A vertical search engine

Name: Michael Ajao-Olarinoye
SID: 10118047
Email: Olarinoyem@uni.coventry.ac.uk
CEM7071-information retrieval course work
Data science and computational intelligence

Introduction

This report shows the process involved in building and designing a vertical search engine. This search engine is one that is built for graduate students who wants to perform a type of research and need to see which professor and schools offer the research. The student would have to search for the research that they want to do and the search engine would check the universities available with the professors that are doing the type of research and show the result with how to contact them, either via email or going directly to their website.

The first question that comes to mind is what a vertical search engine is. The Vertical Search Engine is a device that targets a given area. This expands and subdivides conventional search engines that can gather that incorporate unique knowledge on the Web (Li, Zhao and Huang, 2010). There is a different type of search engines over the world, but the one designed in this report crawls three different schools staff web page and gets the required information.

In the field of information retrieval, it is highly required to know how to get the required data that you need and not to pull the unnecessary once. To be effective and efficient in retrieving data from the different website while not wasting various required resources. In this report, I will also build a text classification application that would also use the dataset gotten from the retrieval of the lecturers' data. I will discuss how I went about to build the following in this report:

- Crawler
- Indexer
- Query processor
- Text classification

First of I had to design the process I want to take in building a vertical search engine that crawls a few school websites and for security and ethical reasons, I had to check if I was allowed to crawl the information that I required by using Robot.txt on each website.

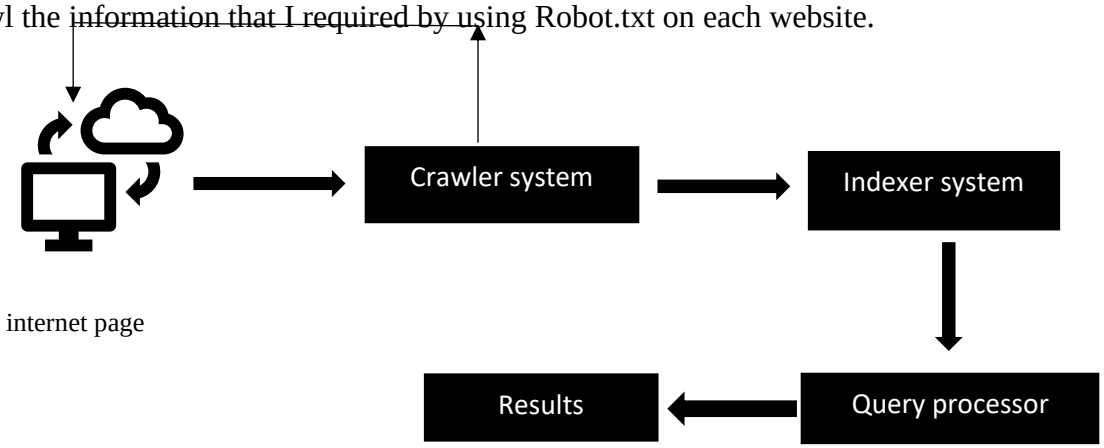


fig1: the workflow diagram of the vertical search engine built

In this project, I used Google colab, and libraries like pandas, nltk, glob, os, and request. I built the search engine using functions in python. The three systems were built to incorporate with each other. Python is a programming language that gives access to a different library to use for crawling the web and building the search engine. The following schools were scrapped:

- University of Wolverhampton - https://www.wlv.ac.uk/about-us/our-staff/?f.A-Z%7CAllDocumentsFill=All&f.School%7CS=School+of+Engineering&f.School%7CS=School+of+Mathematics+and+Computer+Science&form=wlv-facet&query=&collection=wlv-web-our-staff&start_rank=41
- Anglia Ruskin University - https://aru.ac.uk/science-and-engineering/faculty-staff?page=1
- Imperial College London - https://www.imperial.ac.uk/computing/people/academic-staff/

