Classifying Imbalanced Data with AUM Loss

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

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

Garinn Morton
MI Lab, Acronis SCS,
Scottsdale, Arizona, USA

Tyler Thatcher
MI Lab, Acronis SCS,
Scottsdale, Arizona, USA
tyler.thatcher@acronisscs.com

Peter Shaw
Oujiang Laboritory,
Wenzhou, Zhejiang, China
petershaw@ojlab.ac.cn

***Abstract*—We present a significant improvement to a methodology which was described in several earlier articles by Barr, et al. where we demonstrated a workflow which classifies the source code of large open source projects for vulnerability. Whereas in the past, to deal with dearth of minority examples we've applied upsampling and simulation technique, this present approach demonstrates that a clever choice of cost function sans upsampling results in excellent performance surpassing previous results. In this iteration a feed-forward neural network classifier was trained on *Area Under Min(FP, FN) (AUM)* loss. The AUM method is described in Hillman & Hocking. Similar to earlier work, to overcome the *out-of-vocabulary* challenge, an intermediate step *Byte-Pair Encoding* which 'compresses' the data and subsequently, with the compressed data, *long short-term memory (LSTM) network* is used to embed the tokens from which we assemble an embedding of function labels. This results in 128D embedding which along with additional *'interpretable'*, heuristics-based features which are used to classify CVEs. The resulting labeled dataset is extremely sparse, with a minority class consisting of roughly 0.5% of total. Demonstratively, the AUM cost function is undeterred by sparsity of data; this is amply demonstrated by the performance of the classifier.**

***Keywords*-vulnerability detection; static code analysis; common vulnerabilities & exposures (CVE); source code embedding; byte-pair encoding; LSTM; ROC; AUC; classification of imbalanced data; Area Under Min(FP, FN) (AUM)**

Preamble

Software is complex and thus may contain bugs exposing vulnerabilities that can be exploited by attackers to penetrate and harm a computer systems, steal sensitive data, and cause disruptions resulting in significant harm. Thus, it is generally recognized that identifying and fixing bugs will result in reducing vulnerabilities and lessening the impacts and risks of attacks.

I. Introduction

Although the objective of this article is to demonstrate a methodology to deal with a highly imbalanced tagged data, at the risk of repetition we describe the main features of the workflow emphasizing the approach to binary classification in the face of sparsity. This empirical study of multiple large open source projects the *Android OS* demonstrates the feasibility of *static analysis* workflow to rate CVE risk and in general to help identify vulnerabilities of software components. The workflow's main components are:

1) Data prepossessing
2) Byte-pair encoding
3) Embedding with LSTM
4) (Interpretable) feature extraction
5) Classifications, specifically classification of highly imbalanced data with *area under the minimum (AUM).*

Written in standard C and C++ and Java, source code may be regarded as bags of words consisting of tokens, one bag of tokens per function. Much like a natural language, tokens consist of words, variables, functions, literal strings, punctuation andwhatnot. To embed tokens into a Euclidean space, we utilized scrML's LSTM module [1].

The data is appended by tags representing *Common Exposures & Vulnerabilities* (CVE) [2]. The tags are quite sparse; the minority of class consisted of approximately 1.37 percent of the functions while the majority consists of the remaining 98.63 percent. Whereas in past analyses to get over insufficient signal, we've implemented *SMOTE: Synthetic Minority Over-sampling Technique* [3], a procedure that amplifies the signal by simulating positively-tagged examples. However, in this iteration, we've modified the algorithm by bypassing this step by altering the loss function which we describe shortly.

II. Comparison with prior work

As noted, slices of the methodology are described in earlier work [4], [5] & [6], but this workflow differs from the previous one in a significant way. Whereas in earlier work, either an upsampling technique or SMOTE which simulates examples of a minority class were used, neither of those two are used presently. Despite this, the approach presented was successful in improving classifier's performance.

