Learning Edge Properties in Graphs from Path Aggregations

Rakshit Agrawal
University of California, Santa Cruz
Santa Cruz, California
ragrawa1@ucsc.edu

Luca de Alfaro
University of California, Santa Cruz
Santa Cruz, California
luca@ucsc.edu

ABSTRACT

Graph edges, along with their labels, can represent information of fundamental importance, such as links between web pages, friendship between users, the rating given by users to other users or items, and much more. We introduce LEAP, a trainable, general framework for predicting the presence and properties of edges on the basis of the local structure, topology, and labels of the graph. The LEAP framework is based on the exploration and machine-learning aggregation of the paths connecting nodes in a graph. We provide several methods for performing the aggregation phase by training path aggregators, and we demonstrate the flexibility and generality of the framework by applying it to the prediction of links and user ratings in social networks.

We validate the LEAP framework on two problems: link prediction, and user rating prediction. On eight large datasets, among which the arXiv collaboration network, the Yeast protein-protein interaction, and the US airlines routes network, we show that the link prediction performance of LEAP is at least as good as the current state of the art methods, such as SEAL and WLNM. Next, we consider the problem of predicting user ratings on other users: this problem is known as the edge-weight prediction problem in weighted signed networks (WSN). On Bitcoin networks, and Wikipedia RfA, we show that LEAP performs consistently better than the Fairness & Goodness based regression models, varying the amount of training edges between 10 to 90%. These examples demonstrate that LEAP, in spite of its generality, can match or best the performance of approaches that have been especially crafted to solve very specific edge prediction problems.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Theory of computation** → *Graph algorithms analysis*.

KEYWORDS

Path Aggregation; Neural Networks; Edge Learning

ACM Reference Format:
Rakshit Agrawal and Luca de Alfaro. 2019. Learning Edge Properties in Graphs from Path Aggregations. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3308558.3313695

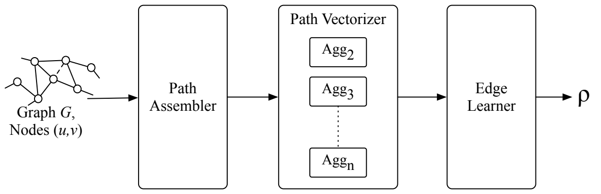

Figure 1: Summarized architecture of LEAP framework

1 INTRODUCTION

Graphs and networks provide the natural representation for many real-world systems and phenomena. In social networks, for instance, different users can be represented by nodes in a graph, and each friendship relation can be represented as an edge. Similarly, in physical systems such as railroad networks or communication networks, each terminal or cellular station is a node in the graph connected to several other terminals. In these graphs, each edge can itself contain significant amount of information. For instance, the presence of an edge in a social network gives a binary signal of presence or absence of a relationship, and a weighted edge on the same network can give a numerical measure of the relationship, hence increasing the degrees of available information. A signed network further contains "positive" and "negative" edge weights, indicating the intensity as well as the direction of a relationship.

The presence and properties of edges in graphs are influenced by several structural factors such as the local neighborhood of the edge, the topology of the graph, and properties and labels associated with surrounding edges in the graph, among others. Machine learning methods can be used to predict the existence of edges, or their properties. For instance, the problem of link prediction is a well explored research area where machine learning methods ranging from heuristics to deep neural networks have been experimented. Similarly, problems around predicting specific edge properties or weights in a graph can also be addressed using learning algorithms.

In this paper, we present a general deep learning framework for learning and predicting edge properties in graphs on the basis of the local neighborhood of the edges. The framework uses the concept of aggregating paths in the graph and is named LEAP (Learning Edges by Aggregation of Paths). A distinctive feature of LEAP is its generality: its ability to learn any kind of edge properties without special need for feature extraction.

In LEAP, for a given graph $G = (V, E)$ where V is the set of nodes and E is the set of edges in the graph, and any two given nodes $(u, v) \in V$, we aim at predicting properties associated with the edge $e_{u,v}$ between the two nodes. For instance, in link prediction, we aim at predicting the presence or absence of this edge, whereas in edge weight prediction, our objective is to predict the weight $w_{u,v}$ associated with the edge $e_{u,v}$. LEAP has a modular architecture consisting of three primary modules: path assembler, path vectorizer, and edge learner (Figure 1). Combined together, these three modules form an end-to-end trainable system. The path assembler gathers paths of different lengths between the node pair (u, v). The pact vectorizer uses an aggregator to summarize the information on the paths into a single vector representation. The edge learner uses the vector representation derived from the path vectorizer, and learns a specific objective for any given edge property.

LEAP's modular architecture makes it easy to implement and experiment. The path aggregators used in LEAP are deep learning modules that take as inputs the raw features of the paths, and produce a trainable aggregation, which is akin to an embedding of the edge set into a vector space. The aggregators thus perform — automatically by computing an embedding — the feature engineering that used to be performed manually, for each property of interest (edge prediction, edge weight prediction, and so on). LEAP, while being an end-to-end learning system that learns embeddings for the nodes and paths itself, can also use pre-trained node embeddings [8, 30], node features, and edge features whenever available.

We present four different kinds of aggregators for LEAP. The aggregators, AvgPool, DenseMax, SeqOfSeq, and EdgeConv, use different neural components and operate at different levels of complexity, focusing on properties like the ordered nature of nodes in a path, and the properties of edges in the paths. We use standard neural modules such as Long Short-Term memory (LSTM) [11] networks, Convolutional Neural Networks (CNN) [19], Pooling operations (Max and Average), and Feed-forward neural networks while constructing our aggregators.

We validate our framework on two specific graph problems: link prediction [21], and edge weight prediction in weighted signed networks (WSN) [18]. In link prediction, we evaluate LEAP on eight real world datasets and present comparisons on the Area under the ROC curve (AUC) score with the current state of the art models WLNM [41] and SEAL [42], and more baseline methods. In the WSN edge weight prediction task presented in Kumar et al. [18] , we evaluate LEAP on three user-user interaction datasets. Two of these datasets refer to Bitcoin trading networks where users provide a rating to other users based on trust. In the third dataset, we learn weights for the votes and sentiment scores assigned by users in Wikipedia to other users when one submits a request for adminship (RfA). We show that LEAP performs similar or better on both these problems against dedicated methods crafted for the specific problems.

The primary contributions of this work can be summarized as follows:

- We present and implement a novel deep learning framework, LEAP, for learning and predicting edge properties of graphs. The framework is general, and it requires no feature engineering, as it relies on deep learning to predict edge properties.
- We define several edge aggregators for LEAP, each suited to particular classes of prediction problems, and we illustrate how LEAP can take advantage of any graph embeddings that may be already available.
- We consider two standard graph prediction problems: link prediction (used, e.g., to predict the formation of connections in social networks) and edge weight prediction (used, e.g., to predict user ratings). We show that LEAP, in spite of its generality, closely matches or improves, on the performance of specialized systems that have been built for these tasks.

In the paper, we will first discuss some related methods for edge property prediction in graphs. We then discuss the motivation behind our LEAP framework and the usage of paths. This is followed by the system design and detailed discussion on aggregators. We then present results from an extensive evaluation over several datasets. We conclude the paper with some considerations on the extensibility and modular design of LEAP.

