# HW 1

```
NAME: Yunzhi Ye
UNI: YY2509
```

## Bref intro

sourch code:

1. Document.py: deal with document, parse xml file, record meta data.
2. IDF.py: calculate DF as well as IDF
3. index.py/index: build up index, store index to file
4. query.py/query: deal with all kinds of user query

used package:

1. stemming.porter3: stem words
2. did `not` use nltk package

statisitical data:

1. index time: 15s
2. index size: 2MB

---

In order to finish `Homework 1`, I implemented following features*:

```
1. parsing given files
2. building invert index
3. accepting user queries, and present query results.
4. calculate index time
5. calculate query time
```

**All the implement are via python*

---

`PARSING FILES` -- Document.py

1. parse xml
2. separate the document into different parts
3. split the text of document into single words

4. record the location of every word in the document
5. stem the words
6. calculate the term frequency

As the homework specification showing us, all the documents are presented in `xml` format. In order to parse these xml files, I defined a function called `parseFile` in class `Document`, and used the package `xml.etree.ElementTree`.

```
import xml.etree.ElementTree as et
…
class Document:
...
    def parseFile(self):
        tree = et.parse(self.filename)
        doc = tree.getroot()
        self.DOC_NO = int(doc.find('DOCNO').text)
        ...
        self.DOC_TEXT = doc.find('TEXT').text
```

this segment of codes split the whole document into different parts, and remove the `tags` of `xml`.

After that, I dealt with the single words. I firstly splited the `Text` part of a document into words by

```
re.split('\W+',self.DOC_TEXT)
```

Before stemming the words, I did some record works. I firstly **record the start point of every single word** in the document (That is for the convenience in query part). This is the second most time-consuming part when preparing for the index (stem the words is the most time-consuming one)

```
def trackLoc(self):
    pre = ' '
    loc = 0
    word = 1
    self.wordLoc = dict()
    for c in self.DOC_TEXT:
        if (re.match('\w',c,0) and not re.match('\w',pre,0)):
            self.wordLoc[word] = loc
            word +=1
        loc+=1
        pre = c
```

And then stem these word by

```
self.words = [stem(word.lower()) for word in self.words]
```

Finally, I calculated the `TF` (term frequency) of the document.

```
def buildTF(self):
    self.TF = dict()
    for word in self.words:
        if(self.TF.has_key(word)):
            self.TF[word] += 1
        else:
            self.TF[word] = 1
    return self.TF
```

---

`BUILD IDF` -- IDF.py

```
1. get documents
2. calculate DF
3. calculate IDF
```

Before calculating the `IDF`(Invert Document Frequency) of a word, I firstly should know the `DF`(Document Frequency). Hence, at the very begining, I should get all the documents, and after getting all the documents, I can easily get the `documents number`.

After I get the files, I started to calculate the `DF` of each word.

```
def buildDF(self,doc):
    for k in doc.TF.keys():
        if self.DF.has_key(k):
            self.DF[k] += 1
        else:
            self.DF[k] = 1
```

Combining `documents number` and `DF`, I calculated the `IDF`.

```
    def buildIDF(self):
    self.freqList = dict()
    if self.DF == None:
        self.buildDF()
    for k in self.DF.keys():
        v = self.DF[k]
        frequency = log(self.doc_count/float(v))
        self.freqList[k] = (v,frequency)
```

---

`INDEX` -- Index.py

```
1. Build index
```

I build the index in this way:

> *index[word]={(Document ID, Location, TF),(Document ID, Location, TF),...}*

`index[word]` : I asigned the index to a `dictionary` data structure in `python`. and the `key` for each item is the word in `string format`.

`Document ID` : it is the `ID` for the document that the word presented in.

`Location` : it is the word location in the document, which I record in the `PARSING FILE` phase.

`TF` : it is the term frequency of the word in the document, which I record in the `PARSING FILE` phase.

The code for building the index is as follow:

```
def buildIndex(self, doc):
    doc_no = int(doc.DOC_NO)
    words = doc.words
    loc= 1
    for word in words:
        item = (doc_no, loc, doc.TF[word])
        if self.wordList.has_key(word):
            self.wordList[word].append(item)
        else:
            itemList = [item]
            self.wordList[word] = itemList
        loc+=1
```

Finally, I stored the whole index structure into a pickle stucture which python provides for `Object Serialization`.

