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*—Deep Learning-based Text Understanding (DLTU) is the backbone technique behind various applications, including question answering, machine translation, and text classification. Despite its tremendous popularity, the security vulnerabilities of DLTU are still largely unknown, which is highly concerning given its increasing use in security-sensitive applications such as sentiment analysis and toxic content detection. In this paper, we show that DLTU is inherently vulnerable to adversarial text attacks, in which maliciously crafted texts trigger target DLTU systems and services to misbehave. Specifically, 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 attack success rate; (ii) evasive – it preserves the utility of benign text, with 94.9% of the adversarial text correctly recognized by human readers; and (iii) efficient – it generates adversarial text with computational complexity sub-linear to the text length. We empirically evaluate TEXTBUGGER on a set of real-world DLTU systems and services used for sentiment analysis and toxic content detection, demonstrating its effectiveness, evasiveness, and efficiency. For instance, TEXTBUGGER achieves 100% success rate on the IMDB dataset based on Amazon AWS Comprehend within 4.61 seconds and preserves 97% semantic similarity. We further discuss possible defense mechanisms to mitigate such attack and the adversary's potential countermeasures, which leads to promising directions for further research.**

I. INTRODUCTION

Deep neural networks (DNNs) have been shown to achieve great success in various tasks such as classification, regression, and decision making. Such advances in DNNs have led to broad deployment of systems on important problems in physical world. However, though DNNs models have exhibited state-of-the-art performance in a lot of applications, recently they 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 [8, 13, 20, 25, 36, 37]. Such discovery has also raised serious concerns, especially when deploying such machine learning models to security-sensitive tasks.

✉ Shouling Ji is the corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23138
www.ndss-symposium.org

In the meantime, DNNs-based text classification plays a more and more important role in information understanding and analysis nowadays. For instance, many online recommendation systems rely on the sentiment analysis of user reviews/comments [22]. Generally, such systems would classify the reviews/comments into two or three categories and then take the results into consideration when ranking movies/products. Text classification is also important for enhancing the safety of online discussion environments, e.g., automatically detect online toxic content [26], including irony, sarcasm, insults, harassment and abusive content.

Many studies have investigated the security of current machine learning models and proposed different attack methods, including causative attacks and exploratory attacks [2, 3, 15]. Causative attacks aim to manipulate the training data thus misleading the classifier itself, and exploratory attacks craft malicious testing instances (adversarial examples) so as to evade a given classifier. To defend against these attacks, several mechanisms have been proposed to obtain robust classifiers [5, 34]. Recently, adversarial attacks have been shown to be able to achieve a high attack success rate in image classification tasks [6], which has posed severe physical threats to many intelligent devices (e.g., self-driving cars) [10].

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 image domain, the perturbation can often be made virtually imperceptible to human perception, causing humans and state-of-the-art models to disagree. However, 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. In general, existing attack algorithms designed for images cannot be directly applied to text, and we need to study new attack techniques and corresponding defenses.

Recently, some mechanisms are proposed towards generating adversarial texts [19, 33]. These work proposed to generate adversarial texts by replacing a word with an out-of-vocabulary one [4, 11, 14]. Although seminal, they are limited in practice due to the following reasons: (i) they are not computationally efficient, (ii) they are designed under the white-box setting, (iii) they require manual intervention, and/or (iv) they are designed against a particular NLP model and are not comprehensively evaluated. Thus, the efficiency and effectiveness of current adversarial text generation techniques and the robustness of

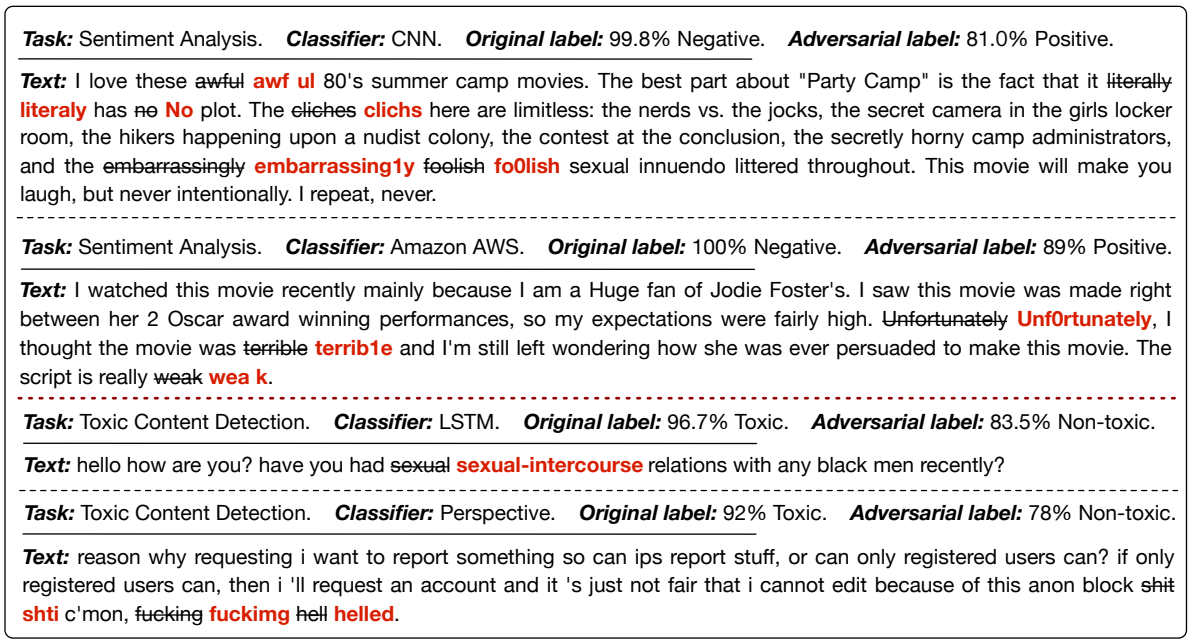


Fig. 1. Adversarial examples against two natural language classification tasks. Replacing a fraction of the words in a document with adversarially-chosen bugs fools classifiers into predicting an incorrect label. The new document is classified correctly by humans and preserves most of the original meaning although it contains small perturbations.

popular text classification models need to be studied.

In this paper, we propose TEXTBUGGER, a framework that can effectively and efficiently generate utility-preserving (i.e., keep its original meaning for human readers) adversarial texts against state-of-the-art text classification systems under both white-box and black-box settings. In the white-box scenario, we first find important words by computing the Jacobian matrix of the classifier and then choose an optimal perturbation from the generated five kinds of perturbations. In the black-box scenario, we first find the important sentences, and then use a scoring function to find important words to manipulate. Through extensive experiments under both settings, we show that an adversary can deceive multiple real-world online DLTU systems with the generated adversarial texts[1], including Google Cloud NLP, Microsoft Azure Text Analytics, IBM Watson Natural Language Understanding and Amazon AWS Comprehend, etc. Several adversarial examples are shown in Fig. 1. The existence of such adversarial examples causes a lot of concerns for text classification systems and seriously undermines their usability.

Our Contribution. Our main contributions can be summarized as follows.

- We propose TEXTBUGGER, a framework that can effectively and efficiently generate utility-preserving adversarial texts under both white-box and black-box settings.
- We evaluate TEXTBUGGER on a group of state-of-the-art machine learning models and popular real-world online DLTU applications, including sentiment analysis and toxic content detection. Experimental results show that TEXTBUGGER is very effective and efficient. For instance, TEXTBUGGER achieves 100% attack success rate on the IMDB dataset when targeting the Amazon AWS and Microsoft Azure platforms under black-box settings. We shows that transferability also exists in the text domain and the adversarial texts generated against offline models can be successfully transferred to multiple popular online DLTU systems.
- We conduct a user study on our generated adversarial texts and show that TEXTBUGGER has little impact on human understanding.
- We further discuss two potential defense strategies to defend against the above attacks along with preliminary evaluations. Our results can encourage building more robust DLTU systems in the future.

[1]We have reported our findings to their companies, and they replied that they would fix these bugs in the next version.

II. ATTACK DESIGN

A. Problem Formulation

Given a pre-trained text classification model $\mathcal{F} : \mathcal{X} \rightarrow \mathcal{Y}$, which maps from feature space $\mathcal{X}$ to a set of classes $\mathcal{Y}$, an adversary aims to generate an adversarial document $\boldsymbol{x_{adv}}$ from a legitimate document $\boldsymbol{x} \in \mathcal{X}$ whose ground truth label is $y \in \mathcal{Y}$, so that $\mathcal{F}(\boldsymbol{x_{adv}}) = t$ $(t \neq y)$. The adversary also requires $S(\boldsymbol{x}, \boldsymbol{x_{adv}}) \geq \epsilon$ for a domain-specific similarity function $S : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$, where the bound $\epsilon \in \mathbb{R}$ captures the notion of utility-preserving alteration. For instance, in the context of text classification tasks, we may use S to capture the semantic similarity between $\boldsymbol{x}$ and $\boldsymbol{x_{adv}}$.

B. Threat Model

We consider both white-box and black-box settings to evaluate different adversarial abilities.

White-box Setting. We assume that attackers have complete knowledge about the targeted model including the model

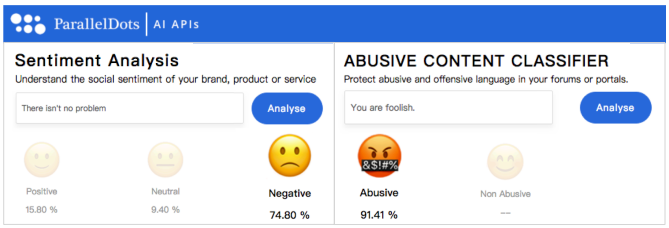


Fig. 2. ParallelDots API: An example of deep learning text classification platform, which is a black-box scenario.

architecture parameters. White-box attacks find or approximate the worst-case attack for a particular model and input based on the kerckhoff's principle [35]. Therefore, white-box attacks can expose a model's worst case vulnerabilities.

