Complementing Machine Learning Classifiers Via Dynamic Symbolic Execution: "Human vs. Bot Generated" Tweets

Sohil L. Shrestha, Saroj Panda, Christoph Csallner
Computer Science and Engineering Department
The University of Texas at Arlington
Arlington, TX, USA
sohillal.shrestha@uta.edu,saroj.panda@mavs.uta.edu,csallner@uta.edu

ABSTRACT

Recent machine learning approaches for classifying text as human-written or bot-generated rely on training sets that are large, labeled diligently, and representative of the underlying domain. While valuable, these machine learning approaches ignore programs as an additional source of such training sets. To address this problem of incomplete training sets, this paper proposes to systematically supplement existing training sets with samples inferred via program analysis. In our preliminary evaluation, training sets enriched with samples inferred via dynamic symbolic execution were able to improve machine learning classifier accuracy for simple string-generating programs.

CCS CONCEPTS

• **Computing methodologies** → **Supervised learning**; • **Information systems** → *Social networking sites*; • **Theory of computation** → *Program analysis*;

KEYWORDS

Program output classification, program analysis, dynamic symbolic execution

ACM Reference Format:
Sohil L. Shrestha, Saroj Panda, Christoph Csallner. 2018. Complementing Machine Learning Classifiers Via Dynamic Symbolic Execution: "Human vs. Bot Generated" Tweets. In *RAISE'18 : IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3194104.3194111

1 INTRODUCTION

The basic question of whether a given program can produce a given string is undecidable in general. Practical instances of this question though are often much more complex. For example, in this paper we examine the problem of estimating if a given tweet has been human-written or machine-generated. As shown in Figure 1, this setting involves not one but a large number of programs. We assume we have access to the source code or binaries of a subset of both the tweet generators (*"bots"*) and libraries used by bots. These may be open source bots or standard bot libraries[1].

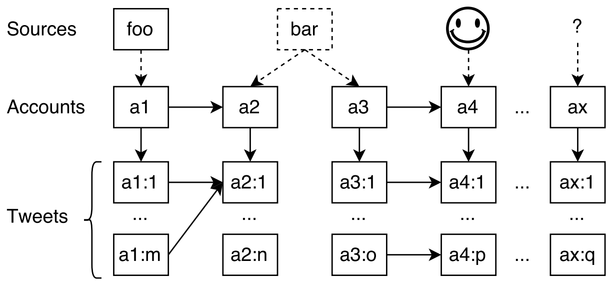

Figure 1: Human vs. bot tweet classification problem: Known items are shown as solid lines, unknown ones are dashed. For example, some Twitter accounts may use known bots (e.g., foo) or proprietary ones (e.g., bar). While much of the accounts, tweets, and their relations is known, the relation between accounts and sources is hidden.

To get a good approximate answer if a given output is program-generated, current approaches completely ignore the available source and binary code, side-step program analysis, and instead rely on a large number of signals that are external to the source code and binaries. In the Twitter example, state-of-the-art work combines over 1,000 signals into a machine learning classifier [30]. Examples of such external signals include how frequently and at what times a given Twitter account publishes tweets, how a tweet is related to other tweets (e.g., via retweets), or how the account interacts with other accounts on the Twitter platform (e.g., as a Twitter follower).

State-of-the-art work first labels each tweet in a training set as either "bot-generated" or "human-written". The labels are typically obtained through trusted parties [16, 20, 31] or collected using the APIs of social networks [21]. Then current work feeds the training tweets into a supervised machine learning algorithm and obtains a tweet classifier. Since labeling is expensive, training sets are relatively small in practice and therefore can only cover a small part of the behavior of tweet generators, which ultimately yields sub-optimal tweet classifiers.

This problem is significant in practice, as humans typically want to know if they are getting fed information produced by another human or by an algorithm. An extreme example is the recent 2016

RAISE'18 , May 27, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5723-4/18/05...$15.00
https://doi.org/10.1145/3194104.3194111

[1]Example libraries include the Microsoft Bot Framework (dev.botframework.com [Accessed Feb. 2018])

U.S. presidential election, where Twitter bots that posed as humans generated many tweets to influence the outcome of the election. But even in a more mundane scenario within the scope of software engineering, it may still be interesting if a comment or feedback about a piece of software has been written by a human user or has been auto-generated, to distinguish genuine user feedback from spam [11, 12, 25–27].

