WSU Libraries Accessibility Project

Project Report

WSU Libraries

WSU Libraries Accessibility Team

Trent Bultsma, Reagan Kelley and Marisa Loyd

TABLE OF CONTENTS

I. Introduction

In accordance with and in shared pursuit of WSU's research exchange mission, we would like to help create a space designed to preserve and share university scholarship [1]. Within a single and shared digital repository, not only do we want a limitless array of knowledge from articles, books, papers, and reports, but we want this digital media to be accessible to all. This starts, first, with these digital documents meeting the standards set out by W3C, an international community trying to bring public work together by providing concrete standards for websites and digital media.

Many digital works are brought to the research exchange repository lacking the initial accessibility standards for digital media set out by the international community, and has resulted in an unknown but copious amount of educational media with sub-optimal accessibility [3]. Our goal is to create an application that can take a pdf and create a modified version that does not change the comprehension or meaning of the work but heightens that document's accessibility to that of W3C standards. We wish to then streamline this process, with it not just assisting a single document, but that of an entire repository, to make the entire WSU research exchange significantly more accessible to all.

II. Background and Related Work

Through researching our project field, we have discovered a lack of automated solutions to the problem of document, and specifically pdf, accessibility. Adobe Acrobat does have some tools for accessibility but they require manual input. For example, when using Adobe Acrobat to make a pdf accessible, the user must first choose the accessibility option in the tools menu, then they have to click on the full check option which opens a pop up window for the user. After this, the user would have to go to the report options of the pop up window, select the page range, which accessibility options to search for and then choose to start checking the document [2]. This manual input required by Adobe Acrobat is something that our team aims to bypass. Our goal is to create a process which automatically updates a pdf's accessibility based on what it lacks without requiring user input.

During our research into the problem of making pdfs accessible, we found that there is an open source repository, pdfminer.six, on GitHub that extracts data from pdf documents [4]. Most specifically, this repository is able to parse, analyze and convert pdfs, extract content as text, html or images, extract tagged content, and extract images. Extracting information from pdfs is a necessary part of our project. However, our focus will be on the document's metadata, color contrast, tags, alternative text for images and reading order in addition to extracting the content and images in the document. Our system will also update these areas instead of simply extracting the information.

To complete this project, our team will need to learn how to extract information from pdfs, how to update the targeted accessibility features based on the requirements of the Web Content Accessibility Guidelines and how to create a new pdf with the updated accessibility features as well as the original information extracted from the pdf.

III. Project Overview

In our quest to bring more accessibility to the WSU research exchange, we have landed on a few key accessibility features to focus on, at least at the start, which can be expanded to other things as our project progresses. These initial features include document metadata, color contrast, tagging, alternative text for images, and reading order. Our desire is to create a fully automated system of taking pdf documents and converting them into these more accessible versions.

Regarding document metadata, the goal is to include the following information. Title, author, subject, keywords, and document language [5]. The title and author can most often be gathered from looking at the first page of the document for things like large or centered text or comparing groups of words with name databases. To determine the subject and keywords, we will need to implement some sort of data mining algorithms to find common words within the document in question that are uncommon among most documents [6].

The feature of tagging documents for software output is similar to html tagging. It involves marking things as a header, paragraph, or entries within a table, before converting the intermediary data gathered from the original pdf. Tagging helps define the reading order, which is another accessibility feature we aim to provide, especially in tables, as well as being used to define alt text for images [7].

For the issue of alternative text on images, we aim to tackle this with some sort of machine learning solution that can generate a description of the contents of images. It is not feasible for us to acquire the data required for a comprehensive image categorization program so we will need to resort to implementing a 3rd party solution. More research is required at this time to find a solution that will not be too costly to realistically implement, be that an open source database of categorized images or a cloud based ai to identify the images for us. This feature does seem like it could be the most time consuming and doesn’t have the highest benefit so for now, we have it at a low priority.

With regards to the reading order of the inputted document, our software solution aims to correctly identify and mark the order in which to read text on the page. This involves selecting the heading first, then title, then body in order. For the body of the document, identifying columns and selecting the order of those or choosing images and descriptions in a certain order will be part of that process. The goal of specifying a reading order for our documents is to provide a better experience for people who use assistive screen readers [8].

The feature of color contrast will be focused on contrast between the text and background color. This will be done by converting all text to 100% black on a white background during the recreation of the pdf in our automated process to be described shortly.

Now with the different initial features defined, we will go over the broader process of bringing those features into reality. Firstly, we will automatically acquire data in the form of pdf documents from the research exchange repository through some means of data harvesting. Next, we will use that data as input to our software that will convert the pdf into a form that can be understood by our automation such as html or a json file. We will then do intermediary processing to ensure the features described above are present. Finally, we recreate the pdf from all that data, resulting in a document that has the same text and image content, but with a much more accessible layout and backend tags/data. This process containing the features described should help us reach our goal of providing more accessibility to users of the WSU research exchange.

IV. Client and Stakeholder Identification and Preferences

Our primary clients are Washington State University Libraries and Anath Jillepalli, our professor for CptS 421. Stakeholders are people who are impacted by a software development project and in this project the stakeholders include Talea Anderson, our primary contact for the project, the employees of Washington State University Libraries, students, professors and researchers [9].

The stakeholders and clients of this project have a few distinct requirements and preferences. For instance, our client, Washington State University Libraries, requires that our team provide a way for employees to process and ensure that documents on Research Exchange, their online repository, meet the Web Content Accessibility Guidelines [10]. The main preference of Washington State University Libraries is that we look at a single collection, identify problems in this collection and then provide a solution to fix the problems identified.

Our stakeholders have slightly different needs, however. Talea and other employees of Washington State University Libraries require that we not only provide a way to process and ensure the documents on Research Exchange meet the Web Content Accessibility Guidelines, but also that we explain our tools and processes to them and show them how to use them. Stakeholders such as students, professors and researchers require that documents on Research Exchange be accessible and readable. The prominent preference for stakeholders is that the documents on Research Exchange have proper tags, alternative text for images, correct metadata and a clear reading order, especially when using screen readers.

V. Team Members - Bios and Project Roles

Trent Bultsma is the team lead for the project. He is a computer science student also pursuing a math minor at Washington State University, interested in graph theory and big data. Trent’s skills include Python, C#, Java, and the Unity game engine. He has industry experience working as a software engineering intern at Schweitzer Engineering Laboratories on circuit board factory automation. For this project, he is responsible for creating the document extractor.

Reagan Kelley is a student at Washington State University, pursuing an undergraduate degree in Computer Science and Philosophy. He has primarily primed his skills in web development and low-level programming designing his own custom graphics engine. He has taken his skills to gain experience in the medical research field, designing web-based applications for clinicians in gerontology. For this project, he is responsible for designing parts of the document transformation pipeline.

Marisa is a student at Washington State University who is studying computer science and is also working to complete a minor in mathematics. Marisa’s interests include linear algebra, calculus and logic and her skills include C, C++ and Java. For this project, she created the document exporter.

VI. System Requirements Specification

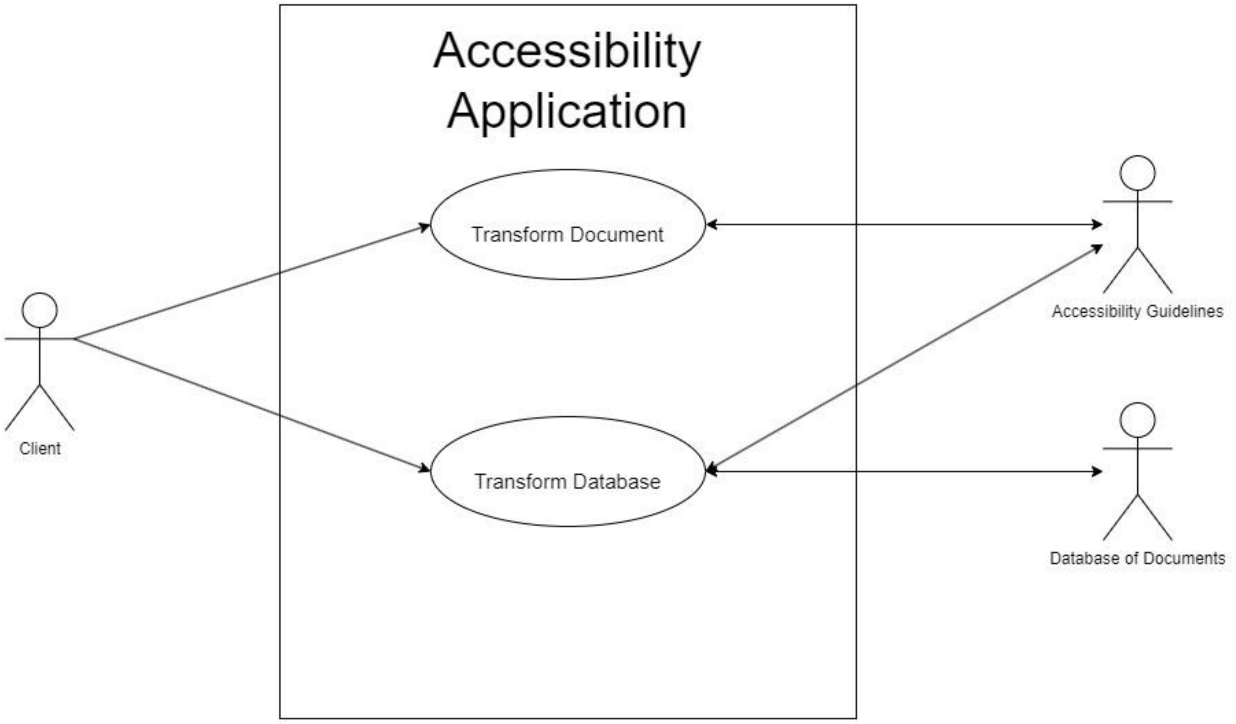
VI.1. Use Cases

Individual Documents

The user starts with a new document, void of prior accessibility checks and filters, and when the user desires to enter it into the database they are then queried by the software, if they would like to transform the document into an accessible version.

The Database

The user desires to transform not just a single document but a segment or all of the database into accessible format. Given desired parameters for database querying, the user will click a button and let it run through the database, taking in documents, and transforming each one of them into an accessible version.



VI.2. Functional Requirements

VI.2.1. Download documents

Obtain Documents for Analysis: We need to be able to access the backend database that contains the many documents for accessibility transformation.

Source: Our client, the WSU libraries, would appreciate the ability to use this accessibility translation software for many documents quickly, not just on a single document. This will require dynamic document retrieval from the database for later translation.
Priority: Priority Level 0: Essential and Required functionality

VI.2.2. Exporting documents

Export Documents to PDF: The software must be able to export documents as to pdfs after all intermediary processing is done. This happens at the end of the process.
Source: WSU research exchange users are our primary stakeholders for this functionality. Creating a readable document for them is important, one that is visually appealing in addition to being accessible, which is the point of the project, is important. Ideally, the outputted document should be equal or better in visual quality to the input document gathered from the research exchange repository.
Priority: Priority Level 0: Essential and Required functionality

VI.2.3. Document metadata

Configuring Metadata from Documents: The application needs to be able to translate all documents into a readable, programmable, and transformable format. This means transforming the data from its visual representation into a text and numerical description that describes the nature of the document, its metadata. No matter the content of the document, this procrustean format will allow quick identification of accessibility errors and later correction.
Source: It is important for the people using our software at the WSU libraries that the speed of this process, the modularization of metadata, accessibility detection and correction, are expedited. If the user wishes to transform an entire database, we need to ensure that our software is rigid, correct, and fast; and this can be ensured by a normalized metadata format.
Priority: Priority Level 0: Essential and Required functionality

VI.2.4. Document tags

Providing Tags for Document Content: Tagging documents involves marking content with labels such as a header, paragraph, row entries within a table, etc. Tagging helps define the reading order, which is another accessibility feature we aim to provide, especially in tables, as well as being used to define alt text for images.
Source: The presence of tags within a document like a PDF allows for clear identification of document elements, which is a mandated accessibility requirement by W3C; a requirement from our client, Talea, at the WSU libraries. Tags are also important to users of the WSU library research exchange because they allow for things like screen readers to function properly.
Priority: Priority Level 0: Essential and Required functionality

VI.2.5. Reading order

Document Reading Order: Our application needs to be able to specify a reading order within the PDF output documents for use with screen readers.
Source: Users of the WSU research exchange who are reading through the documents on the repository need to have a specified reading order to use screen readers effectively. This functionality will satisfy that requirement.
Priority: Priority Level 0: Essential and Required functionality

VI.2.6. Color contrast

Sufficient Color Contrast: Outputted documents should follow a standard of color contrast that is easy to make out words on the page. This requirement is specifically all black text on an all white background for the most effective contrast.
Source: Color contrast is important to users of the WSU research exchange, especially those who have impaired vision or experience difficulty deciphering text with a similar color to the background. The color contrast functionality of our software also fulfills a W3C requirement, which has been requested by our client, Talea, at the WSU libraries.
Priority: Priority Level 0: Essential and Required functionality

VI.2.7. Alternative text on images

Image Alternative Text: The feature of adding alternative text to images would be nice to have if possible, but would probably require a significant amount of effort for a functionality that isn't the most important. It would result in having text embedded in image data in the PDF that describes what is the content of the image [11].
Source: Alternative text is a feature that would be appreciated by users of the WSU research exchange, specifically those who use screen readers, who have difficulty getting the full depth of information from an image. It is also a requirement specified by the W3C standard, which has been requested by our client, Talea, with the WSU libraries.
Priority: Priority Level 2: Extra feature or stretch goal

VI.2.8. Uploading documents to the repository

Document Uploading: One potential functionality for our software is uploading PDF documents back to the WSU research exchange repository after accessibility conversion takes place. Our client, Talea, has mentioned that this feature may not be possible to implement so we will look into it if we have extra time later.
Source: This feature would be helpful for employees at the WSU libraries to reduce their manual time to upload documents after accessibility conversion has been completed.
Priority: Priority Level 2: Extra feature or stretch goal

VI.3. Non-Functional Requirements

V.3.1. Must create pdfs that are accessible

Create Accessible PDFs: PDFs created must follow the accessibility guidelines outlined in the Web Content Accessibility guidelines with a focus on metadata, tags, reading order, color contrast and alternative text for images.

