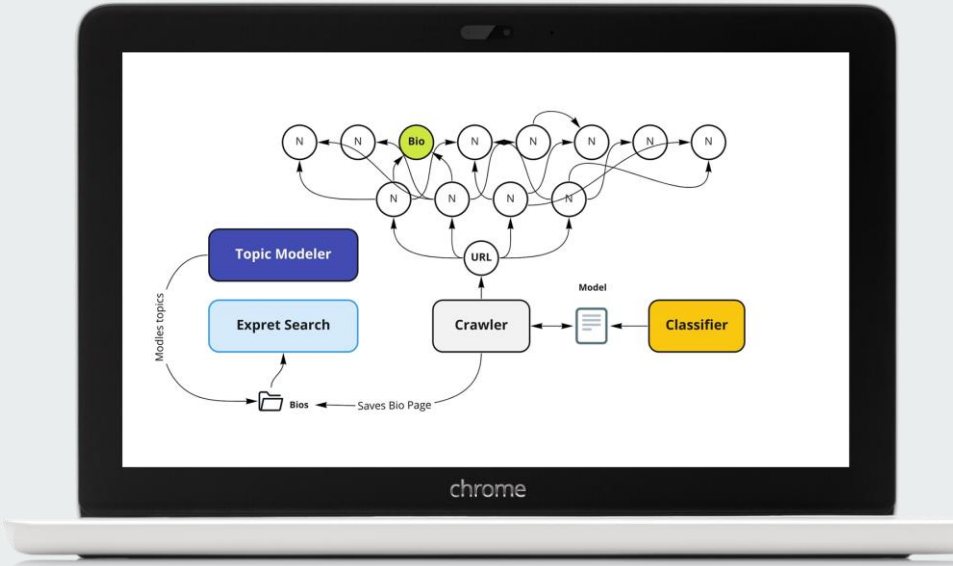
Expert Search

Improvements to Expert Search System: Bio Page Classification, Automated Crawling, and Topic Modeling

Authors:
Bashir Partovi, Karthik Rajagopal, and Mohana Venkata Kalyan Cheerla

Fall 2020 - CS410 - Course Project



Outline

Overview

Bio Classifier

Automatic Crawler

Topic Modeler

End Results

Demos

Overview

Requirements

- ➔ The user should be able to view topics for each search result
- ➔ University websites should be regularly crawled for automatic discovery of bio pages
- ➔ Crawler requires a classification model in order to identify faculty bio pages
- ➔ Classifier should use the provided compiled bios as positive samples and be able to generate negative samples by crawling the web

Assumptions

- Expert Search system runs smoothly on its own
- It is easy to setup Expert Search system locally on developer’s machine
- Expert Search system python runtime could be scaled to the latest version
- MeTA python library is also available in Python 3.8
- Training the classifier is not a computationally intensive task

Solution

Solution description

The system consists of 3 independent components that run concurrently to provide data to Expert Search system to enhance the search results.

Bio Page Classifier utilizes compiled bios as positive training data along with a set of randomly crawled web pages as negative data to train a model to identify faculty bio pages.

The crawler uses such model to crawl universities websites in order to obtain faculty bio pages and stores such pages for Expert Search as seed data.

Topic Modeler runs on stored bio pages in order to model topics to be displayed as part of the search results

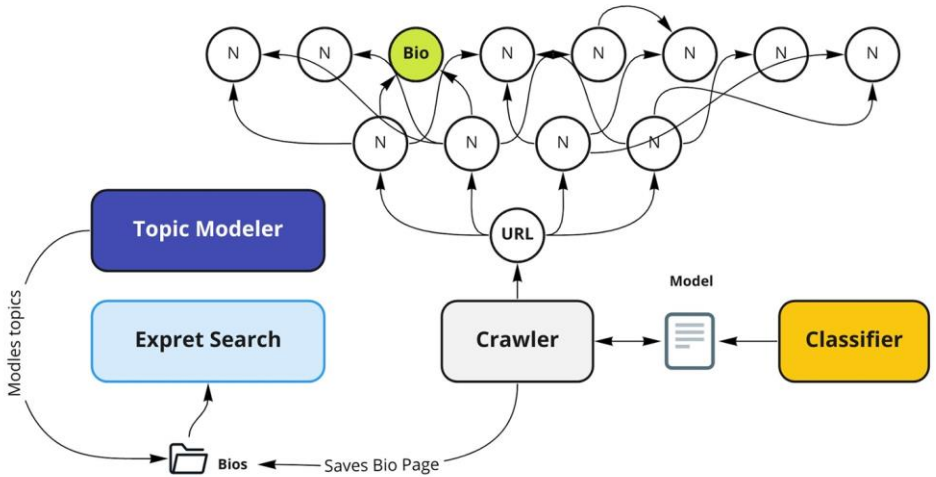
Why it’s better than the existing solution

