Defending Hate Speech Detection against Visual Adversarial Attacks

Saar Tzour-Shaday, Advised by Yaniv Nemcovsky, and Prof. Avi Mendelson

Final Project 02360207 - Advanced Topics in Deep Learning
Department of Computer Science – Technion, Haifa, Israel

Abstract. This project addresses the vulnerability of hate speech detection systems to character-level adversarial attacks, particularly "leetspeak" substitutions that replace standard characters with visually similar symbols. While these obfuscation techniques can significantly degrade model performance, existing defenses often require expensive retraining or fail to generalize across attack variations. I propose a novel model-agnostic preprocessing defense that can be integrated with any pre-trained hate speech detection system without modification. My approach employs a ResNet-18 vision classifier trained on corrupted character images to reconstruct obfuscated words into their original form. The classifier, augmented with corruptions and adversarial training with sparse attacks, achieves 51% accuracy on character recognition. When evaluated on two target models, my deobfuscation procedure successfully restored approximately 50% of adversarially attacked samples and reduced performance degradation by half compared to undefended models. The method recovered about 30% of altered characters while maintaining the semantic meaning required for correct classification. Unlike previous approaches that rely on pixel similarity or require costly adversarial retraining of the entire model, my defense provides an efficient and generalizable solution that can be applied to existing systems with minimal overhead.
Warning: This report contains text that is offensive in nature.

1 Introduction

Background. Hate speech detection systems aim to identify harmful content that promotes violence, discrimination, or hostility against individuals or groups based on attributes like race, religion, gender, or sexual orientation. These systems often rely on deep learning based natural language processing (NLP) models, like BERT, RoBERTa, and GPT, fine-tuned on annotated datasets to classify hate speech.

Despite advancements, hate speech detection systems are susceptible to adversarial attacks, where malicious attackers subtly modify input text to evade detection of abusive content. Common attack strategies manipulate text in different levels: sentence-level (e.g. sarcasm, ambiguity), word-level (e.g. misspelling) and character-level, which is the focus of this project.

character-level adversarial attacks include insertion or deletion of characters, as well as homoglyph and unicode substitutions, which consist of replacing letters with visually akin characters.

Visual character attacks. Visual modifications to text are commonly used in social media to obfuscate offensive comments or as part of specific writing styles, such as "leetspeak." Leet (or "1337"), also known as eleet or leetspeak, or simply hacker speech, is a system of modified spellings commonly used on the Internet. It replaces letters with numbers or symbolds that look similar by their glyphs, via reflection or other resemblance (e.g. turning "E" into "3", or "A" into "4").

Leetspeak attacks work effectively against NLP systems without requiring access to model internals (black box approach). They are easier to implement than computer vision attacks because they simply substitute characters according to predefined rules, without needing complex optimization or gradient calculations. These attacks create words with unfamiliar characters that weren't in the training data, causing models to fragment words into unrecognized pieces and lose their meaning. While humans can visually recognize the modified text, computers process characters individually without visual reasoning capabilities.

These attacks have a key limitation: they lack clear metrics for human readability. Unlike computer vision attacks that use a defined boundary (epsilon) to ensure perturbations remain imperceptible, there's

no established measure for how many letter substitutions make text unreadable to humans. When text becomes illegible to people, the model's confusion becomes justified rather than a vulnerability. Although I attempted to evaluate the legibility of these attacks in this project, the necessary models were unavailable, so this assessment was not completed.

The paper. In this project, I suggest an extension for part of the work "Text Processing Like Humans Do: Visually Attacking and Shielding NLP Systems" by [3]. The authors suggested a visual perturber to generate the leet-speak attack, as well as a few methods for shielding.

