Ossi Jormakka

APPROACHES AND CHALLENGES OF AUTOMATIC VULNERABILITY CLASSIFICATION USING NATURAL LANGUAGE PROCESSING AND MACHINE LEARNING TECHNIQUES

UNIVERSITY OF JYVÄSKYLÄ
FACULTY OF INFORMATION TECHNOLOGY
2019

ABSTRACT

Jormakka, Ossi
Approaches and Challenges of Automatic Vulnerability Classification using Natural Language Processing and Machine Learning Techniques
University of Jyväskylä, 2019, 58 + 4 pp.
Information Systems, Cyber Security, Master's Thesis
Supervisor: Costin, Andrei

Automated vulnerability detection and prediction of vulnerability details may help security specialists to prioritize bug reports and getting earlier fixes to security related software defects. This thesis is about finding vulnerable-like descriptions from any text and classifying vulnerability severities and weakness types. Vulnerability severities are measured using Common Vulnerability Scoring System. Common Weakness Enumeration is a hierarchical list of weakness types that each vulnerability can be classified to. The scoring and weakness type information for known vulnerabilities are available on National Vulnerability Database. Many existing research about vulnerability text-only classification is limited to a narrow area, for example, specific version of Common Vulnerability Scoring System. This thesis gives an overview of classifying bug reports with severities and weakness types altogether. The Scikit-learn library's interfaces were used extensively to implement text preprocessing, machine learning classification, and experiment validation. Experiments include stemming, lemmatization, and numerous text vectorization options and algorithms provided by the library.

The results show that the keyword-based classifier using word 2-grams works as well as One-class Support Vector Machine with lemmatizing using the Term Frequency–Inverse Document Frequency preprocessing method in vulnerability detection. Vulnerability severities can be predicted better for Common Vulnerability Scoring System version 2 than its version 3. The Linear Support Vector Machine classifier got the highest F1-score in predicting both Common Vulnerability Scoring System and Common Weakness Enumeration. This thesis also presents a summary on the latest data available on the National Vulnerability Database data feeds.

Keywords: Common Vulnerability Scoring System, Common Weakness Enumeration, Classification, Scikit-learn, CVE, CVSS, CWE, ML, Machine Learning, NLP, Natural Language Processing

TIIVISTELMÄ

Jormakka, Ossi
Automaattinen haavoittuvuusluokittelu luonnollisen tekstinkäsittelyn ja koneoppimisen menetelmillä
Jyväskylä: Jyväskylän yliopisto, 2019, 58 + 4 s.
Tietojenkäsittelytiede, Kyberturvallisuus, pro gradu -tutkielma
Ohjaaja: Costin, Andrei

Automatisoitu haavoittuvuuksien etsiminen ja haavoittuvuuksien yksityiskohtien ennustaminen voi auttaa asiantuntijoita priorisoimaan ohjelmistovirheitä, joka voi johtaa nopeampaan virheenkorjaukseen. Tässä työssä käytettiin National Vulnerability Database -tietokantaa tutkittaessa kuinka haavoittuvuuskuvauksien perusteella voidaan havaita haavoittuvuuksia mistä tahansa tekstistä sekä ennustaa haavoittuvuuksien vakavuus ja haavoittuvuustyyppi. Common Vulnerability Scoring System -järjestelmä tarjoaa tavan mitata haavoittuvuuksien vakavuuksia. Common Weakness Enumeration -järjestelmä tarjoaa hierarkkisen luokittelun yleisiin haavoittuvuustyyppeihin. Olemassa olevat tutkimukset haavoittuvuuksien tekstiluokittelussa usein rajoittuvat kapeaan alueeseen, esimerkiksi vain johonkin Common Vulnerability Scoring System -järjestelmän versioon. Tämä työ antaa yleiskuvan virheraporttien luokittelusta sekä vakavuuden ja haavoittuvuustyypin ennustamisesta. Työssä pyrittiin käyttämään laajasti tunnettuja tekstin esikäsittelymenetelmiä sekä monia muita Scikit-learn -kirjaston tarjoamia luonnollisen tekstin käsittelyn vaihtoehtoja ja koneoppimismenetelmiä.

Tulokset osoittavat 2-grammin avainsanapohjaisen menetelmän olevan yhtä tehokas kuin yhden luokan tukivektorikone kun esikäsittelynä käytetään Term Frequency – Inverse Document Frequency -painotusta ja sanojen taivutusmuotojen muuttamista perusmuotoon (lemmatizing). Haavoittuvuuksien vakavuuden ennustamisessa saadaan parempia tuloksia Common Vulnerability Scoring System -järjestelmän versiolle 2 kuin järjestelmän versiolle 3. Lineaarinen tukivekorikone saavutti korkeimman F1-tuloksen haavoittuvuuksien vakavuuden ja haavoittuvuustyypin luokittelussa. Lisäksi tässä työssä on yhteenveto uusimpaan National Vulnerability Database -tietokannan tietoon.

Asiasanat: tiedonlouhinta, luokittelu, koneoppiminen, luonnollisen kielen prosessointi, haavoittuvuus

FIGURES

TABLES

TABLE OF CONTENTS

1 AUTOMATIC VULNERABILITY CLASSIFICATION

In field of computer security a vulnerability is weakness that can be exploited by an attacker. Weakness can be any type of defect in a computer system that could lead information security to be compromised. A vulnerability which is unknown to the parties that are responsible of correcting them is called a zero-day vulnerability. Defect reports are often written to a system where the responsible parties can study, reproduce, prioritize, and monitor the status of defect corrections. These reports are short, a few sentences long descriptions about software defects. Some defect reports may expose information about potential vulnerabilities which should be taken into account in prioritization or public visibility. Some defect reporting systems are publicly available. According to Arnold et al. (2009) and Wijayasekara et al. (2012) findings it takes a longer time to incorporate and distribute non-security related software patches than those that are identified as vulnerabilities when they were reported. Wright (2013) et al. concluded that after examining the bug database for the MySQL database software a significant number of previously unknown vulnerabilities were identified.

The National Vulnerability Database (*NVD*) contains information about vulnerability descriptions, security checklists, security related software flaws, misconfigurations, product names, and impact metrics. The NVD database is maintained by the U.S. government and the data is freely available at their data feeds.[1] The data feeds are updated at least daily. The vulnerability descriptions are relatively short sentences about vulnerabilities. These sentences can be used to identify potential vulnerabilities in any other text, including defect reports in defect tracking systems. The security specialists evaluate vulnerability severities and root causes, and this information is also available among the NVD data. Vulnerability severities are expressed using Common Vulnerability Scoring System (*CVSS*) which is based on a several classifications. Vulnerability root causes are expressed using Common Weakness Enumeration (*CWE*) which is a hierarchical tree of hundreds of weakness types. Each vulnerability is identified

[1] https://nvd.nist.gov/vuln/data-feeds

using Common Vulnerabilities and Exposures (*CVE*) identifier number. CVE Numbering Authorities are organizations worldwide that are authorized to assign CVE identifier numbers. Common Platform Enumeration (*CPE*) is a structured naming scheme for systems, software, and packages. The CPE information is included with the other vulnerability information via the NVD data feeds.

In this thesis, the NVD data is used to detect vulnerabilities from text. A few defect databases were selected for study to find security related reports from all the reports. The NVD data is also used to learn and to predict selected machine learning algorithms to classify the CVSS and CWE classifications. The Scikit-learn library was selected to implement classification and to measure the machine learning algorithm performance. Scikit-learn is a free software machine learning library for the Python programming language. The Jupyter Notebook tool was selected to implement all the experiments. The tool is an open-source web application that allows to create and share documents that contain live code, narrative text, and visualizations. The Anaconda platform was selected to manage all the required software packages for this study. Anaconda is a free distribution of the Python and R programming languages for scientific computing that simplifies package management and deployment.

This work is a preliminary study towards a tool which detects and ranks vulnerabilities and also estimates root causes of vulnerabilities using a short human written text information only. The most challenging task to achieve this is to convert human written text suitable for machine learning algorithms. These preprocessing tasks transform the text to features that machine learning algorithms can process. To handle this, a variety of Scikit-learn vectorizers were compared. Another challenge is to detect vulnerabilities from text, knowing vulnerability descriptions only. This is called a one-class or unary classification problem. Traditional statistical or machine learning classification concerns binary classification which requires both positive and negative samples in learning data. It is claimed that one-class classification is successfully applied in numerous realms of academic research and industrial applications (Wang et al., 2017). The results indicate that Scikit-learn's OneClassSVM classifier is capable detecting vulnerabilities as good as the keyword based classifier which was implemented during this work. The problem still remains with relatively high number of false positives. The CVSS classification is mostly a multiclass classification problem but some of the metrics may have two classes only. The results show that CVSS classification is successful with linear classification algorithms having F1-score around 0,816. The challenge is that there are several metrics to classify to gain the CVSS score. Automatic CWE classification is also viable using the same methods as CVSS classification, but the challenge is that there are dozens of separate classes and having low number of samples in many class. An approach was taken in this study to find root categories of those CWE classes. The results in this thesis shares similarities with Han et al. (2017) study estimating CVSS2 scores using word embeddings and 1-layer convolutional neural network.

2 OVERVIEW OF CVSS, CWE, AND CPE

CVSS consists of metrics and classifications, CWE forms a hierarchy of many weakness types, and CPE is a structured scheme which are explained more detailed in this chapter.

2.1 Common Vulnerability Scoring System

Common Vulnerability Scoring System (*CVSS*) is an open industry standard for assessing the severity of security vulnerabilities. The system is maintained by Forum of Incident Response and Security Teams organization. The system maps severity scores to vulnerabilities which allows security specialists to prioritize responses and resources according to threat. Scores are calculated using a formula on several metrics that approximates ease of exploit and the impact of exploit. The scores range is from 0 to 10, with 10 being the most severe. CVSS is composed of three metric groups: Base, Temporal, and Environmental. Each metric group consists of a set of metrics. The set of metrics are expressed in a form of vector. Each metric is abbreviated and separated by a colon character as shown in Figure 1 example CVSS vectors. The base score represents the innate characteristics of a vulnerability. The temporal score represents metrics that change over time due to events which are external to a vulnerability. The environmental score modifies the impact depending on the environment a vulnerability is exposed. Vulnerability databases typically provides the base scores but no temporal or environmental scores. In this thesis the base score is used only.

Figure 1. Examples of CVSS vectors (left: CVSS3, right: CVSS2)

2.1.1 CVSS2 Base Metrics

The specification of CVSS version 2 was published in June 2007. The base metrics consist of six separate metrics. Each metric has predefined classifications, used in a formula to calculate the actual severity score. The CVSS2 metrics and classifications are shown in Table 1. The Access Vector, Access Complexity, and Authentication metrics assess how the vulnerability is accessed and if some extra conditions are required to exploit it. The three impact metrics measures how a vulnerability affects an IT asset. The impacts are independent with each other and describes the loss of confidentiality, integrity, and availability. (CVSS2, 2007).

Table 1. CVSS2 Metrics and Classifications

Metric	Description	Classifications
Access Vector (AV)	"Reflects how the vulnerability is exploited."	Network (N), Adjacent (A), Local (L)
Access Complexity (AC)	"Measures the complexity of the attack required to exploit."	High (H), Medium (M), Low (L)
Authentication (Au)	"Measures the number of times an attacker must authenticate to a target in order to exploit a vulnerability."	None (N), Single (S), Multiple (M)
Confidentiality Impact (C)	"Measures the impact on confidentiality of a successfully exploited vulnerability."	Complete (C), Partial (P), None (N)
Integrity Impact (I)	"Measures the impact to integrity of a successfully exploited vulnerability."	Complete (C), Partial (P), None (N)
Availability Impact (A)	"Measures the impact to availability of a successfully exploited vulnerability."	Complete (C), Partial (P), None (N)

The CVSS2 score is calculated based on the six metrics with classifications as illustrated in Table 1. The CVSS2 base score consists of exploitability and impact metrics and is calculated as follows:

$$\text{"}Impact_{conf} = case\ ConfidentialityImpact\ of\ N:\ 0.0,\ P:\ 0.275,\ C:\ 0.660$$

$$Impact_{Integ} = case\ IntegrityImpact\ of\ N:\ 0.0,\ P:\ 0.275,\ C:\ 0.660$$

$$Impact_{Avail} = case\ AvailabilityImpact\ of\ N:\ 0.0,\ P:\ 0.275,\ C:\ 0.660$$

$$Impact = 10.41 \times (1 - (1 - Impact_{conf}) \times (1 - Impact_{Integ}) \times (1 - Impact_{Avail}))$$

$$f(impact) = 0\ if\ Impact = 0,\ 1.176\ otherwise$$

$$AV = case\ AccessVector\ of\ L:\ 0.395,\ A:\ 0.646,\ N:\ 1.0$$

$$AC = case\ AccessComplexity\ of\ H:\ 0.35,\ M:\ 0.61,\ L:\ 0.71$$

$$Au = case\ Authentication\ of\ M:\ 0.45,\ S\ 0.56,\ N:\ 0.704$$

$$Expl = 20 \times AV \times AC \times Au$$

$$BaseScore = round\_to\_1\_decimal(((0.6 \times Impact) + (0.4 \times Expl) - 1.5) \times f(Impact))\text{"}$$

2.1.2 CVSS3 Base Metrics

The specification of CVSS version 3 was published in June 2015. The major difference to the previous version is that there are eight separate metrics to classify to calculate the actual score. The CVSS version 3 base metrics and classifications are shown in Table 2. (CVSS3, 2015).

Table 2. CVSS3 Metrics and Classifications

Metric	Description	Classifications	Numeric
Attack Vector (AV)	"Reflects the context by which vulnerability exploitation is possible."	Network (N), Adjacent (A), Local (L), Physical (P)	0,85, 0,62, 0,55, 0,2
Attack Complexity (AC)	"Measures the complexity of the attack required to exploit."	High (H), Low (L)	0,44, 0,77
Privileges Required (PR)	"Describes the level of privileges an attacker must possess before successfully exploiting the vulnerability."	High (H), Low (L), None (N)	0,27 / 0,5, 0,62 / 0,68, 0,85
User Interaction (UI)	"Captures the requirement for a user, other than the attacker, to participate in the successful compromise of the vulnerable component."	Required (R), None (N)	0,62, 0,85
Scope (S)	"Scope refers to the collection of privileges defined by a computing authority. These privileges are assigned based on some method of identification and authorization."	Changed (C), Unchanged (U)	Modifies Privileges Required if Scope is Changed
Confidentiality Impact (C)	"Measures the impact to confidentiality of a successfully exploited vulnerability."	High (H), Low (L), None (N)	0,56, 0,22, 0
Integrity Impact (I)	"Measures the impact to integrity of a successfully exploited vulnerability. "	High (H), Low (L), None (N)	0,56, 0,22, 0
Availability Impact (A)	"Measures the impact to availability of a successfully exploited vulnerability."	High (H), Low (L), None (N)	0,56, 0,22, 0

The numeric values in Table 2 are used in equations to calculate the actual score. The base score is a function of the Impact and Exploitability sub score equations.

The score metric modifies classifications' numeric values and also the equations concerning Impact and BaseScore calculations. The CVSS3 base score is calculated as follows:

$$\text{“}Impact_{Base} = 1 - [(1 - Impact_{Conf}) \times (1 - Impact_{Integ}) \times (1 - Impact_{Avail})]$$

$$Exploitability = 8.22 \times AV \times AC \times PR \times UI$$

$$If(Scope = Unchanged):$$

$$Impact = 6.42 \times Impact_{Base}$$

$$If(Scope = Changed):$$

$$Impact = 7.52 \times [Impact_{Base} - 0.029] - 3.25 \times [Impact_{Base} - 0.02]^{15}$$

$$BaseScore = If(Impact \leq 0), 0\ else:$$

$$If(Scope = Unchanged):$$

$$BaseScore = Roundup(Minimum[(Impact + Exploitability), 10])$$

$$If(Scope = Changed):$$

$$BaseScore = Roundup(Minimum[1.08 \times (Impact + Exploitability), 10])\text{”}$$

In this thesis the cvsslib's calculate_vector method was used to calculate the actual CVSS2 and CVSS3 scores.

2.2 Common Weakness Enumeration

Common Weakness Enumeration is a hierarchical list of software weakness types. The CWE is maintained by nonprofit MITRE organization. The latest version 3.2 was published in January 2019. A new version is published approximately annually. The hierarchical lists are divided based on the concept views: Research, Development, and Architectural views. The research concept view describes weaknesses and dependencies with each other to identify theoretical gaps within CWEs. The development concept view organizes weaknesses related to software development. The architectural concept view organizes weaknesses according to common architectural security tactics. Its goal is to identify potential mistakes that can be made in a software development process. In this thesis the Research Concept view was selected as a basis to resolve CWE parent items. An example of the research hierarchy is shown in Figure 2.

