AR EXTRACTOR : AUTOMATICALLY EXTRACTING CONSTRAINTS FROM ANDROID DOCUMENTATION USING NLP TECHNIQUES

A Thesis by

Padma Priya Sakthivel

Bachelor of Engineering, Anna University, 2013

Submitted to the School of Computing
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

December 2022

AR EXTRACTOR : AUTOMATICALLY EXTRACTING CONSTRAINTS FROM ANDROID DOCUMENTATION USING NLP TECHNIQUES

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Science.

_____
Zhiyong Shan, Committee Chair

_____
Hongsheng He, Committee Member

_____
Abu Asaduzzaman , Committee Member

DEDICATION

To my mother Vasanthi and my father Sakthivel for their unconditional love

To my husband John Lupus for his continuous support and encouragement

Last but not the least, to my son Joshua Vihaan for being the light of my life and making me stronger.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and heartfelt appreciation for my advisor, Dr. Zhiyong Shan for all the guidance, support and encouragement throughout my studies. My gratitude extends to my advisor for understanding my passion and interest and providing me a chance to work on this thesis research . A Special thanks for helping me sail through the uncertain visa situations with all kindness and continue my studies and also for providing the opportunity to become Graduate Teaching Assistant at the EECS department that helped me to support my education at Wichita State University.

I am deeply grateful to my dissertation committee members Dr.Zhiyong Shan, Dr. Hongsheng He, Dr. Abu Asaduzzaman. I would like to thank them for their valuable time, suggestions, and insights to shape up the scope of this dissertation.

Next, I would like to thank the Universe and almighty God for giving me strength and patience to accomplish this theses. I would like to express my gratitude to my parents, husband and son. Without their tremendous understanding and encouragement in the past few years, it would have been impossible to finish my Masters.

Finally, I would like to thank my friends for their continous moral support and guidance.

ABSTRACT

When developing android apps, it is difficult for the programmers to follow all programming constrains in Android documents. Programming rules are difficult to be manually identified and documented by programmers. As those rules are not so simple, extraction of these kind of rules will be helpful for programmers to avoid new bugs and violations. Sentences with keywords like should, must, important, note, caution, etc., are possible constraints and can carry significant conditional information, that can be helpful for aspiring android developers. In case of developing complex or large android apps, most of the time it is complicated for the programmers to follow all rules since they do not have these rules documented. This paper proposes a novel method called AR-Extractor (Android Rules Extractor) to automatically extract programming constraints using natural language processing techniques(NLP). NLP is a branch of Artificial Intelligence and it involves a set of statistical techniques for identifying parts of speech, entities, sentiment, text classification and other aspects of text. In this project we worked on extracting important text information from Android Developers website by employing the Parts of Speech tagging to understand the relation between each words in a sentence and identify its structure to refine the unstructured text data into structured consolidated format. It can help programmers to reduce bugs, understand programming parameters, improve software maintainability and reliability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HTML – Hypertext Markup Language

IDE – Integrated Development Environment

IDF – Inverse Document Frequency

NLP – Natural Language Processing

NLTK – Natual Language Tool Kit

PoS – Parts of Speech

TF – Term Frequency

CHAPTER 1

INTRODUCTION

Textual data is unstructured, difficult to manipulate, and ambiguous data. Imagine having to manually go through all of the textual data and extract the most relevant information. The difficult task is not in finding the right documents, but in finding the right information from these documents. Understanding the relationship between entities, understanding how the events have unfolded, or just simply finding hidden gems of information, is clearly what anyone is looking for when they go through a piece of text. We might be looking for names of entities, certain keywords, others would want to extract important highlighted information. Our intentions differ according to our requirements. Overall we want to get more accurate insights across a broader range of documents and sources. Therefore, coming up with an automated way of extracting the information from textual data and presenting it in a structured manner will help us reap a lot of benefits and tremendously reduce the amount of time we have to spend time skimming through text documents.

Data extraction technique refers to the extraction of insightful information from enormous trove of textual data. It involves the retrieval of data from diverse sources. The information extracted is then stored for further analysis or use in a separate database. Text mining (also referred to as *text analytics*) is an artificial intelligence (AI) technology that uses natural language processing (NLP) to transform the free (unstructured) text in documents and databases into normalized, structured data suitable for analysis or to drive machine learning (ML) algorithms.

Without automated data extraction and text mining, text analysis would be burdensome and prone to more human errors. Especially when you have to learn from a tutorial website like Official Android Development Tutorial which consists of hundreds and thousands of webpages, having to go through all of it and learn important dos and don’ts and other important considerations to make

while developing an application. An automated way of extracting constraints from official android developer guide and presenting it in a structured manner will help Android developers when creating android applications. Android developers and programmers will benefit if they have a handy document for important programming constraints that serve as a guideline to help in setting app attributes, parameters and coding.

This is precisely what AR Extractor strives to achieve. In this project, we work on the official Android Developers guide website to extract sentences that provide important information that can be documented for future reference for Android developers. AR Extractor focuses on extracting sentences that contains constraints like should, must, etc., and check if the extracted sentences provide meaningful information and extract previous and next sentences if required for information completeness and sentence coherency. We use Natural Language Processing tools such as PoS tagging and Dependency Parsing to work on the unstructured text from Android Webpages and transform it into a structured one for future reference. This can greatly help the Android programmers to reduce software defects that occur due to rule violation. The extracted information can be used in multiple ways, like it can be categorized and used as a workbook or we can store in a database that can be implemented as a tool in an IDE.

CHAPTER 2

BACKGROUND

Before getting into how we extract the text, we have to understand what by definition is a constraint and learn about NLP and few sub concepts like POS Tagging and how dependencies are interpreted with examples**.**

2.1 Natural Language Processing (NLP)

Natural Language Processing(NLP) is a part of computer science and artificial intelligence which deals with human languages. Natural language processing (NLP) can be thought of as an intersection of Linguistics, Computer Science and Artificial Intelligence that helps computers understand, interpret and manipulate human language. NLP is a component of text mining that performs a special kind of linguistic analysis that essentially helps a machine “read” text. NLP handles written text at level of documents, words, grammar, meaning, and context. NLP considers the hierarchical structure of language: several words make a phrase, several phrases make a sentence and, ultimately, sentences convey ideas. Using text vectorization, NLP tools transform text into something a machine can understand, then machine learning algorithms are fed training data and expected outputs (tags) to train machines to make associations between a particular input and its corresponding output. Machines then use statistical analysis methods to build their own “knowledge bank” and discern which features best represent the texts, before making predictions for unseen data (new texts). Human language is filled with ambiguities that make it incredibly difficult to write software that accurately determines the intended meaning of text or voice data. Homonyms, homophones, sarcasm, idioms, metaphors, grammar and usage exceptions, variations in sentence structure—these just a few of the irregularities of human language that take humans years to learn, but that programmers must teach natural language-driven applications to recognize

and understand accurately from the start, if those applications are going to be useful. NLP uses a different methodology to decipher the ambiguities in human language, including the following: automatic summarization, part-of-speech tagging, disambiguation, chunking, as well as disambiguation, and natural language understanding and recognition. Today NLP is used to create systems like speech recognition, document summarization, machine translation, spam detection, question answering, autocomplete, predictive typing and so on.

2.2 Android Programming Constraints

Constraints are the words that can provide conditional sense, obligation or advice. Few examples are '*should', 'note', 'warning', 'noted', 'recommended', 'need', 'tip', 'remember', 'caution', 'warning', 'hint',* '*make sure', 'cautious*'. Sentences containing these constraints words convey crucial information, obtaining and documenting them can save a lot of time instead of having to look through the whole text.

Many programming constraints are much more complicated. Some sentences with constraints may be correlated or dependent on each other i.e., those sentences will refer to the subject of previous or next sentence. In this case, determiners, or pronouns, e.g., *they, this* are used. It is important to understand the programming rules as well as dependencies between constraints to be able to implement them correctly.

Example : "if the activity returns to the Resumed state from the Paused state, the system once again calls onResume() method. For this reason, you should implement onResume() to initialize components that you release during onPause() and perform any other initializations that must occur each time the activity enters the Resumed state".

For the above example, AR-Extractor will extract the second sentence since it contains the constraint *should.* But the second sentence depends on the previous sentence that is indicated by

the determiner *this (for this reason).* Therefore, both sentences need to be extracted to provide meaningful information. For such cases, sentence structure needs to be analyzed using natural language processing.

Some complex rules may indicate sentence dependency or correlations, i.e., those sentences will refer to the subject of previous or next sentence, in those cases it should be extracted together. Firstly, the tool scrapes HTML files are from android documents and then the annotated sentences are parsed accordingly, checked for constraints, and explored for further dependencies to previous or next sentences using natural language processing techniques and machine learning algorithms. Sentences are scraped by exploiting the part-of-speech (POS) and dependency tags help in leading to a more purposeful extraction containing constraints. Machine learning is used for visualization of constraints and its associated relation.

