TOWARDS A CHARACTERISTICS-AWARE DOCUMENT SEARCH ENGINE

BY

LONG T. PHAM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Kevin C.C. Chang

ABSTRACT

The increasing volume, heterogeneity, and redundancy of the Web create a novel challenge for search engines in which, target documents must satisfy some characteristics. It is increasingly important because there are more and more types of web pages on the Internet nowadays. Current web search engines are fundamentally incapable of addressing the user need because keywords can not express characteristics of target pages. Another alternative is to use vertical search engines, but they can only cover a few popular niches. Thus, we propose Forward Search to empower users with an engine to express not only topics but also characteristics of pages in their queries. Expected results are documents ranked by both topical and characteristic relevance.

Creating Forward Search to have the focus of a vertical engine and the flexibility of web search presents many novel challenges. First, we must represent a document with novel information to support querying for characteristics. Second, we must index both keywords and named entities to quickly locate relevant pages of the target characteristics during query time. Third, we have to design a realistic method to acquire user input about the characteristics of target pages and retrieve documents ranked by both topical and characteristic relevance. Finally, since our system redefines relevance in traditional web search, we must rethink the user interface. This thesis comes with a full-functioning web-based demonstration of our Forward Search at `http://crow.cs.illinois.edu:8080` and five open-source code repositories at `https://github.com/forward-uiuc`.

To my life partner and my family, for their love and support.

ACKNOWLEDGMENTS

First, I would like to thank my adviser, Professor Kevin Chang, for not only the insightful discussions but also the rigorous coaching I needed to dig deeper and think bolder.

Second, I am thankful to Mangesh Bendre, Dr. Yuan Fang, Dr. Vincent Zheng, Denghao Ma, Abhinav Kohar, Herbert Wang, Lam Vu, Guangyu Zhou, Shriyak Sridhar, Zubin Pahuja, Dr. Fanwei Zhu, Professor Yueguo Chen, Alex Aulabaugh, Phuong Cao, and other collaborators at the University of Illinois for working with me, teaching me and encouraging me during the process.

Last, but not least, I would like to thank my life partner and my family for the emotional support I needed to complete this challenging process.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The increasing volume, heterogeneity, and redundancy of the Web create a novel challenge for search engines. Considering the scenarios below:

Scenario 1.1. *Alice is considering buying the newest Sony camera, and she wants to see side-by-side photo comparisons between the camera and its competitors. However, most reviews and comparisons retrieved by traditional search engines are similar and only focus on technical specifications. There are indeed blog posts comparing photos side-by-side, but they scatter in thousands of result pages.*

Scenario 1.2. *Bob is preparing for his graduate school applications, and he wants to collect the homepages of professors with interest in* Information Retrieval. *However, top results from traditional search engines give him just a few professor homepages mixed with many course pages, department news, and faculty directories.*

Scenario 1.3. *Carla usually looks for food recipes on the Internet, but she is only interested in those with step-by-step instructional photos. However, top results from traditional web search engines incorrectly give her pages with how-to videos because they usually contain the keyword "step-by-step".*

In the scenarios, target documents must satisfy some characteristics. It is increasingly important because there are more and more types of web pages on the Internet nowadays. For users, on the one hand, it is an unprecedented opportunity to find the content that is not only topically relevant but also suitable for their unique style and needs. On the other hand, it is overwhelming because pages of desirable characteristics may not be among the top results. The matter becomes worse if the target pages are written by some niches such as bloggers, which are usually ranked lower than those from authority sources such as news agencies.

Current web search engines are fundamentally incapable of addressing the user needs because keywords can not express characteristics of target pages. There are two essential barriers. First, authors do not use keywords to describe characteristics of their pages explicitly. For example, most professor homepages do not contain the word "homepage", many "food blogs" include neither "food" nor "blog" in their websites, and recipes with step-by-step instructional photos do not write "step-by-step." Second, there may be correlations between keywords and characteristics of pages, but they correlate in a non-trivial, indirect and unreliable way. For example, one may add word "phone" to query "professors in information retrieval," because professors usually include their phone numbers in their

homepages. However, such rules are unnatural for users to come up with, and they are unreliable because some authors may write his number without explicitly saying "phone".

Another alternative is to use vertical search engines. However, they can only cover a few popular niches such as rental cars, hotel booking, restaurant reviews, and are limited to a few data sources because they are expensive to create. Moreover, knowing which vertical engine to use for a task is already difficult for most users.

Thus, we propose Forward Search to empower users with an engine to express not only topics but also characteristics of pages in their queries. Expected results are documents ranked by both topical and characteristic relevance.

Creating Forward Search to have the focus of a vertical engine and the flexibility of web search presents many novel challenges.

First, we must represent a document with novel information to support querying for characteristics. We observe that named entities, such as persons, organizations, locations, emails, are intuitive and useful in representing document characteristics. For example, a web page that contains a full name on the page title, some contact info near the header, and a list of publications near the bottom, is likely the personal homepage of a researcher. Compared to keywords, using named entities is more direct as they do not always co-appear. Thus, we propose to model a document with both keywords and named entities.

Second, we must index both keywords and named entities to quickly locate relevant pages of the target characteristics during query time. Inspired by researches in Entity Search, we propose a method to index named entities and their positions in the same inverted index architecture with keyword search technology. The immediate benefit of this design is we can quickly deploy our solution to existing production search engines such as Apache Lucene.

Third, we have to design a realistic method to acquire user input about the characteristics of target pages and retrieve documents ranked by both topical and characteristic relevance. As writing direct queries with keywords and named entities could be challenging for most users, we propose a two-step interface: first, users write queries to retrieve topically relevant documents, and second, they specify a few pages in the search results as examples of the target characteristics. Our system then uses novel algorithms to infer queries from the two steps to maximize the topical and characteristic relevance of search results. As the first step is exactly how users do every day and the second step is optional if they want to narrow down the results to satisfy some characteristics, our system fits in the workflow of traditional web search seamlessly.

Finally, since our system redefines relevance in traditional web search, we must rethink the user interface. In particular, we shall sketch a new search result page to help users quickly identify pages with target characteristics without clicking on the links. This thesis

comes with a full-functioning web-based demonstration of our Forward Search at `http://crow.cs.illinois.edu:8080`.

CHAPTER 2: CONCEPT

2.1 CHARACTERISTIC RELEVANCE

Our ultimate goal is to empower users with a novel tool to better exploit the increasingly heterogeneous Web. Since thirty years ago, web search has made tremendous progress regarding both efficiency and relevancy. Even novice users nowadays can quickly find relevant content from trustworthy sources. However, going beyond top search results is still a painful process, because sorting through the retrieved links is tedious, while revising queries may change the topics. For example, a query like *sony a7 iii review* gives us many good reviews about the camera, but if someone is only interested in the ones containing ISO information for each sample photo, he must go through all search results until finding them. Changing the query to *sony a7 iii review iso* is not helpful because the search engine will prioritize irrelevant reviews about the ISO aspect of the camera. Thus, we propose a new measure of usefulness for search results called characteristic relevance, which should be distinguished with topical one. The new type of relevance is increasingly important now because the Web is more and more topically redundant.

Definition 2.1 (Topical relevance)**.** *Topical relevance $R(q_t, d)$ denotes how well a retrieved document d contains topics in user query q_t.*

Definition 2.2 (Characteristic relevance)**.** *Characteristic relevance $R(q_c, d)$ denotes how well a retrieved document possesses characteristics the user describes in query q_c.*

As both types of relevance are necessary for a document to be useful, we need to measure the combined relevance score $R(q_t q_c, d)$:

$$R(q_t q_c, d) = log(P(q_t q_c|d)) \tag{2.1}$$

$$P(q_t q_c|d) = P(q_t|d) * P(q_c|q_t d) \tag{2.2}$$

The two types of relevance are intuitively independent of each other. In particular, a document may be topically relevant but characteristically irrelevant with a query. And vice versa, it may be topically irrelevant but characteristically relevant with another query. Thus, Equation 2.2 becomes:

$$P(q_t q_c|d) = P(q_t|d) * P(q_c|d) \tag{2.3}$$

$P(q_t|d)$ can be calculated using existing techniques and features. Since keywords cannot represent page characteristics, to calculate $P(q_c|d)$, we need to model documents with additional information, which will be discussed as below.