III. The data

This empirical study of three open source projects: `Android OS v.10`, which is written in C, C++ and Java, the `Linux kernel` which is written in C and `OpenSSL Core Library` which is also written in C. A summary of in tables I and II. Links to the source code is found in [7] (Android OS), [8] (Linux) and [9] (OpenSSL.)

Approximately 1.37 percent of the functions/methods are tagged as CVEs (see section 3.1). `Tags` represent the presence/absence of a CVE where non-CVE is represented with -1, and a CVE with $+1$.

As mentioned earlier, the data in question is *imbalanced.* Indeed, the fraction of functions tagged as CVEs is exceedingly low. The `Linux OS` source code consists of approximately 585,101 functions with 22,260 CVEs; that is approximately 3.8 percent of the records associated with the Linux OS source code are tagged as -1 while the remaining 96.2 percent tagged as $+1$. This kind of imbalanced data is generally considered difficult to *classify.*

A. Common vulnerability exposures (CVE)

`Common Vulnerabilities & Exposures (CVE)` is a database of publicly-disclosed software & firmware vulnerabilities and various risks for exposure to malware and 'bad actors' [2]. Below is code inspection of CVE-2017-0781 [10].

```
void bnepu_release_bcb(tBNEP_CONN* p_bcb)
{
  /* Ensure timer is stopped */
  alarm_free(p_bcb->conn_timer);
    ....
    ....
  while (!fixed_queue_is_empty(
  p_bcb->xmit_q))
  {
    osi_free(fixed_queue_try_dequeue(
    \textbf{p_bcb->xmit_q}));
  }
  fixed_queue_free(\textbf{p_bcb->xmit_q}, NULL);
  p_bcb->xmit_q = NULL;
}
```

Based on the `'commit'` message which says `''Free p_pending_data from tBNEP_CONN to avoid potential memory leaks,''` CVE-2017-0781 is known to cause a memory leak.

IV. Embedding, neural networks & features

A. Autoencoding with Deep learning

`Autoencoding` or `embedding` refers to the process of employing a neural network to assign a numerical vector to an object like text. Neural networks have been applied to natural languages since the 2013 seminal work of Mikolov et al. [11] with a subsequent application to static code analysis in 2019 by Alon et al. [12] who used a different neural network architecture to embed code into a Euclidean space of an appropriate dimension. We employ a recurrent neural network architecture to embed source code tokens.

B. Long Short-Term Memory (LSTM) Networks

Parsing: To extract the token sequences, we first use `Understand` [13] to extract all functions, and then use `srcML` [1] to parse each function into an `Advanced Semantic Tree (AST)` and generate tokens. We have adopted the LSTM language model [14], which we have implemented in this empirical analysis of `Android`, `Linux` and `OpenSSL`.

Autoencoding with LSTM: As shown in Fig. 1, tokens are autoencoded into 128 dimension continuous vector, i.e., $\mathbb{R}^{128}$. See [15] and [16]. Indeed, an LSTM network is a recurrent network which is able to learn sequences of arbitrary lengths [17]. The general architecture of the network used for embedding functions is shown in Fig. 1.

Table I: No. Functions by Project

	`Android`	`Android`	`Linux`	`OpenSSL`	Total
Language	C	Java	C	C	C/Java
Non-CVE	1,099,278	1,143,050	562,841	14,124	2,819,293
CVE	15,602	1,288	22,260	107	39,257
Total	1,114,880	1,144,338	585,101	14,231	2,858,550

Table II: `Functions/Methods.`

`Ancillary`

CVE Files	3,378	917	3,249	*	7,544
Non-CVE Files	92,514	90,942	30,695	1,347	215,498
CVE LOC	10,960,940	5,167,021	5,565,418	*	21,693,379
Non-CVE LOC	27,977,505	11,915,969	15,571,929	386,649	55,852,052
CVE Classes	774	3,536	0	*	4,310
Non-CVE Classes	34,430	170,859	1	*	205,290
CVE Structs	9,499	0	10,821	*	3,378
Non-CVE Structs	56,434	0	28,317	662	85,413

The asterisks (*) represent unknown values which are generally immaterial due to the relative smallness of OpenSSL.

