**FastText Model for Offensive Language Identification**

1. **Overview**

I initially trained the open-source FastText machine learning model using public training datasets called OLID and SOLID. OLID and SOLID are public datasets containing annotated tweet text data. I then evaluated the trained model on the SOLID test dataset. This trained model can be used to determine whether the text data contains offensive language, is targeted offensive language, and the target of the offensive language.

I also used LIME technique to increase the interpretability of the identification results by visually showing which words in the text led to the identification results. This visual demonstration will be integrated into the website of the Trusted Data Exchange platform.

This feature can help platform users identify whether there is offensive language in the text data they upload, what type of offensive language, and what suspicious offensive language words exist.

1. **Environment**

This project depends on the following software and packages:

* Python 3.7
* pandas, numpy, fasttext, re, nltk, sklearn, tqdm, typing, os, random, lime, tkinter

1. **Datasets**

This project uses OLID and SOLID for model training and testing. OLID (Offensive Language Identification Dataset) is a dataset to classify offensive language on social media. It contains 14,100 English tweets that have been manually labeled using a three-level taxonomy:

* Offensive Language Detection
* Categorization of Offensive Language
* Offensive Language Target Identification

Although OLID has been widely used, it's relatively small in size and might be biased toward offensive language as the data was collected using specific offensive keywords. Also, because of the hierarchical nature of its taxonomy, the number of instances decreases at more specific levels, making it difficult to train robust deep learning models.

SOLID (Semi-Supervised Offensive Language Identification Dataset) is a larger dataset, containing over 9 million English tweets. The creation process for SOLID was semi-supervised, using OLID as a seed dataset to avoid time-consuming manual annotation. SOLID, being larger and collected in a more principled manner, can significantly improve the performance of offensive language identification models.

1. **Model**

FastText is an extension to the word-based model that uses subword representations, making it particularly effective in handling the noisy structure of tweets. FastText is a subword model that has demonstrated robust performance across various tasks without extensive hyperparameter tuning. It utilizes a shallow neural model for text classification, similar to the continuous bag-of-words model. Unlike the continuous bag-of-words model that predicts a word based on its surrounding words, FastText predicts the target label based on the words in the sample.

FastText's ability to break words down into smaller chunks (subwords) allows it to better understand and represent words that may have been misspelled or written in a non-standard way, a common occurrence in social media language. This is particularly useful in the context of offensive language detection, where people might deliberately misspell words or use non-standard forms to evade detection.

One major reason to choose FastText is its fast training speed. For large-scale text data, FastText can complete training in a relatively short time. Additionally, FastText uses character-level n-grams features, enabling it to understand words not present in the language model, making it highly suitable for handling social media text full of innovative usages and typos.

However, FastText can't capture relationships between sequences of words (i.e., it doesn't take into account the order of words). For complex semantic texts, FastText might not provide deep and precise understanding.

1. **Training (train\_fasttext.py)**

In the SOLID paper, FastText was trained with bigrams and a learning rate of 0.01 for Levels A and B of the OLID/SOLID taxonomy. For Level C, it is trained with trigrams and a learning rate of 0.09. All tasks use a window size of five and a hierarchical softmax loss. In the implementation, I trained FastText with the default values of hyperparameters as they can achieve similar prediction performance.

The training sets from OLID and SOLID were combined (though it’s optional to use either one of them for training). This combined dataset was then randomly shuffled before being used for model training. Enhancements in the performance for Levels B and C were achieved by implementing an upsampling strategy for the classes that were underrepresented. For each class, I sampled 'K' instances where 'K' represents the count of instances belonging to the most frequently occurring class. This method helped to balance the representation of different classes in the data.

1. **Evaluation (test\_fasttext.py)**

I evaluate the performance of each model by comparing the predicted labels with the actual labels for the SOLID test set. The evaluation results for the FastText model at different tasks are as follows:

* Task A: The model demonstrates high performance with an accuracy of 0.92. It has a precision, recall, and F1-score of 0.99, 0.84, and 0.91 for the 'NOT' class, and 0.86, 0.99, and 0.92 for the 'OFF' class respectively.
* Task B: The model's performance drops compared to Task A, with an overall accuracy of 0.54. For the 'UNT' class, the precision, recall, and F1-score are 0.53, 0.92, and 0.68 respectively. For the 'TIN' class, these metrics are 0.62, 0.15, and 0.24, indicating a lower recall and F1-score.
* Task C: The model's accuracy is 0.60. For the 'IND' class, the precision, recall, and F1-score are 0.48, 0.61, and 0.54 respectively. For the 'GRP' class, these metrics are 0.85, 0.63, and 0.73, showing a relatively good performance. For the 'OTH' class, these metrics are 0.13, 0.30, and 0.19, which are significantly lower, indicating the model's poor performance on this class.

In summary, the FastText model performs well on Task A but demonstrates lower performance on Tasks B and C, particularly struggling with the 'OTH' class in Task C.

1. **Visualization (lime\_popup.py)**

This code uses the LIME (Local Interpretable Model-Agnostic Explanations) technique to explain predictions of a three-level hierarchical approach of machine learning models. These models identify and classify offensive language in text.

LIME provides a way to understand the decisions made by complex machine learning models, which are often seen as black boxes due to their internal complexity. LIME focuses on the predictions of a machine learning model at the local level, i.e., for a specific instance. It approximates the behavior of the original model around the vicinity of the instance using a simpler model (such as a linear model) that is easier to understand.

The LIME technique is applied to each model's prediction independently (only when applicable), providing an explanation for each model's output. LIME's interpretations are local and pertain only to individual predictions, not the entire machine learning model.