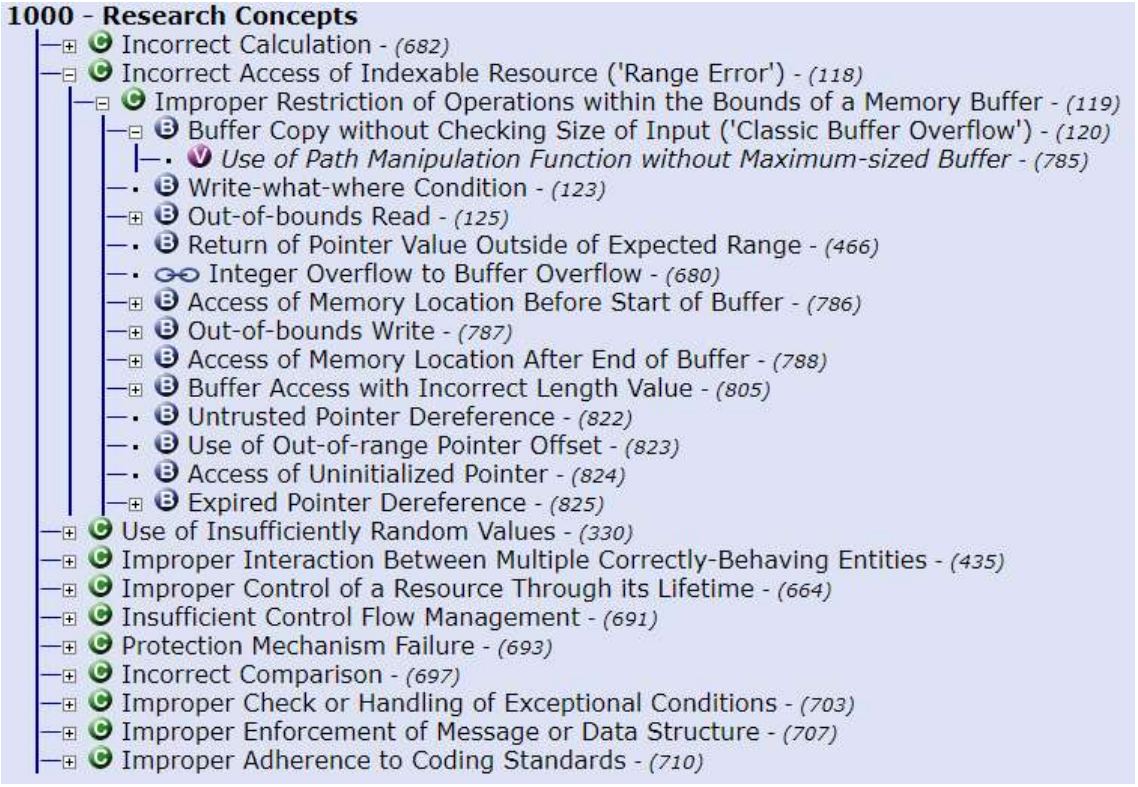

Figure 2. CWE hierarchy by Research Concept View[2]

All the research concept view's root level classes are shown in Figure 2. The tree-like relationships between weaknesses that exist at different levels of abstraction are shown. At the highest level, classes exist to group weaknesses. The classes are weaknesses that are described at more abstract level than the base weaknesses. A variant weakness is described at a very low level of detail, typically limited to a specific language or technology. Also the category type of CWE exists but not in the research concept view. In the view there are 806 CWE entries out of the total 1131 entries.

2.3 Common Platform Enumeration

The CPE is a structured naming scheme for systems, software, and packages. The CPE includes a formal name format, a method for checking names, and a description format for binding text to a name. The CPE Dictionary is hosted and maintained by NIST organization which is part of the Security Content Automation Protocol (*SCAP*). The latest version 2.3 was published in 2013. The CPE is considered to be an industry standard, originally defined by MITRE organization. The CPE formatted string binding is shown in Figure 3. All eleven attribute values must appear in the formatted string binding.

[2] https://cwe.mitre.org/data/definitions/1000.html

```
cpe:2.3:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>:<sw
_edition>:<target_sw>:<target_hw>:<other>
```

Figure 3. CPE Formatted String Binding

The format specifies in which platform vulnerabilities have been found. More detailed explanation about the format is the following:

- **part**: The system type, one of the following:
  a = Application
  h = Hardware
  o = Operating System
- **vendor**: Organization name who developed the product.
- **product**: Product name specified by vendor.
- **version**: Version identifier of the product.
- **update**: Update name of a version specified by vendor, for example, "R2" for Windows 2012.
- **edition**: Vendor specified software edition, for example, "server" or "x86".
- **language**: Language of a software, for example, Finnish.
- **sw_edition**: Software Edition defined by vendor to tailor a particular market or class of end users.
- **target_sw**: Software computing environment where the product operates.
- **target_hw**: Instruction set architecture the product is being identified, for example, "x86".
- **other**: Any other descriptive information which does not fit in any other attribute.

The complete CPE dictionary is freely available at the NVD website which is updated at least daily[3]. The CPEs which are mapped to vulnerabilities are listed among the NVD data feeds. The CPE formatted string binding within the NVD data feeds were used in this thesis.

[3] https://nvd.nist.gov/products/cpe

3 OVERVIEW OF MACHINE LEARNING TECHNIQUES

Text classification or text categorization is the task of assigning one or more predefined classes on unstructured text documents according to their content. Text classification is used for multiple purposes in many different fields: for example, news stories can be organized by subject topics, academic papers can be classified by technical domains, and patient reports in health-care can be indexed in multiple categories. A spam filter can classify an e-mail message as spam or non-spam. Text classification can be manual, simple rule or word based, or it can use machine learning methods to categorize text documents. Based on previous research, text classification performs better using machine learning approaches than a word based approach (Yang, 2000). Structured data refers to information with a high degree of organization which is more easily computable and handled by a computer. Unstructured data is information that does not have a pre-defined data model or organisation of data is not pre-defined. Semi-structured data is a form of structured data that does not conform to the formal structure of data models or forms but contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. (Kantardzic, 2011).

Machine learning methods in automatic text classification shares similarities with the fields of pattern recognition, statistics and data mining. Pattern recognition is a branch of machine learning that focuses on the recognition of patterns and regularities in data. Statistics is a branch of mathematics dealing with the collection, analysis, interpretation, presentation and organization of data. Data mining is the computing process of discovering patterns in large data sets. The goal is to discover a novel information from data. Machine learning can be roughly divided into two wide categories: Supervised or unsupervised learning depending on the learning signal. Supervised learning is the task learning from labelled training data. Unsupervised learning is the task to describe hidden structure in unlabelled training data. Another field of data mining is anomaly detection. The idea is that anomalies compared to a normal baselined data can be somehow detected. Anomaly detection can use supervised or unsupervised machine learning techniques. The Scikit-learn user guide divides anomaly detection into outlier and novelty detection. In outlier detection, the training data contains outliers which are far from the others, and estimators ignore the deviant observations. In novelty detection, the training data is not polluted by outliers (Scikit, 2019).

In terminology of machine learning, classification is usually considered as an instance of supervised learning. The corresponding unsupervised procedure is known as clustering. In classification, the categories are known beforehand and given in advance for each training document. In clustering, groups of samples that naturally belong together are sought. (Witten & Frank, 2005). Binary or binomial classification is the task of classifying the elements of a given set into

two groups. Multiclass or multinomial classification is the problem of classifying instances into three or more classes. In multi-label classification, multiple labels are predicted for each sample. In one-class classification or unary classification objects of a specific class are tried to be identified from all objects by learning from a training set containing only the objects of that class. A unary classifier performance remains relatively stable when a dataset class imbalance increases whereas a binary classifier performance decreases (Bellinger et al., 2012). A balanced dataset is the one that contains equal or almost equal number of samples of each class.

In data mining, dataset is a matrix which consists of rows of samples and columns of features. In text classification, the number of features is typically hundreds or thousands. This type of dataset is called a high dimensional dataset (Kantardzic, 2011). High number of features often leads to sparsity which means having a value of zero for the most instances. Text classification methods must be able to handle the sparsity of data. Sparsity can be taken into account at the feature extraction phase by implementing a dimension reduction technique. A general machine learning classifier model is shown in Figure 4. It represents how unstructured text is transformed into features, first in the training phase and then in the prediction phase.

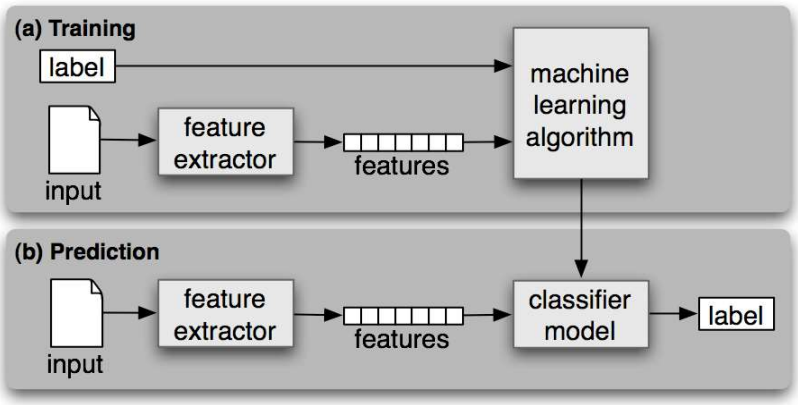

Figure 4. Machine Learning Classifier Model

In training, the features are provided to a machine learning algorithm with labels of the correct classifying results. Machine learning algorithm builds a model internally which is able to distinguish labels seen in the training data. In prediction, based on the trained model a classifier is able to predict labels in a new unseen data provided. A classifier can predict only such labels seen in the training data. The model in Figure 4 illustrates a supervised learning paradigm. In case of one-class classification, the data provided at learning phase contains samples only from one class. A one-class classifier can predict how far a data point is compared to the training data. In this sense the approach can be said as semi-supervised novelty detection (Scikit, 2019). In this thesis traditional machine learning techniques were used only but newer neural network based techniques exists to solve natural language classification problems as well.

3.1 Machine Learning Algorithms

There are many machine learning algorithms available depending on the problem type the algorithm needs to solve. Kantardzic (2011) divides the problem types into six primary data mining tasks: Classification, Regression, Clustering, Summarization, Dependency Modelling, and Change and Deviation Detection. As stated earlier, classification tries to classify items to predefined classes as clustering seeks to identify a finite set of categories. Regression tries to map data items to real-value prediction variables. Summarization is a task that involves methods for finding a compact description for a dataset. Dependency modelling is for finding a local model that describes significant dependencies between data variables. Change and Deviation Detection is used to discover the most significant changes in a dataset. The Scikit-learn user guide (2019) provides a cheat-sheet to select preferable algorithm for a machine learning problem as presented in Figure 5.

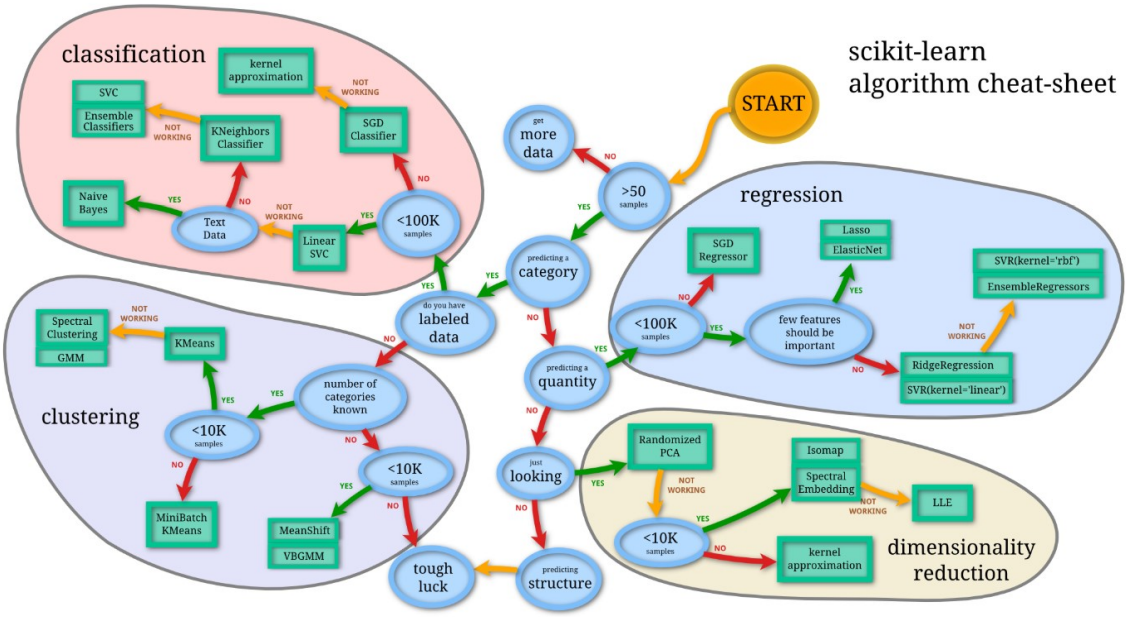

Figure 5. Scikit-learn Algorithm Cheat-Sheet

The cheat-sheet instructs to select an approach solving a machine learning problem. The approaches are not implicit but rather guidelines to lead to the right direction. In this thesis, Naïve Bayes, Stochastic Gradient Descent, Linear Support Vector Machine, and K-nearest Neighbors classification approaches were studied. The Scikit-learn library also provides approaches to for anomaly detection problems. The following approaches were studied: One-class Support Vector Machine, IsolationForest, and LocalOutlierFactor. In this chapter these algorithms are briefly explained from the view of using the programming interfaces, not implementing the actual algorithms.

3.2 One-class Support Vector Machine

Schölkopf et al. (2001) introduced One-class Support Vector Machine (*OCSVM*) to solve a novelty detection problem. In novelty detection, at prediction phase the classifier tries to determine whether a data point can be distinguished from the original data points seen at training phase. OCSVM is an extension of the standard binary Support Vector Machine (*SVM*). Wang et al. (2017) pointed out the importance of selecting the classifier hyperparameters which has a significant influence on its performance. In machine learning, a hyperparameter is a parameter whose value is set before the learning process begins. According to Wang et al. (2017) findings the most important hyperparameters which need to be properly tuned are the following: "The regularization coefficient ν and the Gaussian kernel width σ. The ν controls the upper bound of rejected target data, which is often tuned to reject noise during training, while σ controls the smoothness of decision boundary. An overly large σ or small σ will cause underfitting and overfitting respectively. Improper ν will make the decision boundary distorted by noisy target data or reject excessive target data." In machine learning, underfitting is the classifier decision boundary that is too simple to explain the variance in data. Overfitting is a too complex decision boundary. The both circumstance leads to bad classification performance.

The Scikit-learn library provides OCSVM implementation in the svm module's OneClassSVM class. The implementation is based on the libsvm implementation. The libsvm is an open-source machine learning library written in C++. The most important OneClassSVM's hyperparameters are introduced in Table 3, full hyperparameter list can be found from the user guide[4].

Table 3. OneClassSVM hyperparameters

Hyperparameter	**Description**	**Possible values**
kernel	"Specifies the kernel type to be used in the algorithm. If a callable is given it is used to precompute the kernel matrix."	'linear', 'rbf', 'poly', 'sigmoid', 'precomputed', callable
gamma	"Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma='scale' it uses 1 / (n_features * X.var()) as value of gamma."	float
tol	"Tolerance for stopping criterion."	float
nu	"The fraction of training errors and support vectors. Should be in the interval [0, 1]. By default 0.5 will be taken."	float

The kernel parameter adjusts the function how a support vector hyperplane is calculated. The default value is 'rbf' which stands for radial basis function. In this thesis, different parameter values were tested during hyperparameter tuning.

[4] https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html

3.3 Isolation Forest

IsolationForest is an unsupervised ensemble learning method performing outlier detection in high-dimensional datasets. The method isolates samples by randomly selecting features and split values between the maximum and minimum of the selected features. It can be represented by a tree structure. (Scikit, 2019). The method detects anomalies based on the concept of isolation without employing a distance or density measure. The method returns the anomaly score of each sample. The Scikit-learn library provides many hyperparameters to affect the functionality. Some of the parameters are explained in Table 4, full hyperparameter list can be found from the user guide.[5]

Table 4. IsolationForest hyperparameters

Hyperparameter	Description	Possible values
contamination	"The proportion of outliers in the dataset. Used at learning to define the threshold on the decision function."	float in (0., 0.5), optional (default=0.1)
n_estimators	"The number of base estimators in the ensemble."	int, optional (default=100)
bootstrap	"Whether or not the individual trees are fit on random subsets of the training data sampled with replacement."	boolean, optional (default=False)

According to Liu et al. (2012) Isolation Forest outperforms OneClassSVM and Local Outlier Factor methods detecting global anomalies in performance having a low linear time-complexity and a small memory-requirement. Isolation forest fails to detect local anomalies. They also found that using smaller subsamples builds better isolation models.

3.4 Local Outlier Factor

Local Outlier Factor (*LOF*) is an unsupervised algorithm proposed by Breunig et al. (2000) for finding anomalous data points by measuring the local deviation of a given data point with respect to its neighbors. The algorithm can detect local outliers efficiently from large datasets. An anomaly score of each sample is called Local Outlier Factor. The score tells how isolated the object is compared to surrounding neighbors. Locality is given by nearest neighbors and the distance is used to estimate the local density. The Scikit-learn library provides numerous

[5] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

hyperparameters for LOF. Some hyperparameters are introduced in Table 5, full list of hyperparameters can be found from the user guide.[6]

Table 5. LocalOutlierFactor hyperparameters

Hyperparameter	Description	Possible values
n_neighbors	"Number of neighbors to use by default for k-neighbors queries. If n_neighbors is larger than the number of samples provided, all samples will be used."	int, optional (default=20)
algorithm	"Algorithm used to compute the nearest neighbors: 'ball_tree' will use BallTree 'kd_tree' will use KDTree. 'brute' will use a brute-force search. 'auto' will attempt to decide the most appropriate algorithm based on the learning values."	{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional
contamination	"The proportion of outliers in the dataset. Used at learning to define the threshold on the decision function."	float in (0., 0.5), optional (default=0.1)
novelty	"By default, LocalOutlierFactor is only meant to be used for outlier detection. Set novelty to True if you want to use LocalOutlierFactor for novelty detection."	boolean, default False

3.5 Naïve Bayes

According to the Scikit-learn user guide Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem. A Naive Bayes classifier assumes that the presence of a particular feature is unrelated to the presence of any other feature. Naive Bayes classifiers have worked well in many real-world situations such as document classification and spam filtering. It is alleged that Naïve Bayes requires a small amount of training data to estimate the necessary parameters. Naive Bayes learners and classifiers can be fast compared to more sophisticated methods. Naive Bayes is also known as a bad estimator, so the probability results are not trustworthy. The Scikit-learn library provides many variants of Naïve Bayes implementations. The Multinomial Naïve Bayes (*MultinomialNB*) classifier was selected in this thesis as the classifier has been used in text classification. (Scikit, 2019).