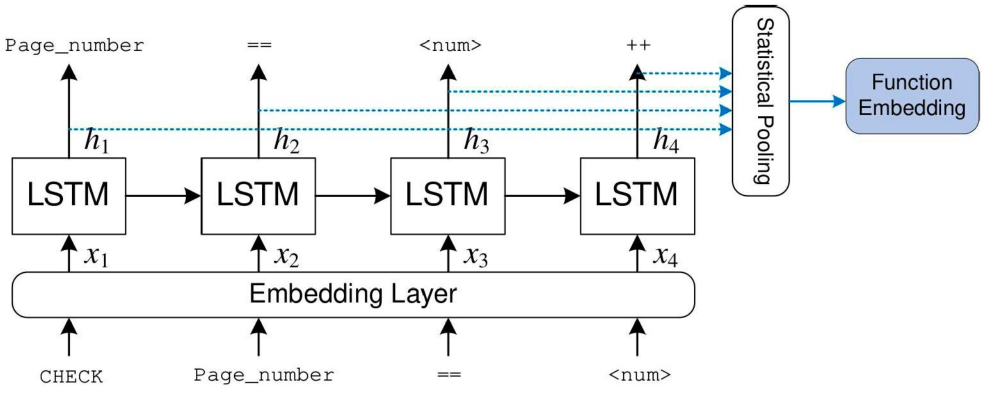
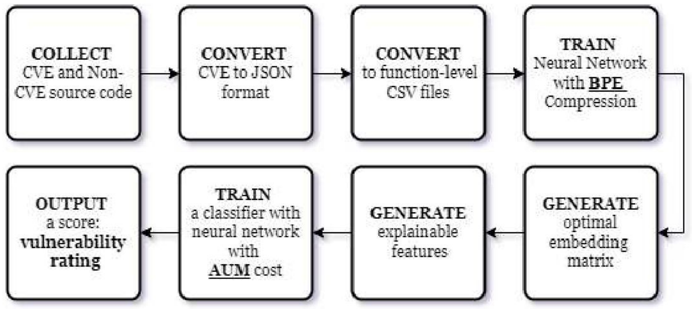


Figure 1: LSTM language model for generating function embedding.



COLLECT CVE and Non-CVE source code
CONVERT CVE to JSON format
CONVERT to function-level CSV files
TRAIN Neural Network with BPE Compression
GENERATE optimal embedding matrix
GENERATE explainable features
TRAIN a classifier with neural network with AUM cost
OUTPUT a score: vulnerability rating

Figure 2: The workflow tool chain for scoring vulnerability.

Explainable Features: In addition to the auto-encoded features, we calculate additional features, some explanatory.

1) `Cosine similarity` which measures the average correlation (cosine) between a feature vector and the mean of the feature vectors.
2) `Number of parameters` is the number of parameters in a function (0 parameter if the input is void)
3) `Cyclomatic complexity` measures branch complexity like `if` and `case`.
4) The `length` is the number of tokens in function body.
5) The `number of loops` like `for` and `while`.
6) `Number of conditionals` is a raw count of `if` and `case`.
7) The `number of system calls` is a count of `UNIX/Linux` calls in a function (0 is none).
8) `Maximal line` is the number of tokens in a longest line (between two successive semicolons.)

V. The process workflow

Fig. 2 describes the process used to rate the risk of code components for vulnerability. This processing toolkit extends on a series of analyses and tool chains previously developed [18]–[21].

This project incorporates two significant improvements:

1) The introduction of 'explainable' features, and more importantly,
2) Implementing of a new, and as we see, effective cost function, ; Area Under Min(FP, FN) (AUM) .

To amplify, given that the data is highly imbalanced, item 2 allows for bypassing a need for upsampling. AUM specifics are discussed below.