Recent work has made significant progress in using machine learning techniques to improve program analysis [3–5, 8, 9, 17]. As an example, recent work on integrating machine learning with symbolic execution has focused on using active machine learning as a custom search strategy in symbolic execution. For example, active learning [1] has been used to drive symbolic execution to learn the input-output specification of stream processing programs [3] and to learn method interface specifications [9]. Our work is different in that we aim to use program analysis to complement an established passive learning workflow that makes sense for a given domain (e.g., a workflow that builds classifiers with supervised (passive) learning from over 1,000 signals that are external to source code and binaries).

To bridge this gap, this paper proposes a technique for complementing an established passive machine learning (ML) workflow with program analysis. Specifically, our techniques can improve the quality of ML-inferred classifiers, by systematically and automatically enriching the sample sets ML-algorithms use to train their classifiers. Concretely, we perform dynamic symbolic execution to systematically and automatically explore program execution paths and obtain additional training samples. By definition, we can automatically label all such additional samples as "bot-generated", as they can be generated by the analyzed bot code.

Our preliminary experiments have indicated that this approach has the potential to improve the accuracy of machine learning inferred classifiers. To summarize, the paper makes the following contributions.

- The paper presents a technique for complementing an established passive machine learning workflow via program analysis.
- The paper reports on an initial experience that indicates that an instance of this approach can improve the accuracy of machine learning inferred classifiers.

This paper is presented in terms of C#, .Net, and the Pex tool [6, 10, 28]. But the underlying techniques are language-independent and can easily be implemented for other high-level languages and tools such as Java and Dsc [13, 14].

2 MINIMAL EXAMPLE

This section illustrates the approach using the hand-crafted minimal Listing 1 foo method. Suppose we have collected a training set of 1,500 tweets and have manually classified them as human-written or bot-generated. For example, this training set contains "bot" labeled texts "a 15" and "a 200", and "human" labeled texts "Not OK" and "a boy". From this training set a supervised ML technique has inferred a classifier that achieves on a fresh production set of 750 human-written plus 750 bot-generated tweets (which do not overlap with the training set) 95% precision, 50% recall, and an overall accuracy of 74%.

```
public String foo(int p) {
  if (p > 0)
    return "a" +" "+ p;  // Path 1
  return "b" +" "+ (−p); // Path 2
}
```

Listing 1: Minimal example bot program foo, which produces simple strings based on input parameter p.

While the initial training set yields a classifier of decent accuracy, we observe that this set consists of only two kinds of text. First, there are texts that cannot be produced by foo (e.g., "a boy"), since each program output starts with "a" or "b" followed by a number. Second, the remaining texts could have been produced by foo, but only by its path 1, since they all start with "a". In other words, since the training set does not sufficiently cover path 2, the resulting classifier will not capture outputs produced by path 2. Basically this classifier works well on examples that are similar to the training set, but struggles for examples produced by path 2.

To get a better classifier, we must add to our training set samples that cover path 2 and re-run the ML algorithm. While various program analysis techniques could be used to get such samples, in this paper we focus on directed path exploration using dynamic symbolic execution (DSE). With DSE we systematically enumerate foo's execution paths, generate output samples for each such enumerated path, and automatically label each such sample as "bot-generated". In many cases (e.g., for foo), DSE achieves high path coverage. We can thus use DSE to systematically and automatically complement the initial training set.

For the foo example, our naive Bayes ML algorithm yields classifier C1. Then we use dynamic symbolic execution to explore foo and produce 375 additional program outputs that cover path 2. Rerunning our naive Bayes yields classifier C2. When comparing C1 and C2 on a single production set of tweets that does not overlap with any training sample, the DSE-enriched classifier C2 achieved higher precision, recall, and accuracy, e.g., yielding a total accuracy of 98% (vs. C1's 74% accuracy). C2's higher accuracy makes intuitively sense, since C2 was trained on a more comprehensive training set, consisting of C1's training set plus additional DSE-inferred samples that also cover path 2.