Black-box Setting. With the development of machine learning, many companies have launched their own Machine-Learning-as-a-Service (MLaaS) for DLTU tasks such as text classification. Generally, MLaaS platforms have similar system design: the model is deployed on the cloud servers, and users can only access the model via an API. In such cases, we assume that the attacker is not aware of the model architecture, parameters or training data, and is only capable of querying the target model with output as the prediction or confidence scores. Note that the free usage of the API is limited among these platforms. Therefore, if the attackers want to conduct practical attacks against these platforms, they must take such limitation and cost into consideration. Specifically, we take the ParallelDots[2] as an example and show its sentiment analysis API and the abusive content classifier API in Fig. 2. From Fig. 2, we can see that the sentiment analysis API would return the confidence value of three classes, i.e., "positive", "neutral" and "negative". Similarly, the abusive content classifier would return the confidence value of two classes, i.e., "abusive" and "non abusive". For both APIs, the sum of confidence values of an instance equal to 1, and the class with the highest confidence value is considered as the input's class.

C. TEXTBUGGER

We propose efficient strategies to change a word slightly, which is sufficient for creating adversarial texts in both white-box settings and black-box settings. Specifically, we call the slightly changed words "bugs".

1) White-box Attack: We first find important words by computing the Jacobian matrix of the classifier $\mathcal{F}$, and generate five kinds of bugs. Then we choose an optimal bug in terms of the change of the confidence value. The algorithm of white-box attack is shown in Algorithm 1.

Step 1: Find Important Words (line 2-5). The first step is to compute the Jacobian matrix for the given input text $\boldsymbol{x} = (x_1, x_2, \cdots, x_N)$ (line 2-4), where x_i is the i^{th} word, and N represents the total number of words within the input text. For a text classification task, the output of $\mathcal{F}$ is more than one dimension. Therefore the matrix is as follows:

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

[2]https://www.paralleldots.com/

Algorithm 1 TEXTBUGGER under white-box settings

Input: legitimate document $\boldsymbol{x}$ and its ground truth label y, classifier $\mathcal{F}(\cdot)$, threshould ϵ
Output: adversarial document $\boldsymbol{x_{adv}}$
1: Inititialize: $\boldsymbol{x'} \leftarrow \boldsymbol{x}$
2: **for** word x_i in $\boldsymbol{x}$ **do**
3: Compute C_{x_i} according to Eq.2;
4: **end for**
5: $W_{ordered} \leftarrow Sort(x_1, x_2, \cdots, x_m)$ according to C_{x_i};
6: **for** x_i in $W_{ordered}$ **do**
7: $bug = SelectBug(x_i, \boldsymbol{x'}, y, \mathcal{F}(\cdot))$;
8: $\boldsymbol{x'} \leftarrow$ replace x_i with bug in $\boldsymbol{x'}$
9: **if** $S(\boldsymbol{x}, \boldsymbol{x'}) \leq \epsilon$ **then**
10: Return None.
11: **else if** $\mathcal{F}_l(\boldsymbol{x'}) \neq y$ **then**
12: Solution found. Return $\boldsymbol{x'}$.
13: **end if**
14: **end for**
15: **return** None

where K represents the total number of classes in $\mathcal{Y}$, and $\mathcal{F}_j(\cdot)$ represents the confidence value of the j^{th} class. The importance of word x_i is defined as:

$$C_{x_i} = J_{\mathcal{F}(i,y)} = \frac{\partial \mathcal{F}_y(\boldsymbol{x})}{\partial x_i} \quad (2)$$

i.e., the partial derivative of the confidence value based on the predicted class y regarding to the input word x_i. This allows us to find the important words that have significant impact on the classifier's outputs. Once we have calculated the importance score of each word within the input sequences, we sort these words in inverse order according to the importance value (line 5).

Step 2: Bugs Generation (line 6-14). To generate bugs, many operations can be used. However, we prefer small changes to the original words as we require the generated adversarial sentence is visually and semantically similar to the original one for human understanding. Therefore, we consider two kinds of perturbations, i.e., character-level perturbation and word-level perturbation.

For character-level perturbation, one key observation is that words are symbolic, and learning-based DLTU systems usually use a dictionary to represent a finite set of possible words. The size of the typical word dictionary is much smaller than the possible combinations of characters at a similar length (e.g., about 26^n for the English case, where n is the length of the word). This means if we deliberately misspell important words, we can easily convert those important words to "unknown" (i.e., words not in the dictionary). The unknown words will be mapped to the "unknown" embedding vector in deep learning modeling. Our results strongly indicate that such simple strategy can effectively force text classification models to behave incorrectly.

For word-level perturbation, we expect that the classifier can be fooled after replacing a few words, which are obtained by nearest neighbor searching in the embedding space, without changing the original meaning. However, we found that in some word embedding models (e.g., word2vec), semantically opposite words such as "worst" and "better" are highly syntactically similar in texts, thus "better" would be considered as

the nearest neighbor of "worst". However, changing "worst" to "better" would completely change the sentiment of the input text. Therefore, we make use of a semantic-preserving technique, i.e., replace the word with its top_k nearest neighbors in a context-aware word vector space. Specifically, we use the pre-trained GloVe model [30] provided by Stanford for word embedding and set $top_k = 5$ in the experiment. Thus, the neighbors are guaranteed to be semantically similar to the original one.

According to previous studies, the meaning of the text is very likely to be preserved or inferred by the reader after a few character changes [31]. Meanwhile, replacing words with semantically and syntactically similar words can ensure that the examples are perceptibly similar [1]. Based on these observations, we propose five bug generation methods for TEXTBUGGER: (1) **Insert**: Insert a space into the word[3]. Generally, words are segmented by spaces in English. Therefore, we can deceive classifiers by inserting spaces into words. (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]. This is a common occurrence when typing quickly and is easy to implement. (4) **Substitute-C (Sub-C)**: Replace characters with visually similar characters (e.g., replacing "o" with "0", "l" with "1", "a" with "@") or adjacent characters in the keyboard (e.g., replacing "m" with "n"). (5) **Substitute-W (Sub-W)**: Replace a word with its top_k nearest neighbors in a context-aware word vector space. Several substitute examples are shown in Table I.

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 8). If the classifier gives the new text a different label (i.e., $\mathcal{F}_l(\boldsymbol{x'}) \neq y$) while preserving the semantic similarity (which is detailed in Section III-D) above the threshold (i.e., $S(\boldsymbol{x}, \boldsymbol{x'}) \geq \epsilon$), the adversarial text is found (line 9-13). If not, we repeat above steps to replace the next word in $W_{ordered}$ until we find the solution or fail to find a semantic-preserving adversarial example.

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**

2) Black-box Attack: Under the black-box setting, gradients of the model are not directly available, and we need to change the input sequences directly without the guidance of gradients. Therefore different from white-box attacks, where we can directly select important words based on gradient information, in black-box attacks, we will first find important sentences and then the important words within them. Briefly, the process of generating word-based adversarial examples on text 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 3.

TABLE I. EXAMPLES FOR FIVE BUG GENERATION METHODS.

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

Algorithm 3 TEXTBUGGER under black-box settings

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

Step 1: Find Important Sentences (line 2-6). Generally, when people express their opinions, most of the sentences are describing facts and the main opinions usually depend on only a few of sentences which have a greater impact on the classification results. Therefore, to improve the efficiency of TEXTBUGGER, we first find the important sentences that contribute to the final prediction results most and then prioritize to manipulate them.

Suppose the input document $\boldsymbol{x} = (\boldsymbol{s_1}, \boldsymbol{s_2}, \cdots, \boldsymbol{s_n})$, where $\boldsymbol{s_i}$ represents the sentence at the i^{th} position. First, we use the spaCy library[5] to segment each document into sentences. Then we filter out the sentences that have different predicted

[3]Considering the usability of text, we apply this method only when the length of the word is shorter than 6 characters since long words might be split into two legitimate words.

[4]For this reason, this method is only applied to words longer than 4 letters.

[5]http://spacy.io

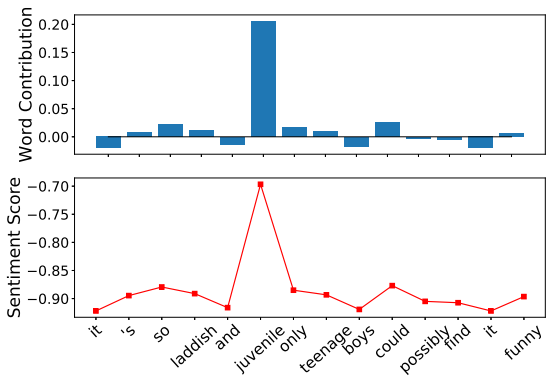


Fig. 3. Illustration of how to select important words to apply perturbations for the input sentence "*It is so laddish and juvenile, only teenage boys could possibly find it funny*". The sentiment score of each word is the classification result's confidence value of the new text that deleting the word from the original text. The contribution of each word is the difference between the new confidence score and the original confidence score.

labels with the original document label (i.e., filter out $\mathcal{F}_l(\boldsymbol{s_i}) \neq y$). Then, we sort the important sentences in an inverse order according to their importance score. The importance score of a sentence $\boldsymbol{s_i}$ is represented with the confidence value of the predicted class $\mathcal{F}_y$, i.e., $C_{s_i} = \mathcal{F}_y(\boldsymbol{s_i})$.

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 comparing the prediction before and after removing a word reflects how the word influences the classification result as shown in Fig. 3. Therefore, we introduce a scoring fuction that determine the importance of the j^{th} word in $\boldsymbol{x}$ as:

$$C_{w_j} = \mathcal{F}_y(w_1, w_2, \cdots, w_m) - \mathcal{F}_y(w_1, \cdots, w_{j-1}, w_{j+1}, \cdots, w_m) \quad (3)$$

The proposed scoring function has the following properties: (1) It is able to correctly reflect the importance of words for the prediction, (2) it calculates word scores without the knowledge of the parameters and structure of the classification model, and (3) it is efficient to calculate.

