Norwegian University
of Life Sciences

Master's Thesis 2022 30 ECTS
Falcuty of Science and Technology

Constructing a Vulnerability Knowledge Graph

Anders Mølmen Høst
MSc Data Science

Acknowledgements

I would like to thank my two supervisors at Simula Research Laboratory, Postdoctoral fellow Guru Prasad Bhandari and Chief Research Scientist Leon Moonen for invaluable help during the work with this thesis. A special thanks to Bhandari who I have had the gratitude of collaborating close with during the work with this thesis. We have enjoyed exciting discussions together about the implications of the thesis and future directions. When I have had a difficult time finding the way forward, Bhandari have patiently pointed me in the right direction. I thank Leon for giving me the opportunity to work with this project and including me in his research group. Leon have provided valuable guidance of technical as well as theoretical character on the challenging problems related to this thesis.

I thank Professor Hans Ekkehard Plesser my supervisor at the Norwegian University of Life Sciences (NMBU) for his support with writing the thesis. I am also grateful for the courses in programming, informatics and data science project which Plesser has been teaching at NMBU. These courses has provided me with the tools to engage in the work with this thesis.

I thank my girlfriend Linn Marcela Fossnes, reminding me of slowing down and enjoying breaks also during stressful times. At last I will thank my family and particularly my parents for their continuous support during the work with this project.

Oslo, February 15, 2022

Anders Mølmen Høst

Abstract

Attackers exploiting vulnerabilities in software can cause severe damage to affected victims. Despite continuous efforts of security experts, the number of reported vulnerabilities is increasing. As of January 2022, the National Vulnerability Database consists of more than 160 000 vulnerability records of known vulnerabilities. These vulnerability records contain data such as vulnerability classification, severity metrics, affected software products, and textual descriptions describing the vulnerability.

The National Vulnerability Database provides a high-quality data source for security analysts learning about known vulnerabilities. However, maintaining this database comes at a high labor cost for the security experts involved. Knowledge graphs is a semantic technology which has the potential to aid in this task. In our work we explore how knowledge graphs are used in the broader field of cyber security. We then propose our own vulnerability knowledge graph for vulnerability assessment where we combine techniques from NLP with Knowledge graph embedding. Although future work on constructing ground truth data is necessary to evaluate and benchmark our experiments, our initial results show entity prediction results of 0.76 in Hits@10 score.

Contents

Acronyms

AP	Averaged perceptron. 21
BPTT	Backpropagation thorugh time. 36
CNA	CVE Numbering Authorities. 60
CPE	Common Platform Enumeration. 14
CRF	Conditional random fields. 26
CVE	Common Vulnerabilities and Exposures. 10
CVSS	Common Vulnerability Scoring System. 11
CWE	Common Weakness Enumeration. 13
IOB	Inside-Outside-Beginning. 20
KG	Knowledge graph. 10
MalKG	Malware Knowledge Graph. 61
MEM	Maximum Entropy Model. 26
MR	Mean rank. 19
MRR	Mean reciprocal rank. 19
NER	Named Entity Recognition. 11, 26
NLP	Natural language processing. 58

NVD National Vulnerability Database. 10

POS Parts of speech. 20

PRE Precision. 18

RE Relation Extraction. 19

REC Recall. 18

RNN Recurrent neural network. 35

VulKG Vulnerability Knowledge Graph. 58

Chapter 1

Introduction

1.1 Software security and vulnerabilities

New applications being released provide users with tools for work, study, entertainment, and other areas. However, while many of these applications are useful, some can also be abused in harmful ways. Software vulnerabilities are flaws in code that an attacker can exploit. Attacks utilizing these vulnerabilities can cause severe damage to the affected victim. The consequences for the victim include financial loss, exposure of sensitive data, or essential production halting.

Developers unintentionally introduce vulnerabilities in their code by lacking understanding of how systems fail. To combat this, training in secure development, use of code reviews, and testing are all important measures (Tsipenyuk et al., 2005).

A critical vulnerability in the application Log4j was published in December 2021 and has received much public attention. Log4j is an open-source logging tool from Apache [1]. The use of Log4j includes logging runtime events and error messages in applications running on servers. The vulnerability was associated with a feature where users could input code instructions to customize text formatting. Instead of using the feature to format logs, an attacker could input malicious code to gain access to the targeted server (Torres-Arias, 2021). The infected server can then be used for mining cryptocurrency or in attacks against organizations or government institutions (Palmer, 2021). Log4j is commonly bundled in other software, making the

[1]https://logging.apache.org/log4j/2.x/manual/index.html

vulnerability affecting a wide range of applications. Not only does it affect the applications including vulnerable Log4j libraries, but also services using these applications (msl, 2021). Consequently, the task of identifying the vulnerable components in order to implement patches becomes more difficult.

On an annual basis, the number of reported vulnerabilities in Common Vulnerabilities and Exposures (CVE) (MITRE, 2021a) is increasing, according to statistics from the National Vulnerability Database (NVD). In 2015, less than seven thousand CVEs were registered, and in 2020 the number had increased to more than 18 thousand (Laboratory, 2021).

Security researchers are constantly searching for better ways to learn why vulnerabilities occur and how to fix these. Accurate classification of known vulnerabilities is essential to achieve this goal. Security related data are heterogeneous and often unstructured, including textual descriptions of vulnerabilities and vulnerable source code. Manually classifying vulnerabilities from these sources is labor-intensive work for security experts. Data-driven security research can contribute to vulnerability classification using machine learning techniques and statistical analysis.

1.2 Motivation and problem statement

CVE is a data set storing vulnerability information as CVE records. The CVE records contain an ID number, a description, and public references. NVD fetches this data and provides additional structure by adding vulnerability type, product configurations, and severity metrics to each record. However, this comes at a high labor cost for security analysts labeling and classifying the records. A knowledge graph (KG) is a graph-based abstraction to represent data as knowledge (Hogan et al., 2021, Ch. 1).

where a schema is defined in advance for relational databases, defining a schema could be postponed in KGs. This allows for more flexibility for the KG to evolve, for example by adding incomplete information (Hogan et al., 2021, Ch. 1).

We believe that knowledge graphs can aid in labeling data in NVD by increasing the degree of automation. In addition, the knowledge graph can also be helpful for data exploration where similar vulnerabilities can be found based on common characteristics.

This thesis constructs a vulnerability knowledge graph by extracting data from NVD. The first part of the knowledge graph is to extract cyber-security entities using a Named Entity Recognition (NER) model, which we train on labeled NVD data. We then propose a rule-based relation extraction (RE) model to extract relations between the entities. The RE model is based on NVD labels and identifies the typical structure of CVE records. The extracted entities and relations between these are triples that combined form our initial knowledge graph. Knowledge graph embedding is the last step of our model. This technique enables us to improve the initial graph by predicting missing entities. We performed KG embedding by training a tensor decomposition model.

Applying the knowledge graph concept to the NVD data, we raise the following research questions:

1. *Can our knowledge graph predict vulnerability types and related terms?*
2. *Can we find relations between existing vulnerabilities using the same knowledge graph?*

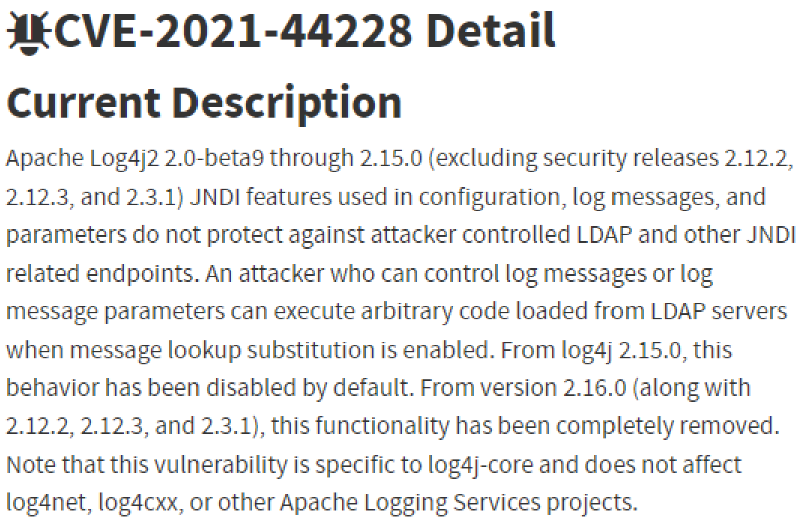
1.3 National Vulnerability Database

CVE stores disclosed vulnerabilities. Each vulnerability record has a unique identification number, a description, and public links. The vulnerability description contains information about the characteristics of the vulnerability and how it can be exploited, and what products and vendors are affected. The public links hold additional publicly available information about the vulnerability and include advisories, solutions, mitigation, and exploit information.

NVD fetches vulnerability records from CVE and adds additional information such as product configurations and vulnerability types. Five figures 1.1 to 1.5 shows how the Log4j vulnerability (CVE-2021-44228) is stored as a vulnerability record in NVD. The first figure, Fig. 1.1 contains the vulnerability description which is the same as in CVE.

Severity metrics is calculated in Fig. 1.2 according to the Common Vulnerability Scoring System (CVSS) [2]. CVE-2021-44228 is a critical vulnerability with a base score of 10.0, the highest possible score.

[2]https://www.first.org/cvss/


CVE-2021-44228 Detail

Current Description

Apache Log4j2 2.0-beta9 through 2.15.0 (excluding security releases 2.12.2, 2.12.3, and 2.3.1) JNDI features used in configuration, log messages, and parameters do not protect against attacker controlled LDAP and other JNDI related endpoints. An attacker who can control log messages or log message parameters can execute arbitrary code loaded from LDAP servers when message lookup substitution is enabled. From log4j 2.15.0, this behavior has been disabled by default. From version 2.16.0 (along with 2.12.2, 2.12.3, and 2.3.1), this functionality has been completely removed. Note that this vulnerability is specific to log4j-core and does not affect log4net, log4cxx, or other Apache Logging Services projects.

Figure 1.1: CVE ID and description of CVE-2021-44228

The Vector string in the bottom of Fig. 1.2 consists of categorical variables which are used as input when calculating the score. CVE-2021-44228 consists of the following nine elements:

CVSS:3.1 CVSS version.

AV:N Attack vector is Network.

AC:L Access complexity is low.

PR:N Privileges required is none.

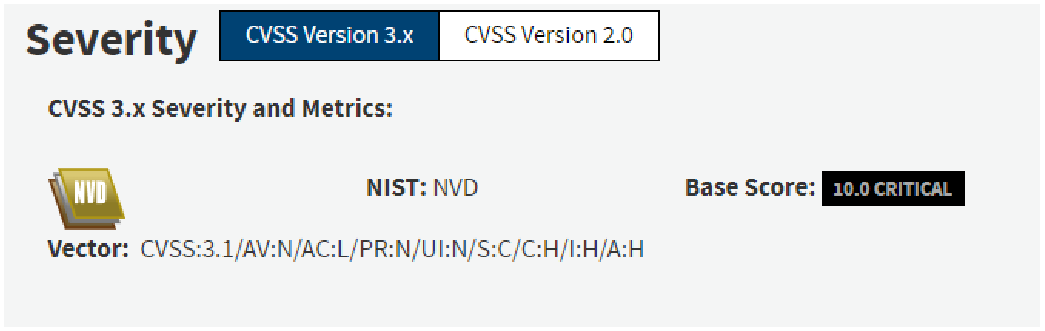
UI:N User interaction is not required.

S:C Scope is changed.

C:H Confidentiality impact is high

I:H Integrity impact is high

A:H Availability impact is high



CVSS 3.x Severity and Metrics:

NIST: NVD

Base Score: 10.0 CRITICAL

Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Figure 1.2: Common Vulnerability Scoring System (CVSS) of CVE-2021-44228

CVSS version 3.1 is the standard for calculating the base score. The Attack vector is through the network, allowing remote attackers to perform an attack through the internet. Attack complexity for this example is low, where the attack complexity measures what conditions outside the attackers' control must be circumvented for the attack to be successful. No privileges are required, so the attacker does not need to be authorized.

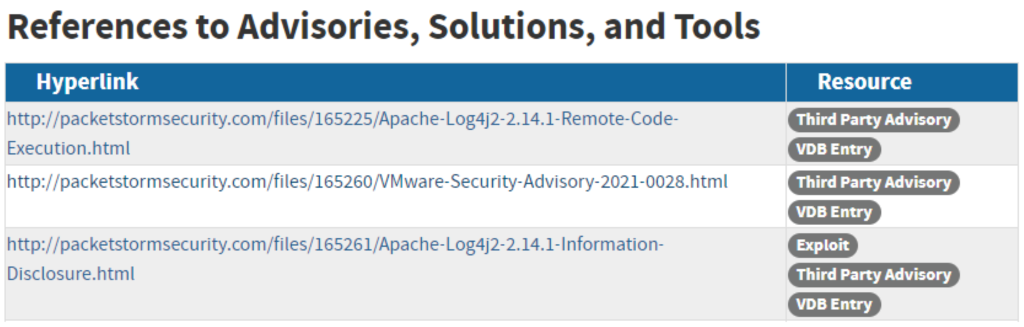
The security scope in CVE-2021-44228 is changed. A security scope consists of security authorities, where a security authority is a mechanism for access control. For example, a single security authority could restrict what processes and which users have access to one database. If the vulnerability affects components outside of the vulnerable components security authority, the security scope is changed (CVS, 2022). In CVE-2021-44228, applications directly using Log4j libraries in their code are vulnerable. In addition, other applications and services using these applications are also vulnerable.

The impact on confidentiality (C), integrity (I), and availability (A) are all high (H). Measures of confidentiality restrict what information is available to whom. For example, only authorized users should have access to read sensitive information. Integrity concerns write access. An example of an integrity measure is ensuring that only authorized users can write information to a particular database. Availability is about making sure that requested resources are available to authorized users (Bruvoll and Arne Sørby, 2020).

Fig. 1.3 shows some of the provided public links, including exploit information and third-party advisories.

Common Weakness Enumeration (CWE) [3] contains a list of software and

[3] https://cwe.mitre.org/



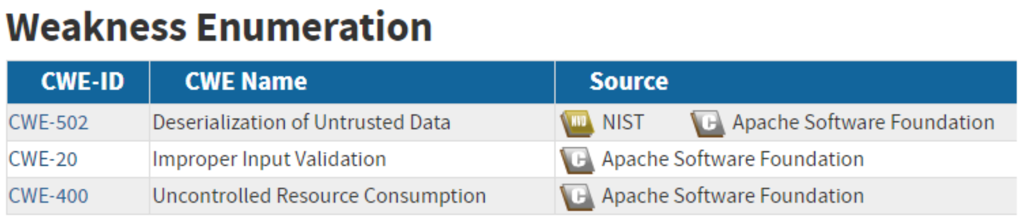
References to Advisories, Solutions, and Tools

Hyperlink	Resource
http://packetstormsecurity.com/files/165225/Apache-Log4j2-2.14.1-Remote-Code-Execution.html	Third Party Advisory VDB Entry
http://packetstormsecurity.com/files/165260/VMware-Security-Advisory-2021-0028.html	Third Party Advisory VDB Entry
http://packetstormsecurity.com/files/165261/Apache-Log4j2-2.14.1-Information-Disclosure.html	Exploit Third Party Advisory VDB Entry

Figure 1.3: Some of the public references in CVE-2021-44228

hardware weakness types. NVD classifies vulnerabilities (CVE) in weakness types (CWE).

CVE-2021-44228 has three CWEs assigned. CVE-2021-44228 allows an attacker to input malicious code without raising an exception. Thus the input is not properly validated, and the weakness ”CWE-20 Improper Input Validation” in Fig. 1.4 is applicable. We refer to MITRE (2021b) for further information about the two other weaknesses CWE-502 and CWE-400, which also apply to CVE-2021-44228.



Weakness Enumeration

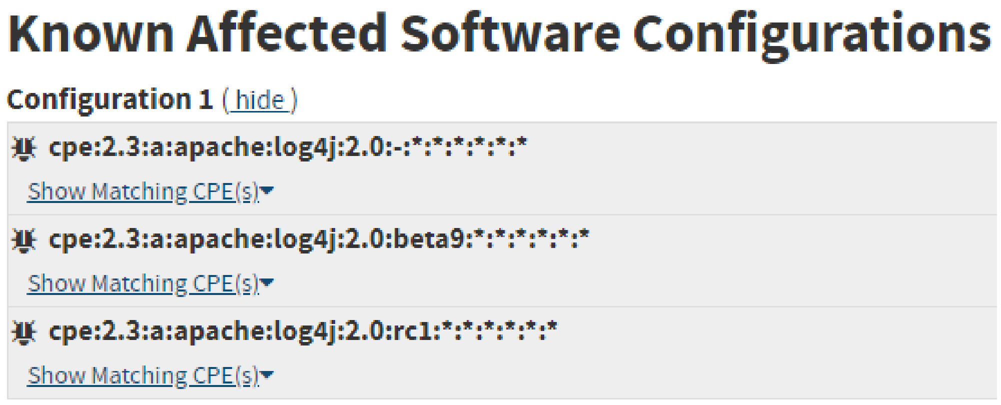
CWE-ID	CWE Name	Source
CWE-502	Deserialization of Untrusted Data	NIST Apache Software Foundation
CWE-20	Improper Input Validation	Apache Software Foundation
CWE-400	Uncontrolled Resource Consumption	Apache Software Foundation

Figure 1.4: CVE-2021-44228 is classified in three weakness types.

Common Platform Enumeration (CPE) is a protocol for identifying and classifying product configurations ([4]). Each CPE string corresponds to a single uniquely defined configuration.

Considering the second line in Fig. 1.5. ”cpe:2.3” tells us that this product configuration has CPE version 2.3. ”a” tells us that the product is

[4]https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe


Known Affected Software Configurations

Configuration 1 (hide)

cpe:2.3:a:apache:log4j:2.0:-:*:*:*:*:*:*

Show Matching CPE(s)

cpe:2.3:a:apache:log4j:2.0:beta9:*:*:*:*:*:*

Show Matching CPE(s)

cpe:2.3:a:apache:log4j:2.0:rc1:*:*:*:*:*:*

Show Matching CPE(s)

Figure 1.5: Some of the affected configurations of CVE-2021-44228

an application. The software vendor is Apache, and the application is Log4j. The application has version 2.0 and update beta9.