[6] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html

3.6 Stochastic Gradient Descent

Stochastic Gradient Descent (*SGD*) is supervised discriminative learning of linear classifiers under convex loss functions. The Scikit-learn user guide tells that SGD has been applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. The classifier can scale to very large matrices with more than 10^5 training samples and more than 10^5 features. The Scikit-learn's SGDClassifier supports multiclass classification by combining multiple binary classifiers in a one versus all (*OVA*) scheme. (Scikit, 2019). The actual classifier can be selected as a hyperparameter. In Table 6 there are some hyperparameters introduced, full listing can be found from the user guide.[7]

Table 6. SGDClassifier hyperparameters

Hyperparameter	Description	Possible values
loss	"The loss function to be used. 'hinge' gives a linear SVM. The 'log' loss gives logistic regression, a probabilistic classifier. 'squared_hinge' is like hinge but is quadratically penalized. 'perceptron' is the linear loss used by the perceptron algorithm. The other losses are designed for regression instead of classification."	str, default: 'hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron'
alpha	"Constant that multiplies the regularization term. Also used to compute learning rate when set to 'optimal'."	float, default=1e-4
tol	"The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for n_iter_no_change consecutive epochs."	float or None, optional (default=1e-3)
shuffle	"Whether or not the training data should be shuffled after each epoch (iteration)."	bool, optional. Default: True

3.7 Linear Support Vector Machine

Support Vector Machines (*SVM*) are a set of supervised learning methods used for classification, regression and outlier detection. The Scikit-learn user guide (2019) says that SVM is effective in high dimensional spaces even in the case of very high number of features compared to number of samples, but the classifier may suffer overfitting. SVM is also memory efficient. SVMs supports specifying different kernel functions for the decision function. Overfitting can be avoided by choosing a kernel and a regularization term. SVM do not directly provide

[7] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

probability estimates. In this thesis the linear kernel was selected for further study. The Scikit-learn's LinearSVC provides the linear SVM. The implementation is based on the liblinear library which is more efficient than the libsvm library based implementation. The LinearSVC supports multiclass classification according to OVA scheme. In Table 7 some important LinearSVC hyperparameters are introduced, full list of hyperparameters can be found from the user guide.[8]

Table 7. LinearSVC hyperparameters

Hyperparameter	Description	Possible values
penalty	"Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to coefficient vectors that are sparse."	string, 'l1' or 'l2' (default='l2')
loss	"Specifies the loss function. 'hinge' is the standard SVM loss while 'squared_hinge' is the square of the hinge loss."	string, 'hinge' or 'squared_hinge' (default='squared_hinge')
dual	"Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features."	bool, (default=True)
tol	"Tolerance for stopping criteria."	float, optional (default=1e-4)
C	"Penalty parameter C of the error term."	float, optional (default=1.0)

The implementation uses a random number generator to select features when fitting the model. It might lead to different results for the same input data. If that happens a smaller value of the tol parameter must be attempted.

3.8 K-Nearest Neighbors

The Scikit-learn user guide (2019) calls Nearest Neighbors classification as instance based learning or non-generalizing learning method. The method does not attempt to generate a general internal model but stores instances of the training data instead. Classification is computed as a simple majority vote of the nearest neighbors of each point. A query point is then assigned the data class that has the most representatives within the nearest neighbors of the point. The algorithm is simple but the method has been successfully applied in number of classification and regression problems such as handwritten digits or satellite image scenes. Nearest Neighbors methods may be useful in classification situations where the decision boundary is very irregular. (Scikit, 2019). The Scikit-learn library provides the KNeighborsClassifier object to implement this type of classifier.

[8] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

3.9 Scikit-learn Programming Interfaces

In the Scikit-learn library, an estimator for classification is a Python class that implements the fit and the predict methods. Fitting an estimator means that the learning data is provided as parameter. The testing data is provided as parameter to be able to predict the classes to which unseen samples belong. As a coding convention, the Scikit-learn estimators follow certain guidelines to make its behavior more predictive and developer friendly.

The library provides Pipeline class to apply sequentially a list of transforms and a final estimator. The intermediate steps must implement the fit and the transform methods. Only a final estimator needs to implement the predict method. The purpose of a pipeline is to ease the processing steps that can be cross-validated.

To implement a new Scikit-learn compatible estimator, the class needs to inherit Scikit-learn's BaseEstimator and also one of the estimator type classes. The type classes are ClassifierMixin, RegressorMixin, ClusterMixin, and TransformerMixin. In case of an estimator for classification the fit, predict, and score methods need to be overwritten. Further information about implementing estimators can be found from the Scikit-learn's developer's guide.[9]

GridSearchCV is a useful class for tuning an estimator hyperparameters or options in a pipeline. The class does exhaustive search over specified parameter values for an estimator. GridSearchCV uses the fit and score methods. It also may use the methods predict, predict_proba, decision_function, transform, and inverse_transform if they are implemented in the estimator used. It does cross-validated search over a parameter grid. Another option is to use the ParameterGrid class directly passing each grid element as a parameter of the estimator's set_params method.

The metrics package contains useful methods to calculate classification scores, for example, the f1_score and roc_auc_score methods. The package also contains the classification_report method to print a report of classification results and the confusion_matrix method which returns a matrix of classification results. Scikit-learn also provides many vectorizer implementations for preprocessing text data.

[9] https://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator

4 OVERVIEW OF TEXT PREPROCESSING TECHNIQUES

History of Natural Language Processing (*NLP*) began in 1950 when Alan Turing handled this topic in his paper Computing Machinery and Intelligence. Since then the NLP has taken its place in fields of computer science, artificial intelligence and computational linguistics. NLP consists of processes required to interact between computer and human languages. In other terms, NLP is feature engineering to transform unstructured text to features and machine learning algorithms for further processing. Kantardzic (2011) splits feature engineering into data preparation and data reduction phases. He estimates that over 50% of effort is used in these phases in the data mining process. Data preparation concerns transformation of raw data, normalizations, data smoothing, handling missing data, and dealing with outliers. Data reduction concerns feature reduction, dimensionality reduction, and value reduction. Feature reduction is important since most of the real world data mining applications are characterized by high dimensional data, where not all of the features are important. Dimensionality reduction refers to mathematical transformations to reduce features, for example, the Karhunen – Loeve's Principal Component Analysis. Value reduction is a reduction in the number of discrete values for a given feature. This is also known as feature discretization. In this thesis the value reduction is implemented transforming CVSS scores into four severity levels.

In NLP, syntactic analysis is used to evaluate how the natural language fits with the grammatical rules. Some widely known NLP techniques are: **Lemmatization** is converting various inflected forms of a word into a single form. **Morphological segmentation** is dividing words into individual units called morphemes. In word segmentation a large piece of continuous text is divided into distinct units. **Part-of-speech tagging** is used to identify the part of speech for every word. Parsing involves undertaking grammatical analysis for the provided sentence. **Stemming** is cutting the inflected words to their root form. **Tokenization** is splitting the text into a sequence of N-gram tokens. N-gram means combining a sequence of N-words into tokens. For example, a 2-gram token contains two subsequent words. A **stop word** is a commonly used word which might not contain any useful information, such as English words "the", "a", "an", "in", et cetera. **Named entity recognition** is a technique that tries to map items in the text to proper names. Many of the introduced techniques are implemented in the Stanford CoreNLP toolkit (Manning et al, 2014). Another toolkit which used in this thesis is Natural Language Toolkit (*NLTK*). It is a platform for building Python programs to work with written natural text. In Figure 6 there is Wijayasekara et al. (2014) technique to process and classify bug reports to regular bugs and bugs that are security related. These preprocessing steps can be used in other text classification purposes as well.

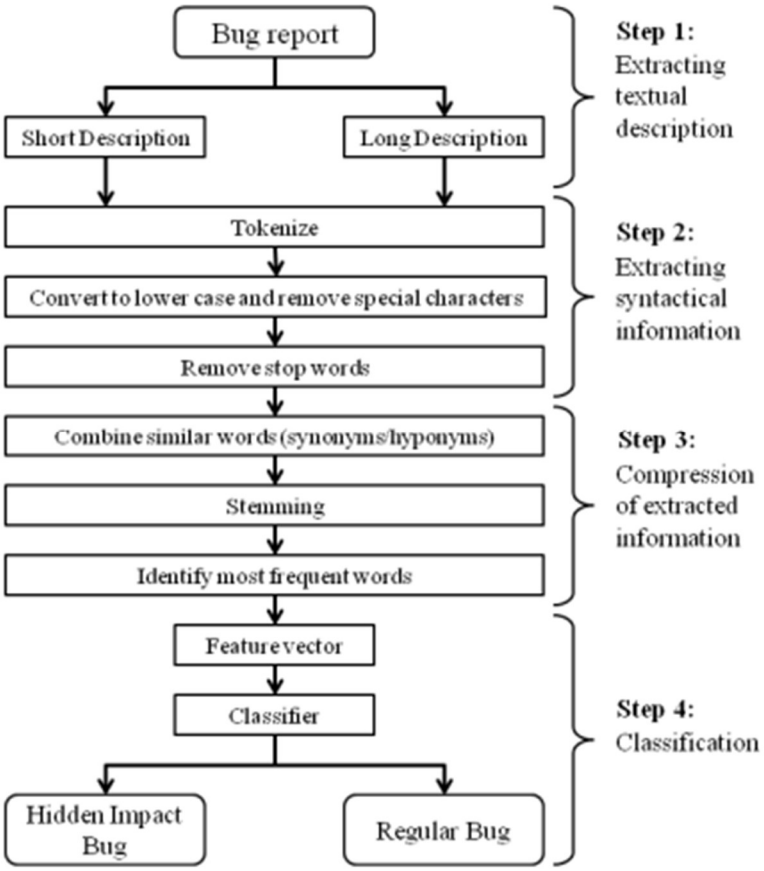

Figure 6. Natural Language Processing Steps (Wijayasekara et al., 2014)

In the figure, Step 1 represents the basic unstructured text extraction. The short description is a title which can be 5-10 words long. The long description is a defect report a few sentences about a bug. In step 2 the syntactical information is extracted in the form of single unique words which are converted to lower case. The extraction process removes words and symbols that might not carry a significant amount of information. This is done by tokenizing the description into a bag-of-words representation and then removing stop words. A simple form of bag-of-words representation is to keep count how many times each word appear in a document. In Step 3 similar words are combined and stemming is performed to identify the most frequent words. Step 4 illustrates how the features are passed to a classifier which makes prediction concerning the text given as input of the whole process. Steps 2 and 3 represents the data preparation and reduction phases in data mining. Wijayasekara et al. (2012) used Wordnet to identify synonyms to combine them. WordNet is a large lexical database of English. They used Porter stemming to cut words into their basic form. To combine words that carry similar information further reduces the number of features in matrix.

4.1 Term Frequency – Inverse Document Frequency

Term Frequency–Inverse Document Frequency (*TFIDF*), is a numerical statistic weighting scheme to reflect how important a word is to a document in a collection or corpus. In linguistics, a corpus is a large and structured set of texts. TFIDF is a very popular weighting scheme used in digital library systems. TFIDF can be calculated as follows:

$$d(i) = TF(wi, d) \times IDF(wi) \tag{4.1}$$

$$IDF(w) = \log(\frac{|D|}{DF(w)}) \tag{4.2}$$

Joachims (1996) explained how TFIDF works as follows: "The term frequency $TF(wi, d)$ is the number of times word w occurs in document d. The document frequency $DF(w)$ is the number of documents in which the word w occurs at least once. The inverse document frequency $IDF(w)$ can be calculated from the document frequency. $|D|$ is the total number of documents. The inverse document frequency of a word is low if it occurs in many documents and highest if the word occurs in only once. The value $d(i)$ of feature wi for document d is then calculated as the product. $d(i)$ is called the weight of word wi in document d. This weighting scheme says that a word wi is an important indexing term for document d if it occurs frequently in it. On the other hand, words which occur in many documents are rated less important due to their low inverse document frequency." His research focused text categorization using Naïve Bayes and TFIDF classifiers.

4.2 Stemming and Lemmatization

Stemming is a crude heuristic process that cuts off the ends of words in the hope of correctly transforming words into its root form. The original stemming algorithm was introduced in 1979 in the Computer Laboratory, Cambridge England. Porter Stemming algorithm is written and maintained by its author, Martin Porter. In this thesis, the NLTK library's Snowball Stemmer was used which implements the Porter stemmer algorithm (Porter, 1980).

Another approach determining a stem of a word is lemmatization. Lemmatization is process of determining the dictionary form of a word based on its intended meaning. It may use a dictionary for word mappings or rule-based approaches. According Camacho-Collados and Pilehvar (2018) research a simple tokenization works equally or better than lemmatization or multiword grouping for text classification with neural network based classifiers. In this thesis, the NLTK library's WordnetLemmatizer was used.

4.3 Text Vectorization

The Scikit-learn library provides many helpful classes in feature extraction from text. In Scikit-learn, the process of turning text documents into numerical features is called vectorization. Feature extraction differs from feature selection: feature extraction transforms arbitrary data into numerical features usable for machine learning. Feature selection is a machine learning technique applied on these features. (Scikit, 2019). Scikit-learn's CountVectorizer provides a basic bag-of-words representation of features. The TfidfVectorizer class provides a TFIDF representation of features. In Table 8 some of important vectorizer parameters are introduced, full list of parameters are available in the user guide.[10]

Table 8. Vectorizer parameters

Parameter	Description	Possible values
stop_words	"If 'english', a built-in stop word list for English is used. If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. If None, no stop words will be used. The max_df parameter can be set to a value in the range [0.7, 1.0] to automatically detect and filter stop words based on intra corpus document frequency of terms."	string {'english'}, list, or None (default)
lowercase	"Convert all characters to lowercase before tokenizing."	boolean, True by default
ngram_range	"The lower and upper boundary of the range of n-values for different N-grams to be extracted. All values of n such that min_n <= n <= max_n will be used."	tuple (min_n, max_n) Default: (1, 1)
min_df	"When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. If float, the parameter represents a proportion of documents, integer absolute counts."	float in range [0.0, 1.0] or int, default=1
token_pattern	"Regular expression denoting what constitutes a token. By default it selects tokens of 2 or more alphanumeric characters. Punctuation is completely ignored and always treated as a token separator."	string, Default: r'\b[^\d\W]+\b'

Stop words are assumed uninformative representing the content and can be removed to avoid them being used in prediction. Stop word lists may contain words that could include important information, for example "vulnerability". Stop word lists are a simple tool managing noise. (Nothman et al., 2018). In this thesis the token pattern's regular expression is changed from default so that only at least three characters long alphanumerical words are extracted as features.

[10] https://scikit-learn.org/stable/modules/feature_extraction.html

The Scikit-learn built-in vectorizers do not account for potential misspellings or word derivations but the functionality can be added by customizing a tokenizer or analyzer. The Scikit-learn user guide (2019) instructs that instead of passing customized methods as constructor parameters more advisable way is to inherit the class and override the member methods: build_preprocessor, build_tokenizer, and build_analyzer. The basic functionality of the member methods are described more detailed in the user guide. [11]

[11] https://scikit-learn.org/stable/modules/feature_extraction.html

5 RELATED WORK

Jacob J. Tyo (2016) concluded in his thesis that performance of each classifier varied greatly between datasets but the Naïve Bayes classifier outperformed all other classifiers in all cases while SVM classifiers were always among the best performing. In the research, three separate NASA's defect databases used as datasets. The research focused detecting hidden impact bugs using supervised and unsupervised machine learning approaches. A hidden impact bug can be defined as vulnerability identified as such after the bug had been disclosed to the public. The outcome of this study was that while unsupervised approach performed well, it was not as effective as the supervised method, achieving a G-Score of only 0,715 where the best supervised approach achieved G-Score of 0,903. The research suggests to continue to explore the generalizability of vulnerability profiles and open source bug database empirical studies should be performed. As an alternative, the research proposes continuing anomaly detection approach towards multiclass classification treating each class of the multiclass problem as a one-class anomaly detection problem. According to the research there are no prior work using unsupervised machine learning to classify software security bug reports exists.

Miyamoto et al. (2015) compared the following algorithms: Naïve Bayes Classifier, Latent Dirichlet Allocation, Latent Semantic Indexing, and Supervised Latent Dirichlet Allocation (*SLDA*) to estimate the CVSS score. They used NVD vulnerability descriptions as a dataset. Their finding was that the SLDA algorithm is the most efficient. They proved even better results adding an annual weight to the algorithm. They concluded that their method cannot detect completely novel vulnerabilities. They also observed that if information about CWE were assigned to the vulnerability, it might be possible to improve accuracy. So, they speculated that it might be possible to create a two-step method which first estimates the CWE-ID from the description and then estimate the CVSS base metrics using their method.

Lamkamfi et al. (2011) found that Naïve Bayes Multinomial classifier is able to achieve stable accuracy the fastest, having only about 250 bug reports of each severity at training phase. They studied GNOME and Eclipse defect databases to predict severity of bug reports but not in the sense of vulnerabilities or security. They study Naïve Bayes, K-Nearest Neighbor, and Support Vector Machines (*SVM*) approaches.

Bozorgi et al. (2010) explored OSVDB and Mitre CVE databases for the most likely exploited vulnerabilities. They used the SVM classifier to distinguish the data into two classes. They found that many of the features are irrelevant but a lot of information is contained in text fields. They used a bag-of-words representation for each text field to transform the text to features.

Peters et al. (2017) developed a tool to find security related bug reports using a few open source project's defect databases. They used stop-word removal and TFIDF preprocessing techniques and compared Random Forest, Naïve Bayes,

Logistic Regression, Multilayer Perceptron, and K-Nearest Neighbors machine learning algorithms. Their results show that the selection of machine learning algorithm varies between the databases. The tool handles the class imbalance problem. Data is said to suffer from class imbalance problem when the class distributions are highly imbalanced. In this context, many classification learning algorithms have low predictive accuracy for infrequent class.

