A Part of Speech Tagger for Software Documentation

Design Document

Team 35
Ali Jannesari, Hung Phan
Joseph Naberhaus
James Taylor
Austin Boling
Ekene Okeke
Ahmad Alramahi
Ethan Ruchotzke
sd21-35@iastate.edu
https://sdmay21-35.sd.ece.iastate.edu/

Revised: 11/15/2020

Executive Summary

Development Standards & Practices Used

ISO-IEC: ISO 12207 - Software Life Cycle: Used to model our project plan.

ISO-IEC: ISO 9001 - Quality Management: Used in conjunction with Agile Software development to ensure our product meets the client's specifications.

ISO-IEC: 15504 - SPICE: Used to asses the software development processes we have and will setup

Summary of Requirements

Study elements of standard POS tagger

Study new elements of software documentation

Implementation of a new POS tagger

Evaluation of the new POS tagger

Applicable Courses from Iowa State University Curriculum

Com S 227 Object Oriented Programming

Com S 228 Introduction to Data Structures

Com S 311 Introduction to the Design and Analysis of Algorithms

Com S 472 Principles of Artificial Intelligence

Math 207 Linear Algebra

Stat 330 Probability and Statistics for Computer Science

New Skills/Knowledge acquired that was not taught in courses

Python

Web scraping

General NLP Theory (for most in group)

Table of Contents

1 Introduction 4

1.1 Acknowledgement 4

1.2 Problem and Project Statement 4

1.3 Operational Environment 4

1.4 Requirements 4

1.5 Intended Users and Uses 4

1.6 Assumptions and Limitations 5

1.7 Expected End Product and Deliverables 5

2 Project Plan 5

2.1 Task Decomposition 5

2.2 Risks And Risk Management/Mitigation 6

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria 6

2.4 Project Timeline/Schedule 6

2.5 Project Tracking Procedures 6

2.6 Personnel Effort Requirements 7

2.7 Other Resource Requirements 7

2.8 Financial Requirements 7

3 Design 7

3.1 Previous Work And Literature 7

3.2 Design Thinking 7

3.3 Proposed Design 7

3.4 Technology Considerations 8

3.5 Design Analysis 8

3.6 Development Process 8

3.7 Design Plan 8

4 Testing 9

4.1 Unit Testing 9

4.2 Interface Testing 9

4.3 Acceptance Testing 9

4.4 Results 9

5 Implementation 10

6 Closing Material 10
6.1 Conclusion 10
6.2 References 10
6.3 Appendices 10

List of figures/tables/symbols/definitions

Figure 2.4.1 Project timeline
Table 2.6.1 Personal effort requirements
Figure 3.7.1 Use case diagram
Figure 3.7.2 Model training pipeline
Table 6.3.3 Tag iteration 1
Table 6.3.3 Tag iteration 2
Table 6.3.3 Tag iteration 3

1 Introduction

1.1 Acknowledgement

Our group would like to acknowledge our TA Hung Phan and our professor Ali Jannesari for giving us this project, as well as pointing us in the right direction to begin work on it. We would also like to acknowledge the Stanford NLP Group, who worked on the NLP, making this project possible.

1.2 Problem and Project Statement

As it stands right now, there isn't a tagger software that can analyze software documentation well. While the base NLP can do this to some extent, it isn't up to the task, as it doesn't fully understand the context of software documentation.

Our solution is to use the learning tools provided by the Stanford NLP group to augment their existing NLP model to meet the requirements that we have laid out. This includes creating a plethora of training data to give the NLP examples of correctly tagged software documentation. Further details of how we will augment the NLP will be found below.

1.3 Operational Environment

According to the Stanford Natural Language Processor Group, running a trained model requires at least 100 MB of memory, but in some cases can require upwards of 7 GB of memory ("The Stanford NLP", 2020). Training a model requires at least 1 GB of memory, but in many situations will require significantly more. Based on our current experiences, having at least 8 GB of memory and a reasonably modern processor is recommended ("The Stanford NLP", 2020).

1.4 Requirements

Corpus of tagged software documentation

- Collect a variety of forms (> 4 types) of software documentation in large quantities (> 25 of each)
- Tag the data in the same schema as the treebank used by the Stanford NLP
- Ensure the data is usable for future works

Augmented Stanford NLP model for software documentation

- Improve accuracy of base Stanford NLP model when run against english within software documentation
- Expand tag set of base Stanford NLP to cover common elements of software documentation
- Build Java and Python APIs for the new model
- Be capable of running on a mid-range machine with 8 GB of Memory and a mid range processor

Paper covering our work

- Produce accuracy comparisons between our model and the base Stanford NLP (both with normal english text, and software documentation)
- Follow writing standards used by Computer Science academia

1.5 Intended Users and Uses

The Intended users of this product are for the people who want a software that understands Natural Language (Stanford NLP Speech Tagger).

