COMP2271: Data Collection and Cleaning, Report

Banner ID: 000810185

0 Preface

0.1 Prerequisites

The codebase has been written entirely in **Python 3.9**. Various elements from this version of Python have been used, so the code will *only work* with Python 3.9.

The following external libraries (with their corresponding versions) have been used:

- bs4==0.0.1
- openpyxl==3.0.7
- lxml==4.6.3
- numpy==1.20.2
- requests==2.25.1
- pandas==1.2.4
- nltk==3.6.2
- sklearn==0.0
- seaborn==0.11.1

All of these can be installed in the usual way with **pip**, i.e.:

```
pip install [MODULE_NAME==VERSION]
```

For example:

```
pip install requests==2.25.1
```

Once these external libraries have been installed, the program may be run from the terminal as follows:

```
python 000810185.py
```

or

```
python3 000810185.py
```

0.2 Notes on Running Time

On average, when the default parameters are used, the program will take around 10 to 13 minutes to complete.

The methods for problem 1 will usually take up to 2 minutes. This runtime can be reduced by increasing the `NUM_WORKERS` parameter - by default, this is set to 100. Note that originally this part of the program took around half an hour to complete, but I optimised it by using asynchronous methods, which resulted in a 94% speed improvement.

The methods for problem 2 generally take up to 30 seconds. This is, of course, contingent on the number of keywords used in the *keywords.xlsx* file as well as the number of articles collected for each. My program has been configured to handle all possible formats of news articles that may be collected from BBC News for each of the original 10 keywords.

The running time of the functions for problem 3 (in particular, my algorithm that facilitates the calculation of *semantic similarity*) is heavily dependent on the parameters used. With the default parameters, the preprocessing/data cleaning stage will take roughly 2 to 3 minutes, and the algorithm itself can take up to 7 minutes. Increasing the `WINDOW_SIZE` and `NEGATIVE_SAMPLE_SIZE` parameters will result in a running time increase. Conversely, by setting the `COLLECT_ALL` parameter to `False`, the running time will be significantly reduced. This is because only highly relevant sentences (i.e., those specifically containing one of the keywords) from the articles will be used by the algorithm (in general, this reduces the vocabulary size from around 12,000 to around 5,000).

Finally, the functions for problem 4 are all very quick as most of the work is done behind the scenes by Seaborn, which has been highly optimised for graphical programming. Generally, this section will take around10 seconds, with a small additional overhead incurred due to the requirement of dimensionality reduction for one of the visualisations.

0.3 Additonal Notes

To avoid any unexpected exceptions, please ensure that any excel or text file that the program must interact with is closed when the program is running.

Moreover, please note that there is additional commentary to be found in the code-base itself, within comments. This specifically applies to the functions for problems 1 and 2, since I have not elaborated on these in this report.

1 The Algorithm

1.1 Introduction

In linguistics, it's often the case that the semantics of a given word is derived from the denotational concept of the *sign* and *signifier*. That is, briefly, the meaning of a word is essentially derived from the idea, or *thing*, represented by that word. However, this does not lend to a convenient computational model. A common and effective solution for deriving word semantics computationally is by analysing words *in relation* to other words. That is, the program can decide the similarity of words based on how often they appear together, or how often they appear in similar contexts (i.e., how frequently they appear with similar words or phrases). Of course, the system undertaking this process is ignorant about the actual conceptual ideas that underlie these contexts, but it is able to mathematically represent them in order to derive word similarity.

Thus, such is how textual semantics can be understood within the realm of computation: not by the literal notions that define meaning (as in humans), but by mathematical representations and the relationships thereof. The notion I have just outlined is well established within the field of linguistic philosophy; it has a name: distributional similarity. Following from the idea that *'you shall know a word by the company it keeps'* [1], representing a word by means of its *neighbours* can be a great source of value in computational linguistics. This is akin to Wittgenstein's proposition that to understand the meaning of words is to accurately predict their usage in text - that is, word meaning is derived from textual context.

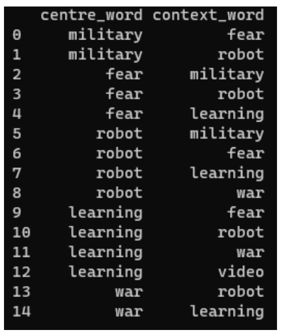
1.2 Word2Vec

To relocate these linguistical concepts into a more computational realm, we can build a **dense vector** for every distinct word (or phrase) which serves to predict the likelihood of other words appearing in the same context as this word. That is to say, the model takes words as inputs and quanitfies them as series of numbers that aim to encapsulate the underlying semantics of the inputted word. This notion, along with those outlined earlier, underscores the **word2vec** algorithm, introduced by Google in 2013 [2].

To achieve the intended outcome of constructing word embeddings, the model requires a large corpus of text - specifically, the corpus should comprise sentences because the essence of the model is *word contexts*. This corpus basically represents the dataset for the algorithm. To preprocess the dataset, some standard natural language processing data cleansing methodologies can be applied, including special symbol and stop word removal, conversion of all text to lower case, and stemming or lemmatisation. These additional methods are generally not entirely required for very large corpora, however I did incorporate them into my implementation.

My implementation of the algorithm uses as its corpus the sentences contained within the news articles collected for problems 1 and 2. The preprocessing/data cleaning methods in my codebase are contained within the class entitled `W2VPreprocessor`. First, because the algorithm is designed to work with singular words and not multi-word phrases, every *keyword* from the original keywords.xlsx file that contains multiple words is encoded such that any space (i.e., " ") is replaced with an underscore (i.e., "_") - for instance, "malicious bot" becomes "malicious_bot". The keywords in the corpus are then *depluralised*, meaning any instance of one of the keywords that end with an "s" is replaced with that keyword with the "s" removed. This means that variants of the same word, which of course contain the exact same underlying semantics, are encoded as the same word embedding. After converting the remainder of the corpus to lower case (which decreases the number of unique words in the corpus), the **NLTK** library [3] was used for lemmatising the words in the corpus, which entails shortening a word to its *prefix* (for instance, "running" would become "run"), followed by word tokenisation. To handle stop words, I found a large set of frequent stop words online [4], and this was used in conjunction with a function I wrote to *subsample the most frequently occurring words* in the corpus, and remove them according to a given threshold.