Han et al. (2017) studied predicting CVSS2 scores based on NVD vulnerability descriptions. They divided the vulnerabilities into four classes by severity from critical to low. In their approach, instead of relying on manual feature engineering, they used word embeddings and a one-layer shallow Convolutional Neural Network (*CNN*) to automatically capture discriminative word and sentence features of vulnerability descriptions for predicting vulnerability severity. They adopted an undersampling strategy to balance the training and testing datasets. They compared their method to TFIDF+SVM, word embeddings+SVM, word embeddings+2-layer CNN, and word embeddings+CNN with Long Short Term Memory (*LSTM*). Their method outperformed the rival methods having F1-score averaged by severity classes 0,816. Word embedding is the collective name for techniques where words or phrases from the vocabulary are mapped to vectors of real numbers. It involves a mathematical embedding from a space with many dimensions per word to a continuous vector space with a much lower dimension. CNN and LSTM are neural network approaches in machine learning. Originally CNN was designed to map image data to an output variable. LSTM was proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber to deal with the exploding and vanishing gradient problems.

Another way finding vulnerabilities in addition to crawling through defect texts is to scan the source code. As defect reports may contain code snippets within the text, Wijayasekara et al. (2012) used a static code analyser to create features in addition to text preprocessing techniques. Li et al. (2018) developed a tool using neural network techniques to detect potential vulnerabilities in source code. The tool is limited to C/C++ programming languages only.

Sanguino and Uetz (2017) analysed CPE dictionary and CVE feeds. They developed a method that recommends a prioritized list of CPE identifiers for a given software product. Based on their observations there are four major issues: CVE entries without CPE references, software products without assigned CPEs, typographical errors, and a lack of synchronization between both datasets may lead to incorrect results output of Vulnerability Management Systems. They propose that NIST, which is responsible for maintaining these repositories, should define a mechanism to overcome these issues.

Ruohonen and Leppänen (2018) studied textual information retrieval techniques to map CVEs to CWEs. Based on the NVD information they attempted to assign CWEs to vulnerabilities in four software products that can be found from the Snyk database. They used traditional text pre-processing techniques and four

weighting systems, including TFIDF, 1-3 grams, and latent semantic analysis to map CWE to vulnerabilities. They got poor precision results but it cannot be generalized to all repositories or programming languages. They found that simple keyword searches based on CVE and CWE identifiers are more robust. Their observation were, that the choice over particular security-related corpora has likely a strong effect upon the vulnerability-CWE mappings and CWEs are not very similar with respect to each other. Even though unigrams gave the best results the trigrams “denial of service” and “NULL pointer dereference”, for example, should attain higher weights than any of the other N-grams.

6 METHODS AND DATA

NVD database was used to study CVSS and CWE classification and also to learn one-class classifiers. To study unary classification, a few defect databases of open source projects were collected. The security related issues were manually labelled by a security specialist to confirm vulnerability keywords can be found from defect reports. Automatic CVSS and CWE classifications were run using NVD data only.

6.1 NVD Data

NVD provides vulnerability data feeds in the JSON format, specified in NVD JSON 1.0 Schema. The data feeds are available in compressed format where files are divided on a yearly basis. Also, the recently added and lately modified vulnerabilities are available in separate files. In this thesis, the latest vulnerabilities and all vulnerability data from 2018 and until end of April in 2019 were downloaded. 18755 vulnerabilities were used as datasets in experiments. In Figure 7 there is an example of a single item of vulnerability data which is downloaded and parsed for further usage. According to Miyamoto et al. (2015) assigning higher weights to recently published CVEs gives better classification results. Also, by narrowing down the dataset size it might save computational power.

```
['CVE-2018-5357',
 'ImageMagick 7.0.7-22 Q16 has memory leaks in the ReadDCMImage function in coders/dcm.c.',
 'CWE-399',
 'AV:N/AC:M/Au:N/C:N/I:N/A:P',
 4.3,
 'CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H',
 6.5,
 ['imagemagick', 'imagemagick', 'canonical', 'ubuntu_linux']]
```

Figure 7. Example of Parsed NVD Data

The data structure presented is a Python list in which items are as follows, listed from the top: CVE identifier, Vulnerability text, CWE class, CVSS2 vector, CVSS2 Score, CVSS3 vector, CVSS3 score, and CPE vendor and component names. Vulnerability texts were preprocessed and passed to machine learning algorithms to predict classifications. The CVSS vectors and CWE classes were used to validate the classifier results. CPE names, 6571 of them, were studied to compare whether excluding or including them have better results. According to Han et al. (2017) vulnerability texts are concise, average length is only 37.5 ± 15.4 words. Not all vulnerabilities have CVSS2, CVSS3 and CWE given in whole dataset. In Figure 8 the data distribution of CVSS2 metrics are presented. There were total of 18124 vulnerabilities that had CVSS2 vector given in whole dataset.

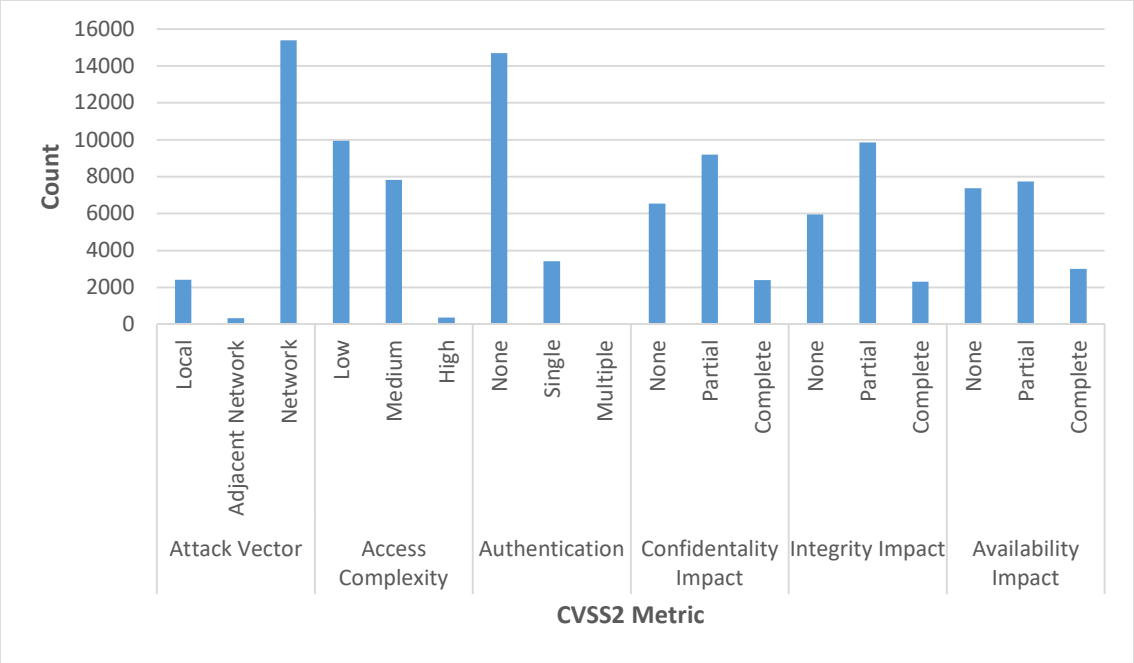

Figure 8. CVSS2 Data Distribution by Vectors

In this thesis, the classification approach tries to predict vectors metric by metric and then calculating the score. In CVSS2 there are six classification predictions to be made based on metrics to calculate the actual score. As Figure 8 indicates there is high class imbalance in the attack vector, access complexity, and authentication metrics. The class "Multiple" of Authentication metric was completely discarded because there are only 4 observations of them. On the attack vector and access complexity metrics there are over 300 observations of low number classes. Major number of vulnerabilities can be exploited from network, having low access complexity with no authentication required by attacker. Partial loss of confidentiality, integrity, and availability are the major impacts the vulnerabilities have. This average type of vulnerability base score can be calculated as 7,5.

In Figure 9 the CVSS3 data distribution by vectors is shown. There are 17928 vulnerabilities in total the CVSS3 vector is given in whole dataset. The classification approach makes eight predictions before the actual CVSS3 score can be calculated. As the figure indicates, there are some classes that have low number of observations but none have to be discarded. On the attack vector there are physical vulnerabilities only 177 and exploitable from adjacent network only a little over 300 observations. Surprisingly, with low impact on availability there are only about 300 vulnerabilities. Most of the vulnerabilities are exploitable from network, having low attack complexity with no privileges required from attacker and having no user interaction required. Mostly, as the scope is not changed the authorization privileges are not elevated by vulnerable components. The major impact is high on confidentiality, integrity, and availability for CVSS3 estimated vulnerabilities. This type of average vulnerability gives base score of 9,8.

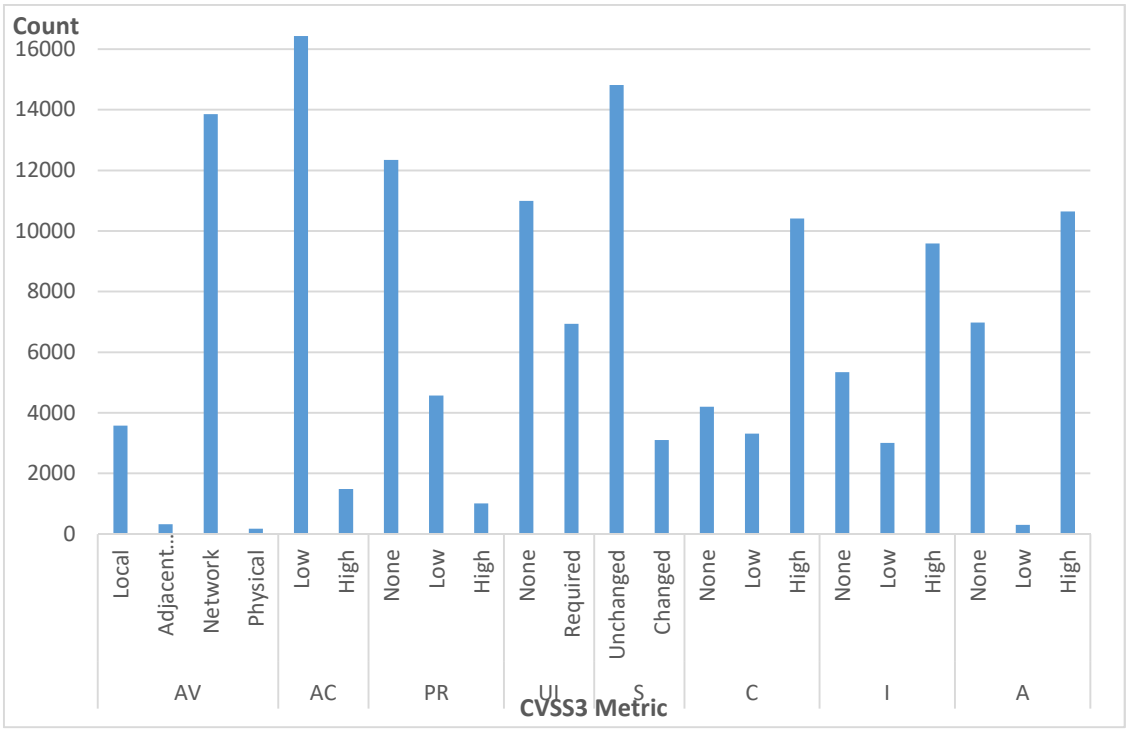

Figure 9. CVSS3 Data Distribution by Vectors

Han et al. (2017) categorized vulnerabilities into four groups by severity as indicated in Table 9. Atlassian security advisories classify vulnerabilities into four severity levels based on CVSS scores. They estimated vulnerability severities based on these categories. In this thesis the severity categories are used in classification after calculating the actual score. In Figure 10 the vulnerabilities by severity are shown in CVSS2 and CVSS3 scores.

Table 9. Vulnerabilities by Severity

Range	Description	Category
9.0 – 10.0	"Vulnerabilities this range are usually exploited straightforwardly since the attacker does not need any special authentication credentials or knowledge about individual victims."	critical
7.0 – 8.9	"Vulnerabilities are usually difficult to exploit. Exploitation of such Vulnerabilities may result in elevated privileges, significant downtime, or compromise of the confidentiality, integrity, or availability."	high
4.0 – 6.9	"Vulnerabilities typically require the attacker to reside on the same local network as the victim or manipulate individual victims via social engineering tactics. Impact of such vulnerabilities is usually mitigated by factors such as user privileges, authentication requirements."	medium
0.1 – 3.9	"Vulnerabilities in the "low" level have very little impact on an enterprise's business. Local or physical system access is always required for exploiting."	low

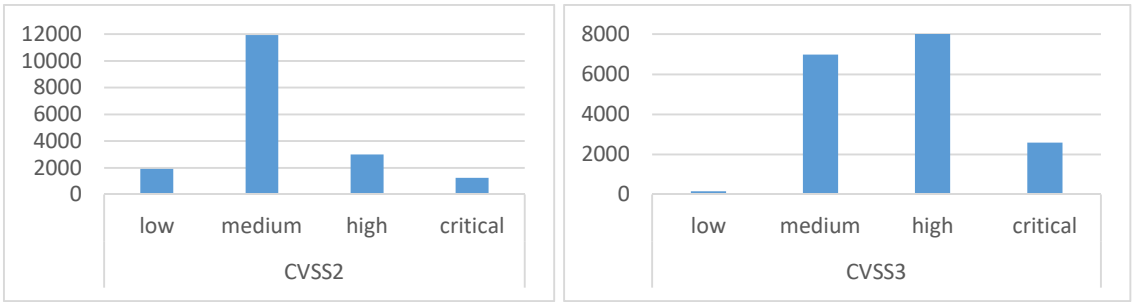

Figure 10. Distribution of CVSS Counts Based on Vulnerability Severity

In this thesis, in CVSS2 score the 11946 vulnerabilities fell into medium category but the rest vulnerabilities were between 1000-3000. In CVSS3 score there were 163 vulnerabilities in low category, 6984 vulnerabilities in medium, 8187 vulnerabilities in high, and 2590 vulnerabilities in critical severity.

There are 18755 vulnerabilities and 105 different CWEs given. In Figure 11 the distribution of CWEs is listed. There are 38 CWEs which are assigned less than 10 times. In top 5 CWEs there 8228 vulnerabilities out of all 18775.

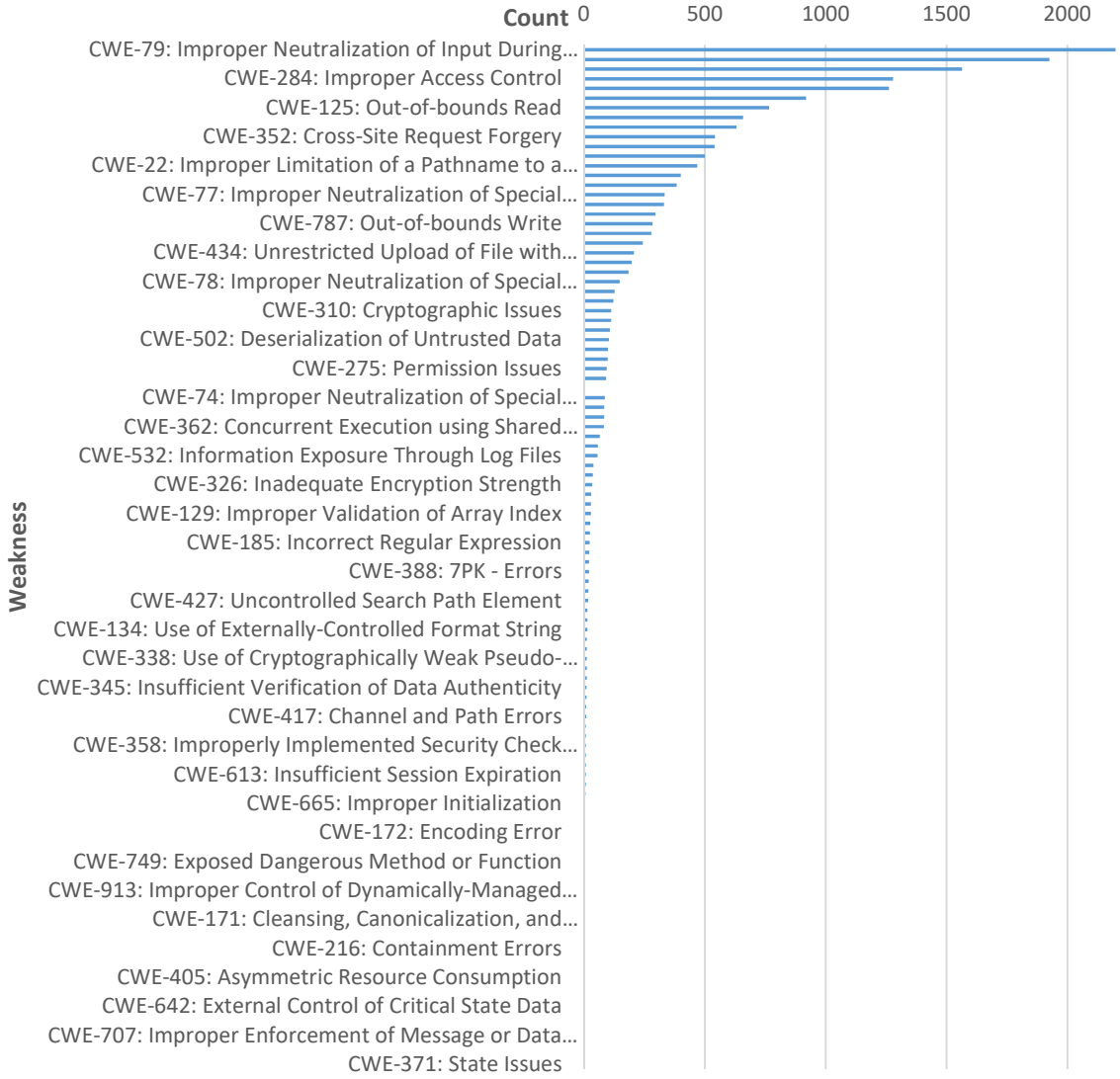

Figure 11. Original Dataset CWE Distribution

The top 5 original dataset CWEs consists of Improper Neutralization of Input During Web Page Generation, Improper Restriction of Operations within the Bounds of a Memory Buffer, Improper Input Validation, Improper Access Control, and Information Exposure. In this thesis, an attempt to balance the CWE data by finding root categories of low number CWEs was taken: If there were less than ten times CWE assigned a search for root category was attempted. This is implemented in the CweFinder class which can be found from Appendix 1. The CWE hierarchy is available in different formats separated in three different views. The research concept view was used in this thesis only. In Figure 12 there is the distribution of root CWEs of the complete dataset presented.

