POLITECNICO DI TORINO

Collegio di Ingegneria Matematica

Corso di Laurea Magistrale
in Ingegneria Matematica

Tesi di Laurea Magistrale

Fuzzy transcript annotation with domain-specific contextual learning

Relatore

firma del relatore (dei relatori)
prof. Laura Farinetti
.....
.....
.....
.....

Candidato

firma del candidato
XU LIANG

Marzo, 2020

Index

Chapter 1

Introduction

In recent years, as the on-line study becomes more and more popular among students at university and people who continue their self-study after work. Both the quantity and the quality of on-line open courses have been continuously increasing.

In Politecnico di Torino, more and more on-line courses have been provided to the students for the purpose of after-class study and the preparation for the exams. As we know, as long as we have a good understanding of the key concepts and topics of a specific course, it will benefit us a lot to obtain a better result during the final exam. As a result, the text mining group of Politecnico di Torino launched a project which implements a search function within each individual video lecture, in order to make it possible for the students can retrieve all the time slots which are related to the word they search. For the implementation of that purpose, a possible way is to give semantic annotation to the transcripts of video lectures, which means to highlight the important words with interpretive notations.

For the purpose addressed above, techniques of related fields such as Natural language processing, semantic web and information retrieval should come into consideration. Especially two of the NLP techniques may become the key to solve the problem : named entity recognition(NER) and named entity linking(NEL). Since the procedure of giving semantic annotation to each word or phrase in the transcript can be seen as the procedure of linking between the raw text word and named entities in knowledge base such as Wikidata or DBpedia. Then there are mainly two different needs of the project. The first one is to recognize as many as possible named entities according to the mentions that appear in the transcript text. And the second one is to link the recognized named entities to knowledge base.

There are many different aspects of work we can do to help the progress of the whole project . And among the different directions, there are mainly two contributions of the work in this thesis to our whole project. The first is to find a feasible way to tackling the problem of misspellings in the transcript, since nowadays the transcripts automatically obtained through means like ‘Google transcribe’ usually still contain plenty of incorrect spellings, which will make the tasks NER and NEL much more difficult. As a result there’s the need of finding a way to recognizing the possible correct spelling of the original word in order to make convenience for the NER or NEL tasks. Here we finally reach to the point

that we utilize tri-gram analyzer in ElasticSearch which is one of the current most popular search engine to implement fuzzy search in DBpedia labels. As a result, we can get a ranked list of named entity labels when searching a raw text word even in the case of mis-spelling.

The second contribution of this thesis is to implement a prototype framework with several different information retrieval methods to obtain a collection of domain-specific named entities. Since to give semantic annotation to video lectures of a specific course, it usually means that most of the named entities which appear in that series of lectures should belong to the same or highly related scientific subjects or fields. According to the principle above, it can be really helpful and meaningful to get a collection of domain-specific named entities and to continuously populate the obtained collection in order to cover as many named entities belong to the specific domain as possible. In this thesis, we found a way of utilizing multiple information retrieval tools and combining them in a heuristic way. And the method we proposed mainly used three different kinds of tools : web scrawler , Google custom search engine and MediaWiki API. In addition, after obtaining the domain-specific text, we use several open named entity extractors to extract named entities, such as DBpedia spotlight, Babelfy, etc.

The article is organized as following : Chapter 2 introduce the reason and the context of the project more in details. Chapter 3 introduce the literature of the work of this thesis and related research areas. Chapter 4 introduce the basic knowledge and part of starting of the tools and methods have been used. Chapter 5 is the part of the methods we proposed and the relevant experiments we have done. And Chapter 6 draws the conclusions of the thesis and depict the further insights for future work.

Chapter 2

Context

The Politecnico di Torino provides many services which can help students about their studies during their university years. One of the most helpful could be the online video lectures service, which has been offered more and more in recent years. As the result, students are not only facilitated to follow the courses comfortably at home in case of illness or some other reasons outside the campus, but also during the study phase, when some topics are not clearly understood at classroom.

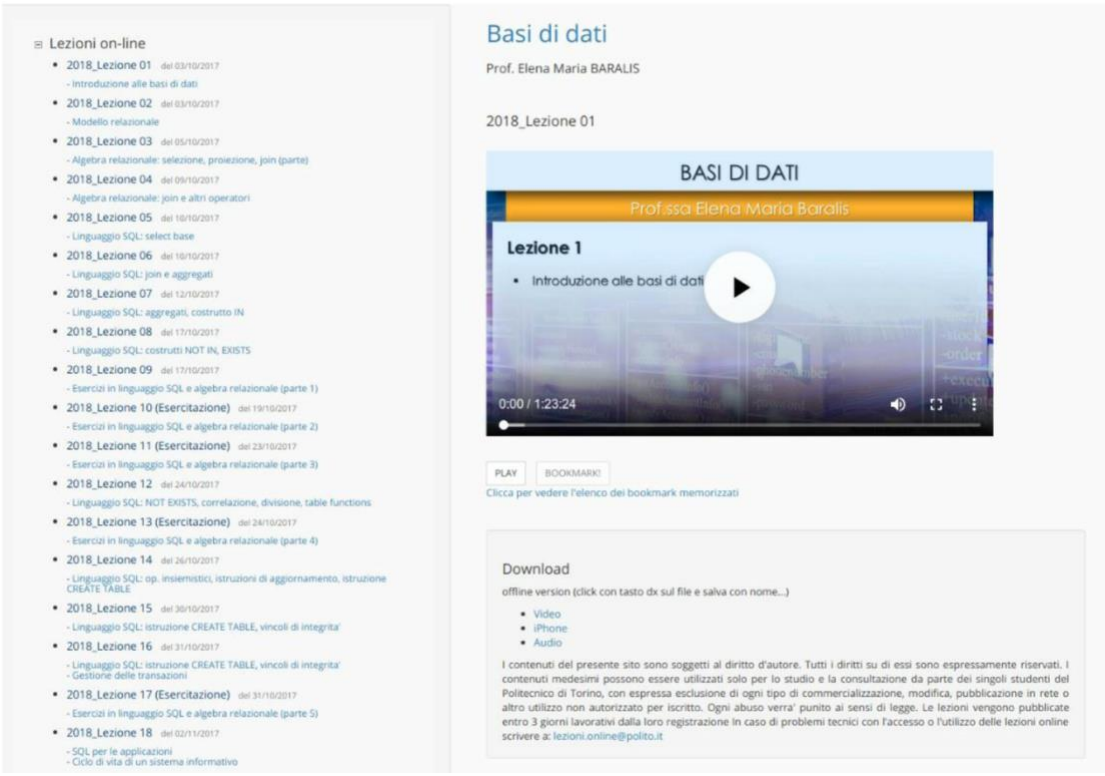

Figure 2.1: the page of the database video lecture site, in which shows the framework that how the video lectures are structured.

On this page, students are able to choose the lecture that they prefer and review the parts of greatest interest for study purposes. In addition, some other functions are also provided, including the brief description of a specific lecture, in order to help students to access to the a specific page, or to conveniently download the video lectures locally in order to watch them offline. Among the different functions, there is a function that could be regarded as really essential: a search bar which allows you searching for some specific content within the course.

With this function, a student could search for a specific interested topic by writing some keywords, which will get a list of links as the result, corresponding to the precise moments in which the professor talked about the topic searched. Then in general, during the final study phase, the preparation for the exam, the students are usually interested in clarifying their understandings on some specific topics and concepts, and often there is no enough time to go through and look for all the paragraphs in which these have actually been depicted, in order to not miss the most essential parts.

The function that we want to implement in some way fulfills this kind of need. And an basic idea is to divide each single lecture into time slots, characterized or labeled by a specific topic, so that, when a certain topic is being searched for, the student could see all the time slots related to the concepts searched. Dividing a lecture into fixed time slots, where each one corresponds to a specific topic, is not appropriate enough, since in general during the procedure of a specific teaching explanation, the professors usually tend to make references to many different topics at the same time, which can be all useful to understand the main topic or the whole field in some degrees.

As the result, the actual goal is to find a way to recognize and extract the key concepts within the video lecture. And the approach is that starting from the transcription of the professor's speech (transcript) during the lecture and having every matching between word and instant of time in which it is pronounced, then analyzing this text in search of key concepts. In this kind of context, these key concepts have a domain specific name in natural language processing : **entities.**

In a word, the main target of our project is to give semantic annotation to the raw text of transcripts using techniques about named entities.

Chapter 3

Literature of related fields

Information extraction (IE) from a text is linked to the problem of simplifying the unstructured or semi-structured content in order to create a structured view of the information present in a free text. In most of the cases this activity concerns processing human language texts by means of **natural language processing** (NLP). IE techniques can be applied to many different fields, such as newspaper, comments in the social networks, digital books on the web, scientific papers, and job advertisements, etc. And what kind of information should IE techniques extract from all these kind of information resource ? There is a core conception in the IE or NLP context which is called named entity that plays a key role. Usually the term named entity identifies a unit information such as the name of something with useful meaning, including people, organizations, locations, etc. With these named entities in a text content, the semantic meaning of the whole sentence or text then can be expressed in a relatively understandable way. The research about named entity has been a key role in the area of NLP for a long time, also in the area of semantic web the case is the same.

And what is semantic web? What is the difference between original World Wide Web and semantic web? In fact, semantic web as a concept was originally expressed by Tim Berners-Lee in 1998, who was also the inventor of World Wide Web in 1990. As described in **reference[1],** the core idea of semantic web is : by adding semantics (metadata) that can be understood by computers to documents on the World Wide Web (such as HTML documents), the entire Internet becomes a universal information exchange medium. And to enable the encoding of semantics with the data, technologies such as Resource Description Framework (RDF) **reference[2]** and Web Ontology Language (OWL) **reference [3]** are used. As a result, named entity extraction from a text and adding semantic content on text, with the support of different ontologies and knowledge bases (KB), such as Wikipedia, DBpedia, and Freebase, has always attracted more and more attention from researchers.

Another area called natural language processing intersects the area of semantic web a lot, which always can be a kind of technique support to help the machine understand and recognize import information of human-being language. But usually the NLP problem can be really complicated to cope with, since there's always big intrinsic ambiguity of human language and characters. As a result, the NLP area usually can be divided into different sub areas or phases, which cooperate organically together to understand natural language.

And usually the NLP area can be divided into these phases :

- lexical analysis: decomposition of a token linguistic expression (in this case the words).
- grammatical analysis: association of the parts of speech to each word in the text.
- syntactic analysis: arrangement of tokens in a syntactic structure
- semantic analysis: is the process of relating syntactic structures, from the levels of phrases, clauses, sentences and paragraphs to the level of the writing as a whole, to their language-independent meanings.

In our project, the NLP techniques that we use are focused on the sub area of lexical analysis. The main purpose of lexical analysis is to cut out words from sentences, find out the individual morphemes of words, and determine their meanings. Lexical analysis includes word segmentation, part-of-speech tagging, named entity recognition, and word sense disambiguation. Lexical analysis includes two aspects: morphology and vocabulary. Generally speaking, morphology is mainly reflected in the analysis of word prefixes and suffixes, while vocabulary is expressed in the control of the entire lexical system. Different languages have different requirements for lexical analysis, for example, there is a large gap between English and Chinese or between Italian and Chinese.

3.1 Named entity recognition (NER)

The reason we introduce NER here is that the task of NER is strongly related to the work we do in our project, in the meanwhile also quite different. As depicted above, NER is one of the main task of lexical analysis. And it has been one of the most popular NLP tasks among these years. Plenty of methods and papers were given about NER, both from statistical learning and deep learning. And what is NER?
Named Entity Recognition (NER), also known as "proper name recognition", refers to identifying entities with a specific meaning in the text, mainly including person names, place names, agency names, and proper nouns. Simply speaking, it is to identify the boundaries and categories of entity references in natural text. As introduced in **reference[4]**, from 1991 to 2006, there was a change of the research hot point from rules and dictionaries to machine learning methods, such as HMM, CRF. Even though with machine learning methods, there's less human work to construct dictionaries or design rules, but there are still plenty of work of feature Engineering. In recent years, there are a lot of works with deep learning methods, for example the bi-LSTM + CNN architecture designed in **reference[5]**, also eliminate the cost of human work for feature engineering.

Usually NER task can be modeled as a classification task for each token in the sequence. And the biggest problem why we cannot directly map the problem in our project to a traditional NER task is that the set of categories of the classification of NER is usually very limited, otherwise there's the need to construct a domain-specific method for each different domain. Usually there are 3 big categories and several small categories for traditional NER task :

- **ENAMEX** : limited to names, acronyms, and other unique identifiers, which are characterized with the TYPE attribute via **PERSON** (personal, family names), **ORGANIZATION** (names of corporations, governments and other organizational entities) and **LOCATION** (names of political or geographical locations, such as regions, cities, continents, mountains, lakes, etc. and astronomical locations).
- **TIMEX** : refers to temporal expressions, which can be characterized by the TYPE attribute, in **DATE** (complete or partial date), **TIME** (complete or partial time of day), **DURATION** (measure of time spent during a known period).
- **NUMEX** : and useful for numeric expressions, including values with units of measure. Through the TYPE attribute it is characterized in **MONEY** (monetary expression), **MEASURE** (standard for measurements, such as age, area, distances, energy, speed, temperature, volume, pressure, etc.), PERCENT (percentages), **CARDINAL** (number or numerical quantity of some object).

