Capstone Report

I, Issar Arab, affirm that I have applied ethics to the design process and in the selection of the final proposed design. And that, I have held the safety of the public to be paramount and have addressed this in the presented design wherever may be applicable.

Issar Arab

Approved by the Supervisor

Dr. Violetta Cavalli Sforza
Acknowledgements

I am using this opportunity to express my gratitude to everyone who supported me throughout my capstone Project. For this occasion, I would like to thank:

Dr. Violetta Cavalli Sforza, who is my capstone supervisor. She helped me and coached me during my capstone project by giving me feedback and tips, and by guiding me with valuable advice through my capstone. I want to thank her for her professional spirit and valuable guidance.

Dr. Yassine Salih Alj, who is the capstone coordinator of the SSE school. I am thankful for his constructive criticism and friendly advice concerning my capstone project.

My final thoughts go to my family and friends, for their priceless moral and financial support throughout my years at Al Akhawayn University, and without whom this project would not have been a success.
# Table of Contents

Acknowledgements .............................................................................................................................. i

Abstract .................................................................................................................................................. iii

1. INTRODUCTION ............................................................................................................................. 1  
   1.1. Project context ............................................................................................................................. 1  
   1.2. STEEPLE Analysis ...................................................................................................................... 1 

2. METHODOLOGY ............................................................................................................................ 3 

3. SYSTEM ARCHITECTURE .................................................................................................................... 5  
   3.1. Text Extraction ............................................................................................................................ 5  
   3.2. OCR Technology .......................................................................................................................... 6  
   3.3. Processing the Extracted Text ...................................................................................................... 8  
   3.4. Applying the Process on the Whole Document ........................................................................... 19  
   3.5. WordNet Usage ........................................................................................................................... 22  

4. Conclusions ......................................................................................................................................... 27 

References ............................................................................................................................................. 28 

Appendix A.1: Pdfminer3k installation ................................................................................................. 29 

Appendix A.2: Script for Automatic Execution of the PDFMiner Command on All the PDF Files in the Folder .................................................................................................................................................. 30 

Appendix A.3: Script for Automatic Execution of the Tesseract-OCR command on All Locked PDF Files ...................................................................................................................................... 31 

Appendix A.4: Script for Automatic Processing of All Extracted Text Files ........................................ 32
Abstract

On average 240 students graduate each year from AUI from the three different schools and in different fields. As a requirement to graduate either with a bachelor or a master degree, students must work on a project and defend it by the end of their academic journey at the university.

Before picking the topic of my capstone design, I started looking at the previous research work done at AUI. The available library resource allowing to check for reports and work done by students is not that helpful and complex to use. Therefore, I decided to implement a simple to use Thesis Search Engine that will have as an added value the following:

- First, automating the indexing of theses, including master theses and project reports, internship reports, and capstones, using Natural Language Processing technologies.
- Second, creating a user friendly and a responsive web application for students. The system can be described as a one-box text field to enter a query and a search button to retrieve relevant documents on the topic of interest.
- Third, advancements of research are based on previously conducted work and results throughout history. This is one of the goals of this system, which is making the retrieval of previous work done by AUI students easy to access. This will enhance the research quality of the university.

The following sections of this report will explain in more detail the system implemented, as well as the technologies used to make it a successful product.
1. INTRODUCTION

1.1. Project context

The main objective of this capstone project is to design and implement a Search Engine that will allow any user to enter a query and retrieve relevant documents from the pool of AUI theses (internship reports, capstones, Master project reports and theses from the three different schools) in the DB.

1.2. STEEPLE Analysis

STEEPLE Analysis is "a model for strategic decision that takes into consideration seven main macro-environmental factors in the activity of analysis, assessment and forecast of the impact of the decision to be made" [1]. STEEPLE stands for societal, technical, environmental, ethical, political, legal and economic factors. In this project, we adopted to STEEPLE model to analyze and assess the potential impact of our solution. The macro-environmental factors related to the project include societal, technical, environmental, ethical, political and economic factors.

- **Technical**: The manner in which the University and the Schools give access to capstone, Master’s Theses or Master’s projects reports do not make it easy to look for reports that address topics similar to those of interest. The emergence of the field of Natural Language Processing has provided tools and technologies that allow text extraction as well as building language models to process and manipulate natural language. The solution will allow retrieval of relevant documents through a one-box search field.

