A Machine Learning Approach for Vulnerability Curation

Yang Chen
Veracode
ychen@veracode.com

Andrew E. Santosa
Veracode
asantosa@veracode.com

Ang Ming Yi
Veracode
mang@veracode.com

Asankhaya Sharma
Veracode
asharma@veracode.com

David Lo
Singapore Management University
davidlo@smu.edu.sg

ABSTRACT

In this article, we report our design and implementation of a system based on machine learning to support the identification of vulnerable open-source software. Our system uses publicly-available data sources including Jira tickets, Bugzilla bugs, Github issues and pull requests, Github commits, mailing lists and reserved CVEs. The prediction results of our machine learning models are used to suggest to a team of security researchers on whether each data item is related to vulnerability or not. Our system provides a complete pipeline from data collection, model training and prediction, to the validation of new models before deployment. The system is executed iteratively in production to generate better models as new input data becomes available. Our system is also enhanced by a self-training process that significantly and automatically increases the size of input dataset used in training new models, opportunistically maximizing the improvement in the quality of the generated models at each iteration. We present evaluation results to illustrate the improvement in the performance of the generated models in one iteration and the effectiveness of self training.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

application security, open source software, machine learning, classifiers ensemble, self training

ACM Reference Format:
Yang Chen, Andrew E. Santosa, Ang Ming Yi, Asankhaya Sharma, and David Lo. 2018. A Machine Learning Approach for Vulnerability Curation. In *ESEC/FSE '19: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 26–30 August, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

As with any software, open-source libraries may contain vulnerabilities. *Software composition analysis* (*SCA*) analyzes the dependencies of an application to list those that are vulnerable, with commercial offerings from various companies [3, 11, 20, 21]. SCA systems employ publicly-available databases such as the *National Vulnerability Database* (*NVD*) to identify vulnerable dependencies of an application. Each vulnerability recorded in the NVD is identified using a *Common Vulnerabilities and Exposure* (*CVE*) id. More often than not, however, the vulnerabilities are reported and/or fixed via various other media, such as issue reports, pull requests, or commit messages, without being identified by NVD. According to SourceClear data [18], close to 50% of vulnerabilities in open-source software are not disclosed publicly with a CVE.

In this article, we report our design and implementation of a production system based on machine learning to support the manual identification of vulnerable open-source software in the context of SCA. As an input to our machine learning system, we use publicly-available data sources, including tickets, bug and issue reports, pull requests, commit data, security-related mailing lists, and a vulnerability database. We use the prediction results of the machine learning system to provide a suggestion to a team of security researchers on whether a data item is related to a vulnerability or not. A *vulnerability-related* item includes discussions on vulnerabilities. For pull requests and commits, it is typically an item that discusses the patching of vulnerabilities. The final labeling of each data item as vulnerability-related or not is done manually by the security researchers. Our system provides a complete pipeline from data collection, model training and prediction, to the validation of new models before deployment.

Our data collection engine collects data from different data sources related to open source projects, by calling the API made available by the data sources. Our data sources include Jira tickets, Bugzilla bugs, Github issues and pull requests, Github commits, mailing lists, and *reserved* CVEs from NVD. The Github commits are collected from the same set of Github projects where we collect the data for the issues and pull requests. A reserved CVE entry is a CVE entry that has been registered, but at present is not yet confirmed as a vulnerability because of incomplete details. It may later be confirmed as a proper CVE. We apply a filtering with regular expression to filter out irrelevant data, since their number is much more than the relevant ones, except for email data, as the ratio of imbalance for emails is low. The regular expression that we use here includes the terms related to computer security, such as *security*, *advisory*, *authorized*, *NVD*, etc.

To bootstrap our system, a team of security researchers initially review the data that pass our regular expression filter and label them as vulnerability-related or not. These initial manually-labeled data are used for training our bootstrap models. The models are then used to suggest to the security researchers the vulnerability relatedness of new input data to help their decision making. This process further produces new labeled data, which are then utilized to generate new and better models iteratively.

We mainly focus on the textual parts of each data item, and for each data source, we use word2vec [10] to perform word embedding. In the training process, we generate one model for each data source we mentioned previously, using a stacking ensemble of classifiers, which are reported to perform well for unbalanced datasets and natural language processing in the literature [28]. The basic classifiers that we use in our stacking ensemble include *random forest* (*RF*), *Gaussian Naive Bayes*, *k-nearest neighbor* (*k-NN*), *support vector machine* (*SVM*), *gradient boosting*, and *AdaBoost*. The output of the six classifiers is passed on to a logistic regression as the meta learner in the stacking ensemble. Given a data item, the model outputs *probability of vulnerability relatedness* (*PVR*), which is a value from 0 to 1. We apply a threshold on the PVR called *PVR threshold* to suggest if the data item is related to vulnerability or not. Given PVR threshold τ, when a PVR of a data item is strictly greater than τ, the system outputs a positive (vulnerability-related) recommendation, otherwise it outputs a negative (non-vulnerability-related) recommendation. The data items that are suggested to be vulnerable are passed on to a team of security researchers, who finally labels each data item as vulnerability-related or not.

As manual labeling is expensive, during each iteration, there is a large amount of data not manually labeled that are made up of two parts. One part is the set of data that fails regular expression filter during data collection and another part consists of the data that is automatically predicted as not related to vulnerability by the current production models. We make our system performs *semi-supervised* learning process; in particular, *self training*, where we re-apply our production models to the unlabeled data in order to significantly increase the amount of labeled data for training new models. For each unlabeled data item, we compute its PVR using the production models and use *high* and *low* thresholds to label them. If the PVR is higher than the high threshold, the item will be labeled as vulnerability-related, and if it is lower than the low threshold, the item will be labeled otherwise. We do not use any data item whose PVR is from the low to the high threshold.

When a suite of new models are trained, we perform the following two kinds of validation to ensure that they have better quality than the models already deployed in production:

- *Performance validation.* Security researchers and self training produce a combined labeled dataset for training new models. We randomly reserve 25% of these data for the purpose of testing the resulting model, and use only 75% of them to actually train the models themselves. After the models are generated, we apply them to the 25% testing dataset for prediction and compute their performance. We deploy a new model into production only when the new model has a better performance than the production model it is replacing.
- *Stability validation.* There is a risk that the prediction results of the new models are too widely different from the results of the production models. To mitigate this risk, we measure the *stability* of the prediction results of the new models by comparing them with the results of the models in production. Here we consider a new model to be stable if, its true positives and true negatives do not vary too much when compared with the production model. To measure this, we calculate our new metric called *deployment stability*, which is the sum of the numbers of the common true positives and common true negatives of the new and production models, divided by the total number of data items. For this measurement, we use a randomly-picked half of the labeled dataset with the same balance of positive and negative labels.