2 RELATED WORK

Graphs have generated intense research interest over the years in machine learning problems. Commonly studied problems include link prediction [21], node classification, and node ranking [31], among others.

With growing interest in deep learning for graphs, several algorithms for learning node representations have been suggested. These include embedding methods such as LINE [33], DeepWalk [30] and node2vec [8]. More neural network based methods for learning node representations include Graph Convolutional Networks [16], GraphSAGE [9], and Graph Attention Networks [35]. These methods are also often adapted for edge-based learning tasks such as link prediction.

Link prediction has been performed with methods ranging from heuristics to deep neural networks. Martinez *et al.* [23] have categorized the existing link prediction methods into the similarity based, probabilistic and statistical, algorithmic, and preprocessing categories. Similarity based methods operate on the intuition of similar nodes having an affinity towards each other. Locally focused methods like Common Neighbors [21] and Adamic-Adar [2] are interpretable simple methods used extensively for link prediction. More complex similarity-based methods include Katz index [14], PageRank [29], and SimRank [13], among others. Al Hasan *et al.* [10] have explored the use of standard machine learning classifiers for link prediction. Factorization method are also used by Menon and Elkan [24] for link prediction.

Weisfeiler-Lehman Neural Machine (WLNM) [41], and Subgraphs, Embeddings, and Attributes for Link prediction (SEAL) [42] present dedicated deep learning systems for link prediction and define the current state of the art in the space.

In weighted signed networks, for predicting edge weights, Kumar et al [18] defined the learning objective and adapted methods like Bias-Deserve [26], Signed Eigenvector Centrality [5], PageRank [29], and more trust based algorithms. Edge based methods are also used in applications such as SHINE [37] and Rev2 [17] where properties from graphs are used in determine dataset specific tasks.

For the tasks of link prediction and edge weight prediction in weighted signed networks, we present comparison of LEAP with many of these methods later in the paper.

3 MOTIVATION

Edges in real world networks are representative of latent properties within the graph as well as properties among the nodes. Intuitively, for any two nodes (u, v) in the graph, the properties of an edge $e_{u,v}$ between them should depend on the characteristics of the nodes themselves. However, the nodes themselves can be characterized by the edges involving these nodes, hence increasing the dependence to neighborhoods. Therefore, $e_{u,v}$ can be affected by several other nodes and edges in the graph and not just the node pair (u, v). In order to learn more about the edge, it is therefore important to explore the neighborhood of u and v.

For example, if the graph represents a professional or social network, the formation of an edge $e_{u,v}$ between u and v may depend on the properties of u and v, and also on the properties of common friends and friend-of-friends, that is, on the properties of paths emanating from u and v. Equally, in a trust network, the amount of trust of a user u on a user v, constituting a label for $e_{u,v}$, can depend not only on the properties of u and v themselves, but also on the other sets of users that trust, and are trusted by, u and v.

In order to learn more about an edge $e_{u,v}$, it is therefore important to explore the neighborhood of u and v, and assemble the nodes and edges from the graph that can impact the edge $e_{u,v}$ the most. In particular, our framework will consider the paths originating at u and ending at v. These paths can involve a large set of nodes and edges, each of which are related to the two nodes by being an intermediary in a path between them. These intermediate nodes and edges give us information on u and v. The framework we present will enable us to learn from this shared neighborhood of u and v when predicting the properties of $e_{u,v}$.

We note that we could also consider paths that originate at u or v, but do not connect u with v; these paths would provide a characterization of one of u and v only. It would be easy to extend our framework to consider these paths also. However, graph embeddings already enable us to summarize properties of individual node neighborhoods; for this reason, our framework considers mainly the *shared* neighborhood around the edge to be predicted.

4 THE LEAP FRAMEWORK

LEAP is an end-to-end deep learning framework for graph edge learning. The core concept driving LEAP is the ability to learn edge properties in a graph simply from the graph structure, without any need for feature engineering. Moreover, in the presence of explicit features, LEAP can use both the available features as well as self-learned representations from the structure of the graph, such as embeddings. The LEAP framework consists of three separate modules: path assembler, path vectorizer, and edge learner; an overview of the system is presented in Figure 2.

LEAP operates on a given graph $G = (V, E)$, where V is the set of vertices (nodes), and $E \subseteq V \times V$ is the set of edges. An edge can be directed or undirected, weighted or unweighted, and signed or unsigned. A path p^l of length l between the two nodes (u, v) consists of a sequence of nodes $u_0, u_1, \ldots, u_l$, with $u_0 = u$ and $u_l = v$. The set of paths $\mathcal{P}^l_{u,v} = \{p^l_1, p^l_2, \ldots, p^l_n\}$ consists of all the paths of length l between the nodes (u, v). For simplicity, we will often use the notation p^l and $\mathcal{P}^l$ when referring to a path $p^l_{u,v}$ and the set of paths $\mathcal{P}^l_{u,v}$ of length l between the vertices (u, v).

The end-to-end learning objective of LEAP is guided by the input of graph G, two specified nodes (u, v), and the target property ρ being predicted about (u, v). For instance, in case of link prediction, where the framework can be used to predict the presence or absence of an edge between the two nodes (u, v), the output prediction $\rho \in [0, 1]$ from the model represents the probability of an edge existing between (u, v). Similarly, in an edge weight prediction task, the model output ρ can be the predicted weight for edge $e_{u,v}$. We now define the separate modules of the framework used for these learning objectives.

4.1 Path Assembler

The first phase of LEAP is an exploration task which gathers data from the graph structure to be used by the subsequent learning modules. Given a graph G and a pair of nodes (u, v), we start by assembling paths of different lengths between the two nodes. As the learning objective is concerned with the pair (u, v), we do not include the possible 1-length path u, v among the paths considered. As a hyper-parameter provided to the system, we define a set $\mathbf{L} = \{l_1, l_2, \ldots, l_k\}$, as the set of path lengths to be used.

For any length $l \in \mathbf{L}$, we now need to collect a set of paths $\mathcal{P}^l$ of length l. For ease of computation, we can limit the size of each subset, as well as the size of $\mathbf{L}$ as required by the problem and the dataset: if there are too many paths between two nodes, a random subset of paths can be extracted for use by the framework. From the perspective of the framework, the processing is independent of these sizes. Once collected, the paths are then made available to the system in the form of k path-sets $\mathcal{P}^l$ with each set consisting of paths of same length l.

