# Data Science Toolbox Assessed Coursework 3

Dan Jones, Kishalay Banerjee, Sam Harding

February 22, 2019

## Abstract

In this report we generate a dataset containing source code from popular Github repositories in Python, R, and Javascript. A number of models are then developed, using Latent Dirichlet Allocation (LDA) to find 'topics' within our dataset. These models are run on a reduced, and large dataset, depending on the models hardware requirements. Next, we posit that the topics found by LDA will best map to the programming languages, rather than more general subjects or themes. We develop an expected/ideal word distribution for each programming language, and use the Jaccard Index to find the closest language for each topic generated by our model.

The trained models are then run on a labelled training dataset where a predicted programming language mixture is output. These are compared against the known programming langauge mixtures via the Kullback–Leibler divergence, a method of measuring the difference from the one probability distribution to a reference. After exploring a number of models, we settle on a comparison between two mixture models, which use differing tokenisation algorithms. We find that our alternative tokenisation algorithm gives the lowest KL-Divergence within our training dataset.

Finally, we find that, since many modern programming share keywords, it is hard for an LDA model to differentiate between them.

---

## Introduction

This project asked us to apply topic modelling in a non-trivial situation, ideally focused on a cybersecurity context. After brainstorming together, we decided to pursue modelling source code, aiming to use LDA to identify and group together code written in the same programming language. In this case, we consider the programming language to be the 'topics' of our model. The documents for our corpus will be Github repos, selected to contain a variety of programming languages. Although the programs will not necessarily be cybersecurity related, we are interested to see if and how successfully our LDA model can group/detect the programming language by only looking at the source code. We plan to measure the success of our method by using known keywords for each language, measuring how well the topics relate to these keyword sets via the Jacard index.

Implementations of the ideas discussed in this report are contained within the Jupyter Notebook "project/report.ipynb" . The report follows the structure of this document:

We have chosen the following **equity split**:

- Dan Jones - 33%
- Kishalay Banerjee - 33%
- Sam Harding - 33%

# Dataset

In an LDA analysis, the usual data set would consist of natural language documents. For our project, our data set is a collection of repositories (containing code in a number of programming languages) from Github. We have considered each repository to be an individual document, and the mixture of topics in that document are the different source code languages it contained. Details of how the data was generated is given in the next section.

## Building a Source Code Corpus

Despite the wide availability of open source code across the web, accessible via repositories like Github and Sourceforge, aggregated datasets suitable for use in machine learning are hard to find. An exception to this is the work by source{d} technology, who have released large archives of public git repositories[6] as well as open source tools for modelling source code in a more structured way [2]. Unfortunately, their ecosystem is primarily in the Go programming language, so proved hard to use for this particular project.

For our work we chose to generate our own data set of program source code, focusing on four key programming languages:

- Python
- R
- Javascript
- Shellcode

The Github API [1] was used to locate the most starred repositories in each of the above programming languages. The source code of each of these repositories was then downloaded, filtered to include only the file types that we are interested in, then saved into a CSV file.

We generated three datasets, each used in different parts of the project:

1. Minimal Dataset (dataset.csv.gz), consisting of the top 10 repositories for each programming language. This consists of $\approx 16,000$ source code files.
2. Completed Dataset (full_dataset.csv.gz), consisting of the top 250 repositories in each of the four categories. This proved too much data to work with on our hardware for most of our use cases, unfortunately.
3. Test Dataset (test_dataset.csv.gz), consisting of the 10th - 20th top repositories.

We used the file extensions of each source file to ensure we only included Python, Javscript, R and Shellcode in the dataset. Further, these file extensions were used to label each source code file in our testing dataset. Our implementation can be found in dataset.py, and usage details are available via it's "help" command line argument.

In order to model program mixtures, we combined the files in each repository to create a single data point. These documents now each contain a mixture of Python, Javascript, R and Shellcode. We then went on to label our test dataset by calculating the percentage of each programming language in each repository. This can be found under the "Generating the Dataset" section of the report notebook.

Due to difficulties in developing a labelled dataset for shellcode, we left it out of our models.

### Pre-processing Program Source Code for Topic Modelling and Alternative Splitting Function

We have chosen to model source code, which provides a unique set of challenges compared to natural language processsing (NLP). In particular, the process of cleaning and normalizing our dataset has a different set of requirements. We should carefully consider the applicability of different NLP techniques to source code. The processes of removing stopwords, as well as lemmatisation and stemming of inflected word forms in natural language shouldn't be universally applied across programs. Natural words such as "if" and "for" should not be neglected in a programming context, and words which are language identifiers like "finally" should not be shortened to "final" via stemming. Program sources are structured documents whose words have a different semantic meaning, depending on their position in text.

The task of pre-processing our data can be approached in two ways. Firstly, consider the case where the programming language is known. Here, we can make use of a language-specific parser to tokenise each document, allowing us to differentiate programming language keywords, identifiers and comments. We can therefore treat identifiers and comments as we would natural language, whilst preserving keywords verbatim.

