SE/CprE 492 Final Report

Group 35

A Part of Speech Tagger for Software Documentation

Bring the power and flexibility of natural language processing to software documentation.

Advisors
Ali Jannesari
Hung Phan

Team Members
Ahmad Alramahi - Lead Developer
Austin Boling - Meeting Facilitator
Joseph Naberhaus - Project Lead
Ekene Okeke - Report Coordinator
Ethan Ruchotzke - Documentation Manager
James Taylor - Linguistics SME

Table of Contents

Introduction

The purpose of this senior design project is to create a part of speech (POS) tagger which can accept software documentation in the form of HTML files. These files are then fed into a training pipeline which constructs a model based on them which can be used to tag future sets of documentation. The main problem that we are trying to solve with a newly created POS tagger is the issue that the current standard part of speech (POS) tagging solutions used in industry are not able to tag natural language alongside code. Our solution to this problem was an augmentation of the existing and commonly used Stanford NLP pipeline which is used for conventional natural language processing (NLP). This final report will go into detail about the structure of this product, as well as an explanation of the product from a usability standpoint.

Project Design

Requirements

Functional

- Input: Previously unseen, untagged software documentation in an HTML or XML-Like format.
- Output: A tagged software document in NLP format. Both English tags and custom code tags are included in the output tags.
- Pseudocode and English descriptions of code are taggable, and fit into the set of custom tags.

Non-Functional

- Creation of a new, custom set of PoS tags which fit the abstract concepts contained in code.
- Retrain the Stanford NLP model using tagged software documentation.
- Translate the manually tagged data into a format usable by the existing Stanford NLP training pipeline.
- Buildup of a large corpus of tagged software documentation.

Constraints

- The solution created must be built on top of, and directly integrated with the existing standard Stanford NLP pipeline.
- The solution must be created and proven viable within two semesters.
    - The project must be usable for future projects; establishing a strong pipeline is a necessity for continuation.
- The solution must be low, to no cost.
    - Stanford NLP is publicly available.
    - Utilization of university resources for the training of the model.

Standards of Development

- ISO-IEC 12207: Software Lifecycle
    - Influences on the design and longevity of this project.
- ISO-IEC 9001: Quality management
    - Influences on the quality and structure of this project.
- ECMA 494: JSON Format
    - Straightforward,easily serializable format used for inter-pipeline communication and output.

Operating Environment

- Java 15 / Java SE 16
- Python 3.8 or above
    - Jnius - integration into java pipeline.
    - Beautifulsoup - easy parsing of XML / HTML files.

System Structure

This system is a pipeline of components where files are scraped, parsed, tokenized, analyzed, and tagged. As this is a complex process, the structure of a pipeline is beneficial, as it allows for transparency between stages and straightforward debugging when the process goes wrong.

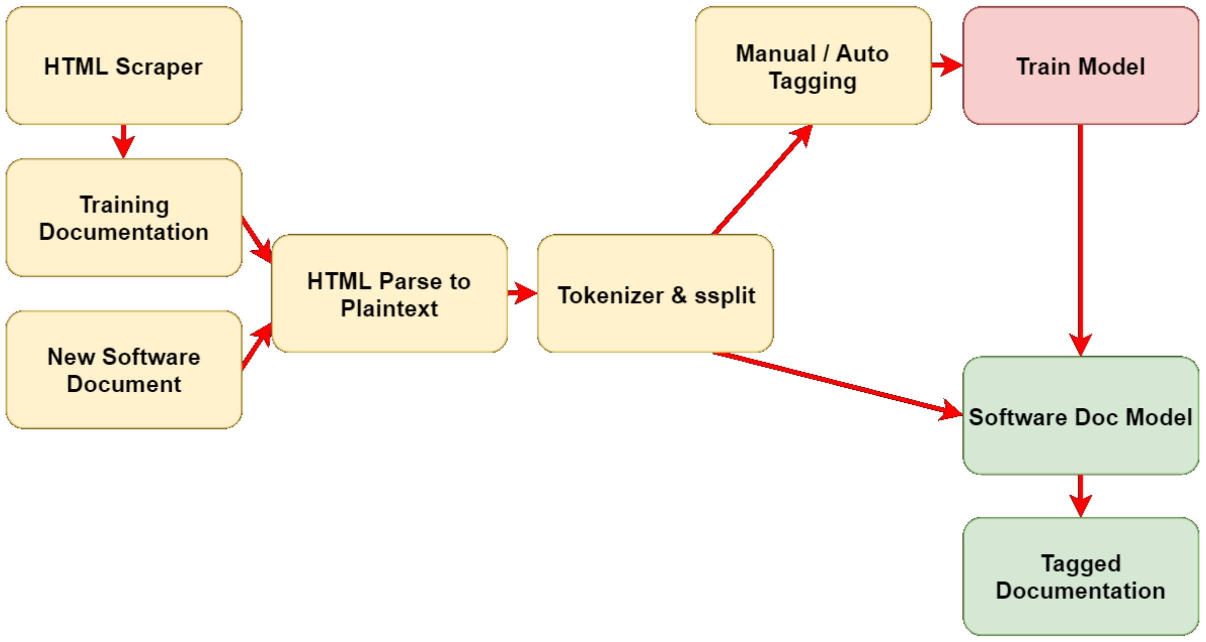


Figure 1: The design's pipeline structure.

