PROJECT REPORT

Improved ExpertSearch System

(Team ZMS)

Zacharia Rupp (zrupp2@illinois.edu)
Mriganka Sarma (ms76@illinois.edu)
Sai Ranganathan (sr50@illinois.edu)

Introduction

The ExpertSearch system is a system to search faculties who are experts in certain research areas or topics from university websites crawled from the web. The goal of our project is to improve this ExpertSearch system in a few ways, including automating the scraping process, adding topic mining for finding faculty research topics, and improving the UI to give improved visualizations and query refinement options.

1. Functional Overview

The improved ExpertSearch system enables the following functionalities:

- Automatic scraping of websites to identify faculty directory webpages and non-directory webpages
- Automatic scraping of the classified faculty directory webpages to further identify faculty bio webpages and non-bio webpages
- Automatic scraping of faculty bio webpages to generate one bio document per faculty and adding to the compiled bios
- Topic Mining from the compiled bios to extract research topics of the faculties
- Display top-5 research topics associated with each retrieved faculty
- Improved email extraction for each faculty
- Refine search query using any of the topics from the displayed topic cloud
- Prepopulate email content when clicked on a faculty's email address

The **Automated Scraper** improves the faculty bio generation process from a vast collection of websites.
The **Topic Miner** adds more structure to the unstructured faculty website data retrieved from a query.
The **enhanced UI** enables succinct visualization of the structured faculty results and provides shortcuts for additional search filters and faculty connection.
Together, these new features improve the utility of the ExpertSearch system to the user.

2. Implementation Details

2.1. Automated scraping process

The Automated Scraper takes a set of known University websites and top 500 Alexa websites as input, performs a series of operations to classify the directory URLs and then to classify the faculty homepages. The automated scraper then scrapes the classified faculty homepages to generate faculty bio documents and adds the bios to the collection.

2.1.1. Inputs:

- University websites
- Top-500 Alexa websites

2.1.2. Outputs:

- Corpus of classified Faculty Directory URLs ***(classified_dir_urls.cor)***
- Corpus of classified Faculty Bio URLs ***(classified_faculty_urls.cor)***
- Documents of bios for each faculty generated by scraping the classified Faculty Bio URLs ***(e.g. 6530.txt)***

2.1.3. Deliverables:

- Automated Scraper ***(auto_scraper.py)***
- Data Handler ***(data_handler.py)***
- Scraper ***(scraper.py)***
- Text Classifier ***(text_classifier.py)***

2.1.4. Component design / Code workflow:

Automated Scraper (auto_scraper.py)

The Automated Scraper module automates the whole flow of generating the faculty bios from the input mixture of “positive” and “negative” URLs in the following sequence of steps:

- Uses the data handler (data_handler.py) to prepare a train and test set of Faculty Directory URLs
- Uses the scraper (scraper.py) to scrape these URLs to prepare the train and test corpus
- Uses the text classifier (text_classifier.py) to build and train a Doc2Vec model on the documents in the train corpus of directory contents
- Uses the text classifier to predict the category of the test URLs as “Directory” or “Non-Directory”
- Saves the classified directory URLs to a file **(classified_dir_urls.cor)**
- Uses the data handler to prepare a train and test set of Faculty Bio URLs
- Uses the scraper to scrape these URLs to prepare the train and test corpus

- Uses the text classifier to build and train a Doc2Vec model on the documents in the train corpus of faculty bios
- Uses the text classifier to predict the category of the test URLs as “Faculty” or “Non-Faculty”
- Saves the classified bio URLs to a file **(classified_faculty_urls.cor)**
- Uses the scraper to scrape the faculty bios from the classified bio URLs
- Generates one document per faculty bio (e.g. **6530.txt**) and saves under **ExpertSearch/data/compiled_bios**

The auto_scraper.py module is the entry point for the complete automatic scraping and bio generation task. The following figure **(Fig. 1)** shows the complete automation flow starting with the input websites till the bio generation completion.

