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

Abhishek Sharma
Veracode
absharma@veracode.com

Asankhaya Sharma
Veracode
asharma@veracode.com

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

ABSTRACT

Software composition analysis depends on database of open-source library vulerabilities, curated by security researchers using various sources, such as bug tracking systems, commits, and mailing lists. We report the design and implementation of a machine learning system to help the curation by by automatically predicting the vulnerability-relatedness of each data item. It 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 to significantly and automatically increase the size of the training dataset, opportunistically maximizing the improvement in the models' quality at each iteration. We devised new *deployment stability* metric to evaluate the quality of the new models before deployment into production, which helped to discover an error. We experimentally evaluate the improvement in the performance of the models in one iteration, with 27.59% maximum PR AUC improvements. Ours is the first of such study across a variety of data sources. We discover that the addition of the features of the corresponding commits to the features of issues/pull requests improve the precision for the recall values that matter. We demonstrate the effectiveness of self-training alone, with 10.50% PR AUC improvement, and we discover that there is no uniform ordering of word2vec parameters sensitivity across data sources.

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, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A Machine Learning Approach for Vulnerability Curation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

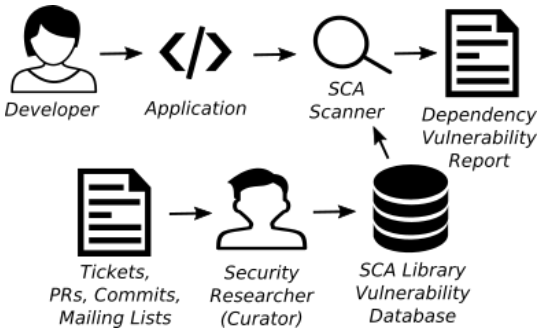

Figure 1: Software Composition Analysis

1 INTRODUCTION

Modern information infrastructure relies heavily on open-source libraries provided by Maven central, npmjs.com, PyPI, etc. Unfortunately, they may contain vulnerabilities that compromise the security of the applications using them. *Software Composition Analysis* (*SCA*) automatically reports vulnerabilities in the open-source libraries used by an application, so that they can be remediated by the application's developers. SCA is widely used in industry, with commercial offerings from various companies [1, 6–8]. Figure 1 shows its typical workflow. A SCA system scans an application to report the vulnerable dependencies (libraries) used, using the data from a library vulnerability database. Security researchers curate this database using various sources, such as tickets and bug reports, *pull requests*, and commit messages [50]. This curation is necessary as many open-source vulnerabilites are not identified in the *national vulnerability database* (*NVD*), the main public information source on vulnerabilities [46]. The design of a state-of-the-art SCA product is discussed in an article [19].

In this article, we report the design and implementation of a machine learning system to help with the curation of a library vulnerability database. This system predicts each input data item as either *vulnerability related* or not, which is then passed as an input to a team of security researchers who curate the database. We train one model for each data source (e.g. Jira tickets), using a stacking ensemble of classifiers, which are reported to perform well for unbalanced datasets and natural language processing in the literature [50]. Our system provides a complete iterative pipeline from data collection, model training and prediction, to the validation

of new models before deployment. We use scikit-learn 0.20 [5] and word2vec [27] from gensim 3.6.0 [2] for word embedding.

Previous approach [50] uses manually-labeled data from the curated database for training new models, however, in this way, a large amount of data remains unused. This unused set includes data items that fail our initial keyword-based filtering, and data items that are predicted as not related to vulnerability by the current production models. To maximize the utilization of these data items, our system performs *semi-supervised* learning [12], in particular *self-training* (as defined by Nigam and Ghani [28]), in which we re-apply our production models to the unlabeled data in order to significantly increase the amount of labeled data for training new models. This opportunistically maximizes the improvement in the quality of models at each iteration of the pipeline.

Our pipeline produces a suite of new models at each iteration. Before deploying these models into production, we need to guarantee that they will perform better than the current production models. For this, we conduct the following two validations:

- *Performance validation.* 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.* We check for each new model that its true positives and true negatives do not vary too much to that of the production model. To measure this, we compute a 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.

In summary, our contribution is on the design and implementation of a machine learning system to support manual curation of a library vulnerability database. Our system supports the whole pipeline of machine learning, including data collection, training, prediction, and validation. The pipeline is executed iteratively to improve the model continually, and has been deployed in production. We use the k-fold stacking ensemble of classifiers from existing work [50], but we advance further in these directions:

(1) From the technical perspective, we designed and implemented a complete system which uses self-training. In our experiment, self-training improves PR AUC by 10.50%.
(2) We introduce a complete model validation methodology with a novel deployment stability metric.
(3) We add emails and *reserved* CVEs (NVD entries that are not yet confirmed as vulnerabilities) [3] into the suite of input data sources, and use more commit features. Previous work [50] uses only commit messages due to concerns about accuracy, however, the quality of the developer significantly affects the quality of the code [26]. We hypothesize that the developers more familiar with security issues are more likely to fix a vulnerability and therefore include Github user name into our commit feature set. We also in addition include patched file paths, and the commit patch itself. For the recall of 0.72 reported by the existing work [50] we more than doubled the precision from 0.34 to 0.69.

In this article, we detail our pipeline, evaluate its performance for one iteration as a case study, with PR AUC improvements of at most 27.59%. Ours is the first of such study across a variety of data sources. We discover that the addition of the features of the corresponding commits to the features of issues/pull requests improve the precision for the recalls that matter. We demonstrate the effectiveness of self-training alone, with 10.50% PR AUC improvement, and we discover that there is no uniform ordering of word2vec parameters' sensitivity across data sources. We show how the deployment stability metric helped to discover an error.

We provide an overview of our system in Section 2. We discuss our data collection and feature engineering in Section 3, and our model training including self-training in Section 4. We discuss our validation methodologies in Section 5 and provide an evaluation of our approach in Section 6 before discussing the limitations of our approach and proposing future work in Section 7. We present related work in Section 8, and conclude our paper in Section 9.

2 OVERVIEW

Figure 2 shows the flow chart of our system. It uses a suite of machine learning models to predict the labels of the collected data for speeding up the manual labeling by security researchers. The system executes iteratively, with a new production suite trained in every iteration with increasing quality. To maximize the performance improvement, in each iteration we employ self-training, where we automatically label residual data not manually labeled, which are then combined with the increasing amount of manually-labeled data to train the next suite. Each new model is then validated against a model for the same data source (e.g., Jira tickets) in the current production suite to ensure that it is better, so that it can replace the current production model.

