MySearchEngine

Table of Contents

Introduction

MySearchEngine has four main components: Controller, Backend, Parser and Test. Controller is in charge of dealing with user command and organize each components work together. Parser is in charge of transform the natural language to structured data. Backend is in charge of index and retrieve structured data. Test is in charge of test functionality. We will show detail of Backend and Parser in following sections, and Controller and Test will be included in user manual. Besides, this project has other helper components to support functionality: Vector has two subclass, and we will see detail in Backend section; SpellChecker class is in charge of detecting spelling mistake and giving out suggestions, we will see detail in SpellChecker section.

Parser

Requirements

1. Find all tokens from input file;
2. Stem tokens as term;
3. Reveal the match rule of the token; (for testing and debug)
4. Report the distribution of token matched by each rule;(for testing and debug)

Design

Since rules are defined, we can extract tokens by Java regex method, and we can just create an regex.txt file to store all rules, so that we can modify rules without change source code. The regex.txt file is look like this:

```
####### rule 1: joined by hyphenate
((\w+\-+\n*)+\w+)
####### rule 2.1: email addressa
(([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6}))
####### rule 2.2: url
((https?:\/\/)([\da-z\.\-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?)
####### rule 2.3: ip address
((?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-
9]|[01]?[0-9][0-9]?))
####### rule 3: single quotation
(\s+'(\w+\s+)+\w+'\s+|^'(\w+\s+)+\w+'\s+)
####### rule 4: Two or more words start with capital letter
(([A-Z]+[A-Za-z\-]*[ \t\x0B\f\r])+([A-Z]+[A-Za-z]*))
####### rule 5: Acronym
(((\w+)\.)+(\w+))
####### extra rule: street name
(([Ss][Tt]\.\s?\w+))
####### extra rule: something like `you're`, `can't`
(\w+'\w*)
####### normal word
(\w+)
```

To make it readable, odd lines are comments, even lines are regex expressions. And each rule has a priority, top rules will be matched first.

Here is a problem we have to consider: If we match each rule separately, a substring will be extract multiple times, for example, if "Java Programming" match two rules ("Java Programming" and "Java"), "Java" will be extracted twice. To avoid this problem, we have two choices:

1. Eliminate matched substring in each iteration and exit when the current string is empty;
2. Concatenate all regexes together by "|", regex automaton will consume matched substring automatically;

The first choice is hard to implement and has potential efficiency problem, so we choose the second one, this will be implemented in **regexTokenize** method.
The **regexTokenize** will return an ArrayList<String>, contain all extracted tokens. Those token are very close to term, but some of them have strange form. For example, newline character will be extracted in "a-\nb", and we want use "-" replace invisible character in token, like using "Search-Engine" instead of "Search Engine", what's more, we want all letters are lowercase. So we add an **cleanUp** method to change all tokens into the form we want.
Finally, we use **stemTerms** method to get stemmed terms. The stem algorithm we use is found on from https://tartarus.org/martin/PorterStemmer/java.txt.
workflow:

Testing

For the testing and debugging purposes(requirement 3, 4), we have to add other method to collect information in runtime and report them.
Since we concatenate all regexes together, we don't know which rule the extracted token belong to.
A simple solution is to enumerate all rules, so we add a **findMatchRule** method: passing a token string and regex, return the matched rule.

```
public static AbstractMap.SimpleEntry<String, String> findMatchRule(
String s, HashMap<String, String> regexs) {
  for (String desc: regexs.keySet()) {
    String expr = regexs.get(desc);
    Matcher m = Pattern.compile(expr).matcher(s);
    if (m.find()) {
      return new AbstractMap.SimpleEntry<String, String>(desc, expr);
    }
  }
  return null;
}
```