Although there already have been many relatively matured and powerful methods and algorithms of NER, but in our case when we want to extract and give some semantic annotations to the transcript of different courses from different subjects, other than if we build NER system for each of the subjects on purpose, we cannot provide a generalized useful NER system for most of the different domains at the same time. But there is still some help point that NER methods and tools can support our project.

3.2 Named entity disambiguation (NED)

Named entity disambiguation or linking is a NLP task to determine the identity of the entity mentioned in a context. This is done by linking the mention to an entity in a knowledge base. And one of the most difficult problem of NED is that a single word or mention may have several different meanings. For example, 'I bought a new iphone in the apple store', in this sentence, apple means the American technology company instead of the fruit. It's quite easy for human being to recognize the actual meaning of the mention. But for machine, this work mainly rely on the capable NED system.

3.2.1 NED approaches glimpse

Many approaches have been proposed to perform NED. All of them can be grouped into three macro-strategies: local, global and collective. The local methods act on each word mentioned, independently from the others, based on the compatibility between the subject and his candidate entities, using some general features to improve the selection. Global and collective methods assume that disambiguation decisions are interactive and that there is consistency between entities that occur in the same text, allowing the use of semantic relationship measures for disambiguation. Collective methods make disambiguation decisions simultaneously, while global methods disambiguate only one mention(entity) at a time.

But whatever these types of approaches of NED, usually there are two phases in a NED system. The first one is candidate generation, which means through some strategy like lookup some entity dictionary derived from a knowledge base or Wikipedia structured links. The second one is to select the best candidate from the set of candidate entities derived from the first step, which usually is more focused by the research interests of the NLP researchers.

Usually a typical local approach is to evaluate the similarity between a mention and the candidate entities, with the context. First, the contextual characteristics of the entities within the text are extracted. Then, they are weighted and represented as numerical features in a model. Finally, every mention in the text is linked to the candidate entity that has the highest similarity with it.

Bunescu and Pasca **reference[6]** described the first system to resolve entity mentions to Wikipedia pages. They proposed a method that uses an SVM kernel to compare the context around the mention, in combination with the estimate of the correlation of the word with the categories of the candidate entities. Each candidate entity is an entry in Wikipedia and its lexical context is the content of the article. Later in TAC 2009, Honnibal and Dale **reference[7]** designed two different kinds of strategies in the phase of candidate generation, the first one is designed to get a minimal set of candidates to get the minimum degree of ambiguity by using a series of Wikipedia dictionary lookups. The other one is the high coverage strategy involves simply looking up the mention string in a set of reverse index of Wikipedia markup which consists of the name, redirection, truncated name and disambiguation dictionaries. And in the phase of ranking the candidates, they experimented two different measures also ,finally they found cosine similarity was better than overlap between the strings. And in this work, they also first introduced BOW(bag of words) in NED to represent both the Wikipedia article and the document of the mention. In TAC-KBP 2011, Cassidy

and Chen **reference[8]** introduced the popularity of the candidate entities, which made the final decision of NED biased on the more popular candidates. Global approaches leverage interdependence between disambiguation decisions of different entities and exploit two main types of information: the context of disambiguation and the semantic relationship between entities. In 2011, Cucerzan **reference[8]** was the first to use an interdependence model between disambiguation decisions of different entities. **In reference[8]** the context of disambiguation is composed of all the contexts of Wikipedia that occur in the text and the semantic relationship is based on the overlap in categories of entities that could be referred to the mention in the text. Wikipedia contexts include inlink labels, outlink labels and the titles of all articles. In 2009, Kulkarni **reference[9]** premised that documents largely refer to topically coherent entities, which means within a single document there may appear many different entities at the same time, but usually the different entities should be in the same or related domain or topic. For example, While Michael Jordan and Stuart Russell can refer to seven (basketball player, footballer, actor, machine learning researcher, etc.) and three (politician, AI researcher, DJ) persons respectively in Wikipedia (as of early 2009), a page where both Michael Jordan and Stuart Russell are mentioned is almost certainly about computer science, disambiguating them completely.

In 2011, Han and Sun **reference[10]** proposed a graph-based collective NED method, which modeled and exploited the global interdependence between NED decisions of different entities that co-occurred in the same document. And they first proposed a graph-based representation, called Referent Graph. In addition, they proposed a collective inference algorithm, which was capable to jointly infer the referent entities of all name mentions by exploiting the interdependence captured in Referent Graph. A little bit more in details, the collective algorithm proposed collects the initial evidence for each mention and then strengthens this evidence by the propagation of the initial evidence through the arcs of the Referent Graph. The initial evidence of each mention is defined as its popularity on the other mentions and its value is the TF-IDF11 score normalized by the sum of the TF-IDF scores of all the mentions in the text.

3.2.2 Brief introduction on knowledge base

In general, the knowledge bases used for implementing NED are divided into two different categories. The first one is curated knowledge base represented by Freebase, Yago2. They extract a large number of entities and entity relationships from knowledge bases such as Wikipedia and WordNet. They can be understood as a structured Wikipedia. In fact, a knowledge base like Wikipedia is still only a small part of the entire Internet information. Another type of knowledge base is extracted knowledge base represented by Open IE and NELL. They extract entity-relationship triples directly from hundreds of millions of web pages. Compared with curated knowledge base, the knowledge obtained in this way is more diverse. But in the meanwhile there is also a certain noise that is directly extracted from the webpage, and its accuracy is lower than that of Curated knowledge base.

Here, we only list three of the most popular knowledge bases :

- **Wikipedia** : is the most popular and large scale multilingual encyclopedia. And is created and evolved by the distributed and collective effort from volunteers all over the world. Each Wikipedia article refers to a distinguished entity, thus the information contained in Wikipedia article can be really useful for NLP tasks about named entities, such as NER, NED. Furthermore, the Wikipedia structure provides a set of useful functions for linking entities using labels, categories, redirect pages, disambiguation pages and links to other Wikipedia pages.
- **DBpedia** : as described in **reference[11]**, **[12],** DBpedia is a multilingual knowledge base created from Wikipedia. By leveraging the structured information in each article of Wikipedia, such as infobox, category hierarchy, geographic coordinates and external links**.**
- **Wikidata :** as described in **reference[13**], Wikidata is Wikimedia's central repository for structured data. This is the place where data, like the number of inhabitants of a country, is stored and made accessible to humans and computers alike. The data is used across all 287 language editions of Wikipedia and its sister projects as well as in projects outside of Wikimedia.

In **reference[14]**, there is a very good survey and comparison between different open knowledge bases, in which much more informative details can be found.

3.3 Practical need of transcript annotation : Fuzzy match

In the literature of our whole project, the last student did the research regarding the problem as a pure ideal NED problem, and in his work he used a large scale clustering method which has a really high computational complexity. Although he has reached some degree of acceptable values of f1 score, precision and recall. But there's still long way to go to improve the whole project in several aspects. The first one is that there should be the capability of fuzzy match on the tokens of the video lecture transcript. It's not possible to get a large number of video transcripts by human labor, since the translation of even only one video audio to text will cost more or less several times of the time duration of the video lecture. Then nowadays, the transcripts of videos are usually obtained by some automatic methods, for example, Google offers a service online called '**Transcribing video**' which can automatically transfer audio in the video to text. But until now the result text obtained from that kind of service still sometimes contains a lot of spelling errors. And since nearly all of the current NED methods leverage the string characteristics of the token itself and also the context of the token which usually the other tokens around it. Then it will introduce a lot of noise as a result of the spelling errors. As a result, there should be some way to automatically revise the spelling errors for traditional NED methods or some other ideas to recognize and disambiguate named entities from the raw text of video transcripts.

That's the reason why we think about leverage the convenient and fast indexing and searching functions of Elastic Search. Especially it can been seen naturally in the phase of candidate generation to use some kind of fuzzy search of ES. In addition, the ranking of the search result in ES is also in some way controllable, which means one can adjust the ranking of search results according to some heuristic criterion of one self by leveraging some characteristics of ES.

3.4 Domain-specific set of contextual keywords learning

In our project, usually at each time there is a series of video lectures which belong to the same course, which means that the helpful named entities in these video lectures of the same course are in the same domain. For example, in the course of machine learning, in one of the lectures, the topic is support vector machine, and in another lecture the topic is Naïve Bayes, then the named entities appear in these two lectures have a high probability to be from the same field, machine learning. Moreover, the specific domain's name usually just directly appears in the title of the course. And in our project, the domains are not absolutely open field, since the video lectures are certainly about a specific subject of engineering, science or social science, etc.

Before starting this part of work, a natural idea was to check whether there were some domain specific keywords set that already existed. But unfortunately, the answer is negative in most of time. Until now there still aren't enough keywords set for most of domains, if there's the need, one should collect the keywords by oneself. But to implement an automatic system to give semantic annotation to video transcripts of many different subjects, this can be a really meaningful and useful task to automatically collect important domain specific keywords as a set, which can be helpful to the following down streaming NLP tasks, such as NER,NED. And now some of the heuristic ideas and methods can play an important role for this target.

Chapter 4

The IR methods and tools used

In this chapter, a number of different tools and methods have been used later will be introduced briefly, such as google custom search engine, MediaWiki api, self-written scrawler to parse books and elastic search.

Usually the different tools have been used for absolutely different purposes separately:

- **Google custom search engine** is a platform provided by Google that allows web developers to feature specialized information in web services, refine and categorize queries and create customized search engines, based on Google search.
- **MediaWiki** is a set of web-based Wiki engines, which are used by all Wikimedia Foundation projects and many wiki sites. MediaWiki software was originally developed for the Free Content Encyclopedia Wikipedia and is today deployed by some companies as an internal knowledge management and content management system.
- A **Web crawler**, sometimes called a spider or spiderbot and often shortened to crawler, is an internet bot that systematically browses the worldwide web typically for the purpose of web indexing (web spidering).
- **Elasticsearch** is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.

But sometimes they also could be utilized together to fulfill some needs of Natural Language Processing or Semantic Web Search functionalities. Since in some way, they all can retrieve or cope with the text information on the web. Among the above tools ,mediawiki is more domain specific since it can be only used for the need of Wikipedia related data, especially for the works which are related to named-entities. But in the meanwhile, the other three are more open domain tools, that means there is a much bigger flexibility of the way one can use them and there are much smaller constraints when using them. For example, one can use google custom search engine to get search result from a limited selected number of web sited not the whole worldwide web, there's nearly absolute flexibility that one can choose whatever website from which he or she retrieve the search result. Another example, one can usually write his or her own web scrawler to get the specific web information they want, in different forms and from different web resources.

4.1 Google custom search engine

As depicted above, google custom search engine is a platform based on google search but which allows the user to define the set of the web sites where they want to get the results from. Since in our case there's the need of the use of name entities of a specific course, here we can define the set of search web sites as the web sites which are most probably related to the specific course and contain plenty of the labels of the related named entities. And in our work , as a basement, we only use a unique web site of **Coursera** which is a famous open course web site that contains a number of courses in different fields such as mathematics, physics and computer science, etc.

Here a brief description about how can one use google custom search engine to only search on the web site Coursera:

- Step 1 : From the Google Custom Search homepage, click Create a custom search engine or New search engine.

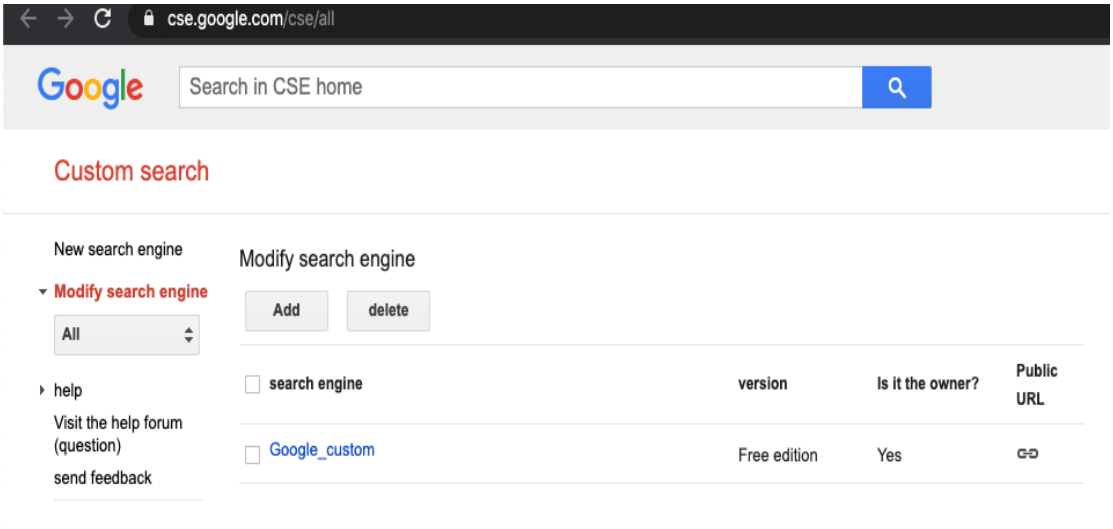

Figure 4.1: first step of creating a google custom search engine.