The main steps of LIME are as follows:

* Select the instance for which you want to explain the model's prediction.
* Perturb your dataset and get the new sample's predictions using the model.
* Weight the new samples according to their proximity to the instance of interest.
* Fit a simple model (like linear regression or decision tree) on the dataset with the sample weights.
* Interpret the simple model to draw conclusions.

The three-level hierarchical approach works as follows:

* Model A: Detects if the text is offensive or not.
* Model B: If Model A classifies the text as offensive, Model B categorizes the offensive language as either targeted or untargeted.
* Model C: If Model B classifies the offensive language as targeted, Model C identifies the target of the offensive language, classifying it as an individual, group, or others (e.g., organizations, events, and issues).

The code also includes a feature to let the user manually enter a text or randomly select a text from the SOLID dataset for prediction and explanation (Figures 1-3).

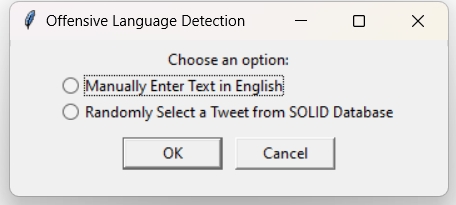


Figure 1. Pop-up window visualizing user options

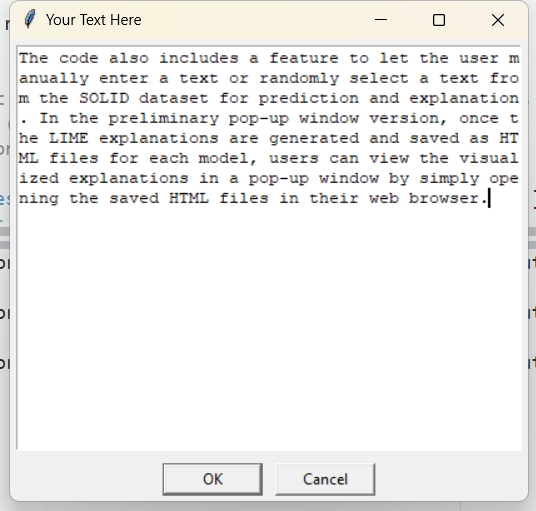


Figure 2. User option 1: manually enter text in English

A screenshot of a computer error

Description automatically generated

Figure 3. User option 2: randomly select a tweet from the SOLID database

In the preliminary pop-up window version, once the LIME explanations are generated and saved as HTML files for each model, users can view the visualized explanations in a pop-up window by simply opening the saved HTML files in their web browser (Figures 4-6).

A screenshot of a computer

Description automatically generatedFigure 4. Visualization of LIME explanations for Model A (classification: offensive or not offensive)

A screenshot of a computer

Description automatically generatedFigure 5. Visualization of LIME explanations for Model B (classification: targeted or untargeted, given offensive)

A screenshot of a computer

Description automatically generatedFigure 6. Visualization of LIME explanations for Model C (classification: individual, group, or others, given targeted)

**Useful Links**

<https://sites.google.com/site/offensevalsharedtask/home>

<https://huggingface.co/tharindu>

<https://huggingface.co/datasets/tharindu/SOLID/tree/refs%2Fconvert%2Fparquet/default/train>

<https://zenodo.org/records/3950379#.XxZ-aFVKipp>

<https://semeval.github.io/>

<https://www.kaggle.com/competitions/jigsaw-toxic-comment-classification-challenge/overview>

<https://www.kaggle.com/competitions/jigsaw-toxic-severity-rating/overview>

<https://www.kaggle.com/datasets/feyzazkefe/olid-dataset>

<https://lilianweng.github.io/posts/2021-03-21-lm-toxicity/>

<https://aclanthology.org/2023.acl-short.66/>

<https://arxiv.org/abs/1602.04938>

<https://arxiv.org/abs/1607.04606>

<https://github.com/marcotcr/lime>

<https://www.oreilly.com/content/introduction-to-local-interpretable-model-agnostic-explanations-lime/>

**README**

This README provides an overview of the directory structure and file descriptions for this project.

* Folders

(1) OLID Dataset: Contains the OLID dataset, which includes the training set, test set labels, and test set tweets.

(2) SOLID Dataset: Contains the SOLID dataset, comprising the training set, test set labels, and test set tweets.

(3) Miscellaneous: Contains additional files, including codes, model outputs, materials, and reference papers. Currently not in use.

* Codes

(1) train\_fasttext.py: A Python script that trains FastText models on OLID and/or SOLID training sets, intending to predict offensive languages at three levels.

(2) test\_fasttext.py: A Python script that tests FastText models on the SOLID test set, and generates classification reports of FastText models across three levels.

(3) lime\_popup.py: A Python script that uses LIME (Local Interpretable Model-Agnostic Explanations) to interpret and visualize the predictions of the model on individual instances.

(4) settings.py: A Python script containing the hyperparameters and other settings used in the other Python scripts.

* Files

(1) results\_fasttext\_dataX\_preprocessY.txt: Six files storing classification reports of FastText models across three levels. The data source and preprocessing status are indicated by 'dataX' and 'preprocessY', respectively.

data1 = OLID only

data2 = SOLID only

data3 = OLID + SOLID

preprocessTrue = with tweet text preprocessing

preprocessFalse = without tweet text preprocessing

(2) model\_fasttext\_a/b/c.bin: These files save the final FastText models.

(3) train\_data\_fasttext\_a/b/c.txt: These files store the processed training datasets before model training. FastText only accepts txt file inputs.

(4) lime\_notes.docx: This text file contains cautionary points on using LIME technique for interpretability and visualisation.

(5) documentation.docx: This Word document provides detailed information about the entire project.