The picture below shows the libraries that were imported to be used for the crawling, indexing and query processor. The libraries will run and import the required codes for building the search engine, each line and steps will be explained further. First, of I will start with crawling and explain the code written to crawl one of the schools, university of Wolverhampton

```
# import all required library
import requests
from bs4 import BeautifulSoup
import csv
import pandas as pd
import os
import glob
os.chdir("/content/drive/My Drive/crawler")
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
import pandas as pd
import numpy as np
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
nltk.download('stopwords')
nltk.download('punkt')
from nltk.corpus import wordnet
from nltk import pos_tag
from nltk.stem import WordNetLemmatizer
import string
```

CRAWLER

Web crawlers are computer programs that search the internet, and 'read' what they find. But in our case we will build our crawler system ourselves using python that will take out the information that is required from each website by studying the Dom of the website (working with HTML), using a library called beautiful soup. Beautiful Soup is a Python library that draws data from both HTML and XML formats. It works with your favourite parser to include idiomatic ways to browse, search, and change the parser tree. Beautiful Soup parses everything that you offer it and does the traversal stuff for you. You might say "Find all the links or classes" by using the find and select function of the beautiful soup library (Richardson 2020).

Our web browser submits to a Web server anytime we access a Web page. This request is called a GET script because we get server files. The server will then return files telling our browser how to render the page for us (Vik Paruchuri 2016).

The image below shows the first function created to crawl around the URL given to it which is the Wolverhampton staff page. The first line of code is the code that creates the CSV file

for Wolverhampton and stores the required rows, that will be crawled from the webpage. The function create_wolv pages the number of pages and loops through all the pages that show the lectures and get all the URL associated with each lecturer. Beautiful soup is called, and the URL is requested using the request library, and gotten and passed with an Html.parser to covert the result into text. The result is the text of the Html structure of the webpage.

```
# create a csv with the following rows to save the crawled files
f = csv.writer(open('wolv.csv', 'w'))
f.writerow(['Name', 'email', 'link', 'research'])

#function that crawls and gets the link to each lecturer individual pages
def create_wolv(max_pages):
    page = 1
    while page <= max_pages:
        URL = "https://www.wlv.ac.uk/about-us/our-staff/?f.School%7CS=School+of+
        source_code = requests.get(URL)
        plain_text = source_code.text
        soup = BeautifulSoup(plain_text, 'html.parser')
        for link in soup.select('h4 a'):
            href = link.get('title')
            #print(href)
            wolv_lect(href)
        page += 1
```

The for loop looks through all the links on the pages, the soup. Select points to the (“h4 a”) the anchor tag of all the h4 tags of the Html to get the href. The href was name title on the webpage and for a test, I will crawl just 1 page which is the first page.

The next function is the function that pulls the name, email, job title, and the research tab that grabs the text on the webpage that talks about each lecturer research focus. Each name and other columns were gotten to by targeting the corresponding Html tag of each and get just only the text. The Try function loops through and finds all the selected tag and gets the text and input it into each corresponding row in the CSV. The function is passed in the other function and the href is passed as an argument.

The image below shows the second function that was built to crawl each staff webpage and saves it into a CSV file format. The method was also used to get the data from the other two universities and saved in all their respective CSV files. With each university crawled using the beautiful soup library.

```
# function that crawls the each indivial pages to get the details required
def wolv_lect(item_URL):
    source_code = requests.get(item_URL)
    plain_text = source_code.text
    soup = BeautifulSoup(plain_text, 'html.parser')

    for details in soup.find_all(class_='content_wrap'):
        try:
            name = details.h2.text.strip()
        except Exception as e:
            name = None
        #print('Name:', name)

        try:
            title = details.h3.text.strip()
        except Exception as e:
            title = None
        #print('Job title:', title)

        try:
            email = details.span.a.text.strip()
        except Exception as e:
            email = None
        #print('Email:', email)

        try:
            research = soup.select('.staff_det li span p')[0].text
        except Exception as e:
            research = None


        f.writerow([name, email, item_URL, research]) # saving the files
create_wolv(50) # number of pages to loop through
```

