Pilot Operating Handbook Semantic Text Analysis

Project Alpha Prototype Report

**The Boeing Company**

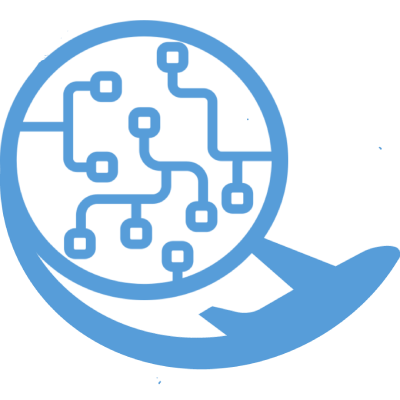


**The Autopilots**

Katie Peterson

Allison Lorphanpaibul

Weiren Lai

Daniel Harker

December 7, 2020

**TABLE OF CONTENTS**

[I. Introduction 3](#_Toc58273290)

[II. Team Members - Bios and Project Roles 3](#_Toc58273291)

[III. Project Requirements 4](#_Toc58273292)

[IV. Solution Approach 6](#_Toc58273293)

[V. Test Plan 10](#_Toc58273294)

[VI. Alpha Prototype Description 16](#_Toc58273295)

[VI.1. Noun Parsing with Context 16](#_Toc58273296)

[VI.1.1. Functions and Interfaces Implemented 16](#_Toc58273297)

[VI.1.2. Preliminary Tests 16](#_Toc58273298)

[VI.2. Database Storage 16](#_Toc58273299)

[VI.2.1. Functions and Interfaces Implemented 16](#_Toc58273300)

[VI.2.2. Preliminary Tests 16](#_Toc58273301)

[VII. Alpha Prototype Demonstration 18](#_Toc58273302)

[VIII. Future Work 19](#_Toc58273303)

[IX. Glossary 20](#_Toc58273304)

[X. References 20](#_Toc58273305)

# Introduction

Artificial intelligence (AI) and machine learning (ML) have become trends of the future; however, the cost of AI and ML is very high, especially in acquiring and maintaining real and production data. Hence, there is a need to be able to support training synthetic data engines like GANs (generative adversarial networks) to create synthetic data on-demand. These engines provide a fast and inexpensive way to produce data that is properly marked and tagged, unlike most real and production data.

Our goal is to develop a semantic text parsing solution that takes vocabulary from Boeing aircraft handbooks, written in Boeing’s *Simplified Technical English,* and use this vocabulary to train GANs. Specifically, the solution will identify and store each noun/noun phrase, along with its designated document name, year, product, location, and 1-50 sentences demonstrating the context of this noun. After training with this dataset, we expect GANS to produce synthetic text data written in Boeing’s *Simplified Technical English* with a 60-80% quality [1].

The project’s focus will primarily lie in the field of *Natural Language Processing (NLP)* - how we teach machines to conceptualize human language. For a GAN that will produce text data, training often involves supplying it with both linguistic rules and a set of vocabulary words. An important aspect of this is to ensure the GAN understands the meaning and context of the nouns in its vocabulary. In English, many nouns have different meanings depending on their context. It is thus important to ensure that when a GAN produces text data, it is using nouns appropriately.

This document serves to provide a summary of the project’s progress thus far and to elaborate on the technical details of our team’s engineering efforts.

# Team Members - Bios and Project Roles

Our team consists of four undergraduate students each studying computer science at Washington State University.

Katie Peterson serves as the team lead and is responsible for the data design of the system. Her previous experience includes working as a software engineering intern at Premera Blue Cross and serving as an undergraduate teaching assistant at WSU.

Allison Lorphanpaibul is responsible for the system’s sentence parsing and user interface. Her previous experience includes working as a software engineering intern at HP and serving as a tutor in the Academic Success and Career Center at WSU.

Weiren (Will) Lai is responsible for the general testing of the system and for developing the CI/CD pipeline. His previous experience includes completing coursework in machine learning, software engineering, and algorithm design/analysis.

Daniel Harker is responsible for the system’s noun parsing. His previous experience includes serving as an undergraduate teaching assistant at WSU and completing coursework in machine learning, web development, and algorithms.

# Project Requirements

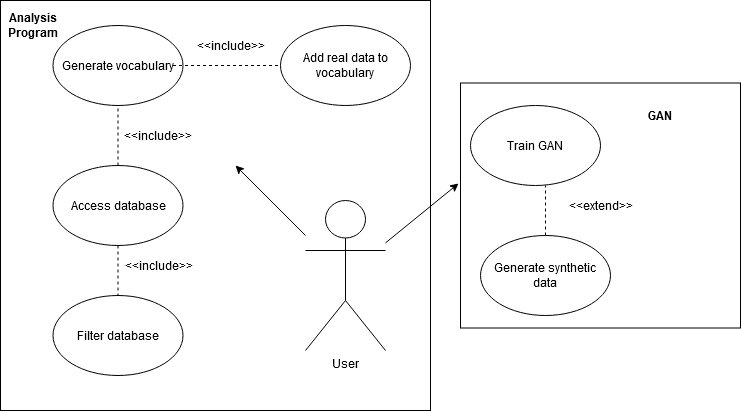
This section will describe the overall requirements of the project that are expected from our team’s final design. It is broken down into three sections: use cases, functional requirements, and non-functional requirements.

