MODULE 3
Natural Language Processing
DSI Team 2

Catherine Gitau, Fanamby Randriamahenintsoa,
Malcolm Wright and Martin Page

Message Screener

Check yourself before you wreck yourself: make your social media posts internet-proof

Table of Contents

Introduction:

Problem Statement:

The online world is just a click away. Anyone can have their thoughts and ideas published on social media in a matter of seconds. However, thoughtless and ill-worded posts can go viral, encourage a backlash, and have real-world implications for the sender.

Proposed Solution:

In many cases, simple feedback on a message might discourage users from publishing heedless content online. Firstly, providing feedback on a message adds friction by requiring the user to review a message before sending it. Secondly, feedback offers specific analyses of the sender's message that can flag potential controversial or insensitive content, providing a helpful reaction to the text. Our product analyses a message for: (1) profane language, which can be masked; (2) emotional tone (sentiment); and (3) the topic space, in line with Twitter's policy for content. The product is accessed through a graphical user interface and has built in database functionality.

Value of Solution:

The product is a self-help tool for social media users that helps them be more mindful of their posts and avoid unnecessary controversy by flagging words, tone and topics that might be considered sensitive.

Features of the Solution:

Feature 1: Profanity Screener

The profanity filter was built on the principle of avoiding the Scunthorpe problem [1], which is the unintentional flagging of innocuous words because they happen to contain a substring of letters that appears to have an obscene or unacceptable meaning (for example, the name Virginia contains the substring 'virgin', the word classic contains the substring 'ass', and the surname Cummings contains the substring 'cum' [2]). This strategy was applied because the profanity screener is designed to help the user check their message; it is not designed to act as a censorship tool. We therefore wanted the profanity screener to be flexible enough to catch commonly used formats of profane words and be useful as a checking tool, but at the same time reduce false positives to avoid annoying the user. The profanity screener thus searches for complete words (defined by word boundaries '\b') and not substrings. Manipulation strategies offer flexibility to find derived and novel profane words.

The profanity screener has the following features that define its search strategy:

- The message is cleaned to ensure that there are no double whitespaces ('\s') between words and the message is analysed in lower case.
- Sentence tokenization (from the **nltk** package) is used to retain the sentence divisions of the message. This ensures that sentence run-over, that is, if the last word of a sentence and the first word of the following sentence happen to form a two-word profanity, is not detected.
- Punctuation is stripped from the message. This ensures that random internal punctuation does not foil the detection of a profane term.
- As an exception to the punctuation stripping, the hyphen is manipulated in various ways in the user's message. The hyphen is (1) left natively in the message, (2) removed, with the

words concatenated into a single word, and (3) replaced by a space. This offers multiple representations of how a hyphenated word might appear in the blacklist database for a match. It also allows novel profane words to be detected if one part of the hyphenated term is profane.

- The words in the blacklist database that contain a hyphen were also permuted, as above, to widen the possibility of finding hits on compound terms that can be represented in different ways.
- Similar to the augmentation of hyphenated words, two-word compound terms might be written in the message with a space between them but appear in the database as a single word. Bigrams of the sentences in the message were generated and the space between the words were removed. The concatenated terms were screened against the blacklist database.
- Two-word compound term might be written in the message as a single word but appear in the database as two words. The database was also augmented by removing the space between two-word compound terms.
- The blacklist terms were formatted with regex that allowed common prefixes and suffixes to be placed before and/or after the lookup term (this includes leading and/or trailing digits), ensuring that derivatives of profane words could also be detected (standard lemmatisation in the **nltk** package did not work on swear words).

The possible terms detected from the above search strategy are scanned again against the original message to identify the format of the profane word *in situ*. That is, because of the manipulations, the word might be represented differently from the hit word that is detected, such as in the case of a novel compound swear word or the existence of leading/trailing digits that are anchored to profane terms. Alternatively, multiple hits could exist for the same profane word if the permutations of the message are also on the blacklist database.

- The detected terms are screened over the message (with hyphens but no other punctuation) to find them in their native form, not preceding nor proceeding a hyphen.
- The detected terms are screened over the message (with hyphens but no other punctuation) to find them preceding or proceeding a hyphen, and the entire (potentially novel) compound word is captured. This search uses the findall method to capture multiple novel words that use the same base profane word.
- Detected terms that are made up of multiple words are screened over the message (with hyphens but no other punctuation) with an indifference to internal hyphens and the word as it is formatted is captured. This search uses the findall method to capture multiple versions of the term.
- Leading and/or trailing digits are recognised as part of the profane term. Before the list of profane words is returned, terms with digits that do not appear natively in the blacklist are stripped of their numbers and returned in a clean format.

