

A collection of scientists in Wales between 1804 and 1919 built
using Natural Language Processing techniques.

CS39440

Taylor Brown (tab23@aber.ac.uk)
Supervisor: Dr Amanda Clare (afc@aber.ac.uk)

May 3, 2019

Declaration of Originality

I confirm that:

- This submission is my own work, except where clearly indicated
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name: Taylor Brown

Date: May 3, 2019

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science department.

Name: Taylor Brown

Date: May 3, 2019

Acknowledgements

I'd like to thank my supervisor, Dr Amanda Clare, for all of the support and assistance I received whilst working on this project. Her aid has been invaluable to the creation of this system. I would also like to thank the National Library of Wales, for the collection of newspapers that they have built, for putting these newspapers in the public domain and for granting me access to their API, that I might better complete this project. Finally, I'd like to thank Shannon Brown, undergraduate at Marjons University, for the aid she provided in the presentation of the web app.

Abstract

This paper discusses the building of a system that presents a collection of scientists who worked in Wales between the years of 1804 and 1919. It looks at how natural language processing techniques can be used for information extraction and in particular the area of named entity recognition, and investigates how this information can be presented to a fictitious user. The design, implementation and tests of three distinct parts of the system, the information extraction software, a database and a web application, are discussed, and includes an investigation into various different technologies and toolkits that could have been used.

Contents

1	Problem Analysis	5
1.1	Preparation	5
1.2	Analysis	6
1.3	Process	8
2	Phase One: Technologies and Experimentation	9
2.1	Information Extraction	9
2.1.1	Planning	9
2.1.2	Coding	10
2.1.3	Testing	11
2.2	Database	12
2.3	Web App	12
3	Phase Two: Building the system	13
3.1	Information Extraction	13
3.1.1	Design	13
3.1.2	Implementation	13
3.1.3	Testing	18
3.2	Database	19
3.2.1	Design	19
3.2.2	Implementation	20
3.2.3	Testing	21
3.3	Web App	22
3.3.1	Design	22
3.3.2	Implementation	22
3.3.3	Testing	23
4	Evaluation	25
4.1	What I have learned	25
4.2	On the Chosen Methodology	25
4.3	Deviations from the Plan	26
5	Bibliography	27
A	Third Party Code and Libraries	29
A.1	NLTK	29
A.2	Spell Corrector	29
A.3	Spell Checking	29
A.4	Human Name Parser	29
A.5	Flask	29
B	Ethics Form	31

C	External Documents	33
C.1	Requirements	33
C.2	NLP Toolkit Comparison	36
C.3	Database normalization	40
C.4	Testing Summary	42
C.5	Web Application Design	47

Chapter 1

Problem Analysis

This chapter breaks down the nature of the project into key sections, and discusses how this project was prepared for. It also includes a section on the chosen methodology and justifies this choice.

1.1 Preparation

Natural language is the name given to how humans communicate with each other. We do not always have the easiest communication principles: the ways in which we speak, or write, or talk over the internet are complicated, and filled with tiny rules that depend on subtle differences such as the relationship between the two in conversation, the time of day, or the method being used to converse. In short: natural language is messy, and software technologies often have difficulty in deriving meaning from most examples of it. Natural Language Processing (NLP) is the art of understanding and deriving meaning from human language, usually automating the task in some way. It is the intersection of computer science, artificial intelligence and human languages, and employs elements from all of these areas to derive meaning from human communication[1].

Natural Language Processing for Online Applications[2] was a key text in building understanding around NLP. It appears to be written for research or development of a toolkit from the ground up and it was a good insight into how such technologies are developed. It helped to build an understanding of how these tools work, which was required to build a foundation upon which the final system could be built. In particular, section 1.3, *Linguistic Tools* served as an excellent introduction into how tokenizers break down a string of characters into words or sentences, how part-of-speech taggers can use either a series of rules, or sets of training data and probability to tag more ambiguous words, and looks at the non trivial task of recognizing noun phrases in sentences - vital for detecting named entities.

If an entity can be described as a noun, or object, then a named entity is essentially a proper noun. These can be people, places, organizations, languages etc. The purpose of this system is to collect information about scientists, so named entity recognition is a focal point for development. In the technologies discussed in chapter Two of this report, named entity recognition cannot occur without part of speech tagging. All words in a language have a given part of speech, which define the purpose of that word in a sentence, in meaning as well as grammatically. There are eight of these parts of speech[3]:

- Noun: person, place, thing or idea.
- Pronoun: a word used in place of a noun
- Verb: an expression of action
- Adjective: modifies or describes a noun or pronoun

- Adverb: modifies or describes a verb, adjective or another adverb
- Preposition: placed before a noun or pronoun to form a phrase that modifies another word within a sentence
- Conjunction: joins words, phrases and clauses
- Interjection: expresses emotion

Every word in the English language falls into one or more of these eight categories. Nouns are primarily the focus for this piece of work, as the majority of the information required from the texts is noun-oriented; proper nouns, nouns and pronouns.

It is worth noting that many of the previous studies and research papers written about NLP, information extraction and named entity recognition have all been primarily focused on the development of the algorithms or models behind these techniques, rather than discussing the practical applications of them. While useful in the information they provide, it was determined that the building of a model from scratch was beyond the scope of this project, and so more practical implementations of these techniques were looked at instead.

NLP excels at extracting meaning from strings of text. This has made it a key area of research for companies like Google[4], where current research topics are:

- Summarization
- Conversation
- Multilingual modeling
- Sentiment analysis

One potential reason these areas are so interesting to global corporations is to turn the extensive sums of information that the human race has collected into something more manageable, so that more meaning can be derived from said information.

Information extraction is the process of deriving specific information from text, and is a primary objective of many systems that employ NLP techniques. A rapidly expanding market that makes use of information extraction is algorithmic trading. According to Investopedia's financial dictionary, algorithmic trading is trading that "uses powerful computers to run complex mathematical formulas for trading." [5] It can be completely automated and set up to consider specific scenarios. Much information that is relevant to making important financial decisions is presented by journalists in various ways, predominantly in English. Information extraction is used to collect pertinent information and present it in a format that can be used to make these trading decisions.

1.2 Analysis

The aim of this project is to produce a collection of scientists living and working in Wales between the years 1804 and 1919, in an attempt to demonstrate the ways in which these scientists contributed to their communities through their actions. It also attempts to provide a method of accessing this information.

A fictitious user was created at the offset of the project, around who's needs the requirements for the project were built. It was assumed that this user had a desire to learn something about Welsh scientists at this time of history, though whether for academic or personal reasons was left unconsidered. The anticipated desires of this user are covered more extensively in appendix C.1. These requirements can be broken down into three categories, ordered according to their relevance to the user:

1. User Interface
2. Data Storage
3. Information Extraction

This user will be able to submit a search through a web app, that will provide them with an extensive list of the results. These results will be retrieved from a database, which will be filled using a program that extracts information from a newspaper collection made available by the National Library of Wales[6].

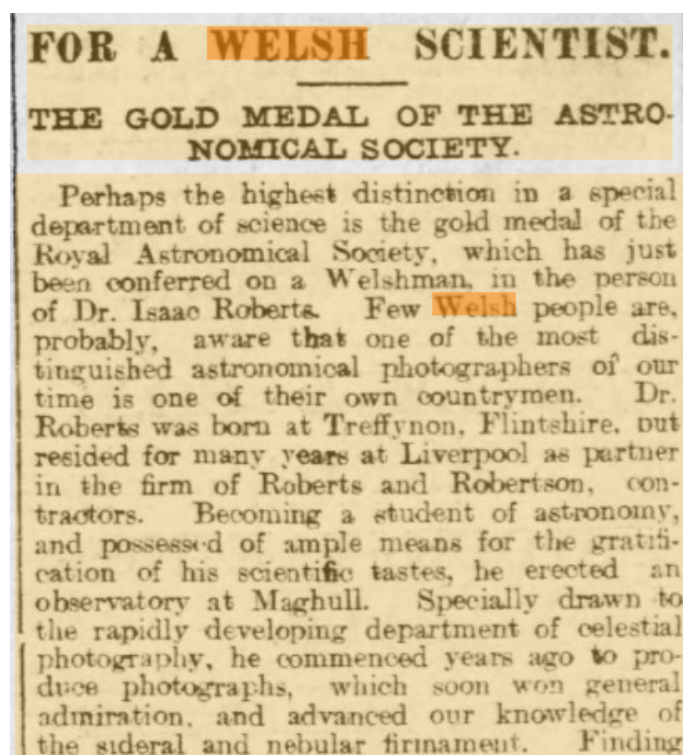


Figure 1.1: An Archive Article

The reading done prior to the commencement of this project considered various different NLP techniques, and how they could apply to this system. This narrowed the scope of the project to focus exclusively on named entity recognition. Some alternative NLP areas were considered, namely sentiment analysis and summarization; the former providing insight into how a scientist was viewed by the press and, given how easily a public's opinion can be swayed by the media it consumes, how the general public would have thought or felt about any particular scientist. Ultimately, it was felt to be unnecessary to the key aims of the project, and risked diverting attention from the creation of the core system.