Then we can find the distribution of matched rules, by using **matchStatistic** method:

```
public static <T extends Collection<String>> void matchStatistic(T rawTerms) {
  HashMap<String, Integer> statistic = new HashMap<>();
  HashMap<String, String> regexs = readRegex("regex.txt");
  for (String i: rawTerms) {
    AbstractMap.SimpleEntry<String, String> x = findMatchRule(i, regexs);
    assert(x != null);
    if (!statistic.containsKey(x.getKey())) statistic.put(x.getKey(), 1);
    else statistic.put(x.getKey(), statistic.get(x.getKey()) + 1);
  }
  for (String i: report.keySet()) {
    String fn = i.replaceAll("#+", "");
    writeReport(fn.split(":")[0].trim(), report.get(i), "report/");
  }
  int tot = 0;
  for (String key: statistic.keySet()) {
    tot += statistic.get(key);
    System.out.printf("Total terms: %d (%s)\n", statistic.get(key), key);
  }
  System.out.printf("Total term: %d\n", tot);
}
```

We may want to find the distribution of a set of tokens or an array or tokens, so here we use generic method. It will write result to files under folder "./report", information in there will be used to generate dictionary for spell checker.

Here is a screen shot of running test case for this part:

```
|eggeek:src$ ./go
|Parser: executing regexTokenize...
|Parser: executing matchStatistic(ArrayList)...
|Total terms: 3503 (####### rule 5: Acronym)
|Total terms: 634 (####### extra rule: street name)
|Total terms: 5799 (####### extra rule: something like `you're`, `can't`)
|Total terms: 1172210 (####### normal word)
|Total terms: 11928 (####### rule 1: joined by hyphenate)
|Total term: 1194074
|Parser: executing matchStatistic(Set)...
|Total terms: 857 (####### rule 5: Acronym)
|Total terms: 154 (####### extra rule: street name)
|Total terms: 1214 (####### extra rule: something like `you're`, `can't`)
|Total terms: 35973 (####### normal word)
|Total terms: 5913 (####### rule 1: joined by hyphenate)
|Total term: 44111
|Parser: executing cleanUp...
|Parser: executing stemTerms...
```

And, We use test/rule_*.txt as input to test parsing, you can browse the test folder to see detail:

```
|eggeek:test$ ls rule*
|rule1.txt  rule2.txt  rule3.txt  rule4.txt  rule5.txt  rule_address.txt  rule_punctuation.txt
```

and you can run test case by specific command. (see example in UserManual)

Backend

Requirements

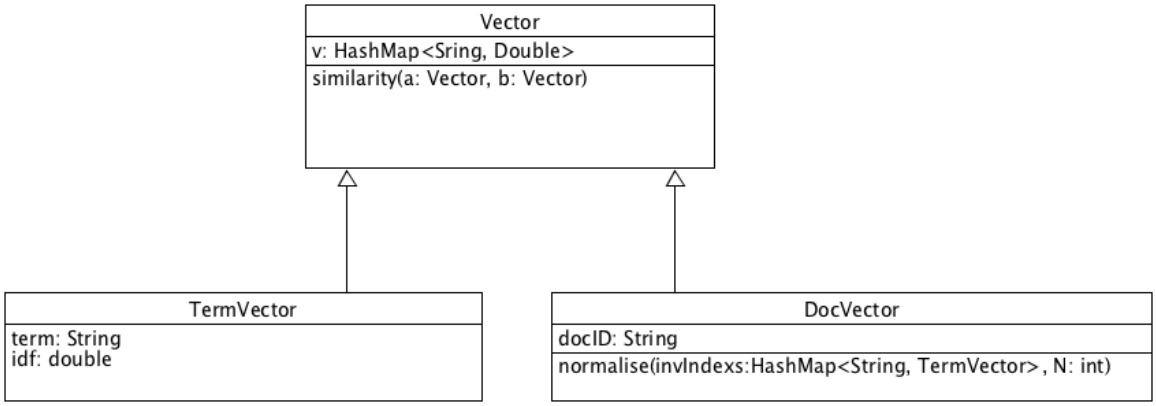
1. Store each documents in a vector form;
2. Calculate similarity between two documents based on vector space;
3. Calculate term's tf in each documents and it's idf, and store the result in index file as invert index;
4. Able to reconstruct term's information from index file;
5. Retrieval documents and sort by rating.

