Interpretable linear models for predicting security vulnerabilities in source code

Toby D. Hocking
Northern Arizona University
toby.hocking@nau.edu

Joseph R. Barr
Acronis SCS, Scottsdale, Arizona, USA
barr.jr@gmail.com

Tyler Thatcher
Acronis SCS, Scottsdale, Arizona, USA
tyler.thatcher@acronisSCS.com

***Abstract*—In our increasingly digital and networked society, computer code is responsible for many essential tasks. There are an increasing number of attacks on such code using unpatched security vulnerabilities. Therefore, it is important to create tools that can automatically identify or predict security vulnerabilities in code, in order to prevent such attacks. In this paper we focus on methods for predicting security vulnerabilities based on analysis of the source code as a text file. In recent years many attempts to solve this problem involve natural language processing (NLP) methods which use neural networks-based techniques where tokens in the source code are mapped into a vector in a Euclidean space which has dimension much lower than the dimensionality of the encoding of tokens. Those embedding-type methods were shown effective solving problems like sentence completion, indexing large corpora of texts, classifying & organizing documents and more. However, it is often necessary to extract an interpretation for which features are important for the decision rule of the learned model. A weakness of neural networks-based methods is lack of such interpretability. In this paper we show how L1 regularized linear models can be used with engineered features, in order to supplement neural network embedding features. Our approach yields models which are more interpretable and more accurate than models which only use neural network based feature embeddings. Our empirical results in cross-validation experiments show that the linear models with interpretable features are significantly more accurate than models with neural network embedding features alone. We additionally show that nearly all of the features were used in the learned models, and that trained models generalize to some extent to other data sets.**

***Index Terms*—vulnerability detection, static code analysis, interpretable linear models, L1 regularization.**

I. Introduction and related work

Every day more parts of society are becoming influenced and controlled by computers. To keep the normal functions of society running as intended, it is therefore important for the code that controls these computers to be running as intended. However, a computer system may not function as intended due to a software vulnerability, meaning a flaw in the code that is exploited by a malicious third party. The concept of a software vulnerability has been formally defined as a fault in the specification, development, or configuration of software such that its execution can violate a security policy [Krsul, 1998]. It is therefore important to be able to identify and fix software vulnerabilities.

Previous research has used white-box univariate statistical tests, based on the internal metrics of code complexity, churn, and developer activity [Shin et al., 2010], as defined by the international organization for standardization [ISO and IEC, 2003]. Proposed complexity metrics for a given function include number of lines of code, cyclomatic complexity, and number of parameters. We limit our study to metrics in this first category. Proposed churn metrics include number of commits in a time period, number of lines changed, and number of new lines added. Proposed developer activity metrics include number of developers who committed changes, number of developers not central to the project, etc, and have a clear prior interpretation in terms of which values are problematic (large number of developers working on one file is claimed to be problematic). That previous paper used univariate significance tests, which are interpretable in terms of what features are correlated with code vulnerabilities, but do not offer any prediction error analysis.

There are a variety of previously proposed methods which use machine learning for predicting vulnerabilities from source code. Yamaguchi et al. [2011] proposed an analysis of the FFmpeg source code, by representing each function as a TFIDF vector of symbols used in that function, then using unsupervised principal components analysis, and computing cosine similarity with a known vulnerability [Salton and McGill, 1983]. Bilgin et al. [2020] proposed machine learning based on an abstract syntax tree, applied to the Draper VDISC data set [Russell et al., 2018], using a multi-task neural network, and comparing with a code2vec representation [Alon et al., 2019]. Harer et al. [2018] proposed learning based on the source code control flow graph, using the random forest learning algorithm. In all of these previous studies, there has been no interpretability analysis (which features are important for making the prediction of vulnerability).

Neural networks have been proposed to learn feature embeddings for vulnerability prediction [Dam et al., 2017]. In previous work, neural networks and boosting algorithms were proposed, with various over-sampling techniques investigated in order to deal with class imbalance [Barr et al., 2020, Barr et al., 2021, Barr and Thatcher, 2022, Barr et al., 2019]. Such algorithms can be considered a black box, so are difficult to interpret in terms of which features are used and important for prediction.

