ReinforceBug: A Framework to Generate Adversarial Textual Examples

Bushra Sabir
University of Adelaide
CREST - The Centre for Research on Engineering Software Technologies
CSIRO Data61

Muhammad Ali Babar
University of Adelaide
CREST - The Centre for Research on Engineering Software Technologies

Raj Gaire
CSIRO Data61

Abstract

Adversarial Examples (AEs) generated by perturbing original training examples are useful in improving the robustness of Deep Learning (DL) based models. Most prior works generate AEs that are either unconscionable due to lexical errors or semantically and functionally deviant from original examples. In this paper, we present *ReinforceBug*, a reinforcement learning framework, that learns a policy that is transferable on unseen datasets and generates utility-preserving and transferable (on other models) AEs. Our experiments show that ReinforceBug is on average 10% more successful as compared to the state-of-the-art attack TextFooler. Moreover, the target models have on average 73.64% confidence in wrong prediction, the generated AEs preserve the functional equivalence and semantic similarity (83.38%) to their original counterparts, and are transferable on other models with an average success rate of 46%.

1 Introduction

Machine Learning (ML) models have attained remarkable success in several tasks such as classification and decision analytics. However, ML Models specifically Deep Learning (DL) based models, are often sensitive to Adversarial Examples (AEs). AEs consist of modified training data samples that preserve the intrinsic utilities of the ML solutions, but influence target classifier's predictions between original and modified inputs (Fass et al., 2019). Recent works (Fass et al., 2019; Li et al., 2020; Biggio and Roli, 2018) have demonstrated that (i) including AEs as a part of training data can enhance the robustness and generalization of the ML models, and (ii) these examples can be utilized to test the robustness of ML models and help understand their security vulnerabilities and limitations. Previous works on generating AEs have attained success in image (Biggio and Roli, 2018) and on a few conventional text classification tasks such as sentiment, text entailment and movie reviews (Li et al., 2018; Jin et al., 2020; Li et al., 2020). Nevertheless, generating AEs for discrete textual data is still a challenge (Jin et al., 2020).

Textual AEs generated by the prior works have the following limitations:

Utility Preservation Textual AEs require to satisfy task-specific constraints such as lexical rules (spellings or grammar), semantic similarity and functional equivalence to original examples. Yet, most of the current state-of-the-art methods do not satisfy these constraints, thereby generating imperceivable AEs for end-users (Zhang et al., 2020). A few recent works (Li et al., 2020; Wang et al., 2019; Li et al., 2018; Jin et al., 2020) have considered semantic similarity constraint, but other constraints have been barely explored (Jin et al., 2020).

Knowledge Transferability Most prior works Alzantot et al. (2018); Jin et al. (2020); Wang et al. (2019) generate one to one example-specific AEs, i.e., for a given example x, they create an example x'. Each instance x is considered independent in a corpus C, and no relationship between different instances in the corpus is assumed. Therefore, the knowledge gained by transforming an example to AE is limited to a single example and is not reused on other examples. This process is both time-consuming and may not generalize the identified vulnerabilities of the target model.

Word Replacement Strategy Most prior works use a single word replacement strategy such as synonym substitution (Alzantot et al., 2018; Jin et al., 2020; Wang et al., 2019) or character perturbation (Gao et al., 2018) to generate AEs. This strategy has two main disadvantages: (i) the AEs generated by using a character-based replacement strategy, contains many spelling errors that result in unnatural text; and (ii) multiple word transformations with its synonym in a synonym-based replacement

strategy affect the language's fluency, making it sound unnatural. For instance, Jin et al. (2020) generated an AE "Jimmy Wales is a big, ~~fucking idiot liar~~ friggin nincompoop deception". This AE sounds unnatural and is grammatically incorrect.

Handling Noisy Datasets Most prior works generated AEs for datasets such as Yelp (Yelp) and Fake News (Kaggle, 2018a) which do not contain many spelling mistakes or out-of-the-vocabulary words. However, human generated natural text is prone to lexical errors. For example, tweets usually contain informal language, misspellings and unknown words. Prior works, such as (Jin et al., 2020; Wang et al., 2019; Alzantot et al., 2018; Li et al., 2020) that considered synonym substitution strategy, cannot deal with such noisy dataset.

Our contribution

We present *ReinforceBug* that addresses the aforementioned limitations of prior works. Our code is available at `https://bit.ly/2QOZDMT`. Our main contributions are summarized as follows:

1. We propose a reinforcement learning framework, *ReinforceBug*, that learns a policy to generate utility-preserving AEs under the black-box setting with no knowledge about the target model architecture or parameters.

2. We evaluate *ReinforceBug* on four state-of-the-art deep learning models Email spam, Twitter spam, Toxic content and Review polarity detection respectively.

