Detection and Identification of Obfuscated Obscene Language with Character Level Transformers

Tobias Renwick^{†,*}, Denilson Barbosa[†]

[†] University of Alberta

Abstract

We present O-Norm (Offense Normalizer), a system for normalizing offensive words and phrases by reversing user-created obfuscations. O-Norm is intended to be a preprocessing tool for pipelines that run on user-created text, particularly those tasks where obscenities may be of significant value such as sentiment, toxicity, and abuse detection. O-Norm is constructed with a purely generated, context-free dataset derived from a curse dictionary. The generative dataset allows O-Norm to be flexible to the addition of new words as language adapts, and also allows it to be readily retrained in other Latin alphabet-based languages as it does not require manual annotations to train. O-Norm is based on a character-level, transformer network which attempts de-obfuscation only on out-of-vocabulary (*OOV*) tokens. In an 80/20 train-test split, O-Norm achieves an F1 score of 89.6% over 141 curses in a generated dataset with 2.16 million unique training points. An inspection of O-Norm’s output on a sample of social media posts from Kaggle’s Jigsaw corpus reveals accuracy of 95.7% on de-obfuscating transformations in toxic user-created text.

Keywords: offensive language, social media, character level, generated data, transformer

1. Introduction

Social media has become a popular domain for NLP research as it is a digitized and real-time stream of the thoughts and emotions of the world. To name just a few examples, attempts have been made to make market predictions [1], detect natural disasters [2], and monitor the corporate image of a company [3], all from social media. One of the challenges in creating corpora from social media is that the text is typically generated for a non-professional setting, by amateur writers. As a result, the text can have a crude vernacular, containing curses, racial slurs, and pejoratives, which are herein referred to as offensive language.

Historically, many media sources have suppressed the appearance of offensive language by removing or replacing the offending text with the help of dictionary-based tools. While such censorship is intended to improve the overall quality of the content, dictionary-based tools are easily defeated by clever writers who exploit the human ability to “read between the lines” and obfuscate offensive words while preserving their intended meaning, as illustrated in Table 1.

Obfuscating offensive words seems to have become the norm across many social media sites, with consequences to the NLP tools that process such content. A token is treated as *IV* (in-vocabulary) by an NLP algorithm if it expects the token with reasonable frequency, while obfuscated tokens are often ignored or treated as *OOV* (out-of-vocabulary), inevitably resulting in lower accuracy. For instance, Eger, Şahin, Rücklé, Lee, Schulz, Mesgar, Swarnkar, Simpson, and Gurevych [4] have shown that obfuscated offensive language can cause a decrease in accuracy for tokenization, part of speech tagging, and sentiment analysis and Mishra, Del Tredici, Yannakoudakis, and Shutova [5], Mishra, Yannakoudakis,

* renwick@ualberta.ca

Table 1. Transformations of an example ‘word’ which avoid censorship by a standard dictionary lookup

Obfuscated	Transform	O-Norm Data
w@d	09 Shift Substitution	yes
w0rd	Leet Speak	yes
w.o.r.d	Inappropriate Punctuation	yes
wo r d	Inappropriate Spacing	yes
wodr	Typoes	yes
wórd	Glyph Substitutions	yes
werd	Phonetic	no
	ASCII Art	no
	Zalgo text	no

and Shutova [6], and Nobata, Tetreault, Thomas, Mehdad, and Chang [7] have shown improvements in accuracy by accounting for character-level features in text, which is theorized to be due, at least in part, to obfuscations and misspellings.

1.1. Background

Fundamental to almost all NLP tasks is the notion of *tokenization*, which has the goal of reducing a piece of text to atomic “chunks”, called tokens, that can properly carry the semantics of the text. Human readers process text at many token levels simultaneously, from document level (knowledge about topics) to character level (plurality for example). Many current state-of-the-art methods for language understanding tasks are based on subword byte-pair encoding (BPE) methods [8, 9] which attempt to split words into common subword blocks with the hope that they can generalize to different forms.