1.4 Framework

The following framework gives an overview of our work constructing a vulnerability knowledge graph. The framework begins with downloading data from NVD and goes through multiple steps before reaching the final step of entity prediction.

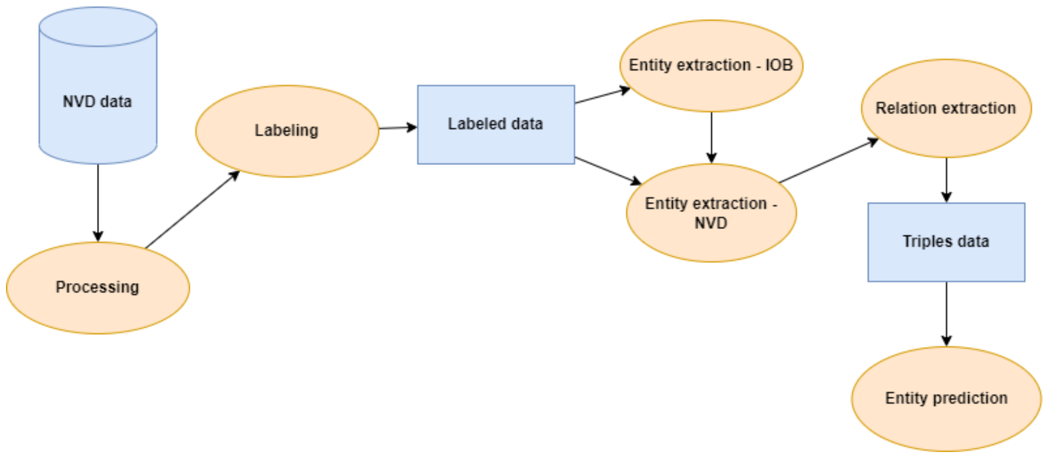


Figure 1.6: Framework showing the steps of our work

Chapter 2

Theory

2.1 Performance metrics

In the following section we present performance metrics used throughout this thesis. In the first part we consider the commonly encountered metrics in machine learning which is accuracy, precision, recall and F1-score. In the second part we look closer into performance metrics commonly used for *entity prediction* in knowledge graph.

2.1.1 Common performance metrics in machine learning

Accuracy is common metric used when evaluating machine learning classifiers on balanced data. Accuracy simply calculates all true predictions and divides this by the total number of predictions. Accuracy is defined:

$$Accuracy = \frac{TP + TN}{FP + FN + TP + TN} \tag{2.1}$$

where

- TP: True positives
- TN: True negatives
- FP: False positives

- FN: False negatives

However, accuracy is not a good metric for unbalanced data sets. For example, consider we are interested in classifying vulnerability related terms from a vulnerability description and separate these from the non-vulnerability related terms. Our data set consists of 1 percent vulnerability related terms and 99 percent non-vulnerability related terms. A model that always predicts non-vulnerability related terms, will achieve 99 percent accuracy. However, such a model is not a good model since it completely fails in predicting vulnerability related terms.

Precision (PRE), recall (REC and F1-score are metrics which is often used for unbalanced data sets. Precision and recall is defined:

$$PRE = \frac{TP}{TP + FP} \tag{2.2}$$

$$REC = \frac{TP}{FN + TP}. \tag{2.3}$$

If we optimize our model for precision, our model will only predict vulnerability related terms if it has a high confidence that the term is a vulnerability related term.

If we optimize our model for recall on the other hand, our model will aim for predicting all vulnerability related terms in our data set. However, this comes at the cost of predicting a larger amount of false positives. Where the false positives are the terms incorrectly predicted as vulnerability related term.

A metric which compromises the wish for high precision and high recall is called F1-score. F1-score is defined:

$$F1 = 2 \times \frac{PRE \times REC}{PRE + REC} \tag{2.4}$$

2.1.2 Performance metrics in knowledge graphs

A triple in a knowledge graph consists of a head, a tail, and a relation between the two. Entity prediction is the task of predicting a tail (or head) given that we know head (or tail) and relation. The goal is to find the true head or tail given all other entities in the graph. Entity prediction algorithms usually return results in the form of a ranked list of possible triples (Balažević et al., 2019). For example, consider we want to predict the head of a triple given 100 possible entities. We assign confidence to all possible triples, and then return the rank of the true triple. The ranking of the true triple are used when calculating performance metrics such as mean rank (MR), mean reciprocal rank (MRR) and Hits@n (Balažević et al., 2019).

MR calculates the average ranking of the true triple over all possible triples. To compute the MRR we first calculate the inverse ranking of the true triple among all possible triples. Then the average of these rankings is computed to get the MRR score. Hits@n calculates the percentage of time the true triple is ranked among the top n possible triples. Hits@10, hits@3 and hits@1 is commonly reported when referring to this metric.

A low MR, a high MRR and a high Hits@n indicates better performance of the entity prediction model. The MR metric could be unstable and is reported to be more negatively affected by individual bad ranked examples. It has thus been more common in recent work to focus on MRR and Hits@n metrics (Rastogi et al., 2021).

2.2 Labeling unstructured data

In order to train supervised machine learning models to perform NLP tasks such as NER or relation extraction (RE), we need labeled data serving as ground truth. Both manually labeling and automatic labeling approaches have been considered for this thesis.

The first approach is to annotate data manually. Although the approach can produce very accurate labeling, it is costly and time-consuming. A corpus consisting of 30 security blogs, 240 CVE descriptions, and 80 security bulletins manually annotated is considered by Joshi et al. (2013) to train a NER model for cybersecurity entities. Joshi et al. (2013) uses the Stanford NER (Finkel et al., 2005) which is a pre-trained NER model. The performance metrics is reported in Table 2.1.

As an alternative approach Bridges et al. (2014) propose a framework for automatic labeling. The data set consists of vulnerability descriptions from NVD, patch and mitigation information taken from Microsoft Security Bulletin, and exploit descriptions from Metasploit Framework (Bridges et al., An HTML Injection Vulnerability in iOrder 1.0 allows the remote attacker to execute Malicious HTML codes via the signup form

Figure 2.1: Vulnerability description of CVE-2021-43441

2014). The vulnerability descriptions from NVD used by Bridges et al. (2014) is approximately thirty times the size of what was used by Joshi et al. (2013). This framework labels data using three approaches.

The first step is using the CPE vector for exact database matching. This is done by matching elements from the CPE vector with words in the vulnerability description. We consider the following vulnerability description;

With the corresponding CPE vector:

cpe:2.3:a:iorder_project:iorder:1.0:*:*:*:*:*:*:*

”iorder” is the product component of the CPE vector above and is of type application. The ”iorder” from CPE is matched with ”iOrder” from the CVE description. The word ”iOrder” will then be labeled as application. Furthermore, 1.0 is a version component of the CPE vector above and is matched with ”1.0” in the CVE description for labeling.

Bridges et al. (2014) uses heuristic rules and regular expressions to label those entities that do not precisely match the CPE vectors. One example of a heuristic rule approach is to use file extensions such as ”.exe” to label the file entities. Another approach is to use standard terminology when referring to versioning, such as ”before 2.5”, ”1.1.4 through 2.3.0” and ”2.2.x”. In addition, Bridges et al. (2014) classifies some of the words as vulnerability-relevant terms. These relevant terms can consist of one-gram, two-gram, or three-gram words. Where an example of a two-gram relevant term is ”remote attacker”. A total of 172 entities with relevant terms are provided in a dictionary which is called a *gazetteer*. When encountering any of these 172 entities in a CVE description, the label is given by the dictionary. The relevant terms provided by the gazetteer are entities particularly relevant to the weakness type (CWE). The labeling we just described is referred to as domain-specific labeling by Bridges et al. (2014).

In addition to domain-specific labeling, Inside-Outside-Beginning-labeling (IOB) and parts of speech (POS) labeling are also performed by Bridges et al. (2014). IOB-labeling is a technique useful for labeling multi-word entities, also called n-grams. The first word of an entity is labeled as ”B” which

Paper	**Precision**	**Recall**	**F1-score**
Bridges et al. (2014)	0.989	0.993	0.994
Joshi et al. (2013)	0.837	0.764	0.799

Table 2.1: Entity extraction performance from two papers. Automatic annotation is utilized in Bridges et al. (2014) while manual annotations is performed in Joshi et al. (2013).

stands for "begin". The following words that belong to the same entity are labeled with "I" for "inside". The words which are not labeled are labeled with "O". For example, the two words in the term "remote attacker" should have IOB labels "B" and "I" correspondingly.

Bridges et al. (2014) evaluate the automatic labels by comparing them against a random sample of manually annotated CVE descriptions. The labeling results reported were 99 percent precision, 77.8 percent recall, and 87.5 percent F1 score (Bridges et al., 2014). Bridges et al. (2014) argues for optimizing their model for high precision where nearly every label found is correctly classified. On the contrary, recall is high if all labels are classified, but many of the entities are classified incorrectly. By choosing this approach Bridges et al. (2014) avoid introducing additional noise to the data. The purpose of high precision in labeling is that this will yield higher performance when later training NER models based on machine learning algorithms on the labeled data.

Bridges et al. (2014) train an average perceptron (AP) classifier on the automatically labeled data. The results are presented in Tab. 2.1 where the reported performance in Bridges et al. (2014) is much higher than what is achieved in Joshi et al. (2013). The average perceptron seems to benefit from the large amounts of data available as a result of the automatic labeling approach (Bridges et al., 2014).

In this thesis automatic labeling based on Bridges et al. (2014) will be used to label data from NVD. In the case of automatic labeling, there are two reasons for training a machine learning classifier for NER on top of already labeled data. These are to improve the performance of the automatic labeling, and to enable recognizing entities from unstructured text where the automatic labeling can not be applied directly. This includes cyber security newsletters, forum posts and raw CVE descriptions which has not yet been analysed and enriched in NVD.

2.3 Encoding of words

If we want to use machine learning algorithms for NER or RE on vulnerability descriptions, we must ensure that these descriptions are in a compatible representation for the algorithms to understand. The first step is tokenization, which is splitting the descriptions into smaller pieces called tokens. The tokens could, for example, be words, part of words, or characters (Pai, 2020). Usually, these tokens must be converted into a numeric representation before machine learning algorithms can be applied. We look into different techniques to represent words as numeric vectors in the following sections. In addition to being useful for NER and RE, a good understanding of how words are encoded is helpful before we continue exploring how we can encode graphs as vectors in Sec. 2.7.

2.3.1 Bag of words

One way of representing words as numeric vectors is the bag of words model (Raschka and Mirjalili, 2019, Ch. 8). Each word in the corpus is assigned an index number in this simple model. One vector can then be used to either represent a single word or a document. This vector is typically sparse, with a length equal to the number of words in the corpus. All indices except those corresponding to words in the document are zero. For example, if a single word is encoded in a corresponding vocabulary of 20 000 words, one element will be equal to one while the 19 999 other elements are zero.

If encoding is done on the document level, the words present in the document are represented by the number of occurrences on the corresponding index. This is the standard bag of words model. Alternatively, in the binary bag of words model, we ignore the occurrences and only consider if the word occurs or not. The indices are then either zero or one depending on the word exists or not (Bose, 2021). This type of encoding is also referred to as one-hot encoding (Chollet, 2018, Ch. 6.1.1).

We demonstrate the binary bag of words with two vulnerability descriptions. CVE-2021-4079:

> Out of bounds write in WebRTC in Google Chrome prior to 96.0.4664.93 allowed a remote attacker to potentially exploit heap corruption via crafted WebRTC packets.

And CVE-2021-4064:

> Use after free in screen capture in Google Chrome on ChromeOS prior to 96.0.4664.93 allowed a remote attacker to potentially exploit heap corruption via a crafted HTML page.

From these examples, we see that the two descriptions share common words such as "Google" and "attacker". We first make an index of all unique words across both CVE descriptions.

Listing 2.1: Binary bow word index

```
{'in': 1, 'to': 2, 'a': 3, 'webrtc': 4, 'google': 5,
  'chrome': 6, 'prior': 7, '96': 8, '0': 9, '4664'
 : 10, '93': 11, 'allowed': 12, 'remote': 13, '
 attacker': 14, 'potentially': 15, 'exploit': 16,
 'heap': 17, 'corruption': 18, 'via': 19, 'crafted
 ': 20, 'out': 21, 'of': 22, 'bounds': 23, 'write'
 : 24, 'packets': 25, 'use': 26, 'after': 27, '
 free': 28, 'screen': 29, 'capture': 30, 'on': 31,
  'chromeos': 32, 'html': 33, 'page': 34}
```

We then encode the descriptions using the binary bag of words and get the following output.

Listing 2.2: Binary bow CVE-2021-4079

```
[0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
  1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
```

Listing 2.3: Binary bow CVE-2021-4064

```
0., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0
 ., 1., 1., 1., 1., 1., 1., 1., 1., 1.]
```

Indexing starts from 0, and thus the first element from each vector is encoded with 0. Comparing the two descriptions we see that for example "webrtc" which has index 4 is represented by 1 in CVE-2021-4079 and 0 CVE-2021-4064.

The challenge with this approach is that expanding the corpus to contain more words will increase sparsity when single documents are encoded.

2.3.2 Term Frequency - Inverse Document Frequency

As was seen in the section above, some of the words were common for both CVEs. When including all CVEs from NVD some words will occur more frequently than others. If we are extracting information from CVEs, the less frequently occurring words are more important to distinquish CVEs from one-another. We say that less frequently occurring words contains more *discriminatory* information ((Raschka and Mirjalili, 2019)). As a consequence, these words should be given more emphasis. Term Frequency Inverse Document Frequency (tf-idf) is a technique to handle this.

We define the term frequency as the number of times a term occurrs in a document. The term frequency is often normalized ((Bose, 2021)) by the total number of words in a document and defined as :

$$tf(t,d) = \frac{n_{t,d}}{\sum_{i=1}^{k} n_{i,d}} \tag{2.5}$$

Here $n_{t,d}$ is the number of occurrences of term, t, in document, d.

We can then define the inverse document frequency as:

$$idf(t,d) = \log \frac{n_d}{1 + d_f(d,t)} \tag{2.6}$$

Where n_d is the total number of documents and d_f is the document frequency, which is the number of documents, d, containing the term, t. The inverse document frequency is logarithmically scaled to ensure that weights assigned to low document frequencies are not too high ((Raschka and Mirjalili, 2019)). It is common to add one in the denominator in (2.6) to avoid division by zero.

The product of 2.5 and 2.6 is the term-frequency-inverse document frequency, defined:

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t,d) \tag{2.7}$$

2.3.3 Word Embedding

Embedding of words is a popular approach to represent words as vectors. Word embeddings are dense numeric representations. Instead of encoding words as a vector with length equal to the size of the vocabulary, the dimensions can be chosen depending on the problem. Typically when dealing with large vocabularies its common with 256-dimensional, 512-dimensional or 1024-dimensional vectors (Chollet, 2018, 6.1.1). This means that for a vocabulary containing 20 000 words, a significant amount of space is saved compared to a one-hot encoding approach. Another characteristic of word embeddings is that they are learned from data. The goal is to construct embeddings such that the distance between two words in a vector space should reflect their semantic relationship. Synonyms should be encoded close, while two unrelated words should be encoded further away. One toy example of a word embedding taken from Chollet (2018) considers the four words: tiger, wolf, cat, and dog. The relationships in a 2-dimensional space is illustrated in Fig. 2.2. On the y-axis, wolf and tiger have higher values since they are both wild animals, while cats and dogs are pets and are thus assigned lower values. On the x-axis, the embedding captures that dog and wolf share that they are both canines while tiger and cat are felines.

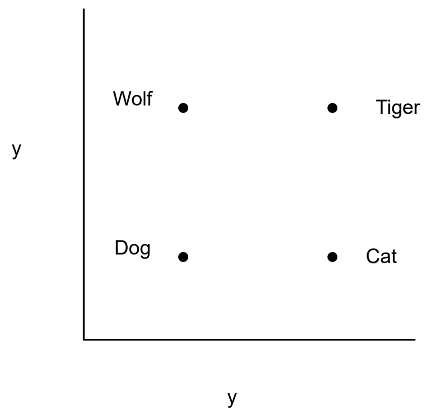


Figure 2.2: Toy example of word embedding

To learn the word embeddings, different approaches exist. A model which have proven success are referred to as the Word2vec algorithm and were proposed by Mikolov et al. (2013). The basic idea is that the model computes the embedding for a particular word by considering a window of surrounding words as context.

2.4 Named Entity Recognition

Named Entity Recognition (NER) is the task of classifying a *mention* also referred to as a term from a document with a label. These terms typically consist of a single or a few words (Kejriwal, 2019). Some terms are ambiguous. For example, this could be ”Linux” which could refer to the ”Linux” organization or the ”Linux” software. These terms need disambiguation, which we will touch upon later in the thesis. In the following, a model for NER, which has been used for extracting cyber security entities by Bridges et al. (2014) is presented.

2.4.1 Maximum Entropy model trained with Averaged Perceptron

The Maximum Entropy Model (MEM) is a supervised machine learning model utilizing a large set of predefined features. The model is a popular choice in sequential tagging problems (Bridges et al., 2014). In our case, the model is used for Named Entity Recognition. Different variants of the model are common. In history-based MEM, the model considers only features until the current word. In Conditional Random Fields (CRF) the ”global” features are considered. Global features refer to features both before and after the current word of each sentence. We can use both alternatives for training a NER model in extracting security-related entities. We present an example of MEM using two history-based features in the following.

A sentence, w, is split in a sequence of words with length n:

$$w = (w_1, ..., w_n) \tag{2.8}$$

The sentence is labeled with the corresponding sequence of tags:

$$t = (t_1, .., t_n) \tag{2.9}$$

Sequences of labels and words form the input to the model.

