# Shakespeare search engine and evaluation



---

## Abstract

In this report, I will explain how I made a search engine of Shakespeare.json dataset, and the method I evaluate it.

The conclution first, this search engine search for **keyword of text content** and return the **play names**, for example, search "hi" return "Hemlet". I use **MAP** to evaluate the search engine, the MAP value is **0.99 (stemming)** and **0.99(without stemming)**

## How to run

The system dependences are **NLTK** and **Ordered_Set**, the latter is a 3rd party library to generate a set without hashing the element, so the order of the elements are fixed, In order to run my code please use **pip install ordered-set**.

The data set Shakespeare_6.0.json is included in the project folder.

There are **2** scripts in the project folder: **pipeline.py** and **pipeline_stem.py**, simply run the scripts to get results, including **individual P@K value, AP for a query and MAP** for the search engine.

You are expected to see these messages while running the script.

> step 1...Read json file step 2...Indexing, It takes up to 5 minutes... step 3...Searching step 4...Evaluation

The mentioned values are coming in the step 4.

# Set up Elastic search

Here is how I set up elastic search before the project start.

1. Get data from the website, run

   `curl -L -o shakespeare.json <https://download.elastic.co/demos/kibana/gettingstarted/shakespeare_6.0.json>`

2. then put data to elasticsearch

   `curl -H 'Content-Type: application/x-ndjson' -XPOST 'localhost:9200/shakespeare/doc/_bulk?pretty' --data-binary @shakespeare_6.0.json`

3. Finish importing

| health | status | index | uuid | pri | rep | docs.count | docs.deleted | store.size | pri.store.s |
|---|---|---|---|---|---|---|---|---|---|
| yellow | open | shakespeare | 4RBYL8m1SQ20N38eUspeNg | 5 | 1 | 111396 | 0 | 21.7mb | 21.7mb |

4. set up mapping for fields

```
PUT /shakespeare
{
    "mappings": {
        "doc": {
            "properties": {
                "speaker": {
                    "type": "keyword"
                },
                "play_name": {
                    "type": "keyword"
                },
                "line_id": {
                    "type": "integer"
                },
                "speech_number": {
                    "type": "integer"
                }
            }
        }
    }
}
```

# Indexing

It's important to decide what to search and how to index, my decision is search a keyword in "text_entry" filed and return all the "play_name"s.

I read the json file line by line, for each line, use `json.loads(line)` to convert json data into string, it helps me to take out information from them.

The json file contains two kind of lines, they come in pairs, but only one of them is useful. The lines are

> - {"index":{"*index*":"*shakespeare*","id":0}}
> - {"type":"act","line_id":1,"play_name":"Henry IV", "speech_number":"","line_number":"","speaker":"","text_entry":"ACT I"}

It is quite obvious that only the **second** line is what I need. The first line contains key "index", that's how I distinguish them and add all second lines to a collection.

```
if("index" in shake_raw):
    pass
else:
    shake.append(shake_raw)
```

Once I have the pure information, I can generate a dictionary to store `"text_entry":"play_name list"` pair, for example: `"Hi":{'Hemlet','Henry V','A midsummer dream'}`, it's the most effecient way to search a keywrod, also I pre-process the data like assignment 1 did.

```
for s in shake:
    if(play != s["play_name"]):
        content = set() # empty the set when finish a play_name
    play = s["play_name"]
PSEUDO: preprocess tokens:
    content.add(token)
dic[play] = content
```

There is a slightly different in **pipeline_stem.py**, a stemming process is added.

**Please note, this indexing step would take 5 to 10 minuts to run, please be patient.**

The complete indexed dictionary will be `"keyword" : "set(play_names)"`

# Prepare 10 queries

I prepared 10 queries as following:

> Bunch, hi, kill, rememberst, slightest, exclamation, bug, homes, eating, childrens.

The expected play names are stored in a dictionary called "expected"

```
expected = {
    "bunch": OrderedSet(['Henry IV', 'Measure for measure', 'Richard III']),
    "hi": OrderedSet(['Much Ado about nothing']),
    "kill": OrderedSet(['Henry IV', 'Henry VI Part 2', 'Henry VI Part 3', 'Alls well that ends well', 'As you like it', 'Antony and Cleopatra', 'Cymbeline', 'King John', 'Julius Caesar', 'King Lear', 'Loves Labours Lost', 'macbeth', 'Measure for measure', 'Merchant of Venice', 'A Midsummer nights dream', 'Much Ado about nothing', 'Othello', 'Pericles', 'Richard II', 'Richard III', 'Romeo and Juliet', 'Taming of the Shrew', 'Timon of Athens', 'Titus Andronicus', 'Twelfth Night', 'Henry V', 'Julius Caesar', 'Merry Wives of Windsor', 'Richard II', 'Richard III', 'Troilus and Cressida', 'Alls well that ends well', 'Merry Wives of Windsor', 'Much Ado about nothing', 'Pericles', 'The Tempest', 'Two Gentlemen of Verona', 'Coriolanus', 'Hamlet', 'Henry VIII', 'King John', 'A Winters Tale', 'Henry VI Part 1', 'A Comedy of Errors', 'Coriolanus', 'macbeth']),
    "rememberst": OrderedSet(['As you like it', 'The Tempest']),
    "slightest": OrderedSet(['As you like it', 'Much Ado about nothing', 'Henry IV']),
    "exclamation": OrderedSet(['Henry VIII', 'Much Ado about nothing', 'King John']),
    "bug": OrderedSet(['Henry VI Part 3', 'A Winters Tale']),
    "homes": OrderedSet(['Coriolanus', 'King John']),
    "eating": OrderedSet(['Henry IV', 'Richard II', 'Titus Andronicus', 'Twelfth Night', 'Richard II', 'Timon of Athens', 'Henry VI Part 2', 'Two Gentlemen of Verona', 'A Winters Tale', 'Julius Caesar', 'Measure for measure']),
    "childrens": OrderedSet(['Henry VI Part 2', 'macbeth', 'Richard III', 'Henry IV', 'Coriolanus', 'Measure for measure', 'Richard II', 'Romeo and Juliet', 'Henry VI Part 1', 'Henry VIII'])
    }
```

