SeqGAN-APG: Sequential Generative Adversarial Networks for Automatic Patch Generation

Bradley Baker
University of New Mexico
Albuquerque, New Mexico
bbradt@unm.edu

ABSTRACT

As software systems grow in size and complexity, the task of manual debugging consumes more time in the development cycle, and opens up to more serious and frequent human error. To address this problem, researchers have developed systems for automated patch generation, which attempt to reduce the manual work required for diagnosing and addressing faulty code. Machine Learning models, which can leverage large amounts of information, for example from large code repositories, promise one possible set of solutions for automated patch generation. In particular, researchers have recently used sequential generative adversarial networks (GANs) successfully in text-based prediction tasks. We thus propose a model for sequential generative adversarial networks for automated patch generation (SeqGAN-APG) which divides the task of patch generation into a two-part adversarial framework which learns the distribution of sequential features in non-buggy code while also learning to generate sequential features which mimic that distribution. Our approach is novel not only in terms of its application of GANs to an untested kind of sequential data, but also in terms of the kinds of sequential features which we implement, going beyond the typical token-based approach.

KEYWORDS

Automatic Patch Generation, Generative Adversarial Networks, Deep Learning

ACM Reference format:
Bradley Baker. 2017. SeqGAN-APG: Sequential Generative Adversarial Networks for Automatic Patch Generation. In *Proceedings of , , ,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As software systems grow in size and complexity, the task of manual debugging consumes more time in the development cycle, and opens up to more serious and frequent human error. So far in 2017, Mozilla has verified 576 newly reported bugs for the Firefox web-browser [2], a complex piece of software containing nearly 18 million lines of code [3]. Though many discovered bugs may only affect the system at a superficial level, some faults can expose serious security vulnerabilities, and thus require immediate attention and resolution - an often tall order in large and complex software systems. In the past decade, researchers have thus begun to turn toward automatic bug-detection and repair in an effort to address the rising difficulties of manual debugging [5]; however, few of the proposed methods have made use of recent advances in machine learning and artificial intelligence in order to leverage information regarding software structure, history, or other relevant data.

As machine learning has been celebrating a wide array of novel applications, many researchers have applied machine learning models to problems in software engineering and design, leveraging large repositories of code as data sets to solve various problems. For example, one popular line of research has translated solutions for movie recommender systems to help Integrated Developing Environments predict API calls while writing programs [7, 9, 33]. Only recently, however, researchers have started to leverage large code repositories to help in automated patch generation and other security-related problems. In this paper, we provide a contribution towards this effort in the form of a Sequential Generative Adversarial Network for Automated Patch Generation (SeqGAN-APG). Our proposed framework utilizes a discriminative model trained in an adversarial setting to learn the underlying distribution of novel sequential features extracted from buggy and non-buggy programs, while the associated generative adversarial model attempts to learn how to generate the kinds of sequential features characteristic to non-buggy code. This represents a novel approach to deep learning for patch generation, contrasting with the auto-encoder based method recently implemented in the Deepfix Patch Generation framework [16].

Though our implementation currently lacks some final details and sufficient time to learn thorough models, which will establish SeqGAN-APG as a full pipeline, our implemented discriminative and generative models do provide sequential predictions for the target distribution of non-buggy programs. In addition to the standard approach of utilizing sequences of tokens to represent program data, our model utilizes a novel application of AST-based sequential features for generating non-buggy code samples, thus taking an important step into the application of alternative sequential features which may better capture the rigid syntactical structures at work in programs.

2 BACKGROUND

In this section, we present some of the background information at work in our presentation of SeqGAN-APG.

2.1 Automated Bug Repair

We can divide the problem of automatic bug repair into a number of sub-problems: ”failure detection (something wrong has happened), bug diagnosis (why this has happened), fault localization (what the root cause is), [and] repair inference (what should be done to fix the problem)” [26]. In addition to this taxonomy, ”repair inference” can be extended to include ”patch generation”[1] if the final

,
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

[1]These methods often target security-related goals [10, 34]

product of the model is a full software patch intended to produce a concrete resolution for a bug or set of bugs within software. In the past, many researchers have approached each of these sub-problems individually (see, for example [11, 18, 22, 24, 27]), while some frameworks attempt to address multiple problems at once, often performing both localization and repair (e.g. [23]). Historically, repair inference has perhaps most frequently been addressed using genetic programs ([6, 14, 17, 20, 21, 36]), although a number of other approaches utilize other methods such as solving problems in Satisfiability Modulo Theory [12, 37], or semantic analysis [28]. Most recently, some state of the art methods have taken some first steps into data mining [19] and even some basic probabilistic methods [23]; however, the problem of bug repair via patch generation has been largely untouched by machine-learning, and thus deserves further exploration.