Summarization could still prove itself a useful strategy in building this collection, as the potential data on any one person collected could range from two or three articles to several hundred if they were particularly famous. Condensing this information into an 'Interesting Facts' portion of a final web app gives the final design the chance to be something more interesting than merely a search engine. Summarization could turn a simple collection into a system that does a rich history in scientific innovation proper justice. Hence, summarization is still being considered a useful tool for the system, though will not be implemented during the project timeline due to tight time constraints.

1.3 Process

The approach to this project was always going to be agile in nature. There was simply too much that had to be learned, and too significant a proportion of that learning would occur as work progressed, to plan everything before implementation and testing began. The agile methodology chosen was a blend of scrum and kanban: scrum for its iterative nature, which would prove useful when the project grew in scope and complexity with time and experience, and kanban as a method of managing workload. A project with very distinct sections, like this one, requires a balanced approach to commencing and completing tasks. It was felt that this project could not be completed without careful management to avoid an overload of work, and the potential ‘burn out’ that goes with it.

Each scrum sprint lasted a week, with a discussion every Monday morning to summarize what was accomplished the week prior, and how to best spend the coming week to move closer to a fully functional system. It was at this point that the kanban board was updated to reflect the issues addressed in the discussion, with relevant tasks that had been completed being moved to the final section of the board. Each discussion focused on areas of planning, development and testing, and referred back to the requirements list in appendix C.1 to decide what new features should be worked on next.

Chapter 2

Phase One: Technologies and Experimentation

This chapter looks at the first phase of the project: the choosing of various technologies. It also investigates and discusses reasons for these choices. It covers approximately the first three weeks of development.

2.1 Information Extraction

2.1.1 Planning

Before a specific language and subsequent toolkit could be chosen, various options had to be investigated. Due to past experience with Java and C++, these were not long considered as language options past a cursory glance. These past experiences were largely negative, and led to a dislike for the languages. The two options that were deeply examined, Ruby and Python, were both languages that would be used in a complex project for the first time. After some brief research, and the realisation that the majority of NLP literature was already designed for Python, the decision was made. Python would be the best language option, and the primary task moved on to determining the best toolkit for NLP.

There are several toolkits that are available in Python for NLP. Some of these have existed for a long time, with an established user base, and plenty of support. Others have sprung up in the last five years or so. Two of the more commonly used toolkits were tested, the Natural Language ToolKit (NLTK) and spaCy, alongside a third that does not see nearly as much use, but whose design is focused primarily around support for multiple languages: Polyglot. Because it was designed for working with multiple languages, and there are newspaper articles written in Welsh as well as English, it was found to be worth investigating. The full report can be found in appendix C.2.

SpaCy is the industry standard for NLP. In the practical applications of NLP discussed in the previous chapter, if Python was the chosen language, then based on sheer numbers, it is likely the technology behind the application worked with spaCy[7]. This is primarily due to its ease of use - it's fairly intuitive with the right documentation. This doesn't necessarily make it better than the other options, merely more common in businesses. Academics, on the other hand, tend to lean towards NLTK for NLP. This is due in no small part to its exhaustive documentation: alongside detailed module documentation, the NLTK is associated with a book[8] on various NLP techniques and how to use the NLTK to accomplish them. It has also been around the longest, with an initial release date of 2001. The main reason academics tend to prefer using NLTK, though, is because it is far easier to train NLTK parsers on user data[9].

2.1.2 Coding

The aim of each program written to test the toolkits was to perform part of speech tagging and Named Entity recognition on the same piece of text. The code written was as simple as possible given what was being attempted. The below example is the test code for spaCy:

Listing 2.1: SpaCy Test Code

```
import spacy
import en_core_web_sm

nlp = en_core_web_sm.load()
text = ("A CURIOUS EXPERIMENT. A distinguished German biologist-Dr. Weisman -is
    making experiments in the way of trying to show that artificial modifications
    made "
    "in living animals may be reproduced in succeeding genera- tions. He has taken 900
    white mice, and cut off their tails with a carving knife,"
    "or some other in- strument, and he hopes in time to produce from these mice that
    will be born tailless. "
    "This is not under. taken because a breed of tailless white mice is urgently
    needed, but to establish a great fact, if it be a fact, in evolution."
    "Whatever success Dr. Weisman may attain, says a correspondent, his attempt iB
    much more on scientific lines than the theory recently set by an amateur
    naturalist,"
    "with much gravity and alleged circumstance, that the Manx or tailless cat is the
    product of a chance cross between the ordinary domestic tabby and the wild
    rabbit."
    "As the Manx cat is a perfect cat in every- thing but its tail, showing nothing of
    the structure or habits of the rabbit, and as the pairing of a long tailed
    animal"
    "with a short tailed animal would not be likely to abolish the tail altogether: as
    the rabbit is entirely herbivorous and the cat almost entirely carnivorous,
    and as "
    "the cat would be much^ more likely to eat the rabbit than to pair with it, the
    amateur naturalist can hardly be said to have brought to light a great
    scientific truth. "
    "What Dr. W eisman will do with his mice remains to be seen. ")
doc = nlp(text)
for ent in doc.ents:
    print(ent.text, ent.label_)
```

Evidently, spaCy is fairly simple to use, and with some tweaking to account for inaccuracies, it might have suited the needs of the project well. This code is an example taken from the spaCy homepage which, considering most of the work for this part of the project was named entity recognition, has led to some concerns that perhaps the wrong toolkit was chosen, and spaCy would have been a better choice given the state of the final system. Nevertheless, the NLTK is inarguably more powerful, if less intuitive for a beginner, and offers far more potential going into the future.

This is what the same code looks like using NLTK:

Listing 2.2: NLTK Test Code

```
from nltk.tokenize import word_tokenize
from nltk import ne_chunk, pos_tag

list = word_tokenize(text)
tagged = pos_tag(list)
entities = ne_chunk(tagged)
```

```
for e in entities:  
    print(e)
```

Whereas spaCy does the bulk of its processing using the imported module `en_core_web_sm.load()`, and does it all at once, the NLTK uses a collection of different modules to the same effect. Though the code appears simple here, understanding what happened at each step was difficult, particularly within the `ne_chunker`.

Expectations were high on the discovery of Polyglot. It offered the potential to process both English and Welsh articles, effectively doubling the amount of data that could be collected. Perhaps the expectation placed upon the toolkit, for a language that has approximately 900,000 speakers in Wales, was too high[10]. Although supporting Welsh in tokenization, language detection and sentiment analysis, it does not support Welsh in part of speech tagging or named entity recognition. However, that doesn't mean it couldn't still have a use in this project; the toolkit was still tested in the same way for the English language as the other two.

Listing 2.3: Polyglot

```
from polyglot.detect import Detector  
from polyglot.text import Text  
from polyglot.transliteration import Transliterator  
from polyglot.downloader import Downloader  
  
text = Text(example)  
print(text.entities)
```

Much like spaCy, it was very easy to use, but lacked the sheer capabilities and the documentation of NLTK. On top of this, it felt unnecessary to use a toolkit with such a heavy emphasis on the integration of multiple languages when it could not be used for one of the only language this project could have benefited from.

2.1.3 Testing

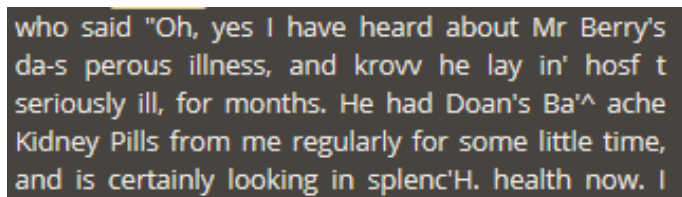
The initial plan was to test each toolkit in as similar a method as possible, with an exception for Polyglot, since it was only considered for use because of its multilingual design. Evidently, upon the realisation that Polyglot could not be used in this way, it was tested in an identical method to the other toolkits. These tests were based off the areas of focus mentioned in the previous chapter: part of speech tagging and named entity recognition. Thus, each test was conducted on the same example text from the National Library of Wales archive, thereby ensuring accurate results for working with actual data were achieved.

These tests also helped to narrow focus for development by determining what was necessary and what was not when attempting to process text. For example, when working with the NLTK, although often considered a vital part of text normalization, the removal of capital letters removes all ability the toolkit has to detect named entities. Another common text normalization technique is to remove *stop words*, words used commonly in a language to ensure sentence structure, but with little inherent meaning themselves. However, when this technique was applied to both the NLTK and spaCy, both toolkits lose a significant percentage of their accuracy.

Furthermore, these tests served as a foundation upon which the final system could be build, regardless of which toolkit was actually chosen. The code required to perform part of speech tagging and named entity recognition would already be written.