The tool chain transforms the data (source code) into a labeled feature matrix where each row represents a function. An embedding into $\mathbb{R}^{128}$, a 128-dimensional Euclidean space achieves optimal performance constrained by practicality. We've discussed embedding calibration in [4] and [21].

The embedding chains several distinct parts:

1) Token extraction with Byte-pair encoding (BPE) (Philip Gage, 1994) [22] is a data compression algorithm which high frequency pairs of consecutive bytes of data are replaced with bytes not occurring within the data. We use the sentencepiece, code in Github [23].
2) Token embedding with an LSTM network [24]
3) Embedding function labels by 'reassembling' the tokens (the tokens produced by BPE procedure) by taking their average embedded value.

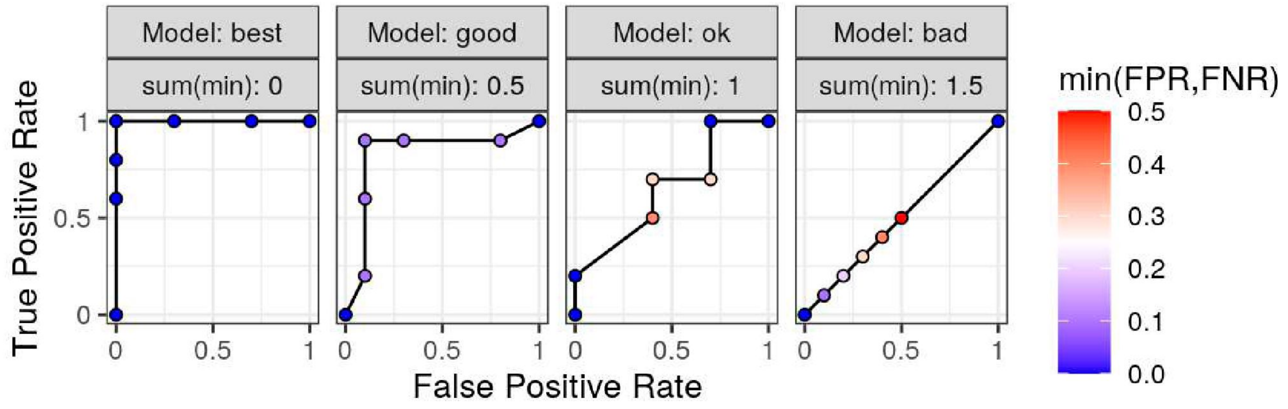


Figure 3: ROC curves for four different binary classifiers, with points on each curve colored using the minimum of False Positive and False Negative rates. The Sum of Min (SM) values over all points on the curve is shown in the panel title, and the proposed AUM loss function is a continuous differentiable relaxation of that quantity (minimizing AUM encourages good ROC curves with small values for sum of min, as in left panel).

