Natural Language Processing and Information Retrieval with Applications in Social Networks Midterm Project

Abdul Gafar Manuel Meque
Institute of Information Science / Address line 1
Nataional Chengchi University / Address line 2
Taiwan International Graduate Program / Address line 3
ameque@iis.sinica.edu.tw

Abstract

The current project is comprised of four parts, namely the installation of a search engine (IR) system, modification of said IR system to improve or change the ranking of search results, test it on a designated dataset, at last a extra section for user interface for searching, focused on relevance feedback. In this approach to tackle the requirements, from the suggested Systems, Haystacksearch, was chosen is implement and used as is, a process requiring a minimal effort on configurations. to further comply with the requirements, some changes where made in both the scoring and ranking modules, and later on integrated with an improved user interface for the results.

1 Introduction

according to (Salton, 1968), *Information Retrieval* is a field concerned with the structuring, manipulation (from storing,organizing to accessing) of information. Over the years many approaches to IR have been proposed with great success, most of them focusing on web search engines. Here I present the Haystack, a system that is tailored for web, relying on the django platform for its web capabilities and an independent search engine. For this particular project haystack is implemented using Whoosh, a symple and easy to setup search engine as the back-end for django-haystack.

2 Haystack Setup and Installation

The installation and configuration of Haystack and whoosh is described at (Lindsley, 2016), a simple process that requires only a single line of shell command each using pip, any python package would the same way. for the haystack, just install the django-haystack package

```
$ pip install django-haystack
```

for the whoosh back-end engine, similarly installed a package

```
$ pip install Whoosh
```

After the packages are installed, the next step was to create a django website, its was accomplished with the following shell commands:

```
$ django-admin startproject ir_project
```

This command created a folder named ir_project, containing another folder with the same name and a manage.py file(is just a command-line utility to allow us interact with this Django project), the inner ir_project directory is the Python package for the project. The next step was the creationg of the actual search application, by using the following command:

```
$ python manage.py startapp ir_search
```

this command in its turn, created a folder named ir_search, with all the relevant/basic Python files the web app.

To allow the interaction amongst Whoosh, haystack and django framework, the following modification of the files generated by the previous command were made:

- change the settings.py
  - adding ”haystack” to the INSTALLED_APPS list

    - adding HAYSTACK_CONNECTIONS variable, point to whoosh engine
- change the models.py file
    - creating a News (Model) class to represent the document name, title and body properties to the Model.
    - live

After this processes are done, we have our app, functional but not doing the search, for that, in the next section the indexing process and searching is described.

3 Indexing and Searching with Haystack Whoosh

3.1 Creating Search Indexes

In Haystack, to determine which data goes to the search index and handles we need need to create SearchIndex objects. they are basically Django models that can be manipulated using fields.

In our implementation there is only one Model, the News model, so only one search index is thus created. To build a SearchIndex, we need to subclass both indexes.SearchIndex and indexes.Indexable, and also define the fields to store data with and define a get_model method. Inside the search_indexes.py file inside the search_ir a NewsIndex class is created, taking into account the structure of the News model. Placing the in that file makes it intuitive for haystack to locate it.

```
import datetime
from haystack import indexes
from search_ir.models import News


class NewsIndex(indexes.SearchIndex,
    indexes.Indexable):
     text = indexes.CharField(document=
    True, use_template=True)
     name = indexes.CharField(model_attr=
    'name')


     def get_model(self):
```

search.indexes.py

Note: Each SearchIndex should have one field set to document document=True, to tell haystack and the backend search engine where to search first.

The SearchIndex file also contains the use_template=True on the text field attribute, which tells haystack that it should look for a template for building the News documents to be indexed by Whoosh. The template is located inside your template directory called search/indexes/search_/news_text.txt and it has the following content:

```
{{ object.name }}
{{ object.text }}
```

newstext.txt

this tells that only that our search result object will only contain two values each, since only those two fields from the documents are considered.

The final step in this process was to build the actual indexes by running the command:

```
$ python manage.py rebuild_index
```

3.2 Creating the search UI

For the search view, an html template is created, to display the search box and populate the search result, the basic and standard for demonstration template looks like the following: the first lines are creating a simple search form with only one input box and a search button.

```
    <h2>Search</h2>
    <form method="get" action=".">
        <table>
            {{ form.as_table }}
            <tr><td>&nbsp;</td>
                <td>
                    <input type="submit"
 value="Search">
                </td>
            </tr>
        </table>
```

search.html

the next few lines are used to get and display search results if any. It should be noted that the result.object.name uses the actual News object returned from the search in and gets its name field.

```
        {% if query %}
            <h3>Search Results</h3>
            {% for result in page.
    object_list %}
                <p>
                    <a href="{{ result.
    object.get_absolute_url }}">{{
    result.object.name }}</a>
                    <a href="{{ result.
    object.get_absolute_url }}">{{
    result.score }}</a>

                </p>
            {% empty %}
```

search.html

the last relavant part of this template is the previous and next links/buttons that are displayed at the bottom of the page, if there is any next or previous page, that is, when the search results are more than the max result per page defined.

```
                {% endfor %}

                {% if page.has_previous or
    page.has_next %}
                    <div>
                        {% if page.
    has_previous %}
                        <a href="?q={{ query
     }}&amp;page={{ page.
    previous_page_number }}">
                            {% endif %}&
    laquo; Previous{% if page.
    has_previous %}</a>{% endif %}
                        |
                        {% if page.has_next
    %}
                        <a href="?q={{ query
     }}&amp;page={{ page.
    next_page_number }}">
                            {% endif %}Next
    &raquo;{% if page.has_next %}</a>{%
    endif %}
```

search.html

After the template search view template the next step was configuring the view module, the view module was updated to have the following data:

```
from django.shortcuts import render
#Search views
from django.views.generic import
    TemplateView

class SearchView(TemplateView):
    template_name = 'search_ir/search.
    html'
search_view = SearchView.as_view()
```

views.py

In this initial implementation there is only one view, so that is the only configuration that needed to be mapped. After mapping the view to the template then the urls.py was updated to specify the url patern for the search view.

```
from django.conf.urls import patterns,
    include, url
from views import search_view

urlpatterns = patterns('',
    url(r'^search$', search_view, name='
    search_view'),
)
```