Our results indicate that the performance of our tool generates more precise android programming rules or constraints automatically.

TABLE 1

EXAMPLE OF CONSTRAINTS

Constraints	Example sentences
Must	You must publish your JSON verification file at the following location: https://domain.name/.well-known/assetlinks.json
Note	In Android 7.0 (API level 24) and higher, the system throws a TransactionTooLargeException as a runtime exception. In lower versions of Android, the system only shows a warning in logcat.
Warning	Warning: If an activity, service, or broadcast receiver uses intent filters your app can't be installed on a device that runs Android 12 or higher.

TABLE 1 (continued)

make sure	Please make sure to set the attributes of receiver method.
recommended	It is recommended to restart phone after the app has crashed
important	The most important step is to bring the image size down mainly for the space reasons
Remember	Remember to display the radio button
need	You'll have a list of behavior changes that you'll need to test and debug your app for.
cautious	Be cautious about sharing passwords and other sensitive information.

2.3 POS Tagging

Part of Speech (POS) tagging is a natural language processing technique for categorizing words in a text corpus corresponding to a specific part of speech, depending on the word context and definition. The words in a sentence belong to different Parts of Speech (POS). There are nine different parts of speech in English: noun, pronoun, verb, article, adjective, adverb, preposition, conjunction, and interjection. The POS determines the characteristic structure of words within a sentence or how a specific word is defined in each sentence. It is possible for a word to have distinct parts of speech in different sentences depending on different contexts.

Thus, the POS tag of a word carries a lot of significance in understanding the meaning of a sentence. We can leverage POS tagging to extract meaningful information from our text using the attribute .pos_ in Spacy (NLP library) and can extract sentences based on nouns, verbs and whatsoever our requirements. But at times extracting information solely based on the POS tags is not suffice. If we want to extract the subject and object from a sentence, we cannot just do that based on their POS tags. For that, it’s important to look at the relationship between words which are called dependencies. Knowing the dependencies will help in analyzing the completeness and

meaningfulness of the textual data. While Part-of-Speech (POS) Tagging assigns different labels known as POS tags to the words in a sentence that tells us about the part-of-speech of the word, Dependency parsing is the process of analyzing the grammatical structure of a sentence based on the dependencies between the words in a sentence. In Dependency parsing, various tags represent the relationship between two words in a sentence. These tags are the dependency tags.

Example : The boy gave the right answer

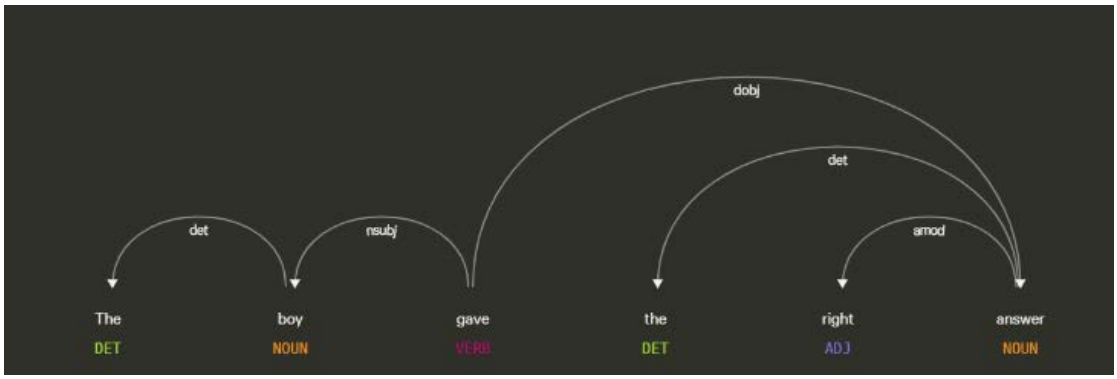

Figure 1 : Example 1 Pos and Dependency tagging

Whereas, in the sentence, you have the right to ask your wants, here the word right is treated as noun.

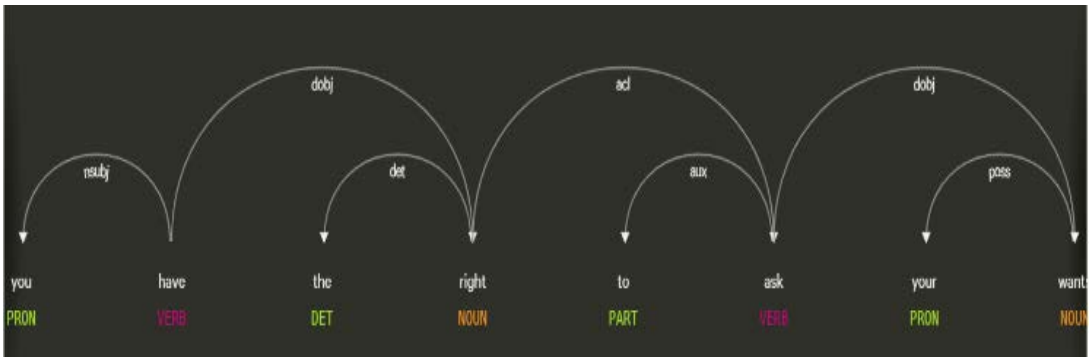

Figure 2 : Example 2 Pos and Dependency tagging

In the above image, the labels below the words are Parts of Speech tags (Noun,Verb,etc.,) and the arrows represent the dependency between two words in which the word at the arrowhead is the child, and the word at the end of the arrow is head. The root word can act as the head of multiple words in a sentence but is not a child of any other word. You can see above that the word ‘took’ has multiple outgoing arrows but none incoming. Therefore, it is the root word. One interesting thing about the root word is that if you start tracing the dependencies in a sentence you can reach the root word, no matter from which word you start. Using POS tags and dependency tags, it’s helpful to analyze relationships between different entities in a sentence and extract dependent sentences i.e., the previous or the next sentences as needed.

2.4 Keywords and Dependencies

We consider words like ‘This’, ‘That’, ‘Them’, ‘Therefore’, ‘Although’, ’however’ in the extracted sentences that contain the constraints. These words imply that the extracted sentence may not give complete information and may depend on the previous or the next sentence for its wholeness and relevance. Words like ‘this’, ‘that’, ‘it’ carry different POS tags and based on the context of the text and their POS tags and dependencies tags will help in determining if previous or next sentence is really needed for the meaningfulness of the text information. We also have few words in the extracted sentences that will help in determining if the extracted sentence is really needed at all. We can skip those type of sentences and refine the final data.

CHAPTER 3

METHODOLOGY

AR-Extractor consists of four steps as shown in Figure 3. The first step is to download HTML documents using a website crawler or spider called Screaming Frog to automatically crawl and download web pages.

In the *data extraction* step, we performed web scraping on android documents to get HTML data. Web scraping method is employed and automated using a python library called beautiful soup that pulls data out of HTML and XML files. It works with a parser of user's choice to provide idiomatic ways of searching, navigating, and reformatting or modifying the parse tree [1]. Our code is designed to pass data scraping based on the HTML tags (e.g., *id, class, a, p*) and using pattern based regular expression. The HTML data is then processed to remove unnecessary information like header, footer, XML, JAVA, JSON, KOTLIN code etc,so that only annotated document is extracted. This process is carried out to filter constraints from the annotated document and to prepare the text for further grouping and analysis.

In the *data processing* step, the sentences with constraints are identified and explored for further dependencies where sentences are usually associated to each other where one sentence might refer to the subject of previous or next sentence. *Spacy*, a natural language processing library is used for analyzing the sentences and their parts of speech (POS) tags to determine if the sentence gives complete information or previous or the next sentences' information needs to be extracted to give a meaningful information.

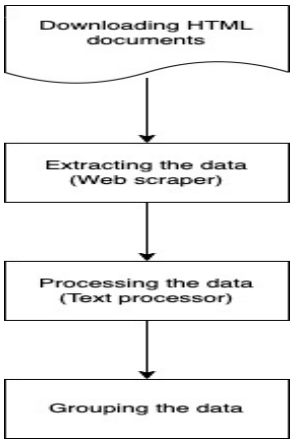

Figure 3 Overall Method

In the *data grouping* step, these set of sentences are pre-processed to obtain their TF-IDF scores. These scores are helpful in vectorizing the sentences. The vectors of these sentences are used as an input to the K-means machine learning algorithm to determine the centroids and thereby datapoints are assigned to the nearest centroids. These are formed as clusters; thus, relevant sentences are categorized for easy use and readability.