VI. Area Under the Min of FP and FN (AUM): surrogate loss for imbalanced binary classification problems

In this section we briefly explain the `AUM surrogate loss function`, which has been shown to result in ROC curve optimization for imbalanced binary classification problems [25]. We assume there is a data set of n labeled examples, and we have learned a function that outputs a real-valued predicted score for each example, $f(\mathbf{x}) \in \mathbb{R}$. The threshold of zero is applied to this score in order to obtain a predicted class, $\text{sign}[f(\mathbf{x})] \in \{-1, 1\}$. `True positives` are examples with positive labels and positive predictions, whereas `false positives` are examples with negative labels and positive predictions. The `ROC curve` is a plot of `True Positive Rate (TPR)` versus `False Positive Rate (FPR)`, when $f(\mathbf{x}) + c$ is used as predictions, for all possible constants $c \in \mathbb{R}$ that could be added to the predicted values. There are up to $n+1$ distinct points on the ROC curve (if there are no ties in predicted scores), and at least two distinct points on the ROC curve (if all predicted scores have the same value, the ROC curve starts at FPR=TPR=0 then jumps diagonally to FPR=TPR=1). Each of the points on the ROC curve has a value for FPR and FNR (=1-TPR). The best point on the ROC curve is the one in the upper left (with FPR=0 and TPR=1 $\Rightarrow$ FNR=0). Good points on the ROC curve tend to have a low values for at least one of FPR and FNR, so it is a reasonable optimization objective to try to minimize total min(FNR,FPR) over all points on the ROC curve (we refer to this quantity as the Sum of Min or SM). For example, Figure 3 shows four ROC curves, each with different values for SM; the ideal ROC curve has a SM value of 0. Like the Area Under the ROC Curve (AUC), this SM optimization objective is a non-convex, piecewise constant function, so it can not be used in gradient descent algorithms directly. The AUM can be understood as an L1 relaxation of this SM function; the AUM is a convex, piecewise linear function that is differentiable almost everywhere, so it can be used in gradient descent learning algorithms. In contrast to the logistic (binary cross-entropy) loss which encourages predictions that result in a correct labeling, the AUM loss encourages predictions that result in a correct ranking (Fig. 4). In fact, predicted values $\hat{y}_i$ need only to be ranked correctly (all negative examples have smaller predicted scores than all positive examples) in order to achieve minimal AUM.

A. Details of AUM computation

Let there be a total of n labeled training examples $\{(x_i, y_i) : x_i \in \mathbb{R}^p, y_i \in \{-1, +1\}, i = 1, 2, ..., n\}$, in a data set or batch. Given a prediction vector $\hat{\mathbf{y}} = [\hat{y}_1 \cdots \hat{y}_n]^\intercal \in \mathbb{R}^n$ we can compute the following false positive and false negative totals for each example $i \in \{1, \dots, n\}$,

$$\text{FP}_i = \sum_{j:\hat{y}_j \geq \hat{y}_i} I[y_j = -1], \quad \text{FN}_i = \sum_{j:\hat{y}_j \leq \hat{y}_i} I[y_j = 1]. \quad (1)$$

In other words, the FP_i, FN_i are the error values at the point on the ROC curve that corresponds to observation i. We sort the observations by predicted value $\hat{y}_i$, yielding a permutation $\{s_1, \dots, s_n\}$ of the indices $\{1, \dots, n\}$ such that for every $q \in \{2, \dots, n\}$ we have $\hat{y}_{s_{q-1}} \geq \hat{y}_{s_q}$. All of the error values FP_i, FN_i, for every $i \in \{1, \dots, n\}$, can then be computed via a modified cumulative sum. We have $n+1$ points (not necessarily unique) on the ROC curve,