Our approach, insofar as it incorporates a discriminative model in association with a generative model for creating patches sits at the intersection of failure detection and patch generation, with additional work remaining in fault localization to apply generated patches to buggy code.

2.2 Program Representations for Machine Learning

In cases where researchers use program repositories as data for machine learning models, the problem presents itself - how do we represent a program so that a model can successfully leverage the information included therein?

Following work in natural language processing, one popular set of approaches encodes program features in terms of sequences and subsequences of tokens present in the program. A simple bag-of-words approach, for example, indicates the presence of particular tokens in a program, without including any sequential information, while a more sophisticated approached might encode the presence of n-length grams, i.e. repeated patterns of tokens. Researchers have used these and other highly sparse, token-based features to successfully encode programs for analysis purposes. For additional discussion of text representations in machine learning see [4, 32].

Another popular representation of program information is the program's abstract syntax tree (AST). The AST of a program provides a graph-based representation of relationships between particular syntactical objects in the program, thus capturing more detailed information about a program's underlying structure, such as relationships between variable types, the scope of certain statements and structures, etc.

Finally, programs as a data objects often contain a number of additional features not included in text data. One could very well include program creation or modification timestamps, input-output evaluations, runtime information, or other program data as features of P. While a large body of work in machine-learning focuses on text-representations alone, the frontier of feature engineering and selection for program data remains wide open for both sequential and non-sequential feature types.

2.3 Generative Adversarial Networks

The underlying model which motivates our proposal of SeqGAN-APG is a Generative Adversarial Network (GAN) [15]. A generative adversarial network is a *joint* learning framework involving two deep neural networks, which trains both models in a joint setting to perform two different, but related tasks: one model, the generative model, attempts to emulate the distribution of a given target data set; the second, discriminative model, attempts to estimate the probability that a given sample came from a training data set rather than from D. The generative model, attempts to act as an adversary for the discriminator, using reinforcement learning techniques to attempt to learn how to 'fool' the discriminator, thus hopefully increasing the robustness of the discriminator as well as the ability of the generator to replicate the target distribution.

Researchers have applied GANs to a number of interesting generative tasks, mostly centered around image generation (e.g. [13]) and other real-valued data. It is only very recently that researchers have begun to explore the application of generative adversarial networks to discrete, sequential information, such as text [30, 39]. The SeqGAN [38] model in particular has illustrated that utilizing Recurrent Neural Networks along with some tweaks to gradient computations for the generative task can provide robust sequence predictions. The idea behind SeqGAN-APG is to use this sequential generative adversarial network to generate sequential samples which mimic the distribution of non-buggy programs, while also training a discriminator which is attempting to distinguish between distributions of buggy and non-buggy programs.

3 METHODOLOGY

In this section we describe the underlying methodology used to formulate, build, and evaluate SeqGAN-APG.

3.1 Model Formulation

The problem of automatic patch generation bears closest resemblance to structured sequence prediction, insofar as the underlying generator strives to predict the correct sequence of tokens which will replace a buggy snippet of code. As mentioned before, we want to train a Sequential Generative Adversarial Network for Automatic Patch Generation (SeqGAN-APG), so we split the task of automatic patch generation into 1) buggy vs. non-buggy discrimination, and 2) sequential feature generation.

Buggy/Non-Buggy Discrimination

Let P be a program or snippet represented by a given set of features $\{F_i\}$ such as ordered sequences of tokens, input-output specifications, runtime information, etc. Additionally, associate a binary label L to each program indicating whether the program does ($L == 1$) or does not ($L == 0$) contain a bug. The basic task of automatic patch generation posits a model M which best emulates the map $M : \{P_{L==1}\} \rightarrow \{P_{L==0}\}$.

The majority of previous approaches to automatic patch generation select a "best" candidate patch from a restricted search-space generated from restricted transformations on the features of P. The model in [23], for example, learns a probabilistic ranking over the features of successfully patched data in order to rank an automatically generated search space for P, created by permuting specific features of P, such as the structure of the Abstract Syntax Tree, or sequences of tokens.

