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  
CSIRO Data61

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 \( x' \leftarrow x + \nu \) with a minor perturbation \( \nu \) such that \( x' \) satisfies a set of perturbation constraints \( P_{\text{const}} \) 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 hyper-parameters \( \theta \). An attacker can only query \( F \) with
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 $t_y \in Y$. Furthermore, our approach should learn a policy $\pi_{\theta(s,a)}$ to perform perturbation $\iota$ 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 ReinforceBug. 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 $Post \subset Post_{const}$, 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 $<s_t, a_t, r_{t+1}, s_{t+1}, v_{a_{t+1}}>$ is sent to the agent and the agent model is updated. Here $v_{a_{t+1}}$ shows the valid actions mask for next state. The $v_{a_{t+1}}$ 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 $C_t = E_1, E_2, \ldots, E_N$, where $N$ is the number of examples in the corpus and $E_i$ is equivalent to the set of words $W_i = w_1, w_2, \ldots, w_n$ 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 $score_{E_i}[i] = PCS(E_i, tg)$ where $tg$ is a ground truth label of $E_i$.

3. **Environment State ($s_t$):** this state $s_t = w_1, w_2, \ldots, w_k$ 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 $Pre \subset P_{const}$ 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 $I_{mwords}$ 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 $I_{widx}$. The $I_{widx}$ 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 $w_{idx}$ ∈ most frequent words in the training dataset vocabulary. It is formally defined as follow.

$$I_{widx} = \sum_{i=1}^{k} \frac{|F(E_i) - F(w_{idx}, E_i)|}{F(E_i)}$$ (1)

If the $I_{widx}$ 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 $< w_{idx}, act_{idx}, rep_{idx} >$, where $w_{idx}, act_{idx}$ 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 $score_{E_i}[k]$ (score state) of each example $k$ in $E_{idx}$. If $score_{E_i}[k] > \nu$, only then it is selected as an example to be perturbed, here $\nu$ 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.
Example aug vs August
Insert a char in a word.
ward generator module returns the reward of change the functional equivalence is ensured before
The new word
the generated
Adv
of
didate examples,
the pseudo-code of the RG module. RG takes can-
Reward Generator (RG)
Algorithm 1 shows
success
by
C
success
actions on the same word cannot be performed in
done by setting the
w
the
w
w
in an
word_month
word_weekday
get_semantic
Addition
Omission
Reposition
get_synonyms
get_semantic
ward the word in a word synonymously related word
ward the word with a word having similar synonyms
if word contains currency symbol replace it with currency word and vice-versa.
if word contains currency symbol replace it with currency word and vice-versa.
return:
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
Initialize thresholds for spellerror η_
gramerror ε, Semantic ε ∈ Post;
N ← total candidate examples;
forall ck ∈ cand do
sem = Semantic(orkg, ck);
spell = spellerror(ck);
gram = gramerror(ck);
if sem > ε and spell < η and grammar (< ε) then
Query target_model;
new_seq = query(ck);
prev_seq = score(ck);
change[k] = prev_seq - new_seq;
if change[k] > 0 then
update the corpus state;
C_{t+1} = E_{idx}[k] ← ck;
update the score state;
score_{t+1}[k] = new_seq;
end Compute Reward;
if (score[k] < ν) then
Advexp.append(ck);
prate[k] = wordperturbed/worden;
rt_{t+1} = (orhscore[k] − score[k]) /
ν*score[k] + spell[k] + gram[k] + prate[k];
else
rt_{t+1} = change[k];
end
end
end
r_{t+1} = r_{t+1}/N;
end
return: Advexp, r_{t+1}, C_{t+1};

Table 1: List of Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homoglyph</td>
<td>Replace a char with visually similar char.</td>
<td>potentially vs potentially</td>
</tr>
<tr>
<td>Insertion</td>
<td>Insert a char into a word.</td>
<td>ward the word in a word synonymously related word</td>
</tr>
<tr>
<td>BitSquating</td>
<td>Replace a character with one bit clearly vs clearly</td>
<td>different char</td>
</tr>
<tr>
<td>Omission</td>
<td>Omit one char.</td>
<td>standard vs standard</td>
</tr>
<tr>
<td>Addition</td>
<td>Add a char on start or end of a word.</td>
<td>classified vs classified</td>
</tr>
<tr>
<td>Reposition</td>
<td>Replace the previous char in the word.</td>
<td>classified vs classified</td>
</tr>
<tr>
<td>get_synonyms</td>
<td>Replace the word with its synonym.</td>
<td>control vs dominant</td>
</tr>
<tr>
<td>get_semantic</td>
<td>Replace the word with a word synonymously related word</td>
<td>success vs success</td>
</tr>
<tr>
<td>get_syntactic</td>
<td>Replace the word with a word having similar synonyms</td>
<td>true vs bat</td>
</tr>
<tr>
<td>currency_word</td>
<td>If word contains currency symbol replace it with currency word and vice-versa.</td>
<td>true vs true</td>
</tr>
<tr>
<td>word_to_num</td>
<td>If word represents a number replace it with the number.</td>
<td>true vs true</td>
</tr>
<tr>
<td>num_to_word</td>
<td>If word is a number replace it to replace word.</td>
<td>true vs true</td>
</tr>
<tr>
<td>word_weekday</td>
<td>If word is a weekday, replace it with an abbreviation or vice versa.</td>
<td>true vs true</td>
</tr>
<tr>
<td>word_month</td>
<td>If word is a month, replace it with a month abbreviation vice versa.</td>
<td>Aug vs August</td>
</tr>
</tbody>
</table>