3. We investigate the transferability of our learned policy on a new dataset.

4. We also examine the transferability of our generated AEs on three other state-of-the-art deep learning models.

2 Related Works

Adversarial attacks are extensively studied in computer vision domain (Biggio and Roli, 2018; Goodfellow et al., 2014). Early works in adversarial text attacks were inspired by Generative Adversarial Networks (GANs) (Wong, 2017; Zhao et al., 2018). Wong 2017 showed that GAN-based reinforcement learning algorithms become unstable after a few perturbations. Later, heuristic-based methods such as word removal (Ebrahimi et al., 2018), Out-Of-Vocabulary (OOV) words (Gao et al., 2018) and synonym replacement (Li et al., 2018; Jin et al., 2020; Alzantot et al., 2018) have been proposed. Among these studies, *DeepWordBug* (Gao et al., 2018) generates AEs by randomly transforming a word by OOV word in an example. This approach is practical in producing AEs efficiently; however, it generates AEs that can be detected by the end-user due to large proportion of lexical errors. An attack framework named *TextBugger* (Li et al., 2018) was proposed to generate adversarial samples using the multi-level approach. It identified important words for each example and replaced them with optimal bugs. The approach considered both character-level and word-level transformations. Another recent attack, *TextFooler* (Jin et al., 2020) generates utility-preserving AEs by replacing an important word in an example with its grammatically equivalent synonym. The study also evaluates the generated AEs against semantic similarity constraint.

In this study, we compare our method with *TextBugger* and *TextFooler*.

3 ReinforceBug

3.1 Definitions

Definition 1 *A **Deep Neural Network (DNN)** is a machine learning model that learns a function $F : X \rightarrow Y$ over training data which maps from input space X to a set of classes Y. F is then evaluated on testing data X' and F predicts the output label y' for $x' \in X'$.*

Definition 2 ***Prediction Confidence Score** PCS(x,y) of model F depicts the likelihood of an input $x \in X$ having a label $y \in Y$. The smaller PCS(x,y) suggests F has low confidence that x has a label y.*

Definition 3 *Given a real example x having a label y, a **utility-preserving** AE against x is an input $x' \leftarrow x + \imath$ with a minor perturbation $\imath$ such that x' satisfies a set of perturbation constraints P_{const} and F predicts an incorrect label for it with high PCS i.e., $y' \leftarrow F(x')$ such that $y' \neq y$ and PCS(x',y')$> \nu$.*

Definition 4 *A **black-box attack** is an attack where an attacker does not know the target model F architecture, training data X or hyperparameters θ. An attacker can only query F with*

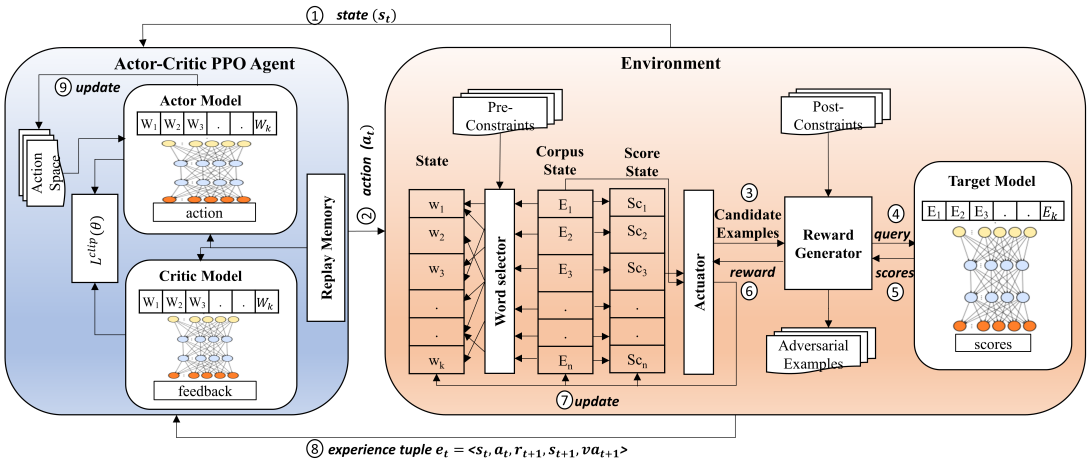


Figure 1: Overall of ReinforceBug

the input examples X' and gets the corresponding $PCS(x, y)$.

Definition 5 *A **non-targeted** attack is an attack in which the adversary's goal is to maximize the misclassification rate of the model F on any generated AE irrespective of its ground-truth label y. i.e., fool the model F to misclassify a spam email as benign email or vice-versa.*

3.2 Problem Formulation

Given a pre-trained target model F, we need to simulate a non-targeted (Definition 5) black-box attack (Definition 4) (Morris et al., 2020) to generate a set of utility-preserving AEs (Definition 3) A_{exp} from a Corpus C with N examples and having corresponding target label $t_g \in Y$. Furthermore, our approach should learn a policy $\pi_{\theta(s,a)}$ to perform perturbation $\imath$ on C such that the model generates utility-preserving AEs (Definition 3) that have semantic similarity ($> \varepsilon$) with the original example but have low perturbation rate (number of words perturbed) and lexical (grammatical and spelling) errors. Moreover, the policy should be transferable on unseen datasets.