V.3.2. Must work with the WSU research exchange repository

Compatibility with Research Exchange: The system must be able to export documents from Research Exchange, WSU's repository, so it has to be compatible with Research Exchange repository.

V.3.3. Able to run autonomously overnight

Run Autonomously: The system shall be autonomously able to create accessible PDFs while left to run overnight.

V.3.4. Files less than 2 Gb

Produce Documents Under 2 GB: Documents produced by the system must be under 2 giga-bytes in size as per storage requirements set by WSU Libraries.

V.3.5. Works on windows machines

Run on Windows: WSU Libraries primarily uses PCs which run the Windows operating system, so the system will be compatible with Windows to ensure our client is able to use it to the fullest extent.

V.3.6. Response time not important/relevant

No Relevant Response Time: Talea has asked us to create a system that the WSU Libraries faculty can run overnight. Thus, response time is not extremely relevant.

V.3.7. For product delivery, send the executable file

Send Executable File: System shall be sent to WSU Libraries in the form of an executable file upon completion of coding and testing.

VII. System Evolution

Some of the fundamental assumptions of our project include the following. One assumption we have made is that we are using the WSU research exchange to get our pdf data. We do not anticipate this changing as we are creating a specialized application for the libraries specifically, but it is an assumption our project is based on. Another assumption is that the program will be run on windows machines. The library has mostly or all windows machines so this will also probably not change, but it is an assumption we will be basing off of to do windows based development. Also, we are assuming that we are unable to upload pdfs to the research exchange, which is why we are holding off on that feature. One last assumption we are basing our project off is that we are unable to work with the PDFs to make them more accessible without recreating them. This is why we are harvesting the data and then recreating the pdf. We did look into the adobe API but it seemed inconvenient and less functional than what is possible with recreating it.

Some points of risk in our project include the following. There is a risk of overwriting data on the repository if we do ever implement uploading, also with the possibility of uploading improper data that loses information, so we will need to be careful about double checking our outputs until we are confident in the quality. Another risk is that the PDF code libraries we are planning to use may not have all needed functionality. From our research, it does seem that the functionality is there, but when we start developing our application it may turn out that some things are not included.

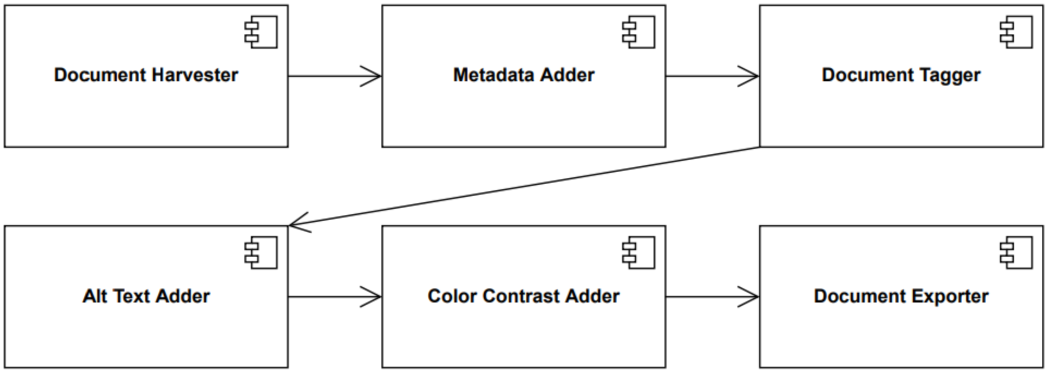
VIII. System Overview

The system will download PDFs from the Research Exchange website, create a new PDF with correct metadata, tags, reading order, color contrast and alternative text for images and then export a new PDF with the corrected accessibility pieces. These pieces of accessibility will be decomposed into three subsystems which are suitable work for one programmer to complete during the project’s creation. Due to the functionality of the system, which will do one functional requirement at a time in a set sequential order, we have chosen to use the pipe and filter architectural model to implement our system. The design of the system will be fairly basic since there is not going to be much interaction between the user and the system aside from obtaining search parameters from the user to determine which document to start with when transforming a segment of documents from Research Exchange, choosing between transforming individual documents or a segment of documents and pressing either the start or pause button.

IX. Architecture Design

IX.1. Overview

The architectural model we have selected is the pipe and filter model. The pipe and filter model involves a series of transformations done on some data like an assembly line to create an overall larger transformation [12]. We have selected this model for our project because there are many different transformations our data needs to go through, those being accessibility features such as adding metadata or alt text, to produce one overall transformation of making the document more accessible. Each of our functional requirements should map to a subsystem, where each subsystem is a transformation or filter in the overall pipeline of our model.

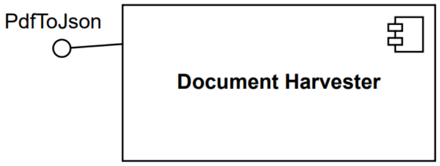


An overview of the components are as follows. The document harvester will pull documents from the repository and translate them into json files. The metadata adder will add metadata to the json file for the document. The document tagger will add tags to the document data. The alt text adder will add alternative text to images in the document. The color contrast adder will add color contrast to the document data. The document exporter will export the document back to a pdf.

IX.2. Subsystem Decomposition

The subsystems are composed into pieces which each perform one transformation on the document data, be that generating it, adjusting it, or exporting it. These highly separate components hand off data to the next component and therefore have almost no coupling since they each do something very unique. The building of data is done in a very serial way so there is some data dependency on that front, but the components themselves are not really dependent upon each other at all besides that they are independent.

IX.2.1. Document Harvester



a) Description

The document harvester will extract documents from the WSU research exchange library and convert them into json files. The produced file will include all text and images from the original document with limited information on the formatting of the document (depending on how accessible the starting document was).

b) Concepts and Algorithms Generated

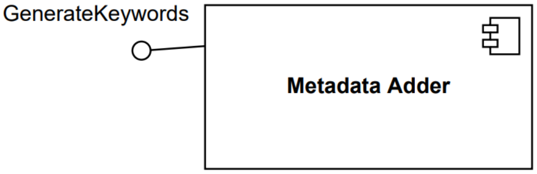
The research exchange provides web endpoints that can be interacted with using GET commands or something similar to them. It is used to grab metadata and documents.

c) Interface Description

<u>Services Provided:</u>

Service Name: PdfToJson(DocumentEndpoint)
Service provided to: Document Harvester [Back-End]
Description: Returns a JSON file containing data within the PDF file.

IX.2.2. Metadata Adder



a) Description

Adds a metadata section to the document data including details such as the title, author, subject, keywords, and document language.

b) Concepts and Algorithms Generated

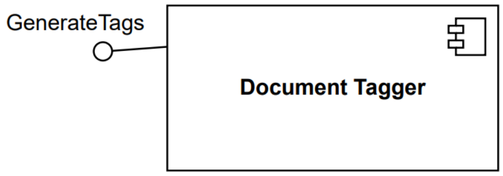
The keywords, if not already harvested from the original document, will be generated by finding commonly occurring words in the document that aren't common in most documents. This looks like picking a keyword that pops up a lot while excluding words like “and” and “the”.

c) Interface Description

Services Provided:

Service name: GenerateKeywords()
Service provided to: Metadata Adder [Back-End]
Description: Finds commonly occurring words in the document to generate keywords.

IX.2.3. Document Tagger



a) Description

Adds a tag section to the document. Each document will have different tags based on the content within. Examples of tags include paragraphs, headings, figures, lists and tables. Some of these tags indicate the reading order for use in things such as screen readers.

b) Concepts and Algorithms Generated

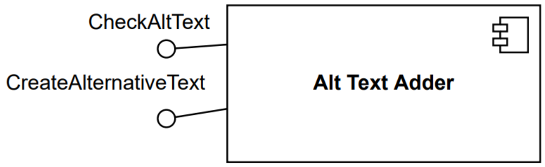
Tags identify the type of content in a PDF and they also store some attributes about the contents. Tags also arrange a hierarchical architecture within the PDF which provides the reading order of the document.

c) Interface Description

Services Provided:

Service name: GenerateTags()
Service provided to: Tag Adder [Back-End]
Description: Identifies the types of content in a document, stores information about each type of content contained within the document and creates a structured reading order based upon this content.

IX.2.4. Alt Text Adder



a) Description

Checks through the images within the document and when necessary adds alternative text to describe the image.

b) Concepts and Algorithms Generated

This will require machine learning algorithms to take in image input and create an intuitive enough understanding of what is going on in the image to provide a description for it. This will probably need to be out-sourced or provided through datasets of images large enough to encapsulate the types of images that will be found within the repository.

c) Interface Description

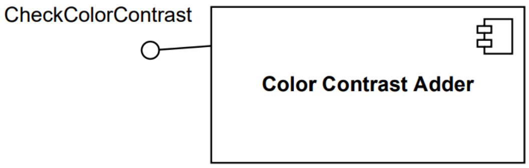
Services Provided:

Service name: CheckAltTest()
Service provided to: Alt Text Adder [Back-End]
Description: Checks each image within a document, which can be found through the document's metadata, and checks to see if there is alternative text provided. If not, it runs an algorithm to generate the appropriate text.

Service Name: CreateAlternativeText(IMAGE_DATA)
Service provided to: Alt Text Adder [Back-End]
Description: Runs the machine learning algorithm (already trained), and returns the generated alternative text that represents the given image.

Services Required:
This requires the Document Harvester's codable metadata extrapolated from PDF. It will also require out-sourced machine learning datasets or libraries.

IX.2.5. Color Contrast Adder



a) Description

Checks through the document text and compares and contrasts between text color and background color and ensures that the difference is adequate to W3C standards. If this is not the case then it changes the metadata, changing either the text color or background so that the contrast is adequate.

b) Concepts and Algorithms Generated

This will require checking through the metadata where text is located and the background at that location. We will need an algorithm that checks both constraints at all points verifying and making changes where needed.

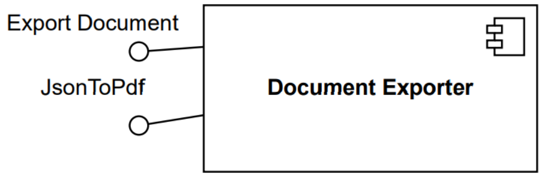
c) Interface Description

Services Provided:

Service name: CheckColorContrast()
Service provided to: Color Contrast Adder [Back-End]
Description: Given a document and its metadata, checks all text and background contrast and fixes it when it is not in-line with W3C standards.

Services Required:
This requires the Document Harvester's codable metadata extrapolated from PDF.

