# Semantically Enhanced Software Traceability Using Deep Learning Techniques

Jin Guo, Jinghui Cheng and Jane Cleland-Huang
University of Notre Dame
Notre Dame, IN, USA
Email: {jguo3, JinghuiCheng, JaneClelandHuang}@nd.edu

***Abstract*—In most safety-critical domains the need for traceability is prescribed by certifying bodies. Trace links are generally created among requirements, design, source code, test cases and other artifacts; however, creating such links manually is time consuming and error prone. Automated solutions use information retrieval and machine learning techniques to generate trace links; however, current techniques fail to understand semantics of the software artifacts or to integrate domain knowledge into the tracing process and therefore tend to deliver imprecise and inaccurate results. In this paper, we present a solution that uses deep learning to incorporate requirements artifact semantics and domain knowledge into the tracing solution. We propose a tracing network architecture that utilizes Word Embedding and Recurrent Neural Network (RNN) models to generate trace links. Word embedding learns word vectors that represent knowledge of the domain corpus and RNN uses these word vectors to learn the sentence semantics of requirements artifacts. We trained 360 different configurations of the tracing network using existing trace links in the Positive Train Control domain and identified the Bidirectional Gated Recurrent Unit (BI-GRU) as the best model for the tracing task. BI-GRU significantly out-performed state-of-the-art tracing methods including the Vector Space Model and Latent Semantic Indexing.**

***Keywords*-Traceability, Deep Learning, Recurrent Neural Network, Semantic Representation.**

## I. Introduction

Requirements traceability plays an essential role in the software development process. Defined as "the ability to describe and follow the life of a requirement in both a forwards and backwards direction through periods of ongoing refinement and iteration" [26], traceability supports a diverse set of software engineering activities including change impact analysis, regression test selection, cost prediction, and compliance verification [25]. In high-dependability systems, regulatory standards, such as the US Federal Aviation Authority's (FAA) DO178b/c [22], prescribe the need for trace links to be established and maintained between hazards, faults, requirements, design, code, and test cases in order to demonstrate that a system is safe for use [38], [23]. Unfortunately, the tracing task is arduous to perform and error-prone [46], even when industrial tools are used to manually create links or to capture them as a byproduct of the development process [14]. In practice, trace links are often incomplete and inaccurate [16], even in safety-critical systems [45], [56].

To address these problems, researchers have proposed and developed solutions for automating the task of creating and maintaining trace links [1], [17], [34]. Solutions have included information retrieval approaches [17], [18], [5], machine learning [47], [30], [30], [51], heuristic techniques [64], [28], and AI swarming algorithms [65]. Other approaches, especially in the area of feature location [20], require additional information obtained from runtime execution traces. Results have been mixed, especially when applied to industrial-sized datasets, where acceptable recall levels above 90% can often only be achieved at extremely low levels of precision [43].

One of the primary reasons that automated approaches have underperformed is the term mismatch that often exists between pairs of related artifacts [10]. To illustrate this we draw on an example from the Positive Train Control (PTC) domain. PTC is a communication-based train control system designed to ensure that trains follow directives in order to prevent accidents from occurring [68]. The requirement stating that *"The BOS Administrative Toolset shall allow the Authorized Administrator to view an On-board's last reported On-board Software Version, including the associated repository name, MD5, and whether the fileset is preferred or acceptable."* is associated with the design artifact stating that *"The Operational Data Panel is used to provide information about the current PTC operations in a subdivision"*. Recognizing and establishing this link requires non-trivial knowledge of domain concepts – for example, understanding that BOS Administrative Toolset contains the Operational Data Panel, each locomotive contains an On-board unit for PTC operation, and that the Operational Data Panel displays the information of locomotives such as the On-board Software Version to the BOS Authorized Administrator. This link would likely be missed by popular trace retrieval algorithms such as the Vector Space Model (VSM), Latent Semantic Indexing (LSI), and Latent Direchlet Allocation (LDA), which all represent artifacts as bags of words and therefore lose the artifacts' embedded semantics. It would also be missed by techniques that incorporate phrasing without understanding their conceptual associations [3], [15]. In fact, most current techniques lack the sophistication needed to reason about semantic associations between artifacts and therefore fail to establish trace links when there is little meaningful overlap in use of terms.

In our prior work we developed *Domain-Contextualized Intelligent Traceability* (DoCIT) [29] as a proof of concept solution to investigate the integration of domain knowledge into the tracing process. We demonstrated that for the domain of PTC systems DoCIT returned accurate trace links achieving mean average precision (MAP) of .822 in comparison to .590 achieved using VSM. However, the cost of setting up DoCIT for a domain was non-trivial, as it required carefully handcrafting a domain ontology and manually defining trace link heuristics capable of reasoning over the semantics of the artifacts and associated domain knowledge. Furthermore, DoCIT depended upon a conventional syntactic parser to analyze the artifacts in order to extract meaningful concepts. The approach was therefore sensitive to errors in the parser, terms missing from the ontology, and missing or inadequate heuristics. As such, DoCIT was effective but fragile, and would require significant effort to transfer into new project domains.

On the other hand, deep learning techniques have successfully been applied to solve many Natural Language Processing (NLP) tasks including parsing [63], sentiment analysis [66], question answering [35], and machine translation [6]. Such techniques abstract problems into multiple layers of nonlinear processing nodes; they leverage either supervised or unsupervised learning techniques to automatically learn a representation of the language and then use this representation to perform complex NLP tasks. The goal of the work described in this paper is to utilize deep learning to deliver a scalable, portable, and fully automated solution for bridging the semantic gap that currently inhibits the success of trace link creation algorithms. Our solution is designed to automate the capture of domain knowledge and the artifacts' textual semantics with the explicit goal of improving accuracy of the trace link generation task.

The approach we propose includes two primary phases. First, we learn a set of **word embeddings** for the domain using an unsupervised learning approach trained over a large set of domain documents. The approach generates high dimensional word vectors that capture distributional semantics and co-occurrence statistics for each word [53]. Second, we use an existing training set of validated trace links from the domain to train a Tracing Network to predict the likelihood of a trace link existing between two software artifacts. Within the tracing network, we adopt a **Recurrent Neural Network** architecture to learn the representation of artifact semantics. For each artifact (i.e. each regulation, requirement, or source code file etc.), each word is replaced by its associated vector representation learned in the word embedding training phase and then sequentially fed into the RNN. The final output of RNN is a vector that represents the semantic information of the artifact. The tracing network then compares the semantic vectors of two artifacts and outputs the probability that they are linked.

Given the need for an initial training set of trace links, our approach cannot be used in an entirely green field domain. However, based on requests from our industrial collaborators, we envision the following primary usage scenarios: (1) Train the tracing network on an initial set of manually constructed trace links for a project and then use it to automate the production of other links as the project proceeds; (2) Train the tracing network on the complete set of trace links for a project and then use it to find additional links that may have been missed during the manual link construction process; and finally (3) Train the tracing network on the trace links for one project, or for specific types of artifacts in one project, and then apply it to other projects and artifact types within the same domain. In this paper we focus on the first scenario.