The version of the algorithm that I designed and implemented follows the *skip-gram* variant, utilising *negative sampling* amongst various other techniques. This is mainly where my implementation differs from the literature in terms of its design and the data structures used. In this report, I will only outline my personal interpretation and implementation, which extensively utilised the **Pandas** and **Numpy** libraries. The dataset is initially constructed from the corpus by essentially selecting the words within the sentences one by one, and building up a Pandas *DataFrame* by appending a row for each of the words neighbouring this selected *centre word* based on the user-specified `WINDOW_SIZE` parameter (which, by default, is set to 2 - thus obtaining 4 neighbouring words, the two preceding it and the two following it). To illustrate this further, **Figure 1** shows the first 15 entries in this DataFrame after it is first initialised.

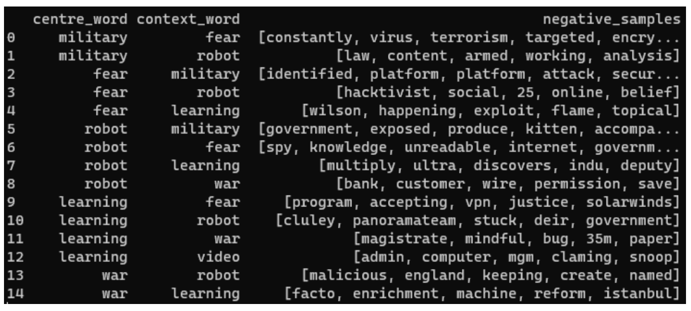


```
   centre_word context_word
0     military         fear
1     military        robot
2         fear     military
3         fear        robot
4         fear     learning
5        robot     military
6        robot         fear
7        robot     learning
8        robot          war
9     learning         fear
10    learning        robot
11    learning          war
12    learning        video
13         war        robot
14         war     learning
```

Figure 1: The skip-gram dataset, when a window size of 2 is used.

This method is precisely what is described as the **skip-gram** approach in the literature. This part of the dataset is key for the algorithm to learn the semantics of each word in the vocabulary. The words on the right column are the *context words* associated with its neighbour in the left column. The algorithm will use this information to discover other words in the dataset that have similar words occurring as the context word. For instance, in the image above we see the word *"military"* appearing alongside *"fear"* and *"robot"*. If this combination of words appears relatively frequently, then other words who also appear beside *"fear"* and *"robot"* will be deemed to be ***similar*** to *"military"* by the algorithm.

Now, this method alone provides the algorithm with a way of figuring out which words are *similar*, but it must also somehow work out which words are *not* similar. In my implementation, I achieved this through the technique of **negative sampling**. By default, the negative sampling parameter in my program is set to 5. This means that, at the start of each epoch, an additional row is added to the DataFrame (which is replaced each epoch) comprising a list of 5 words that *do not* appear in the context of the corresponding centre word. The words within these lists are selected randomly based on a *weight* associated with their frequency within the corpus (see the `get_noise_distribution()` function in my code). An example of the dataset after negative sampling has been applied is shown in **Figure 2**.



```
   centre_word context_word                                    negative_samples
0     military         fear  [constantly, virus, terrorism, targeted, encry...
1     military        robot          [law, content, armed, working, analysis]
2         fear     military  [identified, platform, platform, attack, secur...
3         fear        robot          [hacktivist, social, 25, online, belief]
4         fear     learning      [wilson, happening, exploit, flame, topical]
5        robot     military  [government, exposed, produce, kitten, accompa...
6        robot         fear  [spy, knowledge, unreadable, internet, governm...
7        robot     learning        [multiply, ultra, discovers, indu, deputy]
8        robot          war          [bank, customer, wire, permission, save]
9     learning         fear    [program, accepting, vpn, justice, solarwinds]
10    learning        robot   [cluley, panoramateam, stuck, deir, government]
11    learning          war            [magistrate, mindful, bug, 35m, paper]
12    learning        video            [admin, computer, mgm, claming, snoop]
13         war        robot      [malicious, england, keeping, create, named]
14         war     learning    [facto, enrichment, machine, reform, istanbul]
```

Figure 2: The dataset after applying negative sampling.

In terms of the machine learning aspect of the algorithm, it works via neural networks and the weights are trained via stochastic gradient descent. To elaborate further on what this entails, before the aforementioned methods are applied to construct the dataset, every *unique* word is extracted and assigned a unique integer (in the literature, this would be a *one hot encoded* vector, but for convenience and compatibility with my Numpy/Pandas-oriented implementation, a single integer sufficed). These integers are directly related to indices in two large Numpy matrices, whose lengths correspond with the number of unique words in the vocabulary and the depth of the vectors at each index is specified by the parameters (by default, this is 300 - this is a fairly typical value for the depth of word embeddings). These matrices are the **embedding** matrix and **context** matrix. The embedding matrix is trained to store the final word embeddings of each word in the vocabulary. The context matrix is trained similarly, except the vectors here represent the relative likelihood of the words appearing in the context of another word.

The algorithm operates by going through each example in the dataset and mapping the centre word, context word, and negative sample words onto their corresponding vectors contained within the embedding matrix (for the centre word) and the context matrix (for both the context and the negative sample words). The values contained in the vectors within the embedding matrix are related to the likelihood of a given context vector appearing within the context window of that word. Likewise, the values in the context matrix's vectors are related to the likelihood of the associated word appearing in the context of a given vector in the embedding matrix. During gradient descent, for each training example, the word vector for the centre word has values *increased* that correspond with the likelihood of the context word appearing in its context, and values *decreased* that correspond with the likelihood that each of the negative sample words appears in the context. The process is similar to the vector within the context matrix for the context word.

Once training has completed, the embedding matrix will now contain the final word vectors for each word in the original vocabulary. Due to this way of training, the vectors for words that the algorithm deems to be more *similar* will themselves be more similar - this similarity, between any pair of word vectors, can be calculated by their *cosine similarity*, which I will discuss further in the next section. Although word2vec is a well-established and widely used word embedding algorithm, I stress that the implementation found in my codebase has been written entirely on my own, only following the information found in the original article in addition to a lecture by Stanford [5].

2 Visualisations

The following section outlines the visualisations that I have produced for the processes surrounding my algorithm discussed in the prior section. All of these visualisations were produced using the Seaborn library [6].