- **Ethical**: There are no substantial ethical concerns in this project. The documents on which the search engine operates are public documents that should be viewable by anyone.
• **Societal, Political, and Economic:** The system developed has an indirect impact on these factors, though the impact may not be obvious. Advancements of research in all fields are based on previously conducted work and results throughout history, and one of the goals of this system is to make access to previous work done by AUI students on a specific topic easier. This will enhance the research quality of the university in the three different schools, impacting the technical, societal, political and economic spheres.

• **Environmental:** The project has very limited negative environmental impact insofar as all material is stored and accessed digitally, so the environmental impact is only that of the electricity used to run the system. The reports being accessed are already being stored digitally.
2. METHODOLOGY

As the name says, the project is a Search Engine that allows any user to enter a query and retrieve relevant documents from the pool of AUI theses and projects. The estimated number of alumni is around 3400 students last year, this is the same number as the documents contained in the pool of reports including internship reports, capstones, Master’s projects and theses from the three different schools with an increase of 240 new reports per year. To achieve its purpose, the Search Engine makes use of NLP technologies, such as N-grams\(^1\), k-skip-ngrams\(^2\), Part of Speech (POS) Tagging, and parse trees, as well as the popular lexical database of English, WordNet.

We briefly introduce the project below, using a three-part description:

a. PDF file content extraction:

In this phase, I built a script that I ran over the 3400 documents, which range from 30 pages to 214 pages. The script runs sub-processes to extract text from a pdf document using PDFMiner\(^2\) and copy its content into a text file. The other goal of the script was to organize the structure and the naming of the files created, as well as of the original pdf documents. The script gives a predefined naming structure to the extracted text file and creates a new folder, with the same name, to which the original document will be moved. This structure is meant for easy look up of the original file once a user clicks on a retrieved document on the result page of the application.

---

\(^1\) In the fields of computational linguistics and probability, an **n-gram** is a contiguous sequence of n items from a given sequence of text or speech. The items can be phonemes, syllables, letters, words or base pairs according to the application. The n-grams typically are collected from a text or speech corpus. When the items are words, n-grams may also be called shingles. An n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram". Larger sizes are sometimes referred to by the value of n, e.g., "four-gram", "five-gram", and so on (https://en.wikipedia.org/wiki/N-gram).

\(^2\) In the field of computational linguistics, in particular language modeling, **skip-grams** are a generalization of n-grams in which the components (typically words) need not be consecutive in the text under consideration, but may leave gaps that are skipped over. They provide one way of overcoming the data sparsity problem found with conventional n-gram analysis. Formally, an **n-gram** is a consecutive subsequence of length n of some sequence of tokens \(w_1 \ldots w_n\). A **k-skip-n-gram** is a length-n subsequence where the components occur at distance at most k from each other (https://en.wikipedia.org/wiki/N-gram#Skip-gram).
After skimming through the resulting extracted files, I remarked that no content was extracted from documents that are locked and act as images rather than text. To solve this problem, I opted for the use of an OCR technology. To depict the document that could not be extracted automatically, a script was implemented to go over the resulting files and check if the file size was less than 10KB. When such a file was found, I used the Ghostscript tool to extract each page into an individual picture file. Then, I extracted the content of each page using Tesseract-OCR [3] and appended it to the resulting file.

b. Text Pool Processing:

After all the content was extracted from the reports, NLP technologies were applied on the resulting text using NLTK to generate an indexed file for each document.³

Each step in the indexing of the files is explained in more detail in the sections below.

c. Interface Implementation:

The last step is the interface implementation. The interface allows any user to enter a query and get a list of relevant documents that address the query. If no document is found that matches the query, both a message of apology and a list of suggested concepts will pop up. This list of concepts, sorted lexicographically and derived from WordNet, has as purpose to guide the user in his search by making more successful search queries that will retrieve relevant documents.

---

³ NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum. (http://www.nltk.org/).
3. SYSTEM ARCHITECTURE

Figure 1 shows the flow through the system of a PDF file, in order to be automatically indexed. After indexation, the file is stored in the pool that is the main data resource of the search engine. Each step is described below.