Regardless of the division applied, the idea behind all tokenization is the same: to turn text into a series of tokens that algorithms can reason about. This creates a potential vulnerability to adversarial attacks [4] in that by their nature tokenizers are very sensitive to the actual characters and their order which is used to convey meaning, while humans are much more adaptive, fluidly replacing characters that seem out of place with a more appropriate version. At a point, this idea was incorporated into an online *meme* and given the name “typoglycemia” [10]. To illustrate this effect the chunk of text which follows was circulated, and while the *content of the quote is untrue*, the effect illustrates the depth of human ability:

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn’t mttar in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.[10]

However, no tokenizer today, no matter how large the dataset it was trained on, has a reasonable chance of deciphering this kind of text. Therefore, we believe the best defence is to normalize text before processing. We note that while the models and training techniques we present here could be used to address the example above, we put it to a more realistic use: de-obfuscating offensive language.

Table 2. Tokenization of obfuscated phrases with Spacy [11] and BPE encoding from the Transformers library [9] which results in different token representations for each variant of the same word

Tokenizer	Phrase	Token List
Spacy	curse	‘curse’
Spacy	c u r s e	‘c’ ‘u’ ‘r’ ‘s’ ‘e’
Spacy	c.u.r.s.e	‘c.u.r.s.e’
BPE	curse	‘c’ ‘urse’
BPE	c u r s e	‘c’ ‘Ġu’ ‘Ġr’ ‘Ġs’ ‘Ġe’
BPE	c.u.r.s.e	‘c’ ‘?’ ‘u’ ‘?’ ‘r’ ‘?’ ‘s’ ‘?’ ‘e’

Tokenization is not only vulnerable to spelling alterations, it is significantly affected by punctuation and spacing as these are the primary delimiters in Latin languages — the point at which a new token is supposed to begin. If a tokenizer makes mistakes and separates what a human would see as one token into multiple tokens, this causes a discontinuity in the interpretation and creates another feature that the network needs to learn: how to map these different token groups into the same result.

If the obfuscation in question is to add spacing or inappropriate punctuation between the tokens (e.g., ‘c u r s e’ or ‘c.u.r.s.e’), a human reader will typically understand that the spaces/punctuation need to be collapsed to understand the semantics, but this causes problems for tokenizers. As a concrete example, Table 2 shows example tokenizations by state-of-the-art methods where the word ‘curse’ is either treated as between one to nine distinct tokens. Note also that neither tokenizer reduces any of the inputs to the same token representation.

In a typical NLP processing pipeline, the token that is produced by the tokenizer needs to be resolved to a known vocabulary. The problem caused by tokenization mistakes is compounded here where the mistaken tokens will now likely be resolved to an *OOV* token, a special token that is used for all unknown words. Using these methods, obfuscations can hide the writer’s meaning from even a well-trained tokenizer paired with a large vocabulary as the space of possible obfuscations for a token is quite large. Consider the tokens ‘f!@#’, ‘f@!#’, and ‘f#!@’, which are different permutations of the same characters and represent the same word, i.e they should be mapped to the same token. The task for this obfuscation technique is to map the many different permutations of ‘f***’ to the same (offensive) de-obfuscated word. Failure to do so results in lost meaning as well as many unnecessary *OOV* tokens, which, in turn, increases the dimensionality and the sparsity of the embedding matrix, hampering generalization.

While all of the words and phrases targeted herein are typically deemed offensive, they do not all share the same meaning, and they may not even all be negative in the context in which they are used. For tasks where the primary goal is to detect sentiment [12], or toxic behaviour [13], stronger detection and identification of offensive language may be particularly useful. For example, consider the sentences: “That’s f***** awesome!” and “You’re f###ing awful!”. The normally offensive word, in these contexts, is synonymous with ‘really’ or ‘extremely’, which in itself is not negative, but should still carry weight towards the understanding of the expression.

1.2. Our Approach.

As a solution, herein we present O-Norm, an offensive word normalization system, designed to be used during preprocessing to reduce the amount of *OOV* tokens in user-generated text. O-Norm uses character-level information to clean up text, but with an emphasis on high precision so that it does not harm further processing steps.¹

2. Related Work

While we focus on detecting offensive language, de-obfuscating language has other important applications. Freschi, Seraghiti, and Bogliolo [14] discuss the importance of de-obfuscating text as it relates to detecting spam in e-mail. The authors use a set of manually created rules such as the transformation of the character ‘@’ to ‘a’, and pattern matching via the Damerau distance metric to normalize text. With the normalized text, the authors also examine phonetic translations for matches. Our work uses a small preprocessing step that is along similar lines but uses less aggressive rules as we wish to make fewer assumptions about the intent of the author. For example, we do not consider ‘@’ to be an ‘a’ as ‘@’ is often used in the *shift 09* obfuscation method as a random replacement for any character in a string. Furthermore, the ‘@’ symbol is often used in social media to indicate a reference to a user. Because of these relaxed assumptions, phonetic translations were more difficult to consider for O-Norm, as any non-alpha character has no real pronunciation.