The pipeline is broken into 3 general sections, based on color above. The first section, illustrated in yellow, is the preparatory portion of the pipeline, where data is parsed and prepared for tagging. The red section is the training-only section, dedicated to training a model for later tagging. The final section is green, and dedicated to tagging and outputting formatted information from the tagged document. This is also where model testing is completed.

HTML Scraper & Training Documentation

The HTML scraper is the generator for our documentation used in the training portion of the pipeline. Unlike its counterpart, the new software document, HTML scraping allows the pipeline to gain a large number of documents from a single file of URLs. All scraped HTML is sent forward through the pipeline into the plaintext conversion stage.

New Software Document

Unlike its counterpart, the HTML scraper, a new software document is sent through the pipeline (generally) one at a time. In order to tag a document, it needs to be sent through the pipeline manually, by injecting an HTML file into the plaintext converter.

HTML Parse to Plaintext

This component is the first shared component in the pipeline, and is responsible for converting HTML and XML documents (especially HTML) into an easily parsable plaintext format. Using a list of code indicators, this component removes HTML tags which are specified by the filter, removing useless data. It then converts applicable tags into <code> tags, which allows future components to understand what is code and what is English. This is vital for the rest of the pipeline, as it is extremely important that code taggers aren't used for tagging english, and vice versa.

Tokenizer & Sentence Split

The tokenization component is responsible for splitting the large amount of plaintext into a strongly structured document composed of sentences and tokens. At first this appears to be a straightforward task, but a large amount of coordination must be done with the existing Stanford NLP tokenizer to maintain consistency. In addition, understanding context in regards to code and English blocks was vital to the construction of a consistent, reliable tokenizer.

Manual & Auto Tagging

The manual and auto-tagging portion of the pipeline is strictly used when training a new model. During this portion, auto tagging software is run on the tokenized documents, creating a JSON file with tagged english components and untagged code components. This cuts down the work of manual tagging greatly. Any non-autotagged components are then saved in order to be tagged using the manual tagger.

The manual tagger is responsible for creating a fast, straightforward environment for quickly applying tags to untagged code elements. For our project, this was done by our group, and was the slowest part of development. Unlike large companies, we could only reasonably manually tag 100 articles, leaving us with a small amount of data for training later on. In addition to the manual tagger, other tools were created for ease of manual tagging, like the patcher, which tagged all tokens of a certain type with a certain tag.

Train Model

The most important stage in the pipeline is the actual training of the model, which is actually straightforward. As our tagged data resides in a JSON format, it needs to be converted into an NLP format which the Stanford NLP pipeline is capable of processing. It then is fed into the training pipeline provided by Stanford, and it generates a file containing the model information usable by the Stanford NLP tagger.

Software Doc Model & Tagged Documentation

The final stage of the pipeline is the output and tagging stage where the actual pipeline model is used to evaluate the software document passed through the pipeline. When in training mode, this portion of the pipeline is used for grading and post-training evaluation. While in tagging mode, this portion of the pipeline is used to apply the trained model to the passed in software document. The output of this stage can be in NLP-XML format or in JSON format, and converters exist for both depending on the needs of the consumer.

Security Concerns and Countermeasures

Currently, cybersecurity concerns and countermeasures are a non-factor for our project. Our project is an offline, parts-of-speech tagging tool with no sensitive data or structures to hide. There are also no consequences of security breaches that are not already available or performable. The only part of our project that could become compromised, or have an impact if it is compromised, is the Stanford NLP at its core. However, the Stanford NLP is already publicly available and as such any security risks to breach it, if there are any and have any significant consequences, have no use of being negated.

Development Process

Design References

In figure 2 below, you can see the use case diagram that has motivated our design decisions. It's important to realize the two component nature of our project. The model itself will be usable by any member of the general public that knows basic coding. However, there will also be a set of tools made specifically for training new versions of the model that only experienced developers will be capable of using.