We evaluate our approach on a large industrial dataset taken from the domain of PTC systems to address two research questions:

**RQ1:** How should RNN be configured in order to generate the most accurate trace links?

**RQ2:** Is RNN able to significantly improve trace link accuracy in comparison to standard baseline techniques?

The remainder of the paper is structured as follows. We first introduce deep learning techniques related to the tracing network in Section II. The architecture of the tracing network is described in Section III. Sections IV and V describe the process used to configure the tracing network, our experimental design, and the results achieved. Finally, in Sections VI to VIII we discuss related work, threats to validity, and conclusions.

## II. Deep Learning for Natural Language Processing

Many modern deep learning models and associated training methods originated from research in artificial neural networks (ANN). Inspired by advances in neuroscience, ANNs were designed to approximate complex functions of the human brain by connecting a large number of simple computational units in a multi-layered structure. Based on ANNs, *Deep Learning* models feature more complex network connections in a larger number of layers. A benefit gained from a more complex structure is the ability to represent data features with multiple levels of abstraction; this is usually preferable to more traditional machine learning techniques, in which human expertise is needed to select features of data for training. Backpropagation [58] is widely recognized as an effective method for training deep neural networks; it indicates how the network should adapt its internal parameters to better compute the representation in each layer. Before presenting our approach, we describe fundamental concepts of deep learning techniques, especially as related to NLP tasks. Furthermore, as our interest lies in comparatively evaluating different models for purposes of trace link creation, we describe these various techniques in some depth.

### *A. Word Embedding*

Conventional NLP and information retrieval techniques treat unique words as atomic symbols and therefore do not take associations among words into account. To address this limitation, word embedding learns the representation of each word from a corpus as a continuous high dimensional vector, such that similar words are close together in the vector space. In addition, the embedded word vectors encode syntactic and semantic relationships between words as linear relationships between word vectors [48]. The use of learned word vectors is considered one of the primary reasons for the success of recent deep learning models for NLP tasks [21].

Skip-gram with negative sampling [48], [50] and GloVe [53] are the most popular word embedding models due to the notable improvement they bring to word analogy tasks over more traditional approaches such as Latent Semantic Analysis [48], [53]. Word embedding models are trained using unlabeled natural language text by utilizing co-occurrence statistics of words in the corpus. The Skip-gram model scans context windows across the entire training text to train prediction models [48]. Given the *center* word in the window of size $T$, this model maximizes the probability that the targeted word appears around the center word, while minimizing the probability that a random word appears around the center word. The GloVe model uses matrix factorization; however we do not discuss it further because its performance is equivalent to the Skip-gram with negative sampling approach while it is less robust and utilizes more system resources [41].

### *B. Neural Network Structures*

Deep learning for NLP tasks are typically addressed using neural network techniques. *Feedforward* networks, also referred to as multi-layer perceptrons (MLPs), represent a traditional neural network structure and lay the foundation for many other structures [32]. However, the number of parameters in a fully connected MLP can grow extremely large as the width and depth of the network increases. To address this limitation, researchers have proposed various neural network structures targeting different types of practical problems. For example, convolutional neural networks (CNNs) are especially well suited for image recognition and video analysis tasks [40]. For NLP tasks, Recurrent neural networks (RNNs) are widely used and are recognized as a good fit to the unique needs of NLP [58]. In particular, RNN and its variants have produced significant breakthroughs in many NLP tasks including language modeling [49], machine translation [6], and semantic entailment [66]. In the following sections, we first introduce background about RNN and then discuss several RNN variants that were evaluated in this study.

### *C. Standard Recurrent Neural Networks (RNN)*

RNNs are particularly well suited for processing sequential data such as text and audio. They connect computational units of the network in a directed cycle such that at each time step $t$, a unit in the RNN not only takes input of the current step (i.e., the next word embedding), but also the hidden state of the same unit from the previous time step $t-1$. This feedback mechanism simulates a "memory", so that a RNN's output is determined by both its current and prior inputs. Furthermore, because RNNs use the same unit (with the same parameters) across all time steps, they are able to process sequential data of arbitrary length. This is illustrated in Figure 1. At a given time step $t$ with input vector $x_t$ and its previous hidden output vector $h_{t-1}$, a standard RNN unit calculates its output as

$$h_t = tanh(Wx_t + Uh_{t-1} + b) \tag{1}$$

where $W$, $U$ and $b$ are the affine transformation parameters, and $tanh$ is the hyperbolic tangent function: $tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$.

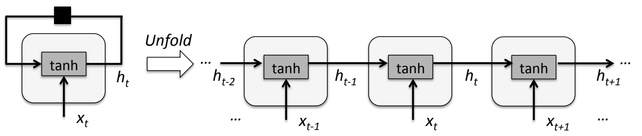


Fig. 1: Standard RNN model (left) and its unfolded architecture through time (right). The black square in the left figure indicates a one time step delay.

A prominent drawback of the standard RNN model is that the network degrades when long dependencies exist in the sequence due to the phenomenon of exploding or vanishing gradients during back-propagation [9]. This makes a standard RNN model difficult to train. The exploding gradients problem can be effectively addressed by scaling down the gradient when its norm is bigger than a preset value (i.e. *Gradient Clipping*) [9]. To address the vanishing gradients problem of the standard RNN model, researchers have proposed several variants with mechanics to preserve long-term dependencies; these variants included Long Short Term Memory (LSTM) and the Gated Recurrent Unit (GRU).

### *D. Long Short Term Memory (LSTM)*

LSTM networks include a memory cell vector in the recurrent unit to preserve long term dependencies[33]. LSTM also introduces a gating mechanism to control when and how to read or write information to the memory cell. A gate in LSTM usually uses a sigmoid function $\sigma(z) = 1/(1 + e^{-z})$ and controls information throughput using a point-wise multiplication operation $\odot$. Specifically, when the sigmoid function outputs 0, the gate forbids any information from passing, while all information is allowed to pass when the sigmoid function output is 1. Each LSTM unit contains an *input* gate ($i_t$), a *forget* gate ($f_t$), and an *output* gate ($o_t$). The state of each gate is decided by $x_t$ and $h_{t-1}$ such that:

$$\begin{aligned} i_t &= \sigma(W^i x_t + U^i h_{t-1} + b^i) \\ f_t &= \sigma(W^f x_t + U^f h_{t-1} + b^f) \\ o_t &= \sigma(W^o x_t + U^o h_{t-1} + b^o) \end{aligned} \tag{2}$$

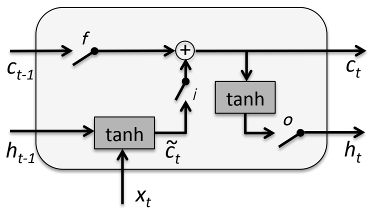
To update the information in the memory cell, a memory candidate vector $\tilde{c}_t$ is first calculated using the *tanh* function. This memory candidate passes through the input gate, which controls how much each dimension in the candidate vector should be "remembered". At the same time, the *forget* gate controls how much each dimension in the previous memory cell state $c_{t-1}$ should be retained. The actual memory cell state $c_t$ is then updated using the sum of these two parts.

$$\begin{aligned} \tilde{c}_t &= tanh(W^c x_t + U^c h_{t-1} + b^c) \\ c_t &= i_t \odot \tilde{c}_t + f_t \odot c_{t-1} \end{aligned} \tag{3}$$

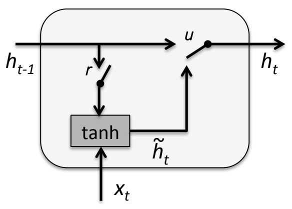
Finally, the LSTM unit calculates its output $h_t$ with an output gate as follows:

$$h_t = o_t \odot tanh(c_t) \tag{4}$$

Figure 2 (a) illustrates a typical LSTM unit. Using retained memory cell state and the gating mechanism, the LSTM unit "remembers" information until it is erased by the *forget* gate;



(a) LSTM



(b) GRU

Fig. 2: Comparison between single units from LSTM and GRU networks. In (2a), $i$, $f$ and $o$ are the input, forget and output gates respectively; $c$ is the memory cell vector. In (2b), $r$ is the reset gate and $u$ is the update gate [13].

as such, LSTM handles long-term dependencies more effectively. LSTM has been repeatedly applied to solve semantic relatedness tasks and has achieved convincing performance [66], [57]. These advances motivated us to adopt LSTM for reasoning semantics in the tracing task.