![Diagram of the indexing process]

**Figure 1:** The flow of indexing a PDF file

3.1. Text Extraction

For text extraction, the tool that was used for that purpose is PDFMiner. PDFMiner is a tool for extracting information from PDF documents. It can extract text from PDF files as HTML, SGML or “Tagged PDF” format. It extracts all the text that is represented as ASCII or Unicode strings. It cannot recognize text drawn as images, which would require optical character recognition. It also extracts the corresponding locations, font names, font sizes, writing direction (horizontal or vertical) for each text portion [3].
For compatibility purposes of the Python version I have on my machine, I installed Pdfminer3k version, a Python 3 port of PDFMiner. For additional detail on how to install PDFMiner, refer to appendix A.1.

PDF Miner is run from a command line interface to extract content from each document Figure 2 shows how I used it to extract the text of my corpus. Since the process will take a lot of time if executed manually, I built a script to run this command for each document in the corpus. The script I implemented is given in Appendix A.2.

![Figure 2: PDFMiner command to extract content from a PDF](image)

3.2. OCR Technology

The content of some documents could not be extracted since the PDF was locked. A locked PDF acts like a picture from the perspective of text extraction. In such cases, an Optical Character Recognition technology is needed to extract text. I opted for Tesseract Open Source OCR Engine [3] to do the work.

Tesseract was originally developed in C at Hewlett-Packard Laboratories Bristol and at Hewlett-Packard Co., Greeley Colorado between 1985 and 1994, with changes made in 1996 to port to Windows, and to C++ in 1998. In 2005 Tesseract was open sourced by HP. Since 2006 it is developed by Google. While fundamentally built in C and C++, it has wrappers for other languages, so it can be used by developers to implement their own OCR application in other languages as well. The latest stable version is 3.04.01, released in February 2016.
The Tesseract Open Source OCR Engine contains an OCR engine (*libtesseract*) and a command line program—(*tesseract*). Tesseract has unicode (UTF-8) support, and can recognize more than 100 languages "out of the box". It can be trained to recognize other languages. Tesseract supports various output formats: plain-text, hOCR\(^4\), and PDF. The tesseract-OCR project does not include a GUI application. In order to get better OCR results, one needs to improve the quality of the image given to Tesseract.

Figure 3 shows the use of Tesseract-OCR to extract content form pictures. Again, since the process will take a lot of time if executed manually, I built a script to run this command for each document in a picture format. The script I implemented is provided in Appendix A.3.

---

\(^4\) hOCR is an open standard of data representation for formatted text obtained from optical character recognition (OCR). The definition encodes text, style, layout information, recognition confidence metrics and other information using Extensible Markup Language (XML) in form of Hypertext Markup Language (HTML) or XHTML (https://en.wikipedia.org/wiki/HOCR).
3.3. Processing the Extracted Text

Following the extraction of text from PDF files, some NLP technologies were applied to get value from the extracted text. This section explains the processing done on the extracted text in detail.

I began by implementing a program in Python that took as an input the extracted content of a document. The program computed the $k$-skip-$n$grams from the extracted text. The purpose beyond computing the $k$-skip-$n$grams was that the skip-grams might get interesting noun phrases that could be of great interest for indexing. Most probably, such skip-grams may match some other $n$-grams and give them higher indexing.

Additionally, both a discussion with my capstone supervisor and a detailed research on the most queried phrases revealed that the main phrases that are valuable to extract are Noun Phrases.

Figure 4 shows the process that was implemented after of a series of modeling iterations.
The input to the process described in this section is the text extracted from the PDF files, as detailed in the previous section. The individual steps in the processing are described below.

- **Sentence segmentation:**

  Using `nltk.sent_tokenize`, I extracted sentence segments into separate lines. The separators used to create segments were the punctuation marks and the newline character. Figure 5 shows an example of a sentence-segmented paragraph picked from the extracted file of a MBA Master thesis after the sentence segmenter was run on it.
Figure 5: Sample sentence segmentation output

- Sentence tokenization:

  Once the document is split into separate sentences and sentence fragments, I extracted tokens from each sentence using `word_tokenize` from the natural language tool kit in python. The output of tokenizing the sentences in Figure 5 is shown in Figure 6.
Figure 6: Sentence tokenization sample output

- **Part of speech tagging of the list of tokens where each list represents a sentence:**