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 "so l id, so l id, so l id, so l id, So l id". 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 AE \( s \) is generated by substituting \( w_{old} \) with \( w_{new} \) in the previously selected examples. 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 AE 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 \( v_{t+1} \) 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 \( C_{t+1} \) 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 \( e_t, C_{t+1}, score_{t+1} \) and \( success_{t+1} \) 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 \( Advexp \), the reward \( r_{t+1} \) of \( α_t \) and updated corpus state \( C_{t+1} \). In this study, we have considered three main post utilities that the generated \( Advexp \) 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 \( ε \), respectively, and the semantic similarity between the original and AE should be \( > ε. \) 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 \( < ν \) or not. When converged case, the example is added
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 $get_{\text{semantic}}$ and $get_{\text{synonym}}$ 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.

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Training Data</th>
<th>Testing Data</th>
<th>Avg Length</th>
<th>Avg Spelling errors</th>
<th>Avg Grammar errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enron (Wiki)</td>
<td>2.0k</td>
<td>2.0k</td>
<td>22.2k</td>
<td>0.3%</td>
<td>6.8%</td>
</tr>
<tr>
<td>Twitter (Kaggle, 2019)</td>
<td>8.1k</td>
<td>3.9k</td>
<td>73.9k</td>
<td>0.1%</td>
<td>6.1%</td>
</tr>
<tr>
<td>Toxic (Kaggle, 2018)</td>
<td>159.5k</td>
<td>12.8k</td>
<td>119</td>
<td>1.0%</td>
<td>5.6%</td>
</tr>
<tr>
<td>Yelp (Yelp)</td>
<td>159k</td>
<td>58k</td>
<td>159</td>
<td>0.8%</td>
<td>24.6%</td>
</tr>
</tbody>
</table>

Table 3: Balanced Accuracy of target models on test datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>WordCNN</th>
<th>CharCNN</th>
<th>BiLSTM</th>
<th>RCNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enron</td>
<td>97.50%</td>
<td>96.50%</td>
<td>96.60%</td>
<td>98.30%</td>
</tr>
<tr>
<td>Twitter</td>
<td>91.44%</td>
<td>92.02%</td>
<td>93.72%</td>
<td>94.15%</td>
</tr>
<tr>
<td>Toxic</td>
<td>69.05%</td>
<td>86.62%</td>
<td>89.61%</td>
<td>88.56%</td>
</tr>
<tr>
<td>Yelp</td>
<td>94.74%</td>
<td>94.44%</td>
<td>95.60%</td>
<td>95.34%</td>
</tr>
</tbody>
</table>
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 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.
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., addi-