Nobata, Tetreault, Thomas, Mehdad, and Chang [7] use an array of features from dictionaries to syntactic as input to a regression model to detect abuse (toxic language and hate-speech) online for enhanced moderation. The authors demonstrate that the most effective single feature is the character level n-gram, falling just 5% F-score behind all of the features combined.

Nobata, Tetreault, Thomas, Mehdad, and Chang [7] also attempt to illustrate the fluidity of abusive language, as they show the degradation of a model over time by training on early data and evaluating more recent data. The authors call this stationary time version the original model, and compare it to an accumulated model, which is trained on all data before the current slice of time, and evaluated at the data contained in each slice, essentially updating the data with time. The accumulated model outperforms the original model on the most recent time slice by approximately 5% F-score, however, the authors note that this is possibly due to the higher volume of training data.

Mishra, Yannakoudakis, and Shutova [6] use a character-level neural network to detect abuse, specifically highlighting the issue of *OOV* vocabulary as an issue to consider in the modelling. The authors present an array of 10 different neural models that operate from character to word level. Most related to our work is a character-based word composition model that attempts to generalize the meaning of unseen words from their context. The network performs this function through a bidirectional LSTM that outputs an embedding, with the hope that the character level embedding can generalize to unseen words.

Eger, Şahin, Rücklé, Lee, Schulz, Mesgar, Swarnkar, Simpson, and Gurevych [4] analyze the effectiveness of visual character embeddings at preventing “adversarial perturbations” in text created by their system, called Viper. The authors create the adversarial perturbations by examining the first 30,000 Unicode characters and associating visual similarity by image-embedding (ICES — image-based character embedding space), semantic similarity by the similarity of the Unicode description (DCES — description-based character embedding space) (e.g. `LATIN SMALL LETTER A` is considered similar to the letter ‘a’), and ECES (easy character embedding space) which is a manually assembled smaller set of perturbations which are visually simple to recognize. The authors demonstrate the effectiveness of

¹The library code for O-Norm is available at github.com/qubies/o_norm.

Viper at decreasing the performance of state-of-the-art NLP systems on standard benchmark tasks. The glyph transformation function in our work is based on glyphs suggested by Viper DCES. Eger, Şahin, Rücklé, Lee, Schulz, Mesgar, Swarnkar, Simpson, and Gurevych [4] concluded that a rule-based replacement using a method where a glyph char was replaced with its nearest neighbour, was the best approach for de-obfuscating these attacks. The downside to this method is that it only considers glyph substitutions, ignoring obfuscations based on punctuation and spacing, or typos. Therefore, their method isn't very practical for de-obfuscating general forms of obscenity.

The connection between character-level models and their ability to handle misspellings and obfuscations is reasonably well-known [5–7]. This general approach for the detection of offensive language emphasizes the importance of character-level n-grams in the identification of toxic or abusive speech as the information lost with word level embedding is seen as too significant. The works above are related to O-Norm but are not directly comparable as they measure their abilities on the end task (typically identification of hateful text). In O-Norm the offensive words are altered, and the final algorithm applied to the text to detect sentiment or hate can still use any level of tokenization. O-Norm's approach is fundamentally a separation of concerns around pre-processing and semantic understanding. To perform this function, our work combines the notion of the importance of character level abstractions noted in Mishra, Yannakoudakis, and Shutova [6] to train a neural network with the generative approach of Eger, Şahin, Rücklé, Lee, Schulz, Mesgar, Swarnkar, Simpson, and Gurevych [4] attempting to produce a network that specializes in recognizing the partial patterns present in obfuscated words.

3. Method

O-Norm's training data is built up from 141 different obscene words and phrases selected from the databases provided by Hatebase [15] and No Swearing [16]. Hatebase [15] is used as a lexicon that focuses on curses and hate speech, while NoSwearing.com [16] is used to augment Hatebase with more current slang offensive terms. Some of the phrases in these lexicons consist of an offensive word attached to a non-offensive one; e.g., 'ship head', which are removed since the token 'head' is *IV*. While O-norm cannot reasonably fully repair a curse like 'sh@@ he@@' to 'ship head' it will resolve to 'ship he@@', however, we hypothesize that obfuscation of the curses alone is the far more likely case.