We collect data from six sources including Jira tickets, Bugzilla reports, Github issues (including pull requests), Github commits, emails from relevant mailing lists and reserved CVEs from NVD. They are then reviewed by security researchers in our company to see which of them are related to vulnerability. The data collected are highly unbalanced, where a very small proportion of the input data are actually vulnerability-related. Due to a large number of unrelated data, a lot of effort of our security team is wasted. To mitigate this issue, we use keyword-based filtering to select data which are more likely to be related to security (detailed in Section 3.1). Only after this filtering step, we use machine learning models to predict the vulnerability-relatedness of the filtered-in data.

To bootstrap the whole process, we first create a suite of initial models. Here, the security researchers reviewed the filtered-in data and labeled them as vulnerability-related or not, directly producing the labeled data (Box B of Figure 2). On this data, we used supervised learning to obtain the initial production models, one for each data source. In the subsequent iterations, the security researchers review only those data that are predicted by the current suite of production models as vulnerability-related and label them. Although we may miss some true vulnerability-related data due to the imprecision of the models, but by using the prediction results, we reduce the manual review and labeling time. In addition to human-labeled data, we have a lot of unlabeled data, consisting of what was predicted as vulnerability-unrelated by the production models, and the data that failed the keyword-based filtering. Their sizes are much larger than the labeled data, particularly for Github commit data. This situation serves as a good use case where semi-supervised learning [12] can be applied. In particular, here we use a class of semi-supervised

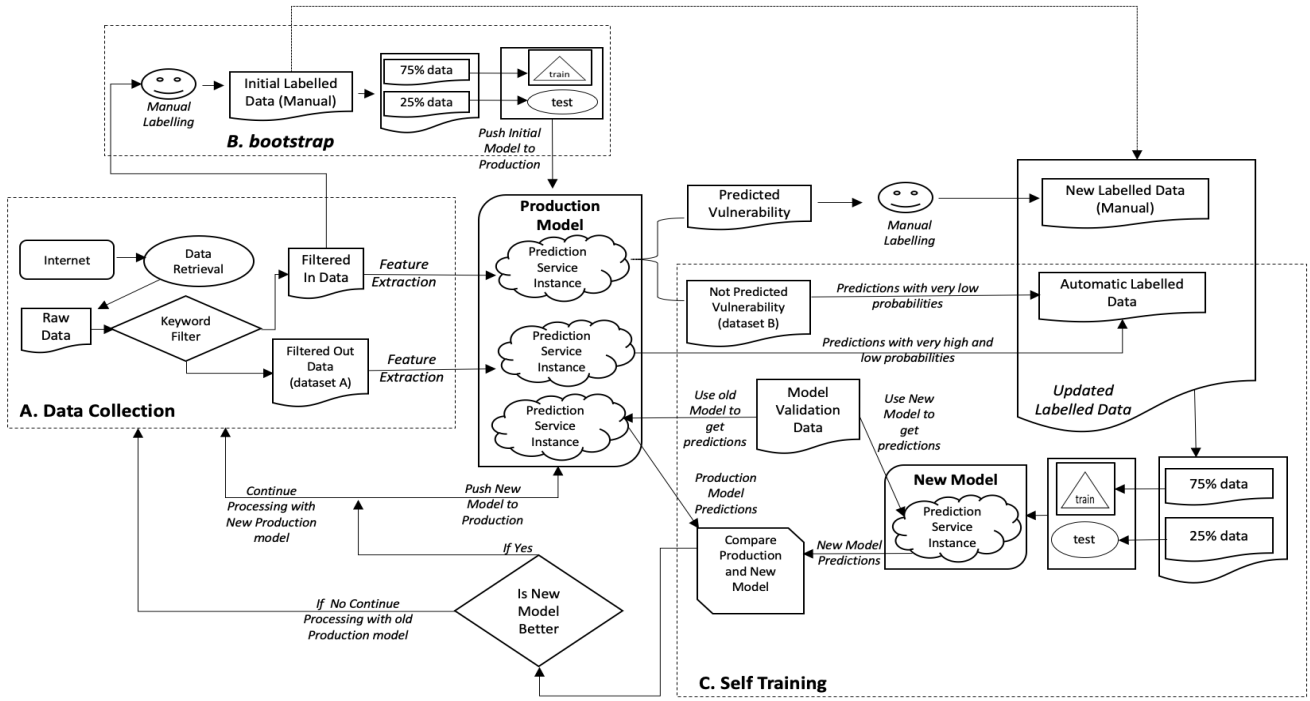

Figure 2: System Pipeline Data Flow Diagram

learning called self-training (as defined by Nigam and Ghani [28]) to automatically label the unlabeled data for training new models.

For self training, we generate automatically-labeled data with the help of our models in production. Firstly, from the dataset predicted as not related to vulnerabilities by the current production model, we filter out items with very low probabilities, and treat them equivalent to a data that would have been labeled by a member of research team as not a vulnerability. Secondly, we also run all the data that was filtered out by keyword based filtering through an instance of our production model. Based on the predictions on this data, we treat the predictions with very high probability as equivalent to data that was labeled as vulnerability by a security research team member, and similarly the predictions with very low probability are treated as equivalent to human labels of not a vulnerability. The automatically-labeled data are then combined with human-labeled data to train new models. Once we have the new models trained their performance on validation data is compared with production models and based on the comparison result, it is decided whether to replace production models. The details of how and what criteria are used to compare are described in detail in Section 5.

3 DATA COLLECTION

3.1 Data Sources

In their day-to-day tasks, the security researchers find information on vulnerabilities from a number of data sources. These include both textual data sources as well as code sources. The data sources are Jira tickets, Bugzilla reports, Github issues (including pull requests), Github commits, mailing lists, and *reserved* CVEs (a CVE entry with an id, but unconfirmed as a vulnerability) from NVD. Table 1 summarizes the data that we collected for the first iteration of our pipeline, including the total collected data sizes (2nd column), the number of data labeled vulnerability-related for training and validating the new model in Section 6.1 (3rd column), and its ratio to the collected data (4th column). The labeled dataset is the result of manual vulnerability curation effort for over a year from 2017 employing a team of 2-3 security researchers. Although we collect the same set of features from Jira Tickets, Bugzilla reports, and Github issues, we separate them as different sources as they have different characteristics. Table 1 also shows that Bugzilla reports are likely to be vulnerability related when compared to the other two, possibly because Bugzilla reports are specifically about bugs which include vulnerabilities, whereas Jira tickets and Github issues in addition also discuss feature requests and implementation details.