They designed three embedding spaces:

1. Description-based character embedding space (DCES) - textual description taken from the Unicode 11.0.0 final names list. The set of nearest neighbors is determined by all characters whose description refer to the same letter in the same case.
2. Easy character embedding space (ECES) - applied only to English upper and lower cases, with a single nearest neighbor each, that forms the same letter but with a diacritic below or above it.
3. Image-based character embedding space (ICES) - uses visual similarity merics to identif nearest neighbors characters. Each character was encoded through 24x24 image representation of the character, stacked row by row to form a 576-dimensional embedding vector.

The attacker introduced in the paper is $VIPER(p, CES)$ (VIsual PERturber), which probabilistically replaces characters with their nearest neighbors. Formally, given an input text, $VIPER$ flips each character with probability p to one of its top 20 visual neighbors in the given CES. The attacks caused performance drops up to 82% in tasks like part-of-speech tagging and grapheme-to-phoneme conversion, while humans retained near-perfect accuracy.

To address these vulnerabilities, the paper proposes three shielding methods: visual character embeddings, adversarial training, and rule-based recovery.

1. Training the model with ICES, the visual character embedding.
2. Applying adversarial training by including visually perturbed data during model training with a perturbation probability p.
3. Implementing rule-based recovery by replacing non-standard characters with their nearest standard neighbors (a-zA-Z) in ICES.

Limitations. The proposed attack as well as the shielding approaches are restricted, because the character embedding assumes pixel similarity, ignores different sizes, translations, rotations, flipping or stretching. Moreover, it is limited to one-to-one character mappings, when attackers often use character combinations. In addition, the adversarial examples they focused on are limited to obscure Unicode characters, when many real-world attacks employ common ASCII characters available on standard keyboards. Another pitfall is that adversarial training and training the model with ICES are too expensive because they require training from scratch. Moreover, they only applied a single type of perturbations (a diacritic below or above the letter), which does not generalize to any type of letter substituion. Generating a diverse set of adversarial examples for training can be computationally expensive and time-consuming. More importantly, models become overly specialized to adversarial examples, and eventually fall behind performances achieved on clean data.

My approach. To address these drawbacks, I introduce a model-agnostic deobfuscation preprocessing defense mechanism, to protect against those character-level leet-speak adversarial attacks. It can be attached to any pre-trained NLP model, with the goal of minimal loss of clean accuracy. My approach recieves the text input, detects adversarial words, and replaces them with the most likely word, aiming to transform it back to the original word.

2 Methods

The main contribution of the project is a model-agnostic visual defense mechanism against character-level adversarial attacks like leetspeak in hate speech detection systems. This mechanism serves as a preprocessing layer that detects and deobfuscates adversarial words before passing them to a pre-trained NLP model. My main goal was to effectively recover the original meaning of perturbed words while maintaining high accuracy on clean data. Specifically, I developed a character classification model based on ResNet-18, trained on grayscale images of characters (1-3 symbols) with various corruptions (e.g., scaling, rotation, translation). I implemented a leetspeak attack generator based on a custom leet character embedding space (LCES) and the VIPER framework (proposed in the paper), which perturbs input text by substituting characters with visually similar alternatives. I designed a preprocessing pipeline that detects suspicious words, converts them to images, classifies each segment, and reconstructs the deobfuscated word to restore the original meaning.

The defenses introduced by the paper rely on pixel similarity, one-to-one mappings, or costly adversarial training. My approach aims to generelize to wider set of substitutions, consisted of more than one character. It aims to be size-invariant, and is robust to many different variants and shapes of symbols that can resemble letters. It is efficient because it only requires training a small ResNet on a small dataset (10,000), instead of adversarially training a large language model with classifier head from scratch. In addition, I suggest an algorithm that only clean adversarial words, without changing already clean words, and can be added to any pre-trained hate speech detector, which means that the clean data accuracy remains the same.

3 Implementation and Experiments

3.1 Implementation

Attack. I crafted by hands a new CES, compsed from a mapping between English letters to characters of length up to 3: *leet character embedding space (LCES)*. I utilized the leet-speak cheet sheet[1], but filtered it a bit because some substitutions they suggested there seemd overly far from being similar to the original letter. I implemented $VIPER$ from scratch, that uses LCES and a given swapping probability p. This attacker replace each character in the sequence independetly with probability p, with some random candidate from the LCES map.