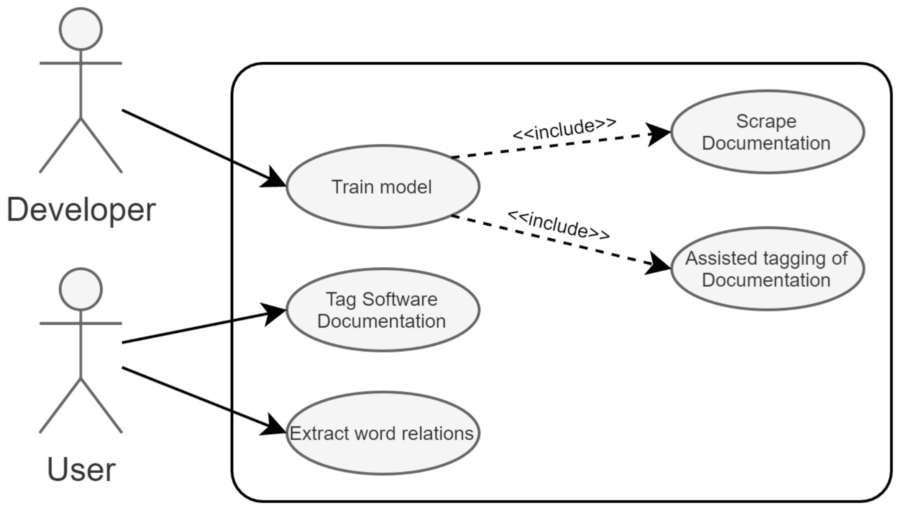


Figure 2: A use case diagram for this project

Completion of the use case for general users was relatively trivial. This is because the Stanford NLP libraries contain APIs that already complete them. For more information on the architecture of the POS models consult the documentation available on the Stanford NLP website.

The bulk of development effort went into the use cases for developers. For this we implemented a pipeline to transform software documentation into a tagged corpus of examples. We then used these to train a CRF model. In figure 3 below, you can see the planned architecture for this pipeline.

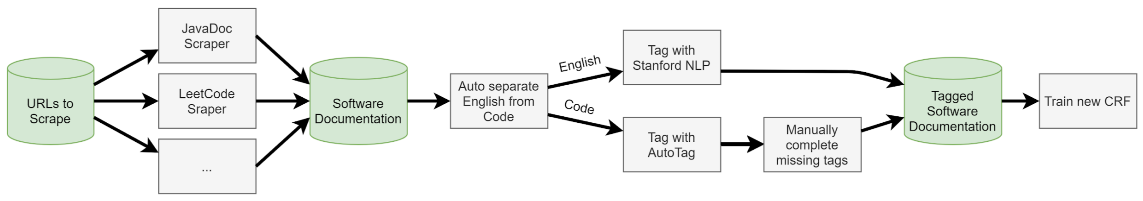


Figure 3: The design pipeline for training the model

Major Development Milestones

Below is the development process used throughout both semesters of work put into this project. Generally, this format was followed closely, with lots of development taking place in the pre-iterative phases.

Milestone 1: Determine Types of Software Documentation

- Find multiple examples of software documentation (English, and code)
- Categorize software documentation based on similar qualities
- Generate at least 2 categories of software documentation

Milestone 2: Collect Software Documentation

- Find at multiple examples of software documentation in each category
- Design a Web Scraper to take software documentation and output it in an easily readable format
- Output the raw HTML data
- Output the documentation segments

Milestone 3: Clean and Pre-process Software Documentation

- Write a parser to generate “blocked” code based on whether the data is in monospace font or not, also based around HTML tags.
- “Blocked” code distinguishes between code snippets and English text
- Take scraped documentation and generate treebanks based on the data.

Milestone 4: Complete initial analysis of Software Documentation

- Pass the raw software documentation through the existing English NLP pipeline
- Analyze the data to find common errors which need to be fixed
- Find common errors in English documentation
- Analyze behavior of NLP on code snippets

Milestone 5: Development and Testing of System Pipeline

- Establish a stable codebase for each component of the pipeline
- Complete component based testing for each component
- Iterate on a component-based basis until all components are functional

Milestone 6: Strengthen and Deepen Tags Related to Software

- Analyze common errors to determine additional tags which may benefit POS analysis
- Come up with use cases for the new tags, along with several examples of their usage
- Develop a complete list of tags related explicitly to software tagging needs based on code.

Milestone 7: Manually Tag Documentation with New Tags

- Test out new tags using the corpus of software documentation
- Iterate on tags as necessary until a good model is found
- Manually tag all software documentation with new tags
- Initially tag documentation with Stanford NLP
- Correct Stanford NLP errors using manual tagging

Milestone 8: Train a new Conditional Random Fields Model

- Put together corpus of training data
- Use Treebank format to convert raw documentation data
- Use Stanford's CoreNLP library to train a new Conditional Random Fields (CRF) Model
- This stage will be iterative as necessary

Milestone 9: Acceptance Testing

- Perform acceptance and accuracy testing on a separate set of data from the training data
- Does the model conform to the accuracy standards set out earlier? If not, revert to milestone 5 and iterate.

Milestone 10: Reporting

- Compile data on accuracy of tested data and prepare reporting data.