By Bridges et al. (2014) the following features have been implemented:

For notional ease, we focus on the previous two domain labels as a single feature, defined as t_{i-1}, t_{i-2}. However, the following formulas can be generalized, including all the mentioned features.

Given an arbitrary word sequence, the conditional probability of a tag sequence is:

$$p(t|w) = \prod_{i=1}^{n} p(t_i|t_{i-2}, t_{i-1}, w_i) \tag{2.10}$$

The probability of a tag is then defined:

$$p(t_i|t_{i-2}, t_{i-1}, w_i) \equiv \frac{e^{f(\bar{t_i}, w_i)*v}}{z(\bar{t_i}, w_i)} \tag{2.11}$$

$f = (f_1, ..., f_m)$ is the feature vector with corresponding weights, $v = (v_1, ..., v_m)$ which are learned from the data. For notional ease, we defined $\bar{t_i} = t_{i-2}, t_{i-1}, t_i$. We want to make sure that the computed probabilities of all possible tags in has sum equal to one. To ensure this we divide by the factor, $z(\bar{t_i}, w_i)$ which is defined:

$$z(\bar{t_i}, w_i) = \sum_{t_i} exp[f(\bar{t_i}, w_i) * v] \tag{2.12}$$

Each feature either fires or not. An example of a feature component could be:

$$f_1(\bar{t_i}, w_i) = \begin{cases} 1 & \text{if } t_{i-2} = \text{Vendor}, \\ & t_{i-1} = \text{Application} \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

The feature above fires if we encounter the tags vendor and application as the two previous tags.

The goal is to find the best set of weights such which maximizes the probability of predicting the correct tag.

The weights are initialized to zero and trained using the averaged perceptron. Since the perceptron learning rule updates the weights for every prediction, Bridges et al. (2014) argues for averaging the weights over multiple iterations. Considering an intuitive example from Bridges et al. (2014) of a model predicting correctly in 9999 examples out of 10 000, but then makes the wrong prediction in the last example. In such a case, a model which has 99.99 percent accuracy will be replaced by a less accurate model Bridges et al. (2014).

Al Algorithm 1 describes the averaged perceptron algorithm which Bridges et al. (2014) uses for training the MEM model and is. Intermediate weights at each iteration are stored in v and accumulated as a weighted sum in v_{tot}. $v_{t-stamp}$ keeps a timestamp for when the different weights were adjusted. The argument maximum of $p(\hat{t}|w, y)$ with respect to the possible tags, $\hat{t}$, is used to decide the prediction y. The weights are only adjusted when a prediction is wrong, and only the weights for the fired features are adjusted.

Listing 1 Averaged Perceptron

```
**Input**: $(w, t)$ = training set
$N_{iter}$ = number of iterations
**Output**: $v_{ave}$ = trained parameter vector
Initialize $iter = 1$
Initialize $i = 0$
Initialize $v = (0, ..., 0)$
Initialize $v_{t-stamp} = (0, ..., 0)$
Initialize $v_{tot} = (0, ..., 0)$
**while** $iter \leq N_{iter}$ **do**
    **for** $(w, t)$ in training set **do**
        Set $y = argmax_{\hat{t}} p(\hat{t}|w, v)$
        **if** $y! = t$ **then**
            $v_{tot} + = [(i, ...i) - v_{t-stamp}] * v$
            $v + = f(w, t) - f(w, y)$
            **for** $j = 1...length(v)$ such that
            $f(w, y)[j]! = 0$ **do**
                Set $v_{t-stamp}[j] = i$
            $i + = 1$
        **else**
            $i + = 1$
    $iter + = 1$
$v_{tot} + = [(i, ..., i) - v_{t-stamp}] * v$
Set $v_{ave} = v_{tot}/i$
**return** $v_{ave}$
```

2.5 Machine learning

In the following we provide an introduction to some basic machine learning algorithms and some more advanced algorithms which extends the basic models. The goal for this section is to provide an understanding of some of the models which could be used as part of our KG. We also provide an introduction to underlying concepts including model weights, net input, loss functions and activation functions which is applicable to many machine learning models.

2.5.1 Perceptron

In the Perceptron model, small random weights, $\boldsymbol{w}$, are initialized and assigned to the input features, x. The net input is then defined as:

$$z = \boldsymbol{w}^T \boldsymbol{x} \tag{2.14}$$

In the unit step function, this net input is compared against a defined threshold, θ.

For simplification it is common to bring θ to the left side of the net input by defining a zero weight, $w_0 = -\theta$ and a corresponding input signal $x_0 = 1$ (Raschka and Mirjalili, 2019, p.21). The unit step function can then be defined as:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.15}$$

And the net input in 2.14 becomes:

$$z = w_0 x_0 + w_1 x_1 + .. + w_m x_m = \boldsymbol{w}^T \boldsymbol{x} \tag{2.16}$$

(Raschka and Mirjalili, 2019, Ch. 2) The output of the unit step function corresponds to a single prediction, $\phi(z) = \hat{y}$. The true value ,y, is compared against the prediction, $\hat{y}$. If the prediction is wrong the weights are updated. This update value is often referred to ase *loss* and defined:

$$\Delta w_j = \eta(y^{(i)} - \hat{y})x_j^{(i)} \tag{2.17}$$

Here η is the learning rate that controls the magnitude of the weight updates for misclassified examples. The learning rate is discussed in more detail in the next section about Adaline 2.5.2. For every prediction, all weights are updated simultaneously. The steps of training the Perceptron model can be summarized in the Perceptron learning rule:

1. Initialize the weights to 0 or small random numbers.
2. For each training example, $\boldsymbol{x}^{(i)}$:
   (a) Compute the output value, $\hat{y}$.
   (b) Update the weights.

(Raschka and Mirjalili, 2019, p. 23)

The rule above corresponds to one iteration of training, also called *epoch*, and training the model usually involves multiple epochs. When 2.17 is zero across all weights for all examples, the algorithm has converged. The model has perfect performance without any misclassifications in such an idealized case. However, for this to be achievable, the data must be *linearly separable*. Examples of linearly and non-linearly separable data are illustrated in Fig. 2.3. Since this is not always the case, it is common to set a maximum limit on how many epochs of training. Another approach is to set a threshold for the number of accepted misclassifications (Raschka and Mirjalili, 2019, p.25).

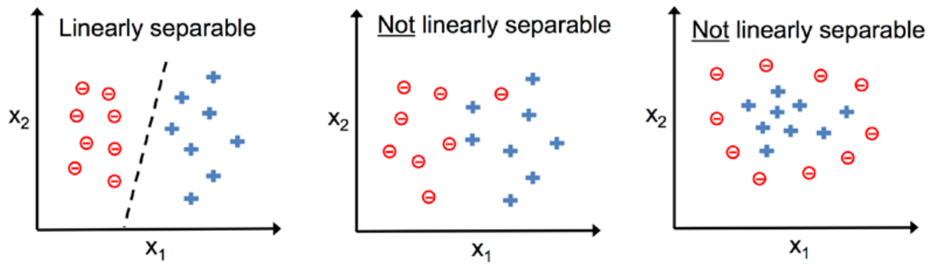


Figure 2.3: Linearly and non-linearly separable data by Raschka and Mirjalili (2019, p.25)/MIT license.

The architecture of the perceptron algorithm is illustrated in Fig. 2.4.

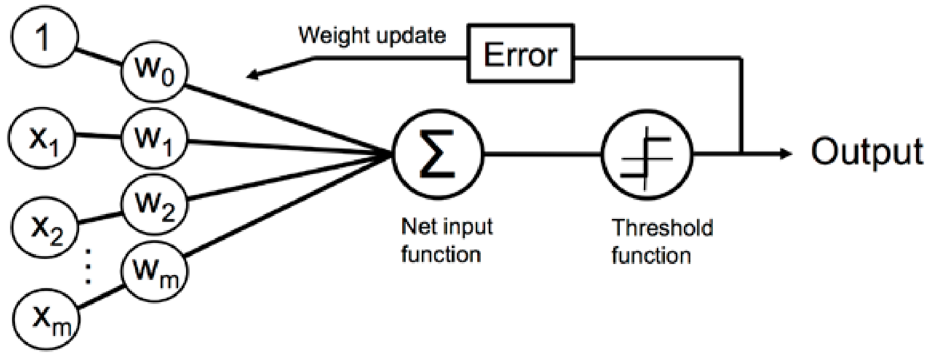


Figure 2.4: Architecture of Perceptron. Figure from Raschka and Mirjalili (2019, p.26)/MIT license.

2.5.2 Adaline

Adaline is a single-layer neural network illustrated in Fig. 2.5 and considered an improvement of the Perceptron model. Adaline is a good starting point to understand more complex neural network architectures.

Adaline learns by optimizing an *objective function*, also called a *loss function*. The loss function of Adaline is the sum of squared errors (SSE) between the true labels and predicted labels defined as:

$$J(\mathbf{w}) = \frac{1}{2}\sum(y^{(i)} - \phi(z^{(i)}))^2 \tag{2.18}$$

where $y^{(i)}$ is the true class label of the i^{th} sample. $\phi(z^{(i)})$ is the predicted label of the i^{th} sample. In Fig. 2.5 each input feature is assigned a weight.

A difference from Perceptron is that a linear activation is applied to the net input function.

$$\phi(\boldsymbol{z}) = \boldsymbol{z} \tag{2.19}$$

However, for Adaline, the above is simply an identity function and the net input does not change. Perceptron computes the loss after applying the unit step function. In Adaline, loss is computed by comparing the true label with the output after linear activation to decide the weight update.

When a final prediction is made, the output from the activation function is compared against the threshold function.

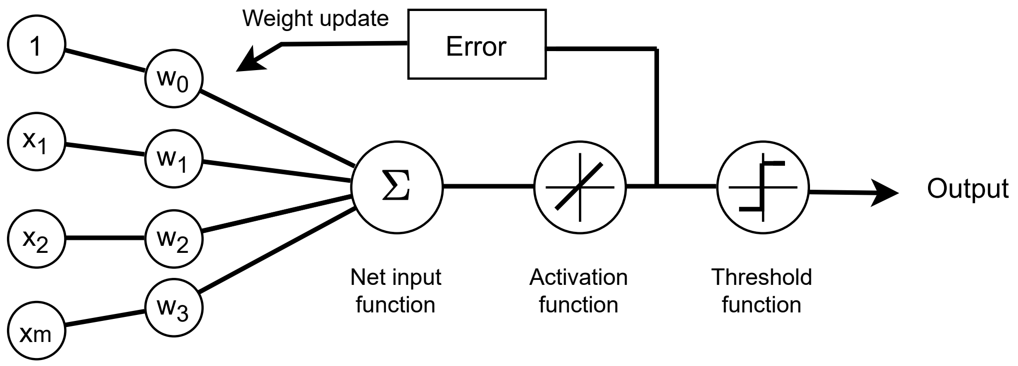


Figure 2.5: Architecture of Adaptive Linear Neuron. Figure based on Raschka and Mirjalili (2019, p.37).

During training, Adaline updates the weights for every epoch using the following update rule.

$$\mathbf{w} := \mathbf{w} + \mathbf{\Delta w}, \quad \text{where } \mathbf{\Delta w} = -\eta \nabla J(\mathbf{w}) \tag{2.20}$$

η is the learning rate which is a hyperparameter. $\nabla J(\mathbf{w})$ is the gradient. Since the cost function is minimized, a minus sign in front of the learning rate is used. This optimization method is called gradient descent, and a step is taken in the opposite direction of the gradient to reach the minimum seen in the left part of Fig. 2.6. Using gradient descent, all weights are updated simultaneously, where the updates are calculated by multiplying each input signal with its corresponding error. To compute the gradient, each individual weight update is found by taking the partial derivative of the cost function with respect to each weight j.

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \tag{2.21}$$

The learning rate is used to adjust the speed of the training. The learning rate must be tuned in order to reach the optimum. A too-large learning rate will overshoot the minimum seen in the right part of Fig. 2.6, a too small learning rate results in slow training with the potential of never reaching the minimum. Interested readers are referred to Raschka and Mirjalili (2019, ch. 2) for additional details regarding the challenges of finding the best learning rate.

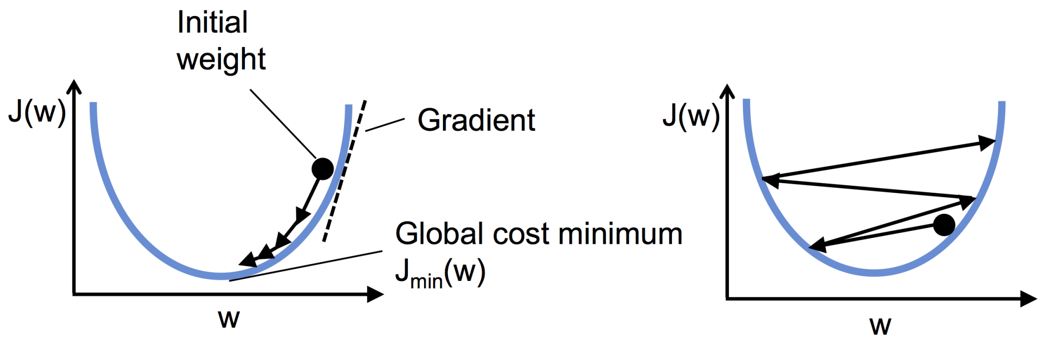


Figure 2.6: Right figure illustrates the optimization using the gradient descent algorithm. Left figure is an example where the algorithm ”overshoots” the minimum as a consequence of a too high learning rate. Figure from Raschka and Mirjalili (2019, p. 44) /MIT license.

2.5.3 Deep learning and Multilayer Perceptron

A Multilayer Perceptron (MLP) is a variant of a multilayer neural network architecture. In addition to an input and an output, layer MLP consists of an arbitrary number of hidden layers. The hidden layers increase the learning capacity of the network compared to Adaline. In addition, MLP uses non-linear activation functions such as the sigmoid function to learn more complex problems (Raschka and Mirjalili, 2019, Ch. 12). A network is commonly considered deep when it consists of two or more hidden layers (Raschka and Mirjalili, 2019, Ch. 12). However, for easier illustration, in Fig. 2.7 we illustrate a MLP with a single hidden layer. The network can easily be generalized to include more layers.

MLP is fully connected with weight coefficients connecting one layer to the next. In Fig. 2.7 each unit in the input layer is connected with every unit in the hidden layer, and each unit in the hidden layer is connected to every unit in the output layer. A bias unit is added to the input layer referred to as $a_0^{(in)}$ in the input layer and $a_0^{(h)}$ in the hidden layer. These bias units are introduced independently in the specific layer and are not dependent on the previous layer. The bias units typically have the value 1 and the corresponding weights are adjusted during training similarly to other weights (Raschka and Mirjalili, 2019, ch. 2).

The training process of MLP can be described as follows. Each of the input signals is activated in the input layer. Then, in the hidden layer, the units here are linear combinations of signals and weights from the previous layer. These units are also activated. In the output layer, the units are linear combinations of hidden units and corresponding weights. This is called forward propagation where the input signal is sent through the network and in each layer different features of the data are learned. For each iteration, the error corresponding to each output unit is calculated as the difference between the unit and the true class label.

Like how forward propagation sends an input signal forward through the network, backpropagation sends an error signal in the opposite direction. The error signal is used for computing the loss gradient at each layer in order to update the weights.

In standard gradient descent, weights are updated after iterating through all training examples. However, when training a multilayer neural network such as MLP this approach can result in slow training. This is due to the many weights that needs to be adjusted. To speed up training, one approach is to use stochastic gradient descent where weights are updated for every training example. Another approach uses mini-batch gradient descent. A mini-batch consists of a subset of the full training data. Weight gradients from each example of the mini-batch are stored. The model is updated by the mean gradient over the mini-batch. (Raschka and Mirjalili, 2019, Ch.12)

There are some challenges of using MLP on our data, including textual descriptions of vulnerabilities. Sentences are sequential data with a time dimension. The order of which the words appear matters. MLP on the other hand, does not consider the time dimension and assumes that all the data are independently distributed.

2.5.4 Recurrent Neural Networks

In modeling sequential data, a set of architectures called Recurrent Neural Networks (RNN) is common. The basic structure of RNN are shown in Fig. 2.8. Similar to a standard feedforward network like MLP mentioned in Sec. 2.5.3, the input layer is connected to the hidden layer. However, in the hidden layer, the units are connected with what is called *recurrent* edges (Raschka and Mirjalili, 2019, Ch. 16). As a result, each hidden unit has two outgoing connections to the next layer's output unit and the hidden unit of the next time step. A simplified illustration is shown in Fig. 2.8.

In order to calculate the loss gradient responsible for updating the weights, the error in each epoch is backpropagated both through the layers, and

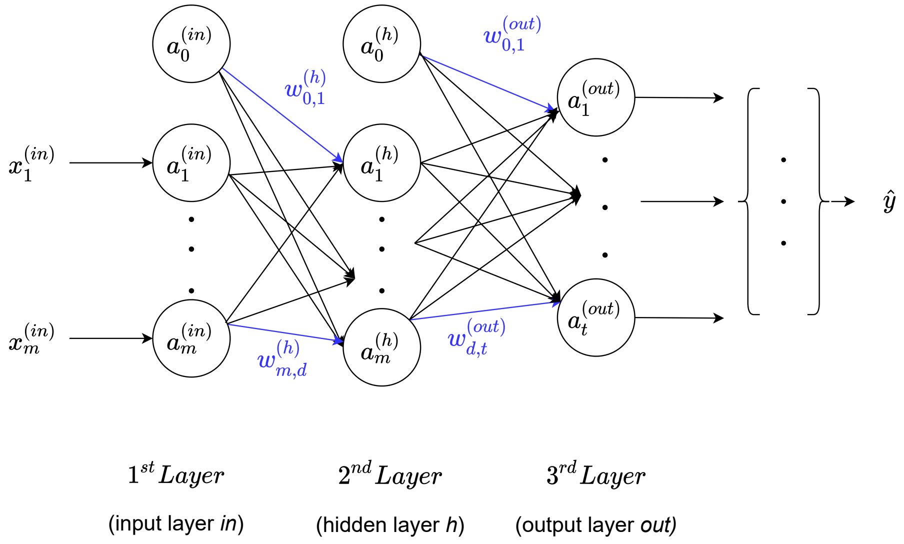


Figure 2.7: Multilayer Perceptron consisting of an input layer, a hidden layer and an output layer. Figure based on Raschka and Mirjalili (2019, p. 388).

through the time steps. This is called backpropagation through time (BPTT). For interested readers we refer to Raschka and Mirjalili (2019, p. 547) for the derivation of the gradients using BPTT. A challenge with BPTT is that the gradients will vanish as the number of time steps increases (Raschka and Mirjalili, 2019, Ch. 16). In an RNN, error signals far back in time will converge to zero. As a consequence of this, RNN has trouble learning long-range dependencies.

