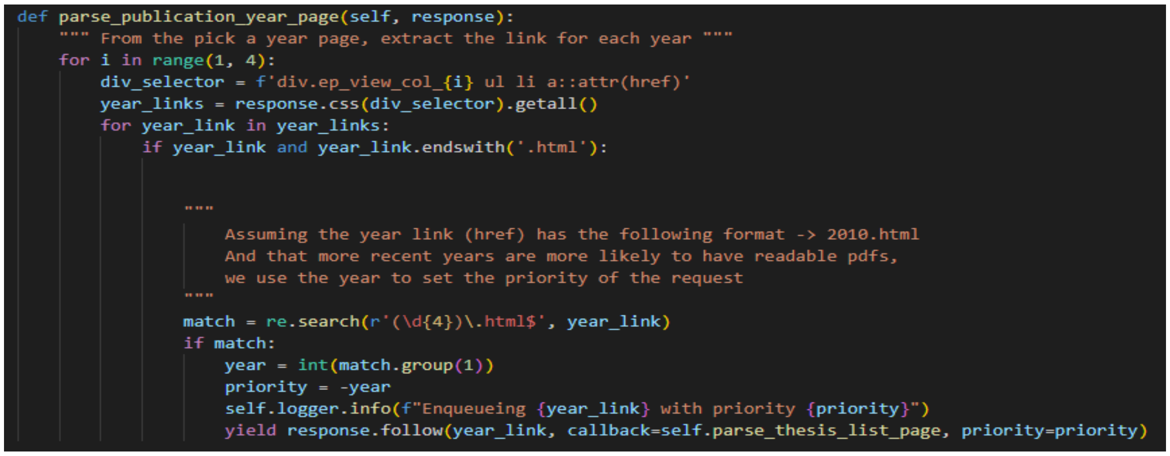
Patrick MacEachen - 40209790
COMP479 - Project 2 demo
2024/12/05

# Important Code snippets

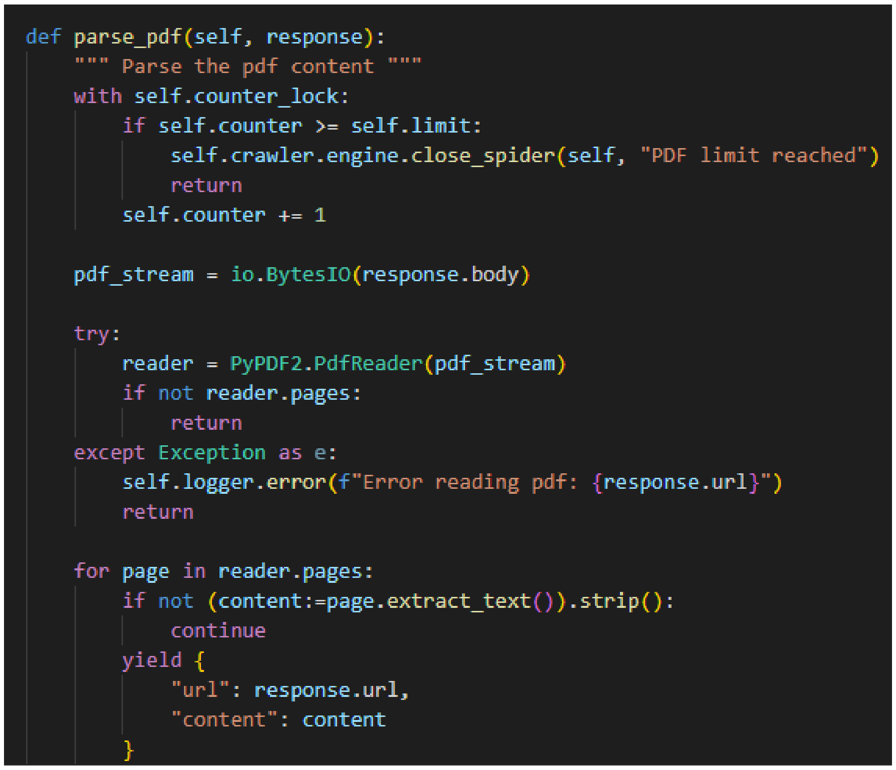
These are two methods from the crawler that I note as important.