2.2 ENTITY-SEMANTIC DOCUMENT MODELING

We observe that named entities, such as persons, organizations, locations, emails, are intuitive and effective in representing document characteristics. For example, a web page that contains a full name on the page title, some contact info near the header, and a list of publications near the bottom, is likely the personal homepage of a researcher. Compared to keywords, using named entities is more direct because we can query for a phone entity instead of keyword "phone". Thus, we propose to model a document as a list of both keywords and named entities.

$$d = \{w_1, w_2...e_1, e_2...\} \tag{2.4}$$

For example, document *phone 217-111-2222* can be modeled as { *phone, 217-111-2222, #phone* }, in which *#phone* represents a named entity of type phone, and it has the same position with keyword *217-111-2222*.

2.3 THE OPTIMIZATION PROBLEM: MAXIMIZATION OF TOPICAL AND CHARACTERISTIC RELEVANCE FOR DOCUMENT RETRIEVAL

The goal of Forward Search is to retrieve documents not only topically but also characteristically relevant with user query. Thus, a document d in corpus D must be ranked using Equation 2.3. It is however difficult to measure the true $P(q_tq_c|d)$ because it requires calibration for $P(q_t|d)$ and $P(q_c|d)$. Within the scope of this thesis, we assume the calibration can be done manually by parameter ξ as follows:

$$P(q_tq_c, d) = P(q_t|d) * P(q_c|d)^{\xi} \tag{2.5}$$

Combined with Equation 2.1, the final relevance score can be calculated as below:

$$R(q_tq_c, d) = R(q_t|d) + R(q_c|d) * \xi \tag{2.6}$$

Intuitively, ξ can be understood as a parameter to control the user preference between the two types of relevance. And Equation 2.6 reflects our retrieval objective to maximize both types of relevance.

Implementing the concept in Equation 2.6 is challenging with multiple components. We will discuss the general architecture in Chapter 3, and then each component in later sections as follows:

- Chapter 4 discusses the architecture for modeling documents $d \in D$ with both named entities and keywords.
- Chapter 5 discusses the architecture for injecting named entities into the existing inverted index structure for keywords.
- Chapter 6 discusses our language to query for both named entities and keywords.
- Chapter 7 discusses the interface to acquire user input about the target characteristics and a simple method to infer q_c from the input.
- Particularly, with the introduction of characteristic relevance, we must rethink the entire user interface to display search results and interact with users, which will be discussed in Chapter 8.
- Finally, evaluation, related work, future work, and conclusion will be discussed in Chapters 9, 10, 11, and 12.

CHAPTER 3: ARCHITECTURE

3.1 DESIGN GOALS

Our primary goal is to enable searching for ad hoc page characteristics. It means that our system must not only understand page characteristics but also retrieve relevant documents quickly. Page characteristics shall be ad-hoc, i.e., defined at query time, because user needs are unlimited, and it is more convenient for them to tell their needs than to memorize a long list of supported types.

Furthermore, we want to support research on Entity Semantic Document Search and related fields. Thus, we will decouple components of our system, and publish them as separate repositories. We will add features to enable Forward Search on a new corpus as quickly as possible. One of our initial ideas is to allow researchers to explore their raw corpus using our system rapidly.

Finally, we want to bring our research into current business practices. Therefore, we will use well-established data structures and technologies as much as possible. When customizing external libraries, we will extend classes or write plugins to increase compatibility with existing projects.

3.2 SYSTEM FEATURES

To use Forward Search for a new domain, the administrator only needs to provide us with URL patterns for web pages in the domain. For example, they may provide us with `*.cs.illinois.edu` to build a search engine for the entire website of Computer Science Department at the University of Illinois. They can also customize the procedure by bypassing some of our components in the workflow. For example, instead of giving URL patterns, they can provide us with a list URLs in their sitemap directly.

The list of features is as follows:

Feature 3.1 (URL Crawling)**.** *Input is URL patterns, and output is a list of URLs obeying the patterns. We provide a Python script, which uses Common Crawl API, for this task.*

Feature 3.2 (DOM Rendering)**.** *Input is URL list, and output is a list of DOM objects. We use Selenium as the connector and Chrome browser as the driver to render the web pages. Please note that as Selenium API is restrictive, we cannot materialize the DOM objects. Instead, we maintain a connection between Chrome browser, which contains the DOM objects, and the next step (annotation) through Selenium.*

Feature 3.3 (Annotation)**.** *Input is a list of DOM objects and output is a list of annotated documents with layout information for each token. We use Stanford NLP as the based annotators. It provides several built-in named entity taggers. We add a new dictionary-based annotator based on their Regex annotator, which is helpful in practical scenarios.*

Feature 3.4 (Indexing)**.** *Input is a list of annotated documents and output is an inverted index. We use the popular Lucene library for the search engine and Elastic Search as the web server. Also, we build an Elastic Search Analysis Plugin for the indexing process.*

Feature 3.5 (Topical Query Translation)**.** *Input is user query containing keywords and named entities, and output is Elastic Search DSL query. We build an Elastic Search API Plugin for the translation process.*

Feature 3.6 (Characteristic Query Translation)**.** *Input is entity-based patterns and output is Elastic Search DSL query. We build an Elastic Search API Plugin for the translation process. Topical and characteristic queries are combined in the plugin using a pre-tuned parameter. The parameter could be provided by users to specify the preference between two types of relevance.*

Feature 3.7 (Characteristics Learning)**.** *Input is a few sample pages from users and output is entity-based patterns. We create a Java program to do the learning.*

Feature 3.8 (User Interface)**.** *We use NodeJS to design the web interface. It supports multiple components of a web search engine such as snippets, pagination, auto-complete, and so on.*

3.3 SYSTEM WORKFLOW

Figure 3.1 shows the entire workflow of Forward Search. It has two parts: the left one is offline and for administrators, and the right one is online and for users. Each rectangle is a feature. Each eclipse is data or information, which is the input or output of a feature. A cylinder is like an eclipse except that it can be updated/expanded. A dotted rectangle is the boundary of a code unit if it contains at least two components.

For easy reference, we add the name of the library we use/customize in each feature. Most libraries are replaceable. However, we use Stanford NLP data structures as well as Lucene Inverted Index intensively in our system, which makes it harder to replace them.

Finally, we would like to stress that extra annotation models can be added at any time, which will then update the database of annotated documents and Inverted Index. It is

visualized in Figure 3.1 as an arrow from annotated documents back to the annotation process. This small feature is very convenient for administrators to test out their annotation models, or for researchers to use Forward Search as a fast ad-hoc text mining engine.

3.4 CODE REPOSITORIES

We created the following code repositories on Github to enable Forward Search:

- `https://github.com/forward-uiuc/Common-Crawl-URL-Searcher` contains our simple Python code to download URL of a particular domain from Common Crawl.
- `https://github.com/forward-uiuc/Entity-Search-Annotation-Indexing` contains our solution for DOM Rendering and Annotation. It also contains the code to generate the import file for Elastic Search and to generate the screenshot for each web page.
- `https://github.com/forward-uiuc/Entity-Elastic-Search-Analysis-Plugin` contains our Elastic Search plugin for customizing the indexing process, which is important for Feature 3.4.
- `https://github.com/forward-uiuc/Entity-Elastic-Search-API-Extension-Plugin` contains our Elastic Search plugin for rewriting queries from our language to DSL.
- `https://github.com/forward-uiuc/Entity-Semantic-Document-Search-Web-Interface` contains the code for web interface design. It uses NodeJS, ReactJS, and SemanticUI for easy customization and deployment.