I then used the Penn Treebank Part-of-Speech tagset, incorporated in the NTLK tool `nltk.pos_tag()`, to tag the tokens in each sentence. A part-of-speech tagger, or POS-tagger, processes a sequence of words, and attaches a part of speech tag to each word. The output of the tagging is stored in pairs consisting of a word and its tag. For example, for the two tokens contained in the noun phrase “the dog”, the output would be:

(“the”, “DT”)  
(“dog”, “NN”)

Figure 7 bellow shows the POS tagging of the tokenized sentences.
Figure 7: List Part of Speech Tagging sample output

- N-grams and skip-ngrams extraction:

Using the n-gram function along with a skip-gram function I implemented, I extracted the grams from the POS-tagged lists. Then, I focused on getting k-skip-ngrams from the tokens of the previous step and stored them in lists. The k-skip-ngrams include also the n-grams among them.

For display purposes, let us apply that concept on only one sentence from the ones resulting from the previous step above (the third one). The result will be a list of pairs where the first element of the pair is the word and the second element is its POS tag using the Penn Tree Bank of NLTK, as follows:
The carrier operates a domestic network in Morocco and it schedules international flights to Africa, Asia, Europe, and North America, and occasionally schedules Hajj services.

Running the `skipgrams()` function on the above list with \( n \) being ‘3’ and \( k \) being ‘3’, we will get the three skip trigrams shown in Figure 8.

![Three skip trigrams of the sentence in figure 11](image)

**Figure 8:** Three skip trigrams of the sentence in figure 11
The contents of Figure 8 are not the complete list of the **three skip trigrams** that were constructed from the above sentence, just those that fit in the screenshot.

> **Grammar definition for the targeted Noun Phrases:**

The next step is to build a list of grammars for the Noun Phrases using rules that cover coherent sentence fragments (chunks).\(^5\)

The rules that make up a chunk grammar use tag patterns to describe sequences of tagged words. A tag pattern is a sequence of part-of-speech tags delimited using angle brackets, e.g. `<DT>?<JJ>*<NN>`. Tag patterns are similar to regular expression patterns. For that purpose and after doing some research, I will specify my grammar following three rules, which use regular expression syntax:

a) \{<DT>?<JJ.*>*<NN.*>+\}

This rule matches any sequence of tokens beginning with an optional determiner, followed by zero or more adjectives of any type, followed by one or more nouns of any type.

b) \{<DT|PP\(\)$>?<JJ>*<NN>\}

This rule matches an optional determiner or possessive pronoun, zero or more adjectives, then a noun.

c) \{<NNP>+\}

This rule matches one or more proper nouns.

These rules cover most of the important NP in the documents.

---

\(^5\) We have seen before part-of-speech tagging which occurs at the word-level tokenization by assigning a tag to a token. A generalization of POS tagging is called **chunking** in which a contiguous sequence of words or tokens is selected and assigned a single tag, example of Noun Phrase (NP).
Elimination of n-grams that do not match the NP Grammar:

Using the above grammar rules on the n-grams we extracted previously, we can chunk the n-grams and transform them into parse trees that will depict all the components of the n-gram and whether there exists a NP matching the specified rule/grammar or not. Figure 9 is a screenshot of the skip-n-grams and their parse trees following the previously defined rules, represented in a Lisp format.

**Code used:**

```python
cp = nltk.RegexpParser(grammar)
result = cp.parse(ng) #ng for n-gram
print(result, "\n")
```

![Lisp parse trees](image)

**Figure 9:** Skip-n-grams parse trees and their matched NPs represented in Lisp format
Figure 10 depicts how the sentence “The little yellow dog barked at the cat” is being tagged using part-of-speech tagging, chunked following the defined grammar and presented in a Lisp format, and then the representation of the Lisp format as a parse tree.

| POS tagging of the sentence | [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
| Chanking | (S

   (NP the/DT little/JJ yellow/JJ dog/NN)

   barked/VBD

   at/IN

   (NP the/DT cat/NN))
| Parse tree |

Figure 10: A sample of a Regular Expression Based NP Chunker

Having used the NP chunker, I pick the noun phrase from each n-gram, display it, store it for later processing, and discard the rest (that is, the text fragments that do not match any NP grammar rules). Figure 11 displays a sample of NPs that figure in each skip-n-grams generated from the previous steps of the sample input.
Figure 11: The NPs that figure in each skip-n-grams generated
Indexation of the results by sorting them by frequency:

The output of the previous step is the input of this one. After the matched n-grams and k-skip-ngrams candidates are selected, the output in Figure 12 will be displayed.