The objective of assembling the paths in length-specific sets is to allow our system to learn properties particular to path lengths. For example, when considering paths of length

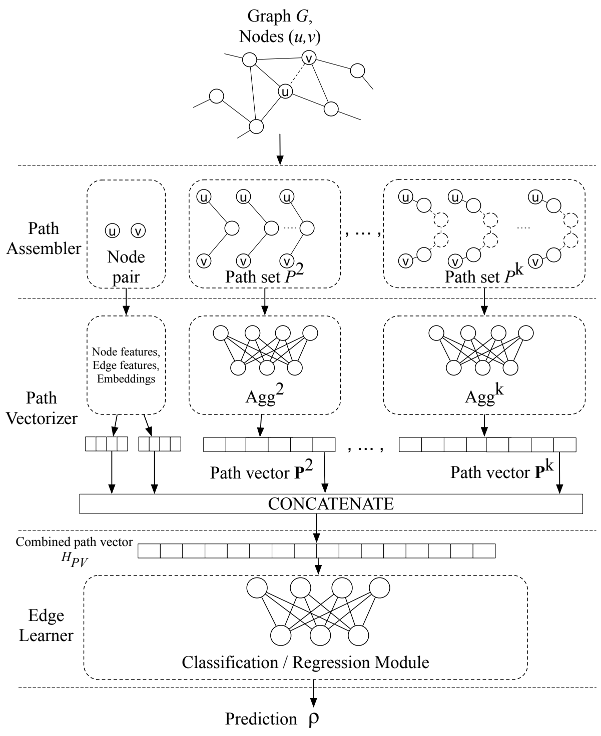
Graph G,
Nodes (u,v)
Path Assembler
Node pair
Path set P^2
Path set P^k
Path Vectorizer
Node features, Edge features, Embeddings
Agg^2
Agg^k
Path vector $\mathbf{P}^2$
Path vector $\mathbf{P}^k$
CONCATENATE
Combined path vector H_{PV}
Edge Learner
Classification / Regression Module
Prediction ρ

Figure 2: Overview of the LEAP framework

$l = 2$, each path between (u, v) differs only by one vertex. By processing such paths together, we increase the capability of the learning modules to focus on a specific variable property at a time, and therefore capture potential factors affecting the predicted output ρ.

The LEAP framework can be generalized to include the exploration of general paths from u, or to v, rather than only paths from u to v.This can be particularly useful when few (or no) paths connect u to v, while the nodes u and v, individually, belong to many paths.

4.2 Path Vectorizer

At the second phase of LEAP, the assembled paths are passed through a deep learning system. This phase is called the path vectorizer. The objective of this phase is to combine all the available information about the node pair (u, v) using the nodes and the paths, and obtain a vector representation that can be used with different edge learning objectives. This module is inspired by the deep learning architectures capable of learning from complex data structures with different dimensionality. For a graph G with N nodes, each node $x \in V$ is given an integer symbol $x_i \in [1, N]$. In order for LEAP to learn from the graph, these symbolic nodes need to be represented as vectors that can be further processed using neural networks. Different methods of vectorization can be used to represent each node. The simplest notation, which retains the symbolic nature of the nodes, can be to use a one-hot vector of size N. In this notation, node $x_i \in V$ is represented as a vector χ_i of size N where $\chi_i[k] = 1$ if $k = x_i$, and $\chi_i[k] = 0$ otherwise. While this representation allows one to identify each node differently, it does not provide the model with any additional information about the node that can be indicative of its relationship to other nodes or the graph structure. For this purpose, instead of the one-hot vectors, in this paper we use the concept of dense embeddings [8, 9, 25, 30]. We represent each node x_i as a dense vector $\chi_i \in \mathbb{R}^K$ of a fixed dimensionality K. This representation of a node, is obtained through a reference lookup Emb by the entire framework, where $\chi_i = \text{Emb}(x_i)$. Similar to the general use of dense embeddings in deep learning systems, the embeddings can either be trained with the entire system, or can be pre-trained using an embedding generation method first, and then later used with the framework. Additionally, if we have set of node features $\mathcal{N}$ representing engineered feature sets for each node, then they can be used in the framework by combining them with the embeddings. For a node x_i, therefore, the embedding representation used by the system will be $\chi_i \Leftarrow \chi_i|\mathcal{N}(x_i)$, where (|) is a concatenation operation.

The nodes and the paths made available by the path assembler are then used into the path vectorizer using the embedding lookup Emb. Each path, represented as a sequence of nodes in their integer representation is passed through the embedding layer, in order to obtain sequences of vectors, with each sequence of length l representing a path of length $l - 1$. At this stage, we have the two concerned nodes u and v, set of path lengths $\mathbf{L}$, the set of paths $\mathcal{P}^l$ for each length l, and the embedding lookup Emb. In case where edge features $\mathcal{E}$ are available for the graph, LEAP can use these features within the learning module as well. We use Algorithm 1 for obtaining a vector representation H_{PV} from these inputs.

Aggregators. In Algorithm 1, the paths are first converted into their vector representation and then passed through the Aggregator. Aggregators are the primary learning units in LEAP, and are discussed in detail in Section 5. For each path length l, an associated Aggregator Agg_l is responsible for processing the paths and learning a vector representation from them. Initially, in the path vectorizer, path sets of different lengths are provided. Since we aim at learning length-wise significant information from these paths, we process the set for each length separately. An aggregator Agg_l is a deep learning module which takes the path set $\mathbf{P}^l$ as an input, and learns a vector representation $\mathbf{h}_l$. Due to this separate modular structure, LEAP can use several different types of aggregators. The aggregator can also use an edge feature set $\mathcal{E}$ when engineered features for the graph edges are also available. In the path vectorizer, the vectors $\mathbf{h}_l$ learned for each length $l \in \mathbf{L}$ are concatenated together along with the vector representations for the input node pair (u, v) to obtain a final vector representation H_{PV}.

4.3 Edge Learner

The last step of LEAP is to perform problem specific learning on the edge $e_{u,v}$ between the two given nodes (u, v). The

Algorithm 1 PathVectorizer

```
Input: Node pair $(u, v)$, Path lengths $\mathbf{L}$,
Path sets $\mathcal{P}^l$ for $l \in \mathbf{L}$, Embedding lookup Emb,
Node features $\mathcal{N}$, Edge features $\mathcal{E}$
$\mathbf{u} \leftarrow \text{Emb}(u)|\mathcal{N}(u)$
$\mathbf{v} \leftarrow \text{Emb}(v)|\mathcal{N}(v)$
for $l$ in $\mathbf{L}$ do
    $\mathbf{P}^l \leftarrow \text{Emb}(\mathcal{P}^l)|\mathcal{N}(\mathcal{P}^l)$
    $\mathbf{h}_l = \text{Aggregator}(\mathbf{P}^l, \mathcal{E})$
end for
$H_{PV} = \text{Concat}[(\mathbf{u}, \mathbf{v}, \mathbf{h}_l)|l \in \mathbf{L}]$
return $H_{PV}$
```

input to the edge learning module is the combined vector H_{PV}. This vector can now be used by any classification or regression method for respective supervised learning problem. For instance, in link prediction problem, where the objective is to detect the presence of an edge between the two nodes (u, v), this module can be used as a binary classifier to classify between *"link exists"* and *"link does not exist"* by predicting a probability $\rho \in [0, 1]$ of a link between (u, v). In general, for any edge based classification, whether binary or multi-class, the module can be used as a classifier with an input vector H_{PV}. The edge learning module can also be used for regression problems. For instance, in edge weight prediction, we can use the output ρ from the module as the predicted weight for the edge. In case of signed edge weight, the same regression module can be used by allowing it to produce both positive and negative values. In multi-class classification, the output ρ can be treated as a vector $\rho \in \mathbb{R}^M$ for M number of classes.