3 BACKGROUND: TWEETS, ML, AND DSE

This section contains necessary background information on social media bots, text classification, supervised machine learning, and dynamic symbolic execution.

3.1 Bots And Social Media

Recent years have seen the explosive growth and adoption of social media platforms such as Facebook and Twitter. Twitter has some 300 million monthly active users[2] whereas Facebook has 2 billion[3]. Both platforms are used by politicians, the traditional mass media,

[2]"Twitter reports 6 pct increase in monthly active users", 26 Apr. 2017, reuters.com/article/twitter-results/twitter-reports-6-pct-increase-in-monthly-active-users-idUSL4N1HY48L [Accessed Feb. 2018]

[3]Josh Constine. "Facebook now has 2 billion monthly users…and responsibility", 27 June 2017, techcrunch.com/2017/06/27/facebook-2-billion-users/ [Accessed Feb. 2018]

companies of all sizes, celebrities, and other thought leaders. Messages on these platforms can go viral and exert vast influence, e.g., by influencing national politics and election outcomes.

While these platforms support various kinds of media, we focus here on text messages such as tweets, since text is relatively easy to analyze. Although many social media bots are benign, there are malignant bots that may distribute spam or malicious messages, e.g., to manipulate public opinion through fake news or propaganda. Classifying these text messages with high accuracy into human-written or bot-generated is an open research problem [30].

3.2 Text Classification And Supervised ML

Text classification sorts documents into predefined categories [7], such as our two categories of human-written and bot-generated. A popular text classification method is supervised machine learning [23], which has been used successfully, for example, to analyze sentiments [2] or to detect spam email [22].

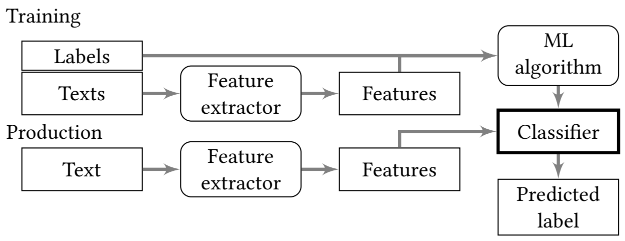

Figure 2: Supervised ML. In the machine learning literature, the production setting is also called "testing".

Figure 2 summarizes a typical classification process based on supervised ML. Provided categorized (or *labeled*) texts, typical approaches perform standard pre-processing steps such as tokenization, removing stop words ('a', 'the', etc.) that are often not significant for classification, and word stemming to convert each word into its canonical form (e.g., mapping "going" to "go"). We have not removed stop words in the experiment involving Listing 1 minimal example as 'a' holds significance while training the classifier and stemming of words is not performed in both of our preliminary experiments.

A key aspect of text categorization is that the number of features (documents or words) can be large. To address this problem, dimension reduction techniques are used, such as choosing a feature subset or mapping features to a new feature set. Common subset methods include term frequency-inverse document frequency (TF-IDF), chi-square, and information gain. For example, to filter out common words, a word receives high TF-IDF if it has high frequency within the given text but a low frequency within the entire corpus.

For text classification, a number of well-known algorithms have been proposed, including k-nearest neighbor (k-NN), Naive Bayes, and support vector machines (SVM). While SVM and k-NN algorithm require parameter tuning for high-quality results, we choose Naive Bayes for our preliminary experiment because of its simplicity and good performance for the task at hand [15].

3.3 Dynamic Symbolic Execution (DSE)

Dynamic symbolic execution (DSE) or concolic execution is a form of symbolic execution that uses an additional concrete program execution to decide branch conditions during symbolic execution, to automatically and efficiently enumerate a program's execution paths [6, 10, 28]. As a DSE engine follows a concrete program execution, the DSE performs a standard concrete execution as, e.g., in the case of analyzing a Java program, any stock Java virtual machine would do. But in addition to this concrete execution, the DSE engine also maintains a symbolic representation of the program state and records each branching decision outcome in terms of expressions over symbolic versions of the program inputs.