Finally, in practicing on a real article from the National Library archive, the conclusion was drawn that the text within the article would have to be manipulated in some way if the system was to have the highest degree of accuracy it could obtain. Notice in the example article ?? there are several inconsistencies with the article: a spelling mistake, or a hyphen where the

article carried a word onto the following line, for example. These cases were not considered to be majorly important, at least as far as scientist extraction goes, but the more articles that were processed, the more likely it becomes that an ill placed hyphen, or a mis-scanned letter will lead to inaccuracies that are easily avoided. Thus, it was determined that some method of checking and correcting the spelling of words would be beneficial.



who said "Oh, yes I have heard about Mr Berry's
da-s perous illness, and krovv he lay in' hosf t
seriously ill, for months. He had Doan's Ba'^ ache
Kidney Pills from me regularly for some little time,
and is certainly looking in splenc'H. health now. I

Figure 2.1: Text spelled poorly

2.2 Database

MySQL was used to build the database. Though SQLite was considered for a time it was felt that the use of a server would be required eventually, when taking into account the potential this system has to exist and grow into the future. SQLite was considered because it works well in low-memory environments and is easy to configure with no need for a server. At the current amount of information, SQLite would have been perfectly adequate, but would have severely limited the capabilities of the final system in other ways. DB-Engines Ranking[11] ranks MySQL as the second most popular DBMS internationally and names it as being behind some of the largest websites and applications. On top of this, it has extensive documentation and guides. It is fast and easy to use, and therefore was thought to be an excellent choice of database management system for this project. There was no discernable difference when comparing PostgreSQL to MySQL, so no reason PostgreSQL couldn't have been used instead.

2.3 Web App

This system was initially planned around the use of Apache and PHP to build the web app. It is a series of tools that are commonly used, so are often standard for web development. After some problems using PHP to access data from the database, a different approach was considered using Flask.

FlaskA.5 is a web micro framework for Python that provides interfaces for other commonly used extensions. In this case, extensions used include *SQLAlchemy* and *wtforms*. After some brief experimentation, which involved setting up some basic HTML pages, and accessing them through a simple web app, it was felt that Flask would be a suitable framework to build with. The aim is also to host the app externally, so an investigation occurred into different options. Heroku and PythonAnywhere were the two webhosts considered: the former was almost immediately discounted as it does not offer native support for a MySQL database, and so time went into setting up using PythonAnywhere.

Chapter 3

Phase Two: Building the system

This chapter focuses on the development of the system, looking at the three distinct parts; the information extraction, the storage of information, and how the information will be made accessible. Each section is broken down into the design, implementation and testing done for each section, as a lot of work done week to week builds upon each other, and there was no clear time period in which each section was worked on. This chapter covers from weeks three to approximately week ten though a lot of this work blurs into the work discussed in the previous chapter.

3.1 Information Extraction

3.1.1 Design

The design of the system was built upon the experimentation discussed in the last chapter. As such, although the foundation for a larger system was already in place, the design and implementation occurred simultaneously. This made for some messy designs: the flowchart shown in figure 3.1 shows how the main method of the software works, task by task. All the data that this system can process starts out in a JSON file where, upon the starting of the program, the text of each article is extracted, and broken down again and again until the system is left with the information it is looking for: human names. For the time being, this and the article metadata is the only information considered important enough to be saved in the database.

Despite Python being an object orientated language, the inclusion of classes were not considered until much later into the project than they should have been. No class diagram has been produced because the classes serve little purpose outside of code readability and improving ease of maintenance.

3.1.2 Implementation

Before the process of information extraction could commence, the articles themselves had to be collected. The National Library of Wales has an API they provided access for the project, and with this a selection of articles could be saved to work with. To narrow the rather broad scope of ‘Science’, five different fields were chosen to focus on: Astronomy, Biology, Chemistry, Mathematics and Physics. These were arbitrary choices, but it was felt that they would provide a good foundation upon which to build the system. There is no reason articles from, say, Psychology or Engineering could not be collected and processed in the same way.

The decision was also made to collect articles with a connection of some kind to one of these five fields and save the results in their own separate files, again for easy processing. The article collection code is kept separate from the information extraction, and the files stored locally because, due to the capabilities of the National Library servers, the API was only accessible

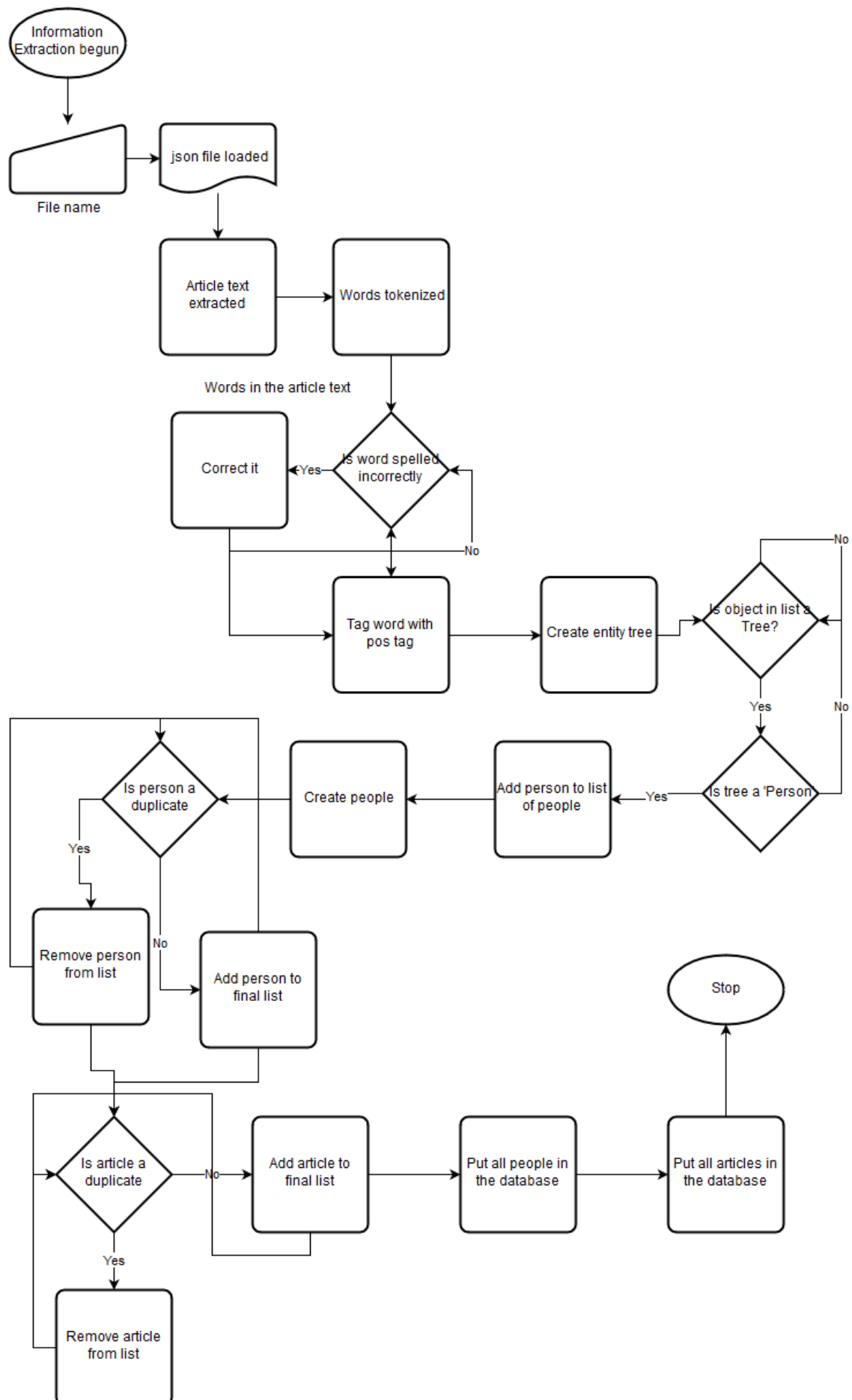


Figure 3.1: Information Extraction Flowchart

to specific IP addresses. Setting up the collected data in such a way allowed development to continue without access to this API. This method of collection came with its own problems, but on the whole has worked well.

Listing 3.1: Information Collection

```
import requests
import json

#p = "chemistry"
response =
    requests.get('http://papurapi.llgc.org.uk/?q=full_text\%3Aastronomy&fq=category
\%3A\%22News\%22&wt=json&indent=true&wt=json&indent=true&rows=999')
data = response.json()
file = open("astronomy.json", "w")
json.dump(data, file, sort_keys=True, indent=4)
file.close()
```

This is a very simple example of how the API can be used: it searches for 999 articles with the keyword *Astronomy*, returns the results in JSON format and saves as a file. This method collected approximately 5000 articles with which to build a collection large enough for the purposes of this project.