IX.2.6. Document Exporter



a) Description

Transforms the metadata from codable data structures back into a usable and readable format: PDF. This document, now fully transformed with accessibility components, is exported or returned to either the end user or back to the database.

b) Concepts and Algorithms Generated

This will require a back propagation or a reverse implementation of document harvesting, where we translate a PDF into a JSON file and other codable formats. We will then need to add database functionality or a means of returning the new document to its original place.

c) Interface Description

Services Provided:

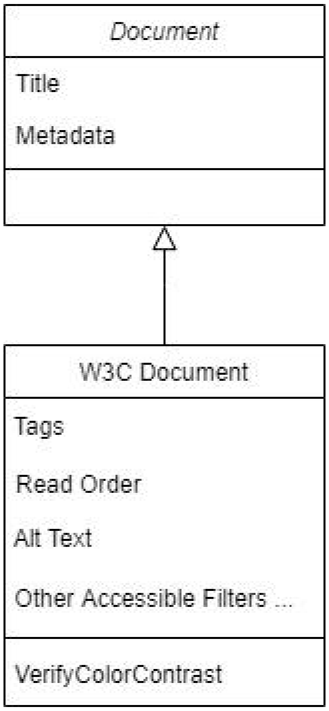
Service Name: ExportDocument(DESTINATION)
Service provided to: Document Exporter [Back-End]
Description: Sends the document back to the receiver

Service Name: JsonToPdf()
Service provided to: Document Exporter [Back-End]
Description: Returns a PDF file manifested by the data within the JSON file.

X. Data Design

We will need to transform a PDF formatted file into a programmable object, and to do this, we will first translate the PDF data into JSON format. We will then parse this data and represent the JSON attribute pairs as a dictionary. To manage and manipulate each PDFs metadata and corresponding JSON dictionary we will need to use a data structure.

We can use object-oriented programming to represent each respective document that goes through our software as class objects. Regardless of if the document meets the accessibility guidelines of W3C, we can substantiate each PDF as a document class. Before this object goes through our assembly line filters, it will need to be transformed into a child class, a W3C Document, which inherits the values of the document but wrapped with accessibility checks. Our filter components that check each accessibility guideline, respectively, will verify that the metadata contains all the accessibility changes it needs, and these checks can be easily managed through a W3C Document's class member variables and functions.



XI. User Interface Design

The user interface design for this project is going to be very basic. Our client, WSU Libraries, has requested the ability to run the program overnight and pause in the morning leaving very little interaction between the program and the user. However, the interaction between the user and the program is very important especially when determining which documents on the Research Exchange website to transform into accessible PDFs.

The program can either be used to transform individual documents, selected by the user, into accessible PDFs or it can be used to transform a segment of documents from the Research Exchange website into accessible PDFs. The initial user interface will provide two buttons, one to transform individual documents and one to transform a segment of documents into accessible PDFs.

In the case of transforming individual documents into accessible PDFs, the user interface will ask the user for the document and ask if the user is sure they want to transform the document into an accessible PDF. If the user selects the 'yes' option, a new and accessible PDF will be created for the user. If the user selects the 'no' option, the user will be prompted for another document to transform.

In the case of transforming a segment of documents, the user interface will have a drop down menu that shows the sections of the Research Exchange website and a bar to type in the name of a pdf to start with as well as a start button and a button to go back to the home page. After either selecting the section of Research Exchange to start with or the pdf name, the user will click the start button. After selecting the start button, the user will be prompted to provide parameters, such as which section of the Research Exchange website to start in or which document to start on, and select the run button which will then pull PDFs from the Research Exchange website based on the user's parameters and create a new, accessible PDF, from the data collected with correct reading order, metadata, tags, color contrast and alternative text for images. The program will do this without user input, so during this time there is no interaction between the user and the program. The user interface will display a pause button and the program will run until the user selects the pause button to stop the program. After the program is paused, the user will be able to either start the program again, starting from where it left off, or go back to the main screen to choose between transforming individual documents and transforming a segment of documents.

After the project is completed, WSU Libraries will be given a flash drive containing an executable file which they can run on their computers. This executable file will be the program. They will not need to install the program, simply transfer the executable file from the flash drive to their computer and open the file to start the program. The user can open the executable file in several ways including double clicking the file after transferring it to the new computer, selecting the file from the start menu, or right clicking on the program and selecting the run option [13]. Upon opening the program for the first time the user's computer will likely ask for permission from the user to run the program. After the program is opened the user will be able to select the start button to begin pulling PDFs from the Research Exchange website and creating new, accessible PDFs which they can upload to the Research Exchange website in place of the old PDFs pulled from the site.

See Appendix A for mock ups of each user interface.

XII. Testing Introduction

XII.1. Project Overview

Our program has six major functionalities for which testing is crucial. These six crucial areas we must test are the document harvester, the document exporter, the document tagger, metadata adder, color contrast adder and the alternative text adder. Testing the document harvester means testing that our program is able to harvest information from pdfs. This feature must be able to extract data from pdfs on Research Exchange and from pdfs supplied by the user. The document exporter must be able to create pdfs from the information extracted along with the metadata, tags, color contrast and alternative text created while the program is running. Testing this feature will mean making sure a document is created and also making sure the information is not only present but also correct. The document tagger will create the reading order of the pdf, which is crucial when using text to speech software. The color contrast adder makes sure that all text is black and the background is white, making the document easier to read. The alternative text adder creates alternative text for images which tells the reader why an image is present and uses a text to speech software to inform the reader of this. The metadata adder will make sure the metadata of the pdf is updated to include the author, keywords, subject and other crucial information to give the reader an idea of what the pdf is about. These four features are crucial in making sure the pdf produced with the document exporter is accessible by the WCAG guidelines. Thus, they must be tested to ensure the documents created are accessible.

XII.2. Test Objectives and Schedule

Our approach to testing this program is to use a combination of automated tests and non-automated tests. The automated tests will be used where they can be, which in this case is just unit and integration testing. These two testing sections allow for the use of unittest to test the functionality of each function. In our program, system testing doesn't allow for automated testing. During this testing phase, our team will be testing the program by using the functions to create documents and go through them by hand to make sure each of the requirements is met. During this phase, the functional and nonfunctional requirements will be tested. Since these requirements don't produce results that can be tested automatedly, we must check them ourselves. This means we will use the document harvester to extract information, create the tags, metadata, color contrast, and alternative text then use the document exporter to create a pdf. We will then examine the tags, metadata, color contrast and alternative text of this new pdf to ensure that each feature is present and correct.

Testing our project will not require many resources. A computer capable of running our program, a pre-downloaded pdf for us to input, and a pdf pulled from the Research Exchange website are all the resources required for testing our project.

Our team has chosen to test as we create our program. This means that every time we add a feature to our project we will be testing that feature. Thus, testing will be concluded not long after our project is finished. The project is set to be delivered to the client at the end of this semester. This means that a flash drive containing the executable program along with the documentation will be delivered to the client in December 2022. There are two major milestones left to this project, milestones two and three. Milestone two is scheduled to be completed November 9th, 2022 and milestone three is scheduled to be completed December 9th, 2022.

XII.3. Scope

The scope of this document is the testing plan, testing strategy and environment requirements of our project. The purpose of this document is to outline and describe our team's testing plan for the project. This document is meant to give the reader a deeper understanding of testing procedures we intend to apply during the testing of our project and also explain why we chose these procedures.

XIII. Testing Strategy

1. Identify the requirements to be tested using the System Requirements Specification section of this document.
2. Upon creating new software updates or modifications, develop test cases that have an expected output satisfying the system requirements.
3. Add the test cases to our automated testing program with their input and expected output.
4. Run unit testing on new or revised code before moving on to integration testing.
5. Failed testing that can not be resolved easily requires the opening of a new issue on Github describing the problem and sequence of events leading to it.
6. Before merging any code branch back into the main branch of the repository, all test cases must be passed. This is known as continuous integration testing.

XIV. Test Plans

XIV.1. Unit Testing

We will be using the python unittest testing framework to conduct validity testing on the core components of our application. This includes the pdf extraction module, all of the accessibility transformation components, and our exporter module. Each of these will be tested atomically and independently. For each component we will provide tests with valid and invalid input and analyze, determine, and check the boundary cases for each component. All of our functional components change the data within a PDF document file. We will test our function according to our expectations in transformations of the file given the accessibility component being tested.

XIV.2. Integration Testing

Similar to unit testing we will use the python unittest testing framework to test all our application's components holistically as it moves through the transformation pipeline. Each accessibility transformation component should be errorless and pass testing regardless of which component in the pipeline came before it. Since each component is independent of the other, our integration testing should show this independence by testing the transformation flow with different transformation orders. In the end, since this is a pipeline software structure, there is little variance in entry points, only variance in data. Integration testing is concerned with how each unit works together, and that will depend on each component being strictly limited to its own personalized task.

XIV.3. System Testing

XIV.3.1. Functional Testing

Functional testing will involve reviewing the functional requirements detailed in the System Requirements Specification section and verifying whether the output of our software satisfies them. Our functional requirements are boolean in nature in that they either are or aren't satisfied and the program does or doesn't provide the functionality. For example, whether the software can download documents from the Research Exchange repository or whether it adds tags to the outputted pdf document. These can easily be verified manually by opening the resulting output and reviewing it visually so automated testing for the overall functional requirements will not be necessary.

XIV.3.2. PerformanceTesting

Performance testing will involve reviewing the non-functional requirements detailed in the System Requirements Specification section and verifying whether the properties of the software are in accordance with them. Most of these requirements can be easily verified manually, including whether the software runs on Windows or is able to run autonomously. The only non-functional requirement that can be tested through automated means is whether output files are less than 2 gigabytes, so that requirement will be integrated into our automated testing program.

XIV.3.3. User Acceptance Testing

User acceptance testing will depend not on just the completion of functional components but the non-functional requirements and the satisfaction of our client. Throughout development we will hold many demos where we will demonstrate the progression of the project and our closing success on providing an application and can provide simple use to the important task of making documents more accessible. Our demos early on will use a CLI, command line interface, to showcase our pipeline and the outputted documents. Later on, we will have a more friendly front-end GUI, graphical user interface, for our client to use. For user acceptance testing, we will rely on the feedback and confirmation by our client during demos that the transformed documents are meeting up to their professional expectations.

XV. Environment Requirements

A test environment is where application testing is conducted to find and fix errors [14]. The testing environment for our project must be able to use unittest to run automated tests. This does not require any special hardware. In python programming, unittest can be used simply by importing the unittest package. This allows us to use the associated testing methods like test fixture, test case, test runner, and test suite to run automated tests on the program [15]. In order to run the unittest package, the testing environment will need to include software that is able to run the python programming language. This includes an integrated development environment that supports python, which, in our case will be Visual Studio Code. This can be downloaded on any computer with internet access, so this will not place a limit on the testing environment.

XVI. Test Results

As of right now, testing our program is done with unittest. As we implement features we create unit tests to test the components found within each feature. The current prototype of our project has implemented unit tests for the pdf extractor and the tag tree features. There are seven unit tests for the pdf extractor of our current prototype, all of which pass. They test the extraction of paragraphs, the data exported to html, extracting fonts and sizes and exporting to html as well as testing whether two paragraphs are equal. There are eight unit tests implemented for the tag tree feature of our current prototype, all of which pass. They test the initialization of a tag tree, the traversal of a tag tree, getting a tag, creating a child, creating a tag and then accessing it as well as creating a tag and accessing it from the previous tag.

XVII. Projects and Tools Used

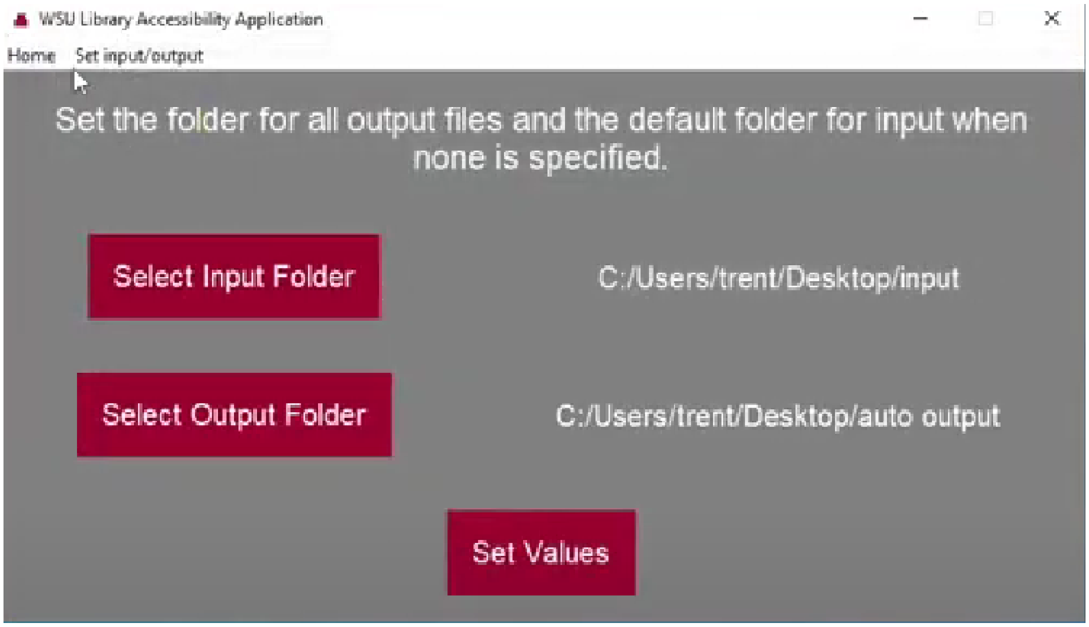
Tools/library/framework	Purpose
pdfminer.six	Extracting text from pdf files
Beautiful Soup	Intermediary html formatting
PyPDF2	Writing metadata to exported pdf files
Detectron2	Neural Network algorithms for document layout detection
Puppeteer	Writing html to pdf
npm	Running node js file

Languages Used		
Python	HTML	Node JS

XVIII. Description of Final Prototype

Our software takes a pdf document through a three-step process: Import and extraction, data transformation, and packaging and exporting. Our application is designed to be a streamlining tool, of which the WSU libraries can use to expedite the accessibility transformation process. We created a simple user interface that allows us to easily send unprocessed documents through our transformation pipeline. Through our UI we can dynamically set the input and output directories of these documents. This is for the purpose of running smaller batches of documents on our local computer and directly analyzing the transformation results.