Step 3: Bugs Generation (line 12-20). This step is similar as that in white-box setting.

III. Attack Evaluation: Sentiment Analysis

Sentiment analysis refers to the use of NLP, statistics, or machine learning methods to extract, identify or characterize the sentiment content of a text unit. It is widely applied to helping a business understand the social sentiment of their products or services by monitoring online conversations.

In this section, we investigate the practical performance of the proposed method for generating adversarial texts for sentiment analysis. We start with introducing the datasets, targeted models, baseline algorithms, evaluation metrics and implementation details. Then we will analyze the results and discuss potential reasons for the observed performance.

A. Datasets

We study adversarial examples of text on two popular public benchmark datasets for sentiment analysis. The final adversarial examples are generated and evaluated on the test set.

IMDB [21]. This dataset contains 50,000 positive and negative movie reviews that crawled from online sources, with 215.63 words as average length for each sample. It has been divided into two parts, i.e., 25,000 reviews for training and 25,000 reviews for testing. Specifically, we held out 20% of the training set as a validation set and all parameters are tuned based on it.

Rotten Tomatoes Movie Reviews (MR) [27]. This dataset is a collection of movie reviews collected by Pang and Lee in [27]. It contains 5,331 positive and 5,331 negative processed sentences/snippets and has an average length of 32 words. In our experiment, we divide this dataset into three parts, i.e., 80%, 10%, 10% as training, validation and testing, respectively.

B. Targeted Models

For white-box attacks, we evaluated TextBugger on LR, Kim's CNN [17] and the LSTM used in [38]. In our implementation, the model's parameters are fine-tuned according to the sensitivity analysis on model performance conducted by Zhang *et al.* [39]. Meanwhile, all models were trained in a hold-out test strategy, and hyper-parameters were tuned only on the validation set.

For black-box attacks, we evaluated the TextBugger on ten sentiment analysis platforms/models, i.e., Google Cloud NLP, IBM Waston Natural Language Understanding (IBM Watson), Microsoft Azure Text Analytics (Microsoft Azure), Amazon AWS Comprehend (Amazon AWS), Facebook fastText (fastText), ParallelDots, TheySay Sentiment, Aylien Sentiment, TextProcessing, and Mashape Sentiment. For fastText, we used a pre-trained model[6] provided by Facebook. This model is trained on the Amazon Review Polarity dataset and we do not have any information about the models' parameters or architecture.

C. Baseline Algorithms

We implemented and compared the other three methods with our white-box attack method. In total, the three methods are: (1) **Random:** Randomly selects words to modify. For each sentence, we select 10% words to modify. (2) **FGSM+Nearest Neighbor Search (NNS):** The FGSM method was first proposed in [13] to generate adversarial images, which adds to the whole image the noise that is proportional to $\text{sign}(\nabla(L_x))$, where L represent the loss function and x is the input data. It was combined with NNS to generate adversarial texts as in [12]: first, generating adversarial embeddings by applying FGSM on the embedding vector of the texts, then reconstructing the adversarial texts via NNS. (3) **DeepFool+NNS:** The DeepFool method is first proposed in [24] to generate adversarial images, which iteratively finds the optimal direction to search for the minimum distance to cross the decision

[6]https://s3-us-west-1.amazonaws.com/fasttext-vectors/supervised_models/amazon_review_polarity.bin

boundary. It was combined with NNS to generate adversarial texts as in [12].

D. Evaluation Metrics

We use four metrics, i.e., edit distance, Jaccard similarity coefficient, Euclidean distance and semantic similarity, to evaluate the utility of the generated adversarial texts. Specifically, the edit distance and Jaccard similarity coefficient are calculated on the raw texts, while the Euclidean distance and semantic similarity are calculated on word vectors.

Edit Distance. Edit distance is a way of quantifying how dissimilar two strings (e.g., sentences) are by counting the minimum number of operations required to transform one string to the other. Specifically, different definitions of the edit distance use different sets of string operations. In our experiment, we use the most common metrics, i.e., the Levenshtein distance, whose operations include removal, insertion, and substitution of characters in the string.

Jaccard Similarity Coefficient. The Jaccard similarity coefficient is a statistic used for measuring the similarity and diversity of finite sample sets. It is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{4}$$

Larger Jaccard similarity coefficient means higher sample similarity. In our experiment, one sample set consists of all the words in the sample.

Euclidean Distance. Euclidean distance is a measure of the true straight line distance between two points in the Euclidean space. If $\boldsymbol{p} = (p_1, p_2, \cdots, p_n)$ and $\boldsymbol{q} = (q_1, q_2, \cdots, q_n)$ are two samples in the word vector space, then the Euclidean distance between $\boldsymbol{p}$ and $\boldsymbol{q}$ is given by:

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} \tag{5}$$

In our experiment, the Euclidean space is exactly the word vector space.

Semantic Similarity. The above three metrics can only reflect the magnitude of the perturbation to some extent. They cannot guarantee that the generated adversarial texts will preserve semantic similarity from original texts. Therefore, we need a fine-grained metric that measures the degree to which two pieces of text carry the similar meaning so as to control the quality of the generated adversarial texts.

In our experiment, we first use the Universal Sentence Encoder [7], a model trained on a number of natural language prediction tasks that require modeling the meaning of word sequences, to encode sentences into high dimensional vectors. Then, we use the cosine similarity to measure the semantic similarity between original texts and adversarial texts. The cosine similarity of two n-dimensional vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as:

$$S(\boldsymbol{p}, \boldsymbol{q}) = \frac{\boldsymbol{p} \cdot \boldsymbol{q}}{||\boldsymbol{p}|| \cdot ||\boldsymbol{q}||} = \frac{\sum_{i=1}^{n} p_i \times q_i}{\sqrt{\sum_{i=1}^{n} (p_i)^2} \times \sqrt{\sum_{i=1}^{n} (q_i)^2}} \tag{6}$$

Generally, it works better than other distance measures because the norm of the vector is related to the overall frequency of which words occur in the training corpus. The direction of a vector and the cosine distance is unaffected by this, so a common word like "frog" will still be similar to a less frequent word like "Anura" which is its scientific name.

Since our main goal is to successfully generate adversarial texts, we only need to control the semantic similarity to be above a specific threshold.

E. Implementation

We conducted the experiments on a server with two Intel Xeon E5-2640 v4 CPUs running at 2.40GHz, 64 GB memory, 4TB HDD and a GeForce GTX 1080 Ti GPU card. We repeated each experiment 5 times and report the mean value. This replication is important because training is stochastic and thus introduces variance in performance [39].

In our experiment, we did not filter out stop-words before feature extraction as most NLP tasks do. This is because we observe that the stop-words also have impact on the prediction results. In particular, our experiments utilize the 300-dimension GloVe embeddings[7] trained on 840 billion tokens of Common Crawl. Words not present in the set of pre-trained words are initialized by randomly sampling from the uniform distribution in [-0.1, 0.1]. Furthermore, the semantic similarity threshold ϵ is set as 0.8 to guarantee a good trade-off between quality and strength of the generated adversarial text.

F. Attack Performance

Effectiveness and Efficiency. The main results of white-box attacks on the IMDB and MR datasets and comparison of the performance of baseline methods are summarized in Table II, where the third column of Table II shows the original model accuracy in non-adversarial setting. We do not give the average time of generating one adversarial example under white-box settings since the models are offline and the attack is very efficient (e.g., generating hundreds of adversarial texts in one second). From Table II, we can see that randomly choosing words to change (i.e., Random in Table II) has hardly any influence on the final result. This implies randomly changing words would not fool classifiers and choosing important words to modify is necessary for successful attack. From Table II, we can also see that the targeted models all perform quite well in non-adversarial setting. However, the adversarial texts generated by TEXTBUGGER still has high attack success rate on these models. In addition, the linear model is more susceptible to adversarial texts than deep learning models. Specifically, TEXTBUGGER only perturbs a few words to achieve a high attack success rate and performs much better than baseline algorithms against all models as shown in Table II. For instance, it only perturbs 4.9% words of one sample when achieving 95.2% success rate on the IMDB dataset against the LR model, while all baselines achieve no more than 42% success rate in this case. As the IMDB dataset has an average length of 215.63 words, TEXTBUGGER only perturbed about 10 words for one sample to conduct successful attacks. This means that TEXTBUGGER can successfully mislead the classifiers into assigning significantly higher positive scores to the negative reviews via subtle manipulation.

[7] http://nlp.stanford.edu/projects/glove/

TABLE II. RESULTS OF THE WHITE-BOX ATTACKS ON IMDB AND MR DATASETS.

Model	Dataset	Accuracy	Random		FGSM+NNS [12]		DeepFool+NNS [12]		TEXTBUGGER	
			Success Rate	Perturbed Word	Success Rate	Perturbed Word	Success Rate	Perturbed Word	Success Rate	Perturbed Word
LR	MR	73.7%	2.1%	10%	32.4%	4.3%	35.2%	4.9%	**92.7%**	6.1%
	IMDB	82.1%	2.7%	10%	41.1%	8.7%	30.0%	5.8%	**95.2%**	4.9%
CNN	MR	78.1%	1.5%	10%	25.7%	7.5%	28.5%	5.4%	**85.1%**	9.8%
	IMDB	89.4%	1.3%	10%	36.2%	10.6%	23.9%	2.7%	**90.5%**	4.2%
LSTM	MR	80.1%	1.8%	10%	25.0%	6.6%	24.4%	11.3%	**80.2%**	10.2%
	IMDB	90.7%	0.8%	10%	31.5%	9.0%	26.3%	3.6%	**86.7%**	6.9%

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

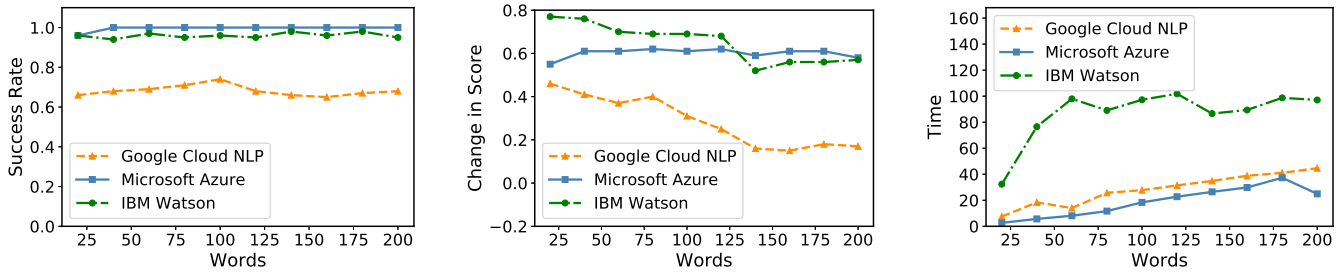
Targeted Model	Original Accuracy	DeepWordBug [11]			TEXTBUGGER		
		Success Rate	Time (s)	Perturbed Word	Success Rate	Time (s)	Perturbed Word
Google Cloud NLP	85.3%	43.6%	266.69	10%	**70.1%**	33.47	1.9%
IBM Waston	89.6%	34.5%	690.59	10%	**97.1%**	99.28	8.6%
Microsoft Azure	89.6%	56.3%	182.08	10%	**100.0%**	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%

TABLE IV. RESULTS OF THE BLACK-BOX ATTACK ON MR.