Once the articles had been collected, they had to be broken down into the information that was deemed important enough to extract: the newspaper title, the title of the article, the date upon which it was published, its unique identifying number, and the text that was printed in the article. This was all very simple to extract, because all of the information was already in JSON format, which Python can treat as a dictionary. The metadata that had to be sorted through to determine what was useful to the project was fairly extensive, though easily navigated. Figure 3.2 is an example of all of the available data about a single article. All of the information that is extracted from these files, with the exception of the article text itself, goes straight into creating a list of Article objects, which has four attributes and no class methods. This class was created to allow the attributes to be easily accessed for any particular article, which makes iterating over a list of these articles the easiest way to put compare them and put them in the database. A People class was also created for this precise reason.

It was mentioned in the previous chapter that some method of checking and correcting spelling would be useful to this project. This was done using a piece of code written by Peter Norvig^{A.2} that compares a word to a dictionary list of words, and generates the likelihood of a misspelled word being one or other of those words. It then attempts to correct the word to be the most likely alternative in two edit attempts. If it cannot, then the word stays misspelled. Without this code, it is unlikely that this software would be able to function properly, given the frequency with which spelling mistakes occur.

The key focus of this software is named entity recognition, which uses the NLTK *ne_chunk()* method to parse words into a shallow Tree structure. Its parameter is a list of part-of-speech tagged words. The chunk parser then looks at the tag attached to a word, and more specifically what the tag starts with. For certain tags, a tree will be built from that tag and its token, and then appended to the end of the larger, main tree. For other tags, they will be appended to the main tree as they are. This creates the shallow tree where child nodes that are not leaves are always entities. [12]

The task of determining the action from a newspaper article could have been approached in one of several ways. Building a classifier from the ground up was considered, and ultimately set aside as largely unnecessary; a similar result could be achieved using a list of potential 'actions', and searching through the text for any one of these. As with every other element of this project, the outcome from this approach would be less accurate than what a properly trained classifier could produce but would suffice as a starting point.

```

{
  "ancestor_UIDs": [
    "llgc-id:3380665",
    "llgc-id:3425722",
    "llgc-id:3425726"
  ],
  "ancestor_titles": [
    "The Cardiff Times",
    "1898-07-09",
    "4"
  ],
  "article_word_count": 24,
  "category": "News",
  "collection_title": "The Cardiff Times",
  "date": "1898-07-09T00:00:00Z",
  "date_day": 9,
  "date_decade": 1890,
  "date_month": 7,
  "date_weekday": 6,
  "date_year": 1898,
  "doc_type": "Newspapers-article",
  "edition_statement": "",
  "full_text": "M Reginald W Phillips, M.A., professor of biology at the University College pf North Wales, has just taken his D.Sc. at the London, University.",
  "id": "3425722-modsarticle87-3425726-1",
  "indexer_version": "348ba644f19dc32320a56c6bd5c6e27a",
  "ingest_code": "ART87",
  "lang": "eng",
  "order": 1,
  "parent_UID": "llgc-id:3425726",
  "phasecode": "apnae041",
  "provider": "WLABNL",
  "region": [
    "Wales, South.",
    "Wales, Mid.",
    "Wales, West.",
    "Welsh Borders (England & Wales)"
  ],
  "rights": [
    "UNKNOWN"
  ],
  "timestamp": "2015-06-15T12:34:18.929Z",
  "title_en": "[No title]",
  "title_en_as": "[No title]",
  "type": "newspapers",
  "useguidelines_cy": "Midi yw statws neu berchnogaeth hawlfraint yr adnodd hwn yn hysbys.",
  "useguidelines_en": "The copyright status or ownership of this resource is unknown."
},

```

Figure 3.2: Article Metadata

Articles from all the different sciences considered for the system were read to decide what could constitute an action, and a few of the results are:

- Experiment
- Lecture
- Examination
- Research

Though many articles were found to reference examination results of teenage and university students in the different sciences, these articles have been found to contain no information relevant to this collection. The *Examination* keyword could be included to offer an easy method of filtering out irrelevant results.

After all the important information has been extracted from the articles, a method of determining who is a scientist and who is not had to be implemented, since many articles mentioned upwards of a half dozen people each. The most obvious and accurate approach would have been building and training a classifier. There was, however, an easier method. Many articles were read, and how each one referred to its scientists was noted. People of note within newspaper articles are often referred to by their full name, and it appeared that scientists were no exception. Two lists of human first and surnames were compiled and, using the module *HumanName*

ParserA.4, name objects were built from the named entities. The decision was made that only people who had a title, a first and a surname within the articles could be considered a scientist, as scientists would have been some of the more ‘respectable’ members of society in the 1800’s, and were therefore more likely to be referred to as such in newspapers. This method of recognizing scientists will inevitably pick up some people who are discussed within the articles and are otherwise people of notability, but this can be tweaked to omit results that exist within a different context based on the sentences they appear in.

Three methods are shown below that work together to create people. First, a name is recognized to be real. A name could be considered ‘real’ if it appears in either the list of first names or surnames. If both parts are considered real, then *createName()* creates a *HumanName* object from these names. At this point, the name segments are in a list of tuples, where they were tokenized and tagged with the appropriate part of speech, so they must first be joined into a single string. Finally, once the name is made, the other attributes of the person can be set, the field of study and the article the person appeared in.

Listing 3.2: Creating People

```
def createName(tuples):
    name_parts = []
    for tup in tuples:
        name_parts.append(tup[0])
        name = HumanName(' '.join(name_parts))
    return name

def isPersonReal(name):
    isReal = False
    with open("C:\\backslash Users\\backslash user\\backslash Documents\\backslash
        major_project\\backslash
        what-made-scientists-in-wales-famous-and-infamous-1804-1919\\backslash
        software\\backslash textfiles\\backslash NAMES.txt", "r") as file:
    with open("C:\\backslash Users\\backslash user\\backslash Documents\\backslash
        major_project\\backslash
        what-made-scientists-in-wales-famous-and-infamous-1804-1919\\backslash
        software\\backslash textfiles\\backslash SURNAMES.txt", "r") as f:
        listOfNames = file.read()
        listOfSurnames = f.read()
        for names in listOfNames:
            if names[0] == name[0]:
                #if the first letter of the current name matches the first letter of
                the wanted name
                #makes the search through over 150000 names not take forever
                if name in listOfNames or name in listOfSurnames:
                    isReal = True
                else:
                    isReal = False
            else:
                continue
        return isReal

def createPeople(listOfPeople, fieldInt):
    lPeople = []
    for p in listOfPeople:
        names = p[0]
        aNumber = p[1]
        n = createName(names)
        if not str(n.first) == "" and not str(n.last) == "": #occasionally, a person
            had no first name. Index out of bounds error if not checked for.
```

```

if isPersonReal(n.first) and isPersonReal(n.last) and not str(n.title) == "":
    #if both names of the person are real names, and the person has a title
    study = fieldOfStudy[fieldInt] #fieldInt increased in the file for loop
    human = person.Person(n, study)
    #comment out the below line for testing purposes - Nothing in article_list
    #human.assignArticle(article_list[aNumber].id)
    lPeople.append(human)

return lPeople

```

3.1.3 Testing

No user testing has been done on this part of the system, as the end user of the collection should not ever need to use it. The only job of the information extraction program is to look for information about scientists, and put that information into a database. The tests discussed in this section can be found in `\software\information_extraction` of the project root directory, where they can be run any time any changes are made to the software.

The test files discussed in this section have been created from five articles in each field of science, producing five smaller sets of data. This has been done to test new and edited methods in the software without waiting the literal hours it takes to search through a file of 999 articles.

Unit Testing

Unit testing was began early into the project timeline: as the need for more complicated methods grew, it was recognized that unit tests were the easiest way to ensure each part of the program ran exactly as expected. Initially writing the tests was harder than anticipated, due largely to a misunderstanding of the different data structures used in the NLTK pos tagging and named entity chunker. Some methods that did not end up used in the final program were also tested, primarily because it was evident at the time of implementation that these methods could serve a purpose to a developer later down the line, when additional information can be extracted.

```

C:\Users\user\Documents\major_project\what-made-scientists-in-wales-famous-and-infamous-1804-1919\software\proof_of_concept>python nlp_testing.py
..article_word_count
category
collection_title
date
date_day
date_decade
date_month
date_weekday
date_year
doc_type
edition_statement
full_text
id
['I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams', 'I A CHEMISTRY DIPLOMA. Mr. Tom Williams']
F..
=====
FAIL: test_findText (__main__.TestTextProcessing)
-----
Traceback (most recent call last):
  File "nlp_testing.py", line 69, in test_findText
    self.assertEqual(a, ['I A CHEMISTRY DIPLOMA. Mr. Tom Williams'])
AssertionError: Lists differ: ['I A[32 chars]iams', 'I A CHEMISTRY DIPLOMA. Mr. Tom William[471 chars]ams'] != ['I A[32 chars]iams']

First list contains 12 additional elements.
First extra element 1:
'I A CHEMISTRY DIPLOMA. Mr. Tom Williams'

Diff is 738 characters long. Set self.maxDiff to None to see it.
-----
Ran 5 tests in 2.891s
FAILED (failures=1)

```

Figure 3.3: findText() unit test error

Figure 3.3 shows a key error experienced when attempting to extract the full article text from one of the json files. The test behind this error found a bug where the same article would be written every time a new instance of *“full text”*: *“”* was found. This bug may have been

missed completely until inspecting the entries in the database were it not for unit testing the method.

