**The NLP Project**

Comparison of different approaches



Author: Arapaut V. Sivaprasad.

Created Date: 02/01/201 Last modified on: 09/01/2021

**DISCLAIMER**

This is an evolving document that records and describes various methods proposed for the NLP project. It may contain errors, misunderstandings and omissions which will get corrected over time. This document was started when the author was still learning the NLP platform project and its objectives. Hence, it is possible that the methods described here may differ significantly from what are required. It is for the sole use of the ABS team involved in this project. No warranty of any kind is provided on the content and its accuracy.

Table of Contents

[PURPOSE OF THIS DOCUMENT 3](#_Toc60741205)

[TL;DR 3](#_Toc60741206)

[INTRODUCTION 3](#_Toc60741207)

[Profanity Analysis 4](#_Toc60741208)

[Python library: better\_profanity 4](#_Toc60741209)

[AWS Custom Entity Recognition 5](#_Toc60741210)

[Observations 6](#_Toc60741211)

[Offensive Language Classification 6](#_Toc60741212)

[Proposed way to use Cosine Similarity 7](#_Toc60741213)

[Sentiment Analysis 7](#_Toc60741214)

[Key Phrase Extraction 8](#_Toc60741215)

[Topics Modelling 8](#_Toc60741216)

[Spam classification – an exercise to develop profanity analysis 8](#_Toc60741217)

[Glossary 9](#_Toc60741218)

# PURPOSE OF THIS DOCUMENT

This document is a summary of the knowledge, findings and potential issues during the learning process of AWS Comprehend. Details about the use of other AWS services (e.g. S3, Athena, Glue, QuickSight, etc.) are described in separate documents and will not be repeated here.

# TL;DR

* The NLP project aims to:
  + Identify and eliminate profanity in text.
  + Identify the emerging issues.
  + Classify user feedback into a finite set of categories.
* Analyses to be implemented:
  + Profanity detection
  + Offensive language classification
  + Sentiment analysis
  + Key phrase extraction
  + Topic modelling
* The system works in the AWS cloud using:
  + S3
  + EC2
  + Comprehend
  + Sage Maker
  + Lambda
* Uses custom Python programs that work in response to triggers.
* EC2 may be used for development:
  + Platforms of increasing computing power available.
  + Parallel computing may be possible.
  + There are execution time limits.
* Profanity analysis:
  + Python package, ‘better\_profanity’
    - Uses exact string matching.
  + Own Python programs using Google’s bad words list.
    - Levenshtein Distance for spelling correction.
    - Vector matching to pick up words/phrases.
  + AWS custom entity recognition.
  + AWS custom classification.
  + Own Python program using Cosine Similarity.
  + Own Python program using reinforced machine learning.
    - Using RAKE to extract key phrases.

# INTRODUCTION

The NLP platform is to examine and cleanse the user feedback from the general population. Such feedback may contain bad language, profanity and other undesirable content. The expectation is that the programs and methods included in this project will automatically cleanse the data or discard the offensive feedback.

Another objective is to identify emerging issues. These are topics identified when there is an influx of feedbacks and/or queries on a particular subject.

Classifying user feedback, comments and queries into distinct classes is important for responding to them by the appropriate department. There should not be too many categories. Ideally, 7 to 20 categories must be aimed for. The system must create a new category if it cannot assign a text to an existing category.

The NLP platform will reside in AWS cloud and use various AWS services including S3, Lambda, EC2, Comprehend, Sage Maker, etc. In this document the focus is on how to use the AWS Comprehend for the analysis and how inhouse Python programs can complement it.

## Background

In the NLP Design document, the following analyses are described. These will be discussed in the sections that follow.

*NLP platform implements the following four analysis:*

*•* *Profanity Analysis - Identify profane words using Better Profanity Python library (NOTE: we originally were going to use Profanity Filter, which is a better library with smarter capability to identify permutation of profane words, but it can’t be used due to this library having a dependency which doesn’t pass vulnerability scanning and another dependency which requires Internet access)*

*• Offensive Language Classification - Identify sentences which contain offensive language using Amazon Comprehend Custom Classification*

*• Sentiment Analysis - Identify sentiment using Comprehend Sentiment Detection*

*• Key phrase Extraction - Identify key phrases using Comprehend Key Phrases Detection*

*• Topics Modelling - Identify topics using Comprehend Topic Modelling*

## Profanity Analysis

This is to detect words/phrases that are obscene in sentences and paragraphs. It is unclear from the design document whether the entire text containing profanity is deleted, or just the words/phrases masked. The following methods will be aimed at identifying the words/phrases within a sentence and, optionally, mask them without discarding the entire sentence.

As given in the design doc (see [above](#a1)) It may be simpler to use the Python library, “better\_profanity” (BP), for this. A quick description of this library, based on the [package description](https://pypi.org/project/better-profanity/), its usage and limitations are given below. Alternatives to using this library are given after that.