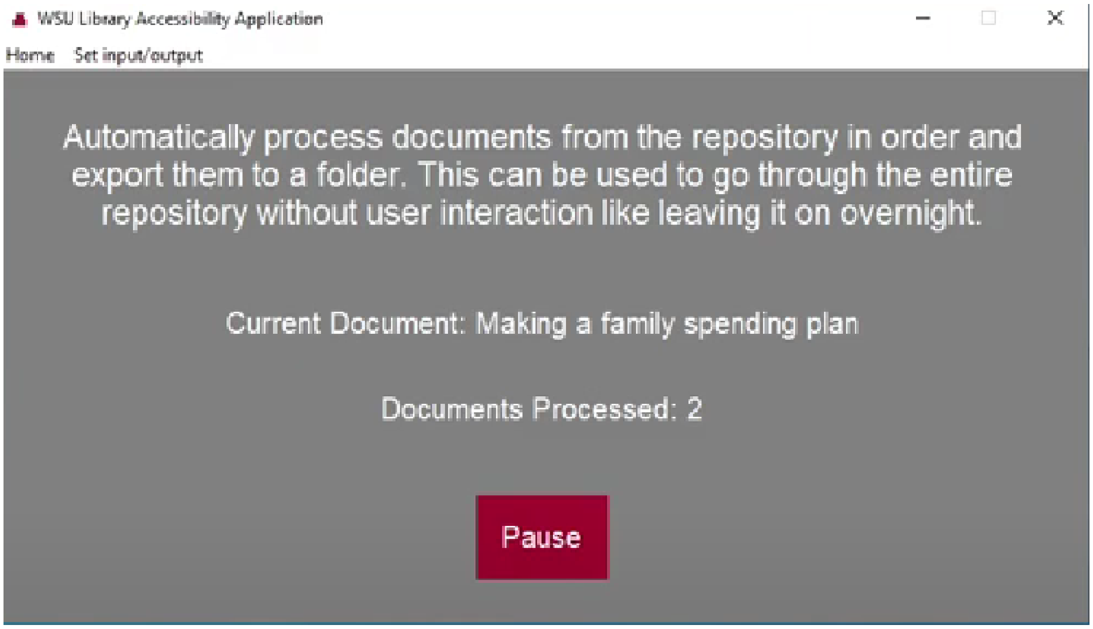
Our user interface also allows us to perform automatic document processing which should allow us to retrieve digital documents directly from the research exchange and run them through our transformation pipeline. In theory these should be sent back to the repository, but for now they are saved in the locally assigned output directory.



Set the folder for all output files and the default folder for input when none is specified.

C:/Users/trent/Desktop/input

C:/Users/trent/Desktop/auto output

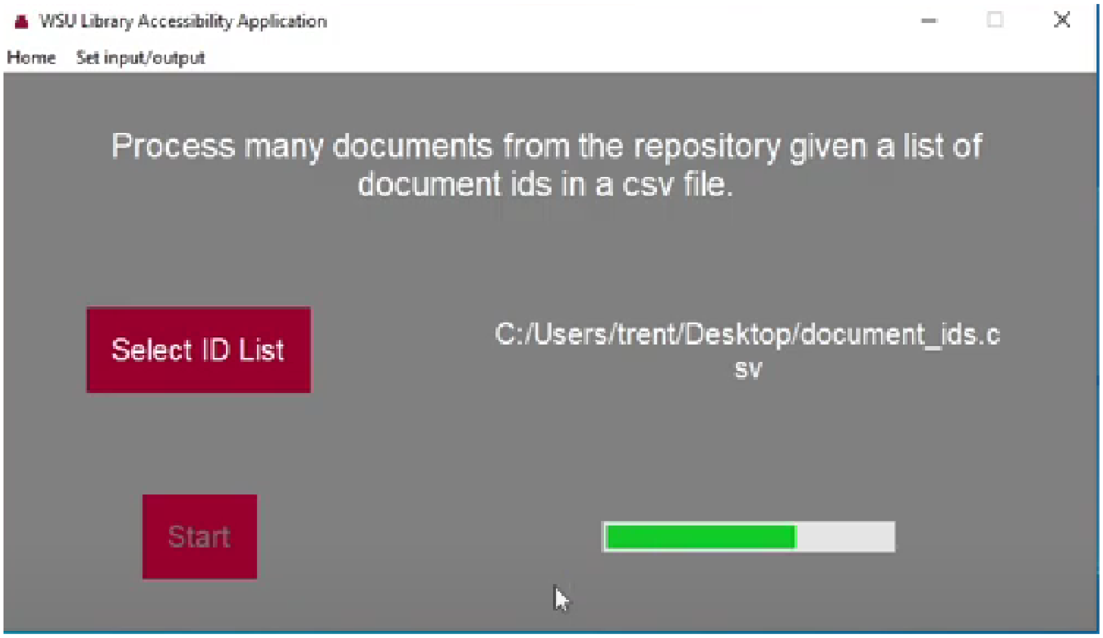


Automatically process documents from the repository in order and export them to a folder. This can be used to go through the entire repository without user interaction like leaving it on overnight.

Current Document: Making a family spending plan

Documents Processed: 2

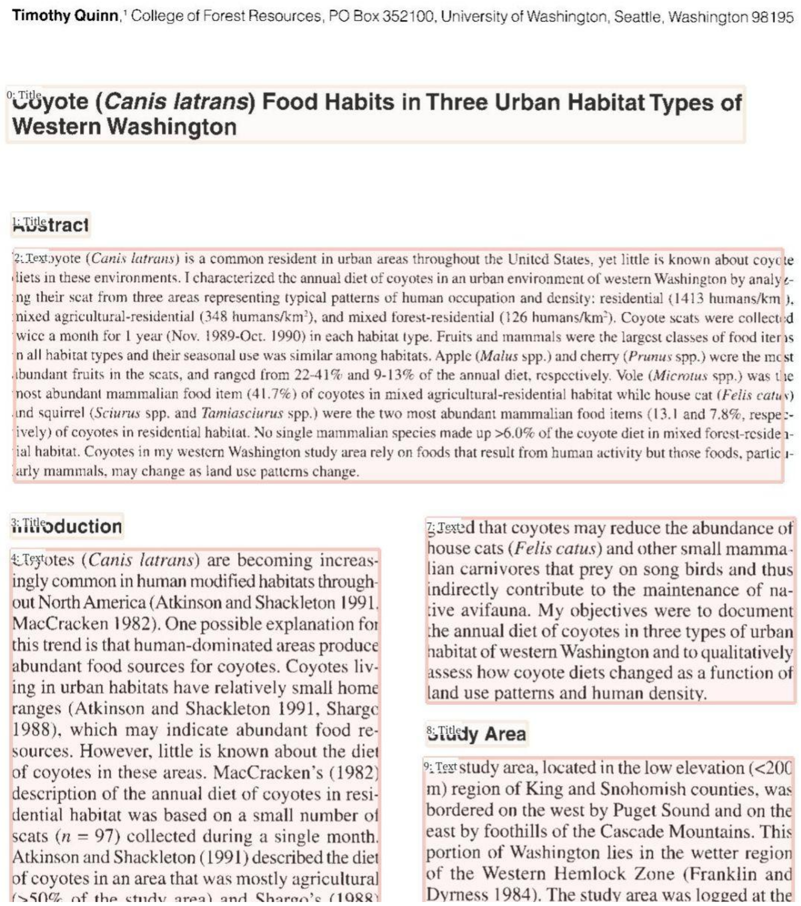
The user interface also provides options to run specified transformation batches. Through csv files which contain row data about documents on the exchange such as their document ID and their titles, we can target specific documents on the repository rather than transforming random documents.



Process many documents from the repository given a list of document ids in a csv file.

C:/Users/trent/Desktop/document_ids.csv

While this is the front-end application for the librarians, we have various debugging tools or tests in place that allow us to refactor or update the system methodically. We have our unit tests which can be run in its totality by running the python unittest function on the test directory. We also have various tools that act as a debugging log for pipeline components. Since our document harvester extracts PDF data through a process of snapshots and the use of AI models, we have images of what that looks like mid-process to show us what the AI is observing. This gives us a sense of where the document harvester may be running into errors. The image below shows what that looks like, where machine learning has used image processing to find paragraph elements in the PDF while also providing an index number with it, giving us the read order of these paragraph layout blocks.



Timothy Quinn,[1] College of Forest Resources, PO Box 352100, University of Washington, Seattle, Washington 98195

Coyote (*Canis latrans*) Food Habits in Three Urban Habitat Types of Western Washington

Abstract

oyote (*Canis latrans*) is a common resident in urban areas throughout the United States, yet little is known about coyote diets in these environments. I characterized the annual diet of coyotes in an urban environment of western Washington by analyzing their scat from three areas representing typical patterns of human occupation and density: residential (1413 humans/km), mixed agricultural-residential (348 humans/km^2), and mixed forest-residential (126 humans/km^2). Coyote scats were collected wice a month for 1 year (Nov. 1989-Oct. 1990) in each habitat type. Fruits and mammals were the largest classes of food items n all habitat types and their seasonal use was similar among habitats. Apple (*Malus* spp.) and cherry (*Prunus* spp.) were the most abundant fruits in the scats, and ranged from 22-41% and 9-13% of the annual diet, respectively. Vole (*Microtus* spp.) was the most abundant mammalian food item (41.7%) of coyotes in mixed agricultural-residential habitat while house cat (*Felis catus*) and squirrel (*Sciurus* spp. and *Tamiasciurus* spp.) were the two most abundant mammalian food items (13.1 and 7.8%, respectively) of coyotes in residential habitat. No single mammalian species made up >6.0% of the coyote diet in mixed forest-residential habitat. Coyotes in my western Washington study area rely on foods that result from human activity but those foods, particularly mammals, may change as land use patterns change.

Introduction

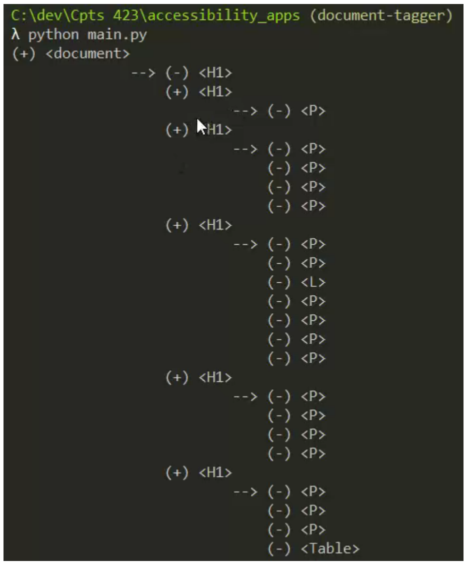
Coyotes (*Canis latrans*) are becoming increasingly common in human modified habitats throughout North America (Atkinson and Shackleton 1991, MacCracken 1982). One possible explanation for this trend is that human-dominated areas produce abundant food sources for coyotes. Coyotes living in urban habitats have relatively small home ranges (Atkinson and Shackleton 1991, Shargo 1988), which may indicate abundant food resources. However, little is known about the diet of coyotes in these areas. MacCracken's (1982) description of the annual diet of coyotes in residential habitat was based on a small number of scats ($n = 97$) collected during a single month. Atkinson and Shackleton (1991) described the diet of coyotes in an area that was mostly agricultural (>50% of the study area) and Shargo's (1988)

ed that coyotes may reduce the abundance of house cats (*Felis catus*) and other small mammalian carnivores that prey on song birds and thus indirectly contribute to the maintenance of native avifauna. My objectives were to document the annual diet of coyotes in three types of urban habitat of western Washington and to qualitatively assess how coyote diets changed as a function of land use patterns and human density.

