Poster: TEXTBUGGER: Generating Adversarial Text Against Real-world Applications

Jinfeng Li*, Shouling Ji*† ✉, Tianyu Du*, Bo Li‡ and Ting Wang§
* Institute of Cyberspace Research and College of Computer Science and Technology, Zhejiang University
Email: {lijinfeng0713, sji, zjradty}@zju.edu.cn
† Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies
‡ University of Illinois Urbana-Champaign, Email: lxbosky@gmail.com
§ Lehigh University, Email: inbox.ting@gmail.com

***Abstract*—In this poster, we present TEXTBUGGER, a general attack framework for generating adversarial texts. In contrast to prior works, TEXTBUGGER differs in significant ways: (i) effective – it outperforms state-of-the-art attacks in terms of success rate; (ii) evasive – it preserves the utility of benign text; and (iii) efficient – it generates adversarial text with computational complexity sub-linear to the text length. Experimental results demonstrate its effectiveness, evasiveness, and efficiency.**

I. INTRODUCTION AND ATTACK DESIGN

Recently, Deep neural networks (DNNs) have been found to be vulnerable against adversarial examples which are carefully generated by adding small perturbations to the legitimate inputs to fool the targeted models [2]. Such discovery has also raised serious concerns, especially when deploying such machine learning models to security-sensitive tasks. While existing works on adversarial examples mainly focus on the image domain, it is more challenging to deal with text data due to its discrete property, which is hard to optimize. Furthermore, in the text domain, small perturbations are usually clearly perceptible, and the replacement of a single word may drastically alter the semantics of the sentence. Therefore, existing attack algorithms designed for images cannot be directly applied to text, and we need to study new attack techniques. In this poster, we propose TEXTBUGGER, a framework that can effectively and efficiently generate utility-preserving adversarial texts against state-of-the-art text classification systems under both white-box and black-box settings. A successful adversarial example is shown in Fig. 1. Due to the limitation of pages, we only detail the process of generating adversarial texts under black-box setting.

Task: Sentiment Analysis ***Classifier:*** Amazon AWS
Original label: 100% Negative ***Adversarial label:*** 89% Positive

Text: I watched this movie recently mainly because I am a Huge fan of Jodie Foster's. I saw this movie was made right between her 2 Oscar award winning performances, so my expectations were fairly high. ~~Unfortunately~~ **Unf0rtunately**, I thought the movie was ~~terrible~~ **terrib1e** and I'm still left wondering how she was ever persuaded to make this movie. The script is really ~~weak~~ **wea k**.

Fig. 1. A successful adversarial example.

Briefly, the process of generating adversarial texts under black-box setting contains three steps: (1) Find the important sentences. (2) Use a scoring function to determine the importance of each word regarding to the classification result, and rank the words based on their scores. (3) Use the bug selection algorithm to change the selected words. The black-box adversarial text generation algorithm is shown in Algorithm 1.

Algorithm 1 TEXTBUGGER under black-box settings

```
Input: legitimate document $x$ and its ground truth label $y$,
    classifier $\mathcal{F}(\cdot)$, threshold $\epsilon$
Output: adversarial document $x_{adv}$
 1: Inititialize: $x' \leftarrow x$
 2: for $s_i$ in document $x$ do
 3:     $C_{s_i} = \mathcal{F}_y(s_i)$;
 4: end for
 5: $S_{ordered} \leftarrow Sort(sentences)$ according to $C_{s_i}$;
 6: Delete sentences in $S_{ordered}$ if $\mathcal{F}_l(s_i) \neq y$;
 7: for $s_i$ in $S_{ordered}$ do
 8:     for $w_j$ in $s_i$ do
 9:         Compute $C_{w_j}$ according to Eq.1;
10:     end for
11:     $W_{ordered} \leftarrow Sort(words)$ according to $C_{w_j}$;
12:     for $w_j$ in $W_{ordered}$ do
13:         $bug = SelectBug(w_j, x', y, \mathcal{F}(\cdot))$;
14:         $x' \leftarrow$ replace $w_j$ with $bug$ in $x'$
15:         if $S(x, x') \leq \epsilon$ then
16:             Return None.
17:         else if $\mathcal{F}_l(x') \neq y$ then
18:             Solution found. Return $x'$.
19:         end if
20:     end for
21: end for
22: return None
```

Step 1: Find Important Sentences (line 2-6). Suppose the input document is $x = (s_1, s_2, \cdots, s_n)$, where s_i represents the sentence at the i^{th} position. First, we segment each document into sentences. Then we filter out the sentences that have different predicted labels with the original document label (i.e., filter out $\mathcal{F}_l(s_i) \neq y$). Then, we sort the important sentences in an inverse order according to their importance score C_{s_i}. The importance score of a sentence s_i is represented with the confidence value of the predicted class $\mathcal{F}_y$, i.e., $C_{s_i} = \mathcal{F}_y(s_i)$.