Each repository has a README file. Most of the code is written in Java. We use either Maven or Gradle to assist package dependency resolution and deployment. Due to space limitation, this thesis contains less detailed instructions than in the README files.

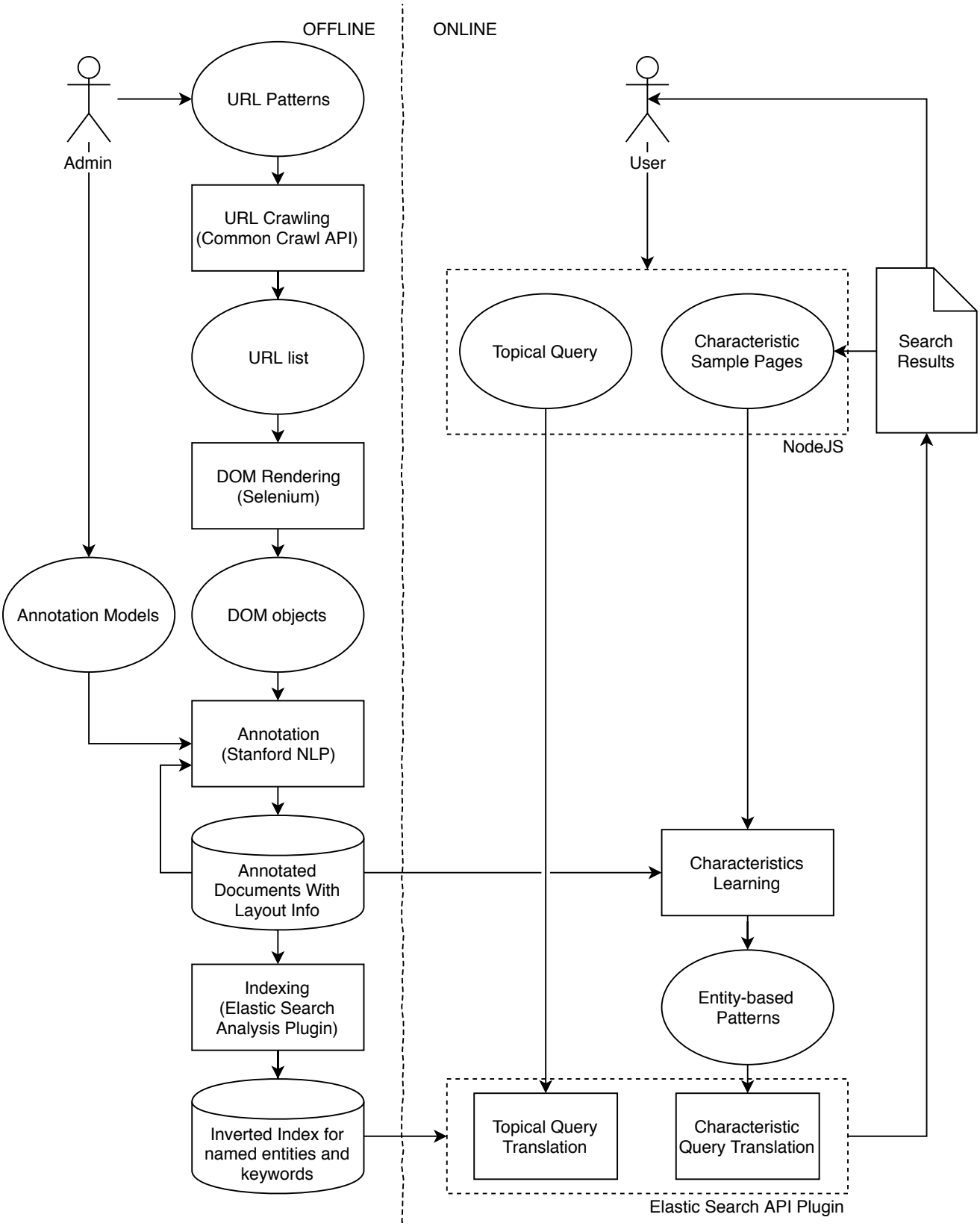

Figure 3.1: Workflow of Forward Search. Each rectangle is a system component. Each eclipse is data or information, which is the input or output of a component.

CHAPTER 4: EXTRACTION

4.1 OBJECTIVES

This step is a combination of DOM Rendering and Entity Annotation. While the former component is to extract layout information, the later one is to label named entities. They are the fuel of our system to enable querying for page characteristics. Our assumption is authors use entities and layout besides content to express the type of pages they are writing. For example, a directory page usually contains a grid or a list of photos, or a homepage usually contains a person's name on the title.

4.2 DOCUMENT MODELING

Similar to the traditional bag-of-words model, we define a document as a list of tokens. To inject information about named entities and layout, we turn each token into a rich token, which may have type and layout (i.e., coordinates and sizes) if available. The benefit of this modeling is we can extend token-document inverted index to incorporate rich information.

4.3 LAYOUT INFORMATION EXTRACTION

To extract layout information, we use Selenium and Chrome driver to render the DOM tree for each web page. We then travel the DOM tree to extract tokens in each DOM element and assign layout information for the tokens if available. In the current implementation, we assign layout information for each token based on the DOM element containing it.

We chose Selenium over other headless browsers because it enables all rendering capabilities of a real browser like Chrome. More importantly, it is actively developed by a big community. Finally, it is harder for websites to detect and refuse automatic access from Selenium than other headless browsers.

Implementing this extraction is non-trivial due to several barriers. First, traveling DOM tree is unnatural with Selenium as well as other headless browsers because their primary goal is to do unit test rather than scraping content. In particular, text nodes that do not have proper tags are not callable in Selenium which makes any built-in DOM traversal in Selenium useless. Thus, we have to do so through executing external JavaScript command to return DOM object `https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/JavascriptExecutor.html`. It is a slow process but probably the only

viable solution now. Second, as annotation algorithms usually require linguistic features such as sentence and paragraph boundaries, we have to extract tokens while maintaining this information. It is our unique challenge as we have to travel DOM tree ourselves to get the text because merely calling existing API *getText* strips all layout information and disconnects the text from the DOM tree. The detailed process is described in Algorithm 4.1.

Algorithm 4.1: Algorithm to tokenize and annotate a web document with layout information

```
Data: A web document
Result: Emit token sequentially, each of which has layout information
Set root as the document web element;
Function TravelDom(root)
    if root is a leaf node then
        Retrieve layout info for root from Selenium;
        Tokenize the text of root and emit each token with layout info;
    else
        /* Must use script instead of XPath to not miss text children   */
        foreach child of root do
            TravelDom(child)
        /* Add delimiters when needed because annotators only work on
           complete sentences                                            */
        if root is a block-level web element and last emitted token is not a delimiter
         then
            Emit sentence separator token ".";
```

4.4 NAMED ENTITY ANNOTATION

Named Entity Annotation or Named Entity Tagging is a long-standing problem. The goal is to label tokens with appropriate tags. The simplest solutions include string matching and regular expression. More sophisticated ones use graphical models, deep learning, and so on. Most techniques are relatively expensive for both training and inference. However, in our setting, we care more about the inference step, and it is can, fortunately, be applied to each small unit of text in parallel.

We can use multiple annotation algorithms but need to have one single data structure. We chose the one from Stanford NLP because it is flexible, and we can directly use existing models from the library.

4.5 STANFORD NLP ANNOTATOR

Stanford Annotation is a unit of Stanford NLP pipeline, which is famous for its tokenizer and its state-of-the-art implementation of Conditional Random Fields for Named Entity Recognition. For English, by default, this annotator recognizes named (PERSON, LOCATION, ORGANIZATION, MISC), numerical (MONEY, NUMBER, ORDINAL, PERCENT), and temporal (DATE, TIME, DURATION, SET) entities (12 classes). If adding its provided REGEX rules, it can support additional fine-grained classes EMAIL, URL, CITY, COUNTRY, STATE_OR_PROVINCE, NATIONALITY, RELIGION, (job) TITLE, IDEOLOGY, CRIMINAL_CHARGE, CAUSE_OF_DEATH (11 classes) for a total of 23 classes. In particular, they enable users to provide customized models and REGEX rules to adapt their annotator to any specific need.