- Step 2 : In the Sites to search box, type one or more sites you want to include in the search results. You can include any sites on the web, even sites you don't own. Don't worry, you can always add more later.

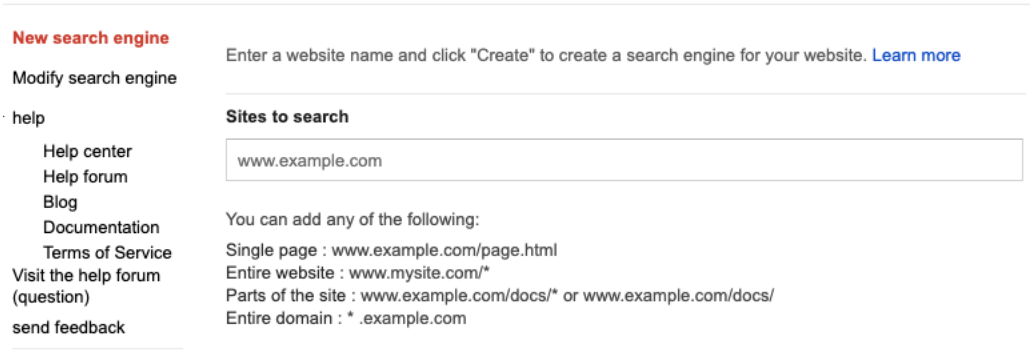

Figure 4.2 : second step of creating a google custom search engine.

- Step 3 : In the Name of the search engine field, enter a name to identify your search engine. For tips on naming your search engine, see Branding guidelines below.

Figure 4.3 : third step of creating a google custom search engine.

- Step 4 : Once you're ready, click Create. Until now the custom search engine itself has already been created.
- Step 5 : Then we go to https://console.developers.google.com/project and create a new project and enable an API, here we choose Google Custom Search API.

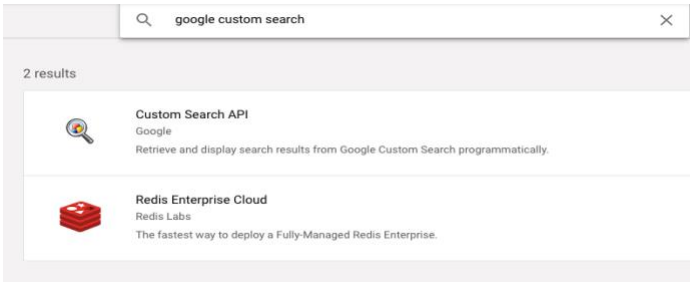

Figure 4.4 : 5th step of creating a google custom search engine.

- Step 6 : Then in the Credentials option we can create an API key for public service access of our created Google Custom Search API.

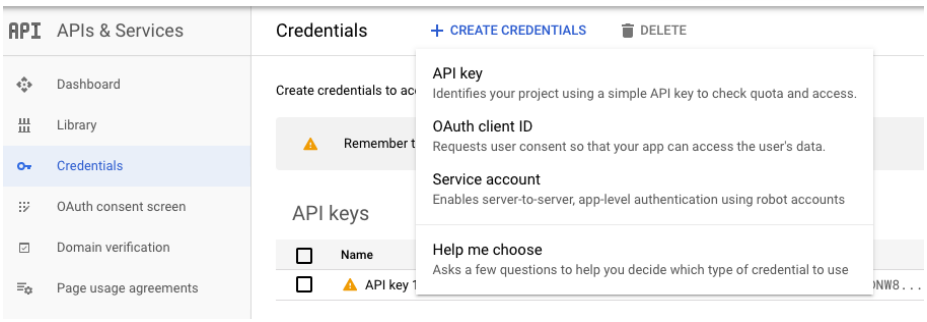

Figure 4.5 : 6th step of creating a google custom search engine.

Until here, we have already created a Google Custom Search Engine API which can be invoked in our own service program in Python, Javascript, etc.

After installing Google Custom Search API module in the python program or the conda environment by command 'pip install google-api-python-client', one can use the API like the picture of code showed below:

```
from googleapiclient.discovery import build
import pprint

#define key
api_key = "Your API Key"
cse_key = "Your Custom search Engine Key"

resource = build("customsearch", 'v1', developerKey=api_key).cse()
result = resource.list(q='simplified python', cx=cse_key).execute()

pprint.pprint(result)
```

Figure 4.6 : part of python script of using google custom search API

The two main parameters here are the API key and the Custom Search Engine key. And here q is the query that one want to search on the custom search engine. To see more details about the parameters to help a more customized use of the API, one can search the detail information on the Google's developers site: https://developers.google.com/custom-search/v1/overview

4.2 MediaWiki

MediaWiki is a **wiki software,** but first we need to know what is actually **wiki**? Wiki is a hypertext system that is open on the World Wide Web and can be co-authored by multiple people. It was first developed by Ward Cunningham in 1995. Ward Cunningham defines the wiki as "a social computing system that allows a group of users to create and connect a set of web pages with simple descriptions". Some people believe that, the Wiki system is a network of human knowledge, which allows people to browse, create and change Wiki texts on the basis of the web, and the cost of creating, changing and publishing is much higher than HTML text small. At the same time, the Wiki system also supports community-oriented collaborative writing, providing necessary help for collaborative writing. Finally, the writers of Wiki form a community, and the Wiki system provides a simple communication tool for this community. Compared with other hypertext systems, wikis are easy to use and open, which helps to share knowledge of a certain area within a community.

MediaWiki is a free and open source wiki suite originally developed for Wikipedia, written in PHP. All non-profit Wikimedia projects under the Wikimedia Foundation, numerous other wiki sites on the Internet, and the MediaWiki homepage are now based on this software.

For making the use of MediaWiki functions in our project, we use an API called pywikiapi, which is one of the python client API of MediaWiki, through which we can get the same results as when we search in the wikidata homepage, it means that we could get a bunch of most relevant named entities by input a probable name of the entity you search.

Here is the brief description of pywikiapi: This is a minimalistic library that handles some of the core MediaWiki API complexities like handling continuations, login, errors, and warnings, but does not impose any additional abstraction layers, allowing you to use every single feature of the MW API directly in the most optimal way. The library was written by the original author of the MediaWiki API itself.

The use of MediaWiki APIwill be something like this, and the use in our case in detail will be illustrated in a next chapter:

```python
from pywikiapi import wikipedia
# Connect to English Wikipedia
site = wikipedia('en')

# Iterate over all query results as they are returned
# from the server, handling continuations automatically.
# (pages whose title begins with "New York-New Jersey")
for r in site.query(list='allpages', apprefix='New York-New Jersey'):
  for page in r.allpages:
    print(page.title)

# Iterate over two pages, getting the page info and the list of links for each of t
for page in site.query_pages(titles=['Test', 'API'], prop=['links', 'info'], pllimi
    print(page.title)
    print(', '.join([l.title for l in page.links]))
    print()
```

Figure 4.7 : part of python script of using MediaWiki API

4.3 Web Crawler on resources of books

Whenever someone wants to retrieve and store some specific information on a specific website, web crawler has always been some kind of first choice. Since in our case, more relevant information with high quality may help a lot the annotation of the text, it's natural to think about whether web crawler could be used. Here, as a basement example, in our project we only extract useful information from one book on Internet.

4.3.1 The composition of python web scrawlers

The Python crawler architecture is mainly composed of five parts, namely the scheduler, URL manager, web page downloader, web page parser, and application.

- **Scheduler:** equivalent to the CPU of a computer, mainly responsible for scheduling the coordination between the URL manager, downloader, and parser.
- **URL manager:** includes the URL address to be crawled and the crawled URL address to prevent repeated crawling of URLs and circular crawling of URLs. There are three main ways to implement URL managers, which are implemented through memory, database, and cache database.
- **Web page downloader:** Download a web page by passing in a URL address, and convert the web page into a string. The web page downloader has urllib3 including login, proxy, and cookies, and requests.
- **Web page parser:** Parsing a web page string, we can extract our useful information according to our requirements, or it can be parsed according to the DOM tree parsing method. There are some different kinds of parsers, such as regular expressions, beautifulsoup, and lxml, etc.

- **Application:** An application composed of useful data extracted from a web page.

Here we will give some brief descriptions of several most popular python libraries to implement a web scrawler.

4.3.2 urllib: the built-in URL handling module of python

urllib module is the URL handling module for python. It is used to fetch URLs (Uniform Resource Locators). It uses the *urlopen* function and is able to fetch URLs using a variety of different protocols.

Urllib is a package that collects several modules for working with URLs, such as:

- urllib.request for opening and reading.
- urllib.parse for parsing URLs
- urllib.error for the exceptions raised
- urllib.robotparser for parsing robot.txt files

In our project, the other libraries could be better choices since the workload are usually much less than using urllib.

4.3.3 Requests: make HTTP requests more human-friendly

As the warning written in the introduction page of the project Requests by the original author Kenneth Reitz : Non-professional use of other HTTP libraries can cause dangerous side effects, including: security flaws, redundant code disorders, reinvented wheel syndrome, documentary disorders, depression, headaches, and even death. Requests allows you to send pure natural, plant-raised HTTP / 1.1 requests without manual labor. You don't need to manually add a query string to the URL, or form-encode the POST data. Keep-alive and HTTP connection pools are 100% automated, and all power comes from urllib3, which is rooted in Requests.

It can be said that the biggest feature of Requests is its simple and direct elegance. Whether it is the request method or the processing of the response result, as well as cookies, URL parameters, and post submission data, all reflect this style.

Here is a simple example:

```
>>> import requests
>>> resp = requests.get('https://www.baidu.com')

>>> resp.status_code
200

>>> resp.headers['content-type']
'application/json; charset=utf8'

>>> resp.encoding
'utf-8'

>>> resp.text
u'{"type":"User"...'
```

Figure 4.8 : an example of using requests.

It can be seen that both the initiation of the request and the corresponding processing are very intuitive.

Here the beloved features of the Requests library are list below :

- Keep-Alive & Connection Pooling
- International Domains and URLs
- Sessions with Cookie Persistence
- Browser-style SSL Verification
- Automatic Content Decoding
- Basic/Digest Authentication
- Elegant Key/Value Cookies
- Automatic Decompression
- Unicode Response Bodies
- HTTP(S) Proxy Support
- Multipart File Uploads
- Streaming Downloads
- Connection Timeouts
- Chunked Requests
- `.netrc` Support

4.3.4 Selenium

What is Selenium? As the official web page says, Selenium automates browsers. That's it! What you do with that power is entirely up to you. Primarily, it is for automating web applications for testing purposes, but is certainly not limited to just that. Boring web-based administration tasks can (and should!) be automated as well. Selenium has the support of some of the largest browser vendors who have taken (or are taking) steps to make Selenium a native part of their browser.

It is also the core technology in countless other browser automation tools, APIs and frameworks.

Selenium was born for testing. But it wasn't expected that in the age of scrawlers, it turned into a good tool for writing a scrawler. To make a brief summarization about Selenium in one sentence: it can control your browser, and it can imitate human beings' action of browsing the web pages.

Although selenium is a good tool to create a scrawler, but it also has some obvious disadvantages:

- Slow. Each time you run the crawler, you open a browser. If it is not set, it will load a lot of things such as pictures, JS, etc.
- Takes up too much resources. Some people say that when you replace Chrome with a headless browser PhantomJS, but the principle is the same, still needs opening a browser, and many websites will verify parameters.
- The requirements on the network will be higher. Selenium loads a lot of supplementary files (like css, js, and image files) that may not be of value to you. This may generate more traffic than just requesting the resources you really need (using a separate HTTP request).
- The crawl scale can't be too large. It's rarely seen any company using Selenium in the production environment.
- Difficult to learn. The cost of learning Selenium is too high. The cost of learning Selenium is many times more difficult than Requests

According to the disadvantages depicted above, in our project we only use Selenium to find the element of changing to the next page and click the element.

4.3.5 Beautiful Soup

Beautiful Soup is a python package for parsing HTML and XML documents (including having malformed markup, i.e. non-closed tags, so named after tag soup). It creates a parse tree for parsed pages that can be used to extract data from HTML, which is useful for web scrawlers.

As a summarization of the process of crawling the web, so that one have a better understanding of the function of BeautifulSoup :

1.Select the url to crawl
2.Log in to this URL using python (urlopen, etc.)
3.Read web page information (read () the information from the page_source)
4.Put the read information into BeautifulSoup

5.Use BeautifulSoup to select tag information, etc. (substitute using regular expressions)

As a HTML parser tool, from the process depicted above, we can figure out that Beautiful Soup only works at the last steps to extract some specific information according to the tags or some other rules.