In summary, our contribution focuses on the design and implementation of a system which uses machine learning to support manual curation for identifying vulnerabilities in open-source software through freely-available data sources. Our system supports the whole pipeline of machine learning task, including data collection, training, prediction, and validation. We iterate this pipeline to improve the machine learning model continually. We enhance our system by a self-training process that significantly and automatically increases the size of labeled dataset used in training new models, opportunistically maximizing the improvement in the quality of the generated models at each iteration. We also introduce novel testing methodologies to compare new and production models. In addition, we package our production models as a REST service that provides predictions using submitted user data. In this article, we detail our pipeline, evaluate its performance for the first iteration as a case study, and experimentally demonstrate the effectiveness of self training for a data subset. Although our core learning engine uses the stacking ensemble of Zhou and Sharma [28], we advance further with building a complete system that also employs semi-supervised learning and uses a novel complete testing methodologies for model evaluation. In addition, we also expand the range of data with emails and reserved CVEs and with more commit features.

We provide an overview of our whole system in Section 2. We discuss the aspects of data collection in Section 3 including data sources and feature engineering, and we discuss model training in Section 4. We discuss the validation of our models in Section 5, and our prediction service in Section 6. We discuss the self-training technique that we use to improve the performance of our models at every iteration in Section 7, and provide an evaluation of our approach in Section 8. We present related work in Section 9, and conclude our paper in Section 10.

2 OVERVIEW

Our system provides a complete pipeline from data collection, feature engineering, training, prediction, to new model testing and deployment. It helps manual identification of vulnerabilities in open-source projects using data from Jira tickets, Bugzilla bug reports, etc. Figure 1 shows the data flow diagram (DFD) of our system. It is an iterative pipeline where we use our machine learning models to predict the labels of new data (B) to help security researchers to actually label them. The labeled data are then used to train new

models using supervised learning (C), after which the resulting models are validated against existing production models (D) to ensure that they are better. The newly-validated models are then deployed in production for predicting the labels of newer data (B), as well as being used in a self-training subsystem (E), which automatically labels residual data not manually labeled to increase the amount of labeled data used in the next iteration for training new models, opportunistically maximizing the performance improvement of the generated models at each iteration. We execute our pipeline iteratively to accumulate more labeled data and gradually improve our production models.

We collect data from six sources including Jira tickets, Bugzilla bugs, Github issues and pull requests, Github commits, emails from security-related mailing lists, and reserved CVEs from NVD (A). Here the data are highly unbalanced, where only a small proportion of the input data are vulnerability-related. To tackle this problem, we use a regular expression to filter the input data to obtain data which have relatively higher probability of being related to security vulnerability. The regular expression includes the terms related to computer security, such as security, advisory, authorized, NVD, etc.

To bootstrap the whole process, however, we need to create some initial models. To do this, a team of security researchers reviewed the filtered-in data and label them as vulnerability-related or not, denoted in Figure 1 using the dotted line (bootstrap 1), directly producing labeled data. On these labeled data, we used supervised learning to obtain the initial models (bootstrap 2), one for each data source. In the bootstrap phase, where there is no production model, we directly use these models in production (bootstrap 3).

In the second and further iterations, we use the production models to predict each data item as either vulnerability related or not (B). The security researchers review those data items that are predicted as vulnerability-related only and label them. Using this method, we miss some true vulnerability-related data due to the imprecision of the production models, but with using the prediction result, we reduce the manual review and labeling time. We then use the new labeled data to generate new models using the same learning approach (C). We note that our system has been gradually evolving over time, and at present we use a set of known-good fixed hyperparameters in the training process, such as we no longer reserve test data for hyperparameter tuning. Once there are new models, we validate them against the production models (D). The purpose of this validation step is to to ensure that the new models to be deployed actually perform better than the production models.

Manually labeling data is expensive, and hence there is a large amount of data that cannot be manually labeled. The first dataset consists of the data that are predicted non-vulnerability-related by the production models, and second dataset consists of the data that fail regular expression filtering. Their sizes can be much larger than the labeled data, particularly for commit data. Such is the situation where semi-supervised learning can be applied [12, 15, 27]. In particular, here we use a class of semi-supervised learning called self training to automatically label the data from both sources (E). For the first dataset, we already have the PVRs computed by the production models, whereas for the second dataset, we apply the production models to compute the PVRs. Different to prediction service (B), here we replace the PVR threshold with high and low thresholds, such that when the PVR output of the production model is higher

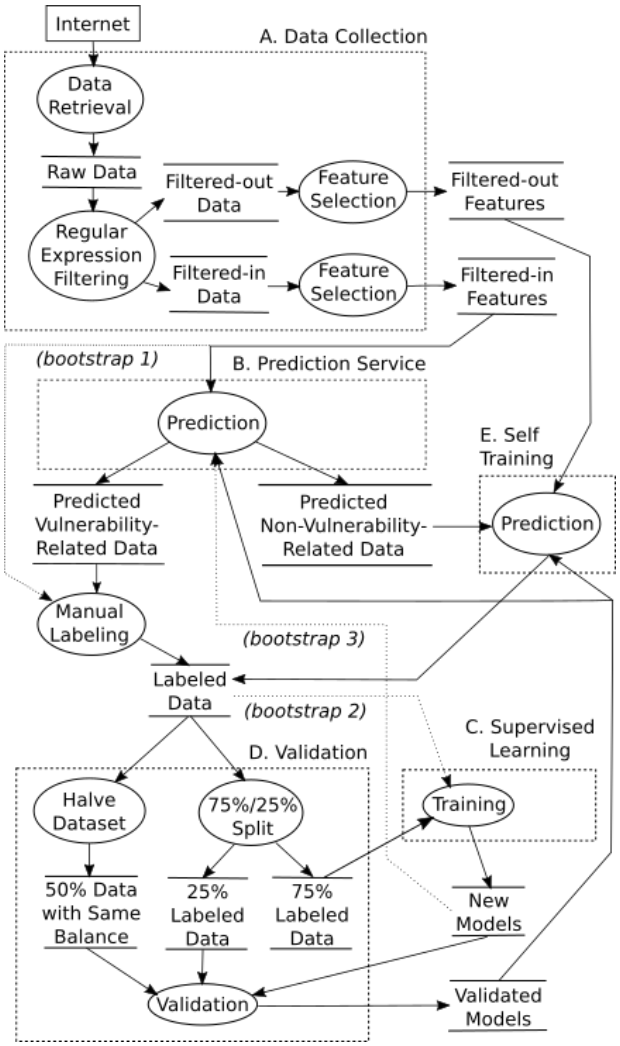

Figure 1: System Pipeline Data Flow Diagram

than the high threshold, we label the data item as vulnerability-related, and when it is lower than the low threshold, we label the data item as non-vulnerability-related. Otherwise, the data item remains unlabeled. We combine this automatically-labeled dataset with the manually-labeled dataset into a single labeled dataset that we use to train our machine learning models for the next iteration. These we detail in Section 7.