---

`QUERY` -- query.py

```
1. support command df
2. support command tf
3. support command doc
4. support command title
5. support command similar
5. support single word or phrase search
6. support combined words or phrases search
```

In `query.py`, I first parsed the users input, in order to know what kind of command the user want to execute.

```
while True:
try:
    var = raw_input('$')
    argv = shlex.split(var.lower())
    if argv[0] == 'df':
        ...
        df(argv[1],index)
    elif argv[0] == 'doc':
        ...
        showDoc(argv[1])
    elif argv[0] == 'title':
        ...
        showTitle(argv[1])
    elif argv[0] == 'tf':
        ...
        termFreq(argv[1],argv[2],tf)
    elif argv[0] == 'freq':
        ...
        freq(argv[1],index)
    elif argv[0] == 'exit':
        break
    else:
        search(argv,wordLoc,index)
except ValueError:
    errInfo()
```

`df`

if the user want to execute command `df`, I look up to the index, to find which document does this word presents in. then I can get several tuples from index in the shape of `(document id, location, tf)`. In a for loop, I can look up the index of the `input word` or the index of all words in the `input phrase`.

For the input word, I just count the different documents id, and then print out.

However, for the input phrase, situation becomes a little complex:

1. Get the index of every word in the phrase, and the `word count` in the phrase
2. For the first word index, I put the tuples (document id, location+(`word count`-1)) in to a set; the second word index, I put the tuples (document id, location+(`word count` -2)) in to another set, and for the following words, doing the same thing.
3. Then I intersect all the set, and count the different document in the result, then print out the number

```
def df(text,index):
    text = text.strip()
    textList = re.split('\W+',text)
    if len(textList) > 0:
        textList = [stem(word) for word in textList]
        setList = list()
        length = len(textList)-1
        for word in textList:
            if index.has_key(word)==False:
                print 0
                return
            wordSet = { (tuples[0], tuples[1]+length) for tuples in
index[word]}
            setList.append(wordSet)
            length-=1
        docNum= setList[0]
        for Docset in setList:
            docNum = docNum & Docset
        doc_final = set()
        for dn in docNum:
            doc_final.add(dn[0])
        print len(doc_final)
```

`tf`

`tf` is much simplier than `df` command. I have stored the term frequcy for each document when doing `PARSING FILE`phase. So for command `tf`, I just need to look up to the record to find the certain data.

```
def termFreq(doc_no,word,tf):
    try:
        doc_no = int(doc_no)
        if tf.has_key(doc_no) ==False:
            print "File "+doc_no+' does not exist.'
            return
        doc_tf = tf[doc_no]
        if doc_tf.has_key(word) == False:
            print 0
            return
        number = doc_tf[word]
        print number
    except ValueError:
        print "File "+doc_no+' does not exist.'
```

### `doc` and `title`

`doc` command and `title` command is straghtforward, I just look up to the document, then print it out.

### `similar`

For `similar` command, query system will return a set of similar words compare to the input one. and all the similar words have a `EDIT DISTANCE` less than 3 from input word

### `search`

`search` command is the most difficult one. Firstly, I seperate the **positive words**(words without prefix `!`) and **negative words**(words with prefix `!`).

For the **negative words**, I look up to the index and find the `document id` of documents which contain the word, then I remove these document id from whole document id pool, then add them to result set.

For the **positive words**, I look up the index and find the `document id`, and for the phrase intersect the result, as in handling the `df`command. the I add the found document id into the result set and with a score attached with each document id.

```
def findPostive(word,score,index):
    word = word.strip()
    wordList = re.split('\W+',word);
    textList = filter(None, wordList)

    doc_final = set()

    textList = [stem(word) for word in textList]
    setList = list()
    length = len(textList)-1
    for word in textList:
        if index.has_key(word)==False:
            return
        wordSet = { (tuples[0], tuples[1]+length) for tuples in index[word]}
        setList.append(wordSet)
        length-=1
    docNum= setList[0]
    for Docset in setList:
        docNum = docNum & Docset
    for item in docNum:
        if score.has_key(item[0]):
            score[item[0]]+= len(textList)
        else:
            score[item[0]] = len(textList)
    targetDoc = dict()
    for doc in docNum:
        if targetDoc.has_key(doc[0]):
            continue
        targetDoc[doc[0]] = (doc[1]-len(textList)+1,len(textList))
    return targetDoc
```

Finally, I sorted the result set subject to the score, and print out every document segment I found.

```
def search(words, wordLoc,index):

    score = dict()

    doc_find = dict()
    for word in words:
        if word[0] == '!':
            negateWord = findNegate(word[1:],score,index)
        else:
            positiveWord = findPostive(word,score,index)
            if positiveWord == None:
                continue
            for doc in positiveWord.keys():
                if doc_find.has_key(doc):
                    continue
                doc_find[doc] = positiveWord[doc]

    PRINT SEGMENT FOUND

    RETURN
```