**Samtla-Char-NER Report**

**Implementation of Character-based Named Entity Recognition into the Samtla System**

*Student: Matthew Ralph,*

*MSc Computer Science project report, Department of Computer Science and Information, Birkbeck College University of London*

[*mralph02@dcs.bbk.ac.uk*](mailto:mralph02@dcs.bbk.ac.uk)

*Supervisor: Dr Dell Zhang*

*With thanks to Dr Martyn Harris*

*This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. The report may be freely copied and distributed provided the source is explicitly acknowledged.*

# Abstract

**Update at end.** Recent approaches to Named Entity Recognition (NER), such as that of (Kuru, Arkan Can and Deniz, 2016), demonstrate that a character-level representation of textual data can yield good results when training a deep learning model. In this project, a set of Hansard debates is aggregated, processed and labelled for use in a Bidirectional Long Short-Term Memory (BLSTM) neural network. The trained model, and the original dataset, is integrated with Birkbeck’s Samtla digital humanities text archiving system, such that the Hansard texts can be browsed in the interface, and previously unseen Named Entities are highlighted.

# Acknowledgements

I would like to express my gratitude to the people who taught me to program in Python by working on real problems, particularly to Ali Lotia and Ogonna Iwunze, whose expertise is matched only by their patience and compassion. I would also like to thank Sergio Gutierrez-Santos, whose instruction in the Java programming language was carefully designed and helped to open up a world of structured code for me, as well as demystifying unit testing.

I am grateful to Dr Martyn Harris for his help and encouragement when exploring this project and its potential integration with Samtla, and to Dr Dell Zhang for his ideas, advice on the academic landscape surrounding Named Entity Recognition, and quick responses to my queries. Petar Konovski’s quick assistance in setting up a server in Birkbeck with library dependencies greatly simplified the process of moving my code onto a suitable sized server, and I am grateful to Systems Group for the use of such a beefy machine.

Finally, I would like to thank my wife for all her help throughout this Master’s programme, while she worked on her own master’s and continued to support so many people.

*Ad Maiorem Dei gloriam*, from whom all language flows. Ps 19v14. **Pange Lingua?**

# List of figures and tables

**Compile at end**

Table of Contents

[0](#_Toc522276672)

[1. Abstract 1](#_Toc522276673)

[2. Acknowledgements 1](#_Toc522276674)

[3. List of figures and tables 1](#_Toc522276675)

[5. Introduction (including background) 3](#_Toc522276676)

[6. Overall Results (trailer) 4](#_Toc522276677)

[7. Software Architecture 4](#_Toc522276678)

[7.1. The Pipeline of tasks 4](#_Toc522276679)

[7.2. Invoke 5](#_Toc522276680)

[7.3. Named Entity Downloading 6](#_Toc522276681)

[7.4. Raw Hansard downloading 7](#_Toc522276682)

[7.5. Hansard processing 7](#_Toc522276683)

[7.6. Hansard chunking 8](#_Toc522276684)

[7.7. Hansard interpolation 9](#_Toc522276685)

[7.8. Hansard Numerification 10](#_Toc522276686)

[7.9. Partition into datasets and sizes 12](#_Toc522276687)

[7.10. Formation of Tensors 14](#_Toc522276688)

[8. Implementation issues 14](#_Toc522276689)

[8.1. Wikipedia data cleanliness 14](#_Toc522276690)

[8.2. Interpolation overlaps 16](#_Toc522276691)

[8.3. NLTK Treebank word span\_tokenize bugs 16](#_Toc522276692)

[8.4. Toy dataset model – tensor sparsity 17](#_Toc522276693)

[8.5. Hansard Presentation issues 17](#_Toc522276694)

[9. Planning 17](#_Toc522276695)

[10. Testing 17](#_Toc522276696)

[10.1. Unit testing 17](#_Toc522276697)

[10.2. Model evaluation 17](#_Toc522276698)

[10.2.1. The ‘mini’ dataset 18](#_Toc522276699)

[10.2.2. The ‘toy’ dataset 19](#_Toc522276700)

[10.2.3. The full dataset 20](#_Toc522276701)

[10.2.4. Cross-validation 20](#_Toc522276702)

[10.3. Overall evaluation 20](#_Toc522276703)

[11. Summary and Conclusions 20](#_Toc522276704)

[11.1. Pre-processing is hard 20](#_Toc522276705)

[11.2. Labelling is hard 20](#_Toc522276706)

[11.3. Sentence tokenization is hard 20](#_Toc522276707)

[11.4. Unfinished work 20](#_Toc522276708)

[12. References 21](#_Toc522276709)

[13. User Manual 21](#_Toc522276710)

[14. Appendix A: Code 21](#_Toc522276711)

[15. Appendix B: List of Invoke tasks 21](#_Toc522276712)

[16. Appendix C: What’s My Work 24](#_Toc522276713)

# Introduction (including background)

The brief for this project was to demonstrate Named Entity Recognition, using the approach cited in (Kuru, Arkan Can and Deniz, 2016), and the Keras implementation of this provided by GitHub user 0xnurl.[[1]](#footnote-1) The target dataset was the Hansard, the record of debates in both of the houses of Parliament in the United Kingdom.[[2]](#footnote-2) This dataset is now available via the Parliament UK Data API,[[3]](#footnote-3) however this API is largely undocumented and was not available at the start of this project. Instead, I used the They Work For You API,[[4]](#footnote-4) which has all debates from 1919 onwards available for download in a parsed XML format annotated with metadata about the speaker. I did not have time to use this high-quality metadata during the project. However, I am grateful for free use of this API which certainly made data preparation easier for me.