1. The use of priority based on the year of publication



```python
def parse_publication_year_page(self, response):
    """ From the pick a year page, extract the link for each year """
    for i in range(1, 4):
        div_selector = f'div.ep_view_col_{i} ul li a::attr(href)'
        year_links = response.css(div_selector).getall()
        for year_link in year_links:
            if year_link and year_link.endswith('.html'):


                """
                    Assuming the year link (href) has the following format -> 2010.html
                    And that more recent years are more likely to have readable pdfs,
                    we use the year to set the priority of the request
                """
                match = re.search(r'(\d{4})\.html$', year_link)
                if match:
                    year = int(match.group(1))
                    priority = -year
                    self.logger.info(f"Enqueueing {year_link} with priority {priority}")
                    yield response.follow(year_link, callback=self.parse_thesis_list_page, priority=priority)
```

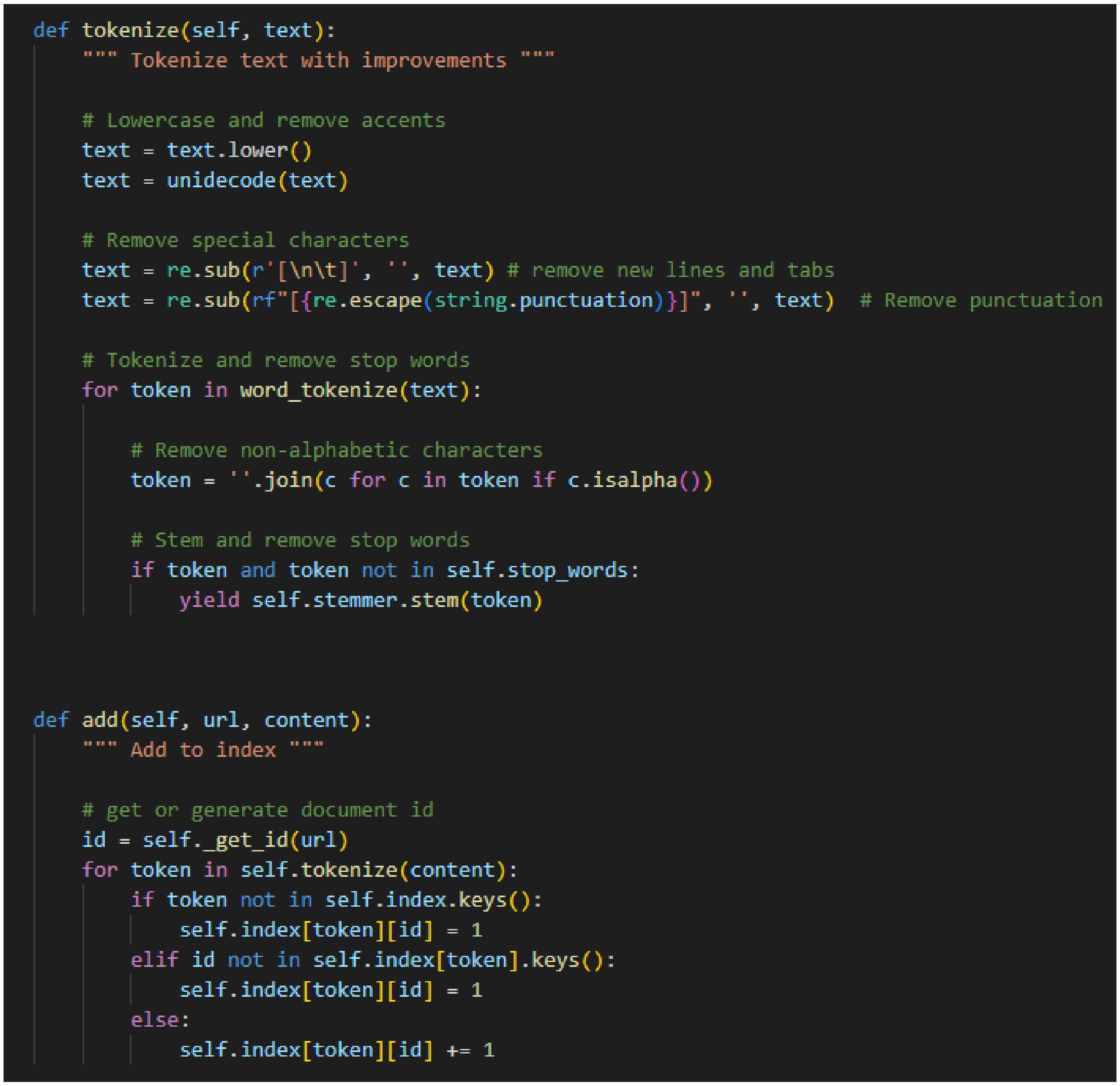
2. The parsing of the pdf



```python
def parse_pdf(self, response):
    """ Parse the pdf content """
    with self.counter_lock:
        if self.counter >= self.limit:
            self.crawler.engine.close_spider(self, "PDF limit reached")
            return
        self.counter += 1

    pdf_stream = io.BytesIO(response.body)

    try:
        reader = PyPDF2.PdfReader(pdf_stream)
        if not reader.pages:
            return
    except Exception as e:
        self.logger.error(f"Error reading pdf: {response.url}")
        return

    for page in reader.pages:
        if not (content:=page.extract_text()).strip():
            continue
        yield {
            "url": response.url,
            "content": content
        }
```