In this paper we study the extent to which software vulnerabilities can be predicted by machine learning analysis of the corresponding source code. Previous work falls into either completely white-box statistical methods with no prediction accuracy analysis, or completely black-box machine learning methods with no interpretability analysis. In contrast we propose building a multivariate predictive model, and selecting important features using L1 regularization. In particular, we are interested in machine learning algorithms which are not only accurate at predicting vulnerabilities, but can also provide an explanation or interpretation about what parts of the source code may be responsible for the vulnerability. The main novelty of our paper is the idea to use L1 regularized linear models to combine the prediction accuracy analysis typical of black-box neural networks with the interpretability analysis typical of white-box statistical methods.

II. Data sets

We investigated open source code data sets based on the Linux kernel (in C programming language), the openssl library (in C), and the `Android OS` source code (C and Java components), link found in Google & The Open Handset Alliance [2020]. For each source code data set we used the `Common Vulnerabilities & Exposure` database to give a binary label to each function: known to be vulnerable, or not Mitre Corporation [2020].

For each data set we computed features for each function, with the goal of using machine learning to predict whether or not a new function should be classified as vulnerable or not. Table I shows for each data set, the total number of functions, and the number of positive and negative labels. Note that we created a "full" data set which consists of all of the data from the four primary data sets (openssl, linux, android_c, android_java), and a "C" data set which consists of just the data from C programming language (openssl, linux, android_c). The "full" data set consists of 2.4 million labeled functions, of which only 17,205 were labeled as positive/vulnerable examples (0.71%). Over all six data sets the percent of positive/vulnerable labels varied from 0.13% for android_java to 1.59% for android_c.

For each function we computed a set of three kinds of features. First, we used previously described methods to compute a set of 128 neural network embedding features using a recurrent network trained to predict the next token in the sequence defined by a function's source code [Barr et al., 2020, Barr et al., 2021, Barr and Thatcher, 2022, Barr et al., 2019]. Second, in order to quantify the complexity of the function [Shin et al., 2010], we computed a set of 8 interpretable features for each function. These interpretable features included the number of parameters/arguments, the number of tokens, and others (details given in Table II). Finally, we computed interaction features for each pair of interpretable features, by multiplying the feature values. We expected that the interaction features should be helpful to obtain increased prediction accuracy using the linear model, but less helpful using the non-linear gradient boosting learner.

III. Algorithms

In this section we briefly explain the interpretable machine learning algorithms that we used in our analysis. First, let $\mathbf{x} = [x_1, \dots, x_d] \in \mathbb{R}^d$ be a vector of d inputs/features, one for each attribute of the source code that we will use to make a prediction (for example, number of comments, number of lines, or one of the learned neural network features). Also, let $y \in \{-1, 1\}$ be the corresponding output, a binary class label (1 if the code has a known vulnerability, 0 otherwise). In this context, our goal is to learn a binary classification function $c : \mathbb{R}^d \rightarrow \{-1, 1\}$, which gives accurate predictions on new/test data. We limit our study to one type of interpretable/explainable learning algorithm: L1 regularized linear models.

A. L1 regularized Logistic Regression

In a linear model, there are two kinds of parameters to learn: $\beta \in \mathbb{R}$ is the intercept, and $\mathbf{w} = [w_1, \dots, w_d] \in \mathbb{R}^d$ is a weight vector (one for each of the input features). The goal is to learn a function $f(\mathbf{x}) = \beta + \mathbf{w}^T\mathbf{x} \in \mathbb{R}$ which outputs a real-valued score. Larger scores are more likely to be positive examples, and the threshold of zero is used to obtain the classification, $c(\mathbf{x}) = \text{sign}[f(\mathbf{x})]$. To obtain an interpretable linear model, we use the Lasso/L1 regularization technique of [Tibshirani, 1996], which computes a simple model in which only a subset of inputs/features are used for prediction. Let $(\mathbf{x}_i, y_i)_{i=1}^n$ be the set of inputs/outputs in the training data, let $\lambda > 0$ be a regularization parameter, and let $\ell(\hat{y}, y) = \log[1 + \exp(-\hat{y}y)]$ be the logistic loss (negative binomial log likelihood / NLL). The L1 regularized logistic regression model parameters are defined as the solution to the following optimization problem:

$$\beta^\lambda, \mathbf{w}^\lambda = \arg\min_{\lambda, \mathbf{w}} \sum_{i=1}^n \ell[\beta + \mathbf{w}^T\mathbf{x}_i, y_i] + \lambda||\mathbf{w}||_1. \quad (1)$$