In addition, we also provide our production models as a service. The prediction system is accessible via a REST API, where the user provides input data with specified data source type, and obtain prediction results. Including this component, our system is a complete end-to-end machine learning system with continual performance improvements over time.

3 DATA COLLECTION

3.1 Data Sources

We implemented a data collection engine to collect data from different data sources of open source projects. Table 1 shows the

Table 1: Dataset Sizes

Data Source	Collected Size	No. Positive	Positive Ratio
Jira Tickets	17,427	911	5.23%
Bugzilla Bugs	39,801	20,250	50.88%
Github Issues/PRs	50,895	5,147	10.11%
Commits	157,450	5,181	3.29%
Emails	20,832	11,756	56.43%
Reserved CVEs	31,056	7,245	23.33%

summary of the data we collected for different data sources for the first iteration of our pipeline. The second column shows the total collected data sizes. The third column shows the final positive labeling counts, where positive data are those labeled vulnerability-related, and the fourth column shows the ratio of those data in all the data that we collected (second column). Although we collect the same set of features from Jira Tickets, Bugzilla bugs, and Github Issues/PRs, we separate them as different sources as they have different characteristics; for example, Bugzilla bugs are often more informative and specific than the other two, the notion of issue severity and the typical number of attachments may also differ between the three. For all of the data sources except emails and reserved CVEs we use a regular expression to filter out irrelevant data items. This is because the data for these sources are highly unbalanced, with a large proportion of them are not relevant to security vulnerability (see the fourth column of Table 1). The regular expression includes a set of terms related to computer security, such as *security*, *advisory*, *authorized*, *NVD*, etc., conjoined with the alternation operator | and is the same for all data sources. We consider any data item with a textual component that matches the regular expression as more likely to be vulnerability-related and we use it for prediction. A data item that does not match is to be used for self training. The numbers shown in the second column include those that pass as well as fail the filtering. We next describe our data collection process for each data source.

Jira Tickets. We collect Jira tickets from a list of servers which contains the most popular Jira servers for open source libraries that are frequently encountered in customer scans. There are 82 Jira servers in the list currently and more will be added as necessary. We show some of them in the first column of Table 2. The data collection engine queries the servers daily through their APIs.

Bugzilla Bugs. Similar with the Jira tickets, as at the time of writing we have collected Bugzilla bug reports from 24 Bugzilla servers for open source libraries that we frequently encounter in customer scans. We list some of them in the second column of Table 2. As shown in Table 1 we discovered that Bugzilla bug reports are likely to be vulnerability related when compared to similar data sources (Jira tickets, Github issues/PRs). A possible reason is that Bugzilla bug reports are specifically about bugs which include vulnerabilities, whereas Jira tickets and Github issues/PRs in addition also discuss feature requests and implementations. Interested readers may visit the sites we list in Table 2 to confirm this.

Github Issues and Pull Requests. We gathered Github issues and pull requests from 20,447 Github library repositories at the time of writing. This set is also based on the libraries that we frequently encounter in customer scans.

Commits. The commit data we use are from the same Github 20,447 repositories. Since one Github issue or pull request may be related with multiple commits, the number of commit data is much more than the number of issue or pull requests. We use the commit data both to improve the prediction results for the Github issues and pull requests models, as well as independently to train models used in predicting vulnerability-related commits.

Emails. We subscribe to open source software projects mailing list from 18 security-related mailing lists. Although these lists are security-related, the content of each message may not be related to vulnerability, such as a release announcement, and therefore identification is still necessary. We list some of the mailing lists in the third column of Table 2. We utilize the email data without regular expression filtering since the ratio of imbalance is low.

Reserved CVEs. NVD's reserved CVE is an important source to find potential vulnerabilities. A CVE entry is marked as reserved when it is not yet confirmed to pertain to an actual vulnerability. A reserved CVE typically only contains a number of URLs to web pages that possibly describe a vulnerability. The web pages may not describe an actual vulnerability, and therefore we still need to identify a reserved CVE as vulnerability-related, although it is already proposed as a description of a vulnerability. Other details of a reserved CVE get populated when it is confirmed as an actual vulnerability and its reserved status removed.

3.2 Feature Engineering and Selection

The feature engineering and selection are based on our experiences on whether the feature actually helps in improving the performance or not. We show the features that we extract from the data in Table 3. From Jira tickets and Bugzilla bug reports, we extract the same sets of features. We use the Github issues and pull requests data to train two different models, depending on whether a Github issues or pull request has associated commits or not. We name the data source without commits as *Github_Basic*, and the data source with commits as *Github_Combined*. For Github_Basic, we extract the same sets of features as for Jira tickets and Bugzilla bugs. For Github_Combined, we in addition extract commit message, Github user name, patched files paths, patch lines added, and patch lines deleted. The reason for using commit data is that we experimentally discovered that including the corresponding commit data, whenever available, for the identification of the vulnerability relatedness of Github issues and pull requests improves the prediction performance.

For commit data, the authors of [28] decide to use only commit messages as the only feature. They do not use committer's name due to concerns about the lack of accuracy. However, there is an indication in the literature that the quality of the developer significantly affects the quality of the code [9]. We postulate that the developers more familiar with security issues are more likely to be assigned the task to correct a vulnerability. In contrast to [28], therefore, we include Github user name into our feature set.

We avoid using dates as features for training our models since they result in highly imprecise predictions. We previously found that most of the Jira tickets, Bugzilla bugs, Github issues and pull

Table 2: Example Data Sources

Jira Servers	Bugzilla Servers	Mailing Lists
https://issues.apache.org/jira	https://bugzilla.suse.com	announce@apache.org
https://jira.sakaiproject.org	https://bugzilla.mozilla.com	dev@jspwiki.apache.org
https://java.net/jira	https://bugs.eclipse.org	security@apache.org
https://jira.sonarsource.com	https://bugzilla.novell.com	replies@oracle-mail.com
https://issues.jboss.org	https://bugs.webkit.org	rubyonrails-security@googlegroups.com
https://0xdata.atlassian.net	https://bugzilla.gnome.org	security@suse.de
https://jira.mongodb.org	https://bugs.kde.org	debian-security-announce@lists.debian.org