* 1. **Use Cases**

|  |  |
| --- | --- |
| Use Case | User generates a constrained vocabulary |
| Pre-condition | (none) |
| Actions | 1. User inputs document written in simplified English 2. System responds by parsing the document 3. System puts parsed vocabulary into database/spreadsheet 4. (optional) User inputs vocabulary as training data for GAN |
| Post-condition | Database has new entry with parsed vocabulary |
| Acceptance tests | Database entry exists |
| Requirements | * Algorithm for noun parsing * Identification of noun information |

|  |  |
| --- | --- |
| Use Case | User filters constrained vocabulary lists |
| Pre-condition | Database/spreadsheet with vocabularies has been generated from the program. |
| Actions | 1. User opens up database/spreadsheet. 0 2. User can query from data using different filters, such as year, date, document, etc. using options provided. 3. The system responds with showing the filtered lists |
| Post-condition | User can easily filter different vocabulary lists |
| Acceptance tests | Correct data is shown |
| Requirements | * Data is stored in a database/spreadsheet * Identification of noun information |

|  |  |
| --- | --- |
| Use Case | User adds real/production data to vocabulary |
| Pre-condition | Database/spreadsheet with vocabularies has been generated from the program. |
| Actions | 1. User inputs vocabulary and real data 2. System produces synthetic data 3. User inputs data for GAN training |
| Post-condition | Vocabulary will deliver 90% quality test data when training the GAN. |
| Acceptance tests | New test data delivers 90% quality |
| Requirements | * Data is stored in a database/spreadsheet |

* + 1. **Use Case Diagram**

Above shows the two different areas of use cases for a user. Our text analysis program (left) will work independently from the GAN (right). The GAN program and real data will be provided by Boeing.

* 1. **Functional Requirements**
     1. **System Input & Output Requirements**

**System Input:** The system must read as input 3-10 Boeing pilot operating handbooks in PDF format.

**Source**: Rakshit Bhatt and Don Brancato of The Boeing Company. Necessary to ensure the nouns that are parsed are used in the correct context for Boeing.

**Priority**: Level 0 - Essential and required functionality

**System Output:** The system must store data collected in a MySQL database or Open Office Spreadsheet.

**Source**: Rakshit Bhatt and Don Brancato of The Boeing Company. Necessary to ensure the data can be located and used to train GANs in the future.

**Priority**: Level 0 - Essential and required functionality

* + 1. **System Parsing & Identification Requirements**

**Algorithm for Noun Parsing:** The system must have a semantic text parsing algorithm, capable of parsing all nouns from text written in Boeing Simplified English.

**Source**: Rakshit Bhatt and Don Brancato of The Boeing Company. Necessary to ensure nouns are collected for use in training GANs.

**Priority**: Level 0 - Essential and required functionality

**Identification of Noun Information:** For each unique noun parsed, the system must also identify the following: 1 to 50 sentences from the various documents that contain the noun, the name of the document(s) the noun appears in, the product associated with the document(s), the publication year of the document(s), and the location of the document(s).

**Source**: Rakshit Bhatt and Don Brancato of The Boeing Company. Necessary to ensure the GANs are trained with context for each noun.

**Priority**: Level 0 - Essential and required functionality

* 1. **Non-Functional Requirements**

**Reliability/Performance – Accuracy Percentages for Real and Synthetic Data**

The program must be able to consistently produce accurate results for the given data. When testing the GANs with real data, we expect to have 40-70% accuracy. When pairing real data with synthetic, this accuracy should jump up to 90%.

**Response Time – Quickly Generate Constrained Vocabulary**

The program needs to be able to generate a constrained vocabulary quickly. Since there are thousands of simplified languages at Boeing, this program will need to be able to create the vocabulary in a timely manner so that it can be used efficiently.

**Readability – Create Simple Databases of Nouns**

With the input to the program being Boeing-provided documents and manuals that are hundreds of pages long, the output needs to be clear and concise. This output will take the form of a database of nouns that can be read by humans and parsed by computers.

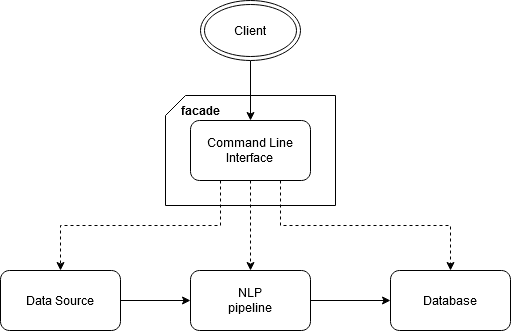
**Usability – Straight-forward Creation of Simplified Languages by User**

The program interface needs to be designed so that any user can select multiple files that will be turned into a simplified language. This design needs to be uncomplicated and straightforward to allow for ease of use.

# Solution Approach

We have divided our solution into three components: architecture design, data design, and user interface design.

* 1. **Architecture Design**

The system will use a simple data pipeline architecture. Raw data from the user will be input at the start of the pipeline, processed by the internal subsystems, and finally be output to a database.

Since our system does not require that much user interaction, the interface layer of the system will be simple. The *facade* design pattern can be employed here to provide a simple way for the user to interact with the system easily, without having to view all the processes under the hood, reducing complexity. To start, we can use the command line as an interface for the user. This pattern will also provide room for extensibility of the interface in the future if there is time as described in [2].

At this time, our system will not require a lot of information hiding, so the interface does not have to be too sophisticated.

* 1. **Data Design**
     1. **Internal Data Structures**

We will be creating objects out of our own classes, as well as the 3 types of containers spaCy includes: Doc, Span, and Token. Below is a summary of the data structures and where to find any spaCy documentation on them:

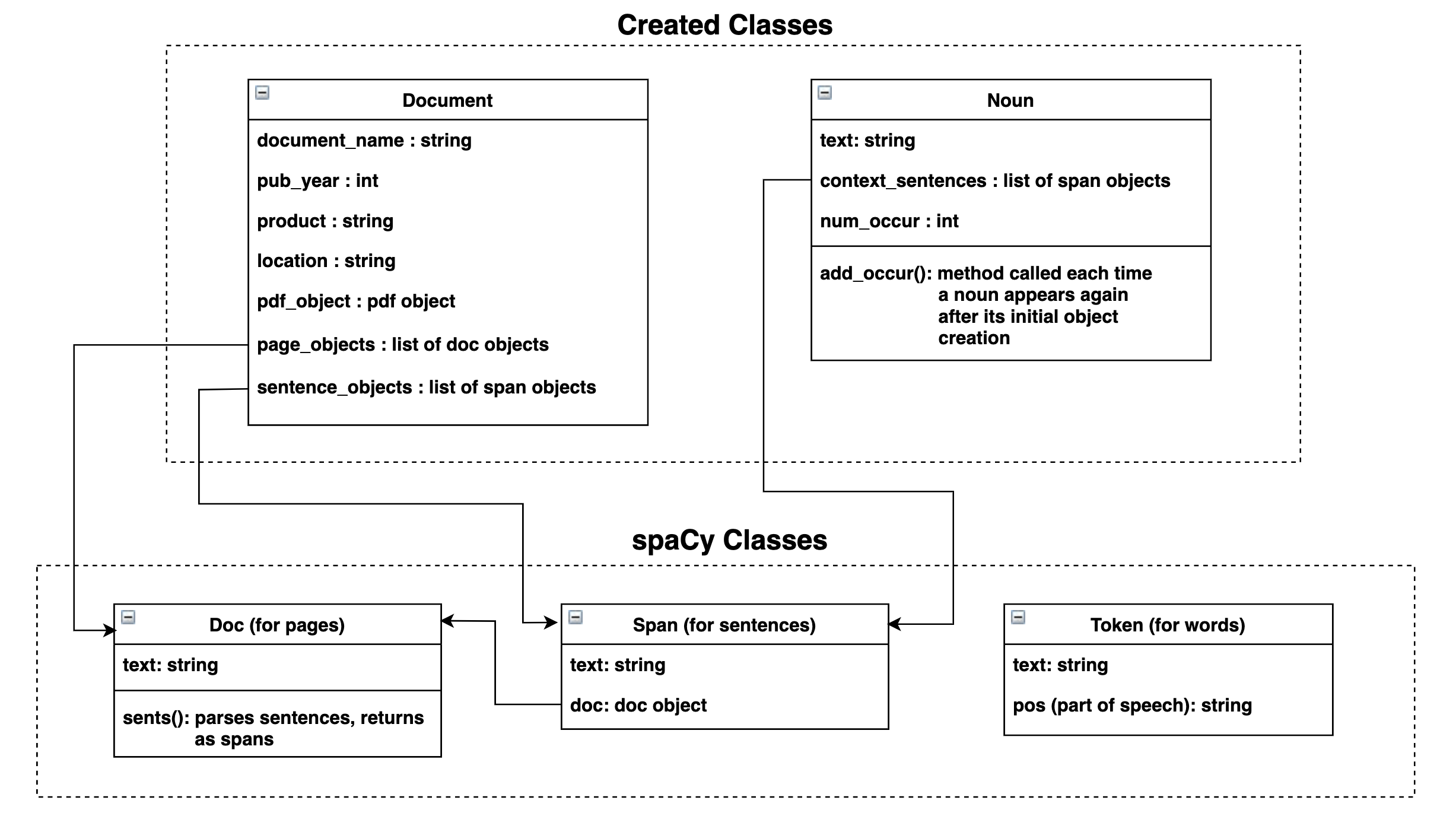
Document objects will be used to store information about the document(s) the user uploads. This includes the document's name, its publication year, the product it is written about, its location, and a list of sentence/span objects found in this document.

A Doc container is a way to store a block of text and perform operations on it. Our program will create a Doc object for each page of text from the PDF the user uploads.

The block of text in the Doc object can then be divided into smaller parts, called Spans. Our program will create a Span object for each sentence found in the block of text. Spans store the sentence as a string, the main document the sentence came from, the Doc (page) the sentence came from, a list of the objects created for nouns in the sentence, and a list of the nouns as strings (to make comparing the nouns easier/more time efficient)

The Span objects can be divided into Tokens. While parsing, a Token object will be created for each word in the sentence.

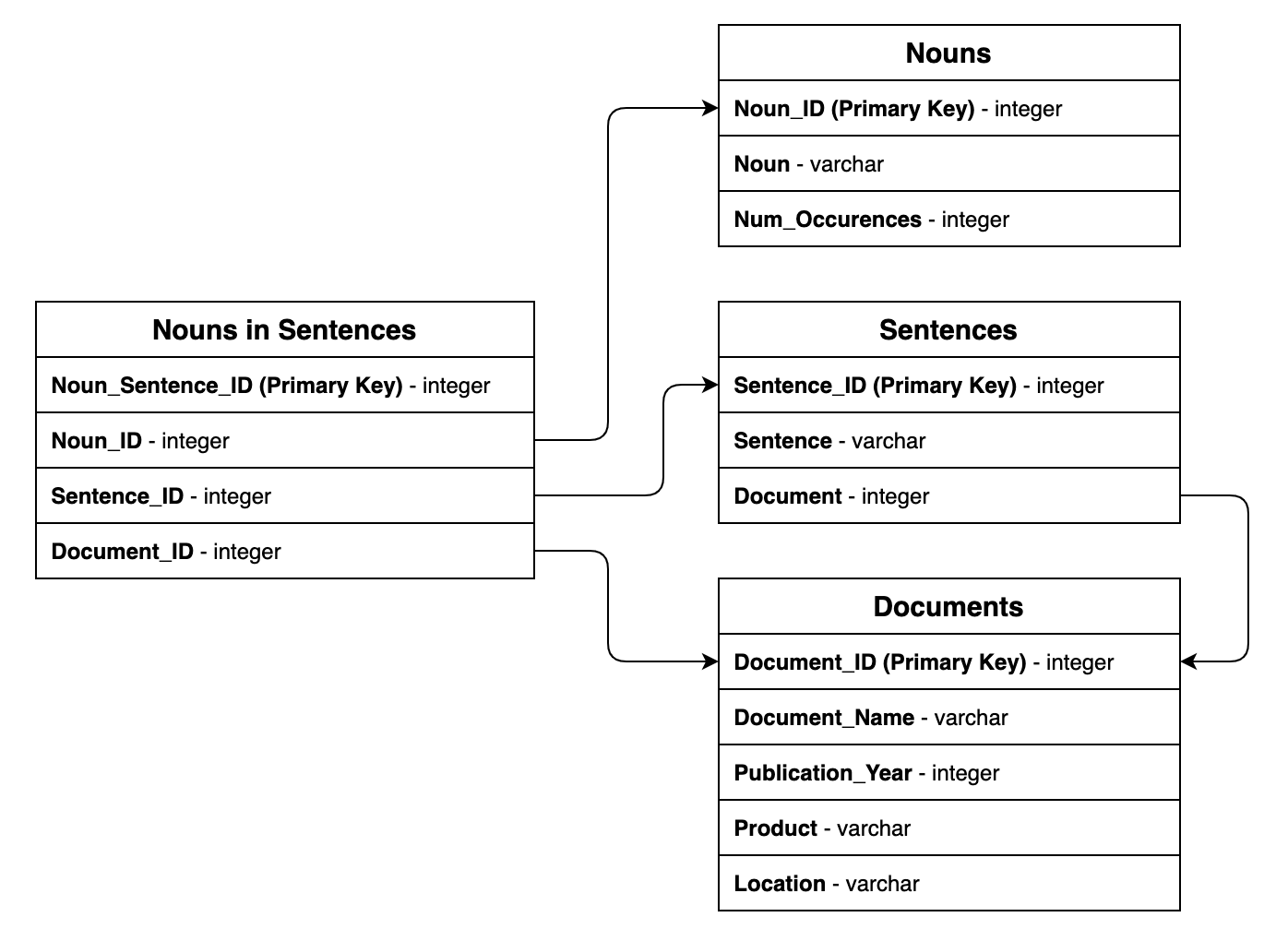
Tokens will then be analyzed to determine if they are a noun or not. If the token is a noun, a noun object will be created for that specific noun. Each noun object has the word stored as a string, the number of times it has appeared, and a list of Span objects - one for each sentence the noun appears in.



* + 1. **MySQL Database**

Our system will export the data it collects to a MySQL relational database. The database will consist of four tables. The first table will contain the name of all documents used as input to the system, along with their publication year, product they were written for, and the location where they can be found. The second will contain a list of all sentences read by the system, along with the document they appear in. The third table will list all unique nouns found by the system and the total number of times they appear in the documents. Finally, the fourth table will relate the documents, sentences, and nouns by listing each noun’s primary key with the primary key of each sentence it appears in, along with the primary key of the document that sentence appears in.

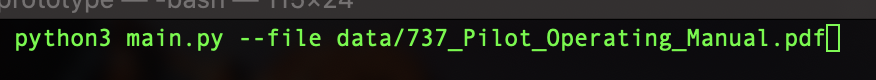
Below is a visualization of the database and its 4 tables:



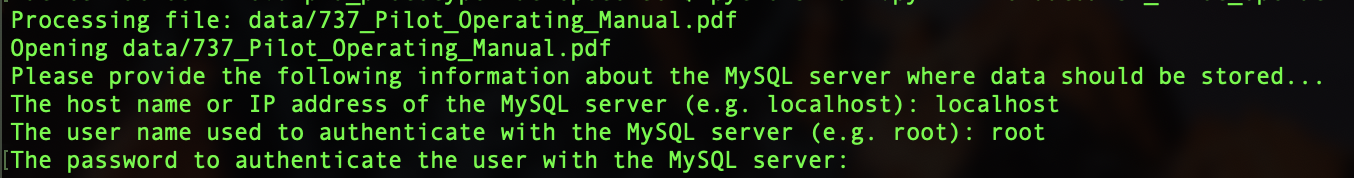
* 1. **User Interface Design**

Based on our client’s expectation and the small amount of user interaction required, our team decided to use a command-line based approach to the user interface.

First, the user will launch the program with the path to the PDF document they want read. As of right now, the program can only read one file at a time. In future versions, we plan to allow multiple files to be read at a time.



After reading and parsing the PDF document, the user will be prompted to provide information about the server they want to create the new database in.



Once created, the database will store the information that was collected from the PDF document. The user will be given the name of the database so they know where to access the data.



# Test Plan

Our team will use several testing methods for our system, and we will follow the test pipeline outlined in the Testing Strategy section below. Most system and unit testing will be written before or in tandem as their corresponding features. Performance, chaos, and user acceptance testing will most likely be done after the development of our working prototype.

Members will be in charge of developing test cases for the features that they are in charge of.

* 1. **Testing Strategy**

We will use GitLab’s CI/CD for our system testing. For other testing, such as performance testing, we may require access to Boeing’s infrastructure to get information such as accuracy metrics.

As our project grows, the pipeline may need to be extended with new required tests. Because we will be adding new features frequently during the initial phases, having a clear workflow for testing will be helpful.

Our testing workflow will be as follows:

|  |  |
| --- | --- |
| 1. | Identify any new test cases. Tests will be mainly derived from the Software Requirements Specification. |
| 2. | Identify which particular tests will be used to test each module. |
| 3. | Write necessary tests. Document all test case information. |
| 4. | Add new changes. Perform the tests. |
| 5. | Submit new changes to unit testing. Document any test data from the testing process. |
| 6. | If unit tests pass, the changes will be eligible for integration and system testing. |
| 7. | Once all system testing is complete, the change is eligible for deployment. |
| 8. | All testing documents and changes will be reviewed by a peer. |
| 9. | Changes can be revised and resubmitted for testing. |
| 10. | Once passing review, changes can be deployed. |
| 11. | In the case of unsuccessful tests, a bug form will be generated describing the test case, the problem encountered, its possible cause, and the sequence of events that led to the problem. If necessary, this form can be submitted for later analysis. |

* 1. **Testing Tools**

For our testing tools, we came up with a few ideas. We have decided to use pytest and unittest or either one. We are also aware of how different they are in comparison.

Unittest provides test cases, test suites, test devices, and related classes of test running programs, making the test clearer, more convenient, and controllable.

To write use cases using unittest, the following rules must be followed:

1. The test file must be imported into unittest first.

2. The test class must inherit unittest.TestCase.

3. The test method must start with “test\_”.

4. The test class must have the unittest.main() method.

However, pytest is a third-party testing framework for python and an extended framework based on unittest, which is more concise and more efficient than unittest.

To write use cases using pytest, the following rules must be followed:

1. The test file name must begin with “test\_” (eg: test\_ab.py).

2. The test method must start with “test\_”. (It is obvious here that pytest is much simpler than unittest for writing test cases. pytest can execute unittest-style test cases without modifying any code of unittest case, which has better compatibility).

Also, there are lots of pytest plug-ins, such as flask plug-ins, which can be used for error reruns of use cases; and xdist plug-ins, which can be used for parallel execution of devices.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Complexity | Compatibility | Plug-ins |
| Unittest | Complex | None | Barely |
| Pytest | Simple | Supports unittest | Many |

* 1. **Unit Testing**

The primary goal of unit testing is to take the smallest unit of testable software in the application, isolate it from the remainder of the code, and test it for bugs and unexpected behavior.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | System | Test Data | Expected Result |
| Test sentence tokenization base case | Sentence Parser | A paragraph | Sentences should be parsed properly |
| Test sentence tokenization no stop punctuation | Sentence Parser | A paragraph with no punctuation | Paragraph should be unchanged |
| Test incorrect command | User Interface | Misspelled/incorrect command | Error message. Help menu displayed. |
| Test table implementation | Database | System run with blank PDF | Empty tables are created |
| Test table accuracy (1 PDF) | Database | System run with PDF containing a single paragraph | Tables are populated properly and with correct data |
| Test table accuracy (multiple PDF’s) | Database | System run with 2 PDF’s each containing a single paragraph | Tables are populated properly and with correct data |
| Test text parsing algorithm | Sentence parser | A PDF written in simplified English that has at least 100 pages | Parsed words with information tagged to each word such as year, location and etc.. |
| Test system response time | File import, sentence parser, user interface | A PDF written in simplified English that has at least 100 pages | output each function’s response time after running the entire system through |

* 1. **Integration Testing**

Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Two or more components are integrated and tested, and when no new faults are revealed, additional components are added to the group.

* + 1. **Test Incorrect Command with Sentence Tokenization**

After knowing that the user interface works for obtaining specific documents to open for reading, we need to be sure it can actually be used for tokenizing the document by sentences. This will be tested using a small sample size, in which the sentences will simply be output to the terminal for confirmation. Once it’s determined that the two components work together, we will be able to use actual Boeing documents.

* + 1. **Test Sentence Tokenization with Noun Parser**

Now that it’s confirmed that we can obtain sentences from an input file, we need to confirm that we can parse nouns from them sentences. This will likely focus mainly on the variable type of the sentences and how that type is compatible with the noun parser program we decide to use. Similar to testing incorrect command with sentence tokenization, the nouns from each sentence will be displayed to the terminal for confirmation. This integration test will start firstly with a single sentence, then progress to a paragraph, to ensure that multiple sentences perform identical to a singular sentence as input.

* + 1. **Test Sentence Tokenization and Noun Parser with Data Storage**

This integration test has two connected parts to it: storing sentences and storing nouns. Since sentences and nouns are linked together across their respective data tables, they’ll need to be stored together. This will be done by storing a sentence in its respective data table and linking each noun detected by the noun parser as described in WA3. Testing will start with a paragraph in order to easily check for accuracy, then increase in size to multiple paragraphs. After it’s determined that these parts work together flawlessly, we will be able to run the program on actual Boeing documents.

* 1. **System Testing**

System testing is a type of black box testing that tests all the components together, seen as a single system to identify faults with respect to the scenarios from the overall requirements specifications. Entire system is tested as per the requirements.

During system testing, several activities are performed:

* + 1. **Functional Testing**

Test of functional requirements (from requirements specification). The goal is to select those tests that are relevant to the user and have a high probability of uncovering a failure.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Functional Requirement | Steps | Test Data | Expected Result |
| Test invalid system input | The system must read as input 3-10 Boeing pilot operating handbooks in  PDF format. | 1. Input a file in the incorrect file format  2. Wait for results  3. Check the database | A file with an invalid extension (doc, csv, etc.) | The system should report an error-- it should not crash. No data should be produced. |
| Test non-simplified English system input | The system must read as input 3-10 Boeing pilot operating handbooks in  PDF format. | 1. Input a file in the incorrect file format  2. Wait for results  3. Check the database | Any pdf with text not written in simplified English. | The system should output to the database. |
| Test output if there is an existing database previously created by system | The system must store data collected in a MySQL database. | 1. Run system with file(s) so that a database is created  2. Run system again with same file(s)  3. Check the database | Any PDF(s) written in simplified English | The system should create a new database and not overwrite the existing database |
| Test text parsing algorithm | The system is able to parse all nouns from text written in Boeing simplified English | 1. Run system with an imported PDF file.  2. Parse the text  3. Check the result in the database. | At least a PDF written in simplified English from Boeing. | The algorithm parsed the text in the PDF and outputted the results into the database. |
| Test nouns with specific requirements | The system must parse and identify 1 to 50 sentences from the various documents that contain the noun, the name of the document(s) the noun appears in, the product associated with the document(s), the publication year of the document(s), and the location of the document(s). | 1. Run the system with an imported file.  2. Parse the files.  3. Identify a list of information about each noun. | PDFs written in simplified English | A dataset that contains a list of nouns and a list of information that is related to each noun such as location, year etc. |

* + 1. **Performance Testing**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Non-Functional Requirement | Steps | Test Data | Expected Result |
| Test the readability of the database the system creates. | With the input to the program being Boeing-provided documents and manuals that are hundreds of pages long, the output must be clear and concise. It will take the form of a database that can be read by humans and parsed by computers. | 1. Run system with specified documents  2. Wait for results  3. Check to ensure the database was created, is accessible, and contains data from each of the 10 documents. | 10 PDF documents written in Boeing simplified English, each with at least 100 pages. | The system creates a database with the necessary amount of memory to store all data from the 10 documents.. The database is able to be opened on the computer and be viewed by the user. |
| Test the accuracy of noun parsing | When parsing a sentence by each word, the system should be able to determine if each word is a noun or not. | 1. Run the system  2. Print nouns and their counts  3. Check the output compared to the known, desired results | Start small with a paragraph. Progress into using multiple paragraphs | The system should generate 100% accuracy in finding nouns. |
| Test the response time of the program. | After inputting Boeing documents, the system is able to run and output a response time to each process accordingly. | 1.Run the system.  2. Check the duration time of each process. | 10 PDF documents with Boeing simplified-English. | The system is able to run and output the time taken for each process. |

* + 1. **User Acceptance Testing**

Our team met with our clients, Don and Rocky, on December 3rd to perform user acceptance testing. We demonstrated the alpha prototype we created and received positive feedback from Don and Rocky. Our project passed the overall user acceptance testing. More details about this demonstration can be read below in section VII. Alpha Prototype Demonstration.

* + 1. **Chaos Testing**

Once the prototype is in a working state, we can perform chaos testing on it. The goal of chaos testing is to cause system failures. To start, we can perform naive chaos testing by simply inputting bad or volatile data into the system. If the system becomes robust enough, we can implement a chaos monkey to automatically and randomly perform tests on the system that will cause it to fail.

# Alpha Prototype Description

## Noun Parsing with Context

### Functions and Interfaces Implemented

Through the command line, the program is able to accept a PDF as input, open the PDF, and extract the text. An object is then created for the document as a whole, as well as an object for each page of the document. From here, the program will parse the text by sentence and create an object for each word. The program then determines which of these words are nouns, and stores them as objects created from the Noun class. The number of times a noun appears in the PDF is also counted. A visualization of the document, page, sentence, word, and noun classes can be seen in the diagram in section IV.2.1.

### Preliminary Tests

So far, the only issue with getting the nouns is the fact that the program doesn’t get all of the text out of the PDF. This issue isn’t due to coding errors but simply that the import (pdfplumber) isn’t doing its job completely. Of the text extracted, all nouns are found, and all sentences are treated as they should be. See section VI.2.2 for screenshots of the results from the text that is extracted.

## Database Storage

### Functions and Interfaces Implemented

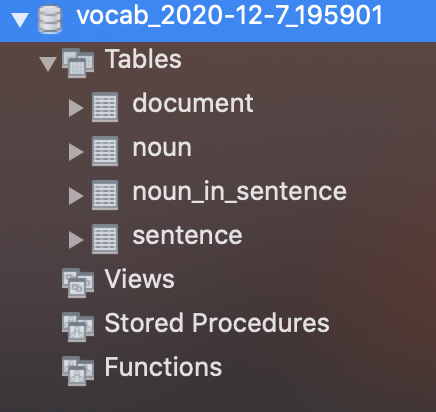
Each noun, sentence, and document is stored in a relational database. As seen in the diagram in section IV.2.2., the document table consists of information regarding the PDF that was uploaded by the user, along with a primary key ID. The sentence table consists of each sentence parsed from the document, along with its own primary key and a foreign key linking to its parent document’s primary key. The noun table consists of each unique noun parsed from the PDF, along with a primary key ID and the number of times the noun occurred in the text.

These primary keys of the elements in each of these tables are then combined in the noun/sentence relation table, where each occurrence of a noun is listed and paired with the sentence and document it came from. For example, if the noun “airplane” appeared 3 times in a document, it would be listed 3 times in the relational table – one for each sentence it appeared in.

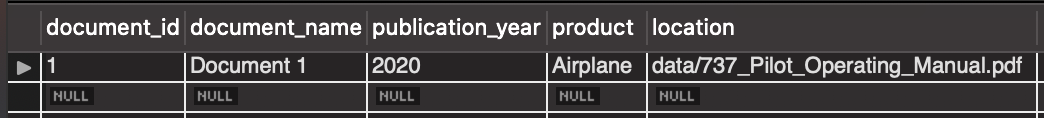
### Preliminary Tests

The database is currently working correctly with nouns and sentences stored, as well as linking all nouns to the sentences they came from.

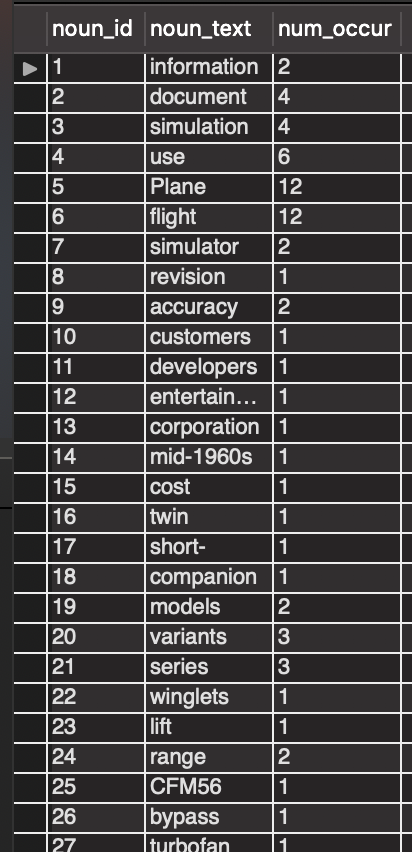
Screenshot of the created database:



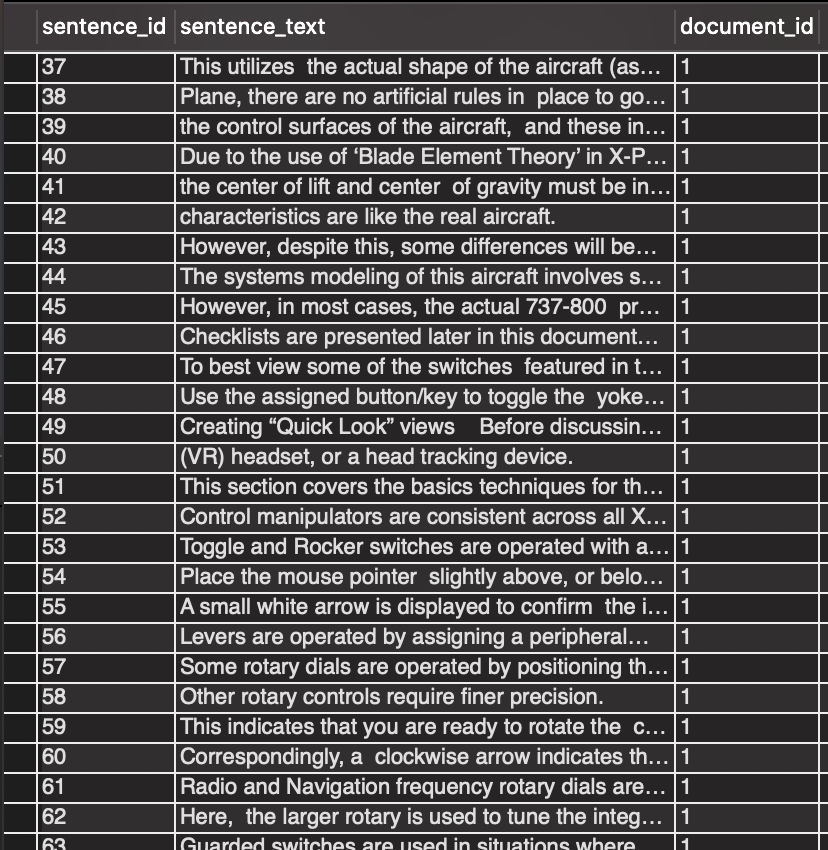
Screenshot of Document Table



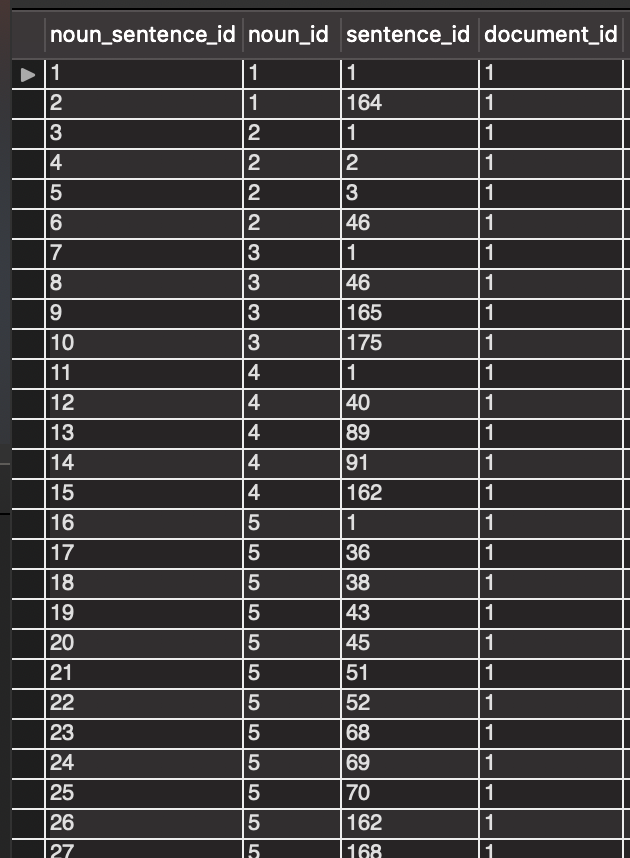
Partial Screenshot of Noun Table



Partial Screenshot of Sentence Table



Partial Screenshot of Noun/Sentence Relation Table



# Alpha Prototype Demonstration

We demonstrated our alpha prototype to our mentors, Don and Rocky, during a meeting on December 3rd. We demonstrated how the program reads the PDF documents and handles noun parsing, as well as inserting data into the database.

After the demonstration, our mentors commented on the program. First, they asked us to investigate how Google has handled noun parsing. Google is a good example in terms of file and language processing. Second, they mentioned setting up syntactic rules is an option as well, and that by setting up syntactic rules, we would have parameters for the program to have a more precise and accurate result. Third, they suggested we read the PDF document into a text file so we could handle the text better. Lastly, they prompted us to look out for misspellings in the Boeing provided handbooks and to be able to handle these misspellings appropriately.

We all agreed on the fact that we should let multiple users try out the program to see if they are able to break it. This is needed since we are going to process a large amount of data and we do not want it to break at any time.

At the end of the meeting, our mentors asked us what we used to extract the PDF text and the team responded with pdfplumber. pdfplumber is a library that can handle PDF format information. It allows us to find detailed information about each text character, matrix, row and lets us extract and visually debug the table.

Overall, it was a helpful, insightful, and meaningful demonstration. We have concluded that we still have a long way to go, but at least we have a good and solid start.

# Future Work

Next semester we will focus on polishing and improving our current prototype.

The following are tasks we plan on doing next semester, in order of priority.

1. **Fix issues with text parsing**

There are a couple of issues in our prototype that need to be fixed as soon as possible: fixing data loss when reading the pdfs and fixing issues with sentence parsing. For fixing pdf reading, our mentors recommended using Jupyter notebooks. For improving sentence parsing, we need to clearly define more syntactical rules to remove sentence parsing errors.

1. **Improve parsing algorithms**

There are many ways we can improve the parsing algorithms in our current prototype besides removing errors from sentence parsing. We need to make sure that nouns are being counted properly. The following are currently things we would like to add:

* Add noun lemmatization to make sure nouns are not counted separately (e.g. airplane and airplanes counted as two separate nouns)
* Handle misspelled nouns
* Handle noun gerunds

1. **Add ability to run on multiple files**

This is something required from the original project description. This should be relatively simple to add using argparser.

1. **Accuracy testing with GAN**

First, we need to combine our generated vocabularies with actual data. Then, we need to input it into a GAN and test for accuracy. We are expecting 90% accuracy. This is an important test metric.

1. **Try different approaches/libraries**

There are many NLP libraries for python. Trying different libraries out may reveal that other libraries/approaches are better suited for our application’s needs.

# Glossary

**Generative Adversarial Networks (GANs) -**  A popular machine learning model used to create synthetic data on-demand.

**Natural Language Processing (NLP)** - The field pertaining to how to program machines to

understand human language.

**Simplified English -** The standard for written aerospace documents. Utilizes a restricted vocabulary and simple grammar rules.

# References

[1] Rakshit Bhatt & Don Brancato, “Pilot Operating Handbook Semantic Text Analysis”, The Boeing Company, Washington, USA, 2020.

[2] Gamma, Erich, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Massachusetts, USA: Addison-Wesley Publishing Company, 1995. [Online]. Available. [https://archive.org/details/dehttps://archive.org/details/designpatternsel00gamm/page/185/mode/2upsignpatternsel00gamm/page/185/mode/2up](https://archive.org/details/dehttps:/archive.org/details/designpatternsel00gamm/page/185/mode/2upsignpatternsel00gamm/page/185/mode/2up). Accessed: October 23, 2020.