Secondly, consider the case where the programming language is unknown. We are unable to parse each document before modelling, and so must treat it as unstructured text. Here, a trade-off must be made between normalising the data set and losing meaning. For example, if we remove stopwords, it is possible key language features such as the "if" keyword will be removed from modelling. Since our key aim is to identify the mixture of programming languages in a document, we are dealing with this case where we do not know the language in use before running our model.

We have two options here. We can either use the standard sklearn parsing function (which looks for strings of alphanumeric characters between punctuation), but this will not consider any potential non-alphanumeric identifiers nor ignore words in the comments of programs. Not being satisfied with this option, we alternatively considered writing our own bespoke parsing function, which would split the programs up more appropriately. We were able to write such a procedure which: removes comments in python, r and javascript; removes line breaks; splits the code string by spaces and opening brackets (in order to separate different codewords and pull out function names); and finally concatenate this information into a new document with spaces inbetween every new word. We can then tell sklearn to tokenize the document only by spaces, preserving our new parsing technique. This function code is written in the "Alternative Splitting Function" of the jupyter report.

## Latent Dirichlet Allocation

LDA is an example of a topic model. Topic modelling is defined as a type of statistical modelling for discovering the abstract "topics" which occur in a collection of documents. Specifically, LDA is a "three level hierarchical Bayesian model, in which each item of a collection is modelled as a finite mixture over an underlying set of topic probabilities" [3]. LDA was first proposed as a tool for research in population genetics, but currently, has found much wider application in the field of machine learning. For LDA to be applied, each 'document' is considered to be composed of a mixture of a number of different 'topics', and the topic distribution is assumed to have a sparse Dirichlet prior. Usually, LDA is applied in the field of Natural Language Processing (for example, identifying words and topics within documents written in English, French etc). However, for our project, we have re-purposed LDA to identify and differentiate between computer programming languages. The basic premise remains the same, but some steps in the usual analysis (especially in the context of data pre-processing) becomes significantly different. Those have been explained in more detail in the next sections.

### Interpretation of LDA Visualisations

The most common (and informative) way to interpret the results of an LDA is the LDAvis library in Python. It is used to visualise the fit of an LDA topic model to a corpus of documents. LDAvis requires 5 arguments as input -

1. The matrix of the estimated probability mass function over the terms in the vocabulary of each topic in the model.
2. The matrix of the estimated probability mass function over the topics in the model for each of the documents in the corpus.
3. The number of tokens observed in each document. Each token is a positive number.
4. The character vector containing the terms in the vocabulary.
5. The frequency of each term across the entire corpus. Each such frequency is a positive integer.

Four sets of visual elements can be displayed while using LDAvis -

1. Default Topic Circles - These circles represent each topic mentioned in the model for the LDA, and their areas are set to proportional to the proportions of the topics across the total number of tokens in the corpus.
2. Red Bars - Each bar represents the estimated number of times a given term is generated by a given topic. When a particular topic is selected, it displays a default number of relevant terms for that particular topic.
3. Blue bars - Each such bar represents the overall frequency of each term in the corpus.
4. Topic-Term Circles - These are circles whose areas are set to be proportional to the frequencies with which a given term is estimated to have been generated by the topics.

All the visual elements in LDAvis represent frequencies, rather than conditional probabilities. For example, a wider bar would signify more frequent terms in the training data, and vice versa.

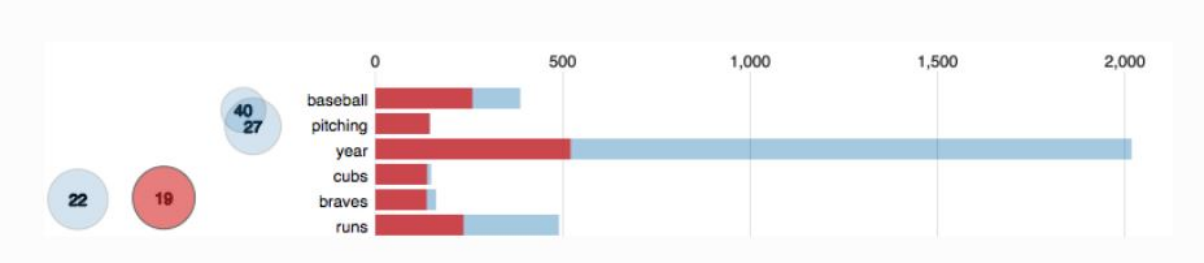


Figure 1: Screenshot of a Typical pyLDAvis visualisation

## Topic Modelling

A Topic Model is a kind of probabilistic generative model that has been used widely in the field of computer science, especially with respect to text mining and information retrieval. Topic modelling originated from Latent Semantic Indexing (LSI). However, there are subtle differences between the two, since, LSI is not a probabilistic model.The key idea behind topic modelling is that documents show multiple topics, and therefore the key question of topic modelling is discovering a topic distribution over each document, and a word distribution over each topic.[5]

In our project, we carried out three types of topic modelling -

For the first model, we carried out topic modelling on individual source files in our Github repositories. In this case, we considered each **document** (from an LDA context) to be **each individual source file** of a particular repository.