- New faculty pages are automatically discovered and indexed
- Search results include topic tags that could be used to create new search queries
- No manual intervention is needed by the administrator to prepare the bio data

General Architecture

The system has 4 separate components:

- **Expert Search**
- **Topic Modeler**
- **Bio Classifier**
- **Automated Crawler**



Bio Page Classifier

Creates a model to identify faculty bio pages

Bashir Partovi

Why a Classifier?

- ➔ Crawler needs to be able to identify faculty bio pages in order to store them for Expert Search consumption
- ➔ Eliminates manual identification of web pages

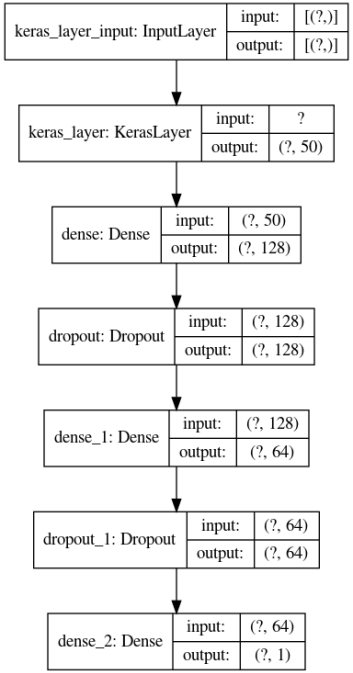
Technology Stack

- ➔ Tensorflow and Keras were used to implement the training algorithm
- ➔ The classifier is written in Python due to ease of use of ML frameworks such as Tensorflow and Keras
- ➔ Compiled bios were used as positive examples and a new set of websites were crawled to obtain negative examples
- ➔ Multiple training algorithms were used to see whether the accuracy increases
- ➔ Reached accuracy of 99.95%

Tensor Layers

The layers consists of

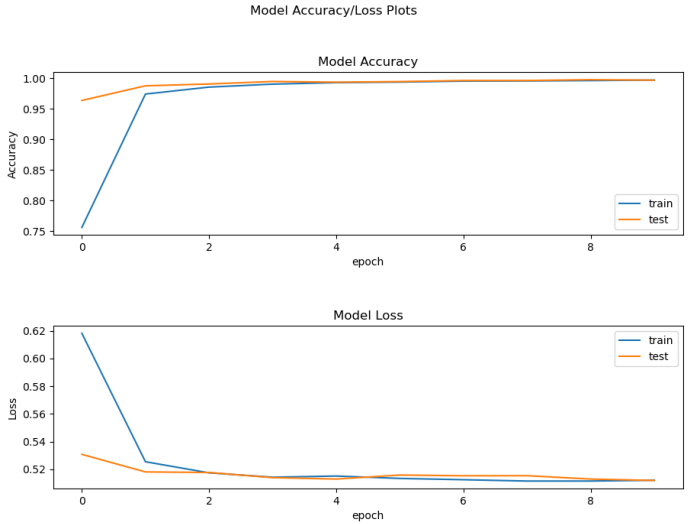
- ➔ An embedding layer (input shape)
- ➔ A dense layer with 128 units and relu activation
- ➔ A dropout layer of 0.5 ratio to avoid overfitting
- ➔ A dense layer of 64 units
- ➔ Another dropout layer of 0.5 ratio
- ➔ A dense layer of 1 unit with sigmoid activation



Classifier in Action

How did it perform?

