# AR-Extractor: Automatically Extracting Constraints from Android Documentation using NLP techniques

Preethi Santhanam[1], Padmapriya Sakthivel[1†] and Zhiyong Shan[1*†]

[1]School of Computing, Wichita State University, 1845 Fairmount St, Wichita, 67220, Kansas, USA.

*Corresponding author(s). E-mail(s): zhiyong.shan@wichita.edu;
Contributing authors: pxsanthanam1@shockers.wichita.edu;
pxsakthivel@shockers.wichita.edu;
†These authors contributed equally to this work.

**Abstract**

When developing Android apps, it is difficult for the programmers to follow all programming constraints described in Android documents. This paper proposes a novel method called AR-Extractor (Android Rules Extractor) to extract the programming constraints automatically from Android developer documents using natural language processing techniques. It can help programmers to reduce bugs, improve software maintainability and reliability.

**Keywords:** Android Developer Guide, Constraint Extraction, Text Dependency, Part of Speech tags

**Regular Research Paper**

## 1 Introduction

As the most popular operating system Android provides plenty of documents for developers. Manually investigating all the textual data and extracting the

most programming constraints is clearly a tedious process.Therefore, an automated way of extracting constraints from official Android developers guide and presenting it in a structured manner will help developers when creating Android applications.

In this paper, we present a novel method called AR-Extractor that extracts programming constraints from the official Android developers guide. Android programs often require and follow programming constraints such as, *if you want to manually handle orientation changes in your app you must declare the "orientation", "screenSize", and "screenLayout" values in the android:configChanges attributes.* If the programmer violates the above rule, then it causes the activity to not start. These kinds of constraints are important for the Android application development.

Our AR-Extractor uses natural language processing technique for sentence extraction. It focuses on extracting sentences with constraints such as *must, should, note, warning, noted, recommended, need, tip, remember, caution, warning, hint, make sure, cautious.* Sentences are scraped by exploiting the part-of-speech (POS) and dependency tags of sentences leading to a more purposeful extraction containing constraints.

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 constraints as well as dependencies between constraints to be able to implement them correctly [1]. The tool methodology is explained in section 2.

Our results indicate that our tool can effectively identify Android programming constraints.

## 2 Methodology

AR-Extractor consists of four steps as shown in Fig 1. The first step is to *download HTML documents* using a website crawler or spider called Screaming Frog to crawl and download web pages automatically. In this paper, we automatically downloaded 500 Android developer guide web pages.

In the *data extraction* step, we performed web scraping on Android developers guide web pages to get necessary HTML data. Web scraping method is automated using a python library called beautiful soup [2]. Our code is designed to pass data scraping based on the HTML tags (e.g., *p, a, span* ) and using pattern based regular expression [3]. This process is carried out to filter constraints from the annotated document.

In the *data processing* step, the sentences with constraints are identified and explored for further dependencies where sentences are usually associated to each other and one sentence might refer to the subject of previous or next sentence. Natural language processing (NLP) library is used for analyzing sentences and parts of speech (POS) tags to determine if the sentence gives

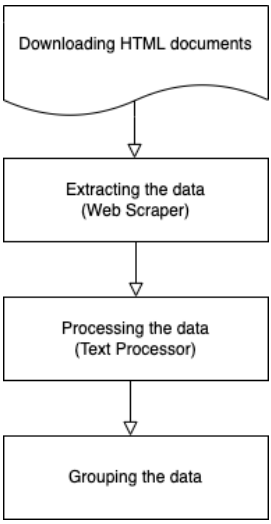


**Fig. 1**: Overall Method

complete information or previous or the next sentences' information needs to be extracted to give a meaningful information.

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 [4]. These are formed as clusters; thus, relevant sentences are categorized for easy use and readability.

## 2.1 Challenges

To our knowledge, there has been no effort so far to automate the extraction of Android programming constraints. A lot of challenges arise while extracting the relevant and meaningful text data. Firstly, it requires a lot of time and effort to go through the data manually for extraction from Android developers guide web pages and then automating the approach using steps as described in Fig 1.

Every webpage is designed in a distinct way, and it requires exploring the Android documentation for the existence of programming 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. It required testing around 100 web pages for element inspection to identify necessary tags using *inspect element* feature. At times, one sentence might refer to the subject of next or previous sentence, so separate NLP rules were written based on POS tags [5] as shown in Table 1.