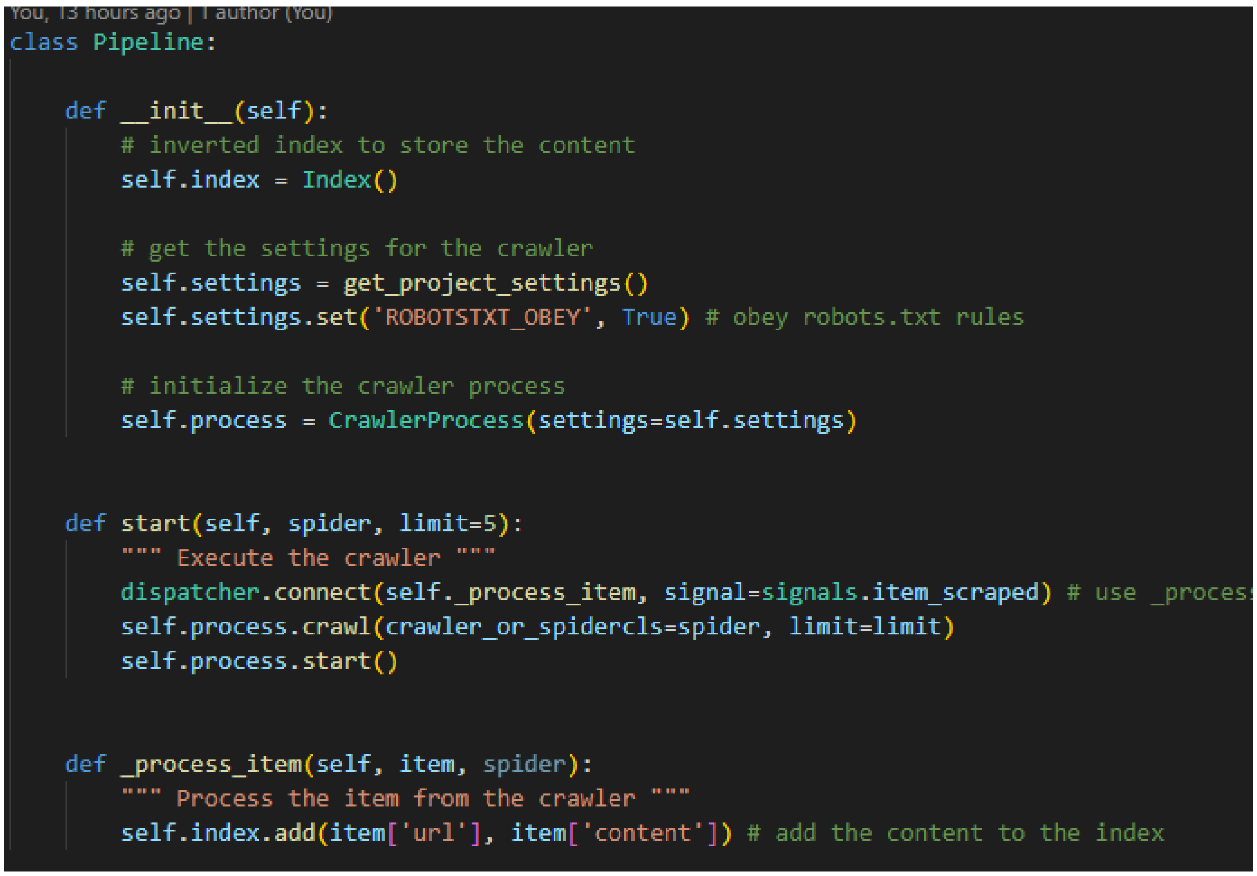
The next photo, are the core methods of the index class (add and tokenize)