Since the edge learner by itself is simply a classification or a regression module, any corresponding learning algorithm can be used here. For maintaining LEAP as an end-to-end trainable deep learning system, we use feed-forward neural networks for the edge learner. These networks can vary in depth by increasing the number of layers, with the first layer receiving input vector H_{PV} and the final layer predicting the output ρ. Given the vector H_{PV}, and parameter N_{EL} as the number of layers, the process of the edge learner is presented in Algorithm 2

Algorithm 2 EdgeLearner

```
Input: Combined Path vector $H_{PV}$, Layer count $N_{EL}$,
$\mathbf{h_{EL}} \leftarrow H_{PV}$
for $c = 1 \ldots N_{EL}$ do
    $\mathbf{h_{EL}} \leftarrow \sigma_c(\mathbf{W}_c \cdot \mathbf{h_{EL}} + \mathbf{b}_c)$
end for
$\rho = \sigma_p(\mathbf{W}_p \cdot \mathbf{h_{EL}} + \mathbf{b}_p)$
return $\rho$
```

For each layer $c = 1 \ldots N_{EL}$, the weight matrix of the neural network layer is represented by $\mathbf{W}_c$ and the bias of the layer is represented by $\mathbf{b}_c$. σ_c refers to the activation function used by the network layers such as *tanh*, *ReLU*, *sigmoid* among others. For the final prediction output layer, the weight $\mathbf{W}_p$ and bias $\mathbf{b}_p$ are used. The activation function σ_f for this layer is decided based on the nature of the problem. In case of binary classification, often the *sigmoid* $\in [0, 1]$ function is used. For multi-class classification, it is common to use the $softmax$ function. In case of normalized edge weight prediction in signed networks, with $w_{u,v} \in [-1, 1]$, we use the $tanh \in [-1, 1]$ activation function to obtain a floating point value representing the predicted signed edge weight.

5 AGGREGATION MODELS

In the previous section, we explained the architecture of LEAP and discussed the requirement of aggregators for performing path vectorization. The concept of using aggregators in graphs is inspired by GraphSAGE [9] where they perform node classification by learning representations for the nodes in a graph. In our system, we have adapted the concept of aggregators for combining paths between two nodes (u, v) and obtaining representations for edges in the graph. Within the path vectorization module, an aggregator Agg_l for paths of length l gets the vectorized path set $\mathbf{P}^l$ as input with an objective of generating output vector $\mathbf{h}_l$.

Since the LEAP framework consists of neural network based layers, each aggregator is itself a deep learning model where the input is a tensor of rank 4 — (batch size, number of paths, path length, node embedding). Each path itself is a sequence of node vectors, and each path set is a set of several such sequences. Therefore, in order to derive single vector representation of the path set, we first need to aggregate all the nodes in the path using aggregator Agg_{node} and then aggregate these paths using the aggregator Agg_{path}. The training of these aggregators is performed with the overall LEAP system using gradient descent based methods. While several other variants are possible, we present four different kinds of aggregators used in this paper.

5.1 AvgPool Aggregator

Our first aggregator follows a simple architecture of combining different vectors together. We call the model for this aggregator AvgPool. This model relies only on the embeddings χ_i for each node x_i under consideration. The model does not have training parameters.

In AvgPool, Agg_{node} concatenates all the node vectors along the path into a single vector. Then, on the set of these derived path vectors, Agg_{path} performs a one-dimensional average pooling operation. The resulting vector $\mathbf{h}_l \in \mathbb{R}^{(l+1)K}$ is therefore a single vector obtained by averaging the paths between the two nodes (u, v) across the paths $k \in K$. The AvgPool aggregator can be summarized as:

$$\mathbf{h}_l = \text{AvgPool}([\overline{(\mathbf{p}_i^l)}, \ \forall \mathbf{p}_i^l \in \mathbf{P}^l]) \tag{1}$$

where AvgPool is the one-dimensional average pooling operation, and $\overline{(\cdot)}$ is the vector concatenation operation which combines multiple vectors by concatenating them together.

$$\overline{([\chi_1, \chi_2, \ldots, \chi_l])} = \chi_1 | \chi_2 | \ldots | \chi_{l+1} \tag{2}$$

where (|) is the concatenation of two vectors. $\mathbf{P}^l$ is the set of vectorized paths of length l in graph G between the two nodes (u, v). $\mathbf{p}_i^l$, indexed by i is an individual path of length l in set $\mathbf{P}^l$.

AvgPool relies on the embeddings for each node, and represents a path as a fixed size vector of all the nodes combined. Since the first and last node of these paths are the nodes u and v respectively, the only changing bits belong to the nodes within the paths. By performing a bitwise pooling operation over these nodes, we can derive mean vector representations for the changing nodes in the path set. Since the embeddings themselves are still trained by the complete framework, the gradients obtained for updating the node embeddings correspond to these average representations and their influence on the final output ρ.

5.2 DenseMax Aggregator

The DenseMax aggregator is a learning model that uses a dense (feed-forward) neural network layer for each path. Similar to AvgPool, in DenseMax, Agg_{node} obtains the representation for each path by concatenating the node vectors into a single long vector. In this model, at Agg_{path}, the path vector is first passed through a dense neural layer. The resulting activations are then passed through a max-pooling operation which helps derive a single vector representation for the paths of length l. Therefore, Agg_{path} in DenseMax consists of a dense neural network layer and a one-dimensional max pooling operation. The operations of the DenseMax aggregator can be summarized as:

$$\mathbf{h}_l = \textsc{MaxPool}([\sigma(\mathbf{W}_l \cdot \overline{(\mathbf{p}_i^l)} + \mathbf{b}_l), \forall \mathbf{p}_i^l \in \mathbf{P}^l]) \tag{3}$$

where $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weight matrix and bias for the dense neural layer. σ is the activation function used by the dense layer. $\overline{(\cdot)}$ is the vector concatenation operation. MaxPool is the one-dimensional max pooling operation which selects bitwise maximum value from multiple vectors to derive a single final vector. The MaxPool operation is used on these representations in order to capture the most activated bits that can affect the final output ρ.

5.3 SeqOfSeq Aggregator

The sequence of nodes from u to v can hold information relevant to the final prediction. For instance, if the existence of an edge between u and v depends on the presence of a path between the two nodes with consecutively increasing edge weight, the sequential order of nodes contains information which holds significance to the final outcome of the model. Therefore, in SeqOfSeq aggregator, we treat the paths as ordered sequences of nodes. We can further consider the path set as a sequence of different paths, if the paths can be ordered using some characteristics. For example, if the edges are labeled by weights, then the total weight of a path p^l is the sum of edge weights for each edge in the path. If the paths from u to v are then sorted according to their total weights, we can process them in a specified order. To this end, the aggregators Agg_{node} and Agg_{path} would need to be sensitive to the order of the inputs.