3.3 System Overview

Fig 1 provides an overview of ReinfoceBug. We model an attack as a reinforcement learning (Sutton and Barto, 2018) process consisting of three main components: an environment, Proximal Policy Optimization (PPO) agent (Schulman et al., 2017) and action space. Firstly, the environment state (s_t) at time t is processed as input by an agent followed by an action $a_t \in A$ determined by an agent to update s_t to the next state s_{t+1}, where A represents an action space (set of valid actions given the state). Subsequently, the environment's actuator acts on the corpus state to construct candidate examples. These examples are then sent to the reward generator module, which is responsible for computing the reward $r_t + 1$ for action a_t. The reward generator applies post constraints $Post \subset P_{\text{const}}$, queries the target model F and obtain scores of the candidate examples to calculate the reward of action a_t. The reward is sent back to the actuator which determines a valid update to the corpus state as well as the next environmental state s_{t+1}. The experience consisting of $< s_t, a_t, r_{t+1}, s_{t+1}, va_{t+1} >$ is sent to the agent and the agent model is updated. Here va_{t+1} shows the valid actions mask for next state. The va_{t+1} is then used by the agent to update an action space A and restrict the agent to only select valid action on the new state s_{t+1}. Each of the modules is discussed below:

3.3.1 Agent

We use a customized version of Proximal Proximate Optimal (PPO) (Tang et al., 2020) Reinforcement Learning (RL) agent with action mask capability. PPO is an enhanced version of Actor-Critic (AC) Model (Grondman et al., 2012) . In AC architecture the agent consists of a separate policy and Q-value network. They both take in environment state s_t at each time step as an input, while actor determines an action a_t among the possible valid action set A and critic yields value estimation $V_t(s_t)$. While an actor uses a gradient descent algorithm to learn a policy that maximizes the rewards R_t, the critic learns to estimate R_t via minimizing a temporal loss. Further, the PPO algorithm avoids large and inefficient policy updates by constraining the new policy updates to be as close to the origi-

nal policy as possible. We have selected this agent because our action and state space is substantially large and to avoid enormous policy updates which make the agent unstable (please refer to Schulman et al. (2017) for more details).

3.3.2 Environment

The environment takes action a_t from the agent as input and outputs the experience tuple e_t, AEs A_{exp} and a flag $done$ depicting the success of the agent in achieving the goal.

3.3.3 States

The environment maintains the following states.

1. *Corpus State (C_t)*: it is given by $C_t = E_1, E_2, ..., E_N$, where N is the number of examples in the corpus and E_i is equivalent to the set of words $W_i = w_1, w_2, ..., w_n$ for an example i at time t.

2. *Score State ($score_t$)*: it is a vector representing the PCS (Definition 2) of target model F on examples E having a ground-truth label tg in corpus C at time t. For instance, given an example E_i the $score_t[i] = PCS(E_i, tg)$ where tg is a ground truth label of E_i.

3. *Environment State (s_t)*: this state $s_t = w_1, w_2, ..., w_k$ is observable by an agent. It consists of k mutable important words in corpus C.

4. *Success rate ($success_t$)*: it is the proportion of utility-preserving AEs compared to all AEs generated by the agent at time t.

3.3.4 Components

The environment contains following components:

Word selector This component takes C state at $t = 0$ and pre-constraints $Pre \subset P_{\text{const}}$ as inputs. Pre are task-specific perturbation restrictions on the specific entities in E_i. For example, spam messages mostly contain URLs, IP addresses, organization names and email addresses pointing to phishing websites. Perturbing these entities in the spam message can change the functional semantics of the message (Ranganayakulu and Chellappan, 2013). Hence, imposing pre-constraints ensures that generated AE preserves the functional equivalence to E_i after applying perturbations. To achieve this, we have designed a *countvectorizer* (sklearn) using a customized tokenizer. The tokenizer finds the list of immutable entities such as URLs and IP addresses in the text using regular expressions or named entity models (Florian et al., 2003). After that, these words are segmented into immutable words Im_{words} using word tokenizer and saved for each example to be utilized later by actuator module. For training *ReinforceBug*, our method first computes the important words from the training datasets as state (s_t) and then learns a policy to identify best actions (a_t) to transform the s_t to a next state (s_{t+1}) such that the success of our attack is maximum.

Our work extends Jin et al. (2020) important words component; however instead of generating example-specific words, our module identifies corpus-specific important words. An important word is selected using a word importance score $I_{w_{idx}}$. The $I_{w_{idx}}$ is calculated as the sum of prediction change in all the examples (k) containing w_{idx} before and after deleting w_{idx}. The candidate words $w_{idx} \in$ most frequent words in the training dataset vocabulary. It is formally defined as follow.

$$I_{w_{idx}} = \begin{cases} \frac{\sum_{i=1}^{k} (\frac{|F(E_i) - F(w_{idx} \notin E_i)|}{F(E_i)})}{k} \end{cases} \quad (1)$$

If the $I_{w_{idx}}$ is > 0, we consider it as an important word. The final list of all the important words is considered as the state st. st and mapping C_{map} that maps each word to the corpus examples are sent back to the environment. For testing, the designed *countvectorizer* is used to transform the testing data onto these selected words.

Actuator This module is responsible to execute an action a_t selected by an agent. Firstly, the actuator transforms action a_t into an action tuple $< w_{\text{idx}}, act_{\text{idx}}, rep_{\text{idx}} >$, where $w_{\text{idx}}, act_{\text{idx}}$ and rep_{idx} depict the index of the word to be replaced, the operation to be performed on that word and replacement word index respectively. Subsequently, example indexes E_{idx} containing the word w_{idx} are obtained by querying the C_{map}. After that, the actuator examines the $score_t[k]$ (score state) of each example k in E_{idx}. If $score_t[k] > \nu$, only then it is selected as an example to be perturbed, here ν represent the PCS threshold. In this way, the examples for which AEs have already been found are not perturbed further and other examples are given a chance. Once these examples are selected, the operation act_{idx} is applied to w_{idx} which results in multiple replacement options.