- ➔ Unfortunately, even though the accuracy is very high, the model overfits the data
- ➔ The accuracy spikes up to 95% on training data at the second epoch
- ➔ Better model? Yes, but it crashed the due to hardware limitation



Challenges

- ➔ Learning curve on Tensorflow and Keras was steep
- ➔ Model overfits the data
- ➔ Local hardware limitation in training the classifier (even crashed Google Colab virtual machine)

Automated Crawler

Regularly runs to discover faculty bio pages by crawling universities websites

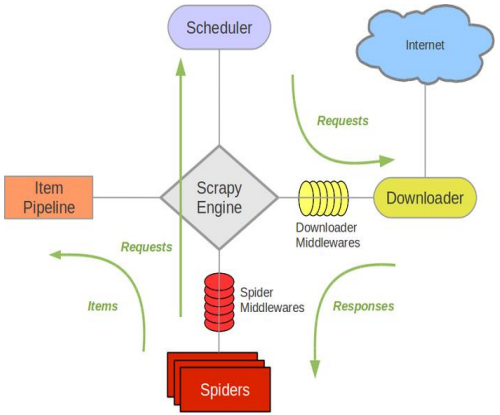
Mohana Venkata
Kalyan Cheerla

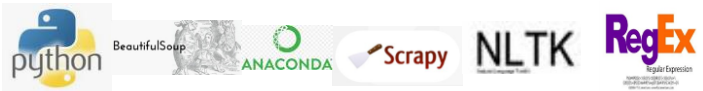
Why an Automated Crawler?

- ➔ Manually searching the web and extracting faculty bio pages are time consuming
  - ◆ It's tedious
  - ◆ Doesn't scale
  - ◆ Navigation is difficult
- ➔ Our Automated Crawler crawls university websites to automatically fetches bio pages
- ➔ It converts the Unstructured Markup text into Semi-structured normalized text data to feed its downstream components

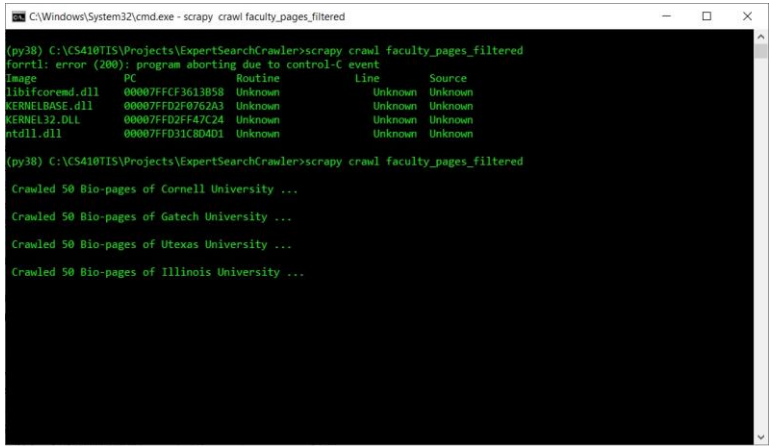
Technology Stack

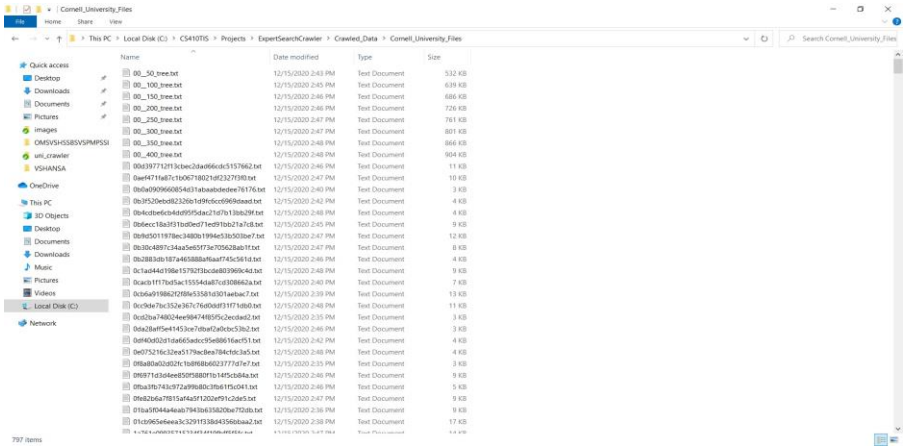
- ➔ The crawler runs on a cron schedule to to passively crawl universities websites
- ➔ The crawler is developed in Python. It utilizes the Scrapy library and follows Breadth First Search (BFS) up to the designated depth specified by the User/Administrator
- ➔ Crawler has Spiders that recursively follow all the web links of the University Website and extracts the web page data along with the Website's "URL Tree" to understand, validate, troubleshoot and trace