While restricting models to predictions on restricted search-spaces can help to retain certain structures on candidate patches, it does so at the expense of excluding possibly successful candidates not available in the search space. Our approach, in contrast, strives to learn the underlying distribution of features in buggy and non-buggy programs, and to use this discriminatory power to drive an underlying generative process for generating a non-buggy program given a set of features for a buggy program. Thus, our approach generalizes from learning a particular map on a known or assumed set of distributions to learning a model which learns estimates of the underlying distributions in order to predict whether or not a given program sample represents a buggy or non-buggy problem. This first task is thus allocated to the *discriminative* part of the Sequential Generative Adversarial Network.

Sequential Feature Generation

We choose to adopt the intuition that many program features include sequential information which may prove important in characterizing the distribution of buggy and non-buggy programs, and to treat the problem of patch generation as a kind of structured sequence generation. This idea follows the example of Deepfix's sequence-to-sequence patch generation [16], which posits a model for translating sequences of buggy tokens to sequences of non-buggy tokens using the Seq2Seq deep auto-encoder [25]. Though the underlying network structure of SeqGAN-APG is fundamentally different than Deepfix, we approach patch generation in fundamentally the same way. Given a buggy program or snippet P, with a set of sequential features $X_1, ..., X_n$, and a set of non-sequential features $\{F_i\}$, our model attempts to predict the best sequence of target features $Y_1, ..., Y_n$ such that the resulting patched program best resembles the learned distribution of non-buggy programs. Doing this in a joint setting allows the generative model to generate these sequential features (such as sequences of tokens or AST-nodes) to best create non-buggy programs or snippets after being given a set of buggy programs as input. Additionally, the sequential nature of the features mean that if a fault is localized, the trained generative model should be able to start at a location prior to the localized fault and generate a set of sequential features, such as correct tokens, which represent a fix to that localized issue.

Model Implementation Details

Due to space constraints we have assembled the parameters and architectural details used to build the SeqGAN-AST network in appendix A.

3.2 Data Processing

Our chosen data set for evaluation is the CodeFlaws data set [35], a recently released set of labeled, buggy and non-buggy programs written in the C programming language. Each file in the CodeFlaws data has a label of either buggy or non-buggy, includes a set of nine input-output evaluations, and matches with a dual file of the opposite label written for the same program functionality. Additionally, each buggy file contains a label identifying the particular kind of bug under the codeflaws proposed taxonomy of bugs [35], which means we could possibly extend learning the binary discriminative task of buggy and non-buggy programs to learn the distributions of features characterizing certain *kinds* of bugs.

Prior to analysis, we remove include statements and comments from each file, assuming that bugs are not located in the usage of particular headers, and aren't influenced by information in comments. This also restricts the vocabulary to only statements which are parsed on compilation, and which do not require the linking of external libraries or header files. We then format the C files so that the contents of each file are all located on a single line, and assemble positive and negative files into separate data sets, where each row in the data set represents one file, i.e. one instance of a bug. Thus, we do not perform any kind of bug localization, treating the entire program as a positive or negative instance of the bug, rather than extracting snippets or labeling lines separately.

This initial approach to data representation requires the length of input and output sequences to be quite long (between 100 and 400 tokens on average), which makes the generation task difficult. This lack of bug localization means we cannot currently associate generated samples with particular fault locations. This major shortcoming short-circuits the final step of the patching pipeline, and represents a major point of remaining work.

Token-Sequence Representation

After assembling C files into the separated data sets as described above, we first generate a data set which encodes programs as sequences of tokens. In each line of the positive and negative files, we first convert all recognized C operators into unique tokens, using the intuition that each operator has a semantic meaning in a similar fashion to C keywords. We then use a pre-built vocabulary of C keywords and operators (based on [1]) to convert tokens into unique hashes, and then extend this vocabulary to hash variable names, constants, literals, and other unique values which appear throughout the program repositories. The resulting sequence of unique hashes, along with the vocabulary for the entire repository, constitute the tokenized data set.

AST-Node-Sequence Representation

In addition to the tokenized representation of program data, we also assemble a data set based on parsing the Abstract Syntax Trees for each program. For each program, we run the Python C-AST Parser [8] in order to generate the associated AST. Once we've generated each AST, we then flatten the AST into an ordered sequence of AST nodes. To do this, we recursively parse through the entire AST in a depth first fashion, adding each encountered node to the sequence and re-encoding edges to children as sparse features in the node itself. We then use the Sklearn vectorizor [29] to convert each node to a set of hashes which encode the unique, nominal features of the given, such as the node's type, its links to other nodes in the AST, or its particular syntactical attributes such as expression or data type. The resulting data set is a tensor with a number of data instances equal to the number of files, with each instance encoding a sequence of feature vectors corresponding to the node located at the given point in the sequence.