### E. Gated Recurrent Unit (GRU)

Finally, the recently proposed Gated Recurrent Unit (GRU) model also uses a gating mechanism to control the information flow within a unit; but it has a simplified unit structure and does not have a dedicated memory cell vector [12]. It contains only a *reset* gate $r_t$ and an *update* gate $u_t$:

$$\begin{aligned} r_t &= \sigma(W^r x_t + U^r h_{t-1} + b^r) \\ u_t &= \sigma(W^u x_t + U^u h_{t-1} + b^u) \end{aligned} \tag{5}$$

In GRU networks, the previous hidden output $h_{t-1}$ goes through the *reset* gate $r_t$ and is sent back to the unit. An output candidate $\tilde{h}_t$ is then calculated using the gated $h_{t-1}$ and the unit's current input $x_t$ as follows:

$$\tilde{h}_t = \tanh(W^h x_t + U^h(r_t \odot h_{t-1}) + b^h) \tag{6}$$

The actual output of a unit $h_t$ is a linear interpolation between the previous output $h_{t-1}$ and the candidate output $\tilde{h}_t$, controlled by the *update* gate $u_t$. As such, the *update* gate balances how much of the current output is updated using $\tilde{h}_t$ and $h_{t-1}$.

$$h_t = (1 - u_t) \odot h_{t-1} + u_t \odot \tilde{h}_t \tag{7}$$

Consequently, a GRU unit embeds long-term information directly into the hidden output vectors. Figure 2 compares the unit structures of LSTM and GRU networks. Despite having a simpler structure, GRU has achieved competitive results with LSTM for many NLP tasks [6], [13], and as such no decisive conclusion has been drawn about which model is better. In this work, we compare the performance of LSTM and GRU in order to identify the most suitable model for addressing the tracing problem.

### F. Other RNN Variables

In addition to modifying the structure within a single RNN unit, the structure of the overall RNN network can be varied. For instance, multi-layered RNNs [52] stack more than one RNN unit at each time step [59] with the aim of extracting more abstract features from the input sequence. In contrast, bidirectional RNNs [60] process sequential data in both forward and backward directions at the same time; this enables the output to be influenced by both past and future data in the sequence. In this study, we also explored the use of two-layered RNNs and bidirectional RNNs for generating trace links.

## III. The Tracing Network

The tracing process comprises several steps: first, the human analyst initiates a trace for a source artifact; second, the similarity between the source artifact and each of the potentially linked target artifacts is computed; third, a list of ranked candidate links are returned; and finally the human evaluates the links and accepts the ones deemed to be correct. The process is repeated for all source artifacts [34]. Out study investigates the effectiveness of various deep learning models and methods for calculating similarities between source and target artifact pairs, with the goal of generating accurate trace links. This is essentially a textual comparison task in which the tracing network needs to leverage domain knowledge to understand the semantics of two individual artifacts and then to evaluate their semantic relatedness for tracing purposes. Valid associations need to be established between related artifacts even when no common words are present. Based on our initial analysis of the strengths and weaknesses of current techniques, we decided to adopt word embeddings and RNN techniques to achieve this goal. Therefore, we first need to learn word embeddings from a domain corpus in order to effectively encode word relations, and then utilize such word embeddings in the tracing network structure to extract and compare their semantics. In this section, we describe our approach for designing and training such a tracing network.

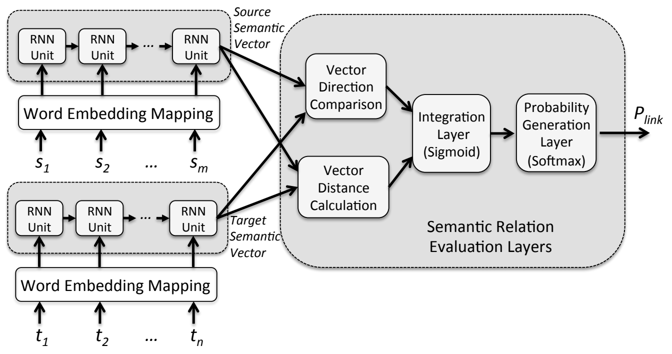


Fig. 3: The architecture of the tracing network. The software artifacts are first mapped into sequences of embedded word vectors and go through RNN layers to generate the semantic vectors, which are then fed into the Semantic Relation Evaluation layers to predict the probability that they are linked.

### A. Network Architecture

The design of the neural network architecture is shown in Figure 3. Given the textual content of a source artifact $A_s$ and a target artifact $A_t$, each word in $A_s$ and $A_t$ is first mapped onto its vector representation through the *Word Embedding* layer. Such mappings are trained from the domain corpus using the Skip-gram model introduced in Section II-A. The vectors of words in source artifact $s_1, s_2, \ldots, s_m$ are then sent to the *RNN* layers sequentially and output as a single vector $v_s$ representing its semantic information. In the case of the bidirectional-RNN, the word vectors are also sent in reverse order as $s_m, s_{m-1}, \ldots, s_1$. The target semantic vector $v_t$ is generated in the same way using RNN layers. Finally, these two vectors are compared in the *Semantic Relation Evaluation* layers.

The *Semantic Relation Evaluation* layers in our tracing network adopt the structure proposed by Tai et al. [66], targeted to perform semantic entailment classification tasks for sentence pairs. The overall calculation of this part of the network can be represented as:

$$\begin{aligned} r_{pmul} &= v_s \odot v_t \\ r_{sub} &= |v_s - v_t| \\ r &= \sigma(W^r r_{pmul} + U^r r_{sub} + b^r) \\ p_{tracelink} &= softmax(W^p r + b^p) \end{aligned} \tag{8}$$

where

$$softmax(z)_j = e^{z_j} / \sum_{k=1}^{K} e^{z_k}, for j = 1, \ldots, K \tag{9}$$

Here, $\odot$ is the point-wise multiplication operator used to compare the *direction* of source and target vectors on each dimension. The absolute vector subtraction result, $r_{sub}$, represents the *distance* between the two vectors in each dimension. The network then uses a hidden *sigmoid layer* to integrate $r_{pmul}$ and $r_{sub}$ and output a single vector to represent their semantic similarity. Finally, the output *softmax layer* uses the result to produce the probability that a valid trace link exists; the result of a softmax function is a K-dimensional vector of real values in the range $(0, 1)$ that add up to 1 (K=2 in this case).