The output of the above search strategy and subsequent *in situ* validation is a list of hit words in lowercase that are in their original format in terms of hyphenation, but with other possible internal

punctuation and leading and/or trailing digits stripped from the text. The output can include novel hyphenated terms if one part of the term is on the profane list. **With the following foils**:

- Random internal numbers render the profane word undetectable.

The final captured terms can be masked in the message where the first and last character of the term remain visible and every other second letter character of the term is substituted by a random symbol. The original sentence case, spacing as well as punctuation (including internal punctuation) is retained. **With the following foils:**

- The missed detection of terms with internal numbers also means that these terms are not masked.
- Of course if a term is not in the blacklist, it cannot be detected even if it is clearly a swear word, such as two profane terms concatenated for make a new word, especially because the profanity screener does not search for substrings. We did try augment the terms in the blacklist database by concatenating every pair of words in the list, but this creates too many new words to scan, many of which are nonsensical.

Feature 2: Sentiment Classification

Sentiment Classification is a supervised machine learning method in NLP regarding understanding whether a sentence is either positive or negative in its tone. In our product, the sentiment analyser provides the user feedback on how their message might sound to others and allows the user to determine if they are coming across in the way they intended.

For this particular task we used **Flair,** which is a simple framework for state-of-the-art NLP. The library allows one to apply NLP models to text such as named entity recognition (NER), part-of-speech (PoS), sense disambiguation as well as classification producing better results than most text classification libraries available.

We first build a basic sentiment analyser, which can detect negative and positive sentiment. We used **Sentiment140** [3] for training our sentiment model. This is a dataset of 1.6 million tweets labeled as 0, negative sentiment or 4, positive sentiment. The tweets were then preprocessed by removing all unrecognised characters, padding out punctuations with spaces as well as removing links from text.

We saved only a fraction of the Sentiment140 data since it is quite large and it takes a lot of RAM when Flair loads it. We decided to train the model using only 0.05% of the whole dataset then split the data into train, test and validation sets. The model was run for 30 epochs and produced an accuracy of **0.76**.

We then built a more sophisticated sentiment classifier which is able to detect the emotions in the text. The emotions we trained our model on were: **sadness**, **anger**, **love**, **surprise**, **fear** and **joy.** We used the Emotions dataset for NLP [4] from Kaggle, the data were then passed through the same preprocessing methods that were used for the sentiment classifier. We again used Flair and the model was run for 20 epochs with final accuracy score of **0.8475**

Feature 3: Topic Identifier

Topic Identification is an NLP technique used to identify and discover topics across text documents. The purpose of the topic identifier feature is to flag topics, covered by a text message, that might be sensitive, according to Twitter's general rules and policies. If one of these topics is identified in the

user's text message, a warning message is then sent to notify them that the content of their message might be sensitive and advise them that they should be careful by giving a link to the Twitter community standards on the subject.

From Twitter's general rules and policies [5], we identified 6 topics that might be sensitive :

- Obscenity,
- Violence,
- Verbal abuse,
- Identity hate speech,
- Hate speech, and
- Offense.

We used the **Kaggle Toxic Comment Classification Challenge** [6] and **Hate Speech And Offensive Language** [7] datasets to train the model, which provided tweets labeled as **'toxic', 'severe_toxic', 'obscene', 'threat', 'insult', 'identity_hate', 'hateful', 'offensive', 'neither'.** We decided to use these data since they can easily be mapped to the sensitive topics that we identified from Twitter policies.

The text data has been processed through a tokenisation approach, a common task on NLP, which consists of separating a piece of text into smaller units called **tokens** that can be either words, characters or subwords. To do so, we used the **Tokenizer API** [8] already provided by keras.

The topic identifier is built up with keras sequential models. We used an **LSTM model**, also known as LSTM Network or Long Short Term Memory network, which is an improvement over RNNs.The LSTM model provided the most reproducible results with a score equal to **0.88**.

Feature 4: Database Functionality

In this challenge, we needed to build a database for the following reasons :

- the profanity screener feature needs to access a defined list of blacklist terms,
- the sentiment analysis feature needs to access a defined list of sentiments,
- the final message that the user submits needs be saved, and
- the test messages need to be saved.

In order to store these useful data, we set up an **SQLite database** managed through the **SQLiteStudio** tool, a free GUI tool for SQLite database management. We chose SQLite because of its easy integration with python through the **sqlite3** module.