In the SeqOfSeq aggregator, we first use an LSTM $\textsc{Lstm}_{inner}$ on each path. From the output activations of $\textsc{Lstm}_{inner}$, we extract a vector representation for the path by performing a max-pooling operation. The aggregator Agg_{node}, in this case, consists of both the $\textsc{Lstm}_{inner}$ and the max-pooling operation. We use a max pool here instead of using only the activation from last timestep of the $lstm_{inner}$: we believe that since our objective is to extract information from the path itself, a max-pooling operation can be more effective in summarizing the path than the final activation. After summarizing each path into a single vector, the sequence of path vectors is processed by a combination of another LSTM $\textsc{Lstm}_{outer}$, followed by a max-pooling operations, as the Agg_{path} aggregator. The SeqOfSeq aggregator can be summarized as:

$$\begin{aligned} H_{inner} &= [\textsc{MaxPool}(\textsc{Lstm}_{inner}(\mathbf{p}_i^l)), \forall \mathbf{p}_i^l \in \mathbf{P}^l] \\ \mathbf{h}_l &= \textsc{MaxPool}(\textsc{Lstm}_{outer}(H_{inner})) \end{aligned} \tag{4}$$

where MaxPool is the one-dimensional max pooling operation, H_{inner} is the intermediate sequence of derived path vectors, and $\textsc{Lstm}_{inner}$ and $\textsc{Lstm}_{outer}$ are the inner and outer LSTMs respectively, used for processing corresponding sequences.

Variants of SeqOfSeq can also be created to use order information only at the paths, or only at the nodes. With the use of sequence learning neural networks such as LSTMs, the SeqOfSeq aggregator is more powerful than the AvgPool or DenseMax aggregators, and it trains a much larger number of parameters.

5.4 EdgeConv Aggregator

Edges of a path can themselves contain significant information. In order to emphasize learning also from the the edges, we propose an aggregator called EdgeConv which focuses on edges while operating over paths. In order to build a learning widget that can operate on the edge, we use a one-dimensional Convolutional Neural Network (CNN) with a window size of 2 that takes as input two consecutive nodes forming an edge. Therefore, when a path is represented as a sequence of nodes, the convolution kernel focuses on all pairs of consecutive nodes along the edge. Given the convolutional results on all pairs of consecutive nodes, we apply a max-pooling operation to compute the overall path label. The aggregator Agg_{node} for EdgeConv, therefore, consists of a one-dimensional CNN and a max-pooling operation. Considering the set of derived paths as an ordered sequence, the Agg_{path} for this case also uses an LSTM and max-pooling operation. Therefore, all the path vectors for paths of length l derived using Agg_{node} are then processed using an LSTM, followed by another max-pooling operation to derive the final vector representation $\mathbf{h}_l$.

EdgeConv can be summarized as :

$$
\begin{aligned}
H_{inner} &= [\textsc{MaxPool}(\textsc{Conv1D}(\mathbf{p}_i^l)), \forall \mathbf{p}_i^l \in \mathbf{P}^l] \\
\mathbf{h}_l &= \textsc{MaxPool}(\textsc{Lstm}(H_{inner}))
\end{aligned} \tag{5}
$$

where MaxPool is the one-dimensional max pooling operation, H_{inner} is the intermediate sequence of derived path vectors, and Lstm is the LSTM module used to learn from different path vectors.

Similar to SeqOfSeq, it is not necessary to treat the paths as an ordered sequence, and different variants of EdgeConv can consider the set of derived paths as an unordered set. The sequence of nodes, however, needs to be ordered in EdgeConv, as it operates consecutively over the edges.

5.5 Aggregator Extensions

The four aggregators presented above provide different levels of complexity and use different neural network modules. Together, they illustrate how the modular nature of LEAP allows us to use different neural network architectures as part of the full system for an end-to-end training. We believe that the use of aggregators is key to achieving an extensible and flexible framework. Specialized aggregators can be trained by focusing on significant properties under concern. The aggregators can also include the new deep learning concepts of attention [3, 22] and memory [40]. Similarly, graph specific neural models like Graph Convolutional Networks [16] and Graph Attention Networks [35] can be adapted as aggregators by representing the assembled path sets as subgraphs. While we train the aggregators along with the entire framework, they can be trained separately using any objective function. In case of transfer learning, a well trained model can be transfered into the LEAP framework and can be used simply as a function without training further. Similarly, a partially trained model can be used as an aggregator, and it can further be trained by the learning objective of the complete framework.

6 EVALUATION

The design of the LEAP system, and the use of aggregators, make it an easy to use end-to-end learning system for any kind of graph. For small graphs with fewer nodes, simpler aggregators with few parameters can be used. For very large datasets, we can construct complex aggregators targeted at several latent properties in the graph. In order to demonstrate the learning abilities of this system, we evaluate it on two commonly studied problems in graphs and social networks — link prediction, and edge weight prediction in weighted signed networks.

6.1 Link Prediction

Graphs and networks evolve over time by creation of newer links between the nodes. Given a graph G and a pair of nodes (u, v), the link prediction problem aims at predicting the probability of existence of an edge $e_{u,v}$ between the two nodes.

Table 1: Summary of the datasets used for evaluation

Type	Name	Nodes	Edges
Link Prediction	USAir	332	2,126
	NS	1,589	2,742
	PB	1,222	16,714
	Yeast	2,375	11,693
	C.ele	297	2,148
	E.coli	1,805	14,660
	arXiv	18,722	198,110
	FB	4,039	88,234
Weighted Signed	Bitcoin-OTC	5,881	35,592
Networks	Bitcoin-Alpha	3,783	24,186
	Wikipedia-RFA	9,654	104,554

6.1.1 Learning Objective. In order to learn this objective using LEAP framework, in the Edge Learner module, we can treat it as a binary classification problem. From the graph $G = (V, E)$, the set of edges E can be considered as the positive sample set. Similarly a set of node pairs $(x_1, x_2) \in V$ can be sampled from the graph where edge $e_{x_1,x_2} \notin E$ is the negative set for classification. A label $\tau = 1$, therefore can be associated with positive pairs and a label $\tau = 0$ is associated with the negative pairs.

6.1.2 Datasets. We evaluate the LEAP Link Prediction model on eight real world datasets. The choice of datasets is motivated by the link prediction results presented in two state of the art models — WLNM [41] (Weisfeiler-Lehman Neural Machine) and SEAL [42] (Subgraphs, Embeddings, and Attributes for Link prediction). The datasets used in this paper are listed in Table 1.

USAir [4] is a network graph for US airlines. Network Science (NS) [27] is the collaboration network for researchers in the subject of network science. Political Blogs (PB) [28] is a political blog network form the US. Yeast [36] is a PPI (protein-protein interaction) network for yeast. C.ele is the neural network of the worm Caenorhabditis elegans [38]. E.coli is the dataset of pairwise reaction network of metabolites in E.coli [43]. arXiv [20] is the collaboration network of research papers on arXiv under the Astro Physics category. Facebook (FB) [20] is the dataset of friend lists from the Facebook social network.

6.1.3 Experiment Setup. We performed an extensive set of experiments on the above mentioned datasets in order to evaluate and compare our framework against state of the art methods in link prediction. For each dataset, we sampled a variable number of data samples into the training set and evaluated the model on the remaining samples. The results presented in this paper adopt the partitioning used in the current state of the art method SEAL [42]. For smaller datasets with less than 2500 nodes, we use 90% of the graph edges and an equal number of negative samples for training, and present evaluation results on the remaining 10% edges and equal number of negative

Table 2: Area under the ROC curve (AUC) comparison of LEAP with baselines. Best LEAP results and best dataset results are highlighted. *n2v* refers to the use of node2vec embeddings with LEAP. *OOM* refers to Out-of-Memory.