O-Norm directly addresses 6 forms of obfuscation, shown in Table 1. These six were chosen as we concentrated primarily on dealing with obfuscations that are easy to perform on a standard 'QWERTY' keyboard. We expect that these selected transformations are far more common than higher effort transformations, such as Zalgo text or ASCII art.

While obfuscations based on phonetics are also relatively easy to accomplish on a standard keyboard, the relaxed tokenization assumptions made by O-Norm (allowing split graphemes) make true phonetic translation challenging. O-norm can detect phonetic obfuscations if they maintain enough structural similarity to the original phrasing, however in future work we would hope to examine the effectiveness of the addition of a phonetic dictionary to O-Norm along similar lines to that employed by Freschi, Seraghi, and Bogliolo [14].

3.1. Dataset

The O-Norm training dataset is composed of entirely synthetic data, which is generated from a provided list of curses. This methodology is flexible, as to train, O-Norm does not require (or receive benefit from) contextual examples of curses. It is possible that a contextually sensitive detector would work better in ambiguous circumstances (e.g. '****'), however, we believe that the difficulty of collecting and maintaining the data would outweigh the gains. Further, while context can help inform if an *OOV* token is a curse, it appears to

not usually be necessary in practice — there is often enough information in the obfuscation itself, where portions of the text are unmasked — s*** — and more common curses take precedence over less common ones.

Table 3. Training examples

Vocabulary Class	Example	Curse?
Single Obscenity	bu11sh1p	Positive
Strong Obscenity	f!@#	Positive
Word	t.a.p.e.s.t.r.y	Negative
Small Word	h@s	Negative
Number	(123.45)	Negative

O-Norm’s training data is built by first selecting input from one of the vocabulary classes in Table 3. Single obscenity examples contain a single offensive word selected at random from the 141 available options. Strong obscenity examples were then added to provide an oversampling of more common curses, encouraging the resolution of potential ambiguities to the more common example. Both of the obscenity classes contain solely positive examples — those that contain curses.

Negative examples are single *non-curse* tokens that are obfuscated, not including the 141 identified curses. The dictionary used for the word examples also doubles as the list of *IV* words. The intent of using plain word examples is to prevent O-Norm from classifying all symbols as identifiers of obscenity and to force it to look for token patterns in the text. Small word examples are all dictionary words less than 5 characters in length, and similar to the strong curses, they were added to prevent the network from de-obfuscating commonly seen small slang text — e.g ‘sass’ into ‘ass’ — they encourage the network to be more certain with smaller words. Number examples were added to help prevent the network from misclassifying numbers as leet speak obfuscations. In initial versions of the networks, numerical expressions such as 45% were transformed into curses.

These 5 classes of examples are not fixed, O-Norm can be adjusted to a specific purpose by providing lists of positive and negative examples that push it towards or away from particular classifications in a fashion similar to the addition of the small words and numerical examples. For example, to run on Twitter data a user might be tempted to add “@user” tags and “#hashtags” as negative examples, however for most data these are probably better filtered with a regular expression beforehand, as their structural requirements make them readily identifiable — ‘[@#]\w{3,}’ or similar.

Once a token is selected, it is passed to the obfuscation methods in Table 4, and one of them is randomly applied. Each transformation method uses a local probability for every possible transformation subset to determine if that subset should be transformed. For

Table 4. Obfuscation Functions in O-Norm

Input	Transformation	Transformed
word	Equal Spacing	w o r d
word	Random Casing	w O r D
word	Swap Letters	w r o d
word	Remove Letter	w o d
word	Leet Speak	w 0 d
word	Unequal Spacing	wo rd
word	Insert Punctuation	w!o!rd
word	0-9 Shift Substitution	w@#d
word	Viper	wórd

example, a **0-9 Shift Substitution** has a local probability to replace a given character with any of `!,@,#,\$,\\%,^,&*,(,.)`. The local probability is randomly assigned on each call to the function, thereby ensuring that the dataset contains examples that vary in the degree of obfuscation performed. Some functions, **0-9 Shift Substitution** for example, have a higher tendency to leave the first letter intact, as this form is more common in obfuscations — ‘`f&%^`’ is more readable than ‘`$u#$`’.