First the CountVectorizer function was used to convert the text to a matrix of token counts. In technical terms, this produces a sparse representation of the counts.

Then, we run the Latent Dirichlet Allocation model on our data, with the number of topics being the number of languages we are looking to identify. Finally, we use the pyLDAvis feature to obtain a visualisation of the model we just created. A screenshot of the topic distribution for this model, as generated by pyLDAvis, is provided below -

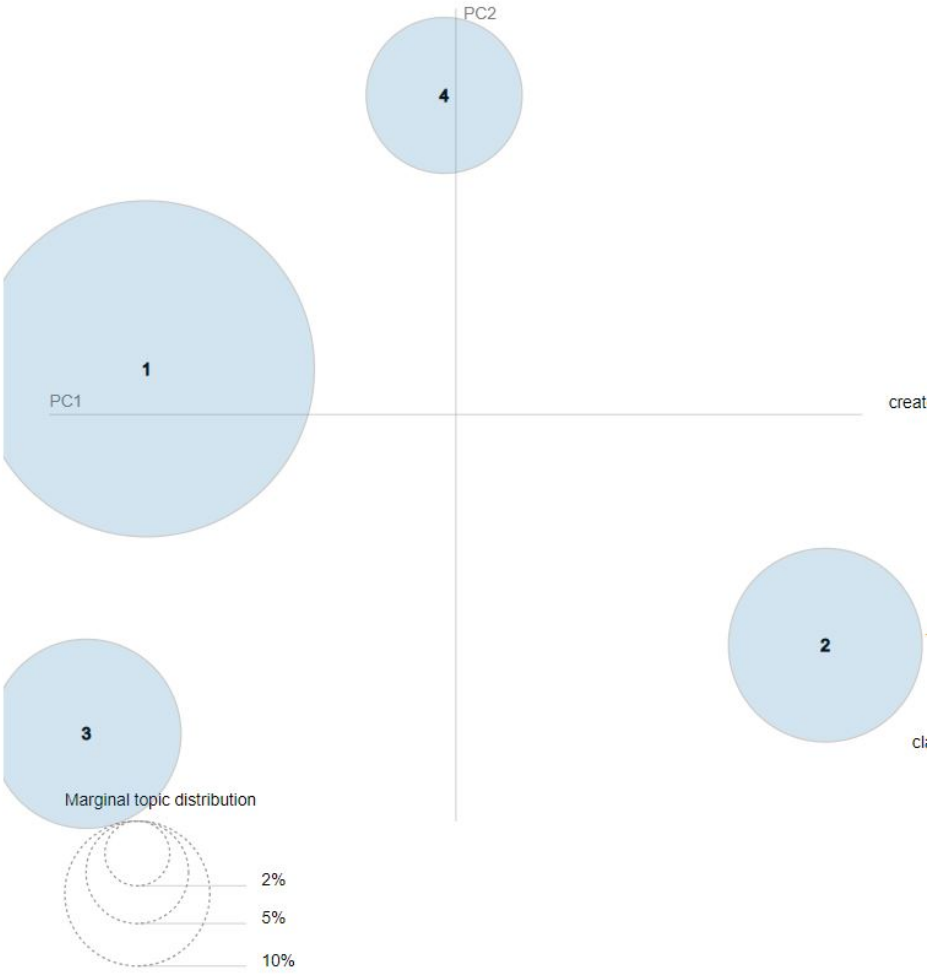


Figure 2: Screenshot of pyLDAvis visualisation of Individual Source File Modelling

Now, since LDA is an unsupervised learning algorithm, it does not tell us which topic corresponds to which programming language. So, our aim now is to map each of these topics to the programming languages. We use Jaccard's Index for this purpose.

We first extract the words displayed by pyLDAvis (according to a decreasing order of popularity) for

each topic. Then, we compute the Jaccard Index for the words for each topic and the list of keywords of each programming language, and store them in a dataframe.

Finally, we construct a heatmap of the dataframe to identify the mappings explicitly. A screenshot of the heatmap is provided below -

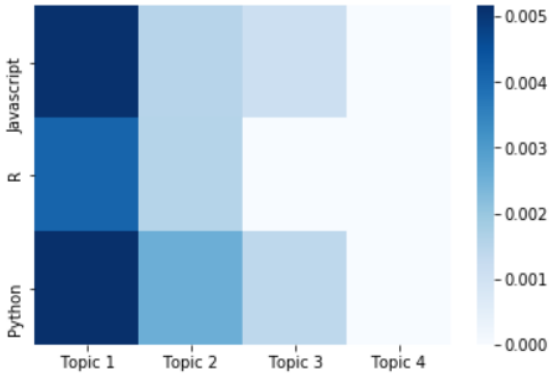


Figure 3: Screenshot of Heatmap for Individual Source Code Modelling

From the heatmap, it can be observed that Topic 1 cannot be clearly mapped to any one particular programming language. This may suggest that the LDA model has been unable to distinguish between some keywords of different programming languages satisfactorily.