Part-Of-Speech Tagger (POS Tagger) is to read text that is given as an input in a Natural language and assign parts of the speech to a word (for an example, noun, verb,and adjective). Some of the uses of this software are as follows:

- Tag POS of software documentation
    - Important in building lemmatizers which are used to reduce word to its root form
- Extracting the relationship between words

We are also creating a suite of tools to assist in training. These will be usable by us and other developers to train new iterations of the model. The use cases of this software will be the following:

- Model training pipeline
- Tool for web-scraping software documentation
- Tool to help tag documentation. Automating as much as possible.

1.6 Assumptions and Limitations

Assumptions:

The inputs to this NLP will be in plain text

It will only be used for English software documentation

The architecture of the existing Stanford NLP is a good base to build our tagger off of

The model will be run locally on a user's personal computers

Limitations:

The model must be runnable on an ordinary mid-range computer (to match performance of existing models)

We will not be augmenting other capabilities of the Stanford NLP model such as the dependency grapher, NER, ect... (per user's requirements)

Users must be familiar with Java or Python to make full use of our model (per user's requirements)

This model will be optimized specifically for software documentation, and not for general NLP (per user's requirements and to improve technical feasibility)

No financial expenditures are expected for this project

1.7 Expected End Product and Deliverables

Corpus of tagged software documentation (November 2020)

Unlike the other two end products, this one covers two of the deliverables given to us by our client:

- Study elements of standard POS tagger (October 2020)
- Study new elements of software documentation-(November 2020)

A collection of software documentation that has been scraped from the internet and manually tagged. All types of documentation deemed relevant and interesting for this project will be included. Additionally, the tagging scheme of the corpus complies with the treebank schema used by the Stanford NLP model.

Implementation of a new POS tagger (February 2021)

A NLP processor tuned for software documentation. It has an expanded tag set that covers the common elements of most programming languages (with some bias towards Java). Internally, it works similarly to the existing Stanford NLP model, but with slight modifications. It's performance will be evaluated and reported along with the release.

Evaluation of the new POS tagger (May 2021)

An academic paper covering the methodology, and results achieved in our project. The paper discusses the basic architecture of the Stanford NLP and what modifications we have made to it. It will also include information about how we gathered a corpus of data to train the data. Finally, it will conclude with the performance and accuracy of our model.

2 Project Plan

2.1 Task Decomposition

The tasks outlined below are the general process the team will be taking in order to complete an additional module to the Stanford NLP processor. The team is following an Agile methodology, and the milestones below are not necessarily linear. In the event of a dependency, it will always be backwards, however a milestone's performance will determine the next milestone undertaken. This is evident in the schedule portion of the design document and the descriptions of tasks.

Major Milestones

1. Documentation Collection and Testing
2. Decision on Methods for NLP Extension
3. Implementation of Methods
4. Evaluation and Acceptance Testing
5. Reporting

Milestone 0: Setup and Testing [1]

- ❏ Complete Installations of Java CoreNLP and Python's Stanza
- ❏ Compile a list of test phrases for initial testing (English)
- ❏ Perform pipeline testing on both Java and Python NLP
    - ❏ Generate comparable outputs in a CSV format
    - ❏ Compare outputs of the pipeline for both versions
    - ❏ Determine next steps from the amount of consistency between platforms
- ❏ Make a decision on whether to use Python or Java's processor
- ❏ Generate documentation related to the extension and understanding of NLP
    - ❏ Make demo documentation for the group's usage

Milestone 1: Determine Types of Software Documentation [1]

- ❏ Find multiple examples of software documentation (English, and code)
- ❏ Categorize software documentation based on similar qualities
- ❏ Generate at least 3 categories of software documentation

Milestone 2: Collect Software Documentation [1]

- ❏ Find at least 5 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 [2]

- ❏ Write a parser to generate "blocked" code based on whether the data is in monospace font or not
    - ❏ "Blocked" code distinguishes between code snippets and English text
- ❏ Take scraped documentation and generate treebanks based on the data.
- ❏ Use Word2Vec to generate training data based on treebanks

Milestone 4: Complete initial analysis of Software Documentation [2]

- ❏ 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: Strengthen and Deepen Tags Related to Software [3]

- ❏ 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

Milestone 6: Manually Tag Documentation with New Tags [3]

- ❏ 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 7: Train a new Markov Model [3]

- ❏ Put together corpus of training data
    - ❏ Use Treebank format to convert raw documentation data
    - ❏ Use Word2Vec to generate training data
- ❏ Use Stanford's CoreNLP library to train a new Markov Model
    - ❏ The new markov model will be subject to acceptance testing
    - ❏ This stage will be iterative as necessary

Milestone 8: Acceptance Testing [4]

- ❏ 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 9: Reporting [5]

- ❏ Compile data on accuracy of tested data.
- ❏ Participate in the creation of a research paper related to the project.