2.1 Data Cleaning and Preprocessing

The articles collected in problem 2 are stored as text files, each contained in a folder dedicated to its corresponding keyword. For instance, the "encryption" folder contains 100 **.txt** files, each of which storing the content of the associated article with the article title and each sentence of its body separated by newlines. This format allows my program to read in the content with ease. A Pandas DataFrame is used to collect this content, where the first column is composed of the keywords (which it obtains from the titles of the folders), and the second column being composed of sentences that have been collected from articles associated with each keyword.

As an additional optional parameter, the program can be toggled to only read in sentences that actually *contain* its corresponding keyword. This facilitates a faster runtime, for both the preprocessing of the corpus and the word2vec algorithm itself, and potentially benefits from the notion that smaller, more relvant corpora can be more effective than larger ones, with more irrelevant data [7]. For the sake of completeness, however, by default, my program is set to use the entire corpus, and the entire corpus was used to gather the following visualisations.

2.1.1 Visualising the corpus size

The number of sentences collected from articles associated with each keyword is illustrated in **Figure 3**, wherein the count (y-axis) denotes the number of occurrences of each keyword (x-axis) in the DataFrame mentioned previously.

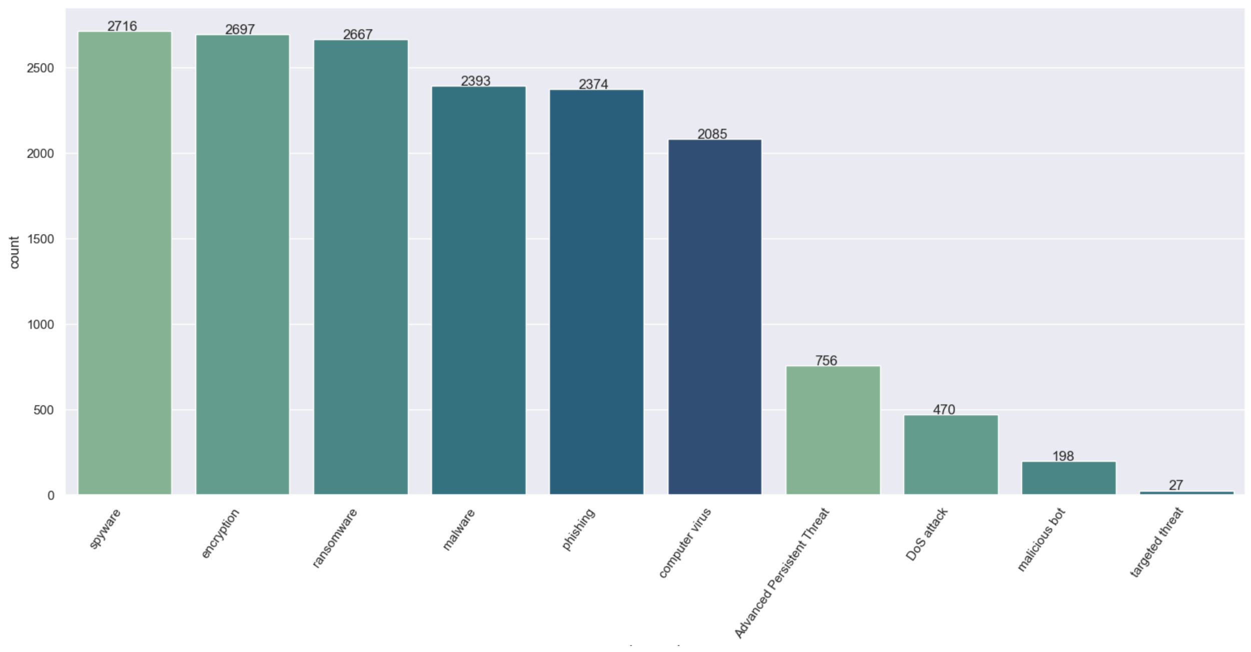


Figure 3: Barplot visualising the number of sentences collected from articles associated with each keyword.

This visualisation was constructed using Seaborn's **countplot()** method. From this bar graph, we can immediately deduce that whilst the most frequently occurring 6 keywords all have relatively similar counts, there is a drastic drop by the 7th. Moreover, it is observable that the keywords with the lowers counts each comprise two or more words, which suggests that it may be easier to attain more data for singular keywords as opposed to multi-word phrases. The implications of this distribution will be further discussed later.

2.1.2 Analysing the sentence lengths

For further analysis of the data prior to data cleaning, the distribution of sentence lengths in the entire corpus is displayed in **Figure 4** - this diagram was constructed using Seaborn's **displot()** method.