Table 3: Features per Data Source

Data Source	Features Used
Jira Tickets, Bugzilla Bugs, Github_Basic	title, body, source, severity, state, labels, comments, attachment number, comment number
Github _Combined	Github_Basic features, commit message, user name, patched files paths, patch lines added, patch lines deleted
Commits	commit message, user name, patched files paths, patch lines added, path lines deleted
Emails	email subject, body, sender, and status
Reserved CVEs	URL, content of the URL link, URL length, content length

Table 4: Word2vec Parameters

Data Source	Window Size	Minimal Count	Vector Size
Jira Tickets	20	50	200
Bugzilla Bugs	20	50	200
Github Issues/PRs	20	50	200
Commits	10	50	200
Emails	10	20	200
Reserved CVEs	10	10	200

requests were predicted as positive (vulnerability-related) with a newly-trained suite of models. We discovered that this was due to the inclusion of date features for training our models, which encompass creation date and last modification date. If one of these two features' value was of the year 2019, the prediction result would be positive. If we changed its value to 2017 or 2018, the prediction result would likely be negative. The root cause was that the prediction results are highly sensitive to these features, and they were represented by milliseconds by the feature extraction, increasing astronomically from the year 2017 to 2019. These two features impact prediction results significantly. We decided that creation and last modification dates are not as relevant to vulnerability compared to other features, and we then dropped these two features.

We clean each textual feature by removing all non-alphabetic characters, except for a commit patch, where we remove brackets and comment delimiters such as #, `//`, `/**/` from the code.

4 MODELS TRAINING

4.1 Word Embedding

Most of the features in our six data sources described in Section 3 are text. Here we use word embedding to map words in text to a vector space as input to our learners. For this, we use word2vec [10], which is one of the widely-used approaches of word embedding in text mining. We use the word2vec implementation that is available in the Python's Gensim library [4]. Word2vec uses neural network to learn vector representation of words based on the context they are used. This is accomplished by either predicting central word from context word, called *continuous bag of words* (*CBOW*), or predicting context words from central word, called continuous *skip-gram*. The output is a dataset which contains a mapping of each word to a vector. To implement our system, we use CBOW.

We train a word2vec model separately for each textual feature of each data source, as each textual feature may have unique characteristics. A commit data item, for example, has normal textual features (commit message, Github user name, patched files paths), as well as code features (patch added, patch deleted). Recall from Section 3.2 that we clean normal textual data by removing all characters that are not letter, and we clean commit patches by removing brackets and comment delimiters. Word2vec also requires a *corpus* for each data source. This corpus is typically obtained from the input data. Generally, larger corpus results in better vector model. For non-commit data sources, we train word2vec model based on the collected data alone, however, for commits, we in addition expand our corpus by using an existing an unlabeled extra commit data already collected, which include data from various sources such as Github. The number of these extra commit data items is 3,212,690.

Each data item has several features. We build a composite vector whose segments represent all and only the features. For a textual features such as the description of a Jira ticket, we calculate the average value of all the vectors that the words are mapped into by word2vec, and append the resulting vector as a representation of the description text in the composite vector. For numeric features such as the number of attachments, we simply append them as an element of the composite vector.

Window size, *minimal count* and *vector size* are three important parameters to train word2vec model. *Window size* is the maximum distance between the current and predicted word within a sentence. *Minimal count* is the count to ignore all words if their total frequency is lower than this value. *Vector size* is the vector size for one word. These parameter values are listed in Table 4 by data sources. We obtained the parameter values by picking the best-performing ones out of several possible values we experimented with.

1 for each $L \in LabeledDataSources$:
2 for $i := 1, \ldots, k$:
3 $L_i :=$ Random subset of L of size $|L|/k$
s.t. $\forall j \cdot 1 \leq j < i \Rightarrow L_i \cap L_j = \emptyset$
4 for $i := 1, \ldots, k$:
5 $L_{test}, L_{train} := L_i, L - L_i$
6 for each $t \in ClassifierTypes$:
7 $basicModel_L^{t,i} := t.fit(L_{train}, labels(L_{train}))$
8 $\overline{v}_t := basicModel_L^{t,i}.predict(L_{test})$
9 $M_i := (\overline{v}_{t_1} \ldots \overline{v}_{t_6})$ s.t. $[t_1, \ldots, t_6] = ClassifierTypes$
10 $M := \begin{pmatrix} M_1 \\ \vdots \\ M_k \end{pmatrix}$
11 $LRModel_L := logisticRegression.fit(M, labels(L))$

Figure 2: k-Fold Stacking Ensemble Pseudocode. The input is *LabeledDataSources*, which is the set of labeled datasets, where each dataset corresponds to a data source. *ClassifierTypes* is the sequence of six basic classifier types: RF, Gaussian Naive Bayes, k-NN, SVM, gradient boosting, and AdaBoost. The procedure outputs a sequence of $LRModel_L$ for all $L \in$ *LabeledDataSources* and a sequence of $basicModel_L^{t,i}$ for all $L \in$ *LabeledDataSources*, $t \in$ *ClassifierTypes*, and $1 \leq i \leq k$.

4.2 Training

In this section we describe our supervised learning approach ((C) of Figure 1) to build vulnerability-relatedness prediction models, one for each data source. For such classification task, there are different types of possible classifiers to use, such as RF or SVM. Here we follow the approach of [28] to use an ensemble of classifiers which has a better performance than each individual classifier. Our *stacking* ensemble consists of six *basic* classifiers and a logistic regression as a meta learner. The six basic classifiers include RF, Gaussian Naive Bayes, k-NN, SVM, gradient boosting, and AdaBoost. The same stacking ensemble is applied to each data source.

For each data item, we represent the positive (vulnerability-related) label using the value 1, and we use the value 0 to represent the opposite label. Since the labels that we use to fit the model are represented by values of 0 or 1, when a model is used to predict a vulnerability relatedness of a data item, the output is therefore a value from 0 to 1, which we call the *probability of vulnerability relatedness* (*PVR*). A PVR is a degree of confidence on whether the data item is related to a vulnerability.