An example of a long-range dependency could be a relation between two words from two different sentences.

2.5.5 Long Short Term Memory

In order to solve the vanishing gradient problem and to learn long-range dependencies, Hochreiter and Schmidhuber (1997) proposed a gradient-based method called long short-term memory (LSTM), which is a variant of RNN. An LSTM consists of memory cells, and the architecture of LSTM is illustrated in Fig. 2.9. The goal is that these memory cells should memorize by preserving information from earlier time steps. An LSTM memory cell is illustrated in Fig. 2.9 and consists of a cell state, $C(t)$, in addition to four

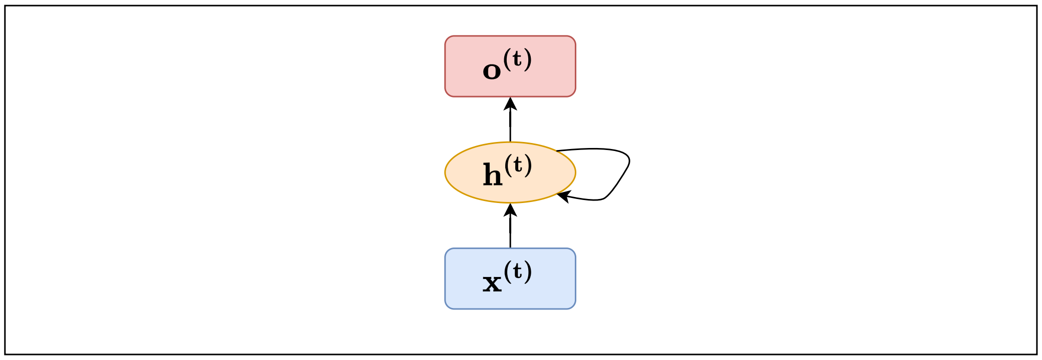


Figure 2.8: Basic RNN architecture based on Raschka and Mirjalili (2019, p. 571).

gates. The *forget gate* controls which information from the previous time to suppress and which information to keep. The forget gate is defined:

$$\boldsymbol{f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f)} \tag{2.22}$$

$\mathbf{W_{xf}}$ and $\mathbf{W_{hf}}$ are weight matrices for the input unit vector of the current time step $\mathbf{x^t}$ and the hidden unit vector of the previous time step $\mathbf{h^{(t-1)}}$ correspondingly. The indices $_{\mathbf{hf}}$ and $_{\mathbf{xf}}$ are vector multiplications. $\mathbf{b_f}$ is the bias vector. The sigmoid activation function, σ is applied to all the components in the forget gate to get a vector with elements between 0 and 1. The input gate, $\mathbf{i_t}$ and output gate, $\mathbf{o_t}$ are defined in the same way.

The candidate value updates the cell state and are given by:

$$\boldsymbol{\widetilde{C}_t = tanh(W_{xi}x^{(t)} + W_{hi}^{(t-1)} + b_t)} \tag{2.23}$$

tanh is the hyperbolic tangent activation function. This activation function returns values between 1 and -1. The cell state can then be computed:

$$\boldsymbol{C^{(t)} = (C^{(t-1)} \circ f_t) \oplus (i_t \circ \widetilde{C_t}),} \tag{2.24}$$

where $\circ$ represents elementwise multiplication and $\oplus$ elementwise addition. Elementwise multiplication between the cell state and the output gate is performed in order to compute the values of the hidden units of the current timestep:

$$\boldsymbol{h^{(t)} = o_t \circ tanh(C^{(t)})} \tag{2.25}$$

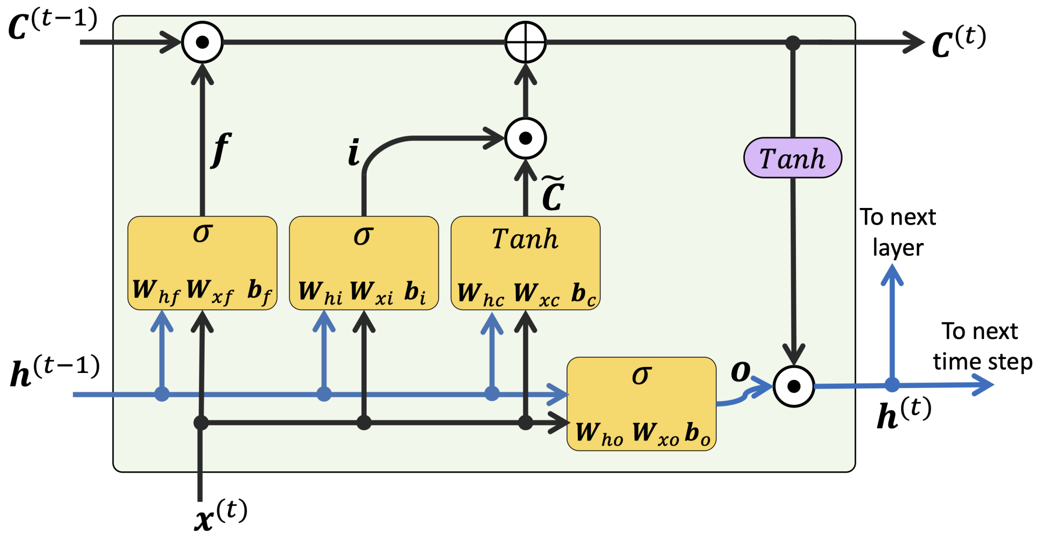


Figure 2.9: LSTM from Raschka and Mirjalili (2019, p. 582)/MIT license.

The hidden units are then passed on to the next time step and the next layer, as in regular RNN. The cell state is passed on to the next timestep.

2.5.6 Bidirectional LSTM

Bidirectional LSTM (BiLSTM) is an architecture consisting of two hidden LSTM layers independently trained with recurrent edges in opposite directions. BiLSTM is based on Bidirectional RNN (Schuster and Paliwal, 1997). In the first layer, information from a memory cell is passed forward to the next memory cell. In the second layer, the memory cells are connected in reverse order, starting from the last step. The ordinary and reverse BiLSTM learns from the same data represented differently.

The forward connected and backward connected layers are combined in the output layer to make the final prediction. This type of architecture is related to ensemble learning, where multiple classifiers capture different patterns in the data (Raschka and Mirjalili, 2019, ch. 7). The architecture is used in the RE model of MalKG by Rastogi et al. (2021).

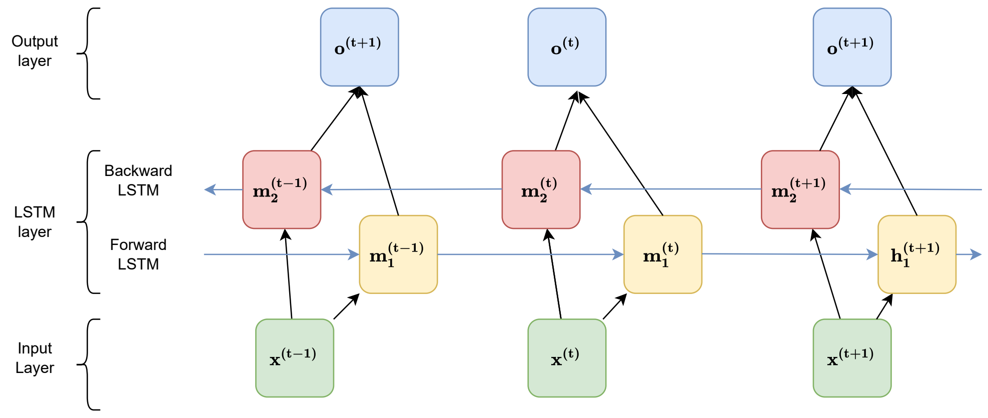


Figure 2.10: Illustration of BiLSTM

2.6 Assess overfitting and bias with learning and validation curves

How the model fits the data is vital for achieving good performance and reliable results in any machine learning problem. A model suffers from high bias if training and validation performance are both low and below the desired or expected performance. A high bias model is illustrated in Fig. 2.11 A. To avoid this, choosing a more complex model or decreasing regularising could help.

The model is overfitted when the trained model has high performance on training data but does not generalize well when making predictions on new unseen data. The model suffers from high variance. If the model is trained and evaluated on the same data repeatedly, the risk is that the model will learn from the test data, and performance on unseen data will suffer. Another reason for overfitting is that the model might be too complex for the data. If the model captures too much of the variation in the training data that does not generalize well to unseen data.

One way to detect overfitting is to compare the training and validation performance as seen Fig. 2.11 B. Here, the gap between the two curves are large, this is a clear sign of overfitting.

If a model overfitts gathering more data could help. This could make the model learn from a larger data set with more variation. As seen in Fig. 2.11 B, performance continues to increase with more training samples.

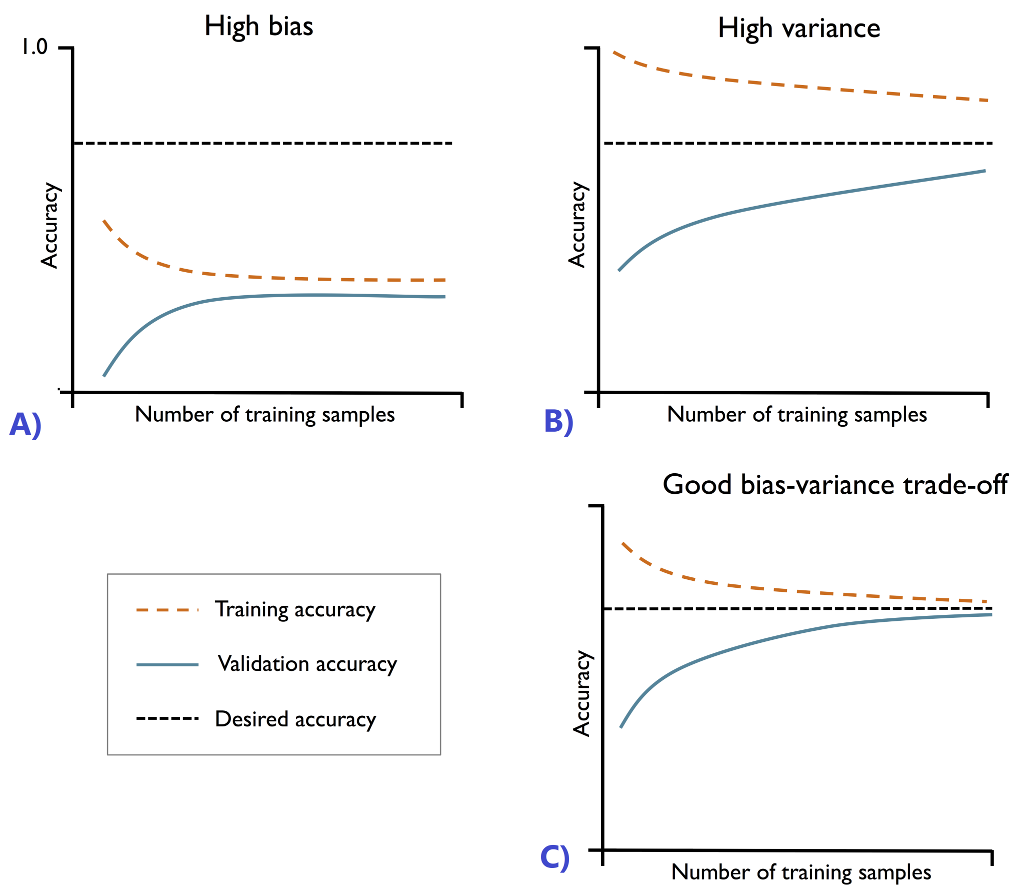


Figure 2.11: Overfitting versus bias. Figure from Raschka and Mirjalili (2019, p. 202) /MIT License

Unfortunately, this is not always feasible given resources or other limitations. Another alternative is choosing a less complex model with fewer parameters, e.g. a linear model. Regularisation can also be used to penalize large weights forcing the model to emphasize extreme cases. The optimal model choice is when training and validation curves are close, and performance is at the desired level illustrated in Fig. 2.11 C (Raschka and Mirjalili, 2019, p. 201f).

2.7 Knowledge graph embedding

We can use machine learning algorithms in several tasks involving knowledge graphs, including entity and relation prediction. As with word embeddings, we usually need numeric vector representations for these algorithms to work.

A first attempt to create a numeric representation of the graph is using a one-hot encoding approach such as the binary bag of words.

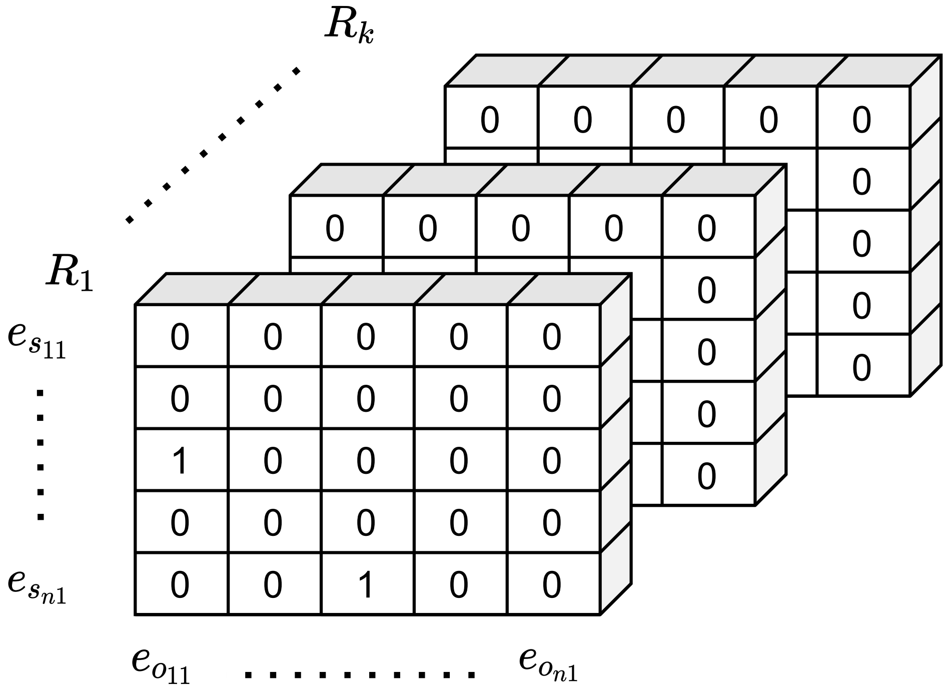


Figure 2.12: Binary tensor. The example tensor consists of three matrices. Each matrix R_i corresponds to one relation between subject entities, the rows, and object entities the columns of the matrix. E.g $e_{s_{11}}$ is the first subject entity of the first relation.

A triple consist of a subject entity an object entity and a relation between these. The subject entities, e_s, and object entities, e_o, can be encoded as vectors $\boldsymbol{e_s}, \boldsymbol{e_o} \in \mathbb{R}^{n_e}$ and the relations as matrices $\boldsymbol{R} \in \mathbb{R}^{n_e \times n_e}$, where n_e is the number of entities. Together this will form a tensor shown in Fig. 2.12. A tensor is a generalization of a matrix. 0-order, 1-order and 2-order tensors refer to scalars, vectors and matrices. Tensors ranging from 3-order into any arbitrary order are called higher-order tensors (Hogan et al., 2021, ch.5.2).

The challenge with a one-hot encoded graph is that if most subject and object entities are not related, these binary tensors will become sparse. KG embedding encodes entities and relations into a dense representation.

Some embedding algorithms encode separately into entity and relation embeddings, while others use a core tensor storing shared information across the embeddings. The dimension of these embeddings is usually much lower than the one-hot encoded vectors and typically ranges between 50 and 1000. As a result of compressing the vectors into lower dimensions, the embedding algorithms are forced to learn from data and can thus capture latent structures of the graph (Balažević et al., 2019).

By capturing latent structures in the graph, knowledge graph embedding algorithms should encode nodes with similar characteristics closer in the vector space compared to unrelated nodes. This feature can be helpful for several tasks related to our problem. In our case, a successful embedding should encode vulnerabilities with similar characteristics in the same neighborhood. We can then use entity prediction to predict missing information about the vulnerability. We will aim to use these embedding to predict vulnerability types.

In the following sections, some embedding algorithms will be presented. These algorithms' strengths and weaknesses are considered to choose the most promising for our use case.

2.7.1 TransE

Translational models is one class of embedding algorithms. The most basic of these is TransE introduced in Bordes et al. (2013). As above we define subject entity, e_s, relation, r, and object entity, e_o, with corresponding embedding vectors in bold notation. Then from a set, S, of triples (e_s, r, e_o), embeddings are constructed in $\mathbb{R}^k$ where k is a hyperparameter representing the embedding dimensionality. The goal is to construct embeddings such that $\mathbf{e_s} + \mathbf{r} \approx \mathbf{e_o}$ when (e_s, r, e_o) holds. In a good embedding, $\mathbf{e_o}$, should be the nearest neighbor of $\mathbf{e_s} + \mathbf{r}$. At the same time, in the case where (e_s, r, e_o) does not hold, $\mathbf{e_s} + \mathbf{r}$ should be far away from $\mathbf{e_o}$. The embeddings are learned from the training set by minimizing the following loss function.

$$\nabla = \sum_{(e_s,r,e_o)\in S} \sum_{(e'_s,r,e'_o)\in S'_{(e_s,r,e_o)}} [\gamma + d(\mathbf{e_s} + \mathbf{r}, \mathbf{e_o}) - d(\mathbf{e'_s} + \mathbf{r}, \mathbf{e'_o})]_+ \tag{2.26}$$

In a corrupted triple (e_s',r,e_o'), either the head or the tail entity has been replaced. The set of corrupted triples S' is defined:

$$S'_{(e_s,r,e_o)} = \{(e'_s, r, e_o)|e'_s \in E\} \cup \{(e_s, r, e'_o)|e'_o \in E\} \tag{2.27}$$

$[]_+$ in (2.26) specifies that only the positive part is considered during optimization. d is a dissimilarity measure assumed to be either the L_1 or the L_2 norm. For each triple in the training set, dissimilarity is measured both for the original triple and the corrupted triple. A good embedding algorithm will output a lower dissimilarity for a true triple, (e_s, r, e_o), than a corrupted triple, (e'_s, r, e'_o). A margin, $\gamma > 0$, is added to the loss function to ensure this.