Thus, we set up a database that is able:

- to contain and to access the list of blacklist terms,
- to contain and to access the list of sentiments,
- to save the final message that users finally submit,
- and to save the test text messages.

The database consists of 4 tables:

- the “sentiment” table which contains the list of sentiments types,
- the “blacklist” table which contains the list of blacklist terms,
- the “message” table which contains the final message that the user submits,
- and the “test_msg” table which contains the test messages.

The database is connected to all the features through a python script made up of a class called **“Database”**. All the features are linked to the database by importing and by instantiating this class. All queries and functions necessary to retrieve or to save data in the database were designed as functions of this class in order to ease the code reusability.

Each of the tables has been populated with initial data, for instance, the “blacklist” table has been populated with a predefined list of terms, the “sentiment” table has been populated with the 6 sentiment types used in the sentiment analysis feature, and the “test_msg” table has been populated with data from Sentiment140.

The database structure is set out below:

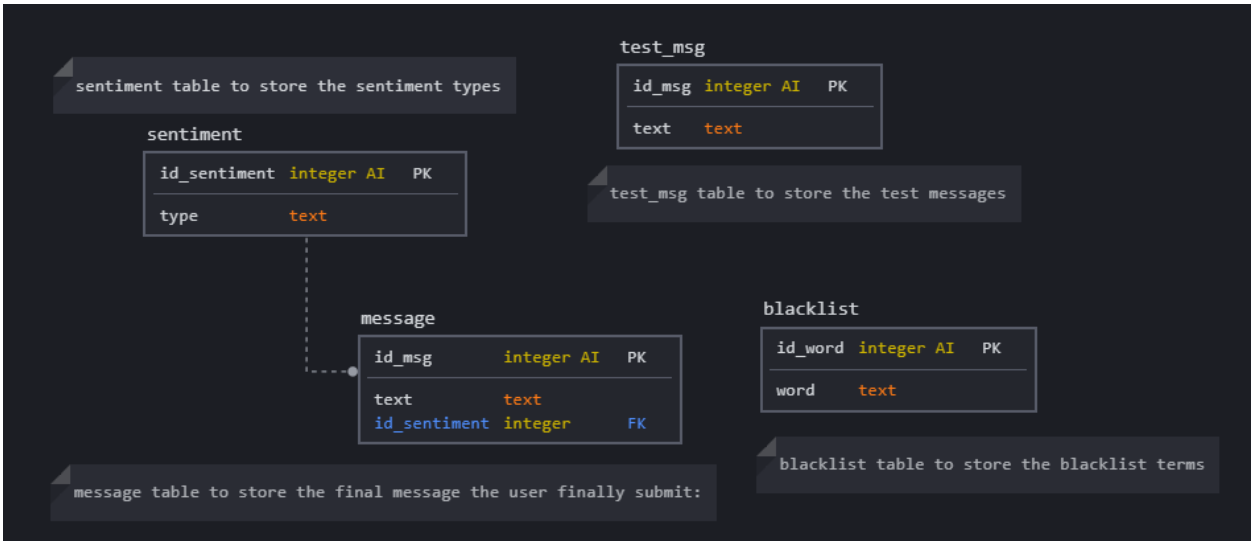

Figure 1: The database structure

Feature 5: Deployment of Graphical User Interface

The features highlighted earlier were deployed on a web-based application, which tied together all the features in one location and allows the user to access the product.

Framework Used

Streamlit was the framework used to develop the graphical user interface. This was selected for three reasons. First, the framework was an installable Python library with built-in functions. This meant that features could be incorporated into the interface efficiently without issues of language compatibility. Second, unlike other interface frameworks, Streamlit was intuitive to work with, only requiring a few lines to implement complex features. Finally, deployment could be done through the

Streamlit website, through a few simple steps. Allowing for a successful online deployment without too many restrictions.

Main Section

The main section of the web app consists of three main components common with the three main features:

- The title area – indicating which feature is currently in use.
- The text input area – which allows the user to input their tweet for analysis.
- The output text region – this displays the results of the particular analyser.

This main space allows for input of the text message as well as sufficient space for outputting the predictions.