A concrete tracing network is built upon this architecture and is further configured by a set of network settings. Those settings specify the type of RNN unit (i.e. GRU or LSTM), the number of hidden dimensions in RNN units and the Semantic Relation Evaluation layers, and other RNN variables such as the number of RNN layers and whether to use bidirectional RNN. To address our first research question (RQ1) we explored several different configurations. We describe how we optimized network settings in Section IV-B. The tracing network is implemented on the Torch framework (http://torch.ch), and the source code is available at https://github.com/jin-guo/TraceNN.

### B. Training the Tracing Network

A powerful network is only useful when it can be properly trained using existing data and when it is generalizable to unseen data. To train the tracing network, we use the regularized negative log likelihood as our objective loss function to be minimized. This objective function is commonly used in categorical prediction models [7] and can be written as:

$$J(\theta) = -\frac{1}{N} \sum_{i=1}^{N} logP(Y = y^i | x^i, \theta) + \frac{\lambda}{2} ||\theta||_2^2 \tag{10}$$

where $\theta$ indicates the network parameters that need to be trained, $N$ is the total number of examples in the training data, $x^i$ is the input of the *ith* training example, $y^i$ is the actual category of that example (i.e. link or non-link); as a result, $P(Y = y^i | x^i, \theta)$ represents the network's prediction on the correct category given the current input and parameters. The second part of the loss function represents a $L^2$ parameter regularization that prevents overfitting, where $||\theta||_2$ is the Euclidean norm of $\theta$, and $\lambda$ controls the strength of the regularization.

Based on this loss function, we used a stochastic gradient descent method [67] to update the network parameters. According to this method, a typical training process is comprised of a number of epochs. Each *epoch* iterates through all of the training data one time to train the network; as such, the overall training process uses all the training data several times until the objective loss is sufficiently small or fails to decrease further. In each epoch, the training data is further randomly divided into a number of "mini batches;" each contains one or more training datapoints. After each batch is processed, a gradient of parameters is calculated based on the loss function. The network then updates its parameters based on this gradient and a "learning rate" that specifies how fast the parameters move along the gradient direction. During training, we adopted an adaptive learning rate procedure [67] to adjust the learning rate based on the current training performance. To help the network converge, we also decreased the learning rate after each epoch until epoch $\tau$, such that the learning rate at epoch $k$ is determined by:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{11}$$

where $\epsilon_0$ is initial learning rate, $\alpha = k/\tau$. In our experiment, $\epsilon_\tau$ is set to $\epsilon_0/100$, and $\tau$ set to 500.

Based on these general methods, a tracing network training process is further determined by a set of predefined *hyper-parameters* that help steer the learning behavior. Common hyper-parameters include the initial learning rate, gradient clip value, regulation strength ($\lambda$), the number of datapoints in a mini batch (i.e. mini batch size), and the number of epochs included in the training process. The techniques for selecting hyper-parameters are described in Section IV-B.

## IV. Experiment Setup

In this section, we explain the methods used to (1) prepare data, (2) systematically tune the configuration (i.e. network settings and hyper-parameters) of the tracing network, and (3) compare the performance of the best configuration against other popular trace evaluation methods.

### A. Data Preparation

To train the word embeddings, we used a corpus from the PTC domain that is comprised of 52.7MB of clean text extracted from related domain documents and software artifacts. The original corpus of domain documents was collected from the Internet by our industrial collaborators as part of their initial domain analysis process. We also added the latest Wikipedia dump containing about 19.92GB of clean text to the corpus and used it for one variant of the word embedding configuration. All documents were preprocessed by transforming characters to lower-case and removing all non-alpha-numeric characters except for underscores and hyphens.

To train and evaluate other parts of the tracing network, we used PTC project data provided by our industrial collaborators. The dataset contains 1,651 Software Subsystem Requirements (SSRS) as source artifacts and 466 Software Subsystem Design Descriptions (SSDD) as target artifacts. Each source artifact contained an average of 33 tokens and described a functional requirement of the Back Office Server (BOS) subsystem. Each target artifact contained an average of 99 tokens and specified design details. There were 1,387 trace links between SSRS and SSDD artifacts, all of which were constructed and validated by our industrial collaborators. This dataset is considerably larger than those used in most previous studies on requirements traceability [5], [44], [34] and the task of creating links across such a large dataset represents a challenging industrial-strength tracing problem. We randomly selected 45% of the 769,366 artifact pairs from the PTC project dataset (i.e. 1,651 $\times$ 466) for inclusion in a training set, 10% for a development set, and 45% as a testing set. Given a fixed tracing network configuration, the training set was used to update the network parameters (i.e. the weight and bias for affine transformation in each layer) in order to minimize the objective loss function. The development set was used to select the best general model during an initial training process to ensure that the model was not overtrained. The test data was set aside and only used for evaluating the performance of the final network model.

Software project data exhibits special characteristics that impact the training of a neural network [30]. In particular, the number of actual trace links is usually very small for a given set of source and target artifacts compared to the total number of artifact pairs. In our dataset, among all 769,366 artifact pairs, only 0.18% are valid links. Training a neural network using such an unbalanced dataset is very challenging. A common and relatively simple approach for handling unbalanced datasets is to weight the minority class (i.e. the linked artifacts) higher than the majority class (i.e. the non-linked ones). However, in the gradient descent method, a larger loss weighting could improperly amplify the gradient update for the minority class making the training unstable and causing failure to converge. Another common way to handle unbalanced data is to downsample the majority case in order to produce a fixed and balanced training set. Based on initial experimentation we found that this approach did not yield good results because the examples of non-links used for training the network tended to randomly exclude artifact pairs that lay at the frontier at which links and non-links are differentiated. Furthermore, based on initial experimentation, we also ruled out the upsampling method because this considerably increased the size of the training set, excessively prolonging the training time.

Based on further experimentation we adopted a strategy that dynamically constructed balanced training sets using sub-datasets. In each epoch, a balanced training set was constructed by including all valid links from the original training set as well as a randomly selected equal number of non-links form the training set. The selection of non-links was updated at the start of each epoch. This approach ensured that over time the sampled non-links used for training were representative and preserved an equal contribution of links and non-links during each epoch. Our initial experimental results showed this technique to be effective for training our tracing network.

TABLE I: Tracing Network Configuration Search Space

| | |
|---|---|
| Word Embedding Source | PTC docs – 50 dim,<br>PTC docs + Wikipedia dump – 300 dim |
| RNN Unit Type | GRU, LSTM, BI-GRU, BI-LSTM,<br>(AveVect as baseline) |
| RNN Layer | 1, 2 |
| Hidden Dimension | RNN30 + Intg10, RNN60 + Intg20 |
| Init Learning Rate (*lr*) | 1e-03, 1e-02, 1e-01 |
| Gradient Clip Value (*gc*) | 10, 100 |
| Regularization Strength $\lambda$ | 1e-04, 1e-03 |
| Mini Batch Size | 1 |
| Epoch | 60 |

### *B. Model Selection and Hyper-Parameters Optimization*

Finding suitable network settings and a good set of hyper-parameters is crucial to the success of applying deep learning methods to practical problems [54]. However, given the running time required for training, the search space of all the possible combinations of different configurations was too large to provide full coverage. We therefore first identified several configurations that were expected to produce good performance. This was accomplished by manually observing how training loss changed during early epochs and following heuristics suggested in [8]. We then created a network configuration search space centered around these manually identified configurations; our search space is summarized in table I. We conducted a *grid search* and trained all the combinations of each configuration in Table I using the training set and then compared their performance on the development set to find the best configuration. We describe our search space below.