Targeted Model	Original Accuracy	DeepWordBug [11]			TEXTBUGGER		
		Success Rate	Time (s)	Perturbed Word	Success Rate	Time (s)	Perturbed Word
Google Cloud NLP	76.7%	67.3%	34.64	10%	**86.9%**	13.85	3.8%
IBM Waston	84.0%	70.8%	150.45	10%	**98.8%**	43.59	4.6%
Microsoft Azure	67.5%	71.3%	43.98	10%	**96.8%**	12.46	4.2%
Amazon AWS	73.9%	69.1%	39.62	10%	**95.7%**	3.25	4.8%
Facebook fastText	89.5%	37.0%	0.02	10%	**65.5%**	0.01	3.9%
ParallelDots	54.5%	76.6%	150.89	10%	**91.7%**	70.56	4.2%
TheySay	72.3%	56.3%	69.61	10%	**90.2%**	30.12	3.1%
Aylien Sentiment	65.3%	65.2	83.63	10%	**94.1%**	13.71	3.5%
TextProcessing	77.6%	38.1%	59.44	10%	**87.0%**	12.36	5.7%
Mashape Sentiment	72.0%	73.6%	113.54	10%	**94.8%**	18.24	5.1%

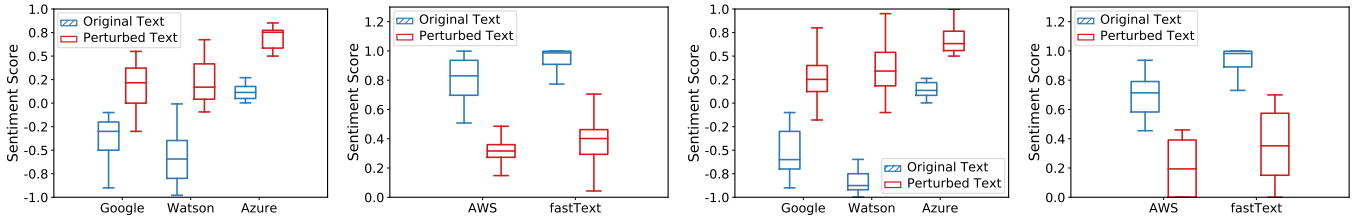
The main results of black-box attacks on the IMDB and MR datasets and comparison of the performance of different methods are summarized in Tables III and IV respectively, and the second column of which shows the original model accuracy in non-adversarial setting. From Tables III and IV, we can see that TEXTBUGGER achieves high attack success rate and performs much better than DeepWordBug [11] against all real-world online DLTU platforms. For instance, it achieves 100% success rate on the IMDB dataset when targeting Azure and AWS platforms, while DeepWordBug only achieves 56.3% and 68.1% success rate respectively. Besides, TEXTBUGGER only perturbs a few words to achieve a high success rate as shown in Tables III and IV. For instance, it only perturbs 7% words of one sample when achieving 96.8% success rate on the MR dataset targeting the Microsoft Azure platform. As the MR dataset has an average length of 32 words, TEXTBUGGER only perturbed about 2 words for one sample to conduct successful attacks. Again, that means an adversary can subtly modify highly negative reviews in a way that the classifier assigns significantly higher positive scores to them.

The Impact of Document Length. We also study the impact of document length on the effectiveness and efficiency of the attacks and the corresponding results are shown in Fig. 4. From Fig. 4(a), we can see that the document length has little impact on the attack success rate. This implies attackers can achieve high success rate no matter how long the sample is. However, the change of negative class's confidence value decrease for IBM Watson and Google Cloud NLP as shown in Fig. 4(b). This means the attack on long documents would be a bit weaker than that on short documents. From Fig. 4(c), we can see that the time required for generating one adversarial text and the average length of documents are positively correlated overall for Microsoft Azure and Google Cloud NLP. There is a very intuitive reason: the longer the length of the document is, the more information it contains that may need to be modified. Therefore, as the length of the document grows, the time required for generating one adversarial text increases slightly, since it takes more time to find important sentences. For IBM Watson, the run time first increases before 60 words, then vibrates after that. We carefully analyzed the generated adversarial texts and found that when the document length is less than 60 words, the total length of the perturbed sentences increases sharply with the growth of



(a) Success Rate (b) Score (c) Time

Fig. 4. The impact of document length (i.e. number of words in a document) on attack's performance against three online platforms: Google Cloud NLP, IBM Watson and Microsoft Azure. The sub-figures are: (a) the success rate and document length, (b) the change of negative class's confidence value. For instance, the original text is classified as negative with 90% confidence, while the adversarial text is classified as positive with 80% confidence (20% negative), the score changes 0.9-0.2=0.7. (c) the document length and the average time of generating an adversarial text.



(a) IMDB (b) IMDB (c) MR (d) MR

Fig. 5. The change of sentiment score evaluated on IMDB and MR datasets for 5 black-box platforms/models. For Google Cloud NLP (Google), IBM Watson (Watson), the range of "negative" score is [-1, 0] and the range of "positive" score is [0, 1]. For Microsoft Azure (Azure), the range of "negative" score is [0, 0.5] and the range of "positive" score is [0.5, 1]. For Amazon AWS (AWS) and fastText, the range of "negative" score is [0.5, 1] and the range of "positive" score is [0, 0.5].

document length. However, when the document length exceeds 60 words, the total length of the perturbed sentences changes negligibly. In general, generating one adversarial text only needs no more than 100 seconds for all the three platforms while the maximum length of a document is limited to 200 words. This means TEXTBUGGER method is very efficient in practice.

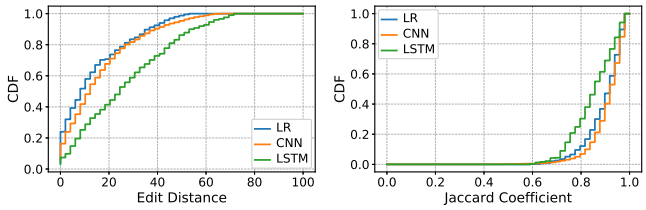
Adversarial Text Examples. Two successful examples for sentiment analysis are shown in Fig. 1. The first adversarial text for sentiment analysis in Fig. 1 contains six modifications, i.e., one insert operation ("awful" to "aw ful"), one Sub-W operation ("no" to "No"), two delete operations ("literally" to "literaly", "cliches" to "clichs"), and two Sub-C operations ("embarrassingly" to "embarrassing1y", "foolish" to "fo0lish"). These modifications successfully convert the prediction result of the CNN model, i.e., from 99.8% negative to 81.0% positive. Note that the modification from "no" to "No" only capitalizes the first letter but really affects the prediction result. After further analysis, we find capitalization operation is common for both offline models and online platforms. We guess the embedding model may be trained without changing uppercase letters to lowercase, thus causing the same word in different forms get two different word vectors. Furthermore, capitalization sometimes may cause out-of-vocabulary phenomenon. The second adversarial text for sentiment analysis in Fig. 1 contains three modifications, i.e., one insert operation ("weak" to "wea k") and two Sub-C operations ("Unfortunately" to "Unf0rtunately", "terrible" to "terrib1e"). These modifications successfully convert the prediction result of the Amazon AWS sentiment analysis API.

Score Distribution. Even though TEXTBUGGER fails to convert the negative reviews to positive reviews in some cases, it can still reduce the confidence value of the classification results. Therefore, we computed the change of the confidence value over all the samples including the failed samples before and after modification and show the results in Fig. 5. From Fig. 5, we can see that the overall score of the texts has been moved to the positive direction.

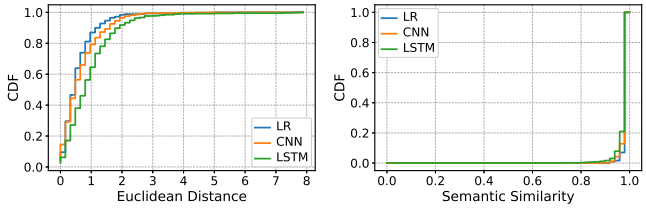
G. Utility Analysis

For white-box attacks, the similarity between original texts and adversarial texts against LR, CNN and LSTM models are shown in Figs. 6 and 7. We do not compare TEXTBUGGER with baselines in terms of utility since baselines only achieve low success rate as shown in Table V. From Figs. 6(a), 6(b), 7(a) and 7(b), we can see that adversarial texts preserve good utility in terms of word-level. Specifically, Fig. 6(a) shows that almost 80% adversarial texts have no more than 25 edit distance comparing with original texts for LR and CNN models. Meanwhile, Figs. 6(c), 6(d), 7(c) and 7(d) show that adversarial texts preserve good utility in terms of vector-level. Specifically, from Fig. 6(d), we can see that almost 90% adversarial texts preserve at least 0.9 semantic similarity of the original texts. This indicates that TEXTBUGGER can generate utility-preserving adversarial texts which fool the classifiers with high success rate.