3.1.1. Negative Examples

Negative examples were needed to prevent over-fitting and labelling all symbols as curses. In our case, they are intended to be those with some inherent variability that may be similar to curses. In training, negative examples are obfuscated along with the curse words, using all the same methods, and the network is tasked with identifying all negative examples as “none”.

The negative examples must be present enough to make the classification of unforeseen tokens more challenging without making the network uncertain of its objective. An example of where overfitting to curses would cause difficulty is where poor punctuation present in the text, for example, words may be accidentally joined by a period (e.g., `word1.word2`) resulting in an *OOV* token, but not usually in a curse. Varying the number of negative examples effectively controls the recall of the final classifier, and the end-user can train models to suit their application in this sense. O-Norm provides 2 models in its release, one trained on 25% negative data, and a more conservative model trained on 50% negative data.

3.1.2. Fine-tuning de-obfuscation

The ability to tune a de-obfuscation tool or any other pre-processing tool is crucial in practical applications of this nature. For example, analysts interested in posts from a specific social media platform that caters to a specific community, with its peculiar jargon, would like to fine-tune their tools to best handle that community. By design, the approach for generating training data for O-Norm allows such fine-tuning. For example, if the analyst observes that some forms of obfuscation are overrepresented in the corpus, O-Norm can be re-trained with proportionally more examples of that class of obfuscations. Similarly, if the analyst observes different kinds of punctuation than those we trained O-Norm with, it would suffice to generate examples of that kind to the training data.

Like any other data pre-processing tool, O-Norm needs to be used with care. For some social media channels (Twitter in particular), the use of hashtags (`#`) and @mentions is ubiquitous. While O-Norm (as trained by us for the experiments reported in this paper) will not necessarily treat every word with these characters as obscene, there is the potential for false positives. Given the prevalence of such *OOV* words in tweets and the availability of high-quality meta-data with Twitter (such as a list of trending hashtags or of users) this pre-processing is unavoidable, but also fairly straightforward. In fact, at the time of writing there exist a large number of pre-processing libraries for popular programming languages meant precisely for that purpose [17–19].

O-Norm’s methods try to be extremely flexible so that it can be used in any different scenario and to perform with high precision so as not to alter the input text. The flexibility comes from the data generation, as no real text is needed to train an instance of O-Norm, all that is needed is a dictionary and a list of curses.

3.2. Networks

O-Norm is tested in two different network configurations, a bi-directional LSTM (Long-Short Term Memory) recurrent network, and a character level transformer network [20] based on the RoBERTa architecture [21].

For the LSTM, the input layer employs a one-hot character level embedding, which is passed into the LSTM layer. The LSTM layer has dropout enabled to encourage confident generalizations. The output of the input sequence is fed to a single dense layer with softmax activation across all of the possible curses with the added option to classify as not a curse.

For the RoBERTa based network, the architecture is identical to the original, however, the tokenizer is altered to character level, and the maximum token length is limited to 50 characters. As a general rule, this input length is quite small for transformers, as the original network has a maximum length of 512 tokens, however since O-Norm relies on what is essentially a word-level window, a much smaller length is required. This shorter length increases the speed of training compared to a length of 512 tokens.

3.3. Pre-Processing

O-Norm employs a regular expression normalization function which restricts input to characters producible on an English QWERTY keyboard and collapses multiple whitespaces to a single space character. Any unknown character is translated to a pipe '|', and URLs are deleted as they are *OOV*.

For some use-cases, it is worthwhile to add additional preprocessing steps. If the text contains many named entities it would be advisable to pre-process them before O-Norm (or add them to the vocabulary). Proper names have shown a disproportionately high false-positive rate in our validation set as they are often *OOV* and at times similar to curses. For example, in our validation set the name 'Koch(s/es)' was routinely de-obfuscated to 'kooch'. As a result, O-Norm has the option to add named entities detected with [Spacy](#) to the vocabulary at the time of processing (for our results we have enabled this option).

3.4. Tokenization

O-Norm uses two levels of tokenization — a word-level token to locate *OOV* tokens, and a character level tokenization which is fed into the networks. Upon ingesting a phrase, O-Norm first reduces it to word-level tokens by splitting on white space and removing book-end punctuation. For each word, O-Norm first considers whether the token is *IV* or *OOV*. If the token is *OOV* it will be passed to the character level network for analysis. Because the character level networks only see *OOV* tokens, O-Norm cannot transform a token that is *IV* into another. The *IV/OOV* mechanism also allows control of how O-Norm operates as the end-user can specify which dictionary to use to perform tokenization, and text-specific words can be white-listed by adding them to the vocabulary.