Study Area

study area, located in the low elevation (<200 m) region of King and Snohomish counties, was bordered on the west by Puget Sound and on the east by foothills of the Cascade Mountains. This portion of Washington lies in the wetter region of the Western Hemlock Zone (Franklin and Dyrness 1984). The study area was logged at the

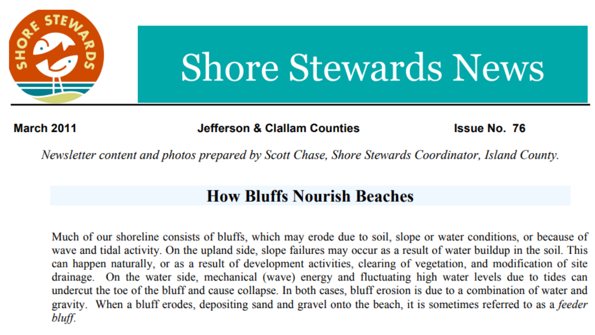
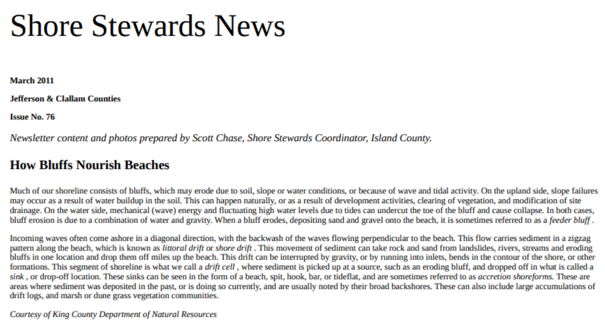
Through the terminal we can also test or observe the output of our accessibility transformation components. For example, we can visualize the constructed tag tree structure from these documents, which will eventually be embedded into the new document itself.



```
C:\dev\Cpts 423\accessibility_apps (document-tagger)
λ python main.py
(+) <document>
              --> (-) <H1>
                  (+) <H1>
                          --> (-) <P>
                  (+) <H1>
                          --> (-) <P>
                              (-) <P>
                              (-) <P>
                              (-) <P>
                  (+) <H1>
                          --> (-) <P>
                              (-) <P>
                              (-) <L>
                              (-) <P>
                              (-) <P>
                              (-) <P>
                              (-) <P>
                  (+) <H1>
                          --> (-) <P>
                              (-) <P>
                              (-) <P>
                              (-) <P>
                  (+) <H1>
                          --> (-) <P>
                              (-) <P>
                              (-) <P>
                              (-) <Table>
```

As a final prototype, we have a working user interface that can process inaccessible documents directly from the repository, which can be done randomly or through selected batches with csv files. We can also process local digital documents by dynamically setting our input and output directories. Beyond the UI, we have unit tests to test each component in the pipeline individually. We also created code that can output mid-processing data in log-like formats for adhoc testing. This includes snapshots of PDF pages, which showcases what the machine learning models are doing, but also tag tree utility calls that allow us to see what the tag tree generation looks like before we output the new accessibility document.

Once a digital document goes through the whole pipeline, the new accessible document should be sent back to the research exchange. For now, it is saved in a local directory. A before and after example is shown below.

BEFORE	AFTER
 Shore Stewards News March 2011 Jefferson & Clallam Counties Issue No. 76 *Newsletter content and photos prepared by Scott Chase, Shore Stewards Coordinator, Island County.* **How Bluffs Nourish Beaches** Much of our shoreline consists of bluffs, which may erode due to soil, slope or water conditions, or because of wave and tidal activity. On the upland side, slope failures may occur as a result of water buildup in the soil. This can happen naturally, or as a result of development activities, clearing of vegetation, and modification of site drainage. On the water side, mechanical (wave) energy and fluctuating high water levels due to tides can undercut the toe of the bluff and cause collapse. In both cases, bluff erosion is due to a combination of water and gravity. When a bluff erodes, depositing sand and gravel onto the beach, it is sometimes referred to as a *feeder bluff*.	 Shore Stewards News **March 2011** **Jefferson & Clallam Counties** **Issue No. 76** *Newsletter content and photos prepared by Scott Chase, Shore Stewards Coordinator, Island County.* **How Bluffs Nourish Beaches** Much of our shoreline consists of bluffs, which may erode due to soil, slope or water conditions, or because of wave and tidal activity. On the upland side, slope failures may occur as a result of water buildup in the soil. This can happen naturally, or as a result of development activities, clearing of vegetation, and modification of site drainage. On the water side, mechanical (wave) energy and fluctuating high water levels due to tides can undercut the toe of the bluff and cause collapse. In both cases, bluff erosion is due to a combination of water and gravity. When a bluff erodes, depositing sand and gravel onto the beach, it is sometimes referred to as a *feeder bluff* . Incoming waves often come ashore in a diagonal direction, with the backwash of the waves flowing perpendicular to the beach. This flow carries sediment in a zigzag pattern along the beach, which is known as *littoral drift* or *shore drift* . This movement of sediment can take rock and sand from landslides, rivers, streams and eroding bluffs in one location and drop them off miles up the beach. This drift can be interrupted by gravity, or by running into inlets, bends in the contour of the shore, or other formations. This segment of shoreline is what we call a *drift cell* , where sediment is picked up at a source, such as an eroding bluff, and dropped off in what is called a *sink* , or drop-off location. These sinks can be seen in the form of a beach, spit, hook, bar, or tideflat, and are sometimes referred to as *accretion shoreforms*. These are areas where sediment was deposited in the past, or is doing so currently, and are usually noted by their broad backshores. These can also include large accumulations of drift logs, and marsh or dune grass vegetation communities. *Courtesy of King County Department of Natural Resources*

XVIII 1. Document Harvester

XVIII 1.1 Functions and Interfaces implemented

We have been able to successfully take a given PDF document and extrapolate it into datasets represented by html dictionaries. We are able to test the effectiveness of this extraction through html files. We use a lot of transformational functions with various parts of the data. To do this, we have implemented a Document class data structure in python that stores these data representations of the document in Paragraph objects that hold text and font data. We have created this class and it is used throughout the application.

XVIII 1.2 Preliminary Tests

We have created unit tests for each of the aspects of the document harvester including reading the document as Paragraph objects and turning those into an html.

XVIII 2. Document Tagger

XVIII 2.1 Functions and Interfaces implemented

This is one of our pipeline functions. It is one that has been the most implemented to this point. We have created a custom data structure, Tag Tree, that acts like a linked list but is built to read document segments in order. The Tag Tree is added to the document class post-function.

XVIII 2.2 Preliminary Tests

We have some automated unit test functions for testing the TagTree building, traversal, and reading.

XVIII 3. Document Exporter

XVIII 3.1 Functions and Interfaces implemented

We have mainly made progress in reading html documents back into PDF format. We have not yet connected this with our Document class but that link should not be a far step.

XVIII 3.2 Preliminary Tests

We have not yet created the unit tests to validate our code for PDF conversion but have custom test functions to check sampled input.

XVIII 4. CLI Application

XVIII 4.1 Functions and Interfaces implemented

As our early front-end application, we created a command-line interface. This CLI allows us to import a local document and run it through our transformation pipeline, and then export it. It gives the user options on executing this. It uses the Document class and calls all the appropriate functions. However, many of those are currently just pass functions.

XVIII 4.2 Preliminary Tests

At this time we do not currently have any unit tests to test the CLI, but we have tested it with sampled input.

XIX. Project Delivery Status

Our project was delivered to our client on Tuesday, April 25th. The project was delivered to Talea Anderson in her office in the Terrell Library. In addition to the delivery of the project, she was given instructions on how to download and use the project. The project was demonstrated and explained in detail to Talea during our final meeting on Thursday, April 27th.

Our project currently exists on each of our laptops, Talea's computer in the library and our GitHub page from this course. The source repository can be found at the following link: https://github.com/WSUCptSCapstone-Fall2022Spring2023/wsulibraries-accessibilityapps. There is no course or project material outside of our personal laptops, the course's canvas page, the library computer or the GitHub repository.

To rebuild the project from the ground up, you would just need access to a computer and access to the GitHub repository. In order to set up the project once you have these two things, you would just need to click the code option in GitHub, download the zip file and open the file. From here, the user will need to double click the setup.bat, which will download the necessary pieces of the project. After this has finished running, anytime the user wants to run the project they will just need to double click run_application.bat to run the project.

XX. Conclusions and Future Work

XX.1. Limitations and Recommendations

The main limitations of our project are found within the formatting of the document exporter and tagging the document. One other known limitation of our project is that it only works with PDF files.

The limitations in exporter formatting include placing images back in the document and general formatting. As of right now, the exporter cannot place images back into the accessible pdf. This can be solved by including the figure tag in the exporter function and using the coordinates of the original picture to place the image back into the PDF. Along with this limitation, there are also some general formatting issues with the exporter. For example, there are sometimes weird characters thrown in and the way the paragraphs are placed into the accessible PDF is very basic and could look more professional. Potential fixes for this include using different OCR to create the tag tree, creating a function to verify that the characters are letters, numbers, or make sense in the current context, although this would probably use machine learning, and updating the layout of the exported document. Updating the layout of the exported document could also be done within the tagging of the document since the exporter uses the tag tree to create the accessible PDF, this, however, would mean that the way a document is tagged would change. However, different OCR might also fix this issue.

The main limitation of the document tagger is that while it recognizes an image and has a tag for a figure, this feature has not been implemented so images are not dealt with. This could easily be fixed by adding the code to implement this. Doing so will also allow the exporter to put images into the accessible PDF and would help with the formatting as well.

When our team was first given this project, we were told we could focus on one type of document to transform. During this time, our team decided to focus on PDF files. Because of this, the program does require and produce PDFs. In order to change this, the exporter would have to have a function specifically for different file types and produce the same file type given. Or have an option to create a PDF based on the original file given, which then would mean that the user interface would just need to accept other file types.

One recommendation our team has for this project is to run the project on windows computers only, since that was the system it was created for. If you try to run the project on a different system you will just end up getting system based error codes and you won’t be able to use the project.

XX.2. Future Work

There are many different areas our project could be improved upon if taken up by others in the future. Here are some of those areas as well as details on what implementation might look like. The keyword generation could be improved as part of the metadata adder by using a machine learning solution. This would require gathering a dataset of words from the repository which are representative of the types of documents on there, and then comparing word frequency within a document to that of the word frequency of the word dataset. Another thing that can be improved is adding images to exported documents. This would require a fairly light amount of work in the document harvester to grab images from the original document using the same system as paragraph detection and then export the images to intermediary files or storing their data for use later in the exporting process. One feature which could be added that would actually be quite complex would be adding alternative text to images, which is a description of the contents of each image embedded into the document data. This requires a significant amount of machine learning type processing involving a large dataset to properly identify the content of images. Realistically, this would probably involve using a cloud service with the dataset and processing power, but there is potential to download a dataset and use that (or to have a pre-trained algorithm). Another feature which can be improved on in the future is exporter formatting. Currently the application exports documents in a very basic looking way and it could be improved with a different font or formatting style (also incorporating pictures into it well). Uploading documents to the repository automatically after processing them is another feature which could be added. This is not very feasible with the current setup of the research exchange because their system requires batch uploads and some manual process type things. These would have to be automated (which would be helpful in general) for automatic uploading from our application to be added. One last feature to mention here that can be added to our project is more intentional processing for tables. Something that actually adds them as a table object in the output as opposed to just raw text extraction or saving it as an image. Those are some of the ways our project can be improved upon and further developed in the future.

XIX. Acknowledgments

Our team would like to thank WSU Libraries for supplying us with this project. We would also like to thank Talea Anderson for meeting with us weekly to discuss the project and its desired attributes. Finally, we would like to thank our professor, Ananth Jillepalli , for his coaching meetings and for providing helpful documentation and feedback throughout the course of our project.

XVIIII. Glossary

Accessibility - When websites, web tools, and software are properly designed and coded, it allows for people with disabilities to use them. W3C, World Wide Web Consortium, provides standards or expectations on how digital media should be presented to ensure those with disabilities can still gain full advantage and understanding of the material. Accessibility in digital media is how well the given software is in accordance with W3C accessibility standards.

Data mining - A process of discovering and analyzing patterns within large data sets. This is done through machine learning, statistics, and data collected through database systems.