In order to implement the model, I had to produce labelled Hansard data. To manually label a few thousand debate documents, required to train even a very basic model, would have been too time-consuming for a project of a few months. So, I used a form of automatic labelling I refer to henceforth as ‘interpolation’, the algorithm for which is explained in section 7.5. Interpolation relied on me having a very large set of Named Entities in my chosen categories or locations, organizations and people. I used the DBPedia SPARQL endpoint[[5]](#footnote-5) and Python’s excellent SPARQLWrapper library[[6]](#footnote-6) to download all the Named Entities on Wikipedia in these categories. There were some data cleanliness issues with this dataset that I never fully overcame, which are detailed in section 8.1.

I used the interpolated (labelled) Hansards to generate a Y tensor. The processed Hansard debates themselves were segmented (or, as I refer to it below, “chunked”) into sentences, and then each character was converted to a number, to create the X tensor. I then used the 0xnurl implementation to train the BLSTM model. An overview of results is given in section 6.

I chose this project because of my interest in linguistics and in humanities texts. In my first degree, Classics, I was fortunate to study linguistic change from Classical Greek to *koine*, the language of the New Testament. I also studied some phenomena of Latin that are markers of a particular gender or class. This project is a tiny step, greatly helped by the labours of TheyWorkForYou, towards mining the value of the Hansard records.

This project is also in part politically motivated. It is vitally important that democratic citizens re-engage with the task of using factual analysis and solid statistics to make important decisions, rather than being emotionally stirred by the language of tyrants. This is a far older problem than the Romans and Ancient Greeks. Learning which companies, places and people we spend most energy talking about as a democracy seems to me, in its own small way, a part of that enormous and essential task.

# Overall Results (trailer)

**Do at end**

# Software Architecture

## The Pipeline of tasks

This project is, in essence, a data pipeline. Data is sourced from Hansard debates and from Named Entities, combined using a variety of algorithms, and then stored in a format that can then be used to predict unseen named entities. As such, it is best visualised using a pipeline flow (see Figure 1). Each element of the pipeline is introduced in more detail in the sections below, along with details of the algorithms and data storage mechanisms used. Implementation difficulties are discussed in section 8.

The outputs of each pipeline step were persisted to disk, either as simple text files, Python pickle objects in binary format, or Keras’ on H5 binary output format. Such persisting is essential when working with a large amount of data, firstly to allow each stage of the output to be validated and checked, and also to ensure the whole pipeline would not need to be run (which takes several days) every time a bug is discovered. As most of the functions in the pipeline do not return pure values but write their results out to disk (using ‘print’ statements only to inform the user of their progress), a unit testing approach was needed that could fake a UNIX filesystem in order to validate the functions were working as expected. This is explained in section 10.1.

Were this system to be ‘productionised’, then all the stages of this flow would be run through a Continuous Integration system such as Jenkins or GoCD which would run the different stages in the correct sequence; in such an arrangement, new Named Entity data and newly produced Hansard records could be fed into the pipeline and used to continuously update the model to account for new data.

## Invoke

As so much of this project’s effort was in collecting data for pre-processing, a command-line driven front-end was preferred over building a Graphical User Interface (GUI) just for the internal tasks of gathering, processing and aggregating data. Given the pipeline structure of the project, it was essential to have a tool that would allow code execution to start at any point in the pipeline, with all the correct dependencies in place, having run any prerequisite tasks required. Invoke[[7]](#footnote-7) was chosen, after some experimentation with Argh, Shovel and Doit.

Invoke was found to support arbitrary library imports from the Python global library and the current project, whereas Doit manipulated the user’s PYTHONPATH and so could not be integrated with a project structured into modules. Invoke also supports the basic features for which one might use a Makefile – a simple command line front-end providing many possible entrypoints into an application, with listed prerequisite tasks which could be called with specified, or default, arguments. In contrast to using Make, the task file itself (tasks.py in the code listing, given in section 13) is in pure python and does not require tab characters for delineation. Most calls were simply Python library imports and function executions, but some required separate command-line invocations e.g. to start PyTest or MyPy (for unit testing and static type analysis, respectively), which Invoke natively supports much more elegantly than Argh or Shovel. A full list of Invoke tasks and their descriptions is found in section 15.

Step numbers below refer to the blue numbers in Figure 1.



Figure 1 pipeline data processing model

## Named Entity Downloading

**Step 1.** Firstly, named entities must be accrued. This is a prerequisite for any automated labelling approach. For locations, the CONLL2003 English dataset was used, together with DBPedia resources of type ‘dbo:Place’. For Organizations, the Amex, Nasdaq and NYSE Stock Exchange company listings were downloaded in Comma Separated Value (CSV) format, as was similar data from the London Stock Exchange, the CONLL2003 English dataset, and DBPedia’s ‘dbo:Organisation’ type. For people, the CONLL2003 English dataset and DBPedia ‘dbo:Person’ type were used, and the New York City Most Popular Baby Names data from Kaggle.[[8]](#footnote-8) Biography-center.com, which was suggested as a naming source by (Klein *et al.*, 2003), no longer has lists of names in an easily-parseable format. However, I suspect that the amount of data in DBPedia has hugely increased since 2003. As Table 2 shows, the size of the DBPedia datasets dwarf the other datasets for all three Named Entity (NE) types.

Table 1 % of NE data from DBPedia

|  |  |
| --- | --- |
| **Dataset** | **% from DBPedia** |
| Locations | 99.8 |
| Organizations | 96.8 |
| People | 99.7 |

**Step 2.** The resulting data had to be cleaned to remove stopwords and some of the more obvious junk data. The data quality issues with the NE datasets are discussed in section 8.1.