Design

Firstly, we should ignore all stop words according to stopwords.txt, all words in stopwords.txt are stemmed, here are reasons:

1. We want the backend is smart enough to find stop words list in the terms set, and terms are stemmed words;
2. The algorithm to find stop words is finding all terms with small idf (less than 1e-4), which is the main purpose of our stop words list, so it should be helpful enough;
3. Although some researchers indicate that using stemmed stop words has negative influence, it's depend on your collection and the purpose of stop words;

Then, for each term, we want store it in some way like: {term, <docID, tf>, <docID, tf> ,... <docID, tf>, idf}, and retrieval it effectively. A suitable data structure to in Java is HashMap, and for each document, we have similar requirement. So we implement Vector class, TermVector class and DocVector class to reuse the code.



In TermVector, we store information in HashMap<String, Double>, key is **docID**, value is **tf**, for the purpose of further calculation, the type of value is Double.
In DocVector, we store information in HashMap<String, Double>, key is **term**, value is **tf**,
In this design, inverted index is easily defined as HashMap<String, TermVector>.

Implement

To calculate similarity, we need two step.
First, we need represent a document by a vector like <tf*idf, tf*idf, ...>, so we implement a normalize method in DocVector class.

```
public Vector normalise(HashMap<String, TermVector> invIndexs, int N) {
    Vector res = new Vector();
    for (String term: v.keySet()) {
      double tfIdf = v.get(term);
      if (invIndexs.containsKey(term)) tfIdf *= invIndexs.get(term).getIDF();
      else tfIdf *= Backend.calcIDF(0, N);
      res.v.put(term, tfIdf);
    }
    return res;
  }
```

We also regard a query as a document, so a query will be represent by a DocVector.
One thing should consider is that a term in query may not in our inverted index, so we add a if-else branch to deal with this case.
Second, we need to the calculation.

```
  public static double smilarity(Vector a, Vector b) {
    double x = 0.0;
    for (String ka: a.v.keySet()) {
      if (b.v.containsKey(ka)) x += a.v.get(ka) * b.v.get(ka);
    }
    double y = a.len() * b.len();
    return x / (y + 0.1);
  }
```

Note that if key doesn't in HasMap means the value is 0, so we just need to consider exist key in the for loop. To avoid dividing by zero, we add 0.1 on y.

Besides, the idf is calculated in Backend class.

```
public static double calcIDF(int df, int N) {
  return Math.log((double)(N + 0.1) / (double)(df + 0.1));
}
```

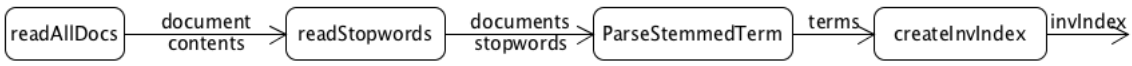
The reason is that the formula is depend on the model we choose, which is related to Backend class. For example, we may switch from idf to wf or ntf.

For requirement 3, we just need to override toString method in TermVector class and write all invIndexs.values() to file.

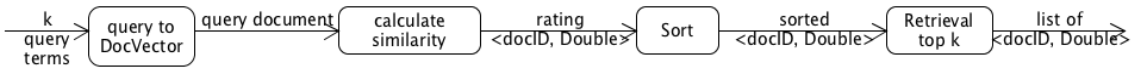
```
@Override
public String toString() {
ArrayList<String> res = new ArrayList<>();
    res.add(term);
    for (String i: v.keySet())
      res.add(String.format("%s,%.3f", i, v.get(i)));
    res.add(String.format("%.3f", idf));
    return String.join(",", res);
  }

// to string
for (TermVector i: outs) {
    content.add(i.toString());
}
// write to file
PrintWriter out = new PrintWriter(indexPath);
for (String i: content) out.println(i);
```

Now we already get all tools we need, rest of part is just implement, see detail in source code and workflow.