This approach to AST-to-Sequence conversion is inspired by the approach outlined in [31], with the essential difference being that we do not convert nodes to program actions prior to parsing.

3.3 Evaluation Protocol + Baselines

Due to the lack of a mechanism for bug localization and patch insertion, to evaluate the success of the generative model, we qualitatively compare the kinds of sequences generated from each model respectively. We compare SeqGAN with an implementation of the

Seq2Seq deep learning library [25], which is the underlying network at work in the Deepfix Automatic Patch Generation suite [16]. We evaluated both SeqGAN-APG and Seq2Seq on both Tokenized and AST-based data sets, and used the same preprocessing steps for both models, except for vocabulary hashing, which Seq2Seq handles internally. We have outlined the detailed parameters for running the Seq2Seq baseline have been outlined in appendix A. In the future, we would like to implement the full Deepfix Automatic Patch Generation suite [16], in addition to other smaller suites used for automatic patch generation; however, since we have not currently resolved the localization sub-problem for our model, this comparison is currently not possible.

In addition to the Generator Baselines, we also wanted to compare the discriminative model with a suite of text classifiers implemented from the sklearn library [29]. We implemented Logistic Regression without normalization and with Elastic Net Regularization, Ridge Regression, a Linear Support Vector Machine with and without feature selection, a simple perceptron classifier, a linear passive-aggressive classifier, bernoulli and multinomial naive-bayes classifiers, a k-nearest-neighbors classifier, a nearest centroid classifier, and a random forest classifier. We performed 10-fold cross validation on all models, and record the accuracy for the binary classification task. We did not perform any hyper-parameter searches for the models, with the chosen hyper-parameters for each model are specified in appendix A. We evaluated the suite of classifiers first on the hashed tokenized text data, and then subsequently on the hashed AST sequences. Unfortunately, due to unresolved issues in the SeqGAN library, and lack of time to invent patches for the software, it was not possible to also output accuracy for the SeqGAN model. Nonetheless, the results indicate differences between the two data processing methodologies, and remain interesting for future analysis. Regardless, without sufficient time to train the networks, or run hyperparameters, it is not likely the more complex models would compare well with the simpler models which we could implement and evaluate within a shorter time-frame.

All tests were run in a 64-bit Ubuntu 17.04 environment with 8 GB of RAM and an Intel(R) i7-4790 core clocked at 3.60GHz. SeqGAN-APG and Seq2Seq were trained using CUDA on an Nvidia GeForce GTX 980 Ti GPU with 6 GB of memory.

4 RESULTS AND DISCUSSION

In this section, we present the initial results for SeqGAN, with the hope of comparing with the various baselines discussed in the previous section.

How does SeqGAN perform in the discriminative task? Figure 1 compares the accuracy measure for a suite of classifiers tested on both tokenized and AST data sets. All classifiers perform better when utilizing the AST-based data. Unfortunately implementation issues in the SeqAST library, which does not by default support accuracy evaluation, prevented direct comparison with the discriminator at this time.

What kinds of sequences do the generative models output?

The localization issue is not the only obstacle preventing the output of currently viable patches for both generators: both deep networks require significant training time (on the order of days),

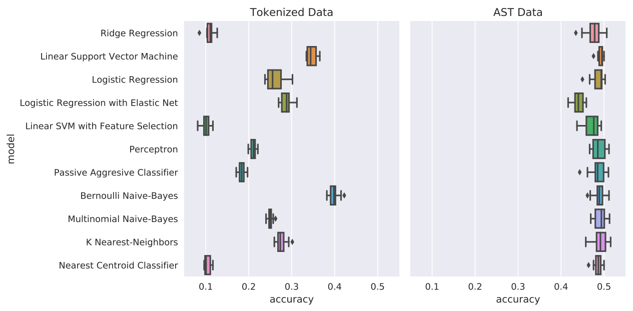

Figure 1: Comparison of Discrimination Accuracy on Different Data Processing Methodologies

as well as robust parameter tuning in order to generate viable patches, or even just viable sequences. Thus, the complexity of the models is itself a barrier to generating robust output after a short training timeframe (only 1000 or 2000 iterations, for example, is not sufficient), which is a common characteristic of most deep-learning frameworks. Access to more robust machines with more GPUs, in addition to a longer timeframe is required to generate the sequences in a more robust ways. Both models, after 1000-2000 iterations, only output sequences of operators. Thus, though the full pipelines do 'work', desirable results would require a longer time-frame.