The disadvantage of Stanford Annotator is it does not allow setting multiple labels for the same token. It means that "Urbana" cannot be a CITY and a LOCATION at the same time, which is very limited in our setting. Thus, we had to extend their code to support this. Now, developers can define as many slots for each token as they want, and assign different labels to different slots.

4.6 DICTIONARY-BASED ANNOTATOR

As in practical scenarios, sophisticated models like Conditional Random Fields or even REGEX-based rules are not available for many domains. However, each domain usually has a high-quality dictionary of target entities. For example, in Huawei Q&A dataset, we are provided with an excellent list of the companies' products. Entities in the list are also unambiguous, which makes string matching techniques reliable. For example, "Mate" is always the name of a phone than a drink in the dataset. Thus, we write a script to convert a dictionary to a set of REGEX rules, which can be used in Stanford NLP Annotation module.

4.7 THE COMPLETE WORKFLOW

We separate the processes of crawling and annotation such that developers can always add new entities to the documents. The index can also be updated, which makes it convenient to tune the index. It also makes Forward Search suitable for ad hoc text mining tasks, when users can quickly use patterns to query text data and add more patterns along the way. The workflow is visualized in Figure 4.1.

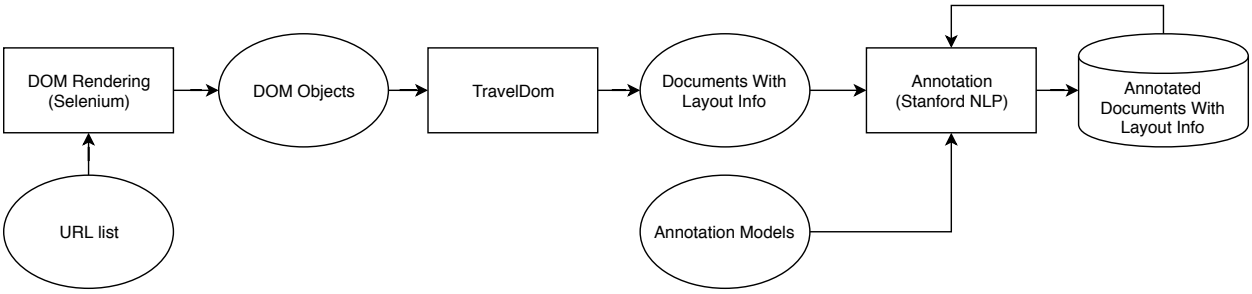

Figure 4.1: Workflow of Extraction. Each rectangle is a system component. Each eclipse is data or information, which is the input or output of a component.

CHAPTER 5: INDEX STRUCTURE

5.1 OBJECTIVES

Input is a list of annotated documents and output is an inverted index. We choose the popular Lucene library for the index engine and Elastic Search as the web server. We customize the process by building an Elastic Search Analysis Plugin.

5.2 INDEX STRUCTURE

We build our index on top of Inverted Index, implemented on Apache Lucene, a free and open-source information retrieval software library. Inverted Index stores pointers from keywords to documents, to speed up keyword search:

$$I(token) : token \rightarrow \{< doc, pos, payload >\} \tag{5.1}$$

In particular, for each token, Lucene stores a list of postings, each of which contains the ID of a document and the position of the token on the document. Developers can also store optional payload information in each posting.

There are two requirements we need, but Lucene does not support. First, besides searching by keywords, we require searching by named entities. Second, besides measuring the distance between tokens by their ordinals, we also want to do so by the actual distance along one of the coordinates.

Our solution is to duplicate fields and to mask Lucene existing concepts with the new semantics. In particular, we create a specialized field for each group of named entities where at most one of them can be labeled. We call the type of fields "entity fields". The benefit is we can have entities and keywords at the same position. For example, token "Illinois" and its corresponding location entity, as well as province entity, can have the same position in our architecture. Formally, the index on 5.1 becomes:

$$\begin{aligned} &I(token) : token \rightarrow \{< doc, pos, payload >\} \\ &Ie(entity) : entity \rightarrow \{< doc, pos, payload >\} \end{aligned} \tag{5.2}$$

In entity fields, only entities have values, which are also their names. For example, sentence "Kevin teaches at the University of Illinois", the field for Stanford NLP's named entity group will analyze it as "PERSON - - - ORGANIZATION ORGANIZATION ORGANIZATION," in which "-" represents stop words which we ignore when indexing. It means that we

duplicate fields before analysis, but the analysis process will not wastefully store the same keywords multiple times across fields.

For coordinates, we further duplicate fields and then replace regular ordinal positional information for each token by the coordinate value. Formally, the index on 5.2 becomes:

$$\begin{aligned}
&I(token) : token \rightarrow \{< doc, pos, payload >\} \\
&Ie(entity) : entity \rightarrow \{< doc, pos, payload >\} \\
&Ix(token) : token \rightarrow \{< doc, x_coordinate, payload >\} \\
&Iex(entity) : entity \rightarrow \{< doc, x_coordinate, payload >\} \\
&Iy(token) : token \rightarrow \{< doc, y_coordinate, payload >\} \\
&Iey(entity) : entity \rightarrow \{< doc, y_coordinate, payload >\}
\end{aligned} \tag{5.3}$$

The storage size is tripled compared to when not indexing coordinates. However, that cost is justified because each token in our index is three-dimensional (ordinal, x-coordinate, y-coordinate) compared to one-dimensional (ordinal) in the traditional inverted index.

The disadvantage of our approach is we cannot query across dimensions. For example, our system cannot look for a token that is both near the word "price" and have the same x-coordinate with an image. It is similar to the limitation when B-Tree index cannot query multi-dimensional data effectively (we must use R-Tree instead).

However, we can alleviate the issue by storing the identity as a payload for each token. If so, to evaluate a query across dimensions, we use the index of a dimension to zoom in a list of candidates, before using the payload to check if a candidate also matches regarding other dimensions. Due to time limitation and as we observe that querying for patterns for each individual dimension is already useful, we will leave support for querying across dimensions to future work.

5.3 ANALYSIS PLUGIN

To implement the approach above, we leverage dynamic schema of Elastic Search and develop an Analysis Plugin, which provides a customized tokenizer called *layout_tokenizer*, that is capable of analyzing field name to map the token to the right masking. The rules are as below:

- If field name follows pattern "_entity_X", it is an entity in group X, and its position is the default one. For example, "_entity_NamedEntityTag" is an entity in a group of 12 default named entities for Stanford NLP, and its position should be its ordinal in the list of all tokens.

- If field name follows pattern "_layout_A_B", it is an entity in group A, and its position is on layout B. For example, "_layout_XPos_RegexNER" is an entity in a group of 12 default REGEX-based named entities, and its position is along X coordinate. We also need a similar "_layout_YPos_RegexNER" for Y coordinate.

