Topic Modelling U.S. Court Cases with High Performance Python

Abdul Z. Abdulrahim

1 Introduction

In this paper, I use Latent Dirichlet Allocation (LDA) to implement a topic model of the U.S. Court Opinions to discover the abstract topics that occur in the collection of documents. I begin by outlining the approach I use to implement the topic model. This includes a brief introduction to LDA and the process required for its implementation. I then discuss the various approaches to optimisation I use to improve the performance of the implementation. These include parallel processing, a method for running processes on multiple CPUs locally and distributed computing, which runs processes across clusters of CPUs that are each treated as computers.

After going through the implementation and various options for optimations, the results show that while significant performance gains can be made from the use of some optimisations methods for computational limitations, we also encounter limitations on memory, cost and networks that can equally hinder improvements to the implementation. This presents a reflection of the issues that will be encountered when working with large datasets such as the U.S. court documents. With over 6 million cases, the U.S. case database provides an opportunity to extract insights on the judicial branch of the country, but processing data at this scale is a difficult task. As a result, considerations must be made on whether the methods of each optimisation is useful in a practical setting where not just the speed of the implementation matters but also the cost and time taken to complete the project as a whole.

In the final section of the paper, I will evaluate the full implementation of the code on the state of Arizona and discuss the results obtained from the model. We see that the LDA model requires an iterative process to obtain insightful results, and as such, it is indeed beneficial to have an implementation that is optimised effectively.

2 Topic Modelling

A large body of text articles is generated every day from various sources, which creates a key challenge for natural language processing i.e. effectively managing, searching and categorising documents by their subjects or themes. These text mining tasks usually include text clustering, document similarity and categorisation of text to find out what themes could be extracted. In text analytics, this is known as Topic Modelling, i.e., given a topic, can we find other documents of text which are similar to it (Olsen, 1993). Therefore, this is a type of statistical modelling for discovering the abstract *topics* that occur in a collection of documents to derive meaningful information (Olsen, 1993; Blei, Ng, & Jordan, 2003).

2.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an example of a topic model used to classify text in a document to a particular topic. It builds a topic per document model and words per topic model, modelled as Dirichlet distributions. The unsupervised probabilistic model is then used to discover underlying themes in a document. LDA assumes that: (i) documents are produced from a mixture of topics and each document belongs to each of the topics to a certain degree; and (ii) each topic is a generative model which generates words of the vocabulary with certain probabilities. To produce a model, we input a *document-term matrix* which keeps the histograms of words (word count) present in each document, and this returns an algorithm of smaller matrices, i.e., a document to topic matrix and a topic to word matrix (Blei et al., 2003; Řehůřek & Sojka, 2010). The Document-Topic matrix accounts for the probability distribution of the topics present in the document. Similarly, a Topic-Words matrix accounts for the probability distribution of words that have been generated from that topic. These are initialised randomly, and then these distributions are improved upon in an iterative process to reach an approximately steady state where these distributions seem logically correct (Olsen, 1993).

2.2 LDA Implementation

The code I prepare to implement the LDA model requires the following data wrangling and pre-processing steps:

- Wrangling: Extracting and parsing through court documents to prepare a clean text

of opinions released by the court for a case.

- Tokenising: Splitting the text into sentences and the sentences into words, then lowercasing the words and removing punctuations.
- Standardising: All words that have fewer than two characters are removed; stopwords are removed; words in the third person are changed to first person and verbs; and future tenses are changed into present and words are reduced to their root form.

Following the steps above, the prepared data is fed into a modelling module from the gensim library. Gensim (Generate Similar) is a free Python library designed to extract semantic topics from documents efficiently (Řehůřek & Sojka, 2010). It is designed to process raw unstructured digital texts (plain text) using various algorithms and includes an LDA modelling module. In terms of the limiting behaviour of the functions/methods used in this paper, i.e. the complexity or processing time, the wrangling of the data approximates to $\mathcal{O}(n)$ time and the LDA approximates to $\mathcal{O}(n * v * k)$ — where n is the number of documents, v is the number of features and k is the number of words. The aim is to therefore significantly reduce the processing time by a magnitude that changes its $Big - \mathcal{O}$ notation. Before implementing the process described above, I outline the optimisation methods used to improve the modelling process.

3 Optimisation Methods

To optimise the preparation and modelling of the topics, I use a variety of single processing, multiprocessing and distributed computing approaches at different stages of the topic modelling process. This is compared and contrasted with alternative approaches that could be used, evaluating the type of computational bottlenecks, the difficulty of implementation, and memory considerations, as well as other limitations encountered.