Table 1: Dataset Sizes

Data Source	Collected Size	No. Positive	Positive Ratio
Jira Tickets	17,427	911	5.23%
Bugzilla Reports	39,801	20,250	50.88%
Github Issues	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%

At the time of writing we have collected Bugzilla reports, Jira tickets, and Github issues (including pull requests) from 24 Bugzilla servers, 82 Jira servers, and 20,447 open-source library repositories hosted on Github. We build our list of servers and repositories based

Use HTTPS to resolve dependencies in Maven Build. This is a security fix for a vulnerability in Apache Maven pom.xml file(s). The build files indicate that this project is resolving dependencies over HTTP instead of HTTPS. This leaves build vulnerable to allowing a Man in the Middle (MITM) attackers to execute arbitrary code on local computer or CI/CD system.

Figure 3: Vulnerability-Related Jira Ticket Title and Body

Bumps [tar](https://github.com/npm/node-tar) from 4.4.1 to 4.4.13. **This update includes security fixes.**
- [Release notes](https://github.com/npm/node-tar/releases)
- [Commits](npm/node-tar@v4.4.1...v4.4.13)
Signed-off-by: dependabot-preview[bot] <support@dependabot.com>

Figure 4: Vulnerability-Related Commit Message

on the open-source libraries that we frequently encounter in customer scans. The set of Bugzilla and Jira servers spans the areas of software security, library, platform and infrastructure, and utilities. Due to lack of space, we list only some representative libraries in the first two columns of Table 2. Figure 3 shows an example of the title and body of a vulnerability-related Jira ticket [10]. The commit data we use are from the same 20,447 Github repositories. Since a Github issue may be related with multiple commits, the number of commit data is much larger than the number of issues (see Table 1). We use the commit data both to improve the results for the Github issue prediction models, as well as independently to train models used in predicting vulnerability-related commits. Figure 4 shows the message of a vulnerability-related commit [9]. To obtain email data, we subscribe to 18 open-source software projects mailing list. We only list some of them in the third column of Table 2 due to lack of space. For reserved CVEs, we retrieve the data from NVD.

Some data may not provide any useful information. The data for most of our sources are highly unbalanced, with a large proportion of them to be irrelevant to security vulnerability (see the fourth column of Table 1). For all of the data sources except emails and reserved CVEs, we perform a keyword-based filtering with security-related keywords, including *security*, *advisory*, *authorized*, *NVD*, etc. Only data items whose text contains any of the keywords are used as inputs to the next stage. Those that do not contain any of the keyword are to be used for self training. The numbers shown in the second column include those that pass as well as fail the filtering. We do not apply this filtering to the email data since from our experience the degree of imbalance is low. There are also other possible ways of filtering, such as based on the timeliness of the data item, however, such complex filterings are unnecessary.

3.2 Feature Engineering and Selection

We show the features of the data that we use in Table 3. Most of the features are textual and used as inputs to word2vec (see Section 4.1), however, there are some numerical features that are directly included in the input vectors. From Jira tickets and Bugzilla reports, we extract the same sets of features, however, we note that the notion of issue severity and the typical number of attachments may differ between various data sources. We use the Github issue data (including pull requests) to train two different models, depending on whether the issue 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 reports. For Github_Combined, we in addition extract commit message, Github user name, patched files paths, patch lines added, and patch lines deleted (we treat program code as text). For Github issues, we use the Github_Basic model for prediction when the data has no commits information, otherwise we use the Github_Combined model. Section 6.1 shows that Github_Combined has a better precision than Github_Basic for the recalls that matter. For reserved CVEs, each item typically only contains a number of web page URLs. We include the set of URLs as feature as each include host names and parameters that may point to security websites. Although a reserved CVE is more relevant to vulnerabilities than some other types of sources, the web pages may not describe a vulnerability, and therefore we also include their content as a feature.

Our feature selection is based on our experiences on whether the feature actually helps in improving performance. For instance, we avoid using date features since they reduce the prediction precision. At one time we observed a low prediction performance due to Jira tickets, Bugzilla reports, Github issues were predicted as positive (vulnerability-related) with a newly-trained suite of models. If a date value was of the year 2019, the prediction result would be positive, and if we changed its value to 2017 or 2018, the prediction result would mostly be negative. Intuitively, the date features should not be as relevant to vulnerability compared to other features, and we thus exclude them as features.

4 MODEL TRAINING

4.1 Word Embedding

Most of the data features that we consider are textual (e.g., commit messages and commit patches for commits data; see Table 3), model training requires input vectors. To map a textual feature to an input feature, we use word2vec [27], a popular approach for word embedding. Here we use the implementation of word2vec in gensim 3.6.0. Before inputting to word2vec, we clean each textual feature by removing all non-alphabetic characters, except for commit patches, where we remove brackets and comment delimiters such as #, //, /**/ from the code. Word2vec also requires a *corpus* for each data source. 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 unlabeled extra commit data already collected from various sources such as Github. The number of these extra commit data items is 3,212,690.

Word2vec employs neural network to learn vector representation of words in one of two ways, either by predicting central word from context word, called *continuous bag of words* (*CBOW*) (which is the approach that we select), or by predicting context words from central word, called continuous *skip-gram*. *Window size*, *minimal count* and *vector size* are three important parameters to train word2vec models. *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

Table 2: Example Data Sources

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

Table 3: Features per Data Source. $*$ = non-textual numeric feature. We input only the textual features to word2vec.

Data Source	Features Used
Jira Tickets, Bugzilla Reports, Github_Basic	title, body, source, severity, state, labels, comments, number of attachments*, number of comments*
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, patch 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 Reports	20	50	200
Github Issues	20	50	200
Commits	10	50	200
Emails	10	20	200
Reserved CVEs	10	10	200

this value. *Vector size* is the vector size for one word. We list the parameter values in Table 4 by data source, as the result of a tuning process. We show in Section 6.3 that the performance of the model is not significantly sensitive to any one of the parameters in Table 4. The output of word2vec is a mapping of each word into a vector. Given a textual feature, we compute a vector average of all the vectors that the words are mapped into. The vector averages of the textual features are combined with numerical values of non-textual features to construct a composite vector that is input to a learner. We train a word2vec model separately for each textual feature as each textual feature may have unique characteristics.