We use the available labeled dataset for training. We randomly split these data using only 75% of them for model training and the 25% rest we reserve for performance testing (Section 5.1). We show our model training pseudocode in Figure 2. The procedure creates a *k-fold stacking ensemble* [28]. In our method, we further randomly split the 75% labeled data part into k disjoint subsets (Lines 2–3 of Figure 2). We choose one of the k subsets as the testing set, and the union of the remaining k-1 subsets as the training set (Line 5 of Figure 2). For each one of the k such split, and for each of the six basic classifiers that we use, we fit a model on the training set (Line 7), and then apply the model for prediction to the testing set (Line 8). Now, for each data item in the testing set, the prediction outputs a PVR from 0 to 1. Since we apply the model to the whole testing dataset, the result of the prediction is a vector of PVRs whose length is the size $|L|/k$ of the testing set. In Figure 2, this is the column vector $\overline{v}_t$ (Line 8). We combine all the column vectors for all basic classifiers into a $(|L|/k) \times 6$ matrix M_i (Line 9). After performing k fittings and predictions for a basic classifier, we obtain a $|L| \times 6$ matrix M whose elements are PVRs. Line 10 of Figure 2 shows the construction of matrix M from $M_1, \ldots, M_k$. We use this matrix, and the column vector of the labels of each data item (denoted as $labels(L)$ in Figure 2) as inputs to a logistic regression to generate a model $LRModel_L$ (Line 11 of Figure 2). We generate one such model for each data source $L \in LabeledDataSource$, and this logistic regression model combined with all of the $6 \times k$ basic classifier models are the outputs of the algorithm. We call this output suite as an *ensemble* model, or simply a *model*. We initially trained new models for all data sources once a week, however, we discovered that their performance does not change significantly on a weekly basis, and therefore we presently only train new models monthly.

5 MODEL VALIDATION

The validation step ((D) in Figure 1) evaluates the performance of the models newly trained by comparing it to the models already deployed in production. With this step we ensure that any new model replacing a production model has a better performance. There are two kinds of tests that we perform. Both involve executing prediction tasks on the test data and computing metrics on the results.

5.1 Performance Validation

As the size of the labeled dataset increases over time, we train the new models with a new labeled superset of the labeled dataset that are used to train the production models. Recall from Section 4.2 that we randomly split the new, larger labeled dataset into two sets: we use 75% of the data for training, and 25% of them for testing. This is the testing data used in the validation step we describe in this section. Here, the metrics that we collect are precision and recall which are defined as follows:

$$precision = \frac{true\ positive}{true\ positive + false\ positive}$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative}$$

Precision reflects the ratio of true positives and all predicted positives. In unbalanced case like ours, it helps to focus on the true vulnerabilities. A high precision would save manual work in identifying false positives. Recall indicates the coverage to identify all truly vulnerability-related items. The higher the recall value, the more real vulnerability-related items are recommended to the security researchers for manual labeling.

Given the same model and dataset, different PVR thresholds give different precision and recall pairs. We iterate over all PVR threshold values from 0.01 to 0.99 with 0.01 increments, and draw *precision-recall* curve with the recall values as the abscissa and precision values as the ordinate. To compare a new model with the

one deployed in production, we compare their precisions given the same recall from the precision-recall curve: the higher the better.

We note that there is another diagnostic tool that could help in comparing two models called *receiver operating characteristic* (*ROC*) curve, however, ROC is not suitable to us since our data have a high ratio of imbalance [2], as shown previously in Table 1.

5.2 Stability Validation

In this section we describe our method to test the commonality of the prediction results provided by a newly-trained model versus the production model. Reports that are too different may indicate problems with the new model. This cannot be done using the performance validation technique described in Section 5.1 as we do not actually compare the similarity of the prediction results between the new and production models. For this validation process, we build a test data for each data source from half of the positively-labeled data, and half of the negatively-labeled data, resulting in a test dataset that has half the amount of the labeled data, but with the same label balance. For the production model, we use the PVR threshold value that is used in the deployment, and for the new model, we select a threshold based on model performance report, where both the precision and recall are higher than those of the production model. Here we define and use a new metric called *deployment stability* as follows:

$$deployment\ stability = \frac{|TP_n \cap TP_p| + |TN_n \cap TN_p|}{test\ dataset\ size}$$

where TP_n and TP_p refer to true positives of the new and production models, respectively, and TN_n and TN_p refer to the true negatives of the new and production models, respectively. This metric measures the coverage of the correct prediction in both models, encompassing both positive and negative data. A high value for this metric indicates that both the new and the production models agree on most of the predictions, so there is not much risk to deploy new model. When deployed, the PVR threshold for new model is the same with the one we use in this test.

6 PREDICTION SERVICE

We deploy the new model into prediction service ((B) of Figure 1) after we confirmed that the new model has a better quality than the current model in production through the validation process (D). Generally, we select a PVR threshold where both the resulting precision and recall of the model are better that those of the production model (see Section 5.2). In the prediction process, the to-be-predicted data are input into the model to produce PVRs. We compare the PVR of each data item with the chosen PVR threshold to predict if it is vulnerability related.

In the prediction service, we perform two types of prediction:

(1) *Batch prediction.* This prediction is done using the daily-collected data that pass regular expression filter. Security researchers review the prediction results and label the data.
(2) *On-demand prediction.* We provide a web interface for on-demand prediction results. For this, we made part of our system available by implementing a RESTful web service to provide predictions based on the user's data. The prediction is performed using the latest-validated models.

Table 5: Labeled and Unlabeled Datasets Sizes

Data Source	Collected Data Size	Labeled Data Size	Unlabeled Data Size
Jira Tickets	17,427	13,028	4,399
Bugzilla Bugs	39,801	22,553	17,253
Github Issues/PRs	50,895	17,230	33,665
Commits	157,450	22,856	134,594
Emails	20,832	16,573	4,259
Reserved CVEs	31,056	18,399	12,657

For Github issues and pull requests, we use the Gthub_Basic model for prediction when the data has no commits information, otherwise we use the Github_Combined model.

7 SELF TRAINING

As mentioned in Section 2, our security researchers label the predicted as vulnerability data only. Besides the labeled data, there are a large amount of unlabeled data which are made up of two parts. One is the set that fails regular expression filter during data collection and another is the one which includes items predicted as non-vulnerability-related during batch prediction. Table 5 shows the labeled and unlabeled data size for each data source.

This is a typical semi-supervised learning scenario [12, 15, 27], where there is a large amount of unlabeled data, and a small amount of labeled data, especially for the commit data source where the number of unlabeled data items is about six times that of labeled data items. We use self training method to label these unlabeled data automatically. Self training is a widely-used technique in semi-supervised learning (see Section 9.3). We use the latest validated models deployed in production to predict all of the unlabeled data. We use different PVR thresholds to label vulnerability-related and non-vulnerability-related items. We set a *high* PVR threshold τ_h and a *low* PVR threshold τ_l. Given a threshold τ for the batch prediction, we have that $\tau_l < \tau < \tau_h$. If the PVR of a data item is greater than τ_h, we label it as vulnerability-related. If the PVR is less than τ_l, we label the data item as non-vulnerability-related. We do not label the data item if its prediction score is between the high threshold and low threshold. We determine the values for τ_h and τ_l based on our previous experiences for different data sources, where the chosen values we expect to result in high prediction accuracy. We note that in this way, the dataset labeled as non-vulnerability-related by the self training mechanism is a subset of the dataset labeled as non-vulnerability-related by the batch prediction, as $\tau_l < \tau$.