2.2 Risks And Risk Management/Mitigation

Because the team is undertaking the project with an Agile methodology, risk is a minor factor due to the fast iteration process. However, risk is still a factor. Below are some of the most important risk factors for the project, and our methods for mitigating the risk. In total, however, Agile development and rapid iteration will allow risk to be mitigated for this project.

Risk Factor 1: Inability to train a model which is acceptable [5-10%]

The inability to train a model which meets acceptance testing standards is the major risk factor associated with this project, however Agile development essentially mitigates this risk for the project. Development of an acceptable model is vital, but iterative. So, even if the model does not meet acceptable accuracy requirements on the first iteration, there will be plenty of sprints available to increase the accuracy.

Risk Factor 2: Inability to create a large corpus of tagged documentation [3%]

The most vital part of machine learning is the creation of a corpus of data. In this project's case, it is possible that the team will be unable to generate a large corpus of data. In the event this happens, the mitigation is simple - scrape for a broader base of documentation. While initially we are only requiring 3 types of documentation, increasing our number of documentation categories will allow more documentation to be available for training, meaning the corpus will be expanded.

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

At least 4 different types of software documentation will be identified

At least 25 instances of each type of software documentation identified will be collected.

At least 25 instances of each type of software documentation will be cleaned.

At least 25 instances of each type of software documentation will be run through the standard POS Tagger to see discrepancies

List of new tags relating to Software Documentation will be finalized.

At least 25 instances of each type of software documentation will have their tags modified manually to match expected

Modified MEMM will be trained with at least 25 instances of each type of manually tagged software documentation

Modified StanfordNLP Tagger will tag Software documentation with additional tags at 90% accuracy

2.4 Project Timeline/Schedule

Figure 2.4.1 below shows the predicted project timeline. Please note that the details of implementation are unknown, so it is just one long 2 month block. Additionally, names of members are not included because we do not yet know who will exactly be doing what.

It can be seen that we plan to be doing mostly information and data gathering over this first semester. This is the most essential part of the project, since we cannot produce a good model without first having good data.

During the next semester, we will start producing models and evaluating them. This will be an iterative process until we are satisfied with the results. Finally, we will release our finalized model, and produce a written report about it.

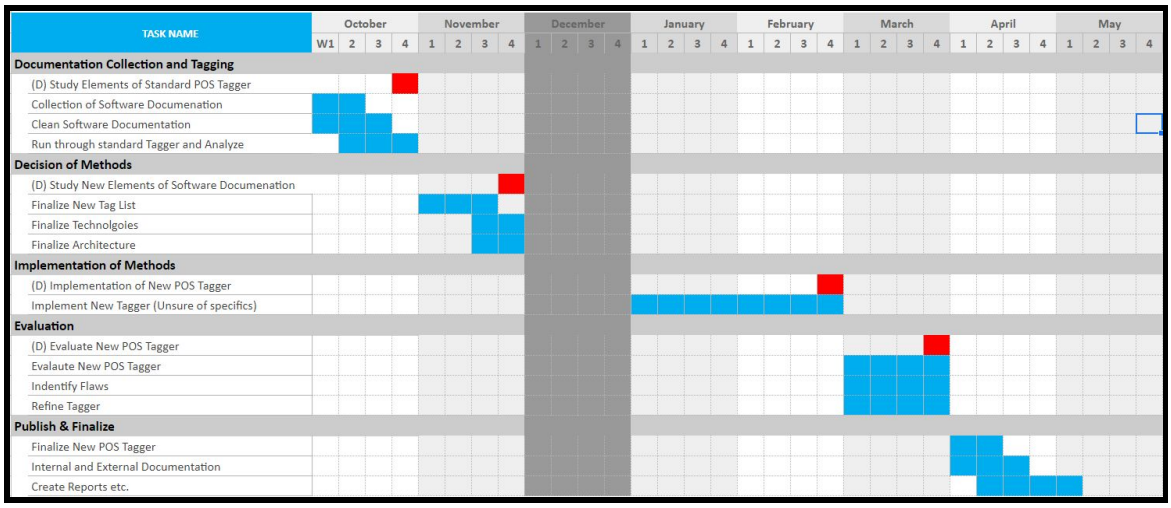

Figure 2.4.1 Project timeline (Deliverables shown with a red box)