Executable file - A program that can be run with a set of instructions or options to make it do something on a computer.

JSON - JavaScript Object Notation; it is a type of file that specifies an object's attributes.

Metadata - Data that provides information about data. This allows us to retrieve descriptive information about a file without looking at the content.

XIX. References

[1] *Rex.libraries.wsu.edu*. [Online]. Available: https://rex.libraries.wsu.edu/esploro/. [Accessed: 20-Sep-2022].

[2] "Create and verify PDF accessibility (acrobat pro)," *Create and verify PDF accessibility, Acrobat Pro*. [Online]. Available: https://helpx.adobe.com/acrobat/using/create-verify-pdf-accessibility.html. [Accessed: 20-Sep-2022].

[3] Pdfminer, "Pdfminer/pdfminer.six: Community maintained fork of pdfminer - we fathom PDF," *GitHub*. [Online]. Available: https://github.com/pdfminer/pdfminer.six. [Accessed: 20-Sep-2022].

[4] "How to extract data from PDF forms using Python." [Online]. Available: https://towardsdatascience.com/how-to-extract-data-from-pdf-forms-using-python-10b5e5f26f70. [Accessed: 20-Sep-2022].

[5] A. Tiwari, "What is PDF metadata: Everything you need to know," *Allyant*, 14-Aug-2022. [Online]. Available: https://allyant.com/pdf-metadata-definition-view-edit-change-remove-importance/. [Accessed: 20-Sep-2022].

[6] "How to extract data from PDF forms using Python." [Online]. Available: https://towardsdatascience.com/how-to-extract-data-from-pdf-forms-using-python-10b5e5f26f70. [Accessed: 20-Sep-2022].

[7] Accessible Document Solutions, "An introduction to PDF tags," *Accessible Document Solutions*, 07-Dec-2020. [Online]. Available: https://accessible-docs.com/tagging-accessible-pdf/. [Accessed: 20-Sep-2022].

[8] B. Chagnon, "Accessible PDFs Have 4 Reading Orders," *The 4 Reading Orders in a PDF.* [Online]. Available: https://www.pubcom.com/blog/2020_08-18_ReadingOrder/reading-orders.shtml. [Accessed: 20-Sep-2022].

[9] "How to define stakeholders for Your Software Development Project," *Award-winning App Development Company*. [Online]. Available:

https://www.conceptatech.com/blog/how-to-define-stakeholders-for-your-software-development-project. [Accessed: 20-Sep-2022].

[10] W. C. W. A. I. (WAI), “Introduction to web accessibility,” *Web Accessibility Initiative (WAI)*. [Online]. Available: https://www.w3.org/WAI/fundamentals/accessibility-intro/. [Accessed: 20-Sep-2022].

[11] S. Brechbühl, “Add an ALT text,” *Accessible PDF*. [Online]. Available: https://accessible-pdf.info/basics/general/add-an-alt-text. [Accessed: 27-Sep-2022].

[12] S. Hasan. “Pipe and Filter Architecture,” *Medium.com,* Available: https://syedhasan010.medium.com/pipe-and-filter-architecture-bd7babdb908 [Accessed 5-Oct-2022].

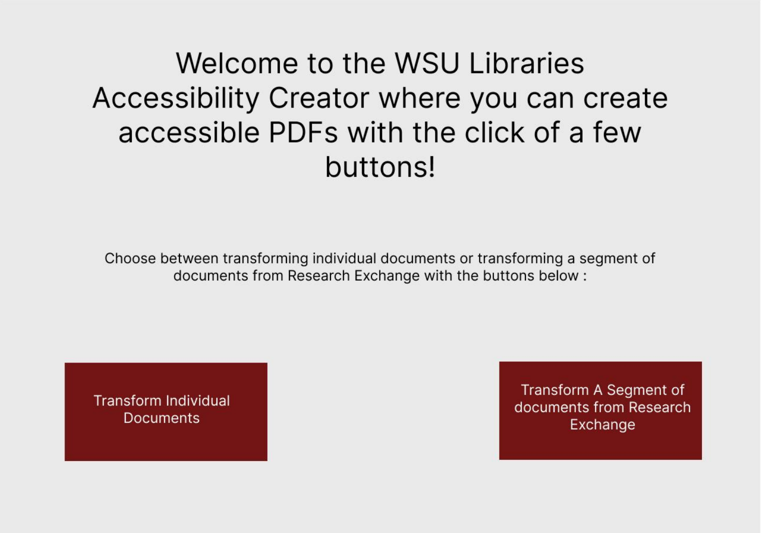
[13] B. Stockton, “What is an executable file & how to create one,” *Help Desk Geek*, 13-Sep-2020. [Online]. Available: https://helpdeskgeek.com/how-to/what-is-an-executable-file-how-to-create-one/. [Accessed: 05-Oct-2022].

[14] A. Marget, “Development and test environments: Understanding the different types of environments,” *Unitrends*, 23-Jul-2021. [Online]. Available: https://www.unitrends.com/blog/development-test-environments. [Accessed: 27-Sep-2022].

[15] “Unittest - unit testing framework¶,” *unittest - Unit testing framework - Python 3.11.0 documentation*. [Online]. Available: https://docs.python.org/3/library/unittest.html. [Accessed: 26-Oct-2022].

XX. Appendices

Appendix A - User Interface Mockups

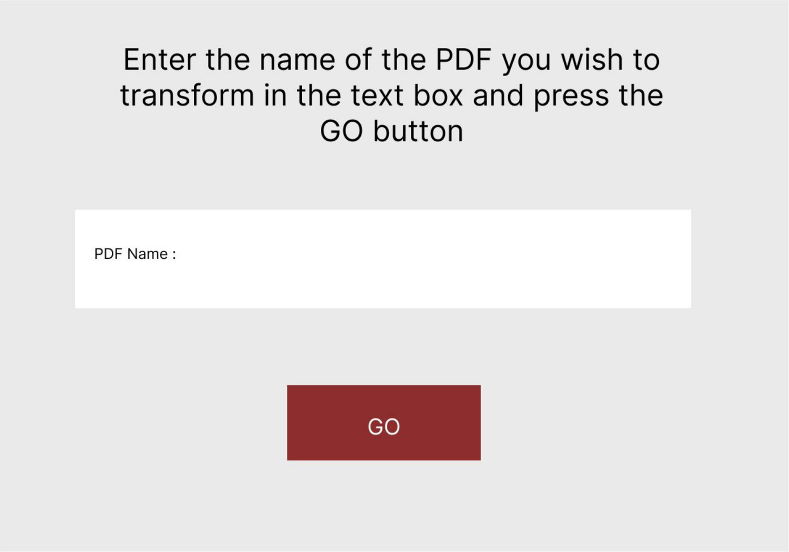


Welcome to the WSU Libraries Accessibility Creator where you can create accessible PDFs with the click of a few buttons!

Choose between transforming individual documents or transforming a segment of documents from Research Exchange with the buttons below :

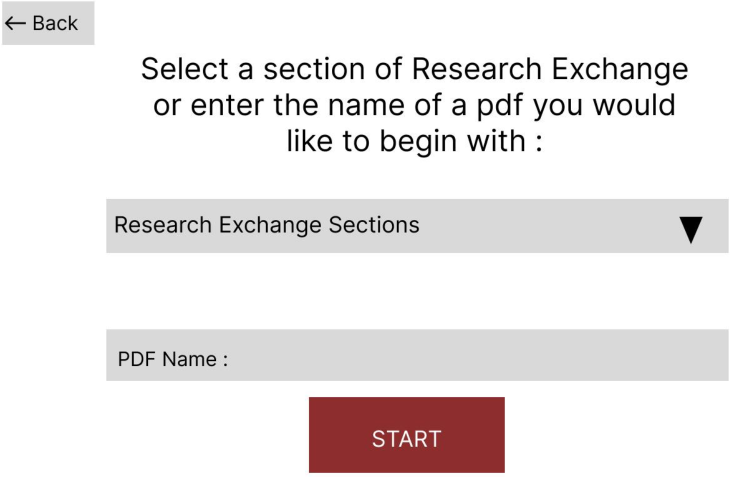
Transform Individual Documents
Transform A Segment of documents from Research Exchange

Main User Interface Mock Up



Enter the name of the PDF you wish to transform in the text box and press the GO button

Individual Document Transformation Mock Up

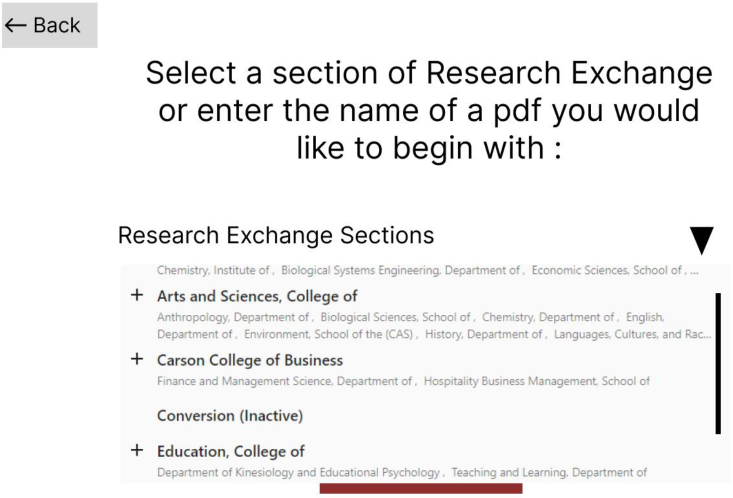


Select a section of Research Exchange or enter the name of a pdf you would like to begin with :

Research Exchange Sections ▼

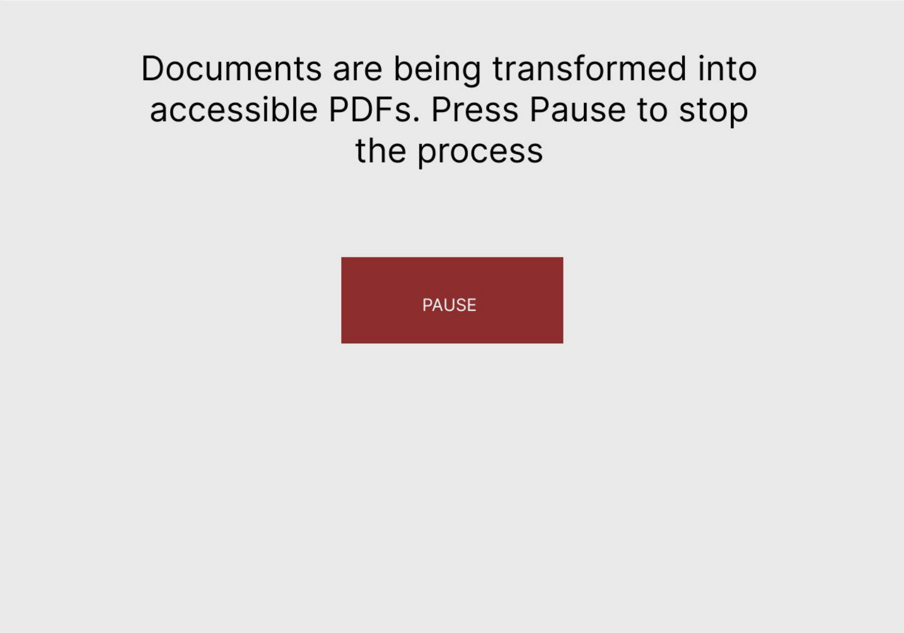
PDF Name :

Segment Transformation Mock Up


← Back
Research Exchange Sections ▼
Chemistry, Institute of , Biological Systems Engineering, Department of , Economic Sciences, School of , ...
+ Arts and Sciences, College of
Anthropology, Department of , Biological Sciences, School of , Chemistry, Department of , English, Department of , Environment, School of the (CAS) , History, Department of , Languages, Cultures, and Rac...
+ Carson College of Business
Finance and Management Science, Department of , Hospitality Business Management, School of
Conversion (Inactive)
+ Education, College of
Department of Kinesiology and Educational Psychology , Teaching and Learning, Department of

Select a section of Research Exchange or enter the name of a pdf you would like to begin with :

Drop Down Menu Mock Up



Documents are being transformed into accessible PDFs. Press Pause to stop the process

Pause Screen Mock Up

Appendix B - Team Information

Trent Bultsma | Email: trent.bultsma@wsu.edu

Reagan Kelley | Email: reagan.kelley@wsu.edu

Marisa Loyd | Email: marisa.loyd@wsu.edu

Appendix C - Example Testing Strategy Reporting

Aspect being tested : PDF Exporter with unittest

Expected result : PDF is generated based on the input html

Observed result : PDF was generated based on the input html

Test result : Pass

Test case requirements : Puppeteer and npm must be installed to run the exporter and unittest must be imported, as well as export_pdf(filename) from pdf_exporter.py