Technological Considerations

Currently, in our project there are a couple areas where technology considerations are significant due to the availability of choice. These two areas are: the language used to implement the PoS tagger and the resources used to train the PoS tagger.

Language Used:

Java - Java is the first of two languages available to us which have been used by others to implement the Stanford NLP. The Java version is implemented using the Stanford CoreNLP and is a low level interface to the trained model. It was discovered during our initial testing that the Java version has a slightly higher accuracy compared to the Python version, but if run incorrectly could take up a very large amount of system resources.

Strengths: Typically takes a lower amount of resources than Python. Higher accuracy tagging.

Weaknesses: Could take a large amount of resources to run.

Trade-offs: Low-level interface increases complexity but also increases control.

Python - Python is the second of two languages available to us which have been used by others to implement the Stanford NLP. The Python version is implemented using Stanza and is a high level interface to the trained model. The Python version runs a local Java server and interfaces with that instead of running the PoS tagger natively in Python. Due to this the Python version uses more resources and does not perform as well as the Java version.

Strengths: Easy to use.

Weaknesses: Takes a larger amount of resources than the Java version to run. Not the native language of the Stanford NLP.

Trade-offs: Trades complexity for ease of use compared to Java.

Training Method:

Developer's Machines - Training the model(s) on the developers' machines utilizes resources that the developers already possess. The developers do not possess the most powerful hardware to accelerate the training times. Due to this, as the model's training data grows in size it quickly becomes apparent that more powerful hardware would be highly beneficial.

Strengths: Utilize resources already owned.

Weaknesses: Could take a significant amount of time to train. Training the model with a growing dataset could potentially easily surpass the technical capabilities of the developer's machines.

Trade-offs: Utilizing the developers' machines would trade training time and performance for ease of use/training and accessibility.

HPC GPU Racks and Clusters - Utilizing GPU racks and clusters specifically designed to provide high performance and significant computing power (particularly for training models) would be a great boon to the project. Utilizing these resources would offload training responsibilities from our own machines to university machines, which are much more powerful and could complete training on a much larger set of data in a smaller amount of time.

Strengths: Faster training time on larger data sets.

Weaknesses: Working with machines that are not our own.

Trade-offs: Utilizing GPU racks and clusters would trade off accessibility of training, although marginally, for more performant training and potentially runs of the program.

Solutions:

Given the above analyses of the technologies to choose from, we have chosen to primarily focus on the Java version of the Stanford NLP (CoreNLP) and utilize GPU racks and clusters to train our models. However, we ended up using our own machines to train the models often, as our dataset was small enough that it became more convenient to use our own machines.

Testing

The testing strategy for this project utilized many different phases based on what the focus of work was at the time of testing. The most testing emphasis was placed on the components of the pipeline, as this project needed to be future-project compatible, and having a working pipeline was vital to this goal.

Integration Testing

The most important aspect of the testing regime of this project was integration testing throughout the pipeline components. As the pipeline was vital not just for our group, but any future groups on this project, integration testing was required. We made several design decisions to ensure integration testing would be easy, most notably the usage of a pipelined design with transparent connections. Each component's inputs and outputs were direct and single-streamed, so any output could be grabbed and read at any time.

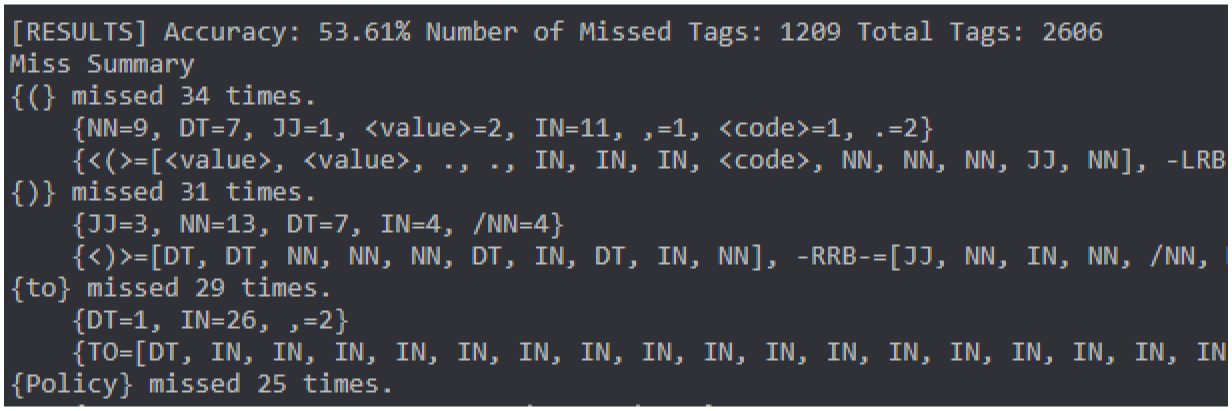
To do integration testing on a component, we moved from front to back through the pipeline. For each component, we supplied the expected input for the component, and manually verified the results of the components operation. While manual testing is typically inappropriate, the dynamic nature of our data resulted in a system in which writing a generic testing platform would have been more work than the components themselves took.