Figure 12: The list of all the Noun Phrases in the sample sentence after applying all the above steps

The list shown in Figure 12 is the set of NP found in the three skip trigrams that we got while applying the skip gram function on the chosen sentence from the file “Aatalla_Sara_2013.pdf”, a SBA MS
Thesis. Figure 13 shows the final sorted list of the most frequent NPs in the three_skip_trigrams we have retrieved from the sentence

The sorting isn’t done lexicographically but by frequency. In otherwords, the program counts the number of occurences of a certain NP and gives it a value/index. The NP with a higher index is ranked in the top of the list then comes the second most frequent and so on. In figure 12, Morocco, Africa, and Asia has three occurences in the list hence they have a higher index and ranked amoung the top frequent NPs with duplicates of those NPs removed and only one representative is kept as it is shown in figure 13.

![Figure 13: Indexed NPs of the sample sentence by frequency](image.png)

3.4. **Applying the Process on the Whole Document**

The steps described in the previous section are applied to each sentence in the document. After results were obtained for an entire document, I chose to display the 100 most frequent NP in the n-grams
as the summary of the original document. This is shown in Figure 14. The idea is that the most frequent words or phrases in the document that can give a person a brief but informative description about the content of the document.

![Image of frequent NPs](image.png)

**Figure 14:** Summary of the document based on the first 100 frequent NPs

After approving each step in the program and applying it to a single document, I applied the process on all the documents of my corpus; a sample processing output is shown in Figure 15
Figure 15: Processing of the whole corpus

Each document processed will result in 100 Noun Phrases, that provide the most frequent information in the document. The result of each document’s processing is stored in a file having the name
summary+theSchool+theDocType+TheActualFileName.txt. For example, the PDF document with the name Aatalla_Sara_2013.pdf was processed by the script and an output indexed file was generated with the name summary+SBA+Master_Thesis+Aatalla_Sara_2013.txt. Those documents generated by the script will then serve as an input for the web application and hence the basis of the AUI Thesis Search Engine.

Appendix A.4 represents the script implemented in Python to perform the automatic indexing of all the extracted text files.

3.5. WordNet Usage

In the Thesis Search Engine, I made use of the WordNet lexical database in order to try to categorize the document using the most frequent Noun Phrases. Figure 16 is a screenshot of the 10 most important synsets of the sample document named Aatalla_Sara_2013.pdf.

![Figure 16: The ten most common synsets found in the sample document based on the summary file](image)

Once all the documents have been processed as described above and WordNet has been used to categorize them into the most relevant topics they address, the list of concepts that the corpus as a whole contains can be computed as shown in Figure 17.
Figure 17: Concepts addressed by the corpus based on WordNet
The Web App always keeps listening for a user to enter a query that will be processed to retrieve the most relevant documents based on WordNet’s synsets. Figure 18 illustrates queries and the most relevant documents retrieved:

![Query and document retrieval example](image)

**Figure 18:** Thesis Search Engine command line interface

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It’s free and open source [4].

Therefore, since it is not user friendly to deploy a program that uses command-line interaction, I thought of building a web app application powered by the same concepts and algorithms in the backend. Having used Python for the text extraction and the pre-processing, I found Django the most appropriate web framework to build the interface of this web application. Figure 19 shows the web interface I have implemented for the AUI Thesis Search Engine using Django.

In case of a student entering a query that retrieves no matches, a list of concepts found in the corpus is displayed lexicographically. This list that, based on the WordNet concepts addressed by the documents, has as purpose to guide the user in formulating queries that are better matched to the document corpus and therefore able to retrieve potentially relevant documents.
Figure 19: Thesis Search Engine user-friendly interface
4. Conclusions

Developing this system was a successful experience for me and a great accomplishment, as it allowed me not only to brush up a great deal of skills I developed during my studies in computer science, but also discover many other exciting technologies. Moreover, by working with a professional supervisor in the field of Natural Language Processing (NLP) who supported me during my capstone project by giving me feedback and tips, I learned how to be responsible of a certain task, although being constrained by a deadline.