3.1 Parallel Processing

Parallel processing is a technique for simultaneously breaking up and running program tasks on multiple processors, thereby reducing processing time (Lanaro, 2013). In general, this is achievable by using multiple processors. This approach is necessary to tackle large-scale problems such as the topic modelling I implement in this paper. With 6.4 million unique

cases tried in the U.S., computing a model using a single processing approach, where each task is completed sequentially, increases the time required to complete a model. Parallel processing tackles two types of bottlenecks — I/O Bound and CPU Bound (Lanaro, 2013).

3.1.1 I/O Bound or CPU Bound

CPU Bound means the speed of the CPU limits the rate at which process progresses. A task that performs calculations on a set of numbers, for example multiplying small matrices, is likely to be CPU bound. I/O Bound, on the other hand, means the rate at which a process progresses is limited by the speed of the Input/Output subsystem. A task that processes data from disk, for example, counting the number of lines in a file is likely to be I/O bound. Multithreading, a type of parallel processing, is useful in Python to address I/O Bound programs. However, in this case, the bottleneck is computational, so threads are not useful for optimisation. Python includes a Global Interpreter Lock (GIL) that ensures that only one Python instructor executes at a time. This is useful to ensure the sequence of code (Lanaro, 2013). However, this means that even with multiple threads only one command executes at a time. While this is fine for I/O operations where the thread can release the GIL while waiting, it does not address computational bottlenecks.

3.1.2 Multiprocessing

To achieve meaningful optimisation, I instead use multiprocessing. Multiprocessing avoids the GIL because each process has its own Python interpreter. This means each process requires its memory, which in turn means there is overhead in starting/stopping processes and communicating data between them (Lanaro, 2013). As a result, it is only beneficial to multiprocess when the data spread across the CPU units is large enough to warrant the overhead, and communication between processes is small enough. Communication between processes is costly and can seriously hinder the performance of parallel programs. In this distributed memory model used for multiprocessing, each process is completely separated from the others and possesses its own memory space. The communication overhead is costlier compared to the shared memory approach in multithreading, as data can potentially travel through a network interface, but multithreading cannot overcome a CPU bottleneck (Lanaro, 2013).

Lastly, for the multiprocessing approach to be useful, we have to ensure our implemen-

tation is embarrassingly parallel — can be split into chunks and executed independent of one another without affecting our results — which is the case here.

3.1.3 Memory Bound

Another important limitation to also introduce now is a Memory Bound program, which means the rate at which a process progresses is limited by the amount of memory available and the speed of that memory access (Gorelick & Ozsvald, 2014). While the multiprocessing techniques I use can significantly speed up certain processes for the modelling, in others, the large amount of memory required for the process creates another bottleneck that can only be avoided by acquiring more memory or other techniques such as distributed computing. It points to a cost-benefit analysis that must be considered when optimising processes as although the distributed computing approach might significantly improve the speeds of the computation; it requires a further implementation that may be limited by time and cost.

3.2 Distributed Processing

3.2.1 Distributed Systems

A distributed system is simply a group of computers working together as to appear as a single computer to the end-user (Lanaro, 2013). This usually means assigning one computer as the *dispatcher* who assigns the tasks and compiles the results, and *workers* who complete the processes. These machines have a shared state connected over a network, operate concurrently and can fail independently without affecting the whole system's up-time. The primary reason for using distributed computing is to run processes faster. When computers are in a distributed system, there is no requirement for them to be physically close together, but the network connecting them is key. It could either be over a local network for physically close computers, or a wide area network if they are geographically distant. A distributed system may include any number configurations, such as mainframes, personal computers, workstations, minicomputers, but they all work as a single computer.

A distributed system offers scalability as it can easily be expanded by adding more machines as needed, and redundancy as several machines can provide the same services, so if one fails, the processes do not stop. The challenge, however, is that the cost of communication increases as we distribute data and computational tasks across a network. Network transfers are extremely slow compared to the processor speed, and when using distributed

processing, it is even more critical to keep network communications as limited as possible. There are strategies to get around this such as processing the data locally and only transferring data when strictly necessary. As a computing cluster may contain hundreds or thousands of machines, probabilistically speaking, faulty nodes become very common. For this reason, distributed systems need to be able to handle node failures appropriately and without disrupting the ongoing work. Various packages aid in managing these issues and thus for this paper, I use the gensim library, which includes a distributed option and Pyro to maintain communication between nodes.