**Step 3.** Simple UNIX utilities ‘cat’ and ‘sort’ were used to deduplicate the aggregated NE lists, and sort them into a large text file for each NE type.

## Raw Hansard downloading

**Step 4.** To download the Hansards in a programmatic manner, the TheyWorkForYou API was chosen. This was on the basis of its high quality documentation, and the availability of Hansard debates in XML format, with enriched metadata tags (created using a Perl parser which ran over the source PDF Hansard documents originally available on the UK Parliament website) naming each speaker and detailing their constituency and party. Unfortunately, there was not time in this project to make use of this extra metadata.

Python’s concurrent.futures.ThreadPoolExecutor implementation was chosen to increase the speed of downloads as this activity is mainly bound by network I/O. Only function invocations, with their required parameters, needed to be provided in order for Python’s concurrent library modules to parallelise the downloads – no manual thread handling code is needed.

## Hansard processing

The files downloaded from TheyWorkForYou are XML files with a lot of markup and metadata which would distract from the Named-Entity-learning task. After failed attempts with bleach.clean,[[9]](#footnote-9) which fails to remove nested HTML tags, the lxml library’s etree module[[10]](#footnote-10) was successfully used to remove all markup and preserve just the text of the debates. In order for lxml to accept the XML files and process them, the encoding of the files and the lxml library’s config had to be set to use UTF-8. Hansard debates use a wide range of characters, including accented letters like é as well as abbreviations like ¾, so it makes sense to pick the most widely-used Unicode encoding standard.

## Hansard chunking

**Step 6.** I use the term ‘chunking’ throughout this report and the codebase, to refer to the process of sentence-segmentation. This avoids any confusion with the word-segmentation tokenizer, which is used in the interpolation algorithm (see section 7.7). The ‘chunker’ used is the NLTK Punkt sentence tokenizer. However, early testing showed that it struggled with the abbreviations used in Hansard, in particular ‘hon.’, which occurs frequently as a shortening of ‘honourable’. The Punkt tokenizer would view this as the end of a sentence, particularly as it often occurred in the context of ‘the hon. Gentleman’, with the following word capitalised.

Training a sentence segmenter on Hansard data with sentence markers would be a project in itself, so I merely passed several common abbreviations to the chunker, as shown in Code Snippet 1.



Code Snippet 1 NLTK Punkt tokenizer prepared with some common abbreviations.

The format chosen for storing sentence-chunks was a separate file, named the same as the original debate file but ending in -spans.txt. This file contained new-line separated tuples of character-offsets for each sentence start and finish. The format originally chosen was simply to write out a new file for each sentence of each debate, and to auto-generate file numbers such that, if a debate was called ‘Public Sector Pay.txt’, the generated sentences would occupy files called ‘Public Sector Pay-chunk-0.txt’, ‘Public Sector Pay-chunk-1.txt’, etc. This format was just as quick to generate but used a huge amount of disk space. Indeed, I had only processed debates as far as May 1966 when the 200GB of space allocated for this project on my laptop ran out. On further investigation, it was noted that the Mac OS HFS+ filesystem will allocate 4k for any new file, as this is its minimum block size. Hence, the sentence segmenting algorithm was creating a large number of very small files (there are 4,714,480 sentences in the total dataset). The minimum size of these files was 4KB each, but the maximum was as large as the longest sentence. I changed approach to use a single file to store just spans, as Figure 2 shows.



Figure 2 chunking process

## Hansard interpolation

**Step 7.** The interpolation algorithm is detailed in Figure 3. Even though the deep learning model we are using is character-based and has no knowledge of word-boundaries, for the interpolation a word tokenizer was used – the NLTK Treebank tokenizer. The reason for this was simply one of performance.

The challenge for interpolation was to find an algorithm that could match against a Python set object (to take advantage of the hashing-based implementation of sets in Python and avoid the full scan that a list would require), while also making the longest possible match. For example, as “Tonbridge” and “Tonbridge Wells” are different locations, we want to ensure that the longer match is found even if the shorter match would be found first with a simple scan through the text. Similarly, even though “Paris” is a location, “Paris Hilton” is a person and should be identified first, even though the Named Entity type is different.

An n-grams approach is taken. For each text, all n-grams are generated using Treebank’s span tokenizer (there were bugs found in this approach – see section 8.3). The default value used for n was 4, and as such 4-word NEs are the longest that we can interpolate. The n-grams are right-padded with Nones (see the Figure 3), so that Named Entities that are less than n words from the end of a sentence can still be matched. We then take each possible suffix of the n-gram, starting with the longest, and attempt to match it against all Named Entity lists – first locations, then organizations, then people.

The end result looks like Code Snippet 2. Note how ‘Railtrack’ has been identified as an Organization (represented by integer ‘2’), while the rest of the phrase is assigned the ‘NULL’ label 0. The text is rendered with its interpolated labels underneath it, using a helper function “hansard-display-interpolated-file” to line up the characters. For a complete list of all Invoke tasks written for this project, see Section 15.



Code Snippet 2 Interpolated Hansard text sample

The main problem encountered with interpolation was that of overlapping Named Entities. This is described, along with its solution, in section 8.2. Of course, the interpolation algorithm was merely designed to provide better-than-nothing labelling. For high quality labelling, human work would be required. The problems with this automatic labelling approach are discussed more generally in section 11.2.

## Hansard Numerification

**Step 8.** To generate the X tensor, the debate texts themselves had to be converted to Numpy arrays of integers. A CharBasedNERAlphabet was generated from the debate texts using hashing buckets as described in section 7.9, and then a simple lookup against this alphabet was used to convert individual Unicode codepoints into integers from the alphabet. This results in much smaller tensors than simply using the Unicode codepoint value directly, as Unicode has a total of 137,374 characters,[[11]](#footnote-11) requiring a 3-byte Integer to store. The CharBasedNERAlphabet had only 160 characters so was much more compact.