For learning the word embeddings, we used the Skip-gram model provided by the Word2vec tool [48]. We trained the word vectors with two settings: 50-dimension vectors using the PTC corpus only and 300-dimension using both PTC and Wikipedia dump. The number of dimensions are set differently because the PTC corpus (38,771 tokens) contains considerably less tokens than the PTC + Wikipedia dump (8,025,288 tokens). While a smaller vector dimension would result in faster training, a larger dimension is needed to effectively represent the semantics of all tokens in the latter corpus.

To compare which variation of RNN best suits the tracing network, we evaluated GRU, LSTM, bi-directional GRU (BI-GRU), bi-directional LSTM (BI-LSTM) with both 1 and 2 layers introduced in Section II-C. The hidden dimensions in each RNN unit were set to either 30 or 60, while the hidden dimensions for the Integration layer were set to 10 or 20 correspondingly. As a baseline method, we also replaced the RNN layers with a bag-of-word method in which the semantic vector of an artifact is simply set to be the average of all word vectors contained in the artifact ("AveVect" in Table I). We also summarize the search space for other hyper-parameters of the tracing network in Table I.

## C. Comparison of Tracing Methods

In practical requirements tracing settings, a tracing method returns a list of candidate links between a source artifact, serving the role of user query, and a set of target artifacts. An effective algorithm would return all valid links close to the top of the list. The effectiveness of a tracing algorithm is therefore often measured using Mean Average Precision (MAP). To calculate MAP, we first calculate the Average Precision (AP) for each individual query as:

$$AP = \frac{\sum_{i=1}^{|Retrieved|} (Precision(i) \times relevant(i))}{|RelevantLinks|} \qquad (12)$$

where $|RetrievedLinks|$ is the number of retrieved links, $i$ is the rank in the sequence of retrieved candidates links, $relevant(i)$ is a binary function assigned 1 if the link is valid and 0 otherwise, and $Precision(i)$ is the precision computed after truncating the list immediately below $i$. Then, Mean Average Precision (MAP) is computed as the mean AP across all queries. In typical information retrieval settings, MAP is computed for the top N returned links; however, for traceability purposes we compute it when returning **all** valid links as specified in the trace matrix. This means that our version of MAP is computed for recall of 100%.

We computed MAP using the test dataset only, and compared the performance of our tracing network with other popular tracing methods, i.e. Vector Space Model (VSM) and Latent Semantic Indexing (LSI). To make a fair comparison, we also optimized the configurations for the VSM and the LSI methods using a Genetic Algorithm to search through an extensive configuration space of preprocessors and parameters [43]. Finally, we configured VSM to use a local Inverse Document Frequency (IDF) weighting scheme when calculating the cosine similarity [34]. LSI was reduced to 75% dimensions. For both VSM and LSI we preprocessed the text to remove non alpha-numeric characters, remove stop words, and to stem each word using Porter's stemming algorithm.

We also evaluated the results by plotting a precision vs. recall curve. The graph depicts *recall* and *precision* scores at different similarity or probability values. The Precision-Recall Curve thus shows trade-offs between precision and recall and provides insights into where each method performs best – for example, whether a technique improves precision at higher or lower levels of recall [11]. A curve that is farther away from the origin indicates better performance.

## V. Results and Discussion

In this section, we report (1) the best configurations found in our network configuration search space, (2) the performance of the tracing network with the best configuration compared against VSM and LSI, and (3) the performance of the tracing network when trained with a larger training set of data.

### A. What is the best configuration for the tracing network?

This experiment aims to address the first research question (RQ1). When optimizing the network configuration of the tracing network, we first selected the best configuration for each RNN unit type; Table II summarizes these results. We found that the best configurations for all four RNN unit types were very similar: one layer RNN model with 30 hidden dimensions and an Integration layer of 10 hidden dimensions, learning rate of 1e-02, gradient clip value of 10, and $\lambda$ of 1e-04. Performance varies for different RNN unit types.

TABLE II: Best Configuration for Each RNN Unit Type

| RNN Unit | Dev. Loss | Word Emb. | $L$ | $D_r$ | $D_s$ | $lr$ | $gc$ | $\lambda$ |
|---|---|---|---|---|---|---|---|---|
| BI-GRU | .1045 | PTC | 1 | 30 | 10 | .01 | 10 | .0001 |
| GRU | .1301 | PTC | 1 | 30 | 10 | .01 | 10 | .0001 |
| BI-LSTM | .1434 | PTC | 1 | 30 | 10 | .01 | 100 | .0001 |
| LSTM | .2041 | PTC+Wiki | 1 | 30 | 10 | .01 | 10 | .0001 |

$L$ – number of layers in the RNN model, $D_r$ – hidden dimension in RNN unit, $D_s$ – hidden dimension in Integration layer, $lr$ – initial learning rate, $gc$ – gradient clipping value, $\lambda$ – regularization strength

Figure 4 illustrates the learning curves on the training dataset of the four configurations. All four RNN unit types outperformed the Average Vector method. This supports our hypothesis that word order plays an important role when comparing the semantics of sentences. Both GRU and BI-GRU achieved faster convergence and more desirable (smaller) loss than LSTM and BI-LSTM. Although quite similar, the bidirectional models performed slightly better than the unidirectional models for both GRU and LSTM on the training set; the bidirectional models also achieved better results on the development dataset compared to their unidirectional counterparts. As a result, the overall best performance was achieved using **BI-GRU** configured as shown in Table II.

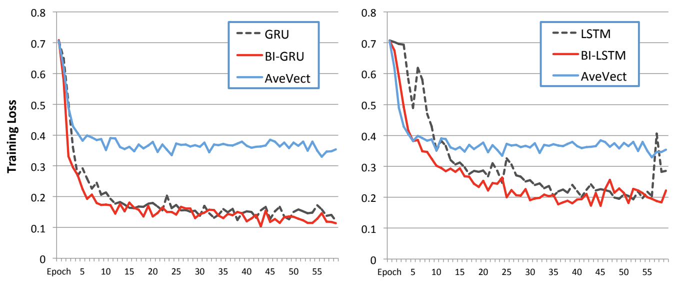


Fig. 4: Comparison of learning curves for RNN variants using their best configurations. GRU and BI-GRU (left) converged faster and achieved smaller loss than LSTM and BI-LSTM (right). They all outperformed the baseline.