## why these words?

- Bunch: this word is the only one that the search engine can't find, the reason is simple, in the play "Richard III", the bunch is not a single word, it is **bunch-backd**, unfortuanlly, the tokenization and stemming can't solve this problem.
- hi: there is only one occurance of this keyword in the dataset, if the search engine can't find it, the result will be 0.
- Kill: This keyword appears alomost in every play, it's good chance to test recall.
- Rememberst: this word is a old style word, I want to test wheather the stemming works for this one.
- Slightest, Exclamation, homes, eating: test for stemming as well.
- bug: no reason
- Childrens: There is no such word now, so I curious about how stemming works on this word.

Additionally, most of these keys has limited values, so I don't have to worry about missing some.

## why ordered set?

It's important to keep the order of elements in the evaluation set, for example

- k=1 False
- k=2 True

  AP is 0.25
- k=1 True
- k=2 False

  AP is 0.75

Set hash the elements so the position of a element is not controlable, therefore, I use the 3rd party library Ordered_Set to keep the elements in original order, it supports & operater and index as well, It's a very good choice for this project.

# Search

Searching is meant to received the play name list of a keyword, for both my seach engine and expected result. So I have 2 functions to do this.

- get predicted result set of my search engine.

```
def get_predicted(keyword):
    try:
        result = index[keyword]
    except KeyError:
        print('no such key.')
        result = set()
    return result
```

- get expected result

```
def get_actual(keyword):
    try:
        dataset = stemmed_expected(expected)
        result = dataset[keyword]
    except KeyError:
        print('no such key..')
        result = set()
    return result
```

An example of a result: `['Henry IV', 'Measure for measure', 'Richard III']`

# Evaluation

I use **MAP** method to evaluate the search engine.

MAP calucate mean value of AP, AP calcuate P@K for each play name. So I implement 2 functions, they are **cal_PK()** and **cal_MAP()**

```
def cal_PK(keyword):
    k = 1
    pk_sum = 0
    while k<= len(get_actual(keyword)):
        predicted = get_predicted(keyword)
        actual = OrderedSet(get_actual(keyword)[:k])
        corret = len(predicted & actual)
        precision = corret/float(k)
        pk_sum += precision
        print("k="+str(k)+"\n"+
"predicted:"+str(predicted)+"\n"+"actual"+str(actual)+"\n"+"precision"+str
(precision))
        k += 1
    try:
        pk = format(pk_sum/(k-1), '.2f')
    except ZeroDivisionError:
        pk = 0.00
print("pk of *"+keyword+"* is "+str(pk)+"\n\n\n")
return pk
```

The k value starts from 1, increase 1 every iteration. in each iteration, the pk_sum get the update by adding precision, which is calculated by comparing the common elements in predicted and expected results.

Here is a example output of word "Bunch":

```
k=1
predicted:{'Henry IV', 'Measure for measure'}
actualOrderedSet(['Henry IV'])
precision1.0
k=2
predicted:{'Henry IV', 'Measure for measure'}
actualOrderedSet(['Henry IV', 'Measure for measure'])
precision1.0
k=3
predicted:{'Henry IV', 'Measure for measure'}
actualOrderedSet(['Henry IV', 'Measure for measure', 'Richard III'])
precision0.6666666666666666
pk of *bunch* is 0.89
```

When k=1, the actual order has only one element, it compares with the predicted result and it matches, so the precision is 1.0, when k = 3, only 2 elements are matched, so the precision is 2/3, so the **AP is 0.89** for this word.

The calculation of MAP is sum the APs and get the mean.

```
def cal_map(dataset):
    map_value = 0
    for keyword in dataset.keys():
        keyword = LancasterStemmer().stem(keyword)
        map_value += float(cal_PK(keyword))
    map_value = format(map_value/len(dataset.keys()), '.2f')
    print('MAP: '+map_value)
    return map_value
```

Iterate every key of the expected dataset and get the mean of APs. It's simple to calculate. The MAP result will be given at the end of the output.

`MAP: 0.99`

## Stemming

I have 2 scripts, with stemming and without stemming. The MAP value is the same.

Both scripts pre-process the dataset with tokenization, remove punctuation and lower case.

In the stemming version, I stem the keyword of the expected dataset, so "homes" become"hom", I was expected stemming could help me solve the problem of "bunch-head", but it is not working.

I was stuck in converting the expected dataset to stemmed dataset for a while because I can't directly change the key by assign a new value, I need to create a new dictionary, here is the code.

```
def stemmed_expected(dataset):
    dic = {}
    for x in dataset.keys():
        y = LancasterStemmer().stem(x)
        dic[y] = dataset[x]
    return dic
```

# Future works

The MAP value for the system is fine, but I still have a lot of work to do in the future.

- The "bunch-head" problem

  I can't get rid of the "-" in between of 2 words by NLTK, the elastic search could find it correctly. In future work, I consider applying `split("-")` to each word.

- Search with **AND, OR, NOT**

I don't have time to implement these functions as I was planning to do so, my expectation is when people search for multiple words, for example, "lose yourself" is actually " lose OR yourself", I can add the result set of "lose" and "yourself" to get the result that contains both words. It should be easy to implement.

- cache the index file

  I could store the indexed dictionary in a file, and read the file when searching, it's much faster than index the dataset every time. I have it when I test my code.