For sufficiently large λ, the L1-norm penalty causes some of the weights in $\mathbf{w}$ to be reduced to zero, which means the corresponding inputs/features do not affect the predicted scores. The linear model is thus interpretable in the sense that there is a certain subset of inputs/features which are used for prediction ($w_j \neq 0$), whereas others are not ($w_j = 0$). We used the implementation provided in R function `glmnet::cv.glmnet` [Friedman et al., 2010], which selects the penalty λ using 10-fold cross-validation. Briefly, the train set is split into subtrain/validation sets; for each split the subtrain set is used to solve (1) for a grid of regularization λ values, and the validation set is used to compute held-out NLL ℓ. The regularization λ value which results in the minimum NLL is selected (intuitively, this is the model with best accuracy on the new/validation data). Typically that results in some intermediate number of features being selected, so the selected model automatically ignores a certain number of inputs/features which are irrelevant for making accurate predictions.

B. Gradient boosting as a non-linear baseline

In addition to running our L1 regularized logistic regression algorithm for interpretability purposes, we ran a gradient boosting algorithm as a non-linear baseline (to get an idea if there were any non-linear trends in the data that were not

	Data set name					
label	openssl	linux	android_c	android_java	C	full
1(vulnerable)	76	2,047	13,796	1,286	15,919	17,205
0(not)	13,120	548,142	852,634	982,334	1,413,896	2,396,230
total	13,196	550,189	866,430	983,620	1,429,815	2,413,435
Percent vulnerable	0.58%	0.37%	1.59%	0.13%	1.11%	0.71%

TABLE I
NUMBER OF LABELS IN EACH OF THE SIX DATA SETS THAT WERE ANALYZED.

Abbreviation	Description
cos	correlation (cosine) on a record with the mean of all others.
par	The number of parameters/arguments of a function
CC	cyclomatic complexity generated by the Python library "lizard"
len	number of tokens (space-delimited) in a function
loo	count of for and while loops.
con	Branch-if complexity. (Number of ifs)
sys	Number of system calls (to a Unix/Linux function)
max	The maximum number of tokens in a line (between two successive semicolons)

TABLE II
ABBREVIATION AND DESCRIPTION FOR EACH OF THE INTERPRETABLE FEATURES.

captured by the linear model). In detail, we used R package SuperLearner [Polley et al., 2021] to train an ensemble of xgboost models [Chen et al., 2022] with the following values for the shrinkage (also called eta/learning_rate) parameter, $\{10^{-4}, 10^{-3.5}, \dots, 10^{-1}\}$. We used binomial family for gradients (also known as binary:logistic objective in xgboost), and 3-fold cross-validation with AUC maximization as the criterion to optimize for model combination. We expected that this gradient boosting learner should be at least as accurate as the linear model, and more accurate if there are non-linear patterns in the data.

C. Cross-validation setup

To evaluate the prediction accuracy of the learned models, we use 10-fold cross validation. In detail, we assign a fold ID from 1 to 10 to each of the 2.4 million observations. For any given fold ID, we set aside the corresponding observations with that fold ID as test sets. We use observations with different fold IDs to create train sets, which are used as input to the learning algorithms. After obtaining the resulting predictive model from the learning algorithm (including hyper-parameter learning/selection), the final model is used to compute predictions and accuracy metrics on the test set(s).