Designing and implementing a script that automatically extracts text from readable and locked PDF files using PDFMiner tool, GhostScript, and Tesseract-OCR was the first challenge I took in this project. Indeed, my computer science curiosity to know more about the field of NLP, got the best of me and led me to work on this project.

After successfully addressing my first challenge, I decided to work on the system implementation of the project, which involved developing a web application as well as document-processing software that gets as input the raw extracted text and as an output an indexed file of the most important noun phrases the pdf document talks about. By the end of the development phase, I managed to implement a responsive Thesis Search Engine web app based on NLP technologies, such as k-skip-ngrams, POS tagging, Grammar parse trees, Python NLTK, and WordNet.

Finally, working on this project helped me define what skills and what knowledge I need to improve in the future and inspired me to explore more career opportunities related to the field of Natural Language and machine learning.
References


Appendix A.1: Pdfminer3k installation

How to Install

- Download the source code.
- Unpack it.
- Run setup.py:
  
  $ python setup.py install

- Do the following test:
  
  $ pdf2txt.py samples/simple1.pdf
import os
import subprocess

school = "SSE"
docType = "Master_Thesis"
studentName = ""

def myScript():
    files = os.listdir("")
    for f in files:
        if f.endswith("").pdf"):"
            #set the path name of the converted doc
            studentName = os.path.splitext(f)[0]
            convertedFileName = school +""+docType+""+studentName
            #Run the command to convert the document
            command1 = "pdf2txt.py -o " +"" +convertedFileName + ".txt" + "" + "" + str(f) + ""

            process = subprocess.Popen(command1, shell=True, stdout=subprocess.PIPE)
            process.wait()
            #create a folder that will contain the original document with the required information
            os.mkdir(convertedFileName)
            #Run a command that will move the pdf to the created folder
            command2 = "MOVE " +"" +str(f) + "" +"" +convertedFileName + ""
            process = subprocess.Popen(command2, shell=True, stdout=subprocess.PIPE)
            process.wait()

myScript()
import os
import subprocess

def myScript():
    fileInProcess = ""
    files = os.listdir("")
    for f in files:
        if f.endswith(".txt"):
            statinfo = os.stat(f)
            if statinfo.st_size>=9997:
                fileInProcess = str(os.path.splitext(f)[0])
                #delete the txt file
                command7 = "\del " "\" + str(fileInProcess) + "\" + "\"
                process = subprocess.Popen(command7, shell=True, stdout=subprocess.PIPE)
                process.wait()
                cwd = os.getcwd() + str(os.path.sep) + str(fileInProcess)
                subFolder = os.listdir(cwd)
            for pdf_f in subFolder:
                if pdf_f.endswith(".pdf"): 
                    command1 = "gswin64 -dNOPAUSE -dBATCH -r600 -sDEVICE=tiffg4 -sOutputFile=scan_%d.tif " "\" + "\"
                    process = subprocess.Popen(command1, shell=True, stdout=subprocess.PIPE)
                    process.wait()
                    #process the images
                    subFolder2 = os.listdir("")
                    cnt = 1
                    for tif_f in subFolder2:
                        if tif_f.endswith(".tif"): 
                            command2 = "tesseract scan_ " "\" + str(cnt) + "\" " + "\" + "\\
                            process = subprocess.Popen(command2, shell=True, stdout=subprocess.PIPE)
                            process.wait()
                            #delete the tif file
                            command3 = "\del scan_ " "\" + str(cnt) + "\"
                            process = subprocess.Popen(command3, shell=True, stdout=subprocess.PIPE)
                            process.wait()
                            if cnt == 1:
                                command4 = "\type scan_ " "\" + str(cnt) + "\" " + "\" + str(fileInProcess) + "\" + "\"
                            else:
                                command4 = "\type scan_ " "\" + str(cnt) + "\" " + str(fileInProcess) + "\" + "\"
                            process = subprocess.Popen(command4, shell=True, stdout=subprocess.PIPE)
                            process.wait()
                            command5 = "\echo,> >\" + "\" + str(fileInProcess) + "\" + "\"
                            process = subprocess.Popen(command5, shell=True, stdout=subprocess.PIPE)
                            process.wait()
                            #delete the txt file
                            command6 = "\del scan_ " "\" + str(cnt) + "\"
                            process = subprocess.Popen(command6, shell=True, stdout=subprocess.PIPE)
                            process.wait()
                            cnt += 1