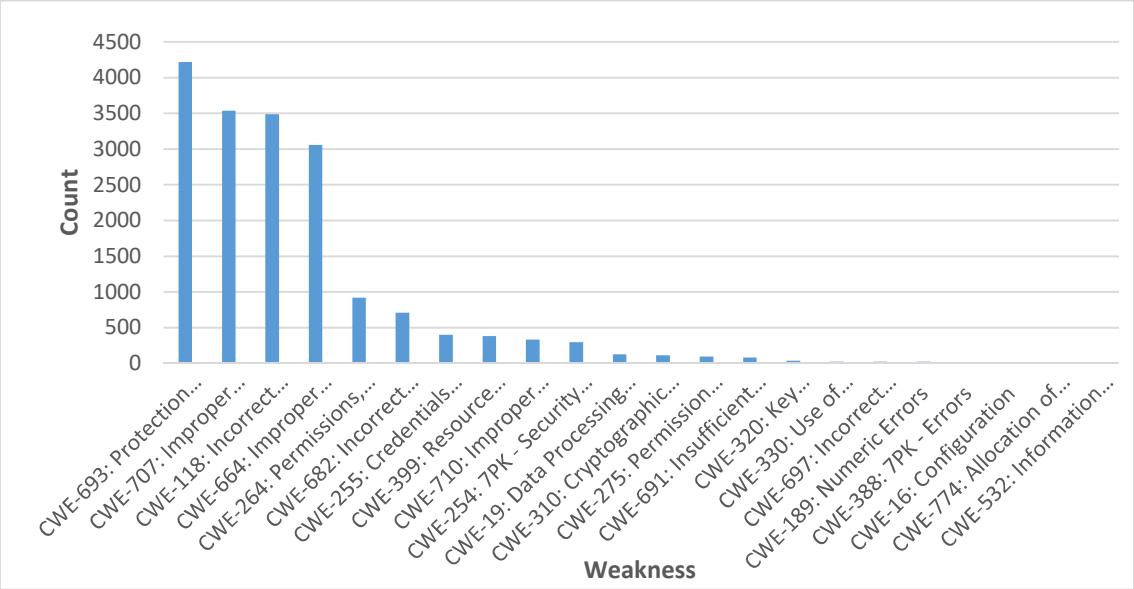

Figure 12. Distribution of Root CWEs on Complete Dataset

In the Research Concept view the root CWEs concentrates mainly around four major weakness types as shown in Figure 12:

- CWE-693: Protection Mechanism Failure
- CWE-707: Improper Enforcement of Message or Data Structure
- CWE-118: Incorrect Access of Indexable Resource
- CWE-664: Improper Control of a Resource Through its Lifetime.

After finding root CWEs for the complete dataset, there were two root items assigned two times only: Information Exposure Through Log Files and Allocation of File Descriptors. After completely discarding CWEs less or equal than ten in numbers the data ended up having 18753 vulnerabilities in 70 different CWEs. The 10-fold cross-validation method requires at least ten observations in each class. The data distribution is presented in Figure 13 which were used in the experiments. The completely discarded CWEs are shown in Table 10. A closer look on the discarded CWEs it reveals many deprecated categories or suggested for deprecation on upcoming versions. There were 631

empty CWEs and 121 CWEs labelled 'NVD-CWE-noinfo' discarded completely as well as they were not describing any vulnerability causes.

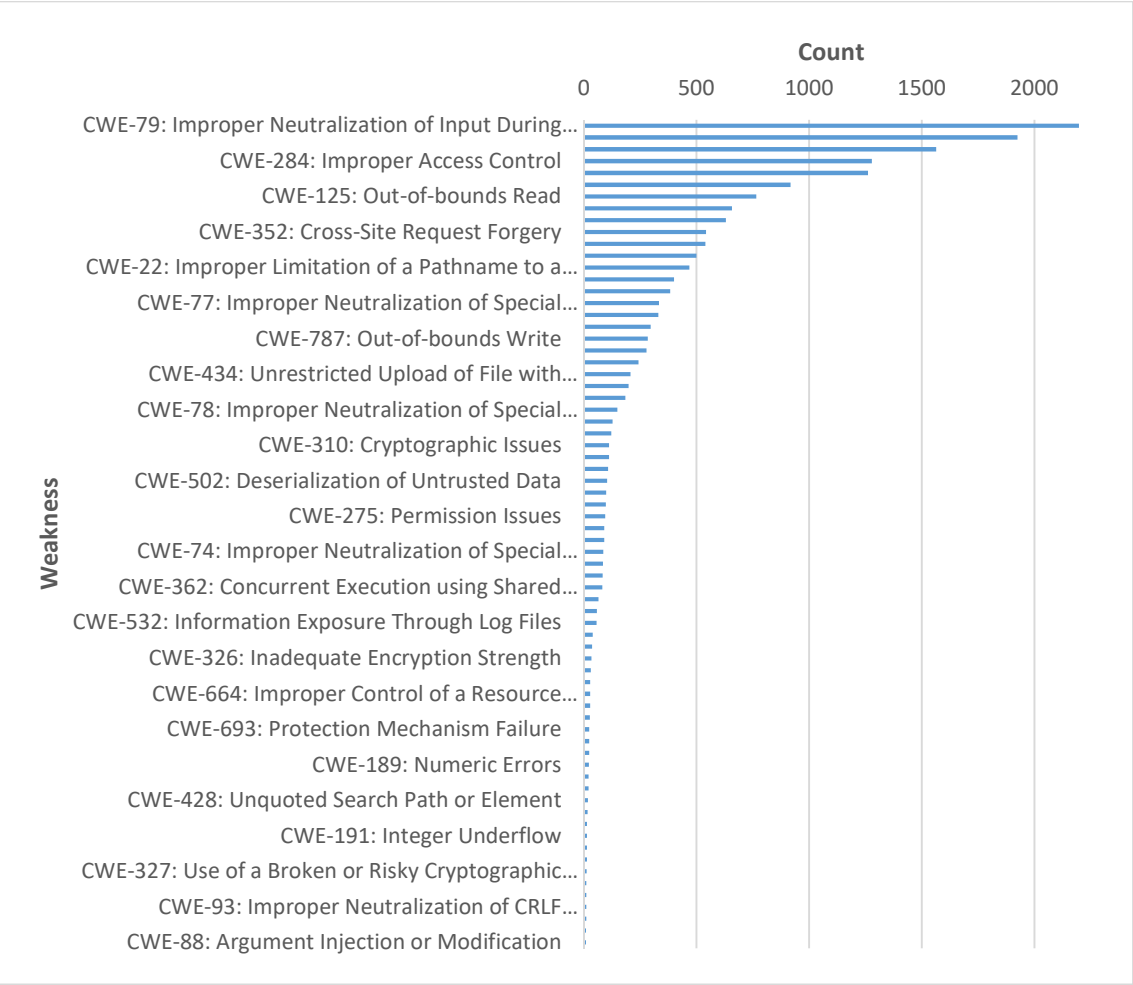

Figure 13. CWE Distribution of Selected Root Items

There were 32 vulnerabilities which were rejected for some reason, for example, rejected by CNA or duplicates which do not contain any useful information for prediction. These vulnerabilities were discarded from the dataset. Also, there were 41 vulnerabilities disputed or rejected by CNA which however contains useful descriptions but this single word was removed. These vulnerability descriptions starts with "** DISPUTED **" or "** REJECT **". Unwanted common words were removed from vulnerability descriptions to make distinction better between a defect and vulnerability. The following words removed in addition to Scikit-learn's default stop words: "issue", "defect", "bug", "fault", "flaw", "mistake", "error", "version", "system", "because", and "before".

Table 10. Details of Discarded and Deprecated CWEs

ID	Name	Description	∑
CWE-18	"CWE CATEGORY: Source Code"	"This entry is being considered for deprecation."	1
CWE-398	"CWE CATEGORY: 7PK – Code Quality"	"Poor code quality leads to unpredictable behaviour. From a user's perspective that often manifests itself as poor usability."	1
CWE-371	"CWE CATEGORY: State Issues"	"Weaknesses in this category are related to improper management of system state."	1
CWE-199	"CWE:CATEGORY: Information Management"	"Weaknesses in this category are related to improper handling of sensitive information."	1
CWE-534	"DEPRECATED: Information Exposure Through Debug Log Files"	"This entry has been deprecated because its abstraction was too low-level. See CWE-532."	2
CWE-769	"DEPRECATED: Uncontrolled File Descriptor Consumption"	"This entry has been deprecated because it was a duplicate of CWE-774. All content has been transferred to CWE-774."	2
CWE-171	"CWE CATEGORY: Cleansing, Canonicalization, and Comparison Errors"	"Weaknesses in this category are related to improper handling of data within protection mechanisms that attempt to perform neutralization for untrusted data."	2
CWE-754	"Improper Check for Unusual or Exceptional Conditions"	"The software does not check or improperly checks for unusual or exceptional conditions that are not expected to occur frequently during day to day operation of the software."	6
CWE-17	"CWE CATEGORY: Code"	"This entry is being considered for deprecation."	7
CWE-91	"XML Injection"	"Entry might need to be deprecated or converted to a general category."	7
CWE-358	"Improperly Implemented Security Check for Standard"	"The software does not implement or incorrectly implements one or more security-relevant checks as specified by the design of a standardized algorithm, protocol, or technique."	7
CWE-444	"Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')"	"When malformed or abnormal HTTP requests are interpreted by one or more entities in the data flow between the user and the web server, such as a proxy or firewall, they can be interpreted inconsistently, allowing the attacker to "smuggle" a request to one device without the other device being aware of it."	8
CWE-682	"Incorrect Calculation"	"The software performs a calculation that generates incorrect or unintended results that are later used in security-critical decisions or resource management."	8
CWE-417	"CWE CATEGORY: Channel and Path Errors"	"This category is being considered for deprecation."	9
CWE-346	"Origin Validation Error"	"The software does not properly verify that the source of data or communication is valid."	9

6.2 Defect Datasets

Defect databases were used to measure one-class classification performance. The NVD vulnerability descriptions were used at learning phase and defect databases were used to predict. In Table 11 there are open source projects defect datasets introduced (Peters et al., 2017) which were used in this study.

Table 11. List of Bug Report Databases

Project	**Domain**	**Start Date**	**End Date**	∑
Chromium	"Web browser called Chrome."	Aug 30 2008	Jun 11 2010	41940
Wicket	"Component-based web application framework for the Java programming."	Oct 20 2006	Nov 9 2014	1000
Ambari	"Hadoop management web UI backed by its RESTful APIs."	Sep 26 2011	Aug 8 2014	1000
Camel	"A rule-based routing and mediation engine."	Jul 8 2007	Sep 18 2013	1000
Derby	"A relational database management system."	Sep 28 2004	Sep 17 2014	1000

The databases contains all together 351 security labelled defect reports and the rest are non-security related, normal defect reports. To make sure the security related reports contain a real vulnerability keywords, security specialist manually labelled these items and 148 items were labelled as potentially vulnerable. 200 randomly selected samples per database, non-security related defect reports were used as a testing set and the potentially vulnerable items were added on the top of that. As a result the testing tests size of 1148 defect reports were used in one-class classification experiments. In Figure 14 there is an example of manually labelled defect which is potentially vulnerable.

```
"Provide way to optionally enable two-way SSL for Server-Agent communication
The two-way SSL mechanism used during server-agent registration exists to prote
ct communication. This is useful in production environments but in typical 'fir
st use' or POC scenarios  having this level of security is not necessary. As we
ll  certificate generation can be problematic causing failures.We need to provi
de a way to make this mechanism optional:1) By default  ship with Server-Agent
Two-Way SSL off.2) At any time post install  a user should be able to turn on T
wo-Way SSL and turn it back off  etc.   "
```

Figure 14. Example of Potentially Vulnerable Defect

6.3 Metrics

The metrics introduced in this chapter are popular metrics to measure and validate classification performance. There are four different outcomes where unary and binary classification can lead to in defect classification (Witten & Frank, 2005, p. 162):

- True Positive (*TP*) is correctly classified potentially vulnerable defect.
- True Negative (*TN*) is correctly classified normal defect.
- False Positive (*FP*) is incorrectly classified potentially vulnerable defect.
- False Negative (*FN*) is incorrectly classified normal defect.

In optimal case, all predictions are correctly classified and there are no incorrectly classified samples. Gates and Taylor (2007) argued a system can be completely useless due to high number of false positives even if all the correctly classified samples are found. Even though their research was about anomalous intrusion detection the same analogy applies here: the false positive rate should not be higher than 1%. The outcomes of classification measures can be used to calculate the true negative rate (*TNR or specificity*), true positive rate (*TPR, recall or sensitivity*), false positive rate (*FPR*), precision, accuracy, F1-score (*F1*), G-score (*G*), Matthews Correlation Coefficient (*MCC*), and Area Under Curve of Receiver Operating Characteristic (*AUC*) as follows:

$$TNR = \frac{TN}{TN + FP} \tag{6.3}$$

$$TPR = recall = \frac{TP}{TP + FN} \tag{6.4}$$

$$FPR = \frac{TP}{FP + TN} \tag{6.5}$$

$$precision = \frac{TP}{TP + FP} \tag{6.6}$$

$$accuracy = \frac{TN + TP}{TN + FP + TP + FN} \tag{6.7}$$

$$F1 = \frac{2 \times recall \times precision}{recall + precision} \tag{6.8}$$

$$G = \sqrt{precision \times recall} \tag{6.9}$$

$$MCC = \frac{TP \times TN - F \times FN}{\sqrt{(TP + F)(TP + F)(TN + FP)(TN + F)}} \tag{6.8}$$

$$AUC = \int_{x=0}^{1} TPR\left(FPR^{-1}(x)\right)dx \quad (6.9)$$

Accuracy, the proportion of correct classifications among all classifications, is a very simple and intuitive measure but in classification and anomaly detection it might lead to too optimistic results. In this sense recall and precision are better measures to describe performance. As a single value of performance is pursued, F1-score is a harmonic mean of precision and recall which suits for binary and multiclass classification. F1-score reaches its best value at 1 and worst at 0. G-score is geometric mean of precision and recall. However, in literature F1-score is more common. F1-score does not take into account the class imbalance and random value of F1-score may vary. To make results comparable with other research the MCC and AUC might be better options. MCC is a measure of the quality of classification which takes into account true and false positives and negatives, and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. A coefficient of +1 represents a perfect prediction, 0 no better than random and −1 indicates total disagreement. Receiver Operating Characteristic curve is a graphical plot of TPR against FPR that illustrates the diagnostic ability of binary or unary classifier as its discrimination threshold is varied. AUC is the area under the curve. The value of AUC varies between 0 and 1. The random value being 0,5. AUC is a widely used measure of performance for classification but has no automatic extension to the multiclass case. MCC can be used as a performance measure in multiclass problems. (Hand, 2009; Jurman et al., 2012). The Scikit-learn library also sets limitations on which metrics to use: The cross_val_score method's scoring parameter supports a limited set of measurement options.[12] Scikit-learn (2019) provides options for averaging each class weighting in multiclass classification. In Table 12 the F1-score averaging options are introduced.

Table 12. F1-score Averaging Options

Option	Description
binary	"Only report results for the class specified. This is applicable only if the targets are binary."
micro	"Calculate metrics globally by counting the total true positives, false negatives and false positives."
macro	"Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account."
weighted	"Calculate metrics for each label, and find their average weighted by the number of true instances for each label. This alters 'macro' to account for label imbalance but it can result in an F-score that is not between precision and recall."
samples	"Calculate metrics for each instance, and find their average."

[12] https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

In this thesis, unary classification is measured using AUC-score and multiclass classification is measured using micro averaged F1-score. The results can be displayed as a two-dimensional confusion matrix with a row and column for each class. Each matrix element shows the number of test examples for which the actual class is the row and the predicted class is the column. All correct predictions are located in the diagonal of the table which is easy to visually inspect the table for prediction errors (Witten & Frank, 2005). Confusion matrices of CVSS classification can be found from Appendix 3.

Cross-validation is a standard statistical model validation technique for assessing how the results of a statistical analysis will generalize to an independent dataset. The goal is to validate how a predictive model will perform in practice. The outcome of cross-validation is reliable only if the learning and testing datasets are separate and a single instance of data belongs to either learning or testing dataset, but not both. The following cross-validation methods can be found from literature (Arlot & Celisse 2010; Kohavi 1995; Kantardzic 2011):

- **Holdout** is a method where a dataset is split into two. One dataset is used for learning and the other for testing. The ratio of the two datasets can be arbitrary but commonly two thirds is used for learning and the rest for testing. This method can be said cross-validation only if validated more than once. Average results of multiple rounds are then used.
- **K-fold cross-validation** is a method where a dataset is divided into K the equal size datasets. One dataset is used for testing and the rest for learning. Validation is performed K times so that the each partial dataset is used for testing at each round.
- **Leave-one-out** is a special case of K-fold cross-validation as K is chosen equal to the samples of a dataset. One single sample is used for testing and the other samples for learning. The validation is repeated K times. This method is computationally expensive.