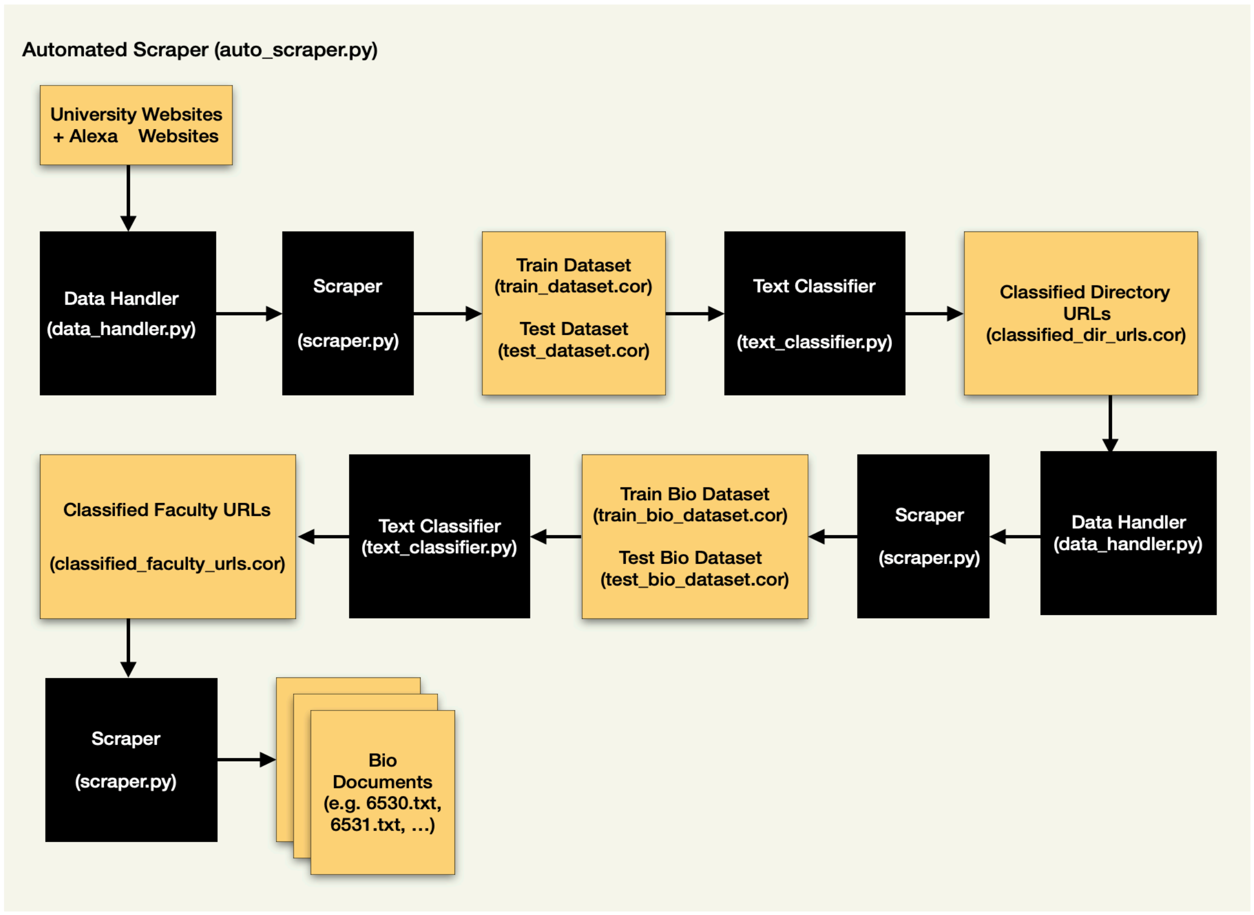
Automated Scraper (auto_scraper.py)
University Websites + Alexa Websites
Data Handler (data_handler.py)
Scraper (scraper.py)
Train Dataset (train_dataset.cor) Test Dataset (test_dataset.cor)
Text Classifier (text_classifier.py)
Classified Directory URLs (classified_dir_urls.cor)
Data Handler (data_handler.py)
Scraper (scraper.py)
Train Bio Dataset (train_bio_dataset.cor) Test Bio Dataset (test_bio_dataset.cor)
Text Classifier (text_classifier.py)
Classified Faculty URLs (classified_faculty_urls.cor)
Scraper (scraper.py)
Bio Documents (e.g. 6530.txt, 6531.txt, …)

Fig. 1: Automation Control Flow / Module interactions

Directory URL Classification

First, let’s explain the Directory URL Classification task with the help of the modules.

Data Handler (data_handler.py)

The Data Handler module first takes the University websites and Alexa websites as input, mixes them and partitions them into test and train URLs. Then uses the scraper module to extract the URL contents into test and train corpus as shown in the figure below **(Fig. 2)**.