For black-box attacks, the average similarity between original texts and adversarial texts against 10 platforms/models are shown in Figs. 8 and 9. From Figs. 8(a), 8(b), 9(a) and 9(b), we can see that the adversarial texts generated by

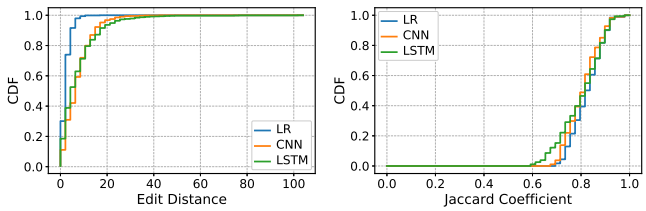


(a) Edit Distance (b) Jaccard Coefficient

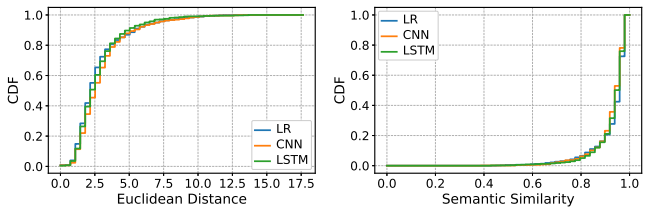


(c) Euclidean Distance (d) Semantic Similarity

Fig. 6. The utility of adversarial texts generated on IMDB dataset under white-box settings for LR, CNN and LSTM models.

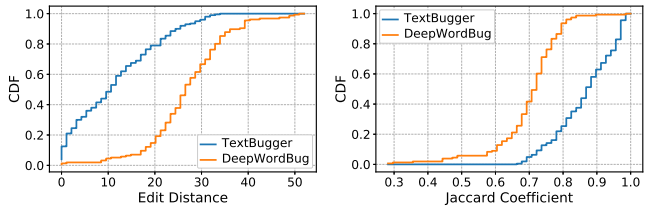


(a) Edit Distance (b) Jaccard Coefficient

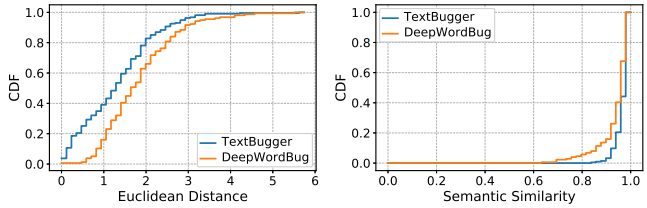


(c) Euclidean Distance (d) Semantic Similarity

Fig. 7. The utility of adversarial texts generated on MR dataset under white-box settings for LR, CNN and LSTM models.

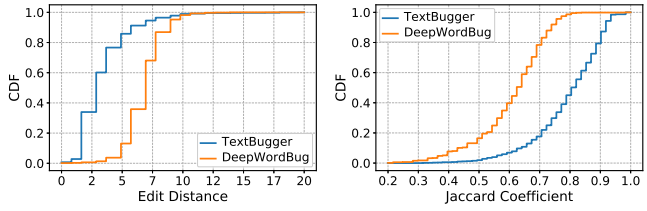


(a) Edit Distance (b) Jaccard Coefficient

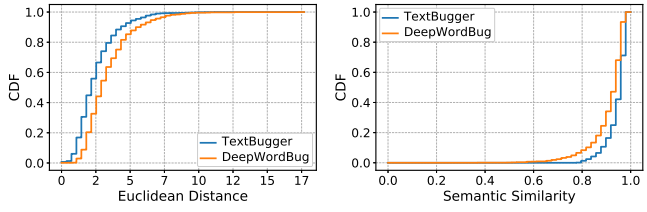


(c) Euclidean Distance (d) Semantic Similarity

Fig. 8. The average utility of adversarial texts generated on IMDB dataset under black-box settings for 10 platforms.



(a) Edit Distance (b) Jaccard Coefficient



(c) Euclidean Distance (d) Semantic Similarity

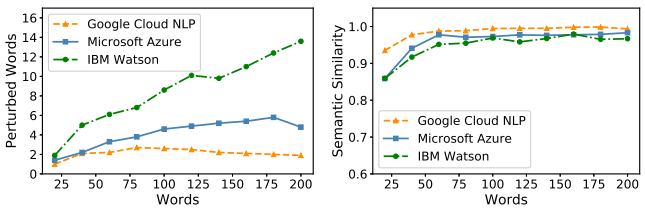
Fig. 9. The average utility of adversarial texts generated on MR dataset under black-box settings for 10 platforms.

TEXTBUGGER are more similar to original texts than that generated by DeepWordBug in word-level. From Figs. 8(c), 8(d), 9(c) and 9(d) we can see that the adversarial texts generated by TEXTBUGGER are more similar to original texts than that generated by DeepWordBug in the word vector space. These results implies that the adversarial texts generated by TEXTBUGGER preserve more utility than that generated by DeepWordBug. One reason is that DeepWordBug randomly chooses a bug from generated bugs, while TEXTBUGGER chooses the optimal bug that can change the prediction score most. Therefore, DeepWordBug needs to manipulate more words than TEXTBUGGER to achieve successful attack.

The Impact of Document Length. We also study the impact of word length on the utility of generated adversarial texts and show the results in Fig. 10. From Fig. 10(a), for IBM Watson and Microsoft Azure, we can see that the number of perturbed words roughly has a positive correlation with the average length of texts; for Google Cloud NLP, the number of perturbed words changes little with the increasing length of texts. However, as shown in Fig. 10(b), the increasing perturbed words do not decrease the semantic similarity of the adversarial texts. This is because longer text would have richer semantic information, while the proportion of the perturbed words is always controlled within a small range by TEXTBUGGER. Therefore, with the length of input text increasing, the perturbed words have smaller impact on the semantic similarity between original and adversarial texts.

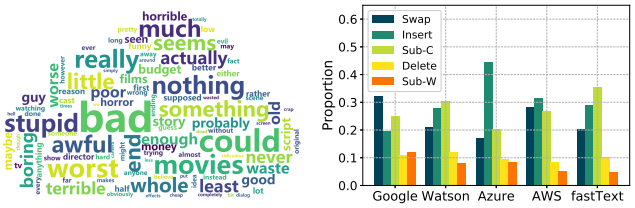
H. Discussion

Toxic Words Distribution. To demonstrate the effectiveness of our method, we visualize the found important words according to their frequency in Fig. 11(a), in which the words



(a) Number of Perturbed Words (b) Semantic Similarity

Fig. 10. The impact of document length on the utility of generated adversarial texts in three online platforms: Google Cloud NLP, IBM Watson and Microsoft Azure. The subfigures are: (a) the number of perturbed words and document length, (b) the document length and the semantic similarity between generated adversarial texts and original texts.



(a) Word Cloud (b) Bug Distribution

Fig. 11. (a) The word cloud is generated from IMDB dataset against the CNN model. (b) The bug distribution of the adversarial texts is generated from IMDB dataset against the online platforms.

higher frequency will be represented with larger font. From Fig. 11(a), we can see that the found important words are indeed negative words, e.g., "bad", "awful", "stupid", "worst", "terrible", etc for negative texts. Slight modification on these negative words would decrease the negative extent of input texts. This is why TEXTBUGGER can generate adversarial texts whose only difference to the original texts are few character-level modifications.

Types of Perturbations. The proportion of each operation chosen by the adversary for the experiments are shown in Fig. 11(b). We can see that insert is the dominant operation for Microsoft Azure and Amazon AWS, while Sub-C is the dominant operation for IBM Watson and fastText. One reason could be that Sub-C is deliberately designed for creating visually similar adversarial texts, while swap, insert and delete are common in typo errors. Therefore, the bugs generated by Sub-C are less likely to be found in the large-scale word vector space, thus causing the "out-of-vocabulary" phenomenon. Meanwhile, delete and Sub-W are used less than the others. One reason is that Sub-W should satisfy two conditions: substituting with semantic similar words while changing the score largely in the five types of bugs. Therefore, the proportion of Sub-W is less than other operations.

IV. ATTACK EVALUATION: TOXIC CONTENT DETECTION

Toxic content detection aims to apply NLP, statistics, and machine learning methods to detect illegal or toxic-related (e.g., irony, sarcasm, insults, harassment, racism, pornography, terrorism, and riots, etc.) content for online systems. Such toxic content detection can help moderators to improve the online conversation environment.

In this section, we investigate practical performance of the proposed method for generating adversarial texts against real-world toxic content detection systems. We start with introducing the datasets, targeted models and implementation details. Then we will analyze the results and discuss potential reasons for the observed performance.

A. Dataset

We apply the dataset provided by the Kaggle Toxic Comment Classification competition[8]. This dataset contains a large number of Wikipedia comments which have been labeled by human raters for toxic behavior. There are six types of indicated toxicity, i.e., "toxic", "severe toxic", "obscene", "threat", "insult", and "identity hate" in the original dataset. We consider these categories as toxic and perform binary classification for toxic content detection. For more coherent comparisons, a balanced subset of this dataset is constructed for evaluation. This is achieved by random sampling of the non-toxic texts, obtaining a subset with equal number of samples with the toxic texts. Further, we removed some abnormal texts (i.e., containing multiple repeated characters) and select the samples that have no more than 200 words for our experiment, due to the fact that some APIs limit the maximum length of input sentences. We obtained 12,630 toxic texts and non-toxic texts respectively.

B. Targeted Model & Implementation

For white-box experiments, we evaluated the TEXTBUGGER on self-trained LR, CNN and LSTM models as we do in Section III. All models are trained in a hold-out test strategy, i.e., 80%, 10%, 10% of the data was used for training, validation and test, respectively. Hyper-parameters were tuned only on the validation set, and the final adversarial examples are generated and evaluated on the test set.