In this thesis, the ten times repeated holdout method was used. Scikit-learn provides the train_test_split method to split a dataset. By default two thirds of the data are used for training. Also, 10-fold cross-validation was used in multiclass classification. The Scikit-learn's cross_val_score method provides the cv parameter to implement this.

Oversampling and undersampling are widely known techniques to handle the class imbalance in datasets. In oversampling the minority class data points are replicated as in undersampling only a part of the majority class data points are taken. There are methods to synthetically generate data points on minority data. In this thesis, undersampling strategy was chosen as oversampling may lead having the same data in learning and testing which further leads too optimistic results.

7 RESULTS

This chapter shows the results of vulnerability detection from open source defect databases and CVSS2 scoring, CVSS3 scoring, and CWE multiclass classification results. The goal of vulnerability detection is to classify whether a text contains keywords used in vulnerability descriptions or not. The multiclass classification results explain how well a vulnerability text can be used to estimate CVSS scores and CWE weakness. The dataset used in experiments contains 18755 vulnerabilities in total having vulnerability descriptions from beginning of 2018 until April 2019. The results shows classification performance comparisons with different classifiers and techniques.

7.1 Vulnerability Detection

Three different machine learning algorithms were tested detecting potential vulnerability descriptions from text which are available in Scikit-learn: OneClassSVM, LOF, and IsolationForest. The NVD data were used in learning and the defect texts in prediction. The results are presented in Table 13 where AUC-score of each algorithm is shown. Elapsed time was also measured during experiments.

Table 13. Vulnerability Detection Performance

Algorithm	AUC-score	Time
OneClassSVM	**0,682**	18,5s
IsolationForest	0,5	8,6s
LocalOutlierFactor	0,492	9,8s

The experiments were run with each algorithm default parameters. IsolationForest and LOF are faster but performance is not better than random. The OneClassSVM classifier was selected for further examination. Different vectorizers with the classifier were tested to find the most effective combination. Also, a keyword-based classifier was implemented to see the difference between machine learning approach and a simple text search. Source code for vulnerability detection and the keyword-based classifier can be found from Appendix 1. The classifier is using Scikit-learn CountVectorizer internally to have bag-of-words representation of vulnerability descriptions. Different N-gram combinations were tested. To classify a text to vulnerability it should contain at least two N-grams tokens of vulnerability descriptions. Stemming and lemmatizing were examined to measure which preprocessing technique gives the best performance. In Table 14 the results of different combinations of vectorizer N-gram ranges in the keyword-based classifier are presented. The min_df parameter, minimum document frequency, was left on its default value. The

parameter makes a vectorizer to ignore terms that have a document frequency strictly lower than the given threshold. The AUC-score, feature count of the vectorizer, and elapsed time are shown.

Table 14. Keyword-based Classifier Performance with N-gram Ranges

N-gram Range	AUC-score	Feature Count	Time
1 – 1	0,5039	8065	121,7s
1 – 2	0,503	44187	124,5s
1 – 3	0,50435	93456	127,3s
2 – 2	**0,7402**	37200	123,1s
2 – 3	0,74	84650	124,7s
3 – 3	0,52	48041	127,7s

The results show that counting 2-grams and not including single words gives much better performance. By having 3-grams in addition it increases the feature count but not performance. The 2-grams only was selected for further examinations in the keyword-based classifier. The classifier with stemming and lemmatizing was compared to different combinations of vectorizers with OneClassSVM. These results are presented in Figure 15. Also, different dataset sizes in learning are shown.

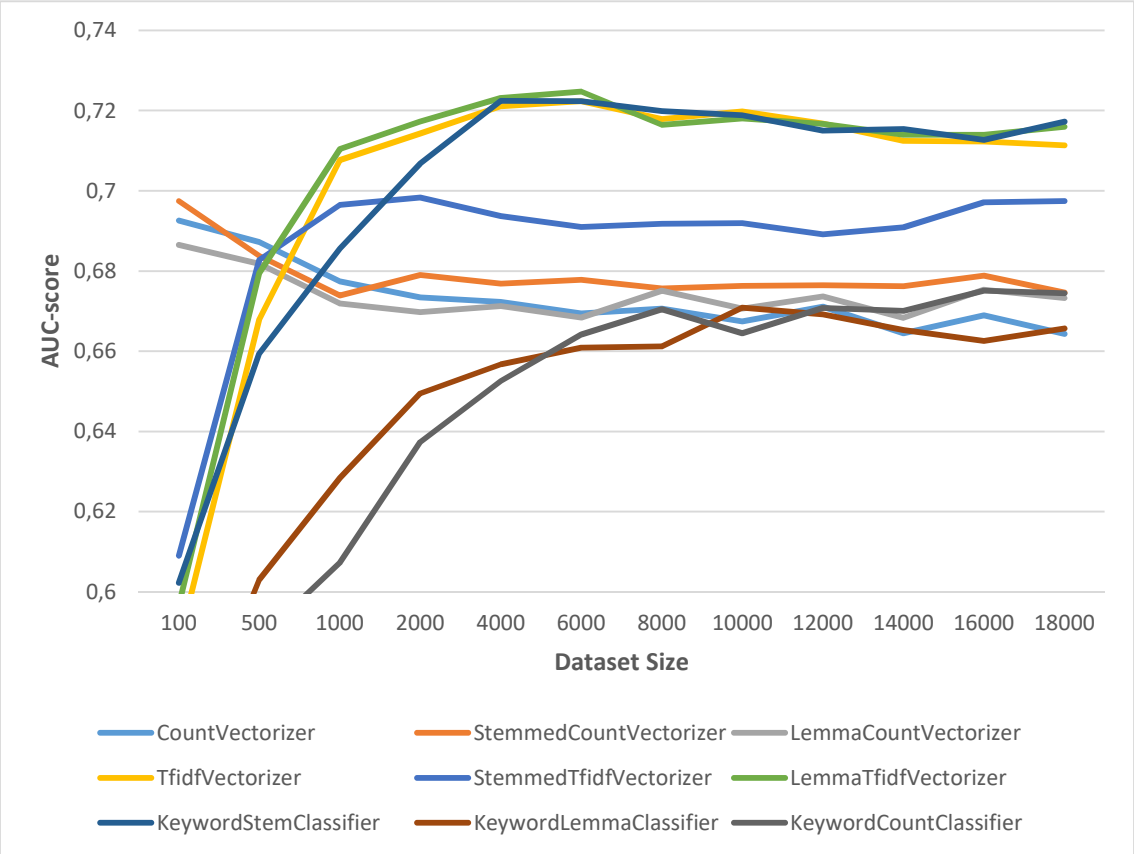

Figure 15. OneClassSVM AUC-score and Dataset Size

The results indicate that using 1000 – 4000 vulnerability descriptions would be enough to reach a stable performance. A plain CountVectorizer and its feature reducing techniques gives the best results at very low number of samples but none of these combinations exceed AUC-score of 0,7. The Keyword-based classifier with a plain CountVectorizer or lemmatizing gives no better results than 0,68 but surprisingly with stemming it gives 0,72 the dataset being 4000 vulnerabilities. OneClassSVM works better with TFIDF weighting than a plain bag-of-words representation. OneClassSVM with stemming does not exceed AUC-score 0,7 but a plain TFIDF performs almost as good as the TFIDF with lemmatizing reaching the score of 0,72 at 4000 vulnerabilities. The TFIDF vectorizer with lemmatizing was selected for further OneClassSVM experiments. In Table 15 AUC-score of different N-gram ranges are shown using OneClassSVM classifier with TFIDF vectorizing and lemmatization. Also, number of vectorizer features and elapsed times are presented.

Table 15. OCSVM+TFIDF+Lemmatizing Performance with N-gram Ranges

N-gram Range	AUC-score	Feature Count	Time
1 – 1	0,699	10401	14,3s
1 – 2	**0,703**	59834	16,7s
1 – 3	0,69	122703	19,1s
2 – 2	0,566	49556	10,9s
2 – 3	0,562	122686	13,6s
3 – 3	0,536	64357	9,3s

The results indicate that using single words as features is almost as good as 2-grams combined. The number of features increases considerably when the N-grams range is increased but having no effects on performance. In Table 16 there are the keyword-based classifier performance with different 2-gram token counts presented. The best performance is obtained when a text contains at least two vulnerable 2-gram tokens.

Table 16. AUC-score of Keyword-based Classifier 2-gram Counts

	1	**2**	**3**	**4**	**5**
Keyword-based Classifier	0,691	**0,731**	0,694	0,686	0,632

The vectorizers provide the min_df parameter to discard words which do not exists on a minimum amount of documents. In Table 17 the effect of different min_df parameters are shown. Increasing the parameter value it has a negative effect on the amount of features. Also, OneClassSVM elapsed time effect can be observed. OneClassSVM and the keyword-based classifier performance is at the highest having min_df = 1.

There are two generic options in Scikit-learn to find optimal combination of classifier hyperparameters: exhaustively by testing all possible combinations using GridSearchCV or randomly selecting a sample in a given number of candidates from a parameter space using RandomizedSearchCV. In this thesis,

the exhaustive approach was selected. In unary classification the exhaustive search must be implemented using ParameterGrid because a different dataset is needed to be passed to the fit method and to the predict method. In Figure 16 the output of exhaustive search is shown.

Table 17. Effect of Minimum Document Frequency Parameter

min_df	OCSVM AUC-score	KeywordStem AUC-score	OCSVM Feature Count	KeywordStem Feature Count	OCSVM Time
1	**0,711**	**0,728**	58943	37028	17,2s
2	0,708	0,705	15532	8217	13,8s
3	0,703	0,69	8573	3995	12,9s
4	0,702	0,671	5961	2545	13,7s
5	0,688	0,657	4665	1963	13,7s
10	0,673	0,618	2425	887	11,6s
20	0,688	0,587	1296	451	11,3s
30	0,686	0,577	951	334	11,4s
40	0,683	0,577	758	257	11,0s
50	0,682	0,563	618	199	10,6s

```
0.7334054054054056: {'clf__kernel': 'linear', 'clf__nu': 0.1, 'clf__shrinking': True, 'clf__tol': 1e-05}
0.7334054054054056: {'clf__kernel': 'linear', 'clf__nu': 0.1, 'clf__shrinking': True, 'clf__tol': 1e-06}
0.7334054054054056: {'clf__kernel': 'linear', 'clf__nu': 0.1, 'clf__shrinking': True, 'clf__tol': 1e-07}
0.7334054054054056: {'clf__kernel': 'linear', 'clf__nu': 0.1, 'clf__shrinking': False, 'clf__tol': 1e-05}
0.7334054054054056: {'clf__kernel': 'linear', 'clf__nu': 0.1, 'clf__shrinking': False, 'clf__tol': 1e-06}
0.7334054054054056: {'clf__kernel': 'linear', 'clf__nu': 0.1, 'clf__shrinking': False, 'clf__tol': 1e-07}
0.7319864864864865: {'clf__kernel': 'linear', 'clf__nu': 0.2, 'clf__shrinking': True, 'clf__tol': 1e-05}
0.7319864864864865: {'clf__kernel': 'linear', 'clf__nu': 0.2, 'clf__shrinking': True, 'clf__tol': 1e-06}
0.7319864864864865: {'clf__kernel': 'linear', 'clf__nu': 0.2, 'clf__shrinking': True, 'clf__tol': 1e-07}
0.7319864864864865: {'clf__kernel': 'linear', 'clf__nu': 0.2, 'clf__shrinking': False, 'clf__tol': 1e-05}
0.7319864864864865: {'clf__kernel': 'linear', 'clf__nu': 0.2, 'clf__shrinking': False, 'clf__tol': 1e-06}
0.7319864864864865: {'clf__kernel': 'linear', 'clf__nu': 0.2, 'clf__shrinking': False, 'clf__tol': 1e-07}
0.6864594594594594: {'clf__kernel': 'linear', 'clf__nu': 0.5, 'clf__shrinking': True, 'clf__tol': 1e-05}
0.6864594594594594: {'clf__kernel': 'linear', 'clf__nu': 0.5, 'clf__shrinking': True, 'clf__tol': 1e-06}
0.6864594594594594: {'clf__kernel': 'linear', 'clf__nu': 0.5, 'clf__shrinking': True, 'clf__tol': 1e-07}
0.6864594594594594: {'clf__kernel': 'linear', 'clf__nu': 0.5, 'clf__shrinking': False, 'clf__tol': 1e-05}
0.6864594594594594: {'clf__kernel': 'linear', 'clf__nu': 0.5, 'clf__shrinking': False, 'clf__tol': 1e-06}
0.6864594594594594: {'clf__kernel': 'linear', 'clf__nu': 0.5, 'clf__shrinking': False, 'clf__tol': 1e-07}
0.7339054054054055: {'clf__kernel': 'rbf', 'clf__nu': 0.1, 'clf__shrinking': True, 'clf__tol': 1e-05}
0.7339054054054055: {'clf__kernel': 'rbf', 'clf__nu': 0.1, 'clf__shrinking': True, 'clf__tol': 1e-06}
0.7339054054054055: {'clf__kernel': 'rbf', 'clf__nu': 0.1, 'clf__shrinking': True, 'clf__tol': 1e-07}
0.7339054054054055: {'clf__kernel': 'rbf', 'clf__nu': 0.1, 'clf__shrinking': False, 'clf__tol': 1e-05}
0.7339054054054055: {'clf__kernel': 'rbf', 'clf__nu': 0.1, 'clf__shrinking': False, 'clf__tol': 1e-06}
0.7339054054054055: {'clf__kernel': 'rbf', 'clf__nu': 0.1, 'clf__shrinking': False, 'clf__tol': 1e-07}
0.7324864864864866: {'clf__kernel': 'rbf', 'clf__nu': 0.2, 'clf__shrinking': True, 'clf__tol': 1e-05}
0.7324864864864866: {'clf__kernel': 'rbf', 'clf__nu': 0.2, 'clf__shrinking': True, 'clf__tol': 1e-06}
0.7324864864864866: {'clf__kernel': 'rbf', 'clf__nu': 0.2, 'clf__shrinking': True, 'clf__tol': 1e-07}
0.7324864864864866: {'clf__kernel': 'rbf', 'clf__nu': 0.2, 'clf__shrinking': False, 'clf__tol': 1e-05}
0.7324864864864866: {'clf__kernel': 'rbf', 'clf__nu': 0.2, 'clf__shrinking': False, 'clf__tol': 1e-06}
0.7324864864864866: {'clf__kernel': 'rbf', 'clf__nu': 0.2, 'clf__shrinking': False, 'clf__tol': 1e-07}
0.6864594594594594: {'clf__kernel': 'rbf', 'clf__nu': 0.5, 'clf__shrinking': True, 'clf__tol': 1e-05}
0.6864594594594594: {'clf__kernel': 'rbf', 'clf__nu': 0.5, 'clf__shrinking': True, 'clf__tol': 1e-06}
0.6864594594594594: {'clf__kernel': 'rbf', 'clf__nu': 0.5, 'clf__shrinking': True, 'clf__tol': 1e-07}
0.6864594594594594: {'clf__kernel': 'rbf', 'clf__nu': 0.5, 'clf__shrinking': False, 'clf__tol': 1e-05}
0.6864594594594594: {'clf__kernel': 'rbf', 'clf__nu': 0.5, 'clf__shrinking': False, 'clf__tol': 1e-06}
0.6864594594594594: {'clf__kernel': 'rbf', 'clf__nu': 0.5, 'clf__shrinking': False, 'clf__tol': 1e-07}
```

Figure 16. Exhaustive Search of OneClassSVM Hyperparameters

OneClassSVM with linear kernel performs the best but with radial basis function the results are very close. Smaller values for the upper bound of training errors and lower bound of support vectors gives better results. Also, it seems the selection of stopping criterion has not much effect. It is good to note that the results on Figure 16 are not cross-validated.

The final experiment tries to find if removing CPE names from datasets gives somewhat better results as shown in Table 18. The results indicate that AUC-score is higher if CPE names are included in both OneClassSVM and the keyword-based classifier.

Table 18. Effect of CPE Names Removal on Classification AUC-score

CPE Name	Included	Excluded
OneClassSVM	**0,729**	0,717
KeywordStemClassifier	**0,732**	0,718

7.2 CVSS Scoring and CWE Classification

Four different machine learning algorithms and techniques were compared classifying CVSS scores and CWE categories. The algorithms are suggested by Scikit-learn for text classification. In Table 19 the CVSS2 classification F1-scores are presented with different vectorizers. The corresponding results for CVSS3 are presented in Table 20. In Table 21 the corresponding results are presented for CWE Classification.

Table 19. CVSS2 Score Classification with Vectorizers

	MultinomialNB	SGDClassifier	LinearSVC	KNeighbors
CountVectorizer	0,75	0,803	0,808	0,73
StemmedCount	0,748	0,803	0,807	0,724
LemmaCount	0,749	0,802	0,810	0,716
TFIDFVectorizer	0,677	0,804	**0,817**	0,718
StemmedTFIDF	0,675	0,803	0,815	0,715
LemmaTFIDF	0,678	0,805	**0,817**	0,716
Time w. TFIDF	16,5s	15,4s	17,5s	30,5s

Table 20. CVSS3 Score Classification with Vectorizers

	MultinomialNB	SGDClassifier	LinearSVC	KNeighbors
CountVectorizer	0,702	0,782	0,792	0,672
StemmedCount	0,7	0,784	0,792	0,67
LemmaCount	0,704	0,784	0,794	0,671
TFIDFVectorizer	0,532	0,77	0,795	0,679
StemmedTFIDF	0,527	0,771	0,795	0,678
LemmaTFIDF	0,533	0,771	**0,796**	0,678
Time w. TFIDF	19s	19,6s	21,8s	40,9s

Table 21. CWE Classification with Vectorizers

	MultinomialNB	**SGDClassifier**	**LinearSVC**	**KNeighbors**
CountVectorizer	0,665	0,796	0,811	0,569
StemmedCount	0,653	0,791	0,808	0,556
LemmaCount	0,665	0,796	0,809	0,574
TFIDFVectorizer	0,541	0,804	**0,812**	0,642
StemmedTFIDF	0,529	0,802	0,811	0,631
LemmaTFIDF	0,544	0,805	0,811	0,643
Time w. TFIDF	48,4s	74,9s	137,8s	56,3s

The results show that the LinearSVC classifier with TFIDF weighting gives the best performance and no remarkable improvement is achieved by stemming or lemmatizing in CVSS2, CVSS3, and CWE classification tasks. Scikit-learn provides HashingVectorizer to create in-memory mapping from string tokens to integer feature indices but performance was considerably lower than CountVectorizers and TFIDFVectorizers. In Figure 17 the SGDClassifier and LinearSVC classifiers are compared with different dataset sizes.