Beautiful Soup transforms complex HTML documents into a complex tree structure. Each node is a Python object. All objects can be summarized into 4 types: Tag, Navigable String, Beautiful Soup, Comment.

4.4 ElasticSearch

Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multi-tenant full-text search engine with an HTTP web interface and schemaless JSON documents. Elasticsearch is developed in Java and released as open source software under the Apache license. Official clients are available in Java, .NET (C #), PHP, Python, Apache Groovy, Ruby, and many other languages. According to the ranking of DB-Engines, Elasticsearch is the most popular enterprise search engine, followed by Apache Solr, which is also based on Lucene.

4.4.1 Full-text search

In text retrieval, full-text search refers to techniques for searching a single computer-stored document or a collection in a full-text database. Full-text search is distinguished from searches based on metadata or on parts of the original texts represented in databases (such as titles, abstracts, selected sections, or bibliographical references).

Since usually the normal databases also can fulfill some need of data storage and data search, why is there still need of full-text search engines? As some **reference**[15] and [16] introduce, there are mainly two reasons:

- Full-text search engines are more suitable for unstructured text queries.
- For document databases like Mongo, full-text search engines are better at maintaining indexes.

Here, we list some of the main advantages of full-text search engine :

- Sub-second search results indicate which files may contain one or more terms (words, numbers, etc.) in millions or billions of user searches. This includes a good search of all text fields, and limited functionality for searching non-text data. This may also include the classification or search results of the payload or search results based on specific values of a specific field.
- Abundant and flexible text query tools and sophisticated ranking functions to find the best documents / records.
- Basic functions for adding, deleting or updating documents / records
- Basic functions for storing data (rather than simple indexing and searching). Not all full-text search systems support this feature, but most include Lucene / Solr

Here, we list some of the main occasions of full-text search engine :

- Large amount of free structured text data (or records containing such data) to search or aspect / category-hundreds of thousands or millions of files / records (or more).
- Supports a large number of interactive text-based queries.
- Demand very flexible full-text search queries.
- Demand for highly relevant search results is not met by available relational databases.
- Relatively less need for different record types, non-text data manipulation, or secure transactions.

4.4.2 Lucene

Lucene is an open source library for full-text retrieval and search, supported and provided by the Apache Software Foundation. Lucene provides a simple but powerful application program interface that can do full-text indexing and searching. In the Java development environment, Lucene is a mature free and open source tool; for its part, Lucene is now and for several years, The most popular free Java information retrieval library.

Lucene is not a complete full-text indexing application, but a full-text indexing engine toolkit written in Java, which can be easily embedded into various applications to implement application-specific full-text indexing / retrieval functions.

There are already many applications whose search function is based on Lucene, such as the search function of Eclipse's help system. Lucene can index text type data, so as long as you can convert the text format of the data you want to index, Lucene can index and search your documents. For example, if you want to index some HTML documents and PDF documents, you first need to convert the HTML documents and PDF documents into text format, then transfer the converted content to Lucene for indexing, and then save the created index file to In the disk or memory, the index file is finally queried according to the query conditions entered by the user. Not specifying the format of the documents to be indexed also makes Lucene suitable for almost all search applications.

In addition, most of the search (database) engines use a B-tree structure to maintain the index. Updating the index will cause a large number of IO operations. Lucene is slightly improved in the implementation: instead of maintaining an index file, it is expanding. During indexing, new index files are constantly created, and then these new small index files are periodically merged into the original large index (for different update strategies, the size of the batch

can be adjusted), so that it does not affect the efficiency of retrieval Under the premise, the efficiency of the index is improved.

Here is a figure from reference[4.4.2.1] which represents the relationship between search applications and Lucene, and also reflects the process of building search applications with Lucene :

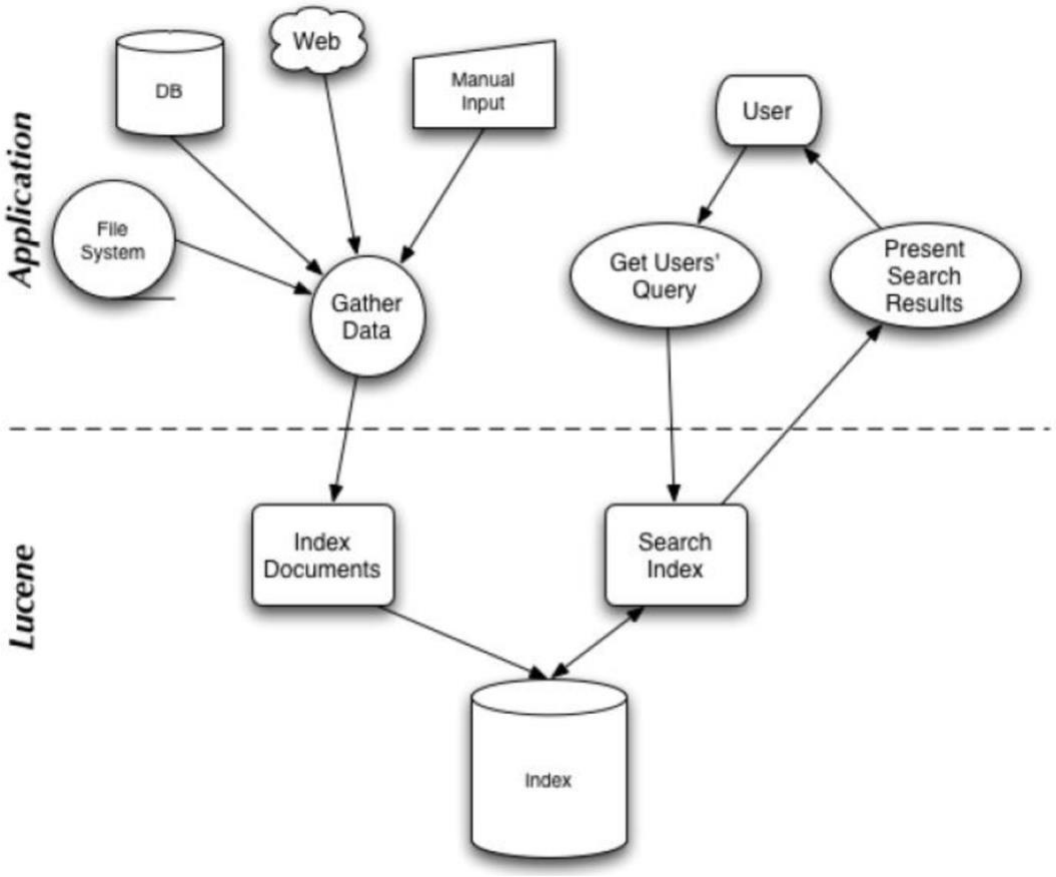

Figure 4.9 : the relationship between search engine application and Lucene

The Lucene package is distributed as a JAR file, here we will introduce briefly the composition of the Lucene Jar file, then we can clearly know the core parts of Lucene :

- Package: org.apache.lucene.document : This package provides some classes needed to encapsulate the documents to be indexed, such as Document, Field. In this way, each document is finally encapsulated into a Document object.
- Package: org.apache.lucene.analysis : The main function of this package is to tokenize documents. Because documents must be tokenized before indexing, the role of this package can be seen as preparation for indexing.

- Package: org.apache.lucene.index : This package provides classes to assist in creating indexes and updating the created indexes. There are two basic classes: IndexWriter and IndexReader, where IndexWriter is used to create an index and add documents to the index, and IndexReader is usedto delete documents in the index.
- Package: org.apache.lucene.search : This package provides the classes needed to perform a search on a built index. For example, IndexSearcher and Hits, IndexSearcher defines the search method on the specified index, Hits is used to save the search results.

From the information above, we can easily figure out that the main parts of Lucene. Document module only works for what is the form of the unstructured input text will be in Lucene. And analysis module works both during the index time and the search time. In the next part of this chapter, the index procedure and search procedure will be introduced briefly separately.

4.4.3 Index documents

Indexing is the core of modern search engines. The process of indexing is the process of processing source data into index files that are very convenient to query. Why is indexing so important? Imagine that you are searching for documents containing a certain keyword in a large number of documents. If you do not create an index, you need to read these documents into memory in order, and then check whether this article contains To search for keywords, this will take a lot of time. Think of the search engine in milliseconds to find the search results. This is because the index is established. You can think of the index as a data structure that enables you to quickly and randomly access the keywords stored in the index, and then find the documents associated with the keywords. Lucene uses a mechanism called inverted index. Reverse indexing means that we maintain a list of words or phrases. For each word or phrase in this list, there is a linked list describing which documents contain the word or phrase. In this way, when the user enters the query conditions, the search results can be obtained very quickly.

4.4.4 Inverted Index

Here, we only give a glimpse on the conception of inverted index, which is the core conception and core part of the indexing part of search engine. And the reason why inverted index was invented is to improve the speed of response of the query when using a search engine. For example, it will cost more or less several seconds to find a query keyword in a 10M file using the traditional forward index, but it only costs several milliseconds using inverted index during the phase of indexing.

Here an example of how does the data structure of inverted index look like :

Documents that we want to index into Lucene(Elastic Search):

1. Politecnico is a good university.
2. The students in Politecnico are smart.
3. Only smart students can come to Politecnico.

Then, all the distinguished words that appear in the three documents above will be stored as a list, and for each of the word, the document ids of the documents where the word appears will be linked to the word ids.

After deleting the stop words, which are not really meaningful from the semantic insight, such as 'is', 'a', 'in', 'the', etc. And also there are also the procedure of stemming which will make the word transform to its original stem version, like 'students' to 'student'. The words left will be stored as a list but with their ids following by the document ids where the words appear.

1	Politecnico	1,2,3
2	good	1
3	university	1
4	student	2,3
5	smart	2,3
6	come	3

But in fact, the structure of inverted index is much more complicated than the description above, since there are also some other kind of useful information stored in the meanwhile, such as TF(term frequency, which means how many times the word appears in a document), position(where the word appears in a document), for the sake of the calculation of the relevance score between the word and the document, which is especially important for the ranking of the search result in a search system.

4.4.5 Search index

Here we only do a brief introduction of the search procedure of Lucene(Elastic Search).First we will give a little example of how does the procedure look like. And then a list of different kind of query types will be depicted and introduced briefly.

In the last part we created an inverted index from three documents, then if we search the word/query 'Politecnico' on the same index, what will happen?

As **reference [4.4.5]** shows, in fact the list of term of the inverted index is a special data structure in Lucene called 'term dictionary'. When the query 'Politecnico' has been searched, first the analysis module should tokenize the query and use some filters(like transform uppercase into lowercase) so that the query will be transformed into a list of terms(here since the example query has only one word, there's actually no tokenization happens), then according to different types of query, the query terms will be searched in the term dictionary and further combine the result of each term in different ways. Finally, there's a model which is in charged of giving a relevance score to each result document in the index, and the ranked list of the result documents will be shown as the final result of search.

Here different types of queries are listed :

- **Boolean query** : A Query that matches documents matching Boolean combinations of other queries, e.g. TermQuerys, PhraseQuerys or other BooleanQuerys.
- **Term query** : A Query that matches documents containing a term. This may be combined with other terms with a BooleanQuery.
- **Phrase query** : A Query that matches documents containing a particular sequence of terms.
- **Fuzzy query :** Implements the fuzzy search query. The similarity measurement is based on the Damerau-Levenshtein (optimal string alignment) algorithm, though you can explicitly choose classic Levenshtein by passing false to the transpositions parameter.At most, this query will match terms up to 2 edits. Higher distances (especially with transpositions enabled), are generally not useful and will match a significant amount of the term dictionary. If you really want this, consider using an n-gram indexing technique (such as the SpellChecker in the suggest module) instead.

4.4.6 ElasticSearch : Expansion and simplification of Lucene

Here we do a brief comparison of Elastic Search and Lucene :

- Luncene is a single-node API, and Elastic Search is distributed.
- Lucene focuses on the construction of the underlying search, while ElasticSearch focuses on enterprise applications.
- Luncene requires secondary development before it can be used. It can't be like Bing or Google, it just provides an interface that needs to be implemented before it can be used.
- Elasticsearch provides other supporting features like thread-pool, queues, node/cluster monitoring API, data monitoring API, Cluster management, etc.

Here we list some core concepts of the distributed Elastic Search :

- **Cluster** : A cluster. An ES cluster consists of one or more nodes, and each cluster has a cluster name as an identifier.
- **Node** : A node is actually an instance of ES, on one machine there can be multiple nodes, but run on different independent enviroments.
- **Index** : A collection of documents.
- **Shard** : An index has one or multiple shards, which looks like a bucket of water are distributed in multiple smaller buckets.

In fact, **each shard of ES is an index of Lucene.** The underneath running principle of each ES shard is absolutely the same as Lucene index, but ES provides some more user-friendly Restful API to hide the complexity of Lucene. And ES provides the possibility of a distributed using of Lucene functions so that it can be more capable for enterprise level use with large scale of data.