For black-box experiments, we evaluated the TEXTBUGGER on five toxic content detection platforms/models, including Google Perspective, IBM Natural Language Classifier, Facebook fastText, ParallelDots AI, and Aylien Offensive Detector. Since the IBM Natural Language Classifier and the Facebook fastText need to be trained by ourselves[9], we selected 80% of the Kaggle dataset for training and the rest for testing. Note that we do not selected samples for validation since these two models only require training and testing set.

The implementation details of our toxic content attack are similar with that in the sentiment analysis attack, including the baselines.

C. Attack Performance

Effectiveness and Efficiency. Tables V and VI summarize the main results of the white-box and black-box attacks on the Kaggle dataset. We can observe that under white-box settings, the Random strategy has minor influence on the final results in Table V. On the contrary, TEXTBUGGER only perturbs a few words to achieve high attack success rate and performs much better than baseline algorithms against all models/platforms.

[8]https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge

[9]We do not know the models' parameters or architechtures because they only provide training and predicting interfaces.

TABLE V. RESULTS OF THE WHITE-BOX ATTACK ON KAGGLE DATASET.

Targeted Model	Original Accuracy	Random		FGSM+NNS [12]		DeepFool+NNS [12]		TEXTBUGGER	
		Success Rate	Perturbed Word	Success Rate	Perturbed Word	Success Rate	Perturbed Word	Success Rate	Perturbed Word
LR	88.5%	1.4%	10%	33.9%	5.4%	29.7%	7.3%	**92.3%**	10.3%
CNN	93.5%	0.5%	10%	26.3%	6.2%	27.0%	9.9%	**82.5%**	10.8%
LSTM	90.7%	0.9%	10%	28.6%	8.8%	30.3%	10.3%	**94.8%**	9.5%

TABLE VI. RESULTS OF THE BLACK-BOX ATTACK ON KAGGLE DATASET.

Targeted Platform/Model	Original Accuracy	DeepWordBug [11]			TEXTBUGGER		
		Success Rate	Time (s)	Perturbed Word	Success Rate	Time (s)	Perturbed Word
Google Perspective	98.7%	33.5%	400.20	10%	**60.1%**	102.71	5.6%
IBM Classifier	85.3%	9.1%	75.36	10%	**61.8%**	21.53	7.0%
Facebook fastText	84.3%	31.8%	0.05	10%	**58.2%**	0.03	5.7%
ParallelDots	72.4%	79.3%	148.67	10%	**82.1%**	23.20	4.0%
Aylien Offensive Detector	74.5%	53.1%	229.35	10%	**68.4%**	37.06	32.0%

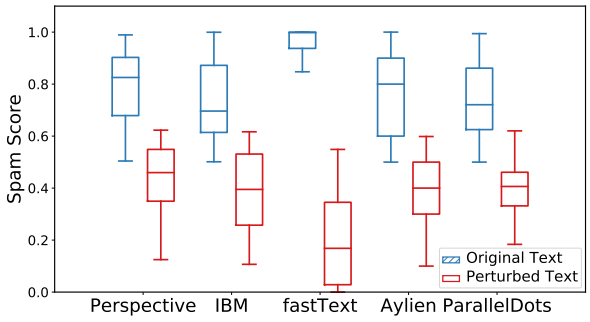


Fig. 12. Score distribution of the after-modification texts. These texts are generated from Kaggle dataset against LR model.

For instance, as shown in Table V, it only perturbs 10.3% words of one sample to achieve 92.3% success rate on the LR model, while all baselines achieve no more than 40% attack success rate. As the Kaggle dataset has an average length of 55 words, TEXTBUGGER only perturbed about 6 words for one sample to conduct successful attacks. Furthermore, as shown in Table VI, it only perturbs 4.0% words (i.e., about 3 words) of one sample when achieves 82.1% attack success rate on the ParallelDots platform. These results imply that an adversary can successfully mislead the system into assigning significantly different toxicity scores to the original sentences via modifying them slightly.

Successful Attack Examples. Two successful examples are shown in Fig. 1 as demonstration. The first adversarial text for toxic content detection in Fig. 1 contains one Sub-W operation ("sexual" to "sexual-intercourse"), which successfully converts the prediction result of the LSTM model from 96.7% toxic to 83.5% non-toxic. The second adversarial text for toxic content detection in Fig. 1 contains three modifications, i.e., one swap operation ("shit" to "shti"), one Sub-C operation ("fucking" to "fuckimg") and one Sub-W operation ("hell" to "helled"). These modifications successfully convert the prediction result of the Perspective API from 92% toxic to 78% non-toxic[10].

Score Distribution. We also measured the change of the confidence value over all the samples including the failed samples before and after modifications. The results are shown in Fig. 12, where the overall score of the after-modification texts has drifted to non-toxic for all platforms/models.

[10]Since the Perspective API only returns the toxic score, we consider that 22% toxic score is equal to 78% non-toxic score.

D. Utility Analysis

Figs. 13 and 14 show the similarity between original texts and adversarial texts under white-box and black-box settings respectively. First, Fig. 14 clearly shows that the adversarial texts generated by TEXTBUGGER preserve more utility than that generated by DeepWordBug. Second, from Figs. 13(a), 13(b), 14(a) and 14(b), we can observe that the adversarial texts preserve good utility in terms of word-level. Specifically, Fig. 13(a) shows that almost 80% adversarial texts have no more than 20 edit distance comparing with the original texts for three models. Meanwhile, Figs. 13(c), 13(d), 14(c) and 14(d) show that the generated adversarial texts preserve good utility in terms of vector-level. Specifically, from Fig. 13(d), we can see that almost 90% adverasrial texts preserve 0.9 semantic similarity of the original texts. These results imply that TEXTBUGGER can fool classifiers with high success rate while preserving good utility of the generated adversarial texts.

E. Discussion

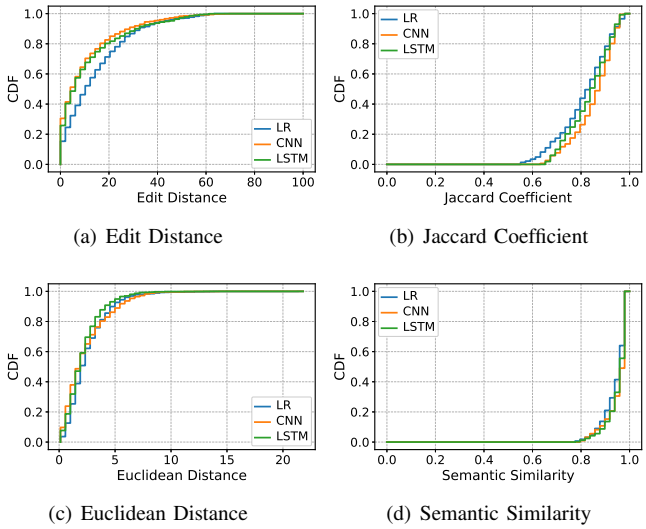
Toxic Words Distribution. Fig. 15(a) shows the visualization of the found important words according to their frequency, where the higher frequency words have larger font sizes. Observe that the found important words are indeed toxic words, e.g., "fuck", "dick", etc. It is clear that slightly perturbing these toxic words would decrease the toxic score of toxic content.

Bug Distribution. Fig. 15(b) shows the proportion of each operation chosen by the adversary for the black-box attack. Observe that Sub-C is the dominant operation for all platforms, and Sub-W is still the least used operation. We do not give detailed analysis since the results are similar to that in Section III.

V. FURTHER ANALYSIS

A. Transferability

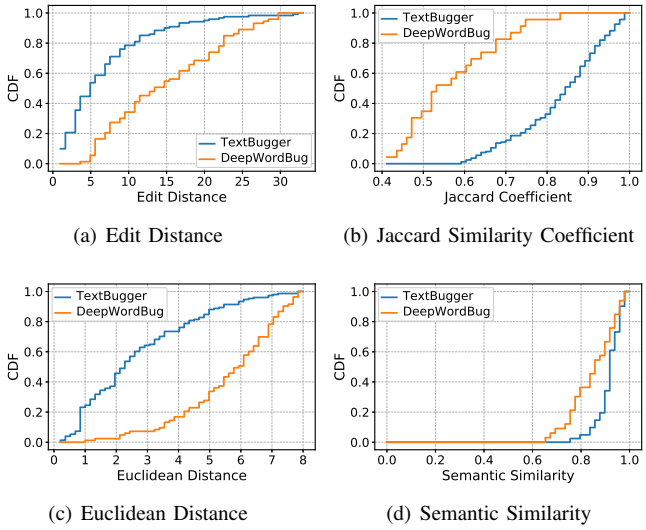
In the image domain, an important property of adversarial examples is the transferability, i.e., adversarial images



(a) Edit Distance (b) Jaccard Coefficient

(c) Euclidean Distance (d) Semantic Similarity

Fig. 13. The utility of adversarial texts generated on the Kaggle dataset under white-box settings for LR, CNN and LSTM models.



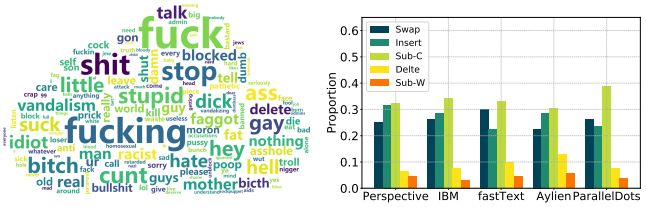
(a) Edit Distance (b) Jaccard Similarity Coefficient

(c) Euclidean Distance (d) Semantic Similarity

Fig. 14. The average utility of adversarial texts generated on Kaggle dataset under black-box settings for 5 platforms.

generated for one classifier are likely to be misclassified by other classifiers. This property can be used to transform black-box attacks to white-box attacks as demonstrated in [28]. Therefore, we wonder whether adversarial texts also have this property.