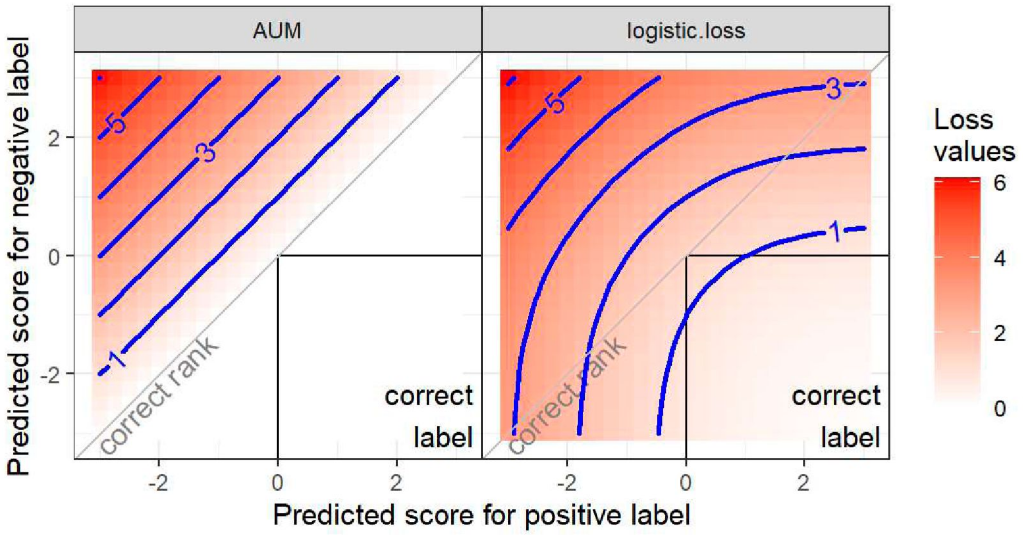


Figure 4: Visualizing AUM (Area Under Min of FP and FN) and logistic loss functions for a positive and negative label. Minima of AUM occur for any predictions that result in correct ranking (predicted score for positive label greater than predicted score for negative label), whereas minimizing logistic loss tends to encourage predictions that result in a correct labeling.

from which the AUM can be computed via

$$\text{AUM}(\hat{\mathbf{y}}) = \sum_{q=2}^{n} (\hat{y}_{s_{q-1}} - \hat{y}_{s_q}) \min\{\text{FP}_{s_q}, \text{FN}_{s_q}\}. \quad (2)$$

B. Classifying Code with AUM Loss

As we noticed earlier, the data is sparse. Of the 2,858,550 functions, a mere 39,257 or approximately 2.11 percent are tagged as CVE ('positive.') Until recently, dearth of minority class (CVEs) hinders a direct approach to modeling, requiring upsampling. Barr et al. have used SMOTE to amplify signal by simulating a minority class [5], [21]. However, it appears that with the AUM cost function a classifier is apt to learn patters and predict accurately.

VII. Model performance with Area Under Min(FP, FN) (AUM) loss

In essence, employing the `AUM loss function` with neural networks is a "black-box" model which shed little light on the significance of the features. We demonstrate that training data consisting of 2,838,550 records with 132 features plus a binary tag of which 39,257 are positively tagged (with $(+1)$) and the remaining 2,819,293 are negatively tagged (with -1). The data set is `highly imbalanced`; the minority class, those tagged with $(+1)$, consisting of approximately 1.73 percent of total. We deploy a two-layer fully-connected, feed-forward neural network classifier with input dimension consisting of 132 nodes, two hidden layers, 250 nodes each and a single binary output layer. Whereas a default loss function is `binary cross entropy (BCE)`, i.e., a function of the form $BCE(\beta) = \sum y_j \log(p_j) + (1-y_j)\log(1-p_j)$ where $p_j = p(x_j;\beta)$ is the probability of $y_j = 1$ conditioned on input vector x_j and β is weight matrix, we deploy `Area Under Min(FP, FN) (AUM) loss` which is previously described. With k=3-fold cross validation, 70-20-10 train-test-validation split, and based on empirics, optimal calibration a batch size of 1,500 and learning rate of 0.00003 an optimal network result in lift of 0.875 at just under 5 percentile, and AUC (area under the ROC curve) of 0.98 (See Fig. 6.)

VIII. Conclusion

Making better distinctions with the AUM loss

This empirical analysis demonstrated that static code analysis using the tool chain described therein might indeed prove to be effective in identifying vulnerabilities that could expose computers and data to unnecessary costly risks. The results obtain for the binary classifier are high. Still an ideal model will result in a `score card` which identify risk factors. For example, we propose to identify properties associated with coding standards like the 'long-method' code-smell of Kent Beck and Martin Fowler [26, 27].

Future work

Clearly static code analysis extends far beyond rating for vulnerability. The tools developed thus far may be put to work to perform tasks like analyzing complexity,

