x-axis represent the package sequence in fourth session (the longest session across eight sessions).

1. The attention allocated to friends changes whether across sessions or within a session. This confirms that the influences of social network change with the change of developer’s interest.
Attention is relatively stable intra a session. This is in line with our hypothesis and reasonable. In general, developers’ interests will not change much within a week (lifespan of sessions in our experiments) and their attention to friends is relatively stable. However, the time span inter sessions may exceed one year, during which the developer’s interest may change, causing developer to focus on different sets of friends in different sessions.

6 CONCLUSIONS
In this article, we first depict the evolving interests, dynamic social influences and attentive dependency influences in software recommendation and then propose a unified software recommendation framework to model them. Graph attention network is introduced and extended to combine attentive software dependency and dynamic social influences, and recurrent neural network is applied to model developers’ evolving interests. Experiment results on real-world datasets indicate the performance and effectiveness of our proposed model. In future, we will consider higher order relationship in dependency network and social network.

DECLARATION OF COMPETING INTEREST
The authors declared that they have no conflicts of interest to this work.

ACKNOWLEDGEMENT
This work is supported by the National Natural Science Foundation of China (Grant No. 61872002, U1936220), the University Natural Science Research Project of Anhui Province (Grant No. KJ2019A0037), the National Key R&D Program of China (Grant No.2019YFB1704101) and the Anhui Provincial Natural Science Foundation (Grant No. 2008085QF307).

REFERENCES
[1] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. In Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI ’09, page 452–461. AUAI Press, 2009.
[2] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In Proceedings of the 26th International Conference on World Wide Web, WWW ’17, page 173–182. International World Wide Web Conferences Steering Committee, 2017.
[3] Makoto Ichii, Yasuhiro Hayase, Reishi Yokomori, Tetsuo Yamamoto, and Katsuro Inoue. Software component recommendation using collaborative filtering. In Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, SUITE ’09, page 17–20. IEEE Computer Society, 2009.
[4] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In 2013 20th Working Conference on Reverse Engineering (WCRE), pages 182–191, 2013.
[5] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. Diversified third-party library prediction for mobile app development. IEEE Transactions on Software Engineering, 2020.
[6] Youhua Jiang, Siyi Yan, Peng Qi, and Yan Sun. Adapting to user interest drifts for recommendations in scratch. In 2020 International Wireless Communications and Mobile Computing (IWCMC), pages 1528–1534, 2020.
[7] Weiping Song, Zhiping Xiao, Yifan Wang, Laurent Charlin, Ming Zhang, and Jian Tang. Session-based social recommendation via dynamic graph attention networks. In Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM ’19, page 555–563. Association for Computing Machinery, 2019.
[8] Bálint Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. In International Conference on Learning Representations, 2016.
[9] Ido Guy, Naama Zwerdling, David Carmel, Inbal Ronen, Erel Uziel, Sivan Yoge, and Shila Ofek-Koifman. Personalized recommendation of social software items based on social relations. In Proceedings of the Third ACM Conference on Recommender Systems, RecSys ’09, page 53–60. Association for Computing Machinery, 2009.
[10] Chen Cheng, Haiqin Yang, Michael R. Lyu, and Irwin King. Where you like to go next: Successive point-of-interest recommendation. In Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI ’13, page 2605–2611. AAAI Press, 2013.
[11] Jarana Manotumruksa, Craig Macdonald, and Iadh Ounis. A deep recurrent collaborative filtering framework for venue recommendation. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM ’17, page 1429–1438. Association for Computing Machinery, 2017.

[12] Disheng Dong, Xiaolin Zheng, Ruixun Zhang, and Yan Wang. Recurrent collaborative filtering for unifying general and sequential recommender. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, pages 3350–3356. International Joint Conferences on Artificial Intelligence Organization, 2018.

[13] Shoujin Wang, Liang Hu, Yan Wang, Quan Z. Sheng, Mehmet Orgun, and Longbing Cao. Modeling multi-purpose sessions for next-item recommendations via mixture-channel purpose routing networks. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, pages 3771–3777. International Joint Conferences on Artificial Intelligence Organization, 2019.

[14] Massimiliano Ruocco, Ole Steinar Lillestøl Skrede, and Helge Langseth. Inter-session modeling for session-based recommendation. In Proceedings of the 2nd Workshop on Deep Learning for Recommender Systems, DLRS 2017, page 24–31. Association for Computing Machinery, 2017.

[15] Xiaolong Xu, Xihua Liu, Zhanyang Xu, Fei Dai, Xuyun Zhang, and Lianyong Qi. Trust-oriented iot service placement for smart cities in edge computing. IEEE Internet of Things Journal, 7(5):4084–4091, 2020.

[16] Hao Ma, Dengyong Zhou, Chao Liu, Michael R. Lyu, and Irwin King. Recommender systems with social regularization. In Proceedings of the Fourth ACM International Conference on Web Search and Data Mining, WSDM ’11, page 287–296. Association for Computing Machinery, 2011.

[17] Tong Zhao, Julian McAuley, and Irwin King. Leveraging social connections to improve personalized ranking for collaborative filtering. In Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM ’14, page 261–270. Association for Computing Machinery, 2014.

[18] Shuiguang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. On deep learning for trust-aware recommendations in social networks. IEEE Transactions on Neural Networks and Learning Systems, 28(5):1164–1177, 2017.

[19] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. Neural graph collaborative filtering. In Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR’19, page 165–174. Association for Computing Machinery, 2019.

[20] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In The World Wide Web Conference, WWW ’19, page 417–426. Association for Computing Machinery, 2019.

[21] Zhiwei Guo and Heng Wang. A deep graph neural network-based mechanism for social recommendations. IEEE Transactions on Industrial Informatics, pages 1–1, 2020.

[22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In International Conference on Learning Representations, 2017.

[23] Petar Velickovi, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lió, and Yoshua Bengio. Graph attention networks. In International Conference on Learning Representations, 2018.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural Computing, 9(8):1735–1780, 1997.

[25] Georgios Gousios. The ghtorent dataset and tool suite. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, pages 233–236. IEEE Press, 2013.

[26] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’16, page 265–283. USENIX Association, 2016.

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In International Conference on Learning Representations, 2014.

[28] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1):1929–1958, 2014.

[29] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17, page 1025–1035. Curran Associates Inc., 2017.