3.1 Datasets

To develop an efficient web scraper, we need a relevant data set. The Android Developer guide web pages has hundreds of links and each of these links hold a handful of information. Few non-essential information is filtered, and the remaining text data is scraped. There are four main steps involved in the process of scraping data and the following steps are undertaken:

- We use a web crawler called screaming frog to automatically download 500 android developer guide web pages.
- *Extracting data* using beautiful soup library with lxml parser and regular expression that can scrape data from various tags and store them in a .txt files, and xlsx files.

- Stored files are used for further *processing* to explore sentences with constraints and its dependencies

- The sentences with constraints are used for data grouping or visualization using machine learning techniques.

3.2 Challenges

To our knowledge, there has been no effort so far to automate the extraction of android programming rules. A fundamental challenge is that not all data from android documentation forms a programming rule. Firstly, it requires a lot of time and effort to manually go through the data for extraction from android documentation and then automating the approach using steps as described in Figure 3. Every webpage is designed in a distinct way, and it requires exploring the android documentation for the existence of programming rules i.e., constraints. Using *inspect element* feature from google chrome, valuable text enclosed within tags are identified and then these tags are used for extraction in the automatic approach.

A lot of challenges arise when it comes to extracting the relevant and meaningful text data. It required testing around 100 web pages for element inspection to identify necessary tags using *inspect element* feature and only then the commonly identified tags for the existence of constraints were used in the automatic approach. To find the tags and its attributes surrounded by specific text, there is a powerful developer tool in *google chrome*, *inspect element* feature is used to inspect structure of the HTML pages.

It needs a lot of time perusing the text to identify the constraints words in the first place and when it comes to dependencies i.e., one sentence might refer to the subject of next or previous sentence, it needs much more time to analyze the sentence structure, their part of speech (POS) tags and dependencies. At times even when the sentence contains keywords such as *this*, *that*, the

sentence may be complete as shown in Table 1. Separate rules for these keywords are written based on their part of speech tags, dependency tags and punctuation.

3.3 Extracting the data

It is always a concern for the programmers to identify valuable rules from the official documentations. To address this problem, a methodology called web scraping is employed to pull out the content of interest from the required web pages. In this paper, it is carefully analyzed, how the programming rules or constraints is pulled-out from the android documentation web pages. The following three aspects of content extraction are taken into account: Beautiful Soup [2] along with lxml parser [3], and regular expression. Figure 5 shows the structure of the web scraper.

***Step 1*:** The HTML documents, i.e., android technical document web pages are given as an input to the *web scraper* as shown in Figure 5. The input is processed in batches with each batch containing 50 HTML files. The downloaded android documentation HTML pages needs to be cleaned before being used for further data extraction.

***Step 2*:** For cleaning and extracting the necessary content from android HTML documents, we use a python library called *Beautiful Soup* that is used for parsing structured data and helps to pull data out of HTML files.

To parse HTML files with beautiful soup, it is required to use parser and pass them in the Beautiful Soup constructor.

For example,

soup = BeautifulSoup (fp, 'lxml')

where *fp->HTML file*

There are a lot of HTML elements or tags such as <*p*>, <*span*>, <*a*> etc. in a web page. *Figure* shows only a crucial part of an android documentation web page that is inspected using *inspect element* feature (explained in section II, B) containing the constraint *note.* Additionally, these elements also have attributes such as *id, title, href, class, etc.* whereas *id* and *class* are the commonly used attributes.

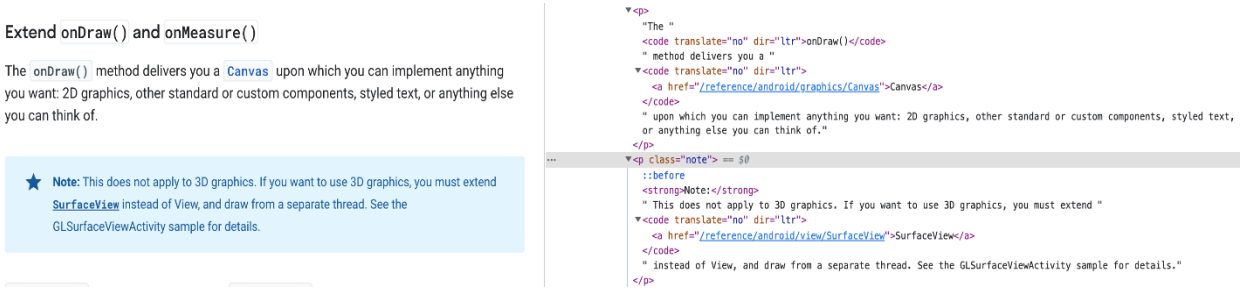

Figure 4 : Inspect Element Feature

In Figure 4, the constraint *note* is enclosed within <*strong*> tag indicating that *note* is of strong importance, but its entire contents are enclosed within the <*p*> tag.

The *web scraper* script extracts three types of contents separately from the android HTML files and are processed individually in the *text processing* step. Following are the contents extracted from three different tags,

- Text encompassed by <*strong*> and its associated tags
- Text encompassed by <*main*> tags
- Tables enclosed within <*table*> *tags*

The *web scraper* is trained in such a way that it extracts only the necessary text from the android HTML documents. Since only the annotated document is considered for extraction, other less important information like *header, footer, script, style, code blocks* enclosed within tags such as *lang-kotlin, lang-json, lang-java and lang-xml, figures, side menus* can be ignored. These kind of

invaluable information can be ignored using *Beautifulsoup.Tag.decompose()*[4] that removes a tag and destroys its contents.

Step 3: In Figure 3, after data cleaning, following three distinct steps are followed for content extraction,

3.3.1 To extract text from and its associated tags:

The programming constraints considered for extraction in this paper are *should, must, important, recommend, make sure, need, note warning, caution.* The constraints i.e., the keywords *note, warning, caution* is commonly enclosed within <*strong*> tag and its contents are within tags such as <*p*>, <*li*>, <*aside*>, <*ul*>, <*div*> .

The constraint *note* is enclosed within <*strong*> tag but its entire associated text may be surrounded by <*p*> tag with attribute *class=" note"*. This complete text related to tag is called as <*strong*> *tag* text extraction. Therefore, the constraints considered for tag text extraction is *note, warning, caution, tip, hint, important.*

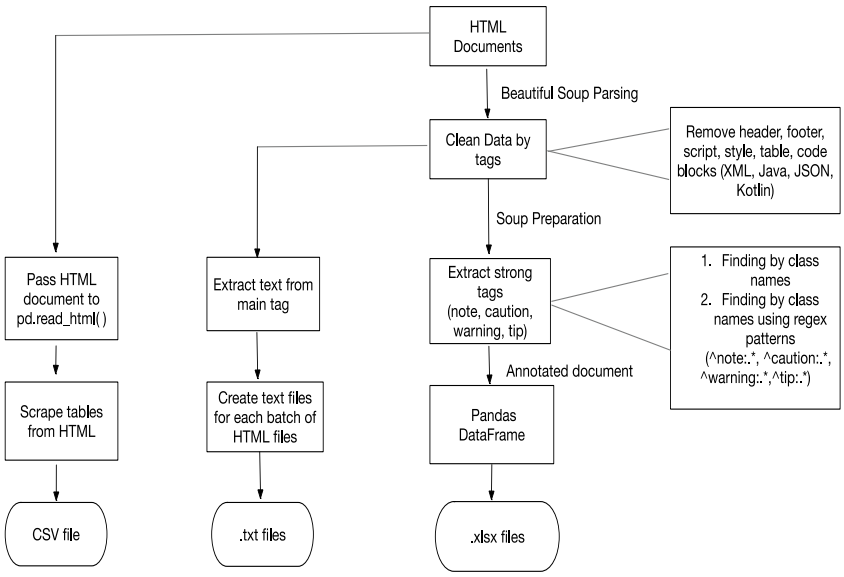

Figure 5 Structure of Web scraper

For example in Figure 8, the constraint *note* has two sentences. Though the second sentence is not related or dependent on the first sentence, both these sentences are important because the *attribute class* contains the constraint *note* surrounded by <*strong*> tag and its contents within <*p*>, so entire text within and its associated tag <p> needs to be extracted. This is the reason for extracting tags text separately because if multiple sentences enclosed by and its related tags are not dependent, the *text processor* script will split the sentences in the tokenization process.

The possible tags considered for existence of <*strong*> tag contents are <*p*>, <*li*>, <*aside*>, <*ul*>, <*div*> and these tags are further explored for the presence of attribute *class* that contain constraints such as *note, warning, caution, tip, hint* which are the common constraints enclosed within <*strong*> tags.

The constraints surrounded by tag are identified using regular expression ^*note:.**, ^*caution:.**, ^*tip:.**, ^*warning.**. For example, suppose we use the regular expression ^*caution*:.* for extracting text within <*strong*> tag, and we analyze the sentence,

Caution: Don't confuse client and server permissions; ensure that the called app has appropriate permissions and verify that you grant the same permissions to the calling app.