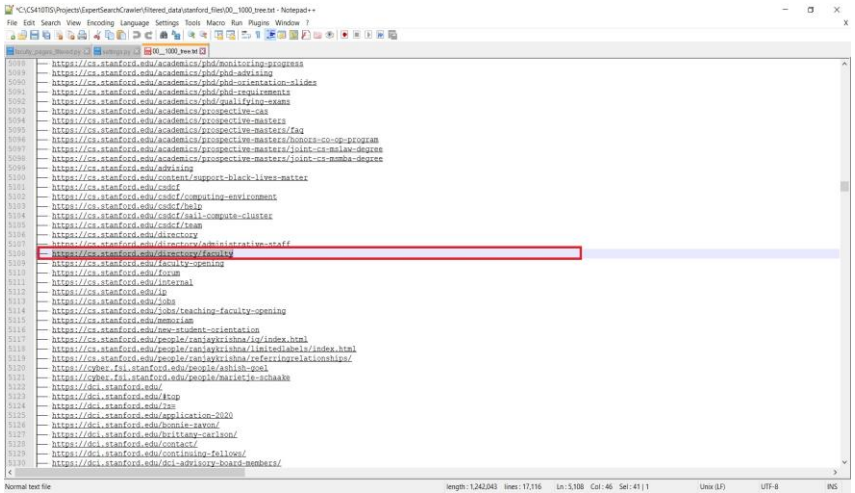


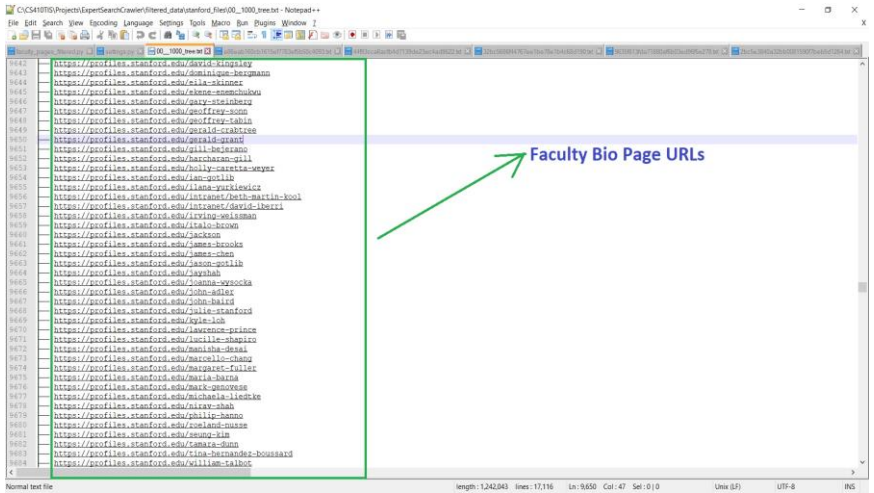
Crawler in Action





Crawler in Action (contd…)





Challenges

- ➔ Crawler runs a long time and drains computational resources on the developer's machine
  - ◆ Parallelization over multiple machines?
- ➔ Optimizing the crawler was challenging
  - ◆ We had to eliminate different branches of crawl tree that were uninteresting, e.g. publications, news, sports

Topic Modeler

Models topics for each search result

Karthik Rajagopal

Why Topic Modeler?