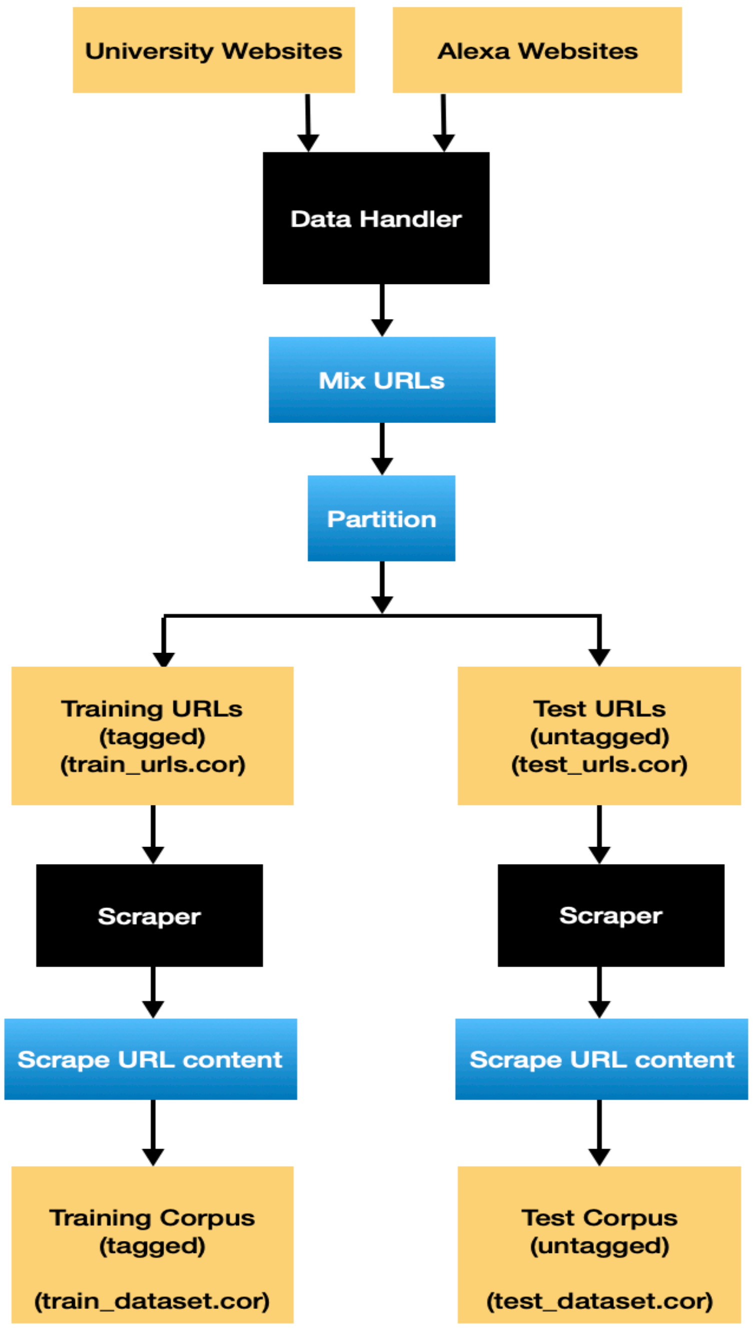
University Websites
Alexa Websites
Data Handler
Mix URLs
Partition
Training URLs (tagged) (train_urls.cor)
Test URLs (untagged) (test_urls.cor)
Scraper
Scraper
Scrape URL content
Scrape URL content
Training Corpus (tagged) (train_dataset.cor)
Test Corpus (untagged) (test_dataset.cor)

Fig. 2: Dataset Preparation for Directory URL Classification

Here's a detailed explanation of the approach used by the Data Handler module to prepare the URLs for the scraper.

- Downloaded the known faculty directory URLs from the sign-up sheet for MP 2.1. These will serve as the "**positive**" examples.
    - About 900 URLs were obtained from the sign-up sheet data, which was partitioned into **500 for training** and **400 for test data**.
- Collected top URLs from Alexa. These will serve as the "**negative**" examples.
    - Collected the global top-50 pages of Alexa.
    - Collected the top-50 pages for 14 different countries. Manually verified that the pages are in English.
    - This gave 750 "negative" URLs.
- When the **AutoScraper** is launched, it invokes the **DataHandler** which mixes and partitions the above URLs into train and test URLs as follows:
    - Training URLs
        - Converts the MP 2.1 sign-up sheet from csv to a file containing only the directory URLs. Performs any cleanup as necessary and labels them as "directory".
        - Combines the top-50 Alexa URLs for 10 countries and labels them as "alexa_dir". Uses these 500 pages for training.
        - Mixes the 500 Faculty Directory training URLs with the 500 Alexa training URLs. Removes duplicates if any. This gives 734 URLs as the final training URLs.
        - The training URLs are saved in the file **train_urls.cor**.
    - Test URLs
        - Combines the top-50 Alexa URLs for 5 countries. Uses these 250 pages for testing.
        - Mixes the 400 Faculty Directory test URLs with the 250 Alexa test URLs. Remove duplicates if any. This gives 548 URLs as the final test URLs.
        - The test URLs are saved in the file **test_urls.cor**.

Scraper (scraper.py)

The Scraper module scrapes the contents from the above train and test URLs and prepares the train and test corpus for the classification task.

The scraper does the following:

- Gets the contents of each URL as text.
- Performs clean-up of non-ascii characters from the content.

- Performs other clean-ups such as substituting newlines, tabs, multiple whitespaces into single whitespace.
- Substitutes contents such as “403 Forbidden”, “404 Not found”, etc. with “Error: Content Not Found”.
- Writes contents of each training URL as a single line of space separated words to the training corpus (“**train_dataset.cor**”).
- Similarly, writes contents of each test URL as a single line of space separated words to the test corpus (“**test_dataset.cor**”).

Text Classifier (text_classifier.py)

The Text Classifier module uses the train and test dataset from above step to classify the Faculty Directory URLs.

The classification module does the following:
- Uses **gensim** to build a Doc2Vec model for feature vector representation of each document.
- Uses the train_dataset.cor to build the vocabulary and train the model.
- Saves the model so that it can be reloaded while running next time on the same dataset.
- Uses **LogisticRegression** as the classifier from scikit-learn module.
- Uses **LogisticRegression** to predict the categories of the test URLs given the test dataset.

Faculty URL classification

Next, let’s look at the Faculty URL Classification task.