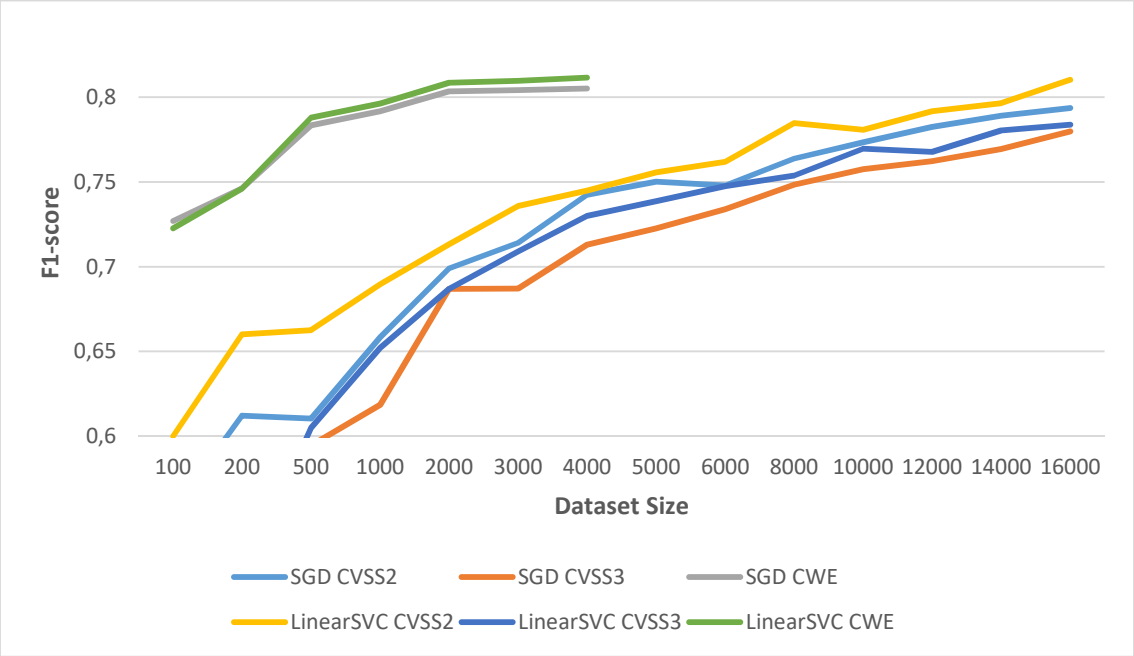

Figure 17. Classifier Performance with different Data sizes in CVSS2, CVSS3, and CWE Classification

In CWE classification F1-score reaches 0,8 at maximum of 2000 samples per class defined. In CVSS classification 0,8 F1-score is exceeded in CVSS2 classification with LinearSVC only. In Figure 17 the data size is the complete dataset size in CVSS experiments and resampled specifically to each class in CWE classification. Negative effect of undersampling can be seen in Figure 17 concerning CWE classification and in Figure 18 concerning CVSS classifications.

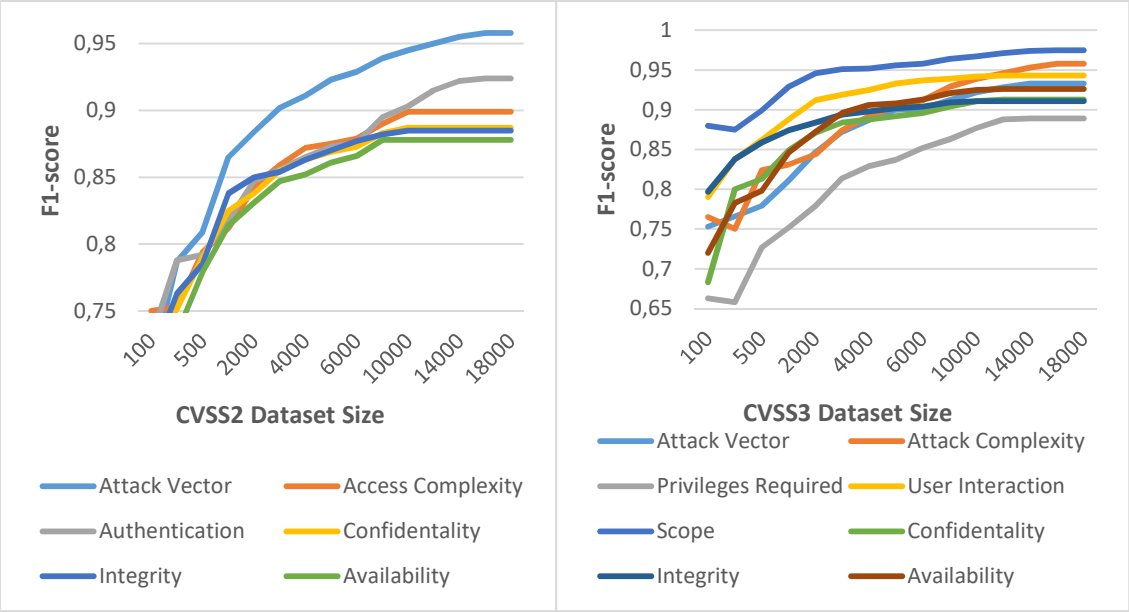

Figure 18. Negative Effect of Undersampling in CVSS2 and CVSS3 Metrics

As indicated in Figures 17 and 18, the undersampling strategy does not give better results. The F1-score varies between 0,87 – 0,975 but combining all metrics using severity categories the highest F1-score can be obtained in CVSS2, having the score 0,81 and in CVSS3 0,783.

As the LinearSVC classifier with a plain TFIDF vectorizing is selected for further experiments the different N-grams ranges were observed. In Table 22 there are F1-scores of CVSS and CWE classifications with different N-gram ranges presented.

Table 22. CVSS and CWE Classification Performance with N-gram Ranges

N-grams	CVSS2	CVSS3	CWE	Feature Count	Time
1 – 1	0,795	0,774	0,796	20059	30,8s
1 – 2	0,814	0,792	**0,813**	135644	62,2s
1 – 3	**0,817**	**0,795**	0,812	293891	105,1s
2 – 2	0,805	0,772	0,787	114286	65,5s
2 – 3	0,804	0,762	0,78	273832	129,2s
3 – 3	0,787	0,723	0,71	159546	115,4s

Very similar results in F1-score can be obtained with 1-2 grams than 1-3 grams but number of features grows higher using wider N-gram ranges. To see the effect of the minimum document frequency in multiclass classification the experiments were made running the df_min parameter values from 1 to 50. In Table 23 the F1-scores with feature counts and elapsed times are shown in CVSS2, CVSS3 and CWE classification. The df_min parameter is explained in Table 8 among the other vectorizer parameters.

Table 23. Effect of the min_df Parameter in Multiclass Classification

min_df	CVSS2	CVSS3	CWE	Feature Count	Time
1	**0,816**	**0,797**	**0,814**	135136	70,4s
2	0,815	**0,797**	0,811	42383	52,3s
3	0,811	0,791	0,81	24060	46,3s
4	0,807	0,789	0,809	16299	44,4s
5	0,804	0,786	0,806	12478	42,1s
10	0,791	0,775	0,803	6277	40,9s
20	0,781	0,775	0,796	3340	39,5s
30	0,777	0,758	0,792	2376	38,6s
40	0,772	0,75	0,786	1858	38,2s
50	0,768	0,748	0,784	1561	38,4s

The best performance in F1-score is gained with lower values of the min_df parameter. However, there are no major effects on performance with higher values of minimum document frequency but the feature count decreases considerably. The processing time also proves a slight, but not considerable decrease. In larger datasets by reducing amount of features it may consume less CPU power.

The final experiment was to test if removing the common platform enumeration names gives better results in multiclass classification. The F1-score results are shown in Table 24 for CVSS2, CVSS3 and CWE classification.

Table 24. Effect of CPE Names Removal on Multiclass Classification F1-score

CPE Names	Included	Excluded
CVSS2	**0,818**	0,812
CVSS3	**0,798**	0,791
CWE	**0,814**	0,813

The results indicate a slightly better but insignificant performance increase in F1-score if CPE names are not removed during the process of removing stop words. The results are very similar in unary and in multiclass classification.

Hyperparameter tuning was carried out for LinearSVC using GridSearchCV but the classifier seemed to achieve the best performance with default settings. Hyperparameter settings can be observed more detailed in Appendix 1.

8 CONCLUSION

Machine learning and natural language techniques were experimented in unary and multiclass classification. In unary classification the one-class classification and anomaly detection algorithms were experimented. Algorithms were selected as suggested by Scikit-learn. The library provides vectorizers to convert word tokens to numeric matrices. Vectorizers and algorithm hyperparameters combined have lots of options and in this work the options were examined thoroughly. The relatively new NVD vulnerability data was used as a learning data of one-class classifiers and also as datasets of multiclass classification experiments. A keyword-based classifier was implemented to compare a non-machine learning solution to the one-class classifiers. Defect databases were used to test the classifiers. Security related defects were manually labelled by a security specialist to verify that the defects truly contained some security related words. The goal of unary classification was to detect potential vulnerabilities from short text descriptions. Multiclass classification experiments were divided predicting CVSS2 and CVSS3 classes and also CWE weaknesses. The CVSS scores consist of exploitability and impact metrics which form distinguishable classes to calculate the actual CVSS score. The scores were mapped in four severity levels to compare the final classification performance. There are hundreds of different CWE classes which describe root causes of weakness types. AUC-score was selected to measure unary classification and micro averaged F1-score was selected to measure multiclassification performance. All the experiments were verified using the 10-fold cross-validation method or its variant.

The OneClassSVM classifier clearly outperforms LocalOutlierFactor and IsolationForest. The radial basis function works as well as the linear kernel and TFIDF weighting works evidently better with OneClassSVM. Vulnerabilities can be detected from text using a keyword-based solution as well. The keyword-based classifier works the best using 2-grams of vulnerability keywords with stemming. AUC-score in unary classification achieved 0,73 at highest. Some unary classification experiments with different datasets can be inspected more detailed in Appendix 2. The results show that vulnerability detection from text is difficult and the results are far from perfect. In many experiments, generation of false alarms arose very high. The classifiers gain a stable level using 4000 to 6000 vulnerability descriptions as learning data.

In multiclass classification the LinearSVC classifier beats the other tested classifiers: Multinomial Naïve Bayes, Stochastic Gradient Descent on top of linear methods, and Nearest Neighbors. The classifier with TFIDF weighting gives a slightly better results than a plain bag-of-words representation. The challenge is that the source data has high class imbalance and effective undersampling could not be achieved. The results show that classifiers work better using the complete dataset without applying a sampling strategy. F1-score got 0,82 in CVSS2 and CWE classification but in CVSS3 the score goes slightly under. Han et al. (2017) got F1-score 0,816 ± 0,052 in their study classifying CVSS2 scores using the word

embedding + 1-layer CNN techniques. In their study the negative effect of undersampling is similar to this work. They found that increasing training data improves the model performance in general but has larger impact on smaller training dataset, and the performance improvement becomes smaller and slower as the training dataset becomes larger. In CVSS classification the challenge is multiple metrics to classify. In CVSS3 there are 8 metrics to classify which all must have high performance to get effective results. Some CVSS classification experiments with confusion matrices can be found from Appendix 3. The challenge classifying CWE is high number of distinguishable classes.

In both, vulnerability detection and multiclass classification the stemming and lemmatization did not have remarkable effect on performance. The techniques are to reduce inflectional variations in words which reduces the number of features. By reducing amount of features it may consume less CPU power. Finally, to make the classifiers more generic by removing CPE names from the datasets was attempted. A slightly better performance was gained by leaving the names within the data. It is possible that thousands of vendor and component names are generic in nature which should be left among the data to gain better class distinction.

All the experiments presented in this paper are published in Github: https://github.com/oz-ds/textvulns.git. Jupyter Notebook implementations for keyword-based classifier, one-class classifier experiments, Common Vulnerability Scoring System, and Common Weakness Enumeration experiments can be found from the repository.

8.1 Discussion

This work aimed to study techniques towards a tool to predict vulnerabilities and their severities from text. The results show that security specialists might benefit from this kind of tool which is using the techniques presented. The tool could give some indications whether a text contains words that are used in vulnerabilities in general, estimation about CVSS score to help prioritization of defects, and also estimation of CWE weakness class. Estimated CWE might further help in prioritization and defect categorization. The experimented techniques can be further developed and better vulnerability detection mechanisms should be researched. As in any empirical study the experiments may suffer validity issues. To mitigate these issues, cross-validation mechanisms are applied. However, low number of bug report databases may threaten validity of vulnerability detection measurements. Another mitigation task was manual labelling of bug reports by security specialist to keep mislabeling in the training data as low as possible.

As the goal of this thesis was more technical, some side-notes can be made based on the NVD data which is presented on Chapter 6. In the CVSS classes there are high class imbalance in many metrics. For example, the most vulnerabilities are exploitable from network and low number of vulnerabilities

are exploitable from adjacent network. Some classes were completely discarded from the data due to low number of observations. The CVSS2 Multiple Authentication metric's vulnerabilities exists only four times on the entire dataset. This metric is changed to Privileges Required in CVSS3. Low number of real world observations in CVSS metrics might indicate need for changes on future versions of CVSS. High class imbalance concerns the CWE data as well. Table 10 lists the bad quality CWEs which were discarded from the experiments. Many of these CWEs are deprecated or suggested for deprecation on upcoming versions.

8.2 Future Work

In this thesis, traditional machine learning approaches were studied and better performance might be obtained by text enrichment and augmentation techniques. Text enrichment involves augmenting original text data with information that it did not previously have. One enrichment technique is part-of-speech tagging where the tags can be used as machine learning features or take additional preprocessing steps concerning to a particular part of speech, for example, take different actions for verbs and nouns. Such techniques were not studied in this work. Also, text normalization could lead to better results. It is a preprocessing technique which goal is to convert word abbreviations and misspellings into normalized form, for example, words "2mrrw" and "tomrw" can be converted to "tomorrow". According to Satapathy et al. (2017) study sentiment classification improved by 4% by applying normalization. The Scikit-learn user guide (2019) provides another interesting approach handling misspellings and derivations by building features using character N-grams. For example, dealing with a corpus of two documents: "words", "wprds". The second document contains a misspelling of the word "words". Usually these are considered two very distinct documents but a character 2-gram representation would find the documents matching in 4 out of 8 features, which may help the preferred classifier decide better. Suitable preprocessing steps are very domain and goal specific and general rule of thumb guidelines cannot be easily defined.

To improve unary classification performance, newer neural network based approaches could be attempted. Perera et al. (2018) developed a novel deep learning based solution for one-class classification. They achieved AUC-score of 0,99 using their method compared to one-class svm having AUC-score only 0,606 detecting American flag images on the Caltech256 dataset. To extend document classification on very large datasets Joulin et al. (2016) developed a new classifier (*fastText*) on sentiment analysis and tag prediction of text using the YFCC100M dataset. They concluded that deep neural networks have in theory much higher representational power than shallow models but it is not clear if simple text classification problems are the right ones to evaluate them.

REFERENCES

Arlot, S., Celisse, A. (2010). *A Survey of cross-validation procedures for model selection*. Statistics surveys 4: 40-79.

Arnold, J., Abbott, T., Elhage, N., Thomas, G. and Kaseorg, A. (2009). *Security impact ratings considered harmful.* 12th workshop on Hot Topics in Operating Systems. USENIX. http://web.mit.edu/tabbott/www/papers/hotos.pdf

Bellinger, C., Sharma, S., Japkowicz, N. (2012). *One-Class versus Binary classification: Which and When?*. 11th International Conference on Machine Learning and Applications. IEEE Computer Society. https://doi.org/10.1109/ICMLA.2012.212

Bozorgi, M., Saul, L., Savage, S., Voelker, G. (2010). *Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits.* Department of Computer Science and Engineering, University of California, San Diego. http://cseweb.ucsd.edu/~saul/papers/kdd10_exploit.pdf

Breunig, M., Kriegel, H., Ng, R., Sander, J. (2000). *LOF: Identifying Density-Based Local Outliers*. ACM SIGMOD International Conference on Management of Data. DOI: 10.1145/342009.335388.

Camacho-Collados, J., Pilehvar, M. (2018). *On the Role of Text Preprocessing in Neural Network Architectures: An Evaluation Study on Text Categorization and Sentiment Analysis*. https://arxiv.org/pdf/1707.01780.pdf

CVSS2. (2007). *A Complete Guide to the Common Vulnerability Scoring System Version 2.0.* Forum of Incident Response and Security Teams. https://www.first.org/cvss/cvss-v2-guide.pdf

CVSS3. (2015). *Common Vulnerability Scoring System v3.0: Specification Document.* Forum of Incident Response and Security Teams. https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf

Han, Z., Li, X., Xing, Z., Liu, H., Feng, Z. (2017). Learning to Predict Severity of Software Vulnerability Using Only Vulnerability Description. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). https://doi.org/10.1109/ICSME.2017.52

Hand, D. (2009). *Measuring classifier performance: a coherent alternative to the area under the ROC curve*. Springer Science+Business Media, LLC

Joachims, T. (1996). *A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization*. Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997

Joulin, A., Grave, E., Bojanowski, P., Mikolov, T. (2016). *Bag of Tricks for Efficient Text Classification*, Facebook AI Research. arXiv:1607.01759. https://arxiv.org/pdf/1607.01759

Jurman, G., Riccadonna, S., Furlanello, C. (2012). *A Comparison of MCC and CEN Error Measures in Multi-Class Prediction.* PLoS ONE 7(8): e41882. doi:10.1371/journal.pone.0041882

Kantardzic, M. (2011). *Data mining: Concepts, models, and algorithms.* John Wiley & Sons. IEEE Press. ISBN: 978-1-118-02912-1.