A consecutive sequence of neighbouring *OOV* tokens is considered a block and they are passed as a group to the networks so that they can be considered as either individual tokens or as singles. This technique is employed to handle scenarios where the user has split a single token in half ('sh ip' etc.). To handle scenarios where an offensive word is split using spacing (e.g. 's h i p') and contains *IV* tokens, O-Norm may consider an *IV* token ('i' in this case) to be *OOV* if it is of a specified short length (1 char in our tests) and it is next to 1 or more *OOV* tokens (not counting punctuation or white space). The tokens which are then passed on to O-Norm are lists of 1 or more *OOV* tokens, which are connected in the text without an intervening *IV* token of sufficient length.

3.5. Prediction

For each sequence of *OOV* tokens, they are passed both individually and as a group through O-Norm’s character level inference networks. O-Norm then returns the highest confidence offensive term and transformation suggestion, along with its span in the text. The threshold value where O-Norm will replace a given token or token-group is configurable to the application. The lower the value of the threshold, the more tokens will be de-obfuscated for our evaluation, we use a threshold of 0.80 for the LSTM model and 0.4 for the transformer model — the transformer model performs with higher precision, and detects more curses with a lower threshold than the LSTM model.

4. Evaluation

To evaluate the model’s performance on the generated data, an 80/20 train-test split was used, with 2,160,000 examples. A test train split method was chosen as we have a large amount of training data and can spare a large portion for testing. Because O-Norm uses randomly generated data, each example was preprocessed as per Section 3.3 before being added to the splits and was added only if it was unique to prevent leakage. In order to be thorough, we also investigated using a 5 fold cross-validation, however, there is minimal difference between the splits due to the amount of data, and the length of the inputs, therefore the data is not reported here.

The results of the test splits are outlined in Table 5, where the transformer (25% negative) model has a slight advantage in overall F1 (80.4% vs. 80.3% and 79.8%), though we believe that this is at the limit of what is possible with the training set, as it contains ambiguities (‘f@#’ could be a curse, or ‘farm’, or ‘fork’ etc.) — we believe that all networks have converged to a similar state. There is, however, one notable exception, in the micro F1 of Transformer (25%) which is the result of a decreased ability to predict the negative examples, it identifies the obfuscated curses with similar precision and greater recall.

Table 5. Test set evaluation on generated data

Model	Sample Count	F1 Macro	Prec Macro	Recall Macro	F1 Micro
LSTM	2,160,000	80.3%	86.0%	76.2%	89.6%
Transformer (50% negative)	2,160,000	79.8%	86.9%	76.2%	89.6%
Transformer (25% negative)	2,160,000	80.4%	86.7%	80.1%	80.4%

4.1. Corpus and Baselines

We evaluated O-Norm in de-obfuscating real, contextual, unseen text using comments from Kaggle’s Jigsaw dataset². As that corpus is not annotated for the task at hand, we ran the candidate methods over a randomized selection of 100,000 toxic posts and manually examined those with changes, or detected curses.

To provide a baseline for the results, we initially ran [profanity-filter](https://pypi.org/project/profanity-filter/)³, a python library that detects curses for the purpose of censorship. The project mainly positions itself as a curse filter, however, it attempts to detect obfuscated curses, as per its documentation, and it identifies the curse with the aid of a Spacy processing pipeline.

Profanity-filter’s method of detection is purely dictionary-based, and therefore only words that are included in its default vocabulary are seen in the text, however, it does add some common obfuscations to its dictionary. Over the 100,000 posts, profanity filter detected

²<https://www.kaggle.com/c/jigsaw-unintended-bias-in-toxicity-classification>

³<https://pypi.org/project/profanity-filter/>

10,496 clear-text curses which it recommended censoring. This number provides an idea of the density of cursing in the text, as there is approximately one unfiltered curse for every ten posts. While many of these are not necessarily curses (for example it censors the words ‘flipping’ and ‘gay’) this is an effect of the dictionary used, which is customizable, therefore these are not seen as errors.

At its default settings, profanity-filter version 1.3.3 detected 55 obfuscated curses in the provided text, however, these were all phonetic derivatives and perfect dictionary matches for the filter. With its extensive dictionary, profanity filter is a high precision tool, and like O-Norm unlikely to damage text, however, as you will see there are far more than 55 obfuscated curses in the text.