Another interesting point we observe is that, the 4th topic seems to be composed entirely of numbers, and thus, does not correspond to any programming language keyword distribution whatsoever.

Note : This model was fully analysed on a smaller subset of the data set we generated, since, the full analysis of the entire data was beyond the hardware capabilities of our laptops.

The second topic model we consider is a type of mixture modelling, wherein, we combine the source code files in each repository into one large file. Thus, in this case, the equivalent of a **document in LDA** is the concatenated contents of an entire repository.

We work on the same subset of data as before. The main difference from the first model, as already indicated above, is that the files in each repository have been concatenated together using the default function in Python.

After the concatenation, we follow the same steps as before in tokenising the data, running the LDA model, creating a visualisation of the results using pyLDAvis, extracting the most popular keywords corresponding to each topic as suggested by the LDA, and computing the Jaccard Indices to map the topics to programming languages.

A screenshot of the pyLDAvis visualisation is provided below -

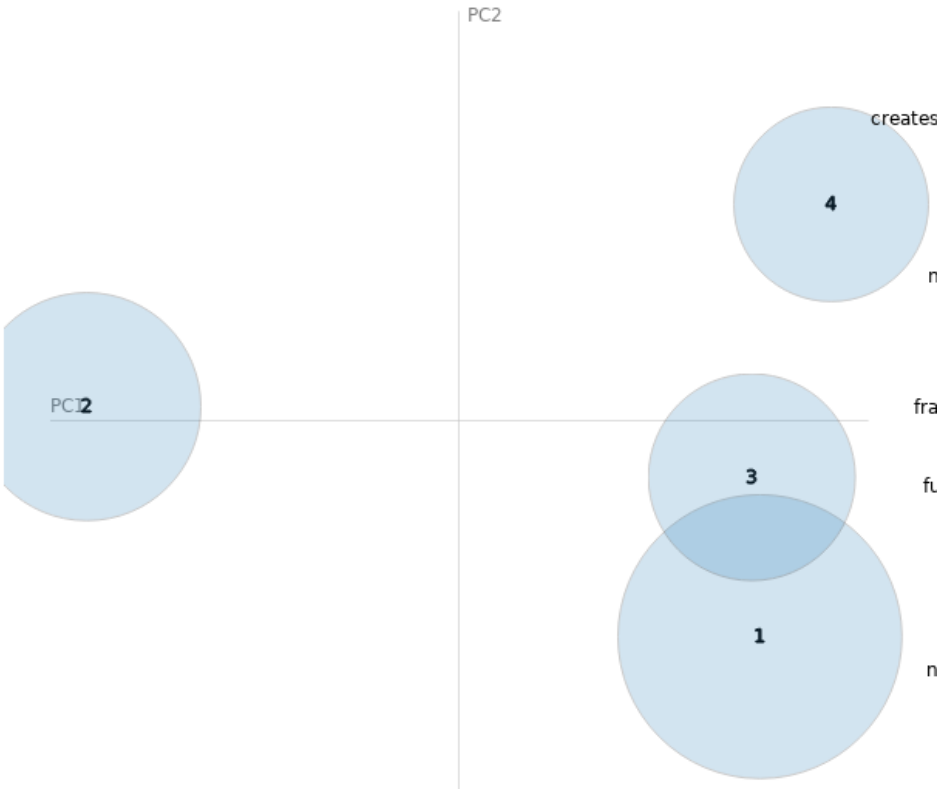


Figure 4: Screenshot of pyLDAvis visualisation for Mixture Modelling

Now, we provide a screenshot of the Heatmap obtained from the results of the Jaccard Indices.

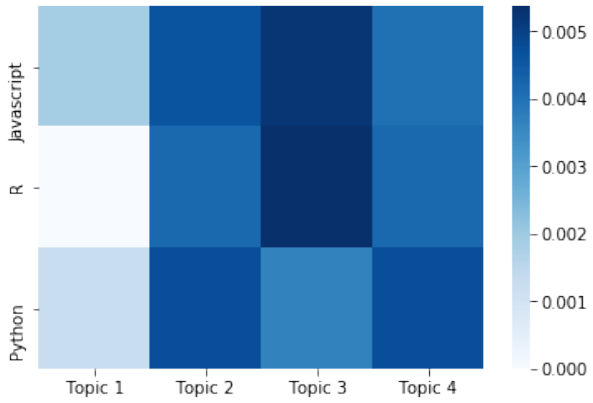


Figure 5: Screenshot of Heatmap for Mixture Modelling

From the Heatmap, we observe that -

- Topic 1 shows the best correspondence with Javascript.
- Topic 2 shows the best correspondence with Python.
- Topic 3 shows the best correspondence with R.
- Topic 4 shows the best correspondence with Python.