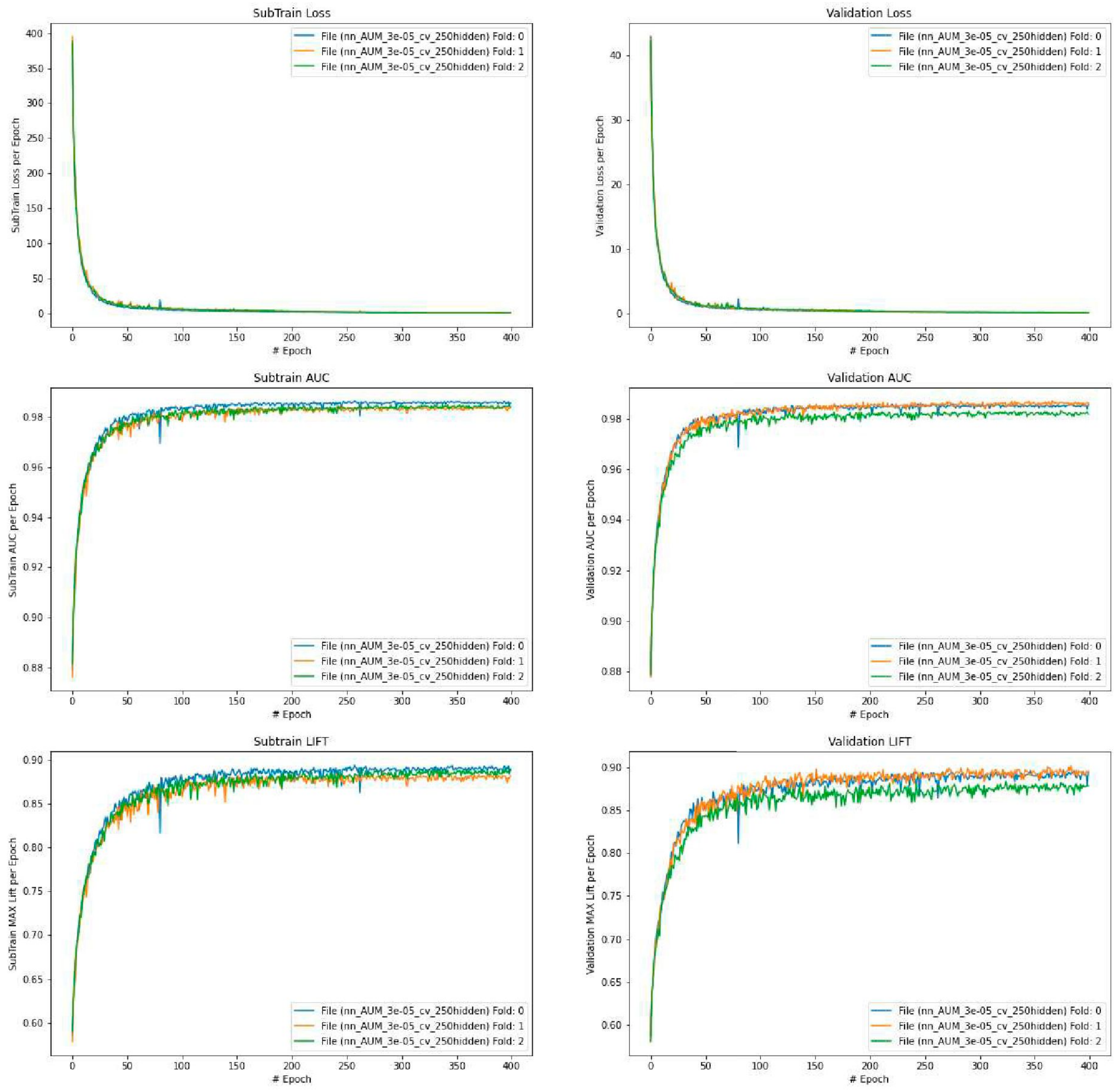


Figure 5: The evolution of AUM loss (top), AUC (middle) and lift (bottom) over 400 epochs.

(textual) data management (storage & retrieval), copyright investigation & plagiarism, etc. Furthermore, we're keen on developing a score card which explains the risks, i.e., provides reasoning for a score.

Acknowledgements. Special thanks to Acronis SCS (USA) for funding the project and for all the support and advice we've received from our friends including Sergey Ulasen and Sanjeev Solanki of Acronis (Singapore) and Neil Proctor of Acronis SCS. Peter Shaw was supported in part by the Oujiang Lab (Zhejiang Lab for Regenerative Medicine, Vision and Brain Health) startup fund.