3. The tokenize and add methods from the Index class.



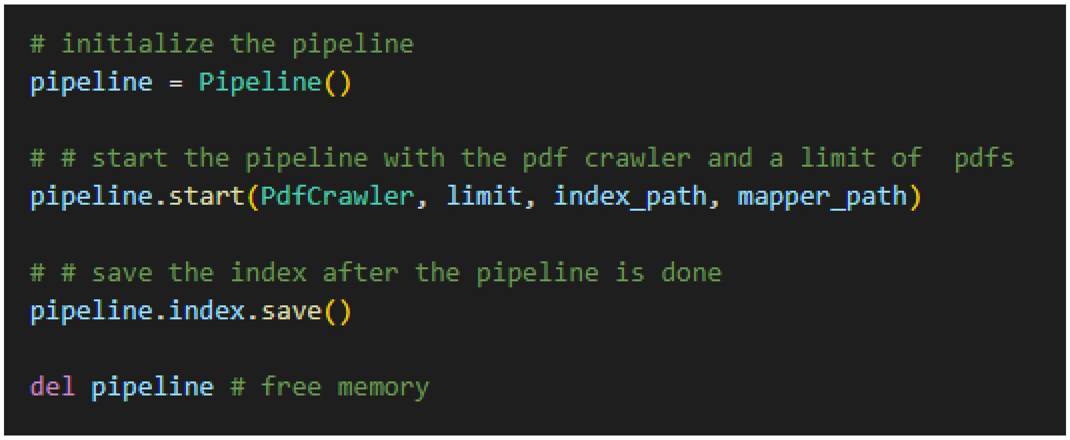
```python
def tokenize(self, text):
    """ Tokenize text with improvements """

    # Lowercase and remove accents
    text = text.lower()
    text = unidecode(text)

    # Remove special characters
    text = re.sub(r'[\n\t]', '', text) # remove new lines and tabs
    text = re.sub(rf"[{re.escape(string.punctuation)}]", '', text)  # Remove punctuation

    # Tokenize and remove stop words
    for token in word_tokenize(text):

        # Remove non-alphabetic characters
        token = ''.join(c for c in token if c.isalpha())

        # Stem and remove stop words
        if token and token not in self.stop_words:
            yield self.stemmer.stem(token)



def add(self, url, content):
    """ Add to index """

    # get or generate document id
    id = self._get_id(url)
    for token in self.tokenize(content):
        if token not in self.index.keys():
            self.index[token][id] = 1
        elif id not in self.index[token].keys():
            self.index[token][id] = 1
        else:
            self.index[token][id] += 1
```

