Austin Bryant
CSC 497/498
Dr. Alex Dekhtyar
Spring/Fall 2021

Automated Detection of Completeness of Acceptable Use Policies for Wireless Networks: Process and Implementation Overview

Project Overview

This project sought to build upon research originally performed by Hayes et al. in *"Towards Improved Network Security Requirements and Policy: Domain-Specific Completeness Analysis via Topic Modeling."* Specifically, effort was focused on data collection, preprocessing, repeatability, and creating a solid foundation for future expansion on this topic. To complete these tasks, I designed and implemented new software solutions for automatically locating, collecting, and preprocessing acceptable use policy (AUP) documents from university websites, before processing with an off-the-shelf Latent Dirichlet Allocation (LDA) model.

Locating & Collecting

A list of universities was first scraped from the NCAA list of schools using the Python *requests* and *BeautifulSoup4* libraries. This produced a list of 1,085 university names in an easily iterable format. From there, these universities were used in a search query using the Google Custom Search API and associated Python library, with the domain restricted to ".edu" domains. Each university was first used in a query specifying that the filetype must be PDF, then later in a query without this restriction. This query was identical for each school, save the actual university name. One example is shown below.

Washington State University "wireless" acceptable use policy filetype:pdf

These result sets were stored in lists and pickled using the Python *pickle* library for easy future access without incurring further charges from API use. Once all queries had been run, a list comprehension was run on the PDF result list to extract the URL from the top result for each school. This was then transformed into a Python set object. The non-PDF results were processed in a similar manner, except each URL was checked to see if it had already been included in the PDF set. Both of these sets were then cast back to lists and pickled for further processing.

Preprocessing

The first step in preprocessing was to filter the lists of URLs through both negative and positive filters. The negative filter contained terms commonly seen in the URLs of non-AUP documents, such as "housing", "syllabus", "catalog", and "guest". This negative filtering removed approximately 50% of the PDF URLs, and 30% of the non-PDF URLs. The positive filter looked for the presence of one or more AUP associated terms in the URL, including "acceptable",

“wireless”, and “network”. This further reduced the size of the lists by 70% for the PDFs and 65% for the non-PDF list.

At this point, the preprocessing steps diverged significantly. Each PDF url had its content scraped by the *requests* library, before being written to a temporary file object in memory, provided by Python’s *tempfile* library. This file was then parsed by *PyPDF2*, a third Python library. To extract the text of the document, each page was iterated over, and all text was collected with the only filtering being a regular expression substitution for multiple newlines with a singular newline. This text was then fed to the primary preprocessor, which is detailed later. The processed text for each page was joined with a newline character before being written to a “.txt” file with the name of the university.

The non-PDF files were scraped using *Selenium* and *BeautifulSoup4*. The URL was loaded into Selenium to avoid anti-bot measures encountered by the *requests* library. The page source was fed into *BeautifulSoup4*, and all text-containing tags were collected into a list, filtered against a blacklist of non-relevant tags, and had their text extracted, stripped of leading and trailing whitespace, and were joined with a single space. This monolithic string was then tokenized to a sentence level, and the resulting list was fed to the primary preprocessor before being saved in a manner identical to the PDF text.

The primary preprocessor had multiple objectives: to remove various artifacts of extraction, recombination and tokenization, to properly split bulleted or numbered lists, and to remove empty or garbled tokens. This was accomplished through a series of regular expressions, splits, and joins, and resulted in text that was as close to the original source as possible while still being easily parsable by a model in the later steps.

The collected preprocessed documents were finally randomly sampled to create a test-train split of 0.8. The training documents were concatenated and saved as an independent file for ease of use.

Further Processing & Vectorization

With the concatenated file generated, the next step in the process was to properly tokenize and lemmatize the input “documents” (in this context, sentences). Each line of the combined document was processed by a *Spacy* “en_core_web_md” natural language processing model, which would tokenize, tag with part-of-speech, and lemmatize each word. These tokens were joined with a single space and the resulting string was appended to a list. This was the most time-intensive portion of training the model, taking over one minute per run.

To properly train a LDA model, the resulting documents first had to be vectorized. I selected *sklearn*’s CountVectorizer for this task, which output a matrix of shape (13584, 6802), with the first dimension corresponding to the number of unique words, and the second corresponding to the number of documents vectorized.

Model Training & Output

The LDA model used was the same as the one used in the original paper, with the parameters used also being identical. Twenty topics of 15 words each were generated through 2,000 iterations. The result vector was then transformed into a human-readable spreadsheet file containing the top words for each topic, as well as their relative weight. The vectorizer and model are also pickled as a tuple to be available for further analysis.

After the model is fully trained, an optionally provided AUP document will be processed in much the same manner, with the main difference being that its vectorization resolution is at the file level rather than the sentence level. The resulting vector is output to stdout and saved to a text file.

Instructions for Use

This program is split into several segments. There are three jupyter notebooks and two Python files: “Scraping.ipynb”, “HTML extraction.ipynb”, “PDF extraction.ipynb”, “generate_AUPs.py”, and “LDA.py”. All the notebooks must have all cells run, and generate_AUPs.py requires no user input. A requirements.txt file is also provided, and all specified packages must be installed.

Scraping is the first one that must be run, and requires a Google API Developer key and configured custom search engine in order to regenerate the search results pickles. The pickled results are provided with the repository, so this is unnecessary.

Both extraction notebooks only require the results pickles, and no further human input. These are the next to be run, and they generate all the AUP documents for later use.

The fourth file to be run is generate_AUPs.py, which creates the merged document and copies the other 20% of the files to a separate directory (“test_aups”) for further analysis.

Finally, LDA.py must be run. This file requires the *spacy* model “en_core_web_md” to be downloaded and takes five optional arguments:

- -m / --model: Specifies a pickled, pre-trained (vectorizer, model) tuple
    - Must specify -d / --document as described below
    - Ignores any model parameter flags specified, as no new model is trained
- -t / --topics: Number of topics the LDA model generates
- -i / --iterations: Number of iterations the model runs
- -w / --words: Number of words output to stdout and the .csv file upon completion
- -d / --document: Specifies an additional document to be vectorized

This file will save the model in the output directory, a .csv of the top words and their relative weights, and a document vector if an additional document is provided.