As any symbolic execution tool, a DSE engine encodes a program path as the conjunction of branch conditions. To explore another path, the tool can manipulate these branch conditions, i.e., by flipping one of the conditions, discarding the subsequent conditions, and feeding the resulting conjunction to an off-the-shelf constraint solver.

The constraint solver (typically a SMT solver) searches for a variable assignment that satisfies the conjunction. The DSE engine then maps the satisfying assignment back to a set of new program input values. DSE is widely used for program analysis and test case generation.

4 OVERVIEW AND DESIGN

At a high level, the goal is to address ML's well-known GIGO (garbage in—garbage out) problem for the special case in which a program is available to produce additional inputs. No matter how fancy the ML algorithm, ML will produce a low-quality classifier, if it is stuck with a bad or incomplete training set. At its heart, an ML algorithm builds a map from text input to label output, based on properties of the training set. If the training set does not represent a domain entirely, the inferred mapping function will have similar holes.

Even when handed a training set without obvious holes, machine learning can still produce bad classifiers if the training set is merely too small [19]. There is no hard and fast rule that suggests the amount of training data required to train a model. Yet this number depends on both the problem complexity (i.e., the unknown underlying function we are trying to derive) and the complexity of the machine learning algorithm. In summary, the larger and the more representative the training set, the better the resulting classifier.

4.1 System Design

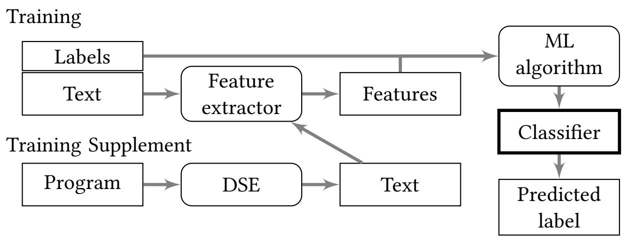

Figure 3: Overview and design

Figure 3 gives an overview of the major components of our approach. In the current setup the only input to our machine learning algorithm is the text. Adding additional signals such as re-tweets, the user's popularity, or the user's posting frequency is subject to future work.

In addition to the training sequence of standard machine learning algorithms shown in Figure 2, our approach has an additional source of training data. Specifically, our approach infers additional training samples from the given program with dynamic symbolic execution.

Although dynamic symbolic execution is computationally expensive and has well-known scaling challenges when faced with loops and complex program conditions, DSE still offers a systematic technique for enumerating many feasible execution paths and often achieves high code coverage. Future work includes replacing DSE with random or search-based test case generation.

4.2 Adapting DSE to Generate Rich Labeled Training Sets

A key observation is that, by default, existing off-the-shelf DSE engines such as Pex are not a good fit for producing rich training sets that are well-suited for ML-training algorithms. The reason for this discrepancy is that DSE engines aim at producing minimal test suites, where each of the program's execution paths is covered by exactly one test case. However this is not a good fit for ML training algorithms, since standard algorithms such as naive Bayes work better if they can work with much more samples per execution path.

To address this mismatch between DSE engines and ML training requirements, we modify the standard DSE process, to emit a configurable number of samples per execution path. Concretely, for each path condition the DSE engine's constraint solvers consider feasible, we do not only request a single solution from the DSE engine's constraint solvers, but a sequence of solutions.

The current prototype implementation simulates this behavior, by annotating the program each time Pex produces a new test case. For example, if Pex produces a test case that feeds parameter value p==1 to the program, our technique annotates the program to ensure that "p!=1" holds for all subsequent Pex explorations and generated test cases.

This work-around is highly inefficient, as it forces Pex to re-execute the program for each sample, even if all samples trigger the same execution path. Re-analyzing each path more than once is not necessary, since such a "p!=1" constraint can be passed to the constraint solver directly, without re-executing the program.

5 INITIAL EXPERIENCE

While a full evaluation is subject to future work, in this paper we focus on the core question if we can use dynamic symbolic execution to systematically improve the accuracy of an existing passive learning scheme for the special case in which the setup contains only a single bot whose source code or binaries are available for program analysis. In short, here the research question is if, for a given program, we can use DSE to strengthen a machine learning inferred classifier.