Table 1: List of Operations

Operation	Description	Example
Homoglyph	Replace a char with visually similar char.	potentially vs potentia1ly
Insertion	Insert a char in a word.	nearly vs n3early
BitSquatting	Replace a character with one bit different char.	clearly vs glearly
Omission	omit one char.	standard vs stanard
Addition	Add a char on start or end of a word.	classified vs classifiedf
Repetition	Repeat the previous char in the word.	requested vs rrequested
get_synonyms	Replace the word with its synonym	control vs dominance
get_semantic	Append the word with a word contextually related word	like vs such like
get_syntactic	Replace the word with a word having similar syntax.	bta vs bat
currency_word	If word contains currency symbol replace it with currency word and vice-versa.	$ vs dollar dollar vs $
word_to_num	If word represents a number replace it with the number.	eigth vs 8
num_to_word	If word is a number replace it to English word.	8 vs eight
word_weekday	If word is a weekday, replace it with its abbreviation or vice-versa	Wednesday vs wed wed vs Wednesday
word_month	If word is a month, replace it with its abbreviation vice-versa	August vs Aug aug vs August

Table 1 provides the list of operations considered. For example, if an operation is Homoglyph and the word indexed by w_{idx} is "solid" then following replacements can be done "so1id,sol1d,s0lid,5olid". The new word w_{new} is selected by rep_{idx}. After w_{new} is selected against the w_{old}, if w_{old} in not in the immutable word list of example k than a candidate AEs are generated by substituting w_{old} with w_{new} in the previously selected examples. In this way the functional equivalence is ensured before generation of AE. The selected candidate examples are then sent to the reward generator. The reward generator module returns the reward of changing the w_{idx} in the selected examples by applying act_{idx} and selecting rep_{idx}. The reward generator also outputs the AEs that satisfies all the post constraints. Finally, the state is updated by replacing the w_{idx} in s_t with index of new word w_{new}. All further actions on w_{idx} are then invalidated. This is done by setting the $va_{\text{t+1}}$ for all actions on the w_{idx} in an $action\_space$ to False. In this way, multiple actions on the same word cannot be performed in one training episode. The $success_{t+1}$ is updated, the episode completes if the $success_{t+1}$ for the corpus state $C_{\text{t+1}}$ has reached the threshold specified by ϕ or all the words in the state have been updated. The module constructs an experience tuple e_t for the agent and returns e_t, $C_{\text{t+1}}$, $score_{\text{t+1}}$ and $success_{\text{t+1}}$ to the environment.

Reward Generator (RG) Algorithm 1 shows the pseudo-code of the RG module. RG takes candidate examples, C_t, time t and post constraints as input and outputs the Adv_{exp}, the reward $r_{\text{t+1}}$ of a_t and updated corpus state C_{t+1}. In this study, we have considered three main post utilities that the generated Adv_{exp} should preserve apart from changing the output of the classifier. Firstly, the percentage of the *spelling* and *grammatical* errors should not be more than η and ι respectively, and the *semantic similarity* between the original and AE should be $> \varepsilon$. If the candidate example meets all these utilities, then the $target\_model$ is queried to obtain the score of the candidate example. RG updates the corpus and score states for an example where the difference between previous (before perturbation) and new (after perturbation) score of an example > 0 .

Subsequently, the reward generator checks if the candidate example score has converged to $< \nu$ or not. When converged case, the example is added

ALGORITHM 1: Reward_Generator

Input: Original examples org, candidate examples $cand$, time t, $Post \subset P_{\text{const}}$;
Output: Adv_{exp}, reward r_{t+1}, $C_{\text{t+1}}$;
Initialize reward $\leftarrow 0$, $Adv_{\text{exp}} \leftarrow \{\}$;
updated_cand, rewards;
if $(t = 0)$ **then**
  $updated\_cand = cand$;
  Query Target Model;
  $score_{\text{t}} \leftarrow target\_model.query(cand)$;
else
  Initialize thresholds for *spellerror* η, *gramerror* ι, *Semantic* $\varepsilon \in$ Post;
  $N' \leftarrow$ total candidate examples;
  forall $c_k \in cand$ **do**
    $sem = Semantic(org_k, c_k)$;
    $spell = \frac{spellerror(c_k)}{wordlen}$;
    $gram = \frac{gramerror(c_k)}{wordlen}$;
    if $sem > \varepsilon$ ***and*** $spell <= \eta$ ***and*** $grammar <= \iota$) **then**
      Query target_model;
      $new\_sc_k = query(c_k)$
      $prev\_sc_k \leftarrow score_t[k]$;
      $change[k] = \frac{prev\_sc_k - new\_sc_k}{prev\_sc_k}$;
      if $change[k] > 0$ **then**
        update the corpus state;
        $C_{\text{t+1}} \leftarrow E_{\text{idx}}[k] \leftarrow c_k$;
        update the score state;
        $score_{\text{t+1}}[k] \leftarrow new\_sc_k$;
      end
      Compute Reward ;
      if $(score_{\text{t}}[k] < \nu)$ **then**
        $Adv_{\text{exp}}.append(c_k)$;
        $p_{rate}[k] = \frac{wordsperturbed}{wordlen}$;
        $r_{t+1} += \frac{\frac{(org_{score[k]} - score[k])}{org_{score[k]}} + sem[k]}{spell[k] + gram[k] + p_{rate}[k]}$
      else
        $r_{t+1} += change[k]$;
      end
    end
  end
  $r_{t+1} = \frac{r_{t+1}}{N'}$;
end
return: $Adv_{\text{exp}}, r_{t+1}, C_{\text{t+1}}$;