Optimization is done using mini-batch stochastic gradient descent. In addition, the L_2 norm of the entity vectors is used for optimization. Without regularization, if the norm of e_s and e_o are increased in (2.26) while γ is kept constant, the effect of γ will diminish, which could result in an artificial minimizing loss. Normalization using the L_2 norm is performed to the entity vectors during optimization. (Bordes et al., 2013)

TransE has some limitations and might, in some cases, be too simplistic. This includes potential flaws when transforming 1-to-n, n-to-1 and n-to-n relations (Hogan et al., 2021, Ch 5.2). We consider an example of a CVE record which consist of multiple vulnerability-relevant terms in its description which is a 1-to-n relation. TransE will aim to assign a similar relation vector to all relevant terms. However, the vulnerability-related terms have different entity embeddings which could be encoded far away from each other.

2.7.2 Tensor decomposition

Tensor decomposition is another approach to create the embeddings. Tensor decomposition involves a set of techniques to decompose a tensor into two or more elementary tensors of lower order. The goal is that the elementary tensors should capture the underlying information from the original tensor (Hogan et al., 2021). As an example, we define a matrix $\boldsymbol{C}$ of $m \times n$ elements as the outer product of two vectors, $\boldsymbol{x}$ and $\boldsymbol{y}$.

$$\boldsymbol{x} \otimes \boldsymbol{y} = \boldsymbol{C} \tag{2.28}$$

C is a *rank-1* matrix. This is because the columns of C are *linearly dependent*. By scaling one column we can get all other columns.

C could then be precisely encoded using $m + n$ elements instead of the full matrix containing $m \times n$ elements. However, most of the times the rank of **C** is more than one. If we assume the rank of **C** to be a matrix of rank r, then we need to sum r rank-1 matrices to precisely decompose **C**. Since this may become computationally expensive for matrices with high rank, *rank decomposition* sets a limit, d, on the rank of **C**. The sum of the d matrices are computed to best approximate **C** in (**??**). Rank decomposition generalized and applied to tensors is referred to as CP-decomposition (Rabanser et al., 2017). To compute the CP-decompositions different algorithms exists further explained in Rabanser et al. (2017).

In knowledge graphs, the initial one-hot encoded representation of the graph mentioned in 2.7 can be decomposed using CP-decomposition.

Three matrices, $\mathbf{X} = [\mathbf{x_1}...\mathbf{x_d}]$, $\mathbf{Y} = [\mathbf{y_1}...\mathbf{y_d}]$ and $\mathbf{Z} = [\mathbf{z_1}..\mathbf{z_d}]$ are components in the decomposition. The i^{th} row of $\mathbf{Y}$ is the embedding of the i^{th} relation. Furthermore, the j^{th} row of $\boldsymbol{X}$ and $\mathbf{Z}$ are both embeddings of the j^{th} entity. We define the tensor decomposition of $\mathcal{G}$:

$$\boldsymbol{x_1} \otimes \boldsymbol{y_1} \otimes \boldsymbol{z_1} + ... + \boldsymbol{x_d} \otimes \boldsymbol{y_d} \otimes \boldsymbol{z_d} \approx \boldsymbol{\mathcal{G}} \tag{2.29}$$

In the binary one-hot encoded graph the same entity also has different embeddings depending on whether it is the subject or object in a particular triple. Many Knowledge Graphs, though, typically aim to assign one unique embedding to each entity (Hogan et al., 2021, Ch.5.2).

DistMult Yang et al. (2015) is a knowledge graph embedding method based on CP-rank decomposition. In DistMult a plausibility function is defined as $\sum_{i=1}^{d}(\mathbf{e}_s)_i(\mathbf{r}_p)_i(\mathbf{e}_o)_i$. Where $\mathbf{e}_s$, $\mathbf{r}_p$ and $\mathbf{e}_s$ are subject entity, relation and object entity embeddings correspondingly. The goal is to construct the embeddings such that the plausibility of positive edges are maximized and the plausibility of negative (false) edges is minimized. Instead of using different matrices for head and tail entities, a single entity matrix $\mathbf{E} = [\mathbf{e_1}...\mathbf{e_d}]$ is applied twice such that:

$$\boldsymbol{e_1} \otimes \boldsymbol{r_1} \otimes \boldsymbol{e_1} + \boldsymbol{...} + \boldsymbol{e_d} \otimes \boldsymbol{r_d} \otimes \boldsymbol{e_d} \approx \boldsymbol{\mathcal{G}} \tag{2.30}$$

One limitation with using the same matrix for subject and object entities is that the direction of the relation is not considered (Hogan et al., 2021, ch. 5.2).

Another embedding approach for knowledge graph embedding which is based on tensor decomposition is TuckER, also mentioned in chapter 4.3. TuckER is considered current state of the art when benchmarked on several generic data sets (Balažević et al., 2019). TuckER is based on Tucker decomposition from Tucker (1966) which decomposes a tensor into one core tensor and a set of matrices. We assume an original third order tensor defined as $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, the core tensor as $\mathcal{G} \in \mathbb{R}^{P \times Q \times R}$ and matrices $\mathbf{A} \in \mathbb{R}^{I \times P}$, $\mathbf{B} \in \mathbb{R}^{J \times K}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$. Tucker decomposition can be formulated as the following a minimization problem:

$$\begin{aligned} & min_{\hat{\mathcal{X}}} ||\mathcal{X} - \hat{\mathcal{X}}|| \quad \text{with} \\ \hat{\mathcal{X}} & = \sum_{p=1}^{P} \sum_{q=1}^{Q} \sum_{r=1}^{R} g_{pqr} \mathbf{a}_p \otimes \mathbf{b}_q \otimes \mathbf{c}_r \\ & = \mathcal{G} \times_1 \mathbf{A} \times_2 \mathbf{B} \times_3 \mathbf{C} \end{aligned} \tag{2.31}$$

$\mathbf{a_p}$, $\mathbf{b_q}$ and $\mathbf{c_r}$ are column vectors of the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ correspondingly. When considering tensors, the outer product is commonly referred to as tensor product. A mode n tensor is another term of a tensor of order n. In the last line of the above formula, $\times_n$, is the tensor product along the n-th mode Rabanser et al. (2017). $\hat{\mathcal{X}}$ is the approximation of $\mathcal{X}$. The core tensor $\mathcal{G}$ stores the level of interaction between the factor matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$. $\mathcal{G}$ is usually a compressed version of $\mathcal{X}$ with lower dimensions. Tucker decomposition can be viewed as a form of higher-order Principal Component Analysis where the factor matrices are the principal components that best capture the variation in each mode. For additional information about tensor products and tensor decomposition approaches we refer to Rabanser et al. (2017).

In Balažević et al. (2019), knowledge graph embeddings are created applying Tucker decomposition to a one-hot encoded graph. Similar to DistMult, two equivalent matrices holds the entity embeddings for both subject and object entities defined as $\mathbf{E} = \mathbf{A} = \mathbf{C} \in \mathbb{R}^{n_e \times d_e}$. The relation embeddings are defined as $\mathbf{R} = \mathbf{B} \in \mathbb{R}^{n_r \times d_r}$ and the core tensor are defined as $\mathcal{W} \in \mathbb{R}^{d_e \times d_r \times d_e}$. n_e and n_r are the number of entities and relations while d_e and d_r are the dimensions of the embeddings.

The scoring function of TuckER is defined in Balažević et al. (2019) as:

$$\phi(e_s, r, e_o) = \mathcal{W} \times_1 \mathbf{e}_s \times_2 \mathbf{w}_r \times_3 \mathbf{e}_o \tag{2.32}$$

In 2.32, the subject embeddings $\mathbf{e}_s$ and the object embeddings $\mathbf{e}_o$ are rows in $\mathbf{E}$. The relation embeddings $\mathbf{w}_r$ are rows in $\mathbf{R}$. In order to obtain the plausibility of a given triple being true, the logistic sigmoid function is applied to each score element in (2.32). (Balažević et al., 2019)

Contrary to DistMult, not all information about relations and entities are stored in $\mathbf{R}$ and $\mathbf{E}$. Some information is instead stored in the core tensor $\mathcal{W}$. Balažević et al. (2019) view the matrices of the core tensor as prototype relations which are linearly combined according to the parameters in each relation embedding. This parameter sharing across relation embeddings will increase computational performance since fewer parameters must be adjusted during training. The core tensor used in TuckER also ensures that the embedding algorithm is *fully expressive*, which means that TuckER have the potential to capture all information from the original graph.(Balažević et al., 2019)

2.8 Hyperparameters and configurations of TuckER

TuckER includes a set of hyper parameters that must be tuned to achieve optimal performance. The learning rate was introduced in Sec. 2.5.2 and is also used in TuckER adjusting the speed of training. Learning rate decay is another hyperparameter used to decrease the learning rate during training. A higher learning rate for earlier epochs could be helpful for computational efficiency, while a lower learning rate for later epochs increases the chance of reaching the optimum without overshooting it. Dropout is a *regularisation* technique and has been applied in neural networks to avoid overfitting (Srivastava et al., 2014). The risk of overfitting in TuckER increases when there are few triples relative to the number of relations Balažević et al. (2019).

When applying dropout in TuckER, a proportion of the embedding weights are randomly set to zero during training.

2.8.1 Batch Normalization

Batch Normalization is a technique introduced to prevent shifts in the weight distribution during training. The idea is that using batch normalization could help reaching a better optimum in addition to making training faster (Raschka and Mirjalili, 2019, Ch. 17 p. 649f). In the following batch normalization is performed to entity embeddings in TuckER. Consider an arbitrary entity embedding $\boldsymbol{e^{[i]}}$ with embedding dimension, c, and batch size, $\boldsymbol{m}$. The batch-wise mean and standard deviation of $\boldsymbol{e^{[i]}}$ is defined:

$$\boldsymbol{\mu} = \frac{1}{m}\sum_i \boldsymbol{e^{[i]}}, \quad \boldsymbol{\sigma^2} = \frac{1}{m}\sum_i (\boldsymbol{e^{[i]}} - \boldsymbol{\mu})^2 \tag{2.33}$$

Embedding $\boldsymbol{e^{[i]}}$ is then standardized over all examples in a batch:

$$\boldsymbol{e^{[i]}_{std}} = \frac{\boldsymbol{e^{[i]}} - \boldsymbol{\mu}}{\boldsymbol{\sigma} + \epsilon} \tag{2.34}$$

The standardized embedding has center mean and unit variance. And the ϵ is a small positive value added to avoid division by zero.

The standardized embedding is then scaled and shifted using two learning parameters, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ to get the batch normalized embedding, $\boldsymbol{y^{[i]}}$:

$$\boldsymbol{y^{[i]}} = \boldsymbol{\gamma} \boldsymbol{e^{[i]}_{std}} + \boldsymbol{\beta} \tag{2.35}$$

Chapter 3

Methods and materials

3.1 Data transformations

3.2 Architecture

We download yearly JSON feeds of CVE records from 2003 until October 2021 from NVD. The data contains vulnerability descriptions which are textual descriptions of the vulnerability records, severity metrics, affected product configurations (CPE), links to references, and classification in vulnerability types. The data contains approximately 150 000 CVEs, 269 unique weakness types, and almost 270 000 unique CPEs, as seen in Table 3.1.

3.3 Processing and labeling

The files must be processed and adjusted before labeling can be performed. We based our approach on code from Bhandari et al. (2021) to process our initial data by removing CVEs without a public reference describing the vulnerability. When a CVE is rejected all public references are removed.

CVE count	CWE count	CPE count
150397	269	266658

Table 3.1: Raw data statistics

CVEs can be rejected for several reasons. One example is when the CVE is a duplicate of another CVE. Another example could be that further research determines that the issue is not a vulnerability after all [1].

We extended the code from Bhandari et al. (2021) by further processing the data in a graphson file consisting of edges and vertices. We did this to make our data compatible with the auto-labeling tool, which we explain in the following paragraph. Each CVE record has outgoing edges to the corresponding CPE objects. Each CVE record has one vertex. CVE ID, CWE ID, description, and labeled data are properties of a vertex. An example of how the data looks like for a single CVE record after processing is shown in listing **??**.

Listing 3.1: CVE-2020-17004 after processing

```
{"edges": [{'_id': 'CVE-2020-17004_to_cpe:2.3:o:
   microsoft:windows_10:-:*:*:*:*:*:*:*',
  '_inV': 'CVE-2020-17004',
  '_label': 'has',
  '_outV': 'cpe:2.3:o:microsoft:windows_10:-:*:*:*:*
    :*:*:*',
  '_type': 'edge'},
 {'_id': 'CVE-2020-17004_to_cpe:2.3:o:microsoft:
    windows_10:20h2:*:*:*:*:*:*:*',
  '_inV': 'CVE-2020-17004',
  '_label': 'has',
  '_outV': 'cpe:2.3:o:microsoft:windows_10:20h2:*:*:
    *:*:*:*:*',
  '_type': 'edge'}],
 "vertices": [{'_id': 'CVE-2020-17004',
 '_type': 'vertex',
 'cwe': 'NVD-CWE-noinfo',
 'description': 'Windows Graphics Component
    Information Disclosure Vulnerability'}
```

The processed CVE records were labeled using the auto labeling technique proposed by Bridges et al. (2014), which was introduced in Chapter 2.2. We used the corresponding source code which can be found in GitHub Bridges

[1]https://www.cve.org/ResourcesSupport/FAQs

et al. (2014) [2]. A few adjustments were made to the original labeling script to make it work with our data. We adjusted the script to the current CPE 2.3 standard from the original CPE 2.2 standard. In addition, we corrected a mistake in the code where the hardware version was initially not extracted.

In listings 3.2 the CVE record from above, listings 3.1, is labeled. The labels which are stored under "tagged_text" is added to the vertex in listings 3.1. Three types of labeling are used. These are IOB labels, domain labels, and POS labels. In listings 3.2, the word "Information" has IOB-label "B", domain label "relevant_term" and POS label "NNP". "NNP" is proper noun, see for example [3].

Listing 3.2: CVE-2020-17004 processed and labeled

```
{'tagged_text': [['Windows', 'O', 'NNS'],
 ['Graphics', 'O', 'NNP'],
 ['Component', 'O', 'NNP'],
 ['Information', 'B:relevant_term', 'NNP'],
 ['Disclosure', 'I:relevant_term', 'NNP'],
 ['Vulnerability', 'O', 'NNP']]}
```

3.3.1 Named Entity Recognition and Relation Extraction

Additional processing was required before we could train the Averaged perceptron model for Named Entity Recognition on the labeled data. We first removed the CPE objects, and then stored the data in lists. For each CVE record, we stored the CVE-ID, CWE-ID, and four sublists corresponding to the words, pos labels, IOB-labels, and domain labels. Using the same example as above, CVE-2020-17004, listings 3.3 show how the data is formatted in Averaged Perceptron.

Listing 3.3: CVE-2020-17004 processed for Averaged Perceptron

```
['CVE-2020-17004',
 'NVD-CWE-noinfo',
 ['Windows', 'Graphics', 'Component', 'Information',
    'Disclosure', 'Vulnerability'],
```

[2]https://github.com/stucco/auto-labeled-corpus
[3]https://cs.nyu.edu/ grishman/jet/guide/PennPOS.html

```
['NNS', 'NNP', 'NNP', 'NNP', 'NNP', 'NNP'],
['O', 'O', 'O', 'B', 'I', 'O'],
['O', 'O', 'O', 'relevant_term', 'relevant_term', '
   O']]
```

We separately trained two averaged perceptron models, one for domain-labeling and another for IOB-labeling, using the same training data for both models. We based most of our implementation on the approach made by Bridges et al. (2014).

When evaluating the model, Bridges et al. (2014) compares the predicted labels with the labels from the auto-labeling tool, which are considered ground truth. The test performance of the two models is reported separately.

We noticed some differences between the paper and the corresponding code from Bridges et al. (2014). Some of the features included in the paper were missing in the code. In both the IOB and domain model, the feature with the part of speech label for the current word were missing. In addition, the domain model was missing the feature with the IOB label of the current word. We included these features in our implementation. We used the same features as provided in the paper by Bridges et al. (2014). Two sets of features were provided for IOB-labeling and domain labeling correspondingly.

Bridges et al. (2014) refer to their implementation as a history-based Maximum Entropy Model. History-based means that the current and previous words and labels are considered. However, as seen in the list of features provided by Bridges et al. (2014) below, words and labels after the current word are also considered.

Features for IOB-labeling

- Unigram features for
  - previous two, current, and following two words
  - previous two, current and following one part of speech tags
  - previous two 'IOB'-tags
- Bigram features for
  - previous two 'IOB'-tags
  - previous 'IOB'-tag and current word

  - previous part of speech tag and current word

- Regular expressions as listed above for
  - previous two, current, and following two words

Features for domain labeling

- Unigram features for
  - previous two, current, and following two words
  - previous two, current, and following 'IOB'-tags
  - previous two domain labels
- Bigram features for
  - previous two domain tags
  - previous domain tag and current word
  - previous 'IOB'-tag and current word
  - previous part of speech tag and current word
- Regular expressions as listed above for
  - previous two, current, and following two words
- Gazetteer features for
  - Software Product
  - Software Vendor

We use the same set of features in our implementation. The regular expression features adjust for variations of the words, this includes words with digits, punctuation, capital letter, or words written as snake-case and camel-case Bridges et al. (2014).