Step 2: Find Important Words (line 8-11). Considering the vast search space of possible changes, we should first find the most important words that contribute the most to the original prediction results, and then modify them slightly by controlling the semantic similarity. One reasonable choice is to directly measure the effect of removing the i^{th} word, since

TABLE I. RESULTS OF THE BLACK-BOX ATTACK ON IMDB.

Targeted Model	Original Accuracy	DeepWordBug [1]			TEXTBUGGER		
		Success Rate	Time (s)	Perturbed Word	Success Rate	Time (s)	Perturbed Word
Google Cloud NLP	85.3%	43.6%	266.69	10%	**70.1%**	33.47	1.9%
IBM Waston	89.6%	34.5%	690.59	10%	**97.1%**	99.28	8.6%
Microsoft Azure	89.6%	56.3%	182.08	10%	**100.0%**	23.01	5.7%
Amazon AWS	75.3%	68.1%	43.98	10%	**100.0%**	4.61	1.2%
Facebook fastText	86.7%	67.0%	0.14	10%	**85.4%**	0.03	5.0%
ParallelDots	63.5%	79.6%	812.82	10%	**92.0%**	129.02	2.2%
TheySay	86.0%	9.5%	888.95	10%	**94.3%**	134.03	4.1%
Aylien Sentiment	70.0%	63.8%	674.21	10%	**90.0%**	44.96	1.4%
TextProcessing	81.7%	57.3%	303.04	10%	**97.2%**	59.42	8.9%
Mashape Sentiment	88.0%	31.1%	585.72	10%	**65.7%**	117.13	6.1%

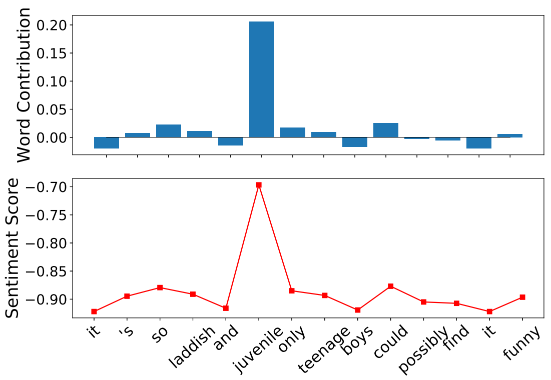


Fig. 2. Illustration of how to select important words to apply perturbations.

Algorithm 2 Bug Selection algorithm

```
1: function SELECTBUG($w, \boldsymbol{x}, y, \mathcal{F}(\cdot)$)
2:     $bugs = BugGenerator(w)$;
3:     for $b_k$ in $bugs$ do
4:         $\boldsymbol{candidate(k)}$ = replace $w$ with $b_k$ in $\boldsymbol{x}$;
5:         $score(k) = \mathcal{F}_y(\boldsymbol{x}) - \mathcal{F}_y(\boldsymbol{candidate(k)})$;
6:     end for
7:     $bug_{best} = \arg\max_{b_k} score(k)$;
8:     return $bug_{best}$;
9: end function
```

comparing the prediction before and after removing a word reflects how the word influences the classification result as shown in Fig. 2. Therefore, we introduce a scoring function that determine the importance of the j^{th} word in $\boldsymbol{x}$ as:

$$C_{w_j} = \mathcal{F}_y(w_1, w_2, \cdots, w_m) - \mathcal{F}_y(w_1, \cdots, w_{j-1}, w_{j+1}, \cdots, w_m) \quad (1)$$