- ➔ We wanted to enhance the Expert Search system with the topic modeling of the faculty bio pages to tag search results with top relevant skills
- ➔ This feature enhances user experience by enabling him/her to start a new search query based on a relevant tag

Technology Stack

- ➔ Spacy is a fast library and provides a concise API to access its methods and properties
- ➔ We used POS Tagging, Named Entity Recognition (NER) and tokenization features of spacy
- ➔ Used Tokenization to process text for use with POS tagging and Named Entity Recognition
- ➔ POS is used for removing the unwanted and noisy words
- ➔ NER feature has helped us find the common entities such as persons, locations, organizations, etc. and distinguish them from noise words in bios
- ➔ TinyDB was used to store the final results of Topic Modeling

Topic Modeler Code Snippet

Code for removing noise word before finding the keywords

```
def isNoise(token):
    is_noise = False
    if token.pos_ in noisy_pos_tags:
        is_noise = True
    elif token.is_stop == True:
        is_noise = True
    elif len(token.string) <= min_token_length:
        is_noise = True
    elif token.string.lower().strip() in stop_words:
        is_noise = True
    elif token.string.strip() in ents:
        is_noise = True
    return is_noise
```

Code for finding NER:

```
ents = [e.text for e in document.ents]
```

Code for finding top keywords

```
from collections import Counter
cleaned_list = [cleanup(word.string) for word in document if not isNoise(word)]
counts=Counter(cleaned_list).most_common(5)
```

Code for Saving data into TinyDB

```
for key in counts:
    a.append(key[0])

db = TinyDB('Topic_Model_Results.json')
db.insert({'File_Name': i, 'Topic_Modelled_word': a})
```

Topic Modeler in Action

Below is sample file we used for Topic Modeling (highlights are the topics):

Home Bio Research Publications Research Group Contact Info Engineering at Illinois Vikram S. Adve Interim Head and Donald B. Gillies Professor|Computer Science Department|University of Illinois at Urbana-Champaign Research Projects Prof. Vikram Adve F ollow the individual project links for more details about each project and relevant publications: ALLVM: Exploring the benefits for software performance, security and reliability if all software on a system (either all userspace software or or userspace+OS software) is available in a rich virtual instruction set that can be analyzed and transformed by sophisticated compiler techniques (think Java bytecode, but for all software). Heterogeneous Parallel Virtual Machine : A compiler infrastructure and parallel program representationfor heterogeneous parallel systems, with the goal of making it much easier to write performance-portable parallel programs. A single program in the HPVM representation can be compiled to GPUsand tomulticore CPUs (with and without) vector extensions, while achieving performance close to separately hand-tuned code for each of those systems. The project is alsoexploring code generation for FPGAs, for specializeddeep-in-memory compute hardware, and developing optimizingcompilersforparallel languages like OpenMP and Domain Specific Languages. Automated Debugging for Software Failures : We are developing automated static and dynamic analysis techniques tounderstand the causes of failures in software systems, in order tohelp programmers diagnose and fix software bugs with as little effort as possible. The project is investigating automated fault localization and diagnosis techniques for both standalone and distributed

You can see he is part of research team and his interest is on compilers. Output from the model:

```
from tinydb import TinyDB, Query
db = TinyDB('Topic_Modelling.json')
result = db.get(Query()['File_Name'] == '2.txt')
print(result)
```

```
{'File_Name': '2.txt', 'Topic_Modelled_word': ['research', 'compiler', 'software', 'students', 'projects']}
```

Challenges

One of the challenges we faced was to setup the Expert Search system working locally.

Our initial goal was to have the modeled topics displayed on the webpage. We tried different options and as we could not do that, we transformed our code to save the results in TinyDB and retrieve data whenever needed.

Setup instructions of the DB has been added in the Git README page.

End Results

In the end ...

- ➔ Unfortunately many of our assumptions were not correct
  - Expert System did not run on Linux or Mac
  - MeTA python library was incompatible with new version of Python
  - Expert Search Docker container continuously crashed
  - Even when the server ran, the indexer did not show any results
- ➔ Classifier overfits the data and generates inaccurate scores on most university pages

Live Demo