Figure 3 Interpolation algorithm

## Partition into datasets and sizes

Any machine learning model requires dataset for training, for configuration of hyperparameters and for validation. In this project, these datasets were named ‘train’, ‘dev’ and ‘test’ respectively.

It is important that the divisions used for these datasets are fairly distributed and do not contain any biases. For instance, if the whole dataset of debates were treated as one linear list from 1919 to the present day, with contiguous segments used for each dataset, this would be a biased distribution, because the use of language changes over time. The model could be trained on early 20th Century English but then validated on 21st Century English, skewing the results.

To avoid this, all the debates’ file names (including the date on which they were spoken in parliament and the subject of the debate) were hashed using Python’s built-in hashing implementation. The file names are strings, and Python’s in-built hash() function takes each character, converts to an integer, and then uses exponentiation and addition to combine them. The modulo of the resulting integer was taken with respect to the number of buckets (which was set at 320), resulting in buckets of equal numbers of debates which are evenly spread with respect to time. The contents of each bucket are saved in bucket list files in the project, so the same datasets can be used consistently (for example, we never want to use data in the “test buckets”, even for manual validation).

In order to convert the 320 numbered buckets into datasets, a simple function was used, which is shown in its entirety in Code Snippet 3. Note that, while we hashed the whole interpolated dataset (all debates from 1919 to August 2018) into 320 buckets, the function only uses the first 8 buckets. This is because, at this stage of the project, a smaller set of files is used to ‘test-drive’ the capabilities of the model. This is referred to as the “Toy” dataset (see section 10.2.2 for an evaluation of this smaller dataset).



Code Snippet 3 Converting bucket numbers to datasets

Note also that the hash-bucketing technique was also used to build an alphabet for the model. When training is started, a set of debates is read in from disk, and the Unicode characters used in those debates are unioned together to make a set which initializes the CharBasedNERAlphabet object used to convert debate texts into a stream of integers for the X tensor (**Step 8** in Figure 1, and referred to as ‘numerification’ in the codebase. See section 7.8). To read in all 66,459 debate files to generate such an alphabet seems wasteful – characters that are not part of the standard Roman alphabet or common English punctuation occur very rarely and give us very few clues about Named Entities. The characters in use, unlike the lexicon, change very little over the decades, so originally the alphabet object was simply built off all the debates from an arbitrary year (1949 to begin with). However, a more principled approach was to use all the debates from one bucket to create the alphabet object. All characters encountered that are not in the alphabet are given the integer for <UNK>, the unknown symbol.

As hoped for, each bucket contains roughly 205 debates, with a standard deviation across all the buckets of 15.5. One consequence of this approach is that all the ‘chunks’ (sentences) in a given debate are placed in the same dataset. That is, for each Hansard debate *h* that exists in a dataset *d* (be it train, test or dev), *all* of *h*’s sentences are found in dataset *d* and none of them are found in a different dataset. This does not seem to present a problem – the main motivation of the bucket-hashing approach was to ensure the datasets’ textual data is distributed across time.

One risk, however, is that longer debates have more sentences – so datasets which happen to have longer debates in, will have more data in them. During the course of this project, this did not present itself as a problem, however as the model became more refined and usable, the hashing approach would need to be revisited, with hashing done at the sentence level.

## Formation of Tensors

**Step 10.** The X and Y tensors are generated from the numerified and interpolated Hansard data respectively. Both are constructed as native Python nested lists by reading from their respective data sources, and then processed using Keras’ pad\_sequences helper function. This accomplishes three things: it left-pads sequences shorter than sentence\_maxlen with 0s, it truncates any sequence longer than sentence\_maxlen, and it converts the nested list structure to Numpy nested arrays of the correct datatypes. The dimension contents for the X and Y tensors are shown in Table 2 and Table 3.

Table 2 X tensor dimensions

|  |  |  |
| --- | --- | --- |
| **Dimension** | **Content** | **Length** |
| 1 | Text Samples | Batch-size (varies depending on dataset) |
| 2 | Characters | Sentence\_maxlen (200) |

Table 3 Y tensor dimensions

|  |  |  |
| --- | --- | --- |
| **Dimension** | **Content** | **Length** |
| 1 | Text Samples | Batch-size (varies depending on dataset) |
| 2 | Characters | Sentence\_maxlen (200) |
| 3 | One-hot array of labels | Number of labels (4) |

For the toy dataset, the Numpy arrays are pickled to disk. This is so they can be used in multiple model training runs with different hyperparameters, without having to regenerate the tensors. The Keras model used from 0xnurl is modified only slightly, so that the dataset’s get\_x\_y function calls a function in the ‘matt’ package, representing a package of library files added to the Keras model as part of this project. My contributions are placed in a separate package in order to clarify exactly what is my contribution to 0xnurl’s model. A file list of all files in this project, who authored them and what they achieve, is found in section 16.

# Implementation issues

## Wikipedia data cleanliness

The datasets downloaded using the DBPedia SPARQL API are a result of volunteer contributions to Wikipedia article content and metadata. As a result, the data is both voluminous and not very clean. Duplicates in the data, like the presence of “Ralph Allwood” and “Ralph Allwood MBE”, are not a problem, as they will improve the coverage of interpolation. However, the DBPedia API contains entries like the following which are wrongly listed as people:

* “(15 July 1914 – 8 November 1927)”
* “(1833-1905)”

And the following are listed as organizations:

* “I” (the first-person pronoun)
* “.” (a single period)

And the following as places:

* “And the”
* “the”

In the case of “(15 July 1914 – 8 November 1927)”, this would appear to be the birth and death dates of a person, wrongly classified as a person’s name. The other examples just seem to be mis-classifications of command words or characters in English.