3.2.2 Gensim Library

In gensim, most of the time-consuming processes are done in low-level routines for linear algebra, inside NumPy, independent of any gensim code. Using the fast BLAS (Basic Linear Algebra) library for NumPy can also improve topic modelling performance significantly (Řehůřek & Sojka, 2010). Thereby, in most cases, a distributed approach is not necessary for optimisation, especially when the costs of acquiring additional computers is considered. In this instance, as I have access to a private server, it is worth evaluating the additional speed gain because the size of the dataset available means most personal computers are unlikely to be able complete a model of all U.S. cases. For communication between the nodes, gensim works well with Pyro (Python Remote Objects). It is a library for low-level socket communication and remote procedure calls in Python. Pyro4 is a pure-Python library, so its installation and use are relatively straight-forward. It simply requires copying the files onto Python's import path.

4 Data

4.1 Caselaw Access Project (CAP)

CAP is a Harvard Law School Library Innovation Lab Project that has collated all official, book-published United States case law — every volume designated as an official report of decisions by a court within the United States (Library, n.d.). It includes all state courts, federal courts, and territorial courts for American Samoa, Dakota Territory, Guam, Native American Courts, Navajo Nation, and the Northern Mariana Islands and dates back to 1658. Each volume has been converted into structured, case-level data broken out by the majority

and dissenting opinion, with human-checked metadata for party names, docket number, citation, and date. It, however, does not include new cases as they are published but is up to date as of June 2018.

As the data is categorised by state, we can access 52 XML dumps which provide data for each state. The states used for the analysis are selected at random and will not cover all 52 states as we only require a sample to demonstrate the bottlenecks encountered, as well as potential optimisations that may be made — and the limitations that should be kept in mind. To that end, the implementation below uses the cases for the states Arizona, District of Columbia, Illinois and Massachusetts as a sample set.

5 Implementation Results

To simplify the modelling and the parallelisation of various bottlenecks in the code, I have defined the process using two classes, `CaseWrangler()` and `TopicModel()`. These encompass the whole operation required to produce an LDA model for the opinions extracted from the cases. As I proceed, I outline each section of the code, the profiling results and the optimisations implemented.

5.1 Case Wrangling

First, we look at the `CaseWrangler()`. This class contains two functions that are intended to: (i) extract the cases from the XML folder containing as zipped file of the cases (`extractCases()`); and (ii) wrangle the case to create a dataframe containing the flattened opinions and relevant meta-data (`wrangleCases()`) (see Appendix for full implementation).

5.1.1 Extract Cases

As the cases for each state is provided in zip files, it restricts the potential in-file parallelisation possible for the sample set. Hence, we run each state file's extraction and de-compression independently — and for multiprocessing, on separate workers — and compile them at the end of the process (with the option to serialise the output).

(a) Single Processing

On running the function to extract cases for our sample set using the single processing approach, it took a total time of `1:59 minutes`. Profiling this function shows that an

average of `0.0003 seconds` is required per case and a bulk of the time is spent in the loop used to extract the files (see Appendix).

(b) Multiprocessing

To parallelise the function, I use the `ProcessPoolExecutor()` from the `concurrent.futures` library which provides a high-level interface for asynchronously executing callables. The resulting total time increases to `2:41 minutes` (see Appendix). Initially, this may appear to be an issue with the initialisation of the multiprocessing module adding overhead to the process — which is partly the case. However, line profiling the code (using `cProfile`) shows that a majority of the time is spent on the decompression and reading of the files. So, it does not appear that parallelisation adds any noticeable overhead.

Further profiling of the memory using `memit` shows that the parallelisation is memory bound. Using a local computer with an available RAM of `8.5 GiB`, the increment of `7.8 GiB` slows the overall process down — as well as other processes running on the computer —, thereby making it less efficient than using a single process approach. This does not come as a surprise as the zipped files collectively amount to `938.8 MiB` which unzipped increases by five to ten-fold. In essence, the multiprocessing approach only yields results if there is enough memory to compute the process efficiently.

Figure 1 depicts a comparison of memory usage as the processes are run. The black lines in the figures, which sum up the total RAM used as the processes are run, show that multiprocessing requires up to three times the amount of memory used to run the single process because each state's file is unzipped simultaneously. Thus explaining why all processes were slowed down.

An alternative approach to this issue could be using a distributed computing optimisation but as discussed earlier, the additional time and cost of implementation do not always merit its use. In this case, as there are further bottlenecks to optimise, I will not implement it at this point.