D. Interpretation methods

We used 10-fold cross-validation, so there are a total of ten linear models that were learned (one for each train/test split). We interpret the ten models in terms of their learned coefficient/weight vectors $\mathbf{w} = [w_1, \dots, w_p]$. Features with coefficient/weight values of zero, $w_j = 0$, are not used at all in the predicted score computation, so can be discarded and interpreted as un-important. Features with non-zero coefficient/weight values, $w_j \neq 0$, contibute to the predicted score computation. For each feature j, we report the number of times the feature has been selected (with a non-zero coefficient) out of the ten possible models/folds, and we also take the mean of the coefficient/weight values across the ten models/folds. More important features will have been selected in more folds, and have larger absolute values for the weights/coefficients.

For each test example $\mathbf{x}$, we can use the predicted score $f(\mathbf{x})$ to rank the example in terms of how likely it is to contain a code vulnerability. In a given test set of functions with possible code vulnerabilities, we can therefore focus our code debugging efforts on the set of examples with the largest predicted scores (say the top 100). For each such test example, we can use the following method to rank the features in terms of order of importance to the prediction as possibly vulnerable. For each feature $j \in \mathbb{R}^p$, we exclude the corresponding entry of the feature vector x_j from the predicted score, and then rank the features in terms of how much the predicted score changes. More formally, we define the predicted score excluding feature j as $f_j(\mathbf{x}) = \beta + \sum_{i \neq j} w_i x_i$. We then subtract this quantity from the actual predicted score, $f(\mathbf{x}) - f_j(\mathbf{x}) = w_j x_j$, which we can sort to obtain a measure of how important feature j was for the vulnerable classification. Large $w_j x_j$ values are associated with features which are very important for the negative classification.

IV. RESULTS

In this section we discuss the results of the computational cross-validation experiments which we performed on the vulnerability prediction data sets.

A. Comparing prediction accuracy of linear and nonlinear models

To determine if the L1-regularized linear model was capturing mostly all of the patterns in the data, we compared its prediction accuracy to the xgboost non-linear baseline. We expected that if there were significant non-linear trends in the data, then the xgboost learner would be significantly more accurate. We used the openssl data set for these comparisons, because it was the smallest data set (and therefore had the smallest time and memory requirements).

For feature representations that used the neural network embedding, we observed significant differences between xgboost with identity weights and L1-regularized linear models with balanced weights (Figure 1). For example, the best test AUC was achieved by the models which used all 164 feature columns, for which xgboost with identity weights was slightly more accurate than cv.glmnet with balanced weights (Test AUC difference = 0.054, p-value in paired one-sided t_9-test = 0.004). However, there was little difference in accuracy between these models when they were only trained using the interpretable and interaction features (Test AUC differences 0.005–0.014, p-values 0.195–0.353). Surprisingly we observed that xgboost with balanced weights always learned a trivial model (test AUC=0.5), and cv.glmnet with identity weights learned a trivial model when trained without the neural network embedding features. Overall these data indicate that the linear models are nearly as accurate as the non-linear xgboost baseline, but there are some non-linear patterns in the embedding features that the linear model was not able to capture.

B. Accuracy when predicting on other data sets

Next, we wanted to investigate the extent to which the learned linear models could be used for prediction in the context of other data sets. To do that we trained linear models using all features, on each of the six data sets and ten train/test splits in 10-fold CV. Each trained model was then used for prediction in each of the six corresponding test sets. We expected that the most accurate models would be the ones trained using the same data as the test set. In agreement with these expectations, we observed that in each test set, the most accurate models had been trained using data from that same set (Figure 2). For example, the most accurate models were in the android_java data set, for which the best models had test AUC of about 0.92 (trained on android_java), whereas the next best had test AUC of only about 0.86 (trained on full data; test AUC difference of 0.0735 with p-value of 10^{-9}). Each data set showed a significant difference between the best model and the next best, with the smallest difference for C data (test AUC difference = 0.008, p-value = 10^{-8}), and the largest difference for openssl data (test AUC difference = 0.0742, p-value = 0.005). Overall these results indicate that it is essential to train models on data which is quite similar to the test set, if optimal accuracy is desired.