Integration testing was by far the most taxing portion of our testing regime. Each component was developed in a somewhat staggered fashion in order to allow the group to split work between programming, tagging, and theoretical model development in a fair way. Each component of the pipeline was iterated on in an agile fashion, with testing being common and the driving factor behind iteration. Once integration testing was complete on each component, the pipeline worked as expected, as the transparent inputs and outputs allowed for extreme flexibility in testing.

Model Testing

Once integration testing was completed on the pipeline it became possible to actually generate and test models. Because of the nature of this portion of the project, iterations were important, but less viable than previously. Lots of debugging and investigation had to go into the building of a model, and analytics tools were created as needed to make debugging results easier.

At the end, model testing was accomplished using a testing module which compared two JSON representations of tagged data, one manually tagged and one tagged by the model being tested. The testing module then compiled results into incorrectly tagged and correctly tagged lists, which were then categorized using a series of dictionaries to make finding causes of incorrect tags easier. An example of our results object is displayed below in figure 4.



```
[RESULTS] Accuracy: 53.61% Number of Missed Tags: 1209 Total Tags: 2606
Miss Summary
{(} missed 34 times.
    {NN=9, DT=7, JJ=1, <value>=2, IN=11, ,=1, <code>=1, .=2}
    {<(>=[<value>, <value>, ., ., IN, IN, IN, <code>, NN, NN, NN, JJ, NN], -LRB
{)} missed 31 times.
    {JJ=3, NN=13, DT=7, IN=4, /NN=4}
    {<)>=[DT, DT, NN, NN, NN, DT, IN, DT, IN, NN], -RRB-=[JJ, NN, IN, NN, /NN,
{to} missed 29 times.
    {DT=1, IN=26, ,=2}
    {TO=[DT, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN, IN
{Policy} missed 25 times.
```

Figure 4: A snippet of results for the final iteration of the model.

Final Results

The final results for our model's efficacy is a 53.61% accuracy tagging rate. The final test for this model was done using 2606 separate tags, with 1209 tags being missed. In terms of files, this is equivalent to 80 files being used for training and 20 files being used for testing.
While initially an upsetting result, upon more investigation we came to realize that a 53% accuracy rate is not bad for the limited dataset we used.

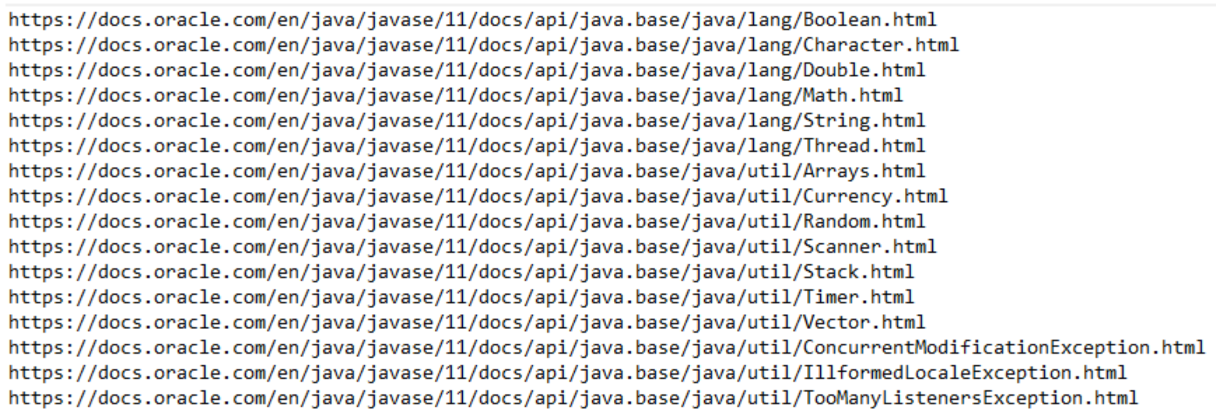
Our group had limited time, and the most time intensive portion of the project was the manual tagging of data for training. Because of the time constraint, only 100 files were tagged in total. This is an extremely small dataset compared to the Stanford NLP dataset, which an entire team of linguists was responsible for putting together. Our model performs far better than random selection of tags, so it is our belief that a future group could simply compile more training data to improve the efficacy of the model in the future.

Appendix 1

Operational Manual

The operation of the pipeline is a multi-step process with the usage of different languages and environments. This short manual will hopefully show you how to operate each component and how to analyze the inputs and outputs of the pipeline components.

1. **Generate a list of URLs you would like to process.**
   The first major component of running the pipeline is organizing and gathering the data you wish to use. While we used the web for finding data, any HTML or SPA page accessible by URL will work here. Simply compile all of these URLs into a text document, delimited by newlines. An example of this file is shown below.