Data Handler (data_handler.py)

The Data Handler module now takes the existing project’s known Faculty Bio URLs and mixes with some Alexa URLs to prepare the train dataset. Uses the classified Faculty Directory URLs from the above step to extract potential Faculty Bio URLs to prepare the test dataset. Then uses the scraper module to extract the URL contents into bio test and bio train corpus as shown in the figure below **(Fig. 3)**.

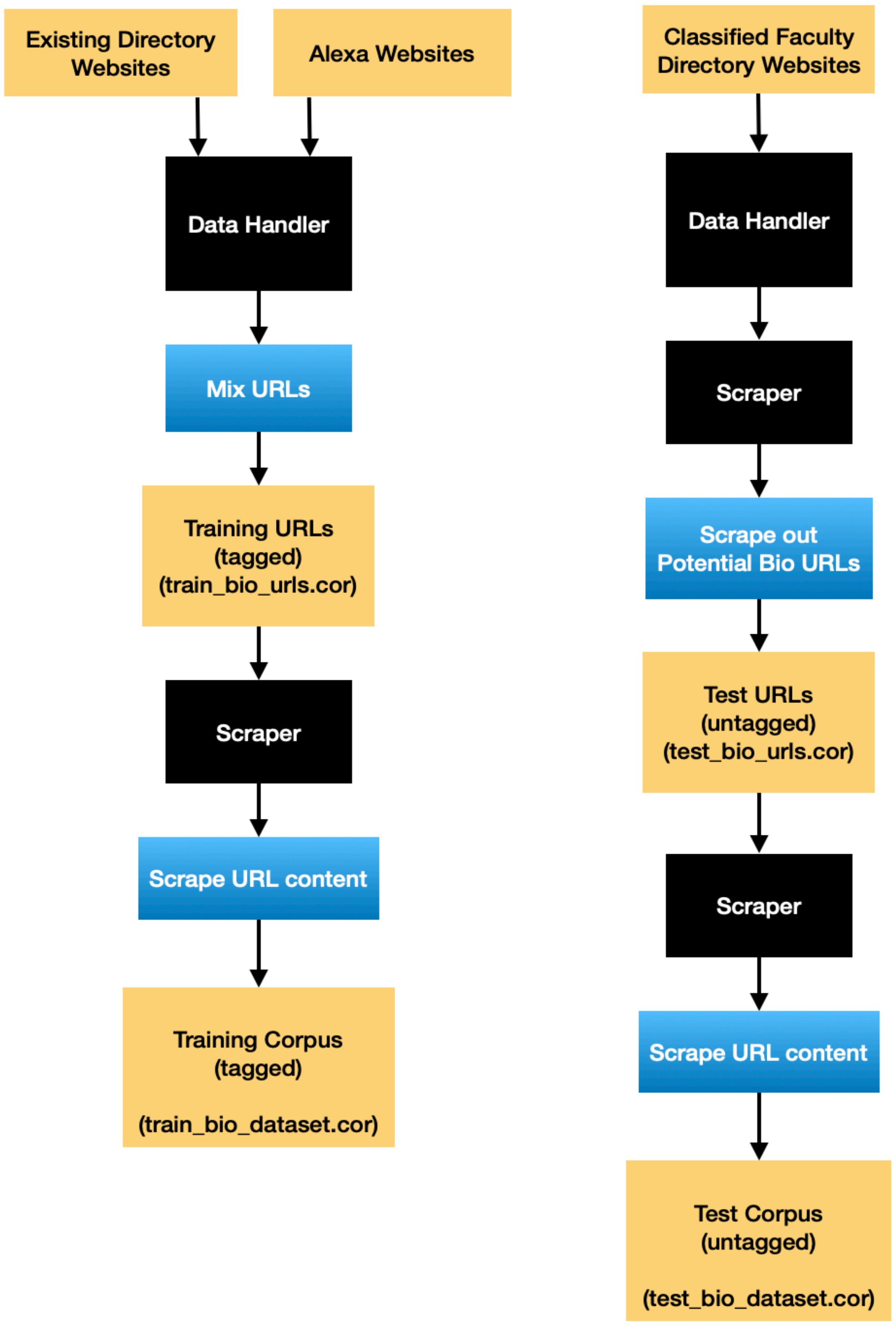
Existing Directory Websites
Alexa Websites
Data Handler
Mix URLs
Training URLs (tagged) (train_bio_urls.cor)
Scraper
Scrape URL content
Training Corpus (tagged) (train_bio_dataset.cor)
Classified Faculty Directory Websites
Data Handler
Scraper
Scrape out Potential Bio URLs
Test URLs (untagged) (test_bio_urls.cor)
Scraper
Scrape URL content
Test Corpus (untagged) (test_bio_dataset.cor)

Fig. 3: Dataset Preparation for Faculty Bio URL Classification

The following approach was used to prepare the dataset:

- Training URLs
    - Use the top 1000 URLs from the currently existing Faculty Bio URLs in the ExpertSearch project as the "**positive**" train URLs.
    - Use 250 URLs from the Alexa test URLs set as the "**negative**" train URLs.
    - Combine them.
    - Tag the faculty bio URLs as "faculty" and save to the file "**train_bio_urls.cor**".
    - Tag the Alexa URLs as "alexa_faculty" and save to the same file.
    - This will be the final file with all the train URLs.
- Test URLs
    - Use the Classified Faculty Directory URLs obtained from the Directory URL Classification task above.
    - Use the scraper to find all potential faculty bio URLs from each of these Directory URLs.
    - Save all these potential faculty bio URLs to the file "**test_bio_urls.cor**".
    - This will be the final file with all the test URLs.