A full report for the tests written can be found in appendix C.4.

Integration Testing

Unit integration testing, rather than system integration testing, then followed. This was designed to test how the different methods interacted with each other - it was noticed early into development that while expected outputs for methods were tested and determined to be properly functional in the unit testing portion of the testing phase, the types of these outputs were consistently confused when moving from method to method. For example, at the point *create-People()* is called, the input for this method is a list of tuples, where each tuple is itself a list of tuples.

The interactions that were focused on in these tests are:

- Text Collection
- Part of Speech tagging, Named Entity Recognition and the creation of scientists
- Repetition of information
- Interactions with the database

Though the final item in this list might be better suited to a system integration test, much of database manipulation relies upon several methods that also need to be tested for the proper interactions. This is why it has been included here.

A full report for the tests written can be found in appendix C.4. However, an example has been included, to showcase how the format and assertion values were decided upon.

The test in question, *testTextCollection*, tests how the *findText()*, *word_tokenize()* and *correctSpelling()* methods interact with each other. It was a logical choice to combine these methods into a single test, as their role in the program is to collect and manipulate an article's text. The code the test uses is identical to the code used in the program, with the exception that it has been isolated to make sense without the context of the rest of the program. Each method was run separately on a test file, which is essentially a scaled down version of a full file, and the resulting output from each method was manually inspected for expected output types and values. In the case of the *correctSpelling()* method, an allowance had to be made for failed test: because this method uses probability, there is a fair chance that an assertion value that passes on one session of testing may fail on another. This is an issue with testing these kinds of methods, and does not reflect an issue in the code itself, nor in how it interacts with other methods. Finally, the output of the test was compared to the assertion list. A similar process was used to build the other unit integration tests.

3.2 Database

3.2.1 Design

The different kinds of information that a user could want to access was defined early in development:

- Information about a scientist
- Information about a field of study
- Science by location

Table 3.2: After Normalization

Scientist ID	Name	Field

Action ID	Action	Date	Location

Newspaper Title	Article Title	Article Date	Article ID

Scientist ID	Article ID	Action ID

The database was built using the MySQL workbench, which was chosen for no other reason that it was simple and quick to use. It was hosted on a server machine, which is the same machine that hosts the web server. This was the setup that the system was built and tested on, but for the final deployment, a server hosted externally should be implemented, to ensure that the web app can function properly. This has the added benefit of being more secure than the test environment on a local machine that also hosts the web app: in the unfortunate event of a security breach of one server, the other would be independently protected. As the information stored in the database is designed to be accessed by the public, the only security risk pertaining to the database server that has been recognized is in the potential tampering with the information stored within it.

3.2.3 Testing

The database, and the code that populated the database, was tested very simply using the same small data sets that the information extraction code was tested with. This was done because it allowed all testing to be performed at once, with each problem being identified as it popped up. A list of problems could be constructed from this, and worked on one by one until solved - this ensured that nothing was started then left incomplete.

Although very little official testing was performed on the database, outside of that mentioned above, several issues were noticed with the information being stored. First, the assumption was made that all articles would have a title, and that titles would have a maximum length of around 50 characters. Neither of these cases were true, and the title field in the Articles table had to be adapted to accept null values for a title, as well as an increase in maximum length to well over a hundred characters.

The database also required refilling several times, because there were several instances where the same article was found as belonging to multiple fields of study. This meant that there would be an attempt to store that article multiple times. Each article has a unique identifier, which is the primary key of the Articles table. There was more than one occasion where population of the database would stop because it was trying to store the same article ID multiple times. This was a problem with an easy fix, but was not found when testing with small samples because it was a rare enough occurrence that random selection of five or six articles from each field did not lead to it happening.

3.3 Web App

3.3.1 Design

The basic design of the web app was decided on fairly early into this project: appendix C.5 shows the initial design work as a simple search bar with a paragraph or two of written text describing what this project aims to accomplish, then subsequent pages show how a query would be returned. In the long term, the series of tables showing results feels a fairly simple way to display the information that has been collected, but it will suffice until such time as a more interesting display can be put together.

The web app source code requires a redesign to improve readability and make it more maintainable. Several classes are currently housed in the same file; two classes that connect to tables in the database, and are extensions of SQLAlchemy's *Model*, and one that creates a form for the homepage of the app. Because the app was produced with the aid of a Flask tutorial, it very quickly grew out of control. The following flowchart was produced alongside the app, in an attempt to better understand how the app needed to function.

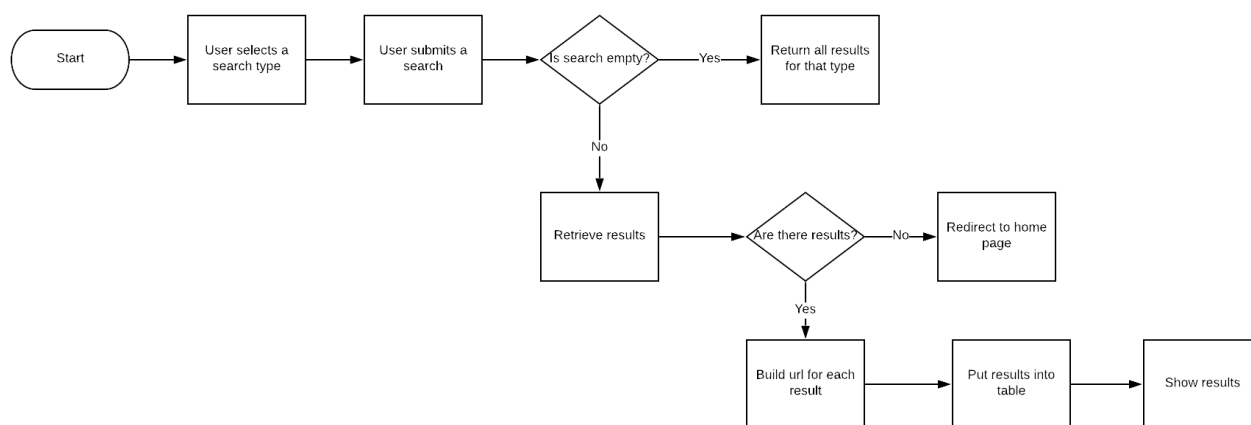


Figure 3.5: Web App Flowchart

It was with figure 3.5 in mind that the app was built, with the primary purpose of acting as a search engine for the database. A secondary purpose was also considered: a reference to the article that the information came from in the first place. This was to take the form of a link to the article in the National Library of Wales archive, where the full thing could be read at the user's leisure.

3.3.2 Implementation

The methods in the following section have been adapted from <https://www.blog.pythonlibrary.org/tag/flask/>, which was used as a guide. Though the code is largely original, the basic premise is not.

The web app implementation began with a basic html page, that loaded with the start up of the app. This page, *home.html*, offers a brief overview of what this project aims to achieve, along with a simple search form. The search form sends a query to the database, which returns a relevant list of results based on which kind of search was made as determined by the user via the drop down menu. The results are returned in a table, with a reference to the article that the searched information came from. This reference contains a hyperlink which takes the user to the National Library of Wales newspaper collection, to the article itself. These links were put together from the unique article ID.

Listing 3.3: Building URLs

```
def returnLinks(resultsList, type):
    if type == 1:
        urls = dict()
        for result in resultsList:
            urlParts = re.split("-", result.articleID)
            url = '/'.join((urlParts[0], urlParts[2], urlParts[3]))
            urls[result.articleID] = url
    elif type == 2:
        urls = dict()
        for result in resultsList:
            urlParts = re.split("-", result.id)
            url = '/'.join((urlParts[0], urlParts[2], urlParts[3]))
            urls[result.id] = url
    return urls
```

In the above section of code, the *type* determines what kind of search is being performed - one pertaining to a scientist, or an article. This difference is made in reference to the article ID for both: stored as the *articleID* in scientists, but *id* in articles. The URL parts shown above are sections of the id that create the url when reading an article on the archives. After breaking down the ID into parts by hyphens, only some of the parts are relevant, so those parts are then rebuilt into part of a url. Finally, the url-suffixes are returned as a dictionary of article IDs and the urls - it was believed at the time that this would make turning the results into a hyperlink easier. These can be attached to the url root, which is the same for every article.