To add more flexible baselines, we also measure the Levenshtein distance, Jaro distance, and Jaro-Winkler distance [22–24] between all *OOV* words and the candidate list of curses used to bootstrap the training of O-Norm (which are well known and publicly available). Levenshtein distance is a well-known measure of edit distance that considers the number of transformations that need to be completed to change one string into another [22]. The Levenshtein distance has a tendency to over classify short strings, therefore for this work we have adapted the following normalization by length for the Levenshtein distance between two strings, a and b :

$$1.0 - \frac{\text{Levenshtein}(a, b)}{\max(|a|, |b|)} \quad (4.1)$$

Jaro distance is intended to measure similar characters in a string, accounting for transpositions [23], while the Jaro-Winkler distance adds the idea that the prefix of a word is less likely to be incorrect, therefore weighting prefixes more [24]. For the normalized Levenshtein distance, we use a threshold of 0.8 to indicate an obfuscation (2 characters incorrect for every 10), while for Jaro and Jaro-Winkler we use a threshold of 0.90.

4.2. Results and Analysis

The Jaro-Winkler distance identified the most ‘obfuscations’ of non-learned methods with 1708 positives within the threshold, however, only 727 of them were correct (Table 6). The Jaro and Levenshtein distances correctly identify the same number of curses — 509 — however, the Jaro distance has 41 more false positives than the Levenshtein edit distance. The data seem to indicate that substitutions (Levenshtein) are a somewhat reasonable method to handle curses substitutions than transpositions, as Levenshtein outperforms both Jaro and Jaro-Winkler distances. For our learned metrics, however, O-norm (transformer-based models) locates more than 3 times as many correct obfuscations as Levenshtein (1410 or 1830 vs. 509), with much greater accuracy (95.7% or 79.6% vs. 78.2%). A large reason for the difficulty faced by edit distance metrics is that short words with a few mistakes quickly get unreasonable scores, and most common curses are short.

The O-Norm transformer variants illustrate the case for the flexibility of O-Norm’s design. By providing the model with fewer negative examples in training (25% vs 50%), we demonstrate the increase in recall in the test corpus from 1474 identified obscenities to 2298, however, the 25% model is far more damaging to the text, as the overall accuracy drops from 95.7% to 79.6%. The 50% negative trained model makes only 64 mistakes on the Jigsaw corpus, most of which are either attributable to currency, bulleting, or proper names — currency and bullets (e.g (4), (5) etc.) cause confusion with the leet speak transformations, and some proper names are left untagged by the Spacy Named Entity Recognition.

We note that while the Jigsaw evaluation lacks a comprehensive metric for recall, at the time of writing we did not have the resources to allow this. If O-Norm’s recall is reasonable, the corpus contains approximately 20 obfuscations for every 1000 posts and this sparsity requires the annotation of a large percentage of the corpus.

Table 6. Jigsaw dataset comparison of detected obfuscations, and their accurate transformations

Detection Method	Identified	Correct	Accuracy
Profanity-filter	55	55	100.0%
Levenshtein	651	509	78.2%
Jaro	692	509	73.6%
Jaro-Winkler	1708	727	51.6%
O-Norm-LSTM	1340	1269	94.7%
O-Norm-Transformer (50% negative)	1474	1410	95.7%
O-Norm-Transformer (25% negative)	2298	1830	79.6%

5. Conclusion

O-Norm is an obscene word de-obfuscation tool based on a RoBERTa like transformer network, trained on a comprehensive synthetic dataset of obfuscated words and phrases to identify hidden obscenities. O-Norm has a low chance of harming downstream models as it only changes tokens that are already *OOV*, which should result in a positive impact on any given task unless the model is reliant on *OOV* tokens. We report experiments showing up to a 95.7% accuracy on a real-world and user-generated dataset, *unseen* during training, with obscene content that had appeared on social media platforms. As there are no previous attempts at our specific task, we compare O-Norm to reasonable baselines that are available to data scientists at the time of writing, showing the effectiveness of our tool.

O-Norm is a simple and customizable tool, which can easily be fine-tuned by the end-user to specific forms of obfuscation. It offers controls for false positives without retraining, namely that a user can directly adjust the threshold for classification, or add domain-specific terminology to the vocabulary, which will classify it as *IV* and therefore it will not be processed. If the user re-trains the system, it can adapt to changing vocabulary, different languages, and specialized use-cases as it requires no annotated data. This area of exploration offers potential pre-processing improvements for many common NLP tasks, particularly those based on social media text, or anywhere obfuscations are plentiful.