To explore this research question, we use the following standard machine learning notions of precision (p), recall (r), accuracy (acc), and F1 score ($f1$).

$$p = \frac{TP}{TP + FP} \quad r = \frac{TP}{TP + FN} \quad acc = \frac{TP + TN}{total} \quad f1 = 2 * \frac{p * r}{p + r}$$

When evaluating a classifier Cx, we apply Cx on unlabeled ("production" or "test") text that is different from the training text samples.

5.1 Results for Motivating Example

For the Listing 1 foo method example, Table 1 shows the results of applying 750 foo-generated and 750 human-written production texts to both C1 and C2. C1 correctly classified 375 foo-generated (true positives, TP) and 728 human-written (true negatives, TN) texts. But C1 falsely classified 22 human-written texts as foo-generated (false positives, FP) and 375 foo-generated texts as human-written (false negatives, FN).

	C1-predicted		C2-predicted	
Actual	Bot	Human	Bot	Human
Bot	TP=375	FN=375	750	0
Human	FP=22	TN=728	34	716

Table 1: Results of applying 1,500 example production texts to classifiers C1 and C2 of the Listing 1 foo method.

The main change between C1 and C2 is that C2 had a DSE-enhanced training set. With this more complete training set, C2's recall improved to 100%, with zero false negatives. Overall, the DSE-enhanced classifier C2 correctly classified 1,466 of 1,500 texts, yielding a 98% accuracy (vs. C1's 74%).

5.2 Sentence Generator Subject

To get a data point besides Listing 1, we applied our approach on the Listing 2 random sentence generator. Based on the "type" parameter, the program randomly generates variable-length sentences, from a (predefined) list of positive words, a list of negative words, or from both lists.

```
Random rand = new Random();
string[] pos = {"able", "adorable", "ageless" /*...*/};
string[] neg = {"aloof", "altercation", "ambiguity" /*...*/};
string[] words = {"I", "welcome", "you" /*...*/};
public string createText(int numWords, int type) {
  string outText = "";
  string randomWord = words[rand.Next(0, words.Length)];
  for (int i=0; i<numWords; i++)
  {
    outText += randomWord + " ";
    if (type == 1)
      randomWord = pos[rand.Next(0, pos.Length)];
    else if (type == 0)
      randomWord = neg[rand.Next(0, neg.Length)];
    else if (rand.Next(1,50000)%3==1)
      randomWord = pos[rand.Next(0, pos.Length)];
    else if (rand.Next(1, 50000) % 3 == 1)
      randomWord=neg[rand.Next(0, neg.Length)];
    else
      randomWord = words[rand.Next(0, words.Length)];
  }
  return outText;
}
```

Listing 2: Random sentence generator (excerpt).

Similar to Listing 1, we first trained classifier C3 on a labeled training set. This training set only contained samples for one value of the type parameter (and therefore a subset of all execution paths), plus samples that cannot be created by the program. Classifier C4 is trained on C3's training set, enhanced with samples generated with the other relevant values of the type parameter, yielding a more representative sample of execution paths and program-generated sentences.

Classifier	Accuracy	Precision	Recall	F1 Score
C3	96.5	100	93.1	96.4
C4	99.4	100	98.8	99.4

Table 2: DSE strengthened the highly-accurate C3 classifier, which yielded C4.

Table 2 shows the results of evaluating C3 and C4 on a 1,500 sentence production set (which did not overlap with the training sets). Both C3 and C4 achieved very high accuracy. This high accuracy levels may be caused by the primitive nature of the Listing 2 sentence generator. This lead to a stark difference between human and bot produced sentences, which enabled even the C3 classifier to be 100% precise and achieve over 90% recall.

While this example is certainly extreme, it illustrates an interesting point: Even for a classifier that is already very accurate, DSE has the potential to strengthen it further.

6 RELATED WORK

The most closely related work is an earlier combination of machine learning and program analysis by Brun and Ernst [5]. They present a technique to find latent code errors in programs. The technique uses a ML model trained on properties of erroneous programs and fixed program versions generated by program analysis. The fault-revealing model then takes program properties as input and selects the one that is likely to indicate program errors.