	USAir	NS	PB	Yeast	C.ele	E.coli	arXiv	FB
Adamic-Adar	0.9507	0.9498	0.9250	0.8973	0.8659	0.9524	-	-
Katz	0.9273	0.9524	0.9306	0.9264	0.8606	0.9329	-	-
PageRank	0.9486	0.9529	0.9374	0.9314	0.9046	0.9548	-	-
node2vec	0.9122	0.9198	0.8621	0.9407	0.8387	0.9075	0.9618	0.9905
Spectral Clustering	0.7482	0.8829	0.8261	0.9346	0.5007	0.9514	0.8700	0.9859
WLK	0.9598	0.9864	OOM	0.9550	0.8965	OOM	-	-
WLNM	0.9571	**0.9886**	0.9363	0.9582	0.8603	0.9706	0.9919	0.9924
SEAL	**0.9729**	0.9761	0.9540	**0.9693**	0.9114	0.9704	0.9940	**0.9940**
LEAP-AvgPool	0.9259	0.9362	0.9555	0.9474	0.9011	0.9484	0.9918	0.9916
LEAP-DenseMax	0.9555	**0.9785**	0.9541	0.9573	0.9050	0.9662	0.9940	0.9914
LEAP-SeqOfSeq	0.9576	0.9635	0.9547	0.9540	0.9153	0.9626	**0.9941**	0.9907
LEAP-EdgeConv	**0.9639**	0.9621	**0.9577**	0.9554	0.9058	0.9614	**0.9941**	0.9908
LEAP-n2v-AvgPool	0.9086	0.9068	**0.9586**	0.9551	0.8909	0.9505	0.9919	0.9920
LEAP-n2v-DenseMax	0.9518	0.9636	0.9564	0.9652	0.9129	**0.9719**	0.9934	0.9914
LEAP-n2v-SeqOfSeq	0.9532	0.9618	0.9571	0.9610	0.9083	0.9662	0.9938	**0.9924**
LEAP-n2v-EdgeConv	0.9547	0.9622	0.9575	**0.9639**	**0.9185**	0.9678	**0.9941**	0.9921

pairs. For relatively larger datasets with more than at least 4000 nodes, we partition the training and evaluation datasets at 50%.

The LEAP[1] system and the aggregators were written in Python with Keras [7] deep learning framework, using the Tensorflow [1] backend. The hyperparameters for each aggregator were selected using multiple trials. We report the results with the best hyperparameters for each setting individually. In all the reported results, we used the path lengths in set $\mathbf{L} = \{3, 4\}$, and used upto 50 paths for each length selected randomly. All the methods were trained using a loss function of binary cross-entropy and the Adam [15] optimizer for gradient descent with a learning rate of 0.001. Each model was trained for upto 30 epochs with early stopping enabled.

6.1.4 Results. We present the results obtained as per the above mentioned experimental setup for LEAP in table 2. The results comprise of LEAP with the four aggregators — AvgPool, DenseMax, SeqOfSeq, and EdgeConv, discussed in the paper. Additionally, we also compare the ability of our model to learn node embeddings themselves against using pre-trained embeddings. For pre-trained embeddings, similar to SEAL, we use the node2vec [8] method first on the graph to derive node embeddings, and then use them with the LEAP framework without updating them further during the system training. We compare these results with three different kinds of methods used for link prediction. We first use the heuristics including Adamic-Adar [2], Katz index [14], and the PageRank [6] algorithm. For feature learning based models, we use spectral clustering [34], and the node2vec [8] algorithm which learned node embeddings and then performs a link prediction task on them. Finally, we compare our system with subgraph-based link prediction methods, defining the current state of the art. These methods include Weisfeiler-Lehman graph kernel (WLK) [32], Weisfeiler-Lehman Neural Machine (WLNM) [41], and Subgraphs, Embeddings, and Attributes for Link prediction (SEAL) [42]. Performance of all the models is compared using the Area under the ROC curve (AUC) metric. We use the settings and results by Zhang et al. [42] for all our baseline methods.

As can be seen in table 2, LEAP performs best or close-to-best on each dataset. Further, we show that an external method for learning node embeddings like node2vec can easily be used within the system. Similarly we can also incorporate known feature vectors for the nodes whenever available.

In comparison to deep learning methods and current state of the art WLNM and SEAL, LEAP achieves equivalent or better performance with the presented aggregators. However, due to its modular nature, the LEAP framework can be highly extended and adapted for different datasets with different latent properties. Further, SEAL requires two prominent steps of node labeling and embedding generation before the neural network can be trained on the graph. In making a learning framework easily deployable on multiple platforms, it is highly advantageous to have an end-to-end trainable system and LEAP provides this particular ability with sufficient modularity to tune the simplicity of the model as required.