Workflow: Create index

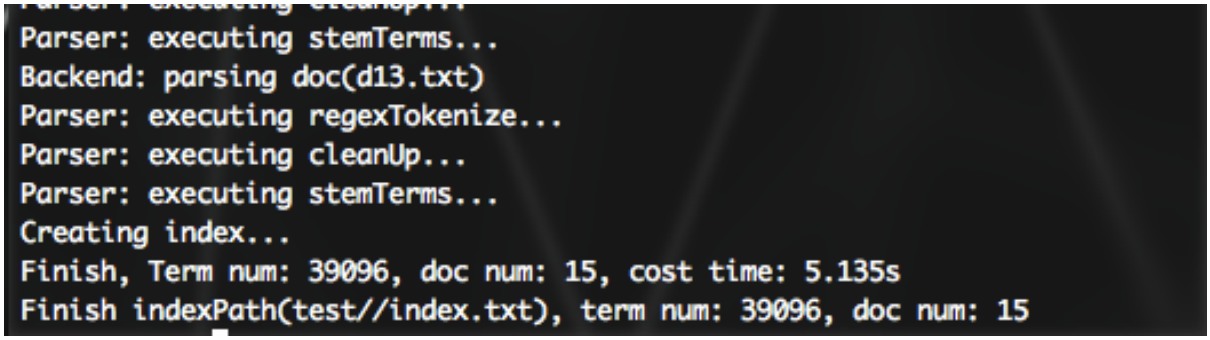


Workflow: Retrieval



Testing

In Backend, we need test create index and load file, the result of those two process should be exactly same, besides, we should profile the running time, it shouldn't be too slow.
We use **testIndex** and **testBackendRead** in Test class to test create index and load file, here is result:



First we performed create index, then we performed read index. The output indicates that each process produced same number of terms and documents, and the running time is 5.135 seconds, which is fast enough.

SpellChecker

Requirements

1. Read dictionary from file;
2. Calculate distance of 2 word;
3. For given word, find the most similar word (or itself, it the word in dictionary);

Analysis

According to the distribution of each rule, most of terms are normal words, so we choose normal words as dictionary.
To calculate similarity, we have two feasible choices: using probability model like Bayes formula or discrete model like edit distance.
The probability model need training set, or generate training set by using algorithm: for each word in dictionary, generate wrong word by simulating each kind of typing mistake, for example, from "word" we can get "wrd", "wrod". But it is hard to guarantee that the wrong word set is complete, we won't be able to detect wrong word if it is not in training set. According to statistic result, the size of dictionary is small, a wrong word may only map to few correct word. Thus, probability model is not suitable in this collection, and we choose discrete model.

Algorithm Design

Classic edit distance calculation is a dynamic programming problem:
given string a, and string b,
dp[ia][ib] represent the minimal number of edit cost when we match substring of a[0:ia] and b[0:ib], and dp can be defined recursively:

```
match at ia and ib:
dp[ia+1][ib+1] = min(dp[ia+1][ib+1], dp[ia][ib]), when a[ia] and
b[ib] already match;
dp[ia+1][ib+1] = min(dp[ia+1][ib+1], dp[ia][ib]+1), when a[ia] and
b[ib] not equal, replace a character.
not match at ia and ib:
dp[ia+1][ib] = min(dp[ia+1][ib], dp[ia][ib] + 1),  delete a[ia]/ or
add in b to match a[ia];
dp[ia][ib+1] = min(dp[ia][ib+1], dp[ia][ib] + 1), delete b[ib]/ or
add in a to match b[ib];
```