5.1.2 Wrangle Cases

To wrangle the extracted cases into text that can be prepared for modelling, the `wrangleCases()` method parses through the files to extract the court name, jurisdiction of case, name of

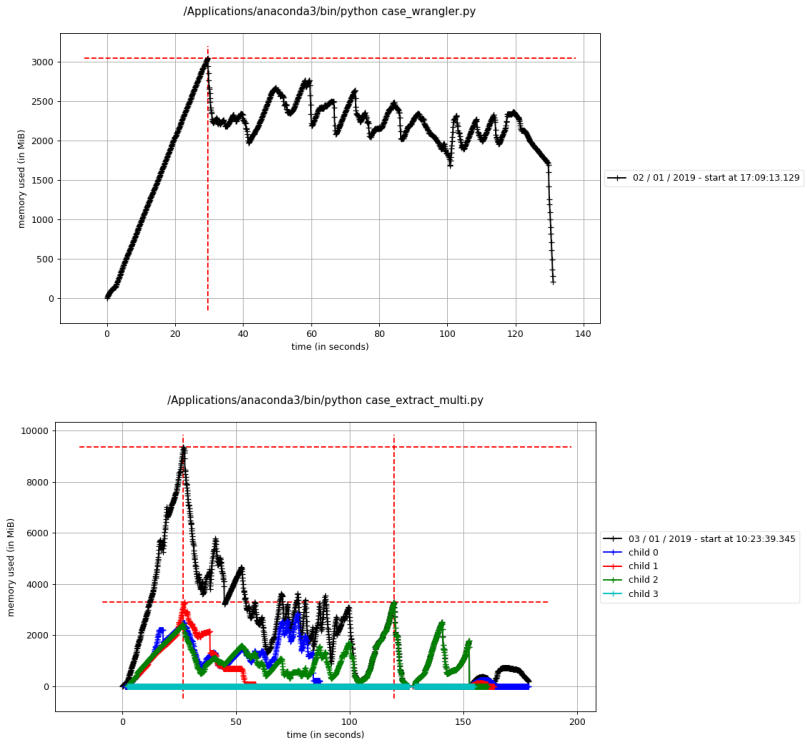

Figure 1: Memory Usage Comparison for Single Processing and Multiprocessing

the case, U.S. citation for the case, date the case was decided and the opinion issued by the judge for the case. These are then flattened into a dataframe that is used to create a text corpus before topic modelling. The full outline of the function used is shown in the Appendix. For the wrangling section, we only need a slice of the whole list created from the previous section. Therefore, I will use the cases for Arizona, which allows us to test the speed and optimisation techniques on a sufficiently large set of `28,182 Cases` without having to loop through all `337,386 Cases`.

(a) Single Processing

Using the single processing approach, it takes roughly `10.3 seconds` to loop through the Arizona cases. This gives us an average time of `0.0004 seconds` per case. We do not encounter a similar memory bottleneck in this optimisation as the incremental memory used for the single processing comes to `164.9 MiB`, which indicates the multiprocessing is unlikely to use up the maximum RAM available.

(b) Multiprocessing

As with the previous section, I use `ProcessPoolExecutor()` to parallelise the `wrangleCases`

`()` method and as expected, we had no memory bottlenecks (incremental memory used, `902.22 MiB`). To achieve optimal speeds for the process, I vary the `max_workers` and `chunksize` variables to identify any improvements in performance. `max_workers` determines the number of processes used, and `chunksize` determined the number of documents or chunks used per processor. The results of the tests are shown in figures 2 and 3 below.

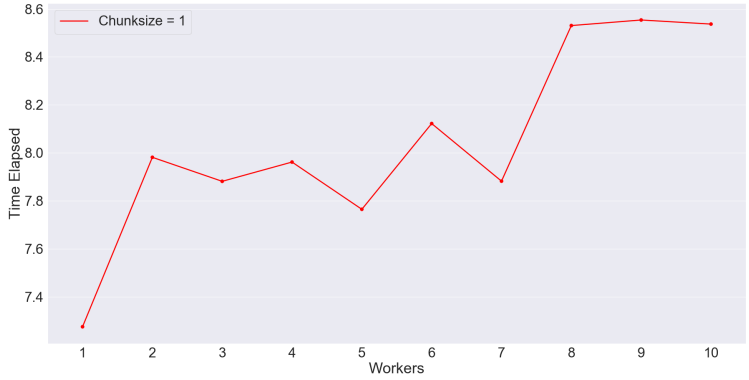

Figure 2: CW.wrangleCases Optimisation (Workers)

Figure 2 shows the changes in speed as the number of workers is increased from `1 to 10` at `chunksize = 1`. We see that initially using one core achieves the fastest speed with `7.3 seconds`. Afterwards, the speed varies moderately until we go past seven workers at which point, the cost of communication across the distributed memory starts to increase the total time.