2.5 Project Tracking Procedures

Tracking procedures/technologies:

Weekly Task Management (Sprint Boards) - **Trello**

Project Version Tracking - **git** repository hosted on **GitLab**

Daily Interaction, Sharing, Teamwork - **Discord**

2.6 Personnel Effort Requirements

Include a detailed estimate in the form of a table accompanied by a textual reference and explanation. This estimate shall be done on a task-by-task basis and should be the projected effort in total number of person-hours required to perform the task.

Task	Textual Reference	Time to complete	Explanation
Documentation Collection and Tagging	The task was distributed as follows: ● Study Elements…. ○ Java- Ahmad and Ekene ○ Python-Ethan and Joseph ● Collection of Software Documentation ○ James and Austin ● Clean Software Documentation ○ James and Austin	4 weeks	● Study Elements of Standard POS Tagger ● Collection of Software Documentation ● Clean Software Documentation ● Run through standard Tagger and Analyze
Decisions of Methods	TBD	4 weeks	● Study New Elements of Software Documentation ● Finalize New Tag List ● Finalize Technologies ● Finalize Architecture
Implementation of methods	TBD	8 weeks	● Implementation of New POS Tagger
Evaluation	TBD	4 weeks	● Evaluate New POS Tagger ● Identify Flaws ● Refine Tagger
Publish and Finalize	TBD	5 weeks	● Finalize New POS Tagger ● Internal and External

			Documentation • Create Reports

Table 2.6.1 Personal effort requirements

2.7 Other Resource Requirements

To extend the Stanford NLP for software documentation tagging, our team requires relatively few resources other than machines and data that our team already possesses or can easily acquire. These resources are broken down into three categories: physical, virtual, and data.

Physical Resources:

- ❏ Mid-range machines for development (modern laptops)
  - ❏ Capable of running a program which may take up to approximately 1 GB of main memory
  - ❏ Most modern processors are more than capable of the development task

Virtual Resources:

- ❏ Mid-to-high-range machines for model training (university computing resources)
  - ❏ Capable of running a program which may take up to approximately 7 GB or more of main memory
  - ❏ Most modern processors are more than capable of the training task, but a high-end processor would drastically reduce the time needed to train
  - ❏ The Stanford NLP is able to use the GPU for training, so a powerful GPU would also drastically reduce the time needed to train

Data:

- ❏ Software documentation of various types from various sources for training
  - ❏ Documentation includes but is not limited to:
    - ❏ Library and package documentation
    - ❏ Documents that mix both natural language, code, and/or pseudo code
    - ❏ Natural language that describes code
  - ❏ Must already be virtual
- ❏ NLP models, documentation, and research

2.8 Financial Requirements

There are no financial requirements necessary for this project.

3 Design

3.1 Previous Work And Literature

Previous Work: POS Tagging

POS Tagging can be approached in many ways, and has been throughout the last few decades.

A rule based tagger only considers the forms of words, and uses that to tag them. Rules could be things like:

- Consider lower/upper case, prefix/suffixes, and word shape
- Words with 'un-' are adjectives
- Capitalized words not at the beginning are Proper Nouns
- Shape of 'number-x' are adjectives

More often, features are used in conjunction with something else, often a graph based model.
The first iteration of these models, a Hidden Markov Model (HMM).
A 2000 improvement of the HMM created specifically for feature extraction in text is the maximum entropy markov model (MEMM). A 2010 creatition known as Conditional Random Fields was another method created to answer the problems that still existed in MEMM.

The industry leading POS Tagger, StanfordNLP[1] uses a bidirectional MEMM called a Cyclic Dependency Network [2]. Our work will introduce new tags and modify some tags from the current standard tagset, the Penn Tree Bank [3]. Then we will train a new model with a tagged documentation corpus.

Previous Work: POS Tagging for Software Documentation

While we have no intention of working from a base on anything done in this specific subfield of POS Tagging, it is worth mentioning the small (but various) amount of work that has been done in the field.

Some work has been done in this field when it comes to tagging StackOverflow questions correctly. [4] An interesting read, but the tags are too specific for what we wish to do. Another with mentioning is a POS tagger for program identifiers[5] like methods, classes, and fields that follow naming conventions. It is useful to think about these kinds of things, but again it is too specific for our purposes.