However, the distinction isn't extremely clear-cut in any of these cases. A reason for this could be, that, since the documents are a **mixture** of various programming languages, it does not do a good job of distinguishing between the keywords of all of them.

An additional metric we calculate for this model is the Kullback-Leibler Divergence. The KL Divergence is a measure of how one probability distribution is different from another probability distribution. It is also termed as Relative Entropy. Mathematically, it is defined as,

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \left( \frac{P(x)}{Q(x)} \right).$$

To implement KL Divergence, we first calculate the percentages of each programming language (R, Python or Javascript) in each document. This would constitute the actual probability distribution of languages in each document. Next, after running the LDA, we calculate the predicted percentages of languages in each document. That constitutes our second probability distribution. Thus, the value of KL Divergence calculated on these two "distributions" would give us a picture of how "close" or "far apart" these distributions are from each other. In other words, we would have an idea of how good the predictions of the LDA have turned out to be.

One of the potential problems in this method is that KL Divergence is calculated for each individual document. Thus, we have 2 lists of values of KL Divergence, and comparison of models thus becomes difficult. So, for each model, we have summed up the absolute values of the KL Divergence to get a single number for comparison.

The KL Divergence tables are provided later in the report.

The third model we considered is also a mixture model. The main difference from the previous mixture model is that, instead of using the default tokenisation algorithm, we developed our own tokenisation algorithm.

Natural Language Processing usually eliminates non-alphanumeric characters like brackets, colons etc, since they aren't considered to be important. However, they play an important role in the syntax of programming languages and may be invaluable as potential identifiers of a particular language.

Thus, we aim to use our own custom-built algorithm, in the hope that it cleans the data in a more programming language - centric manner. The implementation can be found in the "'Alternative Splitting Function" of the Jupyter notebook.
After we carry out tokenisation using this algorithm, the rest of the analysis is the same as the second model. Thus, we calculate KL Divergence for each repository in this model as well, and sum their absolute values to get a number suitable for comparison. As for the previous model, the KL Divergence values list is provided in a subsequent section of this report.

A screenshot of the pyLDAvis visualisation is given below -


Figure 6: Screenshot of pyLDAvis visualisation for the 2nd Mixture Model

Also, we now provide a screenshot of the Heatmap generated from Jaccard Indices of this model -

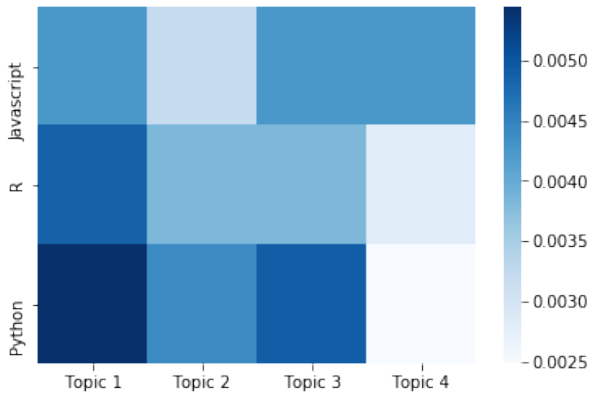


Figure 7: Screenshot of Heatmap for the 2nd Mixture Model

From the Heatmap, it can be observed that this model also does not provide a clear-cut distinction of languages. R cannot be satisfactorily identified in this Heatmap. However, with a bit of approximation, we can draw the following conclusions -
Topic 1 corresponds best with Python.
Topic 2 corresponds best with R.
Topic 3 corresponds best with Python.
Topic 4 corresponds best with Javascript.

## Identifying Program Subjects and Themes

As an aside, we decided to explore the potential for identifying the subject, or theme, of a program using LDA. This is more similar to the classic use of LDA in natural language processing. This was inspired

by the work performed by the Eclipse IDE plugin "'TopicXP" [7] which uses LDA to provide developers contextual information on the latent topics within the libraries they use.

A key modification to our existing models was required. Our previous models used a bag-of-words representation. This choice was important, since it ensured that common words (i.e. the keywords of the programming language), were given priority.

In contrast, when trying to identify subjects and themes, language keywords such as "'if" and "'then"' are not important. Instead, rarer words should be given priority. To do this, we use the term frequency–inverse document frequency (tfidf) statistic. This is the prodcut of two quantities, the term frequency and the inverse document frequency, computed as follows:

$$\text{term-frequency}(t, d) = \frac{\text{number of times t appears in document d}}{\text{number of terms in document d}}$$

$$\text{inverse-document-frequency}(t, d) = \log(\frac{\text{number of documents}}{\text{number of documents containing t}})$$

This better reflects the occurrence of words in natural language and should work well in our context. Since programming language keywords will appear in most documents, it should weight them lower.