When comparing the paper and code by Bridges et al. (2014) we also noticed another difference regarding how the model makes predictions. The Averaged perceptron model keeps a dictionary of unambiguous words from the auto-labeled data in the code. This dictionary contains the unambiguous words as keys and the corresponding labels as values. If the dictionary contains the current word, the corresponding label is the predicted label.

The trained model is then only used when the word is unambiguous. For example, if the word "kernel" is labeled as "relevant term" one place and as "os" another place, the word is unambiguous. The Averaged Perceptron algorithm is used on the unambiguous words.

We trained two models for our named entity recognition model based on the Averaged perceptron. We used the Averaged Perceptron algorithm for training and evaluation on all available data in our first approach. In our second approach, we used the Averaged Perceptron algorithm for training, and then for evaluation, we used the labeled dictionary for the unambiguous words and the Averaged Perceptron for the ambiguous words.

Due to slow training, we restricted our training data to 4000 CVE descriptions. Then we evaluated the model's performance on the remaining data consisting of approximately 146 000 CVE descriptions.

The IOB-labels contain helpful information about multi-word entities. For example, the two words "Information" and "Disclosure" are one entity with "B" and "I" as corresponding IOB-labels. "B" indicates the start of an entity, and "I" indicates that the second word is inside the same entity. IOB-labels do not contain a stop label, but if we encounter a "B" or an "O", we know a new entity has started.

The labeled data are used in the relation extraction model constructing our initial knowledge graph, illustrated in Fig. 3.1. We removed all entities labeled as outside as we did not want to keep these words in our graph. The remaining IOB- and domain-labels were then used to form entities of multiple words. We connect the extracted entities to the corresponding CVE-ID. Since each CVE-ID also has a CWE-ID we also connected the same entities to the CWE-ID. We made rules to extract relations between the entities inside a CVE record. When encountering a product entity, we connect all relevant terms corresponding to the same CVE to this product entity. The CPE vector organizes entities such that vendor comes before application and application before version. We assumed that the same structure was used in the CVE description, and when exploring our data, we found that in approximately two-thirds of the cases, the product follows the vendor and the version follows the product.

We made rules to extract the relation between these entities as follows. When we encounter a vendor entity, we check the following entity. If the following entity is a product that could be either application, hardware or os, we make a relation from the product to the vendor.

We also made relations from product to version similarly. A difference when checking for versions of a product is that we continue looking as long as the next word is version. In this way, we hope that all versions of a related product are extracted from the description.

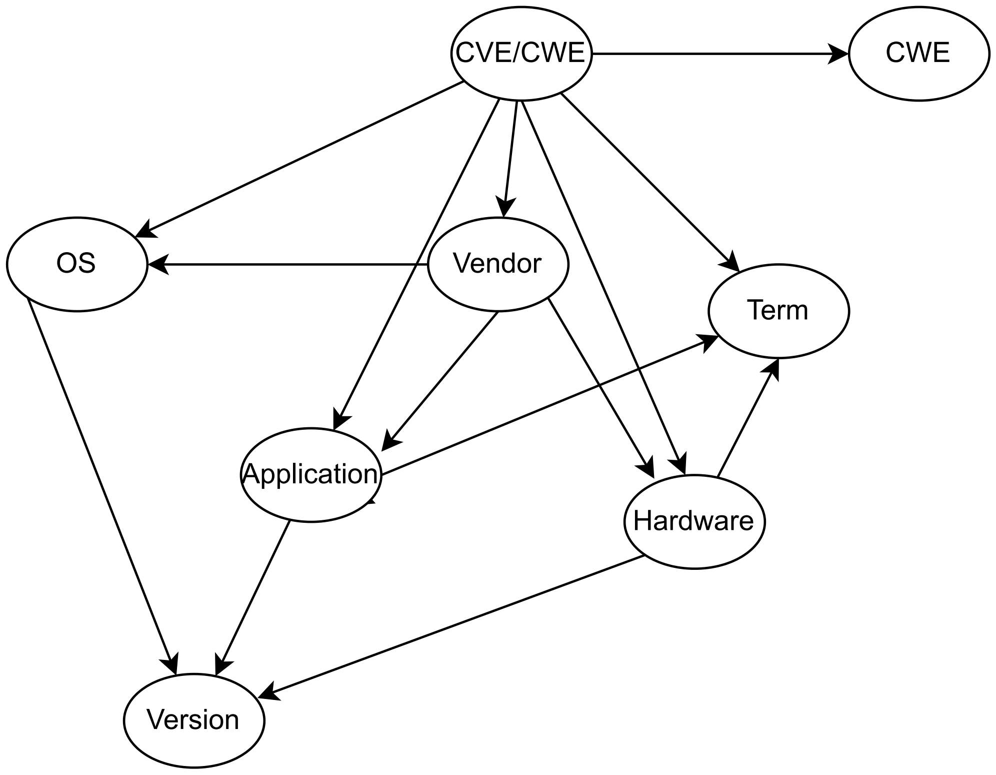


Figure 3.1: Initial knowledge graph after relation extraction is performed.

3.3.2 Entity prediction

Entity prediction using KG embedding is the final step of the proposed architecture. We use the processed relations from the relation extraction model as input in the TuckER model introduced in sec. 2.7.2. Data augmentation were then performed by reversing all the relations. The data set were split into 80 percent for training, 10 percent for validation, and 10 percent for testing. Misclassifications could happen both during entity extraction and relations extraction. However, TuckER assumes that the provided input triples are true.

As mentioned in Sec. 2.8, hyper parameters must be tuned in order for the model to obtain optimal performance. Our first approach was to train the same four combinations of hyperparameters as proposed in (Balažević et al., 2019). All models were run for 100 epochs, and based on the intermediate results, the most promising model was run for additional 200 epochs such that this model was trained for 300 epochs in total.

We evaluated the results from this approach and set up a grid search covering 36 additional hyperparameter combinations focusing on drop out rates and learning rates. Due to slow training of the full data set, we made smaller subsets of the data. These subsets were random samples taken from the training data consisting of 100 000 triples or approximately five percent of the training data. To avoid overfitting, two models were trained and validated for each of the hyper parameter combinations on different samples of the data. We then calculated the mean Hits@10 metric for each combination and ranked them. The parameters from the most promising candidate were used in training a model for 300 epochs on the full data set.

3.4 Specifications of software and hardware

All code regarding this thesis is stored in a GitHub repository [4]. The data is stored in a shared google drive folder. Neptune is a project for tracking machine learning experiments. We used this tool to track output logs, hyper-parameters, performance metrics, and more. Python version 3.6.9 is used in the project, and in requirements.txt, all needed libraries are specified. This file was used to create a virtual environment where all code related to this thesis could be run.

For large jobs, we need high-performance computing power. The computing clusters from the eX3 infrastructure (ex3, 2022) hosted by Simula Research Laboratory in addition to the NBMU Orion HPC Cluster were used (NMBU, 2022). Running jobs on eX3, we used two different partitions depending on the type of job. For jobs involving GPU training with the CUDA platform (CUD, 2013), the dgx2q partition was used. The dgx2q has 16 Nvidia Tesla V100 GPUs where each GPU has 32GB of ram. For long

[4]https://github.com/secureIT-project/anders-msc

sequential jobs, the slowq partition was used. This partition contains 8 single processor nodes with Intel Xeon Silver 4112 CPU. The CPU operates at 2.60GHz with four cores.

Running jobs on NMBU Orion HPC Cluster we used the gpu partition with 4 x 256 GB RAM, 64 cores, 3 Quadro RTX 8000 from Nvidia. A Huawei Matebook D where used for initial prototype testing. The computer has a AMD ryzen 5 processor with 2100 Mhz quad-core processor and 8 GB of ram.

Chapter 4

Related Work

4.1 Related data

CVEfixes is a data set constructed by Bhandari et al. (2021) mining vulnerable and patched code from open source projects stored in the code repositories GitHub, GitLab, and Bitbucket. CVEfixes automatically fetches new commits to the mentioned repositories keeping them up to date with the most recent vulnerability patches. In addition to code-related data, CVEfixes fetches CVE records from NVD and vulnerability types from Common Weakness Enumeration, CWE [1]. The mentioned data sources are combined and stored in a relational database. This rich data set provides multiple opportunities for constructing and refining a vulnerability KG. In this thesis, the focus has been on utilizing vulnerability databases, leaving code-related data for future work.

4.2 Entity and relation extraction of cybersecurity concepts

Gasmi et al. (2019) proposed a method for entity and relation extraction and applied this to the NVD data. The NVD data are labeled using the auto-label tool provided by Bridges et al. (2014). Gasmi et al. (2019) then compares two models of entity extraction where one of them is combining LSTM and CRF (Conditional Random fields) while the other is based on

[1]https://cwe.mitre.org/index.html

CRF alone. The LSTM-CRF is the best performer with an F1 score on the test data reported to be 0.8337 on average for all entity types (Gasmi et al., 2019, Table 4.). Bridges et al. (2014) trains an Averaged Perceptron model and reports the performance in terms of F1 score as 0.965. Comparing the results of Gasmi et al. (2019) and Bridges et al. (2014), the averaged perceptron performs better on the given data. However, the AveragedPerceptron depends more heavily on choosing the best features, which is more labor costly and dependent on domain knowledge (Gasmi et al., 2019).

Gasmi et al. (2019) also proposes three relation extraction models based on the LSTM model. The data set considered consists of all CVE records from 2015 collected from NVD data. The best model performs 0.943 in F1-score. A challenge is that the data is not labeled with relations. Gasmi et al. (2019) uses a semi-supervised approach where important labeled relations are provided as seed values to the model. The iterative algorithm builds on the seed values increasing the number of relations in the process. The approach was originally proposed by Jones et al. (2015) on a data set consisting of 62 news articles, blogs, and updates from a variety of security-related websites. Jones et al. (2015) reports the performance of 0.82 in precision.

4.3 Related Knowledge Graphs

4.3.1 VulKG

The vulnerability knowledge graph (VulKG) was proposed by Qin and Chow (2019). This graph extracts vulnerability information from the NVD. The authors define a vulnerability-related ontology to which knowledge is connected. Qin and Chow (2019) proposes a theoretical framework architecture. The framework includes a Natural language processing (NLP) part and a reasoning part. In the NLP part, entities from CVE descriptions are extracted through Named Entity Recognition (NER) using a BiLSTM-CRF (Bi-directional Long Short Term Memory - Conditional Random Fields) model. This model is based on results by Gasmi et al. (2019) who proposed a NER model for cybersecurity concepts. The model by Gasmi et al. (2019) is trained on automatically labeled data relying on work by Bridges et al. (2014). The extracted entities are aligned with the underlying *VulKG* ontology as seen in Fig. 4.1 in order to find the relations between the entities.

Reasoning is used to find hidden rules in *VulKG* where another goal

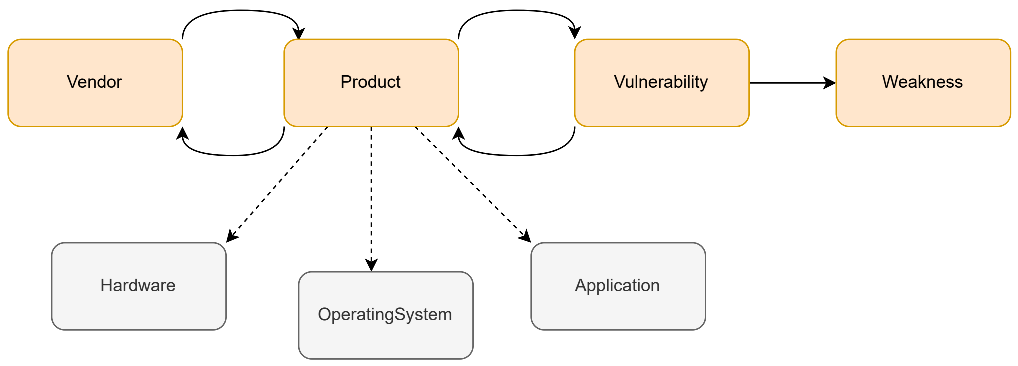


Figure 4.1: VulKG ontology. From Qin and Chow (2019, Fig. 3).

of *VulKG* is to find new weakness chains. If a weakness triggers another weakness, the weaknesses are part of a weakness chain (MITRE, 2021b). (Qin and Chow, 2019). Qin and Chow (2019) finds candidate weakness chains by computing the chain confidence. We define two weaknesses as CWE1 and CWE2. The chain confidence is calculated as the conditional probability of finding CWE2 in a product given that the product already has weakness CWE1. The chain confidence is calculated as follows:

$$\text{Chain Confidence} = \frac{C(\text{Pr with CWE1} \cap \text{Pr with CWE2})}{\text{C(Pr with CWE1)}} \tag{4.1}$$

C(Pr with CWE1), in short, C_1, is the number of products which have weakness, CWE1. $C(\text{Pr with CWE1} \cap \text{Pr with CWE2})$, in short C_{12}, is the number of products which have both weakness CWE1 and CWE2 (Qin and Chow, 2019). To be considered a chain, (Qin and Chow, 2019) requires the chain confidence to be above a threshold of 0.2 and C_1 and C_{12} both to be above 100. Qin and Chow (2019) constructs their knowledge graph from NVD data. Each record of NVD consists of a CVE record enriched with data, including product configurations (CPE) and weakness information (CWE). To connect a product with a weakness, collecting all CVE records with related product configurations is necessary. In order to compute the chain confidence, Qin and Chow (2019) count the CPE objects, which are the nodes of the linking path between the two CWE objects illustrated in Fig. 4.2. By linking two CWE objects sharing CPE objects, we infer that multiple CVE objects are also linked. For example, it is not possible to make a connection from CWE-79 to CWE-918 in Fig. 4.2 without including two CVE objects in the connecting path.

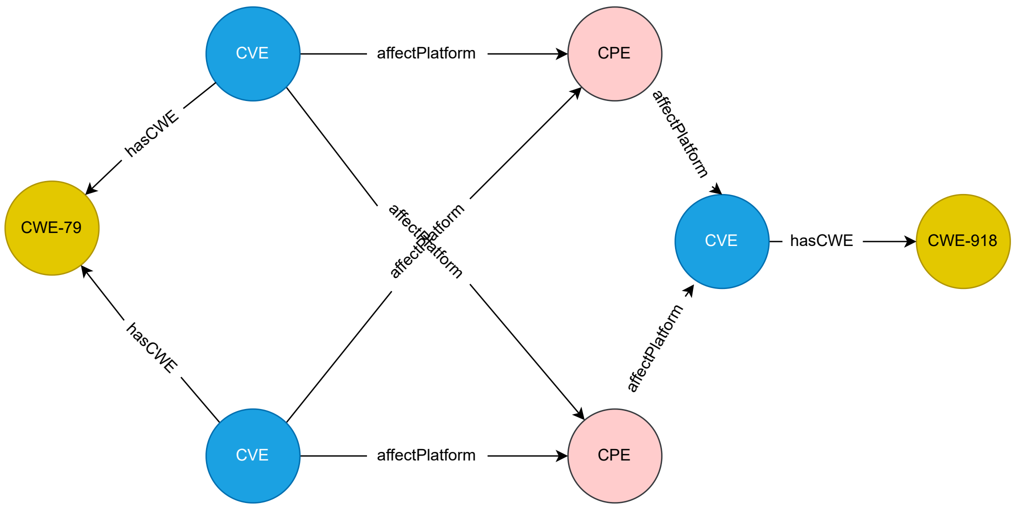


Figure 4.2: VulKG weakness chain. Based on Qin and Chow (2019, Fig. 10a).

How CVE records are assigned in the CVE dataset should follow a set of rules MITRE (2021a). One of these rules states that connecting two CVEs cannot be dependent on each other. Rule 7.2.2 states:

> CNAs MUST NOT assign a CVE ID to a vulnerability that is dependent on another vulnerability. The dependent vulnerability should share the same CVE ID as the vulnerability it is dependent on.

CNA is an abbreviation for CVE Numbering Authorities and CNAs are authorized organizations tasked with assigning new CVE records. Another potential flaw in *VulKG* reasoning concerns graph theory. *VulKG* ontology defines a directed relation from vulnerability (CVE) to weakness (CWE). In Fig. 4.2 we recognize directed edges from the CVEs to the CWEs. Since these relations are not bidirectional, no valid path exists linking CWE-79 with CWE-918.

4.3.2 MalKG

In Rastogi et al. (2021) the author constructs a Malware Knowledge Graph (MalKG). At the time of our replication, MalKG was published open source in GitHub [2]. We cloned the mentioned repository in order to replicate their work.

Our goal with replicating the work of Rastogi et al. (2021) is to understand further how knowledge graphs are used within the broader field of cyber security. Malware and vulnerabilities are closely related, making MalKG a useful starting point before constructing our own knowledge graph.

A large amount of threat intelligence is published by security organizations, government agencies, and research institutions containing information about recent attacks and mitigation strategies. Rastogi et al. (2021) considers a data set consisting of a total of 1100 threat reports, of which 80 were manually annotated to construct a malware knowledge graph. It is also written in the paper by Rastogi et al. (2021) that all available CVE descriptions are part of the data set. However, the provided data set in the GitHub repository does not include these.

Some essential differences between the data set forming our KG and MalKG are worth considering. The data set used in MalKG consists of unstructured threat reports and manually annotated data from these. These threat reports typically span several pages of text describing the malware and its characteristics. On the other hand, CVE records from NVD consist of vulnerability descriptions that only span a few sentences and follow a stricter format. In addition, as introduced in Sec. 1.3, the CVE records in NVD consist of additional data such as vulnerability type, severity metrics, and affected product configurations.