C. Comparing accuracy metrics on the full data set

To determine which features were important for optimal prediction accuracy, we trained models using different feature subsets (interpretable, interpretable and interaction, embedding, all), and then computed predictions and accuracy metrics using 10-fold CV. We expected that the regularized linear models would be most accurate using all of the features, because the built-in L1 regularization automatically excludes any irrelevant features. In agreement with this expectation, we observed that the models trained on all features were indeed the most accurate, using three different accuracy metrics on the test set: lift difference, AUC, and percent accuracy (Figure 3. In particular, we observed that the models trained on all features were significantly more accurate than the models trained using just the neural network embedding features (test AUC difference = 0.008, p-value = 10^{-11}). These data provide convincing evidence that both the interpretable and the neural network embedding features are necessary for optimal prediction accuracy.

D. Interpretation of learned linear model weights

To compare the weights for various features, we created a heat map of the weights which were learned in linear models trained on all of the features (for every one of the six data sets, and taking the mean over all ten cross-validation folds). In our heat map we display a mean weight of zero as white (Figure 4), which represents a weight which was completely ignored in all ten train/test splits. To facilitate comparison and visualization of weights with small absolute magnitude, we display relative weight values in our heat map. More specifically, we first take the log of the absolute value of each weight, then normalize all values to between zero and one, then finally multiply by the sign of the original weight. The result is a set of relative weight values between -1 and 1, such that it is easy to visually distinguish a weight of zero from a weight with small non-zero absolute value (such weights would appear indistinguishable from zero on the original scale). We additionally draw a number on the heat map tile if the number of folds/splits with non-zero weights was between 1 and 9; for visualization clarity we omit drawing this number for 0 folds/splits (appears as a white heat map tile) and for 10 folds/splits (appears as a colored heat map tile).

In Figure 4 there are several trends which are apparent. First, there are five features which are not used at all in any data sets, and these are all interaction features (for example len:con, multiplication of number of tokens by number of if statements). All of the other features (interpretable features and neural network embedding features) were selected in at least one data set and fold, indicating that they are all somewhat useful for prediction. There was only one feature, cos:CC, cosine similarity multiplied by cyclomatic complexity, which was always selected using every data set and fold, and always had the same sign. There were a number of features which were used for prediction in only one data set (for example, the number of loops is only used for prediction in android C data, and the C/full data which contain android C data). Finally there were a number of features which had positive weight using some training data, and negative weight using others. An interesting example is the cos feature (cosine similarity), which had zero weight using openssl data, negative weight using linux data, and positive weight using the other data. This is interesting as it indicates a larger value of the cos feature means increased probability of vulnerability in the context of android functions, but decreased probability of vulnerability in the context of linux functions. There are many dozen other examples of neural network embedding features which had learned weights with opposite signs in different data sets (but these features are not interpretable, so it is not clear what the sign reversal means). Overall our heat map was highly useful