Deobfuscation pipeline. The process end to end described in Algorithm 1.

Splitting the text into words. The first step (Line 1) is breaking the input text into into smaller string segments, while preserving the original characters to allow reconstruction of the string after the deobfuscation. I simply split the text by spaces while keeping the spaces as separate elements, then further splits any word that ends with punctuation marks like . , : ! ?.

Identifying adversarial words. The second step (Line 5) is adversarial words detection. To avoid deobfuscating clean words in the sequence, I designed an algorithm to detect whether a given word is a normal English word or a suspicious, potentially adversarial example. The naïve approach was to use the target model's tokenizer, and check which word is tokenized into the unkown token. This was insufficient, since the odd characters inserted in the middle of the words make it fragmented into a sequence of known tokens. For example, "Bagel" is mapped into a single token, while "!3a9e1" is tokenized into 6 tokens, one for each character, because numbers and symbols have their own token ID. My second attempt was to use a layered approach to cover different edge cases. This part included two iterations, which I will describe now. The first approach utilizes trusted English dictionaries of words and names, provided by the NLTK library. First, it filters out punctuation and very short words, which are unlikely to be adversarial. Then, it checks for mixed-character patterns using a regular expression to flag words that combine English letters with unusual symbols, which often signal obfuscation. To support a wide range of valid English words, including those with suffixes like

[1] https://www.gamehouse.com/blog/leet-speak-cheat-sheet/

Algorithm 1 Deobfuscate Adversarial Example

```
function DeobfuscateAdversarialExample(adversarial_text, letter_classifier, M)
    clean_words ← [ ]
    split_words ← SplitTextToStrings(adversarial_text)
    for all word ∈ split_words do
        if IsValidWord(word) then
            clean_words.append(word)
        else
            all_splits ← SplitStringToSubstrings(word, M)
            substitutions ← [ ]
            scores ← [ ]
            max_substitution ← word
            max_score ← −∞
            for all split ∈ all_splits do
                options ← SubstituteSplit(split, letter_classifier)
                if options ≠ None then
                    for all (substitution, score) ∈ options do
                        if score > max_score then
                            max_substitution ← substitution
                            max_score ← score
                        end if
                        if not IsWordExists(substitution) then
                            continue
                        end if
                        substitutions.append(substitution)
                        scores.append(score)
                    end for
                end if
            end for
            if substitutions = [ ] then
                clean_words.append(max_substitution)
                continue
            end if
            likely_substitution ← substitutions[arg max(scores)]
            clean_words.append(likely_substitution)
        end if
    end for
    clean_text ← Join(clean_words)
    return clean_text
end function
```

"eats" or "walked," it uses the WordNet lemmatizer to reduce words to their base forms and looks them up in the NLTK English dictionary, as well as in list of human names (like "Jane") to ensure they are not falsely flagged as adversarial. Finally, I used a spell checker to filter common misspellings like "valdiation" for "validation" (as they are not leet adversarial). Experiments showed that this approach is too strict for real-world data, as many words were incorrectly flagged as adversarial simply because they were not found in the standard dictionary. This issue often arose with slang, curse words, or informal expressions that are common in online language but absent from curated word lists. To address this, I took a more relaxed approach by simplifying the perturbation mechanism to include only non-letter characters like symbols and numbers. The relaxed version first filters out single-character strings (often punctuation artifacts from text splitting) and strings that consist only of punctuation marks, and trims possessive endings like 's or s'. The core logic relies on a regular expression to detect suspicious tokens that mix letters with non-letter characters (e.g., "h@te" or "l0ve!"). If such a pattern is detected, the word is suspected as adversarial and further processed by the de-obfuscation engine. Otherwise, the word is considered valid and remains untouched. This lighter version improves speed and robustness by quickly filtering out obvious cases while minimizing false detections in informal or noisy text.