At first, such oddities were manually removed, but it was clear that a more principled filtering approach was needed. A number of processing steps were then added to the Named Entity lists once they were downloaded from DBPedia. These are listed in Table 4.

Table 4 DBPedia post-processing tasks on Named Entities

|  |  |
| --- | --- |
| **Task** | **Regex (if applicable)** |
| Remove double quotes |  |
| Left-trim whitespace |  |
| Remove lines that are entirely numbers of symbols | ^[!@£$%^&\*()0-9 ]+$ |
| If whole line starts and ends with brackets, remove them |  |
| If line starts with more than one single quote, remove all single quotes at start of line | (.\*)'{2,}$ |
| If line starts with just whitespace or asterisks, remove them | ^[\* ]+(.\*) |
| Remove words shorter than 4 characters |  |

Finally, if after all processing, all the remaining words in a Named Entity are stop-words (taken from the NLTK Corpus of English stop-words), then whole entry is removed. Of course, this means that some perfectly valid Named Entities like ‘The Who’ cannot be recognised in the interpolation phase. This is a necessary trade-off of cleaning up the data in an automated fashion. Note that words shorter than 4 characters are also removed, before the stopwords step. These tend to be strange stub words like ‘ar’ which are low-value and hard to filter.

DBPedia contains a lot of Chinese, Russian and Arabic names in their respective scripts. This is not a data cleanliness problem, just a phenomenon of Wikipedia’s global reach. There is no principled reason to remove these names from the dataset, but it is unlikely that they would appear in Hansard in their native character-sets.

## Interpolation overlaps

One problem the early incarnation of this algorithm is that earlier Named Entities could be overwritten by later ones. For example, in the phrase ‘The House’, the two-word phrase may be successfully interpolated as a place (referring to the house in which the debate takes place). However, when the algorithm moves on to the word ‘House’, it will label it as an organization (House is the name of two different companies listed on Wikipedia). The resulting labelling is 111122222, with ‘The’ still labelled as a location even though ‘House’ is re-labelled as an organization. Aside from the ambiguity about what the correct labels for the whole phrase are, ‘The’ is now definitely labelled wrongly.

The fix for this problem was to arbitrarily choose the first-matched Named Entity as the correct one. In the case of ‘The House’, the 2-word phrase is labelled as a location. This labelling is then protected – as the n-grams window slides forward to recognise more Named Entities, we keep track of whether the phrase being examined overlaps with a previously recognised Named Entity (to keep track of this, we store recentest\_match\_end, the index of the character at the end of the most recent labelling). If it does, we skip over this n-gram without searching for any more Named Entities. Code Snippet 4 shows the logic (“overlaps” is a helper function which simply compares the first index of the current ngram with recentest\_match\_end and returns a Boolean).



Code Snippet 4 logic to avoid re-interpolating overlapping NEs

## NLTK Treebank word span\_tokenize bugs

NLTK’s span\_tokenize has two open issues on GitHub, one of which was opened only four days ago.[[12]](#footnote-12) “Span\_tokenize” cannot handle some inputs with unbalanced or nested quotation marks. The issue seems to stem from the implementation, which first tokenizes the text into individual words (not offsets), then hunts through the text for each word individually, in order to generate the offsets. The NLTK community on GitHub has submitted a number of fixes to the code used to match the text, but the conversation on the issue as a whole concluded[[13]](#footnote-13) that the only robust solution was to remove the span\_tokenize method completely.

This project is fully reliant on NLTK’s Treebank’s word span\_tokenize to generate offsets used to create ngrams to scour for Named Entities, as described in detail in section 7.7. The character-position indices have to be preserved, in order to generate a Y tensor with NE labels in the same positions as the original characters. In order to side-step the NLTK span\_tokenize bug, I simply searched for all files that had failed interpolation on the first iteration, replaced all occurrences of double-quotes with single-quotes, and then re-interpolated them, as shown in the top-left of Figure 3. This approach is not ideal as it involves changing the raw textual data; given more time, a robust solution to span-tokenizing would be investigated, and relevant code submitted to the NLTK project for review in a Pull Request. Another approach is to completely exclude word-tokenization from the interpolation process, using a sliding character-window over the text to find and label named entities. This option was excluded because of its poor performance and time constraints. Finally, another library could be used to provide span\_tokenize functionality. Spacy.io was investigated, but was not found to have span\_tokenize methods that return indices of the start and end of words.

## Toy dataset model – tensor sparsity

**Choosing a good sentence size - medians**

## Hansard Presentation issues

**E.g. No speaker information due to XML processing**

# Graphical User Interface

In the original proposal for this project, I had hoped to integrate both the named entity recognition model and the pre-processed Hansard debates into Birkbeck’s Samtla system. However, by the time the data was ready for integration, there was no response from the Samtla team until it was too late for any integration work to take place. This could possibly have been avoided if I was more pro-active in determining with the Birkbeck Samtla team a deadline by which integration needed to take place – for a more thorough discussion of my plan’s shortcomings, see section 10.

As outlined in my proposal, a simple Graphical User Interface(GUI) needed to be created, which would at least contain the following basic functionality: allow a user to see a list of all possible Hansard debates on a given date; allow a user to select a Hansard debate from a given date to display all of its text; The Hansard debate displayed should in some way annotate all Named Entities predicted by the Named Entity model which should be run on the server. As this GUI is very much a stub designed to enable demos of the project, no thought is given to the traditional non-functional requirements of performance or security. Performance in particular would be greatly improved with a relational database to index Hansard debates for a particular date, and cached results from the model predictions.