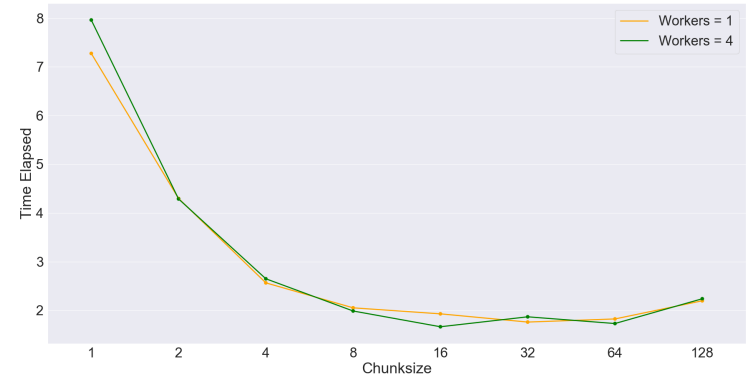

Figure 3: CW.wrangleCases Optimisation (Chunksize)

Figure 3 depicts the changes in speed as we vary the chunksize for one worker and four workers. From the tests, the optimal combination was `max_workers = 4` and `chunksize = 16` which achieved a total time of `1.7 seconds` — over a five-fold decrease in total time from the single processing approach.

5.2 Topic Modelling

The second class created for the model is the `TopicModel()`. As outlined in the pseudo-code below, this class contains four methods that are intended to finalising our model. I noted in the earlier section the steps to completing a model, so each function below serves the purpose of completing the remaining pre-processing steps and creating an LDA model using the gensim library (full implementation can be found in the Appendix).

```
class TopicModel():
    def createCorpus(self, corpuspath, dataframe):
        #Create corpus from dataframe and dump them into folder
        #Read contents into a list and randomly sample them
        #Using methods from NLTK to clean articles
        return docs

    def cleanWords(self, docs):
        #Split and clean words and return as clean documents
        return clean_docs

    def filterWords(self, clean_docs):
        #Filter words for stopwords using NLTK
        #Convert dictionary into Document Term Matrix
        return filtered_dict, term_matrix

    def ldaModel(self, filtered_dict, clean_docs):
        #Create the LDA model using gensim library
        return ldamodel
```

5.2.1 Create Corpus

In the same way classes and methods were created for the data wrangling, the topic modelling class has methods that will be evaluated separately to identify resolvable bottlenecks. First, we review the `tm.createCorpus()` method.

(a) Single Processing

Using the default single processing approach, it took roughly `14.7 seconds` to create a corpus and load it into a list for the next process. As parts of the function inherit methods from the NLTK library, line profiling the code does not produce interpretable results. This is an issue with inheriting classes and methods that will come up again in the later sections of this paper. As for memory, we can see using `memit` that the process uses an incremental memory of `725.2 MiB`.

(b) Multiprocessing

Given the method is not memory bound, I use the `ProcessPoolExecutor` to optimise it (as above) to see the best combination of workers and chunksize. Figure 4 shows the changes in speed as the number of workers is increased from `1 to 10` with a `chunksize = 1`. Although we do not get significant increases in performance, it appears using `max_workers = 4` achieves the best total run time of `12.7 seconds`. Increasing the number of workers thereafter only causes a loss in time performance. Interestingly, it appeared that multiples of 4 achieved the best runtime in any interval of four, i.e. 4, 8, 12 and 16.

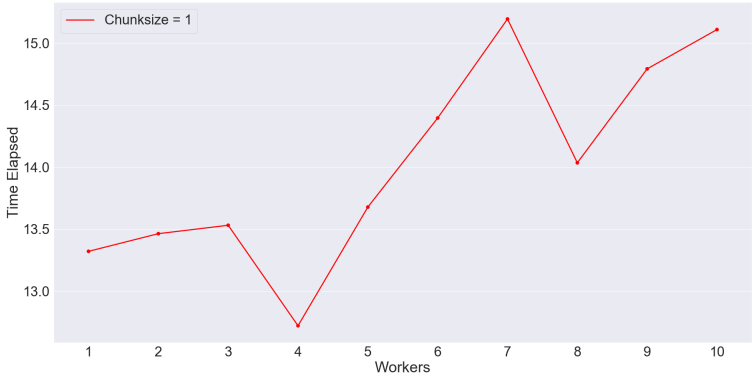

Figure 4: TM.createCorpus Optimisation (Workers)

Figure 5 shows optimal speed as we vary the chunksize for 4 workers over the range `1 to 1024`. The results, while only marginal, achieve the best result with `max_workers = 4` and `chunksize = 256`, which amounted to a total time of `11.4 seconds`. Because we use an external library (NLTK) in this step, it is difficult to diagnose what causes the

limited improvement and no less resolve the issue without going into the source code. Nevertheless, the 21 per cent gain in performance from a single processing approach will make a significant difference when a larger set of cases is used.