Once the query results have been returned, the next step is to put the results in a table and display the table. Two *tables* classes were built using *flask_table*, one for scientists and one for articles, which displayed different results depending on the type of search made. These tables, now filled with the query results, are sent to the *results.html* file, which displays the results.

A key area of difficulty was in deciding how to host the web app. One of the final decisions made for this project was in deciding whether to host the app locally for the project life span, which would be the simpler but less professional option, or in attempting to host externally. The latter option had the added benefit of being accessible from anywhere at any time, which is a vital part of any application designed around use by a larger group. The app was not built with external hosting in mind, so setting up the system to be hosted externally was difficult. In particular, because this app should be accessible from anywhere at any time, the MySQL database had to be hosted externally as well, which meant filling another database. As of the writing of this report, the app still cannot be hosted externally, though it is still an aim of the project and this feature is still in development.

3.3.3 Testing

As detailed in appendix C.4, the website was tested to ensure that different searches return different results that match the search. The same set of search terms were used for each category of search:

- The name *Robert Smith*.
- A partial search based on the name above, *Smi*.
- The field *Chemistry*.
- Part of an article title, *Professor Mag*.
- A full article title, *Death of Professor Magnus*.

These tests passed, and helped make the app easier to use in determining that some method of informing the user that their search was invalid would make their experience using the web app more pleasant. Some of the tests in the testing summary allude to the implementation of other features outside of those already discussed in this report as being useful to a user: in particular, a method of filtering results that have already been found would help navigate those searches that lead to many search results from many different fields, such as searching for an article title.

Manual testing was also useful in determining where the issues with the hyperlinks back to the National Library archive lay: through a series of print statements and trial and error, it was determined that the issue is not with building the URL themselves, nor with redirecting a user to the web page, as both of these features work separately. The issue is in connecting the URL to an individual article ID, and doing this for each article. It is suspected that this issue comes from inexperience in using Flask.

Chapter 4

Evaluation

This chapter discusses an overall evaluation of the project. It includes how I utilized my chosen methodology to its full potential, the work actually completed in the time frame, and my own understanding of the project.

4.1 What I have learned

Before starting this project, I knew nothing about natural language programming. I'd never heard of it, let alone considered what its applications might be or how it could be implemented. In order to complete this piece of work, I had to learn how language works; what different parts of speech were and what they were for, what an entity was and how it differed to a named entity. I also needed to understand what natural language could be used for, that I knew what techniques would be useful to apply to this project and what would not. Towards that end, I turned to web articles designed as a casual introduction into NLP.

Though I had done some work in Python previously, this was my first major piece of work in the language. I suspect that led to me making many mistakes, namely in not utilizing classes until far later in the project than I should have, in writing inefficient code and taking far longer in writing tests than was reasonable. Despite the issues I have had using Python, I do not regret its use as the primary language for the system: I suspect there would have been far more issues had I used Java or C++.

This project introduced a lot of firsts for me: NLP, working with an API, automating database population, using Flask and SQLAlchemy, and using Python's UnitTest.

Flask was particularly difficult to really use properly, despite following a tutorial for most of it, as I didn't understand why I was writing the code that I was until a week or two later. This made it difficult to know where bugs were coming from and, I suspect, is a contributing factor to why the final web app feels underwhelming. In future, I may just stick to a tutorial to provide an introduction to a technology, and use its documentation as the primary source of assistance.

The Python Unittest module proved really easy to use. I did not focus as much on learning what the module was capable of, preferring to only learn what I needed to get by. This is evident in the way every unit test, and even the integration tests, are implemented in exactly the same way.

4.2 On the Chosen Methodology

As described in Chapter One, the chosen methodology for this project was scrum, with some kanban elements where needed. Kanban did not prove as useful to managing the system's workload as was initially expected and was soon cast aside in favour of a more traditional

scrum based system. This was the right decision, as the weekly iterations provided invaluable flexibility to the project organization, without which this system might never have been built to its current level of completion. The amount of work aimed for each week was often severely underestimated to the point where, after the weekly sprint meeting every Monday morning, the work prioritised for that week would be finished by Wednesday or Thursday. This led to an adjustment in how each iteration was organised. Early sprints focused on a single task and its design, implementation and testing, but it was quickly realised to be an inefficient way to work: although a deeper look into any particular feature could not be done until the next iteration, an entirely independent aspect of the system could also be designed, implemented and tested. Had this adjustment taken place sooner in the project timeline, instead of a fixation on finishing one aspect or element of the system, then there may have been more time to implement more features, and the system may have ended up looking and feeling a lot more refined to use.

4.3 Deviations from the Plan

Due to poor time management, emphasis was placed on implementing functional requirements as best as possible, at the cost of not implementing them all. Therefore, while the system can allow for several different searches, it cannot handle dealing with multiple queries at once. While there is the infrastructure to provide a reference link to where a specific article came from, the actual implementation does not work. It was also sorely overestimated how much information on any one scientist would be available from newspapers alone, which meant functional requirement 1.4: *Summarize the information about a specific person in cases where a specific name is returned or searched for* was not feasible within the constraints of this project.

A key aspect that was to be implemented was finding and extracting what scientists did to end up in newspapers: in the beginning of the project, I had hoped to build this into the collection to show some of the ways in which Welsh people contributed to the scientific advancements of the age. It could also have been used to make the collection more interesting to a wider audience; a collection showing scientists in negative ways, as criminals, as chemists accidentally causing fires or explosions, as being the kinds of people who could do anything they wanted simply to see what would happen, could be a collection that excites the minds and imagination of young people, for whom science, particularly that long ago, can seem stuffy and dull. Unfortunately, this was not a feature that I could implement within the constraints of the project: I got so focused on producing a working system, and invested so much time into learning new tools and technologies, that there wasn't enough to devote to this part. I truly regret not working on this feature, and if I have the chance to continue work on this project, it is something I will prioritize.

Chapter 5

Bibliography

- [1] E. Novoseltseva, “How Natural Language Processing Is Used - DZone Big Data.” An introduction to what Natural Language Processing is, this article was the starting point for the preparation of this project.
- [2] P. Jackson and I. Moulinier, *Natural Language Processing for Online Applications: Text retrieval, extraction and categorization. Second revised edition.* Amsterdam, NETHERLANDS: John Benjamins Publishing Company, 2007. This book provided an insight into different areas that natural language processing can be used.
- [3] “The Eight Parts of Speech - TIP Sheets - Butte College.”
- [4] “Language.” This source looks at how Google uses natural language processing, and related areas they are currently researching.
- [5] J. Chen, “Algorithmic trading.” This web page discusses and defines algorithmic trading.
- [6] “For a welsh scientist,” Feb. 1895. Provided is a link to an article on the National Library Archives, provided as an example of what the data looks like from that end.
- [7] “Compare nltk and spacy popularity.” This was useful in determining which of the two main toolkits are more popular generally.
- [8] S. Bird, E. Klein, and E. Loper, “Natural Language Processing with Python,” July 2015. This book focuses on using the NLTK to aid the user in building chunkers, but was found to be helpful in understanding how the tools worked.
- [9] “What are the advantages of spacy vs nltk.” This forum discusses the use of NLTK and spaCy, and reasons one might be used over the other. This source was found useful, as it looks at both tools in a casual, user friendly way.
- [10] “Welsh language,” Apr. 2019. Page Version ID: 892740050.
- [11] “Db-engines ranking.” A source for the popularity of different databases, comparing the use of each one world wide.
- [12] Loper, “Source code for nltk.chunk.named_entity.” This web page was used to understand how the NLTK NE_chunk parser works.

Appendix A

Third Party Code and Libraries

A.1 NLTK

[1]: <http://www.nltk.org/>

This is the toolkit that the whole project depends on. It provided the tools needed to break down the articles and extract the required information from them. The NLTK is licensed under the Apache License V2.0.

A.2 Spell Corrector

[2]: <http://norvig.com/spell-correct.html>

Code from this website was used to correct spelling mistakes in articles that were vital for proper information extraction, without which this project wouldn't have been half as successful as it is. At the time of writing, the website above seems to be inaccessible, but is the same as the one used originally. The same code can also be accessed here: <https://github.com/norvig/pytudes/blob/master/py/spell.py>. This code is licensed under the MIT licence. An edit was made to the code to allow for a third iteration of searching for potential corrections: though this was ultimately not used for this project, it has been left in the source code in case it becomes necessary for future development.

A.3 Spell Checking

[3]: <https://stackoverflow.com/questions/40188226/nltk-spell-checker-is-not-working-correctly>

Code here was used to implement a checker for whether or not a word is misspelled. This was used to prevent the spell checker searching every single word in every single article for misspellings. Code on stack overflow falls under the Creative Commons BY-SA 4.0 license <https://creativecommons.org/licenses/by-sa/4.0/>.

A.4 Human Name Parser

[4]: <https://nameparser.readthedocs.io/en/latest/>

This module was used to build human name objects from strings, simplifying the process of determining whether or not a person was real and, if they were real, whether or not they were a scientist. This module is available for use under the LGPL-2.1 license.