The above sentence will match the regex and will be extracted. In this way, entire text contents within <*strong*> and its associated tags are extracted from android developer guide HTML files and the annotated document is appended to the Pandas DataFrame and then written to excel files (.xlsx files)

3.3.2 To extract text from <main> tags:

The text surrounded by <*main*> tag is scraped and converted into a stream of text files for every batch of android documentation HTML files. The converted text files are passed as an input

to the *text processor* script for exploring dependencies, explained in section IV. Since <table> tags are processed individually, tables are not considered in <main> tag extraction. Text extracted from <main> tags are converted into a stream of *.txt files.*

3.3.3 To extract tables from <table> tags:

In addition, it should be checked whether information from tables is needed and was parsed correctly. For this process, all tables from the given android documentation pages are scraped using pandas read_html function and written to a CSV (Comma Separated Values) file. In Figure 5, acquired android documentation web pages are passed as an input to the pandas read_html function [pandas reference] for table extraction. This function searches for <table> elements and only for table row <tr> and table header <th> rows and table data <td> elements within each <tr> or <th> element in the table. This function attempts to properly handle colspan and rowspan attributes. If the function has a <thead> argument, it is used to construct the header, otherwise the function attempts to find the header within the body (by putting rows with only <th> elements into the header). [pandas dataframe reference]. Since table is a visual grouping of brief information about complicated data, every cell contains important information similar to sentences with constraints. Therefore, extracted tables can be presented in an output in a excel, the extracted tables represented in a excel file format are passed as an input to the text processor script to be analyzed for the presence of constraints such as should, must, note, warning, tip, hint, etc. .txt files are then further chunked into sentences and checked for the occurrence of keywords like previous or next sentences such as this, that, they, however, therefore, instead, etc

3.4 Processing the data

Web scraper script generates three sets of output,

- *.xlsx* files containing *tables,*

- *.txt files* extracted from *HTML files <main> tags, and*
- *.xlsx files* containing <*strong*>and its associated tags text.

In this step, we process each of these files separately and extract only the text containing constraints along with its dependencies. The output of the *web scraper* script is the input to *text processor*. *Text processor* script processes each of the above output separately.

3.4.1 Processing HTML Tables

Tables contain various information like parameters/expressions/attributes/methods and their description, purpose. Web scraper script outputs a list of .csv and .xlsx files containing table information extracted from <table> tags. These .xlsx files are passed to a pandas DataFrame for easy processing. Each excel file with the table information contains a table index, table header and table data.

To extract constraints from these .xlsx files, table rows are iterated to look for the presence of constraints such as *should, must, caution, warning, important* etc., and if these constraints are present, that specific row containing constraints is extracted, and the next step is to look for the table it belongs, to fetch its table header. This is done by checking the previous rows and look for the row that contains naN value in the first column.

If multiple rows containing constraints belong to the same table, it is checked whether its table header is already extracted and if it is extracted, the row is just appended to the previous rows of the same table in the excel sheet. The table rows containing constraints along with the table header output is exported to a different sheet in the same excel workbook. The purpose of exporting the extracted table rows and its headers to the same workbook is to give users, the complete table information and also table rows that contain constraints.

	A	B	C	D
1		0	1	2
2	0		onPlay()	onPause()
3	1	Audio Focus	requestFocus() passing in your OnAudioFocusChangeListener.Always call requestFocus() first, proceed only if focus	
4	2	Service	startService()	
5	3	Media Session	setActive(true) - Update metadata and state	- Update metadata and state
6	4	Player Implementation	Start the player	Pause the player
7	5	Becoming Noisy	Register your BroadcastReceiver	Unregister your BroadcastReceiver
8	6	Notifications	startForeground(notification)	stopForeground(false)
9		Action	Callback	
10	0	ACTION_PLAY	onPlay()	
11	1	ACTION_PLAY_FROM_SE	onPlayFromSearch()	
12	2	ACTION_PLAY_FROM_UI	onPlayFromUri()	
13		Action	Callback	
14	0	ACTION_PREPARE	onPrepare()	
15	1	ACTION_PREPARE_FROM	onPrepareFromSearch()	
16	2	ACTION_PREPARE_FROM	onPrepareFromUri()	
17		Action	Callback	Description
18	0	ACTION_SKIP_TO_NEXT	onSkipToNext()	Next video

Figure 6 : Tables separated by empty header (Nan Value)

3.4.2 Processing the tag text:

HTML tag is a phrase tag which is used to represent the important text of a document on the browser. The element is for content that is of "strong importance," including things of great seriousness or urgency (such as warnings).

This could be a sentence that is of great importance to the whole page, or you could merely try to point out that some words are of greater importance compared to nearby content. As mentioned in the section III, the and its related tags are denoted in the HTML files with the attribute *class = note | caution | warning | special*.

At times, sentences encompassed by and its related tags may be correlated or dependent on other sentences i.e., sentences will refer to the subject of previous or next sentence indicated by keywords such as *this, they, those, however etc*.

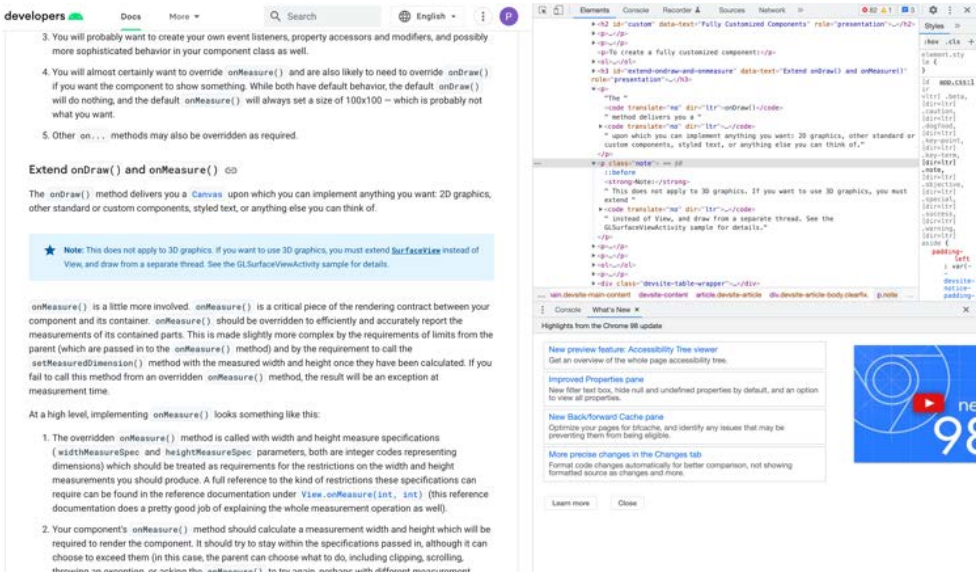

Figure 7 : tags text with dependency

The above keywords may appear anywhere in the sentence, and it is necessary to analyze if the previous sentence is needed for its completeness and right context. We use natural language processing to analyze the sentence structure and categorize words in the sentence using parts of speech (POS) tagging and based on few conditions, we can either extract the previous sentence with tag text or just extract the current tag sentence.

For example, in Figure 8, below sentence with constraint *note* is encompassed by tag and its contents are surrounded by <p> tag with the attribute *class= note*. But the below first

sentence surrounded by <p> tag, refers to the previous sentence that is indicated by the keyword *this.*

Example :"Note: This does not apply to 3D graphics. If you want to use 3D graphics, you must extend SurfaceView instead of View, and draw from a separate thread. See the GLSurfaceViewActivity sample for details."

In this case, it is necessary to extract the below previous sentence,

Previous sentence : "The onDraw() method delivers you a Canvas upon which you can implement anything you want: 2D graphics, other standard or custom components, styled text, or anything else you can think of".

The above sentence needs to be extracted along with the tag text. Since .xlsx files only contain the tag text, extraction of previous sentences requires text files extracted from <main> tags by *web scraper script.*

Thus, processing of tag text requires two inputs.

- The first input is the tag excel sheet and
- The second input contains the list of text files that is extracted from the <main> tags.

The rows in tags excel file are passed to a pandas DataFrame for easy processing. The sentences in the text files are iterated and passed to another DataFrame.

The DataFrame containing the tags are iterated, and each row containing the sentences is analyzed using NLP techniques for identifying keywords like *this, these, however, therefore, hence* and if encountered, the same sentence is searched in the text file DataFrame by looping through the rows. The text files are also iterated and checked for the presence of the tag sentence. If the similar sentence from tag row is found in the text file DataFrame row, text file DataFrame's index will be decreased by 1 to fetch its previous sentence.