All the data gotten is saved into a CSV and I got three CSV files with each corresponding row from all the universities.

```
path='/content/drive/My Drive/crawler' # the path link to my saved files in Colab

all_files = glob.glob(os.path.join(path, "*.csv"))
all_df = []
for f in all_files:
    df = pd.read_csv(f, sep=',')
    df['file'] = f.split('/')[-1]
    df['ID'] = [x for x in range(1, len(df.values)+1)]
    all_df.append(df)

merged_df = pd.concat(all_df, ignore_index=True, sort=True)
```

The image below is the code that loops through the path where all the CSV files are stored using the glob and panda's library. The code merges all the rows of the files and converts it or append into a panda's data frame and also create a file and ID column that show where the files were gotten from and the count that would be required later on for the preprocessing of the indexer and query processor.

```
merged_df = merged_df.astype(str) # converting the datatype to string to avoid future errors in the code
merged_df
```

	ID	Name	email	file	link	research
0	1	Professor Phil Cox	P.W.Cox@wlv.ac.uk	wolv.csv	https://www.wlv.ac.uk/about-us/our-staff/phil-...	nan
1	2	Dr Desmond Case	D.Case@wlv.ac.uk	wolv.csv	https://www.wlv.ac.uk/about-us/our-staff/desmo...	nan
2	3	Martin Eason	M.Eason@wlv.ac.uk	wolv.csv	https://www.wlv.ac.uk/about-us/our-staff/marti...	Design, Materials, Manufacturing
3	4	Muhammad Sayed	M.B.H.Sayed@wlv.ac.uk	wolv.csv	https://www.wlv.ac.uk/about-us/our-staff/muham...	Electronic System Design
4	5	Mr Carl Wilding	C.Wilding@wlv.ac.uk	wolv.csv	https://www.wlv.ac.uk/about-us/our-staff/carl-...	nan

The image above shows the result of the columns gotten from the data frame after converting the data type of each row into strings for ease of processing it later, by changing it with the astype function. The data gotten is 6 columns and 430 rows data frame. It is passed into the indexer as the data frame got.

INDEXER

In Information Retrieval (IR) systems indexing is an essential method. This forms the central functionality of the IR process as it is the first step in IR and helps in the efficient retrieval of information. Indexing restricts the records to the insightful words they hold. This includes a map of the terms and conditions to the relevant documents comprising them. Indexing continues at four stages, including material creation, text tokenization, text terms collection, and index building. The index can be contained in the form of various data structures, including direct index, index of records, lexicon and index inverted. The index may be generated by implementing various algorithms or schemes such as in-memory single-pass indexing, blocked indexing (H. Kaur and V. Gupta, 2016).

Text indexing consists of four stages: defining and handling corpus documents, reading or tokenizing documents, manipulating document words to construct a logical image, and creating index or database data structures (Ounis et al., 2006). Tokenization eliminates the punctuation

characters from the text and is Language-Based, Tokenization parses the documents that split them down to different parts, in general terms (Schütze; Manning and Raghavan, 2008). The function below shows the tokenization process.

```
# this function is created to convert all the text to lower text and to remove punctuation to avoid it to not be case sensitve when searching
def process_string(text):
  text = text.lower() #to lowercase
  text = text.translate(str.maketrans('', '', string.punctuation)) #strip punctuation
  return text
```

The function above is built to process data and change the passes text from the data frame and converting into lower text and removing the punctuation. Which is the beginning process of the indexer that of the search engine.

Search engines prefer not to delete the stop words so the recall can diminish. Popular terms are of little to no useful interest and are thus omitted from the dictionary to index for memory saving. This move includes deleting growing and highly frequent phrases, which are the elimination of stop words

```
# function to tag first charater lemmatize accepts
def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}

    return tag_dict.get(tag, wordnet.NOUN)


stop = stopwords.words('english')
```