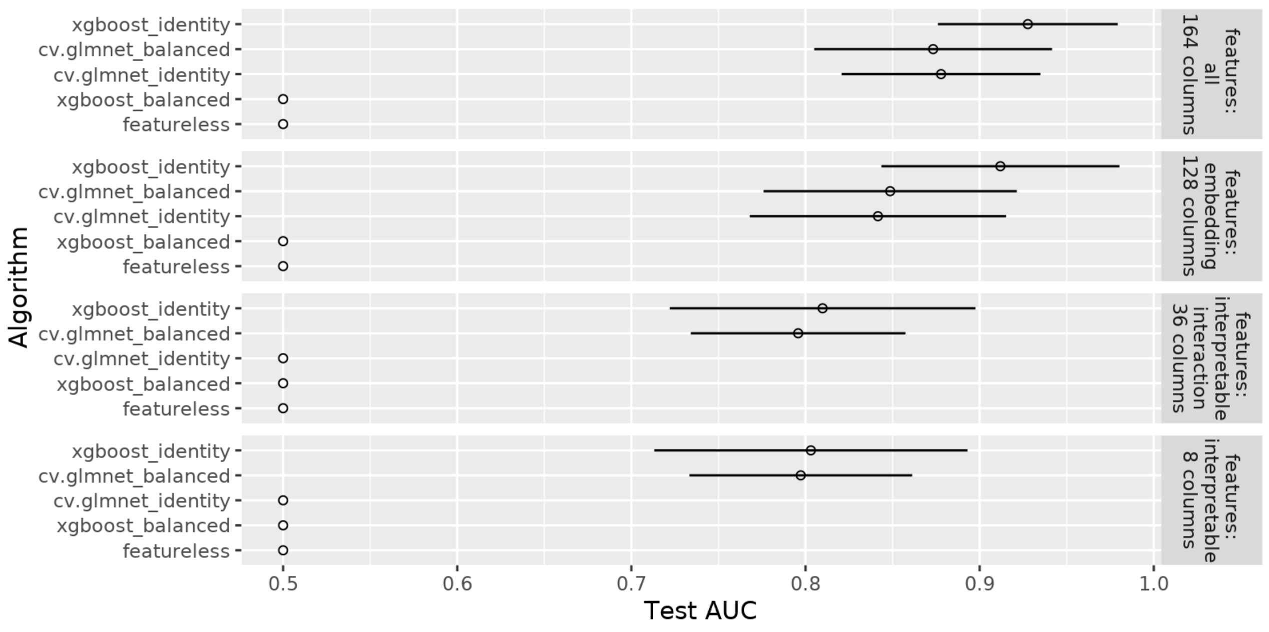

Fig. 1. Test AUC of nonlinear xgboost models and linear cv.glmnet models on the openssl data (mean ± SD over 10 train/test splits).

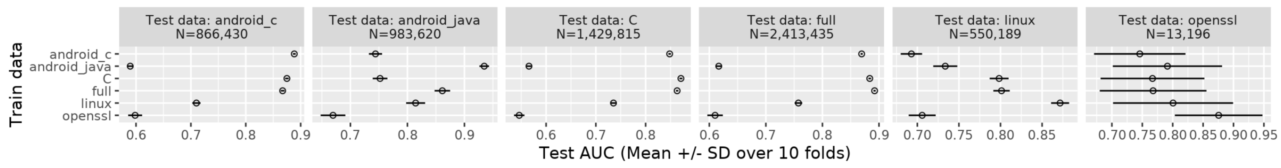

Fig. 2. Comparing test AUC of linear models trained using balanced weights on different subsets of observations. It is clear that in each test set (panels), the most accurate model was trained on data from the same source.

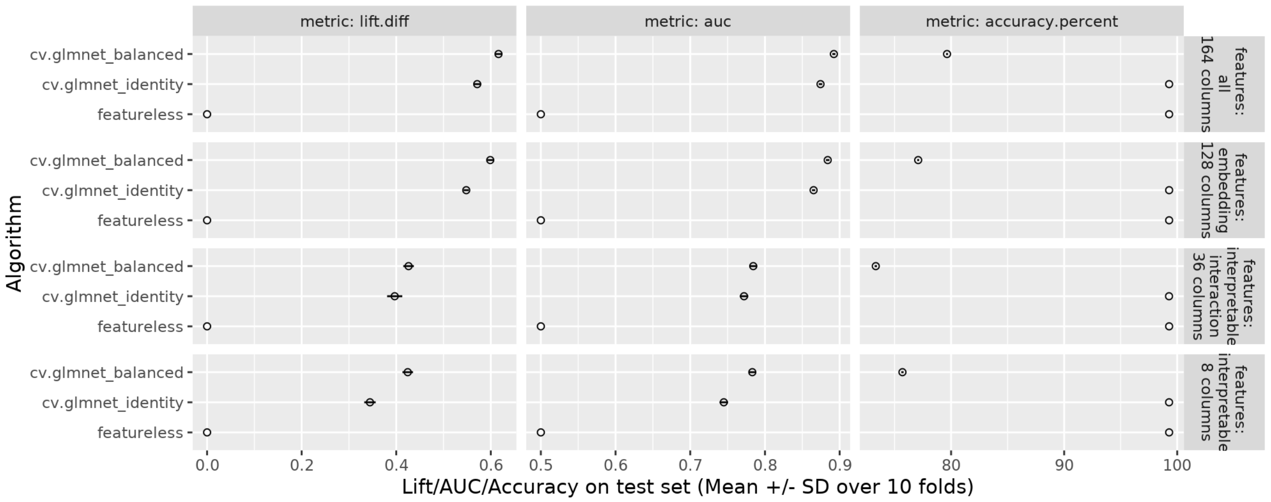

Fig. 3. Comparing three accuracy metrics for linear models trained on the full set of observations, using four different sets of features, and two different loss weighting methods (identity and balanced).

for interpreting which features were used for prediction (or not) using our learned linear models.