Scraper (scraper.py)

Since the ExpertSearch project already contains the faculty bios as documents, the contents of the top 1000 faculty bios **(0.txt … 999.txt)** are copied to the train corpus file ("**train_bio_dataset.cor**").

Then the scraper does the following:

- Scrapes the URLs from line no. 1000 till the end from the file **train_bio_urls.cor** and appends to the train corpus ("**train_bio_dataset.cor**").
- Scrapes the contents of the test URLs (i.e. potential faculty bio URLs) from the **test_bio_urls.cor** file above and adds those contents to the test corpus ("**test_bio_dataset.cor**").

Text Classifier (text_classifier.py)

The classification module does the following:

- Uses **gensim** to build a **Doc2Vec** model for feature vector representation of each document.
- Uses the **train_bio_dataset.cor** to build the vocabulary and train the model.

- Saves the model so that it can be reloaded while running next time on the same dataset.
- Uses **LogisticRegression** as the classifier from scikit-learn module.
- Uses **LogisticRegression** to predict the categories (bio or non-bio) of the test URLs given the test dataset.

Finally, the Scraper module scrapes these classified bio URLs and saves the contents of each bio URL to a new file under *ExpertSearch/data/compiled_bios*.

2.2. Topic Mining

2.2.1. Inputs:

- Generated Faculty Bio documents from the AutoScraper ***(e.g. 6530.txt)***

2.2.2. Outputs:

- Trained topic model ***(lda_mallet_model)***
- Bag-of-words representation of corpus to be used with miner.py ***(corpus_dictionary)***
- Text representation of corpus ***(lda_corpus)***

2.2.3. Deliverables:

- Python script to create topic model and retrieve top-10 terms associated with query topic ***(miner.py)***

2.2.4. Component design / Code workflow:

Topic Miner (miner.py)

The topic miner pulls a topic distribution from a document already mined if the model was trained on the document, otherwise it uses gensim and mallet to create a model from the entire corpus. The process is described below:

Corpus preparation

- Read in compiled bios as strings
- Filter the string representation of each bio to:
    - Remove stop words
    - Extract HTML tags and elements
    - Strip non-alphanumeric characters

    - Strip numbers
    - Strip words that exist in lists of terms extracted from the bios
    - Strip words that exist in a manually defined list of words that were creating incoherent topic clusters (**unwanted_words.txt**)
    - Remove words shorter than four characters
    - Split all words into a list of tokens
- Create list of documents which is comprised of lists of tokens for each document as described above
- Append bigrams and trigrams to each token list for each document
- Create a gensim dictionary from the above documents
- Create a bag-of-word representation of our documents: this will be our corpus.

Model creation

Model creation required a good deal of manual work to ensure that the term clusters were understandable. The process consisted of a lot of trial and error, using the steps below **(Fig. 4)**:

- Create a general model with **gensim.models.ldamodel.LdaModel** class with 10 models
- Visually inspect term clusters to ensure they were meaningful
- If the above criteria were not satisfactory:
    - Tweak corpus construction
- After the above criteria was deemed satisfactory:
    - Using **gensim.models.wrappers.LdaMallet** with the mallet library, I:
        a. Varied number of topics to create new model
        b. Assessed coherence of each model with varying number of topics
        c. Manually inspected output of models with high coherence, looking for term clusters that made intuitive sense and that appeared distinct given knowledge of the separate domains

d. Chose the best model according to above criteria and saved it and the created dictionary for query inference