It is the key and final form of the data that holds a term's postings records. For what term, the list of posts contains the document I'd which correlates to the article containing the word frequency of the phrase in that article. The fields that include the words that also be stored in. When tokens are collected and terms avoid missing, stemming helps to tear the words down to their base types. To simplify the term to specific forms it "strips off" the prefixes and suffixes (generally suffixes).

```
words = entry.text.split(" ")
word = words[0]
sample = {word: [ID]}
print(sample)
```

```
{'muhammad': ['4']}
```

```
for word in words:
  if word in index_test.keys():
    if ID not in index_test[word]:
      index_test[word].append(ID)
  else:
    index_test[word] = [ID]


  print(index_test)
```

```
{'muhammad': ['4'], 'sayed': ['4'], 'electronic': ['4'], 'system': ['4'], 'design': ['4']}
```

Therefore, the output from the phrase pipeline is a list of terms or keywords that end up being placed in the database. The words will attain the properties of exhaustiveness and specificity where exhaustiveness is accomplished by deriving from a text a reasonably large number of definitions, and specificity is achieved by adding more insightful words or removing less insightful ones.

Ultimately, the inverted index is based on both index data structures utilizing indexing algorithms. The Direct Index is the simplest version of an index and stores the words and the term frequency of the terms contained in the paper. It is built to allow the database expansion simple and effective and to be used when clustering the array of documents. Therefore, the direct index stores the term I do, the term frequency, the fields in which the words appear, and the block I would if used. Taking the simplest form for the index is the approach I took and built various functions to pass the data and create an inverted index. The codes in all the images in this section show how the index is built. The images below show the index built bypassing as argument the entry which is the list of the data frame and the index which was gotten from the words ID after processing the data in the indexer.

```
def index_it(entry, index):
  words = entry.text.split()
  ID = entry.ID
  for word in words:
    if word in index.keys():
      if ID not in index[word]:
        index[word].append(ID)
    else:
      index[word] = [ID]
  return index
```

```
def index_all(df, index):
  for i in range(len(df)):
    entry = df.loc[i, :]
    index = index_it(entry, index)
  return index
```

```
def build_index(df, index):
    to_add = transform_df(df)
    index = index_all(df = to_add, index = index)
    return index
```

```
index = build_index(df = merged_processed, index = {})
index
```

```
 'annual': ['2', '3', '4', '7', '8', '18', '19'],
 'social': ['2', '3', '5', '8', '9', '11', '13', '17', '20'],
 'evolutionary': ['2', '20'],
 '22': ['2'],
 'pp983996': ['2'],
 'doi101111j14209101200901706x': ['2'],
 'ingstc': ['2'],
 'bascompte': ['2'],
 'bluthgen': ['2'],
```

QUERY PROCESSOR

The query supports and format passed into the index, the query processor calls for the index to retrieve from all the functions created above and searches to find the exacts words related. It is a simple process that involves an input search for box and processes the passed with the function created called process_string that convert words by removing the punctuation and converting the keys to lower text and retrieves the required data frame by checking the ID against the index of the query retrieved and prints the result. The printed result is the information got that completes the process of the search engine.

```
# Query Processor
irosyndex = index
query = input("Search for:")
query = process_string(query)
retrieved = irosyndex[query]
merged_df[merged_df["ID"].isin(retrieved)]
```

```
arch for:
```

The image above shows A live search bar and when a search word like in this case “data” is inputted the result is shown below by displaying the lecturers that have close to inputted work in the index.

```
# Query Processor
irosyndex = index
query = input("Search for:")
query = process_string(query)
retrieved = irosyndex[query]
merged_df[merged_df["ID"].isin(retrieved)]
```

```
Search for:data
```

	ID	Name	email	file	link	
6	7	Dr Thomas Ings	thomas.ings@anglia.ac.uk	anglia.csv	https://aru.ac.uk//people/thomas-ings	ecolo
26	7	Dr Desmond Case	D.Case@wlv.ac.uk	wolv.csv	https://www.wlv.ac.uk/about-us/our-staff/desmo...	
291	7	nan	nan	imperial.csv	https://wp.doc.ic.ac.uk/szafeiri/	interes

Text classification (IMDB review dataset)

The text classification part is the second part of this project and I’ll be using the IMDB review dataset that has both positive and negative review. Text classification is one of the commonly employed frameworks for natural language processing in numerous business contexts. The dataset is supervised, I will be using a convolutional neural network trained on the data set to

perform the classification to see if one can predict if the review imputed is either positive or negative. For this part, I am also including it in the code and importing the required library to perform the text classification. The dataset contains 50K movie reviews that will be used for this project.

```
df = pd.read_csv("IMDB Dataset.csv")
df.head()
```