References

- [1] J. Bollen, H. Mao, and X. Zeng. “Twitter mood predicts the stock market”. In: *Journal of computational science* 2.1 (2011), pp. 1–8. URL: <https://arxiv.org/pdf/1010.3003&>.
- [2] F. Alam, F. Ofli, and M. Imran. “Crisismmd: Multimodal twitter datasets from natural disasters”. In: 2018. URL: <https://www.aaai.org/ocs/index.php/ICWSM/ICWSM18/paper/viewFile/17816/17038>.
- [3] C. W. Holsapple, S.-H. Hsiao, and R. Pakath. “Business social media analytics: Characterization and conceptual framework”. In: *Decision Support Systems* 110 (2018), pp. 32–45. DOI: <https://doi.org/10.1016/j.dss.2018.03.004>.
- [4] S. Eger, G. G. Şahin, A. Rücklé, J.-U. Lee, C. Schulz, M. Mesgar, K. Swarnkar, E. Simpson, and I. Gurevych. “Text processing like humans do: Visually attacking and shielding NLP systems”. In: *arXiv preprint arXiv:1903.11508* (2019).
- [5] P. Mishra, M. Del Tredici, H. Yannakoudakis, and E. Shutova. “Author profiling for abuse detection”. In: *Proceedings of the 27th international conference on computational linguistics*. 2018, pp. 1088–1098.
- [6] P. Mishra, H. Yannakoudakis, and E. Shutova. “Neural character-based composition models for abuse detection”. In: 2018.
- [7] C. Nobata, J. Tetreault, A. Thomas, Y. Mehdad, and Y. Chang. “Abusive language detection in online user content”. In: *Proceedings of the 25th international conference on world wide web*. 2016, pp. 145–153.

- [8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).
- [9] T. Wolf et al. *HuggingFace’s Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: [1910.03771](https://arxiv.org/abs/1910.03771) [cs.CL].
- [10] Wikipedia. *Typoglycemia*. July 2020. URL: <https://en.wikipedia.org/wiki/Typoglycemia>.
- [11] M. Honnibal and I. Montani. “spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing”. 2017.
- [12] R. Feldman. “Techniques and applications for sentiment analysis.” In: *Commun. ACM* 56.4 (2013), pp. 82–89. URL: <https://airtonbjunior.github.io/mestrado/seminars/presentations/2/Presentation2AirtonV1.pdf>.
- [13] M. Zampieri, S. Malmasi, P. Nakov, S. Rosenthal, N. Farra, and R. Kumar. “Semeval-2019 task 6: Identifying and categorizing offensive language in social media (offenseval)”. In: *arXiv preprint arXiv:1903.08983* (2019). URL: <https://arxiv.org/abs/1903.08983>.
- [14] V. Freschi, A. Seraghi, and A. Bogliolo. “Filtering obfuscated email spam by means of phonetic string matching”. In: *European Conference on Information Retrieval*. 2006, pp. 505–509.
- [15] Hatebase. *Hatebase: Online Database of Hate Speech*. 2019. URL: <http://www.hatebase.org>.
- [16] NoSwearing.com. *Noswearing.com - List of Swear Words, Bad Words, & Curse Words*. 2019. URL: <https://www.noswearing.com/dictionary>.
- [17] E. Loper and S. Bird. “NLTK: The Natural Language Toolkit”. In: *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*. Philadelphia: Association for Computational Linguistics. 2002.
- [18] S. Özcan. *tweet-preprocessor*. URL: <https://pypi.org/project/tweet-preprocessor/>.
- [19] C. Baziotis. *ekphrasis*. URL: <https://github.com/cbaziotis/ekphrasis>.
- [20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. “Attention is all you need”. In: 2017.
- [21] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).
- [22] W. W. Cohen, P. Ravikumar, S. E. Fienberg, et al. “A Comparison of String Distance Metrics for Name-Matching Tasks.” In: *IJWeb*. Vol. 2003. 2003, pp. 73–78.
- [23] M. A. Jaro. “Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida”. In: *Journal of the American Statistical Association* 84.406 (1989), pp. 414–420.
- [24] W. E. Winkler. “The state of record linkage and current research problems”. In: *Statistical Research Division, US Census Bureau*. Citeseer. 1999.