A.5 Flask

[5]: <http://flask.pocoo.org/>

Flask was the framework used for the web app. It is licensed under a BSD license: [http:](http://)

[//flask.pocoo.org/docs/1.0/license/](http://flask.pocoo.org/docs/1.0/license/)

Flask interfaces used in this project are *flask_sqlalchemy* to connect with SQLAlchemy and *flask_wtf* with FlaskForm.

Appendix B

Ethics Form

12/03/2019

For your information, please find below a copy of your recently completed online ethics assessment

Next steps

Please refer to the email accompanying this attachment for details on the correct ethical approval route for this project. You should also review the content below for any ethical issues which have been flagged for your attention

Staff research - if you have completed this assessment for a grant application, you are not required to obtain approval until you have received confirmation that the grant has been awarded.

Please remember that collection must not commence until approval has been confirmed.

In case of any further queries, please visit www.aber.ac.uk/ethics or contact ethics@aber.ac.uk quoting reference number **12316**.

Assesment Details

AU Status

Undergraduate or PG Taught

Your aber.ac.uk email address

tab23@aber.ac.uk

Full Name

Natalie Brown

Please enter the name of the person responsible for reviewing your assessment.

Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment

rrz@aber.ac.uk

Supervisor or Institute Director of Research Department

Appendix C

External Documents

C.1 Requirements

Requirements

tab23

2019-02-14

1 From the User

I would like to use this software to be able to look for scientists who lived in Wales, and who made some amazing contributions to science. Specifically, I would like to be able to find relevant results if I were to search for several different keywords, like a name, field of study, location or a date, or any combination of those. I would also like to be able to sort the results as best I see fit, whether that's alphabetical based on title, or date of publication, or how relevant they are to my keywords. Finally, this software should allow me to read a summary of a specific scientist, as well as provide other locations where I could find out more information if I wanted to.

2 Functional Requirements

2.1 User interface

The user interface in this system is a webpage that contains a simple search bar and, once a query has been submitted, displays extracts of information pertinent to the query. The default ordering for this information will be date of birth of a person, or the date of which an event occurred.

1. Search for a specific keyword, and return all instances of that keyword
2. Search for several keywords, and return instances where those keywords overlap
3. Allow the results to be reordered to best suit the users' needs.
4. Summarize the information about a specific person in cases where a specific name is returned or searched for
5. Provide sources for where the information came from originally, specific to the news article that it was discovered from

2.2 Data storage

All the data gathered will be stored in a database.

1. Store a list of names, dates, and the fields they worked in
2. Return this information when queried
3. Also store events in the person's life, and dates of when they occurred - birth, marriages, children and death, if these can be found.
4. Format general information about a person so that it is returned only when a request for that person specifically is made in some way.

5. Store as many references to where this information has come from as possible, particularly if it did not come from the National Library of Wales newspaper archives

2.3 Information Extraction

This part of the system will search the National Library of Wales newspaper archives, categorize the data found and decide what is relevant enough to store and what is not.

1. Search for names, dates and specific keywords as according to a separate text file
2. Once names are found, search for other articles bearing the same name, and search that text as well
3. Store information found in the data storage system
4. Correct any egregious spelling or scanning mistakes

C.2 NLP Toolkit Comparison

Investigation into Natural Language toolkits

Taylor Brown (tab23@aber.ac.uk)

February 12, 2019

1 Introduction

This report looks at three different toolkits designed for Natural Language Processing (NLP) in Python. These toolkits are:

1. Natural Language ToolKit (NLTK)
2. SpaCy
3. Polyglot

These three were chosen for various reasons. NLTK is the toolkit that has been around the longest: it was first produced back in 2001, and is the toolkit used by academics for its extensive documentation and sheer power. SpaCy is much newer to the NLP scene, and is self confessed to be designed for developers over research[1]. SpaCy offers support in several languages, including German, Italian and Greek as well as English, and offers a very basic tutorial for those new to both spaCy and NLP. Finally, Polyglot is the newest toolkit of the three, and was designed for international use. It was built to support massive multilingual applications, and boasts support for over a hundred languages in various areas of NLP.

This report will compare not only the ease of use and accuracy of the toolkits, but also takes into account the potential needs of the system in mind beyond the current project duration. These factors will be used to decide which toolkit should be used for information extraction in this system. The ease of use test for the toolkit is simple: implement named entity recognition. Since this task is a key feature of the information extraction section of the system, this was recognized to be the best way of determining which toolkit would be best suited to the system.

Each toolkit was tested with the same piece of text, 1, that was taken directly from an article in the archives that are used in the main system. This helps to determine how each implementation will respond to the actual data.

Figure 1: Testing text

A CURIOUS EXPERIMENT. A distinguished German biologist—Dr. Weisman—is making experiments in the way of trying to show that artificial modifications made in living animals may be reproduced in succeeding generations. He has taken 900 white mice, and cut off their tails with a carving knife, or some other instrument, and he hopes in time to produce from these mice that will be born tailless. This is not undertaken because a breed of tailless white mice is urgently needed, but to establish a great fact, if it be a fact, in evolution. Whatever success Dr. Weisman may attain, says a correspondent, his attempt is much more on scientific lines than the theory recently set by an amateur naturalist, with much gravity and alleged circumstance, that the Manx or tailless cat is the product of a chance cross between the ordinary domestic tabby and the wild rabbit. As the Manx cat is a perfect cat in everything but its tail, showing nothing of the structure or habits of the rabbit, and as the pairing of a long tailed animal with a short tailed animal would not be likely to abolish the tail altogether: as the rabbit is entirely herbivorous and the cat almost entirely carnivorous, and as the cat would be much more likely to eat the rabbit than to pair with it, the amateur naturalist can hardly be said to have brought to light a great scientific truth. What Dr. Weisman will do with his mice remains to be seen.

2 Natural Language ToolKit

This toolkit was complicated to get started with: because the toolkit is so big, many modules have to be imported for a series of different, but very related tasks. Though the toolkit seems complicated - largely attributed to its size - the actual implementation is fairly easy and intuitive. Break down sentences into words -; tag words with their associated part of speech -; chunk into entities.

This toolkit took 12.01 seconds to run, and accurately depicted all of the named entities in the text, though the results printed were hard to interpret compared to the other toolkits. It recognised the name of the scientist discussed, Dr Weisman, though only recorded it as a named entity once. It did not detect the Manx cat as any kind of named entity, and it recognised German as a geo-political entity, the label used for nationality amongst other things. The headline was incorrectly labeled as an organization, but this could be as due to the capitalization as to an issue with the chunker and, indeed, when the headline is changed to normal capitalization, the label vanishes.

3 SpaCy

The installation of spaCy was found to be much harder than NLTK, in that where spaCy is a module that requires Visual Studio where NLTK did not. It is very simple for beginners to use; an in depth tutorial is available in the same place as its installation guide. This also includes a detailed explanation of how techniques such as tokenization, part-of-speech tagging and named entity recognition can be implemented. Coincidentally, these are the aspects required for the comparison, and what would be used in the information extraction software, making this tutorial the most helpful of the three.

This toolkit took 30.36 seconds to run on the testing text, and did not miss a named entity in the above text. Several words were recognised as being a named entity incorrectly, such as the Manx species of cat, or the word ‘much’², which is just a mistyped word and is not tagged when spelt correctly. As the final software will spellcheck words, this should not be an issue. Nevertheless, the accuracy is less than desirable, and the code takes far longer to finish than would be preferable.

4 Polyglot

An exception must be made to the testing criteria for polyglot: it was only considered against the other two toolkits because of the potential benefits of working in both Welsh and English for a collection that is designed to highlight the best and brightest of science in Wales. Unfortunately, though polyglot supports 16 different languages in its part-of-speech tagger, and 40 for its named entity chunker, it does not support Welsh in either of these. This was enough to prevent polyglot from being the toolkit of choice. For good measure, it was still tested in the same way as the other two, and found to be lacking in both power and accuracy, though it is fairly easy to use. It appears that the strength of this toolkit really does lie in its support for multiple languages perhaps even at once, as it appears to offer transliteration. Perhaps, if it expands services to include less common languages like Welsh, it would have a place in this project.

This toolkit was significantly faster than the other two, completing its run in just 4.49 seconds. It did, however, only recognise entities that are people, incorrectly identifying the ‘-’ that occurs before the first ‘Dr Weisman’, and not recognising ‘German’ at all. This suggests that accuracy has been sacrificed for speed in this toolkit, and were it to be used for this project, a classifier would have to be built from the ground up.

5 Conclusion

The final choice of toolkit for the information extraction section of the system is NLTK. Though both polyglot and spaCy have their strengths, the former is simply too inaccurate for a system in which a certain level of imprecision must already be taken into account. SpaCy, meanwhile, could be considered an ideal if it did not take significantly longer to produce the same output as the NLTK. Another factor that led to the NLTK being the right choice for this system is its design. It would be fairly easy to build and train a classifier using the NLTK should the system later require one. The only exception would be if polyglot were to offer part of speech tagging and named entity recognition support in Welsh, or a good chunker were built by a native Welsh speaker, which would be worth the effort of improving the accuracy of the English classifier.