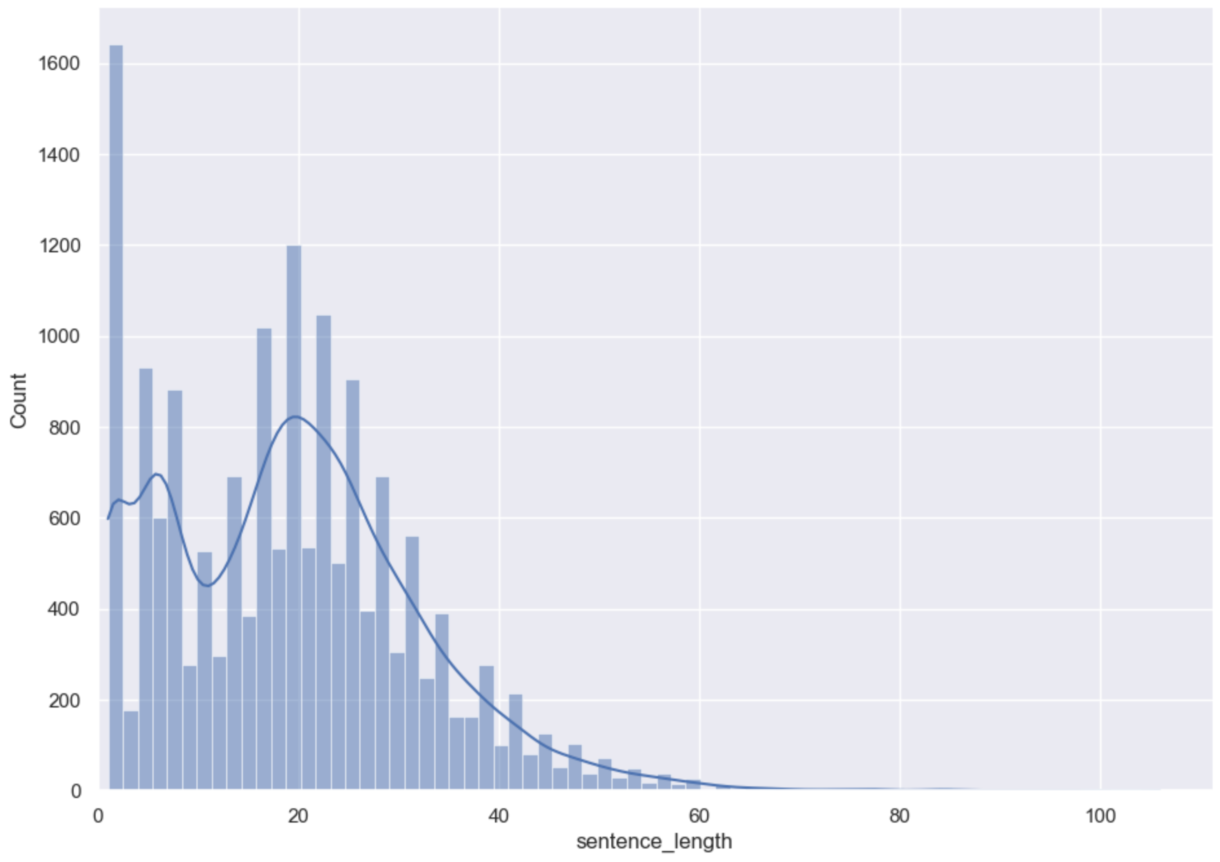


Figure 4: Distplot visualsing the distribution of sentence lengths amongst the corpus.

Various interesting patterns can be observed here. For instance, the majority of sentences are around 20 words long, however, the modal sentence length is 1 word. This is likely due to certain unavoidable aspects of data collection over the web, where anomalies are likely to occur. This particular anomaly will be dealt with during preprocessing. Moreover, a small proportion of sentences have a length greater than 50 to 60, which represents a further anomaly that may require attention during data cleaning. It is also observable that the majority of sentences have an odd number of words.

For a more in-depth analysis of the distribution of sentence lengths, **Figure 5** illustrates the contrasts in the sentence length distributions for each keyword. Interestingly, the patterns noted previously persist here, and as the number of sentences for a given keyword increases, the distribution tends to a similar shape.

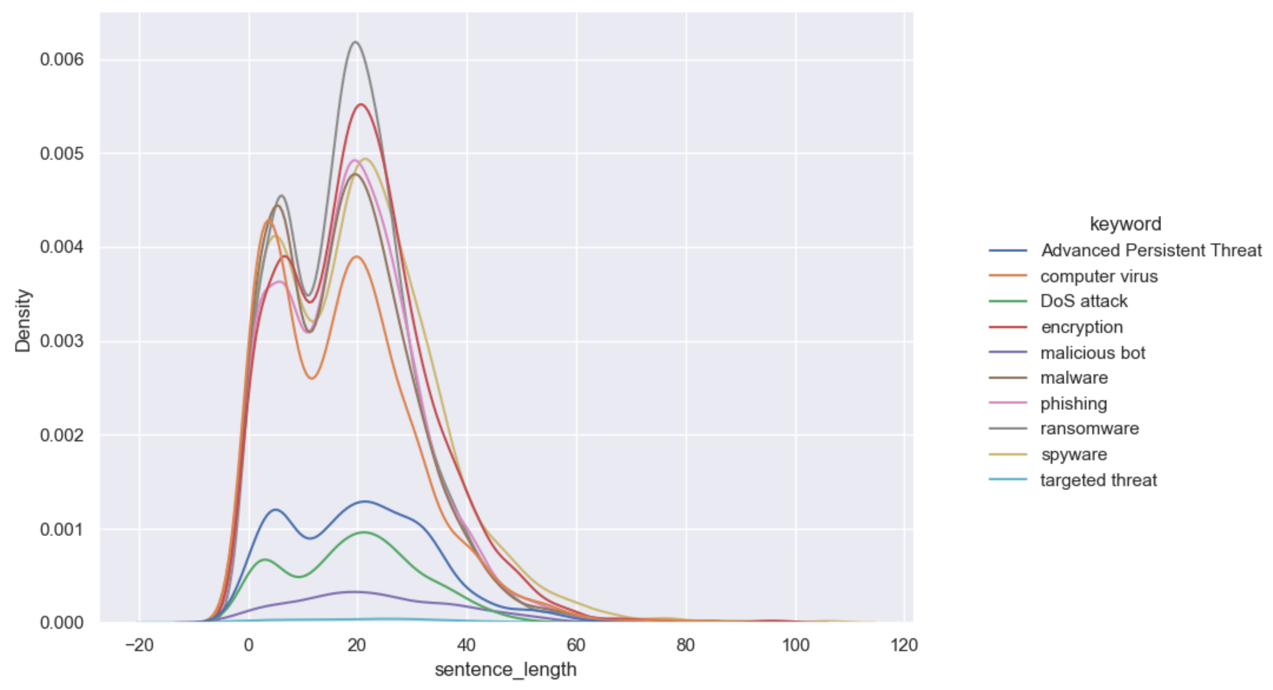


Figure 5: Distplot visualising the distribution of sentence lengths per keyword.

2.1.3 Visualising frequently occurring words and stop word removal

One key stage of natural language preprocessing is the detection and removal of **stop words**. This principle is particularly prevalent within the word2vec algorithm that I implemented, whereby the similarities of words are derived from the likelihood of words appearing in similar contexts. Therefore, frequently occurring words with minimal lexical value are likely to distort, or over-inflate, the resulting semantic similarity scores. To illustrate this problem, **Figure 6** showcases the top 25 most frequently occurring words within the corpus.