myScript()
Appendix A.4: Script for Automatic Processing of All Extracted Text Files

```python
from itertools import chain, combinations
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import copy
from nltk.util import ngrams
import time

grammar = r'NP: {<DT|PP$>|<JJ>*<NN>} {<NNP>+} {<DT>?<JJ.*>*<NN.*>+}

def pad_sequence(sequence, n, pad_left=False, pad_right=False, pad_symbol=None):
    if pad_left:
        sequence = chain((pad_symbol,) * (n-1), sequence)
    if pad_right:
        sequence = chain(sequence, (pad_symbol,) * (n-1))
    return sequence

def skipgrams(sequence, n, k, pad_left=False, pad_right=False, pad_symbol=None):
    sequence_length = len(sequence)
    sequence = iter(sequence)
    sequence = pad_sequence(sequence, n, pad_left, pad_right, pad_symbol)
    if sequence_length + pad_left + pad_right < k:
        raise Exception("The length of sentence + padding(s) < skip")
    if n < k:
        raise Exception("Degree of Ngrams (n) needs to be bigger than skip (k)")
    history = []
```

nk = n+k

# Return point for recursion.
if nk < 1:
    return

# If nk longer than sequence, reduce k by 1 and recur
elif nk > sequence_length:
    for ng in skipgrams(list(sequence), n, k-1):
        yield ng

while nk > 1:  # Collects the first instance of n+k length history
    history.append(next(sequence))
    nk -= 1

# Iterative drop first item in history and picks up the next
# while yielding skipgrams for each iteration.
for item in sequence:
    history.append(item)
    current_token = history.pop(0)
    # Iterates through the rest of the history and
    # picks out all combinations the n-1grams
    for idx in list(combinations(range(len(history)), n-1)):
        ng = [current_token]
        for _id in idx:
            ng.append(history[_id])
        yield ng

# Recursively yield the skigrams for the rest of sequence where
# len(sequence) < n+k
for ng in list(skipgrams(history, n, k-1)):
    yield ng

def ie_preprocess(document):
    sentences = nltk.sent_tokenize(document)
    sentences = [nltk.word_tokenize(sent) for sent in sentences]

    for i in range(len(sentences)):
        sentence = nltk.pos_tag(sentences[i])

        for skg in skipgrams(sentence, n=3, k=0):
            cp = nltk.RegexpParser(grammar)
            result = cp.parse(skg)
            flag = 0
            for n in result:
                if isinstance(n, nltk.tree.Tree):
                    if n.label() == 'NP':
                        if flag == 0:
                            flag = 1
                            np = ''
                        for w in n:
                            if n[0][0] != w[0]:
                                np += ' ' + w[0]
                                np += w[0]
                    else:
                        np += w[0]
                        yield np

    def rw(f, text):
        with open(f, "a") as myfile:
            myfile.write(text)

    mypath = "./extractedText/
    onlyfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]
    for fl in onlyfiles:
        print("Processing document: " + fl)
        file = "./extractedText/"
file += fl
text_file = open(file, 'r')
#text_file.read()
text = ''
cnt = 0
for line in text_file:
    #print (line)
text += line
    #print (text)
    #time.sleep(1)
cnt += 1
print(str(cnt))
time.sleep(0.1)
#text = text_file.read()
newstr = text
s = ''
count = 0
#print("=>"+str(newstr))
normalChar = [',', '!', '.', ',', ';', ':', '"', '%', '-', '
', 't']
for i in range(len(text)):
    if not text[i].isalnum() and text[i] not in normalChar:
        newstr = newstr[:i-count] + newstr[i-count+1:]
count += 1
#print("=>"+str(newstr))

NPList = []
del NPList[:]
for el in ie_preprocess(str(newstr)):
    if len(el) > 2:
        NPList.append(el)

file2 = file = './extractedText/summaries/summary+'
file2 += fl
print("the list contains: " + str(len(NPList)) + ":NP")
c = Counter(NPList)
indexedNPList = []
del indexedNPList[:]
j = 0
for word in c.most_common(100):
    indexedNPList.append(word[0])
rw(file2,word[0])
if indexedNPList[j][len(word[0])-1] != '"' and j!=99:
rw(file2,"
")
j += 1