In our project, since there is the need of **fuzzy search**, which means there may be a lot of misspellings in the transcripts of the video lessons. And if we want to recognize the useful entities inside that kind of transcripts, we should at least have some capability of recognizing entities even though there are misspellings. And in the next chapters, the practical experiments and trying will be discussed more in details.

Chapter 5

Methods proposed

In our context of the project, there are at least two needs that should be fulfilled. Since usually in the transcript of the video lectures there are plenty of misspelling. For example, the final annotation system should recognize the possibility that the string 'logistic regresion' may in fact refers to 'logistic regression', that's the reason why there's the first need of **fuzzy search**.

Secondly, when the students search a query which they are interested in using the search function added to video lecture, they may search a different name of the same entity. For example, maybe during the lecture the professor used the word 'SVM' for convenience, but the students may search 'support vector machine' when using the search function of the video lecture. The final system should have the capability of giving as many as possible the time durations where the professor explained the same specific topic to help a better understanding. That's the reason why the final annotation system should recognize as many as domain-specific named entities and would better also link the mentions to a specific entity in a knowledge base. The procedure of giving the raw text semantic annotation is called text annotation. For implementing a useful search function on video lectures, annotation of the transcript with high quality can be the key problem.

Thus in our project, we did some experiments of fuzzy search using Elastic Search, which is one of the most popular search engine based on Lucene. And there are different ways to do fuzzy search in ES, but the mechanism under different methods of ES are different. From a absolute beginner of ES, as did more and more experiments with ES, we became a better beginner gradually. In the next pages of this chapter, we will introduce the procedure of the experiments we did more in details, such as the beginning with ES, the experiment of creating the index, and the different tries of getting a more acceptable fuzzy search.

As depicted in chapter 3, to annotate the lectures of one single course is a domain-specific task, which means the named entities appear and should be recognized are usually from the same domain or field. As a result, if there are some method that can obtain a collection of domain-specific keywords or named entities, all the other down streaming NLP techniques can be benefited by leverage the domain specific keywords or named entity collection created by our domain specific contextual learning methods.

5.1 Fuzzy search with Elastic Search

The work we did using ES is very experimental, and the version of ES we used is 6.7.2, but there is a annoying fact that the version of ES updates really rapidly which will make old characteristic or functions may not work or exist in the new version. In this part, since our main target is implementing fuzzy search to overcome misspelling in the transcript of video lectures, then this part will focus on the fuzzy match functions and principles in ES, but there are still some preparation steps before the experiments of fuzzy search. In the next pages of this chapter, we will introduce the experiments in a natural order.

5.1.1 A brief start of ES

Here we list the steps of installation of ES as a record :

1.Elasticsearch requires at least Java 8.Before you install Elasticsearch, please check your Java version first by running (and then install/upgrade accordingly if needed) :

```
java -version
```

2.Download the Elasticsearch 6.7.2 tar as follows :

```
curl -L -O https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-

6.7.2.tar.gz
```

3. Extract it as follows :

```
```

4. It will then create a bunch of files and folders in your current directory. We then go into the bin directory as follows :

```
cd elasticsearch-6.7.2/bin
```

5. Now we are ready to start our node and single cluster :

```
./elasticsearch
```

6. Then we can check the basic information(version of ES, name of cluster, etc) of the Elastic Search installed by using :

```
curl -XGET "localhost:9200"
```

Kibana is an open source data visualization dashboard for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts and maps on top of large volumes of data.

For the installation of Kibana :

```
wget https://artifacts.elastic.co/downloads/kibana/kibana-
6.0.0-linux-x86_64.tar.gz
sha1sum kibana-6.0.0-linux-x86_64.tar.gz
tar -xzf kibana-6.0.0-linux-x86_64.tar.gz
cd kibana
```

Now we are ready to start Kibana:

```
./bin/kibana
```

To visualize the utilization of Kibana, we should open a browser and go to the site ' http://localhost:5601'

In our project, we need to import the Wikipedia labels in json form to ES index. Open the Kibana in a browser and click the block called 'Dev Tools', that's where we can interact with Elastic Search using curl-like command .Here, we create a mapping for the json data that will be imported to create the index:

```
PUT /labels
{
 "mappings": {
  "doc": {
       "properties": {
          "text_entry":{"type":"keyword"}
       }
  }
 }
}
```

```
`curl -H 'Content-Type: application/x-ndjson' -XPOST
'localhost:9200/labels/doc/_bulk?pretty' --data-binary
```

5.1.2 Indexing wiki-labels into Elastic Search

First we write a python script to transform the original wiki-labels data which is stored in txt form to json form in order to make it readable to the elasticsearch, and a part of the original labels is like :

```
Christer Englund
Anaemia in pregnancy: occurrence in two economically different
clinic populations of Karachi.
Alison Bell Memorial Award. The nurse as first assistant to the
surgeon: is this a perioperative nursing role?
Curtis Monument In Churchyard, 1 Metre North Of Chancel, Church
Of The Blessed Virgin Mary
[Imported cases of malaria]
Zatoichi and the Chest of Gold
Category:1976–77 in Welsh football
Mesci_5136
Inhibiting ERK Activation with CI-1040 Leads to Compensatory
Upregulation of Alternate MAPKs and Plasminogen Activator
Inhibitor-1 following Subtotal Nephrectomy with No Impact on
Kidney Fibrosis.
The dynamics of cortical folding waves and prematurity-related
deviations revealed by spatial and spectral analysis of
gyrification.
Epiperipatus evansi
gamma-glutamylputrescine synthase
BN112_3529
Orchard Park Projects
Category:Xiriâna lemmas
Cornelius Scipio Asiaticus
```

Figure 5.1 : part of labels which will be indexed in Elastic Search.

But with the raw text form of the wiki labels we cannot directly import it into ES, since it's necessary to transform the raw text to Json format. As a result, a python script was written to transform the raw text version in which each row is a entities' label of a named entity.

After format transformation, the Json format of wiki labels should be like this :


```
{"index":{"_index":"labels","_id":1}}
{"text_entry":"Christer Englund"}
{"index":{"_index":"labels","_id":2}}
{"text_entry":"Anaemia in pregnancy: occurrence in two
economically different clinic populations of Karachi."}
{"index":{"_index":"labels","_id":3}}
{"text_entry":"Alison Bell Memorial Award. The nurse as first
assistant to the surgeon: is this a perioperative nursing role?"}
{"index":{"_index":"labels","_id":4}}
{"text_entry":"Curtis Monument In Churchyard, 1 Metre North Of
Chancel, Church Of The Blessed Virgin Mary"}
{"index":{"_index":"labels","_id":5}}
{"text_entry":"[Imported cases of malaria]"}
{"index":{"_index":"labels","_id":6}}
{"text_entry":"Zatoichi and the Chest of Gold"}
{"index":{"_index":"labels","_id":7}}
{"text_entry":"Category:1976–77 in Welsh football"}
{"index":{"_index":"labels","_id":8}}
{"text_entry":"Mesci_5136"}
```


Figure 5.2 : DBpeida labels that have already been indexed in Json form.

When we were trying to index the whole json data of labels into ElasticSearch, we met the problem that the size of json data 3.1 GB is larger than the default JVM heap size of ElasticSearch 1GB. Then we should revise the setting of heap size, since our max memory of ubuntu is 8 GB , then we set the heap size of ElasticSearch as 4 GB. And we did it in the file jvm.options in the config folder of ElasticSearch :

```
## JVM configuration

################################################################
## IMPORTANT: JVM heap size
################################################################
##
## You should always set the min and max JVM heap
## size to the same value. For example, to set
## the heap to 4 GB, set:
##
## -Xms4g
## -Xmx4g
##
## See https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html
## for more information
##
################################################################

# Xms represents the initial size of total heap space
# Xmx represents the maximum size of total heap space

-Xms4g
-Xmx4g

################################################################
```

Figure 5.3 : Check the heap size of Elastic Search.

But even though we set a heap size which is larger than 3.1 GB (json file size), the "out of binary memory error" continues to occur , then we decided to write a bash script to split the original relative big json file into smaller ones and index them individually :

```
# split the main file into files containing 1000,000 lines max
printf "start split\n";
split -l 1000000 -a 3 -d out.json
/home/parallels/Documents/ElasticSearch_data/split_parts/out_b
ulk;
printf "split done\n";
# send each split file
BULK_FILES=/home/parallels/Documents/ElasticSearch_data/split_
parts/out_bulk*;
i=1;
for f in $BULK_FILES;
do
   curl -H 'Content-Type: application/x-ndjson' -s -XPOST
localhost:9200/labels/type_labels/_bulk --data-binary @$f
   let i+=1;
   echo $i
```

Figure 5.4 : Bash script to index large dataset into ES.

Then if the indexing successed, we can check the basic information of the index through the command : curl -XGET 'localhost:9200/labels?pretty'

Figure 5.5 : successful index creation in ES.

And we can also do some search by search query like :
curl -XGET 'localhost:9200/labels/_search?q=smoothie&pretty' ：

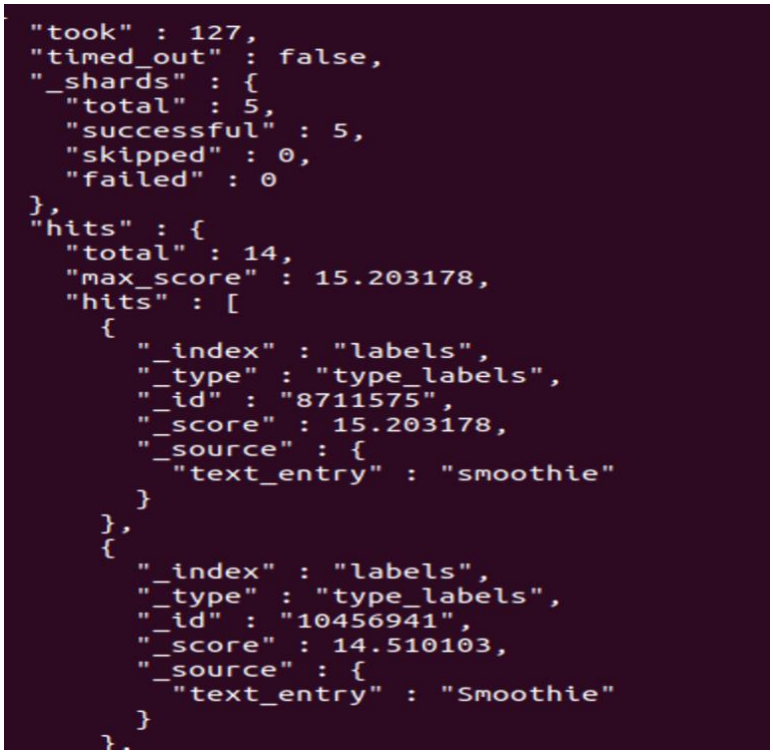

Figure 5.6 : A search example in ES with query ‘smoothie’.

5.1.3 A try to extend the max Levenshtein distance limit of fuzziness more than 2 in ES

After spending a lot of time trying to break the limit of the max Levenshtein distance in ES, we finally find that it seems almost impossible to reach this goal directly in ES by writing scripts or plugins nowadays. But during the trip of this trying, we finally got into the deep details and also tried to use some alternative ways to do the fuzzy search(or simulate the fuzziness more than 2) in ES.

And actually there are several reasons which make this problem difficult. As we know ElasticSearch is based on Lucene which is a strong search engine library written by Doug Cutting, and for the fuzziness part of Lucene, a library or algorithm called 'Levenshtein Automata' was implemented by Robert Muir and used first in Lucene 4.0 in 2011 which made the FuzzyQuery faster 100 times than before.

The story of the implementation of 'Levenshtein Automata' has been posted in the blog of Michael McCandless: Lucene's FuzzyQuery is 100 times faster in 4.0，from which we can find that they both struggled a lot reading the paper of Fast String Correction with Levenshtein-Automata (2002) by Klaus Schulz and Stoyan Mihov, and also implementing the algorithm in Java according to the paper. For

understanding the theory part and the implementation of 'Levenshtein Automata', one can directly check the relevant document of Lucene for a glacne and then the source code of LevenshteinAutomata Class of Lucene in github for the ground-truth details, and from the github repo of Lucene we can figure out that nearly all the fuzzy functions of different fuzzy-relevant Classes are all based on 'Levenshtein Automata'.

Otherwise, there are also some great explanations and implementations of 'Levenshtein Automata'. The first is Levenshtein automata can be simple and fast(2015), in which Jules Jacobs implemented a version of time complexity O(length of string) and an improved version of time complexity O(max edit distance).

The second one is Damn Cool Algorithms: Levenshtein Automata(2010), which is more in details to illustrate the idea of 'Levenshtein Automata', and this blog explained the algorithm in a very intuitive way : first by constructing a NFA(Non-deterministic finite automaton) with the target word(query) and the max edit distance allowed, then since some computational reason the constructed NFA should be converted to a DFA(deterministic finite automaton).Then when the DFA is constructed according to the relevant NFA with the target word and the max allowed edit distance, any incoming word can 'walk' into the 'maze' of the constructed DFA to check whether this word can finally access to one of the acceptable states which means the incoming word is within the distance less than the max allowed edit distance to the target word.