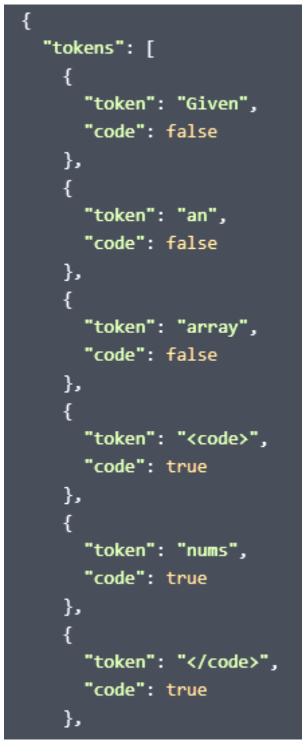
```
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Boolean.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Character.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Double.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Currency.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Stack.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Timer.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Vector.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ConcurrentModificationException.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/IllformedLocaleException.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/TooManyListenersException.html
```

Figure 5: A portion of the URL file for our javadoc scraping

2. **Run the UniversalHTMLParser on the URLs.**
   The UniversalHTML parser is a python application which parses and gathers data from the URLs from the file, processes the HTML into a plaintext format, and then tokenizes and converts the data into a JSON format. The UniversalHTML parser is truly a large, multi-purpose component.

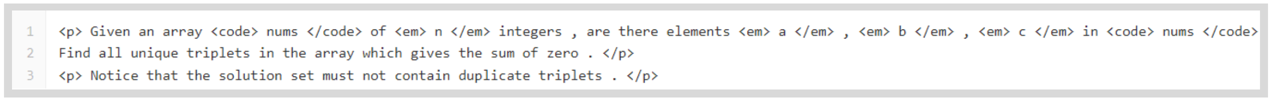
   To run the UniversalHTML parser, you need to install the jnius package in python which allows you to run Java code from a python environment. In addition, you will need to set a couple of options depending on your usage of the HTML parser.

   First, you need to change the rules file (also located in the same directory) to use the conversion rules you wish to use when converting HTML to plaintext. The default rules file is fairly general-purpose, and works well for Javadoc and Leetcode articles.



```
{
  "tokens": [
    {
      "token": "Given",
      "code": false
    },
    {
      "token": "an",
      "code": false
    },
    {
      "token": "array",
      "code": false
    },
    {
      "token": "<code>",
      "code": true
    },
    {
      "token": "nums",
      "code": true
    },
    {
      "token": "</code>",
      "code": true
    },
```

Figure 6(left) : The final output of the UniversalHTMLParser

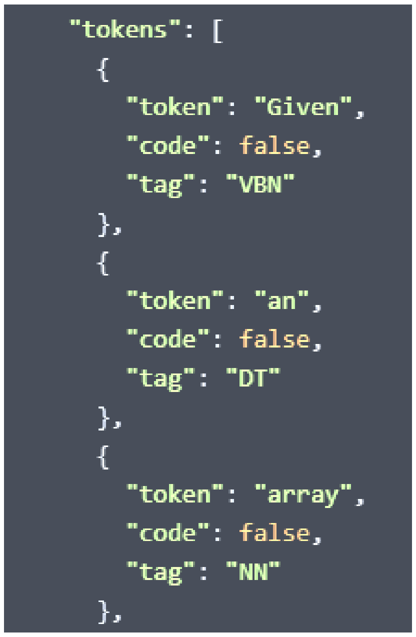


```
<p> Given an array <code> nums </code> of <em> n </em> integers , are there elements <em> a </em> , <em> b </em> , <em> c </em> in <code> nums </code>
Find all unique triplets in the array which gives the sum of zero . </p>
<p> Notice that the solution set must not contain duplicate triplets . </p>
```

Figure 7(above): The intermediate tokenized data from the webpage

Next, you need to direct the HTML parser towards the output and tokenizer directories. These are hardcoded in the main.py file in the main directory. Output will write the tokenized JSON files to that directory. The python file must also have access to the compiled tokenizer class file in order to tokenize the data automatically.

Simply run this file using python to generate a set of JSON formatted files which can then be passed through the rest of the pipeline.



```
"tokens": [
  {
    "token": "Given",
    "code": false,
    "tag": "VBN"
  },
  {
    "token": "an",
    "code": false,
    "tag": "DT"
  },
  {
    "token": "array",
    "code": false,
    "tag": "NN"
  },
```

Figure 8: A portion of the autotagged output

3. **Autotag English and Simple code data.**
   Now that we have JSON files, we need to automatically tag as much data as possible using the stanford NLP english model and our own simple tagging rules. The AutoTagging directory contains the java code necessary to run the autotagger, along with a markdown file with excellent documentation on how to run the code. Once autotagging is complete, the JSON files will be populated with partially tagged data. It is now either the role of the model (if tagging) or the developer (if training) to fill in the rest of the missing tags.