With the rules above and *layout_tokenizer* from our analysis plugin, we can create an Elastic Search schema as in Listing 5.3. Please note that we use *layout_tokenizer* to create two custom analyses: *fulltext_analyzer* and *entity_analyzer*

Listing 5.1: Our Elastic Search Schema

```
PUT /index_name/
{
  "mappings": {
    "document": {
      "properties": {
        "text": {
          "type": "text",
          "term_vector": "with_positions_offsets_payloads",
          "store": true,
          "analyzer": "fulltext_analyzer"
        }
      },
      "dynamic_templates": [
        {
          "entity_type": {
            "match_mapping_type": "string",
            "match": "_entity_*",
            "mapping": {
              "type": "text",
              "term_vector": "with_positions_offsets_payloads",
              "store": true,
              "analyzer": "entity_analyzer"
            }
          }
        },
```

Listing 5.1 cont'd

```
        {
          "layout_type": {
            "match_mapping_type": "string",
            "match": "_layout_*",
            "mapping": {
              "type": "text",
              "term_vector": "with_positions_offsets_payloads",
              "store": true,
              "analyzer": "entity_analyzer"
            }
          }
        }
      ]
    }
  },
  "settings": {
    "index": {
      "number_of_shards": 1,
      "number_of_replicas": 0
    },
    "analysis": {
      "analyzer": {
        "entity_analyzer": {
          "type": "custom",
          "tokenizer": "layout_tokenizer"
        },
        "fulltext_analyzer": {
          "type": "custom",
          "tokenizer": "layout_tokenizer",
          "filter": [
            "lowercase"
          ]
        }
      }
    }
  }
}
```

CHAPTER 6: QUERY LANGUAGE

6.1 OBJECTIVES

The objectives are to support two features below:

Feature 6.1 (Topical Query Translation)**.** *Input is user query containing keywords and named entities, and output is Elastic Search DSL query. We build an Elastic Search API Plugin for the translation process.*

Feature 6.2 (Characteristic Query Translation)**.** *Input is entity-based patterns and output is Elastic Search DSL query. We build an Elastic Search API Plugin for the translation process. Topical and characteristic queries are combined in the plugin using a pre-tuned parameter. The parameter could be provided by users to specify the preference between two types of relevance.*

For example, assuming users want to search for pages where a professor's name is near an image, the questions are how to provide users with an intuitive language to describe that need, and how to map the query to what Elastic Search can understand.

6.2 SPAN QUERY

As we model each named entity as a field in a Lucene document (we can the field "entity field"), searching across fields is crucial for us. However, it is unnatural for Lucene-based search engine because they consider each field of a physical document as a subdocument to match with user queries.

We propose a method to use Lucene for our purpose by duplicating the same text across fields and use Masked Span Query to search across fields `https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-span-field-masking-query.html`. The purpose of the duplication is to make sure that positions of entities and those of tokens are comparable (as they are ordinals in the same text). To this end, we do not need to waste storage for any non-entity token in each entity field. Thus, we remove them during the anlysis process, which is specified in the Elastic Search Analysis plugin mentioned in Chapter 5.

For example, we can query documents where a professor is near an image by the Elastic Search query as in Listing 6.1.

```
GET /test_annotation/_search?
{
  "_source": ["title", "url"],
  "query": {
    "function_score": {
      "query": {
        "span_near": {
          "clauses": [
            {
              "field_masking_span": {
                "query": {
                  "span_term": {
                    "_layout_Y_ProfessorTag": "PROFESSOR"
                  }
                },
                "field": "text"
              }
            },
            {
              "field_masking_span": {
                "query": {
                  "span_term": {
                    "_layout_Y_Type": "sigimg"
                  }
                },
                "field": "text"
              }
            }
          ],
          "slop": 1000,
          "in_order": true
        }
      }
    }
  }
}
```

Listing 6.1: A sample Elastic Search query that looks for pages containing a professor name near and above an image of a significant size along the Y coordinate.

6.3 QUERY LANGUAGE TRANSLATION

Elastic Search provides us with DSL query, which is in JSON format. The query above is an example. It is very expressive but too complicated for users. Thus, we design a simple keyword-like query language to help users write more straightforward and more intuitive queries. We define the grammar in an Elastic Search API plugin.

For example, we can rewrite the DSL query in the last section with *@near (#professor #img)* in our language. The query is much more intuitive and compact compared to DSL query. In particular, DSL queries can quickly go unmanageable with slightly more complicated information needs.

6.4 QUERY GRAMMAR

Our query syntax is as below:

- Each query starts with an operator with @ sign, such as *@near* for ordinal distance, *@near_x* for distance along x coordinate and *@near_y* for distance along y coordinate.
- Then it comes with an optional parameters in square bracket such as *[true 10]* means *in_order = true* and *slop = 10* in the DSL query above.
- Then it comes with a list of keywords or named entities in parentheses.
- Named entities are prefixed with # sign. For example, #professor for *@near_y* essentially means *_layout_Y_ProfessorTag = PROFESSOR* in DSL.

A query can be a combination of multiple queries, and we use Elastic Search Bool compound query to implement this `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html`. An essential benefit of this implementation is we can specify "boost" for each subquery. We tried span query but boosting for sub span queries does not work due to ElasticSearch's design choice. This type of compound query is particularly vital when queries are automatically learned. Intuitively, we could understand each subquery as a feature, all of which are linearly combined. This implementation gives us a sound basis for applying machine learning techniques for query inference.

CHAPTER 7: CHARACTERISTICS LEARNING

7.1 OBJECTIVES

As writing queries with both keywords and entities could be a burden for the majority of users, we must support users with an easier way to describe the target page characteristics. As explained before, it is not hard for users to point and click on a few sample pages of the type they want. The challenge is how to infer queries from those sample pages, and why it is possible.

7.2 ENTITY-BASED PATTERNS

Our key insight is with the appearance of named entities, and queries will become patterns, where a named entity is a slot of the type to fill in. It is interesting because the literature has been using page-specific patterns to extract values, which indicates that patterns represent special characteristics of pages.

7.3 ALGORITHM

We propose Algorithm 7.1 as a simple way to learn the patterns based on frequencies. In the current implementation, we only use patterns composed of two components, one is a keyword in the topical query, and the other is a named entity that is not too far away from the keyword. This is a very simplistic implementation, but the templates could be much more sophisticated. Particularly, to instantiate patterns, we use StanfordNLPPatternMatch (which is also called Token Regex) `https://nlp.stanford.edu/software/tokensregex.html` from Stanford NLP library, which inputs a template and a document, and output the instantiation of the template in the document. For example, we may define a template "#entity .*{max 5 tokens} mining" and the library will output all instantiations of the template in the document, such as "James Peter is interested in data mining" (because there are less than 5 tokens between "James Peter" and "mining"). From there, we can learn patterns such as "#person .*{max 5 tokens} mining" from the template. Please note that the pattern is more specific than the template because #person is more specific than #entity.

Algorithm 7.1: Algorithm to automatically infer entity-based patterns from templates and sample pages

Data: A few sample documents $D = \{d\}$ and a few templates $T = \{t\}$
Result: A list of patterns $P = \{p\}$
Let b as a bag of patterns;
Initialize $b = []$;
foreach *template* t *of* T **do**
    foreach *document* d *of* D **do**
        b.appends(`StanfordNLPPatternMatch`(t,d))
Find high-frequency patterns in b and returns

CHAPTER 8: USER INTERFACE

8.1 DESIGN OBJECTIVES

Since our system redefines relevance in traditional web search, we must rethink the user interface. In particular, we shall sketch a new search result page to help users quickly identify pages with target characteristics without clicking on the links.

8.2 DESIGN CHOICES

8.2.1 Simple extension of traditional search engine to reduce learning curve

Our interface is initially similar to traditional web search because users first need to enter a topical query through keywords and named entities. As this step is exactly how users do every day and the next step is optional if they want to narrow down the results to satisfy some characteristics, our system fits in the workflow of traditional web search seamlessly.

We also provide users with an autosuggest component to remind them about the available named entities.