Splitting each string to all possible substrings up to length 3. The third step (Line 8) is splitting each adversarial word into substrings, based on the assumption that different groupings of characters may represent different valid interpretations. Given a string, I generates all possible ways to split it into substrings, where each substring has a maximum length defined by $M = 3$, which covers most of the leetspeak substitutions. This procedure uses a recursive backtracking approach to explore every valid combination of splits. The result is a list of lists, where each inner list contains one possible split configuration.

Computing the most likely original sequence. The forth step (Lines 9-35) is the core of my algorithm. Given a list of character chunks and a letter classifier, I generate all possible clean word reconstructions along with a probability score for each one. If a chunk contains non-letter characters (like numbers or symbols), I inference the letter classifier to replace it with likely English letters and track the probability of each guess. Chunks that are already plain letters kept as-is. The probabilities of substitutions are aggregated by summing the log probability to build possible words and their likelihood score. Higher score means that the model is more confident in the substitution. The candidates are filtered by checking against the NLTK dictionaries if they are valid words, or side effect mistakes done by the classifier. If no valid word is found, the substitution with the highest score is selected, even if it is not a real word. Finally, all the cleaned words are joined into a single string and returned.

In order to compute the most likelihood substitution for each word, I trained a *letter classifier* that gets an image of characters, and compute the probabilities over the ABC alphabet.

Letter classifier architecture. The model used for classifying images of one to three characters into a corresponding English letter is a custom implementation of the ResNet-18 architecture. I adopted a code from github[2] that defined a ResNet18 architecture for MNIST to implement the architecture in pytorch. This convolutional neural network is specifically adapted to work with grayscale images, such as those generated from leetspeak character crops. The network begins with a 7×7 convolutional layer followed by batch normalization, ReLU activation, and max pooling, which help reduce spatial dimensions while preserving important features. The core of the model consists of four residual blocks, each built from BasicBlock units that allow gradients to pass more easily during training by using skip connections. These blocks progressively increase the number of channels (from 64 up to 512) while reducing spatial resolution, allowing the model to capture increasingly abstract features. After passing through the convolutional backbone, the final feature maps are flattened and fed into a fully connected layer with a number of outputs equal to the number of target classes (26 English letters). The output is raw logits, which are later used to estimate the most likely original letter behind a potentially obfuscated character image by computing the softmax probabilities.

[2] https://github.com/rasbt/deeplearning-models/blob/master/

Training data creation Given a character, it is transformed to a 28×28 grayscale image of text, by applying the following:

- **Base Image**: Each image starts as a blank 28×28 black canvas. Font and Size: Text is rendered using different fonts (Inter, Questrial, SourceSerif4) with size adapted to the number of characters. Font size can be fixed or randomly scaled.
- **Positioning**: Text is placed in the center of the image by default
- **Transformations**: I applied several types of corruptions to the original text, to create diverse and challenging training samples. Scaling, translation and rotations are combined together, while flipping and stretching applied seperately to avoid over-challenging examples (that are no longer similar to the original character). More specifically, I applied the following:
  - **Scaling**: With 70% probability, the text is randomly scaled to different sizes within the image boundaries.
  - **Translation**: With 70% probability, the text is randomly translated vertically to simulate misalignment.
  - **Rotation**: With 70% probability, the text is randomly rotated between -10° and 10°.
  - **Flipping**: Optionally, text images can be horizontally flipped.
  - **Stretching**: Another option stretches the text horizontally to simulate distortion.
- **Normalization**: Final image arrays are scaled to $[0, 1]$ and returned as NumPy arrays.