MalKG extracts information from the mentioned threat reports in the form of triples used to construct a KG. The goal of Rastogi et al. (2021) is to use MalKG to predict unknown threat information. This is done by using knowledge graph embedding 2.7 to encode latent structures of the graph which could be used for entity prediction. The architecture of MalKG is shown in Fig. 4.3

Rastogi et al. (2021) demonstrates use cases of MalKG in two examples. In one of the examples, the malware family of the indicator *intel - update[.]com* is missing. MalKG is queried with the incomplete triple *(intel - update[.]com, indicates, ?)* and returns confidence scores for all possible

[2]https://github.com/malkg-researcher/MalKG

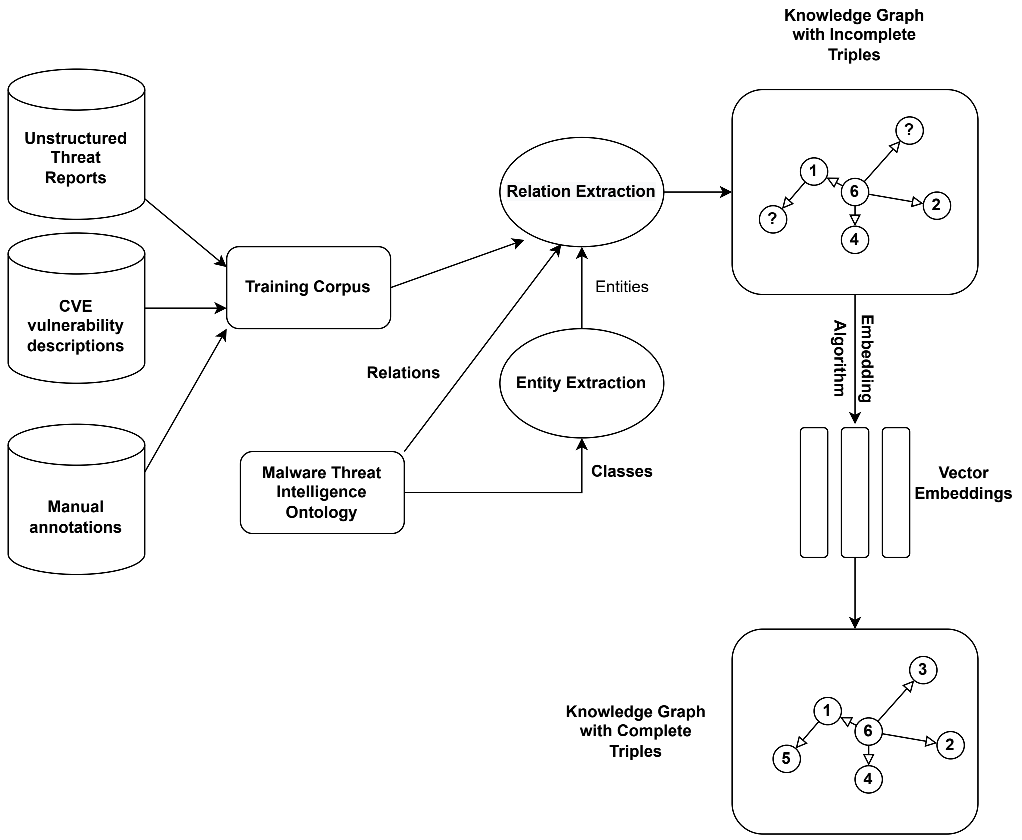

Knowledge Graph with Incomplete Triples
Unstructured Threat Reports
CVE vulnerability descriptions
Manual annotations
Training Corpus
Relation Extraction
Entities
Relations
Entity Extraction
Malware Threat Intelligence Ontology
Classes
Embedding Algorithm
Vector Embeddings
Knowledge Graph with Complete Triples
1
2
3
4
5
6
?

Figure 4.3: MalKG architecture. Figure based on Rastogi et al. (2021, Fig. 3).

entities using KG embedding. The entities are ranked in descending order according to their confidence score. The correct entity in this example was *Stealer* and ranked second among the returned entities in this example.

Two variations of the knowledge graph is proposed in MalKG, MT3K and MT40K. The 80 threat reports, which are manually annotated form the triples in MT3K. MT3K consists of a total of 3 000 triples. MT40K consists of 40 000 triples and is based on the 1100 unstructured threat reports. How the MT40K triples are extracted will be explained soon.

To extract information from the unstructured threat reports Rastogi et al. (2021) first applies NER. Two approaches to NER is used in the MalKG, the Flair framework (Akbik et al., 2019) and SetExpan (Shen et al., 2017).

The Flair framework proposed by Akbik et al. (2019) is an NLP framework providing an interface for model training and hyperparameter tuning. The framework allows for the mixing of word embedding techniques, see 2.3.3. The flair framework also provides pre-trained models. SetExpan is an algorithm that can be used in a range of NLP tasks, including NER. SetExpan takes a set of labeled seed values and an unlabeled corpus as input for training. The goal is then to recognize all entities of the same semantic class (Shen et al., 2017).

Using the Flair framework for NER, precision scores are reported to be between 0.9-0.99 for different classes. We assume that evaluation of the flair framework was performed on the annotated data to achieve the obtained scores. However, the code for this is not included in the repository. No code implementation of SetExpan exists in the repository.

After entities have been extracted, a relation extraction model can find the relations between the entities. The relation extraction model used by Rastogi et al. (2021) is taken from Yao et al. (8 09) and based on a BiLSTM architecture. MT3K is used for training the model. After the model has been trained, the extracted entities from Flair are used as input to the trained relation extraction model to form the triples of MT40K. MT3K is split into training and validation sets (90/10).

No performance metrics are reported from the relation extraction model in the paper by Rastogi et al. (2021). From the code by Rastogi et al. (2021) we found that accuracy is reported as a training metric. Validation metrics included F1 score, precision, and recall. We extended the code also to compute training performance as F1 score, precision, and recall Rastogi et al. (2021).

In the repository from (Rastogi et al., 2021) we find results from training and testing the model. The test F1 score has a max value of 0.1456. A distinction is made between the performance containing these inverse relations and where these relations have been removed. However, in MalKG these scores were almost identical.

The relation extraction model used in MalKG was originally proposed by Yao et al. (8 09). We compare the results from relation extraction in MalKG with the results presented in this paper. Yao et al. (8 09) reports the performance of different relation extraction models tested on data from Wikipedia. The reported scores in the paper proposed by Yao et al. (8 09) range between 0.32 and 0.51 in F1 score (Yao et al., 8 09, p. 7). Considering that these results are significantly higher than what we found indicates that

Model	Data set	Hits@1	Hist@3	Hits@10	MR	MRR
TransH	MT3K	26.8	50.5	65.2	32.34	0.414
TuckER	MT3K	64.3	70.5	81.21	7.6	0.697
	MT40K	73.9	75.9	80.4	202	0.75

Table 4.1: Results for MalKG entity prediction reported by (Rastogi et al., 2021).

relation extraction on threat reports has potential for further improvement.

From MT3K and MT40K new entities can be predicted using knowledge graph embedding. Rastogi et al. (2021) use two embedding algorithms, TuckER (Balažević et al., 2019) and TransH (Wang et al., 2014). TransH is a variation of TransE presented earlier in this thesis. Relations from one entity to another is learned after first projecting the entities on a hyperplane where different relations have distinct hyperplanes (Hogan et al., 2021).

The embedding algorithms are used on both MT3K and MT40K. Since MT40K does not have any ground truth, the models trained on MT40K are tested on MT3K.

The reported scores for entity prediction are calculated using different scoring metrics for both the annotated data set referred to as MT3K and the automatic data set, called MT40K. The reported performance metrics are shown in Table 4.1. The Hits@n metrics is multiplied by 100. In Table 4.1 we see that Rastogi et al. (2021) reports 80.4 in Hits@10 for MT40K. These results can be interpreted as follows, on average, in 80.4 out of 100 times, the true test triple is ranked among the ten triples with the highest confidence score.

The authors compare model performance on MT3K and MT40K to standard data sets such as FB15K (Bordes et al., 2013), FB15k-237 (Toutanova et al., 2015), WN18 (Bordes et al., 2013) and WN18RR (Dettmers et al., 2018). These data sets have been used to benchmark several entity prediction models, such as Bordes et al. (2013), Wang et al. (2014) and Balažević et al. (2019). The performance of the embedding algorithms in MalKG is compared with these because there are no earlier data sets available related to malware knowledge graphs available for comparison. The goal of the authors is that MT3K and MT40K could provide a baseline for future research (Rastogi et al., 2021).

We found some errors when comparing the performance metrics for the

		FB15K		WN18	
Model	Author	Hits@10	MRR	Hits@10	MRR
TransE	Balažević et al. (2019)	0.471	-	0.892	-
	Rastogi et al. (2021)	0.443	0.227	0.754	0.395
TuckER	Balažević et al. (2019)	0.892	0.795	0.958	0.953
	Rastogi et al. (2021)	0.513	0.260	0.806	0.576

Table 4.2: Reported metrics of KG embedding on standard datasets.

standard data sets presented in the paper by Rastogi et al. (2021) with what has been reported by Balažević et al. (2019). For example, as seen in Tab. 4.2 Hits@10 for TuckER on FB15K is reported 0.892 by Balažević et al. (2019) and the same experiment is reported with a Hits@10 of 0.513 by Balažević et al. (2019). Overall the reported scores of the standard data sets seem lower than what has been reported in earlier papers. As a consequence, the reported performance of MT3K and MT40K by Rastogi et al. (2021) appears closer to what is state of the art for the standard data sets.

Our replication results of MalKG includes relation extraction and entity prediction and are presented in 5.

Chapter 5

Results

The following chapter considers the results of our experiments with our vulnerability knowledge graph. The results include two major components: results from entity extraction and entity prediction. In addition, the results from the replication work of MalKG is presented.

5.1 Vulnerability knowledge graph

5.1.1 Named Entity recognition

When training the model, we experienced quadratic growth in the training time. We restricted the training data to 4000 CVE records out of approximately 150 000. The training for a sample size was around 6-8 hours for each model.

We used the remaining 146 000 CVE records to evaluate the performance. These CVE records were used for testing the models with performance metrics shown in Table 5.1.

We evaluate the domain model by first using the auto-labeled input data and then the predictions made by the IOB model. Using predicted IOB-labels as input in the domain model, performance decreases for all four metrics. The most significant decrease is in recall which drops by almost 0.1.

After performance evaluation, we used the trained model for labeling all 150 000 CVE records. The extracted labels were then used as input in the relation extraction model.

In Tab. 5.2 we see that 76 percent of the extracted words are "outside"

Model	F1-score	Precision	Recall	Accurracy
IOB-labeling	0.9304	0.9290	0.9318	0.9657
domain-labeling	0.9398	0.9407	0.9389	0.9818
domain-labeling*	0.8800	0.9245	0.8396	0.9479

Table 5.1: Performance metrics for the averaged perceptron model. domain-labeling* uses the predicted IOB-labels.

labels. These words are not relevant terms according to our model.

Comparing the rest of the labels, we see in Fig. 5.1 that the "relevant_term" label is by far the most frequent domain label occurring more than twice as often as the version, which is the second most frequent label.

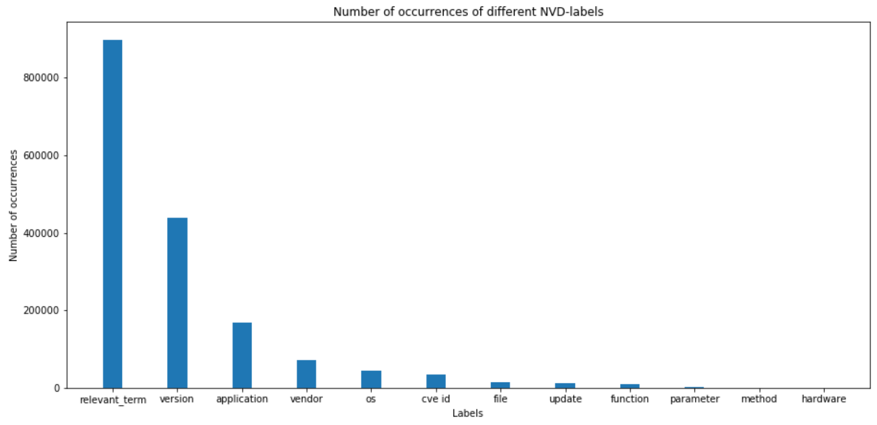


Figure 5.1: Plot domain labels distribution excluding outside tags.

5.1.2 Entity prediction

During the relation extraction, we extracted approximately 2 million triples. We then reversed all triples such that the input to TuckER was 4 million triples in total.

When evaluating TuckER, the input data are assumed to be true. The two best models are presented in Tab. 5.3 with corresponding hyperparameters

label	occurrences	ratio
O	5386747	0.7609
relevant_term	898359	0.1269
version	437991	0.0619
application	167779	0.0237
vendor	71030	0.0100
os	43879	0.0062
cve id	34992	0.0049
file	14843	0.0021
update	11953	0.0017
function	8850	0.0013
parameter	3195	0.0005
method	233	0.0000
hardware	30	0.0000
TOTAL	7079881	1.0000

Table 5.2: Statistics of domain labeling

Id	Hits@10	Hits@3	Hits@1	MRR
TUC-405	0.760	0.728	0.682	0.710
TUC-370	0.757	0.719	0.658	0.694

Table 5.3: Performance metrics for the two best models

Id	TUC-405	TUC-370
num_iterations	300	300
edim	200	200
rdim	30	30
lr	0.001	0.005
input_dropout	0.2	0.2
hidden_dropout1	0.1	0.1
hidden_dropout2	0	0.2
batch_size	128	128
label_smoothing	0.1	0.1
dr	1	0.995

Table 5.4: Hyperparameters of the two best models

Id	Duplicates	Hits@10	Hits@3	Hits@1	MRR
TUC-99	False	0.539	0.458	0.354	0.415
TUC-370	True	0.723	0.657	0.541	0.609

Table 5.5: Performance metrics with and without duplicate triples.

in Tab. 5.4. As seen in Tab. 5.3, TUC-405 performed better than TUC-370 on all performance metrics. TUC-405 was based on the parameters of the best candidate found in the grid search explained in Sec. 3.

Performance curves of the two models TUC-370 and TUC-405 are in the left figure of Fig. 5.2 and follow similar paths. TUC-370 falls a bit behind at approximately 35 epochs before catching up at around 200 epochs. The difference in the Hits@10 metric is only 0.003 but becomes more prominent for the Hits@3 and Hits@1 metric as seen in Tab. 5.3. From the right part of Fig. 5.2 convergence is reached earlier and at around 150 epochs for the broader Hits@10 metric compared to around 250 epochs for Hits@1.

Some triples are extracted multiple times in the relation extraction. For example, this can happen if the same relevant terms are found across multiple CVEs and these CVEs belong to the same CWE. We experienced that keeping these duplicate triples significantly increased performance compared to removing them. The Hits@10 metric of the second-best model TUC-370

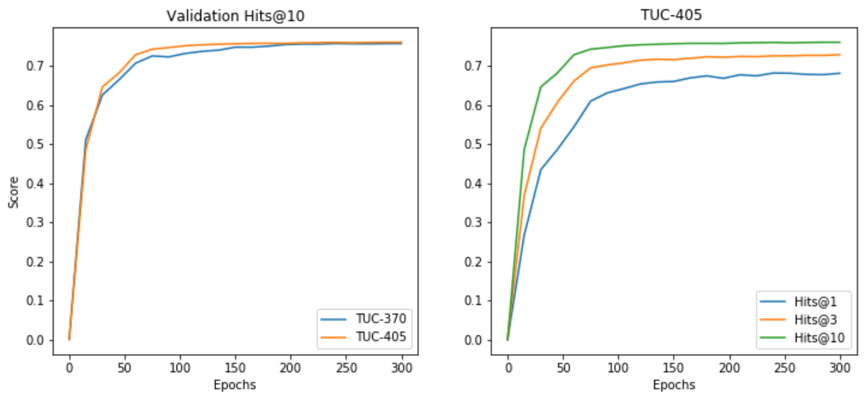


Figure 5.2: Comparison of two best performing configurations (TUC-370 and TUC-405) in the left figure. Different Hits@n metrics of the best model (TUC-405) in the right figure.

was 0.723 and included duplicates. TUC-99 uses the same parameters but removes all duplicate triples. The Hits@10 of TUC-99 is only 0.539 as seen Tab. 5.5.

Training on the complete data set with 300 epochs lasted approximately 36 hours. Evaluating the model was more computationally demanding than training the model. One training epoch took approximately four minutes, while the evaluation took approximately 25 minutes. As a consequence of this, we evaluated our model for every 15 epochs during training. We evaluated our model on both the test and the validation set. The evaluation performance on the two splits were almost identical as seen in Fig. 5.3.

5.2 MalKG

We start by replicating the DocRED relation extraction model of MalKG. When replicating, we use the hyperparameter combination reported in the paper by Rastogi et al. (2021). When replicating the MalKG project, we plot scoring metrics including precision, recall, and f1 score during training of the relation extraction model in Fig. 5.4.

After around 150 epochs, the F1 score stabilizes at around 0.2. The precision and recall score stabilizes at approximately the same value as the F1 score but with more fluctuating curves. Looking at the training performance

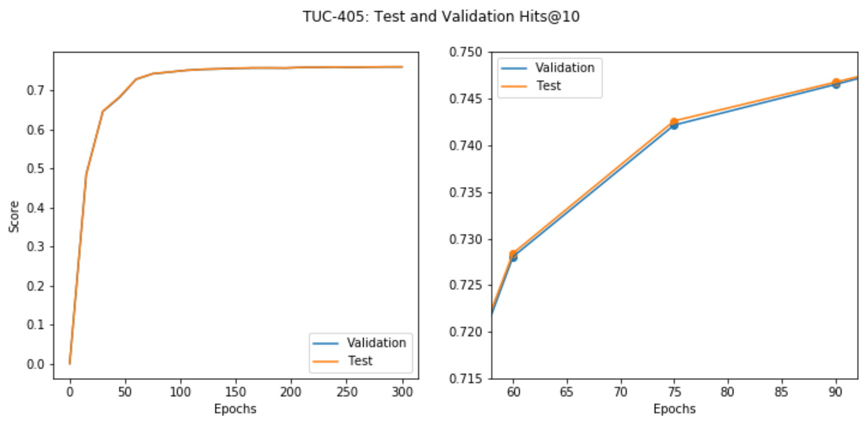


Figure 5.3: Test and validation metrics of the best performing (TUC-405) model. Test and validation metrics are almost identical.