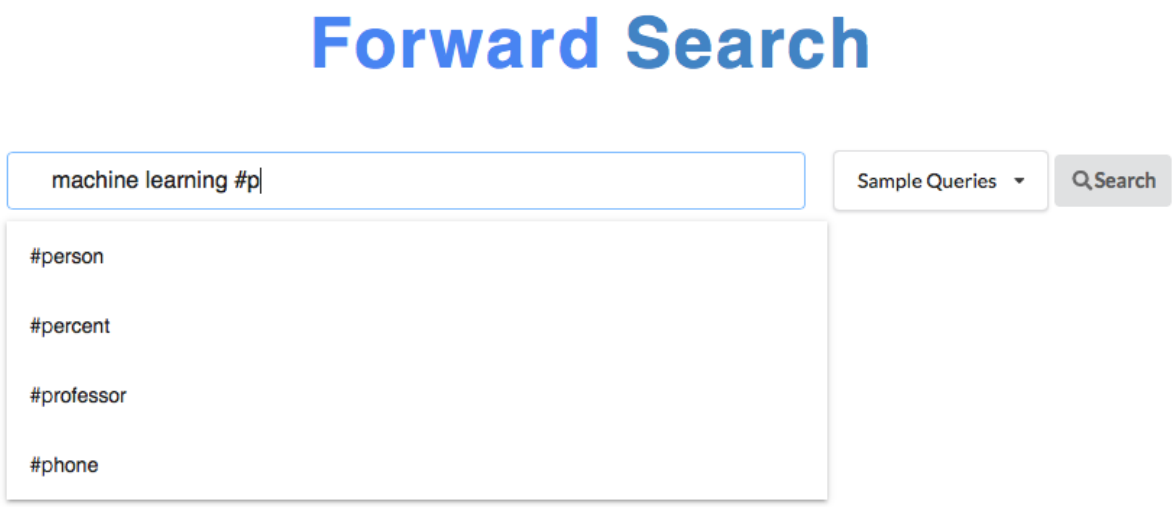

Figure 8.1: Homepage of Forward Search is similar to any other text search engine with a box to enter topical query. We support auto-suggest so that users do not have to remember the names of available named entities

8.2.2 Redesign search results for users to quickly identify types of pages

To highlight the existence of page characteristics, we include a screenshot for each search result. It shows the top viewport of the page, which reveals its characteristics. For example, most users can easily detect a professor homepages by looking at the screenshots. We also show the URL and title in each result because they tell us about the topics of the corresponding page.

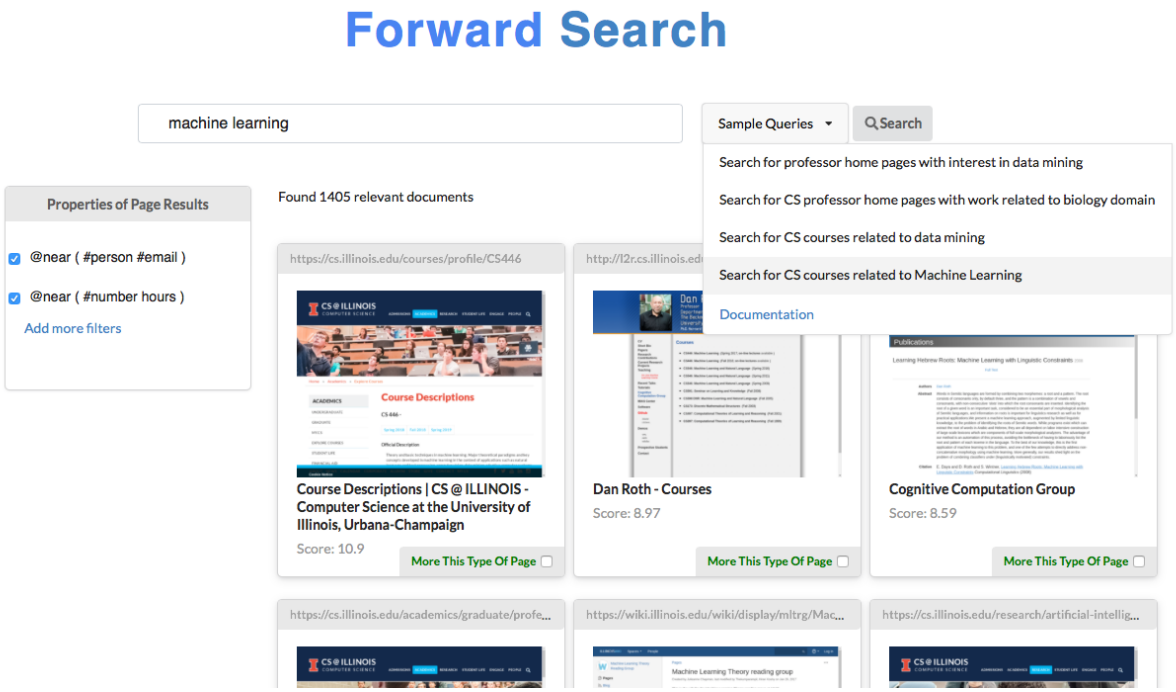

Figure 8.2: Redesigned search result page with the grid layout and page screenshots for users to quickly identify types of pages

8.2.3 Add a novel button *More of This Type* to help users automatically generate queries

As writing direct queries with keywords and named entities could be challenging for most users, we allow users to specify a few pages in the search results as examples of the target characteristics by clicking on "More This Type of Page" buttons. Our system then automatically infer and present subqueries representing characteristics of the pages on the side bar.

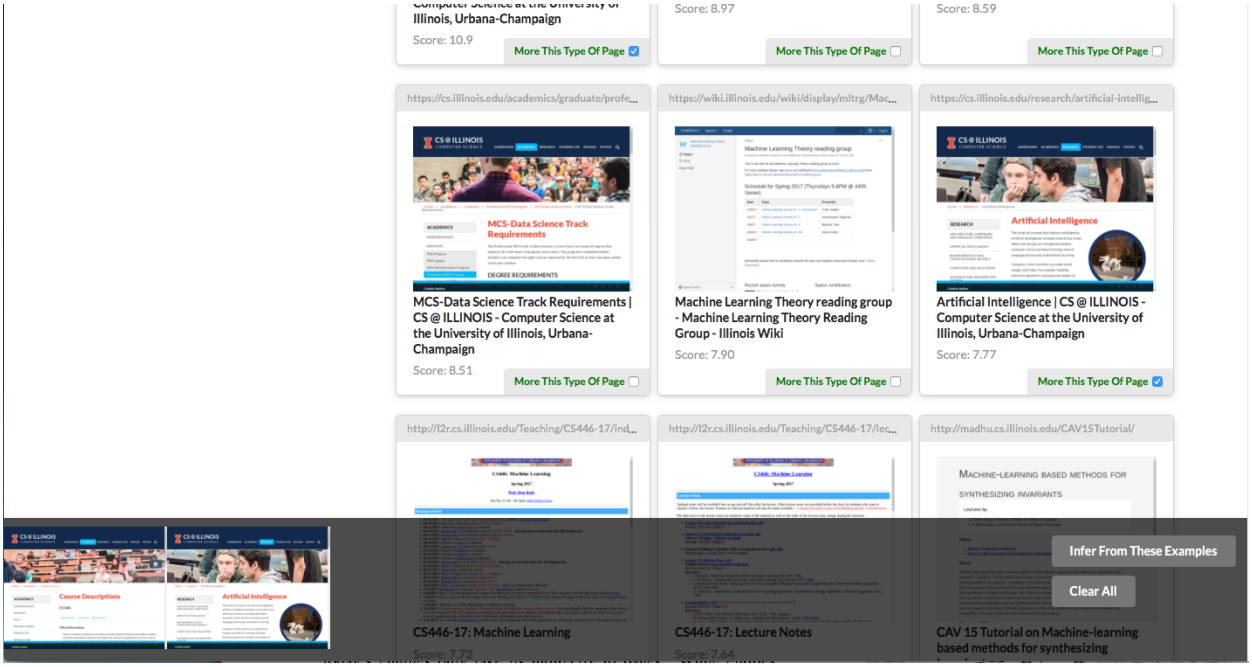

Figure 8.3: Addition of a novel button *More of This Type* to help users automatically generate queries

8.2.4 Addition of a novel side bar to allow users to edit filters similar to vertical search

Users can add properties of target page to the sidebar automatically through inference or manually through typing in an input box. Particularly, users can replace an entity with a value of the type, which resembles the process of filtering results in vertical search. For example, while "@near (#person #organization)" presents expected pattern of a homepage, users can change it to "@near (#person University of Illinois)" to narrow down to only homepages of professors at the University of Illinois.