The model was then enhanced further by considering known reserved words in a programming langauge as stopwords. The results of this model are summarised below:

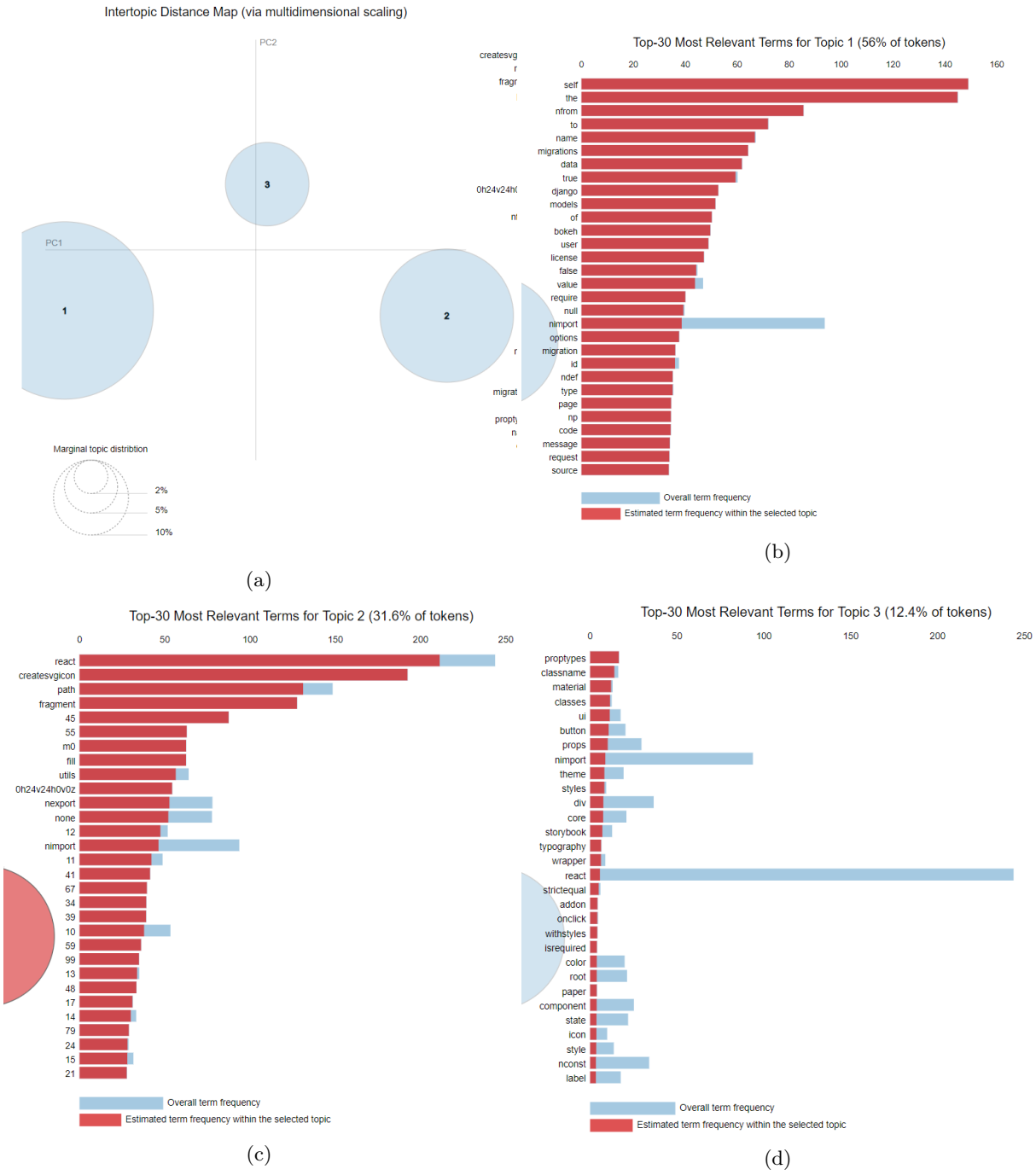


Figure 8: A summary of our second subject/themes model. It is hard to find meaning from the words in each of these topics.

## Mapping Unlabeled Topics to Programming Languages

The Jaccard Index is a statistic used for comparing the similarity and diversity of two sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection, divided by the size of the union of the sample sets:

$$\text{JaccardIndex}(A, B) = \frac{A \cap B}{A \cup B}$$

In modern data science problems, the MinHash locally sensitive Hashing Scheme is used to efficiently compute an accurate estimate of the Jaccard similarity coefficient of different pairs of sets, where each set is represented by a constant-sized signature derived from the minimum values of a hash function.

In our problem, we have used the Jaccard Index as a method of mapping the topics produced by the LDA, to the programming languages we wanted to identify.

To do this, we first created individual lists of common keywords of each programming language, sourced from the language specification manuals on the Internet.

Next, we considered the most common words corresponding to each topic, as obtained from the pyLDAvis visualisation. We tried to keep the list of common keywords and the list of words obtained from the LDA to be nearly similar in length.