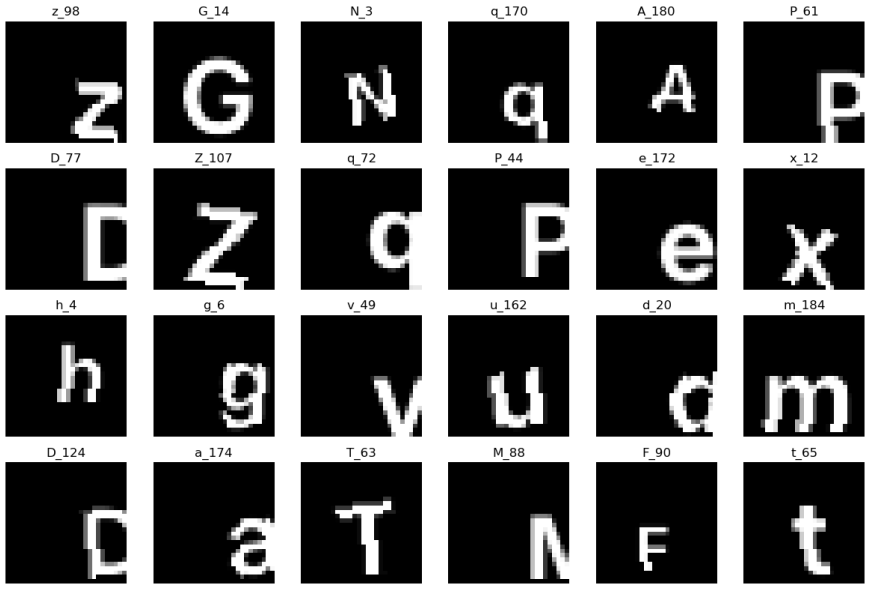

Fig. 1. Samples from the training set of the letter classifier.

Figure 1 demonstrate the different variants generated by applying the described corruptions to English letters images.

Training the classifier. I conducted many iterations until I found the best model. First, I started with a naïve version that is only trained on large ABC letters, without corruptions. Then, I added small letters as well, and then added gradually more and more corruptions and tested how the model performed. The dataset size can vary because of the randomness of the corruptions, that makes each sample uniuque with a high probability. I experimented different sizes of datasets, as well as hyperparameters (number of epoches,

batch size, learning rate). All training images are going through standard scaling by the mean and variance across all images, for better generalization. The model is trained with the Adam optimizer to obtain better stability. The experimentation results of old versions are omitted because this model is only one component of the proposed algorithm, which its main goal is to defend against the attack and not find the best leetspeak classifier, although the performance of this model has a crucial effect on the defence success.

Another important part of the training was label smoothing. In previous iterations, I noticed that the model demonstrates overfitting in terms of too high confidence in classification. For example, when I let it classify the character '0', it was 99% confident that this was the letter Q, although it may also be the letter O. One of the challenges of this task is how to make the model give high probabilites to more than 1 letter, when the symbol resemble a few letters (and not only a single one, such as '!' which is similar to both L and I). To mitigate this and address this challenge, I added a label smoothing to the ground truths - instead of a one-hot vector with the true letter, I used 0.9 probability for the correct one, and the 0.1 is divided by all other classes.

The training set of the mature versions of the model consisted of upper case and lower case letters images, combined with the random corruptions as described above. I initially decided not to include symbols in the training data, and only generate diverse curruptions of the original letters, to be able to evaluate it without leakage. I had only little examples of leet-speak substitutions for each letter, and I wanted to study how the model percieve visually similar characters, like humans are able to do. I am aware that this makes the training set and the test set come from slightly different distributions. However, this immitates better the process that we humen do: we learn to distinguish English letters from symbols, but when we see symbols in text, we understand how to read it because we notice the similarity, while acknowledging the difference.

During the project presentation, the course staff suggested to try and expose the model to symbols in addition to letters, and teach him to classify those as well, but in test time use only the logits of the ABC letters. This is reasonable because by training the model on both letters and symbols (like #, $, $\hat{}$), the model learns to extract a wider range of visual features, including the components that make up leet-speak characters. I experimented this approach (combined with the adversarial training described next), which resulted in 45% test accuracy (the former version was 48%), but showed an undesirable side effect: symbols like '4' and '$', which in the former version were classified correctly to A and S respectively, now presented 0.01 probability on average to a large set of letters. This makes sense, because the model was trained on these characters, and learned to give high probability to 4 and $, and small probabilites to the rest of the characters. Consequently, I decided to stay with the previous approach.