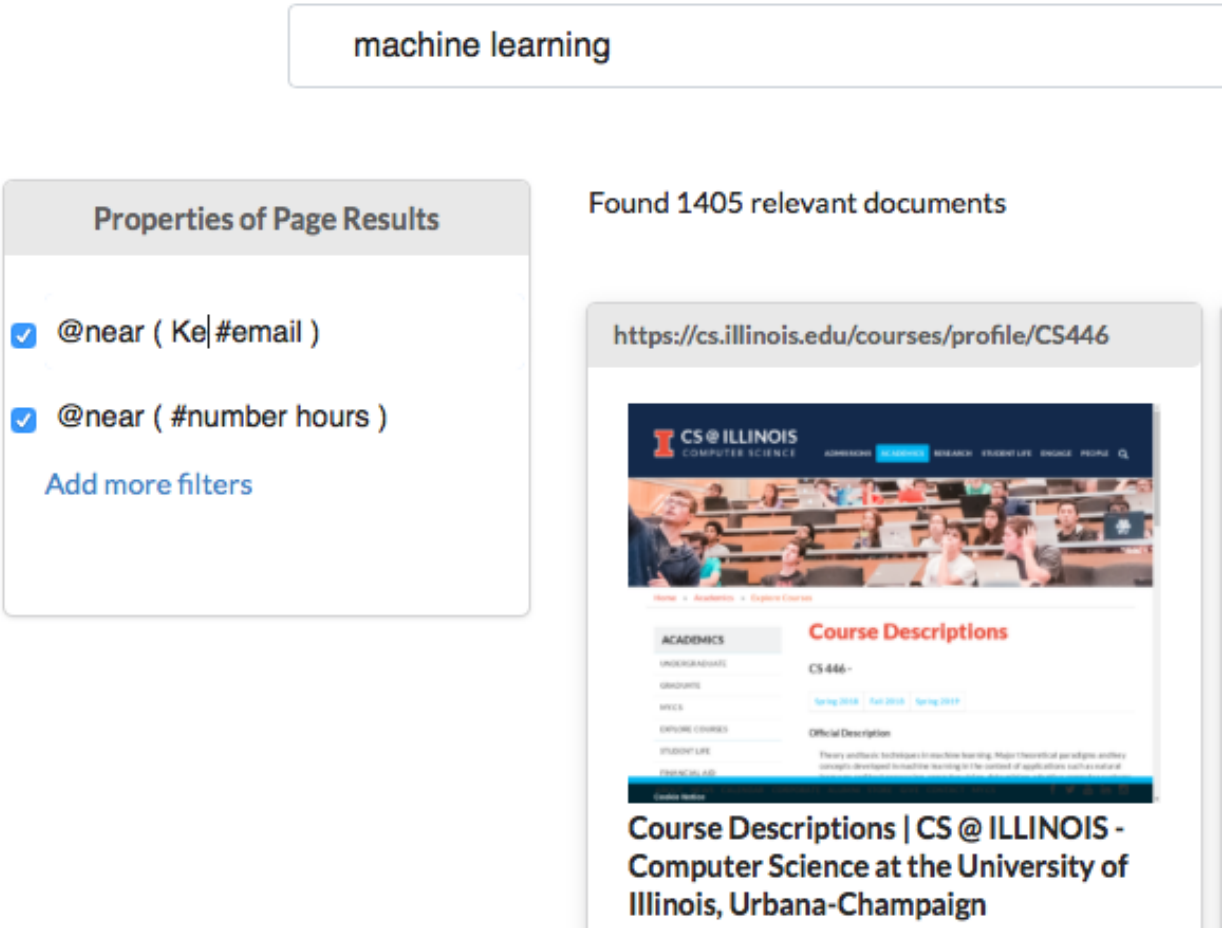

Figure 8.4: Addition of a novel side bar to allow users to edit filters similar to vertical search

CHAPTER 9: EVALUATION

9.1 DATASET

We test the system with a dataset of 10 Computer Science department websites, and each has about 8,000 pages, with a total of 73,228 pages. The dataset represents a common scenario when a developer has a decent set of text data across multiple types/sources of web pages and wants to study the corpus or to build an enterprise search engine on it.

Compared to general web search engines like Google, this scenario does not have metadata such as historical query log, link graph, post date, etc., which makes query-document comparison critically vital for measuring usefulness. Moreover, this type of corpus does not usually have redundancy which makes it difficult to write queries to match the very few documents of interest.

9.2 METHODOLOGY

Our system keeps all of Lucene's capabilities. Thus, with the same index, we will compare how easy and effective our system to Lucene, which is state of the art for enterprise search.

9.3 CASE STUDIES

We will discuss multiple search scenarios users may want to perform on this data set. In particular, we imagine building a search engine for computer science departments, which is potentially useful to gather information for graduate applications and encourage collaboration among departments.

9.3.1 Searching for Professors' Homepages

In Scenario 1.2, Bob wants to search for professor homepages related to his research interest in data mining. The Internet may contain a few curated lists of professors. However, they cannot fulfill his requirements. First, they may be outdated. Second, they, such as faculty directory, contain too many noisy results that are irrelevant with his interest. Third, curated lists are subjective, which likely miss potential professors he wants to work with. For example, a professor in architecture group, who uses data mining to mine system logs, is likely to be excluded in any list of professors in data mining.

Thus, Bob decided to build a search engine across top 10 computer science departments to help him and potentially help his peers. He first tried with Lucene because it is the most popular one for such need.

- He first tries with the obvious query "professor data mining". However, he is so disappointed to see that most top results are news and faculty directories. He took an Information Retrieval course, so he understands it makes sense because those pages contain many instances of keywords "professors" and "data mining."
- Then he tries with rather-hacky query "professor data mining phone" because most professors include phone numbers in his websites. And the first pages are magically filled with professor homepages. He is so happy but quickly realizes the two renown professors in data mining he knows are not in the first few pages. He realizes that their homepages do not contain "phone".

He is so frustrated with Lucene and decides to try Forward Search.

- He starts with topical query "professor mining" to restrict the domain to professors in data mining. The result is similar to the one from Lucene.
- Then he describes a homepage so that the results will give him more homepage results. In particular, he believes most professor's homepage will contain an image aligned with his name either vertically or horizontally, so he writes "@near_x [true 10 0.1] (#img #person)" and "@near_y [true 10 0.1] (#img #person)". Moreover, he believes professors also usually write their names near an organization to describe their work-places, so he adds another subquery "@near [true 10 0.1] (professor #organization)". He is happy to see more professor homepages on top results. Particularly, he found Professor Han and Leskovec on the first page, and many others he does not know before such as Professor Faloutsos and Professor Chang.

Although the search results do not only contain professor homepages, Bob feels happy about them because professors' homepages in the field he cares are among the top results. It is not the case when using Google and Lucene because it has a tiny fraction of homepages in the top search results because they inherently have no way to describe the characteristics of the page he wants.

9.3.2 Searching for Course Homepages

Before deciding his graduate applications, he also wants to learn if the department has multiple courses related to his field of interest.

He starts first with Lucene by typing "data mining course". Unfortunately, the results are filled with the homepages of professors who offer the courses and some department news.

Thus, he seeks for the help from Forward Search. In combination with the topical query above, he also describes a course page by two criteria. First, it should contain a course number, which he knows the format. Second, it should contain credit hour information. Thus, he adds two subqueries: "@near [true 2 0.1] (#number hours)" and "@contains (#course_number)". In particular "#course_number" is an entity not existing by default. Fortunately, Forward Search allows him to quickly annotate the entities and update the index before he runs the queries. The index update is persistent, so next time he can still use "#course_number", which is very convenient.

CHAPTER 10: RELATED WORK

In the literature, there exist approaches to utilize the notion of entity for document search. Apache Hibernate Search [1] models the entire document as an entity with various properties, and allows users to search for entities by keyword-based queries. It supports fuzzy search and ranking by relevance. The benefit is users can search for entities just like searching for documents. Compared to them, in our model, an entity is not a document but a token with a position in the document. Google Enterprise Search [2] models each entity as a field in a document, which enables smart filtering such as filtering all documents mentioning a particular product. Compared to them, an entity in our setting appears in the textual context with other entities and tokens. It enables users to search for documents by describing how entities appear there.