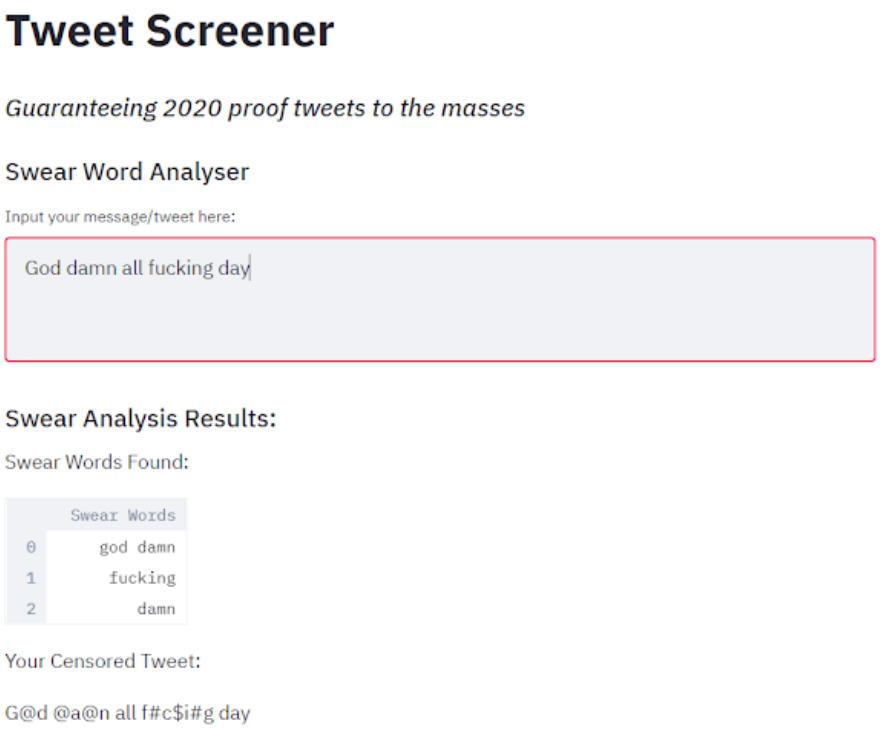

Figure 2: Final version of the GUI to illustrate the main section of the application

Sidebar

The sidebar of the application is more extensive with several operations occurring:

- Visual effects – a brief description of the application and the credits of the team alongside the course logo
- A feature selection area – radio buttons to select the feature to be used in the main area.
- A Publish Tweet Button – this would store the tweet and the emotional sentiment to the database.

The sidebar was utilised as a method to hide the menu and overall description options from the main analysing experience.

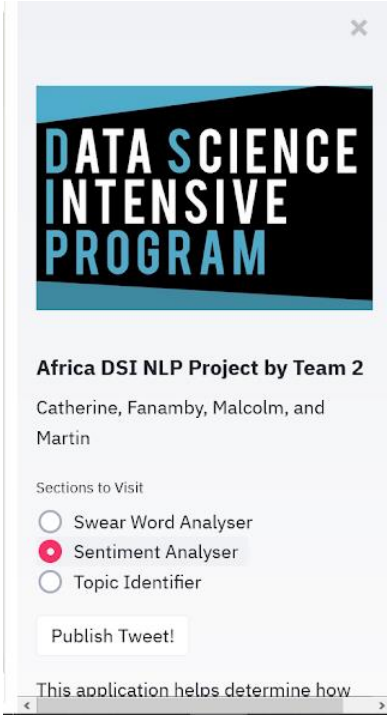

Figure 3: The sidebar of the app

Deployment

To deploy the application we utilised Strealit's in-built sharing functionality. To utilise this service, one of the team members had to be registered with the site. After registering, the app had to invite the member to share to the app, which occurred within the development time.

The application documentation needed to be stored on a public GitHub repository with a requirements.txt file. To deploy, the website would load the necessary packages and libraries as well as the models into their servers. Each app is currently allowed 800 MB of RAM, on a CPU, and 800 MB of dedicated memory. Once deployed the app is accessible to the general public.

Two of the three features deployed without an issue, however, the sentiment analyser was not able to load due to an unpickling issue related to the issue that Streamlit does not support Git large file storage.

Conclusion:

Successes:

- The project had a modular structure with each feature developed in parallel without much dependency on the other features, except for integration with the database and the deployment. This approach has several advantages:
    - Each team member could take ownership of a particular feature and work independently on their task, seeking collaboration where necessary. This meant that each team member could develop the feature to their own vision and the lack of strong dependencies also allowed for a more flexible schedule.
    - This structure also acts as a safeguard, spreading the risk of failure across the different features. If one of the features did not work, the other features would stand as backups to make a leaner product.

- In addition to the modular division of the project, each feature was divided into a hierarchy of tasks, with increasingly complex outputs. The basic version of the feature constituted a minimum viable product that could then be improved upon. This gave our team a working product early in the project's timeline and allowed us to be creative with the features without too much worry about the more complex version of the feature failing and leading to an issue.