Aspect being tested : PDF Exporter naming by user checking

Expected result : PDF should be generated and named based on the input html (example.html)

Observed result : example.pdf was generated based on the input html (example.html)

Test result : Pass

Test case requirements : Puppeteer and npm must be installed to run the exporter and user needs access to the output folder

Aspect being tested : PDF Exporter location by user checking

Expected result : PDF was generated based on the input html and stored in the data/output folder

Observed result : PDF was generated based on the input html and stored in data/output folder

Test result : Pass

Test case requirements : Puppeteer and npm must be installed to run the exporter and user needs access to the output folder

Aspect being tested : PDF extractor font style emphasis detection

Expected result : Font style tags for bold and italics are detected

Observed result : Font style tags for bold and italics are detected

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `` `python -m unittest -v tests.test_pdf_extractor` `` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : Document Tag Tree Generation

Expected result : Tags generate according to optimal structure

Observed result : No observed result

Test result : Fail

Test case requirements : Python unittest module, document_tagger.py, document_layout.py and paragraph.py. Also, to run the test in our directory structure, the user must run the command `` `python -m unittest -v tests.test_pdf_extractor` `` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : PDF Read Order

Expected result : Layout Blocks in the Document Class ordered according to read order

Observed result : No observed result

Test result : Fail

Test case requirements : Python unittest module, document_layout.py and paragraph.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest

-v tests.test_pdf_extractor` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : PDF extractor font style detection

Expected result : Font style names are detected (such as times new roman or whatnot)

Observed result : Font style names are detected

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest -v tests.test_pdf_extractor` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : PDF extractor text attribute detection

Expected result : Text attributes are translated from an html string to a python dictionary

Observed result : Text attributes are translated from an html string to a python dictionary

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest -v tests.test_pdf_extractor` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : PDF extractor cid string conversion

Expected result : Translation of cid encoded strings to utf-8 characters

Observed result : Translation of cid encoded strings to utf-8 characters

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest -v tests.test_pdf_extractor` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : PDF extractor paragraph comparison

Expected result : Paragraph objects with the same content are equal, otherwise not equal

Observed result : Paragraph objects with the same content are equal, otherwise not equal

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest -v tests.test_pdf_extractor` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : PDF extractor paragraph extraction

Expected result : Input pdf file paragraph extraction matches example html extraction equivalent

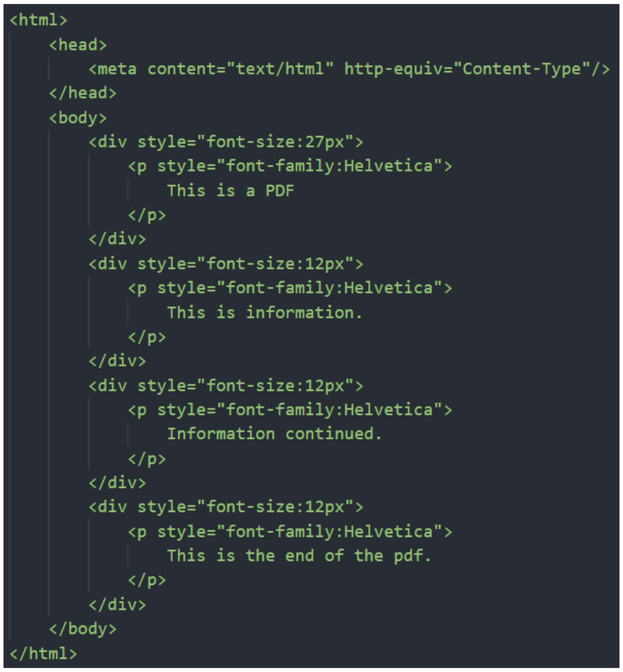
Observed result : Input pdf file paragraph extraction matches example html extraction equivalent

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest -v tests.test_pdf_extractor` from the accessibility_apps directory within the command prompt or power shell.

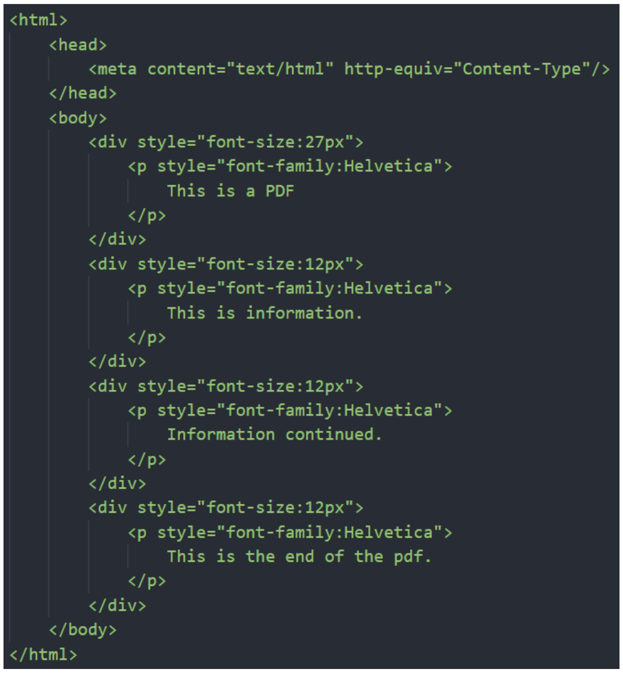
Aspect being tested : PDF extractor html exporting

Expected result :



```
<html>
    <head>
        <meta content="text/html" http-equiv="Content-Type"/>
    </head>
    <body>
        <div style="font-size:27px">
            <p style="font-family:Helvetica">
                This is a PDF
            </p>
        </div>
        <div style="font-size:12px">
            <p style="font-family:Helvetica">
                This is information.
            </p>
        </div>
        <div style="font-size:12px">
            <p style="font-family:Helvetica">
                Information continued.
            </p>
        </div>
        <div style="font-size:12px">
            <p style="font-family:Helvetica">
                This is the end of the pdf.
            </p>
        </div>
    </body>
</html>
```

Observed result :



```
<html>
    <head>
        <meta content="text/html" http-equiv="Content-Type"/>
    </head>
    <body>
        <div style="font-size:27px">
            <p style="font-family:Helvetica">
                This is a PDF
            </p>
        </div>
        <div style="font-size:12px">
            <p style="font-family:Helvetica">
                This is information.
            </p>
        </div>
        <div style="font-size:12px">
            <p style="font-family:Helvetica">
                Information continued.
            </p>
        </div>
        <div style="font-size:12px">
            <p style="font-family:Helvetica">
                This is the end of the pdf.
            </p>
        </div>
    </body>
</html>
```

Test result : Pass

Test case requirements : Python unittest module, pdf_extractor.py, and paragraph.py. Also, to run the test in our directory structure, the user must run the command `` `python -m unittest -v tests.test_pdf_extractor` `` from the accessibility_apps directory within the command prompt or power shell.

Aspect being tested : CSV Metadata Reader

Expected result :

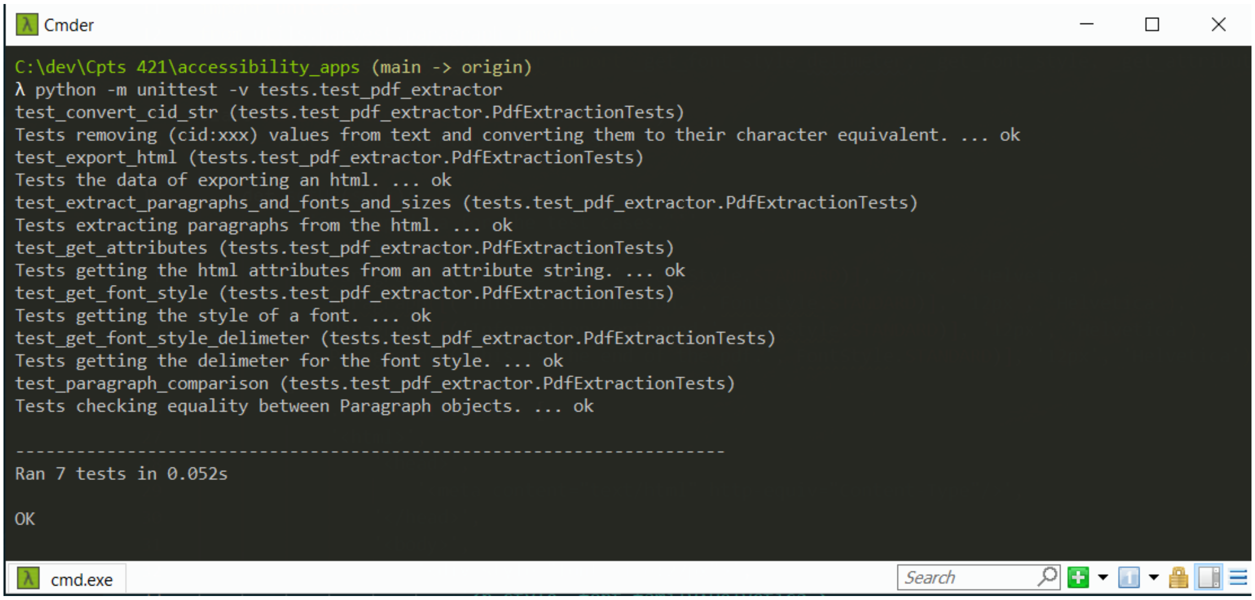
{"ASSET_TITLE" : "Grant County 4H Update, October 2022", “ASSET_PUBLISHDATE" :
"202210", "ASSET_AVAILDATE" : "20230301", "ASSET_RESTYPE" :
"publication.newsletterArticle", "ASSET_KEYWORDS" : "4-H club", "ASSET_PEEREVIEW" :
"NO", "ASSET_LANG" : "eng", "ASSET_COPYRIGHT" :
"http://rightsstatements.org/vocab/InC/1.0/", "ASSET_AFFILIATION" :
”01ALLIANCE_WSU___41_01_01", "ASSET_FORMAT" : "pdf", "ASSET_OPENACCESS"
:"YES", "ASSET_DATEEPUB" : "202210", "ASSET_PORTAL_VISIBILITY" : "TRUE",
“ASSET_PROFILE_VISIBILITY" : "FALSE", "OCREATOR_ORGANIZATION" : "Washington
State University Extension", "FILE_FILEURL" :
"https://s3.us-east-1.amazonaws.com/na-st01.ext.exlibrisgroup.com/01ALLIANCE_WSU/upload
/esploro/talea-test/4h_2022_10.pdf", "FILE_DISPLAY_NAME" : "Grant County 4H Update,
October 2022"}

Observed result :