3.2 Design Thinking

Define Thinking

Define has been the major design thinking component for the semester so far. At the end of the day, the question we are asking the answer to is the most important part of our design. We centered our design around one major question - Is it possible to use existing POS tagging infrastructure to tag software and software documentation? Can we extend existing infrastructure to meet our needs? This has led us down several design routes. One of the most important was the format for training data for existing POS taggers. We are working to manually tag and train existing models, which fits directly within the question we posed during the "define" phase. Specifically, our team has used "define" thinking to our advantage when thinking about the questions of our project. Examples of our usage of define thinking include the design of our initial parsing and scraping units (what exactly are we looking for?) and the construction of our training pipeline (what tools exist for the extension of the CoreNLP?).

Ideate Thinking

So far, our ideate phase has played directly into our scheduling - we are performing agile development, so ideate is by far the most important phase of design. So far, ideate has allowed us to change our methods for gathering data quickly, as well as to make a choice between using the Java platform and the Python platform. Ideate directly correlates with iteration, which is a massive component of this project. Some other concrete aspects of design to come from the "ideate" phase were:

- Moving from manual tagging to automated tagging through a handmade software solution
- Combing javadoc for pure textual information to maintenance of specific tags in HTML
    - This was also extended into the general HTML parser we are working on

3.3 Proposed Design

*** Note: We are using Agile development to complete this task, so the design is not 100% static. Below is the expected design at the current stage of development ***

Our group is proposing the modification of the existing Stanford NLP Part of Speech tagger. This design will rely on several "modules" which all play into the eventual goal - training the existing NLP POS tagger to work with code and software documentation.

I. Code Analyzer

The code analysis tools built for the project will be capable of automatically tagging computer software (actual code) through a simple system of recognizing tokens. This system will take in a JSON file with all formatting information. Each language analyzed simply needs a JSON file for reference. This program will then output the tagged results for every token in the file. The limited testing for this analyzer shows great opportunity in pursuing this.

The code analysis tools will be capable of automatically tagging code - this will allow the team to spend less time manually tagging data, and will allow us to do the smaller task of verifying the results output by the analyzer. The output of this program will be capable of formatting for training data.

II. Web Scraper / HTML Parser

The other half of this project is the analysis of software documentation. Through the internet it is possible to gather a large set of data for training of the existing POS tagger. This sub-design relies on two components. The first component is a web scraper, which is capable of taking software documentation data from the internet. The scraper will be specialized and capable of grabbing data from popular sites with rich documentation, like leetcode, javadoc, and projectEuler.

The second half of this subdesign is the HTML parser. Through the web scraper we will be presented with a mess of HTML data, which is unparsable for existing infrastructure. A parser will need to be created in order to manipulate the data scraped and turn it into plaintext which can be fed into the training formatter. The HTML parser will need to be capable of parsing any selection of scraped HTML while also being capable of selecting tags which the user wishes to maintain (aka <code> tags).

III. Training Formatter

The training formatter subdesign will be necessary for converting the massive amounts of scraped and cleaned data into a format capable of being learned from by the existing POS infrastructure. This formatter will follow the training pipeline available to the team, which uses word2vec in order to vectorize the data contained within the training data. Once the formatter puts the data into a word2vec format, the existing NLP POS tagging structure will be capable of training itself, and building a new model capable of being tested.

IV. Trained NLP POS tagging model

This is the final, and most important subdesign of the project. Using the existing infrastructure available we are able to use the output of the training formatter to train a new POS tagging model based on our needs. The POS tagging platform used will be the Java Stanford CoreNLP library which is an industry standard when it comes to POS tagging.

The Java library for CoreNLP contains interfaces deep into its model, so it is easily capable of being trained by the word2vec data output by the training formatter. Once the trained model is created, it will be measured for accuracy, and iterated on based on results.

Depending on the amount of data output by the scraping and parsing tools, it may become necessary to train the model using existing high performance computing architectures available to the university's computing department.

Contingency Planning

In the event that our system does not meet requirements upon the testing of a "phase" during development, there needs to be a plan in place for next steps. For our project, this is simple. If results are not on par with recommended requirements, we simply need to continue training the model. By stopping often and testing the model, it is possible to see the "diminishing returns" of training and isolate causes of issues. However, this is an AGILE development process, so iteration is baked into the structure of development next semester. Iteration will be the most important part of this project, and testing is an important area to "end" iterations.

3.4 Technology 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 will be using our own machines in the meantime while our data set is small and while we work out the details of setting up the GPU racks and clusters.

3.5 Design Analysis

So far with our current designs, we have found success with tagging code, as well as software documentation. There is a lot of room for improvement, as we need to improve our models, as well as improve the methods we use to train our models, so that we may train our models faster. As it stands now, our software tagger has a success rate of 92% in preliminary trials.