Now, extraction of previous sentence gives a more relevant and meaningful information, since the keyword *this* refers to the previous sentence. The processed sentences are finally exported to an excel sheet.

3.4.3 Processing the text from <main> tags:

The extracted text from <main> tags are converted to a stream of .txt files by *web scraper* script.

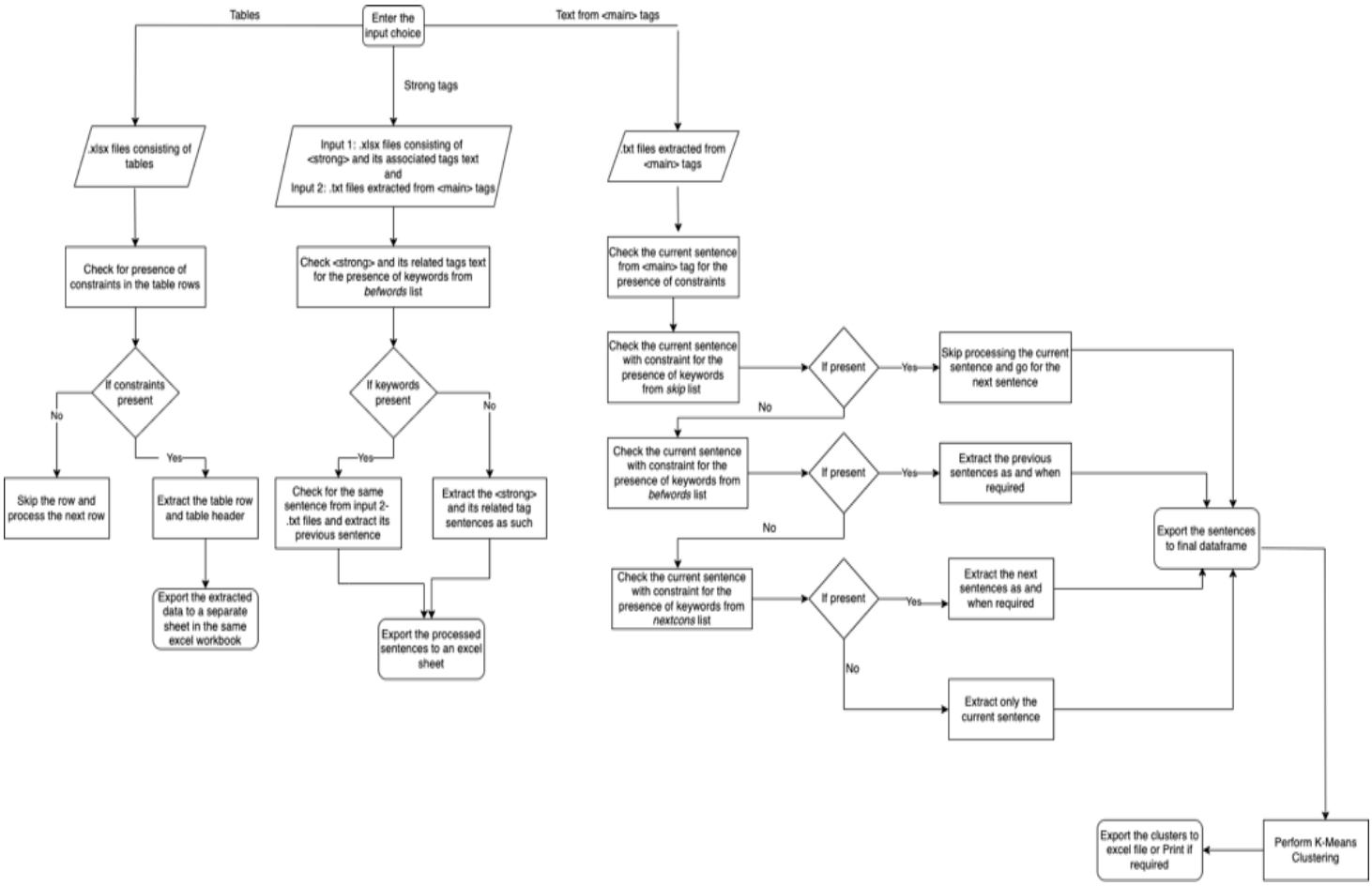

Figure 8 : Structure of Text Processor

We will now look in detail about how the sentences are processed. We created the following three lists,

- A ***befwords*** list that contains a set of keywords that refer to previous sentences i.e., *this, that, these, furthermore, however, hence, otherwise etc.*
- A ***nextcons*** list with a set of keywords that refer to its next or following sentences i.e., *next, after, following, consider an example etc.*
- A ***skip*** list that contains a set of keywords for sentences to be skipped i.e., *following diagram, following snippet, following code, this section, this lesson*. Since we only focus on extracting annotated document for the process.

The processing steps are as follows:

1. The input text files are iterated one by one.
2. Each text file contains a set of paragraphs and tokenized into a list of sentences.
3. The tokenized sentence list is then passed to a DataFrame for easier and faster processing.
4. Each row in the DataFrame with a sentence is checked for the presence of constraints such as *should, must, need, important, remember, noted, recommend, make sure, cautious.*
5. If any constraints are present in the sentence, the current sentence is first analyzed to check for keywords in the *skip* list and if found, the current sentence is ignored or skipped for further processing.

Keywords to look for:
skip= [following diagram, following code, following snippet, this lesson, this guide, this page, figure, this document]

Since this paper is about processing only the annotated text documents of android developer webpages, it is important to look for sentences that refer to code blocks,

diagrams, other unnecessary details, and discard or not extract that particular sentence. For this process, we studied few android webpages and identified some of the common above-mentioned keywords that refer to the code blocks, figures, or diagrams. Hence, after extracting a sentence with constraint, every sentence structure is analyzed and checked for the presence of keywords from the skip list as mentioned.

Example:

"If you want a widget with a button that launches an activity when clicked, you should use the following implementation of AppWidgetProvider".

The above example has a sentence with the constraint should, the text processor script will extract the above sentence due to the presence of constraint. However, this sentence by itself does not provide complete information. It is referring to a code snippet that explains the implementation of AppWidgetProvider and the extracted sentence is not useful without the implementation part. Since we are not concentrating on extracting code snippets, these kinds of sentences are not useful and discarded. Sentences starting with words like this guide, this lesson, this page refers to the whole web page or document and hence will not look relevant to read. Similarly sentences referring to figures or diagrams are also skipped.

6. If the current sentence with constraint doesn't contain keywords from *skip* list, then the next step is to first check the *current sentence* for its dependency to its previous sentence. The current sentence is checked for the presence of keywords from *befwords* list. This is checked by analyzing the sentence structure using natural language processing technique where part of speech (POS) tags and dependency tags are used for checking dependency.

If current sentence contains the keyword from *befwords* list, its POS and dependency tags are checked accordingly as shown in table 1. Then, previous sentence is extracted and again the extracted previous sentence is analyzed for keywords from *befwords*.

Keywords to look for:
befwords=
[*this,that,these,then,however,therefore,then,instead,them,after,next,hence,otherwise,additionally, for example]*

If any dependency is identified, the sentence before the previous sentence is also extracted to get a clear context. The text processor script is developed in such a way that it traverses and extracts up to two sentences before the current sentence, if needed. If the current sentence require or depend on its previous sentence, then only the current sentence is extracted.

<u>Example:</u> *"You can also constrain views that are inside the barrier to the barrier. This way, you can ensure that all views in the barrier always align to each other, even if you don't know which view will be the longest or tallest."*

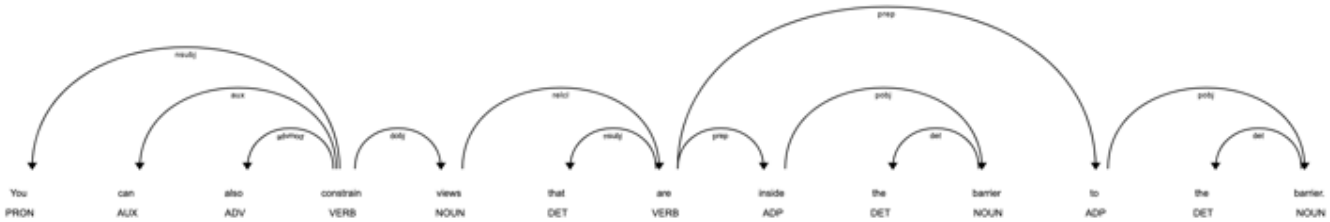

Figure 9 : Sentence tree structure

In the above example, the second sentence contains the constraint ensure and the sentence will be extracted, but the sentence by itself does not provide any complete information. The above sentence structure will then be analyzed where part of speech (POS) tags and dependency tags are used for checking dependency. The second sentence contains the keyword this indicating dependency to the previous sentence about constraining views. Therefore, when we look at its previous sentence (i.e., the first sentence),

You can also constrain views that are inside the barrier to the barrier.