5 CONCLUSION

In this paper, we proposed a novel Sequential Generative Adversarial Network approach to Automated Patch Generation (SeqGAN). In contrast to recent approaches in automated patch-generation, our approach utilizes an adversarial setting to learn how sequential features characterize buggy and non-buggy programs, and then use these features to build sequences of features which represent non-buggy.

In addition to the standard representation of programs as sequences of tokens, we implemented a novel approach to model the sequential features of a program's AST nodes. Current results indicate that AST-based features help simple sequence-based classifiers to better predict whether a given program is buggy or non-buggy. Future work will also implement these features on the fully-trained, optimized generative models.

Beyond insufficient time and infrastructure to fully train and optimize the more complex models, the major missing piece of our framework is a principled strategy for localizing bugs to particular snippets of code in a fashion which would reduce the input and output sequence length, and which would aid in actually applying candidate patches to buggy code. To address this, we would adopt a similar strategy as it utilized in the DeepFix framework [16].

Altogether our framework represents an important first step toward the exploration both of utilizing different sophisticated models with different kinds of features for addressing the problem of automated patch generation. More room remains to attempt different network configurations, and to compare against various other baselines; however, the framework in its primal form is there, requiring only further polishing and few missing pieces before we can apply it in full, generative form.

A MODEL PARAMETERS

In this appendix we provide the hyperparameters used to test the SeqGAN-APG model, as well as the various baselines which we implemented for comparison. If a parameter is not specified, it means the parameter was not specifically configured using the implemented API.

A.1 Generative Models

SeqGAN Table 1 outlines the parameters for the SeqGAN network utilized in SeqGAN-APG.

Seq2Seq In order to assure results prior to submission, we utilized a "small" configuration of Seq2Seq with 128 units in each attention cell. In the encoding layer, we used a 0.8 input keep probability and a 1.0 output keep probability in the RNN. In the decoding layer we used a 0.8 input keep probability and a 1.0 output keep probability. We used the Adam optimizer for back-propogation, with an ϵ tolerance of $8e^{-7}$ and a learning rate of 0.0001. We capped the max sequence length at 100. See [25] for more details on parameter configuration for Seq2Seq.

A.2 Discriminative Baselines

Table 2 summarizes the parameters used for the sklearn v. 0.1.2 implementation of various classifiers.

REFERENCES

[1]

[2] Mozilla firefox bugdays. https://wiki.mozilla.org/Bugdays/Bug-verification. Accessed: 2017-09-10.

[3] Mozilla firefox on openhub. https://www.openhub.net/p/firefox. Accessed: 2017-09-10.

[4] Allahyari, M., Pouriyeh, S., Assefi, M., Safaei, S., Trippe, E. D., Gutierrez, J. B., and Kochut, K. Text summarization techniques: A brief survey. *arXiv preprint arXiv:1707.02268* (2017).

[5] Arcuri, A. On the automation of fixing software bugs. In *Companion of the 30th international conference on Software engineering* (2008), ACM, pp. 1003–1006.

[6] Arcuri, A., and Yao, X. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on* (2008), IEEE, pp. 162–168.

[7] Asaduzzaman, M., Roy, C. K., Schneider, K. A., and Hou, D. Cscc: Simple, efficient, context sensitive code completion. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on* (2014), IEEE, pp. 71–80.

[8] Bendersky, E. eliben/pycparser, Nov 2017.

[9] Bruch, M., Monperrus, M., and Mezini, M. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2009), ACM, pp. 213–222.

[10] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, 2007. SP'07. IEEE Symposium on* (2007), IEEE, pp. 252–266.

[11] Dallmeier, V., and Zimmermann, T. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (2007), ACM, pp. 433–436.

[12] DeMarco, F., Xuan, J., Le Berre, D., and Monperrus, M. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis* (2014), ACM, pp. 30–39.

[13] Denton, E. L., Chintala, S., Fergus, R., et al. Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems* (2015), pp. 1486–1494.