3.6 Development Process

We are following the Agile design practice with leaning towards Scrum. We have found his method to work best, since our project requires a rather significant depth of knowledge. Using Agile allows us to build deeper understanding as needed. In addition, we don't fully know how our data gather and model training pipeline is going to work. Because of the nature of model training, it might take multiple iterations before we get satisfactory results. This makes Agile a good fit for our project.

3.7 Design Plan

In *figure 3.7.1* 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.

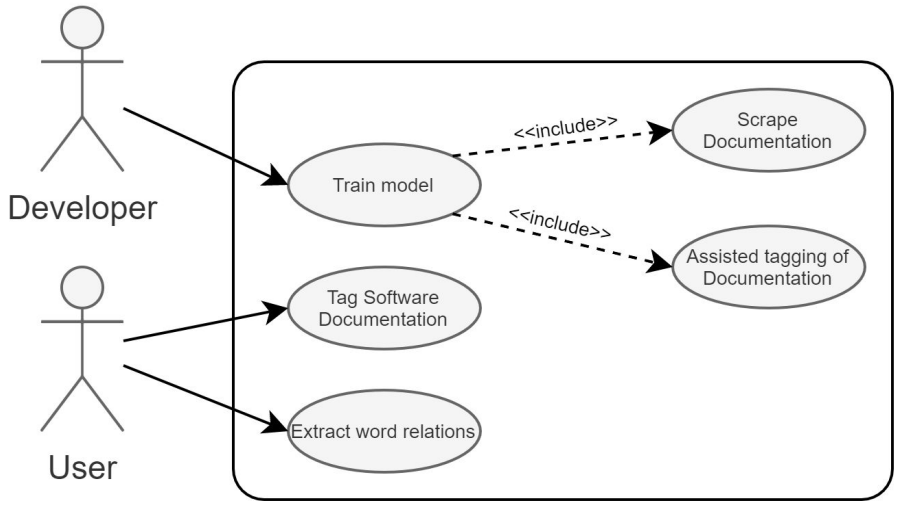

Figure 3.7.1 Use case diagram for the project

Completing the use case for general users will be trivially simple. 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 our development effort is going into the use cases for developers. For this we are going to need a pipeline to transform software documentation into a tagged corpus of examples. We can then use these to train a CRF model. In *figure 3.7.2* you can see the planned architecture for this pipeline.

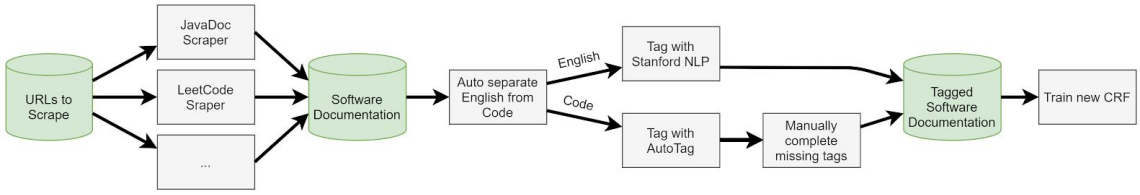

Figure 3.7.2 Model training pipeline

Each step of this pipeline represents a unique Java or Python script. We are following a Unix-like methodology of keeping a program to a single purpose. In the end, we will compose these together with either shell scripts or a master python script. The green databases represent files that will be produced on our computer. We will mostly export to plaintext, CSV, and JSON, since they are both versatile and lightweight.

4 Testing

4.1 Unit Testing

Software units that are being tested for in isolation:

In the context of our project, we need to test the consistency in both the implementations (that is Stanford NLP on python and Stanford NLP in java).

When testing the implementation, we found consistency errors in both of them (mostly punctuation).

We came to the conclusion that Python's and Java's inconsistencies are due to the difference in the interfaces.

4.2 Interface Testing

Within our project, we must "grade" our model as we go along, periodically checking its training by feeding it an unseen test, and grading the result by hand. This allows us to quantifiably test how effective our models are in their various tasks.

4.3 Acceptance Testing

In order to demonstrate that the design requirements are being met and the client is satisfied with the new product, we can involve the customer to be fully involved in the process in acceptance testing.
Examples of ways we can get the customer to involved in Acceptance process is by doing the following

- Sending questionnaires to the clients that have received the version of the program that needs to be tested and receiving the input from the clients
- Interviewing clients that have used the program that is in the process of testing

4.4 Results

As of 11/15/2020, there are no test results in relation to the final goal of the project - a trained NLP model. However, there have been several steps along the way which have encouraged and guided our team's progress so far.