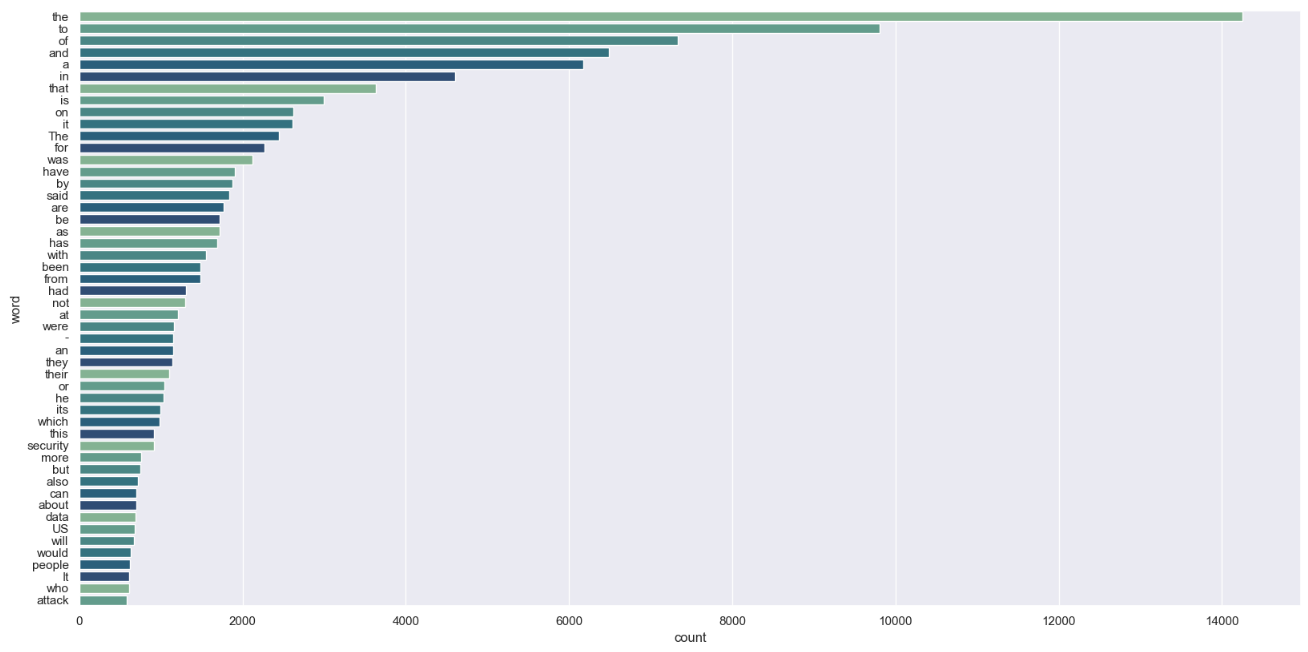


Figure 6: The most frequently occurring words in the corpus, prior to applying data cleaning.

As is shown, the most popular word is "the", followed by many other lexically unhelpful words. In fact, substantially few lexically valuable words appear amongst the most frequent. To elucidate why this is an issue, suppose a word appears very often beside the word "the". It's likely to be the case that many other words, ones that are semantically reasonably distinct from the former, would also appear beside "the". As a result, these words may be given a greater similarity score by the algorithm purely because they are equally likely to appear beside "the", and not because they are genuinely semantically similar.

The effect of data cleaning (i.e., stop word and special symbol removal) on the most common words is demonstrated in **Figure 7**. It can clearly be observed that the value per word in the corpus has now increased.

It's interesting also to note that the most frequent word, following the removal of the stop words, is "security", and many of the other words are related to the concept of Cyber Security (such as "data", "attack", and even many of the original keywords like "ransomware"). This is a good sign as it implies the collected vocabulary is highly relevant in terms of its relationship with the keywords.

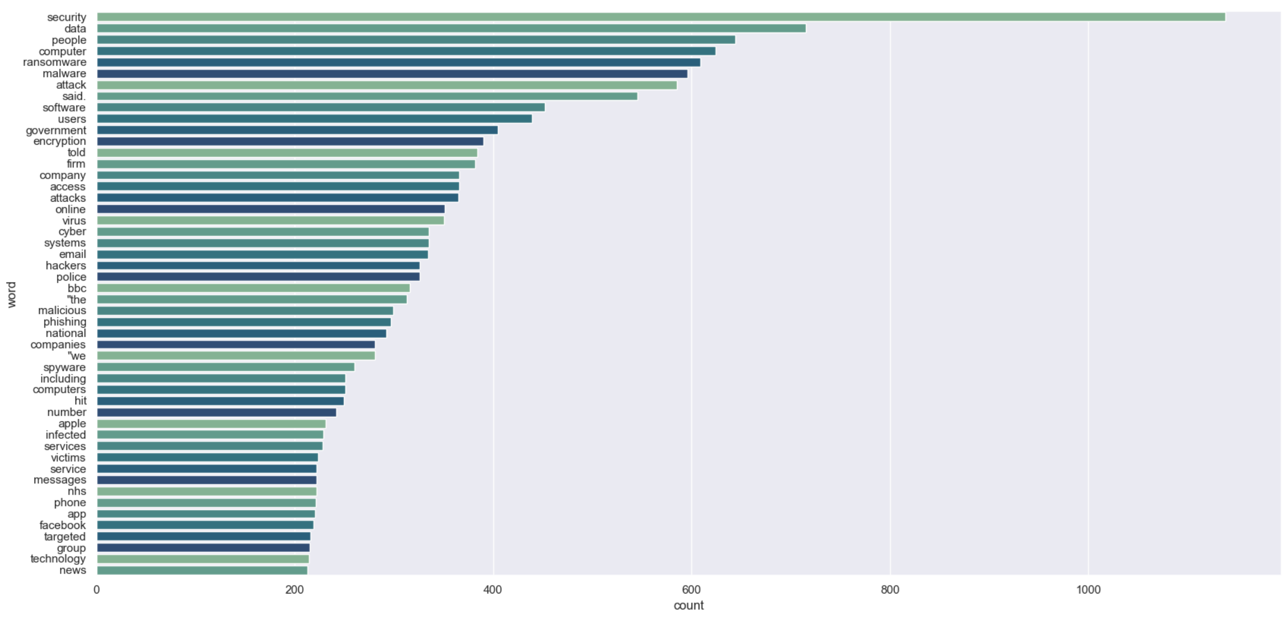


Figure 7: The most frequently occurring words in the corpus, after applying data cleaning.

2.2 Results and Semantic Similarity

Please note that the following semantic distance metrics are highly subject to change as the corpus itself (i.e., being derived from BBC news) is constantly evolving. Therefore, the vocabulary will shift here and there following subsequent runs. Moreover, the algorithm is very sensitive to its parameters, so altering them may lead to results that are distinct from those shown below. That said, the general patterns present in the results shown here do persistently arise (assuming the parameters are set appropriately), albeit with different absolute values.

To clarify the notion of semantic distance that the algorithm I implemented goes by, the similarity scores are obtained by calculating the *cosine similarity* between two word vectors. This is computed via the following formula:

$$\cos\theta = \frac{A \cdot B}{||A|| \times ||B||}$$

where A and B are two word vectors.