4.2 Training

We use supervised learning to build our prediction models using scikit-learn 0.20 [5], one model for each data source. For this task, there are different types of possible classifiers to use, such as *random forest* (*RF*) or *support vector machine* (*SVM*), but here we follow previous work [50] to use an ensemble of classifiers which has a better performance than each individual classifier. Our *stacking*

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

Figure 5: k-Fold Stacking Ensemble Pseudocode. *ClassifierTypes* is the sequence of six basic classifier types: RF, Gaussian Naive Bayes, k-NN, SVM, gradient boosting, and AdaBoost. The procedure's input is the labeled dataset L, and it outputs the models *LRModel* and $basicModel^{t,i}$ for all $t \in$ *ClassifierTypes*, and $1 \le i \le k$.

ensemble consists of six *basic* classifiers, which are RF, *Gaussian Naive Bayes*, *k-nearest neighbor* (*k-NN*), SVM, *gradient boosting*, and *AdaBoost*, and employs a logistic regression as the meta learner.

We use the value 1 to label a vulnerability-related data item, and 0 otherwise. 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 randomly select 75% of the labeled dataset, hereafter called L, for model training and the remaining 25% for performance testing (see Section 5.1). Figure 5 shows the pseudocode for training a *k-fold stacking ensemble* [50] model for each data source. As shown, we further randomly split L into k disjoint subsets (Lines 1–2 of Figure 5). We use $k = 10$, which we discovered to perform better than $k = 12$ used in the literature [50]. 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 4 of Figure 5). For each one of the k such splits, and for each of the six basic classifiers that we use, we fit a model on the training set (Line 6), and then apply the model for prediction to the testing set (Line 7). Now, for each data item in the testing set, the prediction outputs a PVR from 0 to 1. Since we apply the model

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 Reports	39,801	22,553	17,253
Github Issues	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

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 5, this is the column vector $\overline{v}_t$ (Line 7). We combine all the column vectors for all basic classifiers into a $(|L|/k) \times 6$ matrix M_i (Line 8). After performing k fittings and predictions for a basic classifier, we obtain a $|L| \times 6$ matrix M whose elements are PVRs. Line 9 of Figure 5 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 5) as inputs to a logistic regression to generate a model *LRModel* (Line 10 of Figure 5). 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.

When using the models for prediction, we apply a threshold on the PVR called *PVR threshold* to predict if the data item is vulnerability-related. Given PVR threshold τ, when a PVR of a data item is strictly greater than τ, the data item is predicted as vulnerability-related (positive), otherwise it is predicted as vulnerability-unrelated (negative). This prediction is given to the security researchers to finally label the data item.

4.3 Self-Training

SCA security researchers only label the data predicted as vulnerability-related. Besides these, there are actually a large amount of other data which are made up of two parts. One is the dataset that fail the keyword-based filtering (*dataset A*), and another is the dataset predicted as unrelated to vulnerability by the current production model (*dataset B*). Table 5 shows the labeled and unlabeled data sizes for each data source. The existence of a relatively large amount of unlabeled data is a typical scenario where semi-supervised learning [12] is applicable, especially for the commit data source where the number of unlabeled data items is about six times that of labeled data items. Here we label the unlabeled data automatically, and use them for training. This is self-training, a widely-used technique in semi-supervised learning (see Section 8.3).

For the automated labeling, we first need to obtain the PVRs of the *dataset A* and *dataset B*. For *dataset A*, we apply the production models to compute their PVRs, and for *dataset B,* we already have their PVRs computed by the production models, but here we use special PVR thresholds for their labeling. We set a *high* PVR threshold τ_h and a *low* PVR threshold τ_l. Given a threshold τ currently used for prediction in production, 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 vulnerability-unrelated. 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 vulnerability-unrelated by the self-training mechanism is a subset of the dataset predicted as such by the current production model, as $\tau_l < \tau$. We combine the automatically-labeled dataset with the manually-labeled dataset for use in training our models for the next iteration. We use self-training data labels for model training only and not inputting them to the security research team since the labels have unacceptable precision for this purpose, although they are good enough for self-training.

5 MODEL VALIDATION

5.1 Performance Validation

The first validation step ensures that any new model replacing a production model has a better performance. Note that here we have a labeled dataset that keeps increasing in size for every iteration of our pipeline. We randomly select 75% of this dataset for training, and use 25% of it for validation (see Section 4.2)[1]. For validation we collect *precision* and *recall* metrics, where:

$$precision = \frac{|TP|}{|TP| + |FP|} \qquad recall = \frac{|TP|}{|TP| + |FN|}$$

with *TP*, *FP*, and *FN* are respectively the sets of true positives, false positives, and false negatives. Precision is the ratio of true positives vs. all predicted positives. A high precision saves manual work in removing false positives. Recall indicates the coverage to identify all truly vulnerability-related items. The higher the recall, the more likely any real vulnerability-related item gets predicted as such (less false negatives). Here our models are not optimized on one of the metrics, as both precision and recall are important.

Given the same model and dataset, each PVR threshold (0.01, 0.02, . . ., 0.99) fixes a precision and recall pair. Using each pair as a coordinate, we plot a *precision-recall* (*PR*) curve in a Cartesian system with the recall values as the abscissa and precision values as the ordinate. Higher curve indicates better performance. This we use to compare the performance of a new model with another model already deployed in production. There is an alternative diagnostic tool called *receiver operating characteristic* (*ROC*) curve, however, ROC is not suitable to us since our data have a high ratio of imbalance [16] (see Table 1).

5.2 Stability Validation

In the starting phase of our model deployments, to decide if a model in production should be replaced with a model trained on more training data, we used only the metrics of precision and recall as described in 5.1. If a new model had higher precision and recall it was pushed as a production model. However, we found issues with this approach. Sometimes a new model even though having better overall performance on precision and recall metrics, would have very different prediction results from the previous model. In

[1]Using k-fold validation instead of 75%–25% split is doable, however, whether further split on top of the k-fold split already done on the 75% part for the model training (Section 4.2) would improve performance significantly is a subject of future research.

some case many human labeled vulnerabilities, which was being predicted correctly by old models would be misclassified by new model as not a vulnerability. Such erroneous predictions led to our security researchers questioning the usefulness and reproducibility of the production models. This practical challenge forced us to design a new metric we call *deployment stability* which is used a second validation step. We define the metric as follows:

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

where TP_n and TP_p refer to respectively the true positives of the new and the (current) production models, and TN_n and TN_p refer to the true negatives of the new and the production models, respectively. A higher value indicates less risk in deploying the new model as both the new and the production models agree on more of the predictions.

For this validation, we build a test dataset for each data source from half of the positively-labeled data, and half of the negatively-labeled data, such that the test dataset has half the size of the labeled data, but with the same distribution of labels. 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 (when the new model is deployed, this will be the new PVR threshold used in production).