[1]The code is available at https://github.com/rakshit-agrawal/LEAP

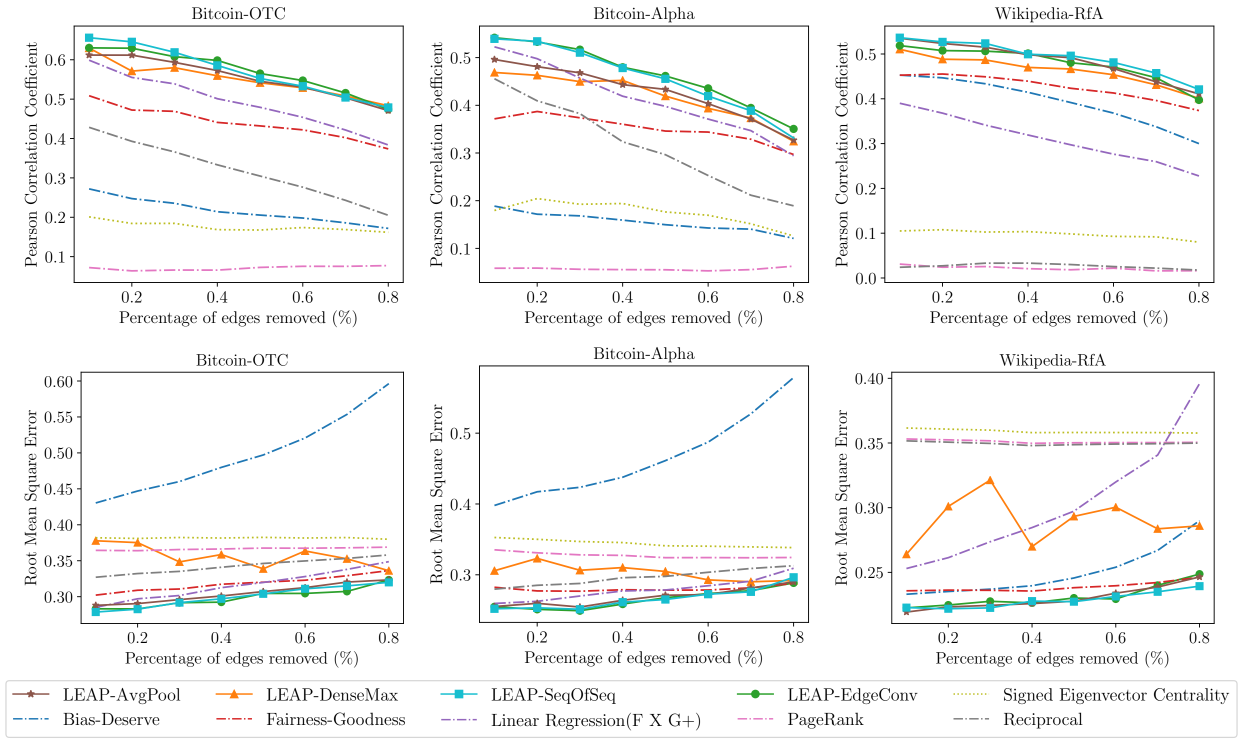

Figure 3: Plots for PCC(top) and RMSE(bottom) on the three datasets for Weighted Signed Networks. The x-axis refers to the percentage of edges removed while training the models. Over all the datasets, and along both the metrics, LEAP based methods show significantly better performance, with the complex aggregators SeqOfSeq and EdgeConv giving the best performance on both the metrics.

6.2 Weighted Signed Networks

In real world datasets representing relations of certain kind among the nodes, the edges can possess different meaningful properties that are of significance to the underlying network. For instance, in a user-user interaction system, each user can have a trust associated with another user. This trust further propagates through the network, affecting the trust between two different users in a certain way. A generalized representation of such networks is obtained through weighted signed networks (WSN). A WSN consists of a graph $G = (V, E)$ where an edge $e_{u,v} \in E$ between two nodes $(u, v) \in V$ has a weight $w_{u,v}$ associated to it. The weight $w_{u,v}$ can be a signed weight, specifying a positive or negative sentiment with a magnitude $|w_{u,v}|$ specifying the intensity of the relation. A WSN graph can either be directed or undirected.

6.2.1 Learning Objective. Kumar et al [18] presented the task of predicting edge weight in WSNs where given a graph G and a node pair (u, v), the objective is to predict the signed edge weight $w_{u,v}$ between the two nodes. For simplicity, the edge weights are normalized to a scale of $w_{u,v} \in [-1, 1]$. In the case of the LEAP framework, the problem of predicting edge weight can be treated as a regression problem for the Edge Learner module. From the graph $G = (V, E)$, we can use the entire set of edges E with the edge nodes $(x_1, x_2) \in V$ for edge $e_{x_1,x_2} \in E$ being the input nodes to the system. Each such node pair can be associated with a regression label $\tau = w_{x_1,x_2}$ using the weights associated with each edge.

6.2.2 Datasets. For evaluation of the LEAP WSN Edge Weight Predictor, we use three real world datasets of user-user interaction networks. These datasets used here are influenced by Kumar et al [18] and are available from SNAP database [20]. The three datasets are listed in Table 1.

Bitcoin-OTC [18] is the "who-trusts-whom" network of people trading Bitcoins on the platform "Bitcoin OTC". The directed signed edge weight between the users on this network refers to the rating given by a user to the other user on the network on a scale from -10 to 10. Bitcoin-Alpha [18] is a similar network for a different trading platform called "Bitcoin Alpha". Wikipedia-RFA [18, 39] is a voting network for Wikipedia request for adminship (RfA). The edges on the network refer to directed votes between users. To associate weights to the signed edges, Kumar et al [18] used the VADER sentiment engine [12] and used the difference between the positive and negative sentiment scores obtained for the vote explanation text in the Wikipedia-RFA dataset.

6.2.3 *Experiment Setup.*

In this evaluation, we present and compare the results for regression on predicting edge weights with $\delta\%$ edges removed. We vary the value of δ between 10% to 80%, with a step size of 10%, specifying the range for partitioning the training and the evaluation datasets.

The system and hyperparameter settings for this task are same as the link prediction model. The only difference in this adaptation of LEAP for a regression task is the choice of loss function. We use the Mean Squared Error (MSE) as the loss function used to compute the gradients for training the Edge Weight Prediction model. The hyperparameter selection was performed using multiple trials, and the best settings were used to report the results.

6.2.4 *Results.*

Similarly to link prediction, we evaluated our system for this objective with the four aggregators — AvgPool, DenseMax, SeqOfSeq, and EdgeConv, discussed in the paper. Neural network based models are not used directly for the task of edge weight prediction in WSN and therefore we compare our results to the heuristic and feature learning based methods previously applied on these datasets.

In the definition of edge weight prediction on WSNs by Kumar et al [18], they adapted several algorithms for this task, providing us with a set of baseline measures on these datasets. We first use a basic method of Reciprocal [18], where the edge weight $w_{u,v}$ is same as that of the reciprocal edge weight $w_{v,u}$ if there exists an edge $e{+}v, u \in E$, and 0 otherwise. We then use two graph algorithms PageRank [29] and Signed Eigenvector Centrality [5]. Each of these algorithms independently learn a score for each node in the graph. The edge weight predicted by these methods simply refers to the difference between the scores obtained for the nodes u and v. Finally, we compare our system with relation specific algorithms used for extracting interaction measures between the nodes. These include Bias-Deserve [26] and Fairness-Goodness [18]. These are iterative algorithms that associate two properties with each node, and learn them by performing sequential iterations on all the nodes in the graph, updating one property at a time. An additional method used in [18] is a Linear Regression model using the above mentioned values as features. This method is identified as Linear Regression (F × G +)

All these methods were evaluated using the experiment setup mentioned above and the results were measures across two standard metrics. We first measure the Root Mean Square Error (RMSE) on the predicted weights to highlight the closeness between predicted and true weights. We then measure the Pearson Correlation Coefficient (PCC) on the predicted weights in order to measure the relative trend in the prediction.

As can be seen in Figure 3, LEAP based methods outperform all the baseline methods on the three datasets, hence defining the new state of the art for edge weight prediction. Moreover, with decreasing number of known edges, the performance of LEAP degrades much slower compared to the other methods.

6.3 Discussion

The results presented above on two edge learning problems prove the ability of LEAP to learn edge properties directly from the structure of the graph. Beyond this, we believe that this framework presents a unique quality of extensibility. LEAP is a highly modular system that can be adapted for any learning objective on the egde properties. In the two tasks presented in the paper, by just making a small change in the edge learner module for each task, the system was able to match the state of the art results. In doing so, it did not require any heuristics or feature extraction methods, and could learn only through the aggregation of paths between the two concerned nodes. Further, LEAP can be trained end-to-end, making it much more convenient to code, use, and maintain, as well as more interpretable since the weights throughout the network are updated using a common objective. The adaptability of LEAP also makes it a potential platform for future research in learning edge properties, and for use of neural networks in graphs.