Due to imprecision risk, we restrict the usage of self-training data labels to model training only and not using them as inputs for the security research team.

8 EVALUATION

8.1 Models Deployment Case Study

In this case study, we use the testing methodologies introduced in Section 5 for illustrating the performance improvements in an iteration. The case study is that of the initial iteration of our pipeline when we introduced self training into the system. This iteration improves the models performance for most of the data sources. Here,

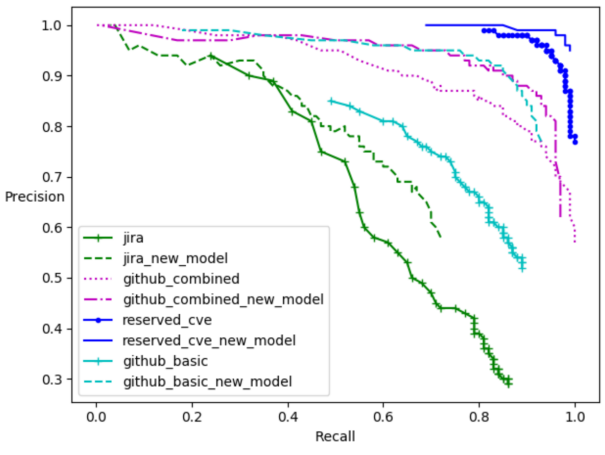

Figure 3: PR Curve for Jira, Github_Combined, Github_Basic, and Reserved CVEs

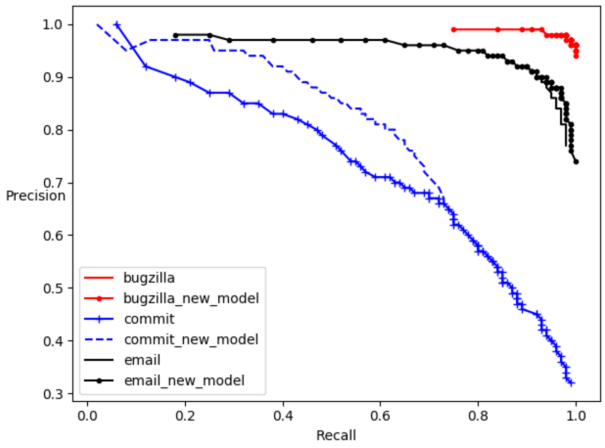

Figure 4: PR Curve for Bugzilla, Email, Commit

we use production models that are trained using a set of manually-labeled data only. The new models are on the other hand, are trained using both the manually-labeled and the self-training-labeled data from the data sources described in Section 3. The manually-labeled dataset used to generate the production models is a subset of the manually-labeled dataset for training the new models.

Performance Evaluation. For performance evaluation, we use the same technique as that of performance testing, which is described in Section 5.1. For each data source, we test the performance of the model with 25% of the reserved labeled data, which are separate from the rest 75% used for training the model itself. To evaluate the new model, we compare its precision and recall with the production model using the precision-recall (PR) curve. Figures 3, and 4 combined show the PR curves for all different data sources. We separate them into two figures for the purpose of clarity. In the figures, each data source has two curves: one for the new model and another for the production model. For each data source, the curve for the new model is mostly above that of the production model. In another word, for each data source, given a specific recall, the precision of the new model is in this case greater than production model. This tendency is the same for all data sources, and it indicates that the new models have better performance. Bugzilla bugs and reserved CVEs have very high precision for all recalls, this is because the input data themselves are mostly vulnerability-related.

Given the same recall, Github_Combined has a better precision than Github_Basic in both the production and new model, indicating that incorporating commit features is useful for predicting vulnerability-relatedness of Github issues and pull requests. For a clearer comparison, Table 6 shows the precision values for both data sources and given the recall values of 0.5, 0.7, and 0.9.

Stability Evaluation. For stability evaluation we use the same technique as for stability testing, which is described in Section 5.2. Here we conduct the test on the half of the labeled data, chosen such that they have the same balance of positives and negatives with the original dataset. Concretely, for each data source this is achieved by randomly selecting 50% of data labeled positive, and 50% of the data labeled negative. We input this same dataset to both the production model and the new model. Table 7 shows the data set sizes details, and Table 8 shows the evaluation results. In Table 8, most of the deployment stability values are around 0.9 which means in most cases, both the production model and the new model agree on 90%

Table 6: Precision of Github_Basic and Github_Combined

Data Source	Recall	Precision	
		Production	New
Github_Basic	0.5	0.85	0.97
	0.7	0.75	0.95
	0.9	0.52	0.85
Github_Combined	0.5	0.95	0.97
	0.7	0.89	0.95
	0.9	0.79	0.88

Table 7: Stability Evaluation Dataset Sizes

Data Source	50% of Positively Labeled	50% of Negatively Labeled	Total Size
Jira Tickets	455	6,059	6,514
Bugzilla Bugs	10,125	1,151	11,276
Github Issues/PRs	2,573	6,042	8,615
Commits	2,590	8,838	11,428
Emails	5,878	2,408	8,286
Reserved CVEs	3,622	5,577	9,199

Table 8: Stability Evaluation Result

Data Source	Deployment Stability
Jira Tickets	0.93
Bugzilla Bugs	0.93
Github_Basic	0.93
Github_Combined	0.77
Commits	0.88
Emails	0.91
Reserved CVEs	0.73

Table 9: Dataset Sizes for Self Training Experiment

	Labeled	Unlabeled	Total
Positive	3,989	849	4,838
Negative	16,011	44,672	60,683
Total	20,000	45,701	65,701

of the prediction results, so there is not too much risk to deploy new model. Github_Combined's deployment stability at 0.77 is lower than others due to a steeper increase in the recall from the prediction to the new model. This is an improvement and we can still confidently replace the current model with the new one. Reserved CVEs' deployment stability at 0.73 is low. We discovered that both models consistently mispredict some amount of positive data as negative. In this case, we do not have a confidence to deploy the new model into production. We discovered the reason is that in this initial iteration, there is an error in our implementation where the features that are used for Reserved CVEs model training become reordered in the prediction stage. This demonstrates that our metric has been effective in helping to discover an error.

8.2 Self Training Experiment

In the case study above, the performance and stability results can be attributed to either the increasing amount of the manually-labeled data used, or the introduced self-training-labeled data. Here instead we present an experiment where the self-training-labeled commit data alone is used to increase the amount of labeled commit data. The objective of this experiment is for demonstrating the effectiveness of self learning for a subset of the problem that we consider. The results motivated our adoption of self training.

For the experiment, we use Github commit data alone. The sizes of the datasets that we use are shown in Table 9. The second column shows the numbers of positive and negative data for the 20,000 manually-labeled commits dataset. The third column shows the sizes of the unlabeled datasets, divided into their eventual positive and negative labelings by self training. Again, this situation is suitable for the application of self training due to the disproportionately larger amount of the unlabeled data.