4. The pipeline class with the start and process item methods.



```
You, 13 hours ago | 1 author (You)
class Pipeline:

    def __init__(self):
        # inverted index to store the content
        self.index = Index()

        # get the settings for the crawler
        self.settings = get_project_settings()
        self.settings.set('ROBOTSTXT_OBEY', True) # obey robots.txt rules

        # initialize the crawler process
        self.process = CrawlerProcess(settings=self.settings)


    def start(self, spider, limit=5):
        """ Execute the crawler """
        dispatcher.connect(self._process_item, signal=signals.item_scraped) # use _process
        self.process.crawl(crawler_or_spidercls=spider, limit=limit)
        self.process.start()


    def _process_item(self, item, spider):
        """ Process the item from the crawler """
        self.index.add(item['url'], item['content']) # add the content to the index
```

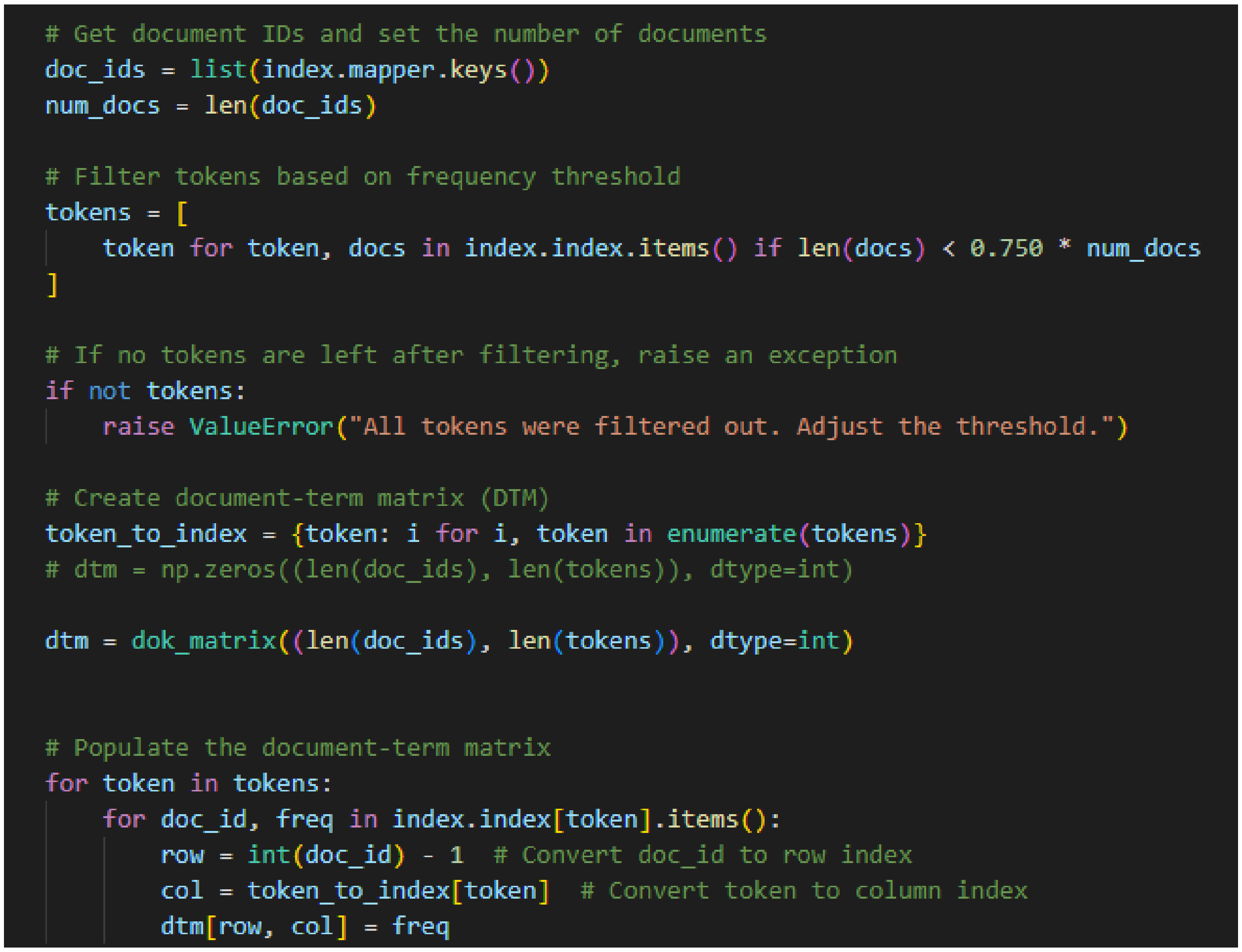
5. The build_index.py file for executing the pipeline.



```
# initialize the pipeline
pipeline = Pipeline()

# # start the pipeline with the pdf crawler and a limit of  pdfs
pipeline.start(PdfCrawler, limit, index_path, mapper_path)

# # save the index after the pipeline is done
pipeline.index.save()

del pipeline # free memory
```