[14] Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (2009), ACM, pp. 947–954.

[15] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems* (2014), pp. 2672–2680.

[16] Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning.

[17] Kim, D., Nam, J., Song, J., and Kim, S. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 802–811.

[18] Kim, S., Zimmermann, T., Pan, K., James Jr, E., et al. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on* (2006), IEEE, pp. 81–90.

[19] Le, X. B. D., Lo, D., and Le Goues, C. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on* (2016), vol. 1, IEEE, pp. 213–224.

[20] Le Goues, C. Automatic program repair using genetic programming. *Diss. University of Virginia* (2013).

[21] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering 38*, 1 (2012), 54–72.

[22] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. Sober: statistical model-based bug localization. In *ACM SIGSOFT Software Engineering Notes* (2005), vol. 30, ACM, pp. 286–295.

[23] Long, F., and Rinard, M. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, pp. 298–312.

[24] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H. Bug localization using latent dirichlet allocation. *Information and Software Technology 52*, 9 (2010), 972–990.

[25] Luong, M.-T., Le, Q. V., Sutskever, I., Vinyals, O., and Kaiser, L. Multi-task sequence to sequence learning. *arXiv preprint arXiv:1511.06114* (2015).

[26] Monperrus, M. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 234–242.

[27] Monperrus, M. Automatic software repair: a bibliography. *University of Lille, Tech. Rep.# hal-01206501* (2015).

[28] Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 772–781.

[29] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12*, Oct (2011), 2825–2830.

[30] Press, O., Bar, A., Bogin, B., Berant, J., and Wolf, L. Language generation with recurrent generative adversarial networks without pre-training. *arXiv preprint arXiv:1706.01399* (2017).

[31] Rabinovich, M., Stern, M., and Klein, D. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[32] Ramya, R., Venugopal, K., Iyengar, S., and Patnaik, L. Feature extraction and duplicate detection for text mining: A survey. *Global Journal of Computer Science and Technology 16*, 5 (2017).

[33] Robillard, M., Walker, R., and Zimmermann, T. Recommendation systems for software engineering. *IEEE software 27*, 4 (2010), 80–86.

[34] Sidiroglou, S., and Keromytis, A. D. Countering network worms through automatic patch generation. *IEEE Security & Privacy 3*, 6 (2005), 41–49.

[35] Tan, S. H., Yi, J., Mechtaev, S., Roychoudhury, A., et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion* (2017), IEEE Press, pp. 180–182.

[36] Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. Automatic program repair with evolutionary computation. *Communications of the ACM 53*, 5 (2010), 109–116.

[37] Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D., and Monperrus, M. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering 43*, 1 (2017), 34–55.

[38] Yu, L., Zhang, W., Wang, J., and Yu, Y. Seqgan: Sequence generative adversarial nets with policy gradient. In *AAAI* (2017), pp. 2852–2858.

[39] Zhang, Y., Gan, Z., Fan, K., Chen, Z., Henao, R., Shen, D., and Carin, L. Adversarial feature matching for text generation. *arXiv preprint arXiv:1706.03850* (2017).

Network Part	**Parameter**	**value**
Generator	Embedding Dimension	32
Generator	Hidden Dimension	32
Generator	Sequence Length	100
Generator	Number of Generated Samples	500
Discriminator	Embedding Dimension	64
Discriminator	Filter Sizes	1 - 20
Discriminator	Num Filters	100 - 200
Discriminator	Dropout Keep Probability	0.75
Discriminator	L2 Regularizer λ	0.2
Both	Batch Size	64
Both	Pre-Training Epochs	10
Both	Training Epochs	50

Table 1: Baseline Parameters for Discrimination Task

Model	**Maximum Iterations**	**Learning Rate**	**Tolerance**	**Number Neighbors**	**Number Estimators**	**NB-α**
Ridge Regression			$1e^{-2}$			
Linear SVM			$1e^{-3}$			
Linear SVM L1-Features			$1e^{-3}$			
Linear SVM L2-Features			$1e^{-3}$			
Logistic Regression	1000	$1e^{-4}$				
Logistic Regression (EN)	1000	$1e^{-4}$				
Perceptron	10000					
Passive Aggressive	1000					
K-Nearest-Neighbors				10		
Nearest Centroids						
Random Forest					1000	
Multinomial Naive Bayes						0.01
Bernoulli Naive Bayes						0.01

Table 2: Baseline Parameters for Discrimination Task