These cosine similarity scores reside between 0 and 1. Higher numbers (i.e., those closer to 1) indicate greater similarity, whereas lower numbers (i.e., those closer to 0) indicate less similarity.

2.2.1 Semantic distances heatmap

In order to visualise the semantic distances between each pair of keywords, the most intuitive approach is via a 2D matrix representation. As part of the coursework, a 2D distance matrix was produced and stored in an excel file, which displays the keywords along each axis and the corresponding semantic similarity scores between each pair. This format neatly translates to a **heatmap**, available as a part of the Seaborn library. The heatmap produced for the aforementioned distance matrix is displayed in **Figure 8**.

Note that care was taken when selecting the colour palette for this heatmap. This colour palette has a hierarchical nature, in that darker colours are related to greater similarity whilst light colours indicate reduced similarity.

Various interesting points can be derived from this visualisation. Intuitively, the scores assigned by the algorithm, as displayed in the heatmap, appear to be reasonably accurate. Of course, the fact that all words have a perfect similarity score with themselves requires no elaboration. The words "malware", "phishing", "ransomware", and "spyware" all have considerable similarities with one another, with scores ranging from 0.75 to 0.82. This is understandable due to the fact that they all represent forms of computer virus. The relatively high similarity that "encryption" has with these words is also understandable as it has a significant relationship with them as a form of protection. Whilst less so than the aforementioned words, "computer virus" also has a reasonable similarity with them.

It's interesting to note how the scores for "malicious bot" and "targeted threat" are all relatively close to 0.5 (which indicated neutrality). This can be explained by both the relatively smaller quantity of articles collected for these terms as well as the general semantic ambiguity that they have. However, the fact that no two words here have been deemed semantically unrelated by the algorithm (i.e., having scores close to 0) makes sense, as they are all related by the concepts of computer science and, more specifically, cybersecurity.

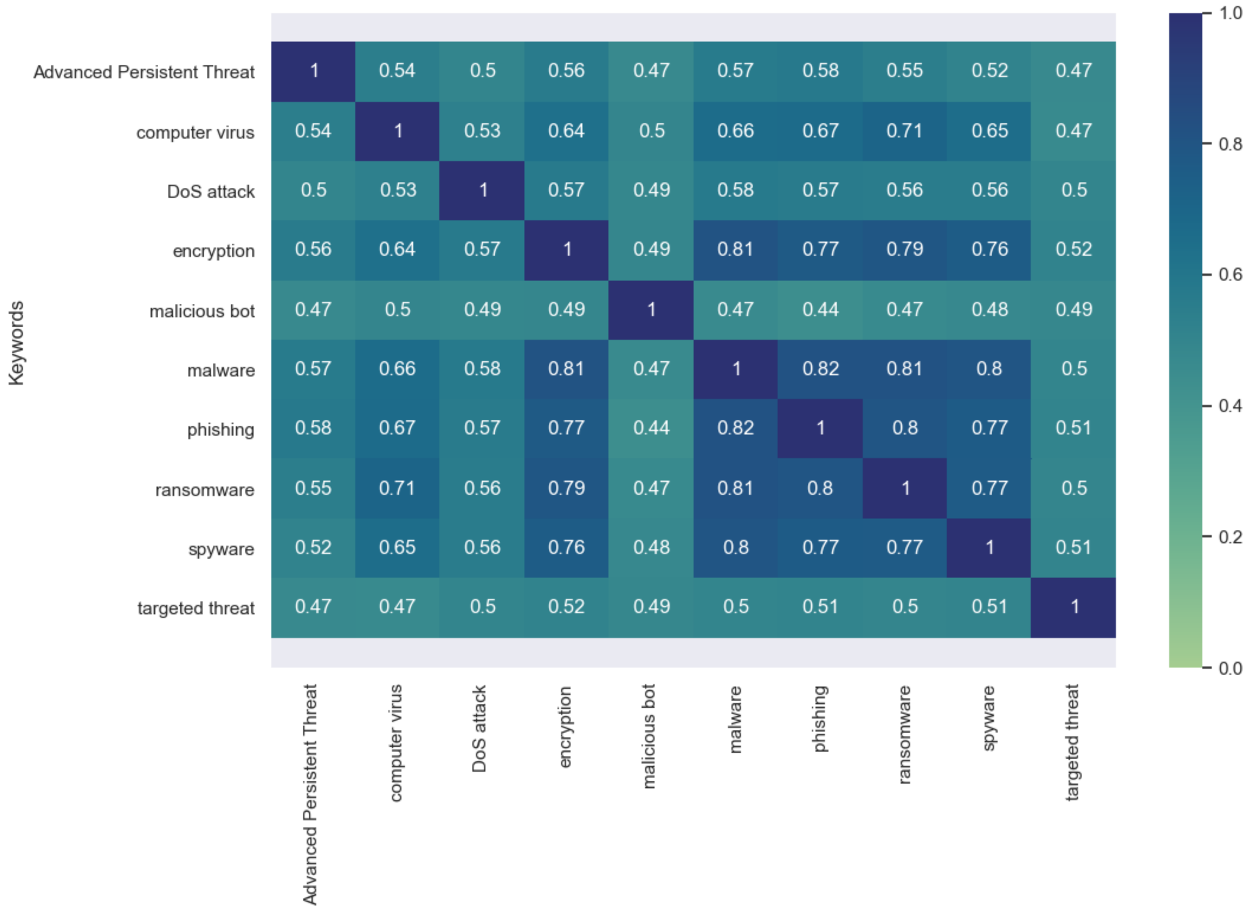


Figure 8: A heatmap showing the raw cosine similarities between each pair of keywords.

2.2.2 Dimensionality reduction and 2D plotting

Another way of visualising the semantic distances between the keywords is by *plotting* them. The original depth of the word embeddings constructed by the algorithm is 300. Of course, it's impractical to plot 300-dimensional vectors using any standard coordinate system. To resolve this, the technique of dimensionality reduction can be used. To elucidate this further, I used **Principle Component Analysis** (PCA) to obtain the 2-deminsional principle components of the embeddings for each keyword. After each word vector has been reduced to 2 dimensions, it's trivial and intuitive to then plot these vectors on a 2D plane. Utilising Seaborn's **relplot()** method, the 2D principal components were plotted and the result of this is shown in **Figure 9**.