### Python library: better\_profanity

#### Usage:

$ pip install better\_profanity

from better\_profanity import profanity

#### Background:

This package uses string comparison instead of regex for matching. This makes it necessary to have all permutations and spelling variations of the offensive words in the list. They state that all modified spellings will be generated when a word in the list is loaded.

*All modified spellings of words in*[*profanity\_wordlist.txt*](https://pypi.org/project/better-profanity/better_profanity/profanity_wordlist.txt)*[[1]](#footnote-1) will be generated. For example, the word, handjob, would be loaded into:*

'handjob', 'handj\*b', 'handj0b', 'handj@b', 'h@ndjob', 'h@ndj\*b', 'h@ndj0b', 'h@ndj@b', 'h\*ndjob', 'h\*ndj\*b', 'h\*ndj0b', 'h\*ndj@b', 'h4ndjob', 'h4ndj\*b', 'h4ndj0b', 'h4ndj@b'

Provided that they have considered all common spelling changes, this method should work. However, if an extra character is added, or one deleted, (e.g. ‘handjobs’ or ‘hndjob’) the system may fail to detect it. I think this is an imperfect method. Using regex to handle it differently will be extremely complex.

Using the **Levenshtein Distance** (**“LD”**) (or “Edit Distance”)methodto correct the spelling to a base word is more accurate. I have created a Python program that does it by using a base list of words. When looking for profanity, this base will be a list of bad words. The software will do an LD match on all words/phrases in the list and pick the right one. So far, tested it using a list of pharmacy brands and got 100% accuracy on a small set of misspelled brands. More tests are required to confirm this. Character additions/deletions as well as spelling mistakes are handled.

Apparently, BP uses a list of just 140 words. It appears to be too few. Google uses a [451-word](https://github.com/RobertJGabriel/Google-profanity-words/blob/master/list.txt) list to ban swear words and bad words. Using this list together with LD will probably give much better results. These methods, LD and BP, could be compared first.

BP can handle Unicode characters which are currently not handled by LD. It will not become an issue if we are handling only English text. Some users may deliberately use Unicode characters to circumvent the testing.

BP can handle it even if the bad words are not separated with space (i.e. situations like ‘good\_bad’ or ---bad\*\*\*, etc.). Apparently, BP is using a substring method to identify the words, which explains the need to have all permutations in the word list. Currently, LD needs the words to be separated by spaces. This requirement may possibly be changed without losing the accuracy.

We must consider the context before calling a word as bad. For instance, the word “sex” could be used in a legitimate context or as a profanity. AWS has a custom Classifier method which can create one or more classes for bad language text. We could say that if the incoming text falls into one of these the word is profanity, otherwise legitimate. Also, a vulgar text is unlikely to contain just one bad word. Perhaps there should be two lists, one for the really bad words that are bad whenever used and another for the dual-purpose words. Presence of a word in the second list must be further clarified by either using the classification or checking whether it goes with a word from the first list.

### AWS Custom Entity Recognition

Using a list of bad words/phrases, and a list of text lines, we can train the model to identify the bad words in a sentence.

The major steps in analysing a text file are as follows:

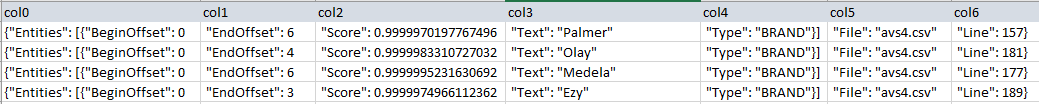
* We will be using ‘AWS Custom Entity Recognition’ under ‘[Analysis Jobs](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-analysis-job) | Analysis Type’.
  + First step is to create a ‘Recogniser’ via ‘[Train New Recogniser](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-custom-entity-recognizer)’.

**Training:**

* Create a training data list of words/phrases. This will have a structure of two columns, ‘text’ and ‘type’. Up to 25 types can be in the same document. There is no limit to the number of lines. Must have at least 200 lines per type to train the model.
* Run the ‘[Train New Recogniser](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-custom-entity-recognizer)’ on the above document. It takes around 30 minutes to complete. The ‘Recogniser’ so created can be used with any testing data.

**Testing:**

* Create a document with one document per line. i.e. each row in a CSV file will be the entire comments from a user. It can have multiple sentences.
* Create and run the [Analysis Job](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-analysis-job).
* Download the output.tar.gz and extract the ‘output’ as ‘output.csv’. It will have a structure as below.



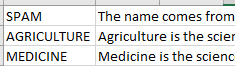
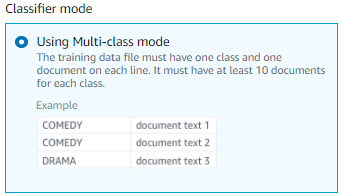
* The above output lists the start/end positions of the words/phrases found in the testing line(s)
* A Python program can parse this and mask the words/phrases in the text lines.