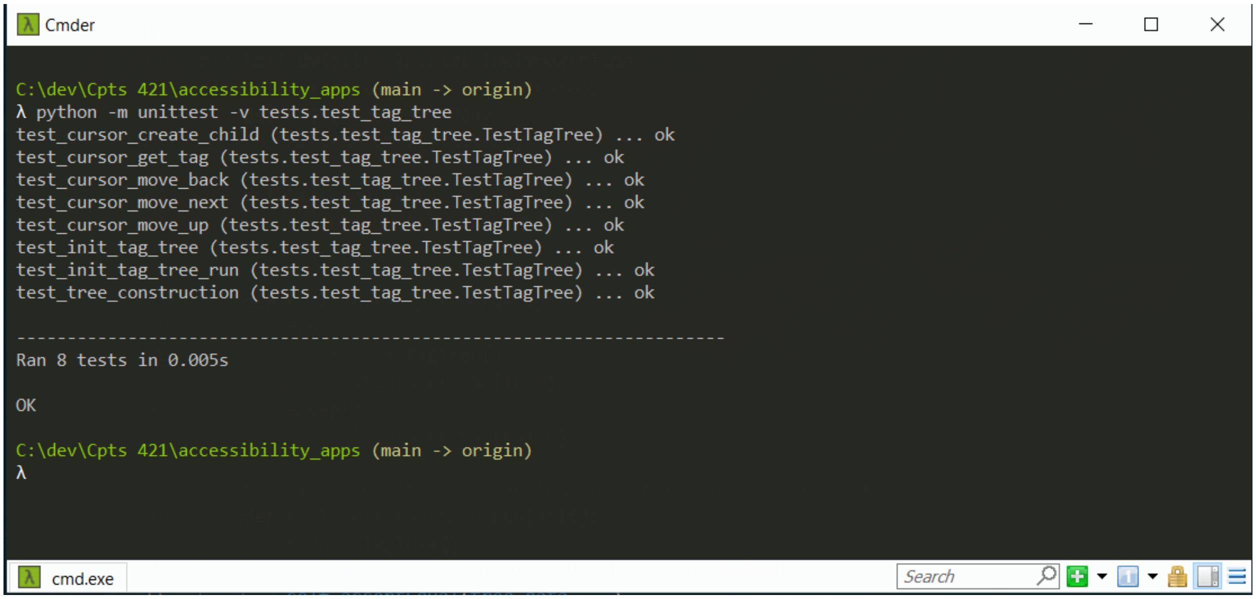
{'ASSET_TITLE': 'Grant County 4H Update, October 2022', 'ASSET_SUBTITLE': '',
'ASSET_DOI': '', 'ASSET_PMID': '', 'ASSET_PUBLISHDATE': '202210', 'ASSET_AVAILDATE':
'20230301', 'ASSET_ACCEPTDATE': '', 'ASSET_COPYRIGHTDATE': '',
'ASSET_CREATEDATE': '', 'ASSET_OPENINGDATE': '', 'ASSET_PERFORMANCEDATE': '',
'ASSET_RESTYPE': 'publication.newsletterArticle', 'ASSET_RESSUBTYPE': '',
'ASSET_KEYWORDS': '4-H club', 'ASSET_PEEREVIEW': 'NO', 'ASSET_LANG': 'eng',
'ASSET_ABSTRACT': '', 'ASSET_COPYRIGHT': 'http://rightsstatements.org/vocab/InC/1.0/',
'ASSET_AFFILIATION': '01ALLIANCE_WSU___41_01_01', 'ASSET_GRANTNOTE': '',
'ASSET_ORIGASSET_ID': '', 'ASSET_FORMAT': 'pdf', 'ASSET_OPENACCESS': 'YES',
'ASSET_DATEEPUB': '202210', 'ASSET_PORTAL_VISIBILITY': 'TRUE',
'ASSET_IDENTIFIER_ADDITIONAL01': '', 'ASSET_PROFILE_VISIBILITY': 'FALSE',
'OCREATOR_ORGANIZATION': 'Washington State University Extension',
'OCONT_ORGANIZATION': '', 'FILE_FILEURL':
'https://s3.us-east-1.amazonaws.com/na-st01.ext.exlibrisgroup.com/01ALLIANCE_WSU/upload/
esploro/talea-test/4h_2022_10.pdf', 'FILE_DISPLAY_NAME': 'Grant County 4H Update, October
2022'}

Test result : Pass

Test case requirements : Python unittest module, metadata_csv_reader.py, metadata_csv_reader.py. Also, to run the test in our directory structure, the user must run the command `python -m unittest -v tests.test_metadata_csv_reader` from the accessibility_apps directory within the command prompt or power shell.



```
C:\dev\Cpts 421\accessibility_apps (main -> origin)
λ python -m unittest -v tests.test_pdf_extractor
test_convert_cid_str (tests.test_pdf_extractor.PdfExtractionTests)
Tests removing (cid:xxx) values from text and converting them to their character equivalent. ... ok
test_export_html (tests.test_pdf_extractor.PdfExtractionTests)
Tests the data of exporting an html. ... ok
test_extract_paragraphs_and_fonts_and_sizes (tests.test_pdf_extractor.PdfExtractionTests)
Tests extracting paragraphs from the html. ... ok
test_get_attributes (tests.test_pdf_extractor.PdfExtractionTests)
Tests getting the html attributes from an attribute string. ... ok
test_get_font_style (tests.test_pdf_extractor.PdfExtractionTests)
Tests getting the style of a font. ... ok
test_get_font_style_delimeter (tests.test_pdf_extractor.PdfExtractionTests)
Tests getting the delimeter for the font style. ... ok
test_paragraph_comparison (tests.test_pdf_extractor.PdfExtractionTests)
Tests checking equality between Paragraph objects. ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.052s

OK
```



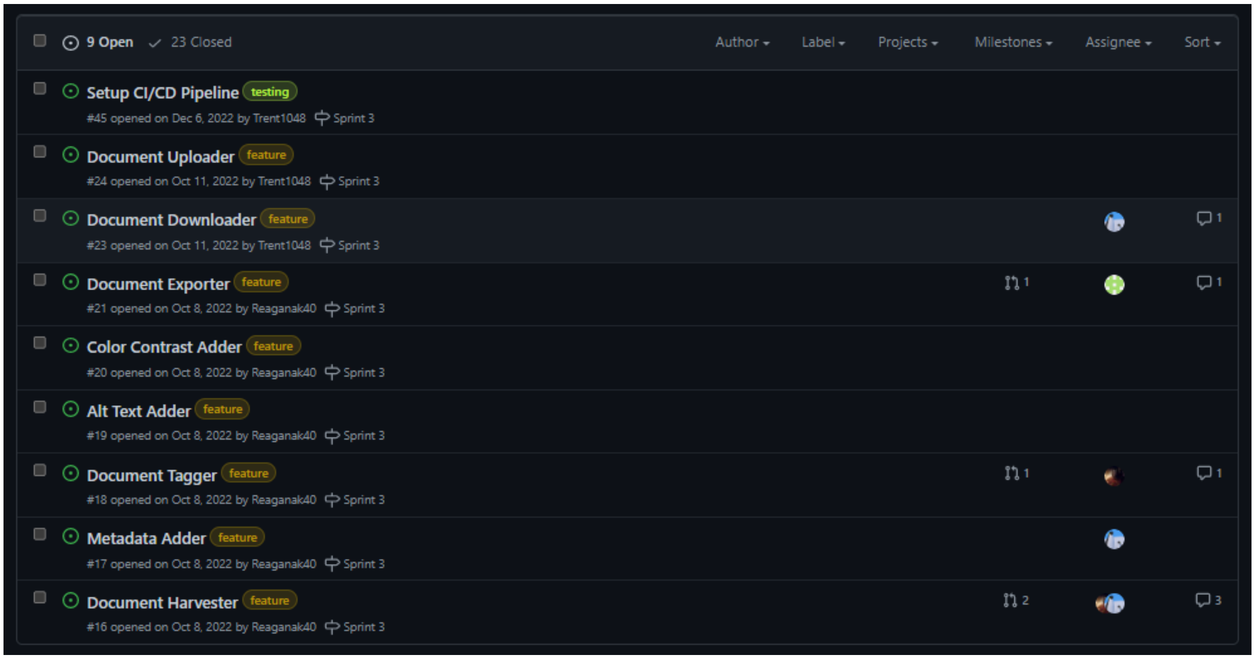
```
C:\dev\Cpts 421\accessibility_apps (main -> origin)
λ python -m unittest -v tests.test_tag_tree
test_cursor_create_child (tests.test_tag_tree.TestTagTree) ... ok
test_cursor_get_tag (tests.test_tag_tree.TestTagTree) ... ok
test_cursor_move_back (tests.test_tag_tree.TestTagTree) ... ok
test_cursor_move_next (tests.test_tag_tree.TestTagTree) ... ok
test_cursor_move_up (tests.test_tag_tree.TestTagTree) ... ok
test_init_tag_tree (tests.test_tag_tree.TestTagTree) ... ok
test_init_tag_tree_run (tests.test_tag_tree.TestTagTree) ... ok
test_tree_construction (tests.test_tag_tree.TestTagTree) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.005s

OK

C:\dev\Cpts 421\accessibility_apps (main -> origin)
λ
```

At this time, user testing is conducted through continuous feedback from our client who has not directly applied our tools, but provides direction so we may get closer to a usable product for end-users.

Appendix D - Project Management

Our team's weekly schedule is to meet on Tuesdays at 1:00pm on Discord as developers to discuss our progress for the week and what next steps we need to take on the project. Then on Thursdays at 1:30pm on Zoom we meet with our client to update her on our project status and check in if she has any questions about the project. We also meet with our professor every few weeks on Zoom for coaching. The team's weekly standup where we meet as developers is most helpful for our group because it allows us to get on the same page and coordinate our contributions for the week. For planning we typically use GitHub issues and projects as described below:

GitHub Issues:



Setup CI/CD Pipeline testing
#45 opened on Dec 6, 2022 by Trent1048 Sprint 3

Document Uploader feature
#24 opened on Oct 11, 2022 by Trent1048 Sprint 3

Document Downloader feature
#23 opened on Oct 11, 2022 by Trent1048 Sprint 3 1

Document Exporter feature
#21 opened on Oct 8, 2022 by Reaganak40 Sprint 3 1 1

Color Contrast Adder feature
#20 opened on Oct 8, 2022 by Reaganak40 Sprint 3

Alt Text Adder feature
#19 opened on Oct 8, 2022 by Reaganak40 Sprint 3

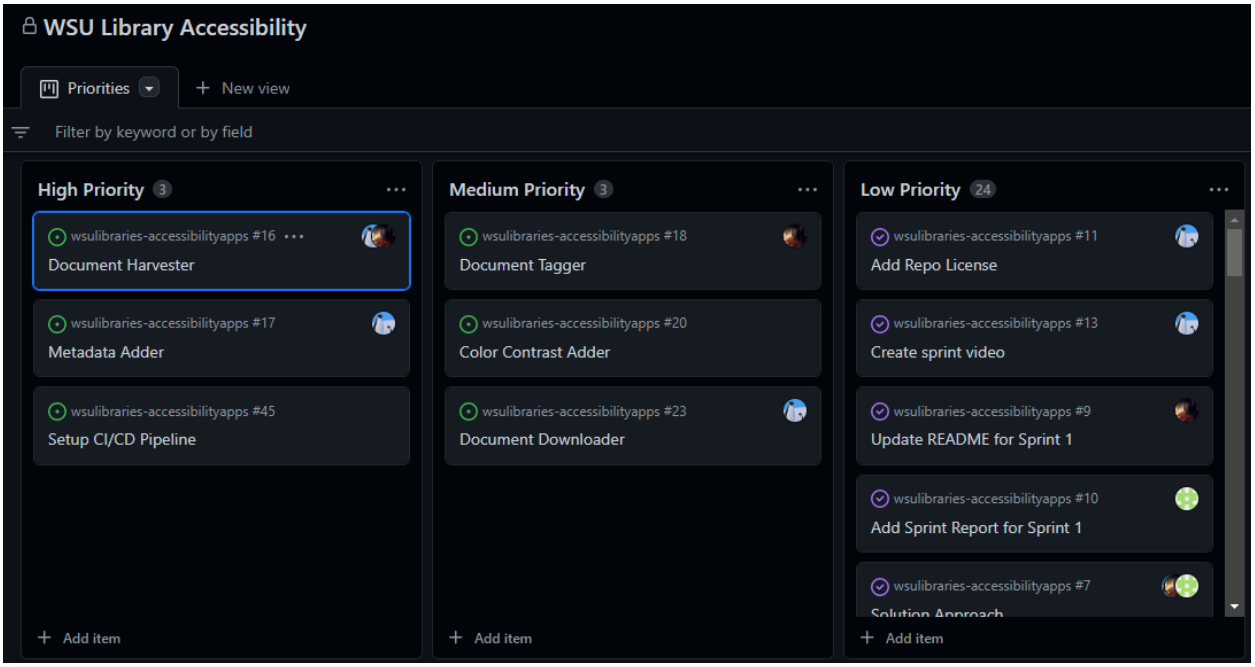
Document Tagger feature
#18 opened on Oct 8, 2022 by Reaganak40 Sprint 3 1 1

Metadata Adder feature
#17 opened on Oct 8, 2022 by Reaganak40 Sprint 3

Document Harvester feature
#16 opened on Oct 8, 2022 by Reaganak40 Sprint 3 2 3

We use GitHub issues to plan out different tasks for each of us to work on that can be claimed by each member of the team.

GitHub Projects:



WSU Library Accessibility

High Priority 3

wsulibraries-accessibilityapps #16
Document Harvester

wsulibraries-accessibilityapps #17
Metadata Adder

wsulibraries-accessibilityapps #45
Setup CI/CD Pipeline

Medium Priority 3

wsulibraries-accessibilityapps #18
Document Tagger

wsulibraries-accessibilityapps #20
Color Contrast Adder

wsulibraries-accessibilityapps #23
Document Downloader

Low Priority 24

wsulibraries-accessibilityapps #11
Add Repo License

wsulibraries-accessibilityapps #13
Create sprint video

wsulibraries-accessibilityapps #9
Update README for Sprint 1

wsulibraries-accessibilityapps #10
Add Sprint Report for Sprint 1

wsulibraries-accessibilityapps #7
Solution Approach

We use GitHub projects to specify the importance of each of our tasks that we have planned or are working on to help us determine what to spend the most time on or which new tasks to take on once we have completed older tasks.