`boundary case is dp[0][0] = 0, and dp[len(a)][len(b)] is the edit cost.`

However, classic algorithm won't perform well in spelling check, for example "hckre" is more close to "scare" than "hacker". The reason is:

1. classic algorithm doesn't consider swap operation, which is very common in real life;
2. classic algorithm gives each position same cost weight, but in real life, people more likely to make mistake at tail rather than head;

Thus, we can improve the classic algorithm by:

1. add swap operation;
2. use other formula to evaluate the cost;

For the swap operation, to keep simple, we only consider the case that swapping adjacent characters.
For the evaluation, we choose the formula: ***log((len+2) / (pos+1))***; (similar to the idf)
where len is the original length of modified string, pos is the edit position.
The reason is: the importance of each character is decrease from head to tail, and this change is not linear, and the range for pos is from 0 to len, so we add "+1" to avoid dividing by zero and "len+2" is designed to give the last character non-zero weight.
We give swap operation very low weight (shrink by 2), because mistake made by swap is quite common, and swapping adjacent character won't change word a lot.

Implement

```
public static double dist(String w, String c) {
  int n = c.length(), m = w.length();
  double[][] v = new double[n+1][m+1];
  for (int i=0; i<=n; i++)
    for (int j=0; j<=m; j++) v[i][j] = INF;
  v[0][0] = 0.0;
  for (int i=0; i<=n; i++) {
    for (int j=0; j<=m; j++) if (Math.abs(v[i][j] - INF) > EPS) {
      if (i < n && j < m && c.charAt(i) == w.charAt(j)) { // match without cost
        v[i+1][j+1] = Math.min(v[i+1][j+1], v[i][j]);
      }
      if (i < n && j < m && c.charAt(i) != w.charAt(j)) { // replace and match
        v[i+1][j+1] = Math.min(v[i+1][j+1], v[i][j] + cost(j, m));
      }
      // swap and match
      if (i+1 < n && j+1 < m &&
          c.charAt(i+1) == w.charAt(j) &&
          c.charAt(i) == w.charAt(j+1)) {
        v[i+2][j+2] = Math.min(v[i+2][j+2], v[i][j] + cost(j, m) / 2.0);
      }
      // delete j
      if (j+1 <= m) v[i][j+1] = Math.min(v[i][j+1], v[i][j] + cost(j, m));
      // add at j
      if (i+1 <= n) v[i+1][j] = Math.min(v[i+1][j], v[i][j] + cost(j, m));
    }
  }
  return v[n][m];
}
```

In the code, w represents "wrong word", c represents "correct word". The n is the length of c and m is the length of w, the index i from 0 to n in outer loop, and index j from 0 to m in the inner loop for each iteration, so the total complexity is O(n*m). Since n and m are very small (usually less than 10), this algorithm should be fast enough.

Testing

In SpellChecker, we need test functionality and execution time, and we will use **testSpellChecker** method in Test class for testing, first we performed **testSpellChecker("hckre", true)** to test classic algorithm performance:

```
Search hckre
Find, hacker, 1.500000
Find, scare, 2.000000
Find, score, 2.000000
Find, havre, 2.000000
Find, care, 2.000000
Find, hoare, 2.000000
Find, acre, 2.000000
Find, core, 2.000000
Find, ochre, 2.000000
Find, hare, 2.000000
Execute time: 0.342000s
```

Then, we performed **testSpellChecker("hckre", false)** to test our new algorithm:

```
Search hckre
Find, hacker, 1.532571
Find, hackers, 1.686722
Find, heke, 1.812379
Find, hacked, 1.966529
Find, hocken, 1.966529
Find, hackney, 1.966529
Find, chores, 1.974404
Find, hackerism, 1.995023
Find, hurki, 2.012884
Find, havre, 2.100061
Execute time: 0.413000s
```

And we can see the output of our new algorithm is more close to the result we are seeking for.