We also found that in three of the four best configurations, the word embedding vectors were trained using the PTC corpus alone. We speculate that one reason the PTC-trained word vectors performed better than the PTC+Wiki-trained vectors is due to differences in content of the two corpora. The PTC+Wiki corpus contains significantly more words that are used in diverse contexts because the majority of articles in the Wiki corpus are not related to the PTC domain. In the case of words that appear commonly in Wiki articles but convey specific meanings in the PTC domain (e.g. message, administrator, field, etc.), their context in more general articles

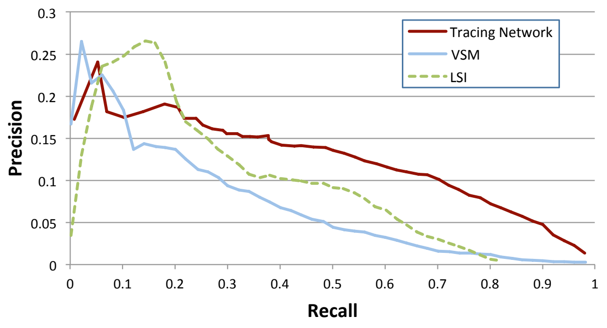


Fig. 5: Precision-Recall Curve on test set – 45% total data

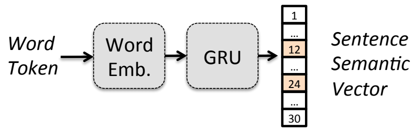
is likely to negatively affect the reasoning task on domain specific semantics when there is insufficient training data to disambiguate their usage. We also tested the tracing network performance using 300-dimension word embedding trained by the PTC corpus. The result is similar to the best configuration found in Table II. These findings suggest that using only the domain corpus to train word vectors with a reasonable size is more computationally economical and can yield better results.

## *B. Does the tracing network outperform leading trace retrieval algorithms?*

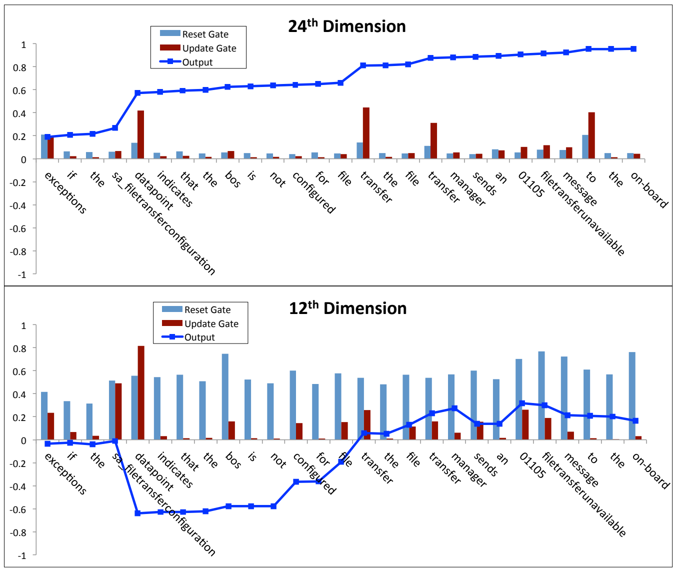
We now evaluate whether the best configuration of the tracing network (i.e. BI-GRU with configuration shown in Table II) outperforms leading trace retrieval algorithms. Links were therefore generated for each source and target artifact pair in the test set (45% total data) and for each source artifact ranked by descending probability scores. Average Precision ($AP$) was calculated using Equation 12. We then compared the APs for our tracing network against those generated using the best performing VSM and LSI configurations. Our tracing network was able to achieve a MAP of .598; this value is 41% higher than that achieved using VSM ($MAP$ = .423) and 32% higher than LSI ($MAP$ = .451). We conducted a Friedman test and found a statistically significant difference among the AP values associated with the three methods ($\mathcal{X}^2(2) = 89.40$, $p < .001$). We then conducted three pairwise Wilcoxon signed ranks tests with Bonferroni p-value adjustments. Results indicated that the APs associated with our tracing network were significantly higher ($M = .598$, $SD = .370$) than those achieved using VSM ($M = .423$, $SD = .391$; $p < .001$) and LSI ($M = .451$, $SD = .400$; $p < .001$); in contrast, there was no significant difference when comparing APs for VSM versus those for LSI.

When comparing the Precision-Recall Curves for the three methods (Figure 5), we observed that the tracing network outperformed VSM and LSI at higher levels of recall. Given the goal to achieve close to 100% recall when performing tracing tasks, this is an important achievement. Precision improved notably when recall was above 0.2. This improvement can be attributed to the fact that the tracing network is able to extract semantic information from artifacts and to reason over associations between them.

While it is almost impossible to completely decipher how GRU extracts a semantic vector from natural language [37], observing gate behavior when GRU processes a sentence can provide some insights into how GRU performs this task so well. As an example, we examine the gate behavior when our best performing GRU processes the following artifact text: *"Exceptions: If the SA_FileTransferConfiguration datapoint indicates that the BOS is not configured for file transfer, the File Transfer Manager sends an 01105_FileTransferUnavailable message to the Onboard."*. The gate behavior varies among dimensions in the sentence semantic vector. In Figure 6, we show the levels of the reset and the update gates and the change of the output values on two of the 30 dimensions after GRU takes each word from this artifact. Unlike LSTM, the GRU does not have an explicit memory cell; however, the unit output itself can be considered as a *persisting memory*, whereas the output candidate in Equation 6 acts as a *temporary memory*. Recall that the **reset gate** controls how much previous output adds to the current input when determining the output candidate (i.e. temporary memory); in contrast, the **update gate** balances the contributions of this temporary memory and the previous output (i.e. persisting memory) to the actual current output. From Figure 6b, we observe that for the 24th dimension, the reset gate is constantly small. This means that the temporary memory is mostly decided by the current input word. The



(a) GRU output at one time step



(b) GRU gate behavior

Fig. 6: The reset gate and update gate behavior in GRU and their corresponding output on the 24th and 12th dimension of sentence semantic vector. The x-axis indicates the sequential input of words while the y-axis is the value of reset gate, update gate and output after taking each word.

update gate for this dimension was also small for most of the input words until the words *datapoint*, *transfer* and *to* came. Those spikes indicate moments when the temporary memory was integrated into the persisting memory. As such, we can speculate that this semantic vector dimension functions to accumulate information from specific keywords across the entire sentence. Conversely, for the 12th dimension, the reset gate was constantly high, indicating that the information stored in the temporary memory was based on both previous output and current input. Therefore, the actual output is more sensitive to local context rather than a single keyword. This is confirmed by the fluctuating shape of the actual output shown in the figure. For example, the output value remained in the same range until the topic was changed from "datapoint indication" to "message transfer". We believe that this versatile behavior of the gating mechanism might enable GRU to encode complex semantic information from a sentence into a vector to support trace link generation and other challenging NLP tasks.

From Figure 5, we also notice that the tracing network hits a glass ceiling for improving precision above 0.27. We consider this to be caused by its inability to rule out some false positive links that contain valid associations. For example, the tracing network assigns a 97.27% probability of a valid link between artifact *"The BOS administrative toolset shall allow an authorized administrator to view internal errors generated by the BOS"* and the artifact *"The MessagesEvents panel provides the functionality to view message and event logs. The panel provides searching and filtering capabilities where the user can search by a number of parameters depending on the type of data the user wants to view"*. There are direct associations between these artifacts: MessagesEvents panel is part of the BOS administrative toolset for viewing message and event log, administrator is a system user and internal errors generated by the BOS is an event. But this is not a valid link because MessagesEvents panel only displays messages and events related to external Railroad Systems rather than to internal events. It is likely that the tracing network fails to exclude this link because it has not been exposed to sufficient similar negative examples in the training data. As we described in Section IV-A, every positive example is used while negative examples (i.e. non-links) are randomly selected during each epoch. We plan to explore more adequate methods for handling unbalanced data problems caused by characteristics of the tracing data in our future work.