### Observations

* It takes around 7 minutes to complete the analysis.
* There is a limit of the number of entities found in each line. Only a maximum of three entities are listed in the output. If more than 3, the corresponding row loses the line number and makes it unusable in further analysis.
* No appropriate visual analysis method could be identified. Hence, resorted to using a Python program to parse the rows in the output and mask the identified words in the original testing document.
* The analysis does not always pick up a word/phrase. A word that was picked up on one line in the testing document may not be picked up in another line.
  + Also, if there are more than one word in the same line, it does not always pick all.
* It probably requires more studies to circumvent the above limitations. Perhaps the training doc is not long enough or some settings in the testing are not done correctly.
* An alternative to using the AWS Training/Testing, a Python program that I had developed for another purpose was modified to use in the analysis. This [program](https://github.com/asivapra/abs/blob/main/AWS_Comprehend/parse_comprehend-results.py) picks up and masks ALL occurrences of the words/phrases in the testing document.
* Since the Python program does not depend on the AWS Comprehend analysis, it is unsure of its place in the ABS-NLP system.
* We can use Lambda to trigger the program. Execution times may become a bottleneck. It takes around 25 minutes to finish scanning a 5,700-line document. In parallel execution, this time can be reduced to 3 or less minutes. For larger documents serial execution may become a limitation and hence parallel execution might be essential. However, it may not be possible to run it in parallel using AWS platform.

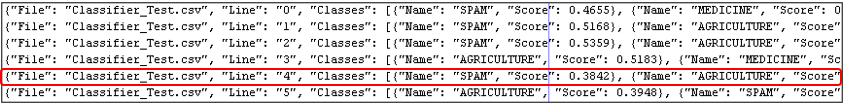
## Offensive Language Classification

The ‘[Custom Classification](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-analysis-job)’ under the AWS Comprehend analysis can classify documents based on their content. For this, a [training](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-classifier) document must be prepared in the following format.



The training document (see above) contained 10 lines in each category. Training took around 30 minutes, and testing the classifier on the testing document, which has the same rows of text as in the training set, took around 7 minutes.

The output is in JSON format as below and gives the scores for each classification. The classification that scores the highest is listed in the third column.



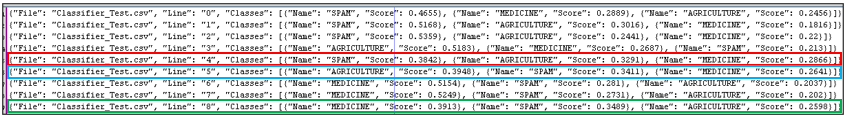
In the sample result, the first three rows were supposd to be “Spam” and the next three, “Agriculture”. However one (11% of the 9 total lines) was incorrectly identified. Since the testing was done with the same text as in the training doc, this should not have happened. The rate of success may improve with more data in the training document. A testing document other than the one used for training must be tested next. It looks like the maximum success percentage would be in mid-nineties.

In one of my projects, where single line texts were compared to each other, the success rate was 99.65%. With large documents of several pages the rate was still around 98%.

It is unclear what logic is used by AWS, but it appears to be based on the prevalence of words/phrases in each doc. Perhaps it can be improved by using ‘Cosine Similarity’ (CS) of the entire text as used in my Python program. I had introduced a method to screen out potentially unproductive comparisons, thereby reducing the execution times from 15 hours to around 6 minutes. This first-level screening could possibly miss some positives if the threshold for the screening was set too low. The false positives among the matched lines were only around 5 in 1500 (0.33%). False negatives were not assessed but it too would have been at around the same rate.

Since the error frequency is not high in the AWS classifier, we can probably combine the output with CS analysis to increase the accuracy of the results. For this, if the scores are close together in a line (e.g. “SPAM” score: 0.3842; “AGRICULTURE” Score: 0.3291) this line could be CS compared against a bunch of lines in each category and take the one that gives the higher CS score.

In the sample output below there are three lines that are borderline cases. We can test all with CS and either confirm or change the result.



This is worth testing and, if successful, will solve the issue of classification straight away.

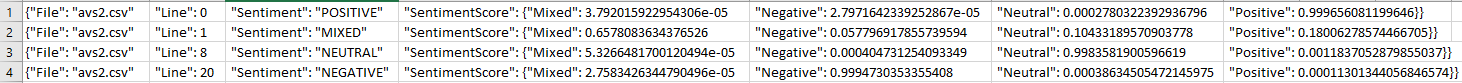
### Proposed way to use Cosine Similarity

If we choose to go by the Cosine Similarity (CS) method, one way to go would be as below.