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive

The dataset is read into the data frame and checked with the head function. The data is split into two containers which are review in the review container and sentiment into the labels container, then the label is transformed using label encoder function. The label is converted into numbers positive (1) and negative (0). The code is shown in the figure below.

```
# container for sentences
reviews = np.array([review for review in df['review']])

# container for labels
labels = np.array([label for label in df['sentiment']])
```

```
# label encoding labels

enc = LabelEncoder()
encoded_labels = enc.fit_transform(labels)

print(enc.classes_)
print(labels[:5])
print(encoded_labels[:5])
```

```
['negative' 'positive']
['positive' 'positive' 'positive' 'negative' 'positive']
[1 1 1 0 1]
```

The data is split using train split function, with the test size been 33% of the dataset which is the reviews and encoded label. While also running a tokenization process on the word index, the training set is converted into word index by using the tokenization function, converting both the

training and testing dataset into sequence and pad sequence. The image below shows the code written to perform the process that was explained above.

```
# specific stopwords
specific_sw = ['br', 'movie', 'film']

# all stopwords
stopwords = stop + specific_sw
```

The code above creates the stopwords that are needed in the processing the dataset.

```
# train-test split
train_sentences, validation_sentences, train_labels, validation_labels = train_test_spl
```

```
vocab_size = len(word_index)
embedding_dim = 100
max_length = 120
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
```

```
# tokenize sentences
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_sentences)
word_index = tokenizer.word_index

# convert train dataset to sequence and pad sequences
train_sequences = tokenizer.texts_to_sequences(train_sentences)
train_padded = pad_sequences(train_sequences, padding=padding_type, maxlen=max_length)

# convert validation dataset to sequence and pad sequences
validation_sequences = tokenizer.texts_to_sequences(validation_sentences)
validation_padded = pad_sequences(validation_sequences, padding=padding_type, maxlen=ma
```

The neural network model which is a convolutional neural network was created using the following code, by initializing the model with a 5 model CNN model. The model will be complied using binary_crossentropy as the loss function and the optimizer as Adam and measured with Accuracy metrics and summarize the model that was created.

```
# model initialization
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    tf.keras.layers.Conv1D(128, 5, activation='relu'),
    tf.keras.layers.GlobalMaxPooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# compile model
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# model summary
model.summary()
```

The result gotten from the model is shown in the image below.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 120, 100)          17165900
_________________________________________________________________
conv1d (Conv1D)              (None, 116, 128)          64128
_________________________________________________________________
global_max_pooling1d (Global (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 24)                3096
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 25
=================================================================
Total params: 17,233,149
Trainable params: 17,233,149
Non-trainable params: 0
_________________________________________________________________
```

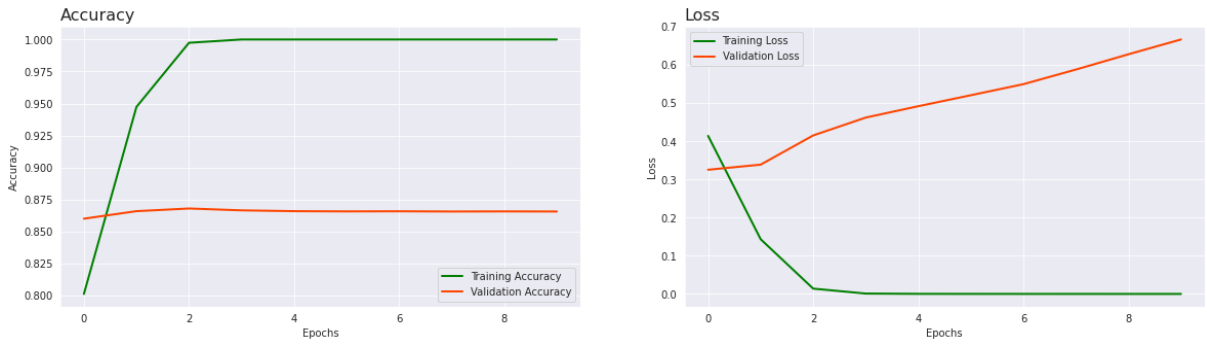
The model is fit to 10 epochs to train and was predicted to have gotten to the 100% accuracy by the 4 epoch while having a validation accuracy of 86%. The image below shows the code written to fit the model to the training data and predicted on the validation data.