### C. *How does the tracing network react to more training data?*

The number of trace links tends to increase as a software project evolves. To explore the potential impact of folding them into the training set, we increased the training dataset to 80%. We randomly selected part of the test data and moved it into the training set to reach 80%, while retaining the remaining data in the testing set. Using the same configuration as described in Section IV-B, we then retrained the tracing network. Because the size of the test set decreased to 10%, we could not make direct comparisons to our previous results. Instead we used both of the trained tracing networks (i.e. trained with 45% and 80% of the data respectively) to generate trace links against the same small test set (sized at 10%). To reduce the effect of random data selection, we repeated this process five times and report the average results.

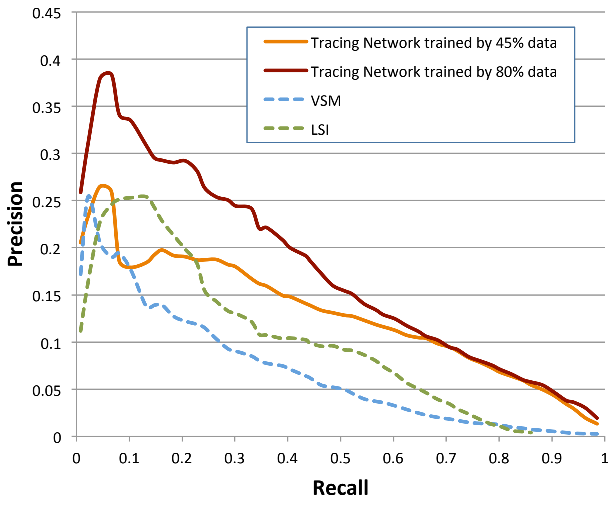


Fig. 7: Precision-Recall Curve on test set – 10% total data.

With a larger training set, the *MAP* was .834 compared to .803 for the smaller training set. Results are depicted in the Precision-Recall Curve in Figure 7. With increased training data, the network can better differentiate links and non-links, and therefore improve both precision and recall. Improvements were observed especially at low levels of recall.

The performance of the tracing network trained using 80% of data was again compared against VSM and LSI for this larger training set. A Friedman test identified a statistically significant difference among the APs associated with the three methods on this new dataset division ($\mathcal{X}^2(2) = 141.11$, $p < .001$). Using pairwise Wilcoxon signed ranks tests with Bonferroni p-value adjustments, we found that our tracing network performed significantly better (***MAP* = .834**) than VSM (***MAP* = .625**; $p < .001$) and LSI (***MAP* = .637**; $p < .001$). As such, we address RQ2 and conclude that in general our tracing network improved trace link accuracy in comparison to both VSM and LSI, and that improvements were more marked as the size of the training set increased. We expect additional improvements by reconfiguring the tracing network for use with a larger training set [8]. However, we also observed that when using the original training set (i.e. 45% of data), our tracing network only outperformed LSI at higher levels of recall as shown in Figure 5 and Figure 7. In future work, we plan to explore the trade-offs between these two methods for specific data features, and to further improve the performance of the tracing network.

## VI. Related Work

In this section we focus on prior work that has integrated ontology, semantics, or NLP into the tracing process. Researchers have attempted to improve bag-of-word approaches such as the Vector Space Model (VSM) [34] by integrating matching terms, project glossaries, and other forms of thesauri [34].

Basic enhancements have included user feedback techniques such as Rocchio [34] or Direct Query manipulation (DQM) [61] to increase or decrease term weights. However, these approaches fail to leverage semantic information.

Other techniques identify terms for briding the term mismatch between source and target artifacts. Dietrich et al. utilized validated trace links to identify frequent item sets of terms occurring across pairs of source and target artifacts, and then used these to augment the text in the trace query [19]. Gibiec et al. [24] approached trace query augmentation by acquiring related documents from the Internet, and then extracting domain related terms. Researchers have also used phrase detection and chunking to search for requirements impacted by change requests [2] or to improve the trace retrieval process [70]. None of these techniques attempted to understand semantics of the artifacts.

Researchers have also explored the use of knowledge bases to create and utilize semantically aware associations in the trace creation process – where a knowledge base include basic domain terms and sentences that describe the relationships between those terms [36],[27],[62]. Data is typically represented as an ontology in which relationships are represented using *AND*, *OR*, *implication*, and *negation* operators [36]. Traceability researchers have proposed the idea of using ontology to connect source and target artifacts [31], [4]. Approaches have been proposed for weighting the evidence for a trace link according to distance between concepts in the ontology [42]. Unfortunately, building domain-specific ontologies is time consuming and ontologies are generally not available for technical software engineering domains.

Finally, while researchers have proposed techniques that more closely mimic the way human analysts reason about trace links and perform tracing tasks [28], [46], there is very limited work in this area. Our prior work with DoCIT, described in Section I, is one exception [28]. DoCIT utilizes both ontology and heuristics to reason over concepts in the domain in order to deliver accurate trace links. However, as previously explained, DoCIT requires non-trivial setup costs to build a customized ontology and heuristics for each domain, and is sensitive to flaws in syntactic parsing. In contrast, the RNN approach described in this paper requires only a corpus of domain documents and a training set of validated trace links.

On the other hand, deep learning has been successfully applied to many software engineering tasks. For example, Lam et al. combined deep neural networks with information retrieval techniques to identify buggy files in bug reports [39]. Wang et al. utilized a deep belief network to extract semantic features from source code for the purpose of defect prediction [69]. Raychev et al. adopted RNN and N-gram to build the language model for the task of synthesizing code completions [55]. We were not able to find work that applied deep learning techniques to traceability tasks in our literature review.

## VII. Threats to Validity

Two primary threats to validity potentially impact our work. First, due to the challenge of obtaining large industrial datasets including artifacts and trace links, and the time needed to experiment with different algorithms for learning word embeddings and generating trace links, our work focused on a single domain of Positive Train Control. As a result, we cannot claim generalizability. However, the PTC dataset included text taken from external regulations, and written by multiple requirements engineers, systems engineers, and developers. The threat to validity arises primarily from the possibility that characteristics of our specific dataset may have impacted results of our experiment. For example, the size of the overall dataset, the characteristics of the vocabulary used, and/or the nature of each individual artifact, may make our approach more or less effective. In the next phase of our work we will evaluate our approach on additional datasets.

Second, we cannot guarantee that the trace matrix used for evaluation is 100% correct. However, it was provided by our industrial collaborators and used throughout their project to demonstrate coverage or regulatory codes. The metrics we used (i.e. MAP, Recall, and Precision) are all accepted research standards for evaluating trace results [34]. To avoid comparison against a weak baseline, we report comparisons against two standard baselines: VSM and LSI and configured them using a Genetic Algorithm.

## VIII. Conclusions

In this paper, we have proposed a neural network architecture that utilizes word embedding and RNN techniques to automatically generate trace links. The Bidirectional Recurrent Gated Unit effectively constructed semantic associations between artifacts, and delivered significantly higher MAP scores than either VSM or LSI when evaluated on our large industrial dataset. It also notably increased both precision and recall. Given an initial training set of trace links, our tracing network is fully automated and highly scalable. In future work, we will focus on improving precision of the tracing network by identifying and including more representative negative examples in the training set.