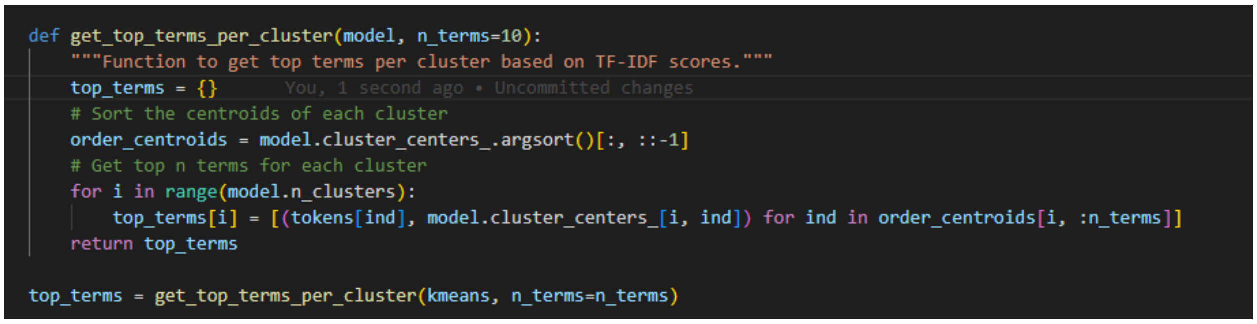
**Note**: To view the output of the index building you can view the files within the index_main directory.

6. The construction of the document-term matrix



```python
# Get document IDs and set the number of documents
doc_ids = list(index.mapper.keys())
num_docs = len(doc_ids)

# Filter tokens based on frequency threshold
tokens = [
    token for token, docs in index.index.items() if len(docs) < 0.750 * num_docs
]

# If no tokens are left after filtering, raise an exception
if not tokens:
    raise ValueError("All tokens were filtered out. Adjust the threshold.")

# Create document-term matrix (DTM)
token_to_index = {token: i for i, token in enumerate(tokens)}
# dtm = np.zeros((len(doc_ids), len(tokens)), dtype=int)

dtm = dok_matrix((len(doc_ids), len(tokens)), dtype=int)


# Populate the document-term matrix
for token in tokens:
    for doc_id, freq in index.index[token].items():
        row = int(doc_id) - 1  # Convert doc_id to row index
        col = token_to_index[token]  # Convert token to column index
        dtm[row, col] = freq
```

7. The function for retrieving the top n terms by tf-idf score



```python
def get_top_terms_per_cluster(model, n_terms=10):
    """Function to get top terms per cluster based on TF-IDF scores."""
    top_terms = {}
    # Sort the centroids of each cluster
    order_centroids = model.cluster_centers_.argsort()[:, ::-1]
    # Get top n terms for each cluster
    for i in range(model.n_clusters):
        top_terms[i] = [(tokens[ind], model.cluster_centers_[i, ind]) for ind in order_centroids[i, :n_terms]]
    return top_terms

top_terms = get_top_terms_per_cluster(kmeans, n_terms=n_terms)
```

Note: for the clustering, you can find the clusters, the top n terms in the clustering directory.
For the clustering with k being the number of departments or faculty, you can find the same results in the clustering_faculty_department  directory.