4. **Manually Tag Data if you are training a model.**
   If you are attempting to create your own model, you will need to manually tag your dataset using the application in the ManualTagger directory. The manual tagger is a java gui application capable of streamlining the manual tagging process. Once again, there is an excellent markdown file in the root directory which will give you step by step instructions on how to set up and run the manual tagger.

   Once the tagger is running, you simply need to direct it towards a directory of JSON files with missing tags. You will then be prompted for each token to manually give it a tag. Other utilities also exist within the application, like listing the number of tags remaining to tag, automatically tagging any tokens which match a string with a tag, and stateful tagging (being able to stop and come back without losing progress).

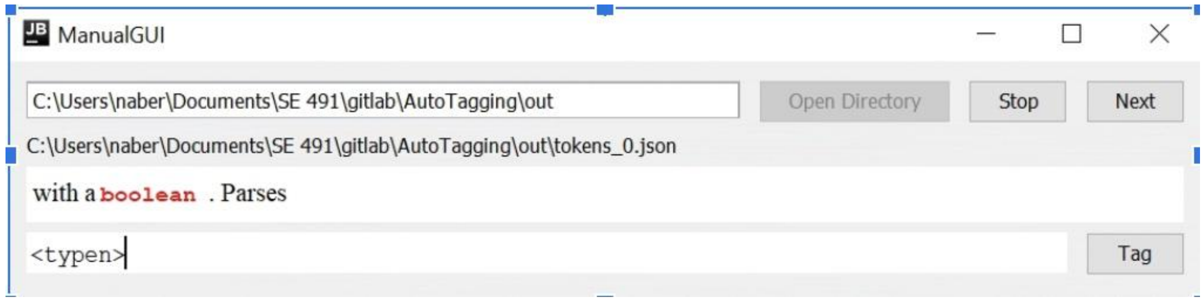
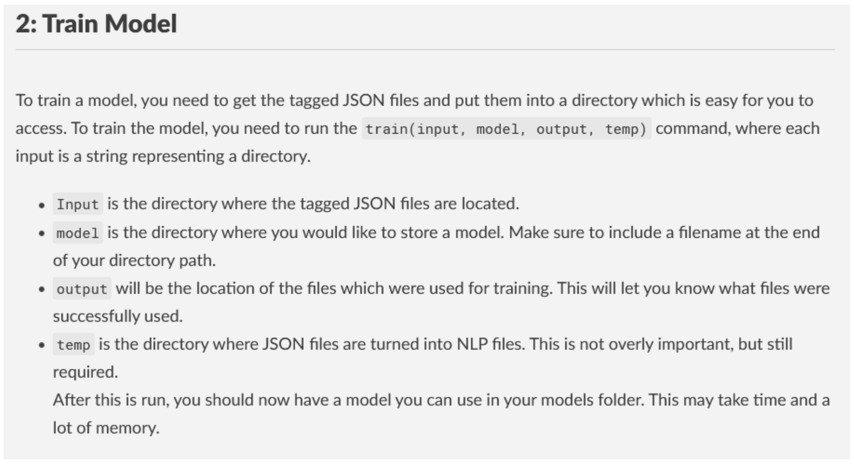

ManualGUI
C:\Users\naber\Documents\SE 491\gitlab\AutoTagging\out
Open Directory
Stop
Next
C:\Users\naber\Documents\SE 491\gitlab\AutoTagging\out\tokens_0.json
with a boolean . Parses
<typen>
Tag

Figure 9: The GUI of the manual tagger. For this example, a "boolean" token is being tagged as a "<typen>" (typename)

5. **Train the NLP model using tagged data.**
   To train the model you will need to navigate to the NLPModel directory of the project. As with other modules so far, there is a markdown file in this directory which will give you much more detailed information about how to run and setup this portion of the pipeline.

   To train a model, you will need to call the "train" function from the "TrainModel" class. This function requires four arguments. The first and third are the input and output directories. THe input directory should contain the tagged JSON files needed for training, and the output directory will contain the JSON files that were used to contribute to the model being trained. The second argument is the model filepath, so that training can be done additively. Finally, the function requires a temporary directory for any temporary files generated during training. Once this function is run, it will create a file in the model file path which contains the newly trained model.



2: Train Model

To train a model, you need to get the tagged JSON files and put them into a directory which is easy for you to access. To train the model, you need to run the `train(input, model, output, temp)` command, where each input is a string representing a directory.

- `Input` is the directory where the tagged JSON files are located.
- `model` is the directory where you would like to store a model. Make sure to include a filename at the end of your directory path.
- `output` will be the location of the files which were used for training. This will let you know what files were successfully used.
- `temp` is the directory where JSON files are turned into NLP files. This is not overly important, but still required.
  After this is run, you should now have a model you can use in your models folder. This may take time and a lot of memory.