Recent work by Davies, Păsăreanu, and Raman combines concolic execution, machine learning, and function fitting to generate system-level inputs with the goal of increasing test coverage in the aerospace domain [8]. Other recent work by Li et al. proposes a new symbolic execution tool that is driven by machine learning based constraint solving, to handle complex path conditions such as function calls [17]. Likewise, Botincan and Domago have used active learning to drive symbolic execution to learn the input-output specification of stream processing programs [3].

While earlier works such as the above have explored combinations of machine learning and program analysis, they have focused on using machine learning to augment program analysis. While this is a very promising research direction, we take the inverse direction and use program analysis to strengthen existing machine learning techniques.

More specific to the problem of sub-optimal classifiers due to the unavailability of labeled training data, an extensively studied and used machine learning technique is active learning [24]. Active Learning is a specialized semi-supervised technique in which a classifier selectively queries the label of the most informative unlabeled data instance, based on different querying strategies, from the user. Unlike classical passive supervised learning algorithms, active learning reduces the labeling effort by choosing a small subset of unlabeled data to re-train the classifier.

Previous work by Tong and Koller [29] as well as Melville and Sindhwani [18] presented ways to reduce the labeling cost of training data. Tong and Koller presented an algorithm to perform pool-based active learning with Support Vector Machines for the text classification problem. Melville and Sindhwani focus on actively selecting the most informative examples and features for labeling.

While our work also has been motivated by the unavailability of comprehensive labeled data that yields sub-optimal classifiers, our work leverages source or binary code via program analysis, to supplement labeled training examples rather than using unlabeled data.

7 FUTURE WORK

The current experimental setting is severely limited, as it assumes that there is only a single bot and we have access to the bot's source or binary code. Future work includes more realistic experiments that are more in line with the original problem setup depicted in Figure 1, i.e., involving multiple bots and bot libraries where only some of them are available for program analysis.

In practice we may not have access to most bots' source or binary code. However even proprietary bots are likely written on top of common libraries. We often have access to these libraries as many of them are widely available, e.g., as open source libraries. So an interesting direction for future work is designing a two-step process. In the first step we analyze the code of a set of common libraries in depth and infer the traits that characterize the libraries. In the second step we then try to observe these libraries' traits in text samples.

Future work also includes exploring alternative program analysis techniques in addition to dynamic symbolic execution (e.g., random or search-based test case generation), to explore additional program execution paths.

8 CONCLUSIONS

Current machine learning approaches ignore programs as a source of such training sets. To address the problem of low-quality machine learning classifiers caused by small or incomplete training sets, this paper proposed to systematically supplement training sets with samples inferred by program analysis. In our preliminary evaluation, training sets enriched with samples inferred via dynamic symbolic execution could significantly improve machine learning classifier accuracy.

REFERENCES

[1] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106.

[2] Erik Boiy and Marie-Francine Moens. 2009. A machine learning approach to sentiment analysis in multilingual web texts. *Information Retrieval* (2009), 526–558.

[3] Matko Botincan and Domagoj Babic. 2013. Sigma*: Symbolic learning of input-output specifications. In *Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 443–456.

[4] James F. Bowring, James M. Rehg, and Mary Jean Harrold. 2004. Active Learning for Automatic Classification of Software Behavior. In *Proc. 2004 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 195–205.

[5] Yuriy Brun and Michael D. Ernst. 2004. Finding Latent Code Errors via Machine Learning over Program Executions. In *Proc. 26th Intl. Conference on Software Engineering (ICSE)*. IEEE Computer Society, 480–490.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically generating inputs of death. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*. 322–335.

[7] Soumen Chakrabarti, Shourya Roy, and Mahesh V. Soundalgekar. 2003. Fast and accurate text classification via multiple linear discriminant projections. *The VLDB Journal* 12, 2 (Aug 2003), 170–185.

[8] Misty Davies, Corina S. Păsăreanu, and Vishwanath Raman. 2012. Symbolic Execution Enhanced System Testing. In *Proc. 4th Intl. Conference on Verified Software: Theories, Tools, Experiments (VSTTE'12)*. Springer, 294–309.