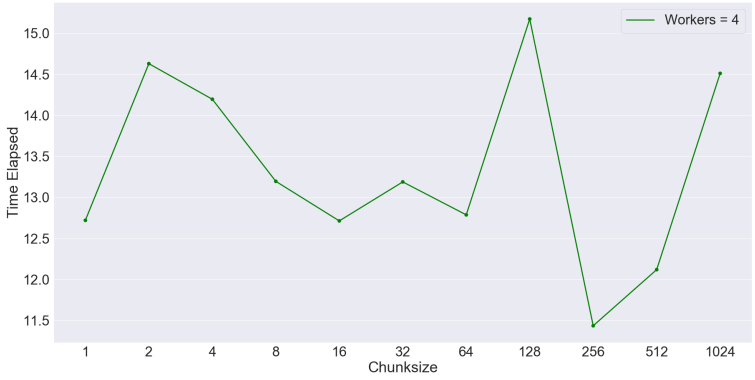

Figure 5: TM.createCorpus Optimisation (Chunksize)

5.2.2 Final Preprocessing

In the final stage of the preprocessing, the term dictionary and term matrix is prepared for the model. It uses a variety of methods from the NLTK and gensim libraries to prepare the data used for the model. As indicated earlier, the use of methods from external libraries limits our ability to diagnose or improve bottlenecks in the code. For this reason, no improvements in performance where achieved at this stage (see Appendix). Using a single process approach, the total time used for the process comes to `3:07 minutes`, compared to a total time of `3:11 minutes` when parallelised. Furthermore, varying the number `workers` and `chunksize` used did not change the perfomance of the process.

This highlights another important point to note when evaluating various approaches to optimisation. While it may be relatively easy to use an pre-assumbled library for some of our processes, they are not always easily parallelised. Moreover, it becomes far more complicated attempting to diagnose the problem as it requires, extensive code review. An alternative approach would be to define the functions/methods ourselves, but this may complicate the process further considering that the code required is quite complex and requires more time to implement. Given that this may be part of an ongoing project, using user-defined code adds a different kind type of implementation overhead, which may require frequent updating if inputs change or the scope of the project changes.

5.2.3 LDA Model

As outlined earlier, the gensim module allows both LDA model estimation from a training corpus and inference of topic distribution on new, unseen documents (Řehůřek & Sojka, 2010). The algorithm is streamed so training documents may come in sequentially. It also runs in constant memory *w.r.t.* the number of documents, so the size of the training corpus does not affect memory footprint, and it can process corpora larger than RAM. Lastly, the implementation may be distributed over a cluster of machines, if available, to speed up model estimation.

In this section, I implement a single process, parallelised and distributed approach to determine the best approach.

(a) Single Processing

The implementation using single processing is relatively straightforward. Having, obtained our term dictionary and matrix, this can be inserted into the function/method. Gensim's default setting creates a model by passing over the documents once, generating `100 topics` and uses a `chunksize` of `2000 documents` which updates the model after each document. Performance can be improved using the single processing model by changing the `chunksize` and decreasing the frequency with which the model updates. However, in this section, I aim to compare the different approaches using the same default settings. So I will not vary `chunksize` here. In our single processing approach, the total run-time comes to `5:38 minutes`. As with the preprocessing, because this is an external library, we are unable to perform line profiling on it. However, we check the amount of memory it uses, which comes to `1.4 GiB` and this poses no issues for the CPU. This is mainly because of the way the LDA module is deisgned in gensim. By running on constant memory, we find that the incremental memory used does not vary much regardless of the approach used.

(b) Multiprocessing

Implementing the multiprocessing approach using the `ProcessPoolExecutor` and 3 `workers` , we see that the run-time decreases slightly to `4:44 minutes`. Using memory profiling, we confirm that the memory footprint more or less stays the same (incremental memory usage, `1.4 GiB`). The performance using `ProcessPoolExecutor` can be improved by increasing the chunksize, but as we aim to compare like-with-like, this is not shown in

this section (see Appendix for time variation with chunksize). What we see from this approach is that the overhead needed to parallelise the process means that it does not operate as efficiently as we would want it.

The gensim library also comes with a pre-assembled multiprocessing module for the LDA. Using this module, we can see further improvements which reduces our total run-time to `3:10 minutes`. Clearly, gensim's parallelisation approach works more efficiently than the native multiprocessor. Varying the chunksize in this case also improves performance as with the `ProcessPoolExecutor`.