We first train a model using the k-fold stacking ensemble approach (see Section 4.2) in a supervised manner and using the 20,000 already-labeled data. We use this model to label the unlabeled data, and add them into our labeled dataset, resulting in total 65,701 labeled commit data. In the labeling process, we use PVR low and high thresholds (see Section 7) to decide whether a particular commit with its PVR value is predicted a positive (vulnerability-related) or a negative (non-vulnerability-related). For the low and high thresholds, we use the values 0.22 and 0.88, respectively. 0.22 is the threshold such that we can use our model to decide that a commit is negative with the high precision 0.93 and 0.88 is the threshold such that we can use our model to decide that a commit is positive with also the high precision of 0.91.

Using the larger 65,701 labeled commit data, we train another k-fold stacking ensemble model. Figure 5 shows the PR curves of both models. For any of the recall values with complete precision data, the model with self training consistently has a better precision.

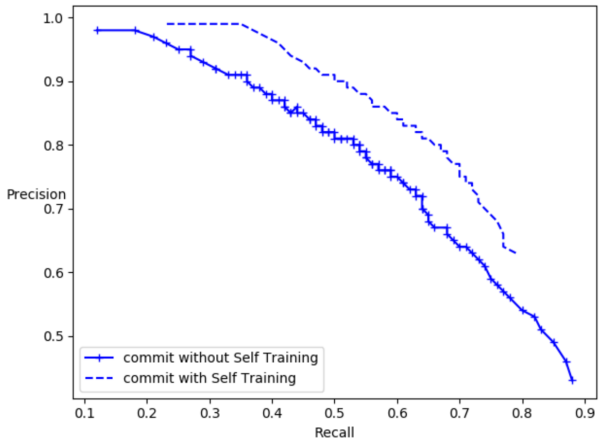

Figure 5: PR Curve for Self Training Experiment

9 RELATED WORK

9.1 Vulnerability Identification

Machine learning technology has been widely used for program analysis [5]. Many of the works are for discovering new vulnerabilities, either automatically [1, 6] or for help in auditing [26], however, there are growing number of works that instead deal with known but unidentified vulnerabilities. The work of Wijayasekara et al. [25] points to the criticality of unidentified vulnerabilities, which they call *hidden impact* vulnerabilities in Linux kernel and MySQL in the period from January 2006 to April 2011. They show that 32% and 64% of respectively Linux kernel and MySQL vulnerabilities discovered during the time period were unidentified vulnerabilities, and that the unidentified vulnerabilities in both software have increased 53% and 10% respectively. Perl et al. [13], for classifies if commits are related to a CVE or not using support vector machine model, but this requires prior identification of the vulnerabilities by CVE ids. Zhou and Sharma [28] explore the identification of vulnerability-relatedness of two kinds of artifacts: commit messages, and issue reports, and pull requests. Zhou and Sharma's approach does not require CVE ids, it discovers hidden vulnerabilities without assigned CVE ids in more than 5,000 projects spanning over six programming languages. For the identification of vulnerability-related commits, Sabetta and Bezzi propose feature extraction also from commit patches in addition to only from commit messages [16], which is the approach taken by Zhou and Sharma [28], however, they only consider commits as data sources. In using commit patch as a feature, similar to Sabetta and Bezzi [16], we also treat patches as natural language text.

Our core learning engine is similar to that of Zhou and Sharma, which uses a supervised machine learning approach for natural languages with the k-fold stacking ensemble, however, we distinguish our work upon theirs in the following ways:

(1) From the technical perspective, we designed and implemented a complete system which uses semi-supervised learning and introduces a novel complete model testing methodologies.
(2) This article expands the range of data being handled, where we add emails and reserved CVEs as data sources, and we

consider more commit data features, including Github user names, patched files paths, and the commit patch itself.

9.2 Learning with Unbalanced Data

Handling unbalanced data is an important research area in machine learning, with a number of literature surveys [7, 8, 19, 24]. The main approaches to unbalanced dataset include *preprocessing*, *cost-sensitive*, and ensemble methods. The preprocessing methods can be further classified into *re-sampling* methods and *feature-selection* methods. The re-sampling methods can be further categorized into undersampling, oversampling, and hybrid methods. Undersampling removes data from the majority class in the dataset to balance the data, while oversampling synthesize data for the minority class in the dataset. In the feature-selection methods, one removes some irrelevant features from the feature space, resulting in more balanced data with only features that are relevant. Cost-sensitive learning assumes higher costs for the misclassification of minority class samples compared to majority class samples, with the algorithm optimizes towards lower cost.

Ensemble method is a popular solution for unbalanced learning [7]. It can be classified into three: bagging, boosting, and stacking [17]. In bagging, the dataset is split into disjoint subsets, and a different classifier is applied to each subset. The results are then combined using either voting for classification, or averaging for regression. In boosting, we serially combine weak classifiers to obtain a strong classifier. In stacking, which includes the k-fold stacking ensemble that we use, the classifiers are coordinated in parallel and their results are combined using a meta classifier or meta regressor, which in our case is logistic regression.

In the area of software engineering, Wang and Yao consider the prediction of defective modules for the next software release based on past defect logs [23]. The data is therefore unbalanced, as the number of non-defective modules is far larger than the defective ones. They consider data re-sampling, cost-sensitive, and ensemble learning methods. One of the best results is provided by boosting method AdaBoost.NC [22]. Rodriguez et al. also review classifiers for unbalanced data for the software defect prediction problem [14]. They consider 12 algorithms, and with using C4.5 and Naive Bayes as base classifiers, and they use Matthew's Correlation Coefficient (MCC) as performance metric. They also discovered that ensemble methods, including SMOTEBoost and RUSBoost provide better results than sampling or cost-sensitive methods.

9.3 Self Training

Self training is a widely-used semi-supervised learning approach, which is applied when the training data set contains a small amount of labeled data, with a large amount of unlabeled data. This situation arises when labeled data are expensive and time consuming to get, yet the unlabeled data are easier to collect. In self training, a supervised learning model is trained based on labeled data and then the model is applied to the unlabeled data to generate more labeled examples. Training is then performed on the new, larger set of labeled data to generate better-performing model. There are works that demonstrate the efficacy of self training in the literature.