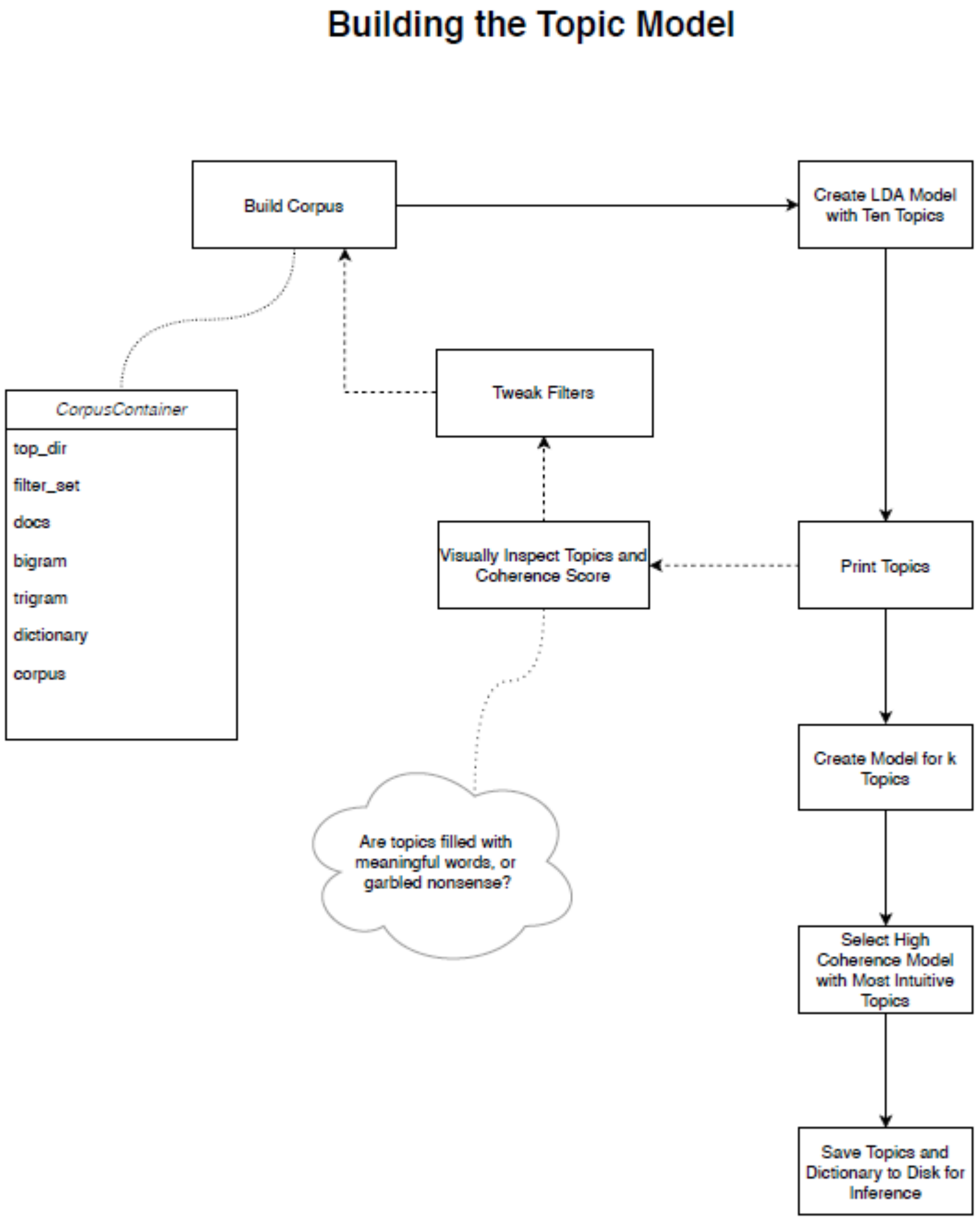
Building the Topic Model
Build Corpus
Create LDA Model with Ten Topics
Tweak Filters
CorpusContainer
top_dir
filter_set
docs
bigram
trigram
dictionary
corpus
Visually Inspect Topics and Coherence Score
Print Topics
Create Model for k Topics
Are topics filled with meaningful words, or garbled nonsense?
Select High Coherence Model with Most Intuitive Topics
Save Topics and Dictionary to Disk for Inference

Fig. 4: Workflow for building an optimal topic model.

Term extraction

With a model and a dictionary, I wrote a method that can infer the topic of a given query and fetch the top-10 terms associated with that topic, a method that can infer the topic of a single document and fetch the top-10 terms associated with that document's topic, and a method that can infer the topics of multiple documents and fetch the top-10 terms associated with each document's topic. These terms will eventually be pushed to the user to help them potentially refine their query as shown below **(Fig. 5)**.