Step 3: Bugs Generation (line 12-20). We propose five bug generation methods for TEXTBUGGER: (1) **Insert**: insert a space into the word; (2) **Delete**: delete a random character of the word except for the first and the last character; (3) **Swap**: swap random two adjacent letters in the word but do not alter the first or last letter; (4) **Substitute-C** : replace characters with visually similar characters (e.g., replacing "o" with "0") or adjacent characters in the keyboard; (5) **Substitute-W** : replace a word with its top_k nearest neighbors in a context-aware word vector space. As shown in Algorithm 2, after generating five bugs, we choose the optimal bug according to the change of the confidence value, i.e., choosing the bug that decreases the confidence value of the ground truth class the most. Then we will replace the word with the optimal bug to obtain a new text $\boldsymbol{x'}$ (line 14). If the classifier gives the new text a different label (i.e., $\mathcal{F}_l(\boldsymbol{x'}) \neq y$) while preserving the semantic similarity above the threshold (i.e., $S(\boldsymbol{x}, \boldsymbol{x'}) \geq \epsilon$), the adversarial text is found (line 15-19). If not, we repeat the above steps to replace the next word in $W_{ordered}$ until we find the solution or fail to find a semantic-preserving adversarial example.

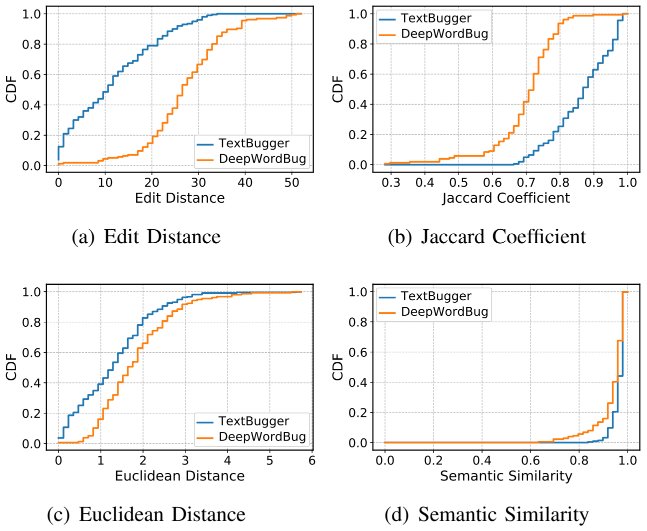


Fig. 3. The average utility of adversarial texts.

II. EXPERIMENTS AND CONCLUSION

We study adversarial attacks against sentiment analysis platforms under both white-box and black-box settings. The main experimental results on IMDB dataset (which contains 25,000 positive and 25,000 negative movie reviews) under black-box setting are shown in Table I and Fig. 3, from which we can see that (i) TEXTBUGGER achieves higher attack success rate than DeepWordBug [1]; and (ii) TEXTBUGGER is more utility-preserving than DeepWordBug in both word-level and vector-level.

REFERENCES

[1] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi, "Black-box generation of adversarial text sequences to evade deep learning classifiers," *arXiv preprint arXiv:1801.04354*, 2018.

[2] C. Szegedy, "Intriguing properties of neural networks," in *ICLR*, 2014, pp. 1–10.

TextBugger: Generating Adversarial Text Against Real-world Applications

Jinfeng Li[1] Shouling Ji[1] Tianyu Du[1] Bo Li[2] Ting Wang[3]

1.Zhejiang University 2.University of Illinois Urbana-Champaign 3. Lehigh University

Introduction

- Recently, Deep neural networks (DNNs) have been found to be vulnerable against adversarial examples generated by adding small perturbations to the legitimate inputs to fool the targeted models [2]. Such discovery has also raised serious concerns, especially when deploying such machine learning models to security-sensitive tasks.
- We present **TextBugger**, a general attack framework for generating adversarial texts. In contrast to prior works, TextBugger differs in significant ways: (i) ***effective*** -- it outperforms state-of-the-art attacks in terms of success rate; (ii) ***evasive*** -- it preserves the utility of benign text; and (iii) ***efficient*** -- it generates adversarial text with computational complexity sub-linear o the text length. Experimental results demonstrate its effectiveness, evasiveness, and efficiency.

TextBugger

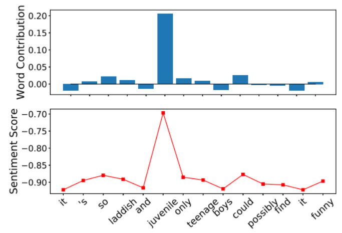
- **Step 1:** Find important sentences.
  - Segment documents into sentences
  - Filter out sentences that have different labels
  - Sort in an inverse order according to their score
- **Step 2:** Find important words.

