Table of Contents

[AWS Comprehend – Usage and Issues 2](#_Toc60490070)

# AWS Comprehend – Usage and Issues

Created on 2 Jan 2021

Author: Arapaut V. Sivaprasad

# INTRODUCTION

This is an evolving document and contains the points learned during the use of AWS Comprehend. Some of the observations may be premature, incomplete or mis-understood. These will change as I gain more experience in its usage.

In the NLP Design document, the following analyses are described.

*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*

# 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 these will not be repeated here.

## 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. This study will be aimed at identifying the words/phrases within a sentence and, optionally, mask them without discarding the entire sentence.

### Steps

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

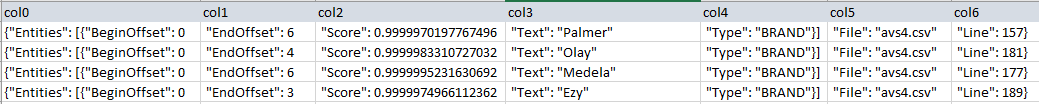
* We will be using ‘AWS Custom Entity Recognition’ under ‘[Analysis Jobs](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create-analysis-job) | Analysis Type’.
  + First step is to create a ‘Recogniser’ via ‘[Train New Recogniser](https://ap-southeast-2.console.aws.amazon.com/comprehend/v2/home?region=ap-southeast-2#create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



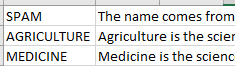
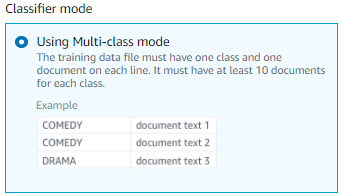
* The above output lists the start/end positions of the words/phrases found in the testing line(s)

### 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 which 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 on one line in the testing document may not have been picked 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 some other 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-ML 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.

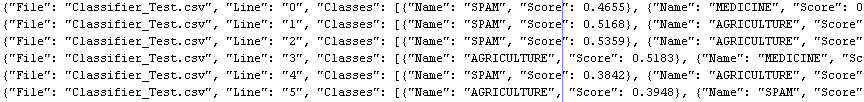
## Offensive Language Classification

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 testing 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, too 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 the one to be chosen.



In the sample result, 1 out of 30 (3.3%) was incorrectly identified. This rate of success may improve with more data in the training document. A testing document other 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, the success rate was 99.65%. With large documents of several pages the rate was lower but 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’ 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. Though it could miss some positives, but was unlikely, the false positives were only around 5 in 1500 (0.33%). False negatives were not assessed but it too would have been at around the abobe rate.

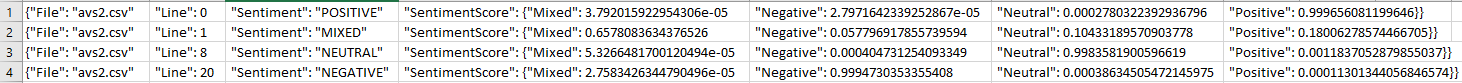
### Proposed way to use Cosine Similarity

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 of 0.91. This 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 to be run as a parallel job, as the serial execution times will far exceed the limits on AWS.

## Sentiment Analysis

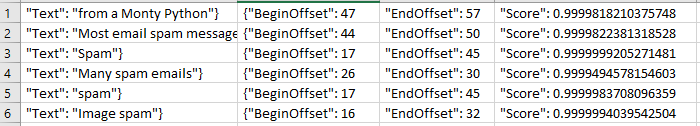
The AWS Comprehend/Sentiment analysis looks at the doc lines for the dominant sentiment. The output file lists the scores for the four classifications, ‘Mixed, ‘Negative’, ‘Neutral’ and ‘Positive and takes the highest score as the sentiment.



When using their sample doc as input, there was 100% accuracy in classifying the sentiment. When my own document was used, there were 5 in 207 (2.4%) incorrect classifications. 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.

## Topics Modelling

This has not yet been studied.

## Spam classification

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 data set (which exists!) I am using a Spam word/phrase set compiled from various sources on a list of documents created from actual spam mails quarantined by my anti-spam software.