* Prepare a list of similar documents (one line per document) with a classification (just like the input for AWS Classifier).
* Prepare the list of doc lines to test (again as for Classifier).
* Compare line by line between the documents and pick the highest CS value. Use a cut-off value and take the scores from 0.91 to 1.00. This cut-off value has been empirically determined from previous experiments.
* Take the classifier ID of the largest CS value line.
* I have a Python [program](https://github.com/asivapra/aipc/blob/main/products_compare_parallel.py) for it but needs modifications to use with long text lines. This program may only be practical if run as a parallel job, as the serial execution times will far exceed the limits on AWS.
* Caveat: This program’s logic may not be universally applicable. The CS scores of large texts (multi-sentences) may not give a clear cut-off value.

## Sentiment Analysis

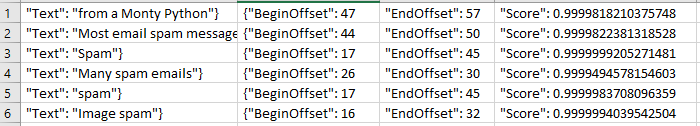
The AWS Comprehend/Sentiment analysis looks at the doc lines for the dominant sentiment. The output file lists the scores for the four classifications, ‘Mixed, ‘Negative’, ‘Neutral’ and ‘Positive and takes the highest score as the sentiment.



When using their sample doc as input, there was 100% accuracy in classifying the texts with sentiments. When my own document was used, there were 5 (2.4%) incorrect ones in 207 classifications. This still appears to be a good result. I do not have a Python program to find the sentiment and therefore cannot compare the results.

## Key Phrase Extraction

This analysis picks out words and phrases and give their indices in the document line. A score is associated with each phrase. It seems to pick up a lot of single words as well as phrases that do not make sense.



It is unclear how this data can be used in the project. Looks like key/phrase analysis is involved in the ‘Classifier’ analysis method. This could be useful in building a list of custom entities.

Using Python’s RAKE (Rapid Automatic Keyword Extraction) the output will give a score associated with each word/phrase. With single words the scores are lower than for phrases. Perhaps we can train the model based on these scores. For instance, take the highest one or two scores and then use manual assessment (reinforced learning) to pick the right phrases. In a [section](#_Spam_classification_–) below this method has been experimented for spam detection.

**Sample results**

s = "But the conspiracy theory carries weight because the rampage is almost indistinguishable from the mindless violence we have already seen carried out by Antifa over the course of 2020. Replace the red caps with red flags and you’d never know the difference."

[('rampage', 1.0), ('indistinguishable', 1.0), ('carried', 1.0), ('antifa', 1.0), ('replace', 1.0), ('difference', 1.0), ('mindless violence', 4.0), ('red caps', 4.0), ('red flags', 4.0), ('conspiracy theory carries weight', 16.0)]

## Topics Modelling

This has not yet been studied.

## Spam classification – an exercise to develop profanity analysis

The purpose of this exercise is to develop a Python program to identify and classify documents belonging to a particular class. Instead of using a Profanity word list (which exists, but I do not have training data) I am using a Spam word/phrase set on a list of documents created from actual spam mails quarantined by my anti-spam software.

The process is iterative. In the first iteration the words/phrases in the list are used to identify the lines in the training doc. Spelling variations are taken care of.

The lines so identified are then parsed for key phrases and two phrases with the highest scores are added to the word list. These phrases will be used in the second iteration and the process is repeated. Eventually, after certain number of iterations, no more new phrases will be picked.

The word/phrase list is then manually edited to keep only the words and phrases relevant to Spam identification.

\_\_\_END OF SECTION\_\_\_

# Glossary

* AWS: Amazon Web Services
* NLP: Natural Language Processing
* NLP Project: The ABS Machine Learning project using NLP
* S3: Simple Storage System from AWS
* EC2: Elastic Computing from AWS
* Lambda:
* Comprehend: AWS service that includes various analyses.
  + Custom Entity Recognition.
  + Custom Classifier.
  + Sentiment Analysis.
  + Key phrases.
* Classifier: AWS Custom Classifier analysis
* Sage Maker:
* Profanity: Bad words and phrases in text.
* Key phrase: Combination of words in text, not necessarily meaningful phrases.
* Topic modelling:
* Cosine Similarity: Mathematical scoring of similarities between texts.
* RAKE: Rapid Automatic Keyword Extraction. A method to extract key phrases from text.
* Better\_profanity: Python library for identifying profanity in text.
  + BP: Better\_Profanity
* Levenshtein Distance: Mathematical method to correct spelling mistakes.
  + LD: Levenshtein Distance
* Edit Distance: Minimum changes, insertions and deletions to reach correct spelling.
* Spam: Email spam

1. This download link does not work. Gets a “Page not Found” error. [↑](#footnote-ref-1)