to the list of valid AE against the original example, and a reward r_{t+1} is computed as shown in line 30 of Algorithm 1. Otherwise, the sum of the change in the scores of an example is added as a reward, as shown in line 31. In line 30, the r_{t+1} represents the summation of reward attained on successfully transforming the original examples into well-constrained AE. It is calculated as a summation of the change in original and current score of the example k, assisted by semantic similarity with the original example and penalized by lexical errors (spelling and grammatical) and perturbation rate p_{rate} for generating each AE. In this way, the agent learns to generate AEs with minimum perturbations and lexical errors and more PCS and semantic similarity. Lastly, the r_{t+1} is normalized by the factor N', so that the agent can learn the impact of action a_t on a single example irrespective of the size of the corpus.

Target Model We have considered a black-box attack (Definition 4) against the target model F. F can be any DNN (Definition 1) that provides PCS (Definition 2) score as an output.

4 Experimental Setup

This section presents our experiment details.

4.1 Datasets

We studied the effectiveness of *ReinforceBug* on three noisy and one conventional text classification tasks respectively. Table 2 enlists the statistics of the considered datasets. Precisely during the target model training, we held out 30% of the training data as a validation set, and all parameters were tuned based on it. After that, the testing dataset was used to evaluate the performance of the model. For training and testing our *ReinforceBug*, we split the dataset into training (70%) and testing dataset (30%). The stratified split was applied to ensure the class distribution on these datasets remains consistent with the actual testing dataset.

4.2 Attacking Target Models

For each dataset, we trained four state-of-the-art models namely Word Convolution Neural Network (CNN) (Jain et al., 2018), Character CNN (Zhang et al., 2015), Word Bidirectional Long Term Short Memory (BiLSTM) (Zhou et al., 2016) and Recurrent CNN (Lai et al., 2015) on the training set. However, we did not consider the recent state-of-the-art BERT (Devlin et al., 2019) model because we found that its performance was significantly low for our considered noisy datasets. Srivastava et al. 2020 also reported that the BERT model's performance notably degrades for noisy text datasets, and further research is required to fine-tune BERT for noisy datasets.

For training the considered models, we used an open-source GitHub repository (Lee). Table 3 shows the performance of each model on the testing set. From these models, we selected the models with best performance accuracy as target models (as highlighted in bold) to train *ReinforceBug* . For the rest of the models trained on a similar dataset, we studied the transferability of our generated AEs. Moreover, we tested our *ReinforceBug* against an unseen dataset to study the transferability of our attack on other datasets. Lastly, for $get\_semantic$ and $get\_synonym$ operations (Pennington et al., 2014) and (Mrkšić et al., 2016) embeddings had been used.

4.3 PPO Agent

We used stable-baselines (Hill et al., 2018) reinforcement learning library to implement our PPO agent. We used a Multilayer perception (MLP) model, as an actor and critic models. For training the agent, we used 30 episodes for each model.

4.4 Utility-Preservation Constraints

To ensure the functional equivalence, we defined immutable tokens as pre-constraints using Named Entity Model provided by Spacy and regular expressions. For Enron dataset, names (Person or Organization), IP, email addresses and URL, while for Twitter and Toxic datasets URL, #Hashtags and @Reference and lastly, for Yelp dataset, names and URL were considered as immutable entities. For

Table 2: The Target Models training and testing data sets statistics

Dataset	Training Data	Testing Data	Avg Length	Avg Spelling errors	Avg Grammar errors
Enron (Wiki)	28.6k	9.9k	244	3.07%	31%
Twitter (Kaggle, 2019)	8.1k	3.9k	15	10.67%	28.46%
Toxic (Kaggle, 2018b)	159.5k	12.4k	70	2.13%	25.60%
Yelp (Yelp)	560k	38k	139	0.81%	21.67%

Table 3: Balanced Accuracy of target models on test datasets

Dataset	WordCNN	CharCNN	BiLSTM	RCNN
Enron	97.50%	96.50%	97.60%	**98.30%**
Twitter	93.44%	92.02%	93.72%	**94.15%**
Toxic	69.05%	86.62%	**89.61%**	88.56%
Yelp	94.74%	94.44%	**95.60%**	95.34%

semantic and lexical equivalence, we defined three post constraints, i.e., the semantic similarity, which were calculated using Universal Sentence Encoder (USE) (Cer et al., 2018) considering $\varepsilon = 0.60$. Moreover, for counting spelling mistakes Garbe was used while for calculating grammar issues, we used language tool (PYPI).

5 Results

Results of our experiments are presented here.

5.1 Attack Evaluation

Table 4 shows the samples of AEs generated by *ReinforceBug* and Table 5 illustrates the main results of our experiments. It can be seen that *ReinforceBug* produces AEs with a comparatively high PCS (i.e., on average 74%), semantic similarity (i.e., on average 83.5%) than other two attacks for all the datasets. However, its success rate is on average 15% less than TextBugger (Li et al., 2018) for the all the models. One reason behind it is that TextBugger generates unrealistic AEs with least PCS (i.e., on average 66%), low semantical similarity (i.e., on average 63%) and significantly high lexical errors (i.e., on average 14% spelling and 27% grammar errors). Additionally, TextBugger on average perturbs more than 59% words of the original text to generate AEs. Such a high perturbation rate is too large to be ignored by the end-user. Moreover, for Enron and Twitter datasets, TextBugger perturbs 9.09% and 97.67% of the URLs present in the text, thus adversely effecting the functional semantics of the text. Therefore, the success rate of TextBugger is an overestimation and deviates from the semantic, functional and lexical constraints.

In comparison with TextFooler (Jin et al., 2020) our method has significantly high success rate (i.e., on average more than 10%) for all the models. It is expected because TextFooler relies on synonym substitutes technique to generate AE, however, for noisy datasets with relative high lexical errors such as Twitter (Table 2) this method tends to fail. Lastly, TextFooler produces 5% more grammatical errors than *ReinforceBug* and similar to TextBugger, TextFooler also perturbs URLs, thus effecting the functional semantics of the generated AEs. These findings suggest that *ReinforceBug* produces effective and utility-preserving AEs (Definition 3).

5.2 Knowledge Transferability

Table 6 shows the transferability of policy learned by *ReinforceBug* on unseen datasets for each task. For benchmarking the results are also compared with the state-of-the-art attacks *TextBugger* and *Textfooler*. It is evident from the results that *ReinforceBug* takes less time to generate adversarial samples as compared to the other models. It is because *ReinforceBug* utilizes the same important word vocabulary selected while learning and the agent has already explored and learned utility-preserving operations on them during training. Additionally, although *TextFooler* and *TextBugger* both perform example-specific perturbation and *ReinforceBug* might suffer from out-of-vocabulary words but still the success rate of *ReinforceBug* on test datasets is more than *TextFooler* attack and less than *TextBugger* which are aligned with our finding in section 5.1. Also, from Table 6 it is evident that our *ReinforceBug* produces utility-preserving AEs with high PCS and semantic similarity (i.e., on average more than 70% and 82% respectively) and comparatively low perturbation rate (on average 8.5%) for all test datasets. It suggests that the important words and transformation learned from the training datasets are transferable to unseen datasets and can be used to generalize the vulnerabilities of the target models.

5.3 Attack Transferability on Models

To determine whether AEs curated based on one model can also fool other models for the same task, we examined the transferability of AEs on other models (see Table 3 for the performance of these models). Table 7 shows the transferability result. AEs generated by our method are more transferable to other models in comparison to the state-of-the-art attacks. However, there is a moderate degree of transferability between models, and the transferability is higher for Twitter and Toxic detection task as compared to Email and Yelp movie-review classification task. Nevertheless, BiLSTM trained on Enron dataset (having 97.60% accuracy) offers more resilience to AEs generated by RCNN by limiting the success rate of the attack to (<17%) for all the attacks, while other models are highly vulnerable to the AEs. It signifies that vulnerabilities exploited by our AEs are task-specific and moderately model-independent.

Table 4: Samples of Examples (red font depicts the change in the original example)

Task: Twitter Spam/Benign	**Original Label** Benign (PCS 100%)	**Adversarial Label** Spam (PCS 98%)	**Task: Toxic/Non-** Toxic Comment	**Original Label** Toxic (PCS 97%)	**Adversarial Label** non-Toxic (PCS 83%)
I should of just gone home yesterday and spent my day off today with my family relatives			Even trolls surely deserve to eat, bastardx.		
Task: Email Spam/Benign	**Original Label** Spam (PCS 100%)	**Adversarial Label** Benign (PCS 96%)	**Task: Positive/** Negative Review	**Original Label** Negative (72%)	**Adversarial Label** Positive (PCS 86%)
Subject:soul mate one of your buddies hooked you up on a date scheduled with another buddy. your invitation:a free dating web site created by women no more invitation:			Premium price for a standard sandwich. Would have been an amazingly enjoyable visit had the deli guy been at least welcoming .		

Table 5: ReinforceBug Attack on Training Dataset

Dataset	**Attacks**	**Success Rate**	**Perturbation Rate**	**% of URLs Perturbed**	**Avg PCS**	**Avg Semantic Similarity**	**Avg Spelling Errors**	**Avg Grammar Errors**
Enron RCNN	TextBugger	**43.09%**	46.64%	35.21%	62.52%	61.51%	13.39%	35.7%
	TextFooler	22.31%	**10.60%**	9.09%	59.76%	76.62%	**3.08%**	21.80%
	ReinforceBug	33.21%	11.09%	**0.00%**	**73.82%**	**85.66%**	3.30%	**9.88%**
Twitter RCNN	TextBugger	**65.98%**	24.74%	97.67%	74.6%	58.8%	25.77%	35.64%
	TextFooler	12.66%	11.89%	99.10%	79.17%	73.79%	5.22%	15.68%
	ReinforceBug	14.99%	**11.29%**	**0.00%**	**82.82%**	**80.86%**	**4.14%**	**13.99**
Yelp BiLSTM	TextBugger	**44.88%**	21.89%	6.25%	58.62%	66.97%	5.22%	16.85%
	TextFooler	33.80%	**4.15%**	4.73%	58.39%	80.60%	**0.74%**	14.13%
	ReinforceBug	39.35%	**3.18%**	0.00%	**63.78%**	**87.78%**	1.88%	**9.68%**
Toxic BiLSTM	TextBugger	**42.11%**	24.78%	45.10%	62.85%	60.83%	11.36%	19.98%
	TextFooler	21.00%	**7.07%**	28.57%	64.17%	76.68%	**1.53%**	14.70%
	ReinforceBug	31.22%	9.43%	**0.00%**	**71.52%**	**78.07%**	3.69%	**6.40%**

Table 6: ReinforceBug Attack on Testing Dataset (Test time is given days-hours:minutes:seconds format)

Dataset	**Attacks**	**Test Time**	**Success Rate**	**Perturbation Rate**	**Avg PCS**	**Avg Semantic Similarity**	**Avg Spelling Errors**	**Avg Grammar Errors**
Enron RCNN	TextBugger	1-18:55:48	**42.42%**	38.68%	62.52%	61.51%	13.39%	17.31%
	TextFooler	2-03:07:54	20.98%	8.12%	61.19%	77.47%	7.23%	10.96%
	ReinforceBug	**1-08:15:21**	31.80%	10.37%	**66.34%**	**88.17%**	**6.86%**	**7.95%**
Twitter RCNN	TextBugger	0-1:30:23	**64.71%**	20.52%	75.6%	58.5%	26.43%	58.17%
	TextFooler	0-1:34:05	12.59%	**9.17%**	79.18%	74.65%	19.71%	14.01%
	ReinforceBug	**0-0:47:29**	12.68%	13.43%	**81.77%**	**75.84%**	**16.97%**	20.17%
Yelp BiLSTM	TextBugger	2-13:26:43	**50.42%**	26.73%	58.73%	66.96%	5.23%	8.21%
	TextFooler	2-16:17:21	39.33%	**4.00%**	58.85%	80.58%	3.15%	5.25%
	ReinforceBug	**1-15:27:12**	46.19%	5.77%	**61.22%**	**89.07%**	3.62%	**4.42%**
Toxic BiLSTM	TextBugger	0-11:57:28	**33.99%**	36.24%	62.52%	61.49%	11.07%	18.65%
	TextFooler	0-15:19:29	16.05%	**7.93%**	65.83%	77.23%	8.37%	7.94%
	ReinforceBug	**0-09:22:49**	30.16%	10.52%	**70.36%**	**78.43%**	**4.83%**	**4.13%**

Table 7: Transferrability of AEs generated by ReinforceBug Attack on other models

	Enron			**Twitter**			**Yelp**			**Toxic**		
	CharCNN	**WordCNN**	**BILSTM**	**CharCNN**	**WordCNN**	**BILSTM**	**CharCNN**	**WordCNN**	**RCNN**	**CharCNN**	**WordCNN**	**RCNN**
TextBugger	**51.0%**	17.3%	**16.5%**	11.7%	30.8%	50.0%	21.8%	21.6%	24.5%	50.9%	43.8%	42.5%
TextFooler	47.0%	12.5%	12.3%	24.0%	41.6%	49.7%	25.6%	22.5%	27.1%	66.3%	59.8%	52.9%
ReinforceBug	47.3%	**27.5%**	16.3%	**53.5%**	**42.6%**	**52.0%**	**34.0%**	**37.2%**	**34.5%**	**70.3%**	**64.9%**	**72.2%**

6 Analysis and Discussion

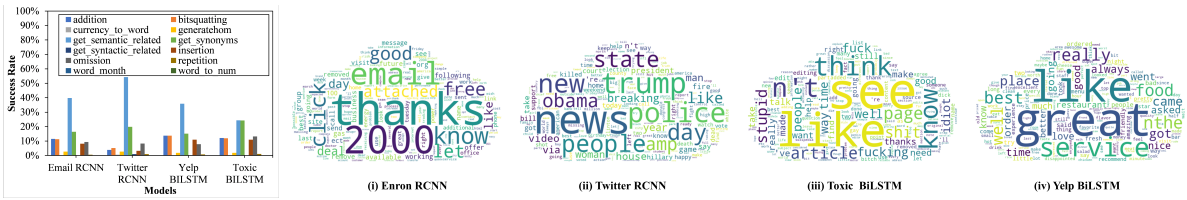
This section analyse the results.

6.1 Operations Distribution

Figure 2a illustrates the proportion of each operation chosen by the *ReinforceBug* to generate utility-preserving AEs. We can see that $get\_semantic$, and $get\_synonyms$ operations are most dominant for all the tasks. One reason could be that $get\_semantic$ is deliberately designed for creating similar contextual adversarial texts without deleting the original important word, while $get\_synonyms$ replaces the word with similar meaning word. That is why the semantic similarity remains intact without impacting the linguistic structure of the text. Other operations, i.e., *addition*, *insertion*, *bitsquatting* and *omission* that cause common typos are moderately chosen, however, *generatehom* is rarely selected by *ReinforceBug*. This happens because it can only replace a character with a visually similar character and produces fewer replacement options for a word than other operations. This reason is also valid of $word\_month$ and $word\_to\_num$ as only few words are either words representing month or numbers in the corpus vocabulary.

6.2 Distribution of words versus the number of examples effected

To demonstrate the knowledge transferability, we visualize the identified important words according to the number of examples affected by their



(a) Operation Distribution (b) Word Cloud of important words versus the number of Examples affected for the target models across Trained Models

Figure 2: Analysis of AEs generated by ReinforceBug

replacements in Figure 2b. Here, the words impacting more examples are represented with a larger font. Figure 2b(i) shows that in emails, words such as 'click', 'attached', 'thanks', and 'deal' are more likely to affect the prediction of the target models by decreasing the spam intent to benign. Whereas for the Twitter dataset, Figure 2b(ii) shows that the targeted RCNN model is more vulnerable to minor perturbation on words such as 'news', 'trump', 'obama', 'state' and 'police'. For toxic content detection, the model decision is manipulated for most examples with words like 'see', 'like', 'think'. Perturbing words like 'stupid' and 'idiot' decreases the toxicity of the text. Lastly, for the yelp dataset, changing words such as 'like', 'great' and 'best' increases the text's negative extent. Therefore, it is evident that models are vulnerable to these words irrespective of a specific example; instead, these vulnerabilities affect multiple examples in the corpus and are transferable to new datasets as seen in (section 5.2) as well as are transferable to other models (section 5.3).

7 Conclusion

Overall, this study proposes *ReinforceBug*, a reinforcement learning-based framework to generate utility-preserving AEs against the state-of-the-art text classifiers under black-box settings. Extensive experiments demonstrate that it effectively generates utility-preserving AEs that are transferable to other models, and the learned policy is transferable to the unseen datasets. It identifies semantic concatenation and synonym substitution attacks as a significant threat to text-classifiers and suggests defences against these attacks should be explored in the future to improve their robustness.

8 Acknowledgements

This work was supported with super-computing resources provided by the Phoenix HPC service at the University of Adelaide.

References

Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2890–2896, Brussels, Belgium. Association for Computational Linguistics.

Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2154–2156, New York, NY, USA. Association for Computing Machinery.

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Brian Strope, and Ray Kurzweil. 2018. Universal sentence encoder for English. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 169–174, Brussels, Belgium. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 31–36, Melbourne, Australia. Association for Computational Linguistics.

Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913.

Radu Florian, Abe Ittycheriah, Hongyan Jing, and Tong Zhang. 2003. Named entity recognition

through classifier combination. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003*, pages 168–171.

Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. 2018. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 50–56. IEEE.

Wolf Garbe. wolfgarbe/symspell: Symspell: 1 million times faster through symmetric delete spelling correction algorithm.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2018. Stable baselines.

Gauri Jain, Manisha Sharma, and Basant Agarwal. 2018. Spam detection on social media using semantic convolutional neural network. *International Journal of Knowledge Discovery in Bioinformatics (IJKDB)*, 8(1):12–26.

Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8018–8025.

Kaggle. 2018a. Fake news, kaggle.

Kaggle. 2018b. Toxic comment classification challenge , kaggle.

Kaggle. 2019. Utkml's twitter spam detection competition , kaggle.

Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*.

Dongjun Lee. dongjun-lee/text-classification-models-tf: Tensorflow implementations of text classification models.

Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2018. Textbugger: Generating adversarial text against real-world applications. *arXiv preprint arXiv:1812.05271*.

Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. BERT-ATTACK: Adversarial attack against BERT using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6193–6202, Online. Association for Computational Linguistics.

John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. 2020. Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 119–126.

Nikola Mrkšić, Diarmuid Ó Séaghdha, Blaise Thomson, Milica Gašić, Lina Rojas-Barahona, Pei-Hao Su, David Vandyke, Tsung-Hsien Wen, and Steve Young. 2016. Counter-fitting word vectors to linguistic constraints. In *Proceedings of HLT-NAACL*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

PYPI. language-tool-python · pypi.

Dhanalakshmi Ranganayakulu and C Chellappan. 2013. Detecting malicious urls in e-mail–an implementation. *AASRI Procedia*, 4:125–131.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

sklearn. sklearn.feature_extraction.text.countvectorizer — scikit-learn 0.23.2 documentation.

Spacy. Linguistic features · spacy usage documentation.

Ankit Srivastava, Piyush Makhija, and Anuj Gupta. 2020. Noisy text data: Achilles' heel of bert. In *Proceedings of the Sixth Workshop on Noisy User-generated Text (W-NUT 2020)*, pages 16–21.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

Cheng-Yen Tang, Chien-Hung Liu, Woei-Kae Chen, and Shingchern D You. 2020. Implementing action mask in proximal policy optimization (ppo) algorithm. *ICT Express*.

Xiaosen Wang, Hao Jin, and Kun He. 2019. Natural language adversarial attacks and defenses in word level. *arXiv preprint arXiv:1909.06723*.

ACL Wiki. Spam filtering datasets.

Catherine Wong. 2017. Dancin seq2seq: Fooling text classifiers with adversarial text example generation. *arXiv preprint arXiv:1712.05419*.

Yelp. Yelp dataset.

Wei Emma Zhang, Quan Z Sheng, Ahoud Alhazmi, and Chenliang Li. 2020. Adversarial attacks on deep-learning models in natural language processing: A survey. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 11(3):1–41.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657.

Zhengli Zhao, Dheeru Dua, and Sameer Singh. 2018. Generating natural adversarial examples.

Peng Zhou, Zhenyu Qi, Suncong Zheng, Jiaming Xu, Hongyun Bao, and Bo Xu. 2016. Text classification improved by integrating bidirectional LSTM with two-dimensional max pooling. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3485–3495, Osaka, Japan. The COLING 2016 Organizing Committee.