Model	Data Set	Author	Test F1 Score
DocRED	MT3K	Ours	0.2291
		Rastogi et al. (2021)	0.1367

Table 5.6: Relation extraction results of our replication compared with the original results reported in the paper by Rastogi et al. (2021)

in Fig. 5.4, all training metrics quickly increase to a ”perfect” score which is an indication of overfitting.

Unfortunately, no official results have been reported in the paper by Rastogi et al. (2021). However, we found results from training and testing the model in the repository by Rastogi et al. (2021). We compare these results with our replication results in Table 5.6. We see that our test F1 score of the replication is higher than what we found in the repository by Rastogi et al. (2021).

In Table 5.7 the results from our replication of entity prediction using TuckER are shown. These results are compared with the reported results from Rastogi et al. (2021) in the same table. We see that the replicated results are much lower than the reported results.

The MT3K consists of around 3000 triples and the MT40K of around 40 000 triples, according to the paper by Rastogi et al. (2021). However, in the repository by Rastogi et al. (2021) we found just about 2000 triples for MT3K and around 15 000 triples for MT40K, indicating that some data are

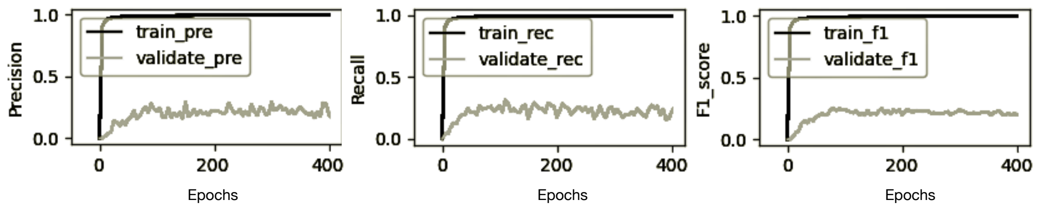


Figure 5.4: Scoring metrics from training of MalKG relation extraction. Training was run for a total of 984 epochs with best f1 score of 0.2291.

Model	Data Set	Author	Hits@1	Hits@3	Hits@10	MR	MRR
TuckER	MT3K	Ours	0.0315	0.0497	0.1275	916.0	0.0611
		Rastogi	0.6430	0.7050	0.8121	7.6	0.6970
	MT40K	Ours	0.2867	0.3172	0.3600	1832.1	0.3113
		Rastogi	0.7390	0.759	0.804	202	0.7500

Table 5.7: Entity prediction using TuckER. Our replication results are compared with the results reported in the paper by Rastogi et al. (2021).

missing.

Chapter 6

Discussion

6.1 MalKG

To increase our understanding of how we can construct a vulnerability knowledge graph, we replicate MalKG. NER and RE are important components of constructing a KG. Unfortunately, in the code by Rastogi et al. (2021), NER was only applied to new unseen data, and no code implementation was provided of how training or evaluating the NER model is done. Regarding the RE model, Rastogi et al. (2021) presents no official results in the paper, but as opposed to NER, performance evaluation was implemented in the code for RE. Rastogi et al. (2021) emphasize the entity prediction part of MalKG the most. In Ch. 5 we experienced that our replication results for entity prediction were far away the results presented by Rastogi et al. (2021). We also experienced that the amount of provided data in the GitHub repository for entity prediction was less than what was stated by Rastogi et al. (2021). It is not unlikely that the missing data could be the reason for the experienced deviations between our results and the results provided by Rastogi et al. (2021). For example, looking at MT40K, the original data set contained 40 000 triples achieving a Hits@10 metric of 0.804 using TuckER on the test data. On the other hand, our replication data set only contained 15 000 triples and achieved a score of 0.360 in Hits@10.

Insights from the paper by Rastogi et al. (2021) have still been helpful for the construction of our KG, even if our replication work were not able to reproduce the original results. Much of the terminology from vulnerability descriptions and threat reports are similar such as vulnerable products, versions, and weakness terms. We have chosen to use the same approach for entity prediction as was done in MalKG.

6.2 NER

We used the same model as Bridges et al. (2014) but our available data of CVE records were larger. The python implementation of averaged perceptron proposed by Bridges et al. (2014) is computationally demanding, therefore we trained our model using only a random sample of 4000 out of the 150 000 CVE records in the full data set. Our performance was a bit lower than what was reported by Bridges et al. (2014). For example, Bridges et al. (2014) reported a score of 0.984 in F1-score compared to 0.940 for our domain model, (second row in 5.1). Overall we believe the 4000 CVE records we used for training represent the full data set well since the descriptions follow a certain structure and contain similar entities. However, from Bridges et al. (2014) they experienced increased performance with more training data. Bridges et al. (2014) trained with 80 percent and tested with 20 percent of the data, which would also be a preferable combination to our model. We believe our performance could also increase with access to more training data.

To make the model training more efficient, Bridges et al. (2014) implemented the model in OpenNLP. A challenge was that Bridges et al. (2014) did not apply much implementation details of how this was done.

Gasmi et al. (2019) proposed different NER and RE models based on the LSTM architecture and applied this to the NVD data. Gasmi et al. (2019) reported a F1-score of 0.8337 applying NER for domain-labeling training on data from 2010 to 2019. Gasmi et al. (2019) used an architecture based on LSTM-CRF but their F1-score were approximately 0.1 less than ours. This was despite the fact that Gasmi et al. (2019) used a much larger data set for training. We thus confirm the statement made by Gasmi et al. (2019) emphasizing that *feature engineered* models outperforms more advanced models like LSTM-CRF when applied to a structured data set like NVD.

Earlier in the theory we introduced the concept of encoding words in numeric representations. Among the different types of word encoding were word embedding. Word embedding is a *feature learning* technique (Raschka and Mirjalili, 2019, Ch. 16) and is commonly used when training neural network architectures based on RNN (Raschka and Mirjalili, 2019, Ch. 16). In our work applying the averaged perceptron, word embeddings has not been used. The reason for this is how the feautres are given and how the model learns optimal weights for these features.

6.3 Relation extraction

Initially our plan was to use a BiLSTM model for RE inspired by Rastogi et al. (2021). The LSTM model has the ability to handle sequential data and extract long range relationships (Raschka and Mirjalili, 2019, Ch. 16) which could be beneficial when extracting relations between words.

Our replication work showed a performance score of this model of just above 0.2 in F1-score. On the contrary, performance on standard data sets has been reported to be twice as high (Yao et al., 8 09). In addition, we did not have any labeled data or a labeling tool for relations similar to the auto-labeling tool (Bridges et al., 2014) used for entities. As a consequence of this, we started from scratch extracting some of the relations we believe is most important in the CVE descriptions. By focusing on a few core relations our hope is that we would get high precision, this approach was inspired by how Bridges et al. (2014) constructed the auto-labeling tool.

As mentioned in Ch. 4, Gasmi et al. (2019) trained an LSTM-model of data of CVE records from 2015 of NVD. A semi-supervised approach for labeling the input data achieved a precision of 0.82. This data was then used as input to the LSTM-model and achieved an F1-score of 0.943 (Gasmi et al., 2019).

Jones et al. (2015) proposes a semi-supervised approach for labeling relations which achived a precision of 0.82. Gasmi et al. (2019) first applies the semi-supervised approach by Jones et al. (2015) for labeling relations. Gasmi et al. (2019) then trains a LSTM model on the labeled data achieving a F1-score of 0.943.

In our approach we do not have any labeled data of relations as ground truth. Are results can thus not be compared with the work of Gasmi et al. (2019) and Jones et al. (2015). For future work we plan to manually annotate a sample of relations in order to measure the precision of the RE model.

Furthermore, we are interested in extracting additional relations from the data. A single CPE vector contains multiple entities. These entities are all related and include vendor, product, version and which OS the vulnerable product are running on. For future improvements of our RE model, we could utilize the CPE vector similarly as they have done in the auto-labeling tool by Bridges et al. (2014). We could then combine the auto-labeling tool with our RE model labeling both relations and entities at the same time.

6.4 Entity prediction

TuckER has achieved state of the art performance of entity prediction on standard data sets (Balažević et al., 2019). In particular, an important characteristic of the model is its ability to capture 1-to-n, n-to-1 and n-to-n relations (Rastogi et al., 2021). The relation from vulnerable product to related terms is an example from our data set of a 1-to-n relationship.

The results from entity prediction using TuckER is shown in Table 5.3. The Hits@1 metric is 0.658 indicating that we are able to predict the correct entity among all other entities in 65.8 percent of the time. The results we achieve is below the results reported by Rastogi et al. (2021). We see for example that they achieve a Hits@10 metric of 0.804 compared to 0.760 in our case, however the gap between our results and the results by Rastogi et al. (2021) increases for Hits@3 and Hits@1. Interestingly, our performance is lower than Rastogi et al. (2021) despite training our model on a much larger data set consisting of approximately 2 million triples compared to 40 000 triples in the MT40K data set from MalKG. Some particular challenges with our data compared to Rastogi et al. (2021) is that it consists of all unique CVE IDs. We suspect that these CVE IDs could be particularly hard for the model to predict because of the level of granularity relative to the limited discriminatory information. Regarding vulnerable versions, we suspect that among 150 000 CVE records, many of the vulnerable versions of different product are overlapping. Predicting vulnerable versions of a product could thus also be challenging for our model.

6.5 An alternative approach

With our KG we can use entity prediction to find missing entities. This includes predicting missing weakness types. As an alternative approach document classification could be used to classify weakness types from vulnerability descriptions. One could for example apply tf-idf to the vulnerability descriptions, and then use the tf-idf scores as input to a machine learning model for classification [Ch. 8](Raschka and Mirjalili, 2019).

6.6 Future directions

Some future improvements have already been mentioned including ways of improving the RE model. We are also interested on improving our KG by extending the knowledge base. We could include data sources such as CWE descriptions and CVSS scoring metrics. In addition, for open source projects in the NVD, it could also be interesting to include commit messages, code metrics and source code of vulnerability fixing commits. CVEfixes (Bhandari et al., 2021) is an example of a relational database providing such multi-granular vulnerability information.

Chapter 7

Conclusion

This work proposes a vulnerability knowledge graph constructed from CVE records from the National Vulnerability Database. We explore the underlying concepts and future opportunities focusing our work particularly on NER, RE and entity prediction. For NER we use the current state of the art model for cyber security entities. We propose a RE model based on specific characteristics of the vulnerability descriptions which we believe could be a starting point for future work. The last part of our framework uses a knowledge graph embedding approach TuckER for entity prediction. We benchmark our performance with MalKG which was proposed by Rastogi et al. (2021) and uses the same approach, TuckER, for entity prediction on threat reports. We experienced that our vulnerability knowledge graph does not reach the same performance level as MalKG, however we believe our KG have potential for future improvements. In particular the RE component of our model could be extended to cover additional relations from the NVD .

Bibliography

2013. Retrieved 2022-02-15 from https://developer.nvidia.com/cuda-toolkit

2021. https://www.microsoft.com/security/blog/2021/12/11/guidance-for-preventing-detecting-and-hunting-for-cve-2021-44228-log4j-2-exploitat

2022. Retrieved 2022-02-15 from https://www.first.org/cvss/specification-document

2022. Retrieved 2022-02-15 from https://www.ex3.simula.no/resources

Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. 2019. FLAIR: An easy-to-use framework for state-of-the-art NLP. In *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. 54–59.

Ivana Balažević, Carl Allen, and Timothy M. Hospedales. 2019. TuckER: Tensor Factorization for Knowledge Graph Completion. In *Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 5184–5193. https://doi.org/10.18653/v1/D19-1522 arXiv:1901.09590

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *International Conference on Predictive Models and Data Analytics in Software Engineering* (Athens Greece). ACM, 30–39. https://doi.org/10.1145/3475960.3475985

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multirelational Data. In *Advances in Neural Information Processing Systems*, Vol. 26. Curran Associates, Inc. https://papers.nips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html

Bishal Bose. 2021. NLP — Text Encoding: A Beginner's Guide. https://medium.com/analytics-vidhya/nlp-text-encoding-a-beginners-guide-fa332d715854

Robert A. Bridges, Corinne L. Jones, Michael D. Iannacone, Kelly M. Testa, and John R. Goodall. 2014. Automatic Labeling for Entity Extraction in Cyber Security. (2014). arXiv:1308.4941 http://arxiv.org/abs/1308.4941

Solveig Bruvoll and Trond Arne Sørby. 2020. Lecture notes in Security in Operating Systems and Software.

François Chollet. 2018. *Deep learning with Python.* Manning Publications Co.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. 2018. Convolutional 2D Knowledge Graph Embeddings. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 11 (Apr 2018). https://ojs.aaai.org/index.php/AAAI/article/view/11573

Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05).* Association for Computational Linguistics, 363–370. https://doi.org/10.3115/1219840.1219885

Houssem Gasmi, Jannik Laval, and Abdelaziz Bouras. 2019. Information Extraction of Cybersecurity Concepts: An LSTM Approach. *Applied Sciences* 9, 19 (Sep 2019), 3945. https://doi.org/10.3390/app9193945

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. arXiv:2003.02320 `http://arxiv.org/abs/2003.02320`

Corinne L. Jones, Robert A. Bridges, Kelly Huffer, and John Goodall. 2015. Towards a relation extraction framework for cyber-security concepts. *Proceedings of the 10th Annual Cyber and Information Security Research Conference* (Apr 2015), 1–4. `https://doi.org/10.1145/2746266.2746277` arXiv: 1504.04317.

Arnav Joshi, Ravendar Lal, Tim Finin, and Anupam Joshi. 2013. Extracting Cybersecurity Related Linked Data from Text. In *2013 IEEE Seventh International Conference on Semantic Computing*. IEEE, 252–259. `https://doi.org/10.1109/ICSC.2013.50`

Mayank Kejriwal. 2019. *Domain-Specific Knowledge Graph Construction.* Springer. `https://doi.org/10.1007/978-3-030-12375-8`

US Information Technology Laboratory. 2021. National Vulnerability Database. Retrieved 2021-04-27 from `https://nvd.nist.gov/`

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]* (Sep 2013). `http://arxiv.org/abs/1301.3781` arXiv: 1301.3781.

MITRE. 2021a. Common Vulnerabilities and Exposures. Retrieved 2021-04-27 from `https://cve.mitre.org/`

MITRE. 2021b. Common Weakness Enumeration. Retrieved 2021-04-27 from `https://cwe.mitre.org/`

NMBU. 2022. NMBU Orion HPC Cluster — NMBU-SUPPORT. Retrieved 2022-02-15 from `https://support.nmbu.no/it-dokumentasjon/nmbu-orion-regneklynge/`

Aravindpai Pai. 2020. `https://www.analyticsvidhya.com/blog/2020/05/what-is-tokenization-nlp/`

Danny Palmer. 2021. Log4j flaw: Attackers are making thousands of attempts to exploit this severe vulnerability. https://www.zdnet.com/article/log4j-flaw-attackers-are-making-thousands-of-attempts-to-exploit-this-sever

Shengzhi Qin and K P Chow. 2019. Automatic Analysis and Reasoning Based on Vulnerability Knowledge Graph. (2019), 17. https://doi.org/10.1007/978-981-15-1922-2_1

Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. 2017. Introduction to Tensor Decompositions and their Applications in Machine Learning. *arXiv:1711.10781 [cs, stat]* (Nov 2017). http://arxiv.org/abs/1711.10781 arXiv: 1711.10781.

Sebastian Raschka and Vahid Mirjalili. 2019. *Python machine learning: machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. http://proquest.safaribooksonline.com/?fpi=9781789955750 OCLC: 1155055478.

Nidhi Rastogi, Sharmishtha Dutta, Ryan Christian, Mohammad Zaki, Alex Gittens, and Charu Aggarwal. 2021. Information Prediction using Knowledge Graphs for Contextual Malware Threat Intelligence. arXiv:2102.05571 http://arxiv.org/abs/2102.05571

M. Schuster and K.K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (Nov 1997), 2673–2681. https://doi.org/10.1109/78.650093

Jiaming Shen, Zeqiu Wu, Dongming Lei, Jingbo Shang, Xiang Ren, and Jiawei Han. 2017. SetExpan: Corpus-Based Set Expansion via Context Feature Selection and Rank Ensemble. In *Machine Learning and Knowledge Discovery in Databases (Lecture Notes in Computer Science)*, Michelangelo Ceci, Jaakko Hollmén, Ljupčo Todorovski, Celine Vens, and Sašo Džeroski (Eds.). Springer International Publishing, 288–304. https://doi.org/10.1007/978-3-319-71249-9_18

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.

Santiago Torres-Arias. 2021. What is Log4j? A cybersecurity expert explains the latest internet vulnerability, how bad it is and what's at stake. http://theconversation.com/what-is-log4j-a-cybersecurity-expert-explains-the-latest-internet-vulnerability-ho

Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. 2015. Representing Text for Joint Embedding of Text and Knowledge Bases. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 1499–1509. https://doi.org/10.18653/v1/D15-1174

Katrina Tsipenyuk, Brian Chess, and Gary McGraw. 2005. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. (2005). https://doi.org/10.1109/MSP.2005.159

Ledyard R. Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (Sep 1966), 279–311. https://doi.org/10.1007/BF02289464

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge Graph Embedding by Translating on Hyperplanes. *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence* 28 (2014). https://ojs.aaai.org/index.php/AAAI/article/view/8870

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. *arXiv:1412.6575 [cs]* (Aug 2015). http://arxiv.org/abs/1412.6575 arXiv: 1412.6575.

Yuan Yao, Deming Ye, Peng Li, Xu Han, Yankai Lin, Zhenghao Liu, Zhiyuan Liu, Lixin Huang, Jie Zhou, and Maosong Sun. 2019-08-09. DocRED: A Large-Scale Document-Level Relation Extraction Dataset. arXiv:1906.06127 http://arxiv.org/abs/1906.06127 version: 3.