White-box

$$J_{\mathcal{F}}(\boldsymbol{x}) = \frac{\partial \mathcal{F}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \left[\frac{\partial \mathcal{F}_j(\boldsymbol{x})}{\partial x_i}\right]_{i \in 1..N, j \in 1..K}$$

$$C_{x_i} = J_{\mathcal{F}(i,y)} = \frac{\partial \mathcal{F}_y(\boldsymbol{x})}{\partial x_i}$$

Black-box



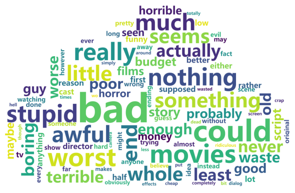
- **Step 3:** Bugs generation.
  - **Insert:** insert a space into the word
  - **Delete:** delete a random character of the word except for the first and the last character
  - **Swap:** swap random two adjacent letters in the word but do not alter the first or last letter
  - **Substitute-C:** replace characters with visually similar characters (e.g., replacing “o” with “0”) or adjacent characters in the keyboard
  - **Substitute-W**: replace a word with its top$_k$ nearest neighbors in a context-aware word vector space

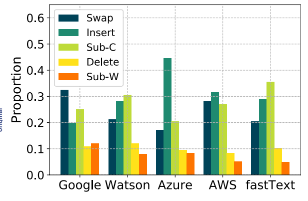
Examples:

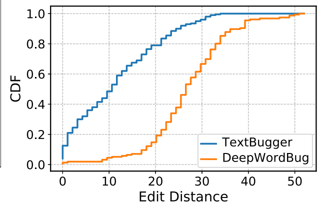
Original	Insert	Delete	Swap	Sub-C	Sub-W
foolish	f oolish	folish	fooilsh	fo0lish	silly
awfully	awfull y	awfuly	awfluly	awfu1ly	terribly
cliches	clich es	clichs	clcihes	c1iches	cliche

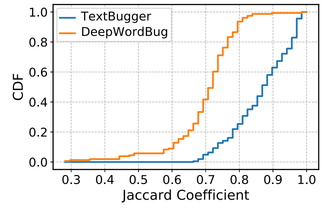
- **Step 4:** Choose the optimal bug to substitute the current word.

Evaluation



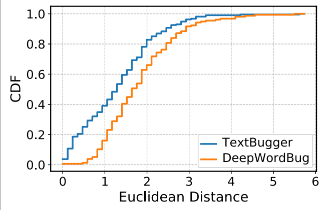


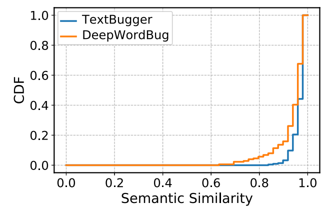




Task: Sentiment Analysis ***Classifier:*** Amazon AWS
Original label: 100% Negative ***Adversarial label:*** 89% Positive

Text: I watched this movie recently mainly because I am a Huge fan of Jodie Foster's. I saw this movie was made right between her 2 Oscar award winning performances, so my expectations were fairly high. ~~Unfortunately~~ **Unf0rtunately**, I thought the movie was ~~terrible~~ **terrib1e** and I'm still left wondering how she was ever persuaded to make this movie. The script is really ~~weak~~ **wea k**.





Targeted Model	Original Accuracy	DeepWordBug [1]			TextBugger		
		Success Rate	Time (s)	Perturbed Word	Success Rate	Time (s)	Perturbed Word
Google Cloud NLP	85.3%	43.6%	266.69	10%	**70.1%**	33.47	1.9%
IBM Waston	89.6%	34.5%	690.59	10%	**97.1%**	99.28	8.6%
Microsoft Azure	89.6%	56.3%	182.08	10%	**100.0%**	23.01	5.7%
Amazon AWS	75.3%	68.1%	43.98	10%	**100.0%**	4.61	1.2%
Facebook fastText	86.7%	67.0%	0.14	10%	**85.4%**	0.03	5.0%
ParallelDots	63.5%	79.6%	812.82	10%	**92.0%**	129.02	2.2%
TheySay	86.0%	9.5%	888.95	10%	**94.3%**	134.03	4.1%
Aylien Sentiment	70.0%	63.8%	674.21	10%	**90.0%**	44.96	1.4%
TextProcessing	81.7%	57.3%	303.04	10%	**97.2%**	59.42	8.9%
Mashape Sentiment	88.0%	31.1%	585.72	10%	**65.7%**	117.13	6.1%

Reference

[1] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi, “Black-box generation of adversarial text sequences to evade deep learning classifiers,” arXiv preprint arXiv:1801.04354, 2018
[2] C. Szegedy, “Intriguing properties of neural networks,” in ICLR, 2014, pp. 1–10.