- We use different NLP techniques in the project. The profanity screener is a rules-based approach to NLP that detects pattern and uses regex. The sentiment analyser and the topic identifier take a modelling approach to NLP, one using Flair and one using keras. In addition to the different modelling approaches, multiple datasets are used for the modelling. Overall this diversity creates a strong and featureful product.

Challenges:

Profanity Screener:

- Getting the regex to work was difficult, especially for the leading and trailing digits. The final solution to deal with digits is a bit messy but we could not find a regex expression to optionally match digits but not return them as part of the capture group (even non-capture groups actually form part of the final capture).

- We ran out of time to make the search work with internal digits. The approach we had in mind would be similar to dealing with internal punctuation marks except that some of the searches would need to be replicated to run on manipulations of the messages that have been stripped of digits.

- The ultimate challenge is the Scunthorpe problem, and indeed no good solution exists to the problem that harmless words can naively contain profane substrings and that some words are legitimate in certain contexts (e.g. Weiner, Schmuck, Libshitz are legitimate surnames). However, building the screener to avoid the Scunthorpe problem did constrain the flexibility of the algorithm, and the approach could be seen as building for the exception rather than building for the general case and then trying to deal with the false positives. We can find only the specific cases that we are looking for; however, this does match the purpose of the screener as a self-help tool and not a censorship device.

Sentiment Analyser:

- Getting the model to work in the deployment was a challenge, even though it worked locally.

Topic Identifier:

- Finding and getting the training data was a bit difficult because we had to look over many data to choose which can be easily mapped to the sensitive topic identified from Twitter's policies and guidelines.

- When building the model, we started with a simple NB-SVM (Naive-Bayes Support Vector Machine) model which had an accuracy of 98%, but over time we realised that its results were not reproducible on other data since its performance decreased to 65% and changed a lot as new data were added. Therefore, we decided to build a keras sequential model from scratch made of a BiLSTM model. This model's performance was quite good, around 98% but it was overfitting especially when the amount of the validation set was set up to more than 20% of

the training set. So, we had to figure out other ways to improve the model's performance. Finally, we ended up with a simple LSTM model and added more Dropout layers to avoid overfitting. This final model provided the most reproducible results with an accuracy of 88%.

Deployment:

- Working with GitHub was challenging, especially when needing to deploy with the Streamlit sharing feature.
- Integrating the code scripts as well as the trained models into the deployment (especially for deployment on the cloud) was difficult. The trained models from the sentiment analyser were very large in file size and we troubleshooted very late that GitHub LFS does not work with Streamlit.
- Overall, more time could have been dedicated to deploying the application, especially bug fixing, as a result of the unpickling issue.

Future directions:

Profanity Screener:

- One could make the screener more flexible by disregarding the Scunthorpe problem and searching for all substrings of the blacklist terms and then trying to work out the exceptions such as by using a whitelist and the context of the message.
- The screener could be improved by using models that analyse the context of the message and make a determination of the words in that context – some words are rude only in certain expressions/parts of speech.
- The screener could be made more robust if it were able to deal with spelling mistakes or other typing errors where the word is still obviously a swear word in the context of the message.
- An added feature of the profanity screener could be to suggest alternative phrases for the flagged terms.

Sentiment Analyser and Topic Identifier:

- The modelling components of the project could be improved with the use of more and diverse data as well as a feature of continuous improvement based on user feedback in the live system.

GUI:

- The code of each feature could be optimised so that the app can load and function more quickly.

Additional features:

- The current features could be bundled into an additional feature that analyses the history of a user's messages. This could offer insights into the degree of profanity used in messages, the general sentiment and the most common topics posted by the user for a certain period in the past.

- A further feature to develop is a fake news classifier or fact checker. The content of a user's message can either be analysed by a fake news classifier model or compared to live fact checking sources to verify the accuracy of claims made in the message.

Appendix

[1] Project GitHub repository: https://github.com/CateGitau/Message_Screener

[2] Project app deployment: https://share.streamlit.io/malcolmrite-dsi/message_screener-1/main/Message_GUI.py

References

[1] https://en.wikipedia.org/wiki/Scunthorpe_problem

[2] https://www.theguardian.com/politics/2020/may/27/anti-porn-filters-stop-dominic-cummings-trending-on-twitter

[3] https://www.kaggle.com/kazanova/sentiment140

[4] https://www.kaggle.com/praveengovi/emotions-dataset-for-nlp

[5] https://help.twitter.com/en/rules-and-policies/twitter-rules

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

[7] https://raw.githubusercontent.com/t-davidson/hate-speech-and-offensive-language/master/data/labeled_data.csv

[8] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer