

# Natural Language Parser and Taxonomy Creation Tool

Final Report



**Team Boeing NLP:** Brandon Christensen, Riley Hunter, and Kadir Nour

**Mentors:** Don Brancato, Rocky Bhatt, Boeing

**Instructor:** Ananth Jillepalli

**Last Revision:** 5/2/2022

## **Table of Contents**

<b><u>Section</u></b>	<b><u>Title</u></b>	<b><u>Page</u></b>
1.0	Introduction .....	3
2.0	Team Members - Bios and Project Roles .....	5
3.0	Project Requirements .....	6
4.0	Software Design.....	13
5.0	Test Plan .....	20
6.0	Projects and Tools Used .....	23
7.0	Description of Final Prototype ..	24
8.0	Product Delivery Status .....	31
9.0	Conclusions and Future Work .....	32
10.0	Acknowledgements.....	34
11.0	Glossary .....	35
12.0	References .....	37
13.0	Appendices .....	38

# **1.0 Introduction**

## **1.1 Project Goals/ Motivation**

The following section until “*Where “We” Fit in*” highlights the points given to us by our mentors on goals/motivations of the project via email correspondence.

*Overall Problem:* In general, current taxonomy services are crowd sourced through companies. Most of these taxonomy services are not extensible outside of their companies context. Additionally, most parsers only service companies and not the individuals as well. Our parser however is largely extensible and can benefit companies and individuals alike for broad or specific taxonomy needs.

*Semantic Parser:* A tool/service that allows computers to understand a constrained vertical including sentences, paragraphs, tables and whole documents. They identify grammar, and establish relationships between terms given context of the term. This is called semantic analysis which is a subset of NLP. This service will be available given a service catalog and would be automatic in nature. The semantic parser is certainly more accurate than any human, or crowd sourced terms with intermediate vocabulary when creating a taxonomy.

*On Demand Access:* Users should be able to use services on demand, independently and inexpensively. Considering this, it makes sense for there to be a cloud-based solution where users can easily shutdown when not using or to use during off hours. Cost saving for such a dynamic would be large.

*LibSci/ Information Science (not data science):* Personnel are provided the task of enterprise vocabulary governance by vertical, and having a large body to work with in a standard format/ output, statistically relevance is high, and data engineering tasks (cleaning data, or dealing with sparsity) is minimized. Parsers can be rule-based, only collecting content that is specified, or run traditionally to parse sentences, or tabular records, to the part of speech – i.e., nouns, noun phrases, determinants and articles, prepositions, adverbs and adjectives.

A parser is technically accurate, more so than humans. Given that TBC is a global organization, parsing can represent what is ‘mostly used in the USA as the base-vernacular’ for the parsing activity. Having vocabulary products based on the actual sentence or record affords the LibSci community with the ability to publish vocabulary by named vertical, and by role, date/time, topic, and location. Vocabulary products are the ingredients of taxonomy; taxonomies are one of the components of effective ontology.

*Masking:* Done by machine, given words or phrases that are ‘controlled’ by GTC/ GPO/ IP for a given nation/state. When the noun is found, it can be either substituted for a more general, less descriptive noun, or deleted/ blackened from the text. Today, humans are used to masking structured data, and the cost, backlog, and error rate is high. If we leave everything to GPO/GTC/IP, sometime next century we can mask content by nation state.

Today, we are unable to accurately mask non-structured data to any volume by humans accurately. Given a taxonomy of words, from the specific and detailed to the most general, a word like molybdenum, a precious metal used to harden certain alloys used in the nose cone of missiles and weapons, we can substitute it with the phrase ‘hardened metal,’ an intermediate term, or the more general still, ‘metal’; thus, a nation state that cannot see molybdenum in TBC

text, can now surely see hardened metal or metal. Again, a taxonomy includes, or can, synonyms, and their general to specific or detailed terms.

*Where “We” Fit in:* Our mentors at Boeing are interested in creating a better data science tool. A tool that can parse text out of a document already exists. A tool that can create categories and relationships between categories already exists. What Boeing is missing is a single tool that can incorporate both of these operations into a seamless and visually appealing process.

Our team has designed a front-end that will walk an expert through each step in the pipe-line process of creating a taxonomy (data gathering, context extraction, categorization, and visualization). Our front-end is interactive, intuitive to use, and it visualizes the process at each step. The expert can save their work in CSV and JSON files. These files will be used in load and to give experts the raw data that our program produces.

The last but most important addition to our data science tool is the creation and use of context with our data. Current tools do not gather context, and this is a huge oversight. In order to have quality information, there must be two components, the data itself, and the data’s context.

For our program, the context we find for each term is: (1) The sentence the term is found, (2) The document the sentence is found, (3) The frequency of the term, and (4) The weight of the term in relation to the amount of documents.

Another problem that context solves is human language being semantically confusing. An expert using data without knowing its context will run into semantic issues understanding the data. An example can be seen if our app was to be used to parse a music genre document. What would happen if the parser pulled out the relevant term “blues”. Without context, the expert could interpret “blues” in multiple ways. It could be a color, it could be a mood/ feeling, or it could be a music genre. Our tool would solve this confusion by giving the expert the context that this data, “blues” is found in.

## **1.2 Mentor Information**

We are working together with two Boeing employees:

### *Don Brancato:*

- Position: Chief Strategy Architect at Boeing
- Contact: Don.Brancato@boeing.com

### *Rocky Bhatt:*

- Position: Enterprise Architect at Boeing
- Contact: rakshit.j.bhatt@boeing.com

## **1.3 Client Information**

Boeing is the world’s largest aerospace company and leading manufacturer of commercial jetliners, defense, space and security systems, and service provider of aftermarket support. They are most well known for their manufacturing and exportation of airliners. Boeing can be organized into three business units: Commercial Airplanes; Defense, Space & Security; and Boeing Global Services.

## **2.0 Team Members - Bios and Project Roles**

Below is information on the members in Team Boeing NLP:

### **Brandon Christensen:**

- Degree: Computer Science
- Project Role: Team Lead, Technical Writer, Full Stack Developer, Software Architect
- Expertise: Python, Javascript, CSS, Flask, React, Communication.

### **Kadir Nour:**

- Degree: Computer Science
- Project Role: Technical Writer, Test Engineer, CSV Loading, Back-End.
- Expertise: Python, Javascript, React

### **Riley Hunter:**

- Degree: Software Engineering
- Project Role: Technical Writer, Containerization, CSV Saving, Back-End.
- Expertise: Python, SQL, Relational Databases, Flask

## **3.0 Project Requirements**

### **3.1 Project Stakeholders**

Our major stakeholder is someone that uses data science tools. Our tool will optimize their results by including context to the data. It will also increase their efficiency by having a single tool that seamlessly incorporates multiple processes into a front-end.

Secondary stakeholders are companies that currently deal with large amounts of data. Our tool could be used in their marketing, security, and database units. The following information are points that were forwarded to us by our project mentors via email (as goals the project will eventually accomplish):

1. Bundling of parser with taxonomy tool to support taxonomy as a service by role, dateTime, topic, location.
2. Parse as a service to support vocabulary products by vertical (audience bizArch, EA, SA, dev/engineering, data science ML/AI authors, InfoSec engineers - ITAM).
3. Parse services for InfoSec to support enhance ABAC and/or Vocabulary-Based Access Control (VBAC) – security to the sentence, or authentication based on vertical vocab use (compare to ABAC authentication to the document or area of interest – gross versus detailed access).
4. Parse Server supports Backend as a service for API and code maintenance of existing applications/services (vocabulary by vertical enables teams to understand data/context, information, knowledge).
5. Mask as a service support – given entity (human, system, machine) that has access to knowledge, information, data in some nation state, ‘control’ vocabulary with masking (replacement with more general term(s) or cover/delete) based on using intermediate or general synonyms of detailed words (e.g., hard material -> metal -> molybdenum).
6. Support Knowledge workers with vocabulary products over information verticals that support knowledge domains.
7. Support Information workers with vocabulary products with data and context by named vertical.
8. Support Data Science, AI, and ML workers with vocabulary products that have words/phrases and context (record and sentence) by table and document.
9. Support SIEM, ITSM, translation services, question answering, ontology induction, automated reasoning services and code generators with parts of speech (nouns, verbs, adjectives, etc.) that are precise to the domain.
10. Support chatbots. Increasingly chatbots are being specified, but suffer when taxonomies of terms are not available for the bot to use, based on the audience location, topic, and role. Given vocabulary products that are the elements of taxonomy, chatbots can simply detect the audience and location, and use a taxonomy by vertical vocabulary product to use the property terms for the named audience (think data catalog of vocab products, or terms).
11. Synonyms of terms found. Metal is top level, then hard metal, then an esoteric name for a hard metal. If the parser finds the esoteric name, we could create a hierarchy, go up the stack, and replace the word with something more generalized. Can also replace classified terms.

## 3.2 Use Cases

### UC1 Story:

John is a pilot for Boeing. He already knows the basics of piloting a 737, but he wants to refresh his knowledge on the take-off procedures. He opens up the 737 manual, and is met with a 200-page document going over every detail there is to know about the aircraft.

John decides to use our tool to summarize the document. John uses our tool to parse all the nouns in the manual. He then creates categories based on the key features of flying the 737 (which he knows because he is an expert in this field). One category is “take-off flight controls”, one is “take-off procedures”, and another is “landing procedures”. He then creates relationships between these categories to link them together.

John now searches for the information he wants. He easily sees all the terms inside the “take-off procedures” category. One of the terms he finds is “lock”, and he wants to see the context that “lock” is in. Our tool allows him to see that “lock” in this context is referring to the procedure of locking his seat belt.

John also looks at the categories that are related to “take-off procedures”. He finds “take-off flight controls” is related by a relationship. He looks at the terms in that category, and is able to find the control he is looking for.

Our tool allows John to quickly look over the relevant information in the document, without having to read through the 200-page document.

### UC1 Source:

Boeing’s 737 flight manual originates this story.

### UC2 Story:

Sarah is a marketing scientist at Amazon. She knows that a user’s search results give information about their interests. She wants to find out what item a particular user might be interested in based on their past searches.

Sarah uses our data science tool to help her see if there are any patterns with this user’s interests. She runs our tool on their search history and parses the nouns. She then sorts the terms by their frequencies, and chooses the ten most frequent. She then categorizes these terms. She finds that this user’s most frequently searched terms can be categorized into three categories: camping, rock climbing, and outdoor cooking supplies. Now Sarah knows exactly what interests this user has.

Sarah then uses our tool on documents that are related to camping supplies. She follows the same process, and she finds that camping supplies can be categorized into tents, sleeping bags, and fishing. In each category, she can sort by frequency, and find the most relevant terms. She uses these terms to offer products to the user that would be relevant to them.

### UC2 Source:

Targeted ads originates this story.

### UC3 Story:

Terry is a data scientist at Google. He wants to improve search results to include more relevant results for a particular user.

Terry runs our tool on this user's search results. He finds that the user searches for soccer frequently. Terry then finds the soccer related searches can be divided into three categories: "soccer balls", "Real Madrid", and "World Cups". Terry looks at the context of the search results for Real Madrid. He finds that the user looks up Real Madrid, but only with context in the time period 2000-2004.

Using this context, the next time the user looks up Real Madrid, Google will return search results of Real Madrid during the time period 2000-2004 first. This way, Google can learn the context of a user to give better search results.

### UC3 Source:

Google search engine originates this story.

### UC4 Story:

Kate is a maintenance staff member working for Splunk. She keeps track of two different companies, one handles real estate, and another handles lumbar. She receives different parsing messages from each vendor indicating their status.

Kate uses our tool on the parsing messages for both companies. Even though the two companies are very different, our tool is flexible and can handle it. Each parser pulls out different vocabularies for the two companies. She can use the vocabularies as a baseline for each company. The terms contain context, so she knows exactly what the messages are referring to (in context to each company). Kate then creates categories for these messages, and relationships between them.

Kate now needs to integrate new devices for the two companies. She runs our tool again after the integration, and compares the new parsing messages to the baseline. If she finds any discrepancies, she can look at the context to find out where exactly the problem is. She can also see which category the discrepancy is in, and find if this might be a problem in one of the related categories.

Kate also uses the baseline multiple times to check for security risks. If for some reason, the baseline changes, she can compare the old baseline to the new one and find out why there is a change. This change might be an error, or it could be malicious code causing the parsing messages to differ.

### UC4 Source:

Splunk IIoT maintenance staff originates this story.



### 3.3 Functional Requirements

Below are the Functional Requirements of our application:

#### FR1 Parse Nouns and Noun Phrases:

- Requirement: The system must be able to parse through text documents and pull out nouns and noun phrases.
- Source: Mentors originated this requirement. Parsing is a main function of the app.
- Priority: *level 0*: Essential and required functionality.

#### FR2 Parse Multiple Documents and Document Types:

- Requirement: The system must be able to parse through multiple documents and document types (.docx, .txt, and .pdf).
- Source: Mentors originated this requirement. The application must be flexible and scalable.
- Priority: *level 1*: Desirable functionality.

#### FR3 Select Which Files to Parse:

- Requirement: The system must be able to choose which files to parse.
- Source: Team originated this requirement. The application must be interactive.
- Priority: *level 1*: Desirable functionality.

#### FR4 Enter Input and Output Folder Locations:

- Requirement: The system must be able to change the input and output folder location.
- Source: Team originated this requirement. The application must be interactive.
- Priority: *level 1*: Desirable functionality.

#### FR5 Enter Taxonomy Name:

- Requirement: The system must be able to change the name of the taxonomy.
- Source: Team originated this requirement. The application must be interactive.
- Priority: *level 1*: Desirable functionality.

#### FR6 View Context of Terms Parsed:

- Requirement: The system must be able to display the term's frequency, weight, sentences, and file location.
- Source: Mentors originated this requirement. The application must integrate context with the data.
- Priority: *level 0*: Essential and required functionality.

#### FR7 Pageable Tables:

- Requirement: The system must have tables which use pagination.
- Source: Team originated this requirement. The application must be efficient and scalable.
- Priority: *level 1*: Desirable functionality.

#### FR8 Categories:

- Requirement: The system must be able to create categories and add/ remove terms from those categories.
- Source: Mentors originated this requirement. The application is interactive and creating categories is one of the main functionalities of the app.
- Priority: *level 0*: Essential and required functionality.

FR9 Taxonomy Graph:

- Requirement: The system must be able to visualize in a graph the nodes (categories).
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 0*: Essential and required functionality.

FR10 Taxonomy Edge Types:

- Requirement: The system must be able to create edge types (relationships).
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 0*: Essential and required functionality.

FR11 Taxonomy Edges:

- Requirement: The system must be able to create edges (relationships) between nodes (categories).
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 0*: Essential and required functionality.

FR12 Taxonomy Delete Edges:

- Requirement: The system must be able to delete edges (relationships) between nodes (categories).
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 0*: Essential and required functionality.

FR13 Taxonomy Nouns in Nodes:

- Requirement: The system must be able to display nouns in nodes (categories)
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 1*: Desirable functionality.

FR14 Taxonomy Edit Edges:

- Requirement: The system must be able to edit edge types (relationships)
- Source: Team originated this requirement. The application is interactive.
- Priority: *level 1*: Desirable functionality.

FR15 Taxonomy Screenshot:

- Requirement: The system must be able to take a screenshot of the relationships graph.
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 1*: Desirable functionality.

FR16 Delete Terms:

- Requirement: The system must be able to delete terms from parsed terms.
- Source: Team originated this requirement. The application is interactive and bad data sometimes gets parsed and needs to be removed.
- Priority: *level 1*: Desirable functionality.

#### FR17 Save to CSV:

- Requirement: The system must be able to save any data created in the front-end into a CSV.
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 0*: Essential and required functionality.

#### FR18 Load from CSV:

- Requirement: The system must be able to load data from a CSV into the front-end.
- Source: Mentors originated this requirement. The application is interactive and visualizes the data.
- Priority: *level 1*: Desirable functionality.

#### FR19 Deploying with Docker:

- Requirement: The system must be able to containerize the service for easy use by others upon deployment.
- Source: Team originated this requirement. We want our program to be easily accessible.
- Priority: *level 1*: Desirable functionality.

### **3.4 Non-Functional Requirements**

Below are the Non-Functional Requirements of our application:

#### NFR1 Scalability:

Our product should be able to scale to handle increasing the document size and the amount of documents.

#### NFR2 Appearance:

Our product should have a clean and streamlined interface. The user should be able to intuitively understand how to use our program because of the UI design choices.

#### NFR3 Reliability:

Our product should produce reliable and consistent results. The results shouldn't change if we run the program multiple times on the same input.

#### NFR4 Accuracy:

Our product should produce accurate results. Our parser should only parse nouns and noun phrases, and return them as cleaned text.

#### NFR5 Reusability:

Our product should be able to be reused. We should have the ability to load and save from .csv, so a user can stop or make edits at any time.

#### NFR6 Python:

Our product should use Python in the back-end to run the main functionality of our app.

#### NFR7 JavaScript:

Our product should use JavaScript to create and render the front-end.

NFR8 Flask:

Our product should use Flask routes to pass information from the front-end and back-end.

NFR9 SpaCy:

Our product must use the SpaCy Python library as our main document parser. It is the industry standard for natural language processing.

## **4.0 Software Design**

### **4.1 Architecture Design**

Below is the final design of our software system:

#### **4.1.1 Overview:**

We used a Pipe-and-Filter architecture approach when creating our program. We begin with data (parsed from the documents), which gets filtered as it goes down the pipeline. The end result is data that has been categorized with relationships created between the categories. The final result is then saved into a CSV and .json file.

*[more information can be found in Appendix A]*

We also use Client-Server architecture in our design. Since our program is being run through a web app, the user can make requests in the front-end, that are then passed through a Flask route and run in the back-end. The results from the back-end are then sent back to the front-end as a response.

*[more information can be found in Appendix B]*

#### **4.1.2 Subsystem Decomposition (Pipe-and-Filter):**

We first decompose and explain our pipe-and-filter architecture below:

##### **Document Corpus (Load):**

- Description: This is the first step in the pipeline. The main functionality is to specify where in the database the taxonomy is, and which taxonomy should be loaded into the app. This information will be saved in the app's main state for later use.
- Services Provided:
  - SelectOutputFolder(): There is an input text box to enter the output folder location (this will be where the user chose to output the taxonomy when they created it). This location will be used in Term Extraction to locate which folder the taxonomy is in.
  - SelectTaxonomyName(): There is an input text box to enter the taxonomies' name (needed if the user created multiple taxonomies and saved them in the same location). This name will be used in Term Extraction to select which taxonomy to load.

##### **Document Corpus (Create):**

- Description: This is the first step in the pipeline. The main functionality is to specify the input and output locations of the taxonomy. The user will select which files to parse and the name of the taxonomy (file name when saving).
- Services Provided:
  - SelectInputFolderLocation(): There is an input text box to enter the input folder location. This should be the location of the text documents the user wants to parse.

- SelectInputFiles(): The app will look into the input folder location and return all the document files. These files are displayed in the front-end, and the user has buttons to either add them to the files to parse, or to remove them.
- SelectOutputFolderLocation(): There is an input text box to enter the output folder location. This should be the location the user wants to save the taxonomy and parsed documents.
- SelectTaxonomyName(): There is an input text box to enter the name of the taxonomy. This is the name of the CSV that will contain the parsed terms and context from all the documents. The CSV will also contain the categories the terms are added to (Category Creation). This name will also be used to name the relationships file (Taxonomy).

#### Term Extraction (Load):

- Description: This is the second step in the pipeline. The main functionality is to load the taxonomy information that was saved to the database into the front-end. The user can view and delete terms they don't want to be included in the taxonomy.
- Services Provided:
  - LoadTaxonomy(): When first entering this page, the app will automatically use the output location and taxonomy name from the app's main state (Document Corpus) to load in the terms, context, categories, graph, and edge types. In the back-end, we use functions that read through our CSV and JSON files, and then populate the app's main state with this information. The user can then click the Refresh Terms button to reload the terms from the back-end.
  - RemoveTerms(): The user might still want to remove terms from the taxonomy at. The user can select a term, and click the Delete Terms button. This will remove the terms from the app's main state and no longer display them in the front-end. These changes will not be stored in the database until the Save Terms button is clicked.
  - SaveTerms(): This will send the terms/ context from the app's main state to the back-end. The back-end will rewrite over the taxonomy in the database without the removed terms.

#### Term Extraction (Create):

- Description: This is the second step in the pipeline. The main functionality is to run the parser and calculate the weights in the back-end. The user can delete terms they don't want to be included in the taxonomy.
- Services Provided:
  - RunParser(): When first entering this page, the app will automatically use the input location, output location, and selected files from the app's main state (Document Corpus) to run the parser. In the back-end, for each document, the parser will find all the noun and noun phrases, their frequencies, sentences, and the file they belong to. For each of these documents, a new CSV file will be created in the output folder location (uses the same name as the document name that was parsed). There will also be a master taxonomy CSV created, that combines all the document's CSV into a single taxonomy (uses the taxonomy name). Terms that are found in multiple documents are appended together into one row (add their frequencies and sentences together).

- CalculateWeights(): This is run immediately after RunParser() is called. The back-end has functions that calculate the weight of each term in the master taxonomy .csv by dividing the frequency of the term by the number of documents parsed.
- RemoveTerms(): This is the same as RemoveTerms() in the Term Extraction (Create) section.
- SaveTerms(): This is the same as SaveTerms() in the Term Extraction (Create) section.

### Category Creation:

- Description: This is the third step in the pipeline. The main functionality is to manually categorize the terms. The user should be an expert in the constrained vertical of the Taxonomy. The expert can make accurate categories and decide which terms go inside them. These categories will be saved in the app's main state, and will later be used as the nodes in the Taxonomy graph.
- Services Provided:
  - CreateCategory(): There is a Create New Category button which will open a modal to select the name of the category. The category name is unique and is added to the app's main state.
  - DeleteCategory(): When the user clicks on a category, the category table (right side of screen) is replaced with a catTerms table displaying the terms in the selected category. A Delete Category button will appear and when it is clicked, the category should be deleted, and the terms that were inside the category should be moved back into the terms table (left side of screen).
  - MoveTermsToCategory(): When the user clicks on a category, a button with an arrow pointing right will appear. After the user selects terms in the terms table (left side of screen), when the button is clicked the terms should be moved out of the terms table and moved into the catTerms table. The term should also be assigned a new parameter, which is the category it belongs to.
  - RemoveTermsFromCategory(): When the user clicks on a category a button with an arrow pointing left will appear. After the user selects terms in the catTerm table (right side of screen), when the button is clicked the terms should be moved out of the catTerms table and back into the terms table. The term should also have the category parameter removed from the app's main state.
  - SaveCategories(): There is a Save Categories button. When clicked the categories are sent in a request to the back-end. The back-end has functions to handle appending each category to the terms in the master taxonomy CSV. At this point, the master taxonomy CSV is finished (unless the user goes back to a previous step in the pipeline and saves).

### Taxonomy:

- Description: This is the fourth and final step in the pipeline. The main functionality is to create relationships between the categories (Category Creation).
- Services Provided:
  - CreateEdgeType(): There is a Create New Edge Type button. When clicked a popup appears to choose a name (relationship name) and select a color. The edge types and their colors are displayed in the edges table as a key for the graph.
  - AddEdge(): When the user selects two nodes while holding the ctrl key (key bind for multiselect), the Add Edge button will become enabled. When this button is clicked, a popup is shown with a drop down menu to select an edge type. The graph will be updated with an arrow pointing from the first node to the second node. The arrow will have the same color as the edge type.
  - DeleteEdge(): When the user selects an edge (relationship), the Delete Edge button will be enabled. If the user clicks this button, then the edge will be removed from the graph.
  - TakeScreenshot(): There is a Take Screenshot button. When clicked, a .png will be downloaded to the browser. The screenshot will contain the graph and the edge types table. This will be useful for users to use as visualization.
  - EditEdge(): When the user clicks on an edge type in the edges table, a popup will appear where the user can update the name and color of the edge type. This will be reflected for every edge in the graph that uses that edge type.
  - DeleteEdgeType(): When the user clicks on an edge type in the edges table, a modal will appear with a Delete button. If the button is clicked, the edge type will be removed from the edges table and every edge in the graph that uses that edge type will be removed.
  - SaveRelationships(): There is a Save Relationships button. When clicked, the graph and edge types will be sent as a request to the back-end. The back-end will create a .json file containing all this information. The name of the file will be the taxonomy name + “\_relationships”.

### **4.1.3 Subsystem Decomposition (client-server):**

Below we decompose and explain our client-server architecture:

#### Front-end (web app):

- Description: This is where the user will interact with our app. Functionality will be accessed by buttons, text input boxes, or graphs. The functions created for this section deal with manipulation and visualization of data in the front-end. All other functionality will be called by requests sent with Flask routes to the back-end. The back-end will respond with responses, which will then be brought into the app's main state and displayed in the front-end.



#### Back-end (server):

- Description: The functions created for this section deal with parsing, weight calculation, loading, and saving. There are also helper functions created to help with the main functions. For example, there are helper functions to clean text when the Parser is being run. There are also helper functions to read and write to and from CSV and JSON files when Load or Save are being called.

#### Database (computer):

- Description: This will be where the data is stored. The text documents to be parsed are located here. This is also where the master taxonomy CSV, individual file CSV, and the relationships/ graph .json will be saved.

## **4.2 Data design**

There are three aspects to the data design of this project: (1) The front-end app's main state, (2) The back-end requests and responses, and (3) The data that is saved in the database. In this section, we will describe the data structures for each.

#### Front-end (web app):

- Data Structures:
  - filesList{}: On the Documents page, there is a dictionary which contains the files that were found in the input folder location. The keys are the document names, and the values are the document file extensions.
  - selectedFiles{}: We needed a separate data structure for files the user has selected to run the parser on. All the files found in the input location should always be displayed, but only the selected files are highlighted. This has the same structure as filesList, for consistency.
  - termDictionary{}: On the Terms Extraction page and beyond, this is the most important data structure. Each key is a term, and the value is another dictionary.

This second dictionary will have four keys, one for context, frequency, weight, and a category (if the term has been added to a category). The values for frequency and weight are doubles calculated in the back-end.

The value for context is a list. There will be pairs of items in this list. The first item in the pair is the name of the file the sentences are found in. The next item in the pair is a list containing all the sentences the term is found in.

The term will only have a category key if the term has been added to a category. The category key is important so we could choose which terms to display in the terms table when editing a category. We did not want to remove terms from the termDictionary, as this should be a single point of truth throughout the app. Instead, by adding a category to the term in the termDictionary, we then can check if there is a category key for a term, and then not display that term in the terms table. In practice, we can remove terms from the term table, without removing the term from the termDictionary. When the user clicks Save Terms, this will be the data structure that is sent to the back-end.

- categories{}: On the Category Creation page, there will be a categories dictionary which will contain the categories created, and the terms that are in these categories. The key will be the category name, and the value will be another dictionary. This second dictionary will have a key for each term, and the value will be the term's context. When the user clicks Save Categories, this will be the data structure that is sent to the back-end.
- edgeTypes[]: On the Taxonomy page, the edge types list will contain the relationships that the user has created. Each item in the list will be a dictionary, with edge type name as the key, and color as the value. This will be used to fill out the edge type table. When the user clicks Save Relationships, this will be sent to the back-end and saved into a JSON file.
- graph{}: On the Taxonomy page, the graph will contain edges and nodes that are displayed in the graph. There is an edge and node key in the graph dictionary. The value for the edges is a list of dictionaries. Each dictionary contains the color of the edge, which node it comes from and goes to, and the name of the edge type. The value of the nodes is also a list of dictionaries which contain the color of the node, the node ID, and the label of the node. This structure will also be sent back when the user clicks Save Relationships, and it will be saved in the same JSON file as edge types.

#### Back-end (server):

- Data Structures:
  - request: The back-end will always receive a request, which is a JSON object containing information that the back-end functions will need as parameters. The request is accessed like a dictionary, and the values are then passed to the back-end functions.
  - response: The back-end will send a response to the front-end after the back-end functions are finished running. The data is also a JSON object. The front-end accesses the response like a dictionary, and sets the app's main state with the information.

#### Database (computer)

- Data Structures:
  - .docx: This is one of the file types that our parser can parse through. This would be a file inside the input folder location.
  - .txt: This is one of the file types that our parser can parse through. This would be a file inside the input folder location.
  - .pdf: This is one of the file types that our parser can parse through. This would be a file inside the input folder location.
  - CSV: All information on the terms (context, frequency, weight, and categories) will be written inside a CSV. The load functions can read CSV files.
  - JSON: All edge types and graph information (nodes and edges) will be saved inside a JSON. The load functions can read JSON files.

### 4.3 User Interface Design

In this section, we go over the design inspirations we used to create our web app. Most of our ideas come from the same article by B. Carrion and team. Below we will point out some of the similarities you can find in our app and theirs.

On the first page, a user can upload documents to parse. Each of these documents are listed with a button to delete it. The buttons are color coded to signify their function (ex. red for delete). There is also a navbar that shows which step in the pipeline you are currently viewing.

*[more information can be found in Appendix C]*

In the second page (term extraction), they paginate their term table. They also display in the table each term's frequency and weight.

*[more information can be found in Appendix D]*

In the third page (categories page), they divide the page into 3 columns. The leftmost column is categories, the middle column is buttons, and the rightmost column is the terms. You can see the terms within each category. In our design, we flipped the order, and moved the terms to the leftmost column. This made more intuitive sense, since you start with the terms table and that should appear first on the page.

*[more information can be found in Appendix E]*

Their taxonomy page is different from ours, but the design choices are the same. They used a zoomable circle packing visual, while we used a graph with nodes and edges. However, both designs include some sort of visualization of the data.

*[more information can be found in Appendix F]*

We go into more details about the design choices specific to our web app in section VI (Description of Final Prototype).

## **5.0 Test Case Specifications and Results**

### **5.1 Testing Overview**

Our testing plan was split into two sections; frontend and backend tests. With the backend we wanted to make sure each module worked as intended. We also tested the full capabilities of our backend with integration tests. For the frontend, we created end-to-end tests using Test-cafe, targeting common user operations.

The backend contains the application logic that delivers a dictionary of parsed nouns, context and frequencies. There are three main modules that produce this outcome:

- Text extraction module
- Spacy module
- Read-write module

We created unit tests for each of these modules. Since we are testing the modules individually we need to provide test data each module was expecting. The text extraction module required a supported file type (DOCX, PDF, TXT). To assert the output was true we need to provide an array that contains a string of the expected text. The spacy module also required a supported file type. To test the accuracy we need to provide a dictionary with noun terms as the key and a dictionary value that contains an array of context and frequency. Finally for the read-write module we need to provide a file as input and an array emulating a CSV row or a dictionary.

For the text extraction module we needed to test if it could handle our three supported file types. Each test consists of a call to the `[file type].get_text(file)` method. The method should correctly discern the type of file the input is and return the extracted text as an array. We use the `assert` keyword to make sure that is the case. There is a test case for each supported file type and a corresponding invalid test case where the input is of the wrong type.

For the Spacy module we needed tests to see the accuracy of the parser. Each test consists of a call to the spacy module's `.get_terms(file)` method. The method returns the parsed documents noun, context and frequency as a dictionary. These sets of tests do not contain `assert` calls since accuracy isn't something that is strictly enforced due to the nature of parsers. However, we log the output of our accuracy on every test run to see if major changes to the metric have occurred.

Lastly, the rewrite module. This is the simplest of the three where we test to see if the creation of a file in the specified location is true. We use the `pathlib` library to assert the existence of a file after a call to the `write_to_csv(file)` and `write_to_json(file)` methods.

The frontend contains all visual aspects of this application. Each component needs to be tested. The components are:

- Create-Load page
- Document page
- Terms page
- Categories page
- Taxonomy page

For the documents page there are two tests. The first one tests if the user is able to successfully type in the input fields for input location, output location and corpus name. Then if they are able to click on the 'Get Files From Input Folder' button. Finally if the button press populates a table with all the files in the folder. The second test is an invalid test. This test inputs data that should not work and will not produce a table of documents after the button click. It asserts that no change has been observed.

For the terms page there are multiple tests. The first one tests whether the terms parsed from the selected corpus populate the table in the middle of the screen. The second test case asserts that given an invalid input folder the table will not populate with values. Next the sentences modal test case checks to see that the user can click on the terms 'Sentences' button. This button should open a popup modal that shows the sentences the noun can be found in. Finally the last test case examines if a term in the table can be deleted.

For the categories page there are multiple tests. The first one tests if the user is able to create a category. This is a multi-step process that requires a modal popup and table insertions. We assert that the modal has appeared and the user is able to input a category name. Then we assert that the last value in the table is the newest category addition. Finally we make sure the name of the category matches the name passed into the modal. The next test case is deleting a category. First we assert that clicking a category changes the available buttons in the 'Edit Categories' center panel. Then we check to see that clicking the 'Delete Category' button removes the selected category from the table. The last two tests focus on the users ability to move terms in and out of a category. We assert that clicking on a category produces a list of clickable terms. Depending on the user's choice we see that clicking on the right arrow button in the center panel reduces the count of terms in the category table and increases the count in the terms table. Vice versa for a click on the left arrow button.

The final test set is for the taxonomy page. The first test checks to see if the user is able to create a relationship type and color. The test asserts a click on the 'Create New Relationship' button opens a pop up modal. This modal allows the user to input a relationship name and color type. The values entered in the modal should correspond with the key value pair added to the relationship table. The next test emulates a user editing the name and color of a relationship in the table. Clicking on the relationship should open the same popup modal as creating to update the values. Another relationship test checks to see if deleting a relationship removes the entry from the table. Finally a test case to see if clicking the screenshot button produces a downloadable .png of the graph visualization.

The last section of our testing plan contains the implementation of a GitHub action workflow that automates the execution of our test cases. We have two .yaml files within the main branch of the project that runs our tests on every push to the branch. We have the option of running the workflow manually in the actions tab as well. The tests run on a Ubuntu server provided by GitHub that requires a fresh build before every test.

## 5.2 Environment Requirements

To run our backend the first technology we need to have installed is python 3.8 or above. This gives us the option to install the pip package manager. From there we would need to have installed all the packages listed in requirements.txt. These files include, spacy for the parsing capabilities, CSV to save noun information and Flask to name a few. Now that we have all the packages required for running our code we need to install the backend test runner, PyTest. With PyTest installed we can run all the tests that reside in our tests directory in a single command.

To run the frontend we first need to have installed node.js 16.13 and above. This gives us the option to install the npm package manager. From there we would need to install the dependencies listed in package.json. The most important dependencies would be react, vis-react and bootstrap. Once all the dependencies are installed we need to install the frontend test runner Test Cafe. With Test Cafe installed we can run all the tests that reside in our Website/tests directory in a single command.

In terms of hardware, the application and tests were run on Windows 10. This application can run comfortably on 8 GB of RAM. The storage space requirements depend on the user's objective, however a single document can produce up to 3 extra documents after a parse with graph visualizations. Users will require a browser and a terminal to start servers.

### 5.3 Testing Results

The test results for the backend show that all main modules are working as intended. The spacy module still needs work when input files are .pdf. This is due to pdf text extraction being a difficult non-standard task. We suspect that .pdf files are not all encoded with the same assumed UTF-8 encoding. When we open a .pdf file using pdfplumber we get terms with hex values that, upon further investigation, correlate to punctuation marks like commas and apostrophes. We conclude that using plain text documents produce the most accurate results with .docx files coming in second. Running the parser on documents around a page long showed speeds of up to .05 second per doc *[more information can be found in Appendix AB]*. We also observed that documents with lengths around 80 showed speeds of 14 seconds. These speeds hold for multi document parsing when using asynchronous multiprocessing. The parsing module can handle running up to 5 documents at a time without draining too many resources on the host machine. Each parse runs at the same base speeds of a single parse. *[for more information see Appendix AC for a full pytest run ]*

## **6.0 Projects and Tools Used**

### **6.1 Libraries, Tools, and Frameworks**

Visual Studio Code	The development environment we used to write our code.
Git Bash	Shell environment we used to run servers (front-end and back-end).
React.js	JavaScript library used to build our user interface.
GitHub	Version Control we used to manage our project.
Discord/ Webex	Team/ mentor communication platforms.
Flask	Web framework we used to connect back-end to front-end (using requests and responses).
Font Awesome	Free font and icon toolkit.
Bootstrap	Free front-end framework for UI templates.
Download.js	Allows JavaScript to download to a browser directory.
html2canvas.js	Script that allows screenshots of DOM elements
react-step-progress-bar.js	Library to create custom progress bars (used in navbar).
react-vis.js	React visualization library (used in Taxonomy graph).
spaCy.py	Free library for advanced Natural Language Processing.
pdfplumber.py	Python library to plumb a PDF for information.
Docker	Containerization for app deployment.

### **6.2 Languages**

Python	CSS	JavaScript	HTML
--------	-----	------------	------

## **7.0 Description of Final Prototype**

### **Database (Early):**

This is how we structure our project at a high-level. There are five main folders: Data, Parser, Taxonomy, Tests, and Website.

*[more information can be found in Appendix G]*

The Data folder is the database, and has functions dealing with reading and writing .csv and .json files. You can also see that this is where the input and output folders are located.

*[more information can be found in Appendix H]*

The Parser and Taxonomy folders are the back-end, and not something the client should have to worry about. The Parser folder has functions for running the parser and cleaning the text. The Taxonomy folder has functions for calculating weights, and reading and writing the graph and category information.

The Website folder is the front-end and what the client interacts with. It has functions for manipulating and visualizing data in the front-end. You can also see that the Website folder is further broken down into three subfolders: Components, Images, and Styles.

Inside the Components folder are the pages and rendering logic. Inside the Images folder are any images to be included in the front-end (a team logo in the navbar for example). In the Styles Folder, there are CSS files for each component in the front-end (including modals). Each component has its own CSS file to improve the code's organization.

*[more information can be found in Appendix I]*

Taxonomy, Data, and Parser each have their own main.py file. The front-end will only call functions inside this file, and all helper functions will be called from there.

Finally, the Tests folder has test cases and test data for our project.

### **Step 0: Create or Load a Taxonomy**

This is the first page in the web app. The user has two options. The top button begins the process of creating a new taxonomy. The bottom button loads a taxonomy that has previously been created. Depending on what the user clicks will affect the app's main state and which Document Corpus page to load.

*[more information can be found in Appendix J]*

### **Step 1: Document Corpus (Create)**

This page is loaded if the user clicked the button to create a new taxonomy. The page can be broken down into top half and bottom half.



The top half of the page contains input text boxes for the input location, output location, and taxonomy name. Each input box has a corresponding submit button which when clicked sets its information into the app's main state. After the submit button has been clicked, the corresponding app's main state information will be displayed underneath their text input box.

The default name for the taxonomy is "master". The taxonomy name must also be unique, and if there is already a taxonomy with the same name in the output location, that file will be overwritten.

The bottom half of the page displays the files to be parsed (from the input location). If the user clicks the Get Files button and the information is entered correctly, then the documents found in the input location will be displayed in the files table. Each row in the files table will be a document, and the document's file type.

Each row will also have a button to either parse the document, or remove that document from the documents to parse. The button will either be labeled Add or Delete, depending on if the document has been added to parse. If a document has been added, then the row will be highlighted blue, and the name will be bolded.

There will also be two buttons on the bottom of the page. These buttons will consistently be at this location from this page forward. The Back button will take you back one step in the pipeline. The Forward button will take you one step forward in the pipeline.

*[more information can be found in Appendix K]*

### Step 1: Document Corpus (Load)

This page is loaded if the user clicked the button to load a taxonomy. There are only two text input boxes.

The first input box is to enter the output location. The label makes more sense as the input location, however, we wanted to keep the locations consistent between loading and saving. Since you save the taxonomy in the output location, then you also load from the output location.

There is also an input box for a taxonomy name. This will tell the program which taxonomy to load if there are multiple taxonomies saved in the same location.

*[more information can be found in Appendix L]*

### Step 2: Term Extraction

This is the second step in the pipeline. From here, either the parser is run (creating a new taxonomy), or loadTaxonomy() is called (loading from a taxonomy). Either way, the UI will look and function the same.

*[more information can be found in Appendix M]*

If the user is creating a new taxonomy, and they click the Refresh Terms button, then the parser will be run in the back-end. After the parser is finished, the weight calculator will

be run in the back-end. The information for the parsed terms and their weights will then be returned in a response to the front-end and appended together into the terms dictionary in the app's main state.

After refreshing the terms, only one thing is saved in the database. When running the parser, each individual document parsed will have a mini CSV file created with the terms that were parsed. These CSV files are later appended together into the master taxonomy CSV. Only these mini CSV files are saved in the database when the parser is run.

If the user is loading a taxonomy, and they click the Refresh Terms button, then the `loadTaxonomy()` function will be called in the back-end. The back-end will have functions for finding and reading the taxonomy CSV and the corresponding graph JSON. This information is returned to the front-end in a response, and saved to the app's main state the same way as create.

To save the master CSV (if this is the first save, then this also creates the master CSV) or any changes made to the terms list (after deleting terms out of the terms table), the user must click the Save Terms button. The term dictionary will be sent as a request to the back-end, and the back-end will run functions that will convert this information into the master taxonomy CSV.

Like the file table from the previous page, each term is clickable, and the row will be highlighted when selected. There are two buttons that will be enabled when a row has been selected. The user can either click Clear Selected, which will unselect all the currently selected rows. The user can also click the Delete Terms button, which will remove the terms from the app's main state terms dictionary. This change will be reflected in the front-end but will not be saved in the back-end until the user clicks Save Terms.

Each term will also have a button labeled Sentences. When this is clicked, all sentences the term is found in and the document the sentences are found in will be displayed in a modal.

*[more information can be found in Appendix N]*

All future tables will also be paginated from this point onward. There are two buttons directly below the terms table. If Next: is clicked, then move forward one page in the table. If Previous: is clicked, move back one page. Disable the buttons if you cannot go backward or forward any more pages.

### Step 3: Category Creation

This is the third step in the pipeline. From here, an expert can create categories and organize the terms into those categories. You need an expert of the corpus that the taxonomy is using in order for the categories to be effective. An expert will use their knowledge and experience to figure out what categories to create, and which terms to add.

The leftmost section will always display the terms in the term dictionary. If a term has been added to a category, then we remove the term from the terms table. If the term is removed from a category, the term will then be redisplayed in the terms table.

The rightmost section will display a table of the categories that have been created. If a category is selected in this table, then the whole section is replaced with a new table that only displays the terms from the selected category.

There are three buttons that can be seen in the middle section. The expert can either Create New Category, Clear Selected, or Save Categories. These buttons are different if the user has currently selected a category from the categories table.

*[more information can be found in Appendix O]*

If the user clicks Create New Category, then a modal will appear asking the expert to name the category. The name must be unique. After the expert confirms the name, the category will then be added to the app's main state category dictionary. The category will also show up in the category table (rightmost section). Each entry in the category table will include the name of the category, and a scrollable table showing the terms in the category.

*[more information can be found in Appendix Q]*

Each term in the terms table is selectable and highlightable, the same as previous sections. The Clear Selected button will also work the same way as previous sections. If an expert has selected terms that are inside a category, then the Clear Selected button will unselect these too.

Save Categories also works similarly to previous sections. The changes in the front-end are only reflected in the database after the expert clicks the Save button. The categories are passed to the back-end as a request. The back-end then has functions that append to rows in the master CSV terms that are now a part of a category. If a row is in a category, the row will have an extra column with the category name the term belongs to.

If the expert clicks on a category in the categories table, then the categories table will be replaced with a terms table that displays that category's terms. The terms in this table are selectable just like the terms in the terms table.

The middle section will have new buttons, Exit Category, Delete Category, Arrow Pointing Left, and Arrow Pointing Right.

*[more information can be found in Appendix P]*

The Exit Category will return the user back to the original Category Creation screen, with the categories table in the rightmost section. The original buttons will also return.

The Delete Category button will remove the category from the app's main state categories dictionary. The terms that were in the category will be moved back to the terms table.

The Left Pointing Arrow button will move terms from a category back to the terms table.

The Right Pointing Arrow button will move terms from the terms table to the category.

#### Step 4: Taxonomy

This is the fourth and final step in the pipeline. From here, the expert can create relationships between the nodes (categories), visualize this data as a graph, and take a screenshot of the graph.

Similar to the Category Creation page, the Taxonomy page can be divided into three sections.

*[more information can be found in Appendix R]*

The leftmost section contains the visualization of the graph. The nodes (categories) and edges (relationships) between the nodes are displayed. The nodes are red and will have a label that corresponds to the category they represent. The edges will have a color that corresponds to the relationship they represent. In the top left corner of the graph, there is also a section that signifies what nodes or edges are currently selected. If the user clicks on either a node or edge, it is highlighted (slightly bigger font size) and will be displayed in the top left corner. The user can also hold the ctrl button on their keyboard to multi select nodes and edges. The leftmost section is slightly larger than it was on the previous page. The graph is the most important information on the page, so it should take up the most space.

The rightmost section contains the edge type table. This section will be used as a key for the graph. When an expert creates a new edge type (relationship), it is added to the app's main state edge type dictionary. This dictionary is displayed in this table. Each row contains the name of the edge type and the color. The color of the color label will change to match the actual color.

Each row is also clickable. When an expert clicks on a row, a modal to edit the edge's name and color appears. If the expert makes a change and clicks Enter, the changes will be made in the app's main state. The change will also be reflected in the graph. Any edges in the graph that were using that edge type will also be updated.

There is also a Delete button in the modal. If the expert clicks this, then the edge type will be removed from the app's main state. Any edges in the graph that use this edge type will also be removed from the graph.

*[more information can be found in Appendix U]*

The middle section contains buttons similar to the previous page. The expert can Create a New Relationship, Add Edge, See Nouns, Delete Edge, Take Screenshot, or Save Relationships.

When Create a New Relationship is clicked, a modal appears asking for the relationship's name and color. The name must be unique. When selecting the color, a color wheel will appear, and the color can be anything. Once the expert clicks Create, the relationship is added to the edge type graph, and displayed in the edge type table.

*[more information can be found in Appendix S]*

Add Edge is only enabled when exactly two nodes in the graph are selected using multi select. If the expert clicks the button, then a modal will appear with a dropdown menu. The dropdown menu will have options for the relationship types that have been created. The expert will select which relationship to add, and after clicking Add, the edge will appear in the graph between the two nodes. The edge is directed, so it will point from the first node clicked to the second node. The edge's color will be decided depending on which relationship was chosen from the dropdown, and the color of that relationship in the edge type table. This information will be added to the app's main state graph dictionary.

*[more information can be found in Appendix W]*

See Nouns is only enabled when exactly one node in the graph is selected. The expert wants to see the context of a category, but they shouldn't have to go back between screens. This button will allow the expert to see the terms in the category.

When the button is clicked, a modal will appear. The modal has the name of the category, and a table of all the terms that have been added to that category.

*[more information can be found in Appendix T]*

Delete Edge is only enabled when one or more edges have been selected in the graph. When the expert clicks this button, the edges that were selected will be removed from the graph. The app's main state graph dictionary will also have the edges removed.

When Take Screenshot is clicked, a .png file will be downloaded to the browser's download folder. The screenshot will contain the graph and the edge type table. The screenshot will be named relationships(x).png. This is a helpful tool to save work, and to visualize the results.

*[more information can be found in Appendix V]*

Finally, the Save Relationships button is similar to the prior save buttons. Nothing in the database has been saved, until this button has been clicked. Once this button has been clicked, then the graph and edge types will be sent to the back-end as a request. The back-end has functions to write this information into a JSON file in the database. The JSON file will be named taxonomy name + "\_relationships".

### Database (End):

We wanted to loop back around to the database to show what it would look like if a taxonomy has been created and saved. For this example, we are creating a games taxonomy. This taxonomy will contain only one document named Game\_types\_and\_genres.pdf. The taxonomy has been created and saved into the database.

We made a subfolder in the Data/Output folder named games. We had our program set the output folder location to Data/Output/games, so that is where the taxonomy and graph have been saved to. Inside this folder, there are three files that have been made.

*[more information can be found in Appendix X]*

The first file is Game\_types\_and\_genres\_nouns.csv. This is the mini CSV that is created when the program parses out the terms. If there were more documents to parse, then there would be more mini CSV files with the “\_nouns” label appended to the end.

The mini CSV contains some extra information that the master CSV will not. The first row will be the name of the document that this was created using. The next row will have information on the total number of nouns and the unique number of nouns found. The next row will have the time it took to parse the entire document, and the time it took to parse each noun on average.

The information following these rows are standard for all the CSV files. The first column will be the term. The next column is the sentences the term is found in. The last column is the frequency of the term found.

*[more information can be found in Appendix Y]*

The next file is Games.csv. This is the master taxonomy CSV. The taxonomy was named “Games” in the program, so the name has been reflected here. This CSV will contain all the information from the mini CSV files appended together.

There will no longer be extra rows containing parsing information. Each row will be a new term entry. The first column in the row is the term. The next column is a list that contains sentences and the documents the sentences come from. The third column contains the frequency. The fourth column is new and it contains the weight. The weight is calculated using the frequency and the number of documents parsed (in this case one). Finally, there might be an item in the last column. If the term has been added to a category, then the category name will be in this column. If the term has not been added to a category, then this column will be empty.

*[more information can be found in Appendix Z]*

The last file in the output folder is Games\_relationships JSON. This is a JSON object that contains information on the graph and the edge types. The graph will have two dictionaries, one for the nodes and one for the edges. These two dictionaries make up the graph. The edge types is a list of the relationship names and their colors.

*[more information can be found in Appendix AA]*

## **8.0 Product Delivery Status**

Our project will be delivered to our clients at Boeing no later than May 7th. We plan to be code complete a month before this deadline, giving us enough time to complete documentation and to run tests. We assume we will find bugs during the testing process, so we want to give ourselves time to fix as many of the bugs as possible.

We will give access to our github repository and the documentation that we have created over to our mentors. Our contact information will also be publicly available on our repository, so we can answer questions about the code in the future.

## **9.0 Conclusions and Future Work**

## 9.1 Limitations and Recommendations (L&R)

The following section describes limitations in our final product as-is. These limitations are categorized as one of three categories: functionality limitations, codebase centric, or frontend limitations. Following the description of each limitations denoted as L# are corresponding recommendations R# which describes how these limitations could be fixed. Note, these L&R's are not by any means prescriptive, we save that for Future Work.

### 9.1.1 Functionality L&Rs

L1: Category generation manually generated by the user.

R1: The process should be semi automatic meaning the application generates the categories however can be edited, deleted or created by the user as well.

L2: Taxonomy generation (relationship making) manually generated by the user.

R2: The process should be semi automatic meaning the application generates the relationships however can be edited, deleted or created by the user as well.

L3: Tables are currently unsortable by the user.

L4: Ability to name the screenshots as a desired filename not included.

L5: Undo and redo buttons not made available for the user to be able to undo or redo certain changes.

L6: Following front-end saving logic not present: that user

L7: Saving logic in front-end could not complete.

- L7.1: Popup modal that user hasn't saved yet.
- L7.2: Should save happen at the beginning at least once.
- L7.3: Logic for not saving all the pages, yet loading for all pages.

### 9.1.2 Codebase L&Rs

L1: Front-end error handling

- L1.1: Needs more modals to explain errors (unique names, etc).
- L1.2: Bad inputs/ outputs to the back-end.
- L1.3: Bad files in input location.
- R1.4: Change into an application, would make choosing folders and files ways easier.
- R1.5: Adding a database would be part of changing to application.

L2: Parser logic, running term multiple times in front-end.

L3: The parser accuracy is still lacking

R3: Dictionary api to filter bad inputs.

L4: Efficiency of the parser on large documents lacking.

- L4.1: Requisite Run time is very large over large documents, only so much reduced algorithmic complexity can do to fix this.



- *R4.1*: Multiprocessing in order to increase efficiency.
- *L4.2*: Parser skips over tables and pictures as they are currently unparsable.
- *R4.2*: Parser should handle tables and pictures.
- *L4.3*: Parser only can handle the English language which makes the program less extensible.
- *R4.3*: Parser should handle different languages.
- *R4.4*: Parser should handle more file types.

### **9.1.3 Frontend L&Rs**

*L1*: Currently node colors in relationships cannot be changed. This affects readability of the output.

*R1*: Change node color

*R2*: The UI could still use work, especially the CSS files when resizing the window

*R3*: Display terms in the nodes in taxonomy page (zoomable circle packing)

## **9.2 Future Work**

There are very many areas where the next group could contribute towards the end product. Note that many of these may stem from L&R's as described above. Currently, there are more than enough next steps for next year's team to work on for the next year, possibly even enough for the group after next. These will be encoded as Next-Step-# encoded with whether tasks are easy, hard, critical or non-critical.

*Next-Step-1*: Parser search functionality, can search a term and it finds the document and sentence.

*Next-Step-2*: Semantic analysis: We have the sentences, so now we should gain the ability to break sentences down into sentence structure (e.g. Noun, Verb phrase, etc).

*Next-Step-3*: Any and all recommendations in section 9.1 can be considered as future work. Especially the following:

- Machine Learning algorithm(s) to automatically create categories and relationships.
- Improved Frontend error handling for users.
- Parser accuracy and efficiency improvements.
- Undo & Redo buttons on each step to be implemented.
- Improved Saving and Loading.

# **10.0 Acknowledgements**

We want to thank our mentors from Boeing, Don and Rocky, for the help and instruction they have given us throughout this project.

We also want to thank our professor AJ for the guidance he has given during our senior year, and for his assistance in preparation for our future careers.

## **11.0 Glossary**

Constrained Vertical: A specific subset of writings pertaining to the same niche subject

Parser: A program that breaks down a string into logical components

NLP: Natural Language Processing, dealing with the interactions between computers and human (natural) language

GAN: Generative Adversarial Network, two Machine Learning neural networks competing with each other

Aboutness: The association of words, looking at their meaning in context to a sentence

IAM: Identity and Access Management, allowing corporations to manage digital identities and control user access

Rule Based System: A set of rules, an interpreter of the rules, memory, and user input

Tokenization: The substitution of data with an equivalent with no extrinsic or exploitable meaning

Stemming: The process of truncating an inflected word to its roots/stem

Lemmatization: The grouping together of inflected forms of a word so they can be analyzed as a single item

POS: Parts of speech are the categories of words based on the rules of the language

Regular Expressions: A sequence of characters for which a search pattern defines

Feature Engineering: A process in preparing and extracting relevant features from a raw dataset

Possessive Noun: A noun that shows ownership and possession in something

Conjunction: A subset of words that are used as a connector between other words, phrases, clauses, or sentences

Independent Clause: A clause (subject and verb) that can stand alone in a sentence; i.e. a complete thought

BERT: State of the art NLP model

Tensors: A mathematical object that describes multi-linear relationships between sets of mathematical objects. Generalization of scalars and vectors

GUI: Graphical User Interface. Allows a user to interact with an application through graphical means as opposed to text-based instructions.

API: Application Programming Interface. Allows two applications to communicate in a secure and reproducible way.

Python: High -level Interpreted program language heavily used in data science.

SpaCy: Industry-strength python library for Natural Language Processing.

Relational Database: Structured data store that uses tables, rows, columns and references.

MySQL: Open-source relational database management system.

Unit Testing: Testing the smallest piece of code that can be logically isolated in a system.

Integration Testing: Checking the interactivity of different sub-system combinations.

Semantic Relations: A relationship between two words that can express the same meaning.

Folksonomy: A classification system where the users can publicly tag items.

NER Tagger: Named Entity Recognition, to tag terms into categories.

Morpho-Syntactic: grammatical categories that have both morphological (structure and form of words) and syntactic (grammatical structure) properties.

Bayesian Hierarchical Taxonomy: A statistical model that is written in multiple levels using the hierarchical model and the Bayes' theorem to observe data and account for uncertainty.

Corpus: All the writings or works of a particular kind on a subject.

BRT: Data model that can be created using binary features and the Bernoulli distribution.

DCM: Multinomial and Dirichlet conjugate distribution.

GPO: A collection of Group Policy settings that defines what a system will look like and how it will behave for a defined group of users.

GTC: GPU Technology Conference

IP: Is the network layer communications protocol in the Internet protocol suite for relaying datagrams across network boundaries.

## **12.0 References**

“Introduction to natural language processing,” *GeeksforGeeks*, 16-Nov-2018. [Online]. Available: <https://www.geeksforgeeks.org/introduction-to-natural-language-processing>.

“Rule-Based Classifier - Machine Learning,” *GeeksforGeeks*, May 06, 2020. <https://www.geeksforgeeks.org/rule-based-classifier-machine-learning>

“The Current State of the Art in Natural Language Processing (NLP),” *ZappyAI*. <https://zappy.ai/ai-blogs/the-current-state-of-the-art-in-natural-language-processing-nlp>

“Machine Learning for Natural Language Processing,” *Lexalytics*, Nov. 25, 2019. <https://www.lexalytics.com/lexablog/machine-learning-natural-language-processing>

M.-S. Paukkeri, A. P. García-Plaza, V. Fresno, R. M. Unanue, and T. Honkela, “Learning a taxonomy from a set of text documents,” *Applied Soft Computing*, vol. 12, no. 3, pp. 1138–1148, Mar. 2012, doi: 10.1016/j.asoc.2011.11.009.

“Natural Language Processing - Machine Learning with Text Data | Pluralsight,” *www.pluralsight.com*. <https://www.pluralsight.com/guides/nlp-machine-learning-text-data>

“spaCy · Industrial-strength Natural Language Processing in Python,” *spacy.io*. <https://spacy.io>

“Difference between Unit Testing and Integration Testing,” *GeeksforGeeks*, May 14, 2019. <https://www.geeksforgeeks.org/difference-between-unit-testing-and-integration-testing>.

Codecademy, “What is a Relational Database Management System? | Codecademy,” *Codecademy*, 2019. <https://www.codecademy.com/articles/what-is-rdbms-sql>

B. Carrion, T. Onorati, P. Díaz, and V. Triga, “A taxonomy generation tool for semantic visual analysis of large corpus of documents,” *Multimedia Tools and Applications*, vol. 78, no. 23, pp. 32919–32937, Jul. 2019, doi: 10.1007/s11042-019-07880-y.

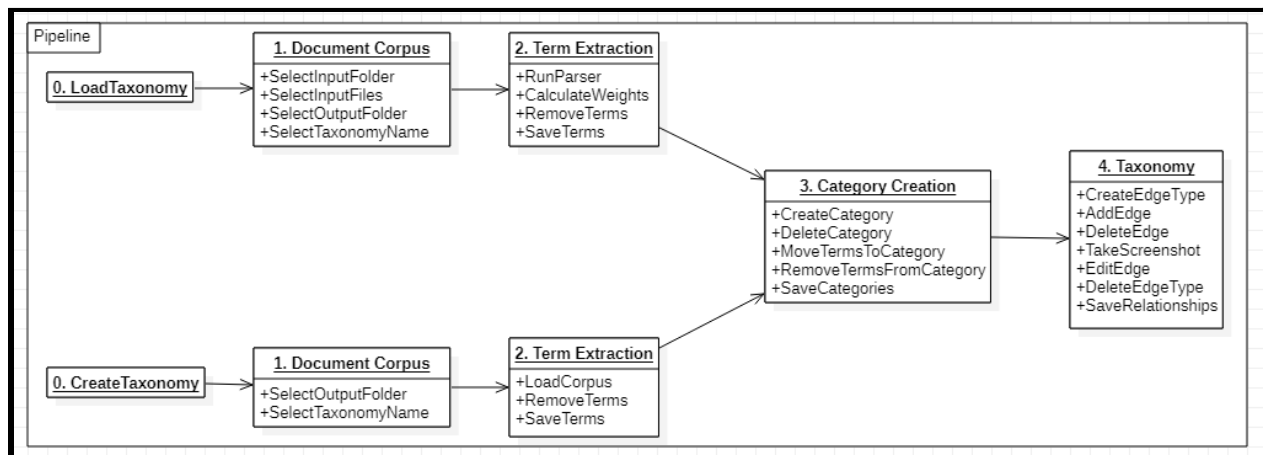
“Introduction to Natural Language Processing,” *GeeksforGeeks*, Nov. 16, 2018. <https://www.geeksforgeeks.org/introduction-to-natural-language-processing>.

“Rule-Based Classifier - Machine Learning,” *GeeksforGeeks*, May 06, 2020. <https://www.geeksforgeeks.org/rule-based-classifier-machine-learning>

Wikipedia Contributors, “Boeing,” *Wikipedia*, Jan. 24, 2019. <https://en.wikipedia.org/wiki/Boeing>

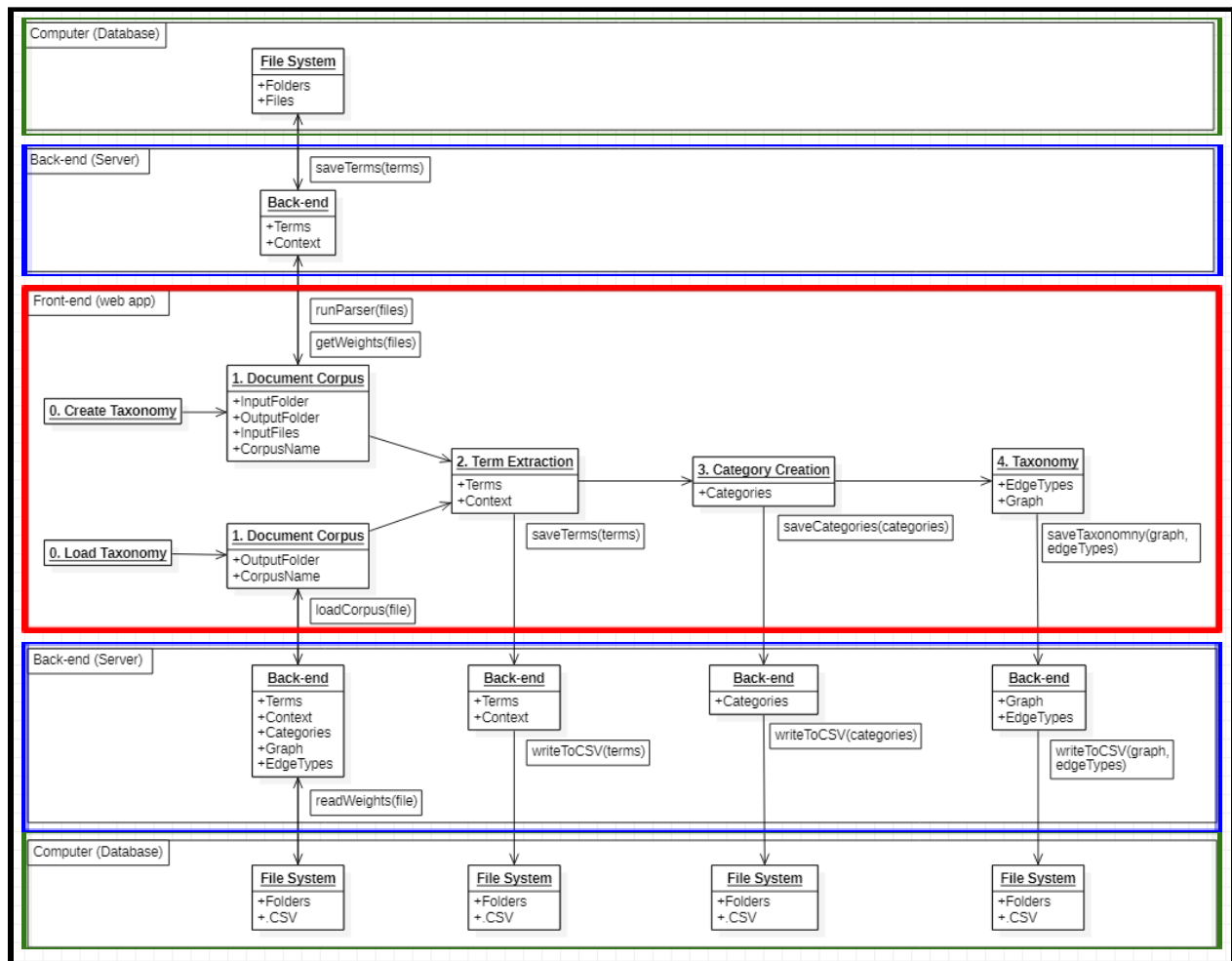
## **13.0 Appendix A - Pipe-and-Filter Design Pattern**

This is the outline for our pipe-and-filter architecture. You can see the basic functions each filter will have. Also notice that Load Taxonomy and Create Taxonomy converge in step 3. This is because the functions from that point on are the same for both.



## **14.0 Appendix B - Client-Server Design Pattern**

This is the outline for our client-server architecture. The red box contains the front-end functionality of our app. The red-box will only interact with the blue boxes. The blue boxes contain the back-end functionality. The functions in the blues boxes will interact with the red box and green boxes. Finally the green boxes contain the functionality for the database. The green boxes can only interact with the blue boxes.



## 15.0 Appendix C - UI Design Inspiration (Step 1)

This is the picture we used to inspire our web app's first step in the pipeline.

The screenshot shows a web application interface for 'Step 1: Document corpus'. At the top, there is a navigation bar with four steps: 1. DOCUMENTS (active), 2. TERMS, 3. CATEGORIES, and 4. TAXONOMY. Below the navigation bar, the title 'Step 1: Document corpus' is displayed. The interface is divided into two tabs: 'File' and 'URL'. The 'File' tab is selected. In the center, there is a large grey box with the text 'Select a document to upload:' and a subtext 'Choose file No file chosen'. Below this, there is a blue 'Upload' button. Below the upload section, there is a green box labeled 'Language:' with a dropdown menu showing 'English'. Below the language section, there is a search bar with the text 'Search' and a magnifying glass icon. To the right of the search bar is a blue 'Reload' button. Below the search bar, there is a table with two columns: 'File' and 'Actions'. The table contains five rows of data, each with a file name and two action buttons: 'preview' and 'delete'.

File	Actions
5b9ad283ad5543a60b412285.txt	<a href="#">preview</a> <a href="#">delete</a>
5b9ad282ad5543a60b411376.txt	<a href="#">preview</a> <a href="#">delete</a>
5b9e83966f96bcf3a123c7f.txt	<a href="#">preview</a> <a href="#">delete</a>
5b9e83966f96bcf3a123caa.txt	<a href="#">preview</a> <a href="#">delete</a>
5b9ad283ad5543a60b4121e7.txt	<a href="#">preview</a> <a href="#">delete</a>

## **16.0 Appendix D - UI Design Inspiration (Step 2)**



This is the picture we used to inspire our web app's second step in the pipeline.

**Step 2: Term extraction**

Parts-of-speech:  
☒ Nouns ☒ Verbs ☒ Adjectives ☒ Adverbs  
 Selection criteria  
☐ Frequency ☒ Weight  
 Threshold (10 %) ?  
 1934 terms selected out of 5278 | Lowest weight: 1

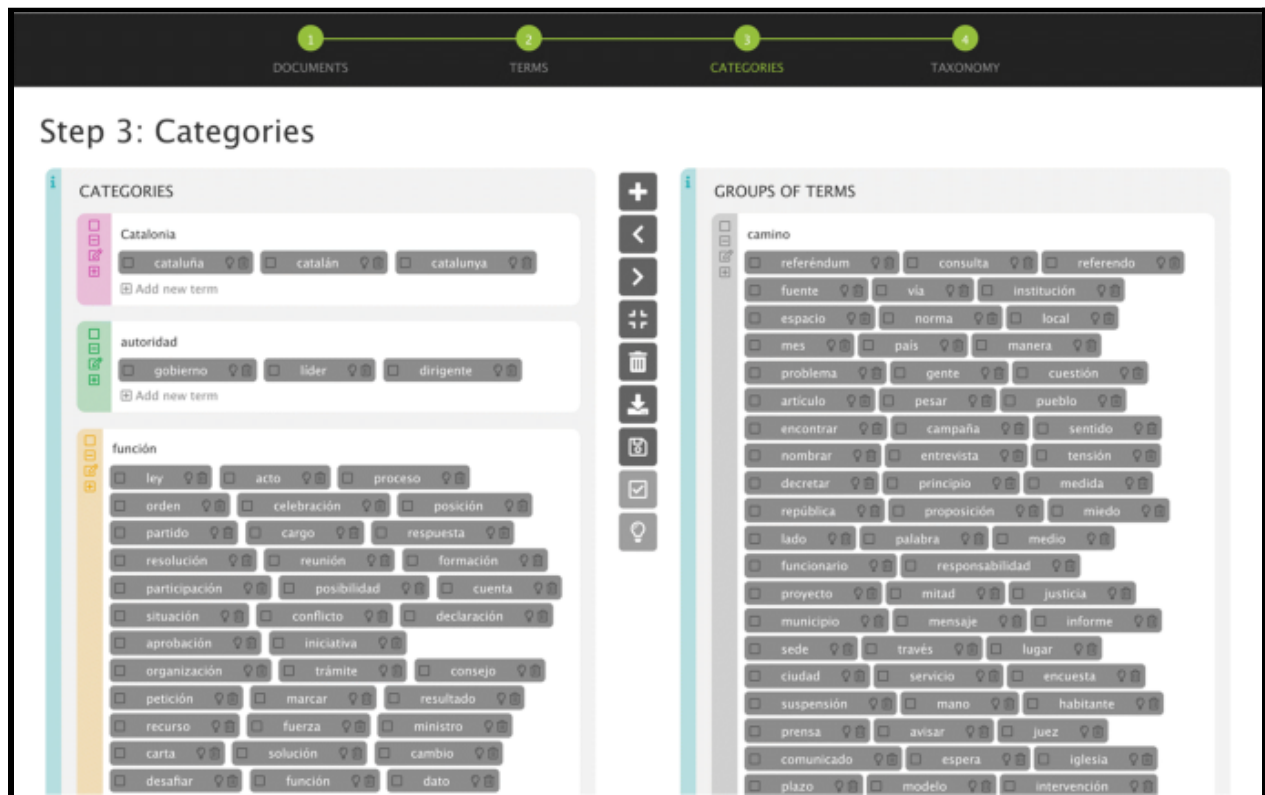
Show 10 entries

Search:

Term	Part-of-speech	Frequency	Docs	Weight
cataluña	Noun	264	42	241.04
catalán	Noun	257	39	217.89
referéndum	Noun	199	43	186.02
ley	Noun	313	25	170.11
gobierno	Noun	212	32	147.48
estar	Noun	156	32	108.52
consulta	Noun	128	39	108.52
Generalitat	Noun	135	30	88.04
presidente	Noun	109	25	59.24
Parlament	Noun	171	15	55.76

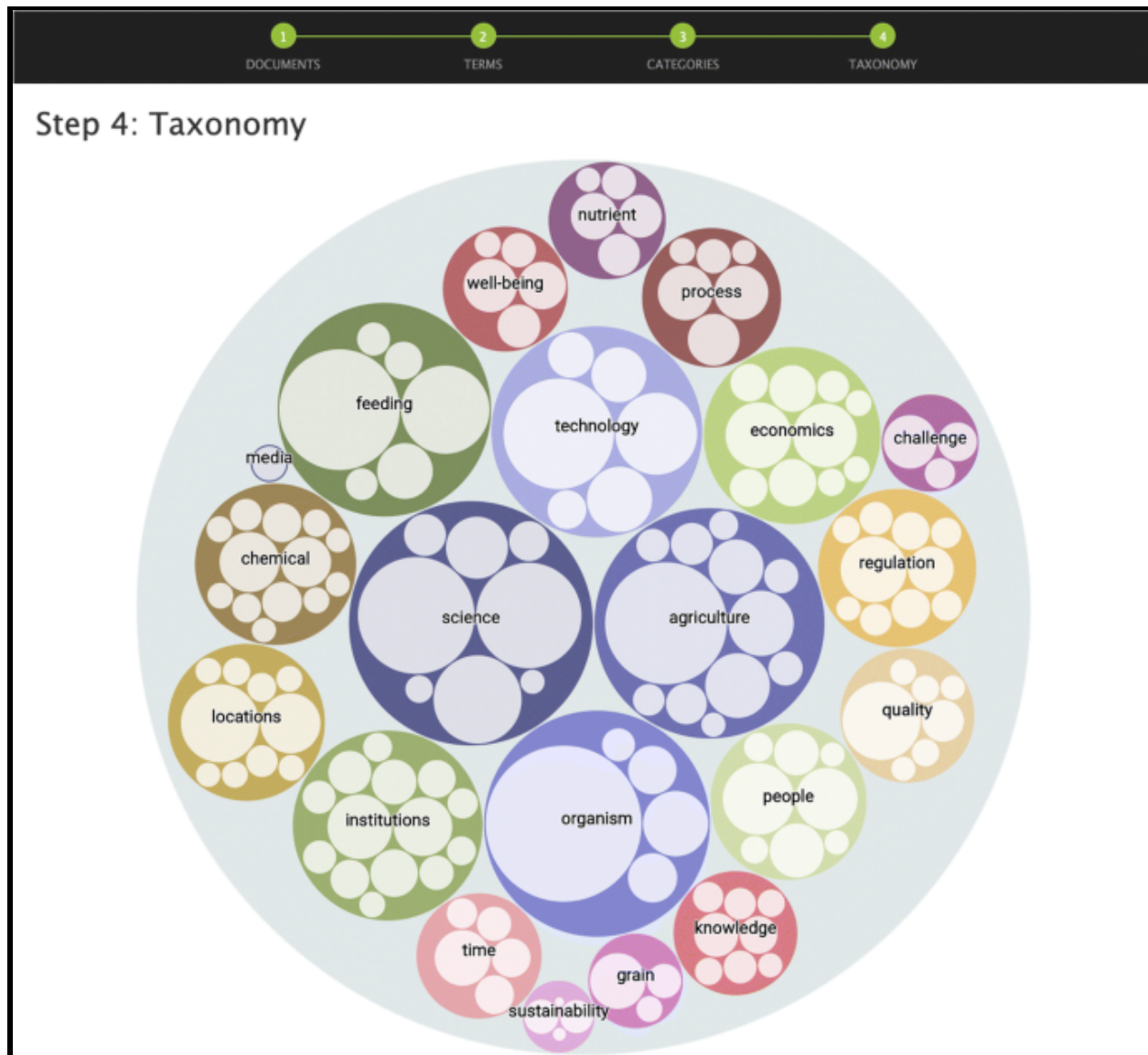
## **17.0 Appendix E - UI Design Inspiration (Step 3)**

This is the picture we used to inspire our web app's third step in the pipeline.



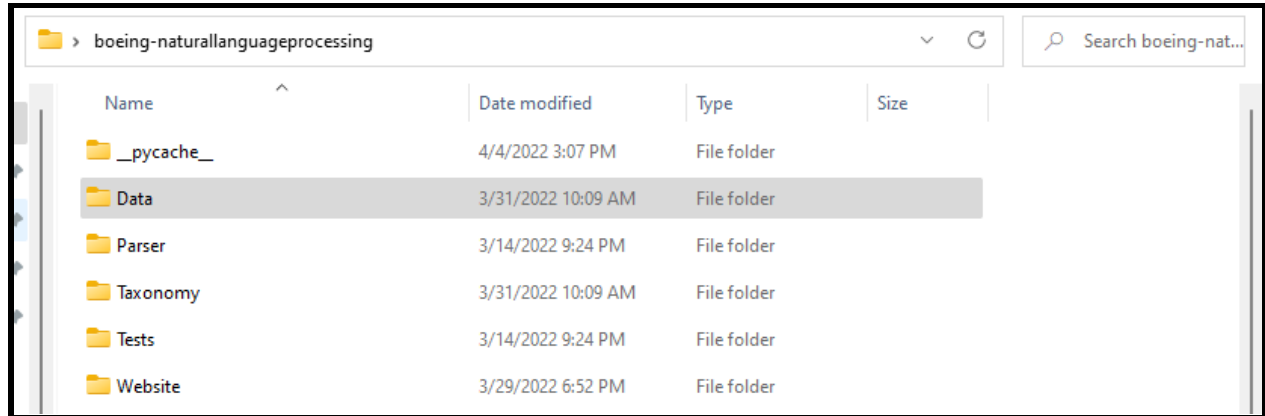
## 18.0 Appendix F - UI Design Inspiration (Step 4)

This is the picture we used to inspire our web app's fourth step in the pipeline.



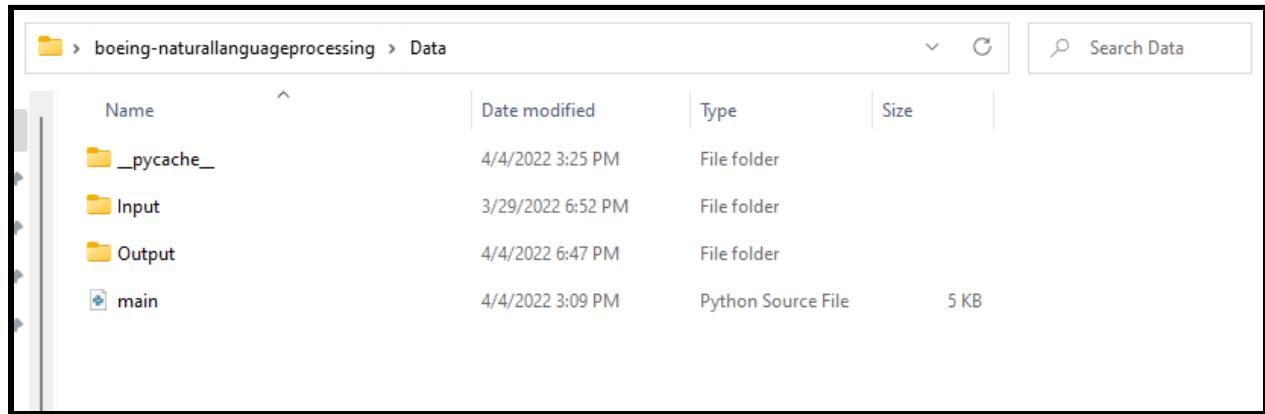
## **19.0 Appendix G - Final Prototype (Database: Project)**

This is the folder structure for the entire project.



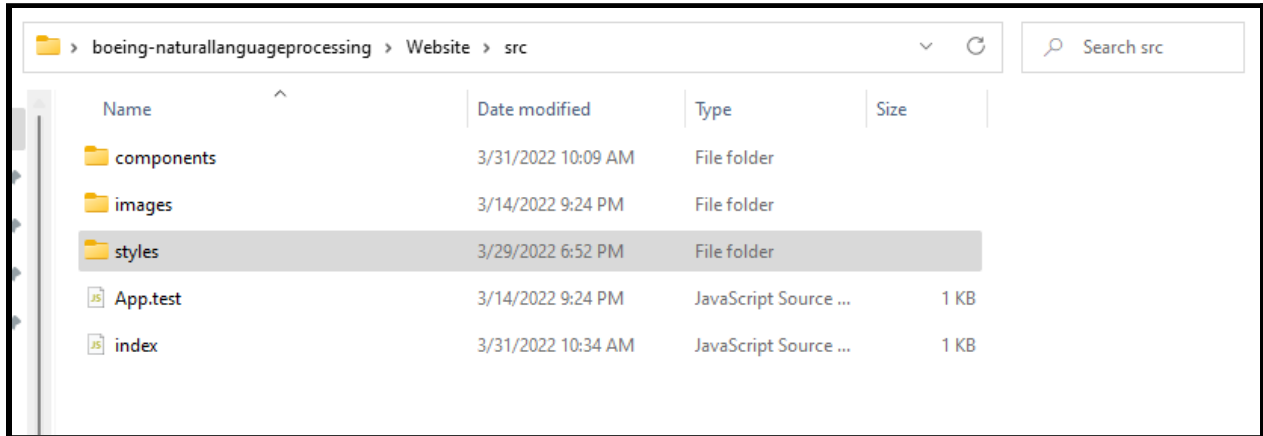
## **20.0 Appendix H - Final Prototype (Database: Data)**

This is the folder structure for the Data folder. Inside the Input folder are documents that are to be parsed. The .csv and .json files that our program creates will eventually be put into the Output folder.



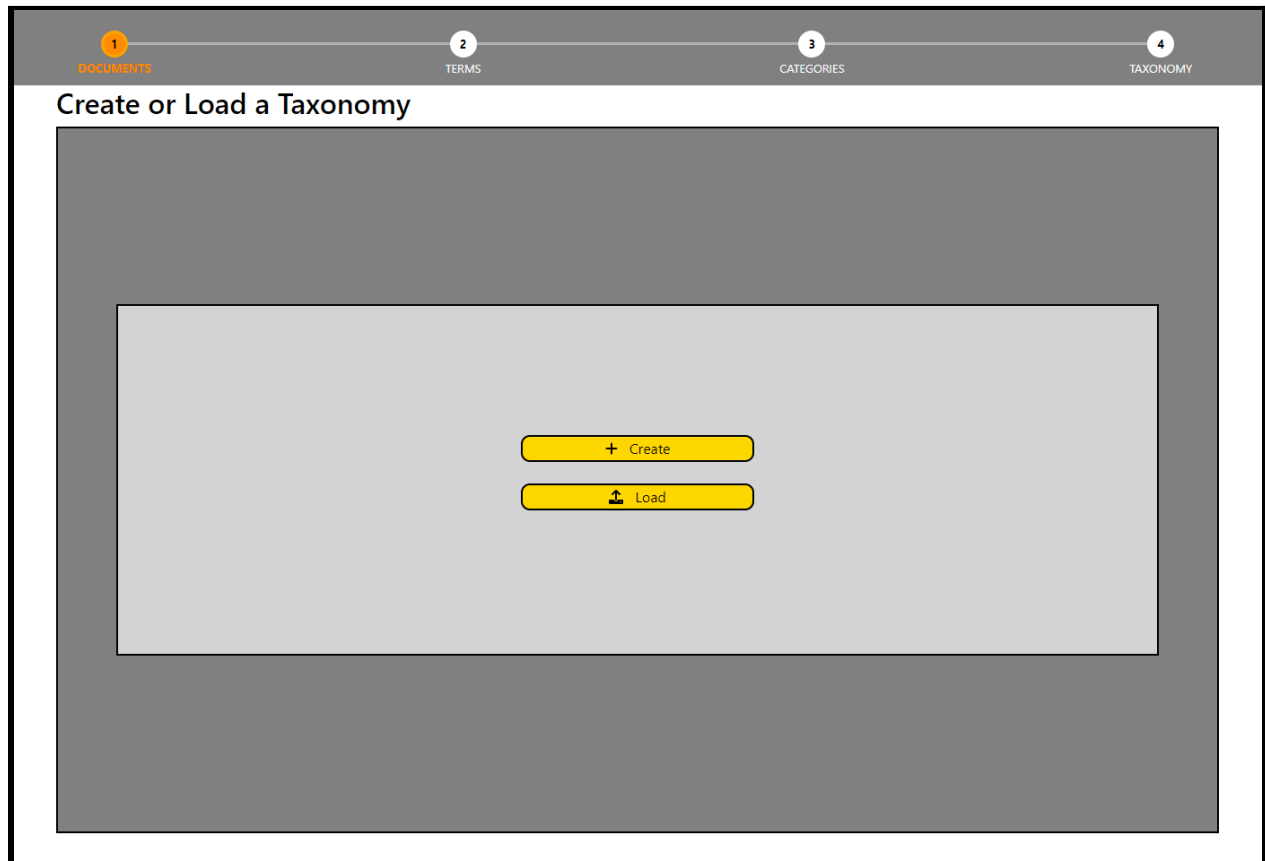
## **21.0 Appendix I - Final Prototype (Database: Website)**

This is the folder structure for the Website folder. There are three subfolders; Components, Images, and Styles.



## **22.0 Appendix J - Final Prototype (Step 0)**

This is the first screen in our web app. The example article by B. Carrion did not have functionality to load a taxonomy, so they didn't need this extra page. From here the user can decide to create a new taxonomy or load an old taxonomy.



## **23.0 Appendix K - Final Prototype (Step 1: Create)**

This is how the completed Document Corpus (Create) page will look. The user has submitted an input location, an output location, and named the taxonomy as master. The user has selected three documents to parse which are highlighted blue.

1DOCUMENTS

2TERMS

3CATEGORIES

4TAXONOMY

## Document Corpus (Create)

File	Add/ Remove
737_Pilot_Operating_Manual.pdf	<input type="button" value="Delete"/>
Futball	<input type="button" value="Add"/>
Games	<input type="button" value="Delete"/>
Music	<input type="button" value="Delete"/>
New Microsoft Word Document.pdf	<input type="button" value="Add"/>
moby-dick-002-chapter-1-loomings.pdf	<input type="button" value="Add"/>

## 24.0 Appendix L - Final Prototype (Step 1: Load)



This is how the completed Document Corpus (Load) page will look. The user has submitted an output location and the name of the taxonomy to load.

1 DOCUMENTS 2 TERMS 3 CATEGORIES 4 TAXONOMY

### Document Corpus (Load)

C:\Users\blcsi\OneDrive\Desktop\boeing-naturallanguageprocessing\Data\Output

Output: C:\Users\blcsi\OneDrive\Desktop\boeing-naturallanguageprocessing\Data\Output

master

Name: master

## **25.0 Appendix M - Final Prototype (Step 2)**

This is how the completed Term Extraction page will look. The terms have been parsed and are displayed in the term table. If this was a load rather than a create, then the terms have been loaded from a taxonomy rather than parsed. The expert has also selected five terms (highlighted blue), and is deciding if they should delete the terms from the taxonomy.

1 DOCUMENTS

2 TERMS

3 CATEGORIES

4 TAXONOMY

### Term Extraction

Refresh Terms:

Save Terms:

Select terms to remove

Noun	Frequency	Weight	Sentences
a 737 or other small aircraft	1	1	<div>Sentences</div>
a very large aircraft	1	1	<div>Sentences</div>
a very short approach	1	1	<div>Sentences</div>
instance	1	1	<div>Sentences</div>
large aircraft rule	1	1	<div>Sentences</div>
pilot	2	2	<div>Sentences</div>
problem	1	1	<div>Sentences</div>
some pilot error	1	1	<div>Sentences</div>
the 777's power	1	1	<div>Sentences</div>
the most technologically advanced commercial aircraft	1	1	<div>Sentences</div>
the power	1	1	<div>Sentences</div>

< Previous:

0

Next: >

Clear Selected

Delete Terms

Back

Forward

## 26.0 Appendix N - Final Prototype (Step 2: Sentences)

While in the Term Extraction page, an expert wants to see the context a term is found in. Each term will have a Sentences button, when clicked a modal will appear with each sentence the term is found in, and the document the sentence came from.

The screenshot displays the 'Term Extraction' interface. At the top, a header reads 'Select terms to remove'. Below this is a table with four columns: 'Noun', 'Frequency', 'Weight', and 'Sentences'. The table lists several terms related to aircraft, with 'problem' highlighted. A modal window is open over the 'problem' row, showing the context of the term. The modal has two columns: 'Location' and 'Sentences'. The 'Location' column shows 'test\_paragraph' and the 'Sentences' column shows the sentence: 'The 777's power can offset some pilot errors, but the power can also cause problems during tight maneuvers.' At the bottom of the interface, there are navigation buttons: '< Previous', '0', 'Next: >', 'Clear Selected', and 'Delete Terms'.

Noun	Frequency	Weight	Sentences
a 737 or other small aircraft	1	1	Sentences
a very large aircraft	1	1	Sentences
a very short approach			Sentences
instance			Sentences
large aircraft rule			Sentences
pilot			Sentences
problem	1	1	Sentences
some pilot error	1	1	Sentences
the 777's power	1	1	Sentences
the most technologically advanced commercial aircraft	1	1	Sentences
the power	1	1	Sentences

problem

Location	Sentences
test_paragraph	The 777's power can offset some pilot errors, but the power can also cause problems during tight maneuvers.

< Previous 0 Next: >

Clear Selected Delete Terms

## 27.0 Appendix O - Final Prototype (Step 3)

This is how the completed Category Creation page will look. The terms table (leftmost section) is carried over from the previous page. You can see the button column (middle section). Finally, you can see the categories table (rightmost section). The expert has made two categories, but has only added terms into the first one. For this to be completed, the expert needs to at least add terms to the second category.

1 DOCUMENTS
2 TERMS
3 CATEGORIES
4 TAXONOMY

### Category Creation

#### Terms

Select terms to move to category

Noun	Frequency	Weight
instance	1	1
large aircraft rule	1	1
pilot	2	2
the most technologically advanced commercial aircraft	1	1
the power	1	1
the world	1	1
they	1	1
tight maneuver	1	1
very tight turn	1	1

Edit Categories

+ Create New Category

Clear Selected

Save Categories

#### Categories

Select a category to edit

Category	Terms
Category1	<div style="border: 1px solid black; padding: 2px;"> a 737 or other small aircraft  a very large aircraft  a very short approach </div>
Category2	

< Previous: 0
Next: >

◀ Back
Forward ▶

< Previous: 0
Next: >

## 28.0 Appendix P - Final Prototype (Step 3: Edit)

While in the Category Creation Page, an expert has clicked on a category that was in the category table (rightmost section). The table has now been replaced with a new section which displays just the terms added to that category. From here, the expert can add more terms to this category, remove terms from the category, or delete the category completely.

Category Creation

Terms

Select terms to move to category

Noun	Frequency	Weight
instance	1	1
large aircraft rule	1	1
pilot	2	2
the most technologically advanced commercial aircraft	1	1
the power	1	1
the world	1	1
they	1	1
tight maneuver	1	1
very tight turn	1	1

Edit Categories

Exit Category

Delete Category

←

→

Clear Selected

Category1

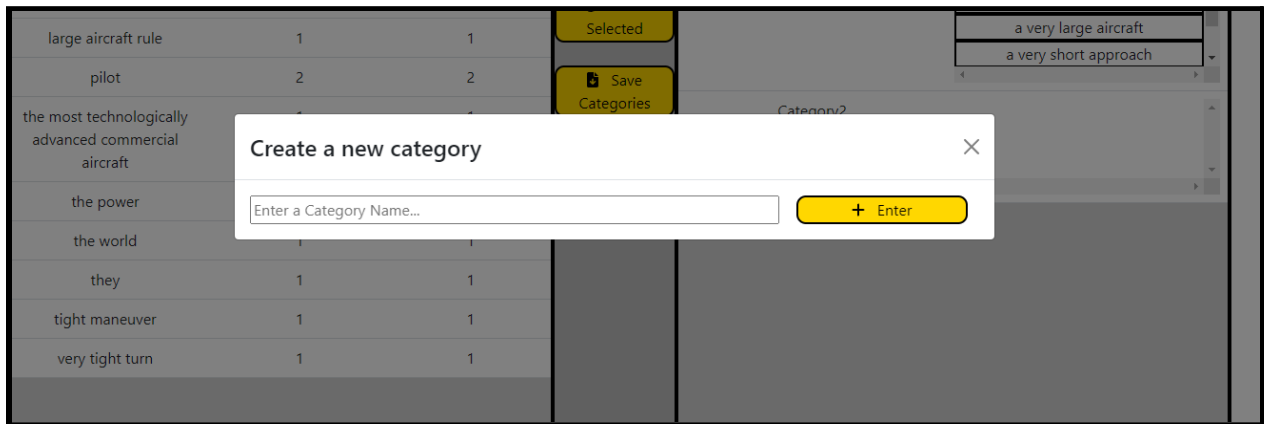
Select terms to remove from category

Terms

a 737 or other small aircraft
a very large aircraft
a very short approach
problem
some pilot error
the 777's power

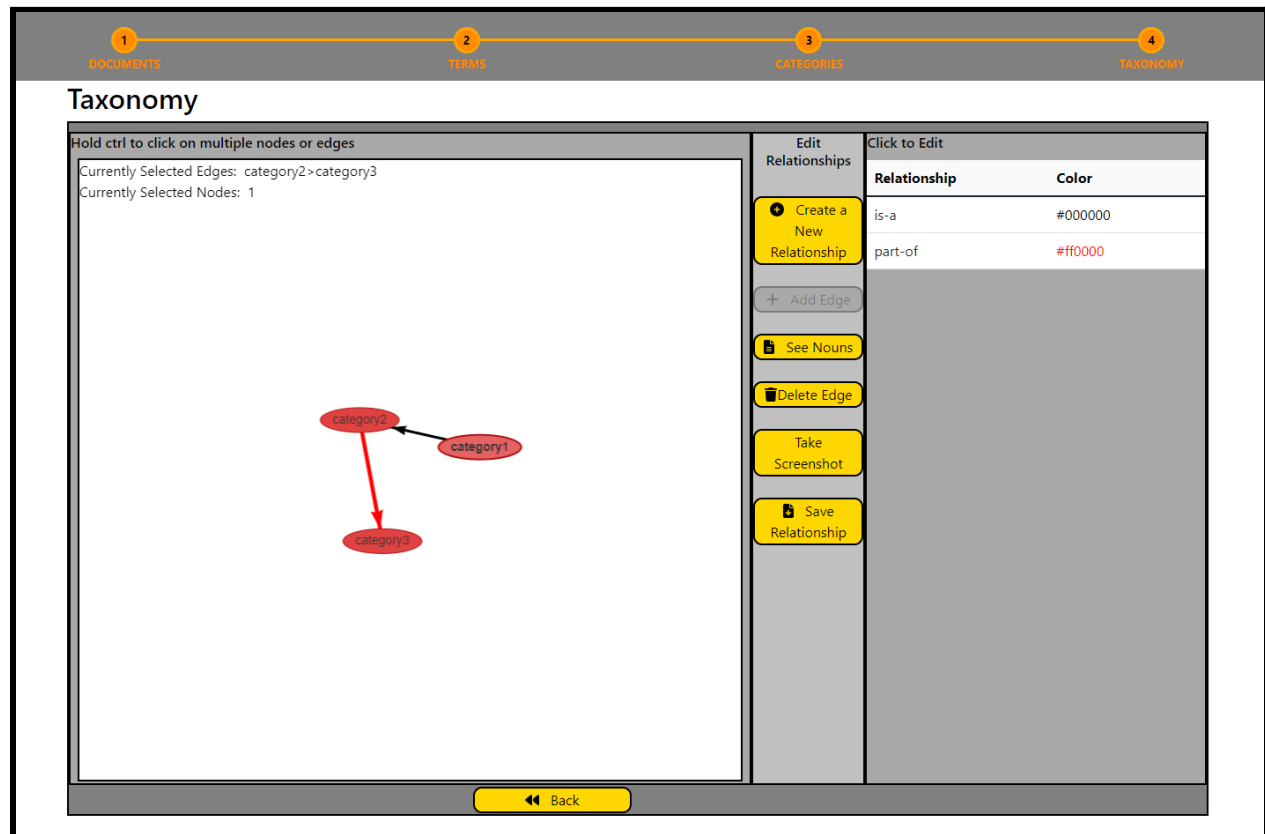
## 29.0 Appendix Q - Final Prototype (Step 3: Create)

While in the Category Creation Page, an expert has clicked the Create New Category button. A modal has now appeared asking the expert to name the category.



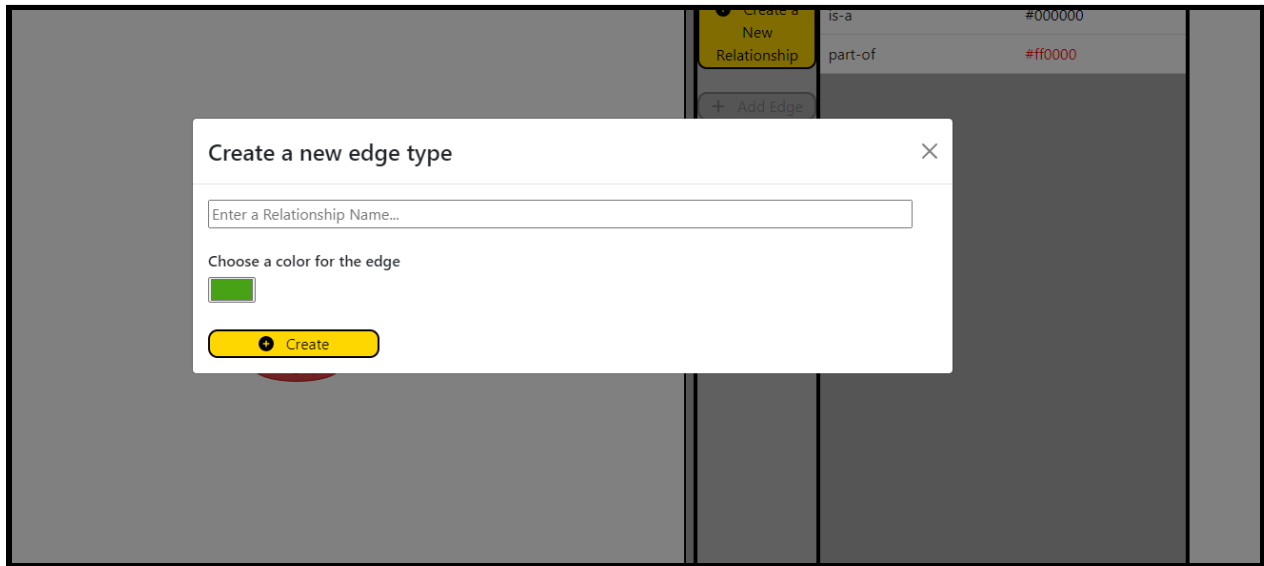
## **30.0 Appendix R - Final Prototype (Step 4)**

This is how the completed Taxonomy page will look. You can see in the leftmost section, there is a graph with three nodes and two edges. The three nodes correspond to three categories that were created in the previous page. The expert then created two relationship types (seen in the rightmost section) and added a relationship between two nodes twice. In the top left corner of the graph, you can also see that the expert has selected the edge between “category 2” and “category 3”, and the node “category 1”. The middle section contains buttons similar to the previous page.



## **31.0 Appendix S - Final Prototype (Step 4: Create)**

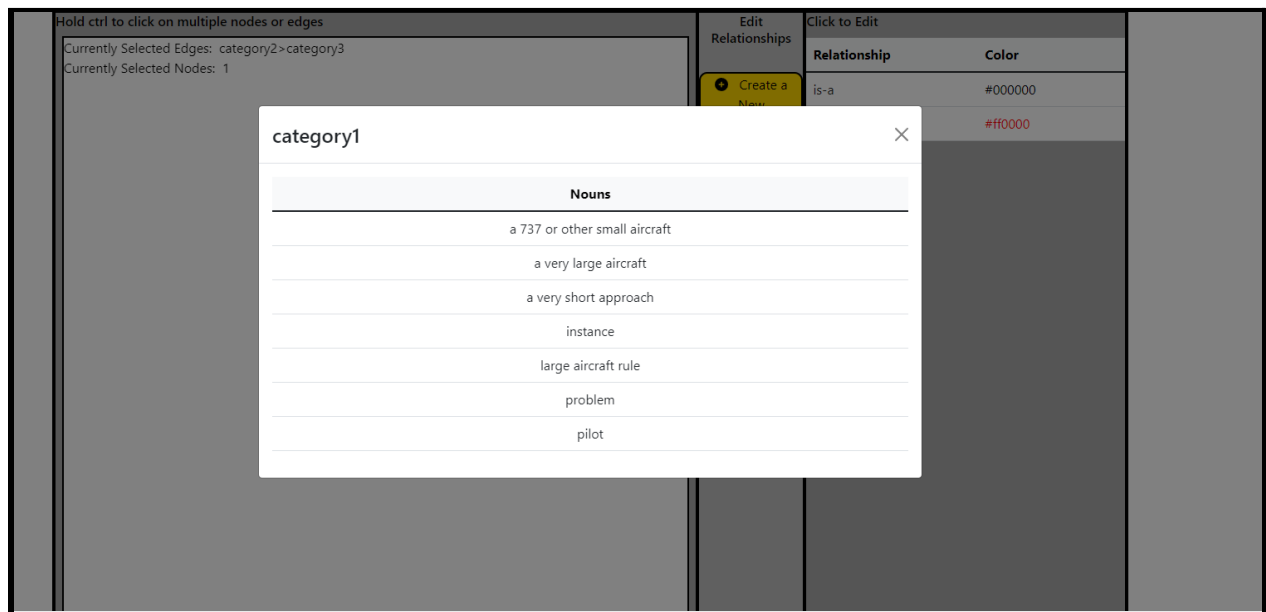
While in the Taxonomy page, the user has clicked on the Create a New Relationship button. This is the modal that will appear, asking the expert to choose a name and a color for the edge type.



## **32.0 Appendix T - Final Prototype (Step 4: Nouns)**

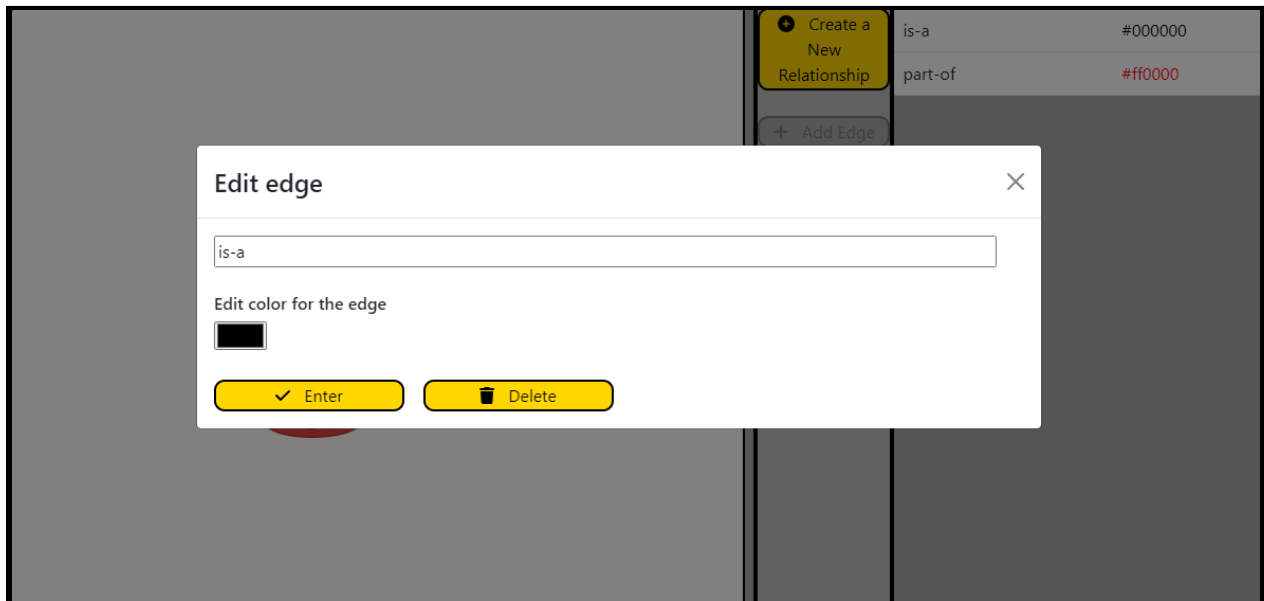


While in the Taxonomy page, the user has selected a node, and has clicked the Show Nouns button. This is the modal that will appear, giving the expert information on what terms are in that category (node).



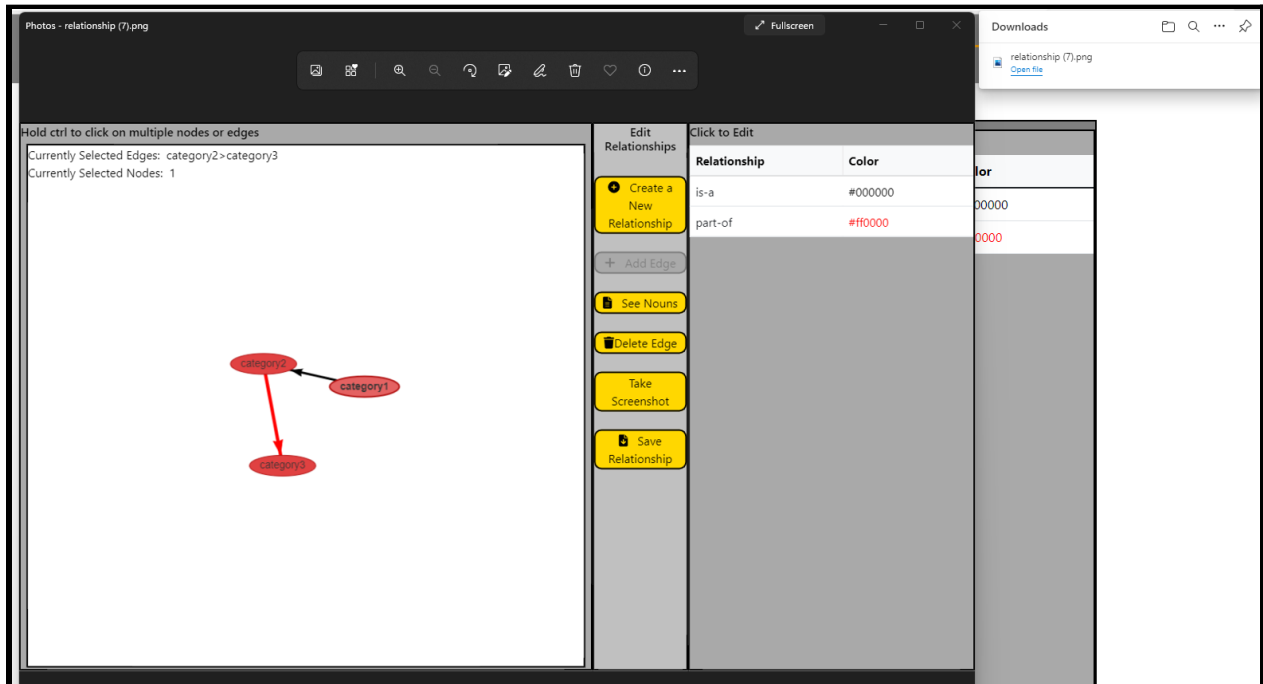
## 33.0 Appendix U - Final Prototype (Step 4: Edit)

While in the Taxonomy page, the user has clicked on an edge type in the edge type table. This is the modal that will appear, asking the expert if they want to edit the name, the color, or to delete the edge type from the table.



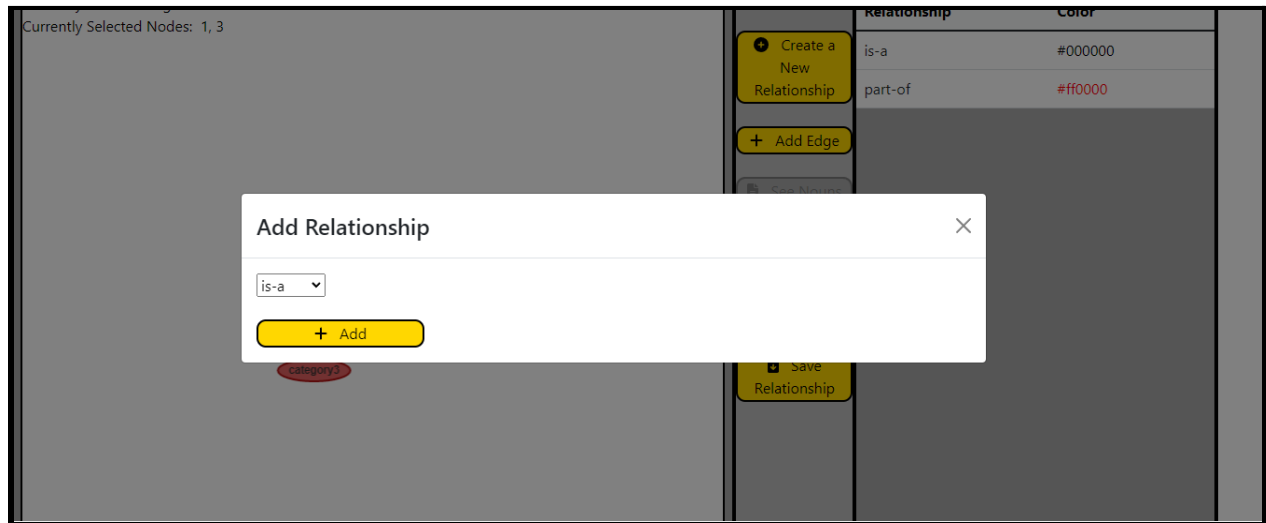
## **34.0 Appendix V - Final Prototype (Step 4: Screenshot)**

While in the Taxonomy page, the user has clicked on the Take Screenshot button. You can see in the top right corner that the screenshot has been saved as relationships.png. We have opened that file with a photo app, and as you can see the graph and the edge type table were included in the screenshot.



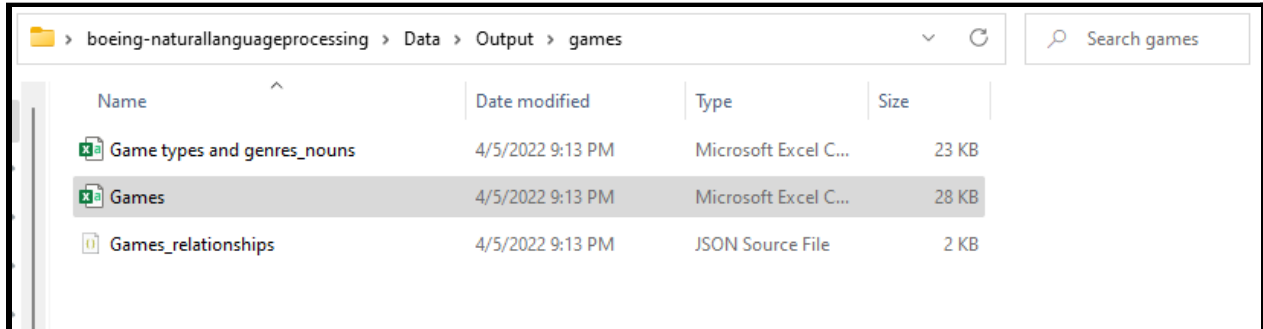
## **35.0 Appendix W - Final Prototype (Step 4: Add)**

While in the Taxonomy page, the user has clicked on the Add Edge button. This is the modal that will appear, asking the expert which relationship type do they want to add between the two nodes they have selected.



## **36.0 Appendix X - Final Prototype (Database: Output)**

This is the folder structure for the Data/Output folder. There will be three main file types. The mini CSV files created for each document parsed. The master taxonomy CSV that combines the mini CSV files and includes weight and category information. The third file type is the graph and edge type JSON for the taxonomy.



Name	Date modified	Type	Size
Game types and genres_nouns	4/5/2022 9:13 PM	Microsoft Excel C...	23 KB
Games	4/5/2022 9:13 PM	Microsoft Excel C...	28 KB
Games_relationships	4/5/2022 9:13 PM	JSON Source File	2 KB

## **37.0 Appendix Y - Final Prototype (Database: Mini)**

This is how the mini CSV will look for each document parsed.

	A	B	C	D
1	Game types and genres_nouns.csv			
2	Unique nouns: 144	Total nouns: 180		
3	Total time: 1.576 sec	Cost per noun: 8.758 ms		
4	game type	['Game Type and Game Genre', 'By Lindsay G	5	
5	game genre	['Game Type and Game Genre', 'By Lindsay G	5	
6	lindsay grace game type	['By Lindsay Grace Game Type Although som	1	
7	the industry	['By Lindsay Grace Game Type Although som	1	
8	a distinct difference	['By Lindsay Grace Game Type Although som	1	
9	video game	['By Lindsay Grace Game Type Although som	1	
10	game story	['When discussing game story, we distinguis	1	
11	we	['When discussing game story, we distinguis	1	
12	a description	['When discussing game story, we distinguis	2	
13	game play	['When discussing game story, we distinguis	2	
14	the narrative content	['When discussing game story, we distinguis	1	
15	the game	['When discussing game story, we distinguis	1	
16	brief list	['The following is brief list of game types:']	1	
17	action	['Action: Games that offer intensity of action	2	
18	game	['Action: Games that offer intensity of action	7	
19	intensity	['Action: Games that offer intensity of action	1	
20	the primary attraction	['Action: Games that offer intensity of action	2	
21	reflex response	['Reflex response is the primary skill needed	1	
22	the primary skill	['Reflex response is the primary skill needed	1	
23	these game	['Reflex response is the primary skill needed	4	
24	the most common action game	['The most common action games are shoot	1	
25	shooter	['The most common action games are shoot	1	
26	doom	['The most common action games are shoot	1	

## **38.0 Appendix Z - Final Prototype (Database: Master)**

This is how the master taxonomy CSV will look.

	A	B	C	D	E	F
1	a comprehensive list	['Game types and genre	1		1 GameCat1	
2	a crime, fantasy genre	['Game types and genre	1		1 GameCat1	
3	a description	['Game types and genre	2		2 GameCat1	
4	a distinct difference	['Game types and genre	1		1 GameCat1	
5	a game genre	['Game types and genre	1		1 GameCat1	
6	a genre	['Game types and genre	1		1 GameCat1	
7	a good adventure game player	['Game types and genre	1		1 GameCat1	
8	a list	['Game types and genre	1		1 GameCat1	
9	a narrative style	['Game types and genre	1		1 GameCat1	
10	a simulation	['Game types and genre	2		2 GameCat1	
11	a single category	['Game types and genre	1		1 GameCat1	
12	a thoroughly entertaining collection	['Game types and genre	1		1 GameCat1	
13	action	['Game types and genre	2		2 GameCat2	
14	action game	['Game types and genre	1		1 GameCat2	
15	adventure	['Game types and genre	1		1 GameCat2	
16	all world	['Game types and genre	1		1 GameCat2	
17	an action, role-playing game type	['Game types and genre	1		1 GameCat2	
18	an explanation	['Game types and genre	1		1 GameCat2	
19	an object	['Game types and genre	1		1 GameCat2	
20	an opportunity	['Game types and genre	1		1 GameCat2	
21	another interesting puzzle game	['Game types and genre	1		1 GameCat2	
22	anyone	['Game types and genre	1		1 GameCat2	
23	baldour's gate	['Game types and genre	1		1 GameCat3	
24	brief list	['Game types and genre	1		1 GameCat3	
25	character	['Game types and genre	2		2 GameCat3	
26	character management	['Game types and genre	1		1 GameCat3	

## **39.0 Appendix AA - Final Prototype (Database: JSON)**

This is how the JSON for the graph and edge types will look.

```

Data > Output > games > {} Games_relationships.json > ...
1  {
2    "graph": {
3      "nodes": [
4        {
5          "id": 4,
6          "label": "GameCat1",
7          "color": "#e04141"
8        },
9        {
10         "id": 5,
11         "label": "GameCat2",
12         "color": "#e04141"
13       },
14       {
15         "id": 6,
16         "label": "GameCat3",
17         "color": "#e04141"
18       }
19     ],
20     "edges": [
21       {
22         "from": 4,
23         "to": 6,
24         "color": "#ff0000",
25         "width": 3,
26         "relationship": "test1",
27         "id": "c21dc022-fd1e-4597-a7f8-f3c46b910c87"
28       },
29       {
30         "from": 6,
31         "to": 5,
32         "color": "#389a1d",
33         "width": 3,
34         "relationship": "test2",
35         "id": "1e122f90-e072-4f33-b431-17d3515349ea"
36       }
37     ]
38   },
39   "relationships": [
40     {
41       "test1": "#ff0000"
42     },
43     {
44       "test2": "#389a1d"
45     }
46   ]
47 }

```

## **40.0 Appendix AB - Speed Test Output**

This is a test run measuring the speed of asynchronous parsing .



```

(.venv) C:\Users\User\Documents\Cap\boeing-naturallanguageprocessing-fork [main = +2 -9 -1 !]> python -m tests.non_functional.parser
_speed
starting speed test for a multiple docx file asynchronously
starting process pool
parsing 737_Pilot_Operating_Manual.pdf
parsing blank.docxparsing EldenRingMount.docx

finished parsing blank.docx
Total time: 0.004 sec
parsing EldenRingStance.docx
parsing test_sentences_10.docx
finished parsing EldenRingMount.docx
Total time: 0.051 sec
finished parsing test_sentences_10.docx
Total time: 0.056 sec
finished parsing EldenRingStance.docx
Total time: 0.073 sec
finished parsing 737_Pilot_Operating_Manual.pdf
Total time: 13.14 sec
Multi async file took 14 sec
(.venv) C:\Users\User\Documents\Cap\boeing-naturallanguageprocessing-fork [main = +2 -9 -1 !]>

```

## **41.0 Appendix AC - Full PyTest Output**

This is a PyTest run showcasing the main tests

```

collected 22 items

tests/integration/test_parse.py::test_accuracy SKIPPED (Helper function) [ 4%]
tests/integration/test_parse.py::test_accuracy_blank PASSED [ 9%]
tests/integration/test_parse.py::test_accuracy_EldenMount PASSED [ 13%]
tests/integration/test_parse.py::test_accuracy_EldenStance PASSED [ 18%]
tests/integration/test_parse.py::test_full_run_single PASSED [ 22%]
tests/integration/test_parse.py::test_full_run_multi PASSED [ 27%]
tests/unit/test_extract.py::test_extract_txt PASSED [ 31%]
tests/unit/test_extract.py::test_extract_txt_invalid PASSED [ 36%]
tests/unit/test_extract.py::test_extract_pdf_file PASSED [ 40%]
tests/unit/test_extract.py::test_extract_pdf_file_invalid PASSED [ 45%]
tests/unit/test_extract.py::test_extract_docx_file PASSED [ 50%]
tests/unit/test_extract.py::test_extract_txt_docx_invalid PASSED [ 54%]
tests/unit/test_read_write.py::test_write_to_csv PASSED [ 59%]
tests/unit/test_read_write.py::test_write_to_json PASSED [ 63%]
tests/unit/test_read_write.py::test_read_weights PASSED [ 68%]
tests/unit/test_read_write.py::test_read_weights_invalid PASSED [ 72%]
tests/unit/test_spacy.py::test_get_terms_blank PASSED [ 77%]
tests/unit/test_spacy.py::test_get_terms_1 PASSED [ 81%]
tests/unit/test_spacy.py::test_get_terms_2 PASSED [ 86%]
tests/unit/test_spacy.py::test_get_terms_invalid_1 PASSED [ 90%]
tests/unit/test_spacy.py::test_get_terms_invalid_2 PASSED [ 95%]
tests/unit/test_spacy.py::test_get_terms_invalid_3 PASSED [100%]

===== warnings summary =====
..\venv\lib\site-packages\docx\section.py:7
  c:\users\user\documents\cap\venv\lib\site-packages\docx\section.py:7: DeprecationWarning: Using or importing the ABCs from 'collections' instead of from 'collections.abc' is deprecated since Python 3.3, and in 3.9 it will stop working
    from collections import Sequence

-- Docs: https://docs.pytest.org/en/stable/warnings.html

```