(c) Distributed Processing

Finally, I use the distributed processing approach to prepare the model. It required a more tedious set up compared to the single and multiprocessing approach as multiple clusters need to be instantiated on the server and selecting an appropriate medium for the clusters to communicate effectively can be challenging. A brief outline of the setup in psuedo-code is shown below (see Appendix for full implementation).

```

    #For each computer in the cluster, run the in the command line
    $ install gensim distributed
    $ export Pyro Serialisers

    #Run Pyro name server on one of the machines
    #Select the workers in the cluster and assign them
    #Select the dispatcher and assign it in the command line

    #Now our cluster is set up and running, ready to accept jobs
    $ Run the LDA in the command line of the dispatcher
    $ Indicate that there are clusters available

```

The results of the distributed processing were varied depending on the strength of the network connection. On a home computer with three clusters as workers, the total run-time came to `0:05:18 minutes`, edging out the single processing approach. Replicating the process on a server with three clusters, however, resulted in a run-time `20:35 minutes`, which improved to `14:04 minutes` when six clusters where used. The significant variation in run-time is likely a result of the quality of the network connection between clusters and available RAM (as the server is shared). On a home computer with cores

used as clusters, communication between each cluster is much easier, thus less overhead is required to consolidate results. Contrastly, when running on the server, the quality of the connection deteriorates making the consolidation much more inefficient and increasing the total run-time.

6 Conclusion

6.1 Summary

Overall, running the complete processes for one state, we can reduce the total run-time by approximately 40 per cent. The table below shows the performance results achieved for Arizona using either the single processing approach or a combination of the best approaches that perform better than the single process approach. These results are particularly helpful when used at scale as they reduce our processing time $\mathcal{O}(n)$ for the data wrangling and pre-processing, and $(\mathcal{O}(n * v * k))$ for the LDA modelling from by approximately $2/5^{ths}$.

Overall Optimisation Results

	Function	Run-time (Single Processing)	Run-time (Optimal)
1	cw.extractCases	00:00:10	00:00:10
2	cw.wrangleCases	00:00:10	00:00:02
3	tm.createCorpus	00:00:14	00:00:11
4	tm.cleanWords	00:01:21	00:01:21
5	tm.filterWords	00:01:32	00:01:32
6	ldaModel	00:05:38	00:03:10
	Total	**00:09:05**	**00:05:26**

6.2 Model Results

The LDA model produced also delivers some interesting results that lend themselves for use in a legal setting. From our results, we can extract human-interpretable topics from the cases, where each topic is characterised by the words they are most strongly associated with. The models produced using the single and multiprocessing approaches differ slightly but this is due to the fact that training is required to stabilise the salient terms identified by the model (see figures 6.2 and 7). In our optimisation, only one pass was used for each model

making the results slightly volatile. The printed topics of the models are characterised by terms which are used to determine what topics are relevant to a document unseen by the model. It is particularly helpful for clustering cases with similar latent themes for further analysis or for identifying changes in legal precedents. Lastly, as an algorithm for identifying document similarity, it also aids in search — an area that is still underdeveloped in legal technology.

Finally, with the help of pyLDAvis, a python library for interactive topic model visualisation, we can also analyse the topics in a topic model that has been fit to a corpus of text data. The interactive web-based visualisation, a snap-shot of which has been included in figures 6.2 and 7, produces the most salient terms in the model, the top three of which are *state*, *trial* and *defendant*. While this may appear innocuous, it may be picking up on the prevalence of cases brought by the State of Arizona in the courts. Governments and states tend to be the single largest prosecutor in courts as they try cases against individuals and entities spanning criminal and civil issues. So the prevalence of cases prosecuted by the State of Arizona is not in fact not a surprise. The results should, however, be interpreted with care as further tweaks to the model may still be required to deduce meaningful insights.