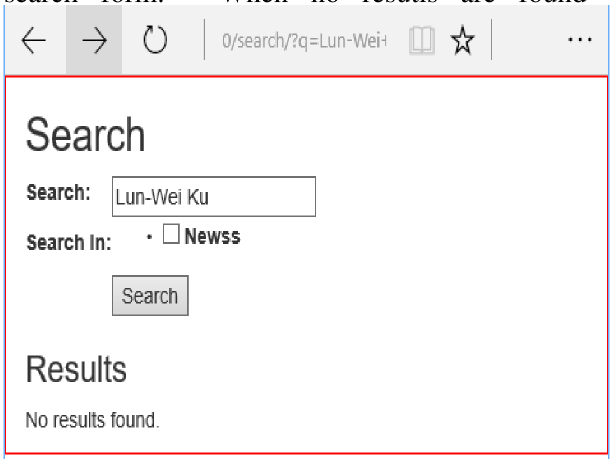
urls.py

3.3 Running The Search WebApp

To run the search app, all that is required is to execute the command

```
$ python manage.py runserver
```

by running this command the web app is deployed and a browser page opens with the search form. When no resutls are found



Search

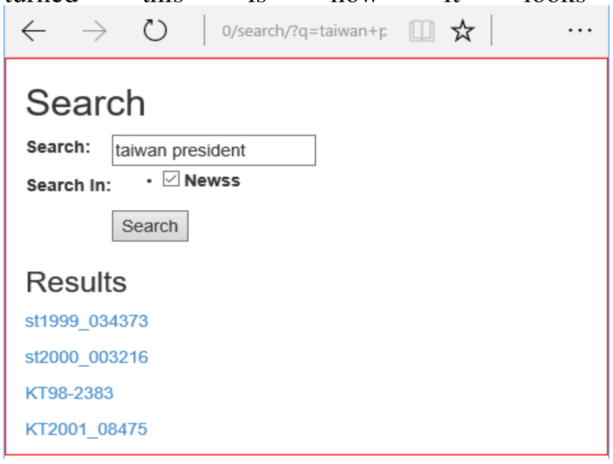
Search: Lun-Wei Ku

Search In: • ☐ Newss

Results

No results found.

when some results are returned this is how it looks



Search

Search: taiwan president

Search In: • ☑ Newss

Results

st1999_034373

st2000_003216

KT98-2383

KT2001_08475

4 Retrieval Result Improvement and Relevance Feedback

For search result improvement and relevance feeback aspects of the project the following modifications were made:

4.1 Retrieval Results Improvement

Replace the use of the default BM25F describe in (Robertson and Zaragoza, 2009) algorith used in whoosh search engine, with the PL2 described in (Harter, 1974) algorithm, which is also based on the divergence from randomness framework. The following image shows the BM25F python code

```python
def bm25(idf, tf, fl, avgfl, B, K1):
    # idf - inverse document frequency
    # tf - term frequency in the current document
    # fl - field length in the current document
    # avgfl - average field length across documents in collection
    # B, K1 - free paramters
```

scoring.py

Now the search engine's scoring mechanism relies on the following function

```python

# PL2 model

rec_log2_of_e = 1.0 / log(2)


def pl2(tf, cf, qf, dc, fl, avgfl, c):
    # tf - term frequency in the current document
    # cf - term frequency in the collection
    # qf - term frequency in the query
    # dc - doc count
    # fl - field length in the current document
    # avgfl - average field length across all documents
    # c -free parameter

    TF = tf * log(1.0 + (c * avgfl) / fl)
    norm = 1.0 / (TF + 1.0)
    f = cf / dc
    return norm * qf * (TF * log(1.0 / f)
```

scoring.py

4.2 UI and Relevance Feedback

The concept behind the user interface in this project is simplicity, so the basic search box an autocomplete, the "More Like This" also know as "Simmilar Results", capabilities are implemented. For the search results, from the simple list of file names of documents matching the query, another simple, yet more informative list contain the names of the document and a slice of it containing the query keywords.

The following piece of code adds the autocomplete capability:

```python
def autocomplete(request):
    sqs = SearchQuerySet().autocomplete(content_auto=request.GET.get('q', ''))[:5]
    suggestions = [result.name for result in sqs]
    # Make sure you return a JSON object, not a bare list.
    # Otherwise, you could be vulnerable to an XSS attack.
    the_data = json.dumps({
```

views.py

a corresponding customized template tag receives the result and on the client side using javascript.

For more like this and result highlight, those capabilities are provided within haystack itself, minor changes where required, like the two lines of template language bellow:

```
{% highlight result.object.text with query max_length 100 html_tag "em" css_class "highlight_me_please" %}
{% more_like_this result.object.text as related_content for "result.object.text" limit 5 %}
```

search.html

5 Conclusion

The final result of all implementations and changes for improvements made in this project can be seen in the final demo search web app, that provides all required aspects from scoring / ranking change, a friendly user interface, all taking into account relevance feedback.

References

S.P. Harter. 1974. A probabilistic approach to automatic keyword indexing.

Daniel Lindsley. 2016. Getting started with haystack.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April.

Gerard. Salton. 1968. *Automatic Information Organization and Retrieval.* McGraw Hill Text.