7 CONCLUSIONS

We presented a novel end-to-end deep learning framework called LEAP for learning edge properties in a graph in this paper. LEAP includes a modular structure consisting of a path assembler, path vectorizer, and an edge learner. For any graph $G = (V, E)$ and two given nodes $(u, v) \in V$, the system learns properties associated with the edge $e_{u,v}$ by aggregating paths between them from the graph. The aggregation is performed using deep learning modules which can be selected based on the dataset and the problem under concern. The system can perform different kinds of supervised learning tasks such as binary or multi-class classification, multi-label classification and regression among others. Being powered by neural modules, the complete framework of LEAP is a layered deep learning model that can be trained end-to-end using gradient descent based methods.

We demonstrate that LEAP can obtain state-of-the-art performance for different learning problems on several real world datasets. For two specific problem of link prediction, and edge weight prediction in weighted signed networks, LEAP shows great performance by matching or improving upon the current state of the art. We also show that the LEAP framework is easily extensible, and can also incorporate node embeddings, node features and edge features into the system. We believe that this system can act as a great platform for experimentation in edge learning, and can be adapted for several different problems. We also believe that the simple architecture of LEAP allows it to be an easily deployable neural model in production environments.

REFERENCES

[1] Martín Abadi and et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Eytan Adamic and Lada A. Adar. 2003. Friends and neighbors on the web. 3 (July 2003), 211–230.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR*

abs/1409.0473 (2014). arXiv:1409.0473 http://arxiv.org/abs/1409.0473

[4] Vladimir Batagelj and Andrej Mrvar. 2006. Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[5] Phillip Bonacich. 2007. Some unique properties of eigenvector centrality. *Social Networks* 29 (2007), 555–564.

[6] Sergey Brin and Lawrence Page. 1998. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks* 30 (1998), 107–117.

[7] François Chollet et al. 2015. Keras. https://keras.io.

[8] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[9] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1024–1034. http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs.pdf

[10] Mohammad Al Hasan, Vineet Chaoji, Saeed Salem, and Mohammed Zaki. 2006. Link prediction using supervised learning. In *In Proc. of SDM 06 workshop on Link Analysis, Counterterrorism and Security.*

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9 (1997), 1735–1780.

[12] Clayton J. Hutto and Eric Gilbert. 2014. VADER: A Parsimonious Rule-Based Model for Sentiment Analysis of Social Media Text.. In *ICWSM*, Eytan Adar, Paul Resnick, Munmun De Choudhury, Bernie Hogan, and Alice H. Oh (Eds.). The AAAI Press. http://dblp.uni-trier.de/db/conf/icwsm/icwsm2014.html#HuttoG14

[13] Glen Jeh and Jennifer Widom. 2002. SimRank: A Measure of Structural-context Similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02)*. ACM, New York, NY, USA, 538–543. https://doi.org/10.1145/775047.775126

[14] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (01 Mar 1953), 39–43. https://doi.org/10.1007/BF02289026

[15] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).

[16] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[17] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and V.S. Subrahmanian. 2018. REV2: Fraudulent User Prediction in Rating Platforms. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM '18)*. ACM, New York, NY, USA, 333–341. https://doi.org/10.1145/3159652.3159729

[18] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. 2016. Edge weight prediction in weighted signed networks. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 221–230.

[19] Yann LeCun and Yoshua Bengio. 1998. Convolutional networks for images, speech, and time series.

[20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[21] David Liben-Nowell and Jon Kleinberg. 2003. The Link Prediction Problem for Social Networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM '03)*. ACM, New York, NY, USA, 556–559. https://doi.org/10.1145/956863.956972

[22] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. *CoRR* abs/1508.04025 (2015). arXiv:1508.04025 http://arxiv.org/abs/1508.04025

[23] Víctor Martínez, Fernando Berzal, and Juan-Carlos Cubero. 2016. A Survey of Link Prediction in Complex Networks. *ACM Comput. Surv.* 49, 4, Article 69 (Dec. 2016), 33 pages. https://doi.org/10.1145/3012704

[24] Aditya Krishna Menon and Charles Elkan. 2011. Link Prediction via Matrix Factorization. In *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part II (ECML PKDD'11)*. Springer-Verlag, Berlin, Heidelberg, 437–452. http://dl.acm.org/citation.cfm?id=2034117.2034146

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. (oct 2013). arXiv:1310.4546 http://arxiv.org/abs/1310.4546

[26] Abhinav Mishra and Arnab Bhattacharya. 2011. Finding the Bias and Prestige of Nodes in Networks Based on Trust Scores. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*. ACM, New York, NY, USA, 567–576. https://doi.org/10.1145/1963405.1963485

[27] Micaleah Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical review. E, Statistical, nonlinear, and soft matter physics* 74 3 Pt 2 (2006), 036104.

[28] M. E. J. Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Phys. Rev. E* 74 (Sep 2006), 036104. Issue 3. https://doi.org/10.1103/PhysRevE.74.036104

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The PageRank citation ranking: Bringing order to the Web. In *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia, 161–172. citeseer.nj.nec.com/page98pagerank.html

[30] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[31] Moshen Shahriari and Mahdi Jalili. 2014. Ranking Nodes in Signed Social Networks. *Social Network Analysis and Mining* 4, 1 (30 Jan 2014), 172. https://doi.org/10.1007/s13278-014-0172-x

[32] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2539–2561. http://dl.acm.org/citation.cfm?id=1953048.2078187

[33] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1067–1077. https://doi.org/10.1145/2736277.2741093

[34] Lei Tang and Huan Liu. 2011. Leveraging social media networks for classification. *Data Mining and Knowledge Discovery* 23, 3 (01 Nov 2011), 447–478. https://doi.org/10.1007/s10618-010-0210-x

[35] Petar VeliÄŊkoviÄĞ, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro LiÃš, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[36] Christian Von Mering, Roland Krause, Berend Snel, Michael Cornell, Stephen G. Oliver, Stanley Fields, and Peer Bork. 2002. Comparative assessment of large-scale data sets of protein-protein interactions. *Nature* 417, 6887 (23 5 2002), 399–403. https://doi.org/10.1038/nature750

[37] Hongwei Wang, Fuzheng Zhang, Min Hou, Xing Xie, Minyi Guo, and Qi Liu. 2018. SHINE: Signed Heterogeneous Information Network Embedding for Sentiment Link Prediction. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM '18)*. ACM, New York, NY, USA, 592–600. https://doi.org/10.1145/3159652.3159666

[38] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (04 June 1998), 440–442. https://doi.org/10.1038/30918

[39] Robert West, Hristo S. Paskov, Jure Leskovec, and Christopher Potts. 2014. Exploiting Social Network Structure for Person-to-Person Sentiment Analysis. *TACL* 2 (2014), 297–310.

[40] Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory Networks. *CoRR* abs/1410.3916 (2014).

[41] Muhan Zhang and Yixin Chen. 2017. Weisfeiler-Lehman Neural Machine for Link Prediction. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, USA, 575–583. https://doi.org/10.1145/3097983.3097996

[42] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 5165–5175. http://papers.nips.cc/paper/7763-link-prediction-based-on-graph-neural-networks.pdf

[43] Muhan Zhang, Zhicheng Cui, Shali Jiang, and Yixin Chen. 2018. Beyond Link Prediction: Predicting Hyperlinks in Adjacency Space. (2018). https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17136