Finally, considering a particular set of keywords (either Javascript, R or Python), we computed its value of Jaccard's Index with the words from each of the topic lists, and stored those values.

Finally, we considered that a particular topic corresponded to a particular language, if the value of the Jaccard Index of that topic was highest with that language.

We followed this technique for all the models we considered.

Finally, we created Heatmaps for better visualisations of the results.

# Conclusion

| Repo | KL Standard Parse | KL Alternative Parse |
|---|---|---|
| 596892 | 0.269011 | 0.662980 |
| 1248263 | 1.928442 | 0.254896 |
| 1790564 | 4.448496 | 0.704039 |
| 4751958 | 8.863396 | 8.586258 |
| 12465340 | 0.281198 | 0.267396 |
| 13523710 | 0.187485 | 0.738181 |
| 14267375 | 0.245021 | 0.338226 |
| 14579179 | 0.371502 | 13.374868 |
| 16146440 | 4.624400 | 3.921856 |
| 17856544 | 0.770869 | 0.046895 |
| 19117456 | 3.366443 | 0.119697 |
| 21289110 | 0.738135 | 0.176492 |
| 23932217 | 0.218933 | 0.372489 |
| 24929423 | 0.295016 | 0.211616 |
| 28556914 | 0.788294 | 4.626866 |
| 33614304 | 0.276975 | 0.703768 |
| 36849200 | 0.284211 | 0.551536 |
| 38226908 | 0.916786 | 0.172001 |
| 45936895 | 2.621928 | 0.509468 |
| 47918643 | 1.968439 | 0.190857 |
| 61412022 | 2.652152 | 0.179364 |
| 69798748 | 5.320805 | 0.014623 |
| 72671522 | 0.130721 | 0.418556 |
| 83222441 | 0.145351 | 0.258941 |
| 84232645 | 4.072407 | 0.549371 |
| 89187780 | 5.052083 | 0.165119 |
| 94911145 | 2.933363 | 0.122228 |
| 128624453 | 4.442435 | 0.711738 |
| Total | 58.214296 | 38.950323 |

A table showing the KL-Divergence between the predicted programming language mixtures, and the known programming languages mixtures.

We observe that our model using an alternative tokenisation algorithm, developed with the parsing of program source code in mind, performs better, when compared to the default algorithm provided by scikit learn (whose focus is mainly on natural language processing). However, we acknowledge that though there has been an improvement, there is still a considerable difference between the actual percentages of programming languages and the estimated values obtained from our mixture models.

## Fundamental Similarities Between Programming Languages

In our analysis, we found that LDA has difficulty distinguishing between programming languages. Topics generated by the model appear to contain large mixes of language words, as shown by the heatmaps we have generated. We hoped for very distinct peaks indicating a strong relationship between topics and languages, but instead we get a very blurred and mixed result.

One possible for this is that the languages we have chosen to model are fundamentally very similar. There is large overlap in the keywords and syntax, meaning that the LDA cannot always detect differences by these words alone. This close relationship can be seen in the following heatmap displaying the Jacard idexes between the three sets of keywords:

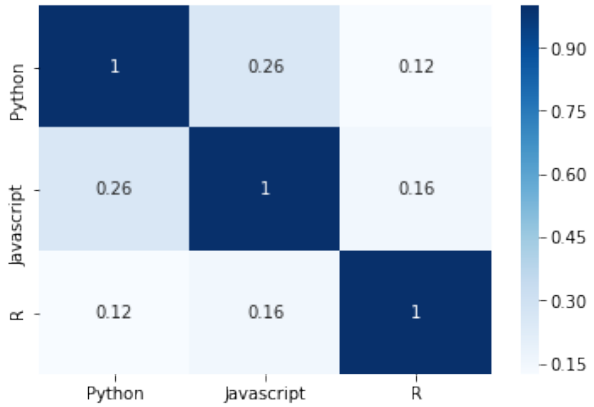


Figure 9: A heatmap showing the Jaccard Index between our programming language keywords.

This shows, for instance, that Javascript and Python share roughly 26% of their keywords, indicating that it might be hard to differentiate between these languages, especially if the overlapping list of keywords are very common phrases (such as ”if” and ”for”). This means that our task was flawed by our choice of languages, and as a result the LDA models we generated are simply not suitable to model and differentiate between such similar programming languages.

## Application to Cybersecurity

Even though the traditional use of LDA has been mainly in Natural Language Processing, it can also be harnessed to perform in cybersecurity contexts, like the search and identification of shell code in program files. This is quite a critical application, since, most malware which infects systems tend to enter through attachments of dodgy emails and spam. Usually, in these cases, the dangerous code is placed within a seemingly innocent looking program, which often helps it to be overlooked.

However, the point remains that the structure and syntax of the shell code is different from that of normal program files.

Thus, if an LDA could be trained to recognise these differences (i.e, the shell code and the normal code could be considered as two different ”topics”, in agreement with LDA jargon), it could potentially be highly useful in eliminating a large number of such cases automatically, without actual human intervention every time.