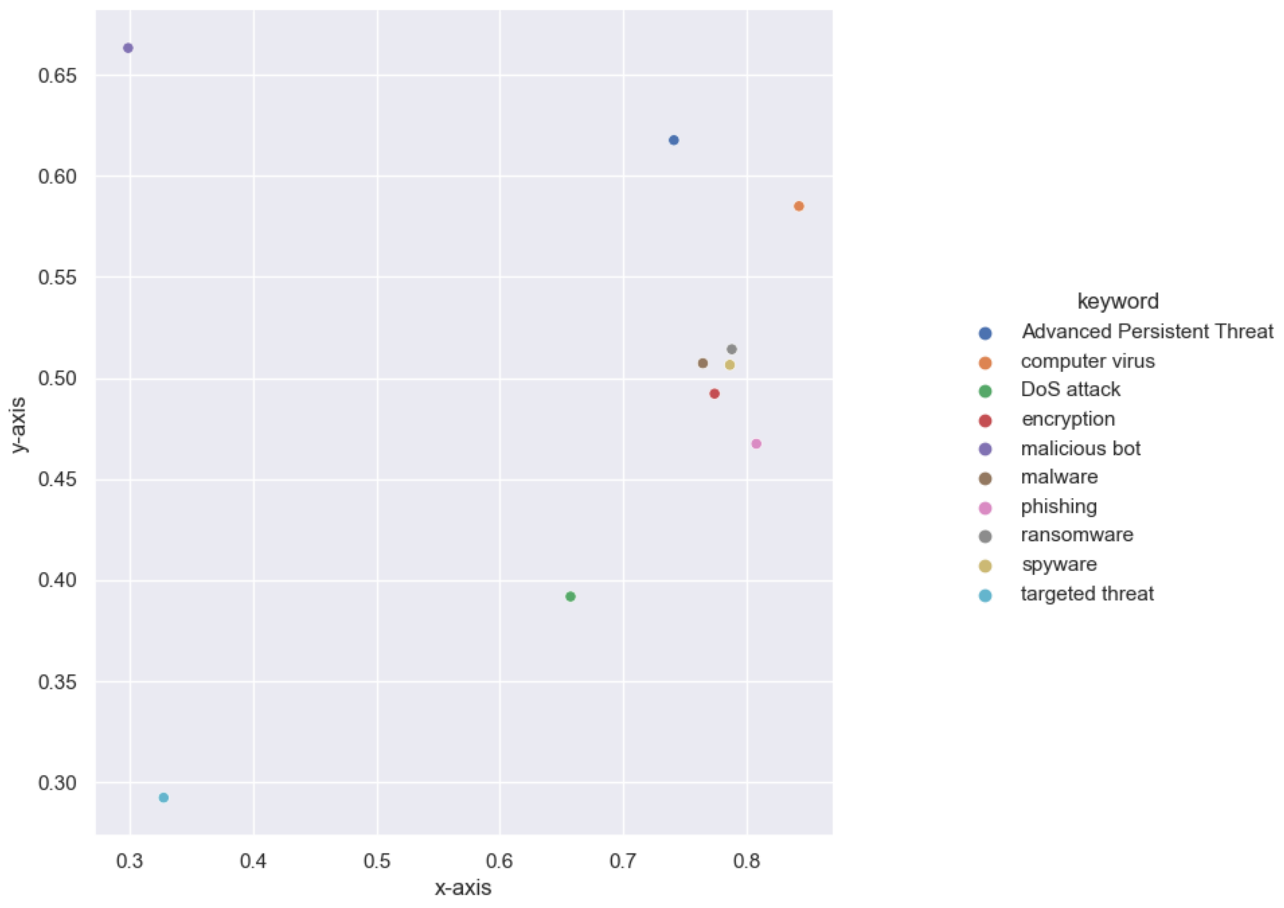


Figure 9: A scatter graph, showcasing a plotting of the word vectors after applying dimensionality reduction to make them 2D.

We can observe that the distances between each word generally correspond with the semantic similarity scores showcased in the heatmap. That is to say, words with greater similarity scores are more likely to appear closer together. Of course, this is not an entirely rigorous visualisation as it relies on the principal components of the word vectors, so some detail contained within the original embeddings may be neglected. That said, they do provide a generally sound notion of the similarities.

Please note that due to the sensitivity of the algorithm, the precise locations of the points are subject to change on subsequent runs, however the general patterns and distances usually remain consistent.

2.2.3 Semantic similarity distributions

The visualisations showcased so far for the semantic distance metrics demonstrate the assigned similarities between the keywords. The following visualisations serve to demonstrate the relative *accuracy* of the semantic similarity scores in general. First, I utilised Seaborn's **violinplot()** method to construct a series of violin plots that illustrate the distribution of sentiment similarity scores between each of the keywords and the remaining words present in the corpus. These violin plots are shown in **Figure 10**.