6 EVALUATION

6.1 Models Deployment Case Study

Here we use the testing methodologies of Section 5 for illustrating the performance improvements in an iteration. The case study is from 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, we use production models that are trained using a set of manually-labeled data only. The new models are on the other hand, trained using both the manually-labeled and the self-training-labeled data from our data sources introduced 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 Validation. Here we use the performance validation technique of Section 5.1 which employs 25% of the available labeled data. Figures 6 and 7 compare the PR curves for all the data sources (separated into two figures for clarity). In both figures, each data source has two curves: one for the new model and another for the production model. Table 6 shows for each data source how much the new model improves the area under curve of the PR curves (PR AUC) for the recalls where we have precision data. The PR AUC tends to increase across all data sources except for Bugzilla reports where the improvement is negligible. To help evaluate our results, we show the ratio of positive data among all labeled data when training the new models in Table 7.

For Jira tickets, the proportion of positive labels is only 6.99% (Table 7), implying extreme imbalance. However, data imbalance has a negative impact on the performance only when it does not match a balanced world. The discussion topics of Jira tickets are by nature unbalanced, encompassing feature and documentation requests, and design and implementation issues unrelated to bugs and vulnerabilities. Therefore, the performances for Jira tickets are still reasonable, although they are comparatively lower (Figure 6).

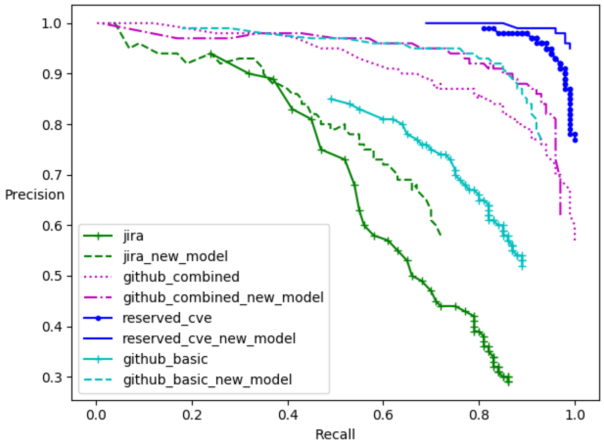

Figure 6: PR Curve for Jira, Github_Combined, Github_Basic, and Reserved CVEs

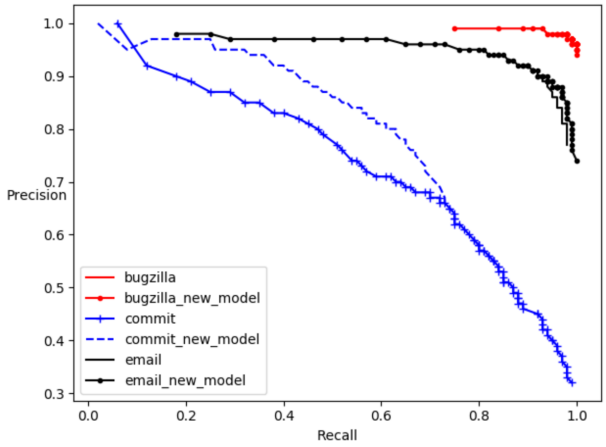

Figure 7: PR Curve for Bugzilla, Email, Commit

Table 6: New Models Improvement over Production Models

Data Source	Recall Range	% PR AUC Inc.
Jira Tickets	0.24–0.72	8.50
Bugzilla Reports	0.90–0.94	0.00
Github_Basic	0.49–0.89	27.59
Github_Combined	0.01–0.97	2.88
Commits	0.06–0.73	8.01
Emails	0.92–0.98	0.95
Reserved CVEs	0.81–0.99	2.52

Table 7: Ratio of Positive Vs. All Labeled for New Models

Data Source	Ratio	Data Source	Ratio
Jira Tickets	6.99%	**Commits**	22.67%
Bugzilla Reports	89.79%	**Emails**	70.93%
Github Issues	29.87%	**Reserved CVEs**	39.38%

From Table 6 we see that the increase in labeled data combined with self-training managed to moderately improve the PR AUC. Assuming the increase in labeled data is small as they have to be labeled manually (recall from Section 4.3 that we do not use the self-training labels as input to researchers), we can conclude that self-training has been effective in improving the model performance for this data source.

The prediction models for Bugzilla reports have very high performance. Bugzilla is specifically for reporting bugs (including vulnerabilities), resulting in a very high proportion of positive labels of 89.79% (Table 7) for training the new model. The world of Bugzilla tickets is by nature unbalanced in the same way as the training data, and hence the high precision. Here, most data items are predicted positive, which matches expectation. Due to the already high performance of the production model, the improvement between the production and new models (Table 6) is negligible.

Figure 6 shows that Github_Basic model has moderately high performance for the new model. The seems to correspond to a moderate data imbalance, were ratio of positively-labeled data is 29.87% (Table 7). Table 6 shows significant PR AUC improvements of 27.59% for Github_Basic. Here also we can assume that increase in manually-labeled data is small and therefore self-training also appears to be effective for this data source.

Given the same recall, Github_Combined has a better precision than Github_Basic for the production model, however, the result is not as straightforward for the new model. Table 8 shows the PR AUC increase for the Github_Combined vs. Github_Basic PR curves. Overall, for all of the recall values for which we have prediction data (0.18–0.93), Github_Combined has a worse performance; however, for the recall ≥ 0.80, it has a better precision than Github_Basic. We discovered that **addition of commit features improves the precision for the recalls that matter**. It appears that the commit messages and code comments in security-related commits have characteristics that are useful for the identification of vulnerability-related Github issues. In Table 6 we observe improvements in the PR AUC when comparing Github_Combined production and new models, suggesting that self-training has been effective.

In Figure 7, we observe that the models for commits have moderate prediction performance. This correlates with moderate data imbalance where the ratio of positive data is 22.67% (Table 7). PR AUC improvement is 8.01% for this data source (Table 6), suggesting that self-training has also been effective. This is confirmed further in Section 6.2 with an experiment without the addition of manually-labeled data in training a new model.

Figure 7 shows a markedly good performance of the email models. Recall from Section 3.1 that there is already a high proportion of vulnerability-related data within the email dataset such that we do not apply keyword-based filtering for it (irrelevant emails in the dataset include release announcements, etc.). Similar to the case of Bugzilla, the result suffers from extreme imbalance where the ratio of positive labels is 70.93% (Table 7), causing most data predicted positive, matching expectation. Table 6 shows that emails has the lowest non-zero PR AUC improvement. The already-high performance for this data source leaves little space for improvement.