And In the next step, we should know how to use the constructed DFA to check which are the acceptable words among the terms from the documents indexed in ES. the documents are indexed in ES with the data structure of inverted index which looks like a HashMap, the terms can be seen as the keys, the ids of documents where a certain term occurs are listed as a posting which can be seen as the value of the key. Then we leave the postings aside now, the terms as the keys are stored as a sorted list in ES(actually the sorting is within a segment), and in ES, during the query time, there's a variant of Binary-Search algorithm has been used to search the position of a certain query term in the sorted list of terms. After all, with both the constructed DFA of the query word and the terms-dictionary of the index in ES which can be seen as a sorted list, there are some methods illustrated in the blog above can relatively check out the set of acceptable terms efficiently.

5.1.4 Current solution : tri-gram analyzer fuzzy search

After spending a lot of time trying to expend the limit of 2 characters in fuzzy match of ES, we temporarily compromise to another methods which is more easier to be implemented, and this method is to use Ngram analyzer of ES, which can generate the ngrams of each token(label), and create the index with all the ngrams, as a result, one can get the original label by searching only a part of it.As described in the official site of ES, the ngram tokenizer first breaks text down into words whenever it encounters one of a list of specified characters, then it emits Ngrams of each word of the specified length. N-grams are like a sliding window that moves across the word - a continuous sequence of characters of the specified length.

Here is an example about how does Ngram analyzer work in ES, with configuration of the ngram tokenizer to treat letters and digits as tokens, and to produce tri-grams (grams of length 3):

```
PUT my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_analyzer": {
          "tokenizer": "my_tokenizer"
        }
      },
      "tokenizer": {
        "my_tokenizer": {
          "type": "ngram",
          "min_gram": 3,
          "max_gram": 3,
          "token_chars": [
            "letter",
            "digit"
          ]
        }
      }
    }
  }
}

POST my_index/_analyze
{
  "analyzer": "my_analyzer",
  "text": "2 Quick Foxes."
}
```

Figure 5.7 : Setting tri-gram analyzer of ES for fuzzy search.

And the result of the analyze with tri-gram will be :

```
[ Qui, uic, ick, Fox, oxe, xes ]
```

If only analyzer is specified in the mapping for a field, then that analyzer will be used for both indexing and searching. If I want a different analyzer to be used for searching than for indexing, then I have to specify both.

Another important and also tricky point is how to set the min and max n-gram limit when using the ngram analyzer(n-gram tokenizer or n-gram filter).Until now I think to find the optimal min and max n-gram numbers can be only achieved by experiments. In the reference link below there is a brief summary about the experience of the author on how to set these two parameters:

Mingram/Maxgram Size

"Notice that the minimum ngram size I'm using here is 2, and the maximum size is 20. These are values that have worked for me in the past, but the right numbers depend on the circumstances. A common use of ngrams is for autocomplete, and users tend to expect to see suggestions after only a few keystrokes. Single character tokens will match so many things that the suggestions are often not helpful, especially when searching against a large dataset, so 2 is usually the smallest useful value of mingram. On the other hand, what is the longest ngram against which we should match search text? 20 is a little arbitrary, so you may want to experiment to find out what works best for you. Another issue that should be considered is performance. Generating a lot of ngrams will take up a lot of space and use more CPU cycles for searching, so you should be careful not to set mingram any lower, and maxgram any higher, than you really need (at least if you have a large dataset)."

----more details in : https://qbox.io/blog/an-introduction-to-ngrams-in-elasticsearch

And there is a kind of statistics which may be helpful to be a reference to choose the min and max n-gram numbers: the distribution of the word length for a specific language, here are some examples in the pictures below, and from which we can see that almost all of the Italian words have a length from 3 to 16, and more than 99% of Italian words have a length from 4 to 15; Meanwhile for English words the case is similar but a little bit different:

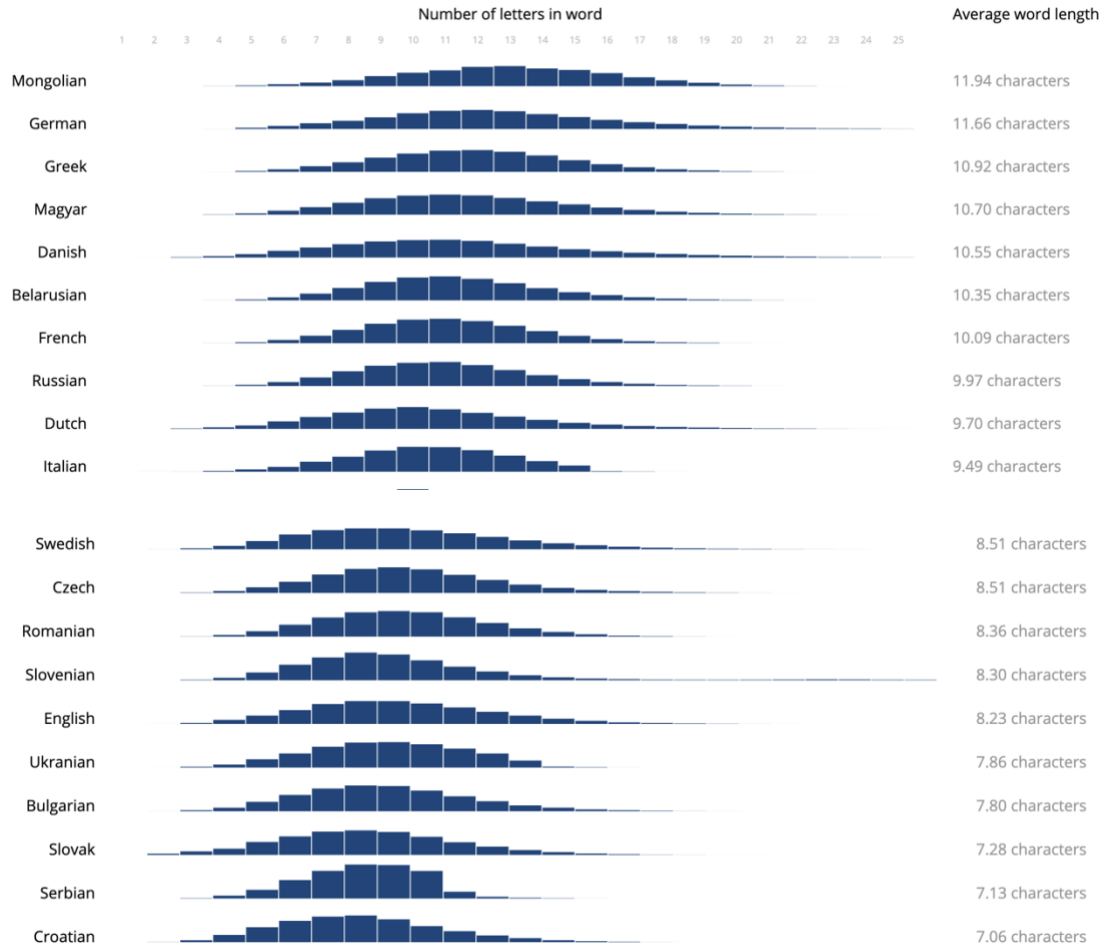

Figure 5.8 : Word length distribution of different languages.

the pictures above are from the site: http://www.ravi.io/language-word-lengths

Another very useful tool is the _explain API in ES, which can help us understand how is a relevance score has been calculated by the calculations of different factors. Especially for the default relevance score calculation **BM25,** which is a variant of traditional **tf-idf** score, the _explain API can explain the details of the calculations of each factors in the calculation of the final BM25 relevance score. But if someone wants to change the default BM25 relevance to use his own custom relevance score, one should write his own Explain Class in Java for the use of _explain API.

Whenever we want to check how the relevance score is calculated in our specific cases for a specific pair of query and document in details, the first tool we should have a try is the '_explain' API. In our case , although we use n-gram tokenizer to do the analysis both in indexing and searching phases, the final relevance score is still calculated using the default BM-25 method in ElasticSearch.

And the use of _explain API is depicted in the official document of ES 6.7.2 here:

https://www.elastic.co/guide/en/elasticsearch/reference/6.7/search-explain.html

5.2 Domain specific contextual learning

This is the second contribution of this thesis for the whole project of giving semantic annotation to the video lecture transcript and implementing the search function to the video lectures. If as many as the named entities or keywords which belong to a specific domain can be automatically found and collected, then the collection of domain specific named entities can in some way maximize the recall of the students' search using the search function provided by our final project. Through implementing a method which leverages several different tools, we finally reach a 98% recall on the named entity labels collection related to the domain 'database'. In addition, this method can be easily expanded to other different subjects or domains, such as math, physics and machine learning, etc.

To make a brief summary of the content we have done in this part, there are mainly two steps. The first one is the way how could we get the information resource which probably contain the information we need, and this part is the work we did in this thesis. The second one is the way to extract the named entity from the information resource obtained in the first step, and this part was done by my college in the project. Then, in this part, we will mainly introduce the work of the first step, but the second part will be also introduced briefly to make a better understand of the whole system.

In the first step that automatically collect the most probably useful resource on the Internet, there are mainly three different tools that we used from very heuristic ideas. And some of the basic introductions were already done in Chapter 4. Here we will further address how we use them to fulfill our need of the whole annotation and search system in details.

5.2.1 Google custom search on Coursera

As we know, **Coursera** is one of the most popular open course platform or website. And Coursera contains plenty of information of different courses from different domains, such as the catalog of the courses, the slides and transcripts. For example, if we want to annotate the a series of video lecture transcripts of a course from Polito, the catalog, the slides and transcripts of all the other open course in the same domain may help a lot. More in details, when we want to annotate the course 'Analysis I' of Polito, the open course 'Analysis I' of MIT, or

some other universities can provide many text information very similar in the semantic way.

But there are many different courses have a name exactly or similar to ‘Analysis I’, and they may all can contribute more or less. Then whether there is a way to retrieve all the relevant courses of a domain? By set the website on which we want to searchin Google custom search(here the website of Coursera), we can retrieve all the courses in Coursera automatically using Google custom search API. Here is an example when people search the query ‘database’ in Coursera website :

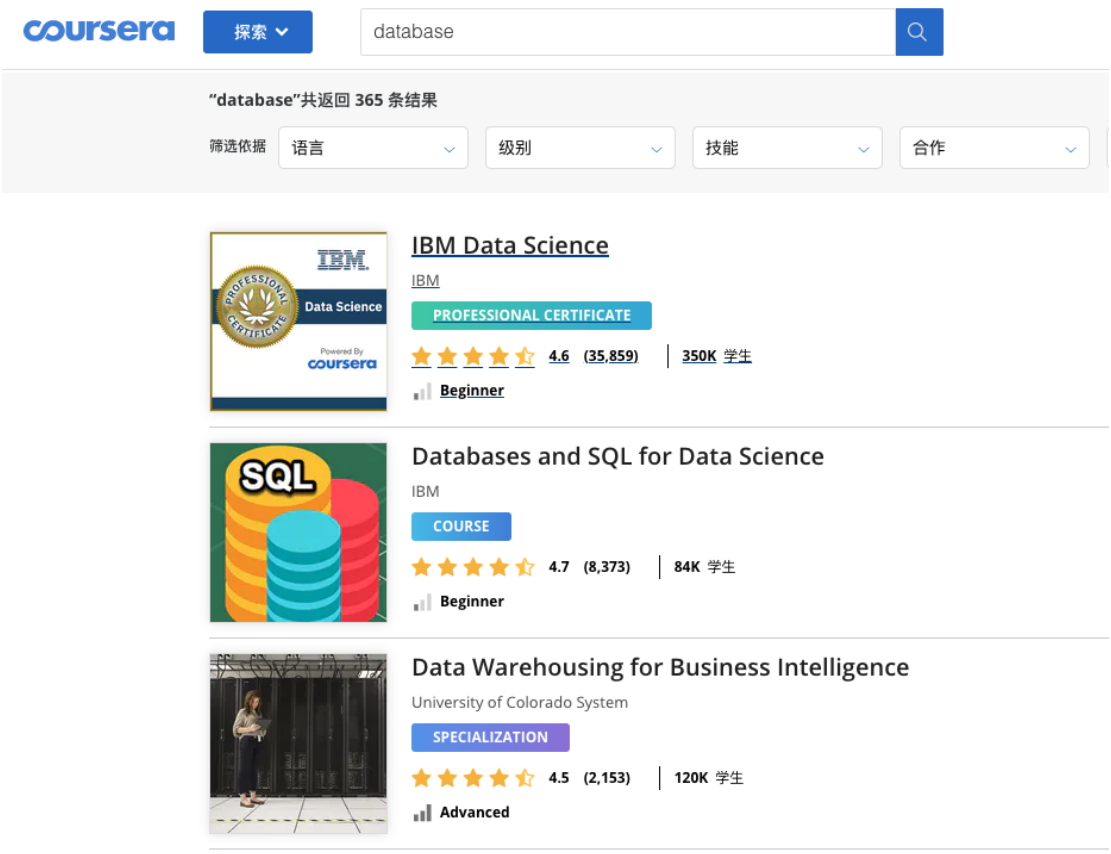

Figure 5.9 : Result list of courses searching ‘database’ in Coursera.