1. Consistency Checking
    a. Consistency checking was the first major component of the project. The results of the consistency checking were excellent, and made the decision of whether to use Java or Python much easier. The python and java versions of CoreNLP performed slightly differently, and consistency errors were found to be minimal. Most errors were found in regards to numbers and punctuation, but other errors led us to believe that the Java CoreNLP was superior, as it made less mistakes in comparison to our manual analysis.
2. Automatic Code Tagging
    a. Automatic code tagging is an ongoing process, but results so far appear to be great. The team has tested several java files along with a java configuration document (in JSON format) and the automatic code tagger appeared to have no problem parsing tokens of java code into their appropriate tags.

4. Leetcode web scraping
    a. Leetcode is the major expected source of software documentation data, and scraping of data from it has been successfully completed. The scraper written is capable of removing data from any leetcode page, and splitting into plaintext and HTML formats. The plaintext scraped data is capable of being passed through a POS tagger successfully, but the results are not good (as expected). The scraped data has shown that the models need to be updated to work with software documentation, especially code snippets.
5. Javadoc HTML parsing
    a. Javadoc HTML documents are capable of being parsed into plaintext components which can be passed into a training formatter. A parser has been created, and is currently being iterated on to parse HTML without flaws. Currently the program is capable of removing everything but method details from the HTML document.
6. Training a model to recognize code elements
    a. There has been limited training on manually tagged software documentation which has been trained on by the NLP POS tagging system in CoreNLP. The POS tagger was capable of tagging java code at >90% accuracy with a dataset of only one file. While this result is promising, it was completed with a very small dataset, so it may not be a relevant result until the dataset we use for training is larger. However, this result was proof that the pipeline used to train the model will work as intended.

5 Implementation Plan

Scraper

The scraping component is one of the closest components to completion. In essence, the scraping component is required in order to gather enough data for the training of a new NLP model. The scraper currently is capable of scraping specific subsets of websites (currently leetcode and javadoc) and returning the HTML raw data. Once the raw data is scraped, we need a parser in order to convert the HTML into usable training data.

While a portion of this falls onto the tokenizer, a large amount of the HTML needs to be stripped, which is accomplished with a near complete parsing application written in python. The parsing application is responsible for stripping HTML tags while also maintaining a whitelist of tags according to what data the training application requires (which is to be determined). Currently the parser is near complete, with only more generic whitelisting required for completion.

Tokenizer

The tokenizer is responsible for breaking up the plaintext data provided by the scraper / parser units and converting them into trainable datasets. Currently, the stanford NLP pipeline has a tokenizer associated with the language being parsed. Similarly, we need to create a custom tokenizer (sentence and word partitioning) in order to fit the scraped data into the training / analysis pipeline.

The tokenizer is still in the design phase, but a majority of the planning has been done. The tokenizer will be written in Java in order to fit it into the existing Java CoreNLP pipeline as a simple module switch. The tokenizer must be capable of analyzing text not only alone, but with "shouldered" HTML tags incorporated. For example, the tokenizer should put "code" emphasis on everything inside of a <code></code> pair.

Trainer

Training the NLP will be done by feeding raw java code into the java tagger, the java tagger will then format and tag the code into the correct format, after it is formatted, it will feed the tagged code into the NLP. We essentially need 1000s of files of java code to train the model. We will have a manual tagger in between the automatic tagger as well as the NLP, allowing us to resolve issues manually.

Tagger

The tagger is the main purpose of our project. Everything else is for the purpose of creating the POS tagging model. Currently, we've successfully tested a simple model with a small amount of training data by using the *CRFClassifier* from the Stanford Core NLP. This is the same classifier that the standard Stanford POS tagger uses. It uses a bidirectional dependency network which we expect will be sufficient for understanding the semantics of software documentation. The trained model will be usable in both Java and Python.

After producing a working tagger we will test it and report its performance when compared to the base Stanford POS tagger. From there, we will iterate on the tagger and training data until the performance of the model is satisfactory.

6 Closing Material

6.1 Conclusion

The work so far:

So far our team has made great progress on every front.

On the front of general decisions/processes, we decided that we were going to introduce new tags that represented software documentation. We have also decided that we would actually use HTML tags in the code to help us determine what is code and what is not (like <code> and).

When it comes to tagging code, we have come up with a new tag set to tag code, as well as a set of rules to tag some Java code. We also experimented with just using conditional random fields to tag software code without any other rules. We currently have two different ways to tag code functional.