Figure 6 indicates that the reserved CVEs' model has a good performance. The new model training data has a relatively good balance of 39.38% (Table 7), therefore, it appears that the performance can be attributed to the good quality of training data. Table 6 shows 2.52% PR AUC improvement for this data source, which also suggests that self-training has been effective, since it is unlikely that the number of labeled reserved CVEs increased significantly between the training of the old and the new models.

Table 8: Github_Combined Improvement over Github_Basic

Recall Range	% PR AUC Increase
0.18–0.93	-0.40
0.80–0.93	0.34

Table 9: Stability Validation Dataset Sizes

Data Source	50% of Positively Labeled	50% of Negatively Labeled	Total Size
Jira Tickets	455	6,059	6,514
Bugzilla Reports	10,125	1,151	11,276
Github Issues	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 10: Stability Validation Results

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

Stability Validation. Here we apply the validation method of Section 5.2. Recall that the test is performed on half of the labeled data for each data source, chosen such that they have the same balance of positives and negatives with the original dataset. Table 9 shows the dataset size details. We input this same test dataset to both the production model and the new model. Table 10 shows the deployment stability results. We observe that 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% of the prediction results, so there is not too much risk to deploy the new model. Github_Combined's deployment stability at 0.77 is lower than most others. This indicates that the new model behaves somewhat differently to the production model on the same input dataset. The likely cause of this is the change in the input data for training, where the new models are trained with more training data, including self-training data. However, although the deployment stability is low, in this case the new model has a much better quality (see Figure 6), and we still replace the current production model with the new model. Reserved CVEs' deployment stability at 0.73 is also 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

Table 11: Dataset Sizes for Self-Training Experiment

	Manual	Self-Training	Total
Positive	3,989	849	4,838
Negative	16,011	44,672	60,683
Total	20,000	45,701	65,701

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 **the deployment stability metric has been effective in helping to discover an error**.

6.2 Self-Training Experiment

In the case study above, although the results suggest that self-training has been effective in improving the performance of the models, the addition of new manually-labeled data in training the new model could have also contributed. To ascertain the effectiveness of self-training, here we present an experiment with two models, one is trained with a dataset of 20,000 manually-labeled Github commits, and another is trained with a larger dataset with only self-training data added to the same manually-labeled dataset. The sizes of the datasets that we use are shown in Table 11. The second column shows the numbers of positives and negatives for the manually-labeled data. The third column shows the sizes of the datasets labeled by self-training. We note that this is the ideal situation for self-training, where the automatically-labeled dataset size is much larger than the size of the manually-labeled dataset.

We first train a k-fold stacking ensemble model (see Section 4.2) in a supervised manner using the 20,000 manually-labeled data. For labeling, we use PVR low and high thresholds (see Section 4.3) to predict whether a particular commit with its PVR is positive (vulnerability-related) or negative (vulnerability-unrelated). For the low and high thresholds, we use the values 0.22 and 0.88, respectively. 0.22 and 0.88 are the thresholds such that we can use our model to decide that a commit is negative, and respectively positive, with high precisions of 0.93 and 0.91, respectively. We next use the supervised training model to label the unlabeled data, and add them into our labeled dataset, resulting in total 65,701 labeled commit data. Using these data, we train another k-fold stacking ensemble model. Figure 8 shows the PR curves of both models. For any of the recall in the range 0.23 to 0.79 with complete precision data, **the model with self-training consistently has a better precision**, with 10.50% PR AUC improvement of the PR curves. This result demonstrates the effectiveness of self-training for commit data.

6.3 Sensitivity to Word2vec Parameters

Here we measure the sensitivity of the model's performance to the values of the word2vec parameters (see Section 4.1). We experiment using Jira tickets', Bugzilla reports', and commits' data sources. To compute the sensitivity, we use the parameter values of Table 4 as a baseline. From the baseline, we change 50% of the value of one parameter in both positive and negative directions, while fixing the values of the other parameters, and we record the performance of the resulting models (one for each change in the positive or negative direction). As the performance measure, we use the PR AUC of the trained models for a uniform recall range for each data source where

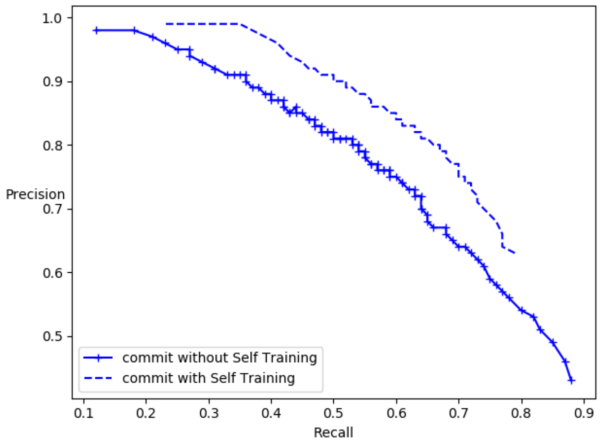

Figure 8: PR Curve for Self-Training Experiment

Table 12: Sensitivity to Word2vec Parameters

Data Source	Parameter	Sensitivity
Jira Tickets	Window Size	0.102
	Minimal Count	0.056
	Vector Size	0.016
Bugzilla Reports	Window Size	0.003
	Minimal Count	0.002
	Vector Size	0.002
Commits	Window Size	0.003
	Minimal Count	0.016
	Vector Size	0.007

we have precision data available (0.1–0.61 for Jira tickets, 0.88–1.0 for Bugzilla reports, and 0.13–0.67 for commits). The sensitivity is computed as the percentage of the absolute change in the PR AUC for each percentage of absolute change in the input value, which is then averaged for both the positive and negative changes. From the results in Table 12, we see that the performance for Jira tickets is most sensitive to the change in window size, minimal count, and vector size, in that order. The prediction performance for Bugzilla reports is already very high (see Figure 7) and is therefore hard to change by tuning the word2vec parameters. For commits, the result is different to that of Jira tickets, where the performance is most sensitive to minimal count, followed by vector size, and lastly window size. Our results indicate that **there is no uniform ordering of sensitivity across data sources**.

7 LIMITATIONS AND FUTURE WORK

We identify the limitations of our work as follows:

- **Limitation to textual data.** A vulnerability can be introduced into an open-source library by a commit, but we cannot identify such vulnerabilities unless they are discussed in any of the data sources. This can be addressed in the future by analyzing library code for vulnerabilities [13, 21, 39, 47].
- **Weakness to data imbalance.** Our results in Section 6.1 indicates that the model performance is still relatively low for Jira tickets despite the use of k-fold stacking ensemble.