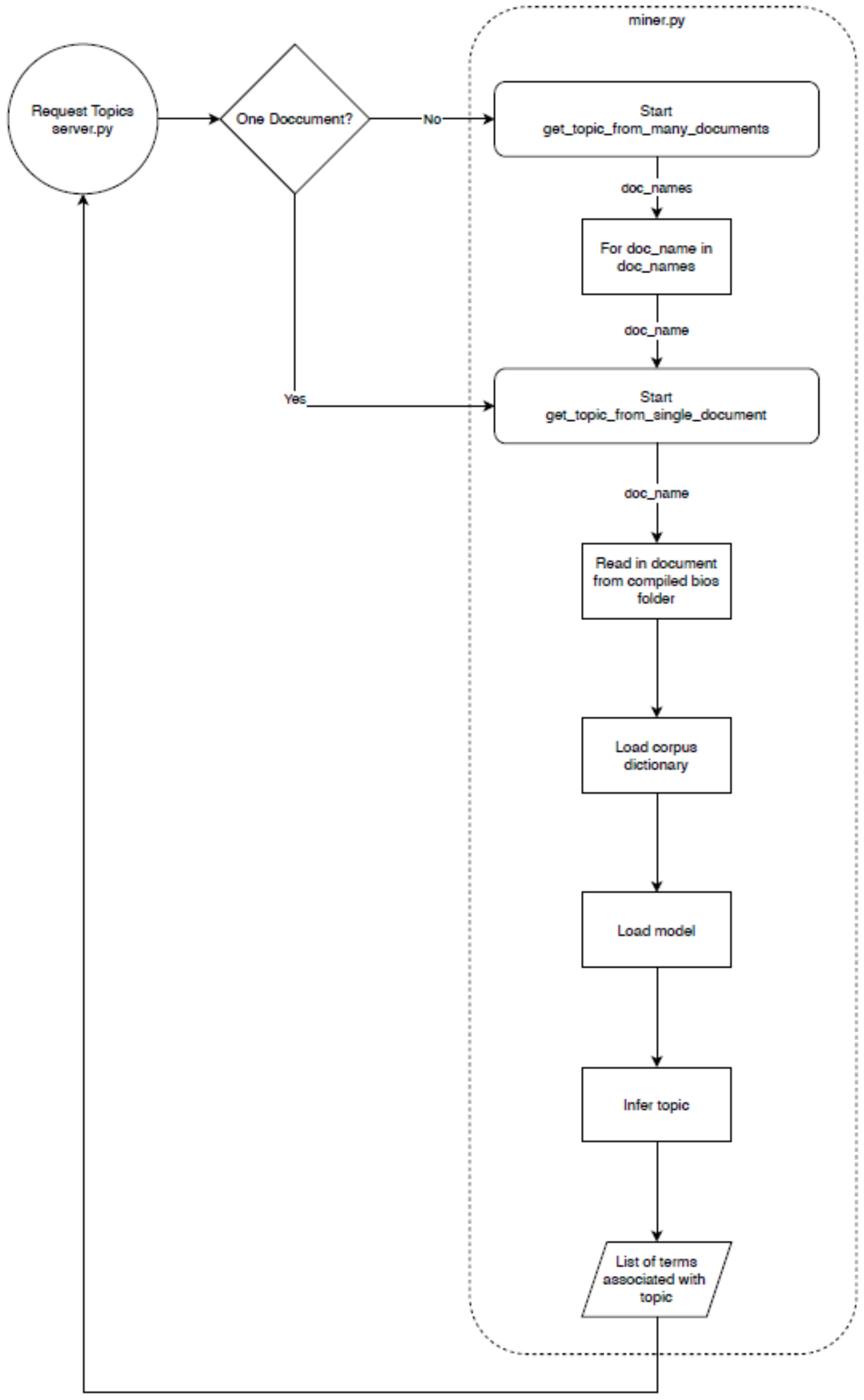
miner.py
Request Topics server.py
One Doccument?
No
Yes
Start get_topic_from_many_documents
doc_names
For doc_name in doc_names
doc_name
Start get_topic_from_single_document
doc_name
Read in document from compiled bios folder
Load corpus dictionary
Load model
Infer topic
List of terms associated with topic

Fig. 5: Extracting topics from documents not included in training set.

2.3. Improved Email Extraction

2.3.1. Inputs:

- Generated Faculty Bio documents from the AutoScraper ***(e.g. 6530.txt)***

2.3.2. Outputs:

- Extracted emails for the faculties

2.3.3. Deliverables:

- Updated email extractor ***(extract-email.py)***

2.3.4. Component design / Code workflow:

Regex Improvement:

- There are certain edge cases that we had noticed in some of the email web pages where the format was different than the traditional email formatting
- Added more regex matches in order to match with these edge cases such as ex. rohini[@]buffalo[DOT]edu

2.4. UI Improvements

2.4.1. Inputs:

- Query terms in the search box
- Topics mined by the topic miner

2.4.2. Outputs:

- Updated UI showing top-5 research topic per faculty
- Updated UI showing topic cloud
- Clickable topic terms for query refinement
- Prepopulated email template on-click email icon

2.4.3. Deliverables:

- Updated server endpoints ***(server.py)***
- Updated UI ***(index.js)***

2.4.4. Component design / Code workflow:

Info Button:

- Information button is created at the top of each of the retrieved faculty.
- When the button is clicked there a table pops up that appears below the selected retrieved faculty
- The table will contain additional information regarding the research topics that the faculty does
- When one of the info buttons associated with a faculty is clicked the other one will close, and the new one will open.

Top 5 Topics Display:

- Display the top 5 topics from the preview for each of the faculty.
- Display these topics in a table format when the information button is clicked
- Underneath each topic is a ‘Learn More’ button which when clicked leads you to a page talking more about the topic in detail from the web.
- Clicking on the “Add to Query” button will refine the current search query to include this topic.

Email Automation:

- Email comes pre-populated with a set subject and body.
- The body talks about one of the research topics that were extracted from the top 5 topics and how the user would like to connect with the faculty regarding research in this topic.

3. Usage Details

The modified project has been tested on Mac and Windows with Python 2.7. Here are the setup instructions for each of these platforms.

3.1. Setup Guide (Mac)

3.1.1. Repo setup

Run the following command on a terminal to clone the github repository.