Kohavi, R. (1995). *A Study of cross-validation and bootstrap for accuracy estimation and model selection.* Proceedings of the 14th international joint conference on artificial intelligence - Volume 2, 1137-1143. IJCAI'95. Montreal, Quebec, Kanada: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-363-8.

Lamkanfi, A., Demeyer, S., Soetens, Q. D., and Verdonck, T. (2011). *Comparing mining algorithms for predicting the severity of a reported bug.* Software Maintenance and Reengineering (CSMR).

Li, Z., Zou, D., Xu, S., Ou, X. Jin, H., Wang, S., Deng, Z., Zhong, Y. (2018). *VulDeePecker: A Depp Learning-Based System for Vulnerability Detection.* Network and Distributed Systems Security (NDSS) Symposium 2018. 18-21 February 2018, San Diego, CA, USA. ISBN 1-1891562-49-5.

Liu, F., Ting, K., Zhou, Z. (2012). *Isolation-based Anomaly Detection.* ACM Transactions on Knowledge Discovery from Data 6(1):1-39. DOI: 10.1145/2133360.2133363.

Manning, C., Surdeanu, M., Bauer, J., Finkel, J. Bethard, S. McClosky, D. (2014). *The Stanford CoreNLP Natural Language Processing Toolkit*. Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 55–60. Association for Computational Linguistics

Miyamoto, D., Yamamoto, Y., Nakayama, M. (2015). *Text-Mining Approach for Estimating Vulnerability Score.* Conference: 2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS).

Nothman, J., Hanmin, Q., Yurchak, R. (2018). *Stop Word Lists in Frr Open-source Software Packages.* Proceedings of Workshop for NLP Open Source Software, pages 7–12. Association for Computational Linguistics.

Peters, P., Tun, T., Yu, Y., Nuseibeh, B. (2017). *Text Filtering and Ranking for Security Bug Report Prediction*. IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2017.2787653

Perera, P., Patel, V.M. (2018). *Learning Deep Features for One-Class Classification.* arXiv:1801.05365. https://arxiv.org/pdf/1801.05365

Porter, M. (1980). *An algorithm for suffix stripping*. Program 14.3 (1980): 130-137.

Ruohonen, J., Leppänen, V. (2018). *Toward Validation of Textual Information Retrieval Techniques for Software Weaknesses*. Proceedings of the 29th International Conference on Database and Expert Systems Applications (DEXA 2018), Regensburg, Springer, pp.~265—277. https://arxiv.org/abs/1809.01360

Sanguino, L. A. B., Uetz, R. (2017). *Software Vulnerability Analysis Using CPE and CVE.* arXiv:1705.05347. https://arxiv.org/abs/1705.05347

Satapathy, R., Guerreiro, C., Chaturvedi, I., Cambria, E. (2017). *Phonetic-Based Microtext Normalization for Twitter Sentiment Analysis*. 2017 IEEE International Conference on Data Mining Workshops

Schölkopf, B., Platt, J., Shawe-Taylor, J., Smola, A., Williamson, R. (2001). *Estimating the Support of a High-Dimensional Distribution.* Neural Computation. Volume 13 Issue 7, Pages 1443 – 1471. MIT Press Cambridge, MA, USA

Scikit, (2019, May 27). *Scikit-learn user guide. Release 0.21.2. Scikit-learn developers*. https://scikit-learn.org/stable/user_guide.html

Gates, C., Taylor, C. (2007). *Challenging the anomaly detection paradigm: A provocative discussion.* Proceedings of the 2006 workshop on new security paradigms, 21-29. NSPW '06. Germany: ACM

Tyo, J. P. (2016). *Empirical Analysis and Automated Classification of Security Bug Reports.* Lane Department of Computer Science and Electrical Engineering. Morgantown, West Virginia. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20160014477.pdf

Wang, S., Liu, Q., Zhu, E., Porikli, F., Yin, J. (2017). *Hyperparameter selection of one-class support vector machine by self-adaptive data shifting*. Pattern Recognition Volume 74, Pages 198-211. Elsevier Ltd. http://dx.doi.org/10.1016/j.patcog.2017.09.012

Wijayasekara, D., Manic, M., Wright, J., McQueen, M. (2012). *Mining Bug Databases for Unidentified Software Vulnerabilities*. 5th International Conference on Human System Interactions. https://doi.org/10.1109/HSI.2012.22

Wijayasekara, D., Manic, M., McQueen, M. (2014). *Vulnerability identification and classification via text mining bug databases.* IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society pp. 3612-3618.

Witten, I. H., Frank, E. (2005). *Data mining: Practical machine learning tools and techniques.* Morgan Kaufmann Series in data management systems. Morgan Kaufmann. ISBN: 9780120884070.

Wright, J., Larsen, J., McQueen, M. (2013). *Estimating Software Vulnerabilities: A Case Study Based on the Misclassification of Bugs in MySQL Server.* Idaho National Laboratory.

Yang, Y. (1998). *An Evaluation of Statistical Approaches to Text Categorization.* INRT Journal.

APPENDIX 1: SOURCE CODE

```
#
# Class to find parent and root cwe
#
# Example:
# finder = CweFinder()
# root_cwe = finder.find_root_cwe(123)
#
import pandas as pd
class CweFinder():
    def __init__(self):
        # CSV file available at https://cwe.mitre.org/data/csv/1000.csv.zip
        self.cwes = pd.read_csv('data/1000.csv', index_col=False)
        self.cwes['CWE-ID'] = self.cwes['CWE-ID'].values.astype(str)

    def find_parent_cwe(self, cwe_id):# method returns the parent cwe of given cwe
        cwe = self.cwes[self.cwes['CWE-ID'] == cwe_id]
        if(cwe.empty):
            return ''
        cwe = cwe.iloc[0]['Related Weaknesses']
        if type(cwe) != str:
            return ''
        s = cwe.find('ChildOf:CWE ID:')#15 characters long
        e = cwe.find(':', s+15)
        if s == -1 or e == -1:
            return ''
        return cwe[s+15:e]

    def find_root_cwe(self, cwe_id):# method return the root cwe of given cwe
        parent = self.find_parent_cwe(cwe_id)
        if len(parent) <= 0:
            return cwe_id
        else:
            return self.find_root_cwe(parent)


#
# LinearSVC hyperparameter tuning
#
from sklearn.model_selection import GridSearchCV
ngram_s = 1
ngram_e = 2
df = 1
t = r'(?u)\b\w*[a-zA-Z]{3,}\w*\b'
vectorizer =
    TfidfVectorizer(stop_words=swds, ngram_range=(ngram_s, ngram_e), min_df=df, token_pattern=t)

classifier = LinearSVC()
pipe = Pipeline([('vect', vectorizer), ('cls', classifier)])

#hyperparameters
parameters = {'cls__loss': ('hinge', 'squared_hinge'),#default: squared_hinge
              'cls__dual': (True, False),#default: True
              'cls__multi_class': ('ovr', 'crammer_singer'),#default: ovr
              'cls__max_iter': (1000, 2000),#default: 1000
             }
gs = GridSearchCV(pipe, parameters, scoring='f1_micro', cv=10, error_score=np.nan)

for i in range(6):# 6 classes on cvss2 metrics
    gs = gs.fit(cvss2_texts[i]['text'], cvss2_texts[i]['label'])
    print(str(i)+':'+str(gs.best_score_))
    print(str(i)+':'+str(gs.best_params_))

#
# Vulnerability detection classifiers and vectorizers
#
def run_test(vzr, cls):
    print(str(cls)[0:str(cls).find('(')] + ' ' + str(vzr)[0:str(vzr).find('(')])

    #get a new testing set
    mixed = vulns_common.get_mixed_dataset(reports['report'], 1000)

    scores = []
    t = time.time()
    for i in range(nfold): #n-fold cross val score
        predicted = pipe.predict(mixed['report'])
        #score = f1_score(y_true=mixed['security'], y_pred=predicted, average='micro')
        score = roc_auc_score(y_true=mixed['security'], y_score=predicted, average='micro')
        scores.append(score)

    scores = np.array(scores)
```

```
    print(str(nfold)+'-fold cross-validated roc-auc-score:' + str(scores.mean()) + '\n')
    print('Time taken: ' + str(round(time.time() - t, 1)) + 's')

for c in classifiers:
    for v in vectorizers:
        pipe = Pipeline([('vect', v), ('clf', c)])
        #get a new learning data before the fit method
        vuln_data = shuffle(nvd_vulns, n_samples=6000)
        vuln_descs = []
        for d in vuln_data:
            if not d[1].startswith('** REJECT'):#some descriptions are rejected by NVD
                vuln_descs.append(d[1])
        print('Dataset to fit:'+str(len(vuln_descs)))
        pipe = pipe.fit(vuln_descs)
        run_test(v, c)

#
# Keyword-based Classifier with stemming
#
class StemmedCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        self.stemmer = SnowballStemmer("english")
        analyzer = super(CountVectorizer, self).build_analyzer()

        return lambda doc: (analyzer(' '.join([self.stemmer.stem(word) for word in doc.split(' ')])))

class KeywordStemClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, min_ngrams=2, max_ngrams=2):
        self.min_ngrams = min_ngrams
        self.max_ngrams = max_ngrams
        unwanted_words                                                                  =
['issue','defect','bug','fault','flaw','mistake','error','version','system','because','before','dispu
ted']
        stop_words = text.ENGLISH_STOP_WORDS#.union(cpe_names)
        stop_words = stop_words.union(unwanted_words)
        self.vectorizer = StemmedCountVectorizer(stop_words=stop_words,
                                                 lowercase=True,
                                                 ngram_range=(min_ngrams, max_ngrams),
                                                 min_df=1,
                                                 token_pattern=r'(?u)\b\w*[a-zA-Z]{3,}\w*\b')

    def fit(self, raw_documents, y=None):
        self.vectorizer.fit(raw_documents)

        return self

    def predict(self, raw_documents, y=None):
        assert (len(self.vectorizer.vocabulary_) > 0), "You must call fit() before predicting data!"
        scores = self.score(raw_documents)

        predictions = []
        for count in scores:
            if count >= 2:#at least two vulnerability n-grams classified as security related
                predictions.append(1)
            else:
                predictions.append(-1)

        return np.array(predictions)

    def word_grams(self, words, min, max):
        s = []
        for n in range(min, max+1):
            for ngram in ngrams(words, n):
                s.append(' '.join(str(i) for i in ngram))
        return s

    def _score_single(self, tokens):
        count= 0
        for index, token in enumerate(tokens):
            if token in self.vectorizer.vocabulary_:
                count = count + 1
        return count

    def score(self, raw_documents, y=None):
        assert (len(self.vectorizer.vocabulary_) > 0), "You must call fit() before scoring data!"
        stemmer = SnowballStemmer("english")
        scores = []
        for index, row in enumerate(raw_documents):
            stems = []
            for word in row.split():
                stems.append(stemmer.stem(word))
            tokens = self.word_grams(stems, self.min_ngrams, self.max_ngrams)
            count = self._score_single(tokens)
            scores.append(count)

        return np.array(scores)
```

APPENDIX 2: VULNERABILITY DETECTION EXPERIMENTS

Keyword-based Classifier

```
Manually labeled vulnerability dataset: 148
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
           1       1.00      0.66      0.80       148

   micro avg       0.66      0.66      0.66       148
   macro avg       0.50      0.33      0.40       148
weighted avg       1.00      0.66      0.80       148

TN=0, FP=0, FN=50, TP=98
Vulnerability learning dataset:3979
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
           1       1.00      0.95      0.97      3979

   micro avg       0.95      0.95      0.95      3979
   macro avg       0.50      0.47      0.49      3979
weighted avg       1.00      0.95      0.97      3979

TN=0, FP=0, FN=211, TP=3768
Test on data/Ambari.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.98      0.93      0.95       971
        True       0.09      0.24      0.13        29

   micro avg       0.91      0.91      0.91      1000
   macro avg       0.53      0.59      0.54      1000
weighted avg       0.95      0.91      0.93      1000

TN=902, FP=69, FN=22, TP=7
Test on data/Ambari.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.98      0.82      0.89       971
        True       0.06      0.41      0.11        29

   micro avg       0.81      0.81      0.81      1000
   macro avg       0.52      0.62      0.50      1000
weighted avg       0.95      0.81      0.87      1000

TN=797, FP=174, FN=17, TP=12
Test on data/Camel.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.97      0.79      0.87       967
        True       0.04      0.28      0.07        32

   micro avg       0.77      0.77      0.77       999
   macro avg       0.51      0.54      0.47       999
weighted avg       0.94      0.77      0.84       999

TN=763, FP=204, FN=23, TP=9
Test on data/Wicket.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.99      0.78      0.87       990
        True       0.01      0.30      0.03        10

   micro avg       0.78      0.78      0.78      1000
   macro avg       0.50      0.54      0.45      1000
weighted avg       0.98      0.78      0.86      1000

TN=773, FP=217, FN=7, TP=3
Test on data/Chromium.csv. Rows: 1000
              precision    recall  f1-score   support

       False       1.00      0.73      0.85       996
        True       0.00      0.25      0.01         4

   micro avg       0.73      0.73      0.73      1000
   macro avg       0.50      0.49      0.43      1000
weighted avg       0.99      0.73      0.84      1000

TN=732, FP=264, FN=3, TP=1
Test on data/Derby.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.94      0.70      0.80       910
        True       0.15      0.52      0.23        88

   micro avg       0.69      0.69      0.69       998
   macro avg       0.54      0.61      0.52       998
weighted avg       0.87      0.69      0.75       998

TN=639, FP=271, FN=42, TP=46
```

OneClassSVM Classifier

```
Manually labeled vulnerability dataset: 148
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
           1       1.00      0.73      0.84       148

   micro avg       0.73      0.73      0.73       148
   macro avg       0.50      0.36      0.42       148
weighted avg       1.00      0.73      0.84       148

TN=0, FP=0, FN=40, TP=108
Vulnerability learning dataset:5967
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
           1       1.00      0.79      0.88      5967

   micro avg       0.79      0.79      0.79      5967
   macro avg       0.50      0.39      0.44      5967
weighted avg       1.00      0.79      0.88      5967

TN=0, FP=0, FN=1281, TP=4686
Mixed dataset: 1148
              precision    recall  f1-score   support

          -1       0.95      0.71      0.81      1000
           1       0.27      0.73      0.39       148

   micro avg       0.71      0.71      0.71      1148
   macro avg       0.61      0.72      0.60      1148
weighted avg       0.86      0.71      0.76      1148

TN=706, FP=294, FN=40, TP=108
Test on data/Ambari.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.97      0.94      0.95       971
        True       0.05      0.10      0.06        29

   micro avg       0.91      0.91      0.91      1000
   macro avg       0.51      0.52      0.51      1000
weighted avg       0.95      0.91      0.93      1000

TN=910, FP=61, FN=26, TP=3
Test on data/Camel.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.97      0.91      0.94       967
        True       0.09      0.28      0.14        32

   micro avg       0.89      0.89      0.89       999
   macro avg       0.53      0.59      0.54       999
weighted avg       0.95      0.89      0.91       999

TN=878, FP=89, FN=23, TP=9
Test on data/Wicket.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.99      0.82      0.90       990
        True       0.02      0.30      0.03        10

   micro avg       0.81      0.81      0.81      1000
   macro avg       0.50      0.56      0.46      1000
weighted avg       0.98      0.81      0.89      1000

TN=811, FP=179, FN=7, TP=3
Test on data/Chromium.csv. Rows: 1000
              precision    recall  f1-score   support

       False       1.00      0.92      0.96       997
        True       0.03      0.67      0.05         3

   micro avg       0.92      0.92      0.92      1000
   macro avg       0.51      0.80      0.50      1000
weighted avg       1.00      0.92      0.96      1000

TN=921, FP=76, FN=1, TP=2
Test on data/Derby.csv. Rows: 1000
              precision    recall  f1-score   support

       False       0.92      0.90      0.91       910
        True       0.14      0.17      0.15        88

   micro avg       0.84      0.84      0.84       998
   macro avg       0.53      0.54      0.53       998
weighted avg       0.85      0.84      0.84       998

TN=819, FP=91, FN=73, TP=15
```

APPENDIX 3: CVSS CLASSIFICATION EXPERIMENTS

CVSS2

Training data:13593 Testing data:4531

	precision	recall	f1-score	support
critical	0.72	0.72	0.72	325
high	0.68	0.71	0.70	728
low	0.76	0.66	0.71	533
medium	0.87	0.89	0.88	2945
micro avg	0.82	0.82	0.82	4531
macro avg	0.76	0.74	0.75	4531
weighted avg	0.82	0.82	0.82	4531

Confusion matrix:

```
[[ 235   39    2   49]
 [  47  517   13  151]
 [   1    5  352  175]
 [  45  194   96 2610]]
```

CVSS3

Training data:13446 Testing data:4482

	precision	recall	f1-score	support
critical	0.60	0.83	0.70	645
high	0.81	0.81	0.81	2061
low	0.51	0.40	0.45	50
medium	0.90	0.78	0.84	1726
micro avg	0.80	0.80	0.80	4482
macro avg	0.71	0.70	0.70	4482
weighted avg	0.81	0.80	0.80	4482

Confusion matrix:

```
[[ 537  101    1    6]
 [ 274 1663    8  116]
 [   3    6   20   21]
 [  81  289   10 1346]]
```