Another good idea that the course staff suggested was to integrate adversarial training with sparse attacks, such as L_0. Sparse adversarial attacks modify only a small number of pixels in the input image, which closely mimics how leet-speak works - replacing or modifying specific parts of characters to create alternatives that humans can still recognize but might fool algorithms. In addition, training the model with the L_0 attack effectively serves as a form of regularization, preventing overfitting to standard character forms and improving generalization to leet-speak variants not seen during training.

The L_0 norm counts the number of non-zero elements in a vector, which creates a discrete optimization problem that cannot be directly solved using gradient-based optimization methods. To cope, I chose to implement a "PGD L_0 attack" - a regular PGD without perturbation bound epsilon, then choosing only k maximal pixels as the final perturbation. This approach offers an elegant solution to the non-differentiability problem. It first runs standard PGD iterations to identify the gradient direction that maximizes the loss. However, instead of applying a uniform perturbation bounded by epsilon, it selectively perturbs only the k pixels with the largest gradient magnitudes. This maintains the sparsity constraint (L_0 norm = k) while still leveraging gradient information to find effective perturbations. Finally, I trained the ResNet18 with 10,504 training images with the corruptions mixture explained above, cross entropy loss with label smoothing, learning rate 0.001, batch size 32 and 20 epochs, while 50% of the batches during training were added the optimal L_0 delta perturbation. This model achieved the best result so far - 51% test accuracy.

3.2 Evaluation

Hate speech dataset. I evaluated my algorithm on the "Hate Speech Detection curated Dataset" from Kaggle, balanced dataset with 360,000 samples per label (0: non-hateful, 1: hateful), created by undersampling non-hateful data and augmenting hateful data using BERT embeddings and WordNet synonym augmentation. I focused only on samples that are tagged as hate speech, as they are my main target in this project, and filtered the dataset to sequences of up to 8 words for faster evaluation.

Target models. Unfortunately, the paper I extend only provided code for training the models they evaluated, but didn't provide those models to reproduce the results. Therefore, I used models from the internet to evaluate my method on:

1. Hate-speech-CNERG/dehatebert-mono-english[3] from Hugging Face, introduced by [1]. I refer to this model as **MONO**. This model is used detecting hate speech in English language. The mono in the name refers to the monolingual setting, where the model is trained using only English language data. It is finetuned on multilingual BERT model.
2. Hate-speech-CNERG/english-abusive-MuRIL[4] from Hugging Face, introduced by [2]. I refer to this model as **MURIL**. It has a BERT architecture and was finetuned on MuRIL (Multilingual Representations for Indian Languages) using English abusive speech dataset. The pretrained model it finetuned on was trained from scratch utilizing the Wikipedia, Common Crawl, PMINDIA, and Dakshina corpora for 17 Indian languages and their transliterated counterparts.

Evaluation metrics. To evaluate the suggested method, I computed the models' accuracy, i.e., the portion of samples which are tagged as toxic and the model classified as toxic, on three different forms during the process: on the original text, on the adversarial text (attacked by $VIPER$ with $p = 0.1$ and $p = 0.2$), and on the clean text after my deobfuscation process. I also computed the edit distance between these sequences, to measure two things: how close the adversarial example is to the original text, and how close the clean text is to the original one. This helps to understand to what extent my method was able to restore the original text.

4 Results and Discussion

4.1 Results

Figure 2 shows the primary results of my work. Both target models had rather low accuracy of 0.81 on the original text, probably because the dataset contains many unfamiliar words (mostly slang), which is a challenging setting. The MURIL model is more robust than MONO, which is observed in the higher accuracy on the adversarial inputs in both experiments. While my method was not able to completely restore the original text, it successfully reduced the performance loss caused by the attack by x2.