In this evaluation, we generated adversarial texts on all three datasets for LR, CNN, and LSTM models. Then, we evaluated the attack success rate of the generated adversarial texts against other models/platforms. The experimental results are shown in Tables VII and VIII. From Table VII, we can see that there is a moderate degree of transferability among models. For instance, the adversarial texts generated on the MR dataset targeting the LR model have 39.5% success rate when attacking the Azure platform. This demonstrates that the adversarial texts generated by TEXTBUGGER can successfully transfer across multiple models. From Table VIII,



(a) Word Cloud (b) Bug Distribution

Fig. 15. (a) The word cloud is generated from Kaggle dataset against the CNN model. (b) The bug distribution of the adversarial texts is generated from Kaggle dataset against the online platforms.

TABLE VII. TRANSFERABILITY ON IMDB AND MR DATASETS.

Dataset	Model	White-box Models			Black-box APIs				
		LR	CNN	LSTM	Watson	Azure	Google	fastText	AWS
IMDB	LR	95.2%	20.3%	14.5%	14.5%	24.8%	15.1%	18.8%	19.0%
	CNN	28.9%	90.5%	21.2%	21.2%	31.4%	20.4%	25.3%	20.0%
	LSTM	28.8%	23.8%	86.6%	27.3%	26.7%	27.4%	23.1%	25.1%
MR	LR	92.7%	18.3%	28.7%	22.4%	39.5%	31.3%	19.8%	29.8%
	CNN	26.5%	82.1%	31.1%	25.3%	28.2%	21.0%	19.1%	20.5%
	LSTM	21.4%	24.6%	88.2%	21.9%	17.7%	22.5%	16.5%	18.7%

TABLE VIII. TRANSFERABILITY ON KAGGLE DATASET.

Model	White-box Models			Black-box APIs				
	LR	CNN	LSTM	Perspective	IBM	fastText	Aylien	ParallelDots
LR	92.3%	28.6%	32.3%	38.1%	32.2%	29.0%	49.7%	54.3%
CNN	23.7%	82.5%	35.6%	26.4%	27.1%	25.7%	52.6%	50.8%
LSTM	21.5%	26.9%	94.8%	23.1%	26.5%	25.9%	31.4%	28.1%

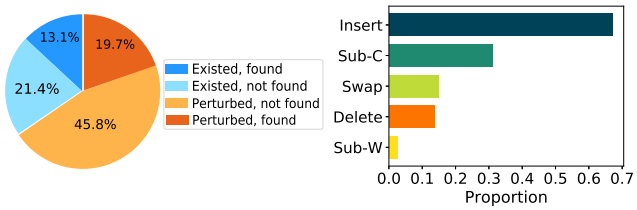
we can see that the adversarial texts generated on the Kaggle dataset also has good transferability on Aylien and ParallelDots toxic content detection platforms. For instance, the adversarial texts against the LR model has 54.3% attack success rate on the ParallelDots platform. This means attackers can use transferability to attack online platforms even they have call limits.

B. User study

We perform a user study with human participants on Amazon Mechanical Turk (MTurk) to see whether the applied perturbation will change the human perception of the text's sentiment. Before the study, we consulted with the IRB office and this study was approved and we did not collect any other information of participants except for necessary result data.

First, we randomly sampled 500 legitimate samples and 500 adversarial samples from IMDB and Kaggle datasets, respectively. Among them, half were generated under white-box settings and half were generated under black-box setting. All the selected adversarial samples successfully fooled the targeted classifiers. Then, we presented these samples to the participants and asked them to label the sentiment/toxicity of these samples, i.e., the text is positive/non-toxic or negative/toxic. Meanwhile, we also asked them to mark the suspicious words or inappropriate expression in the samples. To avoid labeling bias, we allow each user to annotate at most 20 reviews and collect 3 annotations from different users for each sample. Finally, 3,177 valid annotations from 297 AMT workers were obtained in total.

After examining the results, we find that 95.5% legitimate



(a) (b)

Fig. 16. The detailed results of user study. (a) The distribution of all mistakes in the samples, including originally existed errors and manully perturbed bugs. (b) The proportion of found bugs accounting for each kind of bug added in the samples. For instance, if there are totally 10 Sub-C perturbations in the samples and we only find 3 of them, the ratio is 3/10=0.3.

TABLE IX. RESULTS OF SC ON IMDB AND MR DATASETS.

Dataset	Method	Attack Success Rate				
		Google	Watson	Azure	AWS	fastText
IMDB	TEXTBUGGER	22.2%	27.1%	32.2%	20.8%	21.1%
	DeepWordBug	15.9%	12.2%	15.9%	9.8%	13.6%
MR	TEXTBUGGER	38.2%	36.3%	30.8%	31.1%	28.6%
	DeepWordBug	26.9%	17.7%	13.8%	22.1%	10.2%

samples can be correctly classified and 94.9% adversarial samples can be classified as their original labels. Furthermore, we observe that for both legitimate and adversarial samples, almost all the incorrect classifications are made on several specific samples that have some ambiguous expressions. This indicates that TEXTBUGGER did not affect the human judgment on the polarity of the text, i.e., the utility is preserved in the adversarial samples from human perspective, which shows that the generated adversarial texts are of high quality.

Some detailed results are shown in Fig. 16. From Fig. 16(a), we can see that in our randomly selected samples, the originally existed errors (including spelling mistakes, grammatical errors, etc.) account for 34.5% of all errors, and the bugs we added account for 65.5% of all errors. Among them, 38.0% (13.1%/34.5%) of existed errors and 30.1% (19.7%/65.5%) of the added bugs are successfully found by participants, which implies that our perturbation is inconspicuous. From Fig. 16(b), we can see that insert is the easiest bug to find, followed by Sub-C. Specifically, the found Sub-C perturbations are almost the substitution of "o" to "0", and the substitution of "l" to "1" is seldom found. In addition, the Sub-W perturbation is the hardest to find.

VI. POTENTIAL DEFENSES

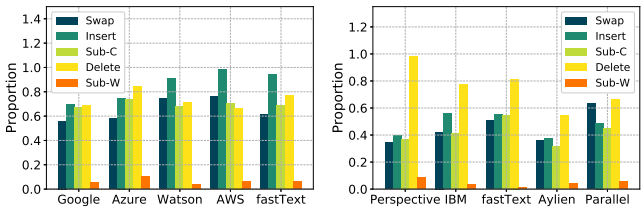
To the best of our knowledge, there are few defense methods for the adversarial text attack. Therefore, we conduct a preliminary exploration of two potential defense schemes, i.e., spelling check and adversarial training. Specifically, we evaluate the spelling check under the black-box setting and evaluate the adversarial training under the white-box setting. By default, we use the same implementation settings as that in Section IV.

Spelling Check (SC). In this experiment, we use a context-aware spelling check service provided by Microsoft Azure[11].

[11]https://azure.microsoft.com/zh-cn/services/cognitive-services/spell-check/

TABLE X. RESULTS OF SC ON KAGGLE DATASET.

Method	Attack Success Rate				
	Perspective	IBM	fastText	ParallelDots	Aylien
TEXTBUGGER	35.6%	14.8%	29.0%	40.3%	42.7%
DeepWordBug	16.5%	4.3%	13.9%	35.1%	30.4%



(a) IMDB (b) Kaggle

Fig. 17. The ratio of the bugs corrected by spelling check to the total bugs generated on IMDB and Kaggle datasets.

Experimental results are shown in Tables IX and X, from which we can see that though many generated adversarial texts can be detected by spell checking, TEXTBUGGER still have higher success rate than DeepWordBug on multiple online platforms after correcting the misspelled words. For instance, when targeting on Perspective API, TEXTBUGGER has 35.6% success rate while DeepWordBug only has 16.5% after spelling check. This means TEXTBUGGER is still effective and stronger than DeepWordBug.

Further, we analyze the difficulty of correcting each kind of bug. Specifically, we wonder which kind of bugs is the easiest to correct and which kind of bugs is the hardest to correct. We count the number of corrected bugs of each kind and show the results in Fig. 17. From Fig. 17, we can see that the easiest bug to correct is insert and delete for IMDB and Kaggle respectively. The hardest bug to correct is Sub-W, which has less than 10% successfully correction ratio. This phenomenon partly accounts for why TEXTBUGGER is stronger than DeepWordBug.

Adversarial Training (AT). Adversarial training means training the model with generated adversarial examples. For instance, in the context of toxic content detection systems, we need to include different modified versions of the toxic documents into the training data. This method can improve the robustness of machine learning models against adversarial examples [13].

In our experiment, we trained the targeted model with the combined dataset for 10 epochs, and the learning rate is set to be 0.0005. We show the performance of this scheme along with detailed settings in Table XI, where accuracy means the prediction accuracy of the new models on the legitimate samples, and success rate with adversarial training (SR with AT) denotes the percentage of the adversarial samples that are misclassified as wrong labels by the new models. From Table XI, we can see that the success rate of adversarial texts decreases while the models' performance on legitimate samples does not change too much with AT. Therefore, adversarial training might be effective in defending TEXTBUGGER.

However, a limitation of adversarial training is that it needs to know the details of the attack strategy and to have sufficient

TABLE XI. RESULTS OF AT ON THREE DATASETS.

Dataset	Model	# of Leg.	# of Adv.	Accuracy	SR with AT
IMDB	LR	25,000	2,000	83.5%	28.0%
	CNN	25,000	2,000	85.3%	15.7%
	LSTM	25,000	2,000	88.6%	11.6%
MR	LR	10,662	2,000	76.3%	23.6%
	CNN	10,662	2,000	80.1%	16.6%
	LSTM	10,662	2,000	78.5%	16.5%
Kaggle	LR	20,000	2,000	86.7%	27.6%
	CNN	20,000	2,000	91.1%	15.4%
	LSTM	20,000	2,000	92.3%	11.0%

adversarial texts for training. In practice, however, attackers usually do not make their approaches or adversarial texts public. Therefore, adversarial training is limited in defending unknown adversarial attacks.