The first sentence has the information about constrain views that are inside the barrier to the barrier. Thus, we are extracting the previous sentence along with the current sentence. In few scenarios, we may have to extract up to two previous sentences to get a clear understanding of sentences.

Example: To play music based on a search query, use the INTENT_ACTION_MEDIA_PLAY_FROM_SEARCH intent.

An app may fire this intent in response to the user's voice command to play music. The receiving app for this intent should perform a search within its inventory to match existing content to the given query and starts playing that content.

In the above example, there are three sentences. The text processor script will extract the last sentence since it contains the keyword should.

The above sentence structure is analyzed, and its POS and dependency tags are used for checking relevance to its previous or next sentences where the sentence also contains the keyword from *befwords* list, this. So, we need to refer its previous sentence,

In this case, the previous sentence also contains with the keyword this, and we must again refer to its previous sentence to get the clear context. Therefore, current sentence is extracted along with the above two previous sentences.

At times, we do have the keyword 'this' in between in the sentence and in this case, we need to analyze if this keyword requires its previous sentence or not.

Example 1:

"Expensive tasks should be safely performed synchronously with this method"

The current sentence requires its previous sentence by analyzing the sentence structure using POS and dependency tags. It is found that sentence contains the keyword from befwords list because it's unclear that what the current sentence method refers to and thus requires its previous sentence for its completeness.

Example 2:

"Because the receiver consumes device resources, use this with care – Here, previous sentence is not needed, the sentence by itself is complete in context as the keyword this in the sentence refers to resources which is present in the current sentence itself"

The difference between both the sentences is the presence of comma (,) in the sentence, if there is a comma, the information is broken down in parts; otherwise, a verbal pause needed to prevent misreading the intended meaning. It sometimes comes after an introductory phrase or clause, and the main part of the sentence is about to begin after the comma. So,

in these cases if ‘this’ is present in a sentence, what it is referring to is mostly in the current sentence itself.

Similarly, another keyword that may have different context and depends on its previous sentence in few scenarios and not need them in other scenarios

Example 3:

“In that case, please make sure the attribute values are written as True”

This sentence requires its previous sentence, we need its previous sentence to understand what that case was.

Example 4:

“*To ensure that all your views are evenly spaced, click Margin in the toolbar to select the default margin for each view that you add to the layout”*.

This sentence does not require its previous sentence as the word that has a different context here in the sentence.

This is quite a challenge, because when we manually read it, we can analyze and figure out the sentence’s dependency based on the context. But how to do this automatically for thousands of text documents?

NLP comes to the rescue here. Natural language processing helps computers communicate with humans in their own language and scales other language-related tasks. For example, NLP makes it possible for computers to read text, hear speech, interpret it, measure sentiment, and determine which parts are important.

Here we use NLP to find the parts of speech (POS) tags of each word present in the sentence and its token dependency. We use SpaCy library to do the POS tagging. The Part of speech tagging, or POS tagging is the process of marking a word in the text to a particular part of

speech based on both its context and definition. In simple language, we can say that POS tagging is the process of identifying a word as nouns, pronouns, verbs, adjectives, etc. Some words can function in more than one way when used in different circumstances. The POS Tagging here plays a crucial role to understand in what context the word is used in the sentence.

Example 5:

An app might have a one-time setup or series of login screens. These conditional screens should not be considered start destinations because users see these screens only in certain cases.

In the above example, the second sentence will be extracted since it contains the constraint *should* but the sentence by itself does not provide any complete information. Therefore, the above sentence structure will be analyzed where part of speech (POS) tags and dependency tags are used for checking dependency as shown in Figure 6. The second sentence contains the keyword *these* indicating dependency to the previous sentence about login screens. Therefore, when we look at its previous sentence (i.e., the first sentence),

"An app might have a one-time setup or series of login screens"

Thus, we are extracting the previous sentence along with the current sentence. In few scenarios, we may have to extract up to two previous sentences to get a clear understanding of sentences.

TABLE 2

SUMMARIZATION OF POS/DEPENDENCY TAGS

Keyword	POS Tagging -Token Dependency	Check Punctuation	Previous sentence(s) required (Yes / No)
this	‘det’/ ‘pobj’	NA	Yes
this	‘nsubj’, dobj	Check for comma (,) in the sentence	If comma present- No If comma not present - Yes
that	‘det’	NA	Yes
that	‘mark’/ ‘nsubj’/ ‘nsubjpass’	NA	No
instead	NA	Check for comma (,) in the sentence	If comma present- Yes If comma not present- No
them	‘dobj’	Check for comma (,) in the sentence	If comma present- Yes If comma not present- No
these	‘det’	NA	No
so	‘cc’	NA	No
so	‘advmod’	NA	Yes
such	‘amod’	Check for comma(,) in the sentence	If comma present- Yes If comma not present- No
such	‘predet’	NA	Yes

7. If the current sentence does not require or depend on its previous sentence, Step 6 is skipped. Now we must check if the current sentence with constraint requires its next sentence.

Keywords to look for:
nextcons= [for example, for instance, this, these, they, it, so]

In this step, we check the *next sentence* for the presence of keywords from *nextcons* list and analyze its POS tags and dependencies to check if the next sentence is independent or dependent on current sentence. If the next sentence is found to be dependent on the current sentence, we extract current sentence along with the next sentence.

<u>Example</u>: *"The name of each package where the change's default state (either enabled or disabled) must have been overridden. For example, if this is a change that is enabled by default, your app's package name would be listed if you toggled the change off using either the developer options or ADB"*.

In the above example, there are two sentences, and the first sentence will be extracted for the presence of constraint *must.* Though the first sentence provides meaningful information by itself, we look at the next sentence, it starts by giving an example for the context mentioned in the first sentence. Thus, second sentence depends on the current sentence and extracting the second example sentence will be helpful. The extracted sentences are combined after the current sentence.

8. The next step is to check whether the extracted sentences are coherent and has continuity. We used sentence transformers to encode the sentences and cosine similarity to calculate the similarity score between the sentences. A score above 0.3 implies that both sentences

have at least a few words in common thereby assuming both sentences have some continuity.

9. Then the sentences are put into a final DataFrame for further grouping using K-Means algorithm.
10. Thus, every batch input is processed, and the above steps are repeated to obtain a set of sentences, and these are appended to final DataFrame.

3.5 Grouping the data

We are using K-means, an unsupervised machine learning algorithm to cluster the text data. K-means algorithm works by calculating centroids and assigns the datapoint to the nearest centroid. The sentences in the final DataFrame are processed for clustering and grouping based on their relevance. We are using K-means, an unsupervised machine learning algorithm to cluster the text data.

Before being able to run k-means on a set of text documents, the documents have to be represented as mutually comparable vectors. Each data in our dataset (DataFrame) is a vector of attributes, i.e., features without a specific label that could assign it to a specific cluster or class. First, the text data is vectorized and vectors of the document are used for calculating the centroids.. To achieve this task, the documents can be represented using the tf-idf score. The tf-idf, or term frequency-inverse document frequency, is a weight that ranks the importance of a term in its contextual document corpus. Term frequency is calculated as normalized frequency, a ratio of the number of occurrences of a word in its document to the total number of words in its document. It's exactly what it sounds like, and conceptually simple, and can be thought of somewhat like a fraction of the document that is a particular term. The division by the document length prevents a bias toward longer documents by "normalizing" the raw frequency into a comparable scale. The

inverse document frequency is the log (no matter the base, because it scales the function by a constant factor, leaving comparisons unaffected) of the ratio of the number of documents in the corpus to the number of documents containing the given term. Inverting the document frequency by taking the logarithm assigns a higher weight to rarer terms. Multiplying together these two metrics gives the tf-idf, placing importance on terms frequent in the document and rare in the corpus.

K-means clustering works by assigning data points to a cluster centroid, and then moving those cluster centroids to better fit the clusters themselves. K-Means algorithm steps are as follows:
Step 1: Randomly pick k centroids from the sample points as initial cluster centers.
Step 2: Assign each sample to the nearest centroid ˆ(j), j 1, . . . , k..
Step 3: Move the centroids to the center of the samples that were assigned to it.
Step 4: Repeat steps 2 and 3 until the cluster assignments do not change or a user defined tolerance or maximum number of iterations is reached.

3.6 Implementation

Due to the distinct structure of Android web pages, it was tedious to provide a generic implementation and involved manual inspection of elements using *inspect element* feature. The prototype is written in Python 3 and integrates the python libraries.

The first generic step is downloading the android documentation web pages and transforming the HTML files to plain text files by following a series of steps mentioned in section III, subsection Due to the distinct structure of android web pages, it was tedious to provide a generic implementation and involved manual inspection of elements using *inspect element* feature.