There is also work about searching for entities. Bing Entity Search [3], Google Knowledge Graph [4] and Facebook Graph Search [5] are examples of searching for entities in a semantic graph. The commonness of the approaches is developers need to create a knowledge graph from text data before enabling entity search. However, creating the semantic graph is challenging, which requires those techniques to process only well-structured data such as Wikipedia [6] or Yago [7]. The result is only popular entities are searchable. Cheng et al. [8, 9] proposes an interesting system to search for any entity on the Web including less well-known ones. Compared to their settings, we use entities as the features to search for documents rather than searching for the entities themselves.

About entity definition, similar to [8, 9], our entities are generic information types such as email, phone, person name, etc. rather than entity instances such as a particular person or a particular organization like in [4, 5, 3]. It makes entity extraction much easier and can be applied to all kinds of entities and text documents. Stanford NLP [10] is a popular library for the task.

There are attempts to leverage web page layout information to measure relevance. Fan et al. [11] generates a screenshot photo of each page to measure the authority score of the page as well as the relevance of user queries. The intuition is junk web pages usually follow some layout patterns such as main content is small and advertisements float around. Moreover, the layout information can be useful to calculate importance of some keywords in a document. For example, keywords are more significant if appearing in the most visible parts of the document, such as header, title, and so on. The authors propose methods to extract the layout features automatically and use them to measure authority and relevance of documents during query time. Our approach of using layout information is fundamentally

different from them in the sense that we use layout information to explain the relationship of elements inside a document, such as entities with entities, instead of explaining relationship of the elements with the documents. As a result, we can capture semantic information inside documents while their layout-based technique can only measure the authority of documents and importance of some keywords in the document.

Our intuition of using patterns to represent characteristics of a web page is similar to RoadRunner [12]. In particular, they assume that web pages of the same site should be similar in layout, and the variances are values specific for each page. Thus, by comparing them, they can induce patterns to extract data values from pages of the site. Our approach is interestingly the opposite of theirs. In particular, while their goal is to extract values by patterns, our goal is to extract patterns from labeled values. We are interested in finding patterns because we want to find pages of some types while that is something already given to them.

Our interface is similar to vertical search engines such as LinkedIn [13] and Zillow [14]. However, attributes of pages are predefined in those websites, while in our setting, users can define those properties by writing subqueries or providing sample pages. Our system can be considered as an on-the-fly vertical search engine when vertical is part of the query. The benefit are two folds. First, users can specify the verticals they want rather than waiting for someone to build a vertical for them, which is limited and often not exactly what they want. Second, any pre-built verticals must extract data before users can query which potentially eliminates useful information, while we only rank results and users can scroll or revise queries to find target information. The challenge is how to provide users with an easy way to specify their desirable verticals, and we propose a method to infer them from sample pages in Chapter 7.

CHAPTER 11: FUTURE WORK

The goal of this thesis is to set a foundation for more research and development of Forward Search. There are many potential improvements.

First, we need more interesting layout features to demonstrate the concept of describing page characteristics. For example, we may label portrait photos or the ones with human faces because knowing their position and size is very useful to detect if a page is a homepage or not. Another group of potential features is HTML layout elements. For example, tables are helpful to detect if a web page contains scientific data or not. Those features are easy to extract with current technologies. Here we provide developers with a search engine to leverage available semantic features for document retrieval.

Second, we need to support searching for content of entities. For example, an email entity becomes much more useful for academic homepage search if it ends at dot-edu. A menu containing publication, research, or teaching is an indication of a professor's page. Content is however tricky to index because we cannot index all possible values. However, we assume that the index can give us few results enough for efficient checking. Please note that we need to do the checking when joining lists of matched documents in Lucene Core to avoid re-searching for matches and retrieving full documents. We propose to store content or value of each entity in its payload `https://wiki.apache.org/lucene-java/Payloads`.

Third, we need to support searching across multiple dimensions. For example, we want to compare both coordinates of entities to know if they are actually near each other. Now, each subquery must be on a single dimension, i.e., x-coordinate, y-coordinate or ordinal, because we can compare positions of different types when joining lists of documents in Lucene core. It is inherently a problem of Lucene because Lucene finds matches by joining sorted lists of positions, which requires that those positions must be on the same dimension. To solve it, we must add additional checking after Lucene joins the lists. As we use Span Query for each subquery, we need to do the checking when Span Query components evaluate candidate spans. Finally, to enable the checking, we must put all positional information on the payload of each token.

Forth, we need a more sophisticated page characteristic learning. The current method instantiates patterns from templates and weights them by frequencies, which seems to be too simplistic. Setting the weight for each learned patterns is difficult because high-frequency and noisy patterns can significantly affect the results. We may also change scoring function of Elastic Search to have better ranking because the current one seems to be not suitable with long queries.

Fifth, we need to have an easier way to set up the entire workflow. Now our code is spread across five repositories. It is helpful to have a single script to install and run Forward Search. That script should support ways to customize the process, such as instead of using Common Crawl to search for URLs, developers can directly provide us with a list of URLs.

Finally, it is possible that there is a simpler structure to incorporate named entities in the index. As discussed in Chapter 5, the reason why we use Masked Span Query is to allow entities and keywords to be able to have the same positions. It is possible that the newer version of Lucene may support this natively. Thus, future developers of the project may want to keep this in mind.

CHAPTER 12: CONCLUSION

We discussed Forward Search as a powerful tool for users to express their information needs to navigate in the increasingly heterogeneous Web. Regarding concept, we propose characteristic relevance as a new measure of the usefulness of search results and propose designs and algorithms to adapt current search engines with the new requirements. Regarding development, we create a live demo at `http://crow.cs.illinois.edu:8080` and publish open-source code on Github `http://github.com/forward-uiuc`. We evaluate the effectiveness of Forward Search by a case study of building academic search engines for top 10 Computer Science departments with a real corpus of 75,000 documents. The preliminary result is promising, which indicates scenarios when our engine is better than the original Lucene library. We also discussed related work and proposed several directions for future work.

REFERENCES

[1] "Apache hibernate search software," 2018. [Online]. Available: http://hibernate.org/

[2] "Google enterprise search," 2018. [Online]. Available: https://enterprise.google.com/search/

[3] "Bing entity search," 2018. [Online]. Available: https://azure.microsoft.com/en-us/services/cognitive-services/bing-entity-search-api/

[4] "Google graph search," 2018. [Online]. Available: https://developers.google.com/knowledge-graph/

[5] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang, "Unicorn: A system for searching the social graph," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1150–1161, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536239

[6] "Wikipedia," 2018. [Online]. Available: https://www.wikipedia.org

[7] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A core of semantic knowledge," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242667 pp. 697–706.

[8] T. Cheng, X. Yan, and K. C.-C. Chang, "Supporting entity search: A large-scale prototype search engine," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247636 pp. 1144–1146.

[9] T. Cheng, X. Yan, and K. C.-C. Chang, "Entityrank: Searching entities directly and holistically," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325898 pp. 387–398.

[10] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ser. ACL '05. Stroudsburg, PA, USA: Association for Computational Linguistics, 2005. [Online]. Available: https://doi.org/10.3115/1219840.1219885 pp. 363–370.

[11] Y. Fan, J. Guo, Y. Lan, J. Xu, L. Pang, and X. Cheng, "Learning visual features from snapshots for web search," *CoRR*, vol. abs/1710.06997, 2017. [Online]. Available: http://arxiv.org/abs/1710.06997

[12] V. Crescenzi, G. Mecca, and P. Merialdo, "Roadrunner: Towards automatic data extraction from large web sites," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=645927.672370 pp. 109–118.

[13] "Linkedin," 2018. [Online]. Available: https://www.linkedin.com

[14] "Zillow," 2018. [Online]. Available: https://www.zillow.com