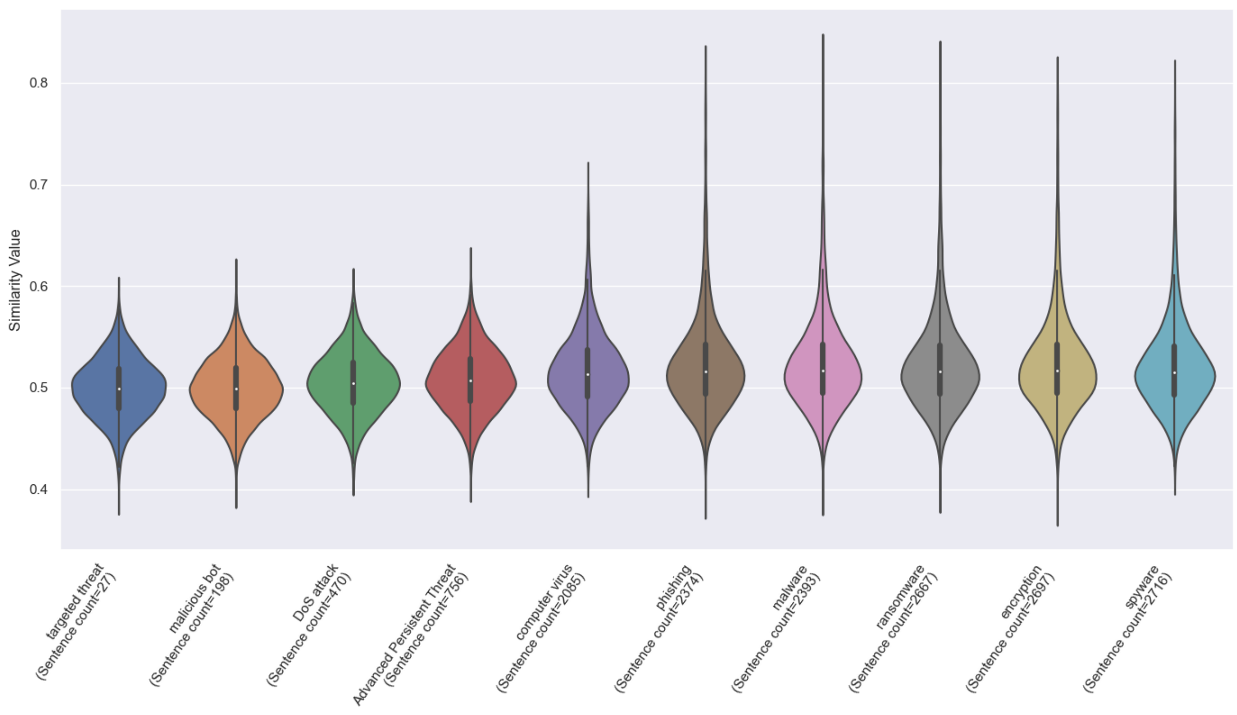


Figure 10: The distribution of similarities with other words in the corpus, for each keyword.

I included the *sentence counts* beside the keywords in the x-axis as this allows us to observe the effect that collecting more sentences for a given keyword has on its resulting similarity distribution. As one would expect, the average similarity scores for each keyword is around 0.5, however, there are certain other points of intrigue that are visible in this plot. Firstly, as the number of associated sentences collected for a given keyword increases, as does the variance of the distribution of similarity scores. This makes sense because as the amount of relevant content for a keyword increases, the algorithm has a greater capability of making a judgment on how similar it is with other words.

The second point of intrigue is that the similarity scores rarely go far beneath 0.4, indicating that the algorithm is unlikely to conjecture that any two words are moderately dissimilar. Although this may appear to be a bug of some kind at first, a valid explanation for this is available. That is, because the corpus is comprised of news articles related to the keywords, and because the keywords all have relatively similar semantic contexts in terms of their relation with Cyber Security, the majority of the vocabulary in the corpus following the removal of the stop words are also related to Cyber Security. This fact is apparent in **Figure 7** from earlier, which illustrates that the most frequently occurring words are more often than not associated with the topic of Cyber Security.

2.2.4 Nearest semantic neighbours

Finally, to further demonstrate the success of the algorithm, I decided to visualise the words with the greatest similarity scores with each keyword. I utilised the **barplot()** method of Seaborn to create a bar for each of the 3 nearest semantic neighbours for each keyword and labeled each bar with the corresponding word. The result is shown in **Figure 11**.

From this diagram, and those preceding it, we can conclude that the word2vec algorithm assigns semantic similarities with reasonable accuracy. The nearest neighbours for each keyword found by the functions I implemented are all generally related to the respective keyword in some way.

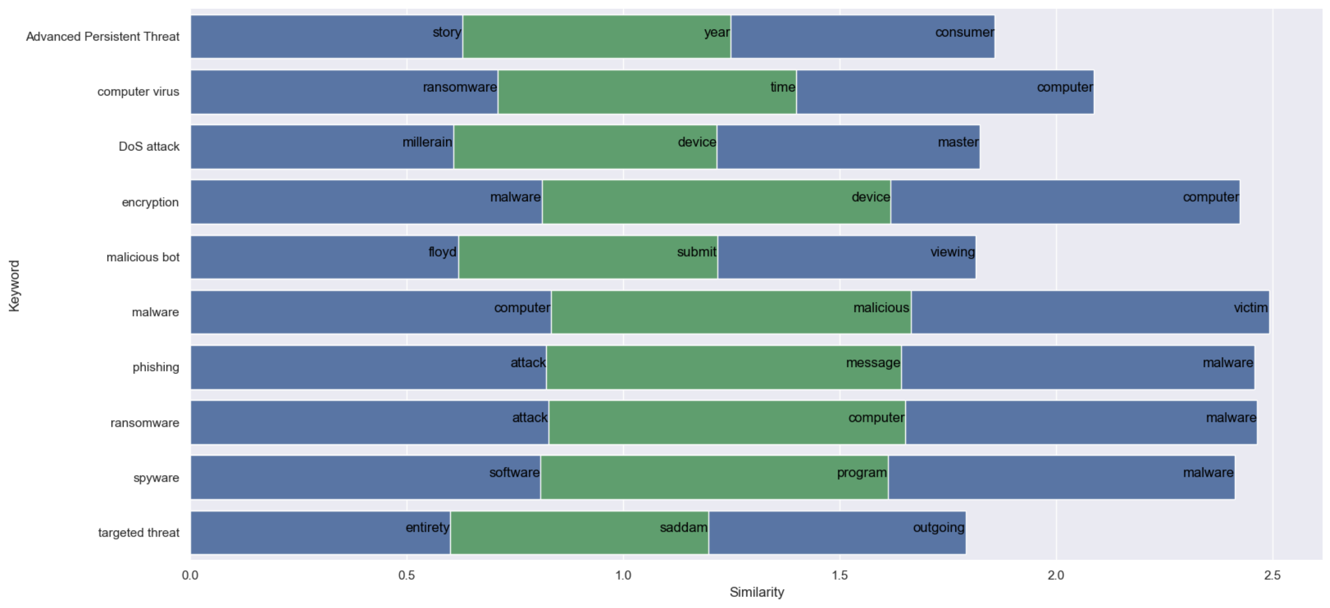


Figure 11: The 3 nearest neighbours to each of the keywords.

3 References

1. Firth, J. R. 1957:11
2. Mikolov et al., "Efficient Estimation of Word Representations in Vector Space"
3. Natural Language Toolkit: https://www.nltk.org/
4. Lists of stop words: https://gist.github.com/sebleier/554280
5. Stanford CS224N Lecture on word2vec: https://www.youtube.com/watch?v=ERibwqs9p38
6. Seaborn: https://seaborn.pydata.org/
7. Dusserre, and Padró, "Bigger does not mean better! We prefer specificity." IWCS (2017).