V. Discussion and Conclusions

In this paper we proposed an interpretable machine learning algorithm for predicting software vulnerabilities. In particular, we proposed using a linear model with L1 regularization, which is interpretable in terms of which features are important (non-zero weights) or not (zero weights) for making vulnerability predictions. We also provided a systematic analysis of several source code data sets, which indicated that both interpretable features, and neural network embedding features, were necessary to obtain optimal prediction accuracy. Finally, we observed that generalization between data sets is possible to some extent, but that the best predictions were obtained by training on data from the same set.

In our paper we considered only interpretable features which measured code complexity (number of lines, loops, etc). For future work, we could expand the set of interpretable features to include measures of code churn and developer activity [Shin et al., 2010]. Additionally, we observed that the xgboost non-linear learner was slightly more accurate than the proposed interpretable linear models, in the context of the openssl data set. For future work, we could expand this analysis to other data sets, and interpretable non-linear learning algorithms such as decision trees.

VI. Acknowledgements

Joseph Barr is grateful for the support and assistance of Acronis SCS and the parent company Acronis for supporting this project. Special thanks goes to Katerina Archangorodskaja of Acronis SCS for her valuable suggestions and constant encouragements.

References

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

J. R. Barr and T. Thatcher. On the vulnerability of large corpora source code. *2022 IEEE 16th International Conference on Semantic Computing (ICSC)*, pages 314–317, 2022.

J. R. Barr, P. Shaw, F. N. Abu-Khzam, and J. Chen. Combinatorial text classification: the effect of multi-parameterized correlation clustering. In *2019 First International Conference on Graph Computing (GC)*, pages 29–36, 2019.

J. R. Barr, P. Shaw, F. N. Abu-Khzam, H. Yin, S. Yu, and T. Thatcher. Combinatorial code classification & vulnerability rating. In *TransAI*, 2020.

J. R. Barr, P. Shaw, and T. Thatcher. Vulnerability analysis of the android kernel. Pre-print arXiv:2112.11214, 2021.

Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou, M. Li, J. Xie, M. Lin, Y. Geng, Y. Li, and J. Yuan. *xgboost: Extreme Gradient Boosting*, 2022. URL https://CRAN.R-project.org/package=xgboost. R package version 1.6.0.1.

H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. Preprint arXiv:1708.02368, 2017.

J. H. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010. doi: 10.18637/jss.v033.i01. URL https://www.jstatsoft.org/index.php/jss/article/view/v033i01.

J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

I. ISO and T. IEC. 9126-2: Software engineering-product quality-part 2: External metrics. *International Organization for Standardization, Geneva, Switzerland*, 2003.

I. V. Krsul. *Software vulnerability analysis*. Purdue University, 1998.

E. Polley, E. LeDell, C. Kennedy, and M. van der Laan. *SuperLearner: Super Learner Prediction*, 2021. URL https://CRAN.R-project.org/package=SuperLearner. R package version 2.0-28.

R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

G. Salton and M. J. McGill. *Introduction to modern information retrieval*. mcgraw-hill, 1983.

Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

Google & The Open Handset Alliance. Android Fluoride Bluetooth stack. https://android.googlesource.com/platform/system/bt, june 2020.

Mitre Corporation. https://cve.mitre.org/cve/, june 2020.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