For the web scraping step, python along with its rich set of utility libraries is considered. The following three aspects of content extraction are considered: Beautiful Soup along with lxml

parser, regular expression. Content extractions are executed on the android documentation web pages with the matching pattern using regular expression. The python library, beautiful soup along with the lxml parser is used for pulling out content from android developer guide web pages. Regular expression is used to find elements by tag names to extract necessary text.

```
soup = BeautifulSoup(open(filename), "lxml")
classes = ['lang-xml', 'lang-kotlin', 'lang-java', 'attempt-right', 'devsite-n
           'devsite-breadcrumb-link gc-analytics-event', 'devsite-thumb-label'
           'devsite-icon devsite-icon-arrow-drop-down','img-caption']

tags = ['pre', 'span', 'a', 'div', 'devsite-content-footer', 'figure', 'figcap
for tag in tags:
    for i in classes:
        for span in soup.find_all(tag, {'class': i}):
            span.decompose()
for script in soup(["script", "style", "table"]):
    script.decompose()
```

Figure 10 : Cleaning unwanted tags

The above code block cleans up unwanted HTML tags so as to remove the contents enclosed in tags containing links, figure, footer and caption, style and script. This will help in filtering the unwanted text thereby focusing only on the relevant text tags .

```
def strongtext_between_tags(soup, filename):
    for strong_tag in strong_tags:

        links = soup.find_all(strong_tag, class_=re.compile("note|caution", re.IGNORECASE))
        if links:
            # pattern=re.compile(r'^Note:.*')
            for str in strong_words:
                pattern = re.compile(str, re.IGNORECASE)
```

Figure 11 : Extracting Strong tags Sentences

The above code block performs extracting the text between strong tags. Strong tags are those that carry important information and they usually go by Tip, Note, Caution. Regular expressions are used to search for specific text patterns and we search for “note” and “caution” to capture the text.

To analyze the extracted text and its sentences, natural language processing techniques are employed as described in the overall methodology. NLP libraries, NLTK and SpaCy library are used. NLTK --the Natural Language Toolkit, is a suite of open-source Python modules, supporting research and development in Natural Language Processing. A variety of tasks can be performed using NLTK such as tokenizing, parse tree visualization, etc. Since we have huge chunks of paragraphs in the form of text files, we need an efficient way to split them into sentences for processing and NLTK goes a long way achieving this.

We also use SpaCy, an open-source Python library, efficient and accurate to tokenize sentences and perform part of speech (POS) tagging. POS tags are used in analyzing the sentence structure along with its corresponding text. The reason behind using two NLP libraries is that each has its own advantage over the other in performing different tasks. In word tokenization and POS-tagging spaCy performs better, but in sentence tokenization, NLTK outperforms spaCy. Using the above NLP libraries, sentence structure is analyzed for checking constraints and dependency.

Sentences are first tokenized into words and the words are individually looped. Based on the POS tags and dependency tags, we decide whether the sentence is dependent on its previous or next sentence. If dependency is found, we go on extracting its dependent sentence. The dependent sentence in turn is processed the same way to maintain the coherency and context of the sentence to look complete and whole. As mentioned above, the text processor works up to 2 sentences before and after to fetch its dependent sentences. We use Pandas Dataframe for easy and fast traversal of the sentences just by navigating the index before and after.

```
def checkbef(df,doc,index,nprev):                                    ⚠ 5  ⚠ 400  ✓ 245
    for tok in doc:
        if tok.text == 'this' and (tok.dep_ == 'det' or tok.dep_ == 'pobj' or tok.dep_
            fullsents = chkbfconst(df,doc,index,nprev)
            return fullsents

        elif tok.text == 'This':
            if doc[0].text == tok.text:
                fullsents = chkbfconst(df,doc,index,nprev)
                return fullsents

        elif tok.text == 'this' and (tok.dep_ == 'nsubj'):
            punct = checkpunct(doc)
            if punct == 0:
                fullsents = doc.text
            else:
                fullsents = chkbfconst(df, doc, index, nprev)
            return fullsents

        elif tok.text == 'Thus':
            if doc[0].text == tok.text:
                fullsents = chkbfconst(df,doc,index,nprev)
                return fullsents
```

Figure 12 : POS and dependency tagging

After extracting the sentences, sentence similarity is examined using SentenceTransformer package which is used to generate the sentence embeddings and with those encodings, cosine similarity score between the sentences can be accurately calculated ensuring coherence and relevance.

```
    emb1 = model.encode(doc_1, convert_to_tensor=True)
    emb2 = model.encode(doc_2, convert_to_tensor=True)
    score = util.cos_sim(emb1,emb2)[0]
```

Figure 13: Sentence encoder to check similarity

CHAPTER 4

EVALUATION

The experiments were performed on a stream of 500 Android technical documentation webpages. From the 500 Android web pages, 3159 constraints were extracted from <main> tags text, 883 constraints from tags and 149 constraints from <table> tags as shown in Table 3.

AR-Extractor reported a total of 4191 true constraints aka true positives. True positive means that all the sentences extracted contains the constraint keywords. Thus the text processor was effective in identifying the sentences with constraints and extracted only those sentences with constraints. True positives are essential for identifying false positives and false negatives to determine effectiveness.

TABLE 3

EVALUATION RESULTS

	True (TP)	Over-reported (FP)	Under-reported (FN)	Precision	Recall	F-measure
<main> tags	3159	0	251	$\frac{3159}{3159+0} = 100\%$	$\frac{3159}{3159+251} = 92.6\%$	$2 * \frac{100 * 92.6}{100+92.6} = 96.1\%$
tags	883	0	69	$\frac{883}{883+0} = 100\%$	$\frac{883}{883+69} = 92.7\%$	$2 * \frac{100 * 92.7}{100+92.7} = 96.2\%$
<table> tags	149	0	0	$\frac{149}{149+0} = 100\%$	$\frac{149}{149+0} = 100\%$	$2 * \frac{100 * 100}{100+100} = 100\%$

All 4191 extracted sentences were manually checked for the presence of false positives. False positive means to check if the text processor has extracted any sentence without the constraint keywords. It was found that no unwanted sentences (sentences without constraint) were present. Therefore, for the overall method, it can be stated that every sentence that was marked as a constraint truly is a constraint, so the approach did not generate any false positives and resulting in a 100% precision.

There is no need to find 'True negatives' in our case, because we don't want to identify or deal with any sentence that does not have any constraints. Hence we are ignoring the True negative case.

When AR-extractor reported 4191 constraints, it under-reported a total of 320 false negatives. False negative means that there were sentences with constraints but missed by AR-Extractor. We consider two cases of False negative here:

Sentences that has constraints keywords present, but not extracted.Though sentences with constraints were extracted, AR-Extractor missed extracting its previous dependency sentences making the current sentence with constraint incomplete.

For false negatives, we identified two reasons.

1) NLP pattern matching might fail to find potential sentences with constraints.
2) Though few sentences with constraints were reported by AR-Extractor, the sentences were incomplete or not useful because they were referring to a code snippet, figure, table or its previous dependency sentences and without them there would not be any wholesome information.

The incomplete sentences with constraints were half and half, i.e., some sentences were partly seen as redundant or not useful.

Below are some of the example sentences that were considered unclear or not useful.

- *"Create a method for starting face detection so you can call it as needed, as shown in the example code below"*

The above sentence is not useful without the code snippet that is indicated by the keywords *code below.*

- "Figure 3 *shows the same set of Preferences split on multiple screens: To link screens with a Preference, you can declare an app:fragment in XML, or you can use Preference.setFragment(). This method is where you should handle displaying the new screen and should be implemented in the surrounding Activity."*

The above example refers to a figure that is indicated by the keyword Figure 3. This example is not completely useful without the figure.

- *"As noted previously, the User Dictionary Provider requires the Android.permission.READ_USER_DICTIONARY permission to retrieve data from it."*

Though AR-Extractor extracted the above sentence due to the keyword *noted*, it did not extract its previous dependency sentence. The above sentence refers to its previous sentence, to something that was said previously which is indicated by the keyword *previously* and, it is also not clear to what extent of its previous sentences the current sentence refers to.

Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances, while recall (also known as sensitivity) is the fraction of relevant instances that were retrieved. Both precision and recall are therefore based on relevance.

Based on our results, text processor was able to achieve 100% precision which means that desired information was extracted successfully and 92% recall because of few sentences with constraints not extracted due to the reasons mentioned above.

4.1 Discussions and Limitations

AR-Extractor has three limitations.