Python Flask was chosen as the web-server to build a very basic site, as it supports Python 3 and is less opinionated than Django. Its only dependencies are Jinja2, which we use to template the HTML files for the site, and Werkzeug for managing the Web Server Gateway Interface (WSGI) HTTP routing.

In the first draft of the GUI, when the user clicks on an entire Hansard debate, the whole debate is first fed into the predict\_str() function of the model, to generate Named Entity labels for the whole text, before displaying the whole annotated text to the user. However, given that running a whole debate text through predict\_str takes about 8 minutes even on the Birkbeck machine learning server, it is clear that this approach could never constitute an acceptable user experience. Of course, the values could all be pre-computed, but this seems at odds with the goal of using a machine-learning model which can be dynamically re-trained and updated. A second approach was chosen, as

# Planning

Lack of detail in the plan. Steps missing from plan, actual timeline used from worklog. Integration is a risk in all IT projects. **Finish**

# Testing

## Unit testing

For all the pre-processing code submitted for this project, the Pytest framework was used to run simple unit tests, validating that functions return expected outputs for given inputs. In this project, most of the main functions used did not return values directly to the caller, but wrote values out to disk, either as text files, as binary Python pickled data, or as Keras h5 database-files. Similarly, many functions expected their input to be a path to a file on disk, from which they would read either text or binary data to continue processing. This approach was taken to support the ‘pipeline’ concept outlined in section 7.1.

In order to validate that a function was writing out expected values to disk, Pyfakefs[[14]](#footnote-14) was used to create a fake filesystem in the context of the unit test. Then the function was run, and the dummy file on the fake file-system was then examined to ensure it had the expected context. Given Pyfakefs’ native integration with Pytest, the written tests do not have to bear much complexity for this setup.

To illustrate this approach, Code Snippet 5 shows a regular unit test written in Pytest. A value is passed into the ‘onehot’ function, and its output is compared against an expected value in a simple equality assertion. Code Snippet 6, by contrast, shows a Pytest unit test with Pyfakefs. The argument passed into test\_interpolate\_one, ‘fs’, is the fake filesystem. The fs.create\_file call is used to create a dummy file in the fake filesystem to feed into the function under test, and an empty dummy file to accept the function’s written output. Once the function (interpolate\_one in this test) is called, the resulting output file is read from the dummy filesystem, and its contents compared to their expected result.



Code Snippet 5 a regular Pytest unit test



Code Snippet 6 A Pyfakefs Pytest unit test

## Model evaluation

The Keras model was evaluated using the Keras ‘evaluate’ method. Of course, the labelled data used to fit the model was both limited by the contents of its data sources, and by the non-human manner of the labelling – the labels were ‘interpolated’ using the algorithm described in detail in section 7.7.

The various different datasets and their sizes are listed in Table 5.

Table 5 Model datasets used

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model name** | **Training dataset size** | **Dev dataset size** | **Test dataset size** | **Epochs** | **Notes** |
| Mini | 4000 | 4000 | 4000 | 2 | Takes first 4000 sentences from ToyV1. |
| ToyV1 | 2323451 | 1233720 | 1157309 | 7 | All validations on Dev dataset returned NaN. |
| ToyV2 | 200000 | 6000 | 6000 | 6 | Uses same data as ToyV1, but with arrays clipped at the limits shown. |

**Sensible baseline: assume everything is NULL. The non-null-labels metric.**

### The ‘mini’ dataset

In order to test out a complete run of the Keras model and verify saving of its state and tracking of its loss scores across epochs, the ‘mini’ dataset was generated. The mini dataset is derived from the ‘toy’ dataset (for which see section 10.2.2), but in the first dimension of the X and Y tensors, only the first 4000 samples are taken for each dataset. So only 4000 sentences are used for train, dev and test. The below figures show the predictably appalling behaviour of this dataset, in terms of accuracy, loss and non-null label accuracy (i.e. accuracy when *not* using the null label, ‘0’, meaning ‘Not a Named Entity).

Note that the ‘mini’ model was only run for two epochs. We can see that accuracy rose and loss decreased. However, the non-null label accuracy also decreased – it appears that in the early epochs of the model, the most efficient way to minimise loss is to label every character as 0, the null label. This observation was also borne out in the ‘toy’ dataset, where non-null label accuracy fell for the first 100k samples or so, before starting to rise.

Attempts to ‘predict’ using the mini dataset also matched this observation. The model has a marked preference for the NULL label. Indeed, all manual validation done using the mini model returned all NULL labels.



Figure 4 Mini dataset accuracy



Figure 5 Mini dataset loss



Figure 6 Mini dataset non null label accuracy

### The ‘toy’ dataset, version 1

The toy dataset was constructed with 8 buckets of the 320 in the dataset, roughly 1600 debates in total. Half of these were used for training, and a quarter each for test and dev. After two epochs, the model stopped training due to a bug discussed below. After this, I trained the model for a further five epochs.



Figure Toy dataset NaN validation accuracy



Figure Toy dataset NaN validation loss



Figure Toy Dataset NaN Validation Non-Null Label Accuracy

As you can see, the graphs above do not show validation accuracy. This is because of a bug encountered with the Toy dataset, where after each epoch, evaluation done against the ‘dev’ dataset leads to a result of NaN (Not a Number). Because of this bug, it was not possible to track the model’s performance on a dataset other than ‘train’ during the fitting process, and hence impossible to detect and avoid overfitting. This was also the reason why the model originally stopped training after two epochs – the Keras EarlyStopping callback was called after two epochs with no improvement in the validation score. In order to train for more epochs, the EarlyStopping configuration had to be removed from the model.