Table 1 gives additional statistics on the experiments. The "adversarial examples" row counts how many obfuscated samples actually fooled the model into changing their classification to valid text instead of toxic. As expected, when p is larger, more samples are able to fool the model. However, their legibility level decreases, and they might also fool a person. The "successful deobfuscations" row counts how many adversarial samples were successfully deobfuscated by my algorithm, i.e., after the process, the model classified them as toxic (when before the process, they were not identified as toxic). Approximately 50% of the samples were cleaned appropriately. The last two rows compare the average edit distance of the original and the adversarial text, against the average edit distance of the original against the clean. A perfect deobfuscation procedure would have gain 0 edit distance, meaning it was able to restore the text completely. My method is not perfect, but it indeed was able to restore 30% of the characters on average.

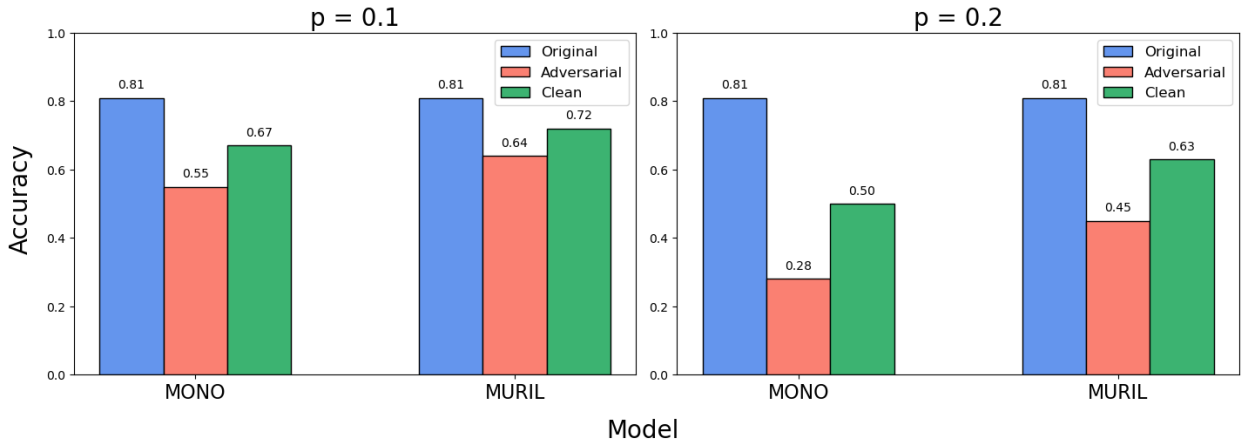

Fig. 2. Models accuracy on original, adversarial, and deobfuscated of first 100 text samples, over two experiments with different attack probability configuration.

Table 1. Experimental results of 100 first samples from the filtered dataset.

	MONO		MURIL	
	$p = 0.1$	$p = 0.2$	$p = 0.1$	$p = 0.2$
Adversarial examples	26	54	20	37
Successful deobfuscations	13	24	9	17
Avg. $EditDist(orig, adv)$	3.47	7.35	3.43	7.41
Avg. $EditDist(orig, clean)$	2.59	5.26	2.63	5.11

Test cases. To further understand the algorithm's limitations, I viewed some successful test cases, i.e., samples of text that their original version was classified as toxic, their adversarial version was missclassified, and the clean version was again classified as toxic. They are presented in Figure 3. These case demonstrate how sometimes not all of the words are restored correctly, but the partial correction is enough to make the target model identify the toxic. For example, in sample number 6, even though the word "unfollow" was replaced by pure gibberish, the rest of the words were cleaned perfectly. Samples 3 and 4 show that the deobfuscation is sensitive to the letter classifier mistakes, when in both examples, the symbol that should have been substituted by 'N' was replaced by 'H', because the classifier gave 'H' a larger probability score.