This can be addressed in the future by limiting our keyword-based filtering to the set of terms appearing in our labeled dataset may reduce input imbalance.
- **Ignoring label change.** The labels in the SCA library vulnerability database may change. We assume in this work that the amount of such change can be ignored.

For more future work, firstly, we can study the usage of alternative machine learning approaches to mitigate data imbalance further (see Section 8.2), such as off-the-shelf ensemble approaches [14, 18, 25]. Secondly, at the moment we train one word2vec model per data source, with the intention of capturing the special characteristics of each data source. We can instead experiment with a single model trained for all data sources, or for a relevant external large corpora. Thirdly, for word embedding, we can also explore word2vec alternatives [17, 31].

8 RELATED WORK

8.1 Vulnerability Identification

There is a large number of works in the area of vulnerability *discovery*, where the problem is in the discovery of *unknown* vulnerabilities, with a variety of approaches including static and dynamic program analyses, and even machine learning [4, 13, 20, 21, 24, 40, 45, 47, 49], as well as attack surface approximation [37, 38]. In contrast, our work is on the *identification* of vulnerabilities, where the vulnerabilities are *known* in various forms such as fix commits, tickets, and mailing list discussions, but not explicitly identified as such. In this area, machine learning techniques are prominent. Wijayasekara et al. [46] point to the criticality of unidentified vulnerabilities, which they call *hidden impact* vulnerabilities. They show that 32% and 64% of Linux kernel and MySQL vulnerabilities discovered from January 2006 to April 2011 respectively were unidentified vulnerabilities, and that their numbers have increased in both software 53% and 10% respectively. Perl et al. [30] classify commits as related to a CVE or not using SVM model, but this requires prior identification of the vulnerabilities in the NVD. Zhou and Sharma [50] explore the identification of vulnerabilities in commit messages and issue reports/pull requests. Their approach discovers hidden vulnerabilities in more than 5,000 projects spanning over six programming languages. Although we use the same k-fold stacking ensemble, we implement a complete iterative pipeline that improves model precision at each iteration with more data sources and features. Our work also uses self learning [28] and a novel deployment stability metric. For the identification of vulnerability-related commits, Sabetta and Bezzi propose feature extraction also from commit patches in addition to commit messages [34]. As ours, they consider patches as natural language text, however, their work is limited in scope to commits data only. Wan considers vulnerability identification from commit messages using deep learning [41] with higher F1 scores compared to Zhou and Sharma's k-fold stacking ensemble [50], however, the models are trained using a dataset labeled without automated filtering. This alone includes 167K Linux kernel commits. In contrast, our approach considers more data sources, and is aimed at practical applicability by employing keyword-based filtering and self learning. This only requires the manual labeling of a small proportion of the data. Beyond identifying vulnerabilities, Chen et al. applies extreme multi-label (XML) learning to identify libraries from vulnerabilities data in the NVD for SCA [15].

8.2 Learning with Unbalanced Data

Handling unbalanced data is an important research area in machine learning, with a number of literature surveys [22, 23, 36, 44]. The main approaches include *preprocessing*, *cost-sensitive learning*, and ensemble. Preprocessing is further classified into *re-sampling* and *feature-selection*. Re-sampling is further categorized into undersampling, oversampling, and hybrid. To balance the data, undersampling removes data from the majority class in the dataset, while oversampling synthesizes data for the minority class in the dataset. Feature-selection removes 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 is a popular solution for unbalanced learning [22]. It can be classified into three: bagging, boosting, and stacking [35]. 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 software engineering, Wang and Yao consider the prediction of defective modules for the next software release based on past defect logs [43]. 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 achieved by AdaBoost.NC ensemble [42]. Rodriguez et al. also review classifiers for unbalanced data for the software defect prediction problem [32]. They consider 12 algorithms, with C4.5 and Naive Bayes as base classifiers. They discovered that ensemble, including SMOTEBoost and RUSBoost provide better results than sampling or cost-sensitive learning. Different to these approaches, we use stacking instead of boosting ensemble to be more adaptive to variations in the input data sources, nevertheless, the literature justifies our usage of ensemble.

8.3 Self-Training

Self-training is a widely-used semi-supervised learning [12] approach. It is applied when the training dataset disproportionately includes only a small amount of labeled data. This situation arises when labeled data are expensive and time consuming to get, yet the unlabeled data are easier to collect. Nigam and Ghani define self-training as an algorithm that initially builds a single classifier using the labeled training data. It then labels the unlabeled data and converts the most confidently-predicted data item of each class into a labeled training example, iterating until all data are labeled [28][2] Nigam et al. use *expectation-maximization* (*EM*), a related algorithm which iteratively uses the initial Naive Bayes classifier to label the

[2]We distinguish self-training from *co-training* [11], where another model for a different set of features is used to automatically label the training data for a model.

unlabeled text data from the web to learn new parameters for the classifier until there is no change in the parameters [29]. The use of EM reduces classification error by up to 33%. Yarowsky uses an algorithm akin to EM for word sense disambiguation [48], however, here problem-specific constraints are applied to build the initial training data as well as filtering the automatically-generated labels at each iteration. The author reports testing accuracy exceeding 95%. Rosenberg et al. [33] use self-training as defined by Nigam and Ghani [28] (which stops when all data are labeled) for object detection with results comparable to traditional supervised learning using a much larger set of fully-labeled data. In our application of self-training, we use high and low PVR thresholds (Section 4.3) to obtain the most-confidently-predicted data, and we stop in just one self-training iteration without attempting to label all data.

9 CONCLUSION

We described a design and implementation of a machine learning system to help security researches curate a library vulnerability database. It trains machine learning models to predict the vulnerability-relatedness of each item from publicly-available data sources including NVD, bug tracking systems, commits, and mailing lists. 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 employ self-training to significantly and automatically increase 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 also proposed a new deployment stability metric to aid the decision in deploying the newly-trained model at each iteration. Our case study of one iteration of the pipeline demonstrated PR AUC improvements of at most 27.59%. Ours is the first of such study. Here we discovered that the addition of the related commit features to Github issues (including pull requests) improve the precision for the recalls that matter, and that there is no uniform ordering of word2vec parameters sensitivity across data sources. We also showed how self-training results in better precision, with 10.50% PR AUC improvement, and how the deployment stability metric was useful to help discover an error.

ACKNOWLEDGMENT

We thank Darius Foo and Spencer Hao Xiao (in alphabetical order) for their comments on our draft and Hanley Shun for data retrieval.