As the picture shows above, there will be a list of courses related to your query ‘database’. And by set the website of Coursera as the site we want to search in CSE, and use the python API of CSE, we can easily launch a query like ‘databse’ in our program, then get the list of search results automatically.

Here is an example to search in Coursera with python script using CSE, the query used here is 'the content of database course' :

```
In [7]: result = resource.list(q='content of database course',cx='013982082896578874498:yj

In [13]: result['items'][0]['link']

Out[13]: 'https://www.coursera.org/learn/database-management'

In [9]: for i in range(10):
            print(result['items'][i]['link'])

https://www.coursera.org/learn/database-management
https://www.coursera.org/learn/database-management/reviews
https://www.coursera.org/courses?query=database
https://www.coursera.org/learn/database-management/reviews?page=17
https://www.coursera.org/learn/database-management/reviews?page=8
https://www.coursera.org/learn/database-management/reviews?page=15
https://www.coursera.org/learn/sql-data-science
https://www.coursera.org/learn/dwrelational/reviews
https://www.coursera.org/learn/dwrelational
https://www.coursera.org/learn/python-databases
```

Figure 5.10 : Using google custom search API to search in Coursera in python.

As the figure above shows, 10 of the result courses have been listed. The maximum number of results in CSE is 500. Then after getting the list of relevant courses in Coursera, some kind of crawler techniques can be used to retrieve and collect useful information in each of the result courses. Here, we use BeautifulSoup which was introduced in Chapter 4 to do this job.

Here is an example of using BS4 to crawl the information by tags :

```
In [7]: soup = BeautifulSoup(r.text, 'html.parser')

In [10]: for div in soup.find_all('div',attrs={'class':'P_gjs17i-o_O-weightNormal_s9jwp

             print(div.text.strip('2m').rstrip(string.digits))

Course introduction video
Course objectives video lecture
Course topics and assignments video lecture
DBMS extensions video lecture
Relational database schema patterns video lecture
Colorado Education Data Warehouse video lecture
Data warehouse standards in health care video lecture
Powerpoint lecture notes for lesson
Powerpoint lecture notes for lesson
Optional textbook
Powerpoint lecture notes for lesson
Powerpoint lecture notes for lesson
Powerpoint lecture notes for lesson
Powerpoint lecture notes for lesson
Overview of software requirements
Overview of database software installation
Oracle installation notes
Making connections to a local Oracle database
SQL statements for Store Sales tables
SQL statements for Inventory DW tables
```

Figure 5.11 : Using BS4 to extract text from the result course page.

Then after this step, we can get text information with high quality which probably contain plenty of labels of named entities related to our query searched in CSE API. The next step is to use some kind of NER or NED tools to extract named entities from the results obtained in the last step. Here we briefly introduce several NER and NED extractors that have been used in our project :

- **Babelfy** : Babelfy is a unified, multilingual, graph-based approach to making Entity Linking and Word Sense Disambiguation, based on a free identification of the candidates' meanings with a heuristic that exploits a dense graph. It is based on BabelNet and jointly carries out disambiguation and entity linking.
- **DBpedia Spotlight** : DBpedia Spotlight is an open source project responsible for developing a system for automatic annotation of DBpedia entities of text in language natural. Provides interfaces for sentence localization (phrase recognition to be noted) and disambiguation (entity link).
- **AIDA** : AIDA was born with the idea that Wikipedia in English is only suitable for disambiguing general English texts such as news articles in English. Developing disambiguation systems for other domains and languages requires a great deal of adaptation to adapt to specific application scenarios. In addition, Wikipedia editions of many languages, such as Arabic, are an order of magnitude lower than the English Wikipedia. Therefore, it is essential to exploit inter-language links to enrich non-English resources.
- **AGDISTIS** : The AGDISTIS framework addresses two of the main drawbacks of the linking framework of current entities: time complexity and accuracy. AGDISTIS, is a framework that achieves the complexity of polynomial time. The framework is knowledge-base-agnostic (that is, it can be implemented on any knowledge base) and is also language independent.

Here is an example of using Babelfy to extract named entity from text :

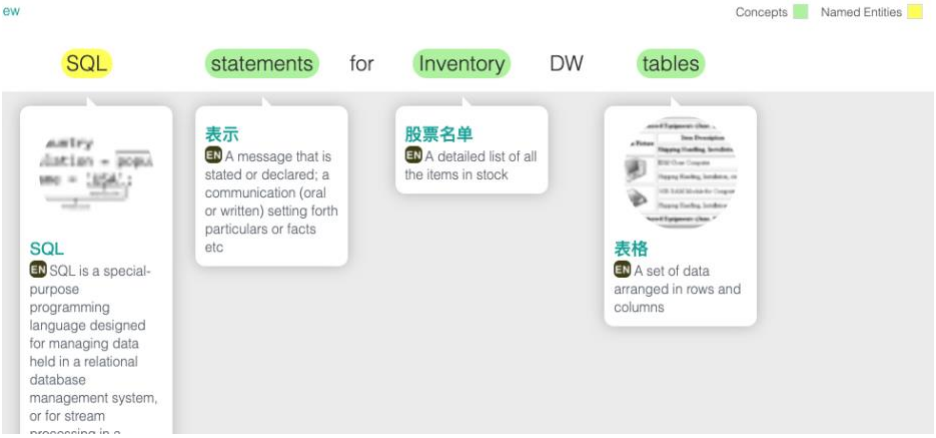

Figure 5.12 : An example using Babelfy extractor to extract named entities.

5.2.2 Web crawler on domain specific books on-line

Besides the domain-specific information obtained from Coursera, there is also the heuristic idea to collect domain-specific information from the books available on the Internet. For the purpose of obtaining text information from the books online, the techniques of web scrawlers must be the first idea comes to one's mind. And usually the books on line have at least two different kinds of versions, pdf and html. In order to get the ability of extracting information from different form of book resource, with a single book 'Database design 2nd edition' reference[5.2] derived from the website https://opentextbc.ca/, which provides both the pdf and html version of this book.

First, we introduce the web crawler implemented with a python script which is dedicated to the extraction of information from the html version of the book. Here, bs4 was mainly used as the parser tool of the html, and selenium introduced in Chapter 4 was used to change to the next page automatically.

```
def extract_page_by_order():

    section_contents = []
    children = soup.select('section')[0].children

    for child in children:

        dict_temp = dict()

        tag_name = child.name
        if tag_name==None:
            continue

        tag_text = child.text.strip().replace('\t\n',' ').replace('\n',' ').replace('\xa0',' ')
        dict_temp['type_tag']=tag_name
        dict_temp['text']=tag_text
        section_contents.append(dict_temp)

    return section_contents
```

```
def parse_book_html():
    count=0
    book_name = driver.find_element_by_xpath("/html/body/div[2]/header/div[2]/nav/h1/a").text
    pages_contents_list=[]
    dict_book = dict()
    dict_book['book_name']=book_name
    dict_book['type']='html'

    while True:
        try:
            section_content_list=extract_page_by_order()

            count+=1

            dict_temp = dict()
            key_1 = 'page_number'
            dict_temp[key_1]=count
            key_2 = 'section'
            dict_temp[key_2]=section_content_list
            pages_contents_list.append(dict_temp)

            dict_book['pages']=pages_contents_list
            print("finish processing the %d pages"%count)

            element_nextPage = driver.find_element_by_xpath("/html/body/div[2]/main/nav/div[2]/a")
            element_nextPage.click()
            time.sleep(2)
```

Figure 5.13 : The main python functions of the web scrawlers

With the python script implemented by us, we can extract and save the useful text information into a more structured Json format, which means within a single Json file, all the meaningful text contained in the whole book can be retrieved, also according to different types of tags.

Here is an example of a part of the result Json file of the book :

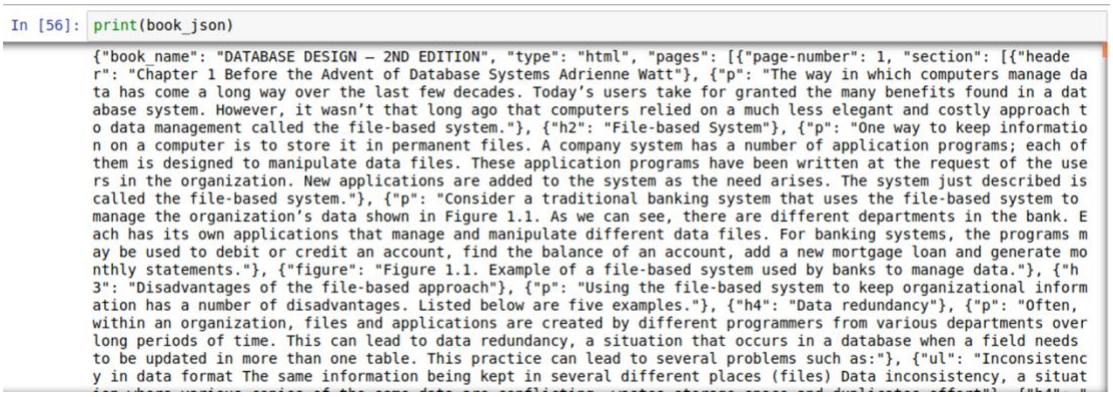

Figure 5.14 : Result json format of book scrawled from the HTML version on line.

Then, we will introduce the way how did we extract text information from the pdf version.

```
import tika
from tika import parser
```

```
%%time
parsed = parser.from_file('calculus-volume-1-6.4.pdf')
print((parsed["content"]))
```

```
2020-02-29 13:50:29,595 [MainThread  ] [INFO ]  Retrieving http://search.maven.org/remotecontent
he/tika/tika-server/1.22/tika-server-1.22.jar to /tmp/tika-server.jar.
```

Figure 5.15 : Using Tika to extract text from pdf version of books on line.

For the extraction of plain text from the pdf version of the book, a python library called Tika has been used in our project. Apache Tika uses existing parsing libraries to detect and extract metadata and structured content from documents in different formats (such as HTML, PDF, Doc). After obtaining the text information of the book using web crawlers and Tika, the next step is also to use different extractors to extract named entities from the text. With a single book, we can achieve a 60% recall of the database domain specific named entity set manually collected previously by our colleges. And if there is some way to implement a generalized web crawler of on-line books someday, a maximized progress of recall can be expected using this heuristic method.

Here is part of the final named entity extraction script we implemented with exist popular extractors :

```
EXTRACTORS = [
    dandelion.DANDELION(credentials_apis['dandelion'][0]),
    babelfy.BABELFY(credentials_apis['babelfy'][0]),
    textrazor.TEXTRAZOR(credentials_apis['textrazor'][0]),
    meaning_cloud.MEANINGCLOUD(credentials_apis['meaning_cloud'][0]),
    #opencalais.OPENCALAIS(credentials_apis['opencalais'][0]),
    #fox.FOX(),
    spacyner.SPACYNER(),
    adel.ADEL()
]

def getEntities(text,extractors=EXTRACTORS,lang='en',credentials_apis=credentials_apis,credential_index=credential
    ensemble_response = {'text':text,'entities':{}}
    for ext in extractors:
        flag = True
        while flag:
            try:
                ext.extract(text,lang=lang)
                flag = False
            except Exception as e:
                global ERR
                ERR = e
                credential_index[ext.name] += 1
                if credential_index[ext.name] >= len(credentials_apis[ext.name]):
                    credential_index[ext.name] = 0
                #if credential_index[ext.name]
                ext.change_credendials(credentials_apis[ext.name][credential_index[ext.name]])
                print(ext.name)
                print(e)
```

```
def getURIsfromEntities(text,extractors=EXTRACTORS,lang='en'):
    uris = set()
    count = 1
    ensemble_response = getEntities(text,extractors,lang)
    if ensemble_response:
        for ext in ensemble_response['entities']:
            for e in ensemble_response['entities'][ext]:
                if 'wikidataUri' in e:
                    uris.add(e['wikidataUri'])
        uris = uris - {''}
    return uris
```

```
In [3]: getURIsfromEntities('today we will talk about SVM and PCA')
Out[3]: {'http://www.wikidata.org/entity/Q282453',
         'http://www.wikidata.org/entity/Q2873'}
```

Figure 5.16 : Part of python script of using extractors to extract named entities.

The example above can show us that with the extractors, we can collect a set of named entity URI from knowledgebase like Wikidata, which will be merged into the whole domain specific context of a specific field or subject.

5.2.3 MediaWiki API to retrieve domain specific context

As introduced in Chapter 4, MediaWiki is a tool to get touch to a lot of different kinds of Wikipedia resources which are contained in the knowledgebase of Wikidata. Here is the official site of Wikidata knowledge base, from which one can view and utilize many features and properties of Wikipedia.