4.2 Discussion

Limitations of my method. First, the algorithm has exponential runtime with respect to the words length. The core of the method involves exploring all possible combinations of character substitutions to reverse leetspeak obfuscation. This results in an exponential increase in computational complexity as the text length increases. Since each ambiguous character can map to multiple possible letters (e.g., '!' to 'I', 'L'), the number of interpretations grows rapidly with text length. As a result, the method is only feasible for short words. For longer words, the computation becomes intractable or requires aggressive pruning heuristics, which may reduce accuracy.

Second, my approach ignores swears and slang, where in practice they should be highlighted more than other words. The algorithm focuses primarily on mapping leetspeak characters back to standard alphabetic

[3] https://huggingface.co/Hate-speech-CNERG/dehatebert-mono-english/
[4] https://huggingface.co/Hate-speech-CNERG/english-abusive-MuRIL

index	p	Original	Adversarial	Clean
1	0.1	you are retarded	yo|_| are ret4rded	you are retarded
2		it said words you fucking mongoloid	][t said words you fL|cking mongol()id	h said words you fucking mongoluid
3		you being a twat imagine	yoü bei/\/g a twa† imagi||e	m being a twat imagihe
4		truly a shithole country	truly a s]~[ithole cou||†ry	truly a shithole couhhry
5	0.2	blah blah ching chong i eat dogs	!3lah b|ah ching ¢hong i ea† dog$	blah dah ching mong i eat dogs
6		unfollow me if part of baby dyke twitter	ü||fo|löw me if pДrt of 13a!3y dŶke t'//itter	dhfdbw me if part of baby dyke twitter
7		so many abu dhabi and ching chong followers	so many a13u dhabi and Ĉ)-(ing chong foll()wers	so many abu dhabi and ching chong follwers
8		i like mongoloid better anyway	i lik& mongöloid be7"|"er anyway	i lik mongdoid bealer anyway

Fig. 3. Test cases of original text, adversarial example produced with the given p, and the clean text resulted from the deobfuscation pipeline.

forms using a classifier, and then checks the plausibility of the decoded text using dictionary-based or language-model-based filtering. However, it does not account for non-standard vocabulary like internet slang or profanities, which are often used in hate speech. As these terms may not appear in traditional dictionaries or may be heavily obfuscated, the system might fail to detect toxic messages that rely on such language.

Lastly, the method is sensitive to the letter classifier's mistakes. The effectiveness of the entire pipeline hinges on the accuracy of the character-level classifier, which is responsible for determining the probability distribution over possible letters for each obfuscated character. If this classifier misclassifies even a few characters, especially in short texts, it can lead to incorrect or nonsensical reconstructions. Since the method explores combinations based on the classifier's output, any errors early in the process can propagate, ultimately affecting the deobfuscation quality.

Conclusion. During this project, even though the phenomenon still occurs and critical, I noticed that there are little studies about deobfuscation of similar characters such as leetspeak from the recent 4-5 years. Modern large language models (LLMs) can now effectively clean leetspeak adversarial text without specialized tools. This capability likely explains the research gap, as general-purpose LLMs have become robust enough to handle these tasks, dedicated text classifiers are becoming less necessary. Nevertheless, this project provided valuable learning opportunities. I gained experience training robust image classifiers and combining them with classical computer science algorithms.

References

1. Aluru, S.S., Mathew, B., Saha, P., Mukherjee, A.: Deep learning models for multilingual hate speech detection. arXiv preprint arXiv:2004.06465 (2020)

2. Das, M., Banerjee, S., Mukherjee, A.: Data bootstrapping approaches to improve low resource abusive language detection for indic languages (2022), https://arxiv.org/abs/2204.12543
3. Eger, S., Şahin, G.G., Rücklé, A., Lee, J.U., Schulz, C., Mesgar, M., Swarnkar, K., Simpson, E., Gurevych, I.: Text processing like humans do: Visually attacking and shielding nlp systems (2020), https://arxiv.org/abs/1903.11508