[9] Dimitra Giannakopoulou, Zvonimir Rakamaric, and Vishwanath Raman. 2012. Symbolic Learning of Component Interfaces. In *Proc. 19th International Static Analysis Symposium (SAS)*. Springer, 248–264.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 213–223.

[11] Emitza Guzman, Rana Alkadhi, and Norbert Seyff. 2017. An exploratory study of Twitter messages about software applications. *Requirements Engineering* 22, 3 (Sept. 2017), 387–412.

[12] Emitza Guzman, Mohamed Ibrahim, and Martin Glinz. 2017. A Little Bird Told Me: Mining Tweets for Requirements and Software Evolution. In *Proc. 25th IEEE International Requirements Engineering Conference (RE)*. IEEE, 11–20.

[13] Mainul Islam and Christoph Csallner. 2010. Dsc+Mock: A test case + mock class generator in support of coding against interfaces. In *Proc. 8th Intl. Workshop on Dynamic Analysis (WODA)*. ACM, 26–31.

[14] Mainul Islam and Christoph Csallner. 2014. Generating test cases for programs that are coded against interfaces and annotations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (May 2014), 21:1–21:38.

[15] Aurangzeb Khan, Baharum Baharudin, Lam Hong Lee, and Khairullah Khan. 2010. A Review of Machine Learning Algorithms for Text-Documents Classification. *Journal of Advances In Information Technology* 1, 1 (Feb. 2010), 4–20.

[16] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[17] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving. In *Proc. 31st IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*. ACM, 554–559.

[18] Prem Melville and Vikas Sindhwani. 2009. Active Dual Supervision: Reducing the Cost of Annotating Examples and Features. In *Proceedings of the NAACL HLT 2009 Workshop on Active Learning for Natural Language Processing (HLT '09)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 49–57. http://dl.acm.org/citation.cfm?id=1564131.1564142

[19] Sarunas J. Raudys and Anil K. Jain. 1991. Small Sample Size Effects in Statistical Pattern Recognition: Recommendations for Practitioners. *IEEE Trans. Pattern Anal. Mach. Intell.* 13, 3 (March 1991), 252–264.

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[21] Matthew A. Russell. 2011. *Mining the Social Web: Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites* (1st ed.). O'Reilly Media, Inc.

[22] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. 1998. *A Bayesian Approach to Filtering Junk E-Mail.* Technical Report WS-98-05. AAAI.

[23] Fabrizio Sebastiani. 2002. Machine Learning in Automated Text Categorization. *ACM Comput. Surv.* 34, 1 (March 2002), 1–47.

[24] Burr Settles. 2009. *Active Learning Literature Survey.* Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

[25] Abhishek Sharma, Yuan Tian, and David Lo. 2015. NIRMAL: Automatic identification of software relevant tweets leveraging language model. In *Proc. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 449–458.

[26] Abhishek Sharma, Yuan Tian, and David Lo. 2015. What's hot in software engineering Twitter space?. In *Proc. 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 541–545.

[27] Leif Singer, Fernando Marques Figueira Filho, and Margaret-Anne D. Storey. 2014. Software engineering at the speed of light: how developers stay current using Twitter. In *Proc. 36th International Conference on Software Engineering (ICSE)*. ACM, 211–221.

[28] Nikolai Tillmann and Peli de Halleux. 2008. Pex - White box test generation for .Net. In *Proc. 2nd Intl. Conference on Tests And Proofs (TAP)*. Springer, 134–153.

[29] Simon Tong and Daphne Koller. 2002. Support Vector Machine Active Learning with Applications to Text Classification. *J. Mach. Learn. Res.* 2 (March 2002), 45–66. https://doi.org/10.1162/153244302760185243

[30] Onur Varol, Emilio Ferrara, Clayton A. Davis, Filippo Menczer, and Alessandro Flammini. 2017. Online Human-Bot Interactions: Detection, Estimation, and Characterization. In *Proc. 11th Intl. Conference on Web and Social Media (ICWSM)*. AAAI, 280–289.

[31] R. Zafarani and H. Liu. 2009. Social Computing Data Repository at ASU. (2009). http://socialcomputing.asu.edu