But the fact remains that for this to work, the LDA would have to be trained on a data set large enough for it to capture all the types of differences accurately. In addition to this, it would also have to ensure that legitimate emails were not flagged as malware, i.e, it would have to ensure a low False Positive Rate (where the flagging of malware is considered to be a positive).

```
    sys.exit(0)

proc = sys.argv[1]
WMI = GetObject('winmgmts:')
p = WMI.ExecQuery('select * from Win32_Process where Name="%s"' %(proc))
if len(p) == 0:
    print "Process " + proc + " not found, exiting!"
    sys.exit(0)

process_id = p[0].Properties_('ProcessId').Value

shellcode = \
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64" \
"\x8b\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e" \
"\x20\x8b\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60" \
"\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b" \
"\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01" \
"\xee\x31\xff\x31\xc0\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d" \
"\x01\xc7\xeb\xf4\x3b\x7c\x24\x28\x75\xe1\x8b\x5a\x24\x01" \
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01" \
"\xe8\x89\x44\x24\x1c\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89" \
"\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45" \
"\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff" \
"\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33\x32\x2e\x64" \
"\x68\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89\xe6\x56" \
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24" \
"\x52\xe8\x5f\xff\xff\xff\x68\x58\x20\x20\x20\x68\x20\x50" \
"\x4f\x43\x68\x63\x74\x6f\x72\x68\x49\x6e\x6a\x65\x68\x6f" \
"\x64\x65\x20\x68\x6f\x6e\x20\x43\x68\x50\x79\x74\x68\x31" \
"\xdb\x88\x5c\x24\x18\x89\xe3\x68\x72\x67\x58\x20\x68\x6e" \
"\x61\x2e\x6f\x68\x6f\x72\x74\x75\x68\x72\x65\x61\x66\x68" \
"\x2e\x61\x6e\x64\x68\x2f\x77\x77\x77\x68\x70\x73\x3a\x2f" \
"\x68\x20\x68\x74\x74\x68\x72\x67\x20\x2d\x68\x6e\x61\x2e" \
"\x6f\x68\x6f\x72\x74\x75\x68\x72\x65\x61\x66\x68\x40\x61" \
"\x6e\x64\x68\x64\x72\x65\x61\x68\x2d\x20\x61\x6e\x68\x75" \
"\x6e\x61\x20\x68\x46\x6f\x72\x74\x68\x72\x65\x61\x20\x68" \
"\x20\x41\x6e\x64\x68\x64\x20\x62\x79\x68\x6c\x6f\x70\x65" \
"\x68\x64\x65\x76\x65\x68\x64\x6c\x79\x20\x68\x50\x72\x6f" \
"\x75\x31\xc9\x88\x4c\x24\x5e\x89\xe1\x31\xd2\x52\x53\x51" \
"\x52\xff\xd0\x31\xc0\x50\xff\x55\x08"


process_handle = windll.kernel32.OpenProcess(0x1F0FFF, False, process_id)
```

Figure 10: Python source code with a malicious payload embedded [4]. As future work, could we use LDA to identify program mixtures like the above?

# Future Work

As an unsupervised learning model, LDA can discover underlying topics in unlabeled data. Nevertheless, "topics" discovered in an unsupervised way may not match the true topics in the data. Therefore, many researchers modified LDA in a supervised learning manner, which can introduce known label information into the topic discovery process. Examples of typical supervised topic models include supervised LDA (sLDA), the discriminative variation on LDA (discLDA), and maximum entropy discrimination LDA (medLDA). A multilabel topic model called labelled LDA (LLDA) extends previous supervised models to allow for multiple labels of documents, and the relation of labels to topics represents one-to-one mapping. Another variant named partially labelled LDA(PLLDA) further extends the concept of LLDA to have latent topics not present in the document labels.

A popular extension of these models is a hierarchical topic model(hLDA). hLDA is an unsupervised hierarchical topic modelling algorithm that is aimed at learning topic hierarchies from data. The need for this model arose from real world applications, where it was observed that topics have correlations among them.

# References


[1] Github developer api, https://developer.github.com/v3/. .

[2] source{d} technology open source https://sourced.tech/open-source/. .

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[4] Andrea Fortuna. Code injection on windows using python, https://www.andreafortuna.org/programming/code-injection-on-windows-using-python-a-simple-example/. .

[5] Lin Liu, Lin Tang, Wen Dong, Shaowen Yao, and Wei Zhou. An overview of topic modeling and its current applications in bioinformatics. *SpringerPlus*, 5(1):1608, 2016.

[6] Vadim Markovtsev and Waren Long. Public git archive: a big code dataset for all. *CoRR*, abs/1803.10144, 2018.

[7] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk. Topicxp: Exploring topics in source code using latent dirichlet allocation. In *2010 IEEE International Conference on Software Maintenance*, pages 1–6, Sep. 2010.