F. Yamaguchi, K. Rieck, et al. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011.

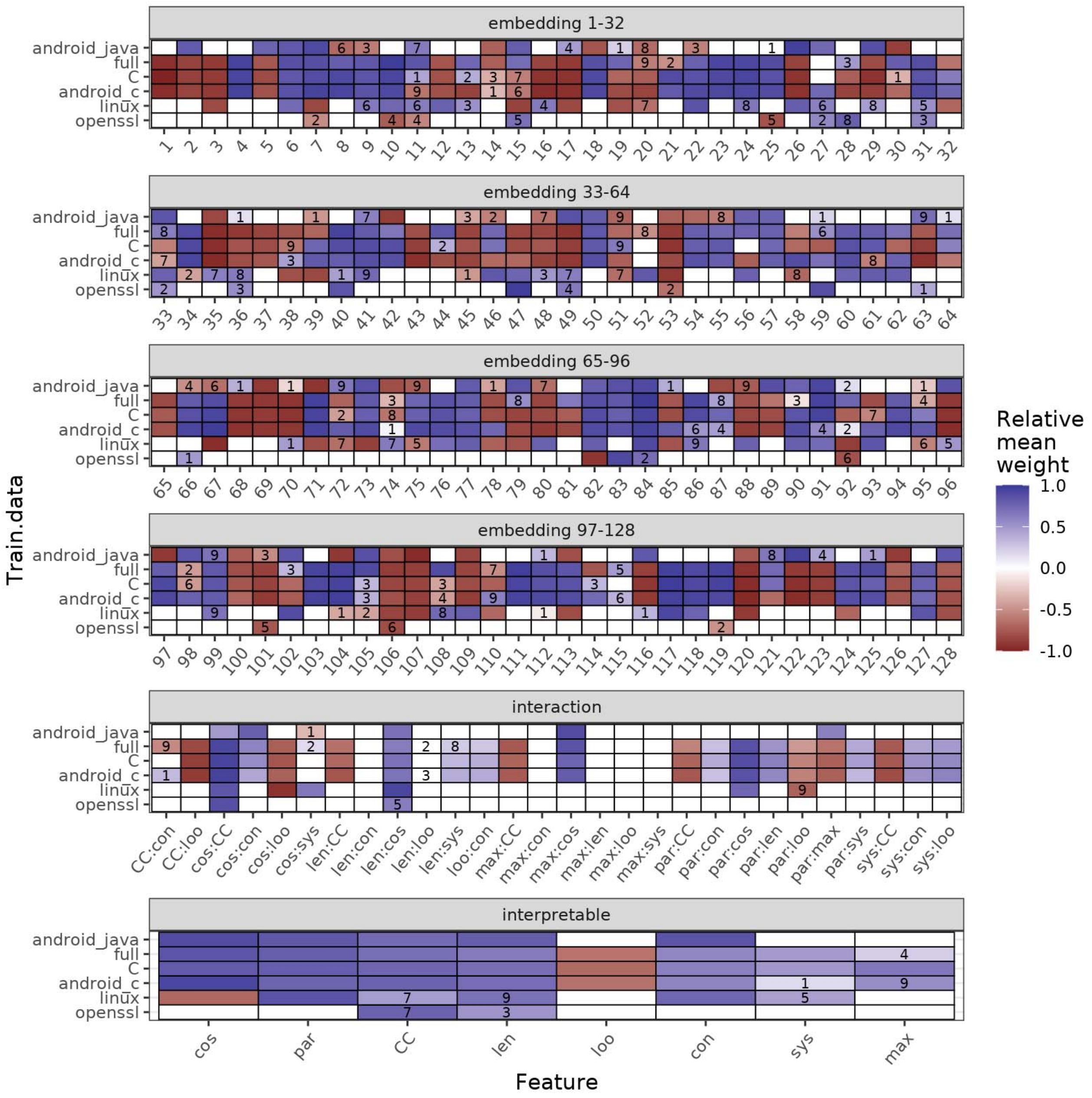

Fig. 4. Heat map of coefficients learned in the linear models trained on six different data subsets, using all of the features. In order to visualize subtle differences between coefficients close to zero, we use a relative color scale to plot the mean of the ten coefficients computed over all train/test splits (darker means larger in absolute value). The number of splits for which the coefficient was non-zero is shown as a number (for 1–9 splits), or white (for 0 splits); otherwise that coefficient was non-zero in all ten splits. Of all the interpretable features, only one (cos) had coefficients of different signs in different data sets (negative in linux, zero in openssl, positive in others). Number of loops had negative coefficient in three data sets (full, C, android_c) and was zero in the others. Other interpretable features had positive or zero coefficients.