REFERENCES

[1] [n.d.]. Black Duck Software Composition Analysis. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html.
[2] [n.d.]. gensim: Topic Modelling for Humans. https://radimrehurek.com/gensim/index.html.
[3] [n.d.]. NVD - Home. https://nvd.nist.gov/.
[4] [n.d.]. rough-auditing-tool-for-security. https://code.google.com/archive/p/rough-auditing-tool-for-security/.
[5] [n.d.]. scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/.
[6] [n.d.]. Software Composition Analysis. https://www.flexera.com/products/software-composition-analysis.
[7] [n.d.]. Software Composition Analysis — Veracode. https://www.veracode.com/products/software-composition-analysis.
[8] [n.d.]. Vulnerability Scanner. https://www.sonatype.com/appscan.
[9] 2019. [Security] Bump tar from 4.4.1 to 4.4.13. https://github.com/bevry/extendr/commit/306cab9a9816f137ac763b8f5ee702a67296bb65.
[10] 2020. Use HTTPS to resolve dependencies in Maven Build. https://issues.apache.org/jira/browse/GORA-642.
[11] A. Blum and T. M. Mitchell. 1998. Combining Labeled and Unlabeled Data with Co-Training. In *11th COLT*. ACM, 92–100.
[12] O. Chapelle, B. Schölkopf, and A. Zien. 2006. *Semi-Supervised Learning*. MIT Press.
[13] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay. 2017. Machine learning for finding bugs: An initial report. In *MaLTeSQuE '17*. IEEE Comp. Soc., 21–26.
[14] T. Chen and C. Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *22nd SIGKDD*. ACM, 785–794.
[15] Y. Chen, A. E. Santosa, A. Sharma, and D. Lo. 2020. Automated Identification of Libraries from Vulnerability Data. In *ICSE-SEIP '20*. ACM.
[16] J. Davis and M. Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *23rd ICML (ACM International Conference Proceeding Series)*, Vol. 148. ACM, 233–240.
[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018).
[18] A. V. Dorogush, V. Ershov, and A. Gulin. 2018. CatBoost: Gradient boosting with categorical features support. *CoRR* abs/1810.11363 (2018).
[19] D. Foo, J. Yeo, X. Hao, and A. Sharma. 2019. The Dynamics of Software Composition Analysis. *CoRR* abs/1909.00973 (2019). arXiv:1909.00973
[20] S. M. Ghaffarian and H. R. 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.
[21] G. Grieco, G. L. Grinblat, L. C. Uzal, S. Rawat, J. Feist, and L. Mounier. 2016. Toward Large-Scale Vulnerability Discovery using Machine Learning. In *6th CODASPY*. ACM, 85–96.
[22] H. Guo, Y. Li, J. Shang, G. Mingyun, H. Yuanyue, and G. Bing. 2017. Learning from class-imbalanced data: Review of methods and applications. *Expert Syst. Appl.* 73 (2017), 220–239.
[23] H. He and E. A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284.
[24] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *FSE '19*. ACM, 695–705.
[25] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *NIPS 30*. 3146–3154.
[26] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *ESEM '13*. IEEE Comp. Soc., 65–74.
[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781
[28] K. Nigam and R. Ghani. 2000. Analyzing the Effectiveness and Applicability of Co-training. In *CIKM '00*. ACM, 86–93.
[29] K. Nigam, A. McCallum, S. Thrun, and T. M. Mitchell. 1998. Learning to Classify Text from Labeled and Unlabeled Documents. In *AAAI '98*. AAAI Press / The MIT Press, 792–799.
[30] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *22nd CCS*. ACM, 426–437.
[31] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. 2018. Deep contextualized word representations. *CoRR* abs/1802.05365 (2018).
[32] D. Rodríguez, I. Herraiz, R. Harrison, J. Javier Dolado, and J. C. Riquelme. 2014. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *18th EASE*. ACM, 43:1–43:10.
[33] C. Rosenberg, M. Hebert, and H. Schneiderman. 2005. Semi-Supervised Self-Training of Object Detection Models. In *7th WACV/MOTION*. IEEE Comp. Soc., 29–36.
[34] A. Sabetta and M. Bezzi. 2018. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *34th ICSME*. IEEE Comp. Soc.
[35] V. Smolyakov. [n.d.]. Ensemble Learning to Improve Machine Learning Results. https://blog.statsbot.co/ensemble-learning-d1dcd548e936.
[36] Y. Sun, A. K. C. Wong, and M. S. Kamel. 2009. Classification of Imbalanced Data: a Review. *IJPRAI* 23, 4 (2009), 687–719.
[37] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. A. Williams. 2015. Approximating Attack Surfaces with Stack Traces. In *37th ICSE*, Vol. 2. IEEE Comp. Soc., 199–208.
[38] C. Theisen, K. Herzig, B. Murphy, and L. Williams. 2017. Risk-Based Attack Surface Approximation: How Much Data Is Enough?. In *ICSE-SEIP '17*. IEEE Comp. Soc., 273–282.
[39] Y. Tian, J. L. Lawall, and D. Lo. 2012. Identifying Linux bug fixing patches. In *34th ICSE*. IEEE, 386–396.
[40] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th ACSAC*. IEEE Comp. Soc., 257.
[41] L. Wan. 2019. *Automated Vulnerability Detection System Based on Commit Messages*. Master's thesis. Nanyang Technological University.
[42] S. Wang, H. Chen, and X. Yao. 2010. Negative correlation learning for classification ensembles. In *IJCNN '10*. IEEE, 1–8.

[43] S. Wang and X. Yao. 2013. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Trans. Reliability* 62, 2 (2013), 434–443.
[44] G. M. Weiss. 2004. Mining with rarity: a unifying framework. *SIGKDD Explorations* 6, 1 (2004), 7–19.
[45] D. A. Wheeler. [n.d.]. Flawfinder Home Page. https://www.dwheeler.com/flawfinder/.
[46] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen. 2012. Mining Bug Databases for Unidentified Software Vulnerabilities. In *5th HSI*. IEEE, 89–96.
[47] F. Yamaguchi, F. Lindner, and K. Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *5th WOOT*. USENIX Assoc., 118–127.
[48] D. Yarowsky. 1995. Unsupervised Word Sense Disambiguation Rivaling Supervised Methods. In *33rd ACL*. Morgan Kaufmann Publishers / ACL, 189–196.
[49] M. Zalewski. [n.d.]. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl/.
[50] Y. Zhou and A. Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *11th FSE*. ACM, 914–919.