References

- [1] <https://spacy.io/usage/spacy-101> What spaCy isn’t: 2019-02-09

C.3 Database normalization

1 Normalization

1.1 First Normal Form

Name	Field	Action	Date	Location	News.Title	Art.Title	Art.Date	Contents	Art.ID

1.2 Second Normal Form

Scientist ID	Name	Field

Scientist ID	Action	Date	Location

Newspaper Title	Article Title	Article Date	Article ID

Scientist ID	Article ID

1.3 Third Normal Form

Scientist ID	Name	Field

Action ID	Action	Date	Location

Newspaper Title	Article Title	Article Date	Article ID

Scientist ID	Article ID	Action ID

C.4 Testing Summary

Summary of Testing

Taylor Brown: tab23

April 30, 2019

1 Introduction

This document summarises the various tests that were performed on the system built in the project *A collection of scientists in Wales between 1804 and 1919 built using Natural Language Processing techniques*. The system is comprised of three different parts, which make up the three different sections in this report.

This system looks at the building of a collection of scientists from newspaper articles published approximately 100 years ago. Articles are downloaded and saved as local files. The article text is analysed and scientists are extracted. Scientists, articles and the different actions that scientists did, the reason they are in the newspaper, are all stored in a database.

The user interface of this system is a web application, built with flask, where a user can search for a name, title of an article or field of study. These searches query the database, and return a table filled with the results. Any search which returns an article ID returns the ID as a link to the article on the National Library of Wales newspaper archive.

1.1 Testing Scope

Covered here are the unit tests for the information extraction software, manual inspection of the information in the database, and webtesting for every aspect of the web app.

1.2 Iterations

The project was developed in a series of sprints, each one improving the system slightly in a different way. Because of this, testing was done regularly and in small batches.

1.2.1 Information Extraction

Unit tests were written for this section whenever a new method was produced that required tests. These tests were ran regularly, and used to determine that a method functioned in the way it was thought to function. They were also updated whenever a method was changed in any way - this includes slight changes to structure as well as complete rewrites. After a change was made, a method was not left alone until it passed its unit test.

1.2.2 Database

The database was manually inspected whenever more data was added to it. These inspections were made to ensure that the data populating the database was of the form and format expected, and checked that newspaper and article titles were of an appropriate length for the constraints set. Given that new data was added in batches regularly, this inspection occurred often.

1.2.3 Web App

Though no unit or system tests were written for this part of the project, the web app was tested manually for functionality - in particular, to ensure that any changes made to the application did not have an adverse affect on reading from the database and displaying the results, since that is the key role of the app.

2 Information Extraction Testing

Unit tests were written for the below list of tasks that this software was designed to complete. These tests **must** be passed before further development can occur. When changing one of these methods in any way, the test must be changed accordingly.

- Documents break down into articles
- Articles break down into text
- The spelling of words is corrected
- Words are tagged with correct part of speech
- Entities are identified
- Names are recognized as names
- People are created

The integration tests for this system look at how various parts of the software work together. The system is broken down into a collection of methods that work together to achieve a defining point in the software. These points are:

- Text collection: tests the `findText()`, `word_tokenize()` and `correctSpelling()` methods together
- People: tests the `tag()`, `entities()` and `createPeople()` methods.
- Duplicity: tests that mentions of the same person in the same article are removed like they should be. Tests that articles that appear in more than one field of study are removed.
- Database: tests that the database is populated with results correctly.

The difference between these tests and the unit tests are that, here, it is already assumed that each method works as expected: integration testing would not have commenced unless the methods passed unit testing. Each test is written as a black box: there is an input of some kind, the methods are run sequentially without external interference, and the output is compared to expected values.

The text collection test uses the `correctSpelling()` method, which causes the test to fail approximately 50% of the time. This is a known issue: if a single test fails with the error *AssertionError: Lists differ: ['POR[126 chars] 'felix', 'Board', 'School', ', ', 'has', 'won'[931 chars] '.'] != ['POR[126 chars] 'felon', 'Board', 'School', ', ', 'has', 'won'[931 chars] '.']*, it is to be ignored.

3 Database

The database was filled with small, sample sets of data initially. These sets were extracts taken from the json files already gathered to test the information extraction, and allowed for any obvious bugs in the population code that could be spotted, fixed and reran without waiting for hours at a time. In this way, it was discovered that the article title field was too small to allow for all potential titles. This field was doubled in size twice before this particular error stopped appearing.

There is only so many issues that will be made known using small sets of data, and one such problem that remained undetected throughout the testing of the database was the occasions where the code would attempt to store an article or person twice. After some inspection of the specific articles causing these problems, the fault was determined to lie with the way articles are written: multiple mentions of the same person in the same article caused them to be saved as the same scientist, and an article appearing in multiple fields of study meant they were saved in both article collection files. This wouldn't have been a problem if the primary key for the articles table was generated, but this field used the article id number, given to the articles by the National Library archive, so this caused another bug. Because both bugs were exceedingly rare, there was a very small change that the test data would trigger them; unfortunately, they did not, and the whole program had to be run again from the start.

4 Web App Testing

Various different aspects of the web app require regular testing to ensure that they work consistently.

- The different kinds of search that can be made
- The handling of unexpected search strings
- The return of information
- The format of returned information
- Navigation around the web app

As seen in 1, the different searches were tested in a variety of ways. Because each search is handled in the same way within the flask app, it would be fair to assume that each search within a selected field that is deemed 'incorrect' would be handled the same way: Every potential search has been tested to ensure that this is the case.

- The example name is *Robert Smith*
- The example field is *Chemistry*
- The example title is *Death of Professor Magnus*

Table 1: Testing the searches

Test ID	Section	Test	Expected Result	Outcome
T-S1:1	Name	Search for Full Name	Results displayed that exactly match the search term: <i>Robert Smith</i>	Seven results displayed for a Mr Robert Smith, his field and the articles the name appears in.
T-S1:2		Search for partial name	Results displayed that match the search term <i>Smi</i>	Sixteen results displayed for various “Smiths”. The same results as T-S1:1 appeared as well.
T-S1:3		Search for field of study	Redirect to home page	Redirect back to home page
T-S1:4		Search for article title	Redirect to home page	Redirect back to home page
T-S2:1	Field	Search for full field	Results displayed for every entry with that field	List of people displayed with a connection to chemistry
T-S2:2		Search for incomplete field	Results displayed that match the field that should have been typed (Test uses <i>chem</i>)	Results displayed that matches the full search meant.
T-S2:3		Search for unavailable field	Redirect to home page	Redirect to home page
T-S2:4		Search for name	Redirect to home page	Redirect to home page
T-S2:5		Search for article title	Redirect to home page	Redirect to home page
T-S3:1	Article Title	Search for full article title	Results displayed for all articles with that title	Three articles returned that match the searched title
T-S3:2		Search for partial title	Results displayed for full titles that match the search term <i>Professor Mag</i>	The same three articles returned for the partial match as for T-S3:1
T-S3:3		Search for name	Display list of articles with name in the title, or redirect to home page if none match.	Redirect to home page
T-S3:4		Search for field of study	Display list of articles with the field of study in title, or redirect to home page if none match.	Displays list of articles with chemistry in title

T-S3:3 and T-S3:4 suggest that some sort of system designed to filter results may be required should the amount of data made accessible by this system grow much more: article titles can have any word combination in them, so finding something specific in a large result set could be difficult. Though all of these tests have passed, it was concluded that some method of informing the user that their search could not find any results might be useful, to make the transition between submitting the search and redirecting back to the home page less jarring. A dialogue box might be the most useful way to do this.

Another test that is run manually is activating several of the hyper links after search results are displayed, ensuring that they link back to the correct article. As of writing, this feature does not work. Why this does not work is understood.

C.5 Web Application Design

	Search
--	--------

...Couple of lines here about the point of this site, and the kinds of information accessible here...

<name only>	Search
-------------	--------

Name	Field of Study
------	----------------

Name attribute contains all names that match the name searched. The name itself is a link to a full list of all the articles a particular name has appeared in.

Field of Study attribute is the area(s) of science a person may have worked in or contributed to.

<location only>	Search
-----------------	--------

Location	Name	Action	Article
----------	------	--------	---------

Location attribute contains only exact matches to the searched entry

Name contains the name of the scientist who worked in that location

Action contains what occurred at that location

Article contains a reference and hyperlink back to the article on the NLW archive website

<field of study>	Search
------------------	--------

Field of study	Name	Location
----------------	------	----------

Field of Study shows a the field of study that was searched for, assuming it was a valid search

Name shows the name of a scientist that worked in the field

Location attribute contains contains the location that the scientist worked in