```
# fit model
num_epochs = 10
history = model.fit(train_padded, train_labels,
                    epochs=num_epochs, verbose=2,
                    validation_split=0.33)

# predict values
pred = model.predict(validation_padded)
```

The code displayed in the image below is the code to create the function that plots the history of the model creating two graphs which are the accuracy and loss by plotting the epochs against both the accuracy and loss function of the created model.

```
def plot_history(history):

  plt.figure(figsize=(20, 5))

  plt.subplot(1, 2, 1)
  plt.plot(history.history['accuracy'], label='Training Accuracy', c='green', lw='2')
  plt.plot(history.history['val_accuracy'], label='Validation Accuracy', c='orangerec
  plt.title('Accuracy', loc='left', fontsize=16)
  plt.xlabel("Epochs")
  plt.ylabel('Accuracy')
  plt.legend()

  plt.subplot(1, 2, 2)
  plt.plot(history.history['loss'], label='Training Loss', c='green', lw='2')
  plt.plot(history.history['val_loss'], label='Validation Loss', c='orangered', lw='2
  plt.title('Loss', loc='left', fontsize=16)
  plt.xlabel("Epochs")
  plt.ylabel('Loss')
  plt.legend()

  plt.show()
```

The result of the model is shown in the graph below by showing both the accuracy and loss function of both the training and validation.



The code shown in the image below shows the testing of the text classification by predicting the label from inputted sentences like the once shown in the image below. The following code process completes the text classification of the IDMB dataset and from the test, it can be seen that the convolutional neural network performed well on the training dataset by predicting “The movie was

very touching and heart whelming" to be positive and "I have never seen a terrible movie like this" to be negative.

```
# reviews on which we need to predict
sentence = ["The movie was very touching and heart whelming",
            "I have never seen a terrible movie like this"]

# convert to a sequence
sequences = tokenizer.texts_to_sequences(sentence)

# pad the sequence
padded = pad_sequences(sequences, maxlen=max_length, padding=padding_type, truncating=t

# preict the label
print(model.predict(padded))
```

```
[[1.000000e+00]
 [1.162032e-04]]
```

This report contains the search engine built and the text classification using neural network and the link to the code is added to the appendix of this report.

Appendix

https://github.com/michaelajao/vertical-Search-engine/blob/master/search_engine.ipynb

https://colab.research.google.com/drive/1TQxtNOkfwRX5uLIYAW-LlAfgBwO5hWQX#scrollTo=HqKImpetFjmp

References

Li, M., Zhao, J., and Huang, T. (2010) Research and Design of the Crawler System in a Vertical Search Engine [online]: IEEE

Richardson, L. (2020) Beautiful Soup: We Called Him Tortoise Because He Taught Us. [online] Available from <https://www.crummy.com/software/BeautifulSoup/> [30 July 2020]

Vik Paruchuri (2016) Tutorial: Python Web Scraping Using Beautiful Soup – [online] Available from <https://www.dataquest.io/blog/web-scraping-tutorial-python/> [30 July 2020]

H. Kaur and V. Gupta (eds.) (2016) 2016 International Conference on Inventive Computation Technologies (ICICT). 'Indexing Process Insight and Evaluation

Ounis, I. A., Plachouras, G., He, V., and Macdonald, B. (2006) 'C. and Lioma, C. (2006) Terrier: A High Performance and Scalable Information Retrieval Platform'. Proceedings of ACM SIGIR-OSIR’06

Schütze, H., Manning, C. D., and Raghavan, P. (2008) Introduction to Information Retrieval [online]: Cambridge University Press Cambridge