- First, it cannot process *links* (i.e., hyperlink or weblink) within the extracted Android HTML documentation. i.e., links enclosed by href attribute:

  <a href="URL"> For example:

  The following HTML excerpt contains a link, whose destination anchor is a link referring to the topic *Optimizations for updating widget content.*

```
<a href="https://developer.android.com/guide/topics/appwidgets/advanced?hl=fi#update-widgets">
Optimizations for updating widget content</a> == $0
```

Figure 14 : href attribute

- Second, since we do not focus on code blocks and extract only the annotated document, AR-Extractor cannot extract constrained sentences within the code blocks. Below is an example *XML comment sentence with the constraint should,* that is present inside the XML code block.

Example : if a user clicks on a shared link that uses the "http" scheme, your app should be able to delegate that traffic to "https".

```
<!-- Make sure you explicitly set android:autoVerify to "true". -->
<intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />

    <!-- If a user clicks on a shared link that uses the "http" scheme, your
         app should be able to delegate that traffic to "https". -->
    <data android:scheme="http" />
    <data android:scheme="https" />

    <!-- Include one or more domains that should be verified. -->
    <data android:host="..." />
</intent-filter>
```

Figure 15 : XML code with constraint should

- Third, sentences with keywords such as *all of this, each of the these, neither of these,* indicate uncertainty (i.e., dependence to previous or next sentence) and will be difficult to process efficiently.

  Example : All of this is important because the synthetic back stack must be realistic

  However, the current sentence itself cannot be processed effectively with uncertain keywords, *all of this.* It is unclear that to what extent *all of this* refers to previous sentences.

The above-mentioned limitations can be solved by writing separate regex and NLP rules for exceptional cases and sentences.

CHAPTER 5

CONCLUSION

In this paper, a novel approach for extracting sentences containing constraints and solving dependencies between them was presented. Relations could be resolved based on a natural language processing technique. NLP framework as well as common data mining algorithms were used for implementing the method. NLP methodology was very effective in performing the POS tagging and obtaining dependency tags, which formed the base to extract relevant and meaningful information.

5.1 Related Works

Detecting constraints and their relations from regulatory documents using NLP techniques [5]: Investigated an approach for detecting constraints and their relations from regulatory documents. Their work denoted that it required manual inspection for the integration of external information for deriving the list of constraint related subjects. Whereas our work has an advantage of extracting up to two previous sentences if the current sentence is dependent on subject of previous sentences leading to sentence completeness without requiring manual inspection of constraint related subjects.

PR-Miner [6] uses data mining technique called as frequent itemset mining to extract implicit programming rules from large software code effectively. In PR-Miner, source code is parsed and then they convert the programming pattern extraction problem into an itemset mining problem. It implemented two techniques for generating programming rules by using the association rule mining algorithm probability of generating false-positive rules is controlled by keeping the minimum support and minimum confidence value high.

Research on Keyword Extraction Algorithm in English Text Based on Cluster Analysis[7] employs data extraction by using text clustering method to automatically extract keywords. Providing keywords will help in text summarization. Text processing is done by word segmentation, pre-processing and converting the word into vectors and calculate the positional importance of the word vectors and finally obtain the relevant keywords, whereas our work focuses on data extraction with our predefined set of keywords and while we look for those sentences with the keywords, we also analyze its coherency and extract previous or next sentence as needed.

Text mining approaches for dealing with the rapidly expanding literature on COVID-19 [8] focuses on retrieving relevant documents from a huge volume of expanding COVID 19 literature through text embeddings, annotations, knowledge graphs and language models to obtain relationship between entities, extract structured evidence from different studies and summarize the evidence and interpret the findings accordingly. This helps to have a centralized corpus of documents related to COVID 19 that can be updated regularly.

5.2 Future Work

The future research directions based on the work, in extraction of android programming constraints could be

• To develop an android studio plugin for providing automatic constraints suggestions in android Integrated Development Environments (IDEs')

• Develop a natural language processing algorithm for detecting sentences with uncertain keywords from android documentation web pages. This would prevent the AR-Extractor from extracting incorrect dependencies for sentences with constraints.

• To write separate regex rules for identifying links enclosed by href attributes. This can be used to unfold and extract numerous android constraints from the enclosed links

5.3 Publication

P. Santhanam, P. Sakthivel and Z. Shan, “AR-Extractor: Automatically Extracting Constraints and their relations from Android Documentation using NLP techniques”, In the proceedings of the 20th International Conference on Software Engineering Research and Practice (SERP’22), Las Vegas, USA, 25. Jul.2022

REFERENCES

REFERENCES

[1] S. Thivaharan., G. Srivatsun., and S. Sarathambekai., "A survey on python libraries used for social media content scraping," *2020 International Conference on Smart Electronics and Communication (ICOSEC)*, 2020.

[2] "Python pandas.read_html() examples," *Python Examples of pandas.read_html*. [Online]. Available: https://www.programcreek.com/python/example/101351/pandas.read_html. [Accessed: 17-Feb-2022].

[3] Parsing XML and HTML with LXML. [Online]. Available: https://lxml.de/parsing.html. [Accessed: 17-Feb-2022].

[4] Real Python, "Beautiful soup: Build a web scraper with python," *Real Python*, 10-Nov-2021. [Online]. Available: https://realpython.com/beautiful-soup-web-scraper-python/. [Accessed: 17-Feb-2022].

[5] K. Winter and S. Rinderle-Ma, "Detecting constraints and their relations from regulatory documents using NLP techniques," *Lecture Notes in Computer Science*, pp. 261–278, 2018.

[6] Z. Li and Y. Zhou, "PR-miner," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[7] Research on Keyword Extraction Algorithm in English Text Based on Cluster Analysis, Jingxia Ma, School of Western Languages and Cultures, Harbin Normal University, Harbin 150025, China

[8] Lucy Lu Wang, Kyle Lo, Text mining approaches for dealing with the rapidly expanding literature on COVID-19, *Briefings in Bioinformatics*, Volume 22, Issue 2, March 2021, Pages 781–799

[9] B. Sluban and M. Grčar, "Url Tree," *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM '13*, 2013.

[10] "Beautiful Soup documentation¶," *Beautiful Soup Documentation - Beautiful Soup 4.9.0 documentation*. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/. [Accessed: 17-Feb-2022].

[11] "Re - regular expression operations¶," *re - Regular expression operations - Python 3.10.2 documentation*. [Online]. Available: https://docs.python.org/3/library/re.html. [Accessed: 17-Feb-2022].

[12] T. S. Zaman and T. Yu, "Extracting implicit programming rules: Comparing static and dynamic approaches," *Proceedings of the 7th International Workshop on Software Mining*, 2018.

[13] “Web scraping in Python,” *The Python Book*, pp. 229–254, 2022.

[14] Real Python, “A practical introduction to web scraping in Python,” *Real Python*, 06-Mar-2021. [Online]. Available: https://realpython.com/python-web-scraping-practical-introduction/. [Accessed: 17-Feb-2022].

[15] B. V. S. Ujwal, B. Gaind, A. Kundu, A. Holla, and M. Rungta, “Classification-based adaptive web scraper,” *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017.

[16] Fatmasari, Y. N. Kunang, and S. D. Purnamasari, “Web scraping techniques to collect weather data in South Sumatera,” *2018 International Conference on Electrical Engineering and Computer Science (ICECOS)*, 2018.

[17] I. Onyenwe, E. Onyedinma, C. Nwafor, and O. Agbata, “Developing products update-alert system for E-commerce websites users using HTML data and web scraping technique,” *International Journal on Natural Language Computing*, vol. 10, no. 5, pp. 01–07, 2021.

[18] K. Jiang and X. Lu, "Natural Language Processing and Its Applications in Machine Translation: A Diachronic Review," 2020 IEEE 3rd International Conference of Safe Production and Informatization (IICSPI), 2020, pp. 210-214, doi: 10.1109/IICSPI51290.2020.9332458.

[19] R. Dhumal Deshmukh and A. Kiwelekar, "Deep Learning Techniques for Part of Speech Tagging by Natural Language Processing," *2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, 2020, pp. 76-81, doi: 10.1109/ICIMIA48430.2020.9074941.

[20] Li, H.; Mao, H.; Wang, J. Part-of-Speech Tagging with Rule-Based Data Preprocessing and Transformer. Electronics 2022, 11, 56. https://doi.org/10.3390/electronics11010056

[21] A. Srivastava and M. Sahami, Text mining: Classification, clustering, and applications. CRC Press, 2009.

[22] K. P. Sinaga and M. Yang, "Unsupervised K-Means Clustering Algorithm," in *IEEE Access*, vol. 8, pp. 80716-80727, 2020, doi: 10.1109/ACCESS.2020.2988796.

[23] J. Qi, Y. Yu, L. Wang and J. Liu, "K*-Means: An Effective and Efficient K-Means Clustering Algorithm," *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, 2016, pp. 242-249, doi: 10.1109/BDCloud-SocialCom-SustainCom.2016.46.