$ git clone https://github.com/sairanga123/CourseProject.git

The directory structure of the project is as below (listing only the files/folders relevant to this project):

```
CourseProject
|___________ ExpertSearch
             |___________ AutoScraper
             |            |___________ data
             |            |___________ auto_scraper.py
             |            |___________ data_handler.py
             |            |___________ scraper.py
             |            |___________ text_classifier.py
             |            |___________ d2v.model
             |            |___________ d2v-bio.model
             |
             |___________ data
             |            |___________ compiled_bios
             |            |___________ expertsearch
             |                         |___________ mallet-2.0.8
             |                         |___________ model_files
             |                         |___________ corpus_dictionary
             |                         |___________ lda_mallet_model
             |                         |___________ lda_corpus
             |                         |___________ miner.py
             |
             |___________ extraction
             |___________ mallet-2.0.8
             |___________ static
             |___________ server.py
```

3.1.2. Project environment setup

The project has been tested on python 2.7. Please setup a python 2.7 environment for running the project. Creating an environment from Anaconda will make many common packages available. So a quick way to start would be to setup a python 2.7 environment from Anaconda.

May need to install many or all of the following python packages depending on what packages the python environment already has.

- gunicorn=19.10.0
- flask=1.1.2
- metapy=0.2.13
- requests=2.25.0
- pytoml=0.1.21
- gensim=3.8.3
- nltk=3.4
- bs4=0.0.1
- lxml=4.6.2
- numpy=1.16.6
- sklearn=0.0

3.2. Setup Guide (Windows)

Windows is currently not supported. If you want to build in Windows, use Windows Subsystem for Linux and follow the steps above.

3.3. Usage Guide

3.3.1. Running the Automated Scraper

Run the AutoScraper to generate the bio documents. This step has been already performed and the generated bio documents have already been added to the **ExpertSearch/data/compiled_bios** folder. Here are the instructions for running the AutoScraper if it needs to be run again with additional input.

To run the automated scraper, first go the ExpertSearch/AutoScraper directory.

```
$ cd ExpertSearch/AutoScraper
```

Then the Automated Scraper can be invoked as follows:

```
$ python ./auto_scraper.py -d -t
```

-d option specifies to generate/regenerate the train and test dataset.
If **-d** switch is used:

- If the dataset already exists, it will be regenerated
- If the dataset doesn't yet exist, it will be generated

If **-d** switch is not used:

- If the dataset already exists, the existing dataset will be used in the subsequent flow
- If the dataset doesn't yet exist, it will be generated even if -d switch is not used

-t option specifies to train/retrain the Doc2Vec model on the train dataset.
If **-t** switch is used:

- If a trained and saved model already exists, the model will be retrained and saved again
- If a trained and saved model doesn't yet exist, it will be trained and saved

If **-t** switch is not used:

- If a trained and saved model already exists, the saved model will be loaded and used for inference in the subsequent flow
- If a trained and saved model doesn't yet exist, it will be trained and saved even if -t switch is not used

3.3.2. Running the Topic Miner

The steps for creating the topic model are documented in **ExpertSearch/data/expertsearch/LDATopicModeling.ipynb**. The model construction is not something that can necessarily be automated because relying on perplexity and coherence scores alone often results in topics that don't make any meaningful sense to a human.

Once the topic model is constructed and saved, miner.py allows the server to load the model and make inferences.

3.3.3. Running the Backend Server

Once, the topic model has been built, we can start the server.
To start the server, go to the ExpertSearch folder.
Then run the following command:

```
$ gunicorn server:app -b 127.0.0.1:8095
```

3.3.4. Running Faculty Search from the UI

Now, launch a web browser and type the following URL:

localhost:8095

3.4. Example Use Cases:

3.4.1. Use Case 1 – Basic use to search faculties

Let's assume that we want to find the faculties that are working on "**text mining**".

Then we'd go to the UI and enter our search string as "text mining".

The existing system would retrieve the top ranked faculty results working on "text mining".

3.4.2. Use Case 2 – Find research interests of the faculty

Now, the improved system will also provide an info button which will bring up an additional table of information for each faculty. This table shows the top 5 research topics the faculty is associated with.

3.4.3. Use Case 3 – Find faculties working on similar topics as the faculty from the initial search results

We now maybe interested in learning more about who are the faculties that are working on any of these research topics. We can quickly search for all the faculties working on this new research topic by simply clicking on the "Add to Query" button for that research topic. This will automatically modify our search query by including that new research topic without having to type it in the search box. The retrieved faculty results will show the list of faculties working on that research topic.

3.4.4. Use Case 4 – Connecting to faculty

Another way we could use the system is to click on the email icon to send an email to the faculty's email address. While we may be at a loss of words for that first email, the system will provide a pre-populated template email which will automatically address the faculty's name and also include reference to the faculty's research area. This will make connecting to an expert faculty just one click away.

4. Contributions

Item	Sub-items	Contributor
Automated Scraping	• Automated Scraper to automate the complete process • Data Handler to prepare the datasets for the text classification tasks • Scraper to scrape the URLs • Text Classifier to classify directory and bio URLs • Function to generate bio documents and add to compiled bios	Mriganka Sarma
Topic Mining	• Topic model • Function to return top-10 words associated with query topic	Zacharia Rupp
Improved Email Extraction	• Added regular expressions to extract emails with atypical forms (e.g. person at place dot com)	Sai Ranganathan Zacharia Rupp
Improved UI	• Display top 5 topics associated with each faculty member • Display cloud of topics • Pre-populate email field when clicked on email address • Improve email extraction part 1.	Sai Ranganathan