# 3 Extracting Data

In this paper, it is carefully analyzed, how the programming constraints are pulled-out from the Android documentation web pages. The steps are as follows:

## 3.1 Step 1

The HTML documents, i.e., Android developer guide web pages are given as an input to the web scraper as shown in Fig 2. The input is processed in batches with each batch containing 50 HTML files. The downloaded Android documentation web pages need to be cleaned before being used for further data extraction.

## 3.2 Step 2

For cleaning and extracting the necessary content from Android HTML documents, we use a python library called beautiful soup. Since only annotated document is used for the process, unnecessary information is removed as shown in Fig 2. The web scraper script extracts three types of contents separately from the Android HTML files and they 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 above three content extractions are discussed in section 3.3, step 3.

## 3.3 Step 3

### 3.3.1 Extracting entire text from $<$ *strong* $>$ and its associated tags

The entire text enclosed by $< strong >$ and its associated tags are considered of high importance. The constraints considered for $< strong >$ tag text extraction is *note, warning, caution, tip, hint, important*. These constraints are commonly enclosed within $< strong >$ tag and its contents are within tags such as $< p >$, $< li >$, $< aside >$, $< ul >$, $< div >$ as shown in Fig 4.

If multiple sentences enclosed by $< strong >$ and its related tags are not dependent, the text processor script will split the sentences in the tokenization process. This is the reason to extract $< strong >$ tags text separately.

For example, Fig 3 shows an example of constraint *note*. In Fig 4, after element inspection, constraint *note* is enclosed within $< strong >$ tag but its entire associated text is surrounded by $< p >$ tag with attribute *class="note"*. This complete text related to $< strong >$ tag is called as $< strong >$ tag text extraction. The constraints surrounded by $< strong >$ tags are identified using regular expression ^note:.*, ^caution:.*, ^tip:.*, ^warning:.*. The extracted $< strong >$ tag text is then written to .xlsx files.

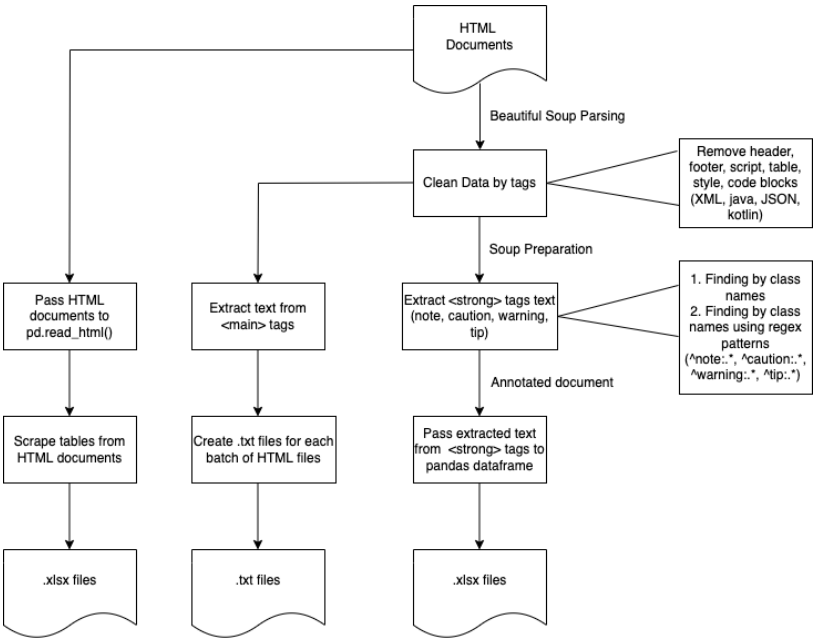


**Fig. 2**: Structure of Web Scraper

### 3.3.2 Extracting 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 HTML files. The converted text files are passed as an input to the text processor script for exploring dependencies.

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.

★ **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.

**Fig. 3**: An example of constraint *note*

### 3.3.3 Extracting text from < *table* > tags

In addition, it should be checked whether information from tables is needed and is parsed correctly. For this process, all tables enclosed by < *table* > tags

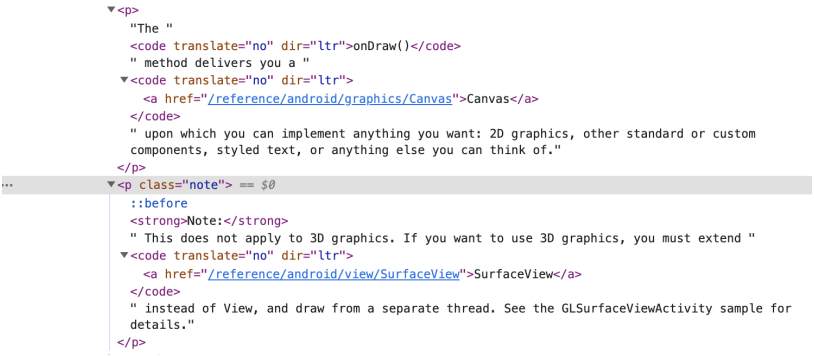


**Fig. 4**: Element inspection of constraint *note*

from the Android documentation pages are scraped using pandas *read_html* function and written to .xlsx files.