Table 5: ReinforceBug Attack on Training Dataset

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Attacks</th>
<th>Perturbation Rate</th>
<th>Avg PCS</th>
<th>Avg Semantic Similarity</th>
<th>Avg Spelling Errors</th>
<th>Avg Grammar Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enron RCNN</td>
<td>TextBugger</td>
<td>45.09%</td>
<td></td>
<td>55.21%</td>
<td>13.39%</td>
<td>35.7%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>73.21%</td>
<td></td>
<td>73.82%</td>
<td>3.08%</td>
<td>9.88%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>30.22%</td>
<td></td>
<td>65.31%</td>
<td>21.80%</td>
<td>10.52%</td>
</tr>
<tr>
<td>Twitter RCNN</td>
<td>TextBugger</td>
<td>65.90%</td>
<td></td>
<td>97.67%</td>
<td>28.71%</td>
<td>5.22%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>12.66%</td>
<td></td>
<td>79.17%</td>
<td>15.68%</td>
<td>5.22%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>11.29%</td>
<td></td>
<td>82.82%</td>
<td>13.99%</td>
<td>5.22%</td>
</tr>
<tr>
<td>Yelp BiLSTM</td>
<td>TextBugger</td>
<td>44.88%</td>
<td></td>
<td>63.78%</td>
<td>21.00%</td>
<td>5.22%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>39.35%</td>
<td></td>
<td>87.56%</td>
<td>5.22%</td>
<td>9.68%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>3.18%</td>
<td></td>
<td>9.28%</td>
<td>19.98%</td>
<td>5.22%</td>
</tr>
<tr>
<td>Toxic BiLSTM</td>
<td>TextBugger</td>
<td>42.11%</td>
<td></td>
<td>64.17%</td>
<td>1.53%</td>
<td>15.70%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>31.22%</td>
<td></td>
<td>71.52%</td>
<td>3.69%</td>
<td>6.40%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>7.97%</td>
<td></td>
<td>78.07%</td>
<td>3.69%</td>
<td>6.40%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Attacks</th>
<th>Test Time</th>
<th>Success Rate</th>
<th>Perturbation Rate</th>
<th>Avg PCS</th>
<th>Avg Semantic Similarity</th>
<th>Avg Spelling Errors</th>
<th>Avg Grammar Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enron RCNN</td>
<td>TextBugger</td>
<td>1-18:55:48</td>
<td>42.42%</td>
<td>58.68%</td>
<td>51.52%</td>
<td>13.39%</td>
<td>17.31%</td>
<td>10.96%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>2-03:07:54</td>
<td>20.96%</td>
<td>16.19%</td>
<td>77.47%</td>
<td>7.23%</td>
<td>10.96%</td>
<td>10.96%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>1-03:20:12</td>
<td>31.80%</td>
<td>10.37%</td>
<td>66.34%</td>
<td>8.97%</td>
<td>21.00%</td>
<td>21.00%</td>
</tr>
<tr>
<td>Twitter RCNN</td>
<td>TextBugger</td>
<td>0-1:30:23</td>
<td>64.71%</td>
<td>20.52%</td>
<td>75.6%</td>
<td>26.43%</td>
<td>58.17%</td>
<td>58.17%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>0-1:34:05</td>
<td>12.59%</td>
<td>9.17%</td>
<td>79.18%</td>
<td>19.71%</td>
<td>14.01%</td>
<td>14.01%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>0-0:47:29</td>
<td>12.68%</td>
<td>13.43%</td>
<td>81.77%</td>
<td>16.97%</td>
<td>20.17%</td>
<td>20.17%</td>
</tr>
<tr>
<td>Yelp BiLSTM</td>
<td>TextBugger</td>
<td>2-13:20:43</td>
<td>50.42%</td>
<td>26.13%</td>
<td>58.13%</td>
<td>5.23%</td>
<td>8.21%</td>
<td>8.21%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>2-16:17:21</td>
<td>39.33%</td>
<td>4.00%</td>
<td>58.85%</td>
<td>3.15%</td>
<td>5.25%</td>
<td>5.25%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>1-15:27:12</td>
<td>46.19%</td>
<td>5.77%</td>
<td>61.22%</td>
<td>3.62%</td>
<td>4.42%</td>
<td>4.42%</td>
</tr>
<tr>
<td>Toxic BiLSTM</td>
<td>TextBugger</td>
<td>0-11:57:28</td>
<td>33.99%</td>
<td>36.24%</td>
<td>62.52%</td>
<td>11.07%</td>
<td>18.65%</td>
<td>18.65%</td>
</tr>
<tr>
<td></td>
<td>TextFooler</td>
<td>0-15:19:29</td>
<td>16.03%</td>
<td>7.93%</td>
<td>65.85%</td>
<td>8.37%</td>
<td>7.94%</td>
<td>7.94%</td>
</tr>
<tr>
<td></td>
<td>ReinforceBug</td>
<td>0-09:22:49</td>
<td>30.16%</td>
<td>10.52%</td>
<td>70.36%</td>
<td>4.83%</td>
<td>4.13%</td>
<td>4.13%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Char CNN</th>
<th>Word CNN</th>
<th>BiLSTM</th>
<th>Char CNN</th>
<th>Word CNN</th>
<th>RCNN</th>
<th>Char CNN</th>
<th>Word CNN</th>
<th>RCNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enron RCNN</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
</tr>
<tr>
<td>Twitter RCNN</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
</tr>
<tr>
<td>Yelp BiLSTM</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
<td>87.3%</td>
</tr>
</tbody>
</table>

6.2 Distribution of words versus the number of examples affected

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

insertion, insertion, bitsquatting and omission that cause common typos are moderately chosen, however, generate_token and word_to_num as only few words are either words representing month or numbers in the corpus vocabulary.
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


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


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


PYPI. language-tool-python · pypi.


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

Spacy. Linguistic features · spacy usage documentation.


ACL Wiki. Spam filtering datasets.

Yelp. Yelp dataset.