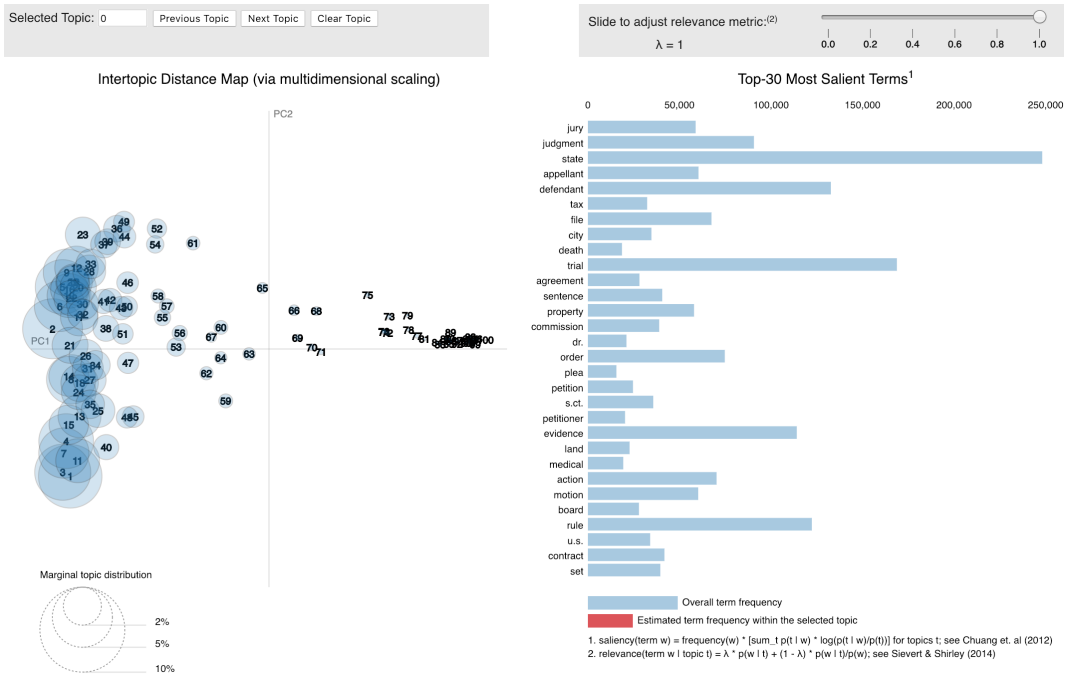

Figure 6: Single Processing LDA Model Visualisation

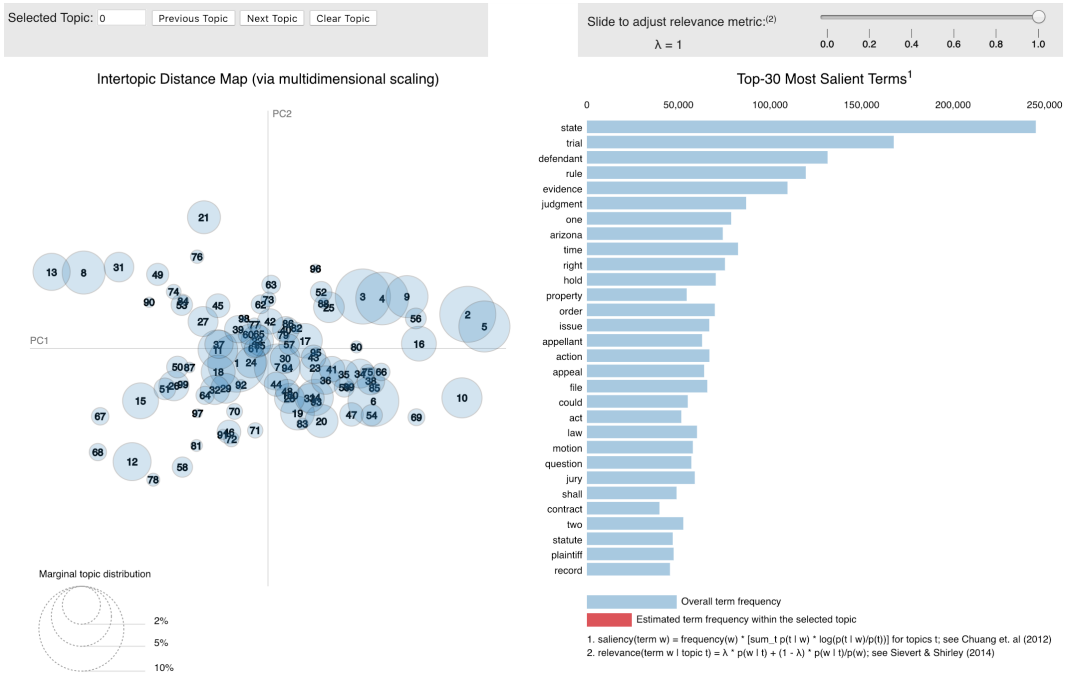

Figure 7: Multiprocessing LDA Model Visualisation

References

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, *3*(Jan), 993–1022.

Gorelick, M., & Ozsvald, I. (2014). *High performance python: Practical performant programming for humans.* " O'Reilly Media, Inc.".

Lanaro, G. (2013). *Python high performance programming.* Packt Publishing Ltd.

Library, H. L. (n.d.). *Caselaw access project.* Retrieved from `https://case.law/`, accessed on 2018-12-30.

Olsen, M. (1993). Signs, symbols and discourses: A new direction for computer-aided literature studies. *Computers and the Humanities*, *27*(5-6), 309–314.

Řehůřek, R., & Sojka, P. (2010, May 22). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (pp. 45–50). Valletta, Malta: ELRA. (`http://is.muni.cz/publication/884893/en`)