The .xlsx files are passed as an input to the text processor script to be analyzed for the presence of constraints.

# 4 Processing the Data

Web scraper script generates three sets of output,

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

## 4.1 Processing HTML Tables

The .xlsx files containing table information from the *web scraper* script 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 and if constraints are present, that specific row containing constraint 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. The extracted table rows with constraints are passed to an excel sheet.

## 4.2 Processing the $< strong >$ tag text

The $< strong >$ and its related tags are denoted in the HTML files with the attribute class = note | caution | warning | special.

In Fig 3 and 4, the two sentences with constraint *note* are surrounded by $< p >$ tag as mentioned in section 3.3.1. But the first sentence within $< p >$ tag refers to its previous sentence which is indicated by the keyword *this*.

In this case, it is necessary to extract its previous sentence for sentence completeness as mentioned below.

*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.*

Thus, processing of < *strong* > tag text requires two inputs.

- The first input is the < *strong* > tag .xlsx file and
- The second input contains the list of text files that is extracted from the < *main* > tags. Processing of text from < *main* > tags text is explained in next section 4.3

The DataFrame containing the < *strong* > tags are iterated, and each row is analyzed for keywords like *this, these, however, therefore, hence* using NLP and if encountered, the same sentence is searched in the text file DataFrame by looping through the rows. If the similar sentence from < *strong* > 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.

## 4.3 Processing the text from < *main* > tags

The extracted text from < *main* > tags are converted into .txt files by web scraper script. We created the following three lists for processing sentences.

- A *befwords* list with a set of keywords refers to previous sentences i.e., *this, that, these, furthermore, however, hence, otherwise etc.*
- A *nextcons* list with a set of keywords refers to its next sentences i.e., *for example, after, following, next 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.*

Each text file contains a set of paragraphs that are tokenized into a list of sentences and then passed to a DataFrame for easier and faster processing.

Firstly, the sentences are checked for the presence of constraints and if any constraints are present in the current sentence, then the sentence is analyzed for keywords in the *skip* list and if found, the current sentence is ignored or skipped for further processing. The following is an example sentence which will be skipped since it refers to a code snippet that explains the implementation.

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

If the current sentence with constraint doesn't contain keywords from *skip* list, then the second step is to check the current sentence for its dependency to the previous sentence. The current sentence is checked for the presence of keywords from *befwords* list. This is done by analyzing the sentence structure using natural language processing technique where part of speech (POS) tags and dependency tags are used for checking dependency [6] as shown in Table 1.

For example: *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*. Therefore, the second sentence structure will be analyzed where it contains the keyword *these* indicating dependency to the previous sentence about login screens. Thus, we are extracting the previous sentence along with the current sentence.

Now, the next step is to check if the current sentence with constraint requires its next sentence and check the next sentence for the presence of keywords from *nextcons* list and analyze its POS tags and dependencies.

*Example: 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 constraint *must*. Though the first sentence provides meaningful information by itself, second sentence starts by giving an example for the context mentioned in the first sentence. Thus, extracting the second example sentence with first sentence will be helpful for understandability. A sample sentence tree structure is omitted to conserve space in this paper but sentences' POS tags and dependencies can quickly be visualized in the browser using displaCy. The next step is grouping the sentences using K-Means algorithm.

# 5 Grouping The Data Using K-means Algorithm

We are using K-means, an unsupervised machine learning algorithm to cluster the text data. Each data in our dataset (DataFrame) is a vector of attributes. First, the text data is vectorized and vectors of the document are used for calculating the centroids [7]. Number of clusters is determined using the elbow method. In this way relevant text data is grouped and put into different categories [8].

# 6 Implementation

A prototypical implementation of the method is described in section 2. The prototype is written in python 3 and integrates the python libraries.

For the web scraping step, the following aspects of content extraction are considered: beautiful soup along with lxml parser, pattern based regular expression.