The tracing network is currently trained to process natural language text. In future work, we will investigate techniques for applying it to other types of artifacts such as source code or formatted data. Finally, given the difficulty and limitations of acquiring large corpora of data we will investigate hybrid approaches that combine human knowledge with the neural network. In summary, the findings we have presented in this paper have demonstrated that deep learning techniques can be effectively applied to the tracing process. We see this as a non-trivial advance in our goal of automating the creation of accurate trace links in industrial-strength datasets.

## IX. Acknowledgments

The work in this paper was partially funded by the US National Science Foundation Grant CCF-1319680.

## References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] C. Arora, M. Sabetzadeh, L. C. Briand, F. Zimmer, and R. Gnaga. Automatic checking of conformance to requirement boilerplates via text chunking: An industrial case study. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, pages 35–44, 2013.

[3] C. Arora, M. Sabetzadeh, A. Goknil, L. C. Briand, and F. Zimmer. Change impact analysis for natural language requirements: An NLP approach. In *23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015*, pages 6–15, 2015.

[4] N. Assawamekin, T. Sunetnanta, and C. Pluempitiwiriyawej. Ontology-based multiperspective requirements traceability framework. *Knowl. Inf. Syst.*, 25(3):493–522, 2010.

[5] H. U. Asuncion, A. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 95–104, 2010.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[7] I. G. Y. Bengio and A. Courville. Deep learning. Book in preparation for MIT Press, 2016.

[8] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[9] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[11] M. Buckland and F. Gey. The relationship between recall and precision. *Journal of the American society for information science*, 45(1):12, 1994.

[12] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[13] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[14] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 55–69, 2014.

[15] J. Cleland-Huang and J. Guo. Towards more intelligent trace retrieval algorithms. In *(RAISE) Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, 2014.

[16] J. Cleland-Huang, M. Rahimi, and P. Mäder. Achieving lightweight trustworthy traceability. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 849–852, 2014.

[17] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *International Conference on Software Maintenance*, pages 306–315, Washington, DC, USA, 2004. IEEE Computer Society.

[18] A. Dekhtyar, J. Huffman Hayes, S. K. Sundaram, E. A. Holbrook, and O. Dekhtyar. Technique integration for requirements assessment. In *15th IEEE International Requirements Engineering Conference (RE)*, pages 141–150, 2007.

[19] T. Dietrich, J. Cleland-Huang, and Y. Shin. Learning effective query transformations for enhanced requirements trace retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 586–591, 2013.

[20] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[21] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[22] Federal Aviation Authority (FAA). *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.*

[23] Food and Drug Administration. *Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices*, 2005.

[24] M. Gibiec, A. Czauderna, and J. Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 245–254, New York, NY, USA, 2010. ACM.

[25] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. I. Maletic, and P. Mäder. Traceability fundamentals. In *Software and Systems Traceability.*, pages 3–22. Springer, 2012.

[26] O. C. Z. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering, ICRE '94, Colorado Springs, Colorado, USA, April 18-21, 1994*, pages 94–101, 1994.

[27] T. Gruber. Ontology. In *Encyclopedia of Database Systems*, pages 1963–1965. Springer US, 2009.

[28] J. Guo, J. Cleland-Huang, and B. Berenbach. Foundations for an expert system in domain-specific traceability. In *21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013*, pages 42–51, 2013.

[29] J. Guo, N. Monaikul, C. Plepel, and J. Cleland-Huang. Towards an intelligent domain-specific traceability solution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 755–766. ACM, 2014.

[30] J. Guo, M. Rahimi, J. Cleland-Huang, A. Rasin, J. H. Hayes, and M. Vierhauser. Cold-start software analytics. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 142–153, 2016.

[31] S. Hayashi, T. Yoshikawa, and M. Saeki. Sentence-to-code traceability recovery with domain ontologies. In J. Han and T. D. Thu, editors, *APSEC*, pages 385–394. IEEE Computer Society, 2010.

[32] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1994.

[33] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[34] J. Huffman Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[35] M. Iyyer, J. L. Boyd-Graber, L. M. B. Claudino, R. Socher, and H. Daumé III. A neural network for factoid question answering over paragraphs. In *EMNLP*, pages 633–644, 2014.

[36] P. Jackson. *Introduction To Expert Systems (3 ed.)*. Addison Wesley, 1998.

[37] A. Karpathy, J. Johnson, and F. Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[38] J. C. Knight. Safety critical systems: challenges and directions. In *24th International Conf. on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 547–550, 2002.

[39] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[40] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[41] O. Levy, Y. Goldberg, and I. Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[42] Y. Li and J. Cleland-Huang. Ontology-based trace retrieval. In *Traceability in Emerging Forms of Software Engineering (TEFSE2013*, San Francisco, USA, May 2009.

[43] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 378–388, 2013.

[44] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):13, 2007.

[45] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang. Strategic traceability for safety-critical projects. *IEEE Software*, 30(3):58–66, 2013.

[46] A. Mahmoud, N. Niu, and S. Xu. A semantic relatedness approach for traceability link recovery. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 183–192, 2012.

[47] A. Mahmoud and G. Williams. Detecting, classifying, and tracing non-functional software requirements. *Requir. Eng.*, 21(3):357–381, 2016.

[48] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[49] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[50] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[51] N. Niu and A. Mahmoud. Enhancing candidate link generation for requirements tracing: The cluster hypothesis revisited. In *2012 20th IEEE International Requirements Engineering Conference (RE), Chicago, IL, USA, September 24-28, 2012*, pages 81–90, 2012.

[52] R. Pascanu, Ç. Gülçehre, K. Cho, and Y. Bengio. How to construct deep recurrent neural networks. *CoRR*, abs/1312.6026, 2013.

[53] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[54] N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS Comput Biol*, 5(11):e1000579, 2009.

[55] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

[56] P. Rempel, P. Mäder, T. Kuschke, and J. Cleland-Huang. Traceability gap analysis for assessing the conformance of software traceability to relevant guidelines. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, pages 120–121, 2015.

[57] T. Rocktäschel, E. Grefenstette, K. M. Hermann, T. Kociský, and P. Blunsom. Reasoning about entailment with neural attention. *CoRR*, abs/1509.06664, 2015.

[58] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[59] J. Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

[60] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[61] Y. Shin and J. Cleland-Huang. A comparative evaluation of two user feedback techniques for requirements trace retrieval. In *SAC*, pages 1069–1074, 2012.

[62] P. Shvaiko and J. Euzenat. Ontology matching: State of the art and future challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1):158–176, 2013.

[63] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

[64] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, 2004.

[65] H. Sultanov, J. Huffman Hayes, and W.-K. Kong. Application of swarm techniques to requirements tracing. *Requirements Engineering*, 16(3):209–226, 2011.

[66] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[67] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[68] US Department of Railroads. *Federal Railroad Administration,PTC System Information, https://www.fra.dot.gov/Page/P0358, Year = Accessed: 2016-08-26.*

[69] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[70] X. Zou, R. Settimi, and J. Cleland-Huang. Improving automated requirements trace retrieval: a study of term-based enhancement methods. *Empirical Software Engineering*, 15(2):119–146, 2010.