While attempting to fix the NaN validation problem, the NumPy arrays used in all datasets were searched for NaN values, infinity values and other non-numeric values, without success. It is noted that this problem was not encountered in the mini dataset. Hence, it may be related to the batch-size used in the toy dataset. This trained model, named ToyV1 in Table 5, did indeed exhibit signs of serious over-fitting. In all manual testing done, it only ever returned the Null label. Its evaluation metrics are shown in Table 6.

Table 6 Evaluation of Toy dataset v1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataset used** | **Dataset size** | **Loss** | **Categorical accuracy** | **Non-Null Label Accuracy** |
| Test | Mini | 0.1386 | 0.9570 | 0.5270 |
| Test | Toyv1 | 0.0885 | 0.9726 | NaN |

As non-null label accuracy is so much lower than the categorical accuracy, it is no surprise that the model had such a marked preference for returning the Null label. Figure 7, Figure 8 and Figure 9 show no validation scores because of these NaN return values.

### The ‘toy’ dataset, version 2

Comparing the Mini and ToyV1 datasets, it is clear that the Mini dataset at least returned validation data, a signal that could be used to detect overfitting during the training epochs. This understanding led to ToyV2, which caps the training data to 500,000 sentences, and test and dev to 6,000 sentences each. The values were picked empirically based on the successful validation feedback from the Mini dataset.



Figure ToyV2 dataset accuracy



Figure ToyV2 dataset loss



Figure ToyV2 dataset non-null label accuracy

### The full dataset

The ‘toy’ dataset’s tensors were 18GB in size as NumPy objects. To train the toy dataset for seven epochs on Birkbeck’s deep Machine Learning server took about 36 hours per epoch, or about eleven days. With 2323449 sentences of data trained every epoch, it is clear that the ‘toy’ dataset actually represents a significant amount of data.

As the toy dataset required 18GB just to load the tensors into memory, it is anticipated that the full dataset would require 720GB or thereabouts. Clearly, to use all the interpolated Hansards into the model (divided into train, test and dev sets) would require using Keras’ fit-generator methods, to generate the tensors on the fly as they were needed, and a large amount of time. It is possible that the NaN validation problems described in 10.2.2 would recur, given the much larger dataset (182,575,072 sentences in total). Or, it is possible that, with a suitably chosen batch-size for the generator, the NaN validation problem could be avoided. It is not clear how long such a model could take to train – if the time taken scaled linearly from the ‘toy’ dataset, which used 8 of the 320 buckets, it could take 1440 hours per epoch, or 60 days. It is hoped that the time taken would not scale linearly, given that the batch size, and the required stochastic gradient descent calculations per batch, would be much smaller.

As of Friday 17th August 2018, I took the decision to prioritise working on a simple GUI interface, instead of writing the generator methods required to train the full dataset. These generator methods could be quite easily derived by wrapping create\_x\_toy and create\_y\_toy in model\_integration.py with a python generator, then returning the Numpy arrays instead of pickling them out to disk.

### Cross-validation

Fortunately, thanks to the hash-bucketing approach to datasets, the Hansard debates are already scrambled, so I could just use contiguous pieces of the NumPy arrays to generate segments. I used Scikit-Learn’s KFold class to generate ten ‘folds’. In each case, a fold was one tenth of the data, used for validation. The other nine tenths were used for training. Because of issues mentioned with using the full dataset in 10.2.3, I used the dataset originally used to train the ‘toy’ model. As a fresh model was used for cross-validation evaluation, this was a fair evaluation on a manageable amount of data.

**Cross-validation results**

## Overall evaluation

Something in context of the Simple-GUI / Hansard

# Summary and Conclusions

## Pre-processing is hard

What to do with the metadata? Don’t want to train on it, but do want it to appear.

## Labelling is hard

Interpolation algorithm won’t abandon one interpolation for a better one – first-found bias for overlaps, location bias

## Sentence tokenization is hard

Taught specific abbreviations to the tokenizer. Still bugs outstanding.

## Unfinished work

Viterbi. Full model. Samtla integration.

# References

**Mendeley generate references.**

# User Manual

**Pipeline.sh list of commands to run to do full processing. GUI instructions.**

# Appendix A: Code

**Copy in all code.**

# Appendix B: List of Invoke tasks

**Update with new tasks**

Table 5 comprises a list of Invoke tasks which could be started within the project, along with a description of the work that they did. The tasks are a combination of environment setup, running automated tests, and the ‘business logic’ of the code – the downloading and processing of Named Entities and Hansard debates. Having one clean, uniform interface for all these tasks greatly simplified the workflow when parts of the pipeline had to be re-run, without the overhead of creating a GUI or integrating with one. Also, having pre-requisites in code avoids the need for repetition. For example, the unit tests are not run unless the virtual environment is set up, and the static type-checker has run already. These tasks are both listed as pre-requisites of the ‘test’ task in Invoke.