Further Improvement of TEXTBUGGER. Though TEXTBUGGER can be partly defended by the above methods, attackers can take some strategies to improve the robustness of their attacks. For instance, attackers can increase the proportion of Sub-W as it is almost cannot be corrected by spelling check. In addition, attackers can adjust the proportion of different strategies among different platforms. For instance, attackers can increase the proportion of swap on the Kaggle dataset when targeting the Perspective and Aylien API, since less than 40% swap modifications have been corrected as shown in Fig. 17(b). Attackers can also keep their adversarial attack strategies private and change the parameters of the attack frequently to evade the AT defense.

VII. DISCUSSION

Extension to Targeted Attack. In this paper, we only perform untargeted attacks, i.e., changing the model's output. However, TEXTBUGGER can be easily adapted for targeted attacks (i.e., forcing the model to give a particular output) by modifying Eq.2 from computing the Jacobian matrix with respect to the ground truth label to computing the Jacobian matrix with respect to the targeted label.

Limitations and Future Work. Though our results demonstrate the existence of natural-language adversarial perturbations, our perturbations could be improved via a more sophisticated algorithm that takes advantage of language processing technologies, such as syntactic parsing, named entity recognition, and paraphrasing. Furthermore, the existing attack procedure of finding and modifying salient words can be extended to beam search and phrase-level modification, which is an interesting future work. Developing effective and robust defense schemes is also a promising future work.

VIII. RELATED WORK

A. Adversarial Attacks for Text

Gradient-based Methods. In one of the first attempts at tricking deep neural text classifiers [29], Papernot *et al.* proposed a white-box adversarial attack and applied it repetitively to modify an input text until the generated sequence is misclassified. While their attack was able to fool the classifier, their word-level changes significantly affect the original meaning. In [9], Ebrahimi *et al.* proposed a gradient-based optimization method that changes one token to another by using the gradients of the model with respect to the one-hot vector input. In [33], Samanta *et al.* used the embedding gradient to determine important words. Then, heuristic driven rules together with hand-crafted synonyms and typos were designed.

Out-of-Vocabulary Word. Some existing works generate adversarial examples for text by replacing a word with one legible but out-of-vocabulary word [4, 11, 14]. In [4], Belinkov *et al.* showed that character-level machine translation systems are overly sensitive to random character manipulations, such as keyboard typos. Similarly, Gao *et al.* proposed DeepWordBug [11], which applies character perturbations to generate adversarial texts against deep learning classifiers. However, this method is not computationally efficient and cannot be applied in practice. In [14], Hosseini *et al.* showed that simple modifications, such as adding spaces or dots between characters, can drastically change the toxicity score from Perspective API.

Replace with Semantically/Syntactically Similar Words. In [1], Alzantot *et al.* generated adversarial text against sentiment analysis models by leveraging a genetic algorithm and only replacing words with semantically similar ones. In [32], Ribeiro *et al.* replaced tokens by random words of the same POS tag with a probability proportional to the embedding similarity.

Other Methods. In [16], Jia *et al.* generated adversarial examples for evaluating reading comprehension systems by adding distracting sentences to the input document. However, their method requires manual intervention to polish the added sentences. In [40], Zhao *et al.* used Generative Adversarial Networks (GANs) to generate adversarial sequences for textual entailment and machine translation applications. However, this method requires neural text generation, which is limited to short texts.

B. Defense

To the best of our knowledge, existing defense methods for adversarial examples mainly focus on the image domain and have not been systematically studied in the text domain. For instance, the adversarial training, one of the famous defense methods for adversarial images, has been only used as a regularization technique in the DLTU task [18, 23]. These works only focused on improving the accuracy on clean examples, rather than defending textual adversarial examples.

C. Remarks

In summary, the following aspects distinguish TEXTBUGGER from existing adversarial attacks on DLTU systems. First, we use both character-level and word-level perturbations to generate adversarial texts, in contrast to previous works that use the projected gradient [29] or linguistic-driven steps [16]. Second, we demonstrate that our method has great efficiency while previous works seldom evaluate the efficiency of their methods [9, 11]. Finally, most if not all previous works only evaluate their method on self-implemented models [11, 12, 33], or just evaluate them on one or two public offline models [9, 16]. By contrast, we evaluate the generated adversarial examples on 15 popular real-world online DLTU systems, including Google Cloud NLP, IBM Watson, Amazon AWS, Microsoft Azure, Facebook fastText, etc. The results demonstrate that TEXTBUGGER is more general and robust.

IX. Conclusion

Overall, we study adversarial attacks against state-of-the-art sentiment analysis and toxic content detection models/platforms under both white-box and black-box settings. Extensive experimental results demonstrate that TextBugger is effective and efficient for generating targeted adversarial NLP. The transferability of such examples hint at potential vulnerabilities in many real applications, including text filtering systems (e.g., racism, pornography, terrorism, and riots), online recommendation systems, etc. Our findings also show the possibility of spelling check and adversarial training in defending against such attacks. Ensemble of linguistically-aware or structurally-aware based defense system can be further explored to improve robustness.

Acknowledgment

This work was partly supported by NSFC under No. 61772466, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Provincial Key Research and Development Program of Zhejiang, China under No. 2017C01055, and the Alibaba-ZJU Joint Research Institute of Frontier Technologies. Ting Wang is partially supported by the National Science Foundation under Grant No. 1566526 and 1718787. Bo Li is partially supported by the Defense Advanced Research Projects Agency (DARPA).

References

[1] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples," *arXiv preprint arXiv:1804.07998*, 2018.

[2] M. Barreno, B. Nelson, A. D. Joseph, and J. Tygar, "The security of machine learning," *Machine Learning*, vol. 81, no. 2, pp. 121–148, 2010.

[3] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, "Can machine learning be secure?" in *ASIACCS*. ACM, 2006, pp. 16–25.

[4] Y. Belinkov and Y. Bisk, "Synthetic and natural noise both break neural machine translation," *arXiv preprint arXiv:1711.02173*, 2017.

[5] B. Biggio, G. Fumera, and F. Roli, "Design of robust classifiers for adversarial environments," in *SMC*. IEEE, 2011, pp. 977–982.

[6] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *S&P*, 2017, pp. 39–57.

[7] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.

[8] M. Cheng, J. Yi, H. Zhang, P.-Y. Chen, and C.-J. Hsieh, "Seq2sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples," *arXiv preprint arXiv:1803.01128*, 2018.

[9] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "Hotflip: White-box adversarial examples for nlp," *arXiv preprint arXiv:1712.06751*, 2017.

[10] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song, "Robust physical-world attacks on machine learning models," *arXiv preprint arXiv:1707.08945*, 2017.

[11] 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.

[12] Z. Gong, W. Wang, B. Li, D. Song, and W.-S. Ku, "Adversarial texts with gradient methods," *arXiv preprint arXiv:1801.07175*, 2018.

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015, pp. 1–11.

[14] H. Hosseini, S. Kannan, B. Zhang, and R. Poovendran, "Deceiving google's perspective api built for detecting toxic comments," *arXiv preprint arXiv:1702.08138*, 2017.

[15] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar, "Adversarial machine learning," in *AISec*. ACM, 2011, pp. 43–58.

[16] R. Jia and P. Liang, "Adversarial examples for evaluating reading comprehension systems," in *EMNLP*, 2017, pp. 2021–2031.

[17] Y. Kim, "Convolutional neural networks for sentence classification," in *EMNLP*, 2014, pp. 1746–1751.

[18] Y. Li, T. Cohn, and T. Baldwin, "Learning robust representations of text," in *EMNLP*, 2016, pp. 1979–1985.

[19] B. Liang, H. Li, M. Su, P. Bian, X. Li, and W. Shi, "Deep text classification can be fooled," *arXiv preprint arXiv:1704.08006*, 2017.

[20] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang, "Deepsec: A uniform platform for security analysis of deep learning model," in *IEEE S&P, 2019*.

[21] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *ACL*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150.

[22] W. Medhat, A. Hassan, and H. Korashy, "Sentiment analysis algorithms and applications: A survey," *Ain Shams Engineering Journal*, vol. 5, no. 4, pp. 1093–1113, 2014.

[23] T. Miyato, A. M. Dai, and I. Goodfellow, "Adversarial training methods for semi-supervised text classification," *ICLR*, 2017.

[24] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *CVPR*, 2016, pp. 2574–2582.

[25] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *CVPR*. IEEE, 2015, pp. 427–436.

[26] C. Nobata, J. Tetreault, A. Thomas, Y. Mehdad, and Y. Chang, "Abusive language detection in online user content," in *WWW*. International World Wide Web Conferences Steering Committee, 2016, pp. 145–153.

[27] B. Pang and L. Lee, "Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales," in *ACL*. Association for Computational Linguistics, 2005, pp. 115–124.

[28] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Asia CCS*. ACM, 2017, pp. 506–519.

[29] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting adversarial input sequences for recurrent neural networks," in *MILCOM*. IEEE, 2016, pp. 49–54.

[30] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014, pp. 1532–1543.

[31] G. Rawlinson, "The significance of letter position in word recognition," *IEEE Aerospace and Electronic Systems Magazine*, vol. 22, no. 1, pp. 26–27, 2007.

[32] M. T. Ribeiro, S. Singh, and C. Guestrin, "Semantically equivalent adversarial rules for debugging nlp models," in *ACL*, 2018.

[33] S. Samanta and S. Mehta, "Towards crafting text adversarial samples," *arXiv preprint arXiv:1707.02812*, 2017.

[34] D. Sculley, G. Wachman, and C. E. Brodley, "Spam filtering using inexact string matching in explicit feature space with on-line linear classifiers." in *TREC*, 2006.

[35] C. E. Shannon, "Communication theory of secrecy systems," *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

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

[37] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," *arXiv preprint arXiv:1801.02610*, 2018.

[38] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *NIPS*. Neural information processing systems foundation, 2015, pp. 649–657.

[39] Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners guide to) convolutional neural networks for sentence classification," in *IJCNLP*, vol. 1, 2017, pp. 253–263.

[40] Z. Zhao, D. Dua, and S. Singh, "Generating natural adversarial examples," in *ICLR*, 2018.