Figure 10: A sample from one of many markdown files throughout the project.

6. **Run the NLP model on the JSON files you wish to tag.**
   If you are interested in tagging a document, you need to use the application in the NLPModel directory. Once again, if you are interested in a more in-depth guide, there is an overview markdown document located in the root of this directory which goes over how to operate the applications in this directory.

   To run the NLP model, you simply need to use the provided stanford NLP interface, while specifying our model as the primary model for usage. To make this process even easier, a "TagDocs" function has been provided in the "Tagging" directory of the project. The output of the tagger will be a JSON file which has been completely tagged. If you would rather have the file in NLP format (XML tags), you can simply remove the conversion.

7. **Test the NLP tagged output.**
   The final portion of the pipeline is only useful if you are evaluating a model's tagging efficacy. Once again, in the NLPModel directory a markdown file is available with better instructions on how to run the testing module.

   The testing module, located in "Testing", essentially does an element-by-element comparison between two parallel JSON files, one "correct" and one "test subject". Any differences in comparison are cumulatively documented and built up for analysis in the "TestResults" class, which is returned from the "Test" function call. This "TestResults" object contains information about the number of missed tags, the types of misses (an ordered list of missed tags based on their number of misses), the actual misses (what tag was used incorrectly), and the overall accuracy of the system. This file should help with any debugging which needs to be done to improve system accuracy.

Appendix 2

Evolution of the Design and Other Versions

Semester 1:Project Design and Main Focus
The first semester was mostly focused on understanding the Standard POS Tagger, collecting relevant software documentation and coming up with our own cleaner version of the documentation for the project. We ran the current standard Stanford NLP tagger and analyzed it with the purpose of understanding how it works. Analyzing the data it produced helped us find common errors in the English documentation and examine the behaviour of NLP on code snippets. This phase saw the system as a black box, with more or less a single interface, however our project developed into a stronger, more pipelined model as the second semester progressed.

We were able to come up with a the following

- New Tag List with several examples of their usage
- a modified pipeline
- a customizable web scraper
- a customizable HTML parser

Semester 2:Project Design and Main Focus
The second semester was mostly focused on the implementation of Methods of the New POSTagger. This includes completing the data gathering pipeline that is the tokenizer and the interleaving of the new components within the pipeline. The remaining months of this semester were focused on evaluating the new POS Tagger and identifying any possible flaws that might be present. Utilizing what was gathered when identifying the flaws to be able to refine the new model. This involved a large amount of iteration and integration testing, which refined and strengthened our pipeline model from the first semester.

Appendix 3

Other Relevant Information

Finalized Set of Software Documentation PoS Tags

This set is our own new tags and excludes the pre-existing standard set of English tags from the Penn Treebank. The Penn Treebank tags are still used as our English tags for this project.

Tag	Description	Example
<am>	Access Modifier	*public static void main()*

<?st>	Conditional Statement	*if (true) { }*
		int i = true ? 4 : 2;

<;>	End of statement	*String hello = “world”;*
<type>	Language type	*class Color* *Object*
<typen>	Type name	*String* hello = “world” class *Color*
<{>	Open block	*if (true) { }*
<}>	Close block	*if (true) { }*
<(>	Open parenthesis (in code)	*if (true) { }*
<)>	Close parenthesis (in code)	*if (true) { }*
<[>	Open bracket	*new String[] {“hello”, “world”};*
<]>	Close bracket	*new String[] {“hello”, “world”};*
<,>	Comma (in code)	*new String[] {“hello”, “world};*
<var>	A variable in code	*String hello = “world”;*
<func>	A function/method	*public static void main()*
<=>	Gets	*String hello = “world”;*
<par>	Parameter of function/method	*public test(String hello)* *function(a)*
<return>	Return statement	*return hello;*

<loop>	Iterative loop	_while (true) { }_
<value>	Value	_int i = 42;_
<”>	Double quotes	_String hello = “world”;_
<’>	Single quotes	_char c = ‘a’;_
<inherit>	Inheritance	_public class foo extends bar_ _public class foo implements bar_
<op_mat>	Mathematical Operator	_int i = ((2 + 2) - ((4 * 3) / 6) % 3);_ _expr++_ _!expr_ _a&b_
<op_rel>	Relational Operator	_if(a ≤ b) { }_
<op_log>	Logical Operator	_if(a || (b && c)) { }_
<op_gets>	Assignment Operator	_int a += 7;_
<.>	Dot (in code)	_foo.bar();_
<error>	Error or exception handling keywords	_try {_ _..._ _} catch (Exception ex) {_ _..._ _}_
<generic_type>	< or > around a generic	Hashmap < String, LongAdder >
<new>	instantiation	_new_ AbstractMap
<comment>	Start or end of comment	_// this is a comment_
<cf>	Control flow	_break; or continue;_