Table 7 List of Invoke tasks used to drive the pipeline

|  |  |
| --- | --- |
| **Task** | **Description** |
| char-ner-create-x | Create an X tensor of Numpy arrays from numerified Hansard data |
| char-ner-create-y | Create a Y tensor of Numpy arrays from onehot vectors from interpolated (labelled) Hansard debates |
| char-ner-display-median-sentence-length | Get the median sentence length of a given dataset |
| char-ner-display-pickled-alphabet | Display the CharBasedNERAlphabet object pickled to disk by char-ner-pickle-alphabet |
| char-ner-pickle-alphabet | Use a small subset of the Hansard debates data to union together all characters used and create a CharBasedNERAlphabet object with a number-to-character mapping |
| char-ner-rehash-datasets | Rehash all the debate data into a different number of buckets – discussed in section 7.7. |
| compile | Run py\_compile on all python files to find compile-time static code problems |
| enable-venv | Enable the Virtual Environment (i.e. a segregated location for pip installs) for this project. A required prerequisite for several other tasks. |
| hansard-chunk-all | Use the chunker on all Hansard debates in the collection – described in section 7.6 |
| hansard-chunk-one | Use the chunked on one Hansard debate, to allow manual validation |
| hansard-display-chunked | Display one Hansard debate, tagged with all its sentence-boundaries, to validate the sentence chunking algorithm. |
| hansard-display-interpolated-file | Display one Hansard debate, with every character tagged by the interpolator as 0 (null), 1 (location), 2 (organization) or 3 (person) |
| hansard-download-all | Concurrently download all Hansard debates from both houses, from a given starting date. |
| hansard-fix-uninterpolated | Find all Hansard debates which did not correctly interpolate due to an NLTK bug with span\_tokenize, and re-interpolate |
| hansard-interpolate-all | Interpolate (label) all Hansard debates using Named Entity data |
| hansard-interpolate-one | Interpolate (label) one Hansard debate for manual validation |
| hansard-numerify-one-to-file | Numerify one Hansard debate – i.e. replace each of its characters with the equivalent integer for this CharBasedNERAlphabet, and store in a file for manual validation |
| hansard-process-all | Do pre-processing steps on all Hansards – discussed in more detail in section 7.5. |
| hansard-process-one | Do pre-processing steps on one Hansard for manual validation. |
| hansard-write-total-number-of-sentences-to-file | Count how many sentences there are in each dataset and write out to file for easy retrieval. This information is used to estimate to the user how long the tensor creation will take. |
| model-minify-toy | Take the toy dataset and truncate the 1st dimension of all the X and Y tensors to the first 4000 samples, to create a mini dataset |
| model-train-mini | Run model.fit() on the Keras BLSTM model, with a batch of the first 4000 samples from the toy dataset. This is to test the end-to-end process of saving and retrieving the model. |
| model-train-toy | Run model.fit() on the Keras BLSTM model, with a toy dataset of 1 320th of the Hansard debates. This is to get an initial indication of the model’s learning capability. |
| ne-data-companies-download-process | Both download and process companies data from DBPedia and other sources |
| ne-data-companies-process | Only do post-processing, data cleansing tasks on companies data, to assist with iteratively improving the cleaning algorithm |
| ne-data-people-download-process | Both download and process people data from DBPedia and other sources |
| ne-data-people-process | Only do post-processing, data cleansing tasks on people data, to assist with iteratively improving the cleaning algorithm |
| ne-data-places-download-process | Both download and process people data from DBPedia and other sources |
| ne-data-places-process | Only do post-processing, data cleansing tasks on places data, to assist with iteratively improving the cleaning algorithm |
| print-debate-titles | Both download and process people data from DBPedia and other sources |
| python-type-check | Run mypy, Python’s static type checker, over all files I wrote in the project which contain type annotations |
| test | First run python-type-check, then run pytest unit tests on the project |

Each task in invoke called out to something else; most tasks invoked a library function from elsewhere in the code base, while some invoked shell commands, for example to de-duplicate and sort the Named Entity lists. In this case, it is faster for the shell to call a GNU C binary and use the ‘sort’ and ‘uniq’ commands, than to use similar functionality in Python. Using the tasks.py file as a dispatcher, without it containing any processing logic itself, ensured that it remained easy to understand and reason with as the project grew.

Once the project reached a scale where the model processing had to be done on cloud compute nodes, the use of a virtualenv also proved worthwhile. By use of a pip freeze file (requirements-freeze.txt in the project), and pyenv to install the same Python version,[[15]](#footnote-15) the DigitalOcean Droplet virtual machine, and later Birkbeck’s own server Venus, could be configured with exactly the same python libraries and python interpreter, even though the system python was different from the laptop used for development work.

# Appendix C: What’s My Work

File tree, colour coded? How many lines of code did I contribute?

1. <https://github.com/0xnurl/keras_character_based_ner> [↑](#footnote-ref-1)
2. <https://hansard.parliament.uk/> [↑](#footnote-ref-2)
3. <http://www.data.parliament.uk/dataset/12> and <http://api.data.parliament.uk/> [↑](#footnote-ref-3)
4. <https://www.theyworkforyou.com/api/> [↑](#footnote-ref-4)
5. <https://dbpedia.org/sparql> [↑](#footnote-ref-5)
6. <https://rdflib.github.io/sparqlwrapper/> [↑](#footnote-ref-6)
7. http://www.pyinvoke.org/ [↑](#footnote-ref-7)
8. <https://www.kaggle.com/new-york-city/nyc-baby-names> [↑](#footnote-ref-8)
9. <https://github.com/mozilla/bleach> [↑](#footnote-ref-9)
10. <https://lxml.de/api/lxml.etree-module.html> [↑](#footnote-ref-10)
11. <http://www.unicode.org/versions/Unicode11.0.0/> [↑](#footnote-ref-11)
12. As of this writing, Thursday 9th August 2018. [↑](#footnote-ref-12)
13. <https://github.com/nltk/nltk/pull/1864> [↑](#footnote-ref-13)
14. <https://github.com/jmcgeheeiv/pyfakefs> [↑](#footnote-ref-14)
15. <https://github.com/pyenv/pyenv> [↑](#footnote-ref-15)