Nigam et al. use self training to classify text from three different real-world domains: newsgroup postings, web pages and newswire articles [12]. They use naive Bayes classifier and *expectation-maximization* (*EM*) algorithms for model training. Their experiment shows unlabeled data contains useful information about target functions, and the use of unlabeled data reduces classification error by up to 33%. Yarowsky uses self training for word sense disambiguation [27]. The algorithm is based on two constraints that words tend to have one sense per discourse and one sense per collocation. It achieves better performance than supervised algorithm given identical training contexts and using the one-sense-per-discourse constraint. Self training is used by Rosenberg et al. [15] for object detection. They point out if unlabeled data are labeled incorrectly by self-training model and added to the labeled data set, the model may be potentially corrupted. So they add the self-training labeled data into labeled data set incrementally and check the result. Their study demonstrates that self-training model can achieve results comparable to a model trained in the traditional supervised learning using a much larger set of fully labeled data.

10 CONCLUSION AND FUTURE WORK

We described a design and implementation of a system based on machine learning to support the identification of vulnerable open-source software. Our system uses publicly-available data sources such as Jira tickets, Bugzilla bugs, Github issues and pull requests, Github commits, mailing lists and reserved CVEs, and it generates machine learning models that predict the vulnerability-relatedness of the input data. The results are used to help a team of security researchers in labeling vulnerability-related data items. The system supports a complete pipeline from data collection, model training and prediction, to the validation of new models before deployment. It is executed iteratively to generate better models as new input data become available. We use self-training process that significantly and automatically increases the size of input dataset used in training new models to maximize the improvement in the quality of the generated models at each iteration. We present evaluation results to illustrate how our approach improves the quality of the models, and to show the effectiveness of self training.

There are several directions for future work. Firstly, to filter input data, we use regular expression matching. It is possible that this is too strict, where we filter out too many input data that are vulnerability-related. As the first direction, we can try loosening and changing the regular expression so that it more precisely filters the input data. Secondly, the use of code features are common in machine learning approaches for program analysis [1, 6, 26]. A direction here is to add the whole program code as another data source which we do not attempt in the current work. Lastly, in this article, we use a stacking ensemble of machine learning classifiers for modeling and prediction. The third future work direction is to experiment with deep learning techniques for this purpose.

ACKNOWLEDGMENT

We thank Spencer Hao Xiao for his comments on our draft.

REFERENCES

[1] Timothy Chappelly, Cristina Cifuentes, Padmanabhan Krishnan, and Shlomo Gevay. 2017. Machine learning for finding bugs: An initial report. In *MaLTeSQuE '17*, Francesca Arcelli Fontana, Bartosz Walter, and Marco Zanoni (Eds.). IEEE Computer Society, 21–26.

[2] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *23rd ICML (ACM International Conference Proceeding Series)*, William W. Cohen and Andrew Moore (Eds.), Vol. 148. ACM, 233–240.

[3] FLEXERA2018 [n. d.]. Software Composition Analysis. https://www.flexera.com/products/software-composition-analysis.

[4] GENSIM2019 [n. d.]. gensim: Topic Modelling for Humans. https://radimrehurek.com/gensim/index.html.

[5] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* 50, 4 (2017), 56:1–56:36. https://doi.org/10.1145/3092566

[6] Gustavo Grieco, Guillermo Luis Grinblat, Lucas C. Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery using Machine Learning. In *6th CODASPY*, Elisa Bertino, Ravi Sandhu, and Alexander Pretschner (Eds.). ACM, 85–96.

[7] Haixiang Guo, Yijing Li, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. 2017. Learning from class-imbalanced data: Review of methods and applications. *Expert Syst. Appl.* 73 (2017), 220–239.

[8] Haibo He and Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284.

[9] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *ESEM '13*. IEEE Computer Society, 65–74.

[10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781

[11] NEXUS2018 [n. d.]. Vulnerability Scanner. https://www.sonatype.com/appscan.

[12] Kamal Nigam, Andrew McCallum, Sebastian Thrun, and Tom M. Mitchell. 1998. Learning to Classify Text from Labeled and Unlabeled Documents. In *AAAI '98*, Jack Mostow and Chuck Rich (Eds.). AAAI Press / The MIT Press, 792–799.

[13] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *22nd CCS*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 426–437. https://doi.org/10.1145/2810103.2813604

[14] Daniel Rodríguez, Israel Herraiz, Rachel Harrison, José Javier Dolado, and José C. Riquelme. 2014. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *18th EASE*, Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit (Eds.). ACM, 43:1–43:10.

[15] Chuck Rosenberg, Martial Hebert, and Henry Schneiderman. 2005. Semi-Supervised Self-Training of Object Detection Models. In *7th WACV/MOTION*. IEEE Computer Society, 29–36. https://doi.org/10.1109/ACVMOT.2005.107

[16] Antonino Sabetta and Michele Bezzi. 2018. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *34th ICSME*. IEEE Computer Society.

[17] Vadim Smolyakov. [n. d.]. Ensemble Learning to Improve Machine Learning Results. https://blog.statsbot.co/ensemble-learning-d1dcd548e936.

[18] SOURCECLEAR2018 [n. d.]. SourceClear. https://www.sourceclear.com/.

[19] Yanmin Sun, Andrew K. C. Wong, and Mohamed S. Kamel. 2009. Classification of Imbalanced Data: a Review. *IJPRAI* 23, 4 (2009), 687–719.

[20] SYNOPSYS2018 [n. d.]. Black Duck Software Composition Analysis. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.

[21] VERACODE2018 [n. d.]. Veracode: Software Composition Analysis. https://www.veracode.com/products/software-composition-analysis.

[22] Shuo Wang, Huanhuan Chen, and Xin Yao. 2010. Negative correlation learning for classification ensembles. In *International Joint Conference on Neural Networks, IJCNN 2010, Barcelona, Spain, 18-23 July, 2010*. IEEE, 1–8.

[23] Shuo Wang and Xin Yao. 2013. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Trans. Reliability* 62, 2 (2013), 434–443.

[24] Gary M. Weiss. 2004. Mining with rarity: a unifying framework. *SIGKDD Explorations* 6, 1 (2004), 7–19.

[25] Dumidu Wijayasekara, Milos Manic, Jason L. Wright, and Miles McQueen. 2012. Mining Bug Databases for Unidentified Software Vulnerabilities. In *5th HSI*. IEEE, 89–96. https://doi.org/10.1109/HSI.2012.22

[26] Fabian Yamaguchi, Felix "FX" Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *5th USENIX Workshop on Offensive Technologies, WOOT'11, August 8, 2011, San Francisco, CA, USA, Proceedings*, David Brumley and Michal Zalewski (Eds.). USENIX Association, 118–127.

[27] David Yarowsky. 1995. Unsupervised Word Sense Disambiguation Rivaling Supervised Methods. In *33rd Annual Meeting of the Association for Computational Linguistics*, Hans Uszkoreit (Ed.). Morgan Kaufmann Publishers / ACL, 189–196.

[28] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *11th FSE*. ACM, 914–919.