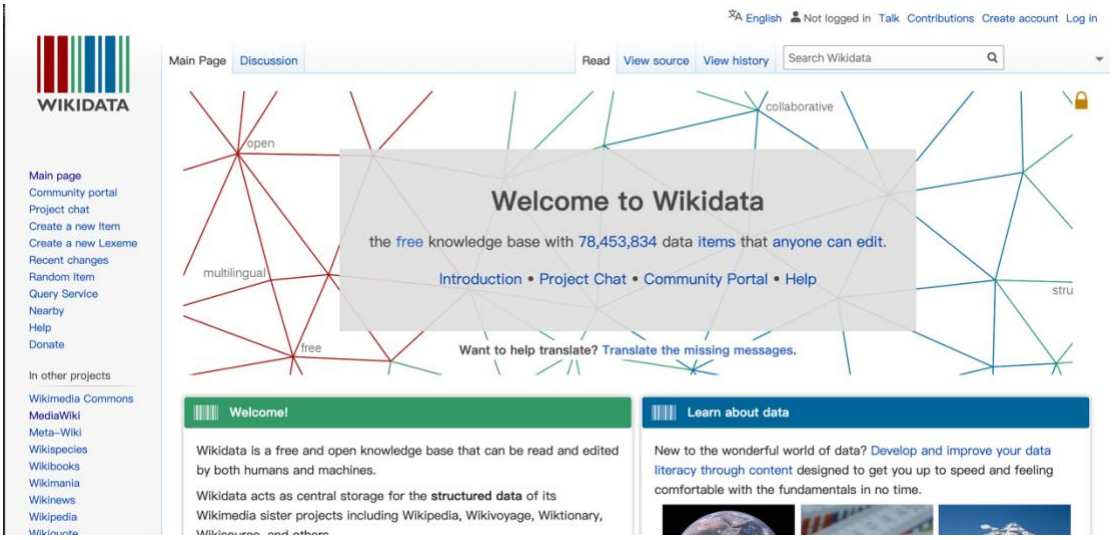

Figure 5.17 : Home page of Wikidata.

Especially with the search box on the up right, one can search a query to retrieve a list of named entities which are most related to the text of the query.

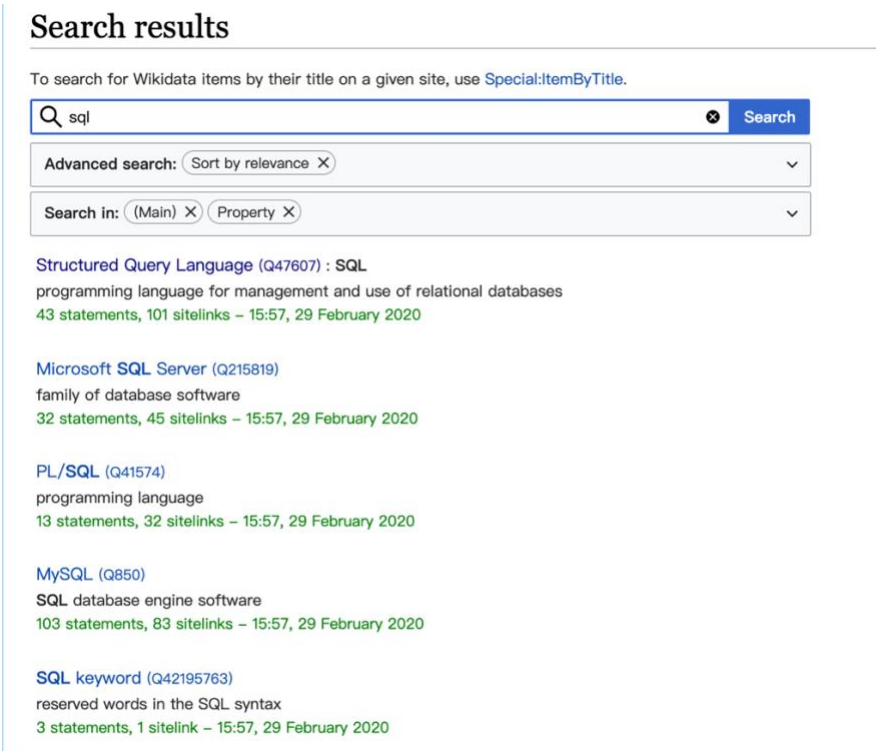

Figure 5.18 : Result list of named entities searching in Wikidata.

From the figure above we can see a part of the result list searching a query of 'sql', which is the name of a basic programming language of database field. And from the result list we can find that this list can be a really high quality resource of named entity set collection for a specific domain, but there should be some manual design of the queries we search and whether there is some way of doing this procedure within python script for the convenience of being a composition of the whole project.

Then after spending some time on investigation, we found that there was a way to use Wikipedia API to retrieve the same search result as in Wikidata search box.

```
S = requests.Session()

URL = "http://www.wikipedia.org/w/api.php"

SEARCHPAGE = "sql"

PARAMS = {
    "action": "query",
    "format": "json",
    "list": "search",
    "srsearch": SEARCHPAGE,
    "srlimit":500
}

R = S.get(url=URL, params=PARAMS)
DATA = R.json()

labels = []
for i in range(len(DATA['query']['search'])):
    print(DATA['query']['search'][i]['title'])
    labels.append(DATA['query']['search'][i]['title'])
```

Figure 5.19 : Using MediaWiki API with python.

Then, we can get the labels of the search result entities, the maximum number of result entities is 500 :

```
SQL
Microsoft SQL Server
MySQL
NoSQL
SQL injection
History of Microsoft SQL Server
SQL Server Express
Join (SQL)
PL/SQL
SQL Server Management Studio
SQL Server Reporting Services
SQL Server Integration Services
Comparison of relational database management systems
Transact-SQL
Data definition language
PostgreSQL
SQL Server
Select (SQL)
Oracle SQL Developer
```

Figure 5.20 : Part of result list of labels searching 'SQL' using Media WikiAPI

As we can see in the figure above, the labels of the result entities are really with high quality, which are some of the keywords most related and important to the domain of database. Also we can get the Wikidata IDs of the result entities :

```
Q47607
Q215819
Q850
Q82231
Q506059
Q28455835
Q1012765
Q2003535
Q41574
Q4048883
Q846897
Q1231051
Q60485
Q1411245
Q1431648
Q192490
Q400708
Q1164001
Q931351
```

Figure 5.21 : Part of result list of entity IDs searching 'SQL' using MediaWiki API

Also make the labels and IDs together, and can be saved for later use :

```
]: for v in zip(labels,list_id):
       print(v)
```

```
('SQL', 'Q47607')
('Microsoft SQL Server', 'Q215819')
('MySQL', 'Q850')
('NoSQL', 'Q82231')
('SQL injection', 'Q506059')
('History of Microsoft SQL Server', 'Q28455835')
('SQL Server Express', 'Q1012765')
('Join (SQL)', 'Q2003535')
('PL/SQL', 'Q41574')
('SQL Server Management Studio', 'Q4048883')
('SQL Server Reporting Services', 'Q846897')
('SQL Server Integration Services', 'Q1231051')
('Comparison of relational database management systems', 'Q60485')
('Transact-SQL', 'Q1411245')
('Data definition language', 'Q1431648')
('PostgreSQL', 'Q192490')
('SQL Server', 'Q400708')
('Select (SQL)', 'Q1164001')
('Oracle SQL Developer', 'Q931351')
```

Figure 5.22 : Part of result list of labels and IDs searching 'SQL' using MediaWiki

After combining all the three named entity resources together, finally we reach a recall of 98% on the named entity set on domain 'database' pre-collected by our colleges. And this heuristic method can be easily generalized to other domains or subjects to collect a contextual named entity set for the later use of down streaming NLP tasks, such as domain specific NER or NED.

Chapter 6

Conclusions and future work

The contribution of the work in this thesis are mainly in two different parts. The first is to find a temporary feasible way of tackling the problem of misspelling, and the second is to find a way to automatically collect the context of a specific domain or subject, which means a set of named entities related to a specific domain.

For the first part of our work about fuzzy search, actually the current solution should be temporary but acceptable, since the tri-gram analyzer works in an relative acceptable way of getting results within a certain Levenshtein distance from the original query, and after a few hours of indexing the DBpedia multilingual labels, each search process cost a time duration only in milliseconds, which means to cope with every word in a single transcript it won't cost a really long time. But there must be a lot of future work can be tried in this part. Since at the first of the work, we also spent a lot of time on implementing a custom relevance score to overcome the limit of the maximum Levenshtein distance 2 in fuzzy search of ES, and we temporarily failed to reach the point in a limited time. But if later on we become more familiar with ES, more research works on ingest API and painless scripting language in ES may lead us to the final target of the customized score more consistent with our ideas.

For the second part of our work about contextual learning of a specific domain. First, the method proposed to automatically learning the context can be future improved. There is a massive improvement can be reached whenever there is a way of creating some kind of general we scrawler which can scrawl different web sites or books in HTML version with different HTML structures. Then not only Coursera but other open course web sites like Udacity, and all other books automatically discovered in HTML version can be parsed and much more useful information can be extracted automatically. In addition, the way of leveraging the result of the set of domain specific named entities can be a much more interesting and challenging task to be done for the down streaming NLP tasks such as NER or NED. For example, we can use the collected set of domain specific named entities as a filter which can help the decision of whether the mention is a named entity and how much does a named entity in knowledge base match the mention appears in the text. But there is still the problem of this collection procedure which will produce so many not domain relevant named entities that can be seen as noise.

As a result, the future work can be biased to expending the leverage of ES when there is more need of coping with misspelling problem. And may also be biased to the utilization of the collected set of domain specific named entities for the named entity disambiguation work to annotate video lecture transcripts.

Acknowledgement

Here I want to express my gratefulness to my parents who support nearly every one of my own choices without condition. And I want to express my gratefulness for my friends both in China and Italy, who always give me any kind of help when I need. And for the work of this thesis, I'm really grateful to Professor Laura who gave me the chance to do some research in the field that I'm interested in. And especially very thankful to my college and friend Lorenzo, since he nearly did all the things he could to make this journey of research well-being and made me see more possibilities of what is a happy life.

Reference

[1] BERNERS-LEE, TIM, et al. “THE SEMANTIC WEB.” Scientific American, vol. 284, no. 5, 2001, pp. 34–43. JSTOR, www.jstor.org/stable/26059207. Accessed 17 Feb. 2020.

[2] "World Wide Web Consortium (W3C), "RDF/XML Syntax Specification (Revised)", 10 Feb. 2004"

[3] "World Wide Web Consortium (W3C), "OWL Web Ontology Language Overview", W3C Recommendation, 10 Feb. 2004"

[4] Nadeau D, Sekine S. A survey of named entity recognition and classification[J]. Lingvisticae Investigationes, 2007, 30(1): 3-26.

[5] Chiu J P C, Nichols E. Named entity recognition with bidirectional LSTM-CNNs[J]. Transactions of the Association for Computational Linguistics, 2016, 4: 357-370.

[6] Bunescu R, Pasca M. Using encyclopedic knowledge for named entity disambiguation[J]. 2006.

[7] Honnibal M, Dale R. DAMSEL: The DSTO/Macquarie System for Entity-Linking[C]//TAC. 2009.

[8] Taylor Cassidy Z C, Artiles J, Ji H, et al. Cuny-uiuc-sri tac-kbp2011 entity linking system description[C]//Proceedings Text Analysis Conference (TAC2011). 2011.

[9] Kulkarni S, Singh A, Ramakrishnan G, et al. Collective annotation of Wikipedia entities in web text[C]//Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. 2009: 457-466.

[10] Han X, Sun L, Zhao J. Collective entity linking in web text: a graph-based method[C]//Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval. 2011: 765-774.

[11] Lehmann J, Isele R, Jakob M, et al. DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia[J]. Semantic Web, 2015, 6(2): 167-195.

[12] Auer S., Bizer C., Kobilarov G., Lehmann J., Cyganiak R., Ives Z. (2007) DBpedia: A Nucleus for a Web of Open Data. In: Aberer K. et al. (eds) The Semantic Web. ISWC 2007, ASWC 2007. Lecture Notes in Computer Science, vol 4825. Springer, Berlin, Heidelberg

[13] Vrandečić D, Krötzsch M. Wikidata: a free collaborative knowledgebase[J]. Communications of the ACM, 2014, 57(10): 78-85.

[14] Nicolas Heist, Sven Hertling, Daniel Ringler, Heiko Paulheim. Knowledge Graphs on the Web -- an Overview arXiv:2003.00719

[15] https://lucidworks.com/post/full-text-search-engines-vs-dbms/

[16] https://qbox.io/blog/elasticsearch-as-a-high-performance-full-text-search-engine

[17] https://www.ibm.com/developerworks/cn/java/j-lo lucene1/index.html

[18] https://lucene.apache.org/core/3_6_2/fileformats.html