For the beginning of our pipeline we have created an html parser that takes any software documentation (written in HTML) and removes all the tags and other HTML bits that we do not think will be useful to our tagger. On our way to get here we made specific scrapers/parsers for JavaDocs and LeetCode that would take the information and then POS tag it.

Goals:

Our overall goal is to create a program that can be given a piece of software documentation and tag both the English parts and the code parts. As a part of this it should be trainable with new data.

Plan of action:

For next semester, we have three big things we need to complete:

1. Create an HTML tokenizer and sentence splitter
2. Tag training data (software documentation)
3. Create our model and tagger as the final step of the pipeline

With these done, we will be most of the way to a fully functional Software Documentation POS Tagger. Some adjustments at this point may need to be made to the tags and the model in order to achieve satisfactory performance (close to the 97% of the Stanford POS tagger).

Justification for plan of action:

1. We decided that we would indeed need an HTML tokenizer to be able to tokenize HTML tags as HTML tags. Natural language tokenizers like CoreNLP are made for, well, natural languages. It is not all that customizable, and currently splits html tags into 3 or 4 parts due to the symbols.
2. We will need good training data that is tagged software documentation to be able to train our model. We can use a mix of the coreNLP tagger and our software rule tagger to quickly tag training data, and then only have a human go over and check for errors. This is far better than a human manually entering every tag.
3. We are confident that given the use of HTML tags and enough good training data, our model - whether it ends up being inside of the coreNLP pipeline or adjacent via Conditional Random Fields - will be an effective (and flexible) software documentation tagger.

6.2 References

See works cited for references and previous works.

6.3 Appendices

Evolution of the Selection of New PoS Tags:

Iteration 1:

Tag	Description	What is it	Example
<c>	Code that doesn't fit into other categories		
<{>	Open brackets/block		{ ...
<}>	Close brackets/block		… }
<;>	End of statement		…;
<.>			foo.bar, foo.bar()
<fun>	Function		foo()
<par>	Parameter		func(foo, bar)
<?st>	Conditional statement		if(foo) { ... }, else { … }, foo ? x:y
<?op>	Conditional operator		<, >, <=, >=, !=, ==, &&, ||
<loop>	Loop		for(...) { … }, while(...) { … }, do { … } while(...);
<var>	Variable		int foo;
<=>	Gets		foo = 3;

Table 6.3.1: *Created by Austin Boling as an exploration of what new tags should be included in the new parser and datasets*

Iteration 2:

Tag	Description	Example
<am>	Access Modifier	*public static void main()*
<?st>	Conditional Statement	*if (true) { }*
<;>	End of statement	*String hello = "world";*
<type>	Language type	*class Color*
<typen>	Type name	*String* hello = "world"
<{>	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)*
<return>	Return statement	*return hello;*

Table 6.3.2: *Created by Joseph Naberhaus as a refined exploration of what new tags should be included and used in the parser and datasets.*

Iteration 3:

Tag	Description	Example
<am>	Access Modifier	*public static void main()*
<?st>	Conditional Statement	*if (true) { }*
<;>	End of statement	*String hello = "world";*
<type>	Language type	*class Color*
<typen>	Type name	*String* hello = "world"
<{>	Open block	*if (true) { }*
<}>	Close block	*if (true) { }*
<(>	Open parenthesis (in code)	*if (true) { }*
<)>	Close parenthesis (in code)	*if (true) { }*
<,>	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)*
<return>	Return statement	*return hello;*

Table 6.3.3: *Created by Joseph Naberhaus as a further, and possibly final, iteration of the new tags to be included and used in the parser and datasets.*

Works Cited:

[1] The Stanford Natural Language Processing Group. Nlp.stanford.edu. (2020). Retrieved 4 October 2020, from https://nlp.stanford.edu/software/lex-parser.shtml.

[2] Feature-Rich Part-of-Speech Tagging with A Cyclic Dependency Network. K. Toutanova et. al. (2003) Retrieved October 2020, from https://web.stanford.edu/~jurafsky/slp3/A.pdf

[3] Alphabetical list of part-of-speech tags used in the Penn Treebank Project. ling.upenn.edu (2020). Retrieved October 2020, from https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html\

[4] Software-specific Part-of-Speech Tagging: An Experimental Study on Stack Overflow. D. Ye et. al. (2014) Retrieved October 2020, from http://www.li-jing.com/papers/POS.pdf

[5] Part-of-speech tagging of program identifiers for improved text-based software engineering tools. S Gupta et. al. (2013) Retrieved October 2020, from https://www.researchgate.net/publication/261487990_Part-of-speech_tagging_of_program_identifiers_for_improved_text-based_software_engineering_tools