References

[1] SRCML, "Srcml SciML," https://www.srcml.org/.

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

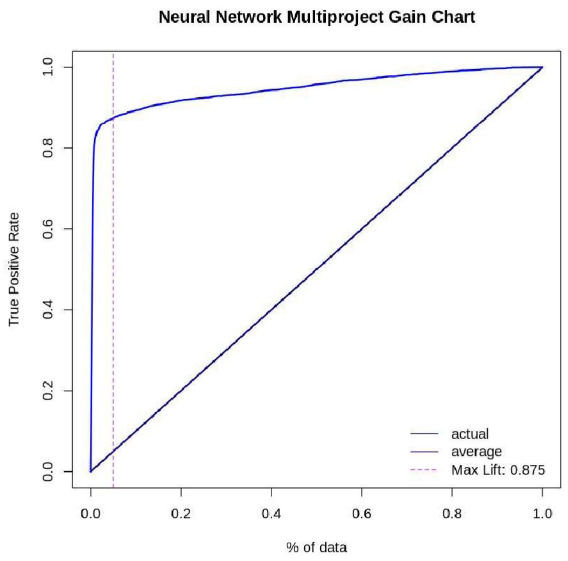


Figure 6: Gains chart: lift 0.875 at a 4.99 percentile.

[3] N. Chawla, K. Boyer, L. Hall, and P. Kegelmeyer, "Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research 16 (2002) 321–357*, 2002.

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

[5] J. R. Barr, P. Shaw, and T. Thatcher, "Vulnerability analysis of the android kernel," *ArXiv*, vol. abs/2112.11214, 2021.

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

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

[8] Linux, "Linux Github," https://github.com/torvalds/linux, 2022.

[9] OpenSSL, "OpenSSL Github," https://github.com/openssl/openssl, 2022.

[10] Mitre Corporation, "CVE-2017-0781," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0781, june 2017.

[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information proc. sys.*, 2013, pp. 3111–3119.

[12] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[13] SciTools, "Scitools SciTools," https://scitools.com/features/.

[14] M. L. Soutner D., "Application of lstm neural networks in language modelling." *Lecture Notes in Computer Science*, vol. 8082, 2013.

[15] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *INTERSPEECH*, 2012.

[16] A. B. Ke Tran and C. Monz, "Recurrent memory networks for language modeling," in *Proceedings of NAACL-HLT 2016, pages 321–331*, 2016.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9(8), pp. 1735–1780, 1997.

[18] 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)*, 2019, pp. 29–36.

[19] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova, "Genome-scale computational approaches to memory-intensive applications in systems biology," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005, pp. 12–12.

[20] R. Jayaraj, G. Raymond, S. Krishnan, K. S. Tzou, S. Baxi, M. R. Ram, S. K. Govind, H. C. Chandramoorthy, F. N. Abu-Khzam, and P. Shaw, "Clinical theragnostic potential of diverse mirna expressions in prostate cancer: A systematic review and meta-analysis," *Cancers*, vol. 12, no. 5, p. 1199, 2020.

[21] J. R. Barr and T. Thatcher, "On the vulnerability of large corpora source code," in *2022 IEEE 16th International Conference on Semantic Computing (ICSC), 314-317*, 202.

[22] P. Gage, "A new algorithm for data compression," *Dr. Dobbs*, 1994.

[23] R. Sennrich, B. Haddow, and A. Birch, "Sentencepiece," 2020.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[25] J. Hillman and T. D. Hocking, "Optimizing ROC curves with a sort-based surrogate loss function for binary classification and changepoint detection," *CoRR*, vol. abs/2107.01285, 2021. [Online]. Available: https://arxiv.org/abs/2107.01285

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: improving the design of existing code, ser," in *Addison Wesley object technology series.* Addison-Wesley, 1999.

[27] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.