For the text processing step, natural language processing techniques are employed as described in the overall methodology. NLP libraries such as NLTK and SpaCy are used. After extraction, sentence similarity is examined using SentenceTransformer package to generate the sentence embeddings. With those

**Table 1**: Summarization of POS tags of keywords and their dependency

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

encodings, cosine similarity score between the sentences can be accurately calculated ensuring coherence and relevance.

# 7 Evaluation

AR-Extractor reported a total of 4191 true constraints aka true positives (TP). True positives are essential for identifying false positives (FP) and false negatives (FN) to determine effectiveness as shown in Table 2

**Table 2**: Evaluation Results

| Tags | True | FP | FN | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|
| $< main >$ | 3159 | 0 | 251 | $\frac{3159}{3159+0} = 100\%$ | $\frac{3159}{3159+251} = 92.6\%$ | $2 * \frac{100*92.6}{100+92.6} = 96.1\%$ |
| $< strong >$ | 883 | 0 | 69 | $\frac{883}{883+0} = 100\%$ | $\frac{883}{883+69} = 92.7\%$ | $2 * \frac{100*92.7}{100+92.7} = 96.2\%$ |
| $< table >$ | 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. Therefore, the approach did not generate any false positives and resulted in 100% precision.

When AR-extractor reported 4191 constraints, it under-reported a total of 320 false negatives (sentences with constraints but missed by AR-Extractor). For false negatives, we identified two reasons. First, NLP pattern matching might fail to find potential sentences with constraints. Second, though few sentences with constraints were reported by AR-Extractor, the sentences were incomplete or not useful without code, or its previous dependency sentences.

## 8 Discussions and Limitations

Our 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: $< ahref = "URL" >$

Second, since we do not focus on processing code blocks and extract only the annotated document, AR-Extractor cannot extract constrained sentences within the code blocks i.e., such as code comments with constraints.

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.

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

## 9 Related Works

Detecting constraints and their relations from regulatory documents using NLP techniques [1]: 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.

PR-Miner [9] uses data mining technique called as frequent itemset mining to extract implicit programming constraints from large software code effectively. [10] implemented two techniques for generating programming constraints by using the association rule mining algorithm.

## 10 Conclusion

In this paper, a novel approach for extracting sentences containing constraints and solving dependencies between them was presented. NLP framework as well as common data mining algorithms were used for implementing the method. Python libraries beautiful soup, spaCy for implementing NLP methodology were very effective in performing the POS tagging and obtaining dependency tags, which formed the base to extract relevant and meaningful information.

## Declarations

This material is based upon work supported by the National Science Foundation under Grant No 2154483.

# References

[1] Winter, K., Rinderle-Ma, S.: Detecting constraints and their relations from regulatory documents using nlp techniques. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pp. 261–278 (2018). Springer

[2] Richardson, L.: Beautiful soup documentation. Dosegljivo: https://www. crummy. com/software/BeautifulSoup/bs4/doc/.[Dostopano: 7. 7. 2018] (2007)

[3] Hunt, J.: Regular expressions in python. In: Advanced Guide to Python 3 Programming, pp. 257–271. Springer, ??? (2019)

[4] Sinaga, K.P., Yang, M.-S.: Unsupervised k-means clustering algorithm. IEEE access **8**, 80716–80727 (2020)

[5] Li, H., Mao, H., Wang, J.: Part-of-speech tagging with rule-based data preprocessing and transformer. Electronics **11**(1), 56 (2021)

[6] Deshmukh, R.D., Kiwelekar, A.: Deep learning techniques for part of speech tagging by natural language processing. In: 2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), pp. 76–81 (2020). IEEE

[7] Kharazmi, M.A., Kharazmi, M.Z.: Text coherence new method using word2vec sentence vectors and most likely n-grams. In: 2017 3rd Iranian Conference on Intelligent Systems and Signal Processing (ICSPIS), pp. 105–109 (2017). IEEE

[8] Rinartha, K., Kartika, L.G.S.: Rapid automatic keyword extraction and word frequency in scientific article keywords extraction. In: 2021 3rd International Conference on Cybernetics and Intelligent System (ICORIS), pp. 1–4 (2021). IEEE

[9] Z.Li, Y.Zhou.: Pr-miner. ACM SIGSOFT Software Engineering Notes **30**(5), 306–315 (2005)

[10] Zaman, T.S., Yu, T.: Extracting implicit programming rules: comparing static and dynamic approaches. SoftwareMining 2018: Proceedings of the 7th International Workshop on Software Mining, 1–7 (2018)